A SCAN-LINE SUBDIVISION APPROACH TO PERSPECTIVE TEXTURE MAPPING

By

DAVID S. SANDERS

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1994

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
December, 1996

A SCAN-LINE SUBDIVISION APPROACH TO

PERSPECTIVE TEXTURE MAPPING

Thesis Approved:

[illegible]

Thesis Advisor

J Chandler

[illegible]

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to thank my advisor, Dr. K. M. George, along with my other committee members, Dr. Chandler and Dr. Hedrick, for their support and faith in my work.

In addition, I would like thank my family and friends for their support during the creation of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In 1974, Ed Catmull introduced an algorithm for displaying realistic 3D surfaces on a computer screen [CATM74]. The algorithm "paints" a digitized picture onto a 3D surface and displays the resulting "textured" surface onto the screen. This is a process which has come to be known as *texture mapping*.

Texture mapping actually involves two mappings: One from *texture space* to *object space*, and then another from object space to *screen space*. Texture space is a coordinate system where texture representations are defined. In this paper, texture space is a 2D coordinate system with coordinates (u,v), where 0<=u,v<=1. As with Catmull's algorithm, the textures in this thesis are represented by an image. Object space, on the other hand, is a coordinate system where objects are defined. For this study, objects are defined in a 3D coordinate system by the coordinates (x,y,z). Last, screen space is basically where screen display information is accessed. In this thesis, screen space is 2D and represented by the coordinates (s_x, s_y). The screen is also represented by a bounded 2D plane cutting through 3D object space, orthogonal to the z-axis.

To simulate how the eye perceives light from objects in the real world, the mapping from 3D object space to 2D screen space may be a perspective mapping or *perspective projection*. With this type of projection, objects which are closer to the screen are made to look larger than those which are farther from the screen. An object's z-coordinate is used to determine how close that object is from the screen, and therefore how far it is from the viewer's eye. Texture mapping which incorporates this perspective projection is called *perspective texture mapping*.

Perspective texture mapping algorithms can generally be divided into three categories: *Perspective mapping, inverse perspective mapping,* and *affine mapping*. Perspective mapping algorithms map forward from texture space to object space and then from object space to screen space, whereas inverse perspective mapping algorithms map in the opposite direction. Affine mappings are translated linear mappings, not perspective mappings, but can be used for perspective texture mapping in rare instances (Chapter 2). Some algorithms take this approach.

The differences between perspective texture mapping algorithms are not just which of the above three categories they fit in, but also what intermediate mappings they use. Catmull's algorithm uses a subdivision approach to map from texture and object space to screen space, whereas other algorithms generally use a parametric mapping approach. Most of the theory that has been developed until now has been based on making the parametric approach more efficient.

In this thesis, I revisit the subdivision approach with a new algorithm called the *scan-line subdivision algorithm* (Chapter 5). The approach simplifies the texture mapping process, and is easily adaptable to different levels of filtering. This allows the algorithm to dynamically trade speed for realism and vice versa.

As with all algorithms, this algorithm has some minor disadvantages. One is that since it texture maps only the basic triangle, 3D object silhouettes are not always smooth. Another is that because it subdivides, it has the overhead of having to implement a stack.

1.1 Organization

Chapter 2 contains a review of the evolution of texture mapping theory, beginning with Catmull subdivision. The chapter divides these theories into three main groups: perspective, inverse perspective, and affine texture mapping. It also covers some filtering methods and issues that apply to texture mapping. Chapter 3 discusses Catmull subdivision. A description for subdividing the cubic curve, as well as the bicubic surface, is presented. Chapter 4 outlines a fast, general purpose, inverse perspective mapping algorithm for texture mapping the basic triangle. Chapter 5 describes the scan-line subdivision method for perspective texture mapping. The chapter describes how this method combines the qualities of Catmull subdivision with the advantages of inverse perspective mapping. The paper concludes with a summary in Chapter 6.

1.2 Keyword Definitions

Several terms used in the thesis are defined below:

Filtering	Refining, altering, and/or reconstructing data.
Object Space	The 3D coordinate system where objects are defined.
Pixel	The smallest "picture element" that can be accessed in screen space.
Projection	A mapping from 3D space to 2D space.
Screen Space	The 2D coordinate system where the computer screen display information is defined.
Texel	The smallest "texture element" that can be accessed in texture space.
Texture Map	A representation of real-world texture, generally in the form of a 2D image.
Texture Mapping	The process of mapping simulated texture onto a 3D object and displaying the textured object onto a computer screen.
Texture Space	The 2D coordinate system where texture is defined.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Texture mapping, the process of mapping simulated texture onto a computer generated surface, was pioneered by Ed Catmull [CATM74]. His technique was to subdivide a mathematical surface, along with a digitized image, until the resulting subdivided surfaces covered only one screen pixel when *projected*, or mapped onto a two dimensional computer screen. The color for each pixel was then taken from the resulting subdivided images, giving the appearance of an image painted onto a surface (see Figure 2.1a).

It has since been shown that texture does not have to be simulated using only an image. Surface bumps can be simulated by mapping a pattern of perturbed surface normals onto an object [BLIN78a]. This process is called *bump mapping*, which is a form of texture mapping. Although many kinds of texture mapping exist, this thesis focuses on a particular type called *perspective texture mapping*.

Perspective texture mapping is the texture mapping of a 3D object which will undergo a *perspective projection* onto a 2D computer screen. This implies that two mappings will take place: One from an image or *texture map* onto a 3D object, and then another from the 3D object onto a 2D computer screen (see Figure 2.1b). Usually these two mappings are composed to form one mapping from 2D texture space to 2D screen space, leaving out the intermediate 3D object space [HECK86].

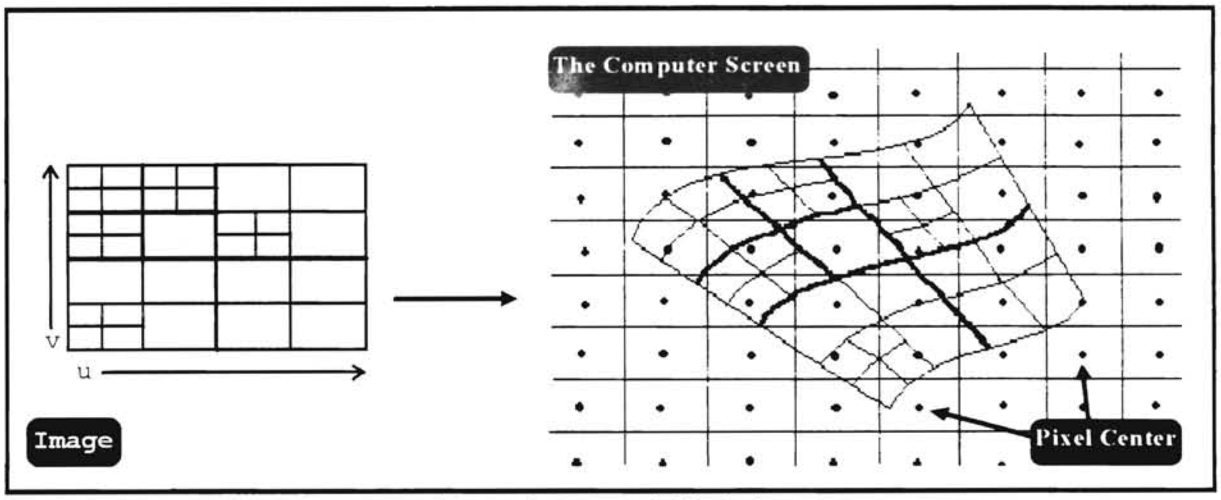


Figure 2.1a Subdivision

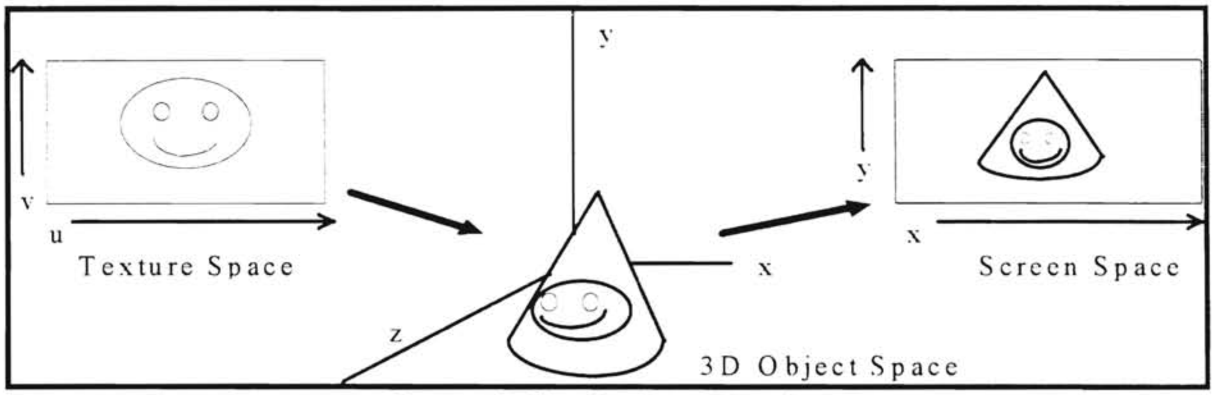


Figure 2.1b Texture Mapping

2.2 Theoretical Development

Since Catmull's subdivision algorithm, other techniques have been developed to implement perspective texture mapping. These techniques can usually be divided into three categories: Perspective mapping, inverse perspective mapping, and affine mapping. One thing common about these techniques is that they strive to mimic the analog world using digital methods. Because of this, digital signal theory plays an important role in perspective texture mapping. The following sections explain some of the techniques used to implement perspective texture mapping, as well as some digital signal processing techniques found in the literature.

