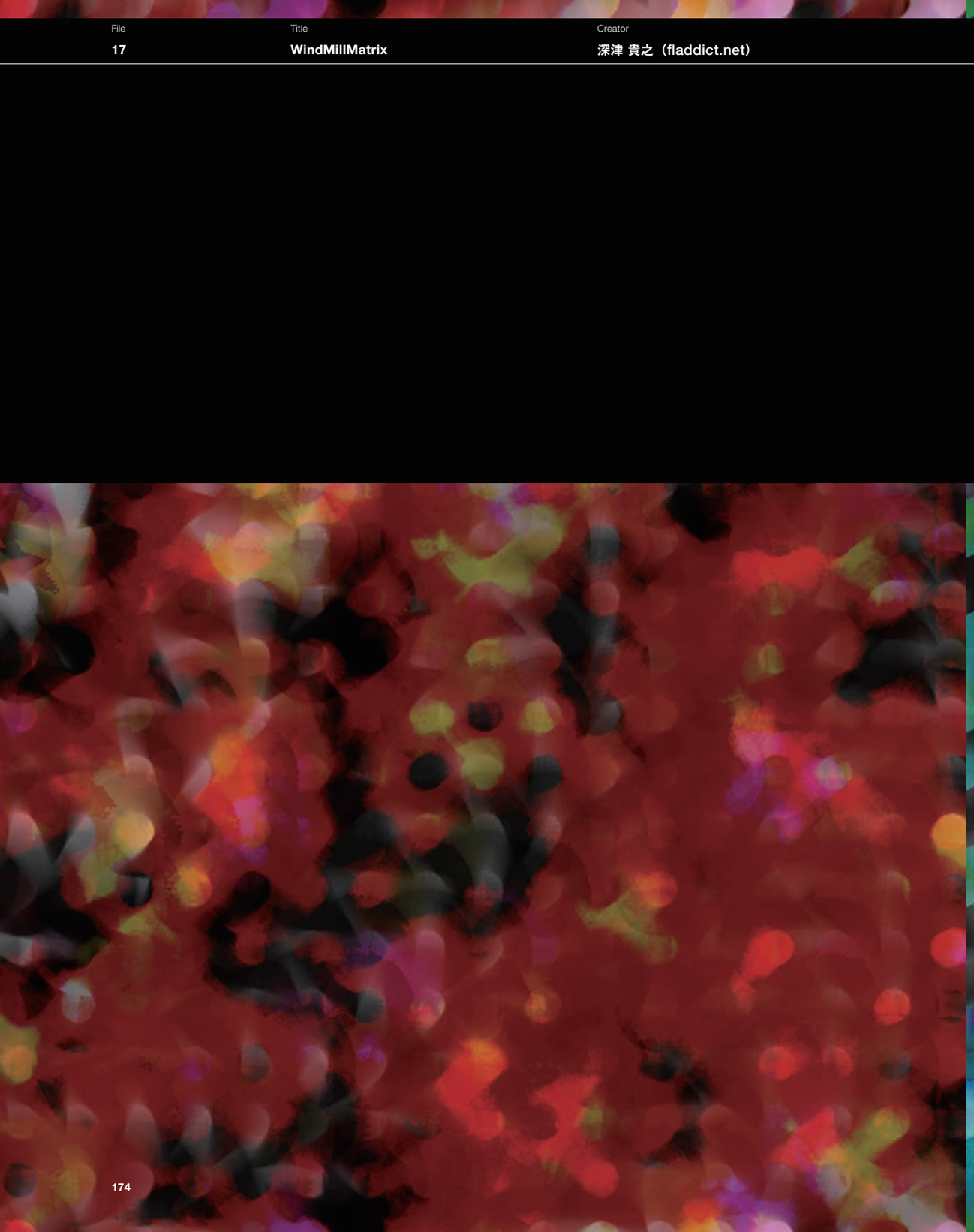


WindMillMatrix

Creator
深津 貴之（fladdict.net）

Flash 8 のビットマップ機能を使う

Flash 8 の描画機能を使った実験の 1 つ。
「オブジェクトを画面に書き出す」→「画面をクリアする」動作で残像エフェクトを実現する。

Title
WindMillMatrix

Sample URL
http://www.fladdict.net/dotfla/windmillmatrix/

Archive
windmillmatrix.zip

File
17

スクリプト
ActionScript 2.0

対応プレーヤー
Flash Player 8以上

制作アプリケーション
Flash 8

発案～デザイン
Flash 8 の描画機能の実験

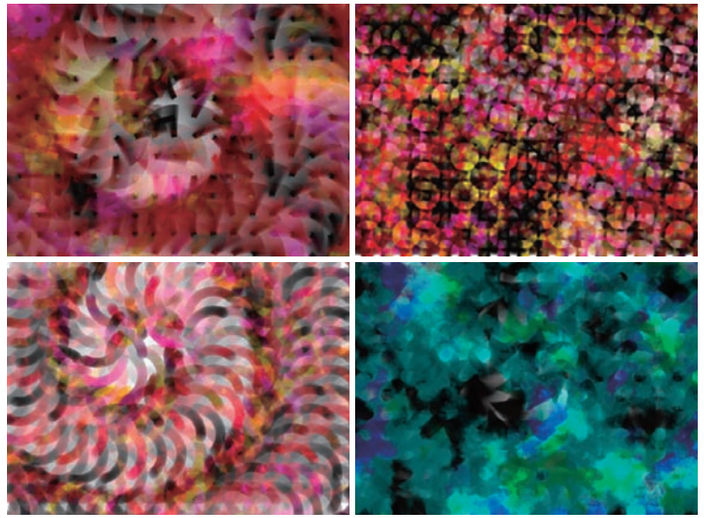
「WindMillMatrix」は、Flash 8 の描画機能の実験として作った万華鏡のような不思議な模様を描くデモです。これは、僕が Flash 8 の新機能を学ぶために作った実験の 1 つです。この章では、比較的小さめのデモを通じて、これ以降の僕のサンプルで使用する Flash 8 の新機能についての大まかな使い方を見ていきたいと思います。

サンプルのコンセプト

とても複雑で不思議な模様を描くこのデモですが、実は行われていることは非常に単純だったりします。この作品で行われているのは単純に N × M の格子状に配置された長方形が、軌跡を残しながらグルグル回転するだけです。ちょうど、縁日の屋台などにズラッと並べられた風車が一斉に回転するようなイメージです。「WindMillMatrix」というタイトルは、そんな様をそのまま名前にしたものです。

ただし、回転する速度はマウスとの距離によって変化します。マウスと近い風車ほどゆっくりと、マウスから離れれば離れるほどすばやく回転するわけですが、この速度のズレが砂の上の波紋のような不思議なウネリを生み出しています。

各要素の回転速度の違いがパターンを生み出す



サンプルのロジック

それほど長いコードではないので、まず全体を見てみましょう。

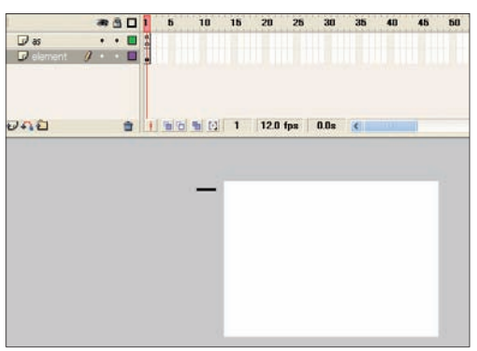
SOURCE WindMillMatrix のコード

```
import flash.display.*;
