# Understanding the Transformation Matrix in Flash 8

## www.senocular.com

Flash Transform Matrix.pdf

### Introduction

Flash 8 has brought to the Flash developer a new, exciting level of control in Flash. Not only can Flash developers now directly manipulate bitmaps within Flash on the fly, but now, they also have complete control over a movie clip's transformations through that movie clip's transform matrix. In prior versions of Flash, that was not possible. Then, certain transformations such as skewing had to be achieved, if by ActionScript, through complicated math dealing with nested movie clips. That is now a thing of the past

Understanding the transform matrix will be the topic of this tutorial. They are used both in manipulating movie clips, and bitmaps when dealing with the new BitmapData object. This makes understanding them quite helpful.

### Matrices and the Transform Matrix

Before getting into how transformation matrices (matrices is plural of matrix) work, it is important to understand what a matrix is. A *matrix* is a rectangular array (or table) of numbers consisting of any number of rows and columns. A matrix consisting of *m* rows and *n* columns is known as an *m* x *n* matrix. This represents the matrix's *dimensions*. You'll commonly seen matrices with numbers in rows and columns surrounded by two large bracket symbols.

$$\begin{bmatrix} 5 & 2 & 4 & 4 & 8 \\ 2 & 1 & 5 & 6 & 5 \end{bmatrix} \quad \begin{bmatrix} 4 & 6 \\ 1 & 2 \\ 8 & 2 \end{bmatrix} \quad \begin{bmatrix} 2 & 5 & 6 \\ 0 & 1 & 5 \\ 3 & 9 & 4 \end{bmatrix}$$

2 x 5 matrix  3 x 2 matrix  3 x 3 matrix

Matrices are used for many different purposes, often to store data or solve problems using certain matrix calculations. Flash uses matrices to define affine transformations.

*Affine transformations* are transformations that preserve collinearity and relative distancing in a transformed coordinate space. This means points on a line will remain in a line after an affine transformation is applied to the coordinate space in which that line exists. It also means parallel lines remain parallel and that relative spacing or distancing, though it may scale, will always maintain at a consistent ratio. Affine transformations allow for repositioning, scaling, skewing and rotation. Things they cannot do include tapering or distorting with perspective. If you're ever worked with transforming symbols in Flash, you probably recognize these qualities.

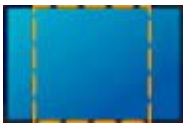
Original | **Yes** | No

The transformation matrices in Flash that define these affine transformations use a 3 x 3 matrix consisting of 3 rows and 3 columns. The positions of values within this matrix are labeled *a*, *b*, *c*, *d*, *tx*, *ty*, *u*, *v*, and *w*. In matrices, these are known as *elements*.

| | | |
|---|---|---|
| a | b | u |
| c | d | v |
| tx | ty | w |

Elements a, b, c, d, tx, and ty are those most important to the transformation matrix. The additional positions, u, v, and w are, at this point, not important but still require to be present in the matrix. Each a, b, c, d, tx, and ty correspond to a specific type of transformation within Flash.

**a** - x scale



**b** - y skew

**c** - x skew

**d** - y scale

**tx** - x translation (position)

**ty** - y translation (position)

The u, v, and w positions are static values that remain at 0, 0, and 1 respectively.

| | | |
|---|---|---|
| x scale | y skew | 0 |
| x skew | y scale | 0 |
| x position | y position | 1 |

**Note:**

In the Matrix class in Flash, u,v, and w are not accessible as properties as a, b, c, d, tx, and ty are.

The effective elements, or *properties* as they will be referred to from this point forward, a, b, c, d, tx, and ty are used in formulas for `x` and `y` when determining the position of a new `x` and `y` value resulting from the transformation applied by a matrix. These formulas are

```
x' = a*x + c*y + tx
y' = b*x + d*y + ty
```

where `x` and `y` represents the original `x` and `y` locations and `x'` and `y'` (read as *x-prime* and *y-prime*) represents the transformed locations. For any point anywhere in an image or movie clip affected by a transformation matrix, these calculations are made to determine the resulting end location for that point.

**Note:**

You may also see a transformation matrix represented as

$$\begin{bmatrix} a & c & tx \\ b & d & ty \\ u & v & w \end{bmatrix}$$

or something similar (notice the u, v, and w properties at the bottom as opposed to the right). This is basically the same matrix, just with a different orientation of elements. Transformations do not change with this difference in orientation, though a few operations are handled differently when using it. The end result still remains the same.

Flash help, at the time of this writing, incorrectly represents a transformation matrix in this manner with the c and b values switched.

## Identity Matrix

When a matrix causes no transformations, you have what is known as a unit or identity matrix.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Any square matrix (a matrix is square if its number of rows equals its number of columns) is an *identity matrix* if all values are 0 except those diagonal from the top left to the bottom right which instead have the value of 1 - the locations where the row position equals the column position. Consider what happens when an identity matrix is used with the formulas for finding a transformed `x` and `y`.

```
x' = a*x + c*y + tx
y' = b*x + d*y + ty

x' = 1*x + 0*y + 0
y' = 0*x + 1*y + 0
```

which reduces to

```
x' = x
y' = y
```

Notice that there is no change resulting from the application of the matrix; `x'` equals `x` and `y'` equals `y`. You can think of an identity matrix for matrices as being the number 1 for *scalars*, or regular numbers. When you multiply 1 by any number, you get the number. The same applies with the identity matrix for transformations. All new matrices in Flash, unless otherwise specified, start out as identity matrices - matrices with no transformations. This includes matrices affecting movie clips. If a movie clip has not been altered or transformed in any way, its transformation matrix is the identity matrix.

## Applying Transformations

Consider what happens if the a property of an identity matrix changes. For example, what would happen if the a property in a Flash transformation matrix, starting out as an identity matrix, is changed from 1 to 2. Remember, the a property references scaling along x.

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

```
x' = a*x + c*y + tx
y' = b*x + d*y + ty

x' = 2*x + 0*y + 0
y' = 0*x + 1*y + 0

x' = 2x
y' = y
```

You now have a situation where all new values of x equal 2 times the old values; `x' = 2x`. A movie clip with the above transformation matrix would have a width double its original.

What if, instead, c was altered and given a value of 1? Remember, c controls skewing along x.

