# 6 Catalogue of Transformations

We conclude with a list of image transformations in the picture language for *popi*. We make full use of the defaults for indexing, so *old* [*x* , *y*] is abbreviated to *old* , and a default assignment *new* [*x* , *y*] = is always omitted. Most of the transformations were illustrated in Chapters **3** and **4**. At the completion of each command, or sequence of commands, listed here the edit buffer will contain the required image. The last few examples are transformations that are harder to express in *popi*, but that can be added as user-defined routines, as shown in Chapter **5** (see *Adding a Display Routine*).

**Making a Negative**

```
Z-old
```

**Logarithmic Correction**

```
Z∗log(old)/log(Z)
```

**Simulated Solarization**

```
(old > Z/2) ? Z-old : old
```

**Contrast Expansion and Normalization**

```
(Z*(old-L))/(H-L)
```

Assumes brightness values in the range *L* to *H* .

**Focus Restoration**

```
5∗old-old[x-1,y]-old[x+1,y]-old[x,y-1]-old[x,y+1]
```

## Blurring

```
(old[x-2,y-2]+old[x-1,y-2]+old[x,y-2]+old[x+1,y-2]+
 old[x-2,y-1]+old[x-1,y-1]+old[x,y-1]+old[x+1,y-1]+
 old[x-2,y  ]+old[x-1,y  ]+old[x,y  ]+old[x+1,y  ]+
 old[x-2,y+1]+old[x-1,y+1]+old[x,y+1]+old[x+1,y+1])/16
```

## Enlarging by an Integer Factor

```
(x<X/5 && y<Y/5) ? old[x*5,y*5] : 0
```
Enlarges by a factor of 5

## Shrinking by an Integer Factor

```
old[x/3,y/3]
```
Shrinks by a factor of 3

## Mirroring

```
old[X-x,y]
```

## Turning the Picture Upside Down

```
old[x,Y-y]
```

## Rotating by $90^o$ Clockwise

```
old[y,X-x]
```

## Rotating by $90^o$ Counterclockwise

```
old[Y-y,x]
```

## Averaging Three Images

```
(one+two+three)/3
```

## Weighted Average

```
(W1*one+W2*two+W3*three)/(W1+W2+W3)
```
Weight factors: W1, W2, W3

## Relief

```
old-old[x+2,y+2]
```

## Arbitrary Grid Transforms

See Chapter **4**, photos 10, 11, and 13.

```
old[x+(64-(old%16)*(old%16))/8, y+(64-(old/16)*(old/16))/8]
old^(old*(128-(x-128)*(x-128)-(y-128)*(y-128)))>>17
old[x+(x%32)-16, y]
```

## Transforms Using Trigonometric Functions

See Chapter **4**, photo 12, and Chapter **3**, expression 3.35.

```
    old[x+sin(X*x/4)*X/8, y+sin(Y*y/4)*Y/8]
    old[x+(X*cos(((x-X/2)*A)*2/X))/6, y]
```

### Transforms Using Polar Coordinates

See Chapter **4**, photos 1, 6, 8, 13, 14, and 19.

```
    old[sqrt(r*R), a]
    old[r, a+r/3]
    old[x(a) * X/A, y(r) * Y/R]
    old[x+((a+r/10)%32)-16, y]
    old[(r*r)/R, a]
    old[r, a+old[r,a]/8]
```

### Composites with Mattes

Straight composites of images, without averaging or fading, can be made in a number of different ways. In the examples below we use two images, named *I*1 and *I*2, and the corresponding mattes, *M*1 and *M*2. Assume that the images have the same background (e.g. portraits against a plain white background). At each point in the final image either *I*1 or *I*2 will be visible. All mattes are zero within the image area they define and *Z* outside of it. This type of image compositing is called the “Porter-Duff algebra.”

| | |
|---|---|
| `(!M1)?I1:I2` | I1 over I2 |
| `(!M2)?I2:I1` | I2 over I1 |
| `(!M1&&!M2)?I1:0` | I1 inside I2 |
| `(!M1&&!M2)?I2:0` | I2 inside I1 |
| `(!M1&&!M2)?0:I1` | I1 outside I2 |
| `(!M1&&!M2)?0:I2` | I2 outside I1 |
| `(!M1&&!M2)?I1:I2` | I1 atop I2 |
| `(!M1&&!M2)?I2:I1` | I2 atop I1 |

The expressions are for non-blurred mattes (cf. Chapter **4** photo 3).

