12-2007

RENDERING PRINCIPAL DIRECTION CONTOUR LINES WITH ORIENTED TEXTURES

Kelly Gallagher
Clemson University, kdgalla@gmail.com

RENDERING PRINCIPAL DIRECTION CONTOUR LINES WITH ORIENTED TEXTURES

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Science

---

by
Kelly Gallagher
December 2007

Accepted by:
Timothy Davis, Committee Chair
Robert Geist
Stephen Hedetniemi

ABSTRACT

In this paper we explore the use of contour lines in computer graphics as a means of conveying shape to the end-user. Contour lines provide an alternative to traditional realistic rendering styles and may even provide a more appropriate visualization for certain situations. For our images, contour line orientation is established in accordance with principal curvature directions.

We present a method for rendering a texture, oriented in the principal curvature direction, across a traditionally-modeled geometric surface that effectively forms suggestive contour lines to enhance the visualization of that surface. We further extend the method to create animated contour textures, wherein lines move across a surface to suggest its shape. We demonstrate how the animation can be made more intuitive and easier to follow through a meaningful generalization of the generated vector space.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Generally the goal of computer graphics is to present information in a form that is visually interpreted by human end-users. In the case of three-dimensional graphics, there is a certain spatial element to this data that we wish to convey. Usually one is required to work within the confines of a two-dimensional display area, which presents a challenge of expressing data in a flat space without loss of depth and shape. Even though we realize that there are three dimensions in our world, we know that the retina of the eye merely interprets a two-dimensional projection of the light that enters the cornea. Therefore, by recreating this projection, it is possible to "fool" the eye into extrapolating three dimensions from two.

It is effective to create a realistic rendering of the data using accurate geometric representations and sophisticated light modeling techniques. If we present our data as it appears in real life, the viewer has a familiar frame of reference in which to interpret the image. Lighting cues are therefore one of the most evocative ways of indicating shape. There are instances, however, where lighting cues can be deceiving or inadequate in a realistic rendering or even in a photograph.

For centuries, illustrators have been developing non-photorealistic methods to more explicitly convey shape. The use of contour lines and selected stroke direction, for example, are often used to indicate that an object has a rounded surface. Even though real objects are seldom adorned with lines that run across the curvature of their shape, the average viewer has no difficulty interpreting what these lines mean.

In this paper we explore various procedures that have been employed to add similar suggestive contour lines to model surfaces in the area of computer graphics. We present a simple method for applying contour lines to a mesh as a texture. These lines are oriented on a surface according to local principal curvature directions for a meaningful and intuitive result that is easy to visually interpret. We further demonstrate that these lines can be animated to add even more definition to the underlying geometry.

We begin in Chapter 2 describing the background of the problem, along with the foundational mathematics. We then move on to a discussion of how to render contour lines through orientable textures in Chapter 3. Chapter 4 documents our design and implementation of a new method for animating these textures to provide a more accurate visualization of the surface features of a complex surface shape, and provide example images from our system. Finally, Chapter 5 suggests future direction for this work.

CHAPTER 2

BACKGROUND

While lighting cues are often sufficient to convey the shape of many three-dimensional models, alternative rendering methods are also exploited when additional information is needed. Aside from simple aesthetic preference, a realistic rendering is occasionally insufficient to inform the user of the precise shape of the subject.

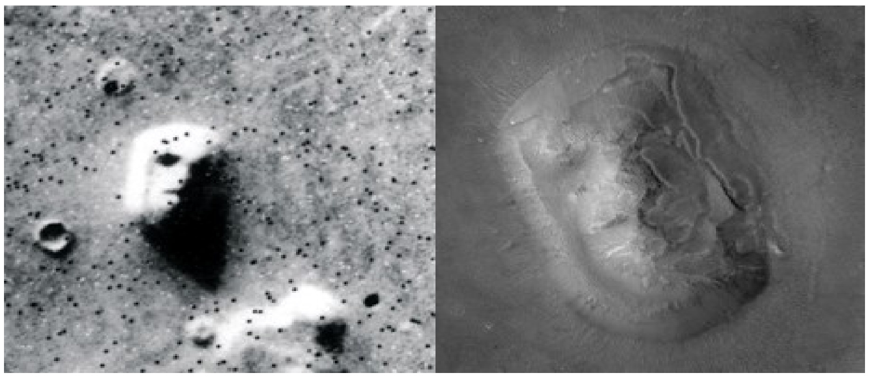

Figure 2.1 *Martian land form* [MSSS 07]

Figure 2.1 shows two photographs of the same Martian land form. While many believe the object in this famous photograph (left) strongly resembles a face, an alternate photograph (right), taken from a different angle and with different lighting conditions, reveals details which were not apparent in the original, including the fact that the object is completely asymmetrical [MSSS 07]. The original photograph demonstrates how lighting can obscure details and lead one to form a false impression.

Numerous types of visual cues can be used to convey information about shape. A multitude of research approaches using non-realistic rendering techniques exploit these visual cues; however, we focus specifically on the use of contour lines.

In this chapter we will examine some of the rendering techniques that have been proposed to convey shape using contour lines. We include both static methods and methods that employ animation to enhance the effect. We particularly focus on techniques that use principal curvature direction (the direction of greatest curvature) to orient contour lines because surface textures oriented along these directions have been shown to be particularly indicative of shape (i.e., observers have a tendency to assume that surface features on an object are aligned in these directions [Girs 00]).

2.1 Principal Curvature Direction

Contour lines must be drawn according to the shape and curvature of the subject. To determine the direction for which the curvature is greatest, we turn to differential geometry, wherein the direction of curvature can be conceptualized as a simple vector tangent to the surface at that point.

For any given point on a smooth surface in three-dimensional Euclidean space, there is a plane which lies tangent to the surface at that point. On this plane we can say that the point is crossed by infinitely-many tangent vectors, which also represent direction. Generally speaking, there is one single direction along which the surface is said to be experiencing the highest degree of curvature. Of course, when this is true, there must also be a direction of least curvature; further, these two vectors will always be orthogonal when they can be measured meaningfully and are called the first and second principal curvature directions (PCD), respectively. Along these direction vectors, the quantified curvature values are called the first and second principal curvatures. A given point on the surface, along with its normal, its two principal curvature direction vectors, and principal curvature values are collectively called the Darboux frame for that point [Pend 05].

It is possible to compute the Darboux frame for any point on a surface as long as that surface is differentiable, except for two intuitive cases: first, if the the surface is completely flat in the local area surrounding that point, and second, if the surface is locally spherical, in which case the surface will curve equally in every direction. In each case the surface is termed umbilical at that point.

To find the Darboux frame, we can use several similar methods based on high-order mathematics surrounding the tangent space of the surface at the point in question. In each case we begin with a typical triangle mesh representation that approximates the surface. The points which we wish to consider are the vertices of these triangles, so for each of the Darboux frames, we already have the point value. In addition, methods for finding the normals at these points are simple and well known.

2.1.1 The Weingarten Matrix

The shape operator, also called the Weingarten matrix, is a 2x2 matrix that concisely represents the tangent space of the surface as represented by the Weingarten equations. This matrix is constructed such that two eigenvalues can be found that are equal to the two principal curvatures *k1* and *k2*, where the greater of the two is the first principal curvature and the lesser is the second principal curvature. The corresponding eigenvectors naturally represent the curvature directions, respectively. This matrix can be defined in six terms.

$$W = \begin{bmatrix} \frac{eG - fF}{EG - F^2} & \frac{fE - eF}{EG - F^2} \\ \frac{fG - gF}{EG - F^2} & \frac{gE - fF}{EG - F^2} \end{bmatrix} \quad \textbf{(2,1)}$$

where

$$e = N_p \cdot \sigma_{uu}(p) \quad \textbf{(2.2)}$$

$$E = \sigma_u(p) \cdot \sigma_u(p) \quad \textbf{(2.3)}$$

$$f = N_p \cdot \sigma_{uv}(p) \quad \textbf{(2.4)}$$

$$F = \sigma_u(p) \cdot \sigma_v(p) \quad \textbf{(2.5)}$$

$$g = N_p \cdot \sigma_{vv}(p) \quad \textbf{(2.6)}$$

$$G = \sigma_v(p) \cdot \sigma_v(p) \quad \textbf{(2.7)}$$

given a point *p* where N_p is the normal of local surface patch *σ(u,v)*.

Goldfeather *et al* [Gold 04] presents several ways to approximate *W,* each of which involves a polynomial representation of *σ(u,v)* in terms of *x,y* and *z* such that the coefficients are terms of *W*. We know all of the points in our mesh representation; therefore the coefficients can be discovered by finding the least squares solution from the resulting set of equations. The most accurate, he discovered, is the cubic approximation:

$$W = \begin{bmatrix} A & B \\ B & C \end{bmatrix} \qquad \textbf{(2.8)}$$

$$z = \frac{A}{2}x^2 + Bxy + \frac{C}{2}y^2 + Dx^3 + Ex^2y + Fxy^2 + Gy^3 \qquad \textbf{(2.9)}$$

Notice the Weingarten matrix in this case is comprised of the first three coefficients of equation 2.9. The normal at this point is also known and can be written in the same terms:

$$N(x, y) = (Ax + By + 3Dx^2 + 2Exy + Fy^2, Bx + Cy + Ex^2 + 2Fxy + 3Gy^2, -1) \qquad \textbf{(2.10)}$$

This form presents us with three equations for each point on the local surface patch. Using the coefficients *A*, *B,* and *C*, we can construct the Weingarten matrix. Then we need only compute the eigenvalues and eigenvectors in order to complete the Darboux frame [Gold 04][Pend 05].

2.1.2 The Second Fundamental Form

The second fundamental form in differential geometry is a roughly equivalent way of describing the shape of a surface using a matrix, defined as the following:

$$L_{II}\,du^2 + 2\,M_{II}\,dudv + N_{II}\,dv^2 \qquad \textbf{(2.11)}$$

where

$$L_{II} = \sigma_{uu} \cdot N,\ M_{II} = \sigma_{uv} \cdot N,\ N_{II} = \sigma_{vv} \cdot N \qquad \textbf{(2.12)}$$

The second fundamental form is also expressed as a matrix:

$$II(u, v) = \begin{bmatrix} L_{II} & M_{II} \\ M_{II} & N_{II} \end{bmatrix} \qquad \textbf{(2.13)}$$

When considering the second fundamental form at a particular point on the surface approximation, it can be used to determine the curvature similarly to the Weingarten matrix.

Interrante [Inte 97] describes a method to compute the principal curvature directions and values from the second fundamental form. We begin with the set of three mutually orthogonal vectors *[e_1, e_2, e_3]* at a point *P* on the surface. e_1 and e_2 are arbitrary vectors on the tangent plane of *P*. The remaining e_3 vector points in the direction of the normal at *P*. The values of the matrix are computed using these vectors.

