6. SCANLINE ALGORITHMS

Scanline algorithms comprise a special class of geometric transformation techniques that operate only along rows and columns. The purpose for using such algorithms is simplicity: resampling along a scanline is a straightforward 1-D problem that exploits simplifications in digital filtering and memory access. The geometric transformations that are best suited for this approach are those that can be shown to be separable, i.e., each dimension can be resampled independently of the other.

Separable algorithms spatially transform 2-D images by decomposing the mapping into a sequence of orthogonal 1-D transformations. For instance, 2-pass scanline algorithms typically apply the first pass to the image rows and the second pass to the columns. Although separable algorithms cannot handle all possible mapping functions, they can be shown to work particularly well for a wide class of common transformations, including affine and perspective mappings. It has been shown that they may be extended to deal with arbitrary mapping functions. This is all part of an effort to cast image warping into a framework that is amenable to fast implementations either in software or in hardware.

Separable algorithms have a high practical importance. This is due to the widespread proliferation of advanced workstations and digital signal processors. This has resulted in dramatic developments in both hardware and software systems. Examples include real-time hardware for video effects, texture mapping, and geometric correction. The speed offered by these products also suggests implications in new technologies that will exploit interactive image manipulation, of which image warping is an important component.

This chapter is devoted to geometric transformations that may be implemented with scanline algorithms. In general, this will imply that the mapping function is separable, although this need not always be the case. Consequently, space-variant digital filtering plays an increasingly important role in preventing aliasing artifacts. Despite the assumptions and errors that fall into this model of computation, separable algorithms perform surprisingly well.

6.1. INTRODUCTION

Geometric transformations have traditionally been formulated as either forward or inverse mappings operating entirely in 2-D. Their advantages and drawbacks have already been described in an earlier chapter. We briefly restate these features in order to better motivate the case for scanline algorithms and separable geometric transformations.

6.1.1. Forward Mapping

Forward mappings deposit input pixels into an output accumulator array. A distinction is made here based on the order in which pixels are fetched and stored. In forward mappings, the input arrives in scanline order (row by row) but the results are free to leave in any order, projecting into arbitrary areas in the output. In the general case, this means that no output pixel is guaranteed to be totally computed until the entire input has been scanned. Therefore, a full 2-D accumulator array must be retained throughout the duration of the mapping. Since the square input pixels project onto quadrilaterals at the output, costly intersection tests are needed to properly compute their overlap with the discrete output cells. Furthermore, an adaptive algorithm must be used to determine when supersampling is necessary in order to avoid blocky appearances upon one-to-many mappings.
6.1.2. **Inverse Mapping**

Inverse mappings are more commonly used to perform spatial transformations. By operating in scanline order at the output, square output pixels are projected onto arbitrary quadrilaterals. In this case, the projected areas lie in the input and are not generated in scanline order. Each preimage must be sampled and convolved with a low-pass filter to compute an intensity at the output. We have already reviewed clever approaches to efficiently approximate this computation. While either forward or inverse mappings can be used to realize *arbitrary* mapping functions, there are many transformations that are adequately approximated when using separable mappings. They exploit scanline algorithms to yield large computational savings.

6.1.3. **Separable Mapping**

There are several advantages to decomposing a mapping into a series of 1-D transforms. First, the resampling problem is made simpler since reconstruction, area sampling, and filtering can now be done entirely in 1-D. Second, this lends itself naturally to digital hardware implementation. Note that no sophisticated digital filters are necessary to deal explicitly with the 2-D case. Third, the mapping can be done in scanline order both in scanning the input image and in producing the projected image. In this manner, an image may be processed in the same format in which it is stored in the frame buffer: rows and columns. This leads to efficient data access and large savings in I/O time. The approach is amenable to stream-processing techniques such as pipelining and facilitates the design of hardware that works at real-time video rates.

6.2. **INCREMENTAL ALGORITHMS**

In this section, we examine the problem of image warping with several incremental algorithms that operate in scanline order. We begin by considering an incremental scanline technique for texture mapping. The ideas are derived from shading interpolation methods in computer graphics.

6.2.1. **Texture Mapping**

*Texture mapping* is a powerful technique used to add visual detail to synthetic images in computer graphics. It consists of a series of spatial transformations: a texture plane, \([u,v]\), is transformed onto a 3-D surface, \([x,y,z]\), and then projected onto the output screen, \([x,y]\). This sequence is shown in Figure 6.1, where the transformation from \([u,v]\) to \([x,y,z]\) and \(p\) is the projection from \([x,y,z]\) onto \([x,y]\). For simplicity, we have assumed that \(p\) realizes an orthographic projection. The forward mapping functions \(X\) and \(Y\) represent the composite function \(p(f(u,v))\). The inverse mapping functions are \(U\) and \(V\).
Texture mapping serves to create the appearance of complexity by simply applying image detail onto a surface, in much the same way as wallpaper. Textures are rather loosely defined. They are usually taken to be images used for mapping color onto the targeted surface. Textures are also used to perturb surface normals, thus allowing us to simulate bumps and wrinkles without the tedium of modeling them geometrically. Additional applications are included in [Heckbert 86b], a survey article on texture mapping.

The mapping between the input and output images is usually treated as a four-corner mapping. In inverse mapping, square output pixels must be projected back onto the input image for resampling purposes. In forward mapping, we project square texture pixels onto the output image via mapping functions \( X \) and \( Y \). Below we describe an inverse mapping technique.

Consider an input square texture in the \( uv \) plane mapped onto a planar quadrilateral in the \( xyz \) coordinate system. The mapping can be specified by designating texture coordinates to the quadrilateral. For simplicity we select four corner mapping, as depicted in Figure 6.2. In this manner, the four point correspondences are \((u_i, v_i)\). The problem now remains to determine the correspondence for all interior quadrilateral points. This task is reminiscent of the interpolation paradigm already considered in an earlier chapter. In the subsections that follow, we turn to a simplistic approach drawn from the computer graphics field.

**Figure 6.2: Four corner mapping.**

### 6.2.2. Gouraud Shading

Gouraud shading is a popular intensity interpolation algorithm used to shade polygonal surfaces in computer graphics [Gouraud 71]. It serves to enhance realism in rendered scenes that approximate curved surfaces with planar polygons. Although we have no direct use for shading algorithms here, we use a variant of this approach to interpolate texture coordinates.
We begin with a review of Gouraud shading in this section, followed by a description of its use in texture mapping in the next section.

Gouraud shading interpolates the intensities all along a polygon, given only the true values at the vertices. It does so while operating in scanline order. This means that the output screen is rendered in a raster fashion, (e.g., scanning the polygon from top-to-bottom, with each scan moving left-to-right). This spatial coherence lends itself to a fast incremental method for computing the interior intensity values. The basic approach is illustrated in Figure 6.3.

For each scanline, the intensities at endpoints $x_0$ and $x_1$ are computed. This is achieved through linear interpolation between the intensities of the appropriate polygon vertices. This yields $I_0$ and $I_1$ in Figure 6.3, where

$$I_0 = \alpha \cdot I_A + (1-\alpha) \cdot I_B$$
$$I_1 = \beta \cdot I_C + (1-\beta) \cdot I_D$$

$0 \leq \alpha \leq 1$
$0 \leq \beta \leq 1$

![Figure 6.3: Incremental scanline interpolation.](image)

Then, beginning with $I_0$, the intensity values along successive scanline positions are computed incrementally. In this manner, $I_{x+1}$ can be determined directly from $I_x$, where the subscripts refer to positions along the scanline. We thus have

$$I_{x+1} = I_x + dl$$

where

$$dl = \frac{(I_1 - I_0)}{(x_1 - x_0)}$$

Note that the scanline order allows us to exploit incremental computations. As a result, we are spared from having to evaluate two multiplications and two additions per pixel. Additional savings are possible by computing $I_0$ and $I_1$ incrementally as well. This requires a different set of constant increments to be added along the polygon edges.

