Animating and Lighting Grass in Real-Time

by

Brook M. Bakay

B.Sc., The University of Regina, 1999
B.A., The University of Regina, 1992

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

. . . . . .

. . . . . .

. . . . . .

THE UNIVERSITY OF BRITISH COLUMBIA

April 22, 2003

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature) _____

Department of Computer Science

The University Of British Columbia
Vancouver, Canada

Date _____

Abstract

I present a simple method to render fields of grass, animated in the wind, in real-time. The technique employs vertex shaders to render displacement maps with Russian-doll style transparent shells. Animation is achieved by translating the surface according to a local wind vector while preserving the length of the blades of grass. Lighting is also done in a vertex shader, with point light sources, accounting for the anisotropic nature of grass, self-shadowing and attenuation. This technique achieves convincing results on current consumer graphics hardware and can be applied to other similar surfaces such as hair and fur.

Contents

List of Tables

List of Figures

Dedication

This thesis and the two and a half years it represents are for Denise. Thank you for repeatedly kicking my ass, and believing in me even when I couldn't stand it. I love you.

I also dedicate this thesis to Barnabas, whom I miss every day. I would trade all of this in a heartbeat for one more walk on the beach with you. I am so sorry I let you down.

Acknowledgements

This work was completed in large part because of the efforts and support of the following people:

Nancy Bakay, Warren Bakay, Graham Bakay, George Bakay, Roger Selby, James Gauthier, Mike Rud, Chris Miller, Tim Miller, Robert Petry, Roger Petry, Norman Petry, Dave Mackie, Dave Panchyk, Wes Moore, Arthur Louie, Alexander Stevenson, Kai Juse, Kevin Allardyce, Dee Jay Randall, Brad Kram, James Hall, Wolfgang Heidrich, Paul Lalonde, Richard Rosenberg, Michiel van de Panne and The Imager Lab at UBC.

Part I

Thesis

Chapter 1

Introduction

The realistic representation of outdoor scenes is a continuing problem in computer graphics. Natural life, flora and fauna, are incredibly complex phenomena to try and realize in a computer graphics context. Additionally, viewers are very familiar with the real thing, and can easily notice when something doesn't look "right". Real-time computer graphics has often relied on being "indoors" – using large occluding walls to facilitate detailed renderings. Outdoor applications, traditionally lacking occluders to cull non-visible geometry, have often used very sparse geometry to describe the landscape, often greatly impacting its believability. In these systems, a field of grass could be reduced to a single texture.

High performance consumer computer graphics hardware has allowed for the display of complex detailed natural phenomena, such as the fur on a bunny [11], at interactive frame rates. However, this work has not yet been able to effectively animate the hair or grass displayed. This work continues in that tradition, while using displacement maps and vertex shaders to leverage current consumer level computer graphics hardware for the animation of complex natural phenomena.

The grass is composed of transparent shells, layered above the landscape. The vertices of these shells are moved in real-time to create the animation. Control of the animation direction, in response to a simulated wind field, is maintained at the vertex level. At each time step, each vertex is moved a distance determined by a global intensity function in accordance with the local wind direction. The wind direction is stored at the vertex level allowing for arbitrarily accurate wind movement over a

Figure 1.1: "Rough winds do shake the darling buds of May." Shakespeare, Sonnets: XVIII

landscape, and arbitrarily complex wind patterns. Vertices are divided into groups that use different intensity values facilitating advanced global wind effects such as waves across the landscape, attenuation in wind intensity or even whirlwinds.

Similarly, the lighting of anisotropic surfaces such as grass or fur has been difficult for real-time applications. Proper lighting is no less important for believability in natural scenes than animation. In fact, without good lighting, the animation itself can be difficult to see. Recent advances in computer graphics hardware have made this a possibility. This technique lights the grass with three different point lights. Two different "normals" are calculated for each vertex, one for diffuse light and one for specular. This, along with an approximation of self-shadowing, amounts to a real-time implementation of Banks [1] ray-traced lighting algorithm. Finally, attenuation is also accounted for by calculating the distance from each vertex to each light. All these

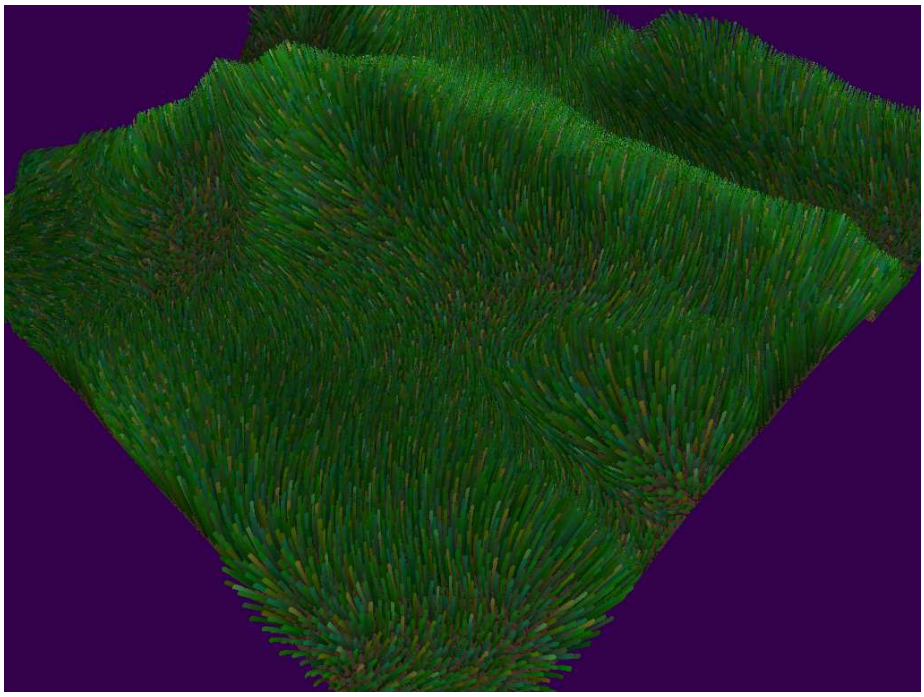

Figure 1.2: A grassy knoll.

lighting calculations are done at the vertex level, without a noticeable drop in speed and produce surprisingly credible results.

This technique is suitable for viewing from above, as in a flight simulator, or for walking in relatively short grass. It is fast, and because animation data is interpolated between vertices of the base mesh, fields of grass can be arbitrarily dense. This technique is also memory efficient as all the shell textures are generated as required from one base texture. Lastly, this technique incorporates vertex shaders available on current consumer grade graphics hardware to further increase speed.

Chapter 2

Previous Work

Previous work of interest to this thesis falls into these categories; Volume textures and animation for natural phenomena and Anisotropic lighting for filiform objects.

2.1 Volume Textures and Animation for Natural Phenomena

Jim Blinn [2] lay much of the groundwork for volumetric textures back in 1982. He described a method to synthesize an image based on the properties of light passing through a volume of particles. Blinn modelled the contributions to the synthesized image of the particles themselves, the background they were occluding and light reflected off them. Simple, probabilistic models resulted in stunning pictures of Saturn's rings and the beginning of using textures to describe complex, three dimensional, natural phenomena.

Building upon Blinn's work, Kajiya and Kay [7] introduced the idea of volumetric textures, which they called "texels" (not to be confused with the current notion of a texel as a texture element, usually in a two dimensional texture). A texel is a cubic space divided into a grid of smaller cubes called "voxels". At each voxel a bundle of information is stored including the density of the material (actually, the amount that the surfaces within the voxel will cover the total area of the voxel after projection), a "frame" consisting of the normal, tangent and binormal vectors, and a light reflection

model for the surface contained within. Since the surface remained the same throughout the texel, namely fur, the reflection model was constant - an ad-hoc cylindrical reflection model.

It should be noted that Kajiya and Kay are not modelling clouds of particles as Blinn was, but rather solid surfaces, sampled in three dimensions. Texels are intended to model complex repetitive geometry, such as the trees on the side of a mountain. A texel is a representative volume, and it is repeated over the surface of the underlying geometry in a thick layer. Because the geometry is not guaranteed to be flat, the texels are "warped" in order to have their sides match up.

Kajiya and Kay's texels were created to solve the problem of aliasing of complex geometries in ray tracing. Rendering a texel involves shooting a ray through it front to back, multiplying transparencies and accumulating intensities (weighted by the current transparency).

Displacement maps were introduced by Cook [3]. Used the context of his shade trees, the location of a surface is a parameter that could be changed based, for example, on the value in a texture. Where bump maps had previously only perturbed the surface normal, displacement maps perturbed the surface location. This amounted to a retesselation of the surface in accordance with the values in a displacement map texture.

Another early form of displacement maps was proposed by Musgrave [15], as a means to increase the speed of ray tracing height fields. Musgrave's "grid tracing" involved tracing a ray through a two dimensional array of square cells, the corners of which map to values in the height field to be rendered. Actual intersection tests need only be performed if the ray's height falls within the maximum and minimum heights of the cell. If so, the cell is divided into two triangles and actual ray/polygon intersection tests are performed. Thus, the polygonal topology of the height field is dynamically generated, or displaced, from the height field data.

Solid textures, an early form of three dimensional texture, were proposed indepen-

dently by Perlin [19] and Peachey [17]. Directly analogous to their two dimensional counterparts, solid textures represent a three dimensional space. Model vertices have indices into that space in much the same way they index into two dimensional textures. The solid textures of both Peachey and Perlin are procedurally generated – the colour values for vertices are evaluated as needed, often with complex and expensive functions. The effect is to easily make any arbitrarily complex object appear as if it were "sculpted" out of the material described by the solid texture.

Neyret [16] first proposed animating texels to increase the realism of a scene. This work was aimed at ray tracing applications in which complex natural phenomena had been represented and animated by physical simulation or particle systems. Neyret pointed out the utility of Kajiya and Kay's [7] texels for this area, and extended their work to allow for animation. Three types of animation were presented; changing the underlying surface mesh, deforming the bounding boxes and swapping out the texels themselves - as frames of animation. The first two types of animation are of interest to this work.

Perbet and Cani [18] discuss animation of volumetric textures to produce realistic grass in the wind. Their slices are perpendicular to the ground's surface, and thus their system is better suited to low views, close to the ground. They precompute a number of postures for each type of grass and send information to each blade regarding which direction to face, and which posture to assume. The two-dimensional textures for the slices are then computed from this information. Having data to control the motion of each blade of grass allows for some very detailed animations, but results in a performance penalty. The speed of this algorithm is dependent on the number of blades of grass in a scene. It would seem that a large amount of texture memory is also required, as each slice uses a unique texture. The authors were able to achieve interactive but not real-time frame rates.