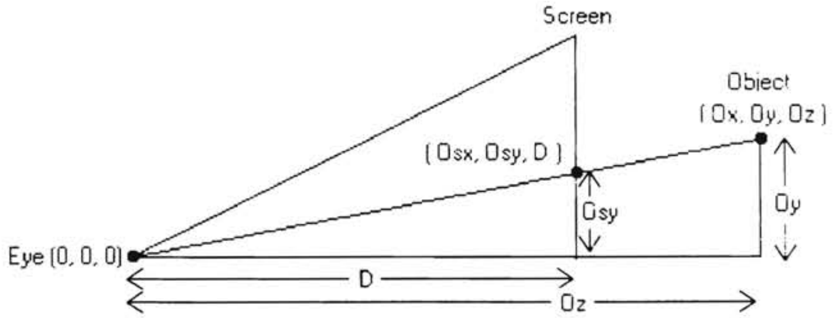
2.2.1 Perspective Mapping

If a 3D surface to be textured can be parameterized, then individual points on that surface can be mapped to a point in texture space and vice versa. Triangles are easily parameterized using the following equations:

$$x = (x_1-x_0)u + (x_2-x_0)v + x_0,$$
$$y = (y_1-y_0)u + (y_2-y_0)v + y_0, \text{ and}$$
$$z = (z_1-z_0)u + (z_2-z_0)v + z_0.$$

Where (x_i,y_i,z_i) are vertices of the triangle, (u,v) are texture coordinates where $0<=u,v<=1$, and (x,y,z) are triangle coordinates.

If a mapping from 2D texture space to 3D object space, such as the one above, is combined with a perspective projection from 3D object space to 2D screen space, the result is a *perspective mapping*. (A perspective projection is a mapping from 3D object space to 2D screen space which simulates perspective. One such mapping can be found via the law of similar triangles (see Figure 2.2.1).)



$$\frac{Osy}{D} = \frac{Oy}{Oz}$$

Ox,Oy,Oz = Object's 3D coordinates.

Osx,Osy = Object's screen coordinates.

D = Eye distance from screen (determined from preferred viewing angle).

$$Osx = D * Ox / Oz$$

$$Osy = D * Oy / Oz$$

Figure 2.2.1 Law of Similar Triangles.

The two mappings mentioned above, the parametric mapping from 2D texture space to 3D object space, and the perspective projection from 3D object space to 2D screen space, can be composed. Using matrix notation, a perspective mapping of a planar texture can be expressed as:

$$[\ xw,\ yw,\ w\ ] = [\ u,\ v,\ 1\ ] \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} \qquad \text{[HECK83]}.$$

Solving the equation above gives:

$$x = \frac{Au + Bv + C}{Gu + Hv + I} \qquad y = \frac{Du + Ev + F}{Gu + Hv + I}$$ [MAXW46].

In the above equation, the coefficients A..I are defined as follows: $A = x_1 - x_0$, $B = x_2 - x_0$, $C = x_0$, $D = y_1 - y_0$, $E = y_2 - y_0$, $F = y_0$, $G = Z_1 - Z_0$, $H = Z_2 - Z_0$, and $I = Z_0$. The values x and y represent the resulting screen coordinates. (Note how closely this resembles the earlier triangle example combined with the perspective projection.)

This method of texture mapping is fairly straightforward, but it can also be very calculation intensive. The reason is that one division per *texel*, or texture pixel, is generally required in order to perform the perspective projection. Since different points on a surface may project to the same screen pixel, averaging of that pixel's corresponding texel values should generally be done. This provides good *antialiasing*, but is also very expensive. (Antialiasing is covered later in the chapter.) Catmull and Smith demonstrate a method of perspective mapping using a unique two-pass shear and scale technique [CATM80].

2.2.2 Inverse Perspective Mapping

The mapping in section 2.2.1 is from texture space to screen space. It has an inverse which maps from screen space to texture space:

$$[\ uq,\ vq,\ q\ ] = [\ x,\ y,\ 1\ ] \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} =$$

$$[\ x,\ y,\ 1\ ] \begin{bmatrix} EI-FH & FG-DI & DH-EG \\ CH-BI & AI-CG & BG-AH \\ BF-CE & CD-AF & AE-BD \end{bmatrix} \qquad \text{[HECK83]}.$$

Instead of scanning texture space to find the corresponding screen coordinates, screen space is scanned to find the corresponding texture coordinates. This method generally requires only one division per screen pixel [SMIT80]. Aoki and Levine demonstrate this method for generating realistic images [AOKI78].

2.2.3 Affine Mapping

Affine mappings are translated linear mappings. A perspective mapping is affine iff $g=h=0$ and $i\neq0$ [HECK91]. In other words, the mapping from 2D texture space to 3D object space is typically affine, but a perspective projection from 3D object space to 2D screen space is not. Texture can be linearly interpolated to fit a 3D object, but a textured 3D object cannot generally be linearly interpolated to fit its projected screen image and still maintain correct perspective. Despite this general rule, there are a few texture mapping methods which do affinely map from 3D object space to 2D screen space while still preserving perspective. One method, *constant-z texture mapping*, takes advantage of the fact that g=h=0 when a plane in 3D space is parallel to the screen plane. It scans a 3D object one z coordinate at a time, holding z constant and affinely mapping that portion of the object which is cut through by this constant-z plane. Another method affinely maps objects, but reinterpolates every few pixels in order to give the appearance of a perspective mapping. These methods are fast, but generally trade realism for speed.

2.2.4 Filtering

The discrete *digital sampling* of a continuous signal, such as light, requires some form of digital signal processing or *filtering* in order to reconstruct lost data [FOLE90]. For the same reason, filtering is also needed when a digitized texture is mapped onto a computer screen. The following sections cover a well known problem in computer graphics called aliasing, and touch on some of the various types of filtering as found in the literature.

2.2.4.1 Aliasing

Aliasing occurs when a signal has unreproducible high frequencies [CROW77][WHIT81]. It causes the infamous jaggy lines that plague the computer graphics world. A mathematical line is continuous, but the computer screen is discrete and generally displays at too low of resolution to reproduce the line accurately. Two solutions to this problem are (1) sample at a higher resolution and (2) do low-pass filtering of the signal before sampling [HECK86].

Sampling the texture at a higher resolution does not necessarily imply that the computer screen needs to be at a higher resolution. If multiple samples of texture values map to a single screen pixel, then the color and intensity of these values can be averaged together and stored at that pixel location. The trade off is aliasing for noise [COOK86].

The second solution is to band-limit the signal before sampling. This means keeping the frequencies of the signal below the Nyquist limit [HECK86]. In other words, lower the frequency and/or resolution of the texture map until it is reproducible by the computer screen when projected. The methods used to do this are well developed for linear mappings [OPPE75], but only a few methods have been introduced for nonlinear mappings such as the perspective projection [HECK86].

2.2.4.2 Space-invariant and Space-variant Filtering

When sampling texture space for use in filtering, often a group of neighboring texels are sampled together. The sample shape and area is determined by the *filter* used. Space-invariant filtering uses a filter shape that remains constant as it moves across the texture map. This form of filtering works best with affine mappings because of their linear correspondence with texture space.

