Fundamentals of Texture Mapping and Image Warping

Master's Thesis
under the direction of Carlo Séquin

Paul S. Heckbert

Dept. of Electrical Engineering and Computer Science
University of California, Berkeley, CA 94720

June 17, 1989

This Postscript version is missing about 40 paste-up figures. To get a complete version, order report no. UCB/CSD 89/516 from the Computer Science Division at the address above.

Abstract

The applications of texture mapping in computer graphics and image distortion (warping) in image processing share a core of fundamental techniques. We explore two of these techniques, the two-dimensional geometric mappings that arise in the parameterization and projection of textures onto surfaces, and the filters necessary to eliminate aliasing when an image is resampled during texture mapping or warping. With respect to mappings, this work presents a tutorial on three common classes of mapping: the affine, bilinear, and projective. For resampling, this work develops a new theory describing the ideal, space variant antialiasing filter for signals warped and resampled according to an arbitrary mapping. Efficient implementations of the mapping and filtering techniques are discussed and demonstrated.

Contents

Acknowledgements

Thanks to Charlie Gunn for clarifying projective geometry, Rick Sayre for his careful reading, and Jules Bloomenthal for detailed critiques between runs on Granite Chief. My readers, Professors Carlo Séquin and Brian Barsky, and Ed Catmull, provided helpful comments. Carlo Séquin and a grant from MICRO generously supported this research. Reference recommendations by Professor Avideh Zakhor and Ken Turkowski, formatting help from Ziv Gigus, and photographic assistance from Greg Ward were also invaluable. The guys of 508-7, John Hartman, Mike Hohmeyer, and Ken Shirriff, created the crucial camaraderie.

This paper is based on research begun at the New York Institute of Technology Computer Graphics Lab, where Lance Williams suggested numerous texture applications, Ned Greene and Dick Lundin demanded faster Omnimax resampling programs, and Ephraim Cohen showed me the 8×8 system for projective mapping inference. Kevin Hunter suggested the decomposition into two square-to-quadrilateral mappings. Discussions of "resize bugs" with Pat Hanrahan and Tom Porter at Pixar underscored the importance of both reconstruction and prefiltering during resampling.

If only there were more time, we could do it all.

Chapter 1

Introduction

In this thesis, we are interested in the modeling and rendering problems common to texture mapping and image warping. We begin by reviewing the goals and applications of texture mapping and image warping.

1.1 Texture Mapping

Texture mapping is a shading technique for image synthesis in which a texture image is mapped onto a surface in a three dimensional scene, much as wallpaper is applied to a wall [Catmull74], [Blinn-Newell76], [Heckbert86b]. If we were modeling a table, for example, we might use a rectangular box for the table top, and four cylinders for the legs. Unadorned, this table model would look quite dull when rendered. The realism of the rendered image can be enhanced immensely by mapping a wood grain pattern onto the table top, using the values in the texture to define the color at each point of the surface. The advantage of texture mapping is that it adds much detail to a scene while requiring only a modest increase in rendering time. Texture mapping does not affect hidden surface elimination, but merely adds a small incremental cost to the shading process. The technique generalizes easily to curved surfaces [Catmull74].

Texture mapping can be used to define many surface parameters besides color. These include the perturbation of surface normal vectors to simulate bumpy surfaces (*bump mapping*), *transparency mapping* to modulate the opacity of a translucent surface, *specularity mapping* to vary the glossiness of a surface, and *illumination mapping* to model the distribution of incoming light in all directions. These applications are surveyed, and their original literature is cited, in [Carey-Greenberg85].

In all of the varieties of texture mapping mentioned above, geometric mappings are fundamental. Two-dimensional mappings are used to define the parameterization of a surface and to describe the transformation between the texture coordinate system and the screen coordinate system. In texture mapping applications the latter mapping is usually fully determined by the 3-D transformation defined by the camera, the modeling transformations describing the geometry of the scene, and the parameterization that maps a texture onto a surface. There are many texture representations, but in this work we restrict ourselves to the most common: discrete 2-D images.

When rendering a textured surface it is necessary to sample the texture image to produce the screen image. Except for scenes viewed with parallel projections, texture mappings are non-affine, so they have nonuniform resampling grids, as shown in figure 1.1 (affine mappings are those composed of rotations, scales, and translations). Antialiasing non-affine texture mappings requires a space variant texture filter, however. Such filters change shape as a function of position, and are much more difficult to implement than space invariant filters. Space variant filters are particularly important for texture mapping applications because of the extreme range of scales involved in many 3-D scenes. (These issues are explored in detail in chapter 3.)

1.2 Image Warping

Image warping is the act of distorting a source image into a destination image according to a mapping between source space (u, v) and destination space (x, y). The mapping is usually specified by the functions $x(u, v)$ and $y(u, v)$. (We use the term "warp" instead of its synonyms "distortion" and "transformation" because of its conciseness and specificity: "warp" specifically suggests a mapping of the domain of an image, while "transformation" can mean a mapping of the image range as well).

Image warping is used in image processing primarily for the correction of geometric distortions introduced by imperfect imaging systems [Gonzalez-Wintz87]. Camera lenses sometimes introduce pincushion or barrel distortions, perspective views introduce a projective distortion, and other nonlinear optical components can create more complex distortions. In image processing, we do image warping typically to remove the distortions from an image, while in computer graphics we are usually introducing one. Image warps are also used for artistic purposes and special effects in interactive paint programs. For image processing applications, the mapping may be derived given a model of the geometric distortions of a system, but more typically the mapping is inferred from a set of corresponding points in the source and destination images. The point correspondence can be automatic, as for stereo matching, or manual, as in paint programs. Most geometric correction systems support a limited set of mapping types, such as piecewise affine, bilinear, biquadratic, or bicubic mappings. Such mappings are usually parameterized by a grid of control points. A survey of mapping and filtering techniques for image processing is found in [Wolberg88].

The appropriate antialiasing filter for non-affine mappings in image warping is, in general, space variant, just as for texture mapping. Filter quality and efficiency is less of a problem for image processing, however, since the mappings used there usually have a more uniform sampling grid than those for texture mapping. When it is known a priori that the sampling grid is nearly uniform, a fixed "interpolation" filter can be used with good results [Gonzalez-Wintz87]. Uniform resampling (often known as "sampling rate conversion") is a fairly common task for 1-D signals such as audio [Crochiere-Rabiner83]. Nonuniform sampling has received much less attention to date.

Figure 1.1: *a) A textured surface viewed in perspective. b) Nonuniform grid of texture samples in the screen.*

1.3 Organization of this Thesis

This thesis is split into two halves: chapter 2 contains a discussion of basic 2-D mappings, and chapter 3 develops and applies a theory for resampling filters. There are also two appendices: the first contains source code for several mapping tasks, and the second summarizes the mathematics of ellipses relevant to filtering.

Chapter 2

Two-Dimensional Mappings

In our discussion of mappings we restrict ourselves, for the most part, to 2-D images mapped to planar surfaces. The simplest classes of 2-D mappings are the affine, bilinear, and projective transformations. These mappings are well known in mathematics and have been used for years in computer graphics. Despite their fundamental importance, however, the latter two mappings have received little attention in the computer graphics literature. Other aspects of texture mapping and image warping, such as texture synthesis and filtering, have received much more attention, probably because they have more apparent impact on image quality than simple 2-D mappings.

We believe that texture mapping would be more common in rendering systems if basic 2-D mappings were more widely understood. In this work we hope to revive appreciation for one class of mappings in particular, the projective mapping.

2.1 Naive Texture Mapping

2.1.1 Coordinate Systems

To discuss texture mapping, we need to define several coordinate systems. *Texture space* is the 2-D space of surface textures and *object space* is the 3-D coordinate system in which 3-D geometry such as polygons and patches are defined. Typically, a polygon is defined by listing the object space coordinates of each of its vertices. For the classic form of texture mapping with which we are concerned, texture coordinates (u, v) are assigned to each vertex. *World space* is a global coordinate system that is related to each object's local object space using 3-D modeling transformations (translations, rotations, and scales). *3-D screen space* is the 3-D coordinate system of the display, a perspective space with pixel coordinates (x, y) and depth z (used for z-buffering). It is related to world space by the camera parameters (position, orientation, and field of view). Finally, *2-D screen space* is the 2-D subset of 3-D screen space without z. When we use the phrase "screen space" by itself we mean 2-D screen space.

The correspondence between 2-D texture space and 3-D object space is called the *parameterization* of the surface, and the mapping from 3-D object space to 2-D screen space is the *projection* defined by the camera and the modeling transformations (figure 2.1). Note that when we are rendering a particular view of a textured surface, it is the *compound mapping* from 2-D texture space

Figure 2.1: *The compound mapping is the composition of the surface parameterization and the viewing projection.*

to 2-D screen space that is of interest. For resampling purposes, once the 2-D to 2-D compound mapping is known, the intermediate 3-D space can be ignored. The compound mapping in texture mapping is an example of an *image warp*, the resampling of a source image to produce a destination image according to a 2-D geometric mapping.

2.1.2 A Naive Implementation of Texture Mapping

To demonstrate the subtleties of 2-D mappings, we outline here an algorithm for texture mapping within a z-buffer polygon renderer [Rogers 85]. This simple algorithm will produce a number of visual flaws that are quite instructive.

We begin by defining the object space coordinates (x, y, z) and texture coordinates (u, v) at each vertex of each polygon. The polygons are transformed to screen space, yielding x, y, and z coordinates for each vertex. We will need to compute z, u, and v at each pixel, and we would like to do this with fast, incremental techniques. It is well known that the perspective transformation maps lines to lines and planes to planes [Foley-van Dam 82], so linear interpolation is appropriate for computing z. Linear interpolation is also used in Gouraud and Phong shading, so it might seem reasonable to interpolate the texture coordinates (u, v) linearly as well. Following this reasoning, we scan convert the polygon into pixels by linearly interpolating x, z, u, and v along the left and right sides of the polygon and linearly interpolating z, u, and v across each scan line. Having texture coordinates (u, v) at each pixel, we can sample the texture array and use the resulting color as the screen pixel value.

The inner loop of this primitive texture mapper will then be:

```
for x = xleft to xright
    if z < ZBUF[x,y] then           is new point closer?
        ZBUF[x,y] = z               update z-buffer
        (r, g, b) = TEX[u,v]        sample texture
        SCR[x,y] = (r, g, b)        write screen
    z = z+dz
    (u, v) = (u, v) + (du, dv)
```

where (a, b, c) denotes a vector. The increment values dz, du, and dv are calculated once per scan line. This code uses the texture as the output pixel color (i.e. no lighting effects) and it point samples the texture (i.e. no filtering).

Close examination of figure 2.2 reveals some of the flaws in the algorithm above. Aliasing (not visible here) would result from a high frequency texture. This can be eliminated by filtering an area surrounding the texture point (u, v) rather than sampling just a single pixel [Feibush-Levoy-Cook80], [Heckbert86b]. More fundamentally, however, these images do not exhibit the foreshortening we expect from perspective (compare with figure 2.3). The textured polygon also shows disturbing discontinuities along horizontal lines passing through each vertex. These discontinuities are artifacts of the linear interpolation of u and v. In animation, the horizontal ripples will move distractingly as the camera rolls, since the ripples are rotation variant, and the lack of foreshortening will make the texture appear to swim across a surface. As we will see, *this naive texture mapper is correct only for affine mappings.*

These problems occur because the texture transformation effected by our linear interpolation of u and v is inconsistent with the geometry transformation used to transform the vertices to screen space. Linear interpolation of (u, v) in a scan line algorithm computes a bilinear mapping, while the actual image warp defined by perspective is a projective mapping. One ad hoc solution to this inconsistency is to continue with linear interpolation, but to finely subdivide the polygon. If the texture coordinates are correctly computed for the vertices of the new polygons, the resulting picture will exhibit less rippling. It is hard to know how finely to subdivide the polygon, however. We can find more satisfactory, theoretically correct, solutions to these problems by studying simple, generic 2-D mappings. Later we will apply the generic techniques to texture mapping.

2.2 Simple Two-Dimensional Mappings

There is a limitless variety of possible 2-D mappings. Mappings for applications of texture mapping or image warping must satisfy the needs of both the user and the implementer of the system. For a designer modeling a 3-D scene or the user of image distortion software, one of the most important criteria for selecting a mapping is predictability. One form of predictability present in the simplest mappings is the preservation of straight lines. If a texture mapping or image warp bends lines then the results are less predictable than those of a line-preserving mapping. Other desirable properties are the preservation of equispaced points, angles [Fiume-Fournier-Canale87], and areas. The implementer of such a system seeks efficient, incremental computations and well-behaved functions (single-valued and invertible). We will see how well the simplest 2-D mappings meet these criteria.

Figure 2.2: *Left: A checkerboard texture. Right: Image produced by naive texture mapper using linear interpolation of u, v. Note horizontal lines of discontinuity passing through vertices (indicated with dashed lines).*

Figure 2.3: *Image produced by projective warp algorithm of §2.3.5 using rational linear interpolation of texture coordinates. Note proper foreshortening.*

A *two-dimensional mapping* is a mapping (transformation) that distorts a 2-D source space into a 2-D destination space. It maps a source point (u, v) to a destination point (x, y), according to the functions $x(u, v)$ and $y(u, v)$. We will discuss three classes of mappings: affine, bilinear, and projective.

Homogeneous Notation

The homogeneous representation for points provides a consistent notation for affine and projective mappings. Homogeneous notation was used in projective geometry [Maxwell46], [Coxeter78] long before its introduction to computer graphics [Roberts66]. The homogeneous notation is often misunderstood so we will take a moment to clarify its use and properties.

In familiar Euclidean geometry we represent points of the real plane $\mathcal{R}^2$ by vectors of the form (x, y). Projective geometry deals with the projective plane, a superset of the real plane, whose *homogeneous coordinates* are (x', y', w). In projective geometry the 2-D real point (x, y) is represented by the homogeneous vector $\mathbf{p} = (x', y', w) = (xw, yw, w)$, where w is an arbitrary nonzero number. Vectors of the form (xw, yw, w) for $w \neq 0$ form the equivalence class of homogeneous representations for the real point (x, y). To recover the actual coordinates from a homogeneous vector, we simply divide by the homogeneous component; e.g., the homogeneous vector $\mathbf{p} = (xw, yw, w) = (x', y', w)$ represents the actual point $(x, y) = (x'/w, y'/w)$. This division, a projection onto the $w = 1$ plane, cancels the effect of scalar multiplication by w. When representing real points with homogeneous notation we could use any nonzero w, but it is usually most convenient to choose $w = 1$ so that the real coordinates can be recovered without division. Projective space also includes the *points at infinity*: vectors of the form $(x', y', 0)$, excluding $(0, 0, 0)$. The points at infinity lie on the *line at infinity*. We will see later how augmentation of the real plane by these points simplifies projective geometry.

In homogeneous notation, 2-D points are represented by 3-vectors and 3-D points are represented by 4-vectors. For affine and projective mappings, we denote points in source space by $\mathbf{p_s} = (u', v', q)$ and points in the destination space by $\mathbf{p_d} = (x', y', w)$.

2.2.1 Affine Mappings

Affine mappings include scales, rotations, translations, and shears; they are linear mappings plus a translation [Foley-van Dam82]. Formally, a mapping $T(x)$ is *linear* iff $T(x + y) = T(x) + T(y)$ and $T(\alpha x) = \alpha T(x)$ for any scalar α, and a mapping $T(x)$ is *affine* iff there exists a constant c and a linear mapping $L(x)$ such that $T(x) = L(x) + c$ for all x. Obviously, linear mappings are a subset of affine mappings.

A general 2-D affine mapping may be written algebraically as:

$$\mathbf{p_d} = \mathbf{p_s}\mathbf{M_{sd}}$$

$$\begin{pmatrix} x & y & 1 \end{pmatrix} = \begin{pmatrix} u & v & 1 \end{pmatrix} \begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}$$

Transform matrices are denoted $\mathbf{M_{ab}}$ where a is the initial coordinate system and b is the final coordinate system. Notice that we have chosen $w = q = 1$ without loss of generality. The 3×3

Figure 2.4: *Affine warps of image at left.*

matrix $\mathbf{M_{sd}}$ has 6 degrees of freedom. Geometrically, the vectors (a, d) and (b, e) are basis vectors of the destination space, and (c, f) is the origin.

As shown in figure 2.4, affine mappings preserve parallel lines and preserve equispaced points along lines (meaning that equispaced points along a line in the source space transform to equispaced points along a line in the destination space, although the spacing in the two coordinate systems may be different). We can invert an affine mapping to find the destination-to-source transformation simply by inverting the mapping matrix (iff $\mathbf{M_{sd}}$ has an inverse). Similarly, affine mappings may be composed by concatenating their matrices.

Since an affine mapping has 6 degrees of freedom, it may be defined geometrically by specifying the source and destination coordinates of three points. The mapping and its inverse will be nondegenerate if the three points are noncollinear in both spaces. To infer the affine transform matrix from a three point correspondence that maps (u_k, v_k) to (x_k, y_k) for $k = 0, 1, 2$, we solve the following matrix equation for $\mathbf{M_{sd}}$:

$$\mathbf{M_d} = \mathbf{M_s}\mathbf{M_{sd}}$$

$$= \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{pmatrix} = \begin{pmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{pmatrix} \begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}$$

The above relation is 6 scalar equations in 6 unknowns a-f, but the system simplifies easily into two 3×3 systems, one for x and one for y.

Affine mappings can map any triangle in source space into any triangle in destination space, or map a source rectangle into a destination parallelogram, but no more general distortions are possible. To warp a rectangle into a general quadrilateral, we will need a bilinear, projective, or some other more complex mapping.

2.2.2 Bilinear Mappings

Bilinear mappings are most commonly defined as a mapping of a square into a quadrilateral. As shown in figure 2.5, this mapping can be computed by linearly interpolating by fraction u along the top and bottom edges of the quadrilateral, and then linearly interpolating by fraction v between the two interpolated points to yield destination point (x, y) [Faux-Pratt79]:

$$(x, y) = (1-u)(1-v)\mathbf{p_{00}} + u(1-v)\mathbf{p_{10}} + (1-u)v\mathbf{p_{01}} + uv\mathbf{p_{11}}$$

Figure 2.5: *Bilinear mapping.*

The general form in matrix notation is:

$$\begin{pmatrix} x & y \end{pmatrix} = \begin{pmatrix} uv & u & v & 1 \end{pmatrix} \begin{pmatrix} a & e \\ b & f \\ c & g \\ d & h \end{pmatrix}$$

A bilinear mapping is affine if $a = e = 0$. The matrix of 8 coefficients may be computed from the four point correspondence of figure 2.5 as follows:

$$\begin{pmatrix} a & e \\ b & f \\ c & g \\ d & h \end{pmatrix} = \begin{pmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_{00} & y_{00} \\ x_{10} & y_{10} \\ x_{01} & y_{01} \\ x_{11} & y_{11} \end{pmatrix}$$

This mapping transforms a unit square into the general quadrilateral $\mathbf{p_{00}}, \mathbf{p_{10}}, \mathbf{p_{11}}, \mathbf{p_{01}}$, which an affine mapping cannot do.

The bilinear mapping has an unusual mix of properties. Because of its linear interpolation, the forward transform from source to destination space preserves lines which are horizontal or vertical in the source space, and preserves equispaced points along such lines, *but it does not preserve diagonal lines*, as shown in figure 2.6. Also, the composition of two bilinear mappings is not bilinear, but biquadratic.