Meyer and Neyret [13] discuss volume visualization using transparent slices com-

posed of textured polygons, and achieve interactive frame rates. They mention that the animation techniques described by Neyret [16] could be applied as well. In their system, a separate texture is stored for each slice, which can lead to memory issues.

Dietrich [4] introduced hardware acceleration of displacement maps. His technique involved rendering the volume texture as a series of transparent slices through the volume. The texture to be applied to the slices is encoded in the alpha channel of the base texture. The alpha value at each texel specifies the height at that texel. This is done simply by specifying a maximum height, and mapping that value to the maximum alpha value of 255. When a shell is rendered, the alpha test is enabled and the alpha compare value is set to the shell's height - a value from 0 to 255. Any texel with an alpha value greater or equal to the shell's height (the alpha compare value) will be rendered. Any texel failing that test will be transparent. This technique is extremely memory efficient because it generates a texture volume from a single two-dimensional texture, but it can lead to visual artifacts at grazing angles, where the viewer can see through the spaces between the slices.

Kautz and Seidel [8] built upon the work of Dietrich [4]. They demonstrated a fast and memory efficient method to re-generate the displacement map from any of the orthographic directions. This had the effect of minimizing the errors at grazing angles, as the best orthographic direction can be chosen each frame based on the view angle. Additionally, they present a best case method to draw the slices perpendicular to the view angle, eliminating the problems with grazing angle views. Unfortunately, their technique to generate shells in arbitrary slicing directions is not applicable here due to the high frequency data contained in the grass textures. We are limited to shells parallel to the model's surface. Shells perpendicular to the surface sample the base texture in one pixel wide strips and would miss a large proportion of the blades of grass. Kautz and Seidel do not address the animation of the volume.

Lengyel [11] produced a convincing furry bunny using several levels of detail, some

of which included volumetric textures. Lengyel's technique was similar to Kajiya and Kay's in that he generated a three dimensional reference texture and tiled it over the surface of the object. He used hardware acceleration to render the fur interactively using a technique very similar to Dietrich's [4]. This involved using Russian-doll style transparent shells as well, however with Lengyel's technique, each shell must have it's own unique texture (to properly represent the reference volume). This technique is memory intensive and not suitable for animation. Lengyel did alleviate the problem of visual artifacts at grazing angles with the addition of "fin" textures. Fin textures are perpendicular to the shell textures and are rendered at silhouette edges. Subsequent work [12] alleviated the memory requirements somewhat, but animation has not been addressed.

2.2 Anisotropic Lighting Of Filiform Objects

The problem of properly lighting a field of grass or a patch of fur has received some attention in recent years. These surfaces are inherently anisotropic – a surface is said to be anisotropic if the intensity of light reflected from it changes when rotating the surface about its normal while keeping the viewer and light directions constant. This is the result of microgeometry – the surface is much more complex at a more detailed level than it appears. Seemingly smooth surfaces may in fact be quite rough when viewed closely enough. This is the effect seen when one looks at the reflections coming from a Christmas tree ornament wound with satin thread, or the grooves in a vinyl record or the changing pictures as you move a children's toy.

Anisotropy must be taken into account in the lighting process. The standard lighting equations will not work for grass. The assumption in these equations is that they are lighting solid, uniform geometry with normals perpendicular to their surface. Kajiya and Kay [7] first proposed a method to light grass and fur. Following the standard lighting model, they separated the contributions of reflected light into diffuse and specular

categories.

Diffusely reflected light follows the Lambertian model in which the amount of reflected light is proportional to the dot product of light and normal vectors. Treating each strand as a cylinder, the total amount of light reflected per unit length of the cylinder can be found by integrating the diffuse equation along the half circle facing the light. Kajiya and Kay found that this result was proportional to the dot product between the light vector and its projection onto the plane perpendicular to the tangent vector of the cylinder.

Kajiya and Kay developed an ad hoc model to represent specular light reflections. They reasoned that a light ray hitting the cylinder will reflect at a mirror angle with respect to the tangent vector. Because the normals extend in all directions perpendicular to the tangent vector, the reflected light forms a cone around the tangent vector with an angle equal to the angle of incidence. The amount of specularly reflected light, then, is proportional to the dot product of the eye vector and the nearest vector in the cone to the eye vector. The sharpness of the specular highlight is controlled by raising this result to some arbitrary power.

Miller [14] proposed creating a "pseudo-reflectance map" to store the reflectance information for an isotropic surface. Miller was specifically interested in surfaces in which the intensity data can be represented simply as a function of the tangent vectors. The map is created by regularly sampling normals on the plane perpendicular to the tangent vector and running them through the regular, isotropic lighting equations. The reflectance for any given orientation of the tangent vector is equal to the average intensity of these samples. The results of this operation are stored in texture maps and applied as needed. Unfortunately, this model assumes an orthographic projection (the eye vector is perpendicular to the eye-space Z-axis for all points). More importantly, the pseudo-reflectance maps need to be recomputed every time the camera moves relative to the lights in the scene, making it unsuitable for interactive applications.

Banks [1] provides a detailed mathematical examination of both the diffuse and specular models provided by Kajiya and Kay. He generalizes the problem using a concept called the codimension. The codimension is simply the difference between the dimensions of the object and the space it occupies. For example, regular surfaces have a dimension of two and exist in three dimensional space. They have, then, a codimension of 1. Lines, such as blades of grass have a dimension of one and exist in three dimensional space, giving them a codimension of two. Banks provides a normalization technique to account for the anomaly of brightness increasing as the codimension increases. As the codimension increases, the normal space increases, causing most light vectors to be closer to the normal space. The amount of reflected light is directly related to this "closeness" and thus a manifold will appear brighter if it occupies a higher dimensional space. Banks' normalization method simply involves raising Kajiya's diffuse term to a suitable power. While Banks does provide a mathematical justification for which power to choose, the concept of exponentiating the diffuse term is ad hoc and has no physical justification. The results are quite convincing, nevertheless.

Banks also accounts for the problem of self-shadowing. Firstly, he adds a component using the underlying surface's normal to the lighting equation. Specifically, he applies the clamp function to the dot product of the normal and light vectors and multiplies the previously computed specular and diffuse intensities by this amount. Observing that light will attenuate as it passes through the material, he added another self-shadowing term to attenuate the intensity exponentially based on the distance travelled. These self-shadowing terms greatly improved the realism of grassy and furry surfaces.

Goldman [5] added terms to the equations to account for directionality of light. Using the previous equations, backlit hair or fur will receive the same lighting values as hair lit from the front. He first computes the relative direction of the light, eye and tangent vectors by taking the cross product of the tangent and light vectors and the cross

product of the tangent and eye vectors. He then takes the dot product of the resulting two vectors. This result is positive when the strand is frontlit – when the light and eye vectors hit the same side of the strand and negative when the strand is backlit – when the light and eye vectors hit opposite sides of the strand.

Goldman represents the amount of light scattering in the forward and backward directions with simple constants. For example, darker hairs will reflect much more light in the backward direction than they transmit in the forward direction. These constants are chosen, ad hoc, for a given type of hair. The constants are used in combination with the directional characteristic described above to account for directional attenuation, resulting in a more believable lighting model.

Lengyel [11] restates the Kajiya/Kay lighting model. He points out that standard hardware lighting could be made to work if two separate normals were used, one for diffuse and one for specular reflections. The correct normal for diffuse lighting is obtained by projecting the light vector onto the plane perpendicular to the tangent vector. The correct normal for specular lighting is obtained by projecting the specular half vector (the average of the light and eye vectors) onto the plane perpendicular to the tangent vector. However, it was impossible to implement this multi-normal technique on hardware available at the time.

Stalling, et al, [20] rework the equations to eliminate the need to calculate normals. They show that a complete lighting solution can be computed from just two terms - the dot product between the light vector and the tangent vector for diffuse lighting and the dot product between the light direction and the camera vector for specular lighting. While the resulting functions would be difficult to compute in real time, Stalling stores them in a single two-dimensional texture map. The previously mentioned dot products serve as texture coordinates. This is quite an elegant solution, and it provides lighting on a per pixel basis, as opposed to per vertex, which should result in more realism. However, the authors provide no means to handle multiple light sources, or

point light sources with attenuation, although multiple passes should allow for multiple light sources.

Heidrich and Seidel [6] generalize the texture lookup method to a variety of different lighting functions. They also apply Stalling's method to anisotropic surfaces other than lines, such as grooved records or satin Christmas tree ornaments. They leverage hardware lighting in order to evaluate the equations using one pass and one texture per light source.

Chapter 3

Method

3.1 Display

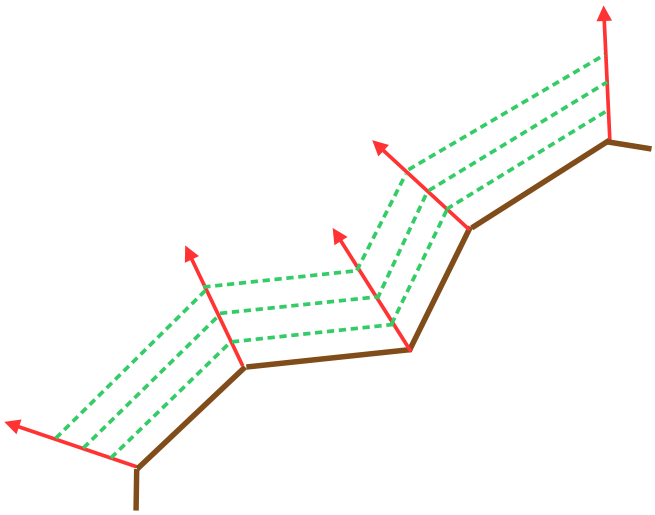

Figure 3.1: Shells are extruded along the normals above the base mesh.

The grass is rendered through Russian-doll style transparent shells. Several copies of the base terrain mesh are "grown out" by displacing the vertices along their associated surface normals in a vertex shader. Alternatively, they could be grown "up" along some constant vector. The shells are transparent except where a blade of grass intersects them. At these points, a cross section of the blade is contained within the texture.