Space-variant filtering, on the other hand, uses a filter size and shape that varies as it moves across the texture map. This type of filtering is good for perspective texture mapping because it accounts for the nonlinear foreshortening caused by the perspective transformation. Space-variant filters are less understood and generally more complex than space-invariant filters [HECK86].

2.2.4.3 Direct Convolution

Computing weighted averages of texture samples in the filtering process is called *direct convolution*. Catmull's subdivision method covered earlier performs an unweighted average of the texture pixels corresponding to each screen pixel. Blinn and Newell improved upon this with a triangular filter and a weighted average [BLIN76]. Feibush, Levoy, and Cook furthered the process with a filter function that allows for several different filter shapes [FEIB80]. Since then, other methods have been developed such as elliptical weighted average (EWA) filtering [GREE86]. This method combines some of the earlier techniques but is less costly [HECK86].

CHAPTER 3

CATMULL SUBDIVISION

In 1974, Ed Catmull introduced a method for texture mapping curved surfaces. The method subdivides a 3D surface patch successively into smaller subpatches until a patch is as small as one screen pixel, at which time it is displayed [CATM74]. These surface patches are defined by the bicubic surface equation, as opposed to other surface representations, because it more closely models smooth, free-form, curved surfaces. Pictures are easily mapped onto these surfaces by subdividing a digitized image along with the surface, and then using the color information from the resulting sub-images to color the screen pixels. Two problems with this method are that the computation time increases roughly as the square of the resolution, and that the application of anti-aliasing techniques is not straightforward [BLIN78].

3.1 The Bicubic Surface

One of the simplest ways to approximate a curved surface is to use planar objects such as polygons. The problem with doing this, however, is that it generally results in a rough-looking surface which has a silhouette made of straight-line segments. Another simple method to approximate a curved surface is to use quadric patches. While smooth in appearance, these surface patches are not suitable for representing free-form surfaces because they do not provide enough degrees of freedom to satisfy slope continuity between patches [CATM74]. The bicubic surface patch, on the other hand, maintains patch continuity while providing the degrees of freedom and smoothness required for modeling arbitrary forms.

Bicubic surface patches are based on a bivariate case of the cubic curve,

$$f(t) = at^3 + bt^2 + ct + d.$$

(The coefficients, a..d, determine the shape of the curve and are found using several methods which are beyond the scope of this thesis.) Subdividing the bicubic surface is much the same as subdividing the cubic curve. The problem is to find f(t) when f(t+h) and f(t-h) are known. First note that for the cubic curve

$$\begin{aligned} f(t \pm h) &= a(t \pm h)^3 + b(t \pm h)^2 + c(t \pm h) + d \\ &= a(t^3 \pm 3ht^2 + 3h^2t \pm h^3) + b(t^2 \pm 2th + h^2) + \\ &\quad c(t \pm h) + d, \end{aligned}$$

and

$$\begin{aligned} f(t+h) + f(t-h) &= 2a(t^3 + 3h^2t) + 2b(t^2 + h^2) + 2ct + 2d \\ &= 2f(t) + 2h^2(3at + b). \end{aligned}$$

Therefore,

$$f(t) = [\ f(t + h) + f(t - h)\ ]/2 - h^2(3at + b),$$

which is the average of the endpoints minus a correction term. The correction term can be generalized in the same manner:

If $g(t) = h^2(3at + b)$ then

$$g(t \pm h) = h^2(3a(t \pm h) + b),$$

and

$$g(t+h) + g(t-h) = 2h^2(3at) + 2bh^2 = 2g(t).$$

Therefore,

$$g(t) = [\ g(t+h) + g(t-h)\ ] / 2.$$

When this is all put together,

$$f(t) = [\ f(t+h) - g(t+h) + f(t-h) - g(t-h)\ ] / 2.$$

The method of subdividing cubic curves can be extended to bicubic surfaces. There are three components which describe the parametric bicubic patch in 3D object space: $X(u,v)$, $Y(u,v)$, and $Z(u,v)$. Each component can be considered as:

$$f(u,v) = F_1v^3 + F_2v^2 + F_3v + F_4\ ,$$

where

$$F_n = a_nu^3 + b_nu^2 + c_nu + d_n,\ \text{for } 1<=n<=4.$$

(Again, the coefficients, $a_n..d_n$, determine the shape of the surface and may be found using several methods which are beyond the scope of this thesis.) As can be seen, the bicubic surface equation is very close to the cubic curve equation, particularly when one of the variables is held constant. Since F_n is a cubic, we know that there is a correction term, G_n, for each F_n. We also note that f is a cubic curve when F_n is held constant, and therefore also has a correction term, g, where

$$g = G_1v^3 + G_2v^2 + G_3v + G_4\ .$$

Four values are needed to represent each defining point during patch subdivision. Catmull arranges these into a "register-square."

f	g
c_f	c_g

In the register-square, f is the value of the function at (u,v), and c_f, g, and c_g are correction terms [CATM74]. The c_f term represents the correction value for f when bisecting in the v direction, while g represents the correction term for f when bisecting in the u direction. Note that since g and c_f are also cubic curves when u or v is held constant, they also need a correction term. The term c_g serves as the correction term for c_f when bisecting in the u direction, and also serves as the correction term for g when bisecting in the v direction. These values are found using the same equations shown above for the cubic curve. As described, bisection is accomplished by holding u constant while bisecting in the v direction, and then by holding v constant while bisecting in the u direction.

3.2 Perspective

So far, a method for subdividing the bicubic surface in 3D object space has been given. In order to display a perspective view of the 3D surface on the computer screen, a perspective transformation must be performed between 3D object space and 2D screen space. This results in a rational bicubic with the surface equation of:

$$F(u,v) = \begin{bmatrix} X(u,v) \\ Y(u,v) \\ Z(u,v) \\ W(u,v) \end{bmatrix},$$

where W(u,v) is called the homogeneous coordinate and is generated by the perspective transformation. The three methods for displaying a perspective surface in screen space are:

1. Divide the surface components by W(u,v), which results in a rational cubic that does not fit into the subdividing scheme.
2. Subdivide X, Y, Z, and W and perform the perspective division at every point, which may considerably increase the complexity of the algorithm.
3. Take only the control points which make up the coefficients of the surface equation through the perspective transformation, then recreate the surface in screen space. This results in a very close approximation of the surface, but not the "correct" surface as described earlier in the chapter.

3.3 Termination

The decision about whether or not a patch is to be subdivided further depends upon the termination conditions. Catmull discusses two termination conditions which are based on patch size and clipping. As explained earlier, subdivision terminates when a patch covers only one screen pixel when projected. Since the edges of patches may be curved, Catmull suggests that a polygon be used to approximate the patch by connecting the four corners of the patch with straight line segments.

This allows faster determination of whether or not the subdivision should continue. The other termination condition, clipping, halts the subdivision when a patch is completely off the screen.

3.4 Hidden Surface Elimination

Hidden surface elimination seeks to avoid displaying surfaces which are behind other surfaces, and therefore out of view. Two methods are described by Catmull to solve the hidden surface problem for bicubic patches. These are the "modified Newell algorithm" and the "z-buffer algorithm." The Newell algorithm [NEWL73] sorts polygons in z-order, and displays the polygons which are furthest from the viewer first. If two polygons intersect so that their z-order is questionable, then they are divided into smaller polygons before being sorted. This method is modified for bicubic surfaces to sort certain control points which define the coefficients of the bicubic equation, rather than sorting some polygon's vertices. "The z-buffer algorithm," on the other hand, uses a buffer of values that represent the closest z-values displayed at each particular screen element. Before a point in object space is displayed at a point in screen space, its z-value is compared with the z-value stored at a corresponding location in the buffer. If the z-value for the point is greater than the z-value in the buffer, then the object is closer to the viewer. In such a case, its z-value replaces the z-value in the buffer and the point is displayed at that screen location. If the z-value for the point is not greater than the z-value in the buffer, then the buffer is left alone and the point is not displayed.

3.5 Mapping an Image onto the Bicubic Patch

Because bicubic surfaces are parametric, images can easily be mapped onto them. Each point on these surfaces are referenced by two variables, u and v, which can be made to correspond to points in texture space. Once a surface patch is completely subdivided and ready for display, the (u, v) coordinates for each corner of the patch may be used to define a sampling area in texture space. This area would then be used by a filtering algorithm to determine the final color value for the screen pixel to be displayed.