The inverse mapping from destination space to source space is not a bilinear mapping either; in fact it is not single-valued. The inverse can be derived by solving for v in the x equation:

$$v = \frac{x - bu - d}{au + c}$$

and substituting into the y equation. The same can be done for u, yielding the two quadratic equations:

$$(au + c)(fu + h - y) - (eu + g)(bu + d - x) = 0$$
$$(av + b)(gv + h - y) - (ev + f)(cv + d - x) = 0$$

so

$$Au^2 + Bu + C = 0$$
$$Dv^2 + Ev + F = 0$$

Figure 2.6: *Bilinear warps of a test grid. a) Grid in source space. b) Warped grid in destination space for quadrilateral 1 (vertices marked with dots). Note curvature of diagonals. c) Warped grid in destination space for quadrilateral 2. Diagonals map to parabolas in destination space. Note foldover of some diagonals (caused by the existence of two roots in quadratic equation).*

where

$$A = af - be \quad B = ex - ay + ah - de + cf - bg \quad C = gx - cy + ch - dg$$
$$D = ag - ce \quad E = ex - ay + ah - de - cf + bg \quad F = fx - by + bh - df$$

We can find (u, v) in terms of (x, y) by evaluating the coefficients A, B, and C above and then computing:

$$u = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}, \quad v = \frac{x - bu - d}{au + c}$$

The inverse transform is multi-valued, and is much more difficult to compute than the forward transform.

2.2.3 Projective Mappings

The *projective mapping*, also known as the perspective or homogeneous transformation, is a projection from one plane through a point onto another plane [Maxwell46]. Homogeneous transformations are used extensively for 3-D affine modeling transformations and for perspective camera transformations [Foley-van Dam82]. The 2-D projective mappings studied here are a subset of these familiar 3-D homogeneous transformations.

The general form of a projective mapping is a rational linear mapping:

$$x = \frac{au + bv + c}{gu + hv + i}, \quad y = \frac{du + ev + f}{gu + hv + i} \tag{2.1}$$

Manipulation of projective mappings is much easier in the homogeneous matrix notation:

$$\mathbf{p_d} = \mathbf{p_s M_{sd}}$$

$$= \begin{pmatrix} x' & y' & w \end{pmatrix} = \begin{pmatrix} u' & v' & q \end{pmatrix} \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

where $(x, y) = (x'/w, y'/w)$ for $w \neq 0$, and $(u, v) = (u'/q, v'/q)$ for $q \neq 0$.

Although there are 9 coefficients in the matrix above, these mappings are homogeneous, so any nonzero scalar multiple of these matrices gives an equivalent mapping. Hence there are only 8 degrees of freedom in a 2-D projective mapping. We can assume without loss of generality that $i = 1$ except in the special case that source point $(0, 0)$ maps to a point at infinity. A projective mapping is affine when $g = h = 0$.

Projective mappings will in general map the line at infinity to a line in the real plane. We can think of this line as the horizon line of all vanishing points, by analogy to the perspective projection. Affine mappings are the special case of projective mappings that map the line at infinity into itself. By defining the projective mapping over the projective plane and not just the real plane, projective mappings become bijections (one-to-one and onto), except when the mapping matrix is singular. For nondegenerate mappings the forward and inverse transforms are single-valued, just as for an affine mapping.

Projective mappings share many of the desirable properties of affine mappings. Unlike bilinear mappings, which preserve equispaced points along certain lines, the projective mappings do not in general preserve equispaced points (figure 2.7). Instead they preserve a quantity called the "cross ratio" of points [Coxeter78]. Like affine mappings, *projective mappings preserve lines at all*

Figure 2.7: *Projective warps of a test grid controlled by the same quadrilaterals as figure 2.6. a) Warped grid in destination space for quadrilateral 1. b) Warped grid in destination space for quadrilateral 2. Note the vanishing points.*

orientations. In fact, projective mappings are the most general line-preserving bijective mappings. Projective mappings may be composed by concatenating their matrices.

Another remarkable property is that *the inverse of a projective mapping is a projective mapping.* This is intuitively explained by reversing the plane-to-plane mapping by which a projective mapping is defined. The matrix for the inverse mapping is the inverse or adjoint of the forward mapping. (The adjoint of a matrix is the transpose of the matrix of cofactors [Strang80]; $\mathbf{M}^{-1} = adj(\mathbf{M})/det(\mathbf{M})$). In homogeneous algebra, the adjoint matrix can be used in place of the inverse matrix whenever an inverse transform is needed, since the two are scalar multiples of each other, and the adjoint always exists, while the inverse does not if the matrix is singular. The inverse transformation is thus:

$$\begin{aligned} \mathbf{p_s} &= \mathbf{p_d M_{ds}} \\ = \begin{pmatrix} u' & v' & q \end{pmatrix} &= \begin{pmatrix} x' & y' & w \end{pmatrix} \begin{pmatrix} A & D & G \\ B & E & H \\ C & F & I \end{pmatrix} \\ &= \begin{pmatrix} x' & y' & w \end{pmatrix} \begin{pmatrix} ei-fh & fg-di & dh-eg \\ ch-bi & ai-cg & bg-ah \\ bf-ce & cd-af & ae-bd \end{pmatrix} \end{aligned}$$

When mapping a point by the inverse transform we compute (u, v) from (x, y). If $w \neq 0$ and $q \neq 0$ then we can choose $w = 1$ and calculate:

$$u = \frac{Ax + By + C}{Gx + Hy + I}, \quad v = \frac{Dx + Ey + F}{Gx + Hy + I} \tag{2.2}$$

Inferring Projective Mappings

In an interactive image warper one might specify the four corners of source and destination quadrilaterals with a tablet or mouse, and wish to warp one area to the other. This sort of task is an ideal application of projective mappings, but how do we find the mapping matrix?

A projective mapping has 8 degrees of freedom which can be determined from the source and destination coordinates of the four corners of a quadrilateral. Let the correspondence map (u_k, v_k) to (x_k, y_k) for vertices numbered cyclically $k = 0, 1, 2, 3$. All coordinates are assumed to be real (finite). To compute the forward mapping matrix $\mathbf{M_{sd}}$, assuming that $i = 1$, we have eight equations in the eight unknowns a-g:

$$x_k = \frac{au_k + bv_k + c}{gu_k + hv_k + 1} \quad \Rightarrow \quad u_k a + v_k b + c - u_k x_k g - v_k x_k h = x_k$$

$$y_k = \frac{du_k + ev_k + f}{gu_k + hv_k + 1} \quad \Rightarrow \quad u_k d + v_k e + f - u_k y_k g - v_k y_k h = y_k$$

for $k = 0, 1, 2, 3$. This can be rewritten as an 8×8 system:

$$\begin{pmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

This linear system can be solved using Gaussian elimination for the forward mapping coefficients a-h. If the inverse mapping is desired instead, then either we compute the adjoint of $\mathbf{M_{sd}}$ or we follow the same procedure, starting from equation (2.2) instead of (2.1), and solve an 8×8 system for coefficients A-H.

In speed-critical special cases, there are more efficient formulas for computing the mapping matrix. The formula above handles the case where the polygon is a general quadrilateral in both source and destination spaces. We will consider three additional cases: square-to-quadrilateral, quadrilateral-to-square, and (again) the general quadrilateral-to-quadrilateral mapping.

Case 1. The system is easily solved symbolically in the special case where the uv quadrilateral is a unit square. If the vertex correspondence is as follows:

x	y	u	v
x_0	y_0	0	0
x_1	y_1	1	0
x_2	y_2	1	1
x_3	y_3	0	1

then the eight equations reduce to

$$
\begin{aligned}
c &= x_0 \\
a + c - gx_1 &= x_1 \\
a + b + c - gx_2 - hx_2 &= x_2 \\
b + c - hx_3 &= x_3 \\
f &= y_0 \\
d + f - gy_1 &= y_1 \\
d + e + f - gy_2 - hy_2 &= y_2 \\
e + f - hy_3 &= y_3
\end{aligned}
$$

If we define

$$
\begin{array}{lll}
\Delta x_1 = x_1 - x_2 & \Delta x_2 = x_3 - x_2 & \Sigma x = x_0 - x_1 + x_2 - x_3 \\
\Delta y_1 = y_1 - y_2 & \Delta y_2 = y_3 - y_2 & \Sigma y = y_0 - y_1 + y_2 - y_3
\end{array}
$$

then the solution splits into two sub-cases:

(a) $\Sigma x = 0$ and $\Sigma y = 0$. This implies that the xy polygon is a parallelogram, so the mapping is affine, and $a = x_1 - x_0$, $b = x_2 - x_1$, $c = x_0$, $d = y_1 - y_0$, $e = y_2 - y_1$, $f = y_0$, $g = 0$, $h = 0$.

(b) $\Sigma x \neq 0$ or $\Sigma y \neq 0$ gives a projective mapping:

$$
\begin{aligned}
g &= \begin{vmatrix} \Sigma x & \Delta x_2 \\ \Sigma y & \Delta y_2 \end{vmatrix} \Big/ \begin{vmatrix} \Delta x_1 & \Delta x_2 \\ \Delta y_1 & \Delta y_2 \end{vmatrix} \\
h &= \begin{vmatrix} \Delta x_1 & \Sigma x \\ \Delta y_1 & \Sigma y \end{vmatrix} \Big/ \begin{vmatrix} \Delta x_1 & \Delta x_2 \\ \Delta y_1 & \Delta y_2 \end{vmatrix} \\
a &= x_1 - x_0 + gx_1 \\
b &= x_3 - x_0 + hx_3 \\
c &= x_0 \\
d &= y_1 - y_0 + gy_1 \\
e &= y_3 - y_0 + hy_3 \\
f &= y_0
\end{aligned}
$$

This computation is much faster than a straightforward 8×8 system solver. The mapping above is easily generalized to map a rectangle to a quadrilateral by pre-multiplying with a scale and translation matrix.

Case 2. The inverse mapping, a quadrilateral to a square, can also be optimized. It turns out that the most efficient algorithm for computing this is not purely symbolic, as in the previous case, but numerical. We use the square-to-quadrilateral formulas just described to find the inverse of the desired mapping, and then take its adjoint to compute the quadrilateral-to-square mapping.

Case 3. Since we can compute quadrilateral-to-square and square-to-quadrilateral mappings quickly, the two mappings can easily be composed to yield a general quadrilateral-to-quadrilateral mapping (figure 2.8). This solution method is faster than a general 8×8 system solver.

Source code to infer a projective mapping from a quadrilateral correspondence is contained in §A.2.

Figure 2.8: *Quadrilateral to quadrilateral mapping as a composition of simpler mappings.*

2.2.4 Comparison of Simple Mappings

As mentioned previously, affine and projective mappings are closed under composition, but bilinear mappings are not. The three basic mapping classes compose as follows:

$\Downarrow$ MAP1 $\circ$ MAP2 $\Rightarrow$	affine	bilinear	projective
affine	affine	bilinear	projective
bilinear	bilinear	biquadratic	rational bilinear
projective	projective	rational biquadratic	projective

Thus, the composition of two bilinear mappings is a biquadratic mapping.

Since (nonsingular) affine and projective mappings are closed under composition, have an identity and inverses, and obey the associative law, they each form a group under the operation of composition. Bilinear mappings do not form a group in this sense.

We summarize the properties of affine, bilinear, and projective mappings below:

PROPERTY	AFFINE	BILINEAR	PROJECTIVE
preserves parallel lines	yes	no	no
preserves lines	yes	no†	yes
preserves equispaced points	yes	no†	no
maps square to	parallelogram	quadrilateral	quadrilateral
degrees of freedom	6	8	8
closed under composition	yes	no, biquadratic	yes
closed under inversion	yes	no, solve quadratic	yes
single-valued inverse	yes	no	yes
forms a group	yes	no	yes
incremental forward mapping	2 adds	2 adds	2 divs, 3 adds‡
incremental inverse mapping	2 adds	1 square root, more	2 divs, 3 adds

†except for horizontal and vertical lines in source space
‡see §2.3.5

For the designer, affine mappings are the simplest of the three classes. If more generality is needed, then projective mappings are preferable to bilinear mappings because of the predictability of line-preserving mappings. For the implementer, the group properties of affine and projective mappings make their inverse mappings as easy to compute as their forward mappings. Bilinear mappings are computationally preferable to projective mappings only when the forward mapping is used much more heavily than the inverse mapping.

2.3 Two-Dimensional Mappings in Texture Mapping

As mentioned earlier, texture mapping consists of a transformation of 2-D texture space to a 3-D surface via the parameterization, and then a projection of that surface into 2-D screen space. Two-dimensional mappings are central to each of these transformations. Parameterization is a 2-D mapping of a texture onto a surface in 3-D, and rendering uses a 2-D mapping to control the resampling of the texture to produce a screen image. We can thus make use of the 2-D mappings discussed above in both modeling and rendering textured surfaces.

2.3.1 Parameterizing Planar Patches

The 2-D mappings described earlier can be used to parameterize polygons and other planar surfaces. A surface *parameterization* describes surface points in 3-D object space in terms of two parameters u and v. We call a parameterized plane a *planar patch*.

Planes are most easily parameterized affinely, by selecting two linearly independent basis vectors in the plane and a point in the plane. These three 3-D points define the 9 degrees of freedom of an affine parameterization or *affine patch*. An affine patch is given by:

$$\begin{pmatrix} x & y & z \end{pmatrix} = \begin{pmatrix} u & v & 1 \end{pmatrix} \begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \end{pmatrix}$$

Just as with purely 2-D mappings, affine patches allow arbitrary triangle-to-triangle and rectangle-to-parallelogram mappings, but not rectangle-to-arbitrary-quadrilateral mappings. Source code to infer an affine mapping from the texture and object space coordinates of a polygon is given in §A.3.

Occasionally one may need more generality than is provided by affine mappings, for example, to distort a rectangular brick texture into a trapezoidal keystone. For this we need bilinear or projective patches (the latter also known as rational linear patches). The general form of a bilinear patch is:

$$\begin{pmatrix} x & y & z \end{pmatrix} = \begin{pmatrix} uv & u & v & 1 \end{pmatrix} \begin{pmatrix} a & e & i \\ b & f & j \\ c & g & k \\ d & h & l \end{pmatrix}$$

and projective patches have the form:

$$\begin{pmatrix} x' & y' & z' & w \end{pmatrix} = \begin{pmatrix} u' & v' & q \end{pmatrix} \begin{pmatrix} a & d & g & j \\ b & e & h & k \\ c & f & i & l \end{pmatrix}$$

(The coefficients $a - l$ are unrelated between these three forms, of course).

Like affine patches, projective patches are always planar. Bilinear patches can be nonplanar, however. The general bilinear patch has 12 degrees of freedom, while planar bilinear patches and projective patches have 11 degrees of freedom. The coefficients of a bilinear or projective patch can be inferred from a four point correspondence between texture space and object space.

Bilinear patches are sometimes used in texture mapping [Hourcade-Nicolas83], [Catmull-Smith80] when nonplanar quadrilaterals are needed. Projective patches are usually preferable for planar quadrilaterals, however, because of their superior properties. Occasionally, practitioners have used bilinear patches when projective patches would have been more appropriate. For example, bilinear parameterizations were proposed for point-in-quadrilateral testing in a ray tracing machine [Ullner83]. In that algorithm, a ray is intersected with the plane of a polygon to determine the intersection point in object space. Next, point inclusion is determined by inverting the bilinear mapping, transforming object space coordinates to texture space, and testing if (u, v) lies within a unit square. The hardware designed was quite complex, as it involved square roots. If projective patches had been used instead then the inverse mapping could have been much simpler. (There are faster methods for point-in-quadrilateral testing than transforming to a unit square, of course.)

2.3.2 Deducing Projective Mappings

If a chain of transformations from texture space to screen space is known, then we can synthesize the compound mapping matrix; there is no need to infer it from vertex correspondences. The compound mapping is most easily found for projective mappings.

We use the following notation:

COORDINATE SYSTEM	REAL	HOMOGENEOUS
2-D texture space	(u, v)	$\mathbf{p_t} = (u', v', q)$
3-D object space	(x_o, y_o, z_o)	$\mathbf{p_o} = (x_o w_o, y_o w_o, z_o w_o, w_o)$
3-D screen space	(x, y, z)	$\mathbf{p}_\sigma = (x', y', z', w)$
2-D screen space	(x, y)	$\mathbf{p_s} = (x', y', w)$

To synthesize the texture-to-screen transform, we concatenate the 3×4 projective parameterization matrix $\mathbf{M_{to}}$ that maps texture space to object space, the 4×4 modeling and (perspective or parallel) viewing matrix $\mathbf{M_{o\sigma}}$ that maps object space to 3-D screen space, and a 4×3 matrix $\mathbf{M_{\sigma s}}$ that discards z to arrive at 2-D screen space:

$$\mathbf{M_{ts}} = \mathbf{M_{to}} \mathbf{M_{o\sigma}} \mathbf{M_{\sigma s}}$$
$$= \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The resulting matrix $\mathbf{M_{ts}}$ is the 3×3 matrix for the projective compound mapping.

2.3.3 Other Parameterizations

There are many other surface parameterization schemes besides those described here; we have covered only the simplest planar parameterizations. For curved surfaces, parameterization ease varies with modeling technique. Parameterizations are inherent when a surface is modeled parametrically [Faux-Pratt79], but can be difficult to construct for implicit surfaces. One method for fabricating a parameterization uses the coordinates of 3-D object space as texture parameters rather than attempt to construct surface-following parameters. These coordinates become arguments to a procedural or tabular solid texture [Perlin85], [Peachey85], and result in objects appearing carved from a solid, rather than covered by a pattern. Two-dimensional textures can also be projected onto surfaces as decals [Bier-Sloan86].

2.3.4 Scanning Textures and Image Warps

To render a texture mapped surface we must compose the parameterization and projection mappings. As shown in the mapping composition table earlier, some combinations of parameterization and projection compose elegantly; others do not. For example, an affine- or projective-parameterized plane viewed with a perspective projection yields a projective compound mapping, but a bilinear-parameterized plane viewed in perspective gives a more complex compound transformation, a rational bilinear mapping. This suggests that perspective projections of textured planes are equally easy to render when an affine or projective parameterization is used, and difficult to render for a bilinear parameterization.

There are three general approaches to drawing a texture mapped surface or performing an image warp: scanning in screen space, scanning in texture space, and scanning in multiple passes. The three algorithms are outlined below:

```
SCREEN ORDER:
    for y
        for x
            compute u(x,y) and v(x,y)
            SCR[x,y] = TEX[u,v]
```

```
TEXTURE ORDER:
    for v
        for u
            compute x(u,v) and y(u,v)
            SCR[x,y] = TEX[u,v]

MULTI-PASS:
    for v
        for u
            compute x(u,v)
            TEMP[x,v] = TEX[u,v]
    for x
        for v
            compute y(x,v)
            SCR[x,y] = TEMP[x,v]
```

where TEX is the texture array, SCR is the screen array, and TEMP is an intermediate array. The multi-pass outline given above is two-pass, but there are variations using three or more passes. Note that resampling should involve filtering, not point sampling, an issue we ignore here.

Screen order, sometimes called inverse mapping, is the most common method. Each pixel in screen space is inverse-transformed to texture space and the texture pixel there is read. When using screen order we need to invert the mapping, which is easy for projective mappings but difficult for bilinear mappings. Screen order scanning is preferred when the screen must be written sequentially, the mapping is readily invertible, and the texture is random access.

The second scanning method is texture order. Scanning in texture order may seem simpler than screen order, since it does not require the inverse mapping, but preventing holes or overlaps in screen space is tricky [Shantz-Lien87]. One solution is to add mapped samples into a screen space accumulator buffer with a filter function. Texture order is preferable only when the texture-to-screen mapping is difficult to invert, or when the texture image must be read sequentially and will not fit in random access memory.

Multi-pass methods decompose a 2-D mapping into two or more 1-D mappings. In the two-pass form, the first pass processes the rows of an image and the second pass processes the columns. Multi-pass methods have been used for both texture mapping [Catmull-Smith80], [Smith87] and for image processing [Wolberg88], to perform geometric correction of images [Fraser-Schowengerdt-Briggs85] or affine image warps [Paeth86]. These methods work particularly well for affine and projective mappings, where the 1-D resampling warps for each pass are affine and projective (rational linear), respectively. Bilinear mappings are possible as well, but they require rational quadratic resampling warps [Catmull-Smith80]. Multi-pass methods are the preferred scanning order when the texture cannot be accessed randomly but it has rapid row and column access, and a buffer for the intermediate image is available.