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
x' = a*x + c*y + tx
y' = b*x + d*y + ty

x' = 1*x + 1*y + 0
```

```
y' = 0*x + 1*y + 0

x' = x + y
y' = y
```

Now, you have an `x'` value that relates not just to `x` but to the change in `y` as well. This means that as `y` increases, so do the resulting values of `x`. This causes a skew in `x`, the aspect of the transformation controlled by the b property.

If you wanted to shift this movie clip in any direction, you could easily change tx or ty in the matrix to do so

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 5 & 10 & 1 \end{bmatrix}$$

```
x' = a*x + c*y + tx
y' = b*x + d*y + ty

x' = 1*x + 1*y + 5
y' = 0*x + 1*y + 10

x' = x + y + 5
y' = y + 10
```

The resulting transformation is still a skew, but now the positions of `x` and `y` are just shifted by 5 and 10 respectively.

You may have noticed that there is no property in a matrix for rotation. Rotation is actually a result of a combination of scaling and skewing. Together, scaling and skewing can distort coordinates to provide what you understand as being rotation.

In simplest terms, any rotation transformation without any other types of transformations applied can be represented with the following matrix

$$\begin{bmatrix} \text{cosine(angle)} & \text{sine(angle)} & 0 \\ \text{-sine(angle)} & \text{cosine(angle)} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where the angle variable represents the amount of rotation applied in the transformation.

In terms of transformation matrices, the application of rotation is often the hardest concept to grasp. Not only is there not a property for rotation as there are for aspects like scaling and skewing, but it relies on 4 other properties (scaling and skewing) to work. That within itself can be hard enough to understand but it only gets worse when you want to consider changing those other properties on top of rotation or even based on that rotation. If you're familiar enough with trigonometry and rotating vectors in 2D space, though, you may see a familiar use of sine and cosine. The same concepts apply.

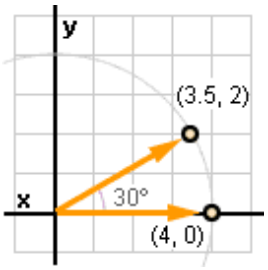
Consider a simple example of a transformation matrix rotating 30 degrees.

$$\begin{bmatrix} \text{cosine(30)} & \text{sine(30)} & 0 \end{bmatrix}$$

| | | |
|---|---|---|
| -sine(30) | cosine(30) | 0 |
| 0 | 0 | 1 |

```
x' = cosine(30)*x - sine(30)*y + 0
y' = sine(30)*x + cosine(30)*y + 0

x' = .87*x - .5*y
y' = .5*x + .87*y
```

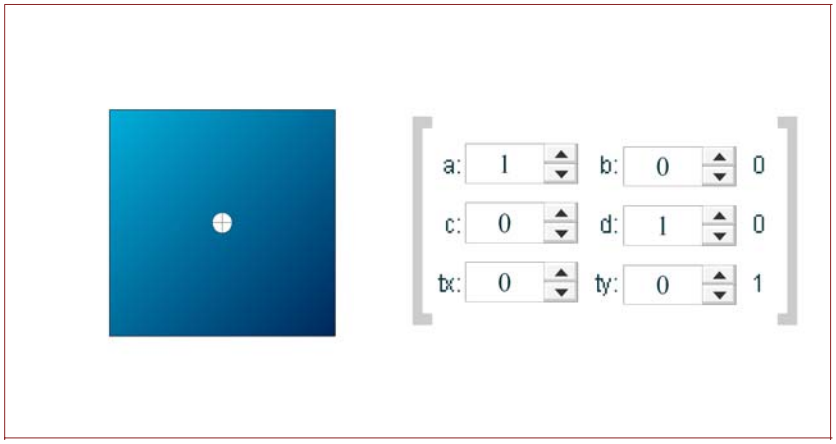
Given the point (4, 0), a rotation of 30 degrees should provide values at about (3.5, 2) for `x` and `y`.




Plugging `x=4`, `y=0` into the formula gives

```
x' = .87*4 - .5*0
y' = .5*4 + .87*0

x' = 3.46
y' = 2
```

Play around with a movie clip's transform matrix and watch how the transformation is applied




## The ActionScript Matrix Class

A matrix in ActionScript is stored in an instance of the Flash Matrix Class. This class is located in the flash.geom package. If you create your own Matrix instance, you would need to use it's full path

```
var my_matrix = new flash.geom.Matrix();
```

or first import the class so that you may reference it using strictly "new Matrix"

```
import flash.geom.Matrix;

var my_matrix = new Matrix();
```

The Matrix constructor allows you to specify the a, b, c, d, tx, and ty properties when the instance is created.

```
import flash.geom.Matrix;

var my_matrix = new Matrix( a, b, c, d, tx, ty );
```

If no arguments are specified, you get an identity matrix.

The properties of the Matrix class are basically just a collection of your main a, b, c, d, tx, and ty properties - those you need to be concerned about when dealing with transformations. Additional methods are also provided to make working with these matrices easier. Some of the more common methods are

```
translate(tx:Number, ty:Number) : Void
scale(sx:Number, sy:Number) : Void
rotate(angle:Number) : Void
identity() : Void
```

Other methods and their descriptions, as well as more information about the Matrix class can be found in Flash help. One method you may not find available is a skew() method. Since skewing is a little atypical of normal transformations, no method is provided. It can be implemented manually through adjusting the b and c properties of a matrix instead.

Each of the methods mentioned above operate on and alter the matrix from which they are called; they do not return a new transformed matrix. Here is a basic rundown.

**translate**

```
translate(tx:Number, ty:Number) : Void
```

The translate method simply moves a transformation by `tx` and `ty`. This process is additive so the tx and ty passed to translate is added on to any existing value of tx and ty in the matrix.

```
my_matrix.translate( 5, 10 );
```

is the same as

```
my_matrix.tx += 5;
my_matrix.ty += 10;
```

**scale**

```
scale(sx:Number, sy:Number) : Void
```

The scale method scales a transform matrix by an `sx` amount in x and an `sy` amount in y. These values are multiplied so a value of 1 represents no scale. One thing to understand about scale, and this applies to rotate as well, is that the values used also affect the matrix's tx and ty properties affecting the transform's position.

```
my_matrix.scale( 1.5, 2 );
```

is the same as

```
my_matrix.a *= 1.5;
my_matrix.d *= 2;
my_matrix.tx *= 1.5;
my_matrix.ty *= 2;
```

**rotate**

`rotate(angle:Number) : Void`

The rotate method rotates a transform matrix by `angle` amount where `angle` is measured in radians. As with scale, rotation is not only applied to the a, b, c, and d properties that affect rotation, but tx and ty as well. The math behind rotation is also a little more complex, as you might have guessed.

