The Amazing Bitmap Transmogrifier

by Keith Rollin

Introduction

This document discusses the techniques involved with basic bitmap transformations. This includes translation (moving all the pixels a uniform distance from one place to another - like Copybits), rotation around a specified point, and scaling all pixels away from a specified point. Collectively, these operations are known as transformations.

The Basics

In achieving fast and effective transformations, you must involve several techniques and tricks.

The basic technique is examine a complete bitmap on a pixel-by-pixel basis, and apply certain formulas to the coordinates of each point. Since the process for each pixel is the same, in our separate discussions of translation, scaling and rotation below, we work with only a single pixel. In our actual program, we use a nested loop to perform these operations on all the pixels.

Since the transformation takes place on a pixel-by-pixel basis, you want your calculations to be as brief as possible. One way to do this is attempt to combine your translation, rotation, and scaling calculations into one simple formula. This will reduce the number of multiplies you have to perform, and increase your transformations tremendously. Coming up with these combined formulas is done through the help of matrixes, which we will examine in the next section.

One problem that comes up if we simply take all the pixels in our source bitmap and map them over to a destination bitmap is that we do not get a solid image. For instance, if we were to scale up a bitmap by a factor of 2, 1 out of every 4 pixels will be white. the trick here is to perform the transformation BACKWARDS. In other words, go through every pixel in the destination, and find the closest pixel in the source that maps to it. This way each pixel in the destination gets the consideration that it's due.

Translation

Translating a point is the easiest of all. All you have to do is add offsets to your point's coordinates:

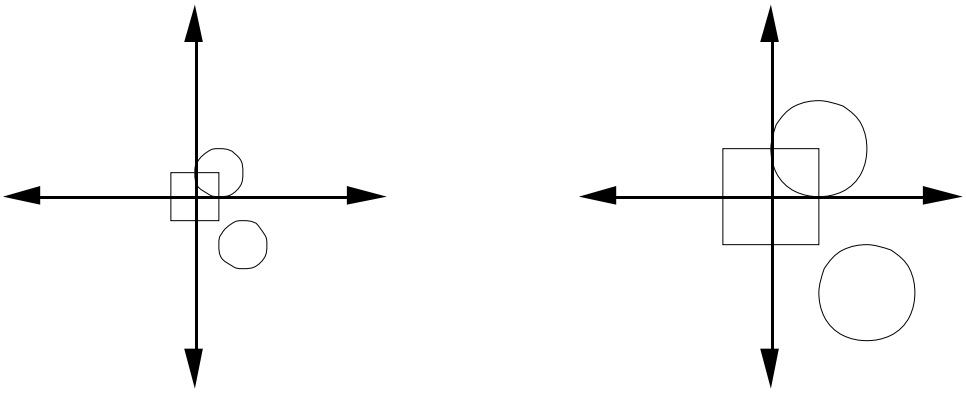
```
x' = x + Dx
y' = y + Dy
```

Scaling

Scaling is almost as easy as translating. All you have to do is multiply your point's coordinates by some scaling factor:

```
x' = (x)(Sx)
y' = (y)(Sy)
```

Please notice that this scales with respect to the origin. In other words, these equations will will cause the following effects:



Scaling by 2 in the X & Y directions

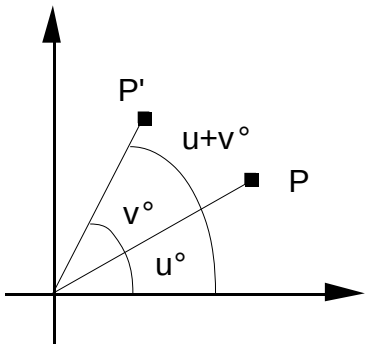
In order to scale relative to another point, one must first translate the entire shape so that that point is at the origin, scale the shape, and then translate this back. As we will see later, this is easily abbreviated into one step.

Reflecting is just a special case of Scaling, where Sx and Sy are either 1 or -1.

