Spatial Transformations

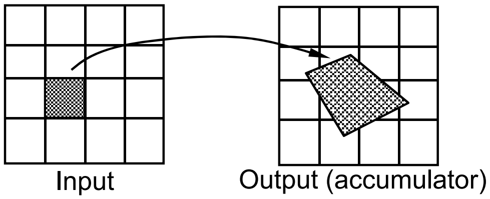
Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

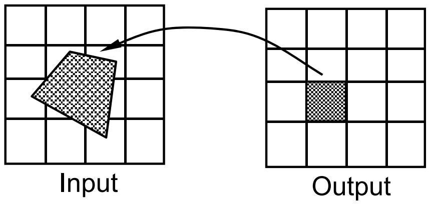
- In this lecture we review spatial transformations:
  - Forward and inverse mappings
  - Transformations
    - Linear
    - Affine
    - Perspective
    - Bilinear
  - Inferring affine and perspective transformations

Forward and Inverse Mappings

- A spatial transformation defines a geometric relationship between each point in the input and output images.
- Forward mapping: $[x, y] = [X(u,v), Y(u,v)]$
- Inverse mapping : $[u, v] = [U(x,y), V(x,y)]$
- Forward mapping specifies output coordinates *(x,y)* for each input point *(u,v)*.
- Inverse mapping specifies input point *(u,v)* for each output point *(x,y).*



Forward mapping



Inverse mapping

Linear Transformations

$$[x, y] = [u, v]\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

$$x = a_{11}u + a_{21}v$$

$$y = a_{12}u + a_{22}v$$

- Above equations are linear because they satisfy the following two conditions necessary for any linear function L(x):

1) L(x+y) = L(x) + L(y)
2) L(cx) = cL(x) for any scalar c and position vectors x and y.

- Note that linear transformation are a sum of scaled input coordinate: they do not account for simple translation.

We want:

$$x = a_{11}u + a_{21}v + a_{31}$$

$$y = a_{12}u + a_{22}v + a_{32}$$

$$[x, y] = [u, v, 1]\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$$

Homogeneous Coordinates

- To compute inverse, the transformation matrix must be square.
- Therefore,

$$[x, y, 1] = [u, v, 1]\begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix}$$

- All 2-D position vectors are now represented with three components: homogeneous notation.
- Third component (w) refers to the plane upon which the transformation operates.
- [u, v, 1] = [u, v] position vector lying on w=1 plane.
- The representation of a point in the homogeneous notation is no longer unique: [8, 16, 2] = [4, 8, 1] = [16, 32, 4].
- To recover any 2-D position vector *p[x,y]* from $p_h=[x', y', w']$, divide by the homogeneous coordinate w'.

$$[x, y] = \begin{bmatrix} \frac{x'}{w'} & \frac{y'}{w'} \end{bmatrix}$$

Affine Transformations (1)

$$[x, y, 1] = [u, v, 1] \begin{bmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{bmatrix}$$

$$\text{Translation} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_u & T_v & 1 \end{bmatrix}$$

$$\text{Rotation} \rightarrow \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Scale} \rightarrow \begin{bmatrix} S_u & 0 & 0 \\ 0 & S_v & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Shear rows} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ H_u & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Shear columns} \rightarrow \begin{bmatrix} 1 & H_u & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

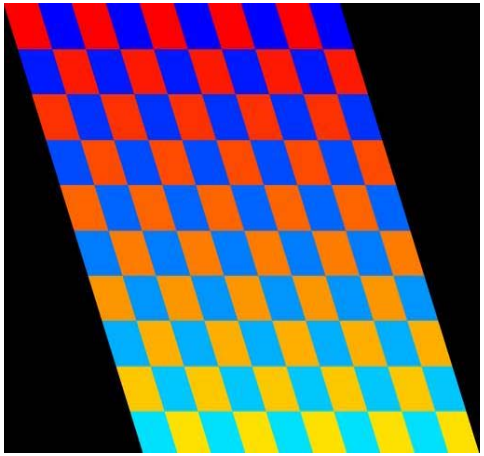
Affine Transformations (2)