2.3.5 Incremental Techniques for Screen Order Scanning

Affine image warps can be computed by linearly interpolating u and v along the sides of a polygon and across each scan line, as implemented in the naive texture mapper presented earlier. Such warps are appropriate for affinely parameterized polygons viewed with parallel projection.

It is now clear how to perform correct texture mapping for projective mappings as well. As discussed earlier, polygons that are viewed in perspective and parameterized either affinely or with a projective mapping will have a projective compound mapping. The algorithm for scanning projective warps is nearly the same as that for affine warps, the only difference being that we cannot linearly interpolate u and v. For projective mappings, u and v are rational linear, not linear, functions of the screen coordinates. So we linearly interpolate the quantities uq, vq, and q, which are linear in x and y, and perform two divisions per pixel:

```
for x = xleft to xright
    if z < ZBUF[x,y] then                is new point closer?
        ZBUF[x,y] = z                    update z-buffer
        (r, g, b) = TEX[uq/q,vq/q]       sample texture
        SCR[x,y] = (r, g, b)             write screen
    z = z+dz
    (uq, vq, q) = (uq, vq, q) + (duq, dvq, dq)
```

Divisions are necessary at each pixel for projective mappings because u and v are rational linear functions of x. (Of course, if $g = h = 0$ then $q = 1$ and the extra work is unnecessary, but this occurs exactly when the projective mapping happens to be affine.) The additional cost of two divisions per pixel is modest even when an inexpensive texture filter (such as point sampling) is used, and is negligible when a more expensive filter is used. When performing projective mappings for image distortion applications rather than texture mapping we would use the same algorithm but without the z-buffer.

Rational linear interpolation of u and v in this way eliminates the ripple, rotation variance, and foreshortening problems mentioned earlier (figure 2.3). This solution to those problems is preferable to polygon subdivision because it totally eliminates the artifacts, at a minor increase in complexity. (Note that Gouraud and Phong shading in perspective, computed using linear interpolation, also give rotation variant artifacts, albeit much less visible than those of texture coordinate interpolation. Rotation variance of these smooth shading techniques can also be fixed using rational linear interpolation.)

2.4 Example: Photomosaics by Image Warping

We now present an application of image warps. Figure 2.9 shows two photographs taken from the same viewpoint but with different camera rotations. The fields of view overlap, so an image mosaic can be created by texture mapping each photograph onto an imaginary plane perpendicular to its line of sight and rendering the 3-D scene with a perspective view from the original viewpoint (figure 2.10). (The photographs are from Florence, site of the renaissance of perspective.) The mapping was found by measuring the coordinates of four pairs of corresponding points in the region of overlap, and using projective mapping inference techniques described in §2.2.3 and §A.2.

Assuming no lens aberration or vignetting, the two images will merge without discontinuity, and we can simulate the photograph that would have resulted from any combination of camera zoom and orientation. In fact, if a panorama of photographs were taken, views in any direction could be synthesized [Greene86].

2.5 Summary

Two-dimensional mappings are a fundamental operation in texture mapping and image warping. In texture mapping, the modeling task of parameterization and the rendering task of scan conversion both use two-dimensional mappings. Image processing uses two-dimensional mappings for the geometric correction of distorted images.

We have discussed the most basic mappings: affine, bilinear, and projective. Affine mappings are very simple and efficient, and may be constructed from any three-point correspondence. The two generalizations of affine mappings, bilinear and projective, can be constructed from any four-point correspondence. When generality beyond an affine mapping is needed, the projective mapping is preferable in most respects. Geometrically, the projective mapping is line preserving; computationally, it is closed under composition and inversion, so it adapts easily to any scanning order. To perform projective texture mapping or image warping, linear interpolation of texture coordinates is inappropriate. Instead we interpolate homogeneous texture coordinates and perform a division at each pixel. The additional cost is often negligible relative to affine mapping.

Figure 2.9: *Images for photomosaic test. Photographs from the interior of the Pitti Palace, Florence. The fields of view overlap, but the perspectives are different, since the camera was tilted up for the top image.*

Figure 2.10: *Photomosaic by image warping. Synthesized image in which figure 2.9 (top) was warped to register with figure 2.9 (bottom), and the two were overlayed.*

Chapter 3

Resampling Filters

3.1 Is Filtering Needed?

Earlier, in our naive implementation of texture mapping, we used point sampling to read data from the texture image: the center of the screen pixel was mapped to texture space and the value at the nearest texture pixel was read. This technique is simple, fast, and works well for some images (figure 3.1). If we point sample images containing high frequencies, however, objectionable moire patterns can result (figure 3.2). This artifact is called *texture aliasing*. The effect is very distracting in animation, where the moire pattern can boil and sparkle from frame to frame. Similar artifacts occur in image warping applications. Such artifacts are intolerable in realistic still images or animation, so we will endeavor to eradicate them.

Aliasing has been studied extensively in the field of digital signal processing. The reader is referred to the elementary [Jackson86], [Hamming83] and advanced [Oppenheim-Schafer75], [Brigham74], [Pratt78], [Dudgeon-Mersereau84] texts for derivations of the theory we summarize in this chapter.

Most of the theoretical research on resampling to date has been driven by audio applications, and is consequently limited to 1-D uniform resampling (i.e. affine mappings) [Crochiere-Rabiner83]. Signal processing theory has been applied by computer graphicists to problems of edge antialiasing [Crow77] with good results. Existing theory is lacking for problems of resampling non-affine mappings, however [Smith83], so most of the filters developed for texture mapping [Catmull74], [Blinn-Newell76], [Feibush-Levoy-Cook80], [Williams83], [Crow84], [Heckbert86b] have been based on filter design heuristics and subjective quality comparisons.

First we will review the causes of aliasing during resampling. We then develop a theory of ideal resampling filters that eliminate aliasing during resampling. We close this chapter with a demonstration of the use of resampling filter theory for texture mapping.

3.2 Sampling: The Cause of Aliasing

The aliasing problem in computer graphics and image processing is due to the digital nature of our computers. Whereas images in nature and geometric models of objects have a continuous spatial

Figure 3.1: *Texture mapping with point sampling. Texture resolution is close to screen resolution, so little aliasing results.*

Figure 3.2: *Texture mapping with point sampling, with more extreme compression. Note moire near horizon.*

Figure 3.3: *Sampling and reconstruction of a continuous signal.*

domain, images in a computer are represented by a discrete array of samples. We will focus on one-dimensional signals for the moment, and return to multidimensional signals (such as images) later.

When a digital signal is created from an analog signal it is *sampled* in space and *quantized* in value. We are unconcerned with quantization here, so we will call continuous-space signals simply *continuous signals* and call discrete-space signals *discrete signals*. Figure 3.3 (left) shows the conversion of a continuous signal a_c into a discrete signal a by periodic sampling. This signal could represent the intensity profile of one scan line of an image, or perhaps an audio signal varying over time. Regardless of the signal's meaning, conversion to and from a discrete representation has a number of consequences.

To explain these consequences we need to define some filter terminology [Jackson86]. A *system* that processes an input signal a to produce an output signal or *response* b is called a *filter*. A filter is called *space invariant* (or time invariant) if a spatial shift in the input causes an equal shift in the output: input $a(x-s)$ gives output $b(x-s)$ for any shift s. A filter is called *linear* if scaling or superposition in its input causes equivalent scaling or superposition in its output: input $\lambda_1 a_1(x) + \lambda_2 a_2(x)$ gives output $\lambda_1 b_1(x) + \lambda_2 b_2(x)$ for all λ_1, λ_2. If a filter is both linear and space invariant then it is uniquely characterized by its *impulse response* $h(x)$, the output resulting from an impulse input. (An impulse $\delta(x)$ is a sample occurring at $x = 0$ and is zero elsewhere). Since any input signal can be represented in the limit by an infinite sum of shifted and scaled impulses, the output of a linear, space invariant filter, by these properties, will be a superposition of shifted, scaled impulse responses. This operation is the *convolution* of the input signal a with the filter h. The convolution of signals f and g is $(f \circledast g)(x) = \int_{-\infty}^{+\infty} f(t)g(x-t)\,dt$. Later we will consider more general, *space variant* filters.

The difficulties with discrete representation become evident when we try to reconstruct a continuous signal from a discrete signal. This is done by convolving the sampled signal with a *reconstruction filter*. The theoretically ideal reconstruction filter is the sinc function, $(\mathrm{sinc}(x) = \sin(\pi x)/\pi x)$. (We will discuss the derivation of the sinc filter later.) As shown in figure 3.3 (right), reconstruction with a sinc does not always recover the original signal. Why not? Judging this figure visually, the approximation appears best toward the left where the original signal varies slowly, but is poor toward the right, where the original signal varies with high frequency. This behavior suggests that reconstruction accuracy is a function of frequency. Aliasing is best understood in terms of a frequency analysis of signals and filters using the Fourier transform [Brigham74].

3.2.1 Frequency Analysis of Aliasing

The units of frequency can be either *angular frequency* in radians per unit distance, denoted ω, or *rotational frequency* in cycles per unit distance, denoted f. The two are related by $\omega = 2\pi f$. We will use angular frequency ω.

The response of a linear, space invariant system to a sinusoidal (complex exponential) input is a complex exponential of the same frequency but altered amplitude and phase. The complex function $F(\omega)$ describing the amplitude and phase of these responses at each frequency ω is the system's *frequency response*. Complex exponentials are the eigenfunctions of the system, and the frequency response is the set of eigenvalues. The frequency response of a filter can be computed as the Fourier transform of its impulse response. The Fourier transform of a signal (as opposed to a filter) is called the signal's *spectrum*. To relate the Fourier transform pair f and F, we write $f(x) \leftrightarrow F(\omega)$, and we say that $f(x)$ lies in the *spatial domain* and $F(\omega)$ lies in the *frequency domain*.

The Fourier transform is defined as

$$F(\omega) = \int_{-\infty}^{+\infty} f(x) e^{-j\omega x}\, dx$$

and the inverse Fourier transform describes the signal in terms of its spectrum:

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} F(\omega) e^{j\omega x}\, d\omega$$

Some other important properties of the Fourier transform:

(1) The Fourier transform of a convolution of two signals is the product of their Fourier transforms: $f \circledast g \leftrightarrow FG$.

(2) The Fourier transform of a product of two signals is the convolution of their Fourier transforms: $fg \leftrightarrow F \circledast G/2\pi$.

(3) When a signal is scaled up spatially, its spectrum is scaled down in frequency, and vice versa. This is an uncertainty principle for signals: a signal cannot be narrow both spatially and spectrally [Hamming83].

(4) The Fourier transform of a real function is a conjugate-symmetric function.

(5) The Fourier transform of a periodic function is a discrete function, and vice versa.

Using frequency analysis we can understand the flaws in our previous reconstruction. As shown in figure 3.4, examination of the spectrum of the signal before and after sampling reveals the problem. Sampling the continuous signal $a_c(x)$ to arrive at the discrete signal in $a(x)$ is equivalent to multiplying it by an impulse train (figure 3.4b): $a(x) = a_c(x)i(x/T)$, where T is the *sample period* and $i(x)$ is a sum of unit-spaced impulses: $i(x) = \sum_{n=-\infty}^{+\infty} \delta(x-n)$. The discrete signal is a product of signals, and multiplication in the spatial domain corresponds to convolution in the frequency domain, so the spectrum of the discrete signal is the convolution of the spectrum of the continuous signal with the spectrum of the sampling grid: $A(\omega) = A_c(\omega) \circledast I(\omega)/2\pi$. But the Fourier transform of an impulse train $i(x/T)$ is another impulse train $I(\omega) = \omega_s i(\omega/\omega_s)$, where $\omega_s = 2\pi/T$ is the spatial *sampling frequency*. Convolution with an impulse train superposes a

Figure 3.4: *Sampling above the Nyquist rate (no aliasing).*

Figure 3.5: *Sampling below the Nyquist rate (aliasing).*

Figure 3.6: *Ideal antialiasing is prefiltering before sampling.*

sequence of translated replicas of the spectrum A_c, making the spectrum of the sampled signal periodic (figure 3.4c).

If the replicated spectra do not overlap in the spectrum of the discrete signal then we can reconstruct the continuous signal by eliminating all replicas except the central one (figure 3.4e). We can do this with a multiplication in the frequency domain, which corresponds to a filtering operation. The type of filter needed for reconstruction is thus a *low pass filter*, a filter that passes low frequencies and stops high frequencies (figure 3.4d). The frequency response of an ideal low pass filter with cutoff ω_c is a box function: $H(\omega) = 1$ for $\omega \leq \omega_c$ and 0 otherwise, and its impulse response is a sinc, as mentioned earlier. For reconstruction we want a cutoff frequency at the *Nyquist frequency* $\omega_s/2 = \pi/T$, which is half the sampling frequency.

If the replicas overlap, as in figure 3.5, it is impossible to extract the original spectrum A_c from A once the replicas are summed. *This is aliasing.* The high frequencies of the continuous signal, when replicated and summed by the sampling process, masquerade, or *alias*, as low frequencies. Aliasing need not occur if the continuous signal is *band limited.* A signal is band limited with *bandwidth* ω_b if it has no frequencies above ω_b: $A_c(\omega) = 0$ for $|\omega| \geq \omega_b$. For a band limited continuous signal, aliasing will not occur if the bandwidth is less than the Nyquist frequency ($\omega_b < \omega_s/2$). An equivalent statement is the Sampling Theorem [Shannon49], which states that a continuous signal with bandwidth ω_b can be reconstructed exactly if the sampling frequency is at least twice the bandwidth ($\omega_s > 2\omega_b$).

3.3 Antialiasing

Typically a continuous input signal is not band limited, yet we would like to perform meaningful processing anyway. There are two general methods for *antialiasing*, that is, reducing aliasing, in this situation: (1) sampling at a higher frequency, and (2) prefiltering the signal.

3.3.1 Sampling at Higher Frequency

Sampling at a higher frequency won't eliminate aliasing if the input signal is not band limited, but it will usually reduce it, since most signals of interest have decreasing amplitude in the high frequencies. Unfortunately, increasing the sampling frequency is often prohibitively expensive in both memory and time. Furthermore, it is impossible to pick a sampling frequency that is adequate for all input signals, since most are not bandlimited.

3.3.2 Prefiltering

The more theoretically correct antialiasing method is *prefiltering*. A prefilter is a low pass filter applied to a signal before sampling (figure 3.6). If h is the prefilter then the new, prefiltered input signal is $a'_c = a_c \circledast h$, and the discrete signal is its sampling: $a = a'_c i$. If we use an ideal low pass filter with cutoff at the Nyquist frequency $\omega_s/2$ then the filtered continuous signal a'_c will be band limited and we can reconstruct it exactly from a, as shown in figure 3.7.

The widths of a filter in the spatial and frequency domains are inversely related. The width of the nonzero portion of an impulse response is called its *support*. A filter with finite support is called a *finite impulse response* (FIR) filter, and a filter with infinite support is an *infinite impulse response* (IIR) filter. The uncertainty principle thus implies that a finite impulse response filter has infinite frequency response and a finite frequency response filter has infinite impulse response. Therefore, an ideal low pass filter, because it has finite frequency response, is not an FIR filter. FIR filters are precisely the class of filters most common in image processing, however, because they are the easiest 2-D filters to design and implement [Dudgeon-Mersereau84]. And since FIR filters are usually implemented using convolution, the expense of which is proportional to the support, we desire filters with small support. Narrowing the support broadens the frequency response, generally decreasing the quality of the filter, however. The uncertainty principle for signals thus causes a tradeoff between the efficiency of an FIR filter and its quality.

When antialiasing in practice, then, no prefilter is an ideal low pass filter, so some aliasing is inevitable during sampling. Antialiasing amounts to the design of prefilters that can be implemented efficiently yet admit minimal aliasing.

In actual systems, reconstruction is done with a less-than-ideal reconstruction filter. The reconstruction filter that smooths between scan lines of a video signal on a cathode ray tube is not a sinc, but is approximately Gaussian. When a non-ideal reconstruction filter is known a priori then the prefilter methods above are inappropriate; we can do better with prefilters that are adapted to the reconstruction filter [Kajiya-Ullner81], [Hummel83].

So far we have spoken only of spatial aliasing. Aliasing can occur in time as well. Such *temporal aliasing* is visible in movies when a fast-moving wheel appears to spin backwards. This "wagon wheel effect" is due to the poor prefiltering performed by film camera shutters that stay open for only a fraction of the frame time. If cameras effected a better temporal low pass filter then a fast-moving wheel would appear as a blur in movies, as it does to the naked eye.

3.3.3 Some Low Pass Filters

Low pass filters are usually designed with the frequency domain constraints shown in figure 3.8 [Jackson86]. As stated earlier, the ideal low pass filter is a box in frequency space, but such a filter is not realizable as an FIR filter, so the ideal is padded with tolerances to make it realizable. The *pass band* is the range of frequencies that are passed relatively unchanged by the filter, the *stop band* is the range of frequencies that are nearly eliminated by the filter, and the *transition band* is the intermediate range of frequencies. High quality filters have small tolerances on the pass and stop bands and a narrow transition band, at the expense of wide support in the spatial domain.

Low pass filters will often ripple in the pass band, the stop band, or in both bands. The central hump of the frequency response is called the *main lobe* and the other ripples are called *side lobes*. The height of the first side lobe relative to the main lobe is one quantitative measure of filter quality.

Figure 3.7: *Ideal prefiltering eliminates aliasing.*

Figure 3.8: *Filter design parameters.*

To design an FIR filter we often start with an IIR filter and *window* it, multiplying the IIR filter by a *window function* of finite support. Window functions are usually even, positive functions that decay smoothly to zero [Hamming83]. Windowing with a box amounts to truncation of the impulse response; this is one of the poorest window functions. We will need to window if we choose an IIR filter such as a Gaussian or sinc.

We now examine three classes of low pass filters, the b-spline filters, Gaussians, and Sinc filters. The following filters are normalized to have unit area in the spatial domain, and have horizontal scale appropriate for reconstruction of unit-spaced samples.

B-Spline Filter

The simplest set of filters are those derived from a box filter by repeated self-convolution. We start with an unweighted average or first order box:

$$b_1(x) = \begin{cases} 1 & |x| \leq 1/2 \\ 0 & |x| > 1/2 \end{cases} \qquad \leftrightarrow \qquad \frac{\sin \omega/2}{\omega/2} = \frac{\sin \pi f}{\pi f} = \mathrm{sinc} f$$

where $f = \omega/2\pi$ is the rotational frequency. Its frequency response is a sinc, as shown at right. This filter, when sampled, is trivial to implement, (involving several additions and one multiplication), but its frequency response is far from ideal, with high side lobes.

We can make higher quality filters by cascading n of these box filters in series. A cascade of filters is equivalent to a single filter whose impulse response is the convolution of all of its component filters, so we define the *nth order b-spline filter* to be:

$$b_n(x) = \overbrace{b_1(x) \circledast b_1(x) \circledast \cdots \circledast b_1(x)}^{n \text{ times}} \qquad \leftrightarrow \qquad \mathrm{sinc}^n f$$

The first two b-spline filters are pictured in figure 3.9 (top).

The b-spline functions are common in signal processing, computer aided geometric design, and probability. Unfortunately, each discipline has its own names for these functions, and there has been little cross-pollination to date, with the exception of a few papers [Hou-Andrews78], [Goldman83],

Figure 3.9: *B-Spline, Gaussian, and Sinc filters in the spatial domain (left) and frequency domain (right). Top: Box (first order b-spline) and tent (second order b-spline) filters. Center: Gaussian filters. Bottom: Sinc filter. From [Blinn89].*

[Heckbert86a]. In the field of splines these functions are known as the uniform b-spline basis functions [Bartels-Beatty-Barsky87]. In probability these functions arise as the probability density of the sum of n uniform random variables.