### 6.2.3. Incremental Texture Mapping

Although Gouraud shading has traditionally been used to interpolate intensity values, we now use it to interpolate texture coordinates. The computed $(u,v)$ coordinates are used to index into...
the input texture. This permits us to obtain a color value that is then applied to the output pixel. The following segment of C code is offered as an example of how to process a single scanline.

```c
dx = 1.0/(x1 - x0); /* normalization factor */
du = (u1 - u0) * dx; /* constant increment for u */
dv = (v1 - v0) * dx; /* constant increment for v */
dz = (z1 - z0) * dx; /* constant increment for z */
for(x = x0; x < x1; x++) { /* visit all scanline pixels */
    if(z < zbuf[x]) { /* is new point closer? */
        zbuf[x] = z; /* update z-buffer */
        scr[x] = tex(u,v); /* write texture value to screen */
    }
    u += du; /* increment u */
    v += dv; /* increment v */
    z += dz; /* increment z */
}
```

The procedure given above assumes that the scanline begins at \((x_0, y, z_0)\) and ends at \((x_1, y, z_1)\). These two endpoints correspond to points \((u_0, v_0)\) and \((u_1, v_1)\), respectively, in the input texture. For every unit step in \(x\), coordinates \(u\) and \(v\) are incremented by a constant amount, e.g., \(du\) and \(dv\), respectively. This equates to an affine mapping between a horizontal scanline in screen space and an arbitrary line in texture space with slope \(dv/du\) (see Figure 6.4).

![Figure 6.4: Incremental interpolation of texture coordinates.](image)

Since the rendered surface may contain occluding polygons, the \(z\)-coordinates of visible pixels are stored in \(zbuf\), the \(z\)-buffer for the current scanline. When a pixel is visited, its \(z\)-buffer entry is compared against the depth of the incoming pixel. If the incoming pixel is found to be closer, then we proceed with the computations involved in determining the output value and update the \(z\)-buffer with the depth of the closer point. Otherwise, the incoming point is occluded and no further action is taken on that pixel.

The function \(tex(u,v)\) in the above code samples the texture at point \((u,v)\). It returns an intensity value that is stored in \(scr\), the screen buffer for the current scanline. For colour images, RGB values would be returned by \(tex\) and written into three separate colour channels. In the examples that follow, we let \(tex\) implement point sampling, e.g., no filtering. Although this introduces well-known artefacts, our goal here is to examine the geometrical properties of this simple approach. We will therefore tolerate artefacts, such as jagged edges, in the interest of simplicity.

Figure 6.5 shows the Checkerboard image mapped onto a quadrilateral using the approach described above. There are several problems that are readily noticeable. First, the textured
polygon shows undesirable discontinuities along horizontal lines passing through the vertices. This is due to a sudden change in $du$ and $dv$ as we move past a vertex. It is an artefact of the linear interpolation of $u$ and $v$. Second, the image does not exhibit the foreshortening that we would expect to see from perspective. This is due to the fact that this approach is consistent with the bilinear transformation scheme. As a result, it is exact for affine mappings but it is inadequate to handle perspective mappings [Heckbert 89].

Figure 6.5: Naive approach applied to Checkerboard.

The constant increments used in the linear interpolation are directly related to the general transformation matrix elements. We then have

\[
\begin{align*}
xw &= a_{11}u + a_{21}v + a_{31} \\
yw &= a_{12}u + a_{22}v + a_{32} \\
w &= a_{13}u + a_{23}v + a_{33}
\end{align*}
\]

For simplicity, we select $a_{33} = 1$ and leave eight degrees of freedom for the general transformation. Solving for $u$ and $v$ in terms of $x$, $y$, and $w$, we have

\[
\begin{align*}
u &= \frac{a_{22}xw - a_{21}yw + a_{21}a_{32} - a_{22}a_{31}}{a_{11}a_{22} - a_{12}a_{21}} \\
v &= \frac{-a_{12}xw + a_{11}yw - a_{11}a_{32} + a_{12}a_{31}}{a_{11}a_{22} - a_{12}a_{21}}
\end{align*}
\]

This gives rise to expressions for $du$ and $dv$. These terms represent the increment added to the interpolated coordinates at position $x$ to yield a value for the next point at $x+1$. If we refer to these positions with subscripts 0 and 1, respectively, then we have
For affine transformations, \( w_0 = w_1 = 1 \) and Eqs.6.6 simplify to
\[
\begin{align*}
du &= u_i - u_0 = \frac{a_{22}(x_i w_1 - x_0 w_0)}{a_{11}a_{22} - a_{12}a_{21}} \\
v &= v_i - v_0 = \frac{-a_{12}(x_i w_1 - x_0 w_0)}{a_{11}a_{22} - a_{12}a_{21}}
\end{align*}
\]
6.6

The expression for \( dw \) can be derived from \( du \) and \( dv \) as follows.
\[
\begin{align*}
dw &= a_{13} \cdot du + a_{23} \cdot dv = \\
&= \frac{(a_{11}a_{22} - a_{12}a_{21})(x_i w_1 - x_0 w_0)}{a_{11}a_{22} - a_{12}a_{21}}
\end{align*}
\]
6.8

The error of the linear interpolation method vanishes as \( dw \to 0 \). A simple ad hoc solution to achieve this goal 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 discontinuities near the vertices. The problem with this method is that costly computations must be made to correctly compute the texture coordinates at the new vertices, and it is difficult to determine how much subdivision is necessary. Clearly, the more parallel the polygon lies to the viewing plane, the less subdivision is warranted.

In order to provide some insight into the effect of subdivision, Figure 6.6 illustrates the result of subdividing the polygon of Figure 6.2 several times. In Figure 6.6a, the edges of the polygon were subdivided into two equal parts, generating four smaller polygons. Their borders can be deduced in the figure by observing the persisting discontinuities. Due to the foreshortening effects of the perspective mapping, the placement of these borders are shifted from the apparent midpoints of the edges. Figure 6.6b, Figure 6.6c, and Figure 6.6d show the same polygon subdivided 2, 4, and 8 times, respectively. Notice that the artefacts diminish with each subdivision.
This approach has been introduced into graphics workstations that feature real-time texture mapping. One such method is reported in [Oka 87]. It is important to note that Gouraud shading has been used for years without major noticeable artefacts because shading is a slowly-varying function. However, applications such as texture mapping bring out the flaws of this approach more readily with the use of highly-varying texture patterns.

6.2.4. Incremental Perspective Transformations

A theoretically correct solution results by more closely examining the requirements of a perspective mapping. Since a perspective transformation is a ratio of two linear interpolants, it becomes possible to achieve theoretically correct results by introducing the divisor, i.e., homogeneous coordinate $w$. We thus interpolate $w$ alongside $u$ and $v$, and then perform two divisions per pixel. The following code contains the necessary adjustments to make the scanline approach work for perspective mappings.
dx = 1.0/(x1 - x0); /* normalization factor */
du = (u1 - u0) * dx; /* constant increment for u */
dv = (v1 - v0) * dx; /* constant increment for v */
dz = (z1 - z0) * dx; /* constant increment for z */
dw = (w1 - w0) * dx; /* constant increment for w */
for (x = x0; x < x1; x++) {
  if (z < zbuf[x]) { /* is new point closer? */
    zbuf[x] = z; /* update z-buffer */
    scr[x] = tex (u/w, v/w); /* write texture value to screen */
  }
  u += du; /* increment u */
  v += dv; /* increment v */
  z += dz; /* increment z */
  w += dw; /* increment w */
}

Figure 6.7 shows the result of this method after it was applied to the Checkerboard texture. Notice the proper foreshortening and the continuity near the vertices.

Figure 6.7: Perspective mapping using scanline algorithm.

6.2.5. Approximations

The main objective of the scanline algorithm described above is to exploit the use of incremental computation for fast texture mapping. However, the division operations needed for perspective mappings are expensive and undermine some of the computational gains. Although it can be argued that division requires only marginal cost relative to antialiasing, it is worthwhile to examine optimizations that can be used to approximate the correct solution. Before we do so, we review the geometric nature of the problem at hand.

Consider a planar polygon lying parallel to the viewing plane. All points on the polygon thereby lie equidistant from the viewing plane. This allows equal increments in screen space (the viewing plane) to correspond to equal, albeit not the same, increments on the polygon. As a result, linear interpolation of u and v is consistent with this spatial transformation, an affine
Scanline Algorithms
mapping. However, if the polygon lies obliquely relative to the viewing plane, then foreshortening is introduced. This no longer preserves equispaced points along lines. Consequently, linear interpolation of \( u \) and \( v \) is inconsistent with the perspective mapping.