The relationship between the second fundamental form and Weingarten matrix is given as

$$II(v, w) = W(v) \cdot w \qquad (2.14)$$

where

v is a point on the surface and *w* is a point in the tangent space at *v,*

II(v,w) is the second fundamental form of the surface over *v,* and .

W(v) is the Weingarten matrix at point *v* [Weis 07].

The eigenvalues of the second fundamental form are, likewise, the first and second principal curvatures [Inte 97].

2.1.3 Gaussian and Mean Curvature

In addition, Huang *et al* [Huan 05] shows how to compute *k1* and *k2* using the mean curvature *H* and gaussian curvature *K*. As it turns out, the mean and gaussian curvature are as follows:

$$K=\frac{L_{II}N_{II}-M_{II}^2}{E_I G_I-F_I^2} \quad \textbf{(2.15)}$$

$$H=\frac{L_{II}G_I-2M_{II}F_I+N_{II}E_I}{2(E_I G_I-F_I^2)} \quad \textbf{(2.16)}$$

L_{II}, M_{II}, and N_{II} are terms of the second fundamental form, defined in equation 2.12. The following three values are terms of the first fundamental form [Pres 07]:

$$E_I=\|\sigma_u\|^2, F_I=\sigma_u\sigma_v, G_I=\|\sigma_v\|^2 \quad \textbf{(2.17)}$$

Once we compute these values, we can solve equations 2.14 and 2.15, and finally for the first and second fundamental curvature values:

$$k_1 = H + \sqrt{H^2 - K}, \qquad \textbf{(2.18)}$$

$$k_2 = H - \sqrt{H^2 - K} \qquad \textbf{(2.19)}$$

For this application it was not necessary to find the corresponding directions.

Here we are also presented with an alternative method of computing the curvatures using a local gradient direction vector and the Hessian matrix over the point on the surface; however, this method is inaccurate when ||*g*|| (a gradient vector of the surface) is 0 or very close to 0 [Huan 05].

2.2 Visualizing Principal Curvature Direction

To simply illustrate the PCD vectors, one can render short strokes across the surface. [Girs 00] demonstrates that graphing a vector field of curvature directions as short lines, while producing results that are rather crude in appearance, can effectively convey the shape of the underlying surface.

Nicer results can be acquired using a technique described by [InFu 97] which can be used to model the strokes such that they conform to the surface. Given a surface representation where we have the Darboux frame at key points, we construct a box of a certain length, width and height, which are user-defined according to the situation at hand. This box is centered around a point *P*, where it is aligned with the orthonormal set of the surface normal at *P* as well as the first and second curvature directions, *k1* and *k2*, such that the height is measured along the direction of the normal, and the length and width along *k1* and *k2*, respectively. The geometry that lies on the intersection of this box and the surface can be used to represent a stroke over the surface. The length and width of the box determine the length and width of the stroke. While a longer stroke is more desirable, the accuracy of the stroke, as a representation of principal curvature direction, degrades as the stroke extends from *P* in either direction, since methods used to determine the principal curvature are very localized. Therefore, it is necessary to choose a length that falls short of perceptible error. Likewise, narrowing the stroke will emphasize its length, but there is a limit on how small the width can become before rendering artifacts occur.

A publication by Interrante *et al* [InFu 97] demonstrates that this is a particularly effective way to visually represent the surface when the surface must be transparent. This transparency makes it difficult to judge the shape of the object based on lighting cues; however, the contour strokes effectively define the surface. At the same time, the sparseness of the strokes makes it easy to see the geometry that lies underneath (see Figure 2.2).

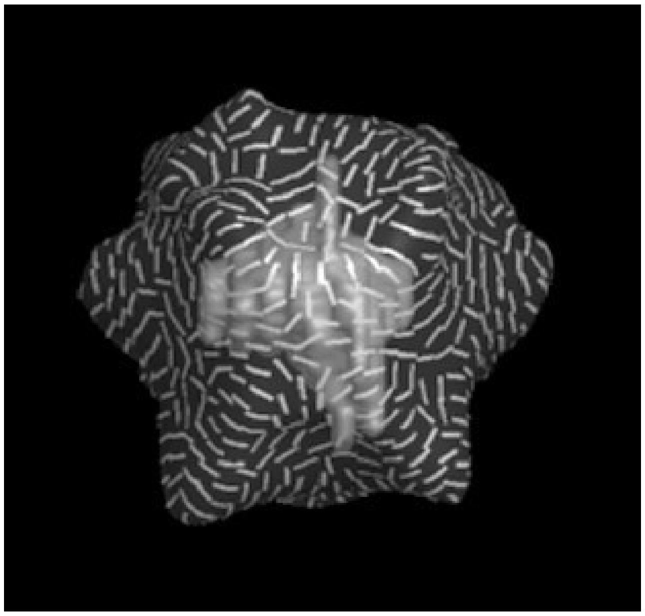

Figure 2.2 *Principle Curvature Direction Strokes* [Infu 97]

A separate field of computer graphics, for which principal curvature data can be applied, is the synthesis of images with the appearance of a pen-and-ink drawing. Here we use the principal curvature to determine the direction of synthesized pen strokes.

When an artist renders in pen, lines are used to outline the shape of the subject, as well as to darken regions in order to give the appearance of shadow from imaginary light sources (often called hatching). Illustrators observe that using arbitrarily oriented parallel lines for hatching can destroy the illusion of shape for an outlined object. On the other hand, if the lines curve with the shape of the object, much like the contour lines that are the subject of this chapter, the illusion of shape can be strengthened. Though other possibilities exist, Hertzmann and Zorin argue that the PCD vector set is a good criterion for determining the orientation of hatching strokes. Not only is it visually effective, but the results are also consistent with real illustrations [HeZo 00]. Even when individual strokes are rendered procedurally, realistic results can be achieved using a tensor field to determine stroke direction [Sals 97].

2.3 Advection

Advection is a mathematical model of distributed movement, often of fluid. Say we have a system in which fluid flows in a manner that is complex and the direction is affected by numerous factors. We can sample the direction of flow at various points in our system and record the direction of the flow as directional vectors. Eventually we collect a representation of the entire system as a vector field. There are multiple methods to create an intuitive visualization of advection, given the vector field as input.

Recall, however, that the topic of this paper concerns visually conveying the shape of surface geometry using contour lines. For our purposes, if we substitute the flow-direction vector field with the set of PCD vectors of a surface, we can create a visualization that will convey the shape of the surface. While these vectors do not represent motion, the application of advection methods to curvature vectors can create visual contour lines that make the overall shape of the surface more apparent to the viewer.

2.3.1 Spot Noise Advection

Spot noise for flow visualization is a simple technique to generate a texture illustrative of the vector field. While the results are not as visually impressive as more advanced procedures, it is useful because it is less computationally demanding. In spot noise advection, dark spots are placed on a texture in a random, but roughly even, distribution and then elongated and oriented according to the underlying vectors. The spots appear as ellipses that are aligned with the direction of the vectors in the vector field. From a distant vantage point, the elongated ellipses blur together and appear as curving lines [deLe 95].

2.3.2 Line Integral Convolution (LIC)

An early method to view vector information was named after the well known digital differential analysis line drawing algorithm. It is called DDA convolution. Using the vector field as a basis for convolving a two-dimensional pixel image, this method creates a reasonably intuitive visualization for a vector field from that image. The procedure is straight forward: for each pixel in the input image, select a line of pixels in the direction of the corresponding vector using the DDA algorithm (this line serves as the convolution mask). Replace the color of that pixel with the average color of the pixels that fall within this line. The mapping between the pixel and the underlying vector is up to the implementation, but the mask can follow a twisting path of several vectors.

Due to the nature of the filter, there will be much greater color correlation in directions that are parallel to the vector directions. When a simple random-dot noise pattern is chosen as the input image, the result is that the image will show streaks in the directions indicated by the vectors. While this illustrates the vector flow, the appearance of such an image is often blocky and disjointed, due to the fact that the algorithm is based on straight lines when for many systems being modeled, the vectors are only approximate measures for what are meant to be continuous curves.

Line integral convolution is a method to model these curves. Simply put, we know that our vectors represent the direction of flow at particular points on our curve; therefore, they must be tangent to the curve, or at least parallel to tangents. If we take the common integral of several direction vectors the result will be a curve, or streamline, that passes through these directions. If we adjust the DDA convolution to use these computed streamlines as the convolution mask at each point, the result will be a smoother pattern.

As described in [CaLe 93], this can be done in a piecewise method where we divide the vector field into segments, called cells, where each represents the influence of a single vector in the field. The part of our filter kernel through this cell is described as

$$h_i = \int_{s_i}^{s_i + \Delta s_i} k(w)\,dw \qquad \textbf{(2.20)}$$

that is, the integral of *k(w)*, the kernel for this cell, over the estimated parametric distance between cells. For each pixel, we construct a filter from the summation of these integrals over the path with a length of *L* pixels. The value *L* is variable and is set as appropriate to the application. It is also suggested that a similar path be traced backwards from the pixel in directions reverse to the underlying vectors to help maintain a smoother stream line [CaLe 93]. The results are shown in Figure 2.3.

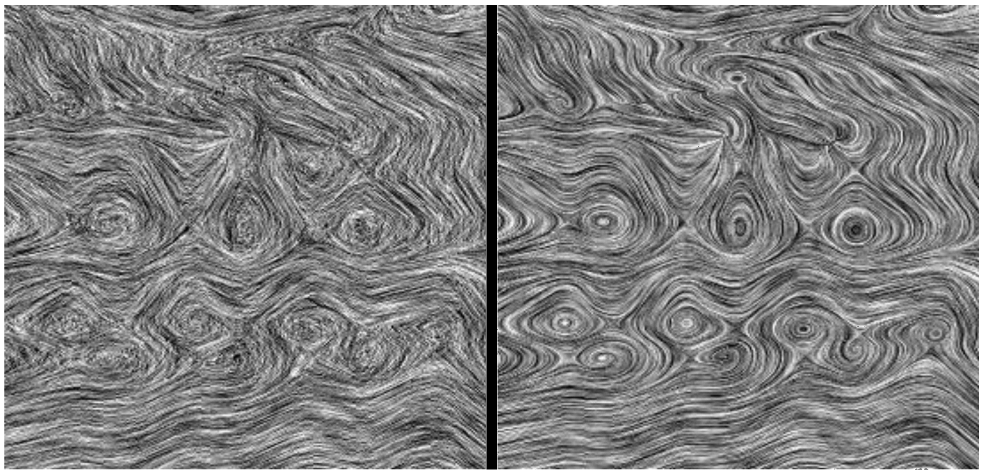

Figure 2.3 *DDA Convolution (left) and LIC (right)* [CaLe 93]

In addition to representing vector flows, the streamlines applied to a three-dimensional surface may resemble contour lines that an artist might draw to define the shape of an object. In effect, the LIC rendering becomes a procedural texture that can enhance the shape of an object. The usual method for applying texture to a mesh, that is, the direct mapping of an image to a mesh via pairs of texture coordinates, can be employed for applying this texture as well. However, this approach works well only if the texture is evenly mapped over the object. Further, popular procedures for automatic texture coordinate generation typically create mirroring or distortion.