```
my_matrix.rotate( Math.PI/4 );
```

is the same as

```
 var sin = Math.sin( Math.PI/4 );
var cos = Math.cos( Math.PI/4 );
var a = my_matrix.a;
var b = my_matrix.b;
var c = my_matrix.c;
var d = my_matrix.d;
var tx = my_matrix.tx;
var ty = my_matrix.ty;
my_matrix.a = a*cos - b*sin;
my_matrix.b = a*sin + b*cos;
my_matrix.c = c*cos - d*sin;
my_matrix.d = c*sin + d*cos;
my_matrix.tx = tx*cos - ty*sin;
my_matrix.ty = tx*sin + ty*cos;
```

**identity**

`identity() : Void`

The identity method converts any matrix into an identity matrix no matter what the prior transformation. In essence, this removes all transformations from the matrix from which it's called.

```
my_matrix.identity();
```

is the same as

```
my_matrix.a = 1;
my_matrix.b = 0;
my_matrix.c = 0;
my_matrix.d = 1;
my_matrix.tx = 0;
my_matrix.ty = 0;
```

## Movie Clip Matrices

When working with movie clips, you have access to the transform matrix affecting the movie clip through the matrix property of that movie clip's transform object.

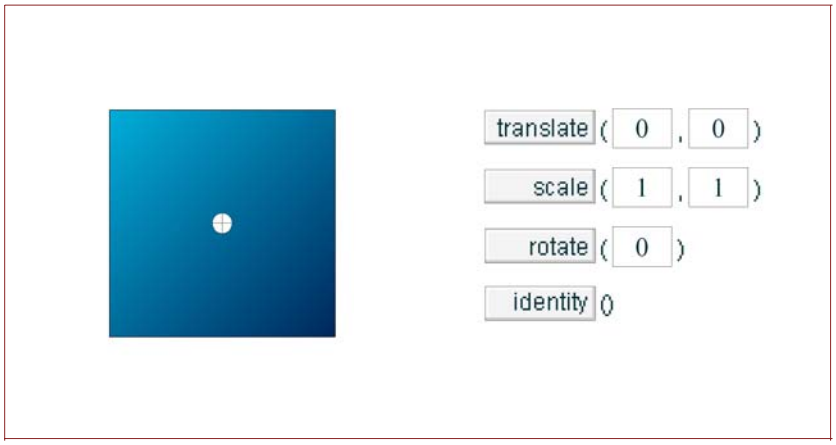
```
var my_matrix = my_mc.transform.matrix;
```

This matrix is actually provided as a copy of the "real" matrix being applied to the movie clip. What this means is that if you attempt to alter properties of that matrix directly, they will not be reflected in the transformations applied to the movie clip. Instead, you would need to work from a copy and then re-assign that altered copy back to the matrix property of the transform object for it to be applied.

```
my_mc.transform.matrix.translate(5, 0); // no change
```

```
var my_matrix = my_mc.transform.matrix;
my_matrix.translate(5, 0);
my_mc.transform.matrix = my_matrix; // changes my_mc's position
```

Interact with this movie clip and see how each method affects it when the matrix operations are called on the movie clip's transform matrix.




**Note:**

Like many other of the new classes bundled with Flash 8, the Matrix class has a `clone()` method which allows you to easily create a copy of an existing matrix. Matrices accessed from a movie clip's transform object are automatically cloned.

When a movie clip is a child of another movie clip and both movie clips have transformations applied to them, the resulting visuals of the child movie clip appear as a combination of its own transforms with those applied by its parent.

## Matrix Multiplication

Matrices are much like complex numbers. You can perform common number operations such as addition and even multiplication. Multiplication, in particular, is an important operation when dealing with transform matrices. Matrix multiplication is what is used to apply the transformations specified in transformation matrices. In fact, the formulas previously mentioned for deriving `x'` and `y'` from a matrix is a result of matrix multiplication on a point or row vector.

A *row vector* is an 1 x *n* matrix - a single row matrix of *n* values. A vector often represents a point in a 2D or 3D space. For our purposes, a vector is a 1 x 3 matrix consisting of values x, y, and the number 1 representing a point in 2D space in Flash.

$$\begin{bmatrix} x & y & 1 \end{bmatrix}$$

Though x and y are the only important values in this vector, the 1 is still necessary much in the same way u, v, and w are in a transform matrix.

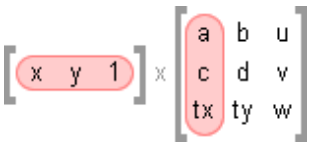
Such a vector can be multiplied by a transformation matrix to have that transformation applied to the point it

represents. With matrix multiplication, however, it is required that the first matrix in the operation have a number of columns equal to the number of rows in the second matrix . Since a vector here is 1 x 3 and the transformation matrix 3 x 3, this requirement is met with row vector on the left side of the equation.

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ tx & ty & 1 \end{bmatrix}$$

The reason for this requirement revolves around how matrix multiplication is evaluated. In matrix multiplication, the resulting matrix is an *m* x *n* matrix where m is the number of rows in the first matrix and n is the number of columns in the second. This requirement also makes order important. For example, given two matrices `A` and `B`, `A*B` does not always equal `B*A`. There are situations where that is the case, and you'll see some of that later on.

Each new value in the resulting matrix is a summation of the products of the values that correspond to the first matrix's columns with the second matrix's rows. For example, given a matrix * vector multiplication, the first value in the resulting vector would be




```
x*a + y*c + 1*tx
```

Doing the same for each row and its respective column in the vector gives the vector

$$\begin{bmatrix} x*a+y*c+1*tx & x*b+y*d+1*ty & x*0+y*0+1*1 \end{bmatrix}$$

or

$$\begin{bmatrix} x*a+y*c+tx & x*b+y*d+ty & 1 \end{bmatrix}$$

You'll notice that this results in the same formula seen before used to evaluate `x'` and `y'`.