import flash.geom.*;

// 初期設定
var element:MovieClip=mc;    // 描画に用いる MovieClip
var yNum:Number = 14;
var xNum:Number = 19;

// 描画用ビットマップ
var screen:BitmapData; // 実際に表示される画像
var cach:BitmapData;    // 一時保存用の画像

var objects:Array; // オブジェクトの情報を格納する配列

init()  // 初期設定の実行

// 初期設定
function init(){
   // 画面表示用 Bitmap の作成
   screen = new BitmapData(Stage.width, Stage.height);
      _root.attachBitmap(screen, 1);

   cach = screen.clone();    // エフェクト用のキャッシュの作成
   reset();
}

// 開始準備
function reset(){
   screen.fillRect( screen.rectangle, 0xffffff)   // 画面の塗りつぶし

   objects = new Array();    // オブジェクト格納用の配列の初期化

      // オブジェクトの作成
   for(var y=0; y<yNum; y++){
        for(var x=0; x<xNum; x++){
             var obj = new Object();
             obj.x = x * 30;
             obj.y = y * 30;
             obj.rotation = 0;
             obj.colR = Math.random()*255;
             obj.colG = Math.random()*obj.colR;
             obj.colB = Math.random()*obj.colR;
             objects.push(obj);
        }
   }
   onEnterFrame = step;
}

// マウスを押したらリセット
function onMouseDown(){
   reset();
}
```