Another application, presented by [Inte 97], involves an arbitrary transparent, blobby object that is only visible by the sparse pattern of lines on its surface. To render this pattern, the author takes advantage of the high resolution of triangle vertices that form the object's surface by simply coloring each vertex according to the computed LIC. The object is still transparent due to the fact that the pattern's opacity is keyed to luminance. A voxel-based, three-dimensional implementation of LIC is used to compute the intended color intensity at each voxel, which is then mapped to the corresponding surface vertex.

In addition, LIC can be used in artistic non-photorealistic rendering. Using the same type of direction vector field generated from PCD vectors, LIC can be used to convolve a Perlin noise pattern to generate a texture that appears similar to brush strokes. This technique is useful for rendering oil-painting and watercolor style images. It can synthesize how an artist might choose the direction of his brush strokes to emphasize shape, and how a brush passing through paint can leave streaks of color in the direction of a stroke [LumM 01].

2.3.3 Fast Line Integral Convolution (FLIC)

Line integral convolution is an inherently expensive operation as multiple integrals must be approximated for each pixel in the target image; therefore, researchers have investigated optimizing the process to reduce calculation time.

A prominent area of redundancy in the algorithm is the overlap of generated streamline filter kernels along the same integral. Consider that the filters for LIC are determined by computing an integral curve over the vector space using numerical methods. Several pixels lay under the integral curve, yet the traditional algorithm computes a new filter kernel for each pixel that falls within this domain. These kernels will overlap and include many identical pixels. To optimize the algorithm for speed, it is possible to store the pixel locations computed for one pixel's kernel and reuse them for other kernels that would fall in the same streamline.

Secondly, even streamlines that do not overlap can be under the influence of the same vectors in the underlying vector field. When this happens the filter kernel will be identical except for a correction term that is part of the formula. Once again, a set of precomputed pixel locations can be stored for the sake of eliminating redundant computation. A procedure that computes LIC using these two optimizations, while yielding identical results, is Fast LIC, and is generally used in place of traditional LIC [StHe 95].

2.3.4 Animating Line Integral Convolution

There are two reasons one might wish to animate an LIC rendering. Obviously if the vector field over which the LIC is rendered changes over time, the visualization should change accordingly. If the vector field is static, animation can still be used to indicate the direction of the flow and illustrate the flow as being in motion. In the case where LIC strokes are used to convey shape, animation could possibly provide a clearer picture. Various methods can be used to achieve this goal.

2.3.4.1 DLIC

Dynamic Line Integral Convolution (DLIC) is an extension of LIC for vector fields that are dynamically changing. DLIC is particularly useful for visualizing electric current, where the current flows in more than one direction. The result is an animation that shows how the flow changes over time.

For this procedure, a second vector field is used which represents the direction of movement for each vector in the initial vector field. Initially a texture pattern of random noise was continually modified according to this second vector field, and traditional LIC was applied on top of this field. While this method works in theory, the random and even distribution of the underlying texture is essential for visually pleasing results. Modifying the texture according to the second vector field eventually caused patterns to appear in the image map, resulting in an overall softening in contrast of the LIC result. The solution is to treat the initial pixel map as a particle system, wherein the method carefully tracks each particle and recreates the texture image at every frame, factoring the overall distribution of color [Sund 03].

2.3.4.2 OLIC

Oriented line integral convolution (OLIC) [WeGP 97] is also a variation of LIC. Rather than averaging the colors in a streamline in order to find the resulting color, the kernel itself is a color ramp. The resulting fade from black to white is a visual indicator of the flow direction. The viewer would typically have a strong predilection to interpret the strokes like a comet, i.e., the light end is leading, while the dark is trailing off. This result represents an advantage over traditional LIC, in which there is no indication of the direction in which the streamlines flow.

This arrangement of color lacks clarity in LIC, where color separates one streamline from another. For the purpose of OLIC, however, a much sparser underlying dot pattern for the source image ensures that each stroke remains distinct against a dark background. Further, these strokes can be animated by continuously re-rendering the image while shifting the phase of the ramp kernel. [BeGr 00] notes that this result can also be achieved simply and efficiently using color-table rotation. A resulting pattern is shown in Figure 2.4.

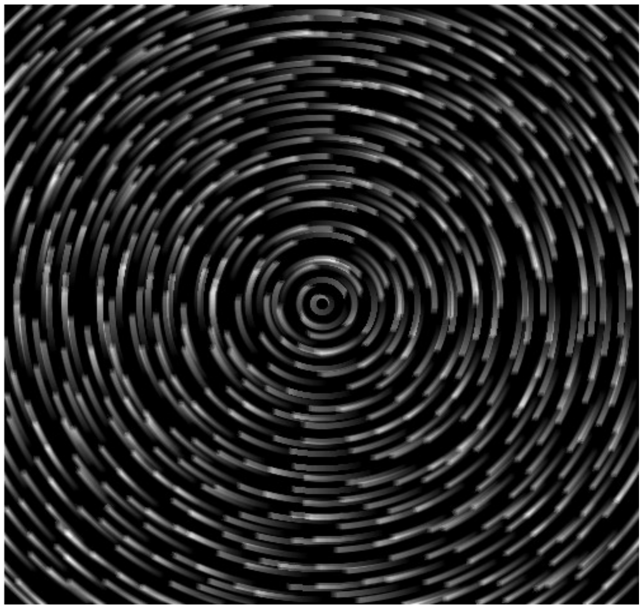

Figure 2.4 *Vector Field Rendered with OLIC Rotating Clockwise* [WeGP 97]

In this chapter we have looked at the research background of computing principle curvature directions for mesh surfaces as a means to create illustrative renders. In the next chapter, we implement the cubic-order algorithm described in Section 2.1.1 and use the resulting PCD vector data in a new texturing method. In Chapter 4 we further demonstrate that the concept of animation presented in Section 2.3.4.2 works well with our goal of animating the surface texture that we create.

CHAPTER 3

RENDERING CONTOUR LINES WITH ORIENTED TEXTURES

Many techniques exist to determine primary curvature direction and also to display the results. Most of the methods presented in the previous chapter rely on volumetric methods to ultimately render the results to the screen. Here we present a simple method to convey the visualization on the surface of input geometry using a repeating texture that is locally applied and oriented.

The procedure divides an input triangle mesh into a collection of small patches of triangles, one for each vertex, which represents the surface local to that vertex. Each patch overlaps and shares triangles with other patches around it. For each patch we determine a vector, which represents the direction in which the patch curvature is the highest and also stipulates the orientation of a texture image (a strongly directional pattern that will appear as contour lines when rendered), which is applied locally to that patch. The textures on the patches will blend together and create a pattern that will globally suggest the shape of the mesh. Though the results are not guaranteed to be as coherent as explicitly determined contour lines, the overall impression is the same.

3.1 Preliminary Information

Before we detail this procedure, we begin with some preliminary information about the specific implementation which led to the results presented in this paper. The procedure is written in the language C#, except where noted. In addition to Microsoft's standard .NET library, we use two open-source libraries that provide useful functionality.

To aid in visualization we use the Irrlicht.NET library, which provides a high-level scene graph for either OpenGL or DirectX rendering. Open GL is chosen, of course, for its traditional role in academics. Irrlicht.NET is used to quickly establish a visual framework in which to display the results of our implementation.

Mapack is a simple mathematics library that provides basic Matrix operations, as well as a least-squares solver and a matrix diagonalization function for finding eigenvalues. All of these features will be used in the process.

3.1.1 The Model

Models for this exercise exist internally in the form of the typical type of triangle mesh used in graphics applications. They are taken from files in Wavefront's OBJ format. This format is easy to parse and suggests a convenient data representation, particularly in referencing triangle corners as indices to vertices, rather than using the vertices themselves. This scheme facilitates determining which triangles share a particular vertex without performing potentially error-prone floating-point comparisons. Additionally, there is no redundant storage of vertex information: if a vertex is updated, it applies to all triangles that share that vertex.

3.1.2 Data Structures

The model is a collection of surfaces, and a surface is represented as a list of vertices and a list of triangles. Each triangle simply stores the indices of the three corner vertices, along with the midpoint and the normal of that triangle. Each vertex stores the Darboux frame for that point, as defined in the previous chapter, which includes the point, the normal at that point, the two principal curvature direction vectors, and the two principal curvature values.

The normals are computed immediately upon loading the model. For each face we create two vectors parallel to the face by selecting two corner vertexes and subtracting the remaining corner vertex from each. Taking the cross product of these two vectors results in an orthogonal vector, which when normalized, represents the normal for that face. The normal for each vertex in the model is approximated by averaging the normals of the faces that share that vertex.

3.1.2.1 A Collection of Surface Patches

For each vertex, there is also a separate local patch which is necessary for computing the principal curvatures. Each patch is a duplicate of the vertex along with its surrounding neighbors. Here a neighbor is defined as any vertex that can be reached from the center vertex by crossing exactly one triangle edge. This set of neighbors is also called the first *n*-ring, or 1-ring, because the remaining triangle edges form a ring around the center vertex.

Once we isolate the geometry that we need to include in a patch, it is duplicated so that during the calculations, each patch can be translated to its own local coordinate space, while the underlying model is preserved. As we shall see, the patch is also useful later for computing texture coordinates when the texturing solution is applied.

3.2 Principal Curvature Directions

Our goal is to compute the Darboux frame for each point on each component. We start by loading each component and computing the normal for each vertex. We then iterate over each vertex and apply the cubic-order algorithm discussed in [Gold 04] and further substantiated by [Pend 05].

For each vertex we obtain a surface patch as described in Section 3.1.2.1. Generally there will be at least three vertices in the neighbor-set, which will be necessary for the calculations. However, if the set has fewer than three, we can obtain a wider sample by combining the neighbor-sets of the included vertices and constructing a larger patch. For each patch we apply calculations to approximate the shape operator for the patch. The eigenvectors and eigenvalues of the shape operator are the first and second PCD vectors and their corresponding values, respectively.

3.2.1 Local Coordinate Space

To simplify the computation, we translate the local patch to a local coordinate space so that the normal of the surface at that point, and the two essentially arbitrary vectors that we derive from the normal, form an orthonormal basis. For now we know that the center point will be at the origin, and the normal of the patch at that point is axis aligned at (0,0,1). We are given the following equations to construct a rotation matrix:

$$R=[r_{1,} r_{2,} r_3]^T \quad \textbf{(3.1)}$$

$$r_1=\frac{(I-n.n^T)i}{\|(I-n.n^T)i\|} \quad \textbf{(3.2)}$$

$$r_2=r_3 \times r_1 \quad \textbf{(3.3)}$$

$$r_3=n \quad \textbf{(3.4)}$$

where

I is the 3x3 identity matrix

n is the normal at the center point *p*

i is the *x*-axis represented as the vector $[1,0,0]^T$.