Although both mappings interpolate the same lines connecting \( u_0 \) to \( u_1 \) and \( v_0 \) to \( v_1 \), it is the rate at which these lines are sampled that is different. Affine mappings cause the line to be uniformly sampled, while perspective mappings sample the line more densely at distant points where foreshortening has a greater effect. This is depicted in Figure 6.8 which shows a plot of the \( u \)-coordinates spanned using both affine and perspective mappings.

![Figure 6.8 Interpolating texture coordinates.](image)

We have already shown that division is necessary to achieve the correct results. There are many applications that look for cheaper approximations on more conventional hardware. Consequently, we examine how to approximate the nonuniform sampling depicted in Fig. 6.8.

The most straightforward approach makes use of the Taylor series to approximate division. The Taylor series of a function evaluated about the point \( x_0 \) is given as

\[
f(x) = f(x_0) + f'(x_0)d + \frac{f''(x_0)d^2}{2!} + \frac{f'''(x_0)d^3}{3!} + \cdots
\]

where \( d = x - x_0 \). If we let \( f \) be the reciprocal function for \( w \), i.e., \( f(x) = \frac{1}{w} \), \( \delta = x - x \), then we may use the following first-order truncated Taylor series approximation [Lien 87].

\[
\frac{1}{w} \approx \frac{1}{w_0} - \frac{\delta}{w_0^2}, \quad \delta = x - x_0
\]

The authors of that paper suggest that \( w_0 \) be the most significant 8 bits of a 32-bit fixed point integer storing \( w \). A lookup-table, indexed by an 8-bit \( w_0 \), contains the entries for \( 1/w_0 \). That result may be combined with the lower 24-bit quantity \( d \) to yield the approximated quotient. In particular, if we let \( a \) be the rough estimate \( 1/w_0 \) that is retrieved from the lookup table, and \( b \) be the least significant 24-bit quantity of \( w \), then from Eq. 6.10 we have \( 1/w = a - a*a*b \). In this manner, division has been replaced with addition and multiplication operations. The reader can verify that an 8-bit \( w_0 \) and 24-bit \( d \) yields 18 bits of accuracy in the result. The full 32 bits of precision can be achieved with the use of the 16 higher-order bits for \( w_0 \) and the low-order 16 bits for \( d \).
6.3. ROTATION

The incremental scanline algorithms described above all exploit the computational savings made possible by forward differences. While they may be fast at computing the transformation, they neglect filtering issues between scanlines. Rather than attempt to approximate the transformation along only one direction, separable algorithms decompose their mapping functions along orthogonal directions, i.e., rows and columns. In this manner, the computation of the transformation is more precise, and the associated resampling remains a straightforward 1-D filtering operation. The earliest separable geometric techniques can be traced back to the application of image rotation. Several of these algorithms are reviewed below.

6.3.1. Catmull and Smith, 1980

Catmull and Smith describe a 2-pass solution to a wide class of spatial transformations in [Catmull 80]. Their work is quite general, including affine and perspective transformations onto planar surfaces, biquadratic patches, bicubic patches, and superquadrics. Image rotation, being an affine transformation, is of course treated in their work. The resulting 2-pass transform decomposes the rotation matrix $R$ into two submatrices, each producing a scale/shear transformation.

$$R = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} = \begin{bmatrix} \cos \theta & 0 \\ -\sin \theta & 1 \end{bmatrix} \begin{bmatrix} 1 & \tan \theta \\ 0 & 1/\cos \theta \end{bmatrix}$$  \hspace{1cm} 6.11

The algorithm first skews and scales the image along the horizontal direction. The result then undergoes a similar process in the vertical direction. This 2-pass approach is illustrated in Figure 6.9. A description of a hardware system to implement this process is found in [Tabata 86].
6.3.2. Paeth, 1986 / Tanaka, et. al., 1986

The most significant algorithm to be developed for image rotation was proposed independently in [Paeth 86] and [Tanaka 86, 88]. They demonstrate that rotation can be implemented by cascading three shear transformations.

\[ R = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \sin \theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\tan(\theta / 2) & 1 \end{bmatrix} \]

The algorithm first skews the image along the horizontal direction by displacing each row. The result is then skewed along the vertical direction. Finally, an additional skew in the horizontal direction yields the rotated image. This sequence is illustrated in Figure 6.10.

The primary advantage to the 3-pass shear transformation algorithm is that it avoids a costly scale operation. In this manner, it differs significantly from the 2-pass Catmull-Smith algorithm which combined scaling and shearing in each pass, and the 4-pass Weiman algorithm which further decomposed the scale/shear sequence. By not introducing a scale operation, the algorithm avoids complications in sampling, filtering, and the associated degradations. Note, for instance, that this method is not susceptible to the bottleneck problem.
Simplifications are based in the particularly efficient means available to realize a shear transformation. The skewed output is the result of displacing each scanline differently. The displacement is generally not integral, but remains constant for all pixels on a given scanline. This allows intersection testing to be computed once for each scanline, noting that each input pixel can overlap at most two output pixels in the skewed image. The result is used to weigh each input intensity as it contributes to the output. Since the filter support is limited to two pixels, a simple triangle filter (linear interpolation) is adequate. Furthermore, the sum of the pixel intensities along any scanline can be shown to remain unchanged after the shear operation. Thus, the algorithm produces no visible spatial-variant artifacts or holes. Finally, images on bitmap displays can be rotated using conventional hardware supporting bitblt, the bit block transfer operation useful for translations. A C program to implement this algorithm is given below.

Figure 6.10: 3-pass shear rotation algorithm.
Rotate image IN about its center by angle ang (in radians)
IN has height h and width w. The output is stored in OUT
We assume that 0 <= ang < \( \frac{\pi}{2} \)
***********************************************************************
rotate(IN, h, w, ang, OUT)
unsigned char *IN, *OUT;
int h, w;
double ang;
{
    int x, y, wmax, newh, neww;
    double sine, tangent, offst;

    /* the dimensions of the rotated image as it is processed are:
       * (h)(w) -> (h)(wmax) -> (newh)(wmax) -> (newh)(neww).
       * +1 will be added to dimensions due to last fractional pixel
       * Temporary buffer TMP is used to hold intermediate image. */
    sine = sin(ang);
    tangent = tan(ang/2.0);
    wmax = w + h*tangent + 1;       /* width of intermediate image */
    newh = w*sine + h*cos(ang) + 1; /* final image height */
    neww = h*sine + w*cos(ang) + 1; /* final image width */

    /* 1st pass: skew x (horizontal scanlines) */
    for(y = 0; y < h; y++) {
        /* visit each row in IN */
        src = &IN[y*w];   /* input scanline pointer */
        dst = &OUT[y * wmax];   /* output scanline pointer */
        skew(src, w, wmax, y*tangent, 1, dst);   /* skew row */
    }

    /* 2nd pass: skew y (vertical scanlines).
       Use TMP for intermediate image */
    offst = (w-1) * sine;   /* offset from top of image */
    for(x = 0; x < wmax; x++) {
        /* visit each column in OUT */
        src = &OUT[x];   /* input scanline pointer*/
        dst = &TMP[x];   /* output scanline pointer */
        skew(src, h, newh, offst - x*sine, wmax, dst);  /* skew column */
    }

    /* 3rd pass: skew x (horizontal scanlines) */
    for(y = 0; y < newh; y++) {
        /* visit each row in TMP */
        src = &TMP[y * wmax];   /* input scanline pointer*/
        dst = &OUT[y * neww];   /* output scanline pointer*/
        skew(src, wmax, neww, (y-offst)*tangent, 1, dst);  /* skew row */
    }
}
/***************************************************************************/
skew(scanline in src (length len) into dst (length nlen)
starting at position strt. The offset between each scanline
pixel is offst. offst=1 for rows; offst=width for columns
***************************************************************************/
skew(src, len, nlen, strt, offst, dst)
unsigned char *src, *dst;
int len, nlen, offst;
double strt;
{
    int i, istrt, lim;
    double f, g, wl, w2;