次のページに続く

```
// 毎フレーム行われる処理
function step(){
    var cach = screen.clone();    // 現在の画面を保存
    screen.fillRect(screen.rectangle, 0xffffff);   // 画面をクリア

    // オブジェクトを描画
    for(var i=0; i<objects.length; i++){
         var obj = objects[i];
         var dist = getDistanceFromMouse(obj);
         obj.rotation += Math.sqrt(dist)*0.004;
         drawElement( obj );
    }

    // 画面に残像エフェクト
    var colt = new ColorTransform(1,1,1,0.99,0,0,0,0);
    screen.draw(cach, new Matrix(), colt);
}

// マウスとオブジェクトの距離を計算して返す
function getDistanceFromMouse(obj){
    var dx = _xmouse - obj.x;
    var dy = _ymouse - obj.y;
    return Math.sqrt(dx*dx + dy*dy);
}

// オブジェクトの内容を元に、エレメントを描画する
//MC を筆としてオブジェクトを描画
function drawElement(obj){
    // 変形マトリックスの作成
    var mat = new Matrix();
    mat.rotate( obj.rotation);
    mat.scale(1,1);
    mat.translate(obj.x, obj.y);
    // 色変換用トランスフォームの作成
    var colt = new ColorTransform(0,0,0,0.2,obj.colR, obj.colG, obj.colB, 0);
    //MC を筆としてオブジェクトを描画
    screen.draw( element, mat, colt);
}
```

サンプルの要素は1フレーム目のスクリプトと MC がすべて



このスクリプトを 1 フレーム目に書き込み、またステージの表示領域外に "mc" という名前で 50 × 8 ピクセルの大きさの長方形のMCを配置します。

準備はこれだけです。基本的に、僕は実験的なコンテンツを作るときには、トゥイーンをほとんど使用しません。こんな感じで、スクリプトだけで実験を行っています。

スクリプト
マウスとの距離で回転速度を変える

それでは、コード内で使われている Flash 8 の新機能を紹介していきたいと思います。

○ ロジックの流れ

スクリプトの大きな流れは、以下の 3 つに分かれます。

- 全体の初期化（init）
- 開始準備
- 毎フレームの処理

● 全体の初期化—init()

まず、コード開始時に一度だけ呼ばれる関数 init() によって初期化が行われます。具体的には、WindMill Matrix で使用する 2 つの BitmapData インスタンスとステージにアタッチされ実際に表示される screen、エフェクト処理の一時記憶に用いられる cach の初期化が行われています。

初期化後は、そのまま関数 reset() が呼び出されます。

● 開始準備—reset()

関数reset()は開始前の下準備を行います。関数init()との違いを説明すると、前者はコードの開始時に 1 回だけ呼ばれる処理を、関数 reset() は画面をリセットするたびに行われる処理が書かれています。マウスクリックによる画面のリセット時には、画面のリセットを行えば済むので、関数 init() は呼ばずにこの関数だけが呼ばれます。

関数 reset() では、3 つの処理が行われます。まずは画面を白く塗りつぶして画像をリセットします。次に、WindMillMatrix の各風車の持つ位置や回転、色といった情報を格納するオブジェクトを for 文で作成し、配列に保存しています。最後に、onEnterFrame で毎フレーム行われる処理として、関数 step() を定義しています。

○ 毎フレームの処理

関数 step() で呼ばれるフレームごとの処理は、さらに細分化すると次の 3 つに分けられます。

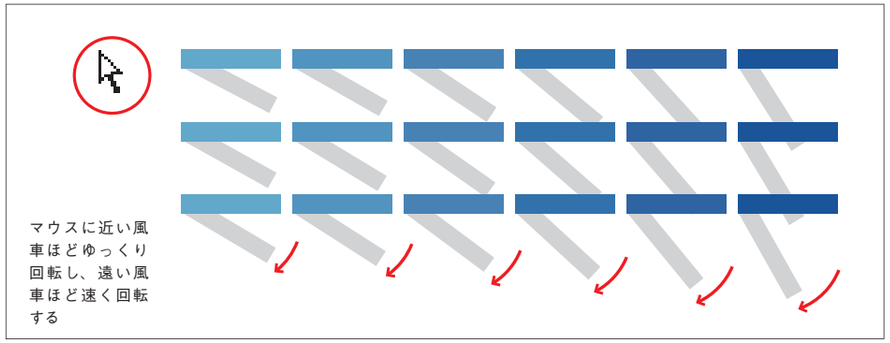
- 各エレメントの移動
- 各エレメントの描画
- 前フレームの画像を利用したモーションブラー