```
x' = a*x + c*y + tx
y' = b*x + d*y + ty
```

**Note:**

When using a matrix of the alternative orientation, multiplication is an operation that is handled a little differently. Instead of using row vector * matrix, you would use matrix * column vector. A *column vector* is just like a row vector only written as a column in the form of a *m* x 1 matrix.

$$\begin{bmatrix} a & c & tx \\ b & d & ty \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

The end result is remains the same, only a it exists as a column vector instead of a row vector.

Vectors are commonly used in matrix multiplication to find a new point resulting from an applied transformation. But matrices can also be multiplied by other matrices to create a new transformation matrix that combines the transformations of those multiplied. This is the same process used in determining a new transformation of a child movie clip when both itself and its parent have transformations applied to them.

Consider 2 transformation matrices to be multiplied.

$$\begin{bmatrix} a1 & b1 & 0 \\ c1 & d1 & 0 \\ tx1 & ty1 & 1 \end{bmatrix} \times \begin{bmatrix} a2 & b2 & 0 \\ c2 & d2 & 0 \\ tx2 & ty2 & 1 \end{bmatrix}$$

Since the first has columns equal to the number of rows in the second, multiplication is possible. The same method used with multiplication with the vector is used here only some additional columns are provided by the second matrix that need to be considered in the calculation.

$$\begin{bmatrix} a1 & b1 & 0 \\ c1 & d1 & 0 \\ tx1 & ty1 & 1 \end{bmatrix} \times \begin{bmatrix} a2 & b2 & 0 \\ c2 & d2 & 0 \\ tx2 & ty2 & 1 \end{bmatrix}$$

After each row and column as been multiplied and added, you get the following

$$\begin{bmatrix} a1*a2+b1*c2+0*tx2 & a1*b2+b1*d2+0*ty2 & a1*0+b1*0+0*1 \\ c1*a2+d1*c2+0*tx2 & c1*b2+d1*d2+0*ty2 & c1*0+d1*0+0*1 \\ tx1*a2+ty1*c2+1*tx2 & tx1*b2+ty1*d2+1*ty2 & tx1*0+ty1*0+1*1 \end{bmatrix}$$

which reduces to

$$\begin{bmatrix} a1*a2+b1*c2 & a1*b2+b1*d2 & 0 \\ c1*a2+d1*c2 & c1*b2+d1*d2 & 0 \\ tx1*a2+ty1*c2+tx2 & tx1*b2+ty1*d2+ty2 & 1 \end{bmatrix}$$

You can see that this continues to follow the same style of 3x3 transformation matrices where u, v, and w are 0, 0, and 1.

We can now apply this to a simple example. Here are two transformation matrices, one which scales x by 200% and one which skews y by a factor of 1.

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 2*1+0*0+0*0 & 2*1+0*1+0*0 & 2*0+0*0+0*1 \\ 0*1+1*0+0*0 & 0*1+1*1+0*0 & 0*0+1*0+0*1 \\ 0*1+0*0+1*0 & 0*1+0*1+1*0 & 0*0+0*0+1*1 \end{bmatrix}$$

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

In the resulting transformation matrix, the skew factor has been multiplied by 2.

Now lets see what happens when the order of the matrices are switched. Do you think the result will be the same?

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1*2+1*0+0*0 & 1*0+1*1+0*0 & 1*0+1*0+0*1 \\ 0*2+1*0+0*0 & 0*0+1*1+0*0 & 0*0+1*0+0*1 \\ 0*2+0*0+1*0 & 0*0+0*1+1*0 & 0*0+0*0+1*1 \end{bmatrix}$$

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

The results are, in fact, not the same. The scaling is no longer being applied to the skew, but has instead simply been included with it.

**Note:**

Remember any matrix multiplied by the identity matrix will result in no change of the original matrix.

Don't be worried if a lot of this seems overwhelming. You don't really need to know how matrix multiplication works. In fact, Flash's Matrix class provides methods for performing these operations for you, both on points (`transformPoint()`) as well as matrices (`concat()`). Chances are you will never have to manually perform multiplication as see above. Nevertheless, knowing how it works can help provide a better overall understanding.

```
import flash.geom.Matrix;

var scale_matrix, skew_matrix;

scale_matrix = new Matrix(2, 0, 0, 1, 0, 0);
skew_matrix = new Matrix(1, 0, 1, 1, 0, 0);
skew_matrix.concat(scale_matrix);
trace(skew_matrix); // (a=2, b=0, c=2, d=1, tx=0, ty=0)

scale_matrix = new Matrix(2, 0, 0, 1, 0, 0);
skew_matrix = new Matrix(1, 0, 1, 1, 0, 0);
scale_matrix.concat(skew_matrix);
trace(scale_matrix); // (a=2, b=0, c=1, d=1, tx=0, ty=0)
```

Like translate, scale, rotate, and identity, concat also alters the current matrix instance as opposed to return a new matrix.

When multiplying with row vectors, Flash point instances can be used with a matrix's transformPoint command.

```
import flash.geom.Matrix;
import flash.geom.Point;

var scale_matrix, original_point, scaled_point;

scale_matrix = new Matrix(2, 0, 0, 1, 0, 0);
original_point = new Point(10, 10);
scaled_point = scale_matrix.transformPoint(original_point);
trace(scaled_point); // (x=20, y=10)
```

Notice that with `transformPoint`, a new, transformed point is returned. Also notice that the order in which the multiplication is written in Flash using either `concat` or `transformPoint` is opposite that written when using matrices. For example given two matrices `A` and `B` and a point `P`

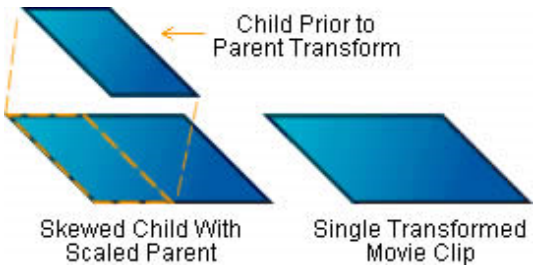
```
A*B -> B.concat(A);
P*A -> A.transformPoint(P);
```

In terms of the parent-child movie clip relation with matrix multiplication, you have

```
Child.concat(Parent);
```

## Concatenated Matrices

Lets say we have 2 movie clips, the first of which is a child of the second. Take the first, child movie clip and skew it along x by a factor of 1. Then, for the second, parent movie clip, scale it along x by 200%. The resulting appearance of the child movie clip from the main timeline is a movie clip that appears to be both scaled and skewed. This same appearance can be duplicated by a single clip not nested within a transformed parent. The transform matrix of such a clip would equal that of the original child movie clip's transform matrix multiplied by the transform matrix belonging to it's parent.




A parent movie clip's transform matrix is used in matrix multiplication to further alter the transformations of its children. Keep in mind that the actual matrix associated with a child movie clip does not contain these changes. It remains unique to that movie clip and contains only the transformations applied to it directly, not any of those that its parent or any other parent movie clips may be adding on to that. Flash does, however, provide an additional property in a movie clip's transform object that provides all transforms from all parents in the form of a single matrix. This is the `concatenatedMatrix` property. It represents the result of multiplication of all the matrices affecting a nested movie clip.