A degenerate case appears when the normal is aligned with the *x* axis. In this case we must alter the procedure to use a different axis [Pend 05].

3.2.2 The Cubic-Order Parametric Representation

The cubic-order parameterization of the shape operator was detailed previously in section 2.2.1. Recall that we have an equation expressed as a cubic order polynomial with seven terms. In addition, expressed in similar terms, is an equation for both the x and y components of the normal at that point. Also recall that we must ensure that at least three vertices are included in the extracted neighbor-set for each patch. Now we can find the coefficients using all of the values of x and y that we have in the patch vertices. With each set of values substituted in each of these three equations, we have enough equations to solve for the coefficients. To solve the series of equations, we use the Mapack Library's Solve() method, which determines the least squares solution.

Note that center point p, for which we are determining the shape operator, should not be included in the patch. Because its x, y and z value are all zero, this point is not useful for solving the equations, and will cause the Solve() method to fail.

3.2.3 Retrieving the Principal Curvature Direction Vectors and Values

The shape operator, or Weingarten Matrix, is also defined in the previous chapter Section (2.2.1) where it is composed of only the first three of the computed coefficients. Diagonalizing this matrix using Mapack provides us with the necessary values. The resulting 2x2 diagonal matrix will contain the first and second principal curvature values. One need only determine which is greater to distinguish the primary from the secondary principal curvature value. This procedure is also used in determining which column in the other resulting 2x2 matrix is the first PCD vector, and which is the second.

3.2.4 Dealing With Error

Several issues arise when computing the principal direction curvature. Some of these are identified in the literature along with ways of dealing with them. One issue already mentioned is the visible and abrupt reversal of first and second PCD vectors, which results when vectors are undefined on a surface that is locally spherical or flat. Our algorithm is imprecise; therefore, when a local surface patch meets, or is extremely close to meeting, either of these criteria, the result is left to the fate of rounding error. If the surface in question has several vertices that fit this category, the results can be inconsistent. One can resolve these situations by simply substituting the result with a consistent reference vector. Such problematic vertices can be identified by comparing the primary and secondary curvature values, which will be very close.

3.3 Applying Texture

Once we have computed the Darboux frame for each vertex in the target geometry, that information can be used to create a pattern on the surface of the model that will cue the viewer to its overall shape. To apply contour lines or marks to the surface of the model, we use texture mapping since it is supported by most graphics hardware for rendering in real-time and by graphics libraries through a straightforward process.

The challenge, however, is establishing the mapping between the model vertices and image-map points. Currently, procedural methods exist to apply textures to simple objects such as spheres and cylinders. Although more generally applicable procedures covering an arbitrary surface without visible distortion can be difficult.

3.3.1 General Texture Parameterization

Surface parameterization is a general term that, in the case of procedural texturing, refers to a surface being "unfolded" and projected onto a plane. Because it is much easier to map points of a flat image to points on a plane, an accurate parameterization can be very useful in texture mapping.

One type of parameterization is the conformal map, which is based on methods of conformal geometry that allow any arbitrary surface, without holes or self intersection, to be mapped first to a sphere, and then to a flat surface. The mapping is called a "conformal mapping" and when such a mapping exists between two surfaces, they share a "conformal equivalence." [Hake 00] presents a method in which the conformal mapping is approximated from a number of discrete elements, corresponding to vertices on the surface. Since the model is represented as a triangle mesh, rather than a continuous surface, the approximation is especially appropriate. Theoretically the conformance map is represented as a partial differential equation, but here it is approximated as a matrix of linear equations that can be easily solved using a least-squares solution [Hake 00].

To apply this method cleanly, the mesh must have spherical topology for direct spherical mapping. The mesh will therefore need to be divided into segments. Further, even a topologically spherical segment will need to be divided further to achieve the ultimate planar mapping. When a model is sufficiently divided, the resulting segments will be homeomorphic to disks. The planar mapping of a model segment for texture-mapping is referred to as a “chart.” The collection of charts that comprise the complete geometry is called the “atlas.”

Along the lines which segment the mesh, discontinuity will appear in the texture mapping. The model, therefore, should be divided in areas where a natural line already exists. Cusps in the model can be detected as areas where adjacent triangles are facing each other at a sharp angle. A predetermined threshold can be used to determine how concave a cusp will need to be for it to be selected [Levy 02].

There are generally two categories of inaccuracy that affect an approximate parameterization: angular distortion and area distortion. The conformity map described above preserves global orientation such that angular distortion is not problematic. Area distortion, on the other hand is occasionally observed. This artifact is caused by a disproportionate ratio between the areas of a triangle and the region of the texture image that is mapped that triangle. The result is a stretching or compressing of the texture image over part of the model.

To deal with this phenomenon, evenness of the area distortion can be controlled by finding the optimal solution to an objective function that minimizes the deviation. This remedy is not always appropriate, however, because maximizing the area conformity without compromising the angular conformity is not always possible [Dege 03].

Rather than attempting to create an accurate parameterization of the entire surface, one can also break down the surface into smaller pieces and apply the texture mapping locally to each component. This technique, called “lapped texturing,” can be used in cases where a repetitive pattern is applied to the model. Here, the overall texture map is constructed of a collection of overlapping patches, each displaying an iteration of the same repeating pattern, as in Figure 3.1.

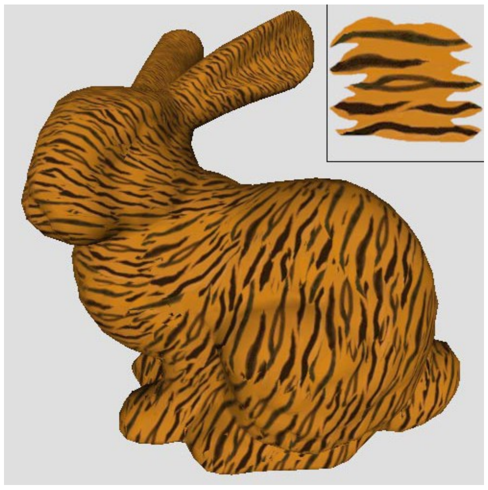

Figure 3.1 *Lapped Texturing* [Prau 00]

The applicable texture element is part of a repeating pattern and is surrounded with an alpha-channel to fade out the edges. When the texture is mapped to adjoining surface patches, it results in smooth overlapping areas without visible seams.

According to the algorithm put forth by [Prau 00], we begin at a random triangle on the surface and create a corresponding mapping to a random region of the texture image according to a predetermined spatial ratio. From that triangle we “grow” the patch by spreading out to incorporate other triangles into the mapping until we reach the extents of the single texture element, or we reach a point where adding new faces will introduce an excessive amount of distortion. Once we reach this point, we have completed a patch and we proceed to select a new random triangle that has not been covered yet. Iteration continues until all triangles are covered.

As before, the resulting patch segment must be homeomorphic to a disk. When adding the next triangle to the patch, the algorithm considers how the triangle will affect the topology of the resulting patch, and determines whether the triangle will be added to the patch.

Another consideration is the orientation of the texture. Unless the texture is isotropic, there will be a correct direction for the texture orientation. In the application described, a user-specified tangent vector is provided for key vertices and interpolated to all vertices. For any given vertex, this vector indicates the local "up" vector for the orientation of the texture. Since a patch has multiple vertices and vectors as such, the texture can change direction over the course of the patch. The algorithm allows for the intentional distortion of the textured image to improve the appearance of the texture orientation [Prau 00].

3.3.2 Simple Method for Rendering Contour Lines as Texture

We can think of contour lines as a repeating pattern across the surface of a model that are locally oriented according to primary curvature direction. The algorithm described above can therefore be adapted to the task of rendering contour lines. For a texture image, we use a pattern of parallel lines. These lines will be oriented, according to the algorithm, by the orientation vectors provided at each vertex. In this case we use the primary PCD vectors that we compute in section 3.2. Ideally, the lapped texturing algorithm should yield impressive results; however, for the sake of expediency, we use a simplification of this method.

Recall that in the course of computing the Darboux frames, we represented the triangle mesh as a set of patches consisting of a center point surrounded by its 1-ring. Because a patch is generated for every vertex on the surface of the mesh, these patches clearly overlap. We propose that this patch can be used as part of the lapped texture algorithm in place of the dynamically “grown” patches that cover a mesh, which greatly simplifies the algorithm in several ways. First, we no longer need patch growing criteria to be considered for each triangle on the mesh. Second, given the size and nature of the patch is generally guaranteed to be of the correct topology, the “unwrapping” of the patch is now trivial as well and is often unnecessary. Any patch with a reasonably gradual curve will exhibit very little visible distortion when compared to a flattened version of the same patch.

One drawback to using this simplified algorithm is due to the much smaller patch size that generally results. Most obvious will be that for a mesh with a high triangle resolution, the repeating pattern will appear to be very small in the context of the full model. Also, a smaller patch introduces more border area, and the border around the patch is an area where aliasing and seams are visible. We also have less room to shape the texture according to local orientation. Due to the high degree of blending we employ for our method, however, these effects are not highly visible and so do not result in a significant loss of image quality.

3.3.2.1 The Algorithm Explained

To map the texture onto the patch, we must find texture coordinates for the vertex on the patch. We first orient the patch to a local coordinate space according to the orientation vector (or “up” vector) for the vertex in the center (here we let the orientation vector be the principal curvature direction). A useful orientation would lay the patch flat on the *xy* plane so that *x* and *y* coordinates would roughly correspond to *uv* texture coordinates. This result could be achieved by translating the patch to the same local coordinate space that was used in Section 3.2.1. Essentially, what we will have is an orientation basis for the patch's center vertex, consisting of the mutually orthogonal normal, and primary and secondary PCD vectors, translated to the canonical basis.

In the algorithm to compute the Darboux frame for the center point, the principal curvature direction is found originally as a two-dimensional vector, to which a translation is applied to map it to three-dimensional space. We use this two-dimensional vector now to finish computing an orientation for the surface patch. Once we have translated the patch so that the normal is aligned with the *z* axis, the patch can simply be rotated again along the *z* axis.

To find the angle between the PCD vector and the *y* axis, we use the following:

$$\cos\theta = \frac{e1 \cdot Y}{\|e1\|\|Y\|} \quad \textbf{(3.5)}$$

$$\sin\theta = \frac{\|e1 \times Y\|}{\|e1\|\|Y\|} \quad \textbf{(3.6)}$$

In this case *e1* is our principal curvature direction vector and *Y* is the global *Y* axis (0,1,0). Note that we can simplify these formulas because both vectors *e1* and *Y* should be normalized. Once we have determined the angle, a simple rotation matrix can be constructed.