3.6 Basic Problems

One of the problems with this method is that the computation time increases roughly as the square of the screen resolution. For example, a square of 2x2 pixels needs only one subdivision, or 4^0 subdivisions. A square of 2^2x2^2 pixels needs 4^0+4^1 subdivisions, and a square of 2^nx2^n pixels needs:

$$\sum_{i=0}^{n-1} 4^i \quad \text{or } (4^n-1)/3 \text{ subdivisions [CATM74].}$$

At each point in the subdivision, a division by two for each surface component is performed, and a stack is used to store the four component values: f, g, c_f, and c_g. Also, each surface component must be transformed by the perspective projection when a termination condition is met. Because the surface to be mapped is curved, some points on the surface may hide behind others. This means that not every subdivision results in a displayed point.

Another problem occurs when filtering or anti-aliasing the sampled display values of the surface patch. These processes require techniques for determining what is visible in each raster element square, or pixel, and a method for storing and combining intensity values at each square to get an average [CATM74]. After termination, a subdivided patch to be filtered and displayed may lie between pixel centers, etc. Because of these types of issues, the filtering and anti-aliasing processes needed are not completely straightforward.

CHAPTER 4

FAST SCAN-LINE BASED TEXTURE MAPPING

Since Catmull's subdivision algorithm, methods have largely been focused towards improving the parametric approach to perspective texture mapping. Because of its properties, inverse perspective mapping has been one of the most widely accepted methods to date. This chapter describes and examines the general purpose inverse perspective mapping algorithm.

4.1 Inverse Perspective Mapping

As explained earlier, inverse perspective mapping is the mapping from screen space to object space, and then from object space to texture space. These two mappings are usually composed to form one mapping from screen space to texture space. Inverse perspective mapping differs from perspective mapping in that screen space is scanned to obtain a texture value, rather than the other way around. For each screen coordinates scanned, a texture coordinate is found which represents the center of a group of texture coordinates which also map to that screen coordinate (see Figure 4.1). This group of texture coordinates should not be ignored, since their combined values represent the final value of the current screen pixel being scanned. A filtering method is commonly used to decide which of these texture coordinates should be included, and how to combine their values to form the final screen value. As a general rule, the greater the number of texture values used to represent the final screen pixel value, the more realistic the texture mapping appears and the slower the algorithm performs.

Filtering seeks to balance between speed and realism. Inverse perspective mapping algorithms are popular because they are easily adjusted to balance between the two. For example, if speed is most critical, then only the center texture value may be used to color its corresponding screen pixel. This results in a crude representation of a texture, but is fast and generally recognizable. On the other hand, if realism is most critical, more texture values may be used to represent each screen pixel. The final image would appear more realistic because more data values would be used to represent the mapped texture. For this same reason, the algorithm would generally be slower in producing the image.

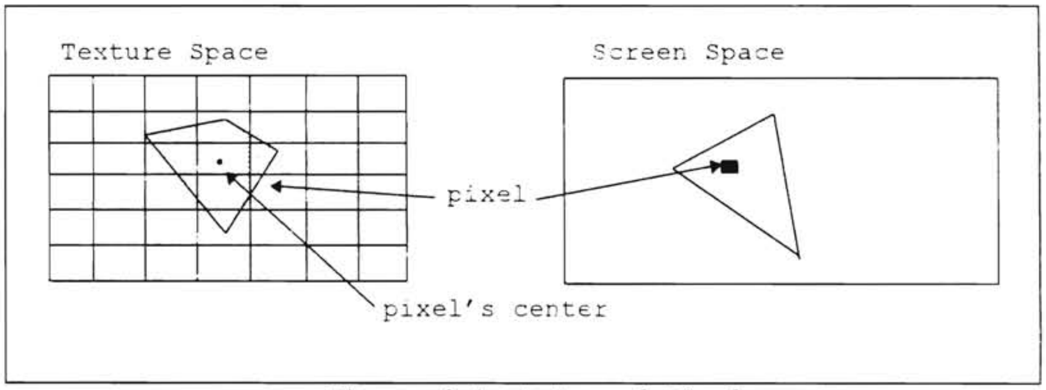


Figure 4.1 A Mapped Pixel

4.2 Representing Surfaces with Triangles

Almost any surface may be represented by a mesh of adjoining triangles. Triangles are easy to work with because they are both simple and planar. Planar objects such as triangles are usually used in inverse perspective mapping because they are easily parameterized. One of the problems with using triangles to represent surfaces, however, is that if a surface is non-planar, a triangle mesh can only roughly approximate it. This means that objects and their silhouettes may look somewhat rugged. One of the solutions to this problem, however, is to use smaller triangles to represent the surface. This is another trade-off between speed and realism. If smaller triangles are used to represent a surface, then more triangles must also be used to represent that surface. If more triangles are used, the algorithm may be slower, but the final image is generally more realistic. However, even in the real world, objects which seem smooth can appear rough when viewed through a microscope.

4.3 Texture Mapping in Scan-line Order

Inverse perspective mapping algorithms commonly scan screen space in scan-line order (left to right, top to bottom). This is because an algorithm which generates pixel values in scan-line order has two main advantages. The first is that because the intensity of each pixel is computed completely before moving on to the next, anti-aliasing computations are relatively easy to perform [BLIN78b]. The second main advantage is that these scan-line algorithms are more suitable for hardware implementations because they generate the intensities in the same order as a computer monitor scans them out onto the screen [BLIN78b]. The next sections describe and examine a general purpose inverse perspective mapping algorithm which performs fast scan-line based texture mapping of a basic triangle.

4.4 The Five Primary Steps

A general purpose scan-line algorithm for the inverse perspective mapping of a basic triangle can be divided into five primary steps:

Step 1. Calculate the mapping coefficients (section 2.2.2). This step matches the 2D rectangular texture space to the triangle in 3D object space. This process is made easier by associating each vertex of the triangle with a corresponding texture coordinate. Since we know that the three vertices of the triangle lie on the plane for which we are trying to find coefficients, we only need to solve a set of linear equations:

$$X(u_i,\ v_i) = A * u_i + B * v_i + C = x_i \ ,$$
$$Y(u_i,\ v_i) = D * u_i + E * v_i + F = y_i \ , \text{ and}$$
$$Z(u_i,\ v_i) = G * u_i + H * v_i + I = z_i \ ,$$

where $i=0..2$, and $(u_i,\ v_i)$ and $(x_i,\ y_i,\ z_i)$ are known. The coefficients, A..I, are then used to find the coefficients a..i (see section 2.2.2).

Step 2. Sort the three triangle vertices, (x_i, y_i, z_i), and their corresponding texture coordinates, (u_i, v_i), by their projected screen y values, Sy_i, where $i=0..2$. Designate vertex A the vertex with the smallest Sy_k, vertex B the vertex with the next smallest Sy_k, and vertex C the vertex with the largest Sy_k.

Step 3. Divide the triangle vertically into two sides. Make Side 0 the edge from vertex A to vertex C, and side 1 the edge from vertex A to vertex B to vertex C (see Figure 4.4).

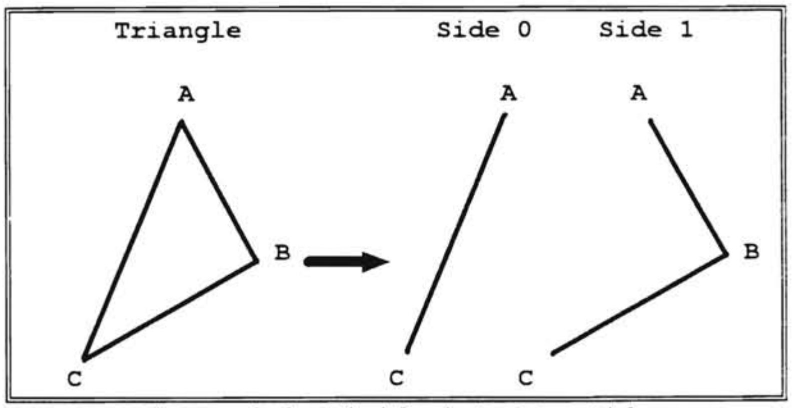


Figure 4.4 Divide into Two Sides

Step 4. Find the screen values for each of the two sides. This is done by linearly interpolating between the projected vertices of the triangle in order to find the three edges of the triangle on the screen. In other words, interpolate between vertex A and vertex C's screen values to find side 0's screen values, and interpolate between vertex A and vertex B's screen values as well as between vertex B and vertex C's screen values to find side 1's screen values (see procedure *find_side()* in appendix A).

Step 5. Scan the screen values between each side, finding their corresponding texture values. This involves holding each side's similar screen y values constant, and incrementing through the screen x values between them. Note that this step is considered the scan-line step, since it is done for each screen y value of the projected triangle in scan-line order. The texture values are found by using the equation in section 2.2.2.

