

If you're comfortable with linear algebra (matrix mathematics), you can represent any arbitrary combination of 2-D scalings, skews, rotations and translations using standard matrix operations.

The idea is that you can represent a point as a column vector

$$\hat{p} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix},$$

and any 2-D scaling, skew, rotation or translation operation \mathcal{S} by a 3x3 matrix with this form:

$$\mathcal{S} = \begin{pmatrix} a & c & tx \\ b & d & ty \\ 0 & 0 & 1 \end{pmatrix}.$$

Then the operation $\mathcal{S}(\hat{p})$ is just the standard matrix-times-vector $\mathcal{S} \times \hat{p}$:

$$\begin{pmatrix} a & c & tx \\ b & d & ty \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax + cy + tx \\ bx + dy + ty \\ 1 \end{pmatrix}.$$

We throw out the '1' at the bottom (which will always be 1) to get the transformed point.

Compositions (i.e. apply one operation, then another) are multiplications of the corresponding matrices. If you want to transform by \mathcal{S} and then \mathcal{T} :

```
public var composition:Matrix = S.clone;
composition.concat(T);
foo.transform.matrix = composition;
```

you can multiply the two matrices $\mathbb{T} \times \mathbb{S}$:

$$\begin{pmatrix} a_t & c_t & tx_t \\ b_t & d_t & ty_t \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} a_s & c_s & tx_s \\ b_s & d_s & ty_s \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_t a_s + c_t b_s & a_t c_s + c_t d_s & a_t tx_s + c_t ty_s + tx_t \\ b_t a_s + d_t b_s & b_t c_s + d_t d_s & b_t tx_s + d_t ty_s + ty_t \\ 0 & 0 & 1 \end{pmatrix}$$

[Yikes! well, see my demo for an Actionscript implementation. It's not so bad.] Notice the order: since we want to end up applying "S and then T" we use $\mathbb{T} \times \mathbb{S}$ -- applied to a point it gives

$$(\mathcal{T} \circ \mathcal{S})(\hat{p}) = (\mathbb{T} \times \mathbb{S}) \times \hat{p} = T \times (\mathcal{S} \times \hat{p}) = \mathcal{T}(\mathcal{S}(\hat{p}))$$

as desired.

An inverse transformation is the standard matrix inverse: the actionscript

```
public var sInv:Matrix = S.clone;
sInv.invert();
foo.transform.matrix = sInv;
```

is pronounced in math as

$$\begin{pmatrix} a & c & tx \\ b & d & ty \\ 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} d/\det & -c/\det & (c ty - d tx)/\det \\ -b/\det & a/\det & (b tx - a ty)/\det \\ 0 & 0 & 1 \end{pmatrix}$$

where $\det = (ad - bc)$. (Note in passing that the last column is the deltaPointTransform of the inverse matrix on the (tx, ty) column.)

An important point: if you look at what Actionscript actually does with pointTransform(s), concat(s) and invert(s), the Matrix object is consistent with the transformation matrix

$$\mathcal{S} = \begin{pmatrix} a & c & tx \\ b & d & ty \\ 0 & 0 & 1 \end{pmatrix}$$

and not with b and c switched as it appears in the documentation. Also, if you read the excellent tutorial (with much better

pictures) at

<http://www.senocular.com/flash/tutorials/transformmatrix/>,

please note that you have to represent **points as row vectors** and **operations as multiplying on the right**: $\mathcal{S}(\hat{p})$ is $\hat{p} \times \mathbb{S}$ and $\mathcal{T}(\mathcal{S}(\hat{p}))$ is $\hat{p} \times \mathbb{S} \times \mathbb{T}$. The way I've done it above is more consistent with how a transformation matrix is usually represented, but the mathematics at senocular is consistent. (If you remember how the transpose of a matrix works, he's just working with transposed matrices.)