Once the patch is in this orientation, texture coordinates can be applied. We take the simplistic approach of projecting the texture flat onto the patch. Iterating through the 1-ring points on the patch, we discover the minimum and maximum *x* and *y* values that the patch covers and simply map them to established *uv* constraints of 0 and 1.0. We can then find the texture coordinates for any given point *P* using the the following formula:

$$(u,v)=\left(\frac{x_P}{(x_{max}-x_{min})},\frac{y_P}{(y_{max}-y_{min})}\right) \qquad \textbf{(3.7)}$$

where

u and v are the (u,v) texture coordinates of P

x_p and y_p are the x and y component of P

x_{max}, x_{min}, y_{max}, and y_{min} represent the extrema of x and y values for all points on the patch.

The edges of each patch also need to be made transparent so that the textured patches blend together to form a continuous textured surface. We can accomplish this task by adding transparent alpha-channel data to the texture image. Several image format types support multiple levels of transparency in the form of an alpha-channel.

Additionally, the alpha channel can also be stored in a separate image map file. We create the alpha channel data so that the center of the texture is opaque but the edges fade from opaque to completely transparent.

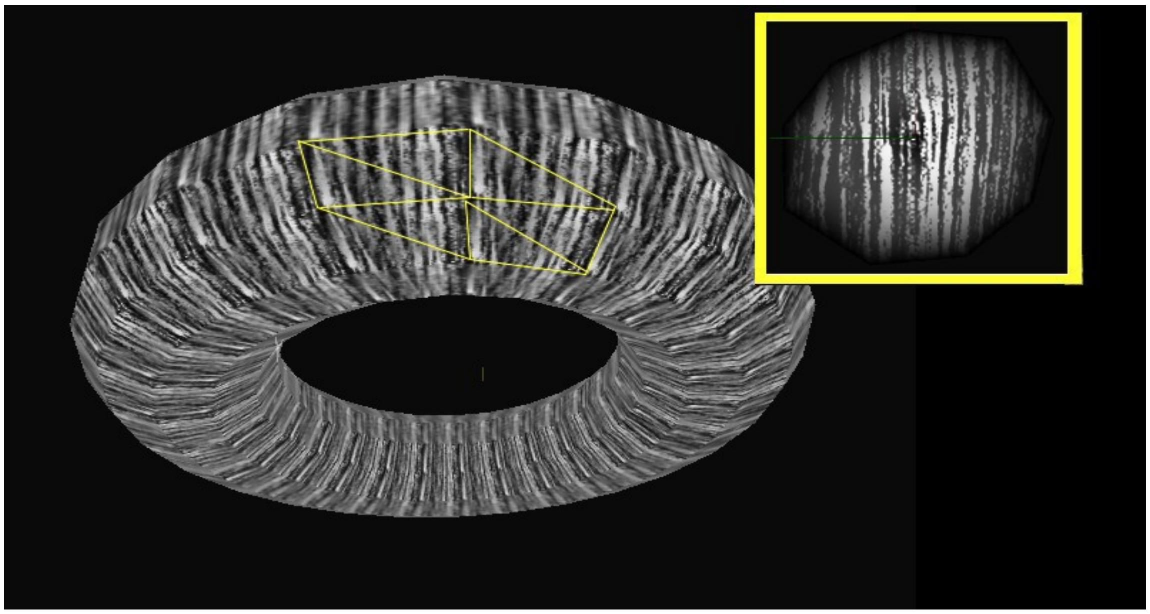

Figure 3.2 *Torus Model with Highlighted Patch and Input Texture*

A similar effect can be achieved using vertex alpha values as well. Generally, the graphics renderer will also support setting color values, including an alpha value, for each vertex in the mesh. During the render, the vertex color will be modulated with texture color to create the color that will appear on the screen. Setting the center vertex of the patch to opaque, and the edges to completely transparent, will also produce the desired result. Even though this leaves much of the patch transparent, the amount of overlap will essentially cross-fade the patches together, resulting in a full level of opacity at every point on the continuous surface. We implement both methods to achieve different rendering effects.

Figure 3.2 shows the results of applying a highly oriented texture to a simple torus model using this technique. An example patch is highlighted. The texture is centered at the center point of the patch and the edges are faded to transparency. Due to overlap of textured patches, no transparent portion of any patch is visible. The texture appears to be solid and continuous.

3.3.2.2 Area Distortion

Not evident in Figure 3.2 is the possibility of area distortion when using this method. We examine an example for which significant area distortion is present due to the translation of the entire texture to each patch. In the case of the torus, all of the triangles that comprise the model are of comparable size. In practice however, patches in the model can have a wide variance in size, according to the size of the triangles they encompass. With a triangle mesh, areas that require more detail often contain smaller triangles than other parts of the mesh.

A render of the same procedure applied to a different model, a vase, shows a definite shrinkage of the texture at the neck of the vase, as compared to the body. The surface of the neck across the circumference happens to be segmented the same as the body; therefore, triangles at the neck are smaller then those at the body. The proposed texturing method currently fits the texture to the extents of the patch to which it is being applied, which results in scaling the texture to the patch size.

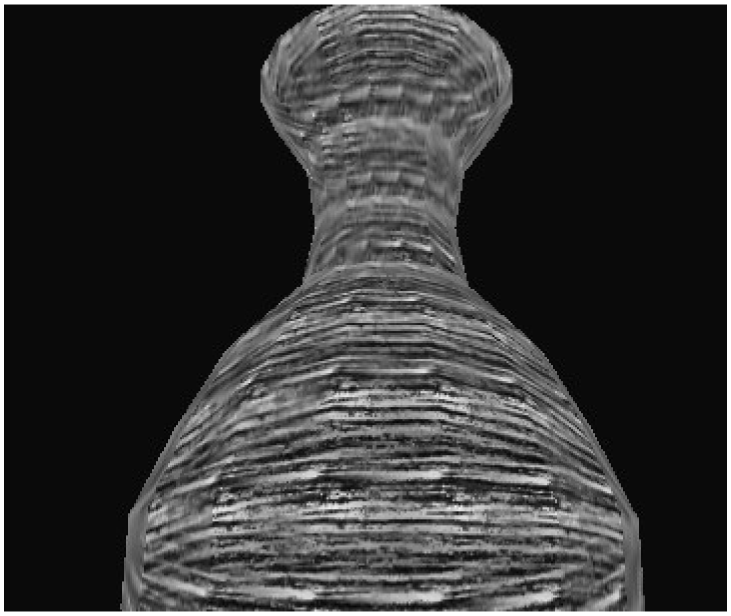

Figure 3.3 *Model Vase Showing Area Distortion*

Alternately, texture coordinates can be scaled adaptively so that the patch uses a portion of the texture that is proportionate to its area. Previously we computed the *u,v* coordinates for each patch in terms of the maximum and minimum *x* and *y* values for the patch vertex locations. Instead we can survey all of the patches in the mesh and find the global extrema for use in the formula. Using the global *x* and *y* maximum and minimum, as opposed to the local, will scale all of the textures to the same frame of reference, but for smaller patches, parts of the texture edge will be cut off. This may not matter, however, if the texture is a regular pattern.

Another evident problem is a potentially distracting periodic pattern due to numerous repetitions of the texture. While distracting patterns from low-frequency components of a texture are also a limitation of the lapped texture on which we have based our algorithm, the fact that we require more repetitions to cover the model exacerbates the problem. While carefully selecting a texture with an evenly distributed appearance can mitigate the problem, it is inherent in any application of a repeating texture. We use a texture comprised of straight stripes of the same width, which is even enough so that such patterns are not distracting enough to deter from the perception of orientation. Even when the stripes on each patch do not line up, the underlying directionality of the texture is still evident.

Now that we have a system of applying oriented textures to a mesh, we describe, in the next chapter, how this method can be extended to render animations that will further enhance the illusion of shape. In Chapter 4 we explore a simple procedure to animate the contour lines over a surface. In addition, we divide the surface to achieve a cleaner animation that is easier to interpret visually.

CHAPTER 4

ANIMATION AND SEGMENTATION

To expand upon the texturing method introduced in the previous chapter, we now demonstrate how these textures can be animated to enhance the impression of shape. Further, the results can be clarified using a technique to segment the model.

For animation, there is less of a precedent to draw from, for while one might see countless illustrations which include contour lines, examples showing contour lines that are specifically animated for the purpose of conveying shape are uncommon. Computer graphics research has shown that rendering an animation of a particle system traveling across an ambiguously-lit three-dimensional surface can enhance the perception of the shape of that surface [LuSM 02]. This “kinetic visualization” is based on the casual observation of flowing water and how its motion indicates the rocks beneath. In addition to computer graphics research, psychological research in this topic has shown that users were more readily able to identify key topographical points when viewing surfaces with the kinetic element added [LuSM 02].

4.1 Simple Animation

Recall that in Chapter 2 we discussed animation methods such as OLIC that can achieve adequately illustrative results simply through color manipulation over a simple image map. A similar procedure can be used to animate the image map used in the oriented texturing algorithm presented in this paper. One would then apply the animated texture image to the geometry, rather than the static image assumed in the previous chapter. Much like the kinetic visualization method, our animation will be highly directional. The animation would be repeated over the surface for every patch; therefore, the animation must be one that is visually effective despite being distributed in repetition.

The animated texture is similar to the line pattern used in the previous chapter. The texture is composed of a series of stripes, but now these stripes are drawn as a gradient ramp of grayscale colors. In each frame, the color ramp is moved and the colors of the stripes are rotated accordingly giving the appearance of motion. In this method, the absolute minimum number of stripes per repetition is three, while much smoother results can be obtained using six.

4.1.2 Determining Direction

The actual direction that the texture is observed to move is related directly to the texture orientation. The rendering of the Stanford bunny in Figure 4.1, shows the striped pattern oriented in the direction of the locally computed vectors. The motion vector for each patch is therefore in the secondary PCD. Unfortunately, the animation presents a problem with directionality not present when applying the texture statically. Though we compute a single secondary PCD vector for each patch, the curvature in the reverse direction is essentially identical; therefore, we cannot be certain which direction will be selected for the secondary PCD. This uncertainty is not a problem with a static texture, which is symmetrical in that direction. The animation is directional, however, and this ambiguity in orientation direction produces disruptions in the resulting animation.

To eliminate these unnecessary vectors, the algorithm to find the secondary PCD can be adjusted slightly to pull from only one half of the available tangent space. Any result that falls outside the half can be easily reversed by multiplying the vector by -1. Though the edges of this artificial range will still introduce ambiguity, this adjustment will greatly reduce the number of disparately oriented animated patches.