Rotation

If you can remember your high school algebra, rotating is also pretty easy. Again, we will look at the case of doing this with relation to the origin; rotating about any point is performed by translating to the origin, rotating, and then translating back, and can all be done in just one step.

In this case, we want to rotate point `P(x,y)` `v` degrees to point `P'(x',y')`.



Rotate u+v degrees

The coordinates of `P` can be expressed:

```
x = r cos(u)
y = r sin(u)
```

Similarly, P' can be expressed:

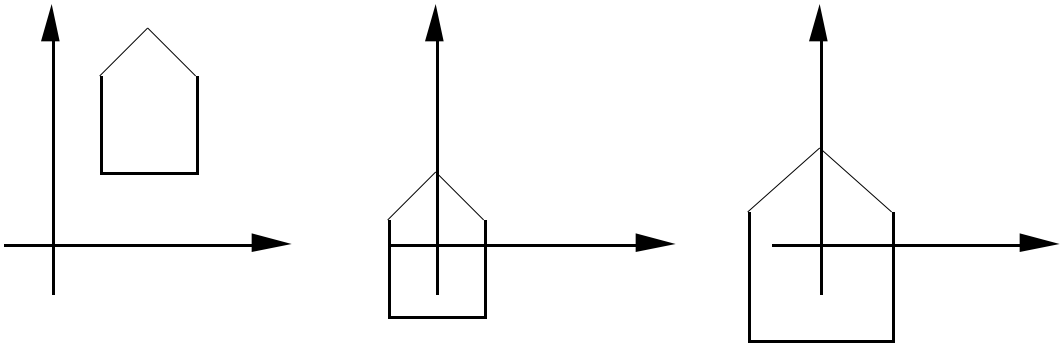
```
x' = r cos(u + v) = r (cos(u)cos(v) - sin(u)sin(v))
y' = r sin(u + v) = r (cos(u)sin(v) + sin(u)cos(v))
```

Distributing `r`, and substituting in `x` and `y`, we get:

```
x' = r cos(u)cos(v) - r sin(u)sin(v)
   = x cos(v) - y sin(v)
y' = r cos(u)sin(q) + r sin(u)cos(v)
   = x sin(v) + y cos(v)
```

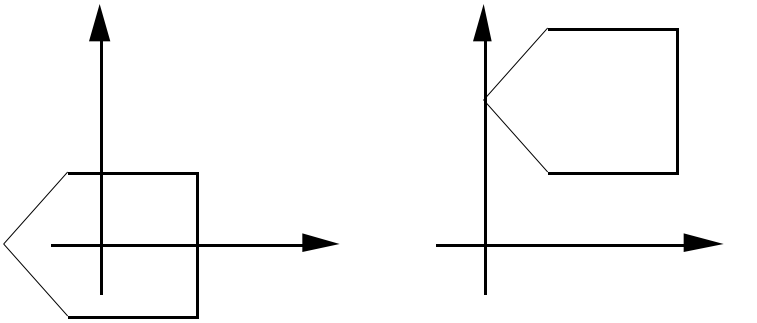
Putting it all together

We now have everything we need to perform general transformations of a bitmap. In general, we translate whatever it is we are transmogrifying to the origin, scale it, rotate it, and then move it back. This is shown below:



Step 1 **Step 2** **Step 3**

Step 1 shows our original position. We have defined a shape of a house using five points. In Step 2, the house has be translated to the origin by adding the same offset to each of those five points. Step 3 shows the house scaled upwards by a small amount. This was done by multiplying the coordinates of each of the points by a constant value.



Step 4 **Step 5**

In Step 3 we show the house rotated about the origin by applying the formula described in the last section for rotating. Finally, the house is translated back to its original position, giving us the result in Step 5.

Let's now follow the life of one of the coordinates through this process. In this case, we'll look at the lower left hand corner of the house, and assume that it's location is (2, 3), and that the origin of the house is at (4, 6).

