TECHNOTE:
Decomposing a QuickDraw GX Mapping

by Lawrence D'Oliveiro
ldo@waikato.ac.nz
The University of Waikato, New Zealand

A QuickDraw GX mapping, being a 3-by-3 matrix, can specify any 2D linear transformation. It is easy enough to build up such a transformation from a sequence of *primitive transformations*, namely translations, scalings, rotations, skews and perspective distortions — GX provides calls to construct nearly all of these components (the exception is perspective, for which there is library code in the GX SDK). Sometimes there is a need to go the other way: given an arbitrary linear transformation, can you break it into a sequence of pure translations, scalings, skews and perspective distortions? This Technote will show you how.

Intended Audience

This Technote is aimed at those who already have some basic understanding of QuickDraw GX graphics, including how to make use of GX mappings. The exposition will take more of an intuitive, hand-waving approach, with little pretense at rigorous derivation.

See Also

Inside Macintosh: QuickDraw GX Environment and Utilities, “QuickDraw GX Mathematics” chapter.

Mac OS SDK CD, ":QuickDraw™ GX:Programming Stuff:GX Libraries:" folder, MappingLibrary.c, the `PolyToPolyMap` routine.

Approaching the Solution

In general, there is no unique decomposition of a mapping into primitive components. For example, if you perform a scaling followed by a translation, you can switch the order if you adjust the translation x and y distances by appropriate amounts. Similarly, a rotation can be represented by a combination of two skews in appropriate directions.

For this reason, we have to decide beforehand on an arbitrary, but convenient, mix of component primitives, and on an ordering for them. Now, a GX mapping consists of nine numbers:

```
typdef struct {
 Fixed map[3][3];
} gxMapping;
```

One of these (`map[2][2]`) is a weighting for the other numbers, so it can be arbitrarily fixed at 1. That leaves eight numbers to be determined.

- A rotation about the origin requires the specification of one number—an angle.
- A translation is specified by two numbers, being the x and y displacements.
- A scaling about the origin is specified by two numbers, being the x and y scale factors.
- A skew about the origin can have different x and y skew factors. We can simplify things by requiring that both the x and y skew factors be equal; any asymmetry can be dealt with by bringing in a rotation as well.
- A perspective distortion requires the specification of two numbers, which can be loosely described as x and y distortion factors.

It so happens that one of each of the above adds up to exactly eight numbers. These do not directly map to the eight elements of the mapping to be determined, but the latter can be uniquely determined from the former, so this is not a problem.

So I will concentrate exclusively on solutions consisting (in order) of

- a scaling **S** about the origin
- a symmetrical skew **K** (no net rotation)
- a rotation **R** about the origin
- a translation **T**
- and a perspective distortion **P**.

If **M** is the original mapping, then the above sequence of transformations can be put in matrix terms as follows:

$$\mathbf{M} = \mathbf{S}\cdot\mathbf{K}\cdot\mathbf{R}\cdot\mathbf{T}\cdot\mathbf{P}$$

The matrix product, represented by the "·" symbol, corresponds to the GX `MapMapping` call. **Note:** I'm assuming that **M** is an invertible matrix: it mustn't map the plane onto a single line or a point.

Thus, the prototype of the function we want to implement can be written as:

```
void DecomposeMapping
  (
 const gxMapping *M, /* input mapping */
 gxMapping *S, /* output scaling component */
 gxMapping *K, /* output skew component */
 gxMapping *R, /* output rotation component */
 gxMapping *T, /* output translation component */
 gxMapping *P, /* output perspective component */
 Fixed *SkewAngle, /* output skew angle in degrees */
 Fixed *RotationAngle /* output rotation angle in degrees */
  )
  /* decomposes a mapping into pure scaling, skew, rotation,
 translation and perspective components. */
```

Suppose the mapping **M** is represented in conventional matrix fashion as

$$\begin{bmatrix} m_{0,0} & m_{0,1} & m_{0,2} \\ m_{1,0} & m_{1,1} & m_{1,2} \\ m_{2,0} & m_{2,1} & m_{2,2} \end{bmatrix}$$

, where the subscripts are the corresponding indexes

into the `map` array in the `gxMapping` record. You will remember that this matrix represents a 2D transformation in *homogeneous* coordinates—the entire matrix can be scaled by an arbitrary factor, so long as the same factor is applied to all elements, and it will still represent the same transformation. If the bottom-rightmost element $m_{2,2}$—the *weighting*—is equal to 1, then the matrix is said to be *normalized*, which simply means we don't have to worry about the weighting.

The component matrices can be represented as follows:

S has the form $\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$, where s_x and s_y are the horizontal and vertical scale factors.