A single texture is used to generate all the shell textures, encoding the "height" of the grass in the alpha channel. The ground has no height and thus all ground texels in the texture have an alpha value of zero. Texels representing grass have non-zero alpha values depending on their respective heights, up to a maximum of 255. As shown in

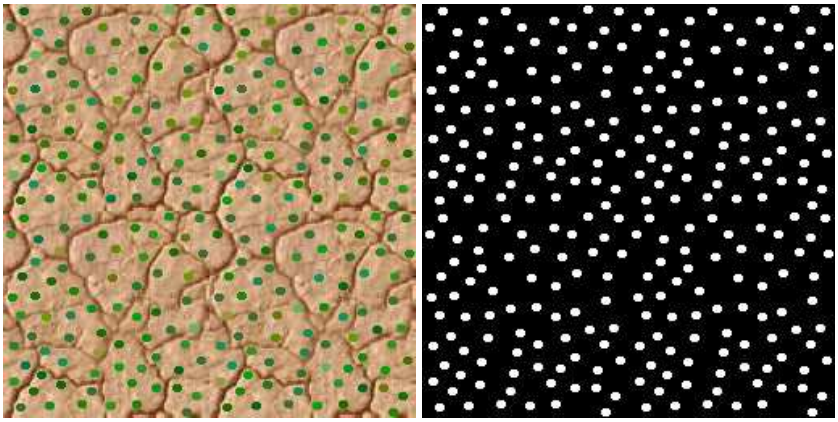

Figure 3.2: The grass texture map with alpha channel. The green dots in the left image are extruded above the surface to form blades of grass. The heights of the blades are encoded in the alpha channel, represented by the white dots in the image at right.

Figure 3.2, the white "dots" in the alpha channel of the texture map correspond to the green dots representing a cross-section of a blade of grass.

The shells are rendered in bottom to top order. The alpha test is enabled and the alpha compare value is set to the shell's height before drawing each shell. The first shell, the base ground mesh, is composed of all the texels in the texture, so the alpha compare value is set to zero (with the alpha test method set to greater than or equal). Before the rendering of each subsequent shell the alpha compare value is set to a higher value. For example, if 10 shells were being rendered, the first shell above the base mesh would be rendered with an alpha compare value of 255/10 = 25. Any blade of grass with a height greater than one tenth of the maximum height would have a cross section included in that shell's texture. Which is to say, any texel with an alpha value greater than or equal to 25 would be rendered in the shell.

Figure 3.3: Close view of grass rendered with 16 shells.

```
Pseudocode:
begin
  enable alpha test
  set alpha test method to greater than or equal
  for i = 0, i < NUM_SHELLS, i++
     set alpha compare value to i/(NUM_SHELLS)*255
     render the shell
  end for
end
```

As Dietrich [4] points out, a state change could be avoided using the alpha channel of the diffuse colour to store the current shell height instead of repeatedly changing the alpha compare value.

3.2 Animation

Animation is implemented by moving each vertex along its "wind vector" – a vector stored with each vertex. Wind vectors are computed in a preprocessing step.

There are several ways of generating appropriate wind vectors, ranging from heuristics to artist painting to a proper fluid dynamics simulation of wind moving over a landscape. This animation algorithm will only consider the contribution of this wind vector that is perpendicular to the local surface normal. This restriction would be easy to overcome with additional per-vertex data, which would, however, degrade the performance slightly. Another good reason to keep the wind vector perpendicular to the local surface normal is that it can be calculated in the vertex shader, and thus not need to be precalculated.

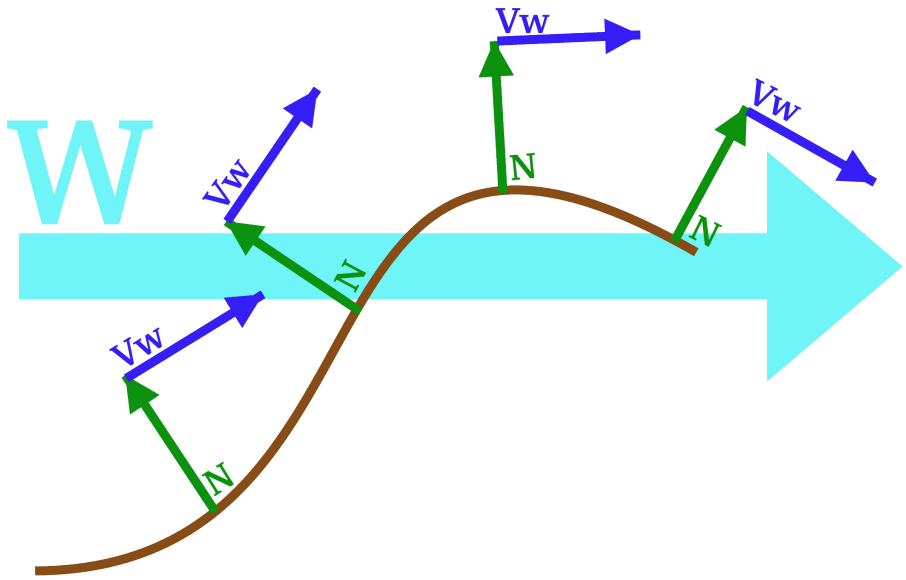

Figure 3.4: Computing the wind vectors. The global wind direction for each of the vertices in this example is given in turquoise. The local wind vectors are in blue and are always perpendicular to the normals (in green).

For the examples in this work I have used a simple point source for the wind. To create the wind vectors, project the vector from each vertex to the wind source onto the plane perpendicular to the normal vector by subtracting from the vector its projection onto the normal.

The wind vector is given by:

$$\vec{V_w} = \vec{W} - \left(\vec{W} \cdot \vec{N}\right) \vec{N}$$

Where V_w is the wind vector at the vertex, W is the vector from the vertex to the wind source and N is the normal vector. All vectors are assumed to be normalized.

Given a wind direction, every vertex is moved along its normal vector and along its wind vector (perpendicular to the normal vector) every frame. The amount moved along these two vectors preserves the inter-shell distance, and thus the length of each blade of grass. In the absence of wind, the distance between shell vertices (along the normal) is a constant equal to the maximum height of the grass divided by the number of shells rendered. With the addition of wind, and thus vertex movement in a direction perpendicular to the normal, this constant inter-shell distance must be preserved or blades of grass will appear to grow and shrink as they animate. Each blade of grass is composed of segments of this constant length that are tilted appropriately in the wind. A windless moment would have all the segments "tilted"at zero degrees, and a moment of maximum wind would have the final segment tilted at 90 degrees. As we move along the blade from bottom to top, the tilt angle increases from zero degrees to a maximum of 90 degrees representing a segment moved into alignment with the wind – parallel to the surface of the landscape. I increment the tilt angle a constant amount between shells. This need not be so, and one could vary the "stiffness" of the grass by changing this increment. "Floppy" grass would do almost all of its bending in the first few shells, whereas stiff grass would do its bending further up the stalk. As shown in Figure 3.5, each segment forms the hypotenuse of a right angle triangle. Therefore, for each shell, the amount to move each vertex along its normal is given by:

$$n_i = \sum_{j=0}^{i} S \cdot \cos(\theta); \quad with\ \theta = \frac{i}{N} \cdot \frac{\pi}{2} \cdot I$$

Where i is the current shell, I is the current wind intensity at this vertex, S is the inter-shell distance and N is the total number of shells being rendered.

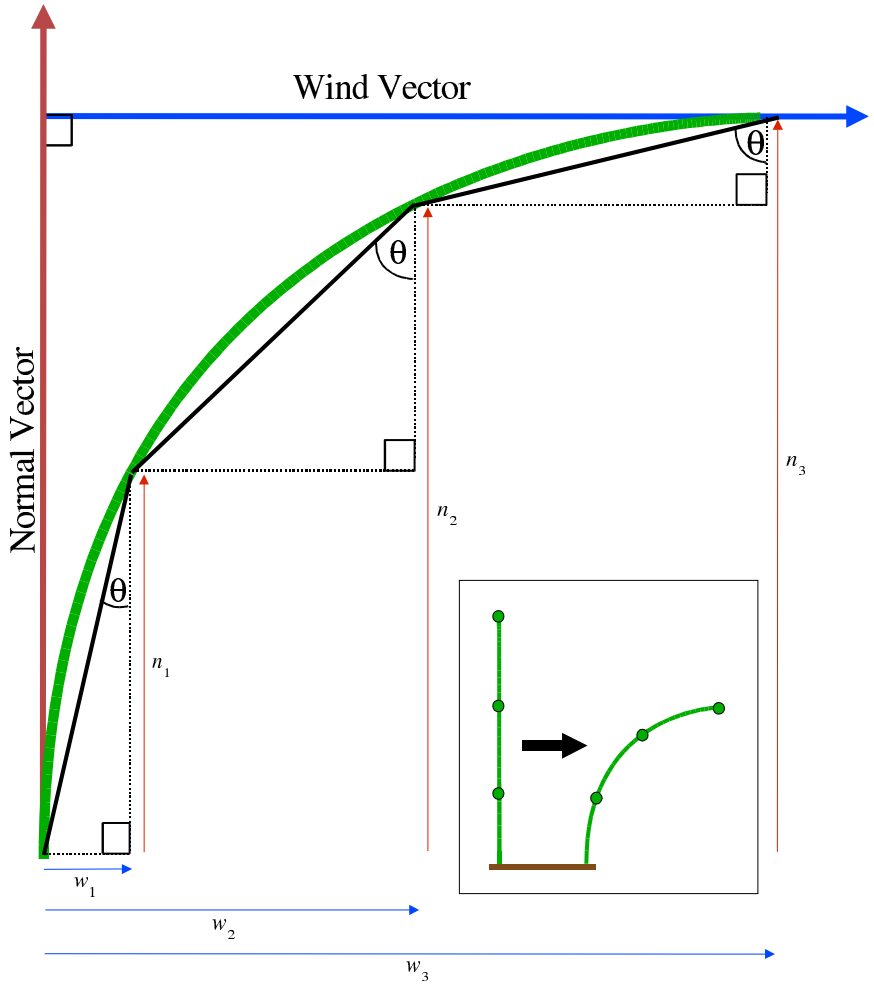

Figure 3.5: The length-preserving function to bend a blade of grass. This blade of grass is rendered with four shells, including the ground. The blade is composed of segments of constant length (the hypotenuses of the right angle triangles) which are tilted in response to the wind intensity. A shell is created by moving each vertex a distance along its normal (n_i) and along its wind vector (w_i).