The first step is to move the center of the house to the origin. This is done by adding the offsets -4 and -6 to the X and Y coordinates respectively. This moves our pet point to (2 - 4, 3 - 6), or (-2, -3).

The next step involves scaling the house. In this case, I expanded the house by 6/4 horizontally, and 8/6 vertically. Applying this to our point, we get:

```
x' = -2(6/4) = -3
y' = -3(8/6) = -4
```

The third step is to rotate the house 90 degrees around the origin. Using our formulas above, we get:

```
x' = x cos(v) - y sin(v)
   = (-3)cos(90) - (-4)sin(90)
   = (-3)(0) - (-4)(1)
   = 4

y' = x sin(v) + y cos(v)
```

```
= (-3)sin(90) + (-4)cos(90)
= (-3)(1) + (-4)(0)
= -3
```

Our point is now at (4, -3). Finally, we translate the whole house back to where it came from by adding the opposite amounts we used to move it to the origin in the first place. Doing this, we get (4 + 4, -3 + 6), or (8, 3). And sure enough, looking at my original drawings in MacDraw (where there's a handy grid), that's exactly where that sucker is!

Get Down, Get Composite!

OK, we've shown that we can be handy around a computer; now lets really show off!

Very briefly, the path that the X coordinate of our sample point above took was:

```
x' = (x-4)(6/4)cos(90) - (y-6)(8/6)sin(90) + 4                [1]
```

Examining this equation tells us a couple of things. For one, rotating by 90 degrees is a snap if you remember that `cos(90)` is always `0`, and `sin(90)` is always `1`. This makes our equation:

```
x' = (6-y)(8/6) + 4                                           [2]
```

Unfortunately, equation [2] is specific to rotating 90 degrees, and equation [1] is specific to translating/scaling/rotating/translating, in that order. Also, there are one too many multiplies and one too many divides in there. In a time critical loop like the one we're going to be in when processing an entire bitmap, the time spent doing these extra operations really adds up. By getting fancy, we can reduce this to 2 multiplies and an add no matter how convoluted our transformation gets.

The first thing we do is put our rules for rotating, scaling, reflecting, and translating into matrix form. For example, for rotating, we can express this as:

```
P' = RP, where R =   [ cos(v)  -sin(v) ]
                     [ sin(v)   cos(v) ]
```

Performing the math, we get:

```
P' = RP = [ cos(v)  -sin(v) ][ x ]
          [ sin(v)   cos(v) ][ y ]

   = [ (x cos(v) - y sin(v))    (x sin(v) + y cos(v)) ]
```

Which agrees with what we showed at the beginning of this document. Unfortunately, the translation transformation cannot be expressed as a 2 x 2 matrix. However, a certain trick allows us to introduce a 3 x 3 matrix function which performs the transformation. By expressing our point P with the coordinates (x, y, 1), we can use the following matrix for translation:

Translating:

```
                       [  1   0 Tx  ]
P' = TP, where   T =   [  0   1 Ty  ]
                       [  0   0  1  ]
```

Similarly, for the other functions, we get:

Rotating:

```
                       [ cos(v)  -sin(v)   0 ]
P' = TP, where   T =   [ sin(v)   cos(v)   0 ]
                       [   0       0      1 ]
```

Scaling:

$$P' = SP, \text{ where } S = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now the fun really begins! With these matrices, we can mix and match in any order that we want in a very general fashion. For instance, using the example of transforming the point (2, 3) above, we can express the translation, scaling, rotating, and re-translating as:

$$P' = (T2)(R)(S)(T1)(P)$$

Expanding out $(T2)(R)(S)(T1)$ to its proper matrices:

$$\begin{bmatrix} 1 & 0 & Dx \\ 0 & 1 & Dy \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} \cos(v) & -\sin(v) & 0 \\ \sin(v) & \cos(v) & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -Cx \\ 0 & 1 & -Cy \\ 0 & 0 & 1 \end{bmatrix}$$