Step five has a nested loop which scans a total of N screen pixels which make up the projected triangle. Because of this, step five has the most influence over the time complexity of the algorithm, O(N). A detailed algorithm for performing fast inverse perspective mapping in scan-line order is given in appendix A.

4.5 Shading

Many shading methods require that a surface normal be known for particular points on a surface. Since shading is a form of texture mapping, it can usually be done during the texture mapping process. Inverse perspective mapping algorithms, however, are not easily tailored to find surface normals. These algorithms generally linearly interpolate between a set of given surface normals to find each particular surface normal. In other words, an affine mapping is used to map the normals. The argument is that shading does not have to be as visually exact as basic texture mapping. Although this is true in many instances, it is desirable for some applications to have objects whose surface normal values do not change while the object moves across the computer screen.

CHAPTER 5

THE SCAN-LINE SUBDIVISION ALGORITHM

The scan-line subdivision algorithm combines the qualities of Catmull's subdivision method with the advantages of the general purpose inverse perspective mapping algorithm. Subdivision is a good, simple method for finding values between points, but it does not have the scan-line qualities that algorithms may need for less expensive hardware implementations. Subdivision can also be very calculation intensive if the algorithm has computational redundancy or complex subdivision calculations. Inverse perspective mapping, on the other hand, allows for a scan-line approach to perspective texture mapping, but is not very suitable for finding surface normals for use in shading. Also with inverse perspective mapping, if the surface shape to be texture mapped does not easily conform to rectangular texture space, aligning the two might not be very straightforward. Where one algorithm has disadvantages, the other has advantages. The scan-line subdivision approach explained in this chapter attempts to take advantage of this fact by combining strengths from both methods.

5.1 Scan-line Subdivision

Scan-line subdivision is very different from Catmull's subdivision method. In fact, it more closely matches the generic inverse perspective mapping algorithm covered in chapter four. Like the inverse perspective mapping algorithm, it works best with planar objects such as triangles. In the author's subjective opinion, there are two major reasons for not directly using curved surfaces to model objects. One is that using these surfaces to model objects can make an algorithm very complex, and another is that many algorithms which use curved surfaces to model objects resort to planar approximations at a certain point in the process anyway. (Still, there are many applications for which using curved surfaces to model free-form objects is preferred. The author leaves altering the scan-line subdivision approach to be used with curved surfaces for later study.) The scan-line subdivision algorithm described in this chapter subdivides a triangle first in the y direction, and then in the x direction. The advantage to subdividing this way is that it avoids computations by needing only to compute the perspective transformation of the object's y components once for each "scan-line" being bisected in the x direction. This somewhat compares with the scan-line order imposed by the inverse perspective mapping algorithm. However, since the subdivision is not completely done in scan-line order, inverse perspective mapping maintains an advantage in this area.

5.2 The Four Basic Steps

A scan-line subdivision algorithm for perspective texture mapping the basic triangle can be divided into these four basic steps:

Step 1. Sort the three triangle vertices, (x_i, y_i, z_i), and their corresponding texture coordinates, (u_i, v_i), by their projected screen y values, Sy_i, where i=0..2. Designate vertex A the vertex with the smallest Sy_k, vertex B the vertex with the next smallest Sy_k, and vertex C the vertex with the largest Sy_k.

Step 2. Divide the triangle vertically into two sides. Make Side 0 the edge from vertex A to vertex C, and side 1 the edge from vertex A to vertex B to vertex C (see figure 4.4). Make each side an array of *n* vertices, where n corresponds to the maximum amount of "vertical" subdivisions that may be done (see section 5.3).

Step 3. Vertically subdivide each "side" of the triangle. This requires bisecting the edge from vertex A to vertex C for side 0, and bisecting the edges from vertex A to vertex B to vertex C for side 1. This is accomplished by adding their components and then dividing the result by two. For example, bisecting the x component between two points, A and B, would give C, the center point's x component where

$$C(x) = (A(x) + B(x)) / 2.0.$$

Any criteria may be used to stop the subdivision. When the subdivision is complete, store the resulting vertices in the "side" arrays, side(0,Sy_i) and side(1,Sy_i), where $0<=i<=n$.

Step 4. Horizontally subdivide between side(0,Sy_i) and side(1,Sy_i), where i=0..n. This step is considered the "scan-line subdivision" step, particularly when Sy_i corresponds to each scan-line y value. Again, any criteria may be used to stop the bisection. When the criteria is met, use the subdivided texture coordinates which are associated with each of the subdivided vertices to color the screen pixels at (Sx_i,Sy_i).

5.3 Termination

The termination conditions used to stop the subdivision in steps three and four largely depend on the filtering process used. If the termination condition is set to stop the subdivision when the projected end-points are only one pixel apart, then the algorithm would closely match the generic inverse perspective mapping algorithm in chapter four.

This condition leaves the rest of the texture mapping up to a space-variant filtering process which would then use the resulting texture coordinates as center points from which to work. (For simplicity, the algorithm covered in this chapter uses this condition to halt the subdivision.)

A form of dynamic filtering can be performed by allowing further subdivisions based on certain termination conditions. For instance, by basing the termination condition on a distance between each endpoint's projected screen coordinate, a space-variant filtering can be performed. This distance value can be varied to balance between speed and realism. A small distance value results in greater realism because more texture values are used to create the final image. A large distance value, on the other hand, results in a faster algorithm because not as many subdivisions would be performed. One type of crude space-variant filtering that can be used in this instance is a simple weighted average of the texture values which make up each pixel.

5.4 Similar Algorithms

As one may have noticed, the steps outlined above match closely with the steps given in chapter four which outline the basic inverse perspective mapping algorithm (steps one through four correspond with steps two through five). In fact, even though this algorithm is a perspective mapping algorithm as opposed to an inverse perspective mapping algorithm, it is closely based on the generic inverse perspective mapping algorithm structure. The main differences are in the final two steps. The scan-line subdivision algorithm uses subdivision to find the texture values between the line segments rather than parameterization.

5.5 Scan-line Subdivision vs. Inverse Perspective Mapping

The scan-line subdivision algorithm has three main advantages over the inverse perspective mapping algorithm. The first is that the criteria for stopping the subdivision can easily be altered to allow for a form of dynamic filtering (see section 5.3). If speed is not critical, more subdivisions may be done, resulting in more texture values being used in creating the final image. This generally results in better image quality.

Another advantage to the scan-line subdivision algorithm is that code for keeping track of each point's surface normal may be easily added at almost no extra cost, resulting in more accurate object shading. This is done by associating each vertex of the triangle with a surface normal, and allowing the algorithm to subdivide the surface normal vector along with the vertex values.

The third advantage involves ease of parallelization. Generally, recursive algorithms are more straightforward when it comes to converting them for use with multiple processors. The bulk of the scan-line subdivision algorithm uses preorder recursion to bisect between two points.

The disadvantages of the scan-line subdivision algorithm over the inverse perspective algorithm include the fact that the algorithm is not a complete scan-line algorithm. That is, it is set to texture map top to bottom, but not necessarily left to right. Another disadvantage is that the scan-line subdivision algorithm has stack overhead, and although both algorithms have the same O(n) time-complexity, it is slightly more calculation intensive. The bulk of the divisions used in the scan-line subdivision algorithm are divisions by two, which is often faster than general-case floating point division. These divisions may also be converted to a multiplication by a constant for those computers which multiply floating point numbers faster than dividing them.

5.6 Scan-line Subdivision vs. Catmull Subdivision

Scan-line subdivision has many advantages over Catmull subdivision. One is that the algorithm is much less complex. This is largely because the scan-line subdivision algorithm subdivides simple planar objects rather than curved surfaces, but also because it subdivides in a scan-line fashion. If not done properly, subdivision of a surface into four sub-surfaces can result in computational redundancy resulting from common points which are found by subdividing separate, but adjacent surfaces. The scan-line subdivision algorithm avoids this by bisecting the surface in the y direction, and then in the x direction. Computation is reduced because planar objects are simpler to bisect and do not "fold over" on themselves.

Unlike the scan-line subdivision algorithm, however, Catmull's algorithm works for curved surfaces. This is an advantage for applications which need more accurate approximations of free-form objects. However, even Catmull's algorithm determines which screen pixels are covered by a curved surface patch by using the quadrilateral formed from the patch's corner points to approximate it.