Similarly the amount to move each vertex along its wind vector is:

$$w_i = \sum_{j=0}^{i} S \cdot \sin(\theta); \quad with\ \theta = \frac{i}{N} \cdot \frac{\pi}{2} \cdot I$$

Since the shells are drawn in bottom to top order, a running total is kept and the sums need not be recalculated for each shell. Each blade of grass, then, is the sum of its segments.

The wind intensity can be represented by any function the user chooses. For best results it should be periodic, continuous and vary between -1.0 and 1.0. Values outside this range would cause "over-bending" in the grass, which in some cases may actually be a desired effect. For this work I implemented several different wind functions including a cubic function and a spline. Not surprisingly, it is easier to fashion the spline into an appropriate function and thus it makes for more convincing results. After Neyret [16], I created a function with a strong attack (a steep up-slope) and a gradual falloff (a gentle down-slope). This function results in quite convincing gusts of wind across the landscape.

Position	Wind Vector	Texture Coordinates
x	x	u
y	y	v
z	z	Animation Step
w	w	a

Table 3.1: Data submitted with each vertex.

The final piece of data sent along with each vertex is an integer representing its animation "step". The world is divided into segments based on their distance from the wind source. Blades of grass close to the source will experience the effects of a sudden spike in wind intensity before distant blades. This allows for a more realistic animation – including "waves" moving across the field, or attenuation of wind intensity.

The vertices are moved in a vertex shader. Values representing the amount of movement along the wind vector and the normal are stored in registers for each of the animation steps present. The shader references the values using the integer sent with each vertex, multiplies the appropriate vector with each value and and adds the result to the vertex position. This process is repeated for every shell rendered. The movement values must change as we travel up a blade of grass – the tip will be affected by the wind more than the base near the root.

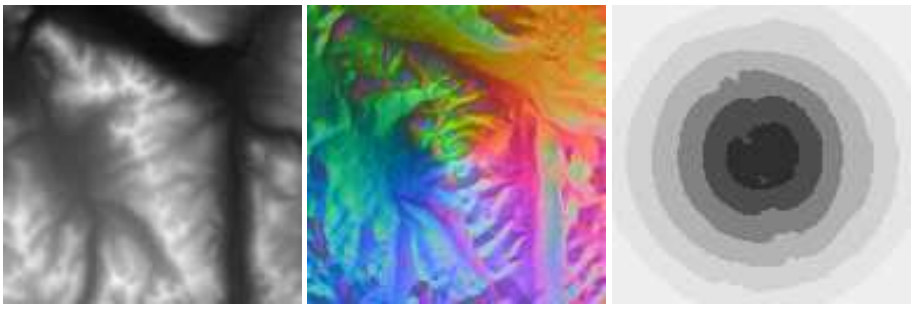

Figure 3.6: Images representing wind vectors. The wind source is roughly in the centre of the images. The original height map is shown at left. The X,Y and Z components of the vector are mapped to the red, green and blue components, respectively, of the centre image. The animation step to which each vertex belongs, in this case representing its distance from the wind source, is shown at right.

Variable	Type	Frequency
S	float	once
AnimStep	int	pre-process
Normal	4-vec	pre-process
Wind Vector	4-vec	pre-process
Vertex Pos	4-vec	each frame, each shell, each vertex
n_i,w_i,θ	float	each frame, each AnimStep, each shell
I	float	each frame, each AnimStep

Table 3.2: Computation frequencies.

3.3 Lighting

3.3.1 Diffuse

The first step in creating a believable lighting model is accounting for surfaces that reflect light in a diffuse manner. This has been a tried and true part of the standard lighting equation in computer graphics for many years. An ideally diffuse surface (with no specular highlights) follows Lambert's Law: the intensity is determined by the cosine of the angle between the surface normal and the vector to the light. One of the goals here is to make use of graphics hardware, and this diffuse equation would be a good candidate, as most recent video cards can evaluate it

directly. The problem is the normal vector. There is no actual geometry being being rendered, and thus we don't have proper normals for the blades of grass. The normals of the shells are equivalent to the tangent vectors of the blades and are perpendicular to the normals we need.

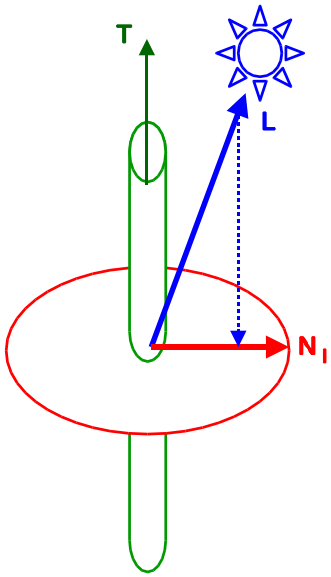

Figure 3.7: Diagram of the diffuse light equation.

Modern computer graphics hardware is flexible enough that it is possible to derive the proper normal on the fly using vertex shaders. The method I use is the one first proposed by Kajiya and Kay [7] and expanded upon by Banks [1]. This technique was originally used for ray tracing in non-interactive applications. It is an ad hoc technique, without a solid foundation in physics. Each segment of each blade of grass is treated as a very thin cylinder, with an infinite number of normals projecting out in all directions perpendicular to the tangent vector. The tangent vector of each segment is obtained by submitting the previous shell's bend constants along with the current set. From these the position of the previous shell's vertex is determined, and the tangent vector obtained by subtracting the previous position from the current one.

The most appropriate normal, the normal that will give the maximum intensity of reflected light, is determined by projecting the light vector onto the plane of normals to the blade of grass

by subtracting from the light vector its projection onto the tangent.

For diffuse lighting, then, the projected light vector is given by:

$$\overrightarrow{N_l} = \vec{L} - \left(\vec{T} \cdot \vec{L}\right) \vec{T}$$

Where N_l is the projection of the vector to the light onto the normal disk, L is the vector from the vertex to the light and T is the tangent vector. All vectors are assumed to be normalized. The intensity of the diffusely reflected light is equal to the cosine of the angle between this normal and the light vector.

The dot product of the normal and the light vector is given to the LIT function in the vertex shader.

3.3.2 Specular

Diffuse surfaces are not sufficient for a believable lighting model. Another component is required to describe shiny surfaces - the specular component. Specular lighting describes the shape and intensity of highlights on the surface reflected toward the eye of the viewer. This is especially true of anisotropic surfaces such as grass or fur, which have a distinctive specular reflectance. A cursory examination of hair or a fabric-wound Christmas tree ornament reveals that the shape of the highlights and their behaviour as the surface moves are considerably different than in the isotropic case.

While evaluating a complex BRDF that accurately describes the physical properties of specular highlights in grass is not possible in real-time, very convincing results can be obtained with an ad hoc model. The Kajiya/Kay [7] or Banks [1] model is quite useful. Originally intended for non-interactive ray tracing applications, modern shaders allow its use in real-time. The commonly-used Blinn approximation of specular reflection involves computing the half-vector between the vector to the light and the vector to the viewer. This is given by:

$$H = \frac{\vec{L} + \vec{V}}{||\vec{L} + \vec{V}||}$$

The half-vector is the direction of maximum highlights, which is to say that the intensity of the specular reflection will be maximized when the normal vector is the same as the half-vector.

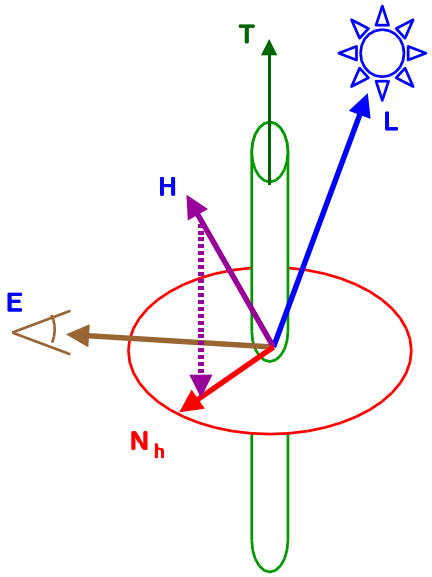

Figure 3.8: Diagram of the specular light equation.

This intensity falls off sharply as the two vectors diverge, and this falloff is usually expressed as the cosine of the angle between the the two vectors raised to some arbitrary power. Again it is important to understand these fundamental lighting equations in order to leverage their implementations in current consumer graphics hardware. Hardware support requires the cosine of the angle between the half-vector and the normal vector. As explained with respect to diffuse lighting above, in the plane perpendicular to the tangent vector, there are an infi nite number of vectors to choose from to be the normal vector. The most appropriate vector, the vector resulting in the maximum specular intensity is given by projecting the half-vector onto this plane by subtracting from the half-vector its projection onto the tangent vector. For specular lighting, then, the new normal vector is given by:

$$\overrightarrow{N_h} = \vec{H} - \left(\vec{T} \cdot \vec{H}\right) \vec{T}$$

Where N_h is the projection of the half vector onto the normal disk, H is the specular half vector and T is the tangent vector. All vectors are assumed to be normalized. The dot product of

the normal and the half-vector is given to the LIT function in the vertex shader along with the specular exponent.

3.3.3 Self-Shadowing

An important factor in the believability of the lighting model is shadows. Viewers are quite familiar with shadows in the natural world. Without some attempt at representing shadows, the grass can be difficult to distinguish and "flat" in appearance. However, a field of grass is very complex, and modelling the actual shadows cast from blades of grass onto each other is not possible in real-time. This is further complicated by the lack of actual geometry in this system - there are no "real" blades of grass to compute shadows from.

I use an approximation proposed by Banks [1] which involves a simple observation: the closer a portion of a blade is to the ground, the more likely it is to be in shadow, and thus the less light it should receive from any lights lighting the scene.

A reasonable approximation to actual shadows can be obtained with a simple linear ramp based on the vertex's height above the surface. The closer it is to the maximum height, the topmost shell, the more light it receives. Visually, it improves the image if the minimum amount of light is clamped to some value. After the intensity of light at the vertex is determined using the specular and diffuse equations above, this intensity is multiplied by a self-shadowing term. This term is given by:

$$S = \frac{D_{cur}}{D_{max}} * (1 - S_{min}) + S_{min}$$

Where S is the self-shadowing term, D_{cur} is the current distance above the ground, D_{max} is the maximum distance above the ground (the height of the topmost shell) and S_{min} is the clamp on the smallest possible self-shadowing term.