B-spline filters of low order ($n = 1$ or 2) make relatively poor low pass filters because the envelope of their frequency responses decay only as ω^{-n}. Nevertheless, these two are perhaps the most commonly used low pass filters in image processing because of the simplicity of implementation. In two dimensions, the second order box filter is called *bilinear interpolation*.

Gaussian Filter

The impulse response and frequency response of a Gaussian filter have similar shape, as shown in figure 3.9 (center):

$$g_{\sigma^2}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2} \qquad \leftrightarrow \qquad G_{\sigma^2}(\omega) = e^{-\sigma^2\omega^2/2} = \frac{\sqrt{2\pi}}{\sigma} g_{1/\sigma^2}(\omega)$$

For reconstruction of unit-spaced samples we want a standard deviation of about $\sigma = 1/2$.

The b-spline filters are FIR approximations to the Gaussian. By the Central Limit Theorem of probability [Bracewell78], as n increases, b_n broadens and approaches a Gaussian (normal distribution) in shape. The standard deviation of these functions grows as the square root of n: $b_n(x) \approx g_{\sigma^2}(x)$ where $\sigma = \sqrt{n/12}$. An FIR approximation to the Gaussian can also be made by windowing. Its tails diminish rapidly, so a Gaussian can be truncated (windowed with a box) with little loss of quality.

Sinc Filter

The sinc filter is an ideal low pass filter with a sharp cutoff in its frequency response:

$$\mathrm{sinc}(\omega_c x/\pi) = \frac{\sin(\omega_c x)}{\pi x} \qquad \leftrightarrow \qquad b_1\left(\frac{\omega}{2\omega_c}\right) = \begin{cases} 1 & |\omega| \leq \omega_c \\ 0 & |\omega| > \omega_c \end{cases}$$

The sinc filter has an ideal frequency response but its impulse response decays slowly (with an envelope of $1/x$), so windowing causes significant degradation of the filter.

3.4 Ideal Resampling Filters

For texture mapping and image warping, the sampling grids of the source and destination images differ, so the image must be resampled. To avoid aliasing during resampling, we need to filter the source image before it is sampled. The *resampling filter* required depends on the mapping that defines the warp being performed (figure 3.10). We refer to the entire process of filtering and sampling as *resampling*.

Most previous work on resampling filters has concentrated on the special case of affine warps, for which the resampling filter is space invariant. A common task in audio processing is the conversion of a discrete 1-D signal from one sampling rate to another [Crochiere-Rabiner83]. The terminology

Figure 3.10: *Forward and inverse mapping controlling a warp.*

of earlier researchers differs somewhat from ours: where they perform "interpolation" or "decimation" at a "uniform sampling rate" during "sampling rate conversion" for a "multirate" task, we perform "magnification" or "decimation" according to "affine mappings" during "resampling" for a "warping" task. For image processing we find our terminology more appropriate and concise.

For their applications, Crochiere and Rabiner use 1-D space invariant resampling filters, but since we are interested in resampling filters for arbitrary 2-D mappings, our filters must be space variant, requiring new theory.

3.4.1 General, Ideal Resampling Filters

We define *proper sampling* of a continuous signal as sampling at a frequency meeting the Nyquist criterion, and *ideal resampling* as a process that inputs a properly sampled signal, warps it, and outputs a properly sampled signal – with minimum information loss. As shown in figures 3.11 and 3.12 (top), the procedure for ideal resampling consists of four steps: reconstruction, warp, prefilter, and sample [Smith83]:

1. Reconstruct the continuous signal from the discrete input signal.
2. Warp the domain of the continuous signal.
3. Prefilter the warped, continuous signal.
4. Sample this signal to produce the discrete output signal.

A closely related procedure is described by [Clark-Palmer-Lawrence85]. The problem they address is the reconstruction of a continuous signal from a discrete signal sampled with nonuniform spacing, whereas our problem, resampling, involves the creation of a uniformly sampled discrete output signal from a uniformly sampled discrete input. Their goals are somewhat different, but their method, like ours, describes a sampling grid in terms of a mapping.

We will derive the resampling formula for general, n-dimensional signals. Given an input signal $f(\mathbf{u})$, a forward mapping $\mathbf{x} = \mathbf{m}(\mathbf{u})$ from source coordinates $\mathbf{u}$ to destination coordinates $\mathbf{x}$, the inverse mapping $\mathbf{u} = \mathbf{m}^{-1}(\mathbf{x})$, a reconstruction filter $r(\mathbf{u})$ and a prefilter $h(\mathbf{x})$, we want to compute the output signal $g(\mathbf{x})$. For 2-D (image) resampling, the source coordinates are $\mathbf{u} = (u, v)$ and the

Figure 3.11: *The four steps of ideal resampling: reconstruction, warp, prefilter, and sample.*

destination coordinates are $\mathbf{x} = (x, y)$. Theoretically, the reconstruction filter and prefilter will be ideal low pass filters (we will later approximate them by FIR filters).

The progression of signals is:

discrete input signal	$f(\mathbf{u})$, integer $\mathbf{u}$
reconstructed input signal	$f_c(\mathbf{u}) = f(\mathbf{u}) \circledast r(\mathbf{u}) = \sum_{\mathbf{k} \in \mathcal{Z}^n} f(\mathbf{k})\, r(\mathbf{u} - \mathbf{k})$
warped signal	$g_c(\mathbf{x}) = f_c(\mathbf{m}^{-1}(\mathbf{x}))$
continuous output signal	$g_c'(\mathbf{x}) = g_c(\mathbf{x}) \circledast h(\mathbf{x}) = \int_{\mathcal{R}^n} g_c(\mathbf{t})\, h(\mathbf{x} - \mathbf{t})\, d\mathbf{t}$
discrete output signal	$g(\mathbf{x}) = g_c'(\mathbf{x})\, i(\mathbf{x})$

When resampling we never actually construct a continuous signal; we compute its value only at the relevant sample points (figure 3.12, bottom). The steps above suggest a multi-pass approach in which we convolve, then warp, then convolve, but we can reorder the computations to group together all the operations needed to produce a given output sample. Expanding the above relations

Figure 3.12: *Resampling filter block diagram. Top: conceptual model. Bottom: implementation.*

in reverse order, we have, for $\mathbf{x}$ on the integer lattice:

$$\begin{aligned}
g(\mathbf{x}) &= g_c'(\mathbf{x}) \\
&= \int_{\mathcal{R}^n} f_c(\mathbf{m}^{-1}(\mathbf{t}))\, h(\mathbf{x}-\mathbf{t})\, d\mathbf{t} \\
&= \int_{\mathcal{R}^n} h(\mathbf{x}-\mathbf{t}) \sum_{\mathbf{k}\in\mathcal{Z}^n} f(\mathbf{k})\, r(\mathbf{m}^{-1}(\mathbf{t})-\mathbf{k})\, d\mathbf{t} \\
&= \sum_{\mathbf{k}\in\mathcal{Z}^n} f(\mathbf{k})\, \rho(\mathbf{x},\mathbf{k})
\end{aligned}$$

$$\text{where: } \rho(\mathbf{x},\mathbf{k}) = \int_{\mathcal{R}^n} h(\mathbf{x}-\mathbf{t})\, r(\mathbf{m}^{-1}(\mathbf{t})-\mathbf{k})\, d\mathbf{t} \tag{3.1}$$

We call $\rho(\mathbf{x},\mathbf{k})$ the *resampling filter*. The resampling filter $\rho(\mathbf{x},\mathbf{k})$ specifies the weight of the input sample at location $\mathbf{k}$ for an output sample at location $\mathbf{x}$. It is space variant in general.

This equation expresses the resampling filter as a destination space integral of the prefilter with a warped reconstruction filter, but we can also express it as the source space integral of a warped prefilter with the reconstruction filter. Assuming $\mathbf{m}^{-1}(\mathbf{x})$ is invertible, we can change variables by substituting $\mathbf{t} = \mathbf{m}(\mathbf{u})$. We call the following the *Resampling Formula*:

$$\rho(\mathbf{x},\mathbf{k}) = \int_{\mathcal{R}^n} h(\mathbf{x}-\mathbf{m}(\mathbf{u}))\, r(\mathbf{u}-\mathbf{k}) \left|\frac{\partial \mathbf{m}}{\partial \mathbf{u}}\right| d\mathbf{u} \tag{3.2}$$

where $|\partial\mathbf{m}/\partial\mathbf{u}|$ is the determinant of the Jacobian matrix [Strang80]. In 1-D, $|\partial\mathbf{m}/\partial\mathbf{u}| = dm/du$, and in 2-D,

$$\left|\frac{\partial \mathbf{m}}{\partial \mathbf{u}}\right| = \begin{vmatrix} x_u & x_v \\ y_u & y_v \end{vmatrix}$$

where $x_u = \partial x / \partial u$, etc. Formula (3.2) for the resampling filter will often be more convenient for computation than formula (3.1).

3.4.2 Affine Mappings and Space Invariant Resampling

In the special case of affine mappings, the forward mapping is $\mathbf{x} = \mathbf{m}(\mathbf{u}) = \mathbf{l}(\mathbf{u}) + \mathbf{c}$ and the inverse mapping is $\mathbf{u} = \mathbf{m}^{-1}(\mathbf{x}) = \mathbf{l}^{-1}(\mathbf{x}) + \mathbf{c}^{-1}$ where $\mathbf{l}(\mathbf{u})$ and $\mathbf{l}^{-1}(\mathbf{x})$ are linear mappings and $\mathbf{c}$ and $\mathbf{c}^{-1}$ are constants. The Jacobian of an affine mapping is a constant transformation matrix $\mathbf{J} = \partial\mathbf{m}/\partial\mathbf{u}$ interrelating source and destination spaces. The linear mappings can be expressed in terms of the Jacobian: $\mathbf{l}(\mathbf{u}) = \mathbf{uJ}$ and $\mathbf{l}^{-1}(\mathbf{x}) = \mathbf{xJ}^{-1}$. For affine mappings, the identity $\mathbf{l}^{-1}(\mathbf{x}-\mathbf{y}) = \mathbf{m}^{-1}(\mathbf{x}) - \mathbf{m}^{-1}(\mathbf{y})$ allows the resampling filter to be expressed as a convolution:

$$\begin{aligned}
\rho(\mathbf{x},\mathbf{k}) &= \int h(\mathbf{x}-\mathbf{m}(\mathbf{u}))\, r(\mathbf{u}-\mathbf{k}) \left|\frac{\partial\mathbf{m}}{\partial\mathbf{u}}\right| d\mathbf{u} \\
&= \int h'(\mathbf{l}^{-1}(\mathbf{x}-\mathbf{m}(\mathbf{u})))\, r(\mathbf{u}-\mathbf{k})\, d\mathbf{u} \qquad \text{where } h'(\mathbf{u}) = \left|\frac{\partial\mathbf{m}}{\partial\mathbf{u}}\right| h(\mathbf{l}(\mathbf{u})) = |\mathbf{J}|\, h(\mathbf{uJ}) \\
&= \int h'(\mathbf{m}^{-1}(\mathbf{x})-\mathbf{u})\, r(\mathbf{u}-\mathbf{k})\, d\mathbf{u} \\
&= \rho'(\mathbf{m}^{-1}(\mathbf{x})-\mathbf{k}) \qquad \text{where } \rho'(\mathbf{u}) = h'(\mathbf{u}) \circledast r(\mathbf{u})
\end{aligned}$$

The resampling filter for an affine mapping is thus space invariant, or more precisely, the filter $q(\mathbf{u},\mathbf{k}) = \rho(\mathbf{m}(\mathbf{u}),\mathbf{k})$ is space invariant, since

$$q(\mathbf{u}-\mathbf{t},\mathbf{k}-\mathbf{t}) = \rho'(\mathbf{m}^{-1}(\mathbf{m}(\mathbf{u}-\mathbf{t})) - (\mathbf{k}-\mathbf{t})) = \rho'(\mathbf{u}-\mathbf{k}) = q(\mathbf{u},\mathbf{k})$$

The output signal is therefore a convolution of the input with the resampling filter:

$$\begin{aligned}
g(\mathbf{x}) &= \sum_{\mathbf{k}} f(\mathbf{k})\, \rho'(\mathbf{m}^{-1}(\mathbf{x})-\mathbf{k}) \\
&= (f \circledast \rho')(\mathbf{m}^{-1}(\mathbf{x}))
\end{aligned}$$

3.4.3 Two-Dimensional Affine Mappings

The equations above apply in any number of dimensions. Specializing for 2-D affine mappings, we have the forward and inverse mappings:

$$\begin{pmatrix} x & y & 1 \end{pmatrix} = \begin{pmatrix} u & v & 1 \end{pmatrix} \begin{pmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{pmatrix}$$

$$\begin{aligned}
\begin{pmatrix} u & v & 1 \end{pmatrix} &= \begin{pmatrix} x & y & 1 \end{pmatrix} \begin{pmatrix} A & D & 0 \\ B & E & 0 \\ C & F & 1 \end{pmatrix} \\
&= \begin{pmatrix} x & y & 1 \end{pmatrix} \frac{1}{ae-bd} \begin{pmatrix} e & -d & 0 \\ -b & a & 0 \\ bf-ce & cd-af & ae-bd \end{pmatrix}
\end{aligned}$$

The Jacobian is the constant matrix:

$$\frac{\partial\mathbf{m}}{\partial\mathbf{u}} = \mathbf{J} = \begin{pmatrix} a & d \\ b & e \end{pmatrix}$$

and the Jacobian determinant is $|\mathbf{J}| = ae - bd$. The output signal is the convolution:

$$\begin{aligned} g(\mathbf{x}) = g(x,y) &= (f \circledast \rho')(\mathbf{m}^{-1}(\mathbf{x})) \\ &= (f \circledast \rho')(Ax+By+C,\ Dx+Ey+F) \end{aligned}$$

where the resampling filter is:

$$\rho' = h' \circledast r$$

and the warped prefilter is:

$$\begin{aligned} h'(\mathbf{u}) = h'(u,v) &= |\mathbf{J}|\, h(\mathbf{uJ}) \\ &= (ae-bd)\, h(au+bv,\ du+ev) \end{aligned}$$

3.4.4 Summary: Resampling Affine Mappings

We have seen that the mapping, the sampling grid, and the resampling filter are all closely interrelated. The interrelationships are particularly simple for affine mappings. For an affine mapping, ideal resampling reduces to a space invariant filter, which can be computed easily using convolution. For a given mapping $\mathbf{m}(\mathbf{u})$, the following statements are thus equivalent:

- mapping is affine
- Jacobian of mapping is constant
- ideal resampling filter is space invariant
- ideal resampling filter is a convolution
- resampling grid is uniform

3.5 Image Resampling in Practice

In practice, analytic signal representations are rare, so we usually must perform filtering numerically. Consequently, the filters used for resampling must have finite impulse response, and the low pass filters used for reconstruction and prefiltering must be less than ideal. Computation of the mapping itself or the integral implied by the resampling formula can also be impractical. These realities force us to approximate the ideal resampling filter.

3.5.1 Resampling by Postfiltering

If the mapping is not affine then the resampling filter is in general space variant. Space variant resampling filters are much more difficult to implement than space invariant filters, so one might be tempted to sample before filtering in such cases. This method, called *postfiltering*, is a simpler alternative to resampling with prefiltering:

1. Reconstruct the continuous signal from the discrete input signal.
2. Warp the domain of the input signal.

3. Sample the warped signal at high resolution.

4. Postfilter the high resolution signal to produce a lower resolution output signal.

The computational advantage of postfiltering is that the postfilter is discrete and space invariant.

Postfiltering eliminates aliasing if the sampling frequency meets the Nyquist criterion, but many continuous signals are not so conveniently band limited. Band unlimited signals are common in texture mapping. Continuous, synthetic images such as textured, infinite planes viewed in perspective have unlimited high frequencies near the horizon. Increasing the sampling frequency may reduce, but will not eliminate, the aliasing in such images.

3.5.2 Image Processing Considerations

Image processing manipulates two-dimensional signals with spatial coordinates x and y and frequency coordinates ω_x and ω_y. Most "images" in image processing are intensity images, in which values are proportional to light intensity. These intensity images are always nonnegative since "negative light" does not exist. Filters with negative lobes will occasionally output negative pixel values which must be corrected in some way.

When displaying a picture antialiased with the linear theory above, it is essential that displayed intensity be proportional to the computed intensity. Most video displays have a nonlinear response approximately described by a power law of the form (displayed intensity) = (input)$^\gamma$, where $2 < \gamma < 3$ depending on the monitor. To compensate for this nonlinearity we must perform *gamma correction*. The most popular correction method uses pixel values proportional to intensity and a hardware colormap containing the function (monitor input) = C(pixel value)$^{1/\gamma}$ [Catmull79]. Without gamma correction, efforts toward antialiasing are hampered.

3.5.3 Filter Shape

A general 2-D filter has an impulse response (or *point spread function*) of two variables $h(x, y)$. Two common classes of image filter are those that are *separable* into a product of 1-D functions: $h(x, y) = h_x(x)h_y(y)$ or circularly symmetric: $h(x, y) = h_r(\sqrt{x^2 + y^2})$. The Gaussian is unique in that it is both circularly symmetric and separable: $e^{-r^2} = e^{-x^2-y^2} = e^{-x^2}e^{-y^2}$.

We describe the shape of 2-D filters in terms of their *profile* (their cross sectional shape) and their *regions* (the shape of one or more iso-value contours), as shown in figure 3.13. The ideal low pass filter for reconstructing images sampled on a square grid is the 2-D sinc: $\mathrm{sinc}(x)\mathrm{sinc}(y)$, whose frequency response is a square box (square region, box profile).

In algebraic terminology, we call the destination space region corresponding to a source space region the *image* of the source space region, and the source region corresponding to a destination region the *preimage* of the destination region (figure 3.14).

3.5.4 Resampling Filter Shape

When we design a resampling filter we are free to choose the filter's region in one space (either source space or destination space), but the mapping determines the region in the other space. The

Figure 3.13: *Profile and regions of a 2-D filter impulse response. (For the elliptical Gaussian filter pictured, the regions are ellipses and the profile is a Gaussian).*

Figure 3.14: *For non-affine mappings, the preimages of circular screen space regions (bottom) are varying shapes in texture space (top).*

Figure 3.15: *A nonuniform resampling grid and the corresponding space variant filter regions. Circular regions in screen space (left) map to elliptical regions in texture space (right). Since the texture regions vary in size, eccentricity, and orientation, the filter is space variant. Dots mark pixel centers.*

ideal resampling filter is defined by the resampling formula:

$$\rho(\mathbf{x}, \mathbf{k}) = \int h(\mathbf{x} - \mathbf{m}(\mathbf{u}))\, r(\mathbf{u} - \mathbf{k}) \left| \frac{\partial \mathbf{m}}{\partial \mathbf{u}} \right| d\mathbf{u}$$

This filter does ideal reconstruction and prefiltering, and it is adaptive to the mapping being performed. Depending on the mapping, the sampling grid could be nonuniform and the ideal resampling filter could have arbitrary, space variant shape (figure 3.15). Its prefilter and reconstruction filter are 2-D sinc functions, so its support will in general be infinite (IIR).

In practice, resampling filters are approximations to the ideal, with finite support and a limited class of region shapes. Resampling filters can be categorized by the approximations they make to the shape of the reconstruction filter, prefilter, and mapping function.

For resampling we are most frequently interested in the source regions for square or circular destination regions. These are the region shapes most common in 2-D filtering, corresponding to separable and circularly symmetric filtering, respectively. Square regions in destination space have a curvilinear quadrilateral preimage in source space for an arbitrary mapping. For projective or affine mappings the preimage of a square is a quadrilateral or parallelogram, respectively. Circular destination regions have an arbitrary preimage for general mappings, but for projective or affine mappings the preimage is a conic curve or ellipse, respectively. The conic resulting from a projective mapping will be an ellipse except when the destination circle crosses the "horizon line", the image of the line at infinity, in which case the conic will be a parabola or hyperbola. These correspondences

Figure 3.16: *Source regions for various destination regions and mapping classes.*

are summarized in figure 3.16 and in the table below.