    /* process left end of output: */
either prepare for clipping or add padding */

    istrt = (int) strt;               /* integer index */
    if(istrt < 0) src -= (offst*istrt); /* advance input pointer for clipping */
    lim = MIN(len+istrt, nlen);       /* find index for right edge */
    for(i = 0; i < istrt; i++) {       /* null output pixels at left edge */
      *dst = 0;                        /* pad with 0 */
      dst += offst;                   /* advance output pointer */
    }
    f = ABS(strt - istrt);            /* weight for right straddle */
    g = 1. - f;                      /* weight for left straddle */
    if(f == 0.) {                     /* simple integer shift: no interpolation */
      for(; i < lim; i++) {           /* visit all pixels in valid range */
        dst = *src;                   /* copy input to output */
        src += offst;                /* advance input pointer*/
        dst += offst;                /* advance output pointer */
      }
    }
    else {                           /* fractional shift: interpolate */
      if(strt > 0.) {                /* weight for left pixel */
        w1 = f;
        w2 = g;
        *dst = g * src[0];            /* first pixel */
        *dst += offst;               /* advance output pointer */
        i++;
      }
      else {                        /* weight for right pixel */
        w1 = g;
        w2 = f;
        if(lim < nlen) lim--;        /* increment index */
      }
      for(; i < lim; i++) {          /* visit all pixels i valid range */
        dst = w1*src[0] + w2*src[offst]; /* linear interpolation */
        dst += offst;                /* advance output pointer */
        src += offst;                /* advance input pointer */
      }
      if(i < nlen) {                 /* src[0] is last pixel */
        *dst = w1 * src[0];          /* src[0] is last pixel */
        dst += offst;                /* advance output pointer */
        i++;
      }
      for(; i < nlen; i++) {         /* visit pixels at right edge */
        *dst = 0;                     /* pad with 0 */
        dst += offst;                /* advance output pointer */
      }
    }

6.4. 2-PASS TRANSFORMS

Consider a spatial transformation specified by forward mapping functions $X$ and $Y$ such that

$$[x, y] = T(u, v) = [X(u, v), Y(u, v)]$$

The transformation $T$ is said to be separable if $T(u, v)=F(u)G(v)$. Since it is understood that $G$ is applied only after $F$, the mapping $T(u,v)$ is said to be 2-pass transformable, or simply

Scanline Algorithms
2-passable. Functions $F$ and $G$ are called the 2-pass functions, each operating along different axes. Consequently, the forward mapping in Eq. 7.4.1 can be rewritten as a succession of two 1-D mappings $F$ and $G$, the horizontal and vertical transformations, respectively.

It is important to elaborate on our use of the term separable. As mentioned above, the signal processing literature refers to a filter $T$ as separable if $T(u,v) = F(u) \cdot G(v)$. This certainly applied to the rotation algorithms described earlier. The definition we offer for separability is consistent with standard implementation practices. For instance, the 2-D Fourier transform, separable in the classic sense, is generally implemented by a 2-pass algorithm. The first pass applies a 1-D Fourier transform to each row, and the second applies a 1-D Fourier transform along each column of the intermediate result. Multi-pass scanline algorithms that operate in this sequential row-column manner will be referred to as separable. The underlying theme is that processing is decomposed into a series of 1-D stages that each operate along orthogonal axes.

The most general presentation of the 2-pass technique appears in the seminal work described by Catmull and Smith in [Catmull 80]. This paper tackles the problem of mapping a 2-D image onto a 3-D surface and then projecting the result onto the 2-D screen for viewing. The contribution of this work lies in the decomposition of these steps into a sequence of computationally cheaper mapping operations. In particular, it is shown that a 2-D resampling problem can be replaced with two orthogonal 1-D resampling stages. This is depicted in Figure 6.11.

6.4.1. First Pass

In the first pass, each horizontal scanline (row) is resampled according to spatial transformation $F(u)$, generating an intermediate image $I$ in scanline order. All pixels in $I$ have the same $x$-coordinates that they will assume in the final output; only their $y$-coordinates now remain to be computed. Since each scanline will generally have a different transformation, function $F(u)$ will usually differ from row to row. Consequently, $F$ can be considered to be a function of both $u$ and $v$. In fact, it is clear that mapping function $F$ is identical to $X$, generating $x$-coordinates from points in the $[u,v]$ plane. To remain consistent with earlier notation, we rewrite $F(u,v)$ as $F_v(u)$ to denote that $F$ is applied to horizontal scanlines, each having constant $v$. Therefore, the first pass is expressed as

$$[x,v] = [F_v(u),v], \quad F_v(u) = X(u,v)$$

where $F_v(u) = X(u,v)$. This relation maps all $[u,v]$ points onto the $[x,v]$ plane.

6.4.2. Second Pass

In the second pass, each vertical scanline (column) in $I$ is resampled according to spatial transformation $G(v)$, generating the final image in scanline order. The second pass is more complicated than the first pass because the expression for $G$ is often difficult to derive. This is due to the fact that we must invert $[x,v]$ to get $[u,v]$ so that $G$ can directly access $Y(u,v)$. In doing so, new $y$-coordinates can be computed for each point in $I$.

Inverting $f$ requires us to solve the equation $X(u,v) - x = 0$ for $u$ to obtain $u = H_x(v)$ for vertical scanline (column). Note that contains all the pixels along the column at $x$. Function $H_x$, known as the auxiliary function, represents the $u$-coordinates of the inverse projection of,
Digitales Video

the column we wish to resample. Thus, for every column in $I$, we compute $H_x(v)$ and use it together with the available $v$-coordinates to index into mapping function $Y$. This specifies the vertical spatial transformation necessary for resampling the column. The second pass is therefore expressed as

$$[x, y] = [x, G_x(v)], \quad G_x(v) = Y(H_x(v), v), u = H_x(v)$$

where $G_x(v)$ refers to the evaluation of $G(x, v)$ along vertical scanlines with constant $x$.

The relation in Eq. 6.15 maps all points in $I$ from the $[x,v]$ plane onto the $[x,y]$ plane, the coordinate system of the final image.

Figure 6.11: 2-pass geometric transformation.

6.4.3. 2-Pass Algorithm

In summary, the 2-pass algorithm has three steps. They correspond directly to the evaluation of scanline functions $F$ and $G$, as well as the auxiliary function $H$.

The horizontal scanline function is defined as $F_x(u) = X(u,v)$. Each row is resampled according to this spatial transformation, yielding intermediate image $I$. The auxiliary function $H_x(v)$ is derived for each vertical scanline in $I$. It is defined as the solution to $u = X(u,v)$ for $u$, if such a solution can be derived. Sometimes a closed form solution for $H$ is not possible and
numerical techniques such as the Newton-Raphson iteration method must be used. As we shall see later, computing $H$ is the principal difficulty with the 2-pass algorithm.

Once $H_x(v)$ is determined, the second pass plugs it into the expression for $Y(u,v)$ to evaluate the target $y$-coordinates of all pixels in column $x$ in image $I$. The vertical scanline function is defined as $G_v(v) = Y(H_x(v),v)$. Each column in $I$ is resampled according to this spatial transformation, yielding the final image.

6.4.4. An Example: Rotation

The above procedure is demonstrated on the simple case of rotation. The rotation matrix is given as

$$[x, y] = [u, v] \cdot \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix}$$

6.16

We want to transform every pixel in the original image in scanline order. If we scan a row by varying $u$ and holding $v$ constant, we immediately notice that the transformed points are not being generated in scanline order. This presents difficulties in antialiasing filtering and fails to achieve our goals of scanline input and output.

Alternatively, we may evaluate the scanline by holding $v$ constant in the output as well, and only evaluating the new $x$ values. This is given as

$$[x, v] = [u \cdot \cos \theta - v \cdot \sin \theta, v]$$

6.17

This results in a picture that is skewed and scaled along the horizontal scanlines.

The next step is to transform this intermediate result by holding $x$ constant and computing $y$. However, the equation $y = u \sin \theta + v \cos \theta$ cannot be applied since the variable $u$ is referenced instead of the available $x$. Therefore, it is first necessary to express $u$ in terms of $x$. Recall that $x = u \cos \theta - v \sin \theta$, so

$$u = \frac{x + v \cdot \sin \theta}{\cos \theta}$$

6.18

Substituting this into $y = u \sin \theta + v \cos \theta$ yields

$$y = \frac{v + x \cdot \sin \theta}{\cos \theta}$$

6.19

The output picture is now generated by computing the $y$-coordinates of the pixels in the intermediate image, and resampling in vertical scanline order. This completes the 2-pass rotation. Note that the transformations specified by Eqs.6.17 and 6.19 are embedded in Eq. 6.12. An example of this procedure for a 45° clockwise rotation has been shown in Figure 6.11.