In this notation (Cx, Cy) is the center of of our shape - the one that will be moved to the origin before we scale and rotate; Sx and Sy are the scaling factors in the x and y directions; v is the amount of rotation; (Dx, Dy) is the destination of our shape's chosen center - these can be the same as (Cx, Cy). Multiplying the first two matrices, we get:

$$\begin{bmatrix} \cos(v) & -\sin(v) & Dx \\ \sin(v) & \cos(v) & Dy \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -Cx \\ 0 & 1 & -Cy \\ 0 & 0 & 1 \end{bmatrix}$$

Doing the next multiplication, we get:

$$\begin{bmatrix} (Sx)\cos(v) & -(Sy)\sin(v) & Dx \\ (Sx)\sin(v) & (Sy)\cos(v) & Dy \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -Cx \\ 0 & 1 & -Cy \\ 0 & 0 & 1 \end{bmatrix}$$

This is starting to look a little messy, so lets make a little notational substitution. Let:

$$\begin{aligned} r11 &= (Sx)\cos(v) \\ r12 &= -(Sy)\sin(v) \\ r21 &= (Sx)\sin(v) \\ r22 &= (Sy)\cos(v) \end{aligned}$$

This gives:

$$\begin{bmatrix} r11 & r12 & Dx \\ r21 & r22 & Dy \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & -Cx \\ 0 & 1 & -Cy \\ 0 & 0 & 1 \end{bmatrix}$$

Carrying out the final multiplication, we get:

$$\begin{bmatrix} r11 & r12 & (-Cx)(r11)+(-Cy)(r12)+Dx \\ r21 & r22 & (-Cx)(r21)+(-Cy)(r22)+Dy \\ 0 & 0 & 1 \end{bmatrix}$$

This, then, is the final matrix by which to multiply the coordinates of our points.

$$\begin{bmatrix} r11 & r12 & (-Cx)(r11)+(-Cy)(r12)+Dx \\ r21 & r22 & (-Cx)(r21)+(-Cy)(r22)+Dy \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} (r11)(x) + (r12)(y) + (-Cx)(r11)+(-Cy)(r12)+Dx \\ (r21)(x) + (r22)(y) + (-Cx)(r21)+(-Cy)(r22)+Dy \\ 1 \end{bmatrix}$$

Which just means that our new `P'(x',y')` is:

```
x' = (r11)(x) + (r12)(y) + (-Cx)(r11)+(-Cy)(r12)+Dx
y' = (r21)(x) + (r22)(y) + (-Cx)(r21)+(-Cy)(r22)+Dy
```

The astute observer will note, however, that everything we've just done is useless. Back at the beginning, we said that best results were obtained by running the transform backwards! In other words, we needed something like:

```
P  = (T1')(S')(R')(T2')(P')
```

Expanding out `(T1')(S')(R')(T2')` to its proper matrices:

```
[ 1  0  Cx ][ 1/Sx  0  0 ][ cos(-v) -sin(-v)  0 ][ 1  0 -Dx ]
[ 0  1  Cy ][  0  1/Sy 0 ][ sin(-v)  cos(-v)  0 ][ 0  1 -Dy ]
[ 0  0   1 ][  0    0  1 ][    0        0     1 ][ 0  0   1 ]

[ 1/Sx  0  Cx  ][ cos(-v) -sin(-v)  0 ][ 1  0 -Dx ]
=  [  0  1/Sy Cy  ][ sin(-v)  cos(-v)  0 ][ 0  1 -Dy ]
[  0    0   1  ][    0        0     1 ][ 0  0   1 ]

[ (1/Sx)cos(-v)   -(1/Sx)sin(-v)  Cx  ][ 1  0 -Dx ]
=  [ (1/Sy)sin(-v)   -(1/Sy)cos(-v)  Cy  ][ 0  1 -Dy ]
[       0                0         1  ][ 0  0   1 ]



[ r11  r12  Cx  ][ 1  0  -Dx ]
=  [ r21  r22  Cy  ][ 0  1  -Dy ]
[  0    0    1  ][ 0  0    1 ]

[ r11  r12  (-Dx)(r11)+(-Dy)(r12)+Cx ]
=  [ r21  r22  (-Dx)(r21)+(-Dy)(r22)+Cy ]
[  0    0               1            ]

[ r11  r12  tx ]
=  [ r21  r22  ty ]
[  0    0    1 ]
```