Source Regions for Various Destination Regions and Mapping Classes:

	DESTINATION REGION	
MAPPING CLASS	*square*	*circle*
general mapping	arbitrary	arbitrary
projective mapping	quadrilateral	conic curve
affine mapping	parallelogram	ellipse

3.5.5 Local Affine Approximation

To simplify the resampling formula we substitute for the mapping $\mathbf{m}$ its *local affine approximation*. The local affine approximation to $\mathbf{m}(\mathbf{u})$ in the neighborhood of $\mathbf{u}_0 = \mathbf{m}^{-1}(\mathbf{x}_0)$ is:

$$\mathbf{m}_{\mathbf{u}_0}(\mathbf{u}) = \mathbf{x}_0 + (\mathbf{u} - \mathbf{u}_0)\mathbf{J}_{\mathbf{u}_0}$$

where

$$\mathbf{J}_{\mathbf{u}_0} = \frac{\partial \mathbf{m}}{\partial \mathbf{u}}(\mathbf{u}_0)$$

is the Jacobian matrix evaluated at $\mathbf{u}_0$. The above approximation is most accurate in the neighborhood of $\mathbf{u}_0$. In the resampling formula for $\rho(\mathbf{x}_0, \mathbf{k})$, the prefilter $h(\mathbf{x}_0 - \mathbf{m}(\mathbf{u}))$ weights the integral most heavily in the neighborhood of $\mathbf{u} \approx \mathbf{m}^{-1}(\mathbf{x}_0) = \mathbf{u}_0$, where the approximate mapping is most accurate, so we can use the local affine mapping to find a approximate resampling filter. The impulse response of the approximate filter thus has the form:

$$\rho_{\mathbf{u}_0}(\mathbf{x}_0, \mathbf{k}) = \int h(\mathbf{x}_0 - \mathbf{m}_{\mathbf{u}_0}(\mathbf{u}))\, r(\mathbf{u} - \mathbf{k}) |J_{\mathbf{u}_0}|\, d\mathbf{u}$$

Since $\mathbf{m}_{\mathbf{u}_0}$ is an affine mapping, this integral is a convolution, simplifying evaluation of the resampling formula. The resulting filter is still space variant, however, since the local affine approximation and its associated filter are position-dependent.

3.5.6 Magnification and Decimation Filters

The approximations to the reconstruction filter, prefilter, and mapping must often be crude for the filter resulting from the resampling formula to be practical. To simplify the integral, it is common to assume that either the reconstruction filter or the prefilter is an impulse (delta function).

We cover several classes of approximate resampling filters below. First we discuss the special cases of magnification and decimation filters, which respectively ignore the prefilter and reconstruction filter. Then, returning to the general case, we derive a resampling filter with a Gaussian profile and elliptical region.

Magnification Filters

When the mapping magnifies the source image, the size and shape of the ideal resampling filter is dominated by the reconstruction filter (figure 3.17b). If we assume the prefilter is an impulse $h = \delta$, then the *magnification filter* is:

$$\rho(\mathbf{x}, \mathbf{k}) \approx r(\mathbf{m}^{-1}(\mathbf{x}) - \mathbf{k})$$

This is an unwarped reconstruction filter centered on $\mathbf{u} = \mathbf{m}^{-1}(\mathbf{x})$. Its support is independent of the mapping, typically 1-4 pixels. Such filters are called interpolation filters in image processing. Common magnification filters are the box (nearest neighbor), bilinear, and cubic b-spline [Hou-Andrews78].

Decimation Filters

When the mapping shrinks the source image, the size and shape of the ideal resampling filter is dominated by the warped prefilter (figure 3.17c), so if we assume the reconstruction filter is an impulse $r = \delta$, then the *decimation filter* is:

$$\rho(\mathbf{x}, \mathbf{k}) \approx \left| \frac{\partial \mathbf{m}}{\partial \mathbf{u}} \right| h(\mathbf{x} - \mathbf{m}(\mathbf{k}))$$

This is a prefilter centered on $\mathbf{u} = \mathbf{m}^{-1}(\mathbf{x})$, warped according to the mapping $\mathbf{m}$. Its support in source space is proportional to the reciprocal of the scale factor of the mapping. Since the scale factors of mappings are unlimited, a decimation filter could have arbitrary support: one pixel or a million pixels.

Figure 3.17: *The region of an affine resampling filter is the convolution of the regions of its warped prefilter h' and its reconstruction filter r. When magnifying, the reconstruction filter dominates, and when decimating, the warped prefilter dominates. Behavior for non-affine mappings is similar.*

Figure 3.18: *The ideal resampling filter (left) is always wider than both the warped prefilter and the reconstruction filter. If a decimation filter is used on a mapping that magnifies in some direction then the filter can miss all sample points (right).*

The region of a decimation filter is completely determined by the destination region of the prefilter and the mapping class. Prefilters are usually chosen to have a square or circular destination region, in which case the source region will be a parallelogram or ellipse, respectively.

Magnification/Decimation Transitions

Many filter implementations for texture mapping and image warping use a combination of magnification and decimation filters, switching from decimation to magnification when the scale of the mapping rises above 1. This rule is ambiguous in 2-D, however, where the scale is given not by a scalar, but by a 2×2 Jacobian matrix. In 2-D, a mapping could decimate along one axis but magnify along another. An ad hoc solution to this difficulty is to use a magnification filter when both axes are magnified, and a decimation filter otherwise. This is not totally satisfactory, however. If a mapping magnifies along just one axis, then its decimation filter could become so narrow that it misses all sample points (figure 3.18). In this case we can artificially fatten the filter along that dimension to have support of at least one pixel.

As an alternative we can develop unified classes of resampling filters that make smoother transitions from magnification to decimation. To do this we cannot assume that either the reconstruction filter or prefilter is an impulse. Before we investigate one such unified resampling filter, based on Gaussians, we must first lay some groundwork regarding filter warps.

3.5.7 Linear Filter Warps

When a 1-D filter's impulse response is widened by a factor a, its amplitude must be scaled by $1/a$ to preserve unit area under the filter. Thus, widening $h(x)$ by factor a yields $h(x/a)/a$. The analogous rule for broadening a multidimensional filter by a transformation matrix $\mathbf{A}$ is $|\mathbf{A}|^{-1}h(\mathbf{x}\mathbf{A}^{-1})$.

When a filter's impulse response is widened, its frequency response narrows. For the transform pair $h(x) \leftrightarrow H(\omega)$, broadening by a gives: $h(x/a)/a \leftrightarrow H(a\omega)$. To find the multidimensional generalization, we simply evaluate a Fourier transform:

$$|\mathbf{A}|^{-1}h(\mathbf{x}\mathbf{A}^{-1}) \qquad \leftrightarrow \qquad \int_{-\infty}^{+\infty} |\mathbf{A}|^{-1}h(\mathbf{x}\mathbf{A}^{-1})e^{-j\mathbf{x}\omega^T}\,d\mathbf{x}$$

where $\boldsymbol{\omega}$ is the frequency vector (ω_x, ω_y). Substituting $\mathbf{x} = \mathbf{y}\mathbf{A}$ yields

$$|\mathbf{A}|^{-1}\int_{-\infty}^{+\infty} h(\mathbf{y})e^{-j\mathbf{y}\mathbf{A}\omega^T}|\mathbf{A}|\,d\mathbf{y} = \int_{-\infty}^{+\infty} h(\mathbf{y})e^{-j\mathbf{y}(\omega\mathbf{A}^T)^T}\,d\mathbf{y}$$

so

$$|\mathbf{A}|^{-1}h(\mathbf{x}\mathbf{A}^{-1}) \qquad \leftrightarrow \qquad H(\omega\mathbf{A}^T)$$

3.5.8 Elliptical Gaussian Filters

We can avoid the problematic transitions between decimation and magnification filters if we restrict ourselves to prefilters and reconstruction filters with elliptical regions and Gaussian profile (figure 3.19). Elliptical filter regions are a natural choice for resampling, since the ideal resampling filter can be approximated using a mapping's local affine approximation, and ellipses are closed under affine mappings. These *elliptical Gaussian filters* have a number of useful properties for image resampling.

As discussed in appendix B, the equation $Ax^2+Bxy+Cy^2 = F$ describes a family of concentric ellipses, so we can construct filters of elliptical region and Gaussian profile using an impulse response of the form:

$$\alpha e^{-(Ax^2+Bxy+Cy^2)}$$

The derivation is simplest if we define elliptical Gaussians as warps of circular Gaussians. Using quadratic forms and other matrix notation (summarized in §B.1) simplifies the algebra considerably.

Definition

A circular Gaussian of unit variance is:

$$g_{\mathbf{I}}(\mathbf{u}) = \frac{1}{2\pi}e^{-\frac{1}{2}\mathbf{u}\mathbf{u}^T} = \frac{1}{2\pi}e^{-\frac{1}{2}(u^2+v^2)}$$

Figure 3.19: *Impulse response of an elliptical Gaussian filter.*

where $\mathbf{u} = (u, v)$.

If we apply a warp $\mathbf{x} = \mathbf{uM}$, where $\mathbf{M}$ is a 2×2 matrix, then by the warping law, the warped filter is:

$$|\mathbf{M}|^{-1} g_{\mathbf{I}}(\mathbf{xM}^{-1}) = \frac{1}{2\pi|\mathbf{M}|} e^{-\frac{1}{2}\mathbf{xM}^{-1}\mathbf{M}^{-1T}\mathbf{x}^T} = \frac{1}{2\pi|\mathbf{M}|} e^{-\frac{1}{2}\mathbf{x}(\mathbf{M}^T\mathbf{M})^{-1}\mathbf{x}^T}$$

We define the impulse response of the elliptical Gaussian filter with variance matrix $\mathbf{V}$ as:

$$g_{\mathbf{V}}(\mathbf{x}) = \frac{1}{2\pi|\mathbf{V}|^{1/2}} e^{-\frac{1}{2}\mathbf{x}\mathbf{V}^{-1}\mathbf{x}^T} = \frac{1}{2\pi\sqrt{AC - B^2/4}} e^{-\frac{1}{2}\begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix}^{-1} \begin{pmatrix} x \\ y \end{pmatrix}}$$

where

$$\mathbf{V} = \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix}$$

Then the warped filter can be written:

$$|\mathbf{M}|^{-1} g_{\mathbf{I}}(\mathbf{xM}^{-1}) = g_{\mathbf{M}^T\mathbf{M}}(\mathbf{x})$$

Functions of this form arise in statistics, where they are called *multivariate Gaussian density* functions, and are used to model the joint probability distribution of correlated random variables [Rao73].

Variance Matrix of an Elliptical Gaussian

Note that for all elliptical Gaussian filters constructed this way, $\mathbf{V}$ is symmetric and $|\mathbf{V}| = AC - B^2/4 = |\mathbf{M}|^2 > 0$. The converse is true as well: for any nonsingular, symmetric matrix $\mathbf{V}$, there is an elliptical Gaussian filter, since any such matrix can be decomposed into the form $\mathbf{V} = \mathbf{M}^T\mathbf{M}$ (see §B.7). We call $\mathbf{V}$ the variance matrix since it plays a role analogous to the scalar variance of a 1-D Gaussian. The eigenvalues of $\mathbf{V}$ are the variances along the principal axes (major and minor axes) of the ellipse and the eigenvectors are the principal axes. If $|\mathbf{V}|$ were zero or negative then the filter's regions would not be ellipses but parabolas or hyperbolas, respectively.

Frequency Response of a Circular Gaussian

The Fourier transform of a unit variance circular Gaussian filter is:

$$\begin{aligned} g_{\mathbf{I}}(\mathbf{x}) \qquad \leftrightarrow \qquad G_{\mathbf{I}}(\omega) &= \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-\frac{1}{2}\mathbf{x}\mathbf{x}^T} e^{-j\mathbf{x}\omega^T}\, d\mathbf{x} \\ &= \frac{1}{2\pi}\int_{-\infty}^{+\infty} e^{-\frac{1}{2}(\mathbf{x}+j\omega)(\mathbf{x}+j\omega)^T - \frac{1}{2}\omega\omega^T}\, d\mathbf{x} \\ &= \frac{e^{-\frac{1}{2}\omega\omega^T}}{2\pi}\int_{-\infty}^{+\infty} e^{-\frac{1}{2}\mathbf{y}\mathbf{y}^T}\, d\mathbf{y} \qquad \text{where } \mathbf{y} = \mathbf{x} + j\omega \end{aligned}$$

but

$$\int_{-\infty}^{+\infty} e^{-\frac{1}{2}\mathbf{y}\mathbf{y}^T}\, d\mathbf{y} = \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} e^{-\frac{1}{2}(x^2+y^2)}\, dx\, dy = 2\pi$$

so the Fourier transform of a unit variance circular Gaussian is a scaled unit variance circular Gaussian:

$$G_{\mathbf{I}}(\omega) = e^{-\frac{1}{2}\omega\omega^T} = 2\pi g_{\mathbf{I}}(\boldsymbol{\omega})$$

Frequency Response of an Elliptical Gaussian

We can find the frequency response of an elliptical Gaussian from that of a circular Gaussian using the warping properties of the Fourier transform. Since $|\mathbf{A}|^{-1}h(\mathbf{x}\mathbf{A}^{-1}) \leftrightarrow H(\omega\mathbf{A}^T)$, we have:

$$|\mathbf{M}|^{-1} g_{\mathbf{I}}(\mathbf{x}\mathbf{M}^{-1}) \qquad \leftrightarrow \qquad G_{\mathbf{I}}(\omega\mathbf{M}^T) = e^{-\frac{1}{2}\omega\mathbf{M}^T\mathbf{M}\omega^T}$$

If we define

$$G_{\mathbf{V}}(\omega) = e^{-\frac{1}{2}\omega\mathbf{V}\omega^T}$$

then the transform pair can be rewritten:

$$g_{\mathbf{M}^T\mathbf{M}}(\mathbf{x}) \qquad \leftrightarrow \qquad G_{\mathbf{M}^T\mathbf{M}}(\omega)$$

Summarizing, the frequency response of an elliptical Gaussian with any nonsingular, symmetric variance matrix $\mathbf{V}$ is a scaled elliptical Gaussian with variance matrix $\mathbf{V}^{-1}$:

$$\frac{1}{2\pi|\mathbf{V}|^{1/2}} e^{-\frac{1}{2}\mathbf{x}\mathbf{V}^{-1}\mathbf{x}^T} = g_{\mathbf{V}}(\mathbf{x}) \qquad \leftrightarrow \qquad G_{\mathbf{V}}(\omega) = e^{-\frac{1}{2}\omega\mathbf{V}\omega^T} = 2\pi|\mathbf{V}|^{1/2} g_{\mathbf{V}^{-1}}(\boldsymbol{\omega})$$

Written in scalar notation,

$$\frac{1}{2\pi\sqrt{AC - B^2/4}}\, e^{-\frac{1}{2}(Cx^2 - Bxy + Ay^2)/(AC - B^2/4)} \qquad \leftrightarrow \qquad e^{-\frac{1}{2}(A\omega_x^2 + B\omega_x\omega_y + C\omega_y^2)}$$

Convolution

Elliptical Gaussians are closed under convolution. This is most easily proven in the frequency domain:

$$\begin{aligned} g_{\mathbf{V}_1}(\mathbf{x}) \circledast g_{\mathbf{V}_2}(\mathbf{x}) \quad \leftrightarrow \quad G_{\mathbf{V}_1}(\boldsymbol{\omega})G_{\mathbf{V}_2}(\omega) &= e^{-\frac{1}{2}\omega\mathbf{V}_1\omega^T} e^{-\frac{1}{2}\omega\mathbf{V}_2\omega^T} \\ &= e^{-\frac{1}{2}\omega(\mathbf{V}_1+\mathbf{V}_2)\omega^T} \\ &= G_{\mathbf{V}_1+\mathbf{V}_2}(\omega) \end{aligned}$$

so

$$g_{\mathbf{V}_1}(\mathbf{x}) \circledast g_{\mathbf{V}_2}(\mathbf{x}) = g_{\mathbf{V}_1+\mathbf{V}_2}(\mathbf{x})$$

Thus, variances add when elliptical Gaussians are convolved, as in 1-D.

Linear Warps

Any linear warp $\mathbf{y} = \mathbf{xK}$ of an elliptical Gaussian yields another elliptical Gaussian:

$$\begin{aligned}
|\mathbf{K}|^{-1} g_{\mathbf{V}}(\mathbf{y}\mathbf{K}^{-1}) &= \frac{1}{|\mathbf{K}| 2\pi |\mathbf{V}|^{1/2}} \, e^{-\frac{1}{2}(\mathbf{y}\mathbf{K}^{-1})\mathbf{V}^{-1}(\mathbf{y}\mathbf{K}^{-1})^T} \\
&= \frac{1}{2\pi |\mathbf{K}^T|^{1/2} |\mathbf{V}|^{1/2} |\mathbf{K}|^{1/2}} \, e^{-\frac{1}{2}\mathbf{y}\mathbf{K}^{-1}\mathbf{V}^{-1}\mathbf{K}^{-1T}\mathbf{y}^T} \\
&= \frac{1}{2\pi |\mathbf{K}^T\mathbf{V}\mathbf{K}|^{1/2}} \, e^{-\frac{1}{2}\mathbf{y}(\mathbf{K}^T\mathbf{V}\mathbf{K})^{-1}\mathbf{y}^T} \\
&= g_{\mathbf{K}^T\mathbf{V}\mathbf{K}}(\mathbf{y})
\end{aligned}$$

Elliptical Gaussian Resampling

If we use elliptical Gaussian filters as the reconstruction filter and prefilter, and employ the local affine approximation to the mapping, then the resampling filter will be an elliptical Gaussian. Since elliptical Gaussians are closed under convolution and linear warps, the resampling formula is easily evaluated. For affine warps, the resampling filter is:

$$\rho(\mathbf{x}, \mathbf{k}) = \rho'(\mathbf{m}^{-1}(\mathbf{x}) - \mathbf{k}) \qquad \text{where } \rho' = h' \circledast r \text{ and } h'(\mathbf{u}) = |\mathbf{J}| h(\mathbf{uJ})$$

In matrix notation, the mapping is $\mathbf{x} = \mathbf{m}(\mathbf{u}) = \mathbf{uJ} + \mathbf{c}$. If the reconstruction filter and prefilter are elliptical Gaussians $r(\mathbf{u}) = g_{\mathbf{V}_r}(\mathbf{u})$ and $h(\mathbf{x}) = g_{\mathbf{V}_h}(\mathbf{x})$ then the warped prefilter is an elliptical Gaussian:

$$h'(\mathbf{u}) \;=\; |\mathbf{J}| h(\mathbf{uJ}) \;=\; |\mathbf{J}| g_{\mathbf{V}_h}(\mathbf{uJ}) \;=\; g_{\mathbf{J}^{-1T}\mathbf{V}_h\mathbf{J}^{-1}}(\mathbf{u})$$

as is the entire resampling filter:

$$\rho'(\mathbf{u}) \;=\; (h' \circledast r)(\mathbf{u}) \;=\; g_{\mathbf{J}^{-1T}\mathbf{V}_h\mathbf{J}^{-1}}(\mathbf{u}) \circledast g_{\mathbf{V}_r}(\mathbf{u}) \;=\; g_{\mathbf{J}^{-1T}\mathbf{V}_h\mathbf{J}^{-1}+\mathbf{V}_r}(\mathbf{u})$$

The result is a simple approximation to the ideal resampling filter. We can easily verify that the filter is shaped appropriately at all scales. The determinant of the transformation matrix $\mathbf{J}$ measures the scale of the mapping. When $|\mathbf{J}| < 1$, we are decimating, and $|\mathbf{J}^{-1}|$ is large, so $\mathbf{V}_h$ is scaled up to dominate the variance matrix. When $|\mathbf{J}| > 1$, we are magnifying, and $|\mathbf{J}^{-1}|$ is small, so $\mathbf{V}_h$ is scaled down, and $\mathbf{V}_r$ dominates.