○ 各フレームごとの処理

● オブジェクトの更新

WindMillMatrix では、各風車の回転はマウスとの距離によって決定されています。すべての風車は for 文によって操作され、関数 getDistanceFromMouse() で求められた距離をベースに計算しています。

風車の回転



SOURCE 風車の回転部分のスクリプト

```
// オブジェクトを描画
for(var i=0; i<objects.length; i++){
     var obj = objects[i];
     var dist = getDistanceFromMouse(obj);     // マウスとの距離を計算
     obj.rotation += Math.sqrt(dist)*0.004;
     drawElement( obj );
}
```

関数getDistanceFromMouse()では、オブジェクトのプロパティx、yとマウスの座標との距離を計算して返しています。2点間の座標の距離は以下の式で求めることができます。

$$\sqrt{(\text{x方向の距離} \times \text{x方向の距離}) + (\text{y方向の距離} \times \text{y方向の距離})}$$

これで求められた式を、Math.sqrt()で√をしたところに0.004をかけて、回転速度としています。この式だと、たとえばマウスからの距離が100pxでも

$$\text{回転速度} = \sqrt{100 * 0.004} = 0.04$$

となり、ほとんど回転してないように見えるかもしれません。

これは、objのプロパティrotationに記録している回転角度が0～360度の通常の角度ではなく、1回転が0～6.28というラジアンという単位を用いているためです。なぜ普通の回転ではなくラジアンを用いるかの詳しい説明は、Matrixクラス（173ページ参照）の説明の中で行います。ここでは、角度には2種類ある程度に覚えていてください。

● オブジェクトの描画

オブジェクトをビットマップに描画する処理は、先ほどのfor文の中からオブジェクト1つごとに関数drawElement()を呼び出すことで行っています。関数drawElement()の中では、オブジェクトの持つx、y、rotationといったプロパティから変形情報を、colR、colG、colBからRGBの色情報を作成し、BitmapDataのscreenに描画しています。変形情報や色情報を扱うMatrixやColorTransformクラスについての詳しい説明は後述します。

SOURCE オブジェクトの描画

```
function drawElement(obj){
    // 変形マトリックスの作成
    var mat = new Matrix();
    mat.rotate( obj.rotation);
    mat.scale(1,1);
    mat.translate(obj.x, obj.y);
    // 色変換用トランスフォームの作成
    var colt = new ColorTransform(0,0,0,0.2,obj.colR, obj.colG, obj.colB, 0);
    //MCを筆としてオブジェクトを描画
    screen.draw( element, mat, colt);
}
```

● モーションブラーエフェクト

最後に、WindMillMatrixで行われている、BitmapDataを利用した簡単な残像エフェクトについての説明をします。毎フレームの頭、画面をクリアしたりオブジェクトを描画する前に、まず画面（screen）をビットマップcachに複製します。

SOURCE 画面をビットマップcachに複製

```
var cach = screen.clone();   // 現在の画面を保存
screen.fillRect(screen.rectangle, 0xffffff);       // 画面をクリア
```

そして風車を描画した後に、先ほどキャプチャした画像を少し薄くしてscreenへと描画します。こうすることで、全体的にぼやけたような残像の効果を出すことができるわけです。

SOURCE キャプチャした画像をscreenへ描画

```
    var colt = new ColorTransform(1,1,1,0.99,0,0,0,0);
    screen.draw(cach, new Matrix(), colt);
```

これが、WindMillMatrixの大まかな流れです。

ここで使用したBitmapData、Matrix、ColorTransformといった新機能は、以降の僕のすべてのサンプルで使用しますので、次にもう少し詳しい使い方と原理を紹介したいと思います。