Where:

```
r11 =  (1/Sx)cos(-v)
r12 = -(1/Sx)sin(-v)
r21 =  (1/Sy)sin(-v)
r22 =  (1/Sy)cos(-v)
tx  =  (-Dx)(r11)+(-Dy)(r12)+Cx
ty  =  (-Dx)(r21)+(-Dy)(r22)+Cy
```

This tells us that, given a point P'(x', y') in the destination, the source pixel that corresponds to it is:

```
x = (r11)(x') + (r12)(y') + tx
y = (r21)(x') + (r22)(y') + ty
```

Getting into the Routine

Now that we've finished our homework, we can get down to implementing the equations as part of a routine. The following C routine does this. It accepts two pointers to bitmaps, a point that specifies the center for rotation and scaling, a point that specifies the destination of the center, the amount (in degrees) of rotation, and the amount of scaling to be performed in the X and Y directions:

```
#define     PI          (3.14159265358979323846)

pascal void DoTransform(BitMap *sourceBM, BitMap *destBM, Point center,
         Point destination, short rotation, extended sx, extended sy)
{
   short    i;
   short    j;
   short    newX, newY;
   Boolean     isSet;
   long     *longPtr;

   short    width = destBM->bounds.right;
   short    height = destBM->bounds.bottom;

   extended r11 =  (1/sx) * cos(rotation * (PI/180));
   extended r12 = -(1/sx) * sin(rotation * (PI/180));
   extended r21 =  (1/sy) * sin(rotation * (PI/180));
   extended r22 =  (1/sy) * cos(rotation * (PI/180));

   extended tx = -destination.h*r11 - destination.v*r12 + center.h;
   extended ty = -destination.h*r21 - destination.v*r22 + center.v;

   ClearBitMap(destBM);

   for ( j = 0; j < height; j++ ) {
      for ( i = 0; i < width; i++ ) {

         newX = i*r11 + j*r21 + tx + .5;
         newY = i*r12 + j*r22 + ty + .5;

         isSet = myGetPixel(sourceBM, newX, newY);
         if (isSet) mySetPixel(destBM, i, j);
      }
   }
}
```

The two core routines - myGetPixel and mySetPixel - are left for you to implement. To be 100% compatible with future versions of QuickDraw, these routines should call Get(C)Pixel and PaintRect. However, doing so is extremely slow - on the order of 7 minutes to transform a full picture. By reading and setting the bitmap's bits directly, you can get a 12-fold increase in performance; the same transformation will take about 35 seconds. By performing some optimizations on the routine above, you can reduce this to about 25 seconds. Such optimizations are shown in the accompanying sample program.

NOTE: these times are on a Mac II when using the MC68881 directly and compiling for the MC68020. They were also obtained by performing the transformations operations - which are optimized for pictures that are mostly white - on a picture that is 75% black.

Where to go from here

There are many things you can do when playing with these routines for your own purposes. Some of these ideas may appear in future versions of the sample program.

- Include cumulative transformations. Right now, the sample program always performs the transformations on the original bitmap. You can modify it so that it can optionally perform additionally transformations on an already transformed bitmap.

- Correctly calculate the size of the destination bitmap. Right now, the sample program forces the destination bitmap to be the size of a MacPaint picture, regardless of how much is needed to hold the final image.

- Use Fract and Fixed data types and the transcendental functions described in Inside Mac IV instead of extendeds. This may give acceptable performance on machines that don't sport a 68881.

- Include printing code. For MacApp programs, this (ahem) is trivial.

- Add an assembly version that accesses the 68881/68882 more efficiently