This unified scheme for reconstruction and prefiltering guarantees that the resampling filter is always wider than both the reconstruction and prefilters, so there is no danger that the filter will fall between samples.

To implement elliptical Gaussian filters in practice, we truncate the Gaussian to give it a finite impulse response. Typically, the ideal reconstruction filter and prefilter both have identity variance, so $\mathbf{V}_r = \mathbf{V}_h = \mathbf{I}$, and $\rho'(\mathbf{u}) = g_{\mathbf{J}^{-1T}\mathbf{J}^{-1}+\mathbf{I}}(\mathbf{u})$.

In summary, the closure property of ellipses under affine warps suggests their appropriateness as regions for resampling filters. Elliptical Gaussian filters are closed under affine warps and convolution,[2] so they form a convenient class of filters for image resampling. Elliptical Gaussian filters may be used for non-affine warps by using the local affine approximation to a mapping as determined by the Jacobian at each destination pixel. This is ideal to the extent that the Gaussian is an ideal low pass filter and the mapping is affine.

3.5.9 An Implementation of Elliptical Gaussian Filters

We conclude with an application of resampling filter theory: an implementation of elliptical Gaussian filters.

As mentioned earlier, the preimage through a projective mapping of a circular region in screen space is (usually) an elliptical region in texture space, and the preimage of a circle through an affine mapping is an ellipse. This observation inspired the author to develop the *Elliptical Weighted Average (EWA) Filter* in an earlier work [Greene-Heckbert86]. The original derivation of the EWA filter was based on heuristics, but now, using the theory of resampling filters, we can justify most of those choices and improve on others.

Elliptical Weighted Average Filter

We will first summarize the EWA filter as it was originally conceived (the following is excerpted with minor changes from [Greene-Heckbert86]).

We assume the texture is being resampled by screen order scanning. At each screen pixel we compute the Jacobian of the mapping. We think of this Jacobian as two basis vectors in texture space, with their origin at the preimage of the screen pixel's center. The Jacobian constitutes a local affine approximation to the mapping. The shape of the EWA filter region is then the (arbitrarily oriented) ellipse defined by these two vectors (figure B.2). The filter profile is stored in a lookup table for speed, so it could be arbitrary. Since the EWA filter gives independent control of region and profile, it is an example of filter design by the transformation method [Dudgeon-Mersereau84].

The EWA filter is a direct convolution method, so its cost is proportional to the area of the texture region, like other high quality texture filters [Heckbert86b], but its coefficient of proportionality is lower than most. To speed computation, EWA uses a biquadratic function of the following form as its radial index:

$$Q(u,v) = Au^2 + Buv + Cv^2$$

where $u = 0, v = 0$ are the translated texture coordinates of the center of the ellipse. When $AC - B^2/4 > 0$ and $A > 0$, this function is an elliptical paraboloid, concave upward. By proper choice of the coefficients A, B, and C, the contours of this function can match any ellipse. Points inside the ellipse satisfy $Q(u,v) < F$ for some threshold F, so Q can be used for both ellipse inclusion testing and filter table indexing. In texture space, the contours of Q are concentric ellipses (figure 3.20), but when mapped to screen space, they are nearly circular. We can associate with each elliptical contour the radius r of its corresponding screen space circle. Since Q is quadratic, Q is proportional to r^2. It would be most natural to index a filter profile function by r, but this would

[2] Elliptical Gaussian filters with nonsingular variance matrices form a group under convolution.

Figure 3.20: *Regions of an elliptical Gaussian (or EWA) filter are concentric ellipses. Dots are texture pixel centers. Algorithm scans pixels inside box; only those inside the contour $Q = F$ contribute to the weighted average.*

require the computation of $r = \sqrt{Q}$ at each pixel. To avoid this cost, we pre-warp the filter lookup function $f(r)$ to create a weight lookup table:

$$WTAB[Q] = f(\sqrt{Q})$$

The profile used in the implementation was a Gaussian: $f(r) = e^{-\alpha r^2}$, for which $WTAB[Q] = e^{-\alpha Q}$. A table length of several thousand was used.

To evaluate Q efficiently, the method of finite differences is employed. Since Q is quadratic, two additions suffice to update Q for each pixel. The following pseudocode implements the EWA filter for monochrome pictures; it is easily modified for color. Integer variables are in lower case and floating point variables are in upper case.

```
Let texture[u,v] be a 2-dimensional array holding texture image
< Compute texture space ellipse center (U0,V0) from screen coordinates (x,y) >
< Compute (Ux,Vx) = (u_x,v_x) and (Uy,Vy) = (u_y,v_y) >
Now find ellipse coefficients: (see appendix B)
A = Vx*Vx+Vy*Vy
B = -2*(Ux*Vx+Uy*Vy)
C = Ux*Ux+Uy*Uy
F = (Ux*Vy-Uy*Vx)^2
< scale A, B, C, and F equally so that F = WTAB length >

Ellipse is AU^2+BUV+CV^2 = F, where U = u-U0, V = v-V0

EWA(U0,V0,A,B,C,F)
begin
    < Find bounding box around ellipse: u1 <= u <= u2, v1 <= v <= v2 >
    NUM = 0
    DEN = 0
    DDQ = 2*A
    U = u1-U0
    for v=v1 to v2 do begin                   scan the box
        V = v-V0
        DQ = A*(2*U+1)+B*V                    Q(U+1,V)-Q(U,V)
        Q = (C*V+B*U)*V+A*U*U
        for u=u1 to u2 do begin
            if Q<F then begin                 ignore pixel if Q out of range
                WEIGHT = WTAB[floor(Q)]
                NUM = NUM+WEIGHT*texture[u,v] read and weight texture pixel
                DEN = DEN+WEIGHT              denominator (for normalization)
            end
            Q = Q+DQ
            DQ = DQ+DDQ
        end
    end
    if DEN=0 then return bilinearfilt(U0,V0)  if ellipse fell between pixels
    return NUM/DEN
end
```

The above code requires 1 floating point multiply, 4 floating point adds, 1 integerization, and 1 table lookup per texture pixel. This implementation can be optimized further by removing redundant calculations from the v loop and, with proper checking, by using integer arithmetic throughout.

Higher Quality Elliptical Weighted Average

In retrospect, we recognize that an EWA filter with Gaussian profile is an elliptical Gaussian resampling filter. The coefficients A, B, and C computed in the pseudocode above form a conic matrix $\mathbf{Q}$ (§B.1) whose inverse is the variance matrix of the elliptical Gaussian:

$$\mathbf{V} = \mathbf{Q}^{-1} = \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix}^{-1} = \mathbf{J}^{-1T}\mathbf{J}^{-1} \qquad \text{where } \mathbf{J}^{-1} = \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix}$$

Since there is no term for reconstruction, the old EWA is a decimation filter. In effect, the reconstruction filter is an impulse, and the prefilter a Gaussian. One would expect such a filter to occasionally "fall between the pixels" for scale factors of 1 or greater, and this indeed happens. With the resampling theory presented here, we can correct this flaw. The change is trivial: instead of $\mathbf{V} = \mathbf{J}^{-1T}\mathbf{J}^{-1}$, we use $\mathbf{V} = \mathbf{J}^{-1T}\mathbf{J}^{-1} + \mathbf{I}$. With this change in ellipse shape, the new EWA filter is a unified resampling filter that is, in effect, using Gaussians for both reconstruction and prefiltering. The necessary modifications to the preceding pseudocode are as follows:

```
A = Vx*Vx+Vy*Vy+1
B = -2*(Ux*Vx+Uy*Vy)
C = Ux*Ux+Uy*Uy+1
F = A*C-B*B/4
```

Optimized Elliptical Weighted Average

Depending on the mapping, some screen pixels can map to immense texture regions requiring excessive computation time with the algorithm above. In such cases it is of great benefit to use EWA on an image pyramid [Heckbert86b]. An image pyramid (figure 3.21) is a sequence of images $f_0(x,y)\ldots f_K(x,y)$ whose base level image f_0 is the original texture image of resolution $M \times N$ pixels, and each pyramid level f_k is made by decimating the level below, f_{k-1}, by a factor of two [Williams83], [Rosenfeld84]. The resolution of image f_k is thus $M/2^k \times N/2^k$, and there will be $K = \log_2(\min(M,N)) + 1$ pyramid levels. To filter a circle of diameter d using an image pyramid we choose a level in which the circle is small (but not so small that accuracy is lost). A reasonable criterion is to choose the pyramid level in which the diameter is at least 3 and less than 6 pixels. Using a pyramid filtering in level k is thus 4^k times faster than filtering in level 0 (filtering without a pyramid). Using the level selection rule above, we can filter any circle with a minimum of 9 and a maximum of 25 pixel accesses, independent of the size of the circle in the original texture image.

Filtering elliptical regions isn't as simple as filtering circular regions. An infinite checkered plane in perspective (figure 3.2) is a particularly difficult test scene, since the image contains unlimited high frequencies at many orientations. Pixels near the horizon map to very eccentric ellipses, some of which are 5 pixels wide and 5000 pixels long in images of this sort. We define eccentricity to be $e =$ (major axis)/(minor axis). If the pyramid level selection rule chooses the level in which the minor axis (the smallest diameter) of the ellipse is between 3 and 6 pixels, then an ellipse with eccentricity e will lie in a rectangle of area at least $9e$. Since eccentricities are unbounded for arbitrary mappings, this area is unbounded, so the cost of EWA filtering of ellipses is unbounded. For large eccentricities (say > 50) the algorithm will run very slowly. Note that it is unacceptable

Figure 3.21: *3-D image pyramid.*

Figure 3.22: *4-D image pyramid.*

to choose the pyramid level based on the major axis of the ellipse, since this could scale down the ellipse so much that the minor axis of the ellipse becomes less than one pixel, producing inaccurate results.

One approach to speed filtering of eccentric texture regions is the expansion of the image pyramid to contain a greater variety of pre-warped images. A standard (3-D) image pyramid can be generalized to 4-D by scaling an image differentially in x and y, as shown in figure 3.22. While a 3-D image pyramid filters using square regions, a 4-D pyramid filters using axis-aligned rectangles. A 3-D pyramid is indexed by three parameters: x, y, and $scale$, while a 4-D pyramid is indexed by four: x, y, $xscale$, and $yscale$. One can efficiently filter an axis-aligned rectangle with a 4-D pyramid, but for arbitrarily oriented ellipses the same problems remain as for a 3-D pyramid. Arbitrarily oriented ellipses could be filtered efficiently if the pyramid were expanded even further, yielding a 5-D *angled pyramid*, which contains an image filtered with both differential scaling and rotation. Angled pyramids are indexed by the five parameters of an ellipse: x, y, major radius a, minor radius b, and angle θ (figure B.1). Experimentation with nonstandard pyramids is an area of active research [Heckbert86b]; it is not yet known if their time benefits relative to standard pyramids compensate for their increased storage costs.

Even if we restrict ourselves to standard pyramids, other optimizations are possible for speeding the filtering of eccentric ellipses. If we are willing to sacrifice quality then we can simply clamp the eccentricities to some limit e_{max}. Ellipses can be rounded by either fattening, shortening, or both. Fattening the ellipse (increasing its minor axis) will cause lateral blurring, and shortening the ellipse (decreasing its major axis) will cause longitudinal aliasing. It is probably best to opt for blurring. Tests show that a threshold of $e_{max} = 20$ or 30 produces very good results. Note that clamping the eccentricity of the EWA filter bounds the computation required for one filtering operation, thereby making EWA a constant cost filter. Figure 3.23 contrasts the results for two eccentricity thresholds.

Eccentricity clamping is achieved in the implementation as follows. From the variance matrix containing the coefficients A, B, C of the ellipse's implicit equation we can find the major and minor radii, a and b, of the ellipse. If the eccentricity $e = a/b$ exceeds a threshold eccentricity e_{max} then we fatten the ellipse by modifying its minor radius: $b' = a/e_{max}$, and compute a new variance matrix. Equations for converting the implicit formula for an ellipse into its radii and angle, and vice versa, are given in §B.3 and §B.4.

3.6 Future Work

There are a number of areas for future research. As mentioned, antialiasing prefilters are usually implemented using FIR designs. Perhaps IIR filters would make more efficient prefilters and resampling filters.

We have seen how the theory of resampling filters can aid the development of high quality, efficient, space variant filters for texture mapping and image warping. The theory can hopefully help us design even higher quality, more efficient filters in the future. We have pursued the elliptical Gaussian here because its closure properties make it an elegant resampling filter. There are, however, a number of 1-D filters of higher quality than the Gaussian [Hamming83], [Turkowski88].

We would like to find less drastic measures than eccentricity clamping to bound the expense of future resampling filters. Highly nonlinear mappings can lead to concave filter regions [Fournier-

Figure 3.23: *Top: EWA filter with eccentricity clamped to 1 (a circular filter). Note lateral blurring along vertical edges. Bottom: EWA filter with eccentricity clamped to 16 or less. Note improved resolution of diagonal frequencies (horizon at right).*

Fiume88], which would be poorly filtered using a local affine approximation, so we should also find rules to determine when the local affine approximation is acceptable, and when it is not.

Conclusions

We have seen that the fundamentals of texture mapping and image warping have much in common. Both applications require (1) the description of a mapping between a source image and a destination image, and (2) resampling of the source image to create the destination image according to a mapping function. The first task is a geometric modeling problem, and the latter is a rendering and image processing problem.

We have demonstrated the desirability of certain properties in a mapping: preservation of parallel lines, straight lines, and equispaced points, and closure under composition and inversion. We have found that affine mappings, while being the least general of the mapping classes considered, enjoy all of these properties, while projective mappings have more generality and yet retain most of these properties. Projective mappings are needed for texture mapping if a perspective camera transformation is used. We have shown that projective warps can be performed very efficiently using rational linear interpolation.

To improve the quality of rendering for texture mapping and image warping, we have developed a new theory of ideal image resampling. This theory describes the filter shape needed for perfect antialiasing during the resampling implied by an arbitrary mapping. We have explored one class of filter that conforms nicely to this theory, the elliptical Gaussian, but there are undoubtedly others. In addition to assisting the design of higher quality, more efficient filters, this theory should provide a standard against which existing filter schemes can be compared.

Appendix A

Source Code for Mapping Inference

Source code to infer projective mappings from a four point 2-D correspondence, and to infer affine mappings from a three point 3-D correspondence, is given below in the C programming language.

A.1 Basic Data Structures and Routines

Below are listed the polygon and vertex data types, and some generic routines for manipulating 3×3 matrices.

A.1.1 poly.h

```
#define POLY_NMAX 8                     /* max #sides to a polygon */

typedef struct {                        /* A POLYGON VERTEX */
    double u, v;                        /* texture space coords */
    double sx, sy;                      /* screen space coords */
    double x, y, z;                     /* object space coords */
} poly_vert;

typedef struct {                        /* A POLYGON */
    int n;                              /* number of sides */
    poly_vert vert[POLY_NMAX];          /* vertices */
} poly;
```

A.1.2 pmap.h

```
/* pmap return codes */
#define PMAP_BAD       -1     /* mapping doesn't exist (bad correspondence) */
#define PMAP_AFFINE     0     /* mapping is affine */
#define PMAP_PROJECTIVE 1     /* mapping is projective */
```

A.1.3 mx3.h

```
double  mx3d_adjoint();

/*  |a b|
 *  |c d|
 */
#define DET2(a,b, c,d) ((a)*(d) - (b)*(c))
```

A.1.4 mx3.c

```
/*
 * mx3.c: routines for manipulating 3x3 matrices
 * These matrices are commonly used to represent a 2-D projective mapping.
 */

#include <math.h>
#include "mx3.h"

/* mx3_adjoint: b = adjoint(a), returns determinant(a) */

double mx3d_adjoint(a, b)
double a[3][3], b[3][3];
{
    b[0][0] = DET2(a[1][1], a[1][2], a[2][1], a[2][2]);
    b[1][0] = DET2(a[1][2], a[1][0], a[2][2], a[2][0]);
    b[2][0] = DET2(a[1][0], a[1][1], a[2][0], a[2][1]);
    b[0][1] = DET2(a[2][1], a[2][2], a[0][1], a[0][2]);
    b[1][1] = DET2(a[2][2], a[2][0], a[0][2], a[0][0]);
    b[2][1] = DET2(a[2][0], a[2][1], a[0][0], a[0][1]);
    b[0][2] = DET2(a[0][1], a[0][2], a[1][1], a[1][2]);
    b[1][2] = DET2(a[0][2], a[0][0], a[1][2], a[1][0]);
    b[2][2] = DET2(a[0][0], a[0][1], a[1][0], a[1][1]);
    return a[0][0]*b[0][0] + a[0][1]*b[0][1] + a[0][2]*b[0][2];
}

/* mx3_mul: matrix multiply: c = a*b */

mx3d_mul(a, b, c)
double a[3][3], b[3][3], c[3][3];
{
    c[0][0] = a[0][0]*b[0][0] + a[0][1]*b[1][0] + a[0][2]*b[2][0];
    c[0][1] = a[0][0]*b[0][1] + a[0][1]*b[1][1] + a[0][2]*b[2][1];
    c[0][2] = a[0][0]*b[0][2] + a[0][1]*b[1][2] + a[0][2]*b[2][2];
    c[1][0] = a[1][0]*b[0][0] + a[1][1]*b[1][0] + a[1][2]*b[2][0];
    c[1][1] = a[1][0]*b[0][1] + a[1][1]*b[1][1] + a[1][2]*b[2][1];
    c[1][2] = a[1][0]*b[0][2] + a[1][1]*b[1][2] + a[1][2]*b[2][2];
    c[2][0] = a[2][0]*b[0][0] + a[2][1]*b[1][0] + a[2][2]*b[2][0];
    c[2][1] = a[2][0]*b[0][1] + a[2][1]*b[1][1] + a[2][2]*b[2][1];
    c[2][2] = a[2][0]*b[0][2] + a[2][1]*b[1][2] + a[2][2]*b[2][2];
}
```

```
/* mx3d_transform: transform point p by matrix a: q = p*a */

mx3d_transform(p, a, q)
double p[3], q[3];
double a[3][3];
{
    q[0] = p[0]*a[0][0] + p[1]*a[1][0] + p[2]*a[2][0];
    q[1] = p[0]*a[0][1] + p[1]*a[1][1] + p[2]*a[2][1];
    q[2] = p[0]*a[0][2] + p[1]*a[1][2] + p[2]*a[2][2];
}
```

A.2 Inferring Projective Mappings from 2-D Quadrilaterals

In texture mapping applications when the camera transformation (object space to screen space transform OS) is not known, or in image warping applications with no intermediate 3-D object space, a projective warp can be inferred from the screen and texture coordinates of the four vertices of a quadrilateral (§2.2.3). (We use the terms "screen space" and "texture space" here, but this code applies to general 2-D quadrilateral mappings.)

Below is a table listing computational cost of the critical subroutines, measured by the counts of divides, multiplies, and adds.

OPERATION ⇒ ⇓ SUBROUTINE	/	×	+
`pmap_quad_quad` (double projective case)	4	65	69
`pmap_square_quad` (affine case)	0	0	10
`pmap_square_quad` (projective case)	2	10	21
`mx3_adjoint`	0	18	9
`mx3_mul`	0	27	18

Following is a routine *pmap_poly* to find the projective mapping from a four point correspondence.