Flash 8 の新機能
BitmapData、Matrix、ColorTransform の使い方

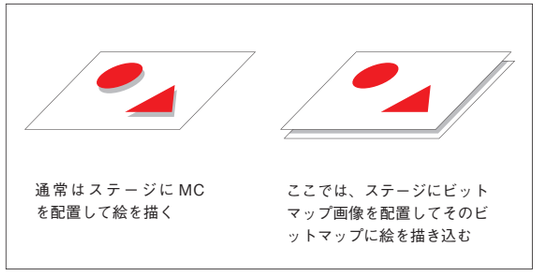
ビットマップに対する描画

Flash 8 の新機能によるもっとも大きな変化は、ビットマップ画像をスクリプトで直接制御できるようになったことです。従来の Flash では、ステージ上に MC を配置することで絵を作っていました。しかし、BitmapData を使うことで画面には新しくビットマップ画像のキャンバスだけを配置し、すべての絵はビットマップ上で描くということも可能となったのです。

ビットマップに直接描写することの最大のメリットは大きく分けて２つあります。

１つは表示速度と表示できる画像の量といったパフォーマンス面、もう１つは表示されている画像をキャプチャし、それを再加工することでさまざまなエフェクトが作り出せるということです。

BitmapData による描画



通常はステージに MC を配置して絵を描く

ここでは、ステージにビットマップ画像を配置してそのビットマップに絵を描き込む

SOURCE Bitmap への描画

```
var myBitmap = new BitmapData(Stage.width, Stage.height);
_root.attachBitmap(myBitmap, 1);
myBitmap.draw( myMovieClip, new Matrix());
```

```
import flash.display.BitmapData;
var myBitmapData = new BitmapData( 幅、高さ、透明が有効かどうか、色 );
_root.attachBitmap( myBitmapData, 深度 );
```

注意しなければならないのは、いわゆる通常のattachMovie()と異なり、関数attachBitmap()によってアタッチされたBitmapData は必ず座標 (0,0) に 100%の大きさ、角度 0 で設置されるということです。スクリプトによって設置位置を変えたり、後から移動させることはできないということです。

そのため、アタッチしたBitmapData を移動させたり拡大したい場合には、BitmapData を MC の内側にアタッチし、コンテナとなった MC のほうを動かします。

BitmapData の変形

BitmapData は必ず、位置 (0, 0) に大きさ 100%、回転 0 で設置される。

位置や大きさなどを後から変えたい場合には、BitmapData を MC 内にアタッチし MC の方を変形させる

ビットマップ画像への描画

ステージ上にアタッチした Bitmap へ実際に何かを描画するときには、draw() という関数を用います。たとえばステージ上にある myMovieClip を Bitmap 上に描画するためには、以下のように関数の第１引数に myMovieClip をわたします。

後述のMatrixやColorTransformなどを第２、第３パラメータとして用いることで、ビットマップに描き込む画像に対して拡大縮小や色の変化を行うことができます。

Matrix による画像の変形

Flash 8 から新しい画像変形を行うクラスとして、Matrix というものが提供されるようになりました。この Matrix が何なのかということを詳しく説明しようとすると、数学的な知識が必要となってしまうので、ちょっと乱暴ですが、ここではMatrixはMCの _x、_y、_xscale、_yscale、_rotation の５つの機能を１つにまとめたものだと思ってください。一般的にMatrixを使用する場合は、Matrix のインスタンスの位置や回転、拡大などを設定した後に MC の transform.matrix プロパティへと割り当てます。

SOURCE Matrix による画像の変形

```
import flash.geom.Matrix;

var myMatrix = new Matrix();

myMatrix.scale(sx, sy); // 拡大を定義
myMatrix.rotate(rad);   // 回転を定義
myMatrix.translate(dx, dy);  // 移動を定義

myMovieClip.transform.matrix = myMatrix; //matrix を割り当て
```

Matrixの変形情報を設定するには、scale（拡大 / 縮小）、rotate（回転）、translate（移動）という３つの関数で行います。ただし、Matrix を使う場合、通常の _x、_rotation、_xscale といったプロパティによる変形とは大きく違う点が３つあります。

- 拡大 / 縮小するときは 100% の大きさを 1 として扱う
- 回転するときは、0 ～ 360 度（degree）の代わりに 0 ～ 2 π（radian）の単位を用いる
- 拡大 / 縮小、回転、移動を行う順番で、変形の内容が変化する

以下、順に見ていきます。

● 拡大 / 縮小

Matrix と通常のプロパティによる変形の違いの 1 つは、拡大 / 縮小で用いる単位の違いです。

通常の _xscale、_yscale による設定では、ご存知のとおりオリジナルの大きさを 100 として値を決定します。

SOURCE _xscale、_yscale による値指定

```
myMovieClip._xscale = 50;
myMovieClip._yscale = 30;
```