```
var parent_matrix = parent_mc.transform.matrix;
var child_matrix = parent_mc.child_mc.transform.matrix;
var childconcat_matrix = parent_mc.child_mc.transform.concatenatedMatrix;

child_matrix.concat(parent_matrix);

trace(child_matrix); // (a=2, b=0, c=2, d=1, tx=0, ty=0)
trace(childconcat_matrix); // (a=2, b=0, c=2, d=1, tx=0, ty=0)
```

You can think of the `concatenatedMatrix` property being to matrices what `globalToLocal` is to points, it provides the transformations needed to match that of the concatenated transformation of the targeted matrix. Using a concatenated matrix, you can transform a movie clip in the main timeline to match the transformation of a nested movie clip with multiple transforms applied by multiple parents.

## Inverse Matrices

You also have the capability of removing all transformations from a nested movie clip to make it seem as though it is not being transformed at all. The concatenated matrix alone, however, will not provide this for you. What you need is an inverse matrix.

An *inverse matrix* is a square matrix where there exists another unique matrix that, when multiplied with the inverse, results in the identity matrix. Both matrices are then considered *invertable* and are inverses of each other. All Flash transformation matrices are invertable meaning for each transformation matrix, there exists another matrix that when multiplied by the transformation matrix will give you the identity matrix. Given the identity matrix `I` and a matrix `A`, there exists a matrix `B` so that

```
A*B = I
```

and

```
B*A = I
```

The method of obtaining an inverse matrix manually is a little complex. It involves finding the adjoint of the matrix and dividing it by its determinant. The process is slightly simpler given the form of our affine transformation matrices but the details of which is still beyond the breadth of this tutorial. It reduces to the following.

Given the matrix

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ tx & ty & 1 \end{bmatrix}$$

its inverse can be found using

$$\begin{bmatrix} d/(a*d-b*c) & -b/(a*d-b*c) & 0 \\ -c/(a*d-b*c) & a/(a*d-b*c) & 0 \\ (c*ty-d*tx)/(a*d-b*c) & -(a*ty-b*tx)/(a*d-b*c) & 1 \end{bmatrix}$$

As with matrix multiplication and the `concat` method, the Matrix class in Flash is able to provide a method that makes life a bit easier in finding inverse matrices. Using the `invert()` method, you can easily convert a matrix into its inverse with minimal hassle.

```
import flash.geom.Matrix;

var my_matrix = new Matrix(2,3,5,7,2,4);
trace(my_matrix); // (a=2, b=3, c=5, d=7, tx=2, ty=4)

my_matrix_inverse = my_matrix.clone();
my_matrix_inverse.invert();
trace(my_matrix_inverse); // (a=-7, b=3, c=5, d=-2, tx=-6, ty=2)

my_matrix_inverse.concat(my_matrix);
```

```
trace(my_matrix_inverse); // (a=1, b=0, c=0, d=1, tx=0, ty=0)
```

As seen above, multiplying the inverse by the original matrix results in the identity matrix.

Using an inverse matrix, a movie clip can have all transformations from all parent movie clips removed by multiplying its current matrix by the inverse of its concatenated matrix.

```
var concat_matrix = parent_mc.child_mc.transform.concatenatedMatrix;
concat_matrix.invert();
concat_matrix.concat(parent_mc.child_mc.transform.matrix);
parent_mc.child_mc.transform.matrix = concat_matrix;
```

## Manipulating Transformation Matrices

Now that most of the basics of transformation matrices has been covered, its about time that some of this knowledge is put to use. It has only been with Flash 8 that now, through ActionScript, we can manipulate movie clips as easily as you could in the Flash IDE (Integrated Development Environment), all thanks to transform matrices.

First we can work with what we know. The basic methods for matrix transformation are `translate`, `scale`, and `rotate`. If you've forgotten, there is no skew. How about we change that?

In order to make a custom skew method that functions like the other methods, you would need to consider that the skew be applied on top of the other transformations of the matrix including translation. If you attempted to set the skew properties (b and c) of a matrix that wasn't the identity matrix directly, it is possible that, given whatever transformations currently exist within the matrix, you could corrupt the transformation beyond a simple skew. Luckily, matrix multiplication is a process that would apply one transformation to another on top of any transformations that already existed within the original matrix. That means adding a simple skew is just a matter of multiplying an existing matrix by a skewed identity matrix.

```
function skew(matrix, x, y){
        matrix.concat( new flash.geom.Matrix(1, y, x, 1, 0, 0) );
}
```

This simple `skew` function uses `concat` with a new identity matrix whose c and b values are set to the `x` and `y` values passed. As a result, the original matrix is multiplied by the new skewed matrix skewing it appropriately regardless of any prior transformations. Like translate, scale, and rotate, this skew function also affects the current translation properties of the matrix. The only difference here is that, since this skew method is not part of the Matrix class in Flash, its usage requires that the target matrix be passed as an argument.

```
skew(my_matrix, 1, 0);
```

One thing you'll notice about methods like `scale` (and this new `skew` method) is that transformations applied using these are not based around the current transforms of the matrix to which they are applied. They take into account the preexisting transformations, but don't base their scale or skew on those transformations. To more clearly demonstrate what I'm talking about, consider a movie clip rotated 45 degrees. When considering a scale being applied to this clip, there are conceptually two possible outcomes; scaling as though there were no rotation or scaling based on the rotation.


Using `scale` and the other Matrix class methods, you get scaling on top of that rotation as though there was no

rotation as seen in the first variation, not that as shown in the second. If you wanted scaling to work with the rotation, like seen in the second, you'd actually want scaling to be applied *before* the rotation. Think about it in terms of the parent-child relationship; when the child movie clip is scaled, the scaling is in line with the rotation of the parent. This is contrary to scaling applied by the `scale` method which acts more like a rotated child being scaled by the parent. The matrix transform methods work as parent transformations adding to the existing children. To get the opposite affect, all you need to do is use matrix multiplication but change the order of in which the matrices are multiplied.

For this switch-around, you'll need to start with the method's transformation and then multiply in the original transformation on top of that. For the starting matrix, you simply use the desired method on an identity matrix. With that, we can create new methods that affect "inner" transformations.