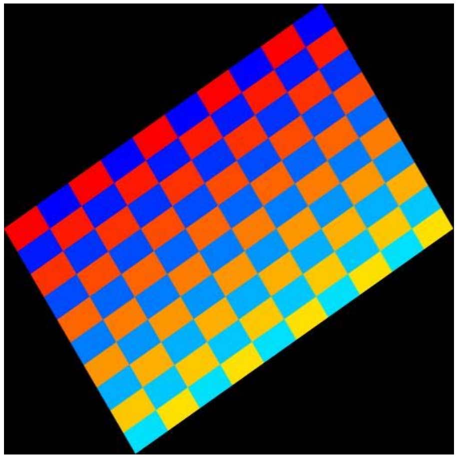
- Affine transformation have 6 degrees of freedom: a_{11}, a_{21}, a_{31}, a_{12}, a_{22}, a_{23}.
- They can be inferred by giving the correspondence of three 2-D points between the input and output images. That is,

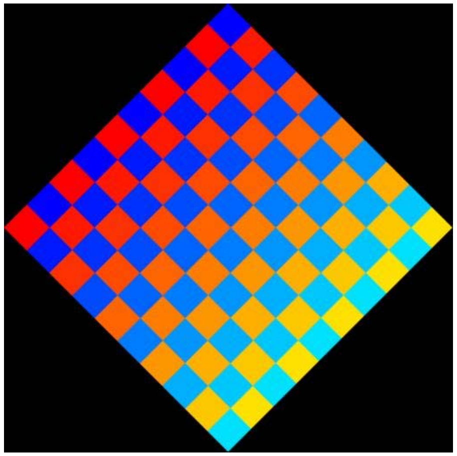
$$\left.\begin{array}{lcl}(u_1,v_1) & \rightarrow & (x_1,y_1)\\(u_2,v_2) & \rightarrow & (x_2,y_2)\\(u_3,v_3) & \rightarrow & (x_3,y_3)\end{array}\right\}\ 6\,\text{constraints}: (3\,\text{for}\ u \rightarrow x,\ 3\,\text{for}\ v \rightarrow y)$$

- All points lie on the same plane.
- Affine transformations map triangles onto triangles.

Affine Transformations (3)


Skew (shear)


Rotation/Scale


Rotation

Perspective Transformations (1)

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

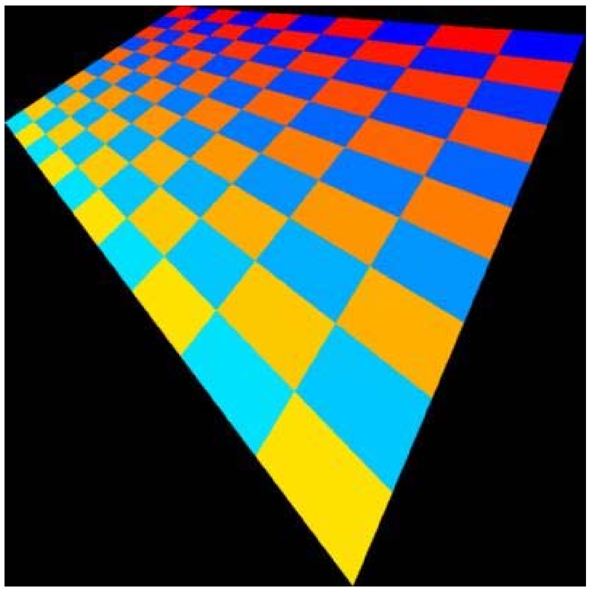
$$x = \frac{x'}{w'} = \frac{a_{11}u + a_{21}v + a_{31}}{a_{13}u + a_{23}v + a_{33}}$$