CHAPTER 6

CONCLUSION

Perspective texture mapping algorithms mostly differ in the type of mapping they incorporate between texture and object space. The two primary mappings that have been studied are the parametric mapping and mapping via subdivision. A popular parametric mapping approach is that of inverse perspective mapping in scan-line order (chapter 4). Until this study, the only method for texture mapping via subdivision is the one introduced by Ed Catmull in 1974 (chapter 3).

One of the advantages of subdivision over other approaches is that it is very dynamic. The goal of subdivision is to divide problems into smaller problems, until each problem is easier to solve. With texture mapping, the problem is that of balancing between speed and realism. For example, if an object to be texture mapped is in motion, the speed at which it is texture mapped may be more critical than how realistic it appears. Subdivision can provide a dynamic balance between speed and realism based on the amount of subdivision performed (chapter 5).

Although the parametric approach is more static, it can be used to texture map very efficiently and does not need a stack. One of the most efficient texture mapping methods based on the parametric approach is the inverse perspective mapping method discussed in chapter four. Unlike Catmull subdivision, this method texture maps in scan-line order, which is desirable for most inexpensive hardware implementations. The method is very simple, mainly because it only texture maps planar objects as opposed to the more complex curved surfaces that Catmull's algorithm maps. One of the problems with this method, however, is that it is not easily adapted to find surface normals for use in shading.

With subdivision, surface normals are easily computed along with other points.

The *scan-line subdivision* method introduced in this paper combines the advantages of both inverse perspective mapping and perspective mapping via subdivision (see Figures in Appendix C). It is closely based on the scan-line structure of the inverse perspective mapping algorithm, but maintains the desirable properties associated with subdivision. With this method, surface normals are easily computed for use in shading, and a form of dynamic filtering is possible based on the number of subdivisions performed. Although the algorithm is not quite as efficient as the inverse perspective mapping algorithm, it has the advantages of subdivision that make it preferable.

6.1 Future Work

One obvious extension to the scan-line subdivision approach is to apply the method to curved surfaces. As is, the algorithm works only with objects that are based on the triangle. To apply the approach to curved surfaces, a method of separating the surface's screen image into "sides" is needed. This task may be challenging because the curve property that makes these surfaces desirable also makes working with them difficult. For instance, the screen image of a projected curved surface patch has edges that do not always correspond to the actual edges of the patch. Multiple sides would need to be found, as opposed to just two, and these sides may not correspond to the actual edges of the surface patch.

BIBLIOGRAPHY

[AOKI78] Aoki, M., and M. Levine, "Computer Generation of Realistic Pictures," *Computers and Graphics,* Vol. 3, 1978, 149-161.

[BEAT82] Beatty, J.C., and K.S. Booth, eds., *Tutorial: Computer Graphics,* Second Edition, IEEE Comp. Soc. Press, Silver Spring, MD, 1982.

[BIER86] Bier, E., and K. Sloan, "Two-part texture mapping." *IEEE Computer Graphics and applications,* September 1986, 40-53.

[BLIN76] Blinn, J.F., and M.E. Newell, "Texture and Reflection in Computer Generated Images," *CACM,* 19(10), October 1976, 542-547. Also in BEAT82, 456-461.

[BLIN78a] Blinn, J.F., "Simulation of Wrinkled Surfaces," *SIGGRAPH 78,* 286-292.

[BLIN78b] Blinn, J.F., *Computer Display of Curved Surfaces,* Ph.D. Thesis, Department of Computer Science, University of Utah, Salt Lake City, UT, December 1978.

[BUIT75] Bui-Tuong, Phong, "Illumination for Computer Generated Pictures," CACM, 18(6), June 1975, 311-317. Also in BEAT82, 449-455.

[CATM74] Catmull, E., *A Subdivision Algorithm for Computer Display of Curved Surfaces,* Ph.D. Thesis, Report UTEC-CSc-74-133, Computer Science Department, University of Utah, Salt Lake City, UT, December 1974.

[CATM80] Catmull, E., and A.R. Smith, "3-D Transformations of Images in Scanline Order," *SIGGRAPH 80,* 279-285.

[COOK86] Cook, Robert, "Antialiasing by Stochastic Sampling," *ACM Trans. Graphics,* 1986.

[CROW77] Crow, Franklin, "The Aliasing Problem in Computer Generated Shaded Images," *Comm. ACM,* Vol. 20, Nov. 1977, 799-805.

[FEIB80] Feibush, Eliot, M. Levoy, and R. Cook, "Synthetic Texturing Using Digital Filters," *Computer Graphics* (Proc. SIGGRAPH 80), Vol. 14, No. 3, July 1980, 294-301.

[FOLE90] Foley, J., A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics: Principles and Practice, Second Edition,* Addison-Wesley, Reading, MA, 1990.

[FREE80] Freeman, H. ed., *Tutorial and Selected Readings in Interactive Computer Graphics,* IEEE Comp. Soc. Press, Silver Spring, MD, 1980.

[GANG82] Gangnet, M., D. Perny, and P. Coueignoux, "Perspective Mapping of Planar Textures," *Eurographics 82,* 57-71.

[GREE86] Greene, Ned, and P. Heckbert, "Creating Raster Omnimax Images from Multiple Perspective Views Using the Elliptical Weighted Average Filter," *IEEE CG&A,* Vol. 6, No. 6, June 1986, pp.21-27.

[GOUR71] Gouraud, H., "Continuous Shading of Curved Surfaces," *IEEE Trans. On Computers,* C-20(6), June 1971, 623-629. Also in FREE80, 302-308.

[HECK83] Heckbert, P.S., "Texture Mapping Polygons in Perspective," Tech. Memo No. 13, NYIT Computer Graphics Lab, Apr. 1983.

[HECK86] Heckbert, P.S., "Survey of Texture Mapping," CG & A, 6(11), November 1986, 56-67.

[HECK89] Heckbert, P.S., *Fundamentals of Texture Mapping and Image Warping,* Master's thesis, UCB/CSD 89/516, CS Dept, UC Berkeley, May 1989.

[HECK91] Heckbert, P.S., and H. Moreton, "Interpolation for Polygon Texture Mapping and Shading," *State of the Art Computer Graphics: Visualization and Modeling,* Rogers, D., and R. Earnshaw, eds., Springer-Verlag, New York, 1991, 101-111.

[MAXW46] Maxwell, E.A., *The Methods of Plane Projective Geometry, Based on the Use of General Homogeneous Coordinates*, Cambridge U. Press, London, 1946.

[NEWL73] Newell, M., R. Newell, and T. Sancha, "A New Approach to the Shaded Picture Problem," Proceedings of the ACM, 1973 National Conference.

[OPPE75] Oppenheim, Alan, and R. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, N.J., 1975.

[SMIT80] Smith, A.R., "Incremental Rendering of Textures in Perspective," SIGGRAPH 80: *Animation Graphics Seminar Notes*, July 1980.

[SMIT83] Smith, A.R., "Digital Filtering Tutorial for Computer Graphics," parts 1 and 2, SIGGRAPH 83: *Introduction to Computer Animation Seminar Notes*, July 1983, 244-261, 262-272.

[WHIT81] Whitted, Turner, "The Causes of Aliasing in Computer Generated Images," SIGGRAPH 81: *Advanced Image Synthesis Seminar Notes*, Aug. 1981.

APPENDIX A

INVERSE PERSPECTIVE MAPPING IN SCAN-LINE ORDER

The following pseudo-Pascal represents the basic code for a general purpose inverse perspective mapping algorithm which texture maps in scan-line order. These functions and procedures are outlined in chapter 4.

{ Type *vertex* is a structure containing both an (x,y,z) and a (u,v) coordinate. }

Procedure Name: texture_map

Comment: Texture maps in scan-line order using an inverse perspective mapping.

Input: An array of type *vertex* with three elements containing the triangle's three vertices.

Output: A texture mapped triangle (on the screen).