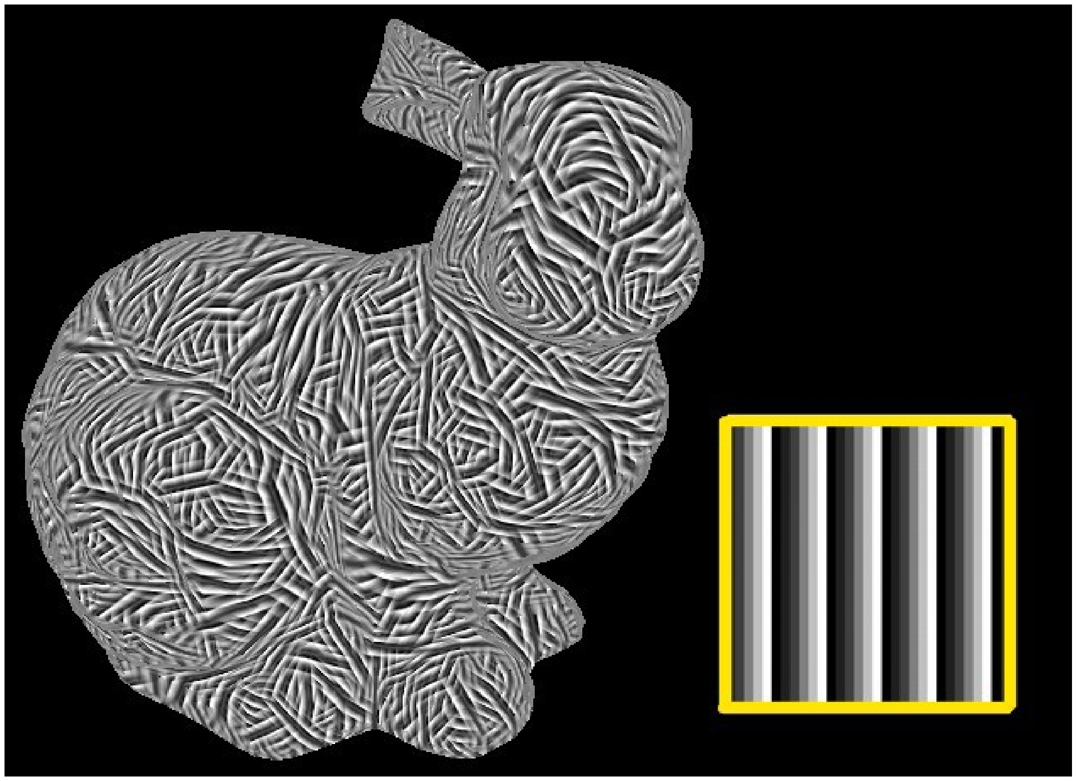

Figure 4.1 *Model Textured with the Six-Stripe Color Ramp*

Additionally the direction of the PCD vector can be purposefully manipulated to control the direction of animation to suit the application at hand. However, there is no general scheme that is considered definitively correct. In the general case where the animation is allowed to flow in the default direction, the result will be animation that is generally coherent over patches with similar orientation.

4.2 Segmentation for Clearer Animation

While we focus on techniques that utilize principal curvature directions, when a subject is sufficiently complex, these methods often provide us with complex data sets that are difficult to visually interpret and may result in a display that is unsatisfactory to convey the intended information. The rendering of the Stanford bunny in Figure 4.1 is arguably one such case.

We endeavor to find a solution that is more intuitive; hence we consider how a human being might approach the same problem. Given the body of a horse for example: an artist wishes to illustrate the horse using contour lines to convey shape. It is unlikely that the artist will survey a large sample of points on the horse's surface and consider the curvature separately for each one. Though this task is easy enough for a computer program, more than likely the artist will *generalize* the curvature of the object.

An artist might consider the legs of a horse and decide that they look roughly cylindrical, so the contour lines should wrap around the legs. The body of the horse can be similarly generalized, but the artist is likely to draw those lines arcing in a different direction, according to the curvature. In fact, illustrations in which contour lines are rendered show that an artist will tend to use cylindrical approximations to determine the direction of the contour lines. In addition, contour lines typically do not reflect small details and variations in the surface curvature, but rather give a broad indication of the overall shape [HeZo 00].

The use of approximation and generalization in programmatic techniques may yield results that are more familiar to an observer. This prospect is especially promising for use in animation, where a quick interpretation of any given frame is crucial to perception as a whole.

Consider how one might then animate contour lines in an intuitive way. We might expect to see contour lines move across the length of the object, perpendicular to the direction over which they are oriented. Given this constraint, however, there are still two possible directions that lines can travel (we can say “forwards” and “backwards”, in relative terms). In the case of the horse, we may expect the lines to begin at the center and extend outward to the extremities. Lines would travel from the body upward across the neck and downward toward each of the hooves.

Here we outline a procedure to render animated contour lines over an arbitrary smooth surface according to this concept. The process is to divide the geometry into parts. For each part we generalize the principal curvature direction and render lines oriented along those directions. Finally we animate these lines, determining the best direction for the lines to flow.

4.2.1 Approximate Convex Decomposition

Approximate convex decomposition is the name of the method used to segment the model. Convex decomposition, in general, is the practice of decomposing an arbitrary mesh into the minimal number of purely convex components. The technique is very useful for adapting an algorithm that would normally require a simple or convex mesh, and applying it to a mesh of arbitrary complexity. Pure convex decomposition is not practical here, however. Many of the types of meshes we intend to use are complex enough that they could possibly break down into tens of thousands of parts.

One technique an artist uses to determine how he will render depth cues is to imagine a complex subject as a collection of basic shapes, but it would be overwhelming for an artist to imagine a subject as a collection of many thousands of shapes. The artist therefore considers only an approximation of the geometry. Since we intend to obtain results that are intuitive to observe, it makes sense that we do the same. In fact, one concept of approximate convex decomposition subdivides the model into meaningful parts.

One particularly effective algorithm for this subdivision was developed by [LiAm 04, LiAm 06]. Viewing the results in figure 4.2, it is apparent that by using approximation to decompose the model, an intuitive separation can be achieved. In the case of the example elephant model, the components are identifiable as the head, tusks, trunk, ears, body, and legs. The fact that all of these components are nameable indicates that a human being would conceptualize a similar decomposition.

This algorithm is a three-dimensional extension of their method to approximate convex decomposition of two-dimensional polygons. The algorithm involves two basic steps. First, the concavity of the geometry must be evaluated to determine where key features are located. Second, the features are used to determine where to divide the model, and along what lines.

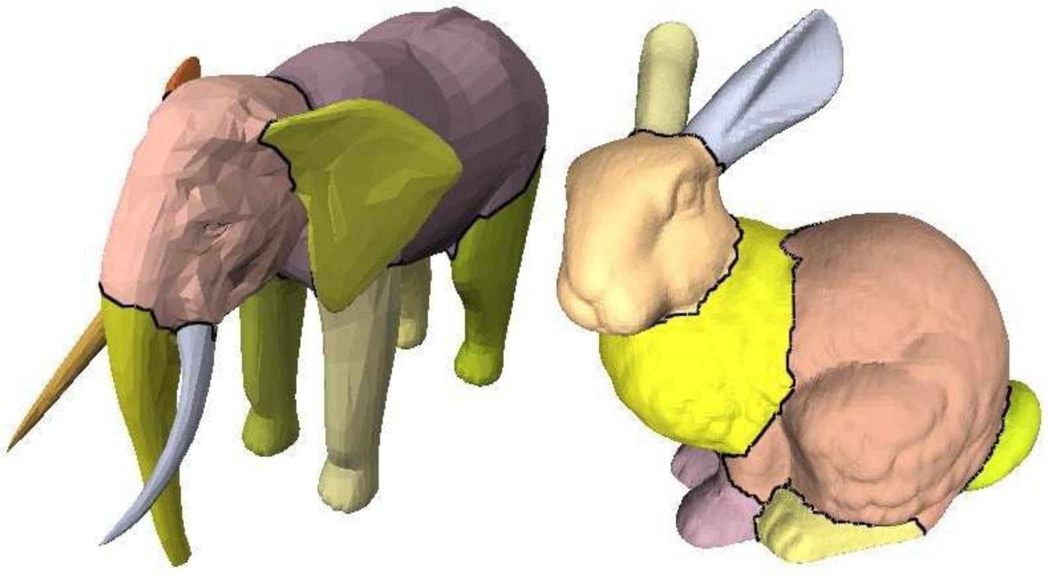

Figure 4.2 *Approximate Convex Decomposition* [LiAm 04]

Measuring the concavity of the model involves the concept of bridges and pockets. Bridges are defined as expansive convex approximations of an object that broadly cover an object's surface, intersecting at some points and sweeping over others. When these bridges enclose an object in space, they collectively form a convex hull around that object. Pockets are the concave geometry underneath the bridges. The concavity of a pocket can be measured by surveying the distances between vertices on the surface of the pocket, and a plane implied by the surface of the bridge. Areas of measured concavity are called features. Note that a bridge is constructed over the original surface model, whereas a pocket is an identified portion of the original surface of existing geometry.

In order to segment the model into convex segments, we divide the model at the features. Because the intent is to make this division arbitrarily approximate, we sort the features according to their measured concavity. Beginning with the area of greatest concavity, we divide the model at each subsequent feature until a specified threshold is reached. To determine how the model is divided, key points, called knots, are identified in the concavity of a feature and are grouped together. Eventually a line is formed that indicates where the division is to be made.

This algorithm will generally work for surfaces that are identified as "genus zero," or contain no "handles" (also referred to as homological loops). These areas of geometry are likened to the handle of a coffee cup, where a branching segment rejoins the model, forming a hole. If the model has such topography, it must be reduced to a set of genus zero models. This creates a preliminary step where this type of topography must be identified and processed [LiAm 06].

4.2.2 Implementation

For this project, we use a preexisting implementation of this algorithm by [Ratc 07]. The code is written in C++ and is confined to the standard (std) library. Though originally intended to be an implementation of the algorithm described above [Ratc 07], an alteration to the procedure reduces its usefulness to decomposing a model into meaningful parts.

The intent of this application is to create a simplified version of the input model constructed from a small number of convex hulls. The simplified model is to be used in physics simulations as a reduced-polygon surrogate for the complex geometry from which it was constructed. The result can be used in collision detection to reduce the number of polygon-to-polygon collisions that need to be computed. Because the application does not require meaningful feature extraction, this element of the implementation was not fully developed. Instead, the program recursively divides the model in half for a user-specified number of iteration steps. The pieces are converted into convex hulls and then rejoined in cases where the resulting rejoining would also be convex. The output is a number of convex hulls that approximate the shape.