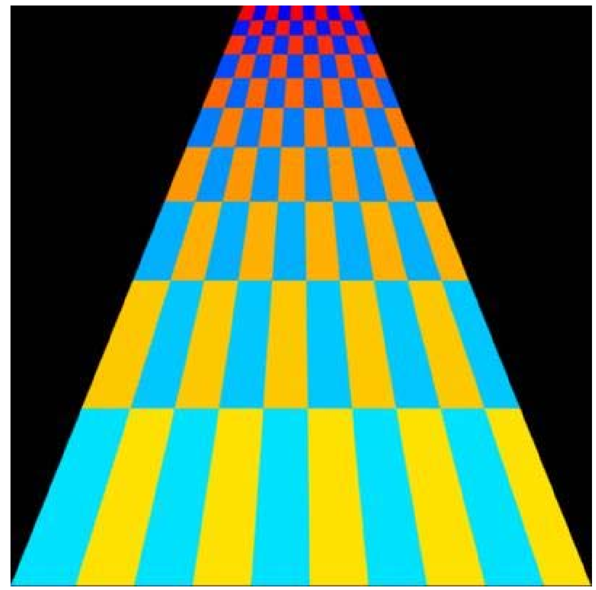
$$y = \frac{y'}{w'} = \frac{a_{12}u + a_{22}v + a_{32}}{a_{13}u + a_{23}v + a_{33}}$$

$$\begin{bmatrix} a_{13} \\ a_{23} \\ a_{33} \end{bmatrix} \text{not necessarily} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \text{as in affine transformations}$$

- Without loss of generality, we set a_{33}=1.
- This yields 8 degrees of freedom and allows us to map planar quadrilaterals to planar quadrilaterals (correspondence among 4 sets of 2-D points yields 8 coordinates).
- Perspective transformations introduces foreshortening effects.
- Straight lines are preserved.

Perspective Transformations (2)



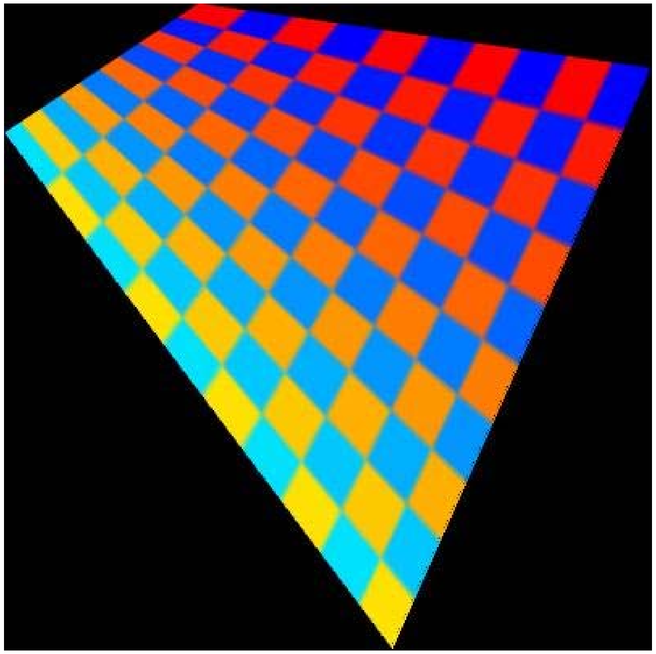


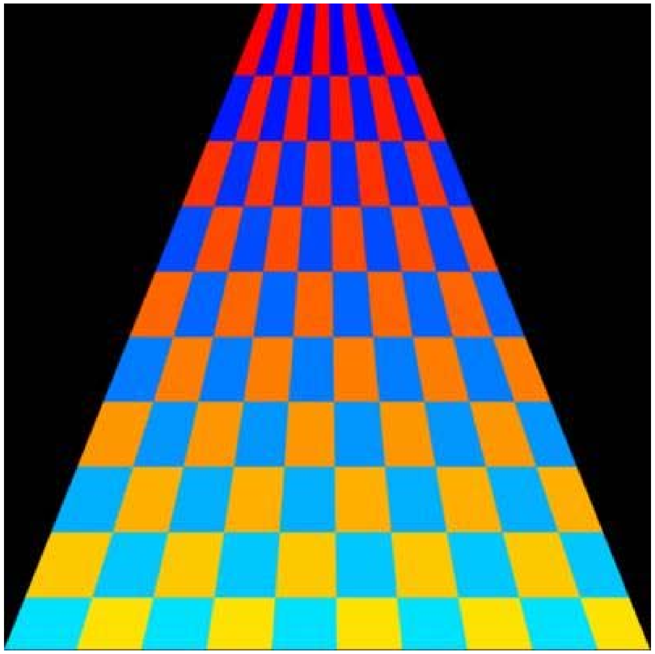
Bilinear Transforms (1)