Globals: *screen_height* is the height of the screen in pixels.

```
procedure texture_map(vertex(3)  :vertex)
var a, b, c, d, e, f, g, h, i       :real;
var w, u, v                         :real;
var sy(3)                           :array of real;
var t, cur_sx, cur_sy               :integer;
var svert(3), side(2, screen_height):array of real;
var side0, side1                    :pointer to real;
```

```
begin

      // Step 1.  Setup the coefficients..

      setup_coeffs(vertex(), a, b, c, d, e, f, g, h, i);


      // Step 2:  Sort the three triangle's vertices by

      //          their projected y value.

      for t := 0 to 2 do

            sy(t) := project_y(vertex(t));

      sort(vertex(), sy(), svert());


      // Steps 3 and 4 - divide triangle vertically into two

      //          sides and find the screen values for each

      //          of the two sides.

      find_side(side(0,), svert(0), svert(2), sy(0), sy(2));

      find_side(side(1,), svert(0), svert(1), sy(0), sy(1));

      find_side(side(1,), svert(1), svert(2), sy(1), sy(2));


      // Step 5 - Scan the screen values between each side, while

      //          finding and plotting the texture values.


      // First make sure we are going to scan in scan-line order

      if (side(0, sy(1)) > side(1, sy(1))) then begin

            side0 := side(1,);  side1 := side(0,);

      else begin

            side0 := side(0,);  side1 := side(1,);

      end;
```

```
    for cur_sy := sy(0) to sy(2) do
        for cur_sx := side0->cur_sy to side1->cur_sy do
        begin
            w := g * cur_sx + h * cur_sy + i;
            u := (a * cur_sx + b * cur_sy + c) / w;
            v := (d * cur_sx + e * cur_sy + f) / w;
            display_point(u, v, cur_sx, cur_sy);
        end;
end.
```

```
Procedure Name:   setup_coeffs
Comment:      Sets up coefficients for use in the parametric mapping
              of the texture plane as transformed to fit a 2D triangle
              in 3D object space ( see section 2.2.2 ).
Input:        An array of three elements which contain the three triangle
              vertices.
Output:       The nine coefficients a, b, .., i.
Globals:      None.
Procedure setup_coeffs(vertex() :array of vertex, var a, b, c, d, e, f,
                       g, h, i  :real)
var ax, ay, az, bx, by, bz          :real;
var cu, cv, du, dv                  :real;
var j, k, l, m, n, o, p, q, r       :real;
var x(3), y(3), z(3)                :array of real;
begin
    // First we must find the coefficients for the parametric
    // equation of the (u, v) texture plane as if it where
    // transformed to fit the triangle in object space.
// We find these using what we know about the (u,v) and
// (x,y,z) coordinates of the triangle.  This section can
// be eliminated if three vertices are added to the triangle
// structure which get updated along side the triangle, however,
// this means transforming six vertices instead of just three
// whenever the triangle is transformed..
ax := vertex(1).x - vertex(0).x;
ay := vertex(1).y - vertex(0).y;
az := vertex(1).z - vertex(0).z;
bx := vertex(2).x - vertex(0).x;
by := vertex(2).y - vertex(0).y;
bz := vertex(2).z - vertex(0).z;
cu := vertex(1).u - vertex(0).u;
cv := vertex(1).v - vertex(0).v;
du := vertex(2).u - vertex(0).u;
dv := vertex(2).v - vertex(0).v;

j := bx * cv / (du * cv - cu * dv);
k := ( ax - j * cu) / cv;
l := vertex(0).x - j * vertex(0).u - k * vertex(0).v;
m := by * cv / (du * cv - cu * dv);
n := ( ay - m * cu) / cv;
o := vertex(0).y - m * vertex(0).u - n * vertex(0).v;
p := bz * cv / (du * cv - cu * dv);
q := ( az - p * cu) / cv;
r := vertex(0).z - p * vertex(0).u - q * vertex(0).v;

x(0) := l;  y(0) := o;  z(0) := r;
x(1) := j + l;  y(1) := m + o;  z(1) := p + r;
x(2) := k + l;  y(2) := v + o;  z(2) := q + r;
```

```
        // Second, we find the coefficients for the inverse
        // mapping as shown in section 2.2.2.
        a := y(2) * z(0) - y(0) * z(2);
        b := x(0) * z(2) - x(2) * z(0);
        c := x(2) * y(0) - x(0) * y(2);
        d := y(0) * z(1) - y(1) * z(0);
        e := x(1) * z(0) - x(0) * z(1);
        f := x(0) * y(1) - x(1) * y(0);
        g := a + d + y(1) * z(2) - y(2) * z(1);
        h := b + e + x(2) * z(1) - x(1) * z(2);
        i := c + f + x(1) * y(2) - x(2) * y(1);
end;
```

Function Name: project_x

Comment: Maps an x-value from 3D object space to 2D screen space.

Input: A vertex structure.

Output: Projected screen x value.

Globals: *Eye_distance_from_screen* is defined from the desired view angle.

```
Function project_x(V    :vertex)    :real;

begin
        project_x := Eye_distance_from_screen * V.x / V.z;
end;
```

Function Name: project_y

Comment: Maps from 3D object space to 2D screen space.

Input: A vertex structure.

```
Output:      Projected screen y value.

Globals:     Eye_distance_from_screen is defined from the
             desired view angle.

Function project_y(V    :vertex)    :real;
begin
      project_y := Eye_distance_from_screen * V.y/V.z;
end;


Procedure Name:   sort

Comment:     Sorts vertices and y values using brute force, which trades
             efficient space usage for a better execution time.
             Each vertex is sorted by its corresponding y value.

Input:       An array of three vertex structures, vertex(0..2), and an
             array of three corresponding y values, sy(0..2).

Output:      A sorted array, svert(0..2), of the three vertex structures,
             vertex(0..2), which correspond to a sorted array of the
             three y values, sy(0..2).

Procedure sort(vertex() :array of vertex, var sy() :array of real,
             var svert() :array of vertex)
var   temp  :real;
begin
      if (sy(0) <= sy(1)) and (sy(0) <= sy(2)) then begin
             svert(0) := vertex(0);
             if (sy(1) <= sy(2)) then begin      // Sorted order: 0, 1, 2
                   svert(1) := vertex(1);
                   svert(2) := vertex(2);
             else begin                          // Sorted order: 0, 2, 1
                   svert(1) := vertex(2);
```

```
            svert(2) := vertex(1);
            temp := sy(1);
            sy(1) := sy(2);
            sy(2) := temp;
        end;
else if (sy(1) <= sy(0)) and (sy(1) <= sy(2)) then begin
        svert(0) := vertex(1);
        temp := sy(1);
        if (sy(0) <= sy(2)) then begin        // Sorted order: 1, 0, 2
            svert(1) := vertex(0);
            svert(2) := vertex(2);
            sy(1) := sy(0);
        else begin                             // Sorted order: 1, 2, 0
            svert(1) := vertex(2);
            svert(2) := vertex(0);
            sy(1) := sy(2);
            sy(2) := sy(0);
        end;
        sy(0) := temp;
else if (sy(2) <= sy(0)) and (sy(2) <= sy(1)) then begin
        svert(0) := vertex(2);
        temp := sy(2);
        if (sy(0) <= sy(1)) then begin        // Sorted order: 2, 0, 1
            svert(1) := vertex(0);
            svert(2) := vertex(1);
            sy(1) := sy(0);
            sy(2) := sy(1);
```

```
          else begin                        // Sorted order: 2, 1, 0
                svert(1) := vertex(1);
                svert(2) := vertex(0);
                sy(2) := sy(0);
          end;
          sy(0) := temp;
      end;
end;
```

```
Procedure Name:   find_side
Comment:     Given two screen coordinates, find the screen coordinates
             that lie on the line which runs between the two points.
             Store the x values in an array which is indexed by the y
             values.
Input:       Two vertices, their corresponding y values, and an array of
             type real which is at least as large as the screen's height.
Output:      The array is passed back with its updated values.
Procedure find_side(var side() :array of real, A, B :vertex, sy1, sy2
                    :real)
var   sx1, sx2, inv_slope, sy        :real;
begin
      sx1 := project_x(A);
      sx2 := project_x(B);
      inv_slope := (sx2 - sx1) / (sy2 - sy1);
      for sy := sy1 to sy2
      begin
            if ((sy >= screen.min_y) and (sy <= screen.max_y)) then
                  side(sy) := sx1 + (sy - sy1) * inv_slope;
```

```
          end;

      end;
```

APPENDIX B

THE SCAN-LINE SUBDIVISION ALGORITHM

The following pseudo-Pascal represents the basic code of the scan-line subdivision algorithm. These functions and procedures are outlined in chapter 5.