For the purpose of our experiment, the implementation was altered to output the divided original geometry, rather than the convex hulls. This did not represent a significant modification to the algorithm, however. The problematic aspect of this altered procedure is that divisions in the surface are no longer guaranteed to be in places of greatest surface concavity. While results can be improved by altering the operating parameters (in particular, setting the number of recursive iterations to be very high), arbitrary divisions in the model are difficult to eliminate. Models processed using this algorithm are not ideal, but they should be adequate for our purpose. One small consolation is that the altered procedure no longer has the “genus zero” requirement that constrained the original.

More specifically, the actual procedure takes the following steps to segment the model:

1) Generate a convex hull around the model by generating bridge planes over the surface.
2) Roughly estimate the difference in volume between the geometry and its hull by tracing the distance from points across the surface of the model to nearest points on the hull.
3) Compare this difference to a predetermined threshold. If the difference is small enough, consider the model to be convex.
4) If the difference is greater than the threshold, divide the model in half along an arbitrary, axis-aligned plane. Repeat steps 1-4 until no more recursion can be done. This process will generate a number of roughly complex pieces.
5) For each pair of adjacent pieces, join them temporarily and create a hull around them, the same as step 1.
6) Estimate the difference in volume between the combined pieces and their hull, the same as in step 2. If the difference is less then the threshold, combine the pieces permanently. Together they form a roughly convex object.
7) If the threshold is exceeded, separate the pieces again.
8) Repeat steps 5-8 for all new combinations created.

In addition our procedure introduces a new potential source of error. To compute the curvature extrema at a point as discussed, we require that the point be part of a continuous surface, at least locally. Since we divide the model into segments, there are now several vertices that rest on the edges of the surface segments. Values computed at these vertices will not be accurate. When possible, this scenario can be avoided by computing the curvature extrema before segmenting the model. Short of that, reconstructing a local surface patch if corresponding points can be found on the adjoining segments would not be difficult. In our case, since we intend to generalize this information for each segment, these vertices can be ignored as long as enough surface vertices remain to indicate the shape of the surface segment.

For all intents and purposes, it is easy to determine whether a vertex is at the edge of the surface segment or not. Recall that according to our data structure, a triangle is represented as a set of three indices to vertices. If we are examining a vertex at a point on the surface that is completely surrounded by triangles, i.e., in a locally continuous portion of the surface, then each triangle is necessarily adjacent to two others. Therefore, in addition to the center vertex that is shared by every triangle in the neighbor-set, each triangle will share its remaining vertices with two other triangles in the neighbor-set. If a triangle contains a vertex that is not shared by another triangle in the neighbor-set, then the vertex in question is at an edge of the segment.

4.3 Determining Direction and Order

Now that the model is segmented, the final step is to simplify the PCD vector field for the sake of the animation. To do this we find the mean of the untranslated PCD vectors in each Darboux frame. Recall that the Darboux frame stores the vertex location, normal, and PCD vectors and values. For each of these frames, we additionally store the untranslated PCD vectors because they are defined relative to the same basis, and are thus comparable to one another. We also store the original *R1* and *R2* rotation vectors, so that the new PCD vector can be translated back to each of the original locations.

The result is a greatly simplified surface texture; however, the surface maintains key directionality from PCD calculations, making it advantageous over non-adaptive contour line rendering schemes. Adjusting parameters in the segmentation algorithm allows control of the number of segments into which the model is subdivided. More segments will demonstrate more accuracy, but lose clarity. Figure 4.3 illustrates the result of this procedure. The broad dark lines show the separation of segments.

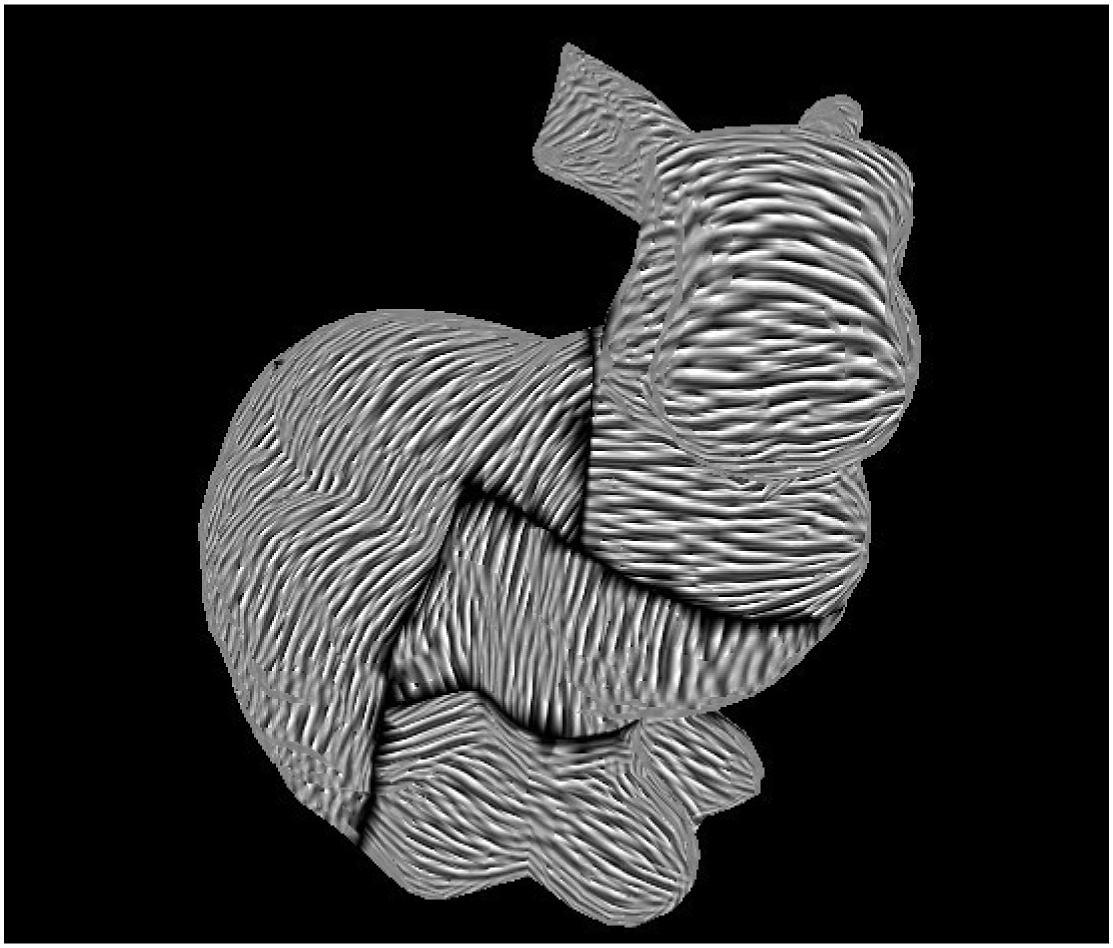

Figure 4.3 *Segmented Model with Generalized Oriented Texture*

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Results

While several methods have been developed to construct a visualization of a model that is based on principal curvature directions to enhance the appearance of shape, these involve either geometric construction or procedurally generated texture images. We successfully demonstrate, however, that such a visualization can be created simply by affecting the orientation of a repeating texture across the model's surface. We further demonstrate that this viewing experience can be enhanced through the concept of kinetic visualization.

The texturing method, despite its limitations, is versatile due to the fact that it uses simple texturing and is thus able to exploit the wide variety of texture features offered by graphics rendering software and devices. After the initial time taken to compute the PCD vector field, the method is also quite capable of rendering in real-time, both with and without the added element of animation.

Chapter 2 discussed algorithms for visualizing PCD vectors that are useful for conveying the shape of transparent objects. The algorithm presented in this paper is also capable of this feature, given the proper alpha-channel map and sparseness of stripes. Figure 5.1 shows a rendering of the Stanford bunny in blue bands. The model contains a teapot which can be seen between the bands. The input image is a simple pattern of two thin blue stripes on a transparent background. When mapped to a surface, it bears resemblance to other methods that render surface notches over a PCD vector field.

Figure 5.2 shows how the method to segment and simplify the model texture presents an intuitive render with the cost of some precision. This simplification enhances the effect of the animation because it creates a smooth area of coherent flow. The divide between the model segments is clearly visible, which will need to be corrected, but the renders are functional as a proof-of-concept.

