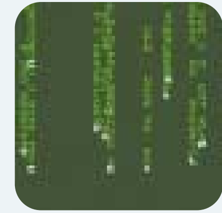
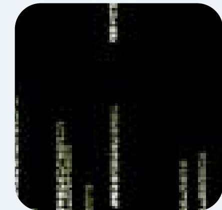
Text Effects

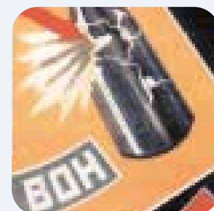
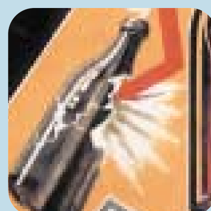
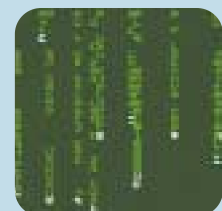
About the Effect

Text is the primary method of conveying information over the web. However, there's much more to text than that: it is a major visual tool. Text is traditionally static, but as Flash is such a great animation tool, it would be criminal not to explore the world of animated text.

This chapter will demonstrate how creating great memorable text effects can easily be achieved in Flash. You soon see that the meaning of text is often not as important as the way it is actual displayed.





The world of text

Text is important because it is a primary method of effectively communicating certain kinds of information. Even in these days of sound, animation, 3D virtual worlds, and streaming video, people like to read.

Text has not been immune to the march of progress though. We no longer use boring old plain text, because we understand that there is much more that can be communicated with this medium. Just like the spoken word has an associated non-vocal level of communication that we call body language, written text also has additional non-textual communication channels other than the actual words themselves. The choice of line spacing, text styles (bold, italic, etc.), color, and so on, can add additional levels of emphasis, importance, or simply readability. But that's only the tip of the iceberg, because the overall graphic design and layout of your text can speak volumes...

Modern graphic design arguably took off in the post-1917 Soviet Union with the Constructivist and Modernist movements. In the 1980s, armed with computers and software, designers such as Neville Brody took inspiration from the original styles to create new typographical conventions.

Compare and contrast the Soviet-Modernist typographical style with typography used in modern magazines and books, and you'll soon begin to see how influential the style actually is, and how much we now take it for granted!

For our purposes, the important idea to grasp is the way in which text can be treated as a **graphic entity**. It can form part of the final graphics rather than just exist as something that is meant to be read, and can in many cases (such as logo design) be the *only* graphic in a design.

Text in motion

Macromedia Flash is different from traditional print based graphic design because we are playing with **Motion Graphics**. As well as the rules of graphic and logo design, we can also experiment with movement. Take a look at some previous Flash-texters:

Yugo Nakamura http://surface.yugop.com/

See also http://yugop.com/ver2/ for the archive Flash site seen here:

Another good example is this text effect by the**void** (www.thevoid.co.uk), which we'll take a look at later.

For a more extreme example of the use of text on the web, go to the Designers Republic at www.thedesignersrepublic.com, where the whole design is based around text or icons that seem to be built up from text. This is an interesting crossover (and discussion point) between a print-based design and what we would normally associate with Flash graphics.

As well as what has already been created in the Flash world, one of the richest sources of animated text is **cinema**. The opening titles of many popular films contain memorable text effects, and they are a good source of text effects to use in a program like Flash.

For example, one of the most memorable effects in recent movies is the ‘text waterfall’ from *The Matrix*. Another instantly recognizable effect is the scrolling opening text at the start of each episode of the *Star Wars* series. We’ll recreate both of these effects in this chapter, and the images above are our versions.

Implementing text effects in Flash

We will be looking at a number of different ways to create text effects in Flash.

Tween based effects

Text effects are one of the easiest animations to set up if you are only using tween-based transitions. This is because the graphics are already there for you; all you have to do is select a font and prepare it for your tween. More compelling effects can be created if you mix your text animations with other effects to generate an overall mood. For example, the opening titles to David Fincher’s film *Seven* combine distressed text, hard gritty graphics, and an appropriate soundtrack to create an overall dark, foreboding atmosphere.