The stages derived above are directly related to the general procedure described earlier. The three expressions for $F$, $G$, and $H$ are explicitly listed below.
The first pass is defined by Eq. 6.17. In this case, $F(v) = u \cos \theta - v \sin \theta$. The auxiliary function $H$ is given in Eq. 7.4.7. It is the result of isolating $u$ from the expression for $x$ in mapping function $X(u,v)$. In this case, $H(v) = (x + v \sin \theta)/\cos \theta$. The second pass then plugs $H(v)$ into the expression for $Y(u,v)$, yielding Eq. 6.19. In this case, $G(v) = (x \sin \theta + v)/\cos \theta$.

### 6.4.5. Another Example: Perspective

Another typical use for the 2-pass method is to transform images onto planar surfaces in perspective. In this case, the spatial transformation is defined as

$$[x', y', w'] = [u, v, 1] \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}, \quad x = x'/w', \quad y = y'/w' \quad 6.20$$

where $x$ and $y$ are the final coordinates in the output image. In the first pass, we evaluate the new $x$ values. giving us

$$[x, v] = \begin{bmatrix} a_{11}u + a_{21}v + a_{31} \\ a_{13}u + a_{23}v + a_{33} \end{bmatrix}, \quad 6.21$$

Before the second pass can begin, we use Eq. 7.4.10 to find $u$ in terms of $x$ and $v$:

$$ (a_{13}u + a_{23}v + a_{33}) \cdot x = a_{13}u + a_{21}v + a_{31}$$

$$ (a_{13}x - a_{11}) \cdot u = -(a_{23}v + a_{33})x + a_{21}v + a_{31}$$

$$ u = \frac{-(a_{23}v + a_{33})x + a_{21}v + a_{31}}{(a_{13}x - a_{11})} \quad 6.22$$

Substituting this into our expression for $y$ yields

$$ y = \frac{a_{12}u + a_{22}v + a_{32}}{a_{13}u + a_{23}v + a_{33}}$$

$$ = \frac{[-a_{12}(a_{23}v + a_{33})x + a_{12}a_{21}v + a_{12}a_{31} + [(a_{31}x - a_{11})(a_{23}v + a_{33})]}{[-a_{13}(a_{23}v + a_{33})x + a_{13}a_{21}v + a_{13}a_{31} + [(a_{31}x - a_{11})(a_{23}v + a_{33})]]}$$

$$ = \frac{[(a_{13}a_{22} - a_{11}a_{23})x + a_{13}a_{12}x - a_{11}a_{12}a_{32}v + (a_{13}a_{32} - a_{11}a_{33})x + (a_{13}a_{13} - a_{11}a_{13})]}{(a_{13}a_{21} - a_{11}a_{23})v + (a_{13}a_{31} - a_{11}a_{33})} \quad 6.23$$

For a given column, $x$ is constant and Eq. 6.23 is a ratio of two linear interpolants that are functions of $v$. As we make our way across the image, the coefficients of the interpolants change (being functions of $x$ as well), and we get the spatially-varying results shown in Figure 6.13.

### 6.4.6. Bottleneck Problem

After completing the first pass, it is sometimes possible for the intermediate image to collapse into a narrow area. If this area is much less than that of the final image, then there is insufficient data left to accurately generate the final image in the second pass.
phenomenon, referred to as the bottleneck problem in [Catmull 80], is the result of a many-to-one mapping in the first pass followed by a one-to-many mapping in the second pass.

The bottleneck problem occurs, for instance, upon rotating an image clockwise by 90° since the top row will map to the rightmost column, all of the points in the scanline will collapse onto the rightmost point. Similar operations on all the other rows will yield a diagonal line as the intermediate image. No possible separable solution exists for this case when implemented in this order. This unfortunate result can be readily observed by noting that the cos term in the denominator of Eq. 6.18 approaches zero as θ approaches 90° hereby giving rise to an undeterminable inverse.

The solution to this problem lies in considering all the possible orders in which a separable algorithm can be implemented. Four variations are possible to generate the intermediate image:

- Transform u first.
- Transform v first.
- Rotate the input image by 90° transform u first.
- Rotate the input image by 90° transform v first.

In each case, the area of the intermediate image can be calculated. The method that produces the largest intermediate area is used to implement the transformation. If a 90° rotation is required, it is conveniently implemented by reading horizontal scanlines and writing them in vertical scanline order.

In our example, methods (3) and (4) will yield the correct result. This applies equally to rotation angles near 90° or instance, an 87° rotation is best implemented by first rotating the image by 90° noted above and then applying a –3° rotation by using the 2-pass technique. These difficulties are resolved more naturally in a recent paper, described later, that demonstrates a separable technique for implementing arbitrary spatial lookup tables [Wolberg 89b].

6.4.7. Fold over Problem

The 2-pass algorithm is particularly well-suited for mapping images onto surfaces with closed form solutions to auxiliary function $H$. For instance, texture mapping onto rectangles that undergo perspective projection was first shown to be 2-passable in [Catmull 80]. This was independently discovered by Evans and Gabriel at Ampex Corporation where the result was implemented in hardware. The product was a real-time video effects generator called ADO (Ampex Digital Optics). It has met with great success in the television broadcasting industry where it is routinely used to map images onto rectangles in 3-space and move them around fluidly. Although the details of their design are not readily available, there are several patents documenting their invention [Bennett 84a, 84b, Gabriel 84].

The process is more complicated for surfaces of higher order, e.g., bilinear, biquadratic, and bicubic patches. Since these surfaces are often nonplanar, they may be self-occluding. This has the effect of making $F$ or $G$ become multi-valued at points where the image folds upon itself, a problem known as foldover.
Foldover can occur in either of the two passes. In the vertical pass, the solution for single folds in $G$ is to compute the depth of the vertical scanline endpoints. At each column, the endpoint which is furthest from the viewer is transformed first. The subsequent closer points along the vertical scanline will obscure the distant points and remain visible. Generating the image in this back-to-front order becomes more complicated for surfaces with more than one fold. In the general case, this becomes a hidden surface problem.

This problem can be avoided by restricting the mappings to be nonfolded, or single-valued. This simplification reduces the warp to one that resembles those used in remote sensing. In particular, it is akin to mapping images onto distorted planar grids where the spatial transformation is specified by a polynomial transformation.

Once we restrict patches to be nonfolded, only one solution is valid. This means that only one $u$ on each horizontal scanline can map to the current vertical scanline. We cannot attempt to use classic techniques to solve for $H$ because $n$ solutions may be obtained for an $n^{th}$-order surface patch. Instead, we find a solution $u = H_x(0)$ for the first horizontal scanline. Since we are assuming smooth surface patches, the next adjacent scanline can be expected to lie in the vicinity. The Newton-Raphson iteration method can be used to solve for $H_x(1)$ using the solution from $H_x(0)$ as a first approximation (starting value). This exploits the spatial coherence of surface elements to solve the inverse problem at hand.

The complexity of this problem can be reduced at the expense of additional memory. The need to evaluate $H$ can be avoided altogether if we make use of earlier computations. Recall that the values of $u$ that we now need in the second pass were already computed in the first pass. Thus, by introducing an auxiliary frame buffer to store these $u$'s, $H$ becomes available by trivial lookup table access.

In practice, there may be many $u$'s mapping onto the unit interval between $x$ and $x+1$. Since we are only interested in the inverse projection of integer values of $x$, we compute $x$ for a dense set of equally spaced $u$'s. When the integer values of two successive $x$’s differ, we take one of the two following approaches.

Iterate on the interval of their projections $u_i$ and $u_{i+1}$, until the computed $x$ is an integer.

Approximate $u$ by $u = u_i + a (u_{i+1} - u_i)$ where $a = x - x_i$.

The computed $u$ is then stored in the auxiliary frame buffer at location $x$.