$$[x, y] = [uv, u, v, 1]\begin{bmatrix} a_3 & b_3 \\ a_2 & b_2 \\ a_1 & b_1 \\ a_0 & b_0 \end{bmatrix}$$

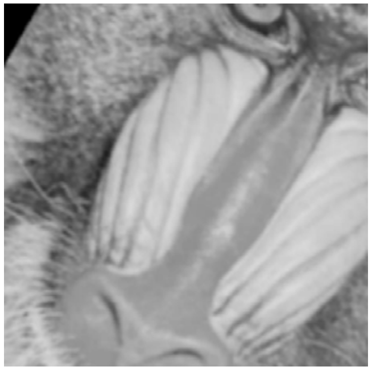
- 4-corner mapping among nonplanar quadrilaterals (2nd degree due to *uv* factors).
- Conveniently computed using separability (see p. 57-60).
- Preserves spacing along edges.
- Straight lines in the interior no longer remain straight.

Bilinear Transforms (2)

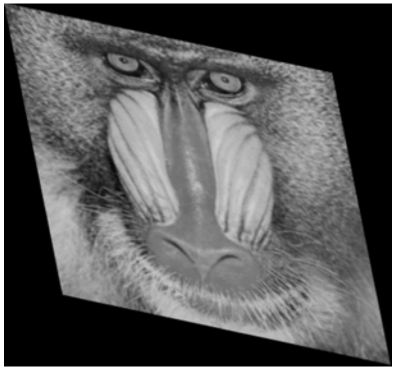




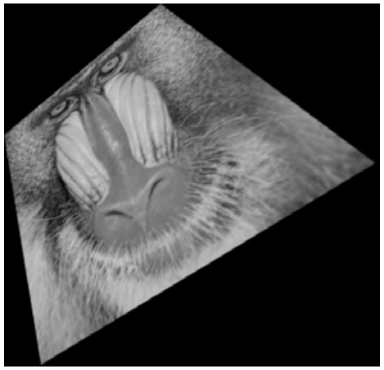
Examples



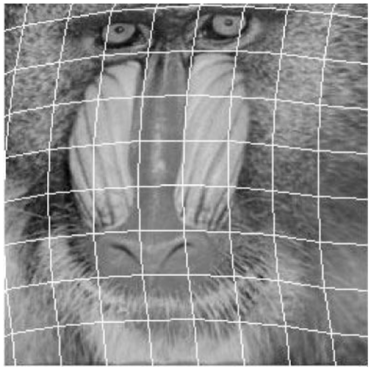
Similarity transformation (RST)



Affine transformation



Perspective transformation



Polynomial transformation

Scanline Algorithms

Prof. George Wolberg
Dept. of Computer Science
City College of New York

Objectives

- In this lecture we review scanline algorithms:
  - Incremental texture mapping
  - 2-pass Catmull-Smith algorithm
    - Rotation
    - Perspective
  - 3-pass shear transformation for rotation
  - Morphing

Catmull-Smith Algorithm

- Two-pass transform
- First pass resamples all rows: $[u, v] \rightarrow [x, v]$

  $[x, v] = [F_v(u), v]$ where $F_v(u) = X(u, v)$ is the forward mapping fct
- Second pass resamples all columns: $[x, v] \rightarrow [x, y]$

  $[x, y] = [x, G_x(v)]$ where $G_x(v)=Y(H_x(v), v)$
- $H_x(v)$ is the inverse projection of x', the column we wish to resample.
- It brings us back from *[x, v]* to *[u, v]* so that we can directly index into Y to get the destination y coordinates.