3.3.4 Attenuation

Finally, for believable point light sources, attenuation of the light intensity over distance must be taken into account. The atmosphere scatters light as it passes through it, and thus the intensity of reflected light depends upon the distance from the light to the surface. In reality, the energy that

reaches a surface falls off as the inverse square of the distance. However, as is often the case in computer graphics, the physically correct result is not necessarily the most desirable. Often we are using a point source light to simulate another type of light, or for aesthetic reasons, we want more control over light attenuation.

It is common in computer graphics to represent the attenuation of light over distance as the sum of three functions; a constant attenuation, a linear attenuation and a quadratic attenuation. These functions are weighted as the user chooses. This convention has support in hardware. The DST vertex program instruction returns not only the distance, but the distance squared. To compute the attenuation, we need simply store the weights in a vector and compute the dot product of this vector with the vector returned by the DST instruction. Taking the reciprocal of this value gives us the amount by which to scale the reflected intensity.

Also inherent in the standard graphics lighting equation is a constant factor used to approximate ambient light. The actual interaction of light with objects in a room is infinitely complex, and not representable in real time. Ambient light is a gross simplification and simply represents the minimum amount of light that any surface in a scene will receive. It is important to note that the diffuse component should be added after the attenuation, as diffuse light is not dependent upon any one light source.

3.4 MIP Maps

Most current consumer graphics hardware supports MIP maps as a method of antialiasing textures. MIP maps are an example of a rare occurrence in computer graphics - a technique that improves visual quality and increases rendering speed, albeit at the cost of memory. To compute the colour of a texture mapped pixel, the pixel is back projected onto the texture. Often there is not a one to one relationship between pixel and texels - specifically, problems arise when the pixel projects onto several texels. Severe aliasing can occur if the nearest texel is chosen, and filtering or combining the affected texels can be quite slow, especially if a large number are involved.

MIP maps were introduced to solve both problems. Essentially, many versions of the texture are created, each one a quarter the size of the previous one until one of the dimensions of the

texture is a single texel in size. These smaller subtextures can be created by a number of means, and perhaps the simplest and most common is the simple application of a box filter - the four texels of the parent image are averaged to form to resultant single texel in the child image. During rendering, when a pixel back projects onto several texels, a lower MIP map level is chosen with the goal of a pixel to texel ratio of 1:1.

Such a scheme will not directly work with the displacement map textures used in this work. Box filtering would be highly inappropriate - the textures used are characterized by high frequency details, namely the dots representing cross sections of the blades of grass. Averaging texels will result in the loss of the cross sections as they are blended with the non-grass portions of the texture. The texture in Figure 3.2 would have its green dots blended with the surrounding brown texels representing the ground. The cross-sections would disappear and thus, the blades themselves would disappear. Additionally, the alpha values of the four texels would be blended. Since the height to displace each texel is encoded in the alpha channel, the very topology of the displacement map would change.

Certainly better filtering techniques than a box filter can be applied, but any that do not specifically preserve the cross sections will not be appropriate and it seems unlikely that any general-purpose filter that is unaware of the cross sections can be guaranteed to do so. For the purpose of creating MIP maps, a custom filter was used. The filter scans the alpha values of the texture map and identifies cross sections (dots) by their non-zero alpha values. All the contiguous texels in each dot are identified and the u and v extents of the dot are determined. Then this dot space is filtered. Each two by two texel square maps to a single texel in the next MIP map level. If two or more of the four texels belong to the dot (ie. have non-zero alphas) the lower level texel is a dot texel. This texel will have the average of the colours and the maximum of the alpha values of the dot texels. If this is not the case, the lower level is not a dot texel. Its alpha value is set to zero and its colour is simply an average of the 4 texels in the level above. Better techniques could certainly be applied, but are not necessary, because the errors introduced in the non-dot texels of lower MIP map levels are not noticeable due to the surfaces' distance from the viewer.

This process is repeated until a cross-section is not representable in the lower level MIP map. At this point, no further levels may be generated.

This difficulty with MIP maps is similar to the one faced by Klein [9]. Klein used simulated

brush strokes to render non-photorealistic environments as if they had been painted. These brush strokes were contained in textures, and these textures were placed on three dimensional geometry. This had the unfortunate side effect of making brush strokes on surfaces far from the viewer appear smaller. This breaks the illusion of a painting, in which brush strokes should be more or less the same size on the screen, no matter where they appear in the three dimensional space of the scene. Klein solved the problem by creating custom MIP maps, called art maps. The strokes in each MIP map level vary in size by powers of two just as the size of the MIP maps themselves change. This keeps the size of the brush strokes in screen space relatively constant, but does not preserve the strokes themselves between levels. Klein's technique is not suitable here because the dots on the textures must be preserved between MIP map levels or the illusion of blades of grass will be lost, but the custom MIP maps generated to solve the problem are very much in the same tradition.

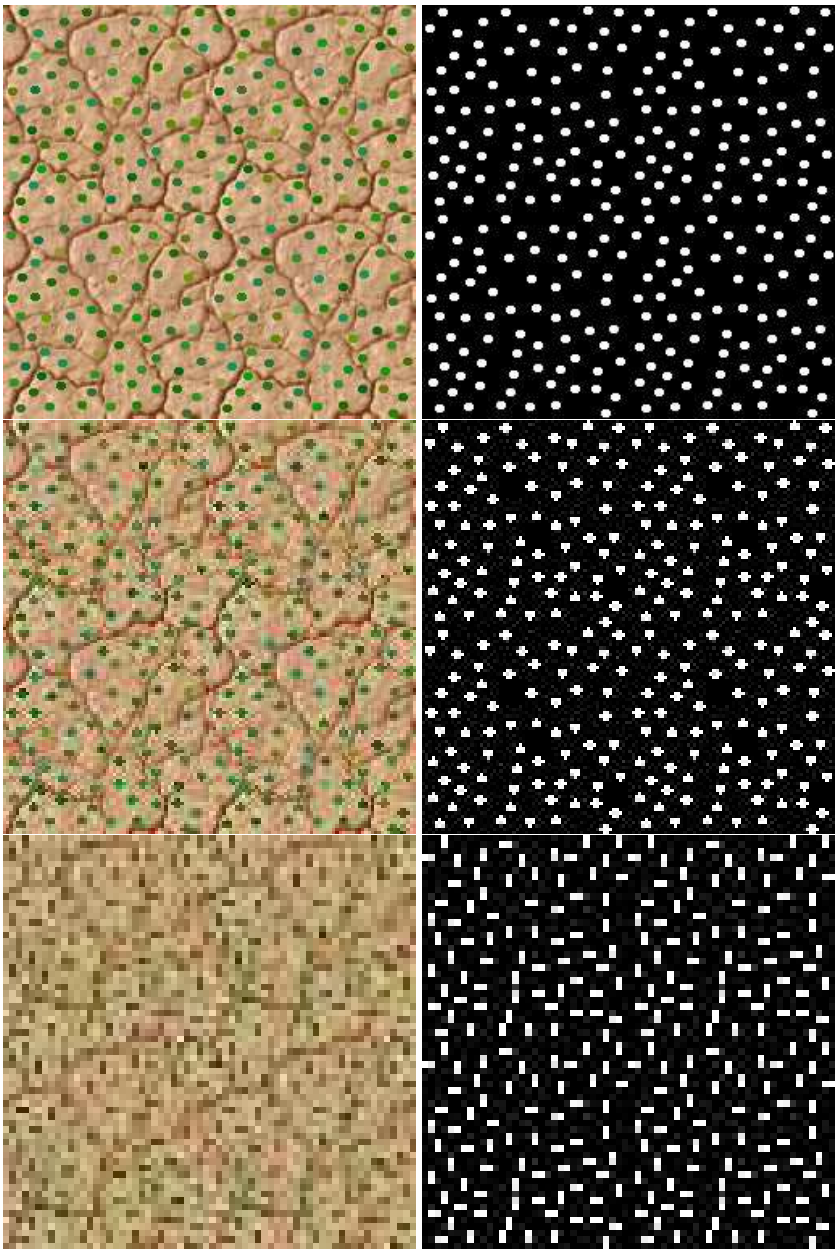

Figure 3.9: MIP maps. A sequence of custom MIP maps with the RGB values on the left and the alpha values on the right. From top to bottom: the original 256 x 256 texture, the first MIP map level of 128 x 128, and the second MIP map level of 64 x 64.

Chapter 4

Results

4.1 Performance

The test machine for the following results has a 1.7 Ghz Intel Pentium processor with 512 Megabytes of memory. The video card is a 32 Megabyte NVidia GeForce3. All tests were done in 32 bit colour with a 16 bit Z-buffer. EAGL, a proprietary graphics API of Electronic Arts, Inc., as described by Lalonde and Schenk [10], was used for the rendering.

Frame rates in this application vary depending on the coverage of the landscape on the screen. These numbers represent near worst-case values. The frame rate increases as one pulls away from the landscape, causing the landscape to cover less of the screen, indicating a fi ll rate limitation to the method. However, when close to the landscape, the frame rate also increases when some geometry can be culled. The screen coverage of the landscape for the tests can be seen in Figure 4.1.

No attempt at optimization has been made, so these results could likely be improved upon. Fill rate capabilities of consumer graphics hardware continue to increase as well. This technique has already proven itself useful for consumer games.

		Resolution	
Shells	Triangles	640 x 480	1024 x 768
1	512	293.5	130.8
9	4608	47.0	22.8
17	8704	25.6	12.4
33	16896	13.4	6.5
65	33280	6.9	3.3

Table 4.1: Results in frames per second.

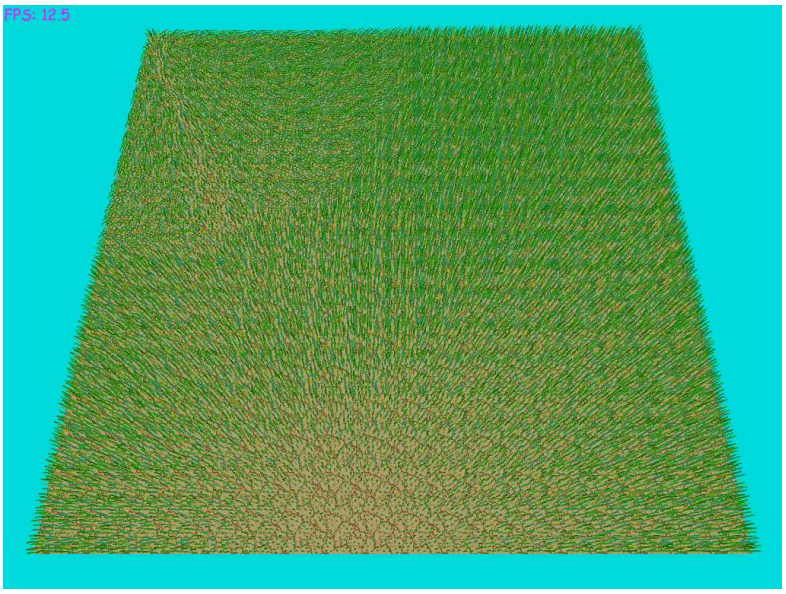

Figure 4.1: Screen coverage for benchmarking.