```
function innerScale (matrix, x, y){
        var scale_matrix = new flash.geom.Matrix();
        scale_matrix.scale(x, y);
        scale_matrix.concat(matrix);
        return scale_matrix;
}

function innerSkew (matrix, x, y){
        var skew_matrix = new flash.geom.Matrix();
        // assume skew function for matrices is defined
        skew(skew_matrix, x, y);
        skew.concat(matrix);
        return skew_matrix;
}

function innerRotate (matrix, angle){
        var rotate_matrix = new flash.geom.Matrix();
        rotate_matrix.rotate(angle);
        rotate_matrix.concat(matrix);
        return rotate_matrix;
}
```

Take notice that these methods actually return a matrix instead of altering the original. For the purposes of these examples, it was just easier to that. Here's how one could be applied to a movie clip.

```
var my_matrix = my_mc.transform.matrix;
my_matrix = innerScale(my_matrix, 2, 1);
my_mc.transform.matrix = my_matrix;
```

In using these transformations, you probably noticed that translation was not affected. This is because the base transformation actually had no translation, so whatever transformation was reapplied from the original matrix just resulted in using the original translation values from that matrix.

What if, however, you don't want translation to be affected in the original methods as well? Flash actually already provides such a method that ignores translation when translating points. It's the `deltaTransformPoint()` method and it acts just like the `transformPoint` method except that transformations are not applied to the translation properties. Other matrix transform methods do not have similar counterparts, but you could easily provide that functionality simply by storing the original location of a matrix prior to transformation and just reassign those values afterwards. That's right, no complicated matrix operations involved.

```
function deltaScale (matrix, x, y){
        var position = new flash.geom.Point(matrix.tx, matrix.ty);
        matrix.scale(x, y);
        matrix.tx = position.x;
        matrix.ty = position.y;
}
```

```
function deltaSkew (matrix, x, y){
        var position = new flash.geom.Point(matrix.tx, matrix.ty);
        // assume skew function for matrices is defined
        skew(matrix, x, y);
        matrix.tx = position.x;
        matrix.ty = position.y;
}

function deltaRotate (matrix, angle){
        var position = new flash.geom.Point(matrix.tx, matrix.ty);
        matrix.rotate(angle);
        matrix.tx = position.x;
        matrix.ty = position.y;
}
```

These methods, like the original transform methods, but unlike the inner methods, transform the values within the matrix in which they are used instead of returning a new matrix.

Alternatively you could manually apply the transformations to each property of a matrix manually as outlined in the definitions of the original methods, but it's usually easier to just save and reassign the last translation values, especially for operations like rotation where the calculations can be a bit extensive.

```
function deltaRotate (matrix, angle){
        var sin = Math.sin( angle );
        var cos = Math.cos( angle );
        var a = matrix.a;
        var b = matrix.b;
        var c = matrix.c;
        var d = matrix.d;
        matrix.a = a*cos - b*sin;
        matrix.b = a*sin + b*cos;
        matrix.c = c*cos - d*sin;
        matrix.d = c*sin + d*cos;
}
```

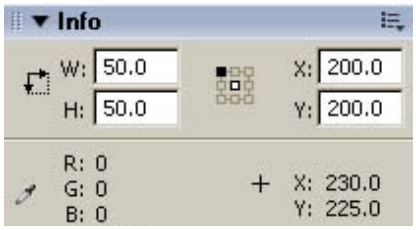
## Transformation Conversions

Sometimes you may need to compare transformation matrix values with those that exist in the Flash IDE or to those used in more generic ActionScript properties. Transformation matrix values, however, do not always correlate with their other Flash counterparts. Translation, for instance, relates directly to those values expressed as x (_x) and y (_y) position in Flash, but scale, skew, and rotation are handled slightly differently.

### translation

Translation is, for the most part, the easy one. The tx and ty properties in a matrix directly relate to the _x and _y properties provided by ActionScript to reflect a movie clip's position within it's parent timeline. The values present in the Flash IDE in the property inspector, however, can differ.

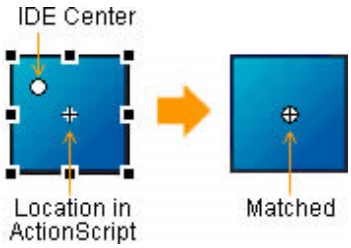
The Flash IDE allows for position to be based on two separate locations. That of the top left of the movie clip or that of the movie clip's center. The option of using either is presented in the coordinate grid of the info panel



A: 100%

This center option does not always represent the local 0,0 location, or *origin*, of the movie clip in question, though. It's actually related to the transformation center point provided by the transform tool and is represented as a small black and white circle in Flash. By default this is placed in the center of a movie clip instance but can be adjusted using the transform tool. In order for translation values, or even _x and _y for that matter, to match up with this transformation center point, it would need to be placed on the origin of the movie clip which is represented by a black and white cross.

The transformation center point is not accessible by ActionScript, so keep that in mind as you are working with movie clips in Flash. It is often best to keep the registration point aligned with the movie clip's origin so you have a direct relation with the values you see in the Flash IDE and those accessible through ActionScript.

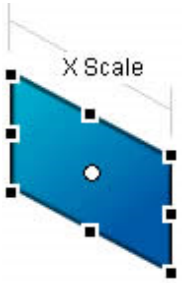



```
var my_matrix = my_mc.transform.matrix;

trace("x: " + my_matrix.tx);
trace("y: " + my_matrix.ty);
```

**scale**

Scale mostly relates directly to the a and d properties of a transformation matrix but is complicated by the effects of skewing. Without skewing, the a and b properties directly relate to the scale properties presented by flash - those in the IDE's transform panel and those given by _xscale and _yscale in ActionScript - with the simple exception of a matrix's scale being based on 1 while Flash bases scale on 100 for a scale of 100%.

When skewing is involved, the scale of an axis changes as it continues to stretch based on the relationship to the other axis, either x to y or vice versa. Because of these relations, its only a matter of using Pythagoreans theorem to acquire the correct scale factor



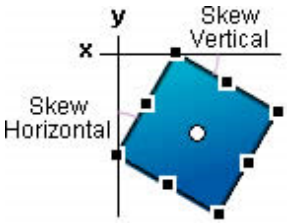

```
var my_matrix = my_mc.transform.matrix;

trace("x scale: " + Math.sqrt(my_matrix.a * my_matrix.a + my_matrix.b * my_matrix.b));
trace("y scale: " + Math.sqrt(my_matrix.c * my_matrix.c + my_matrix.d * my_matrix.d));
```

**skew**

Prior to transformation matrices in Flash 8, skew values were not accessible through ActionScript. The Flash IDE, however, did, and still does, provide values for skew in the transform panel in the form of values of

rotation (in degrees). This is slightly different than the values used to interpret this transformation in the actual transform matrix. Values there represent a ratio by which one axis scales according to the other. They are not rotation or angle values.

The rotation values given by Flash represent the angles created on the opposite axis as a result of that axis being skewed. For example, skewing the x axis creates an angle between the now tilted y axis and its original vertical orientation.




Depending on the amount of skew, this angle varies. Using an arbitrary point along either axis and transforming it with a movie clip's matrix will allow us to obtain the value of this rotation using arctangent. An additional 90 degree offset is needed with x skew compensate for the orientation of the y axis.