Example: Rotation (1)

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

Pass 1: $[x, v] = [u\cos\theta - v\sin\theta, v]$

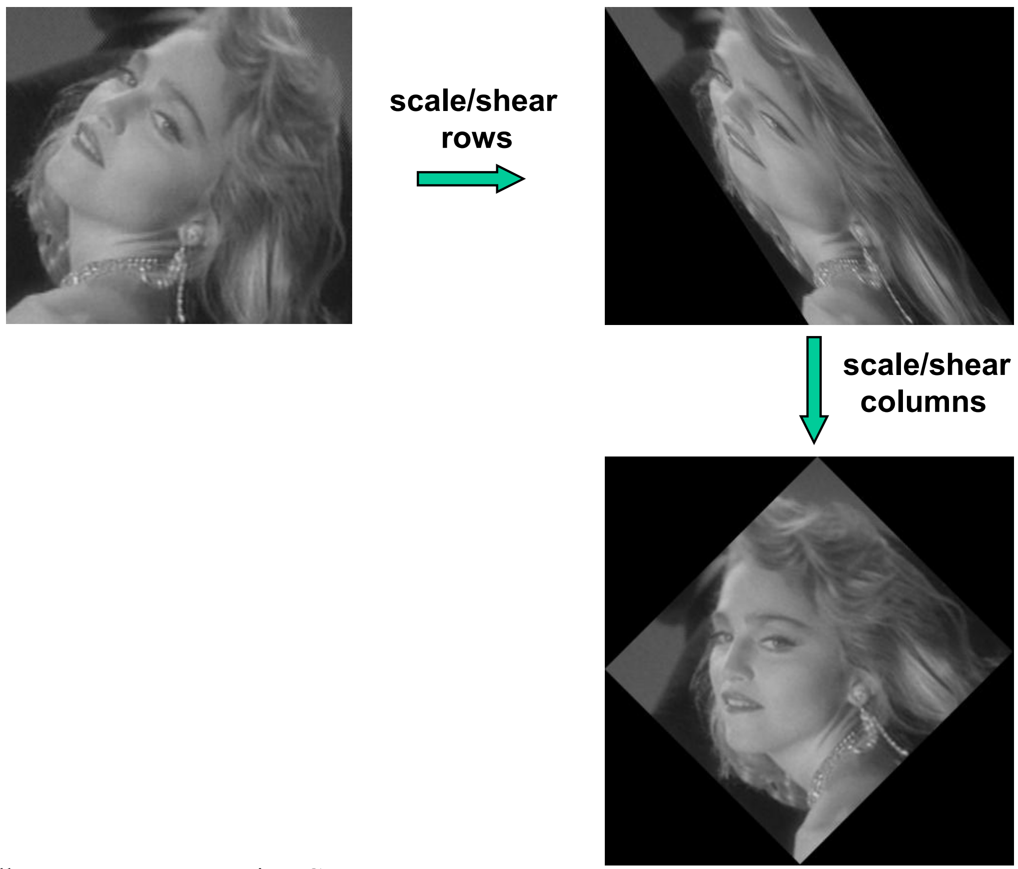
Pass 2: a) Compute $H_x(v)$. Recall that $x = u\cos\theta - v\sin\theta$

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

b) Compute $G_x(v)$. Substitute $H_x(v)$ into $y = u\sin\theta + v\cos\theta$

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

2-Pass Rotation



3-Pass Rotation Algorithm

- Rotation can be decomposed into two scale/shear matrices.

$$R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ -\sin\theta & 1 \end{bmatrix} \begin{bmatrix} 1 & \tan\theta \\ 0 & \dfrac{1}{\cos\theta} \end{bmatrix}$$

- Three pass transform uses on shear matrices.

$$R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -\tan\left(\dfrac{\theta}{2}\right) & 1 \end{bmatrix} \begin{bmatrix} 1 & \sin\theta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\tan\left(\dfrac{\theta}{2}\right) & 1 \end{bmatrix}$$

- Advantage of 3-pass: no scaling necessary in any pass.