K will take the form $\begin{bmatrix} \cos\psi & \sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$, where ψ is the skew angle; this form of skew has no net rotation or scaling effect.

R is $\begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$, where θ is the rotation angle.

T is $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$, where t_x and t_y are the horizontal and vertical offset amounts.

And **P** can be represented as $\begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & 1 \end{bmatrix}$.

You'll notice that all these components are normalized, whereas the original matrix **M** is not required to be. The problem is perspective: you can't seem to deal with it without introducing unnormalized matrices at some point. Requiring that **M** be normalized to start with doesn't help, which is why I haven't bothered. Instead, the code below will include an explicit normalization step after separating out the perspective component.

The Solution, Part 1: Perspective

As a first step, let us determine **P**. We can represent the product of the other components, **S**·**K**·**R**·**T**, as a matrix with no perspective components, thus:

$\begin{bmatrix} m_{0,0} & m_{0,1} & 0 \\ m_{1,0} & m_{1,1} & 0 \\ m_{2,0} & m_{2,1} & w \end{bmatrix}$. Note that this is not normalized, because of the way perspective transforms work (we'll be able to work entirely with normalized matrices once we get perspective out of the way). Also note the weighting element is not the same as in **M**.

Thus, to determine **P**, we have to solve the equation

$$\mathbf{M} = \begin{bmatrix} m_{0,0} & m_{0,1} & 0 \\ m_{1,0} & m_{1,1} & 0 \\ m_{2,0} & m_{2,1} & w \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & 1 \end{bmatrix}$$

which has three unknowns, u, v and w. It so happens we can cleverly

rearrange this (exercise: how come?) as follows:

$$\mathbf{M} = \begin{bmatrix} m_{0,0} & m_{0,1} & 0 \\ m_{1,0} & m_{1,1} & 0 \\ m_{2,0} & m_{2,1} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & w \end{bmatrix}$$

thereby putting all three unknowns into the same matrix. We can then solve for the unknowns by rearranging:

$$\begin{bmatrix} 1 & 0 & u \\ 0 & 1 & v \\ 0 & 0 & w \end{bmatrix} = \begin{bmatrix} m_{0,0} & m_{0,1} & 0 \\ m_{1,0} & m_{1,1} & 0 \\ m_{2,0} & m_{2,1} & 1 \end{bmatrix}^{-} \cdot \mathbf{M}$$

which can be implemented in a straightforward fashion using the standard GX functions `InvertMapping` and `MapMapping`. Here's the code for this part of the `DecomposeMapping` function:

```
int i, j;
gxMapping temp1, temp2;
Fract w;

temp1 = *M;
temp1.map[0][2] = 0;
temp1.map[1][2] = 0;
temp1.map[2][2] = fract1;
*P = temp1;
InvertMapping(P, P);
MapMapping(P, M);
w = P->map[2][2];
P->map[2][2] = fract1; /* P is done */
for (i = 0; i < 3; ++i)
    for (j = 0; j < 3; ++j)
      /* I'd use NormalizeMapping, but it doesn't work */
         temp1.map[i][j] = FractDivide(temp1.map[i][j], w);
temp1.map[2][2] = fract1;
  /* temp1 is product of remaining components */
```

The Solution, Part 2: Translation

Having just done perspective, the most difficult component, the next one, translation, is the easiest:

```
ResetMapping(T);
T->map[2][0] = temp1.map[2][0]; /* X translation component */
T->map[2][1] = temp1.map[2][1]; /* Y translation component */
temp1.map[2][0] = 0;
temp1.map[2][1] = 0;
```

The Solution, Part 3: Scaling

Assume for a moment that there is no skew component, and consider the product **S**·**R**:

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x\cos\theta & s_x\sin\theta & 0 \\ -s_y\sin\theta & s_y\cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If you sum the squares of the elements in the top row, you get s_x^2. And if you sum the squares of the ones in the middle row, you get s_y^2. The same statements can be made if there is a pure skew and no rotation to go with the scaling; hence, they also hold true for a combination of skew and rotation.