```
var my_matrix = my_mc.transform.matrix;

var px = new flash.geom.Point(0, 1);
px = my_matrix.deltaTransformPoint(px);
var py = new flash.geom.Point(1, 0);
py = my_matrix.deltaTransformPoint(py);

trace("x skew: " +((180/Math.PI) * Math.atan2(px.y, px.x) - 90));
trace("y skew: " +((180/Math.PI) * Math.atan2(py.y, py.x)));
```

**rotation**

If you know skew, you know rotation. Flash uses the x skew angle for rotation.

```
var my_matrix = my_mc.transform.matrix;

var px = new flash.geom.Point(0, 1);
px = my_matrix.deltaTransformPoint(px);

trace("rotation: " +((180/Math.PI) * Math.atan2(px.y, px.x) - 90));
```

## Applications

With the basics down, its time that matrices can be put to work in some (somewhat) real-world applications of this knowledge, both in manipulating movie clips and for use in BitmapData operations.

**Shaking Smilies**
Here, a group of smiley movie clips are wildly distorted within a couple of nested, transforming movie clips. When clicked, each will appear to break out of the transformations of its parents and present itself undistorted to the user by using invert with concatenated matrices.
[ view | download ]

As previously demonstrated, a nested, transformed movie clip can be fairly easily taken out of its transformations using `invert` and a concatenated matrix. This example does that but adds in the animation of the movie clip transforming from its original transform to that of the new, inverted transform. It does this using a custom `matrixInterpolate()` method.

The Point class in Flash has an `interpolate()` method that allows you to find any point along the line connecting two other points. Basically, this is a tween method that finds those "in-between" values. The Matrix class does not have such a method, so one was created for this example. All it does is return a new matrix with values in between each of the properties of two other matrices given a value t which is between 0 and 1.