3-Pass Rotation

Software Implementation

- The following slides contain code for **`initMatrix.c`** to produce a 3x3 perspective transformation matrix from a list of four corresponding points (e.g. image corners).
- That matrix is then used in **`perspective.c`** to resample the image.
- The code in **`resample.c`** performs the actual scanline resample.

initMatrix.c (1)

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * initMatrix:
 *
 * Given Icorr, a list of the 4 correspondence points for the corners
 * of image I, compute the 3x3 perspective matrix in Imatrix.
 */
void initMatrix(imageP I, imageP Icorr, imageP Imatrix)
{
  int     w,  h;
  float  *p, *a, a13, a23;
  float  x0, x1, x2, x3;
  float  y0, y1, y2, y3;
  float  dx1, dx2, dx3, dy1, dy2, dy3;

  /* init pointers */
  a = (float *) Imatrix->buf[0];
  p = (float *)   Icorr->buf[0];

  /* init u,v,x,y vars and print them */
  x0 = *p++;     y0 = *p++;
  x1 = *p++;     y1 = *p++;
  x2 = *p++;     y2 = *p++;
  x3 = *p++;     y3 = *p++;
```

initMatrix.c (2)

```
w = I->width;
h = I->height;
UI_printf("\nCorrespondence points:\n");
UI_printf("%4d %4d %6.1f %6.1f\n", 0, 0, x0, y0);
UI_printf("%4d %4d %6.1f %6.1f\n", w, 0, x1, y1);
UI_printf("%4d %4d %6.1f %6.1f\n", w, h, x2, y2);
UI_printf("%4d %4d %6.1f %6.1f\n", 0, h, x3, y3);

/* compute auxiliary vars */
dx1 = x1 - x2;
dx2 = x3 - x2;
dx3 = x0 - x1 + x2 - x3;
dy1 = y1 - y2;
dy2 = y3 - y2;
dy3 = y0 - y1 + y2 - y3;
```

initMatrix.c (3)

```
  /* compute 3x3 transformation matrix:
   * a0 a1 a2
   * a3 a4 a5
   * a6 a7 a8
   */
  a13  = (dx3*dy2 - dx2*dy3) / (dx1*dy2 - dx2*dy1);
  a23  = (dx1*dy3 - dx3*dy1) / (dx1*dy2 - dx2*dy1);
  a[0] = (x1-x0+a13*x1) / w;
  a[1] = (y1-y0+a13*y1) / w;
  a[2] = a13 / w;
  a[3] = (x3-x0+a23*x3) / h;
  a[4] = (y3-y0+a23*y3) / h;
  a[5] = a23 / h;
  a[6] = x0;
  a[7] = y0;
  a[8] = 1;
}
```

perspective.c (1)

```
#define X(A, U, V)       ((A[0]*U + A[3]*V + A[6]) / (A[2]*U + A[5]*V + A[8]))
#define Y(A, U, V)       ((A[1]*U + A[4]*V + A[7]) / (A[2]*U + A[5]*V + A[8]))
#define H(A, X, V)       ((-(A[5]*V+A[8])*X+ A[3]*V + A[6]) / (A[2]*X - A[0]))


/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * perspective:
 *
 * Apply a perspective image transformation on input I1.
 * The 3x3 perspective matrix is given in Imatrix.
 * The output is stored in I2.
 */
void perspective(imagP I1, imageP Imatrix, imageP I2)
{
  int    i, w, h, ww, hh;
  uchar  *p1, *p2;
  float  u, v, x, y, xmin, xmax, ymin, ymax, *a, *F;
  imageP II;

  w = I1->width;
  h = I1->height;
  a = (float *) Imatrix->buf[0];
```

perspective.c (2)

```
xmin = xmax = X(a, 0, 0);
x = X(a, w, 0); xmin = MIN(xmin, x);      xmax = MAX(xmax, x);
x = X(a, w, h); xmin = MIN(xmin, x);      xmax = MAX(xmax, x);
x = X(a, 0, h); xmin = MIN(xmin, x);      xmax = MAX(xmax, x);

ymin = ymax = Y(a, 0, 0);
y = Y(a, w, 0); ymin = MIN(ymin, y);      ymax = MAX(ymax, y);
y = Y(a, w, h); ymin = MIN(ymin, y);      ymax = MAX(ymax, y);
y = Y(a, 0, h); ymin = MIN(ymin, y);      ymax = MAX(ymax, y);

ww = CEILING(xmax) - FLOOR(xmin);
hh = CEILING(ymax) - FLOOR(ymin);