Thus, here's how to extract the scaling:

```
ResetMapping(S);
S->map[0][0] = FixedSquareRoot
  (
        FixedMultiply(temp1.map[0][0], temp1.map[0][0])
    +
        FixedMultiply(temp1.map[0][1], temp1.map[0][1])
  );
S->map[1][1] = FixedSquareRoot
  (
        FixedMultiply(temp1.map[1][0], temp1.map[1][0])
    +
        FixedMultiply(temp1.map[1][1], temp1.map[1][1])
  );
ResetMapping(&temp2);
temp2.map[0][0] = FixedDivide(fixed1, S->map[0][0]);
temp2.map[1][1] = FixedDivide(fixed1, S->map[1][1]);
```

```
 /* temp2 is inverse of S */
MapMapping(&temp2, &temp1);
  /* temp2 now contains only skew and rotation */
```

The Solution, Part 4: Skewing and Rotation

Now all that's left is the skew and rotation. The product of these two is

$$\mathbf{K \cdot R} = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\psi\cos\theta - \sin\psi\sin\theta & \cos\psi\sin\theta + \sin\psi\cos\theta & 0 \\ \sin\psi\cos\theta - \cos\psi\sin\theta & \sin\psi\sin\theta + \cos\psi\cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos(\psi + \theta) & \sin(\psi + \theta) & 0 \\ \sin(\psi - \theta) & \cos(\psi - \theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Thus, we can extract the remaining components as follows:

```
double_t AngleSum, AngleDiff, AngleRadians;
#define RadiansToDegrees (180.0 / 3.141592653589)

AngleSum = atan2
  (
     (double_t)temp2.map[0][1],
     (double_t)temp2.map[0][0]
  );
AngleDiff = atan2
  (
     (double_t)temp2.map[1][0],
     (double_t)temp2.map[1][1]
  );
AngleRadians = (AngleSum + AngleDiff) / 2.0; /* skew angle */
ResetMapping(K);
```

```
K->map[0][0] = X2Fix(cos(AngleRadians));
K->map[0][1] = X2Fix(sin(AngleRadians));
K->map[1][0] = K->map[0][1];
K->map[1][1] = K->map[0][0];
*SkewAngle = X2Fix(AngleRadians * RadiansToDegrees);
AngleRadians = (AngleSum - AngleDiff) / 2.0; /* rotation angle */
ResetMapping(R);
R->map[0][0] = X2Fix(cos(AngleRadians));
R->map[0][1] = X2Fix(sin(AngleRadians));
R->map[1][0] = -R->map[0][1];
R->map[1][1] = R->map[0][0];
*RotationAngle = X2Fix(AngleRadians * RadiansToDegrees);
```

and that's it!

Further Notes

When you try to reconstruct the original mapping from the components, you may encounter a couple of surprises.

First of all, the numbers making up the result may be completely different from the ones in the original mapping. This can happen if the mapping includes a perspective component, or was not normalized. If you look closer, you should find that the numbers bear a constant ratio to the ones in the original mapping, which means that in fact they represent the same mapping.

After accounting for this ratio, you may find that the numbers still differ slightly from the ones in the original mapping. This is because all the arithmetic was being carried out to a finite amount of precision, as is usually the case with computer arithmetic, so the results were not being represented exactly.

For example, consider the following output from an MPW tool I wrote to test the above code. The nine elements of each mapping are written out on a line in row order, with the input mapping on the first line:

```
DecomposeMapping 1 2 1.5 3 4 1.5 6 7 1
Original: 1 2 1.5 3 4 1.5 6 7 1
Scale: -17.88855 0 0 0 -40 0 0 0 1
Skew: 0.64075 0.76775 0 0.76775 0.64075 0 0 0 1
Rotation: 0.97325 0.22975 0 -0.22975 0.97325 0 0 0 1
Translation: 1 0 0 0 1 0 -48 -56 1
Perspective: 0.25 0 -0.375 0 0.25 0.375 0 0 1
Skew angle: 50.15242
Rotation angle: 13.28253
Reconstructed: -0.25 -0.5 -0.3749935627 -0.75 -1 -0.375002861 -1.5 -1.75 -0.25
```

As you can see, the elements of the reconstructed mapping need to be

multiplied by a factor of -4 to make them equal to the original mapping. Even then, the third and sixth numbers (the perspective elements) are not exactly equal to the original values, there being a small discrepancy in the fifth decimal place. This is an occupational hazard with most computer arithmetic: you just have to watch out for it, and not expect exact equality. This could be mitigated by getting rid of most of the fixed-point arithmetic, instead converting all the numbers to reals, using standard real arithmetic, and then converting back; this will reduce the rounding errors, but it cannot eliminate them.

Further quirk: A matrix which does a mirror-image reflection will be interpreted by the above code as having a skew angle close to 90°. It might be more sensible to negate one of the elements of the scaling component instead. For example, when the code computes `S->map[0][0]`, this could be negated if `temp1.map[0][0]` is negative, and similarly `S->map[1][1]` could be negated if `temp1.map[1][1]` is negative.

Acknowledgments

Thanks to Crispin Gardiner for first showing me how skews can add up to rotations. A salute goes to Cary Clark for letting loose the snowball that turned into QuickDraw GX, sucking so many other projects into its path. And fond remembrances go to Diana Rigg and Patrick Macnee, for teaching me the meaning of gxStyle.