6.5. 2-Pass Mesh Warping

The 2-pass algorithm formulated in [Catmull 80] has been demonstrated for warps specified by closed-form mapping functions. Another equally important class of warps are defined in terms of piecewise continuous mapping functions. In these instances, the input and output images can each be partitioned into a mesh of patches. Each patch delimits an image region over which a continuous mapping function applies. Mapping between both images now becomes a matter of transforming each patch onto its counterpart in the second image, i.e., mesh warping. This approach, typical in remote sensing, is appropriate for applications requiring a high degree of user interaction. By moving vertices in a mesh, it is possible to define arbitrary mapping functions with local control. In this section, we will investigate the use of the 2-pass technique for mesh warping. We begin with a motivation for mesh warping.
and then proceed to describe an algorithm that has been used to achieve fascinating special effects.

6.5.1. Special Effects

The 2-pass mesh warping algorithm described in this section was developed by Douglas Smythe at Industrial Light and Magic (ILM), the special effects division of Lucasfilm Ltd. This algorithm has been successfully used at ILM to generate special effects for the motion pictures *Willow*, *Indiana Jones and The Last Crusade*, and *The Abyss* (Winner of the 1990 Academy Award for special effects). The algorithm was originally conceived to create a sequence of transformations: goat→ ostrich→ turtle→ tiger→ woman. In this context, a transformation refers to the geometric metamorphosis of one shape into another. It should not be confused with a cross-dissolve operation which simply blends one image into the next via point-to-point color interpolation. Although a cross-dissolve is one element of the effect, it is only invoked once the shapes are geometrically aligned to each other.

In the world of special effects, there are basically three approaches that may be taken to achieve such a cinematic illusion. The conventional approach makes use of physical and optical techniques, including air bladders, vacuum pumps, motion-control rigs, and optical printing. The next two approaches make use of computer processing. In particular, they refer to computer graphics and image processing, respectively.

In computer graphics, each of the animals would have to be modeled as 3-D objects and then be accurately rendered. The transformation would be the result of smoothly animating the interpolation between the models of the animals. There are several problems with this approach. First, computer-generated models that accurately resemble the animals are difficult to produce. Second, any technique to accurately render fur, feathers, and skin would be prohibitively expensive. On the other hand, the benefit of computer graphics in this application is the complete control that the director may have over each possible aspect of the illusion.

Image processing proves to be the best alternative. It avoids the problem of modeling the animals by starting directly from images of real animals. The transformation is now achieved by means of digital image warping. Whereas computer graphics renders a set of deforming 3-D models, image processing deforms the images themselves. This conforms with the notion that it is easier to create an effective illusion by distorting reality rather than synthesizing it from nothing. The roles of the two computer processing approaches in creating illusions are depicted in Figure 6.12.
Figure 6.12: Two approaches to computer-generated special effects.

The drawback with the image processing approach is the lack of control. Since the distortions act upon what is already present in the image, the input scenes must be carefully selected and choreographed. For instance, movement of an animal may cause difficulties in alignment with the next animal in the sequence, or present problems with occlusion and shadows. Nevertheless, the benefits of the image processing approach to special effects greatly outweigh its drawbacks.

Special effects is one of many applications in which the mapping functions are conveniently specified by laying down two sets of control points: one set to select points from the input image, and a second set to specify their correspondence in the output image. Since the mapping function is defined only at these discrete points, it becomes necessary for us to determine the mapping function over all points in order to perform the warp. That is, given \( X(u_i, v_i) \) and \( Y(u_i, v_i) \) for \( 1 \leq i \leq N \), we must derive \( X \) and \( Y \) for all the \((u, v)\) points. This is reminiscent of the surface interpolation paradigm presented in Chapter 3, where we formulated this problem as an interpolation of two surfaces \( X \) and \( Y \) given an arbitrary set of points \((u_i, v_i, x_i)\) and \((u_i, v_i, y_i)\) along them.

In that chapter, we considered various surface interpolation methods, including piecewise polynomials defined over triangulated regions, and global splines. The primary complication lied in the irregular distribution of points. A great deal of simplification is possible when a regular structure is imposed on the points. A rectilinear grid of \((u, v)\) lines, for instance, facilitates mapping functions comprised of rectangular patches. Since many points of interest do not necessarily lie on a rectilinear grid, we allow the placement of control points to coincide with the vertices of a nonuniform mesh. This extension is particularly straightforward since we can consider a mesh to be a parametric grid. In this manner, the control points are indexed by integer \((u, v)\) coordinates that now serve as pointers to the true position, i.e., there is an added level of indirection. The parametric grid partitions the image into a contiguous set of patches, as shown in Fig. 6.13. These patches can now be fitted with a bivariate function to realize a (piecewise) continuous mapping function.
6.5.2. Description of the Algorithm

The algorithm in [Smythe 90] accepts a source image and two 2-D arrays of coordinates. The first array, $S$, specifies the coordinates of control points in the source image. The second array, $D$, specifies their corresponding positions in the destination image. Both $S$ and $D$ must necessarily have the same dimensions in order to establish a one-to-one correspondence. Since the points are free to lie anywhere in the image plane, the coordinates in $S$ and $D$ are real-valued numbers.

The 2-D arrays in which the control points are stored impose a rectangular topology to the mesh. Each control point, no matter where it lies, is referenced by integer indices. This permits us to fit any bivariate function to them in order to produce a continuous mapping from the discrete set of correspondence points given in $S$ and $D$. The only constraint is that the meshes defined by both arrays be topologically equivalent, i.e., no folding or discontinuities. Therefore, the entries in $D$ are coordinates that may wander as far from $S$ as necessary, as long as they do not cause self-intersection. Figure 6.14 shows vertices of overlaid meshes $S$ and $D$. 

![Figure 6.14 Mesh of patches.](image)
The 2-pass mesh warping algorithm is similar in spirit to the 2-pass Catmull-Smith algorithm described earlier. The first pass is responsible for resampling each row independently. It maps all \((u,v)\) points to their \((x,v)\) coordinates in the intermediate image \(I\), thereby positioning each input point into its proper output column. In this manner, the intermediate image \(I\) is defined whose \(x\)-coordinates are the same as those in \(D\) and whose \(y\)-coordinates are taken from \(S\) (see Figure 6.15). The second pass then resamples each column in \(I\), mapping every \((x,v)\) point to its final \((x,y)\) position. In this manner, each point can now lie in its proper row, as well as column. We now describe both passes in more detail.

### 6.5.2.1. First Pass

The first pass requires the output \(x\)-coordinates of all pixels along each row. This information is derived directly from \(S\) and \(I\) in a two-phase process. We let \(S\) and \(I\) each have \(h\) rows and \(w\) columns. In practice, these dimensions are much smaller than those of the source image. For reasons described later, the source, intermediate, and destination images all share the same dimensions, \(h_{in} \times w_{in}\). Since the control point coordinates are only available at sparse positions, the role of the two-phase process is to spread this data throughout the source image. This makes it possible for all pixels to have the \(x\)-coordinate data necessary for resampling along the horizontal direction.

![Figure 6.15: Intermediate grid \(I\) for \(S\) and \(D\) [Smythe 90].](image)

In the first phase, each column in \(S\) and \(I\) is fitted with an interpolating spline through the \(x\)-coordinates of the control points. A Catmull-Rom spline was used in [Smythe 90] because it offers local control, although any spline would suffice. These vertical splines are then sampled as they cross each row, creating tables \(T_S\) and \(T_I\) of dimension \(h_{in}\) (see Figure 6.16). This effectively scan converts each patch boundary in the vertical direction, spreading sparse coordinate data across all rows.
The second phase must now interpolate this data along each row. In this manner, each row of width $w$ is resampled to $w_0$, the width of the input image. Since $T_S$ and $T_I$ have the same number of columns, every row in $S$ and $I$ has the same number of vertical patch boundaries; only their particular x-intercepts are different. For each patch interval that spans horizontally from one x-intercept to the next, a normalized index is defined. As we traverse each row in the second phase, we determine the index at every integer pixel boundary in $I$ and we use that index to sample the corresponding spline segment in $S$. In this manner, the second phase has effectively scan converted $T_S$ and $T_I$ in the horizontal direction, while identifying corresponding intervals in $S$ and $I$ along each row. This form of inverse point sampling, used together with box filtering, achieved the high-quality warps in the feature films cited earlier.

