

Covers  
Flash MX 2004

# FLASH HACKS™

*100 Industrial-Strength  
Tips & Tools*



O'REILLY®

*Sham Bhangal*

HACK  
#10

## A Custom Color Transform Class

Create a custom class to perform color transformations.

We saw in the preceding two hacks that there are [several common color transformations \[Hack #8\]](#) you may want to perform on a target clip. Furthermore, we saw that this requires a bit of housekeeping, including [setting up timers and callbacks \[Hack #9\]](#). The nature of the task suggests that it is a great candidate to be implemented as a custom class. The class can take care of all the grunt work, allowing us to perform a color transformation or fade using a few simple calls to the custom class's methods. In this hack, we implement a custom color transform class in ActionScript 2.0 using object-oriented programming (OOP) instead of the procedural timeline-based code seen in the preceding hacks.

### An Object-Oriented Transformation

Using ActionScript 2.0 requires Flash MX 2004 or Flash MX Professional 2004. You must set the ActionScript Version to ActionScript 2.0 under the Flash tab in the File → Publish Settings dialog box. Furthermore, the custom *Transform* class we'll develop must be placed in an external plain-text file named *Transform.as* (both the capitalization of the name and the *.as* extension are mandatory). You can create and edit such a file in Flash MX Professional 2004 if you select File → New → ActionScript File. If using Flash MX 2004, you'll need an [external text editor \[Hack #74\]](#). The *.as* file should be placed in the same folder as the *.fla* file that makes use of the *Transform* class. For many more details on ActionScript 2.0 classes and object-oriented development, see *Essential ActionScript 2.0* by Colin Moock (O'Reilly).

Although we can't give a full course on OOP and ActionScript 2.0 here, this custom color transform class can be used even if you don't understand OOP. And we'll examine several aspects of the code after the code listing.

Here is our object-oriented version, implemented as a custom *Transform* class, which must be stored in an external *Transform.as* file:

```
// This ActionScript 2.0 code must go in an external Transform.as file
class Transform {
    // NEG_TRANS inverts the color values.
    // NEUTRAL_TRANS resets the color values.
    // BLACK_TRANS sets the color values to black.
    // WHITE_TRANS sets the color values to white.
    // RATE sets the rate the effects will run at in ms.
    private static var NEG_TRANS:Object = {ra:-100, rb:255,
        ga:-100, gb:255, ba:-100, bb:255, aa:100, ab:0};
    private static var NEUTRAL_TRANS:Object = {ra:100, rb:0,
        ga:100, gb:0, ba:100, bb:0, aa:100, ab:0};
```

```
private static var BLACK_TRANS:Object = {ra:100, rb:-255,
    ga:100, gb:-255, ba:100, bb:-255, aa:100, ab:0};
private static var WHITE_TRANS:Object = {ra:100, rb:255,
    ga:100, gb:255, ba:100, bb:255, aa:100, ab:0};
private static var RATE:Number = 50;

private var interval:Number;
private var startTime:Number;
private var colorObj:Color;
// Constructor accepts target clip to which to apply transforms
public function Transform(targetClip:MovieClip) {
    colorObj = new Color(targetClip);
}
// Inverts the color values
public function invert(duration:Number):Void {
    applyTransform(NEG_TRANS, duration);
}

// Resets the color to the default values set in the authoring tool
public function reset(duration:Number):Void {
    applyTransform(NEUTRAL_TRANS, duration);
}

// Performs a fade to black over specified duration in ms
public function fadeToBlack(duration:Number):Void {
    applyTransform(BLACK_TRANS, duration);
}

// Performs a fade to white over specified duration in ms
public function fadeToWhite(duration:Number):Void {
    applyTransform(WHITE_TRANS, duration);
}

// Function to initiate a fade and set up an interval to
// complete it over time.
private function applyTransform(transObject:Object,
    duration:Number):Void {
    var getTrans:Object = colorObj.getTransform();
    var diffTrans:Object = new Object();
    startTime = getTimer();
    for (var i in transObject) {
        diffTrans[i] = (transObject[i] - getTrans[i]) / (duration / RATE);
    }
    // Use the form of setInterval() that invokes a method of an object,
    // so that instance properties are in scope (the object is this).
    // First parameter is the object (this) on which to invoke the
    // method specified by the second parameter (in this case
    // "transition", which must be passed as a string).
    // Third parameter is interval duration in ms.
    // Fourth, fifth, and sixth parameters get passed to transition()
    interval = setInterval(this, "transition", RATE, transObject, diffTrans,
        duration);
}
```

```
// This method applies each step of the color transformation.
private function transition(transObject:Object, diffTrans:Object,
duration:Number):Void {
    var getTrans:Object = colorObj.getTransform();
    for (var i in diffTrans) {
        getTrans[i] += diffTrans[i];
    }
    colorObj.setTransform(getTrans);
    if (getTimer() - startTime > duration) {
        // Complete the final step in the transition
        colorObj.setTransform(transObject);
        // Clear the interval to stop the effect
        clearInterval(interval);
    }
    // Force the screen to refresh between frames
    updateAfterEvent();
}

public function die():Void {
    // Perform any cleanup code here
}
}
```