Not surprisingly, this technique is limited by the fi ll rate of the graphics hardware. Adding shells, or increasing resolution, has a larger negative impact on performance than adding geometry to the scene. Adding shells does, of course, add geometry as well, and frame rate increases were observed when some of the geometry can be culled. As Lengyel noted [11], a covering of grass or fur allows for a much less detailed base mesh than would otherwise be needed. The complexity of the base mesh is, to a large extent, lost when grown out into transparent shells.

The fi ll rate limitation of the technique essentially makes the lighting "free". No perceptible change was noticed when the lighting code for 3 point lights was added to the vertex shader.

4.2 Quality

The contribution of this work lies mainly in visual quality - the performance is real-time, but those numbers can be manipulated as the user chooses. Decreasing the number of shells, or the complexity of the underlying mesh will increase frame rate. There are undoubtedly countless optimizations possible as well.

Rendering the grass as concentric transparent shells is quite useful. This technique's ef-

fectiveness is independent of the number of blades of grass in a scene, which is an important consideration given the complexity of a field of grass. However, using shells leads to problems at grazing angles where viewers can see "through" the blades themselves. For grass viewed from the top, as in a flight simulator, or relatively short grass, the technique requires no modifications. In other cases, steps would have to be taken to compensate, for example, generating the shells in a manner that makes them perpendicular to the viewer. Kautz [8] has shown some results in this area, although he notes that his technique will not work with highly detailed displacement maps, such as those used here. It may be possible to modify his technique, however. Lengyel [11] used "fin" textures to help with silhouette edges, where the "see through" problem is the most noticeable. This technique will not work without some modification as well, however, because these fin textures would need to be dynamically generated in order to account for the animation of the grass.

The illusion of real blades of grass is quite convincing, and it is greatly helped by the animation. Dividing the vertices into animation frames was quite successful as it allows for very complex patterns. It is very important to preserve the length of the blades of grass as they animate, as the viewers willingness to suspend disbelief is easily lost. A gust across the prairie can be quite believable as shown in figure A.1. A limitation of the technique is that blades of grass can never intersect, or blow in opposing directions past each other. As the vertices move closer to each other, the blades of grass simply "squish" together.

As important as the animation is the lighting. In order to maintain the illusion, the viewer must be able to pick out individual blades of grass. A number of techniques were applied to facilitate this, the simplest of which was to make the blades be slightly different colours, by having the cross section dots on the texture be different colours. Another technique is the Bank's self-shadowing approximation. This had a two-fold effect. Firstly it made it easier to distinguish adjacent blades. As they bend together, the differences in lighting, based on how high they are from the base mesh, help accentuate the individual blades of grass. Also, the lighting model is simply more realistic with self-shadowing. The viewer expects grass near the ground to be darker than grass farther up.

For the point source lights to be credible, attenuation had to be taken into account. Fortunately, anisotropy has no bearing on attenuation which is simply based on the distance between

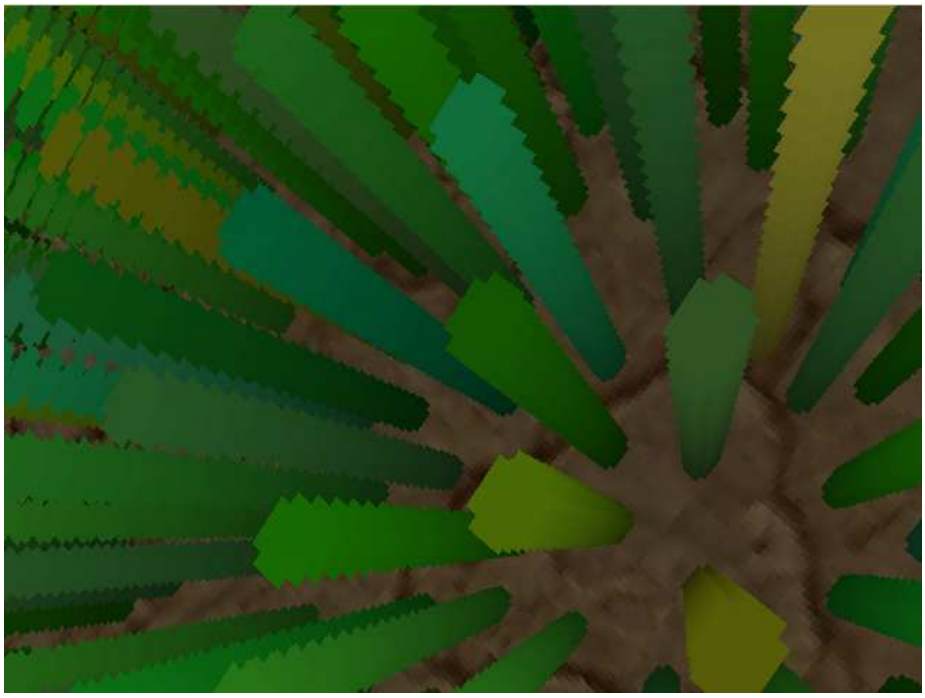

Figure 4.2: The Banks approximation of self-shadowing. It is a simple linear ramp from a clamped minimum value to allowing the full amount of reflected light.

the vertex and the light. Thus, built-in hardware techniques to compute the distance were easily leveraged to create a quite believable effect.

Perhaps the most important factors in creating the illusion of grass blowing in the wind were the diffuse and specular reflection terms. People see anisotropic surfaces reflecting light every day, from hair commercials on television to Christmas ornaments hanging from trees, and are astute in detecting if a surface does not look right. Often, anisotropic reflection is painted in to the hair of models in an attempt at believability. Now, this lighting can be done in real time and the results are quite satisfying. Although the Kajiya/Kay model is not physically based, it produces impressive results. In fact, proper specular and diffuse reflection is probably more important in creating a believable sense of animation than the movement of the shells themselves. It is difficult to show the dynamic effect created by the animation and lighting in real-time, in this paper, but please refer to the sequence shown in figure A.1.

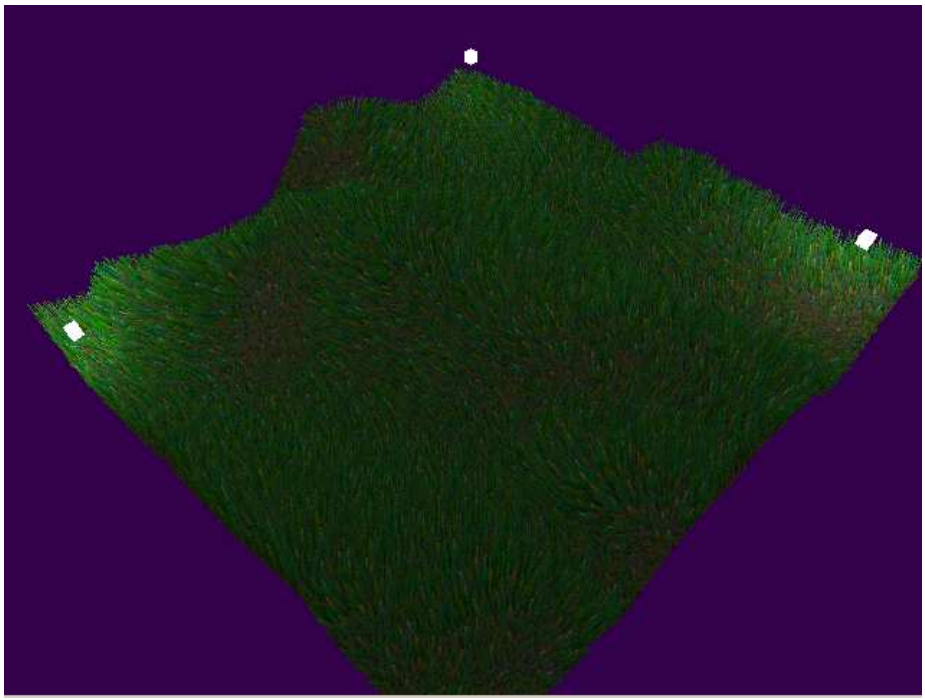

Figure 4.3: Point source lights with attenuation.

Bibliography

[1] David C. Banks. Illumination in diverse codimensions. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 327–334. ACM Press, 1994.

[2] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, pages 21–29, 1982.

[3] Robert L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, 1984.

[4] Sim Dietrich. Elevation maps. Technical report, NVidia Corporation, 2000.

[5] Dan B. Goldman. Fake fur rendering. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 127–134. ACM Press/Addison-Wesley Publishing Co., 1997.

[6] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 171–178. ACM Press/Addison-Wesley Publishing Co., 1999.

[7] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. *Computer Graphics*, 23(3):271–280, July 1989.

[8] Jan Kautz and Hans-Peter Seidel. Hardware accelerated displacement mapping for image based rendering. In *Proceedings of Graphics Interface*, pages 61–70, June 2001.

[9] Allison W. Klein, Wilmot Li, Michael M. Kazhdan, Wagner T. Corr?a, Adam Finkelstein, and Thomas A. Funkhouser. Non-photorealistic virtual environments. In *Proceedings of*

the 27th annual conference on Computer graphics and interactive techniques, pages 527–534. ACM Press/Addison-Wesley Publishing Co., 2000.

[10] Paul Lalonde and Eric Schenk. Shader-driven compilation of rendering assets. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 713–720. ACM Press, 2002.

[11] Jerome Lengyel. Real-time fur. In *Proceedings of the Eurographics Workshop On Rendering*, pages 243–256, June 2000.

[12] Jerome Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 227–232. ACM Press, 2001.

[13] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. In George Drettakis and Nelson Max, editors, *Eurographics Rendering Workshop 1998*, pages 157–168, New York City, NY, July 1998. Eurographics, Springer Wein. ISBN 3-211-83213-0.