For each pixel $P$ in intermediate image $I$, box filtering amounts to weighting all input contributions from $S$ by their fractional coverage to $P$. For minification, the value $P$ is evaluated as a weighted sum from $x_0$ to $x_1$, the leftmost and rightmost positions in $S$ that are the projections (inverse mappings) of the left and right integer-valued boundaries of $P$:

$$P = \frac{\sum_{x=x_0}^{x=x_1} k_x S_x}{x_1 - x_0}$$

where $k_x$ is the scale factor of source pixel $S_x$, and the subscript $x$ denotes the integer-valued index that lies in the range $\text{floor}(x_0) \leq x \leq \text{ceil}(x_1)$. The scale factor $k_x$ is defined to be

$$k_x = \begin{cases} 
\text{ceil}(x) - x_0 & \text{floor}(x) < x_0 \\
1 & x_0 \leq x \leq x_1 \\
x_1 - \text{floor}(x) & \text{ceil}(x) > x_1
\end{cases}$$

Figure 6.16: Creating tables $T_S$ and $T_I$ [Smythe 90].
The first condition in Eq. 6.25 deals with the partial contribution of source pixel $S_x$ when it is clipped on the left edge of the input interval. The second condition applies when $S_x$ lies totally embedded between $x_0$ and $x_1$. The final condition deals with the rightmost pixel in the interval in $S$ that may be clipped.

The summation in Eq. 6.24 is avoided upon magnification. Instead, some interpolation scheme is applied. Linear interpolation is a popular choice due to its simplicity and effectiveness over a reasonable range of magnification factors.

Figure 6.17 shows the effect of applying the mesh in Figure 6.13 to the Checkerboard image. In this case, $S$ contains coordinates that lie on the rectilinear grid, and $D$ contains the mesh vertices of Figure 6.13. Notice that resampling is restricted to the horizontal direction. The second pass will now complete the warp by resampling in the vertical direction.

Figure 6.17: Warped Checkerboard image after first pass.

### 6.5.2.2. Second Pass

The second pass is virtually identical to that of the first pass. This time, however, we begin by fitting an interpolating spline through the $y$-coordinates of the control points in each row of $I$ and $D$. These horizontal splines are then sampled as they cross each column, creating tables $T_I$ and $T_D$ of height $h$ and width $w_{in}$. Interpolating splines are then fitted to each column in these tables. This facilitates vertical resampling to occur in much the same way as horizontal resampling was performed in the first pass. The collection of vertical splines fitted through $S$ and $I$ in the first pass, together with the horizontal splines fitted through $I$ and $D$ in the second pass, are shown in Figure 6.18. The warped Checkerboard image, after it comes out of the second pass, is shown in Figure 6.19.

### 6.5.2.3. Discussion

The algorithm as presented above requires that all four edges of $S$ and $D$ be frozen. This means that the first and last rows and columns all remain intact throughout the warp. As we shall discover shortly, this seemingly limiting constraint has important implications in the simplicity of the algorithm. Furthermore, if we consider the border to lie far beyond the region of interest in the image, then the frozen edge constraint proves to have little consequence on the class of warps that can be achieved.
In examining this 2-pass mesh warping algorithm more closely, it is worthwhile to compare it to the 2-pass Catmull-Smith transform. In the latter case, the forward map was given only in terms of the input coordinates $u$ and $v$. Although nonfrozen edges were allowed, this formulation placed a heavy burden in computing an inverse function after the first pass. After all, after the first pass warps the $(u,v)$ data into the $(x,v)$ coordinate system, direct access into mapping function $Y(u,v)$ is no longer possible without the existence of an inverse. The 2-pass mesh warping algorithm, on the other hand, defines the forward mapping function in terms of two tables of control point coordinates. This formulation permits a straightforward use of interpolating splines, as described for the two-phase first pass.

Although the first pass could have permitted the image boundaries to be nonfrozen, difficulties would have surfaced for an equally simple second pass. In particular, each column in $I$ and $D$ would no longer be guaranteed of sharing the same number of horizontal splines that can be fitted in the vertical direction by just one spline. A single vertical spline in the second phase of the second pass proves most useful. It avoids boundary effects around discontinuities that would otherwise arise as a nonfrozen, possibly wiggly, edge is scan

---

**Figure 6.18:** Splines fitted through $S$, $I$, and $D$ [Smythe 90].

**Figure 6.19:** Warped Checkerboard image after second pass.
converted in the vertical direction. Clearly, slicing such an edge in the vertical direction would produce alternating intervals that lie inside and outside the mesh. Therefore, the frozen edge constraint is placed in order to make the process symmetric among the two passes, and simplify filtering problems in the second pass.

Like the Catmull-Smith algorithm, there is no graceful solution presented to the foldover problem. In fact, the user is refrained from creating such warps. Furthermore, there is no provision for handling the bottleneck problem. As a result, it is possible for distortion to arise when the warps contain large rotational components. This places additional constraints on the user.

6.5.3. Examples

The 2-pass mesh warping algorithm described in this section has been used to produce many fascinating warps. The primary application has been in the transformation between objects. Consider two image sequences of equal length, $F_1(t)$ and $F_2(t)$, where $t$ varies from 0 to $N$. They are each moving images depicting two creatures, say an ostrich and a turtle. The original state of the metamorphosis begins at $F_1(0)$, with the first image of the ostrich. As $t$ approaches $N$, the output $H(t)$ progresses towards $F_2(N)$, an uncorrupted image of the turtle at the end of the sequence. Along the way, the output is produced by warping corresponding images of $F_1(t)$ and $F_2(t)$ in some desired way, as specified by their respective control grids. As a matter of convenience, we shall drop the argument $t$ from the notation in the remaining discussion. It should be understood that when we speak of the image or grid sequences, we refer to one instance at a time.

For each image in the two sequences, grids $G_1$ and $G_2$ are defined such that each point in $G_1$ lies over the same feature in $F_1$ as the corresponding point in $G_2$ lies over $F_2$. $F_1$ is then warped into a new picture $F_{1w}$ by using source grid $G_1$ and destination grid $G_1$, a grid whose coordinates are at some intermediate stage between $G_1$ and $G_2$. Similarly, $F_2$ is warped into a new image $F_{2w}$ using source grid $G_2$ and destination grid $G_2$, the same grid that was used in making $F_{1w}$. In this manner, $F_{1w}$ and $F_{2w}$ are different creatures stretched into geometric alignment. A cross-dissolve between them now yields a frame in the transformation between the two creatures. This process is depicted in Figure 6.20, where boldface is used to depict the key frames. These are frames that the user determines to be important in the image sequence. Control grids $G_1$ and $G_2$ are precisely established for these key frames. All intermediate images then get their grid assignments via interpolation.
One key to making the transformations interesting is to apply a different rate of transition between $F_1$ and $F_2$ when creating $G_t$, so different parts of the creature can move and change at different rates.

Figure 6.21 shows four frames of Raziel's transformation sequence from Willow that warps an ostrich into a turtle. The more complete transformation process, including warps between a tiger and a woman, is depicted in the image on the front cover. The reader should note that the warping program is applied only to the transforming creatures. They are computed separately with a black background. The warped results are then optically composited with the background, the magic wand, and some smoke.

The same algorithm was also used as an integral element in other special effects where geometric alignment was a critical task. This appeared in the movie Indiana Jones and the Last Crusade in the scene where an actor underwent physical decomposition, as shown in Figure 6.22. In order to create this illusion, the ILM creature shop constructed three motion-controlled puppet heads. Each was in a progressively more advanced stage of decomposition. Mechanical systems were used to achieve particular effects such as receding eyeballs and shriveling skin. Each of these was filmed separately, going through identical computer-controlled moves. The warping process was used to ensure a smooth and undetectable transition between the different sized puppet heads and their changing facial features and receding hair. This appears to be the first time that a feature film sequence was entirely digitally composited from film elements, without the use of an optical printer [Hu 90].

In The Abyss, warping was used for facial animation. Several frames of a face were scanned into the computer by using a Cyberware 3D video laser input system. The resulting images consist of range data denoting the distance of each point from the sensor. Although this data can be used to directly generate 3D models of a human face, such models prove cumbersome for creating realistic facial animations with effective facial expressions. As a result, the range data is left in its 2D form and manipulated with image processing tools, including the 2-pass mesh warping algorithm. Each of the facial images is used as a key frame in the animation process. Meshes are used to define and control a complex warp in each successive key frame. In this manner, an animation is created in which one facial expression naturally moves into another. After the frames have been warped in 2D, they are rendered as 3D surfaces for viewing [Anderson 90].