/* allocate mapping fct buffer */
x =  MAX(MAX(w, h), MAX(ww, hh));
F = (float *) malloc(x * sizeof(float));
if(F == NULL) IP_bailout("perspective: No memory");

/* allocate intermediate image */
II = IP_allocImage(ww, h, BW_TYPE);
IP_clearImage(II);
p1 = (uchar *) I1->buf[0];
p2 = (uchar *) II->buf[0];
```

perspective.c (3)

```
/* first pass: resample rows */
for(v=0; v<h; v++) {
      /* init forward mapping function F; map xmin to 0 */
      for(u=0; u< w; u++) F[(int) u] = X(a, u, v) - xmin;

      resample(p1, w, 1, F, p2);
      p1 += w;
      p2 += ww;
}

/* display intermediate image */
IP_copyImage(II, NextImageP);
IP_displayImage();

/* init final image */
IP_copyImageHeader(I1, I2);
I2->width  = ww;
I2->height = hh;
IP_initChannels(I2, BW_TYPE);
IP_clearImage(I2);
```

perspective.c (4)

```
/* second pass: resample columns */
for(x=0; x<ww; x++) {
        p1 = (uchar *) II->buf[0] + (int) x;
        p2 = (uchar *) I2->buf[0] + (int) x;

        /* skip past padding */
        for(v=0; v<h; v++,p1+=ww) {
                if(*p1) break;              /* check for nonzero pixel */
                u = H(a, (x+xmin), v);      /* else, if pixel is black */
                if(u>=0 && u<w) break;      /* then check for valid u  */
        }

        /* init forward mapping function F; map ymin to 0 */
        for(i=0; v<h; v++) {
                u = H(a, (x+xmin), v);
                u = CLIP(u, 0, w-1);
                F[i++] = Y(a, u, v) - ymin;
        }
        resample(p1, i, ww, F, p2);
}
IP_freeImage(II);
}
```

resample.c (1)

```
/* ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * Resample the len elements of src (with stride offst) into dst according
 * to the monotonic spatial mapping given in F (len entries).
 * The scanline is assumed to have been cleared earlier.
 */
void resample(uchar *src, int len, int offst, float *F, uchar *dst)
{
  int     u, uu, x, xx, ix0, ix1, I0, I1, pos;
  double  x0, x1, dI;

  if(F[0] < F[len-1])
        pos = 1;         /* positive output stride */
  else  pos = 0;         /* negative output stride */

  for(u=0; u<len-1; u++) {
        /* index into src */
        uu = u * offst;

        /* output interval (real and int) for input pixel u */
        if(pos) {/* positive stride */
                ix0 = x0 = F[u];
                ix1 = x1 = F[u+1];
                I0  = src[uu];
                I1  = src[uu+offst];
        }
```

resample.c (2)

```
else {  /* flip interval to enforce positive stride */
        ix0 = x0 = F[u+1];
        ix1 = x1 = F[u];
        I0  = src[uu+offst];
        I1  = src[uu];
}

/* index into dst */
xx = ix0 * offst;

/* check if interval is embedded in one output pixel */
if(ix0 == ix1) {
        dst[xx] += I0 * (x1-x0);
        continue;
}
/* else, input straddles more than one output pixel  */

/* left straddle */
dst[xx] += I0 * (ix0+1-x0);
```

resample.c (3)

```
        /* central interval */
        xx += offst;
        dI  = (I1-I0) / (x1-x0);
        for(x=ix0+1; x<ix1; x++,xx+=offst)
                dst[xx] = I0 + dI*(x-x0);

        /* right straddle */
        if(x1 != ix1)
                dst[xx] += (I0 + dI*(ix1-x0)) * (x1-ix1);
  }
}
```