Scripted text transitions

Sometimes you simply want to liven up a block of text by making it appear in a novel way. Everything from ticker tape or typewriter effects, to the perspective scroll seen at the beginning of *Star Wars* can be used to liven up text that may otherwise remain unread. Such transitions are widely used in banner adverts.

Stand alone text effects

There are occasions where the text effect itself is the star of the piece. Flash toys such as text mouse followers are created for their own sake. Although such effects are sometimes ridiculed as nothing more than Flash eye-candy, they have useful applications in areas such as preloaders, where you want to maintain the user’s attention for 30 seconds while your main site loads up.

A point worth emphasizing is that the graphic components of text effects (the individual text characters that you will be animating) are very easy to create. This means that you can quickly set them up and concentrate on the mechanics of the underlying effect itself. This makes text-based animated effects a good starting point for many animations you have in mind, irrespective of whether the finished effect will involve text. Many animated effects will use text as placeholders for the real graphic elements, which you may not want to create until you have got the animation itself working.

Okay, let's get cracking! We will start gently with some general information on how Flash handles text, and then move quickly onto some typical effects themselves.

Flash and text

Flash handles fonts in three different ways and it's not always obvious which one you're forcing Flash to adopt. An appreciation of this issue is therefore crucial when creating text effects.

The following information is not directly associated with any of the text effects we will be looking at later in this chapter. However, you'll need this information to integrate them into your SWFs without adding massive font download times, or to avoid you trying to work out why your fonts aren't visible as they should be on certain machines.

The different ways of handling fonts is split into three groups: **System fonts**, **Device fonts,** and **Embedded fonts**.

System fonts

System fonts are taken from the user's machine (they are not included as part of the SWF download). There are three system fonts, called **_sans**, **_serif**, and **_typewriter**, corresponding to Times New Roman/Times (for PC/Mac systems), Arial/Helvetica, or Courier New/Courier. Flash doesn't treat system fonts in the same way as it treats vector shapes. In particular, it doesn't have access to the point and line information of each font shape, and cannot therefore rotate, resize, or mask the text. System fonts also don't appear with anti-aliased edges due to this same reason. The upside of this is that Flash can render them very quickly and, as mentioned, they don't increase the size of your SWF.

To use a system font you would do the following:

1. Using the Property inspector, select either _sans, _serif, or _typewriter in the Font drop-down menu:

2. Still in the Property inspector, hit the Character… button to bring up the **Character Options** window. Select No Characters.

   Now, here's the rub. Because you cannot treat system fonts as graphics, some text effects will not work if they rely on this feature, so be careful when using system fonts.

Device fonts

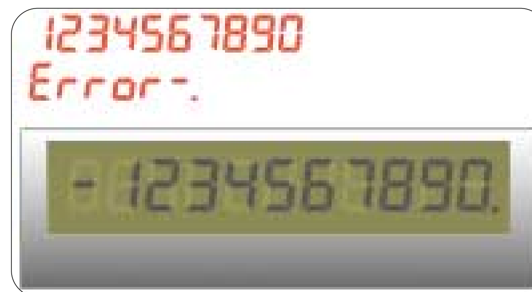
Device fonts are just like system fonts, except that if the font you have selected is not available on a particular user's machine, Flash will default to _sans, _serif, or _typewriter. The process of selecting a device font is the same as that for system fonts, but you choose any font *except* _sans, _serif, or _typewriter. For example, if I use the left-hand, non-standard font shown below as a device font, there is a very good chance that most users will not have this font installed. Although the text will appear correctly on my development machine, the text will look as shown on the right for the vast majority who don't have my eclectic taste in fonts...

Embedded fonts

Embedded fonts are, as their name suggests, embedded into the final SWF. This has the disadvantage of generating a larger SWF file for the user to download, but has the advantage that the final SWF will look the same for *all* users. It also means that Flash can treat the text as any other vector graphic, because the point and line information for the characters is saved as part of the SWF.

To embed a font, you need to bring up the Character Options window as before, but this time select either All Characters or Only. If you select All Characters, Flash will embed the entire font. For some fonts, this can easily be anything up to an additional 40-50k per font, so don't select this unless you're actually using the entire font!