A.2.1 pmap_poly.c

```
#include "poly.h"
#include "pmap.h"
#include "mx3.h"

#define X(i) p->vert[i].sx      /* screen coordinates */
#define Y(i) p->vert[i].sy

#define U(i) p->vert[i].u       /* texture coordinates */
#define V(i) p->vert[i].v

/*
 * pmap_poly: find the projective mapping ST from screen to texture space
```

```
 * given the screen and texture coordinates at the vertices of a polygon,
 * which are passed in the fields (sx,sy) and (u,v) of p->vert[i], respectively.
 */

pmap_poly(p, ST)
poly *p;
double ST[3][3];
{
    double scr[4][2];                   /* vertices of screen quadrilateral */

    if (p->n!=4)
        ERROR("only do quadrilaterals at the moment\n");

    /* if edges 0-1 and 2-3 are horz, 1-2 and 3-0 are vert */
    if (V(0)==V(1) && V(2)==V(3) && U(1)==U(2) && U(3)==U(0)) {
        scr[0][0] = X(0); scr[0][1] = Y(0);
        scr[1][0] = X(1); scr[1][1] = Y(1);
        scr[2][0] = X(2); scr[2][1] = Y(2);
        scr[3][0] = X(3); scr[3][1] = Y(3);
        return pmap_quad_rect(U(0), V(0), U(2), V(2), scr, ST);
    }

    /* if edges 0-1 and 2-3 are vert, 1-2 and 3-0 are horz */
    else if (U(0)==U(1) && U(2)==U(3) && V(1)==V(2) && V(3)==V(0)) {
        scr[0][0] = X(1); scr[0][1] = Y(1);
        scr[1][0] = X(2); scr[1][1] = Y(2);
        scr[2][0] = X(3); scr[2][1] = Y(3);
        scr[3][0] = X(0); scr[3][1] = Y(0);
        return pmap_quad_rect(U(1), V(1), U(3), V(3), scr, ST);
    }

    /* if texture is not an orthogonally-oriented rectangle */
    else return pmap_quad_quad(p, ST);
}

/*
 * pmap_quad_quad: find the projective mapping ST from screen to texture space
 * given the screen and texture coordinates at the vertices of a quadrilateral.
 *
 * Method: concatenate screen_quad->mid_square and
 * mid_square->texture_quad transform matrices.
 * The alternate method, solving an 8x8 system of equations, takes longer.
 */

pmap_quad_quad(p, ST)
poly *p;
double ST[3][3];
{
    int type1, type2;
    double quad[4][2], MS[3][3];
    double SM[3][3], MT[3][3];          /* screen->mid and mid->texture */

    quad[0][0] = X(0); quad[0][1] = Y(0);
    quad[1][0] = X(1); quad[1][1] = Y(1);
```

```
    quad[2][0] = X(2); quad[2][1] = Y(2);
    quad[3][0] = X(3); quad[3][1] = Y(3);
    type1 = pmap_square_quad(quad, MS);
    if (mx3d_adjoint(MS, SM) == 0.)
        ERROR("pmap_quad_quad: warning: determinant=0\n");

    quad[0][0] = U(0); quad[0][1] = V(0);
    quad[1][0] = U(1); quad[1][1] = V(1);
    quad[2][0] = U(2); quad[2][1] = V(2);
    quad[3][0] = U(3); quad[3][1] = V(3);
    type2 = pmap_square_quad(quad, MT);

    if (type1==PMAP_BAD || type2==PMAP_BAD) return PMAP_BAD;

    mx3d_mul(SM, MT, ST);

    return type1|type2;                 /* PMAP_PROJECTIVE prevails */
}
```

A.2.2 pmap_gen.c

```
/*
 * pmap_gen.c: general routines for 2-D projective mappings.
 * These routines are independent of the poly structure,
 * so we do not think in terms of texture and screen space.
 */

#include <stdio.h>
#include "pmap.h"
#include "mx3.h"

#define TOLERANCE 1e-13
#define ZERO(x) ((x)<TOLERANCE && (x)>-TOLERANCE)

#define X(i) quad[i][0]         /* quadrilateral x and y */
#define Y(i) quad[i][1]

/*
 * pmap_quad_rect: find mapping between quadrilateral and rectangle.
 * The correspondence is:
 *
 *      quad[0] --> (u0,v0)
 *      quad[1] --> (u1,v0)
 *      quad[2] --> (u1,v1)
 *      quad[3] --> (u0,v1)
 *
 * This method of computing the adjoint numerically is cheaper than
 * computing it symbolically.
 */

pmap_quad_rect(u0, v0, u1, v1, quad, QR)
double u0, v0, u1, v1;          /* bounds of rectangle */
double quad[4][2];              /* vertices of quadrilateral */
```

```
double QR[3][3];                /* quad->rect transform (returned) */
{
    int ret;
    double du, dv;
    double RQ[3][3];            /* rect->quad transform */

    du = u1-u0;
    dv = v1-v0;
    if (du==0. || dv==0.) {
        ERROR("pmap_quad_rect: null rectangle\n");
        return PMAP_BAD;
    }

    /* first find mapping from unit uv square to xy quadrilateral */
    ret = pmap_square_quad(quad, RQ);
    if (ret==PMAP_BAD) return PMAP_BAD;

    /* concatenate transform from uv rectangle (u0,v0,u1,v1) to unit square */
    RQ[0][0] /= du;
    RQ[1][0] /= dv;
    RQ[2][0] -= RQ[0][0]*u0 + RQ[1][0]*v0;
    RQ[0][1] /= du;
    RQ[1][1] /= dv;
    RQ[2][1] -= RQ[0][1]*u0 + RQ[1][1]*v0;
    RQ[0][2] /= du;
    RQ[1][2] /= dv;
    RQ[2][2] -= RQ[0][2]*u0 + RQ[1][2]*v0;

    /* now RQ is transform from uv rectangle to xy quadrilateral */
    /* QR = inverse transform, which maps xy to uv */
    if (mx3d_adjoint(RQ, QR)==0.)
        ERROR("pmap_quad_rect: warning: determinant=0\n");
    return ret;
}

/*
 * pmap_square_quad: find mapping between unit square and quadrilateral.
 * The correspondence is:
 *
 *      (0,0) --> quad[0]
 *      (1,0) --> quad[1]
 *      (1,1) --> quad[2]
 *      (0,1) --> quad[3]
 */

pmap_square_quad(quad, SQ)
double quad[4][2];      /* vertices of quadrilateral */
double SQ[3][3];        /* square->quad transform */
{
    double px, py;

    px = X(0)-X(1)+X(2)-X(3);
    py = Y(0)-Y(1)+Y(2)-Y(3);
```

```
    if (ZERO(px) && ZERO(py)) {         /* affine */
        SQ[0][0] = X(1)-X(0);
        SQ[1][0] = X(2)-X(1);
        SQ[2][0] = X(0);
        SQ[0][1] = Y(1)-Y(0);
        SQ[1][1] = Y(2)-Y(1);
        SQ[2][1] = Y(0);
        SQ[0][2] = 0.;
        SQ[1][2] = 0.;
        SQ[2][2] = 1.;
        return PMAP_AFFINE;
    }
    else {                              /* projective */
        double dx1, dx2, dy1, dy2, del;

        dx1 = X(1)-X(2);
        dx2 = X(3)-X(2);
        dy1 = Y(1)-Y(2);
        dy2 = Y(3)-Y(2);
        del = DET2(dx1,dx2, dy1,dy2);
        if (del==0.) {
            ERROR("pmap_square_quad: bad mapping\n");
            return PMAP_BAD;
        }
        SQ[0][2] = DET2(px,dx2, py,dy2)/del;
        SQ[1][2] = DET2(dx1,px, dy1,py)/del;
        SQ[2][2] = 1.;
        SQ[0][0] = X(1)-X(0)+SQ[0][2]*X(1);
        SQ[1][0] = X(3)-X(0)+SQ[1][2]*X(3);
        SQ[2][0] = X(0);
        SQ[0][1] = Y(1)-Y(0)+SQ[0][2]*Y(1);
        SQ[1][1] = Y(3)-Y(0)+SQ[1][2]*Y(3);
        SQ[2][1] = Y(0);
        return PMAP_PROJECTIVE;
    }
}
```

A.2.3 mapping_example.c

```
/* example use of pmap_poly to infer projective mapping from a quadrilateral */

#include "poly.h"
#define SIZE 50

main()
{
    static poly p = {
        4,
      /* u  v  sx  sy */
        {0, 0,  0,  0}, /* vert[0] */
        {1, 0, 40,  0}, /* vert[1] */
        {1, 1, 30, 30}, /* vert[2] */
        {0, 1, 10, 20}  /* vert[3] */
```

```
    };
    double ST[3][3];
    double scr[3], tex[3];
    int x, y;

    /* compute screen to texture transform ST */
    pmap_poly(&p, ST);

    /* scan a square area of screen space, transforming to texture space */
    for (y=0; y<SIZE; y++)
        for (x=0; x<SIZE; x++) {
            scr[0] = x;
            scr[1] = y;
            scr[2] = 1.;
            mx3d_transform(scr, ST, tex);
            tex[0] /= tex[2];
            tex[1] /= tex[2];
            printf("scr(%d,%d) transforms to tex(%g,%g)\n",
                x, y, tex[0], tex[1]);
        }
}
```

A.3 Inferring Affine Parameterizations from 3-D Polygons

Following is a routine *pmap_param* that computes the 3×4 texture space to object space transformation matrix (TO) using the texture space and object space correspondence at three selected vertices of a polygon. When concatenated with the 4×3 object space to screen space matrix OS, the 3×3 texture to screen transform TS results. The adjoint of this matrix gives the screen to texture transform ST.

A 3×4 homogeneous matrix form for affine parameterizations is used rather than the 3×3 form discussed in section §2.3.1 for greater compatibility with the 4×4 matrices used elsewhere in a typical 3-D modeling system. This matrix TO transforms from homogeneous 2-D texture points to homogeneous 3-D object space points:

$$(\, x' \quad y' \quad z' \quad w \,) = (\, u \quad v \quad 1 \,) \begin{pmatrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & l \end{pmatrix}$$

A.3.1 pmap_param.c

```
#include <math.h>
#include "mx3.h"
#include "poly.h"
#define MINDEL 1e-7

/*
 * pmap_param: find 3x4 affine parameterization that maps texture space
 * to object space.  The texture and object space coordinates of each
 * vertex are passed in the fields (u,v) and (x,y,z) of g->vert[i], resp.
```

```
 */

pmap_param(g, TO)
poly *g;
double TO[3][4];        /* texture to object transform */
{
    int i;
    double del, pu, pv, qu, qv, p, q, am, bm, cm;
    poly_vert *a, *b, *c;

    /* find three consecutive vertices with noncollinear texture coords */
    a = &g->vert[g->n-2];
    b = &g->vert[g->n-1];
    c = &g->vert[0];
    pu = b->u-a->u; pv = b->v-a->v;
    for (i=0; i<g->n; i++, a++, b++, c++, pu=qu, pv=qv) {
        qu = c->u-b->u; qv = c->v-b->v;
        del = pu*qv-pv*qu;
        if (fabs(del)>MINDEL) break;
    }
    if (i>=g->n)
        ERROR("degenerate polygon");

    /*
     * compute texture to object transform from those three vertices.
     * (note that the correspondences at other vertices are ignored;
     * they are assumed to obey the same affine mapping)
     */
    am = DET2(b->u, c->u, b->v, c->v);
    bm = DET2(c->u, a->u, c->v, a->v);
    cm = DET2(a->u, b->u, a->v, b->v);
    p = b->x-a->x;
    q = c->x-b->x;
    TO[0][0] = DET2(p , q , pv, qv);
    TO[1][0] = DET2(pu, qu, p , q );
    TO[2][0] = a->x*am+b->x*bm+c->x*cm;
    p = b->y-a->y;
    q = c->y-b->y;
    TO[0][1] = DET2(p , q , pv, qv);
    TO[1][1] = DET2(pu, qu, p , q );
    TO[2][1] = a->y*am+b->y*bm+c->y*cm;
    p = b->z-a->z;
    q = c->z-b->z;
    TO[0][2] = DET2(p , q , pv, qv);
    TO[1][2] = DET2(pu, qu, p , q );
    TO[2][2] = a->z*am+b->z*bm+c->z*cm;
    TO[0][3] = 0.;
    TO[1][3] = 0.;
    TO[2][3] = del;
}
```

A.3.2 param_example.c

```
/* example use of pmap_param to compute parameterization of a polygon */

#include "poly.h"

main()
{
    static poly p = {
        4,
      /* u  v  sx  sy  x  y  z */
        {0, 0,  0,  0, 4, 2, 6},        /* vert[0] */
        {1, 0, 40,  0, 5, 3, 8},        /* vert[1] */
        {1, 1, 30, 30, 5, 7,12},        /* vert[2] */
        {0, 1, 10, 20, 4, 6,10}         /* vert[3] */
        /*
         * note: sx and sy are ignored,
         * This test polygon is a square in texture space (u,v)
         * and a planar parallelogram in object space (x,y,z)
         */
    };
    double TO[3][4];
    double obj[4], u, v;
    int i;

    /* compute texture to object transform TO */
    pmap_param(&p, TO);

    for (i=0; i<p.n; i++) {
        u = p.vert[i].u;
        v = p.vert[i].v;
        obj[0] = u*TO[0][0] + v*TO[1][0] + TO[2][0];
        obj[1] = u*TO[0][1] + v*TO[1][1] + TO[2][1];
        obj[2] = u*TO[0][2] + v*TO[1][2] + TO[2][2];
        obj[3] = u*TO[0][3] + v*TO[1][3] + TO[2][3];
        obj[0] /= obj[3];
        obj[1] /= obj[3];
        obj[2] /= obj[3];
        printf("tex(%g,%g) transforms to obj(%g,%g,%g)\n",
            u, v, obj[0], obj[1], obj[2]);
    }
}
```

Appendix B

Ellipses

Here we summarize the analytic geometry of ellipses. Ellipses can be described in both implicit and parametric forms. The ellipse is one class of conic curve, so most of the general formulas for ellipses apply to other conics (hyperbolas and parabolas) as well.

B.1 Implicit Ellipse

The implicit equation for a conic is a second order polynomial in two variables:

$$Ax^2 + Bxy + Cy^2 + Dx + Ey - F = 0$$

Since this equation has six coefficients, any nonzero scalar multiple of which is equivalent, a conic has five degrees of freedom. We can assume without loss of generality that $A \geq 0$. Conics can be classified according to the sign of the *discriminant* $\Delta = AC - B^2/4$. The conic is an ellipse when $\Delta > 0$, a parabola if $\Delta = 0$, and a hyperbola if $\Delta < 0$.

We will deal exclusively with ellipses (conics) centered on the origin, in which case $D = E = 0$, and we have the *canonical conic*:

$$Ax^2 + Bxy + Cy^2 = F$$

A formula that is quadratic in every term in this manner is known as a *quadratic form*. This conic equation can be written in matrix form to facilitate transformations [Roberts66]:

$$\mathbf{p}\mathbf{Q}\mathbf{p}^T = F$$

$$\begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = F$$

where $\mathbf{Q}$ is the *conic matrix*. A conic centered on the origin is uniquely specified to within a scale factor by any 2×2 symmetric matrix $\mathbf{Q}$ and a scalar F. Conics centered on the origin thus have three degrees of freedom. The conic is an ellipse iff both eigenvalues of this matrix are positive (the matrix is positive definite).

Figure B.1: *Ellipse parameterized by major radius a, minor radius b, and angle θ.*

B.2 Transforming an Implicit Ellipse

Given a linear transformation:

$$\begin{aligned} \mathbf{p} &= \mathbf{p}'\mathbf{M} \\ &= \begin{pmatrix} x & y \end{pmatrix} = \begin{pmatrix} x' & y' \end{pmatrix} \begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix} \end{aligned}$$

we can transform the conic $A'x'^2 + B'x'y' + C'y'^2 = F'$. The coefficients of the new conic $Ax^2 + Bxy + Cy^2 = F$ are found by substituting $\mathbf{p}' = \mathbf{p}\mathbf{M}^{-1}$ into the conic equation:

$$\mathbf{p}'\mathbf{Q}'\mathbf{p}'^T = F' \qquad \text{where } \mathbf{Q}' = \begin{pmatrix} A' & B'/2 \\ B'/2 & C' \end{pmatrix}$$

$$\begin{aligned} \mathbf{p}\mathbf{M}^{-1}\mathbf{Q}'\mathbf{M}^{-1^T}\mathbf{p}^T &= F' \\ \mathbf{p}\mathbf{Q}\mathbf{p}^T &= F = F' \end{aligned}$$

where the matrix of the transformed conic is:

$$\mathbf{Q} = \mathbf{M}^{-1}\mathbf{Q}'\mathbf{M}^{-1^T}$$

B.3 Ellipse Analysis by Rotation

An ellipse is most intuitively parameterized by its major radius a, minor radius b, and angle θ (figure B.1). An orthogonal ellipse (with $\theta = 0$) satisfies the familiar equation $(x/a)^2 + (y/b)^2 = 1$, which is an implicit conic equation with no cross term: $A' = 1/a^2$, $B' = 0$, $C' = 1/b^2$, $F' = 1$. We can determine the ellipse parameters a, b, and θ by rotating an arbitrary ellipse until the cross term vanishes.

Starting with the orthogonal ellipse $\mathbf{p}'\mathbf{Q}'\mathbf{p}'^T = F'$ with conic matrix

$$\mathbf{Q}' = \begin{pmatrix} A' & 0 \\ 0 & C' \end{pmatrix}$$

and transforming it by a rotation

$$\mathbf{M} = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

we get the rotated ellipse $\mathbf{p}\mathbf{Q}\mathbf{p}^T = F$ where

$$\mathbf{Q} = \mathbf{M}^{-1}\mathbf{Q}'\mathbf{M}^{-1^T}$$

$$= \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} A' & 0 \\ 0 & C' \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

$$= \begin{pmatrix} A'\cos^2\theta + C'\sin^2\theta & (A'-C')\sin\theta\cos\theta \\ (A'-C')\sin\theta\cos\theta & A'\sin^2\theta + C'\cos^2\theta \end{pmatrix}$$

The rotated ellipse thus has coefficients:

$$\begin{aligned} A &= A'\cos^2\theta + C'\sin^2\theta \\ B &= (A'-C')\sin 2\theta \\ C &= A'\sin^2\theta + C'\cos^2\theta \\ F &= F' \end{aligned}$$

Note that rotation leaves the trace of the matrix (the sum of the diagonal elements) and the discriminant (the matrix determinant) unchanged:

$$\begin{aligned} &\text{trace} = A + C = A' + C' \\ &\text{discriminant} = AC - B^2/4 = A'C' - B'^2/4 = A'C' \end{aligned}$$

We can invert the equations above to find the orthogonal ellipse in terms of the rotated ellipse:

$$A - C = (A'-C')(\cos^2\theta - \sin^2\theta) = (A'-C')\cos 2\theta$$

so

$$\frac{B}{A-C} = \tan 2\theta$$

and

$$\theta = \frac{1}{2}\tan^{-1}\left(\frac{B}{A-C}\right)$$

Substituting one invariant, $C' = A + C - A'$ into the other, we get a quadratic equation in A':

$$A'^2 - (A+C)A' + (AC - B^2/4) = 0$$

so the coefficients of the orthogonal ellipse are:

$$A', C' = \frac{A + C \pm \sqrt{(A-C)^2 + B^2}}{2}$$

There are two solutions since a rotation of 90 degrees leaves the ellipse orthogonal but swaps A' and C'. These two values are the eigenvalues of the conic matrices $\mathbf{Q}$ and $\mathbf{Q}'$. The orthogonal ellipse can be rewritten $(A'/F)x^2 + (C'/F)y^2 = 1$ so the major and minor radii are:

$$a = \sqrt{\frac{F}{A'}} \qquad b = \sqrt{\frac{F}{C'}}$$

where A' and C' are chosen such that $A' \le C'$ and $a \ge b$. Note that these formulas allow the radii of an ellipse to be determined from its implicit coefficients without trigonometry.