[14] G. S. P. Miller. From wire-frames to furry animals. In *Proceedings on Graphics interface '88*, pages 138–145. Canadian Information Processing Society, 1988.

[15] F. K. Musgrave, C. E. Kolb, and R. S. Mace. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 41–50. ACM Press, 1989.

[16] Fabrice Neyret. Animated texels. In Dimitri Terzopoulos and Daniel Thalmann, editors, *Computer Animation and Simulation '95*, pages 97–103. Eurographics, Springer-Verlag, September 1995. ISBN 3-211-82738-2.

[17] Darwyn R. Peachey. Solid texturing of complex surfaces. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 279–286. ACM Press, 1985.

[18] Frank Perbet and Maric-Paule Cani. Animating prairies in real-time. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 103–110. ACM Press, 2001.

[19] Ken Perlin. An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 287–296. ACM Press, 1985.

[20] D. Stalling, M. Zöckler, and H.-C. Hege. Fast display of illuminated fi eld lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, 1997.

Appendix A

Animation Sequence

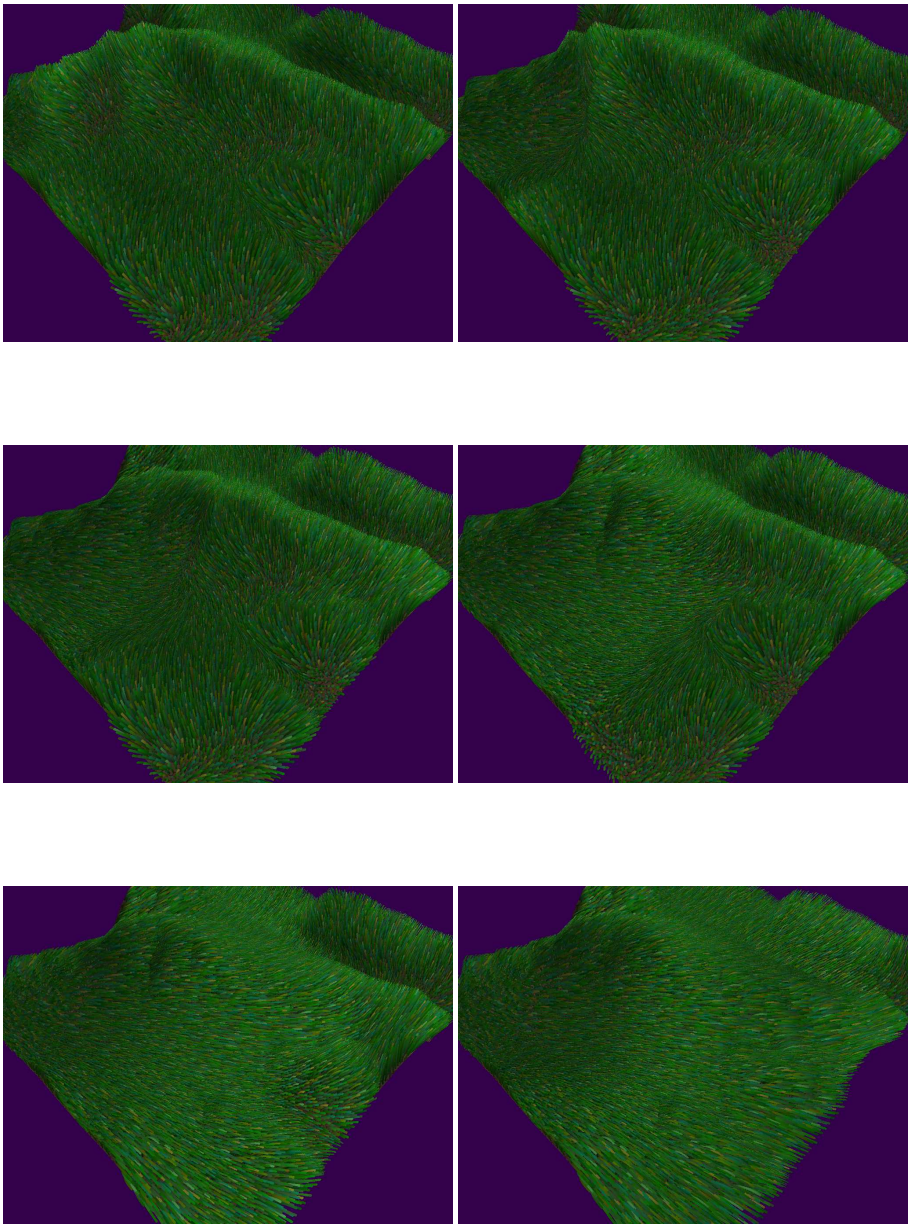

Figure A.1: A gust of wind across the prairie. These frames show a wind gust originating on the left side of the screen and moving toward the right side of the screen. Frame order: top left, top right, middle left, middle right, bottom left, bottom right.