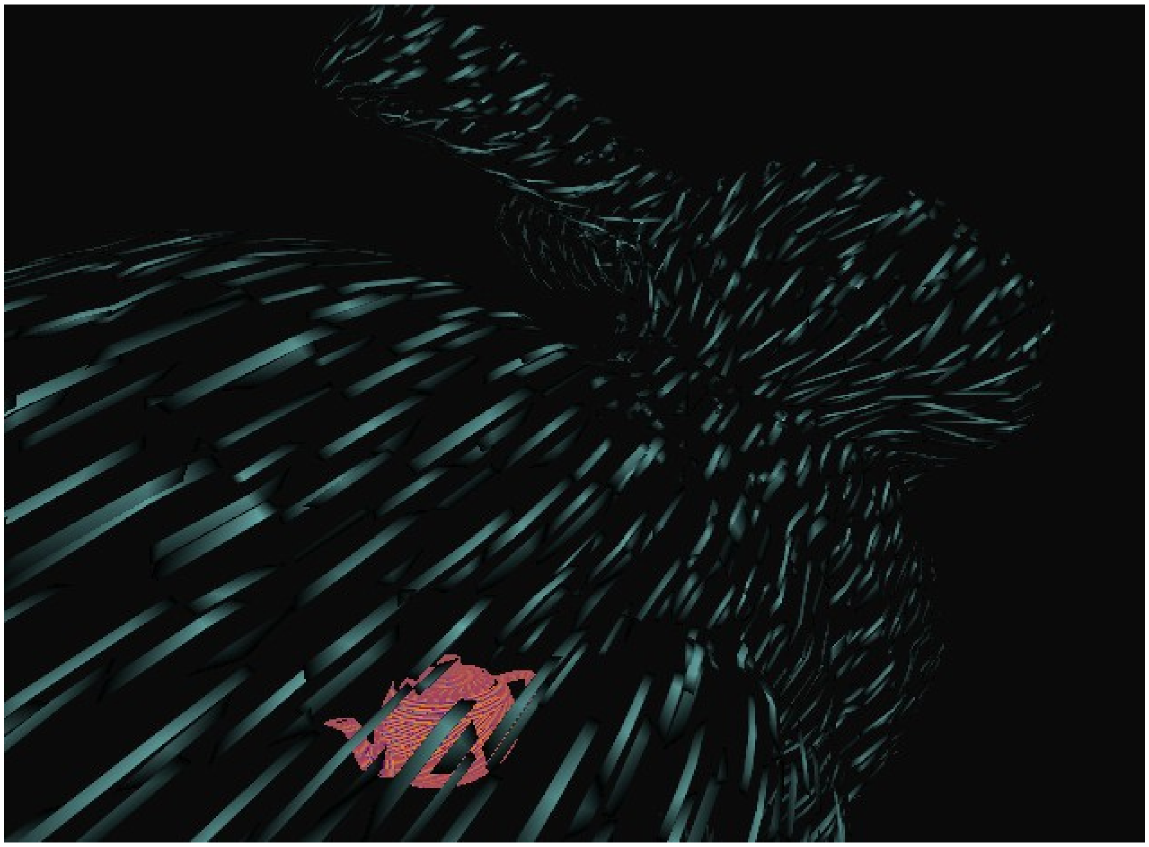

Figure 5.1 *Transparent Rabbit Showing Enclosed Red Teapot*

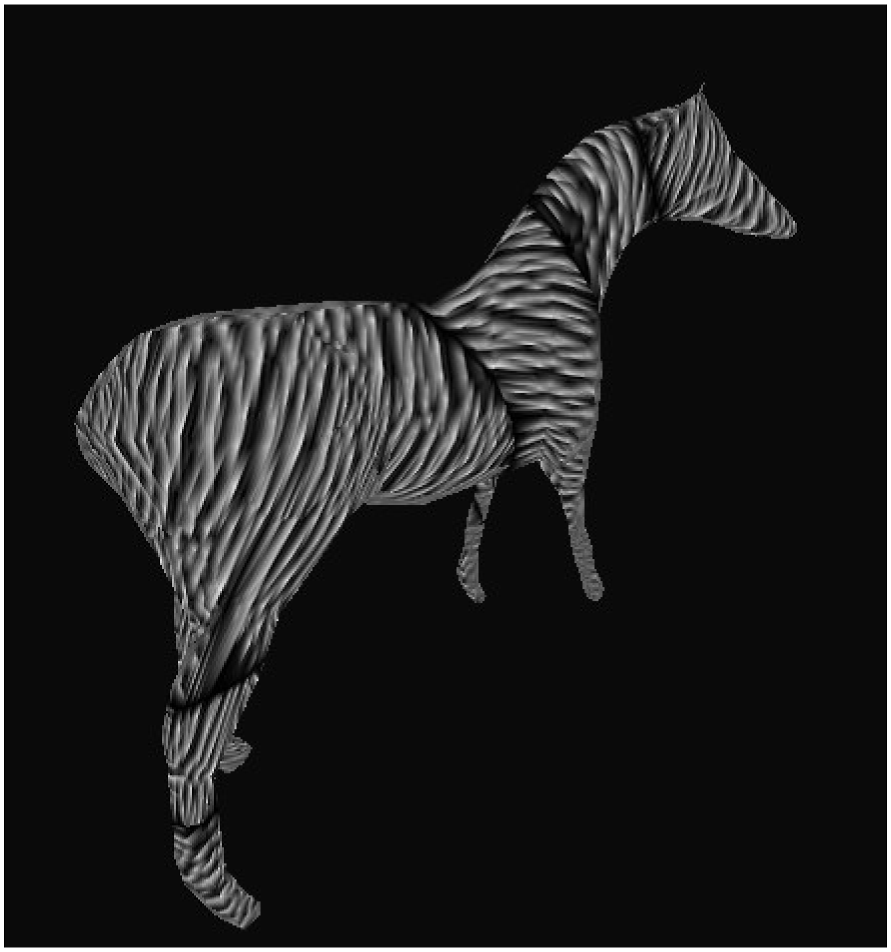

figure 5.2 *Segmented Horse*

In conclusion, these techniques are useful for rapid development of real-time interactive visualizations based on PCD vectors. If the set of PCD vectors were precomputed and stored to file, a negligible amount of time would be required to reconstruct this visualization. In addition, rendering of the visualization is dependent only on common-place techniques that are supported by most graphics rendering hardware.

5.2 Future Work

The project detailed in this paper could benefit from future work. Specifically, the techniques used to apply the texture coordinates, and also for segmenting the model into smaller components, are in their primacy. Certainly they can both be improved to create more impressive results. Convex segmentation may even be implemented through the use of the PCD vector space as surface areas with concavity will presumably contain troughs with a high degree of negative curvature.

Currently the texture application is limited in size by the triangle density of the target mesh. The algorithm is also susceptible to blockiness in appearance, due to the rigid orientation and placement of the mesh. Texturing could be improved if texture stripes could be distorted to bend more gradually with the input vectors. More coherency of the stripes could also be achieved by means of a more sophisticated surface parameterization algorithm. We believe that if the true algorithm described in [Prau 00] were applied, the results would be greatly improved.

Finally, the effectiveness of the results have not been formally verified by end user testing. Testing the results against use from multiple users in various applications would validate the work beyond informal acceptance.

In this paper we have demonstrated the beginnings of an effective algorithm to render visuals that are evocative of shape without relying on illumination. With these enhancements, this work has the potential to provide meaningful and necessary detail to models with complex surface characteristics.

REFERENCES

[BeGr 00] S. Berger and E. Groller. *ColorTable Animation of Fast Oriented Line Integral Convolution for Vector Field Visualization*, Proceedings of the 8th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media 2000, 2000, pp. 4-11.

[CaLe 93] B. Cabral and L. Leedom. *Imaging Vector Fields Using Line Integral Convolution,* Computer Graphics (SIGGRAPH 1993 Proceedings), Vol. 27, No. 4, 1993, pp. 263-272.

[Dege 03] P. Degener, J. Meseth, and R. Klein. *An Adaptable Surface Parameterization Method,* Proceedings of the 12th International Meshing Roundtable, 2003, pp. 201-213

[deLe 95] W.C. de Leeuw and J. Wijk, *Enhanced Spot Noise for Vector Field Visualization*, Proceedings of Visualization '95, 1995, pp. 233-239

[Girs 00] A. Girshick, V. Interrante, S. Haker, and T. Lemoine. *Line Direction Matters: An Argument for the Use of Principal Directions in 3D Line Drawings,* Proceedings of NPAR 2000, pp. 43-52.

[Gold 04] J. Goldfeather and V. Interrante. *A Novel Cubic-Order Algorithm for Approximating Principal Direction Vectors,* ACM Transactions on Computer Graphics (SIGGRAPH 2004 Proceedings), Vol. 23, No. 1, 2004, pp. 45-63.

[Hake 00] S. Haker, S. Angenent, A. Tannenbaum, R. Kikinis, G. Sapiro, and M. Halle. *Conformal Surface Parameterization for Texture Mapping,* IEEE Transactions on Visualization and Computer Graphics, Vol. 6, No. 2, 2000, pp. 181-189.

[HeZo 00] A. Hertzmann and D. Zorin. *Illustrating Smooth Surfaces*. Computer Graphics (SIGGRAPH 2000 Proceedings), Vol. 34, No 3, 2000, pp. 517-526.

[Huan 05] A. Huang, R. Summers, and A. Hara. *Surface Curvature Estimation for Automatic Colonic Polyp Detection,* Proceedings of SPIE -- Volume 5746: Medical Imaging 2005: Physiology, Function, and Structure from Medical Images, Amir A. Amini, Armando Manduca, Editors, 2005, pp. 393-402.

[Inte 97] V. Interrante. *Illustrating Surface Shape in Volume Data via Principal Direction-Driven 3d Line Integral Convolution,* Computer Graphics (SIGGRAPH 1997 Proceedings), Vol. 31 No. 1, 1997, pp. 109–116.

[InFu 97] V. Interrante, H. Fuchs, and S. Pizer. *Conveying the 3D Shape of Smoothly Curving Transparent Surfaces via Texture,* IEEE Transactions on Visualization and Computer Graphics, Vol. 3, No. 2, 1997, pp 98-117.

[Levy 02] B. Levy, S. Petitjean, N. Ray, and J. Maillot. *Least Squares Conformal Maps for Automatic Texture Atlas Generation,* Computer Graphics (SIGGRAPH 2002 Proceedings), Vol. 21, No. 3, 2002, pp. 362-371.

[LiAm 04] J.-M. Lien, N. Amato. *Approximate Convex Decomposition* Proceedings of the Twentieth Annual Symposium on Computational Geometry, 2004, pp. 457-458.

[LiAm 06] J.-M. Lien, N. Amato. *Approximate Convex Decomposition of Polyhedra,* Technical Report, TR06-002, Parasol Laboratory, Department of Computer Science, Texas A&M University, 2006, http://parasol.tamu.edu/groups/amatogroup/research/app-cd/ (2007)

[LumM 01] E. B. Lum and K.-L. Ma. *Non-Photorealistic Rendering Using Watercolor Inspired Textures and Illumination*. PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications, 2001, pp. 322-330.

[LuSM 02] E.B. Lum, A. Stompel, and K.-L. Ma. *Kinetic visualization: A technique for illustrating 3d shape and structure,* Proceedings of IEEE Visualization, 2002, pp. 435-442.

[MSSS 07] *Mars Orbiter Camera: Highest-Resolution View of "Face on Mars,"* Malin Space Science Systems web page, http://www.msss.com/mars_images/moc/extended_may2001/face/index.html (2007).

[Pend 05] N. Pendluru. *3D Shape Representation Using Principle Direction Oriented Textures,* Master's thesis, Clemson University, 2005.

[Prau 00] E. Praun, A. Finkelstein, and H. Hoppe. *Lapped Textures,* Computer Graphics (SIGGRAPH 2000 proceedings), Vol. 34, annual, 2000, pp. 465-470.

[Pres 07] A. Pressley. Elementary Differential Geometry. Springer-Verlag, London Ltd. 2007.

[Ratc 07] J. Ratcliff http://codesuppository.blogspot.com/2006_08_01_archive.html (2007).

[Sals 97] M. P. Salisbury, M. T. Wong, J. F. Hughes, and D. H. Salesin, "*Orientable Textures for Image-Based Pen-and-Ink Illustration*," Computer Graphics (SIGGRAPH 1997 proceedings), Vol. 31, Annual, 1997, pp. 401-406.

[StHe 95] D. Stalling, and H.-C. Hege. *Fast and Resolution Independent Line Integral Convolution*, Computer Graphics (SIGGRAPH 1995 proceedings), Vol 29, Annual, 1995, pp. 249-256.

[Sund 03] A. Sundquist. *Dynamic Line Integral Convolution for Visualizing Streamline Evolution*, IEEE Transactions on Visualization and Computer Graphics, Vol. 9 No. 3, 2003, pp. 273-282.

[WeGP 97] R. Wegenkittl, E. Groller, G. Purgathofer. *Animating flow fields: rendering of oriented line integral convolution*. Computer Animation '97,. IEEE Computer Society Press, 1997, pp. 15-21.

[Weis 07] Weisstein, E. *Second Fundamental Form*. MathWorld-A Wolfram Web resource http://mathworld.wolfram.com/SecondFundamentalForm.html, (2007)