B.4 Ellipse Synthesis by Transforming a Circle

An arbitrary ellipse can be created by scaling and rotating a circle. Starting with the implicit equation for a circle with conic matrix

$$\mathbf{Q}' = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

and $F' = 1$, we scale the circle differentially to have radii a and b and rotate it to have angle θ using the transformation matrix:

$$\mathbf{M} = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} a\cos\theta & a\sin\theta \\ -b\sin\theta & b\cos\theta \end{pmatrix}$$

The resulting ellipse has conic matrix:

$$\begin{aligned} \mathbf{Q} &= \mathbf{M}^{-1}\mathbf{Q}'\mathbf{M}^{-1^T} \\ &= \begin{pmatrix} A & B/2 \\ B/2 & C \end{pmatrix} = \frac{1}{ab} \begin{pmatrix} b\cos\theta & -a\sin\theta \\ b\sin\theta & a\cos\theta \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b\cos\theta & b\sin\theta \\ -a\sin\theta & a\cos\theta \end{pmatrix} \frac{1}{ab} \\ &= \frac{1}{a^2b^2} \begin{pmatrix} b^2\cos^2\theta + a^2\sin^2\theta & (b^2-a^2)\cos\theta\sin\theta \\ (b^2-a^2)\cos\theta\sin\theta & a^2\cos^2\theta + b^2\sin^2\theta \end{pmatrix} \end{aligned}$$

The ellipse thus has coefficients:

$$\begin{aligned} A &= a^2\cos^2\theta + b^2\sin^2\theta \\ B &= (b^2 - a^2)\sin 2\theta \\ C &= a^2\sin^2\theta + b^2\cos^2\theta \\ F &= a^2b^2 \end{aligned}$$

B.5 Parametric Ellipse

There are many parameterizations of an ellipse, but the following trigonometric one seems to be the most natural.

A parametric equation for a unit circle is:

$$\mathbf{p}(t) = (\, x(t) \quad y(t)\,) = (\, \cos t \quad \sin t\,)$$

Figure B.2: *Ellipse shape is determined by two basis vectors (u_x, v_x) and (u_y, v_y) which form rows of the Jacobian for the mapping from (x, y) space to (u, v) space. The basis for a given ellipse is not unique; pictured here are three bases of different phase for the same ellipse. Center parameterization is canonical because its basis vectors are orthogonal.*

When transformed by a scale and rotation we get the parametric ellipse:

$$\begin{aligned}\mathbf{p}' = \mathbf{p}\mathbf{M}_s\mathbf{M}_r &= \begin{pmatrix}\cos t & \sin t\end{pmatrix}\begin{pmatrix}a & 0\\ 0 & b\end{pmatrix}\begin{pmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{pmatrix}\\ &= \begin{pmatrix}a\cos\theta\cos t - b\sin\theta\sin t, & a\sin\theta\cos t + b\cos\theta\sin t\end{pmatrix}\end{aligned}$$

B.6 Converting Parametric to Implicit

Given a parametric circle in one coordinate system, $\mathbf{p} = (x, y) = (\cos t, \sin t)$, and a linear transformation to another coordinate system

$$\begin{aligned}\mathbf{p}' &= \mathbf{p}\mathbf{M}\\ &= \begin{pmatrix}u & v\end{pmatrix} = \begin{pmatrix}\cos t & \sin t\end{pmatrix}\begin{pmatrix}u_x & v_x\\ u_y & v_y\end{pmatrix}\end{aligned}$$

the circle transforms to an ellipse in the second coordinate system. The ellipse is centered on (0,0) and passes through the points (u_x, v_x) and (u_y, v_y) at parameter values t with a 90 degree phase difference (figure B.2). The matrix $\mathbf{M}$ is the Jacobian of the mapping from (x, y) space to (u, v) space.

The implicit equation for this ellipse can be found by substituting $\mathbf{p} = \mathbf{p}'\mathbf{M}^{-1}$ into the identity $\mathbf{p}\mathbf{p}^T = \cos^2 t + \sin^2 t = 1$:

$$\begin{aligned}\mathbf{p}'\mathbf{M}^{-1}\mathbf{M}^{-1^T}\mathbf{p}'^T &= 1\\ \mathbf{p}'\mathbf{Q}\mathbf{p}'^T &= 1\end{aligned}$$

where

$$\mathbf{Q} = \mathbf{M}^{-1}\mathbf{M}^{-1^T} = \frac{\begin{pmatrix}v_y & -v_x\\ -u_y & u_x\end{pmatrix}\begin{pmatrix}v_y & -u_y\\ -v_x & u_x\end{pmatrix}}{(u_x v_y - u_y v_x)^2} = \frac{\begin{pmatrix}v_x^2 + v_y^2 & -u_x v_x - u_y v_y\\ -u_x v_x - u_y v_y & u_x^2 + u_y^2\end{pmatrix}}{(u_x v_y - u_y v_x)^2}$$

So the ellipse has implicit coefficients:

$$\begin{aligned}A &= v_x^2 + v_y^2 \\ B &= -2(u_x v_x + u_y v_y) \\ C &= u_x^2 + u_y^2 \\ F &= (u_x v_y - u_y v_x)^2\end{aligned}$$

B.7 Converting Implicit to Parametric

The trigonometric parameterization for an ellipse has four degrees of freedom, but the implicit form has only three. The extra degree of freedom in the parametric form results from the arbitrary phase of the basis vectors of the ellipse (figure B.2); there is a one-dimensional space of basis vector pairs for a given ellipse. To eliminate this ambiguity we choose as a canonical basis the major and minor axes of the ellipse, which are always perpendicular. Such a basis, and such a parameterization, are unique within permutations and sign changes of the basis vectors.

The basis vectors of the ellipse are the rows of the Jacobian $\mathbf{M}$:

$$\mathbf{M} = \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix}$$

The Jacobian is related to the implicit conic matrix by

$$\mathbf{Q} = \mathbf{M}^{-1}\mathbf{M}^{-1^T}$$

This conic matrix is closely related to the *first fundamental matrix* of a surface in differential geometry [Faux-Pratt79]. We wish to solve the above equation for $\mathbf{M}$ under the constraint that the basis is orthogonal, i.e. $\mathbf{M}^{-1^T}\mathbf{M}^{-1}$ is diagonal. If its rows are orthogonal, then $\mathbf{M}$ can be written in the form

$$\mathbf{M} = \mathbf{\Lambda}\mathbf{R} = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

so the conic matrix is

$$\mathbf{Q} = \mathbf{M}^{-1}\mathbf{M}^{-1^T} = \mathbf{R}^{-1}\mathbf{\Lambda}^{-1}\mathbf{\Lambda}^{-1^T}\mathbf{R}^{-1^T} = \mathbf{R}^{-1}\mathbf{\Lambda}^{-2}\mathbf{R}$$

since $\mathbf{\Lambda}$ is diagonal and $\mathbf{R}$ is orthonormal. If we can extract $\mathbf{\Lambda}$ and $\mathbf{R}$ from any conic matrix then we can solve for its Jacobian. This can be done by finding the diagonal form for $\mathbf{Q}$,

$$\mathbf{Q} = \mathbf{S}^{-1}\mathbf{A}\mathbf{S}$$

where $\mathbf{A}$ is the diagonal matrix of the eigenvalues of $\mathbf{Q}$, and the columns of $\mathbf{S}$ are the corresponding eigenvectors [Strang80]. If the eigenvectors are chosen to have unit length then we can equate $\mathbf{R} = \mathbf{S}$ and $\mathbf{\Lambda}^{-2} = \mathbf{A}$.

The eigenvalues of a 2×2 symmetric matrix $\mathbf{Q}$ are

$$\mathbf{A} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} = \begin{pmatrix} (q+t)/2 & 0 \\ 0 & (q-t)/2 \end{pmatrix}$$

where

$$p = A - C$$

$$q = A + C$$

$$t = \operatorname{sgn}(p)\sqrt{p^2 + B^2}$$

and

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ +1 & x \geq 0 \end{cases}$$

When $t \neq 0$, The unit eigenvectors of $\mathbf{Q}$ are the columns of

$$\mathbf{S} = \begin{pmatrix} \sqrt{\frac{t+p}{2t}} & \operatorname{sgn}(Bp)\sqrt{\frac{t-p}{2t}} \\ -\operatorname{sgn}(Bp)\sqrt{\frac{t-p}{2t}} & \sqrt{\frac{t+p}{2t}} \end{pmatrix}$$

The eigenvectors of $\mathbf{M}$ are then given by $\mathbf{\Lambda} = \mathbf{A}^{-1/2}$:

$$a, b = \lambda_1^{-1/2}, \lambda_2^{-1/2} = \sqrt{\frac{2}{q \pm t}}$$

so the Jacobian is:

$$\mathbf{M} = \begin{pmatrix} u_x & v_x \\ u_y & v_y \end{pmatrix} = \mathbf{\Lambda R}$$

$$= \begin{pmatrix} \sqrt{\frac{2}{q+t}} & 0 \\ 0 & \sqrt{\frac{2}{q-t}} \end{pmatrix} \begin{pmatrix} \sqrt{\frac{t+p}{2t}} & \operatorname{sgn}(Bp)\sqrt{\frac{t-p}{2t}} \\ -\operatorname{sgn}(Bp)\sqrt{\frac{t-p}{2t}} & \sqrt{\frac{t+p}{2t}} \end{pmatrix}$$

$$= \begin{pmatrix} \sqrt{\frac{t+p}{t(q+t)}} & \operatorname{sgn}(Bp)\sqrt{\frac{t-p}{t(q+t)}} \\ -\operatorname{sgn}(Bp)\sqrt{\frac{t-p}{t(q-t)}} & \sqrt{\frac{t+p}{t(q-t)}} \end{pmatrix}$$

When $t = 0$, the ellipse is a circle, so basis vectors of any phase will be orthogonal. A convenient Jacobian in this case is:

$$\mathbf{M} = \begin{pmatrix} 1/\sqrt{A} & 0 \\ 0 & 1/\sqrt{A} \end{pmatrix}$$

The parametric ellipse defined by a Jacobian is:

$$\mathbf{p}'(s) = (u(s),\ v(s)) = (u_x \cos s + u_y \sin s,\ v_x \cos s + v_y \sin s)$$

References

[Bartels-Beatty-Barsky87] Richard Bartels, John Beatty, Brian Barsky, *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*, Morgan Kaufmann Publishers, Palo Alto, CA, 1987.

[Bier-Sloan86] Eric A. Bier, Kenneth R. Sloan, Jr., "Two-Part Texture Mappings", *IEEE Computer Graphics and Applications*, vol. 6, no. 9, Sept. 1986, pp. 40-53.

[Blinn-Newell76] James F. Blinn, Martin E. Newell, "Texture and Reflection in Computer Generated Images", *CACM*, vol. 19, no. 10, Oct. 1976, pp. 542-547.

[Blinn89] James F. Blinn, "Return of the Jaggy", *IEEE Computer Graphics and Applications*, vol. 9, no. 2, Mar. 1989, pp. 82-89.

[Bracewell78] Ronald N. Bracewell, *The Fourier Transform and Its Applications*, McGraw-Hill, New York, 1978.

[Brigham74] E. Oran Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[Carey-Greenberg85] Richard J. Carey, Donald P. Greenberg, "Textures for Realistic Image Synthesis", *Computers and Graphics*, vol. 9, no. 2, 1985, pp. 125-138.

[Catmull74] Edwin E. Catmull, *A Subdivision Algorithm for Computer Display of Curved Surfaces*, PhD thesis, Dept. of CS, U. of Utah, Dec. 1974.

[Catmull79] Edwin E. Catmull, "A Tutorial on Compensation Tables", *Computer Graphics* (SIGGRAPH '79 Proceedings), vol. 13, no. 2, Aug. 1979, pp. 1-7.

[Catmull-Smith80] Edwin E. Catmull, Alvy Ray Smith, "3-D Transformations of Images in Scanline Order", *Computer Graphics* (SIGGRAPH '80 Proceedings), vol. 14, no. 3, July 1980, pp. 279-285.

[Clark-Palmer-Lawrence85] James J. Clark, Matthew R. Palmer, Peter D. Lawrence, "A Transformation Method for the Reconstruction of Functions from Nonuniformly Spaced Samples", *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-33, no. 4, Oct. 1985, pp. 1151-1165.

[Coxeter78] H. S. M. Coxeter, *Non-Euclidean Geometry*, U. of Toronto Press, 1978.

[Crochiere-Rabiner83] Ronald E. Crochiere, Lawrence R. Rabiner, *Multirate Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[Crow77] Franklin C. Crow, "The Aliasing Problem in Computer-Generated Shaded Images", *CACM*, vol. 20, Nov. 1977, pp. 799-805.

[Crow84] Franklin C. Crow, "Summed-Area Tables for Texture Mapping", *Computer Graphics* (SIGGRAPH '84 Proceedings), vol. 18, no. 3, July 1984, pp. 207-212.

[Dudgeon-Mersereau84] Dan E. Dudgeon, Russell M. Mersereau, *Multidimensional Digital Signal Processing*, Prentice Hall, Englewood Cliffs, NJ, 1984.

[Faux-Pratt79] I. D. Faux, M. J. Pratt, *Computational Geometry for Design and Manufacture*, Ellis Horwood Ltd., Chichester, England, 1979.

[Feibush-Levoy-Cook80] Eliot A. Feibush, Marc Levoy, Robert L. Cook, "Synthetic Texturing Using Digital Filters", *Computer Graphics* (SIGGRAPH '80 Proceedings), vol. 14, no. 3, July 1980, pp. 294-301.

[Fiume-Fournier-Canale87] Eugene Fiume, Alain Fournier, V. Canale, "Conformal Texture Mapping", *Eurographics '87*, 1987, pp. 53-64.

[Foley-van Dam82] James D. Foley, Andries van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1982.

[Fournier-Fiume88] Alain Fournier, Eugene Fiume, "Constant-Time Filtering with Space-Variant Kernels", *Computer Graphics* (SIGGRAPH '88 Proceedings), vol. 22, no. 4, Aug. 1988, pp. 229-238.

[Fraser-Schowengerdt-Briggs85] Donald Fraser, Robert A. Schowengerdt, Ian Briggs, "Rectification of Multichannel Images in Mass Storage Using Image Transposition", *Computer Vision, Graphics, and Image Processing*, vol. 29, no. 1, Jan. 1985, pp. 23-36.

[Goldman83] Ronald N. Goldman, "An Urnful of Blending Functions", *IEEE Computer Graphics and Applications*, vol. 3, Oct. 1983, pp. 49-54.

[Gonzalez-Wintz87] Rafael C. Gonzalez, Paul Wintz, *Digital Image Processing* (2nd ed.), Addison-Wesley, Reading, MA, 1987.

[Greene86] Ned Greene, "Environment Mapping and Other Applications of World Projections", *Graphics Interface '86*, May 1986, pp. 108-114.

[Greene-Heckbert86] Ned Greene, Paul S. Heckbert, "Creating Raster Omnimax Images from Multiple Perspective Views Using The Elliptical Weighted Average Filter", *IEEE Computer Graphics and Applications*, vol. 6, no. 6, June 1986, pp. 21-27.

[Hamming83] R. W. Hamming, *Digital Filters*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[Heckbert86a] Paul S. Heckbert, "Filtering by Repeated Integration", *Computer Graphics* (SIGGRAPH '86 Proceedings), vol. 20, no. 4, Aug. 1986, pp. 317-321.

[Heckbert86b] Paul S. Heckbert, "Survey of Texture Mapping", *IEEE Computer Graphics and Applications*, vol. 6, no. 11, Nov. 1986, pp. 56-67.

[Hou-Andrews78] Hsieh S. Hou, Harry C. Andrews, "Cubic Splines for Image Interpolation and Digital Filtering", *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. ASSP-26, no. 6, Dec. 1978, pp. 508-517.

[Hourcade-Nicolas83] J. C. Hourcade, A. Nicolas, "Inverse Perspective Mapping in Scanline Order onto Non-Planar Quadrilaterals", *Eurographics '83*, 1983, pp. 309-319.

[Hummel83] Robert Hummel, "Sampling for Spline Reconstruction", *SIAM J. Appl. Math*, vol. 43, no. 2, Apr. 1983, pp. 278-288.

[Jackson86] Leland B. Jackson, *Digital Filters and Signal Processing*, Kluwer Academic Publishers, Boston, 1986.

[Kajiya-Ullner81] James T. Kajiya, Michael K. Ullner, "Filtering High Quality Text for Display on Raster Scan Devices", *Computer Graphics* (SIGGRAPH '81 Proceedings), vol. 15, no. 3, Aug. 1981, pp. 7-15.

[Maxwell46] E. A. Maxwell, *The Methods of Plane Projective Geometry, Based on the Use of General Homogeneous Coordinates*, Cambridge U. Press, London, 1946.

[Oppenheim-Schafer75] Alan V. Oppenheim, Ronald W. Schafer, *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[Paeth86] Alan W. Paeth, "A Fast Algorithm for General Raster Rotation", *Graphics Interface '86*, May 1986, pp. 77-81.

[Peachey85] Darwyn R. Peachey, "Solid Texturing of Complex Surfaces", *Computer Graphics* (SIGGRAPH '85 Proceedings), vol. 19, no. 3, July 1985, pp. 279-286.

[Perlin85] Ken Perlin, "An Image Synthesizer", *Computer Graphics* (SIGGRAPH '85 Proceedings), vol. 19, no. 3, July 1985, pp. 287-296.

[Pratt78] W. K. Pratt, *Digital Image Processing*, John Wiley and Sons, New York, 1978.

[Rao73] C. Radhakrishna Rao, *Linear Statistical Inference and Its Applications*, 2nd ed., John Wiley, NY, 1973.

[Roberts66] Lawrence G. Roberts, *Homogeneous Matrix Representation and Manipulation of N-Dimensional Constructs*, MS-1405, Lincoln Lab, Lexington, MA, July 1966.

[Rogers85] David F. Rogers, *Procedural Elements for Computer Graphics*, McGraw-Hill, New York, 1985.

[Rosenfeld84] Azriel Rosenfeld, *Multiresolution Image Processing and Analysis*, Springer, Berlin, 1984.

[Shannon49] Claude E. Shannon, "Communication in the Presence of Noise", *Proc. IRE*, vol. 37, no. 1, Jan. 1949, pp. 10-21.

[Shantz-Lien87] Michael Shantz, Sheue-Ling Lien, "Shading Bicubic Patches", *Computer Graphics* (SIGGRAPH '87 Proceedings), vol. 21, no. 4, July 1987.

[Smith83] Alvy Ray Smith, *Digital Filtering Tutorial for Computer Graphics, Parts I and II*, Lucasfilm Technical Memos 27 & 44, Pixar, San Rafael, CA, Mar. 1983, July 1983.

[Smith87] Alvy Ray Smith, "Planar 2-Pass Texture Mapping and Warping", *Computer Graphics* (SIGGRAPH '87 Proceedings), vol. 21, no. 4, July 1987, pp. 263-272.

[Strang80] Gilbert Strang, *Linear Algebra and Its Applications*, 2nd ed., Academic Press, New York, 1980.

[Turkowski88] Ken Turkowski, *Several Filters for Sample Rate Conversion*, Technical Report No. 9, Apple Computer, Cupertino, CA, May 1988.

[Ullner83] Mike K. Ullner, *Parallel Machines for Computer Graphics*, PhD thesis, California Institute of Technology, 1983.

[Williams83] Lance Williams, "Pyramidal Parametrics", *Computer Graphics* (SIGGRAPH '83 Proceedings), vol. 17, no. 3, July 1983, pp. 1-11.

[Wolberg88] George Wolberg, *Geometric Transformation Techniques for Digital Images: A Survey*, TR CUCS-390-88, Dept. of CS, Columbia U., Dec. 1988.