Matrix の関数 scale(sx, sy) で同様の定義を行う場合は、オリジナルの拡大率を 1 として設定します。つまり、_xscale で 50 を指定したければ、その 100 分の 1 の値、0.5 を用いることになります。

SOURCE Matrix.scale による値指定

```
var myMatrix = new Matrix();
myMatrix.scale( 0.5, 0.3);
myMovieClip.transform.matrix = myMatrix;
```

上のコードと下のコードでは、まったく同じ変形を行って比較しています。

● 回転

拡大率と同様に、回転の際も Matrix では異なる単位を使用します。通常のプロパティ _rotation では 0 ～ 360、いわゆる度数（degree）を用います。いっぽう、Matrix ではラジアン（radian）という単位を用います。ラジアンは 0 ～ 2 π という値を 1 回転とする単位です。

度数をラジアンに変換するには、

```
ラジアン＝度数＊Math.PI / 180
```

という式を用います。逆にラジアンから度数への変換は

```
度数＝ラジアン / Math.PI＊180
```

という計算から導くことができます。

以下のコードは、まったく同じ変形を行います。

SOURCE _rotation による回転

```
myMovieClip._rotation = 40;
```

SOURCE Matrix.rotation による回転

```
var myMatrix = new Matrix();
myMatrix.rotation( 40 * Math.PI / 180 );
myMovieClip.transform.matrix = myMatrix;
```

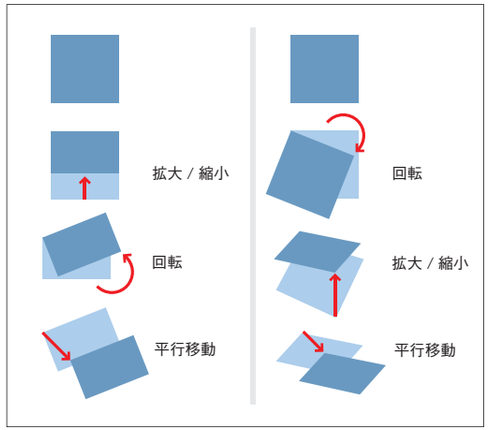

● 変形の順番

Matrix を用いた変形の最も大きな特徴は、拡大、回転、移動を行う順番で変形がまったく変わってしまうということです。Matrix で、いわゆる MC で行われる変形と同じ結果を期待したい場合は、必ず拡大 / 縮小、回転、移動という順番で行います。

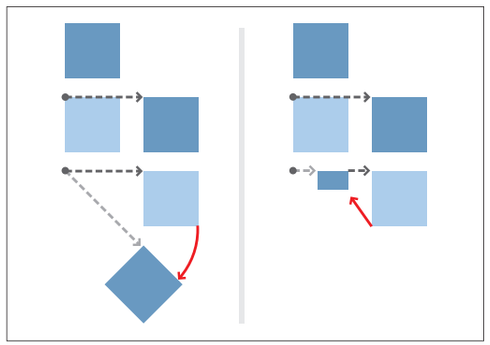
SOURCE Matrix による変形

```
import flash.geom.Matrix;

var myMatrix = new Matrix();

myMatrix.scale(sx, sy); // 拡大を定義
myMatrix.rotate(rad);   // 回転を定義
myMatrix.translate(dx, dy); // 移動を定義

myMovieClip.transform.matrix = myMatrix; //matrix を割り当て
```

Matrix の変形



左：移動後に回転、右：移動後に縮小



なぜかというと、Matrix を用いた変形は、必ずつねに原点を基準とした変形となるからです。ちょっとわかりにくいので、図を使って説明すると以下のようになります。

図「Matrix の変形」を見てください。左は推奨する順番、Y 方向に縮小→回転→移動の順で行った図です。右は、回転→ Y 方向に縮小→移動という順で変形を行ったものです。縮小と回転の順番を変えただけですが、2 つの結果は大きく違います。

なぜこのような違いがでるのでしょうか。前述したように、Matrix の変形では、つねに原点の座標が用いられます。そのため、回転後の Y 方向縮小といった例では、Y 方向の縮小がステージを基準とした方向、尺度で行われるため、ひし形に歪んでしまうわけです。