### Arbitrary Composites

See Chapter **3**, expressions 3.15, 3.28, 3.31, 3.32, and 3.33.

```
    (one[x,y] > Z/2) ? one[x,y] : two[x,y]
    (x>X/2)? old : old[X-x,y]
    (x*two[x,y] + (X-x)*one[x,y])/X
    (x<X/3)?two:(x>2*X/3)?one:((x-X/3)*one+(2*X/3-x)*two)*3/X
    (y<Y/3)?two:(y>2*Y/3)?one:((y-Y/3)*one+(2*Y/3-y)*two)*3/Y
```

### Plotting a Grid

| | |
|---|---|
| `(x%7>1)?(y%7>1)?0:Z:Z` | Evenly spaced, thick white lines. |
| `(x%7)?(y%7)?0:x/2:x/2` | Thinner grid, fading from left to right. |

### Routine-1: Oil Transfer

An example library routine that can be linked with the image editor. This particular transformation was used for Chapter **4**, photo 7. We will use the macro definitions for *New* and *Old* in also the other routines that are listed here.

```
#define N       3
#define New     src[CURNEW].pix
#define Old     src[CUROLD].pix

oil()
{       register int x, y;
        register int dx, dy, mfp;
        int histo[256];

        for (y = N; y < DEF_Y-N; y++)
        for (x = N; x < DEF_X-N; x++)
        {       for (dx = 0; dx < 256; dx++)
                        histo[dx] = 0;

                for (dy = y-N; dy <= y+N; dy++)
                for (dx = x-N; dx <= x+N; dx++)
                        histo[Old[dy][dx]]++;

                for (dx = dy = 0; dx < 256; dx++)
                        if (histo[dx] > dy)
                        {       dy = histo[dx];
                                mfp = dx;
                        }
                New[y][x] = mfp;
}       }
```

Note that the values in array histo can be updated faster if you avoid counting the same pixels more than once in a single sweep across the width of the image.

**Routine-2: Picture Shear**

See Chapter **4**, photo 2. The routine uses a standard library function *rand*() to draw random numbers.

```
shear()
{       register int x, y, r;
        int dx, dy, yshift[DEF_X];

        for (x = r = 0; x < DEF_X; x++)
        {       if (rand()%256 < 128)
                        r--;
                else
                        r++;
                yshift[x] = r;
        }

        for (y = 0; y < DEF_Y; y++)
        {       if (rand()%256 < 128)
                        r--;
                else
                        r++;
                for (x = 0; x < DEF_X; x++)
                {       dx = x+r; dy = y+yshift[x];
                        if (dx >= DEF_X || dy >= DEF_Y
                        ||  dx < 0 || dy < 0)
                                continue;
                        New[y][x] = Old[dy][dx];
}       }       }
```

**Routine-3: Slicing**

See Chapter **4**, photo 9.  For the definitions of *New* and *Old* see Routine-1.

```
slicer()
{       register int x, y, r;
        int dx, dy, xshift[DEF_Y], yshift[DEF_X];

        for (x = dx = 0; x < DEF_X; x++)
        {       if (dx == 0)
                {       r = (rand()&63)-32;
                        dx = 8+rand()&31;
                } else
                        dx--;
                yshift[x] = r;
        }
        for (y = dy = 0; y < DEF_Y; y++)
        {       if (dy == 0)
                {       r = (rand()&63)-32;
                        dy = 8+rand()&31;
                } else
                        dy--;
                xshift[y] = r;
        }

        for (y = 0; y < DEF_Y; y++)
        for (x = 0; x < DEF_X; x++)
        {       dx = x+xshift[y]; dy = y+yshift[x];
                if (dx < DEF_X && dy < DEF_Y
                &&  dx >= 0 && dy >= 0)
                        New[y][x] = Old[dy][dx];
}       }
```

**Routine-4: Tiling**

See also Chapter **4**, photo 15.  The routine can be made more interesting by also randomly varying the size of the tiles.

```
#define T 25    /* tile size */

tiling()
{       register int x, y, dx, dy;
        int ox, oy, nx, ny;
        for (y = 0; y < DEF_Y-T; y += T)
        for (x = 0; x < DEF_X-T; x += T)
        {       ox = (rand()&31)-16;    /* displacement */
                oy = (rand()&31)-16;

                for (dy = y; dy < y+T; dy++)
                for (dx = x; dx < x+T; dx++)
                {       nx = dx+ox; ny = dy+oy;
                        if (nx >= DEF_X || ny >= DEF_Y
                        ||  nx < 0 || ny < 0)
                                continue;
                        New[ny][nx] = Old[dy][dx];
}       }       }
```

**Routine-5: Melting**

See Chapter **4**, photo 5. This transformation “melts” the image in place. It does not use the edit buffer *new*, so the two buffers should not be swapped after the transformation completes.

```
melting()
{       register int x, y, val, k;

        for (k = 0; k < DEF_X*DEF_Y; k++)
        {       x = rand()%DEF_X;
                y = rand()%(DEF_Y-1);

                while (y < DEF_Y-1 && Old[y][x] <= Old[y+1][x])
                {       val = Old[y][x];
                        Old[y][x] = Old[y+1][x];
                        Old[y+1][x] = val;
                        y++;
}       }       }
```

**Routine-6: Making a Matte**

Image mattes were used, for instance, to make photos 2 and 19 in Chapter **4**. For a portrait on a fairly light background, a first approximation of a matte can be made with the following routine. It will have to be touched up with a normal paint program. Experiment with different values for *G*.

```
#define G       7.5     /* gamma factor */

extern double pow();    /* the C-library routine */
matte()
{       register x, y;
        unsigned char lookup[256];

        for (x = 0; x < 256; x++)
                lookup[x] = (255. * pow(x/255., G)<3.)?255:0;
        for (y = 0; y < DEF_Y; y++)
        for (x = 0; x < DEF_X; x++)
                New[y][x] = lookup[Old[y][x]];
}
```