```
function matrixInterpolate(m1, m2, t){
        var mi = new flash.geom.Matrix();
        mi.a = m1.a + (m2.a - m1.a)*t;
        mi.b = m1.b + (m2.b - m1.b)*t;
        mi.c = m1.c + (m2.c - m1.c)*t;
        mi.d = m1.d + (m2.d - m1.d)*t;
        mi.tx = m1.tx + (m2.tx - m1.tx)*t;
        mi.ty = m1.ty + (m2.ty - m1.ty)*t;
        return mi;
}
```

Combining this with an onEnterFrame event, the smilies can animate from their original, nested transforms to the inverted transform and back again.

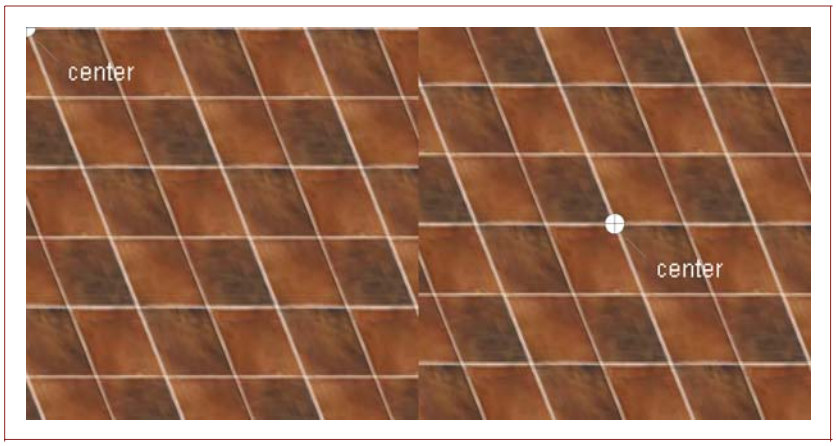
One thing about the inverted transform, however, is that it is not just an inverted concatenated matrix. Just inverting it would make it an identity matrix and throw the movie clip to position 0,0 with normal scale (small for the smilies in this example). Instead, another matrix is combined with the resulting inverted identity to place the selected smiley in the location on the right at a large scale.

**Mouse Walking on the Floor**
This example shows how a transform matrix can be used with `beginBitmapFill()` distort a a tiled image (used for the floor) as it shifts up the screen and an equally distorted mouse movie clip appears to walk on it. Additionally, it shows how an object (can of cheese) can be moved along with the transformed floor.
[ view | download ]

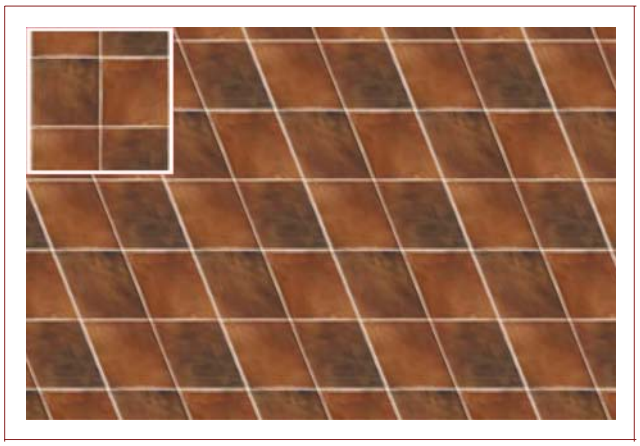
The following image is used for the tiled floor in this example.

Using `beginBitmapFill`, it will be drawn in a square that is sized to be with width and height of the movie (300 x 300). For the skewing effect, a matrix is used with `beginBitmapFill` to apply a skewed transformation that changes based on the mouse's x location.

When dealing with matrix transformations in `beginBitmapFill`, as with any bitmap operation, the "center" of the transformation will be located at the upper left of the bitmap being used. In this example, the center of the transformations is desired to be in the center of the movie. Using `translate`, that center can change from the upper left to the center of the movie. Because `beginBitmapFill` will tile the image being drawn, no imagery is lost as a result of the change in position. Instead, you just get a shift which may or may not even be initially evident. In applying a skew, however, the difference becomes more obvious.




You'll have to wonder, if translating centers the point of origin for the skew, how does the floor appear to be moving? Wouldn't that relate to translating? In fact, it would, but not with that `beginBitmapFill`. Two drawings with `beginBitmapFill` are actually used to get the effect of the floor. One is used center and skew the floor, and the other, drawn beforehand, is used to alter the original bitmap to shift its graphics by the amount of movement needed - translating it. This altered bitmap is then used as the bitmap drawn into the skewed floor



With that, the floor appears to move correctly regardless of the skew and always from the center of the movie.

The other elements in the movie are the mouse and a can of cheese. The mouse is simply placed at the center of the floor and set to match the skew applied to the floor. This transforms it with the floor.

The cheese is a little different. It actually doesn't have any transformations applied to it. This is because it is actually sticking straight up as opposed to laying down on the floor. If on the floor it would shift with the floor, but since the (perceived) vertical axis does not change, neither does the can of cheese. It does, however, need to move with the floor.

As the cheese moves vertically with the floor at the same level of shifting applied to the original tile bitmap, it will need to make sure that horizontally it remains at the correct location. This location will vary based on the amount of skew as well as its distance from the center of the skew - the center as placed by the translation of the skewed `beginBitmapFill`. In fact, when vertically positioned with that center, the horizontal position is not affected by the skew at all. Only incrementally from that position does the skew affect it. What you end up getting is a formula that looks something like this:

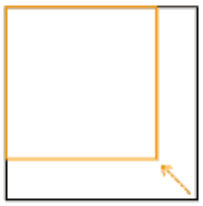
```
cheese_x_offset = center_x + (cheese_y - center_y) * skewAmount;
```

As the difference between the cheese's y position and the y center, the amount of skew affects it more making the offset of the cheese from the x center increasingly greater.

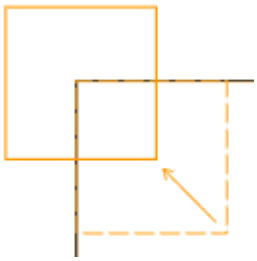
**Black Hole**
By rotating and scaling a matrix used in the BitmapData's `draw()` method, the effect of a black hole can be created, sucking in and enveloping all imagery seen on the screen.
[ view | download ]

There are 3 kinds of transformations applied to the matrix causing the black hole effect, rotation, scale and translation. Rotation gives the swirling effect, scale makes the graphics scale down and sink inward, and translation makes sure it's all centered within the screen. The bitmap representing the black hole continues to, not only draw the fountain of stars emitting from the position of the cursor, but also its last graphical appearance which gives the effect of the trail that eventually gets sucked into the transformations

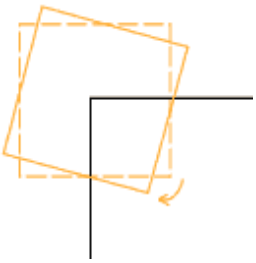
Rotation and scaling are pretty straightforward, but when it comes to translation, some extra effort is needed to make sure that translation is actually correct to take in account not only scaling but also rotation. Remember, the center of a bitmap matrix is in the upper left. This black hole effect is centered in the middle of the bitmap/screen. This is handled is through a multi-step process that involves scaling, centering based on the scale, rotating, and repositioning.



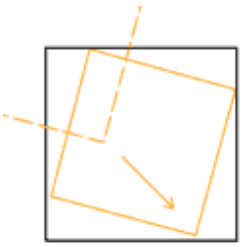
**1**. First, the matrix is scaled.



**2**. Then, based on that scale, the matrix is translated to position it's center at 0, 0 (center is based on the bitmap it will be affecting).



**3**. From that position it is rotated. Because of the previous shift in translation, rotation is based around the center.



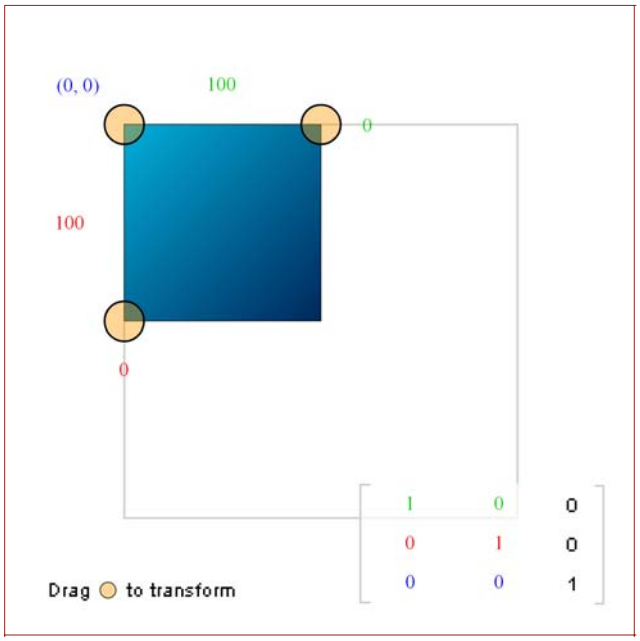
**4**. Finally, the matrix is translated back so that its center is centered within the effect.

**3D Picture Cube**
The classic rotating cube using images for its sides. Thanks to Flash 8 and having access to a movie clip's transform matrix, this example becomes much simpler to implement.
[ view | download ]

Rotating cubes of images like these have been around since Flash 5, and maybe even before that. The idea behind this effect is transforming movie clips so that they match the sides of a cube being rotated in a 3D space

(sans perspective). Before Flash 8, movie clip nesting was needed to get the transformations needed, notably in terms of skewing a movie clip. Now, however, with Flash 8, the effect is easier than ever to pull off thanks to developers being able to directly access a movie clip's transform matrix. In fact, once you get past the 3D, altering the transform matrix is nothing - no complicated equations or trigonometry needed. All you have to do is find the difference of position of some points making up the side of your cube and you pretty much have it.

Transform the following movie clip using 3 points to define the transformation and witness the obvious relationship between the difference of positions and the transformation matrix.




## Conclusion

This tutorial has covered transformation matrices; what they are and how they can be used. It discussed the Matrix class in Flash and the methods it has used to manipulate matrices. Though some of the math behind those operations has also been covered, it is most likely not knowledge that you will need to rely on when you decide to code with matrices.

Hopefully you now at least have a better understanding of the goings on of transform matrices, and with it, a new level of control in using and manipulating elements in Flash.