Two additional examples of mesh warping are shown in Figure 6.24 and Figure 6.25. They serve to further highlight the wide range of transformations possible with this approach.

6.5.4. Source Code

A C program that implements the 2-pass mesh warping algorithm is given below. It warps input image IN into the output image OUT. Both IN and OUT have the same dimensions: height IN_h (rows) and width IN_w (columns). The images are assumed to have a single channel consisting of byte-sized pixels, as denoted by the unsigned char data type. Multi-channel images (e.g., color) can be handled by sending each channel through the program independently.
The source mesh is supplied through the 2-D arrays Xs and Ys. Similarly, the destination mesh coordinates are contained in X_d and Y_d. Both mesh tables accommodate double-precision numbers and share the same dimensions: height T_h and width T_w.

The program makes use of ispline_gen and resample_gen, two functions defined elsewhere in this book. Function ispline_gen is used here to fit an interpolating cubic spline through the mesh coordinates. Since it can fit a spline through the data and resample it at arbitrary positions, ispline_gen is also used for scan conversion. This is simply achieved by resampling the spline at all integer coordinate values along a row or column. The program listing for ispline_gen can be found in Appendix 2. The function takes six arguments, i.e., ispline_gen (A,B,C,D,E,F). Arguments A and B are pointers to a list of (x,y) data points whose length is C. The spline is resampled at F positions whose coordinates are contained in D. The results are stored in E.
Once the forward mapping function is defined, function resample_gen is used to warp the data. Although an inverse mapping scheme was used in [Smythe 90], we choose a forward mapping formulation because it conveniently allows us to demonstrate algorithms derived earlier. Although the segment of code given there is limited to processing horizontal scanlines, we now treat the more general case that includes vertical scanlines as well. This is accommodated with the use of an additional parameter that specifies the offset from one pixel to the next. Horizontal scanlines have a pixel-to-pixel offset of one, while vertical scanlines have an offset equal to the width of a row. The function resample_gen (A,B,C,D,E) applies the mapping function A to input scanline B, generating output C. The input (and output) dimension is D and the inter-pixel offset is E. The function performs linear interpolation for magnification and box filtering for minification. This is equivalent to the reconstruction and antialiasing methods used in [Smythe 90]. Superior filters can be added within this framework by incorporating the results of Chapters 5 and 6.

Figure 6.22: Donovan's destruction sequence from Indiana Jones and the Last Crusade.

Figure 6.23 Facial Animation from the Pseudopod sequence in The Abyss.
Figure 6.24 A warped image of Piazza San Marco.

Figure 6.25 A caricature of Albert Einstein.
/****************************************************
Two-pass mesh warping based on algorithm in [Smythe 90].
Input image IN has height IN_h and width IN_w.
Xs,Ys contain the x,y coordinates of the source mesh.
Xd,Yd contain the x,y coordinates of the destination mesh.
Their height and width dimensions are T_h and T_w.
The output is stored in OUT. Due to the frozen edge
assumption, OUT has same dimensions as IN.
/*****************************/
warp_mesh(IN, OUT, Xs, Ys, Xd, Yd, IN_h, IN_w, T_h, T_w)
unsigned char *IN, *OUT;
double *Xs, *Ys, *Xd, *Yd;
int IN_h, IN_w, T_h, T_w;
{
    int a, b, x, y;
    unsigned char *src, *dst;
    double *x1,*y1,*x2, *y2,*xrow1,*yrow1,*xrow2,*yrow2,*map1,*map2,*indx,*Ts,
        *Ti,*Td;
    /* allocate memory for buffers: indx stores indices used to sample
     * splines;
     * xrow1, xrow2, yrow1, yrow2 store column data in row order for
     * ispline_gen();
     * map1, map2 store mapping functions computed in row order in
     * ispline_gen() #
     */
    a = MAX(IN_h, IN_w) +1;
b = sizeof(double);
    indx = (double *) calloc(a, b);
xrow1 = (double*) calloc(a, b);yrow1 = (double*) calloc(a, b);
xrow2 = (double *) calloc(a, b);yrow2 = (double *) calloc(a, b);
map1 = (double *) calloc(a, b);map2 = (double *) calloc(a, b);

    /* First pass (phase one): create tables Ts and Ti for x-intercepts of
     * vertical splines in S and I. Tables have T_w columns of height IN_h
     */
    Ts = (double *) calloc(T_w * IN_h, sizeof(double));
    Ti = (double *) calloc(T_w * IN_h, sizeof(double));
    for(y=0; y<IN_h; y++) indx[y] = y;

    for(x=0; x<T_w; x++){
        /* visit each vertical spline */
        /* store columns as rows for ispline_gen */
        for(y=0; y<IN_h; y++) {                /* y<T_h; y++ */
            xrow1[y] = Xs[y*T_w + x];yrow1[y] = Ys[y*T_w + x];
            xrow2[y] = Xd[y*T_w + x];yrow2[y] = Yd[y*T_w + x];
        }
        /* scan convert vertical splines of S and I */
        ispline_gen(yrow1, xrow1, T_h, indx, map1, IN_h);
        ispline_gen(yrow2, xrow2, T_h, indx, map2, IN_h);
        /* store resampled rows back into columns */
        for(y=0; y<IN_h; y++) {                /* y<T_h; y++ */
            Ts[y*T_w + x] = map1[y];
            Ti [y*T_w + x] = map2[y];
        }
    }

    /* First pass (phase two): warp x using Ts and Ti. TMP holds intermediate
     * image. */
    TMP = (unsigned char *) calloc(IN_h, IN_w);
    for(x=0; x<IN_w; x++) indx[x] = x;
    /*indices used to sample horizontal spline */
}
for(y=0; y<IN_h; y++) {
    /* visit each row */
    /* fit spline to x-intercepts; resample over all columns */
    x1 = &Ts[y * T_w];
    x2 = &Ti[y * T_w];
    ispline_gen(x1, x2, T_w, indx, map1, IN_w);
    /* resample source row based on map1 */
    src = &IN[y * IN_w];
    dst = &TMP[y * IN_w];
    resample_gen(map1, src, dst, w, 1);
}
/* free buffers */
cfree((char *) Ts);
cfree((char *) Ti);

/*
 * Second pass (phase one): create tables Ti and Td for y-intercepts of
 * horizontal splines in I and D. Tables have T_h rows of width IN_w
 */
Ti = (double *) calloc(T_h * IN_w, sizeof(double));
Td = (double *) calloc(T_h * IN_w, sizeof(double));
for(x=0; x<IN_w; x++) indx[x] = x;
/* indices used to sample horizontal splines */
for(y=0; y<T_h; y++) {
    /* visit each horizontal spline */
    /* scan convert horizontal splines in I and D */
    x1 = &Xs[y * T_w]; y1 = &Ys[y * T_w];
    x2 = &Xd[y * T_w]; y2 = &Yd[y * T_w];
    ispline_gen(x1, y1, T_w, indx, & Ti[y * IN_w], IN_w);
    ispline_gen(x2, y2, T_w, indx, &Td[y * IN_w], IN_w);
}
/* Second pass (phase two): warp y using Ti and Td */
for(y=0; y<IN_h; y++) indx[y] = y;
for(x=0; x<T_w; x++) {
    /* store column as row for ispline_gen */
    for(y=0; y<T_h; y++) {
        xrow1[y] = Ti[y * IN_w + x];
        yrow1[y] = Td[y * IN_w + x];
    }
    /* fit spline to y-intercepts; resample over all rows */
    ispline_gen(xrow1, yrow1, T_h, indx, map1, IN_h);
    /* resample intermediate image column based on map 1 */
    src = &TMP[x];
    dst = &OUT[x];
    resample_gen(map1, src, dst, IN_h, IN_w);
}
cfree((char *) TMP); cfree((char *) indx);
cfree((char *) Ti); cfree((char *) Td);
cfree((char *) xrow1); cfree((char *) yrow1);
cfree((char *) xrow2); cfree((char *) yrow2);
cfree((char *) map1); cfree((char *) map2);