There's a lot of code there, so let's take a closer look. But first, you may wonder why we didn't add the methods of our custom *Transform* class to the *MovieClip* or *Color* class. In ActionScript 1.0, doing so would have been common and even recommended. But in ActionScript 2.0, the preferred approach in most cases is to create a custom class rather than enhancing existing classes.

If you are not familiar with ActionScript 2.0 syntax, notice the `class` keyword used to define the class. Several variables are declared for the class and its instances, outside of any of the methods within the class. The *static properties* or *class properties*, defined with the keyword `static`, are defined once for the class (here, we've initialized various common transformation types and the refresh rate at which we want our transitions to work, `RATE`, is set to 50 milliseconds).

The remaining variables declared without the keyword `static` are *instance properties* (i.e., each instance of the class maintains its own value for the property). The `private` keyword identifies class properties and instance properties that are not to be accessed from outside the class. Variables declared within a method, such as `getTrans`, which is declared within `applyTransform()`, are local variables. The datatypes for all variables, properties, parameters, and method return types are specified using a colon followed by the datatype, such as `:Number`.

It is good practice to use a standardized variable-naming convention in all code, but because of the precise structuring and datotyping of OOP code, it is perhaps more important here than in other coding styles. Constant variable names (i.e., our static datatypes) have been defined in UPPERCASE, and non-constants have been defined in so-called “camelCase” (a mixture of upper- and lowercase).

Next, notice the constructor function, *Transform()*, optional in ActionScript 2.0, which is used to initialize instances of the class. Our constructor accepts the target clip to be used later by other methods of the class. The class then defines several public methods that can be invoked on instances of the class, such as *invert()* and *fadeToWhite()*, and private methods that are for internal use only.

Notice in particular the form of the *setInterval()* invocation. In this case, the first parameter passed is an object. We invoke *setInterval()* with the keyword *this*, which represents the current object (i.e., the instance of the *Transform* class on which *applyTransform()* was invoked). The second parameter is the name of the method to invoke on *this*, namely “transition” (which must be specified as a string). Thus, at the appropriate time, the *Transform.transition()* method is invoked on the current instance, *this*. Invoking a method on the current instance ensures that instance properties, such as *interval* and *colorObj*, are in scope within *transition()*. The fourth and fifth parameters passed to *setInterval()*, *diffTrans* and *duration*, are passed as parameters to *transition()*, when it is invoked. The *transition()* method performs the specified transform over the specified duration and clears the interval when complete.

To use the code, first instantiate an instance of the *Transform* class, as follows (where *myVideo\_mc* is an existing movie clip whose instance name has been set in the Properties panel):

```
var transformer:Transform = new Transform(myVideo_mc);
```

Then, invoke any methods of the class on the object:

```
transformer.invert(3000); // Invert the colors for 3 seconds
transformer.fadeToWhite(2000); // Fade to white over 2 seconds
```

To clean up when you are done, use:

```
transformer.dispose();
delete transformer;
```

Consider enhancing this custom class to implement additional features, such as:

- Making the RATE value user-definable.
- Adding controls to repeat the transitions. This would allow blinking or other repetitive effects.

- Adding more complex methods that involve more than one movie clip, allowing cross-fades between two movie clips. This would introduce some powerful effects useful with video.

The use of ActionScript 2.0 allows you to create new features that can be accessed via methods of new classes. This is the preferred way of building up libraries of commonly used features not native to ActionScript.

For many designers, the ActionScript 2.0 OOP coding style may initially seem a little long-winded because more lines of code seem to be concerned with building up the code structure than actually solving the problem at hand, especially for modest classes. This extra structure is, however, a real advantage in the long term. It gives you a structured way to make your code flexible enough to reuse in several different applications and also makes your code easier to transfer to other users (which is particularly useful in a design environment consisting of an ActionScript coder and several non-scripting designers).

Finally, although OOP code may appear longer than other styles of code, the compiled bytecode can be more efficient [Hack #100]. Flash Player 7 is optimized for OOP code. Benchmarking of well-written OOP code against procedural ActionScript shows that OOP can increase performance through the greater use of local variables and passing of data as arguments, both of which tend to create more optimized bytecode.