{ Type *vertex* is a structure containing both an (x,y,z) and a (u,v) coordinate. }

```
Procedure Name:   scan_line_subdivide
Input:        An array of type vertex with three elements containing the
              triangle's three vertices.
Output:       A texture mapped triangle (on the computer screen).
Globals:      screen_height is the height of the screen in pixels.
Procedure scan_line_subdivide(vertex(3) :vertex)
var svert(3)                    :array of vertex;
var side(2, screen_height)      :array of vertex;
var sy(3)                       :array of real;
var i, cur_sy                   :integer;
begin
      for i := 0 to 2 do        // Step 1 - Sort by projected y values
            sy(i) := project_y(vertex(i));
      sort(vertex(), sy(), svert());
                                // Steps 2 and 3 - (sub)divide vertically
```

```
    bisect_side(side(0,), svert(0), svert(2), sy(0), sy(2));
    bisect_side(side(1,), svert(0), svert(1), sy(0), sy(1));
    bisect_side(side(1,), svert(1), svert(2), sy(1), sy(2));
                                    // Step 4 - subdivide horizontally
    for cur_sy := sy(0) to sy(2) do
    begin
        bisect_inner(side(0, cur_sy), side(1, cur_sy));
    end;
end.
```

Function Name: project_x

Comment: See appendix A.

Function Name: project_y

Comment: See appendix A.

Procedure Name: sort

Comment: See appendix A.

Procedure Name(s): bisect_side and do_bisect_side

Comment: Subdivides or *bisects* one side of a triangle, storing the resulting coordinates in an array. *bisect_side* sets up the bisection, while *do_bisect_side* actually implements the bisection.

Input: Two vertices which represent the ends of a line segment, which denotes one side of a triangle, are passed. The corresponding projected screen y values of the two vertices are also passed, as well as an array of type *vertex* which is capable of holding at least *n* elements. In this example, *n* is the height of the screen in pixels. Procedure do_bisect_side has an extra parameter called *depth* which keeps track of the recursion depth.

Output: The passed array is updated with the new coordinates resulting from the bisection.

Globals: *screen.min_y* and *screen.max_y* denote the minimum and maximum y values that the screen can plot. *max_depth* represents the maximum recursion depth allowed.

```
Procedure bisect_side(var side() :array of vertex, A, B :vertex,
                        sy1, sy2  :real)
begin
    If ((sy2 >= screen.min_y) and (sy2 <= screen.max_y)) then
        side(sy2) := B;                 // Store the "odd" vertex
    do_bisect_side(side(),A,B,1,sy1,sy2);     // Begin the bisection
end;


Procedure do_bisect_side(var side() :array of vertex, A, B :vertex,
                           depth  :integer, sy1, sy2  :real)
var   sy, disty    :real;       // disty = distance between sy1 and sy2
var   C            :vertex;     // C = temporary vertex storage
begin
    // Do necessary clipping
    if (((sy1 > screen.max_y) and (sy2 > screen.max_y)) or
        ((sy1 < screen.min_y) and (sy2 < screen.min_y)) or
            (depth > max_depth)) then
```

```
begin
      depth := depth - 1;
      return;
end;
disty := sy2 - sy1;      // Find the distance between sy1 and sy2
// Check to see if "stop recursion" criteria has been met.
// In this case, check if sy1 and sy2 are one pixel apart.
if ((disty >= -1.0) and (disty <= 1.0)) then
begin
      if ((sy1 >= screen.min_y) and (sy1 <= screen.max_y)) then
            side(sy1) := A;          // Store A in array..
      depth := depth - 1;
      return;
end;
// Do actual bisection
C.x := ( A.x + B.x ) / 2.0;
C.y := ( A.y + B.y ) / 2.0;
C.z := ( A.z + B.z ) / 2.0;
C.u := ( A.u + B.u ) / 2.0;
C.v := ( A.v + B.v ) / 2.0;

sy := project_y( C );    // Find screen y value for the new point

do_bisect_side(side(),A,C,depth+1,sy1,sy); // Preorder recursion
do_bisect_side(side(),C,B,depth+1,sy,sy2);

depth := depth - 1;
```

end;

Procedure Name(s): bisect_inner and do_bisect_inner

Comment: Subdivides or *bisects* between two vertices of type *vertex*. These two vertices are expected to lie on the same y plane, and the subdivision is expected to stop when the projected x screen values of each subdivided point are only one pixel apart. The resulting (u, v) coordinates are used to find the color for the resulting screen coordinates.

Input: Two vertices which represent the ends of a line segment are passed. Procedure do_bisect_side has a parameter called *depth* which keeps track of the recursion depth, as well as two other parameters which correspond to each passed vertex's projected screen x value.

Output: A texture mapped scan-line (on the screen).

Globals: *screen.min_x* and *screen.max_x* denote the minimum and maximum x values that the screen can plot. *cur_sy* is the current screen y value for the scan-line. *max_depth* represents the maximum recursion depth allowed.

```
Procedure bisect_inner(A, B :vertex)
var   sx1, sx2    :real;        // Temporary screen x value storage
begin
      sx1 := project_x( A );
      sx2 := project_x( B );
      if ((sx2 >= screen.min_x) and (sx2 <= screen.max_x)) then
            display_point(B, sx2, cur_sy);
      do_bisect_inner(A,B,1,sx1,sx2);      // Begin the bisection
```

```
end;


Procedure do_bisect_inner(A, B :vertex, depth :integer, sx1, sx2 :real

var   C            :vertex;

var   sx, distx    :real;

begin

      // do necessary clipping

      if ((sx1 > screen.max_x) and (sx2 > screen.max_x)) or

            ((sx1 < screen.min_x) and (sx2 < screen.min_x)) or

                  (depth > max_depth)) then

      begin

            depth := depth - 1;

            return;

      end;

      distx := sx2 - sx1;

      if ((distx >= -1.0) and (distx <= 1.0)) then

      begin

            if ((sx1 >= screen.min_x) and (sx1 <= screen.max_x)) then

                  display_point(A, sx1, cur_sy);

            depth := depth - 1;

            return;

      end;

      // Do actual bisection..

      C.x := ( A.x + B.x ) / 2.0;

      // C.y does not need to be bisected.. (not used)

      C.z := ( A.z + B.z ) / 2.0;

      C.u := ( A.u + B.u ) / 2.0;
```

```
        C.v := ( A.v + B.v ) / 2.0;

        sx := project_x( C );

        do_bisect_inner(A, C, depth+1, sx1, sx);
        do_bisect_inner(C, B, depth+1, sx, sx2);

        depth := depth - 1;
end;
```

APPENDIX C

PICTURES

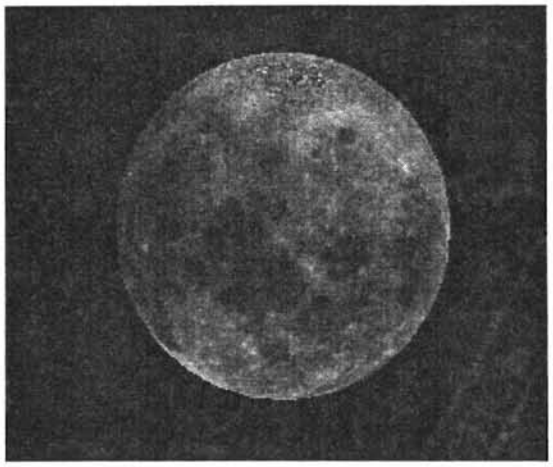


Figure C.1

Triangle meshed sphere textured with a map of the moon.

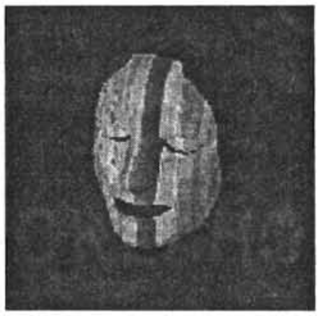


Figure C.2

Texture map of a wooden mask.

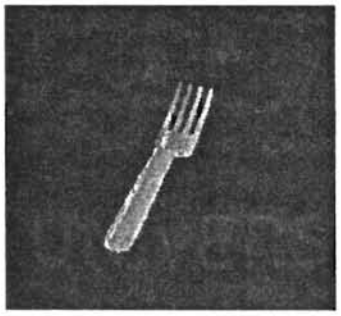


Figure C.3

Texture map of a wooden fork.

2

VITA

David Sterling Sanders

Candidate for the Degree of

Master of Science

Thesis: A SCAN-LINE SUBDIVISION APPROACH TO PERSPECTIVE TEXTURE MAPPING

Major Field: Computer Science

Biographical:

Education: Graduated from Broken Arrow High School, Broken Arrow, Oklahoma, in May, 1990; received Bachelor of Science degree in Computing and Information Science with a minor in Mathematics from Oklahoma State University, Stillwater, Oklahoma in May, 1994. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December, 1996.

Experience: Employed at Creative Labs, Inc. as a technical support agent, Lead Agent, and Agent in Charge of the Legacy Sound group, 1993 to 1996.