イメージ的には、「回転を確定した後に、あらためて原点を基準に拡大縮小を行った」といった感じでしょうか。

このため、移動してから回転した場合や、移動してから拡大する場合などでも、形状が歪んでしまいます。

Matrix による変形は、MC などに直接適用する場合の他、BitmapData に MC などを描画する場合の位置指定を行うために用いられます。BitmapData の関数 draw() では、MC の持つ位置や回転はすべて無視されてしまうので、変形はすべて Matrix を用いて、パラメータとして指定しなければなりません。

myMovieClip という MC をビットマップの座標（100,100）に配置する場合には、以下のように記述します。

SOURCE myMovieClip をビットマップの座標（100,100）に配置

```
var myMatrix = new Matrix(); // 描画に用いる Matrix を設定
var myMatrix.translate(100,100);
myBitmapData.draw( myMovieClip, myMatrix);
```

WindMillMatrix（とそれ以外の僕のすべてのサンプル）では、このようにして Matrix で MC を変形させながらビットマップに描画をしていきます。

○ ColorTransform による色のコントロール

ColorTransform は、Flash 8 以降で Color オブジェクトの代わりとして推奨されている新しいクラスです。ColorTransform を使用するときには、まず行頭で flash.geom.ColorTransform パッケージをインポートしてから、以下のように使用します。

SOURCE ColorTransform の使用

```
import flash.geom.ColorTransform;
var myColorTransform = new ColorTransform(rm, gm, bm, am, ro, go, bo, ao);
```

何やらパラメータが多すぎてむずかしそうに見えますが、実際は思ったよりシンプルです。8 つのパラメータはすべて、古い RGBA の値から新しい RGBA を計算するために用いられます。その肝心の計算式は

新しい Red ＝（古い Red ＊ rm）＋ ro
新しい Green ＝（古い Green ＊ gm）＋ go
新しい Blue ＝（古い Blue ＊ bm）＋ bo
新しい Alpha ＝（古い α ＊ am）＋ ao

というようになります。最初の 4 つの引数がそれぞれ元の RGBA の値と掛け算され、その後に残りの 4 つの引数が足し算されることで新しい色が計算されます。

この ColorTransform は、MC の transform.colorTransform プロパティに割り当てて直接色を変更したり、BitmapData の描画時に描画するオブジェクトの色を変更するために使用されます。

SOURCE ColorTransform で直接色を変更する

```
myMovieClip.transform.colorTransform = myColorTransform;
```


SOURCE BitmapData の描画時に色を変更する

```
myBitmapData.draw( myMovieClip, new Matrix(), myColorTransform);
```

たとえば、MC などにまったく新しい色を与えたい場合は、オリジナルの色はまったく不要ですので、以下のように掛け算に用いる最初のパラメータをすべて 0 にし、残りの 4 つで指定します。

SOURCE MC にまったく新しい色を与える

```
var myColorTransform = new ColorTransform(0, 0, 0, 0, newR, newG, newB, newA);
```

新しい Red ＝（古い Red ＊ 0）＋ newR
新しい Green ＝（古い Green ＊ 0）＋ newG
新しい Blue ＝（古い Blue ＊ 0）＋ newB
新しい Alpha ＝（古い α ＊ 0）＋ newA

あるいは、今の色の α チャンネルだけを半分にしたければ、元の RGB には 1 をかけて保存して、α チャンネルにのみ 0.5 をかけます。

SOURCE α チャンネルの値のみ半分にする

```
var myColorTransform = new ColorTransform(1, 1, 1, 0.5, 0, 0, 0, 0);
```

新しい Red ＝（古い Red ＊ 1）＋ 0
新しい Green ＝（古い Green ＊ 1）＋ 0
新しい Blue ＝（古い Blue ＊ 1）＋ 0
新しい Alpha ＝（古い α ＊ 0.5）＋ 0

ColorTransform は、MC への設定だけでなく BitmapData への描画時への色の指定でも使用されます。なぜなら BitmapData の関数 draw() では、MC に適用されている ColorTransform は無視されるので、新たに定義しなければならないからです。

以下の例は、真っ赤に塗りつぶした MyMovieClip を myBitmapData に描画します。

SOURCE 真っ赤に塗りつぶす

```
var myColorTransform(1,0,0,1,0,0,0,0);
myBitmapData.draw( myMovieClip, new Matrix(), myColorTransform);
```

いかがでしたでしょうか。このサンプルでは、コンセプトや手法の説明よりも、以後のサンプルコードで使う Flash 8 の新機能が把握できることを目的にしてみました。

以降のサンプルをとおして、新しい機能を使ってみてください。