Appendix B

Shader Source

```
#define _WVP VAR0
#define POSITION  VAR1
#define VCOLOUR  VAR2
#define TEXCOORD  VAR3
#define NORMAL  VAR4
#define MODELMATRIX  VAR5
#define LIGHTBLOCK   VAR6
#define LIGHTCONST  VAR7
#define SELF_SHADOW VAR8
#define WINDDIR VAR9
#define ANIMFRAME VAR10
#define BENDCONSTS VAR11
#define _WV VAR12
#define CAMERAPOS VAR13  ;not used
#define ATTENUATION  VAR14
#define WVIT VAR15

#define LIT  r0
#define TMP r1
#define NEWPOS r2
#define NEWNORM r3
#define VERT_TO_LIGHT  r4
#define TMP_ATTEN r5
#define HALF_VECTOR  r6
#define DIFFUSE r7
#define SPECULAR r8
#define VERTDATA r9
#define WINDDIST  x
#define NORMDIST y
#define OLDWINDDIST  z
```

```
#define OLDNORMDIST  w

#define EYE_TO_VERT  r11

#define LIGHTRESULT3  r7

; Define the offsets into the lightblock
#define LIGHT1POS c[a0.x + 0 + BASE6]
#define LIGHT2POS c[a0.x + 1 + BASE6]
#define LIGHT3POS c[a0.x + 2 + BASE6]
#define LIGHTCOLOUR1 c[a0.x + 3 + BASE6]
#define LIGHTCOLOUR2 c[a0.x + 4 + BASE6]
#define LIGHTCOLOUR3 c[a0.x + 5 + BASE6]
#define AMBIENTCOLOUR c[a0.x + 6 + BASE6]

; ------------------------------------------------------
; Extrude vertex along the normal by offset.y
; ------------------------------------------------------

mov a0.x,ANIMFRAME.x
mov VERTDATA,c[a0.x + BASE11]
mul NEWPOS,VERTDATA.NORMDIST,NORMAL
add NEWPOS,POSITION,NEWPOS

; ------------------------------------------------------
; Move vertex along the Wind Vector by offset.x
; ------------------------------------------------------
mad NEWPOS.xyz,VERTDATA.WINDDIST,WINDDIR,NEWPOS

; ------------------------------------------------------
```

```
; Compute the new normal for the segment of grass
; (Tangent vector)
; ------------------------------------------------------
; Find out where the old position was
mul TMP,VERTDATA.OLDNORMDIST,NORMAL
add TMP,POSITION,TMP
mad TMP.xyz,VERTDATA.OLDWINDDIST,WINDDIR,TMP
; Normal is new position minus old position
add NEWNORM,-NEWPOS,TMP

; ------------------------------------------------------
; Set the address register and output the TEXCOORD
; ------------------------------------------------------
mov a0.x, LIGHTCONST.x
mov oT0, TEXCOORD

; ------------------------------------------------------
; Transform Coordinates into clip space
; ------------------------------------------------------
dp4 oPos.x, NEWPOS, c[0 + BASE0]  ; WVP
dp4 oPos.y, NEWPOS, c[1 + BASE0]  ; WVP
dp4 oPos.z, NEWPOS, c[2 + BASE0]  ; WVP
dp4 oPos.w, NEWPOS, c[3 + BASE0]  ; WVP

; ------------------------------------------------------
; Transform Coordinates into camera space
; ------------------------------------------------------
dp4 TMP.x, NEWPOS, c[0+ BASE12] ; _WV Matrix
dp4 TMP.y, NEWPOS, c[1+ BASE12] ; _WV Matrix
dp4 TMP.z, NEWPOS, c[2+ BASE12] ; _WV Matrix
```

```
dp4 TMP.w, NEWPOS, c[3+ BASE12] ; _WV Matrix
mov NEWPOS, TMP ; Can prolly get rid of this ins if need be

; ------------------------------------------------------
; Move NORMAL into camera space and NORMALIZE it
; ------------------------------------------------------
;Transform Normals into camera space
;The following NEWNORMS were NORMAL
dp4 TMP.x, NEWNORM, c[0+ BASE15] ; WVit
dp4 TMP.y, NEWNORM, c[1+ BASE15] ; WVit
dp4 TMP.z, NEWNORM, c[2+ BASE15] ; WVit
mov NEWNORM, TMP ; Can prolly get rid of this ins if need be

;Re-normalize the normal
dp3 NEWNORM.w, NEWNORM, NEWNORM
rsq NEWNORM.w, NEWNORM.w
mul NEWNORM, NEWNORM, NEWNORM.w

; ------------------------------------------------------
; Create vector from camera to vertex and NORMALIZE it
; ------------------------------------------------------
dp3 EYE_TO_VERT.w, NEWPOS, NEWPOS
rsq EYE_TO_VERT.w, EYE_TO_VERT.w
mul EYE_TO_VERT, -NEWPOS, EYE_TO_VERT.w

; Copy in the material power for the "lit" instruction
mov LIT.w,SELF_SHADOW.w

;******************************************************
;******************** LIGHT 1 ***************************
```

```
;*******************************************************

; -------------------------------------------------------
; Build Vector from Vertex to Light and NORMALIZE it
; -------------------------------------------------------
add VERT_TO_LIGHT,LIGHT1POS,-NEWPOS
dp3 TMP.w,VERT_TO_LIGHT,VERT_TO_LIGHT
rsq VERT_TO_LIGHT.w,TMP.w

; -------------------------------------------------------
; Get the attenuation
; -------------------------------------------------------
dst TMP,TMP.w,VERT_TO_LIGHT.w
dp3 TMP.w,TMP,ATTENUATION
rcp TMP_ATTEN.w, TMP.w

; Normalize vertex to light vector
mul VERT_TO_LIGHT, VERT_TO_LIGHT,VERT_TO_LIGHT.w

; -------------------------------------------------------
; calculate and NORMALIZE the DIFFUSE NORMAL
; -------------------------------------------------------
; NORMdiff = L - (T.L)T  where T = Tangent, or Normal
dp3 TMP.w, NEWNORM, VERT_TO_LIGHT
mad TMP,-TMP.w,NEWNORM,VERT_TO_LIGHT
dp3 TMP.w,TMP,TMP
rsq TMP.w,TMP.w
mul TMP,TMP,TMP.w

; -------------------------------------------------------
```

```
; DOT this vector with LIGHT to get diffuse intensity
; ------------------------------------------------------
dp3 LIT.x,TMP,VERT_TO_LIGHT

; ------------------------------------------------------
; Calculate the Half Vector (Eye + Light)/2
; (don't need to divide)
; ------------------------------------------------------
add HALF_VECTOR, EYE_TO_VERT, VERT_TO_LIGHT
dp3 HALF_VECTOR.w,HALF_VECTOR,HALF_VECTOR
rsq HALF_VECTOR.w,HALF_VECTOR.w
mul HALF_VECTOR,HALF_VECTOR,HALF_VECTOR.w

; ------------------------------------------------------
; Project Half Vector onto Tangent Disk to get SPECULAR NORMAL
; ------------------------------------------------------
; NORMspec = H - (T.H)T where H is half vector and T is Normal
dp3 TMP.w, NEWNORM,HALF_VECTOR
mad TMP,-TMP.w,NEWNORM,HALF_VECTOR
dp3 TMP.w,TMP,TMP
rsq TMP.w,TMP.w
mul TMP,TMP,TMP.w

; ------------------------------------------------------
; DOT this vector with HALF VECTOR to get specular intensity
; ------------------------------------------------------
dp3 LIT.yz,TMP,HALF_VECTOR

; ------------------------------------------------------
; Calculate diffuse and specular factors (magical LIT ins)
```

```
; ------------------------------------------------------
lit TMP,LIT

; ------------------------------------------------------
; Scale the factors by the attenuation
; ------------------------------------------------------
mul TMP, TMP, TMP_ATTEN.w

; ------------------------------------------------------
; Add the AMBIENT colour
; ------------------------------------------------------
mov DIFFUSE,AMBIENTCOLOUR

; ------------------------------------------------------
; Add the DIFFUSE colour (* diffuse intensity)
; ------------------------------------------------------
mad DIFFUSE, LIGHTCOLOUR1,TMP.y,DIFFUSE
; FIXME: have separate DIFFUSE and SPECULAR colours

; ------------------------------------------------------
; Add the SPECULAR colour (* specular intensity)
; ------------------------------------------------------
mul SPECULAR, LIGHTCOLOUR1,TMP.z



;*******************************************************
;******************** LIGHT 2 ***************************
;*******************************************************

; ------------------------------------------------------
```

```
; Build Vector from Vertex to Light and NORMALIZE it
; ------------------------------------------------------
add VERT_TO_LIGHT,LIGHT2POS,-NEWPOS
dp3 TMP.w,VERT_TO_LIGHT,VERT_TO_LIGHT
rsq VERT_TO_LIGHT.w,TMP.w

; ------------------------------------------------------
; Get the attenuation
; ------------------------------------------------------
dst TMP,TMP.w,VERT_TO_LIGHT.w
dp3 TMP.w,TMP,ATTENUATION
rcp TMP_ATTEN.w, TMP.w

; Normalize vertex to light vector
mul VERT_TO_LIGHT, VERT_TO_LIGHT,VERT_TO_LIGHT.w

; ------------------------------------------------------
; calculate and NORMALIZE the DIFFUSE NORMAL
; ------------------------------------------------------
; NORMdiff = L - (T.L)T  where T = Tangent, or Normal
dp3 TMP.w, NEWNORM, VERT_TO_LIGHT
mad TMP,-TMP.w,NEWNORM,VERT_TO_LIGHT
dp3 TMP.w,TMP,TMP
rsq TMP.w,TMP.w
mul TMP,TMP,TMP.w

; ------------------------------------------------------
; DOT this vector with LIGHT to get diffuse intensity
; ------------------------------------------------------
dp3 LIT.x,TMP,VERT_TO_LIGHT
```

```
; ------------------------------------------------------
; Calculate the Half Vector (Eye + Light)/2
; (don't need to divide)
; ------------------------------------------------------
add HALF_VECTOR, EYE_TO_VERT, VERT_TO_LIGHT
dp3 HALF_VECTOR.w,HALF_VECTOR,HALF_VECTOR
rsq HALF_VECTOR.w,HALF_VECTOR.w
mul HALF_VECTOR,HALF_VECTOR,HALF_VECTOR.w

; ------------------------------------------------------
; Project Half Vector onto Tangent Disk to get SPECULAR NORMAL
; ------------------------------------------------------
; NORMspec = H - (T.H)T where H is half vector and T is Normal
dp3 TMP.w, NEWNORM,HALF_VECTOR
mad TMP,-TMP.w,NEWNORM,HALF_VECTOR
dp3 TMP.w,TMP,TMP
rsq TMP.w,TMP.w
mul TMP,TMP,TMP.w

; ------------------------------------------------------
; DOT this vector with HALF VECTOR to get specular intensity
; ------------------------------------------------------
dp3 LIT.yz,TMP,HALF_VECTOR

; ------------------------------------------------------
; Calculate diffuse and specular factors (magical LIT ins)
; ------------------------------------------------------
lit TMP,LIT
```

```
; ------------------------------------------------------
; Scale the factors by the attenuation
; ------------------------------------------------------
mul TMP, TMP, TMP_ATTEN.w

; ------------------------------------------------------
; Add the DIFFUSE colour (* diffuse intensity)
; ------------------------------------------------------
mad DIFFUSE, LIGHTCOLOUR2,TMP.y,DIFFUSE
; FIXME: have separate DIFFUSE and SPECULAR colours

; ------------------------------------------------------
; Add the SPECULAR colour (* specular intensity)
; ------------------------------------------------------
mad SPECULAR, LIGHTCOLOUR2,TMP.z,SPECULAR


;*******************************************************
;******************** LIGHT 3 ***************************
;*******************************************************

; ------------------------------------------------------
; Build Vector from Vertex to Light and NORMALIZE it
; ------------------------------------------------------
add VERT_TO_LIGHT,LIGHT3POS,-NEWPOS
dp3 TMP.w,VERT_TO_LIGHT,VERT_TO_LIGHT
rsq VERT_TO_LIGHT.w,TMP.w

; ------------------------------------------------------
; Get the attenuation
```

```
; ------------------------------------------------------
dst TMP,TMP.w,VERT_TO_LIGHT.w
dp3 TMP.w,TMP,ATTENUATION
rcp TMP_ATTEN.w, TMP.w

; Normalize vertex to light vector
mul VERT_TO_LIGHT, VERT_TO_LIGHT,VERT_TO_LIGHT.w

; ------------------------------------------------------
; calculate and NORMALIZE the DIFFUSE NORMAL
; ------------------------------------------------------
; NORMdiff = L - (T.L)T  where T = Tangent, or Normal
dp3 TMP.w, NEWNORM, VERT_TO_LIGHT
mad TMP,-TMP.w,NEWNORM,VERT_TO_LIGHT
dp3 TMP.w,TMP,TMP
rsq TMP.w,TMP.w
mul TMP,TMP,TMP.w

; ------------------------------------------------------
; DOT this vector with LIGHT to get diffuse intensity
; ------------------------------------------------------
dp3 LIT.x,TMP,VERT_TO_LIGHT

; ------------------------------------------------------
; Calculate the Half Vector (Eye + Light)/2
; (don't need to divide)
; ------------------------------------------------------
add HALF_VECTOR, EYE_TO_VERT, VERT_TO_LIGHT
dp3 HALF_VECTOR.w,HALF_VECTOR,HALF_VECTOR
rsq HALF_VECTOR.w,HALF_VECTOR.w
```

```
mul HALF_VECTOR,HALF_VECTOR,HALF_VECTOR.w

; ---------------------------------------------------------
; Project Half Vector onto Tangent Disk to get SPECULAR NORMAL
; ---------------------------------------------------------
; NORMspec = H - (T.H)T where H is half vector and T is Normal
dp3 TMP.w, NEWNORM,HALF_VECTOR
mad TMP,-TMP.w,NEWNORM,HALF_VECTOR
dp3 TMP.w,TMP,TMP
rsq TMP.w,TMP.w
mul TMP,TMP,TMP.w

; ---------------------------------------------------------
; DOT this vector with HALF VECTOR to get specular intensity
; ---------------------------------------------------------
dp3 LIT.yz,TMP,HALF_VECTOR

; ---------------------------------------------------------
; Calculate diffuse and specular factors (magical LIT ins)
; ---------------------------------------------------------
lit TMP,LIT

; ---------------------------------------------------------
; Scale the factors by the attenuation
; ---------------------------------------------------------
mul TMP, TMP, TMP_ATTEN.w

; ---------------------------------------------------------
; Add the DIFFUSE colour (* diffuse intensity)
; ---------------------------------------------------------
```

```
mad DIFFUSE, LIGHTCOLOUR3,TMP.y,DIFFUSE
; FIXME: have separate DIFFUSE and SPECULAR colours

; ------------------------------------------------------
; Add the SPECULAR colour (* specular intensity)
; ------------------------------------------------------
mad SPECULAR, LIGHTCOLOUR3,TMP.z,SPECULAR



; ------------------------------------------------------
; Mult by Self-Shadowing Term
; ------------------------------------------------------
; SelfShadow = NORMDIST/MAXHEIGHT
;              * (1-SELF_SHADOW_MIN)+SELF_SHADOW_MIN
rcp TMP.x,SELF_SHADOW.x
mul TMP.x,VERTDATA.NORMDIST,TMP.x
mov TMP.y,SELF_SHADOW.y
add TMP.y,LIGHTCONST.y,-TMP.y
mad TMP.x,TMP.x,TMP.y,SELF_SHADOW.y
;cap at 1.0
add DIFFUSE,SPECULAR,DIFFUSE
;mov DIFFUSE,SPECULAR
min DIFFUSE,LIGHTCONST.yyyy,DIFFUSE
mul oD0,DIFFUSE,TMP.x%
```

Shader Source