If you select Only, you can select only part of the font for embedding, and this is recommended because it limits the font download to just the characters you actually need. For example, if you were building a LCD display for a simple Flash calculator, you would only need to embed the number characters, minus and decimal point symbols (so that you can show floating point and negative numbers), and the letters to write the word 'error'.



Okay, on with the effects...

Timeline effects

Fonts are essentially nothing more than filled shapes with no outline strokes, and this makes them very easy to animate. In many cases, you don't even have to use ActionScript, and can stick with nothing more complex than a few tweens, and perhaps some masking. The following examples illustrate a number of possibilities for quick no-brainer text effects.

Perspective scroll

This effect shows that motion tweens are about more than just animating motion. As well as movement, you can also add scaling and color/transparency changes, and for this effect we'll be using three out of the four (motion, color change, scaling) in a single effect.

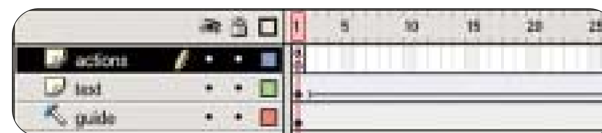
The titles to the *Star Wars* films include a perspective scroll effect. The images below are the same effect, but this time created as a very simple single Flash tween, plus some easy experimentation with the new Flash MX distort option in the Free Transform tool.

To see how we built this animation, have a look at `text01.fla`, and our version of the final SWF `starwars.swf`.

The timeline consists of three layers:

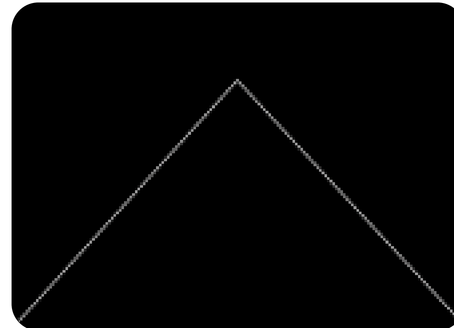
- An `actions` layer. Although this layer contains a script, it is there simply to draw our star field; it has little to do with our text effect.
- A `text` layer, which contains the perspective text tween, extending from frame 1 to 180.
- A guide layer called `guide` that contains some construction lines used in creating our effect.

Here's how it was put together, starting from scratch.



1. Create a new movie with the frame rate and stage size at the normal defaults (12 fps, 550x400 pixels – Ctrl+J will open the Document Properties). Change the background color to black and add two new layers (Insert>Layer), arranging them as in the above image. (Change the bottom layer to a guide layer from the Layer Properties dialog, which is opened from Modify>Layer…, or by choosing Guide from the drop-down menu that appears when you right-click on a layer name.)

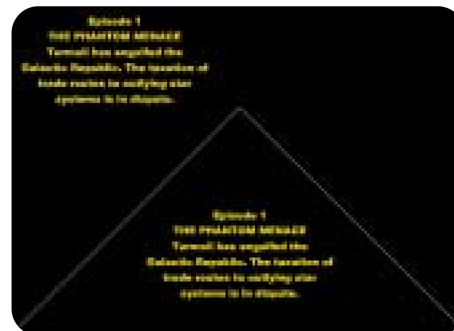
2. In the `guide` layer, add a couple of perspective lines with the Line tool (N) as shown here, starting from the two lower corner points and meeting at the top center of the stage.



3. Next, create your text within a static text field on frame 1 of `text` layer, placing it around the middle of the pyramid formed by the perspective lines. Use a system font (make sure you don't use embedded fonts, which should be the default in any case). Because the perspective distortion we will be creating may make some thinner text fonts difficult to read, you are advised to go for a bold font (we used Arial Black, a thick bold version of Arial).

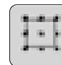
4. We will be breaking this text apart in a moment, so it would be a good idea to keep a version of this text in a safe place if we need to edit the text later. Select the text field, copy it using CTRL+C, and place the copy at the top left-hand corner of the `guide` layer.

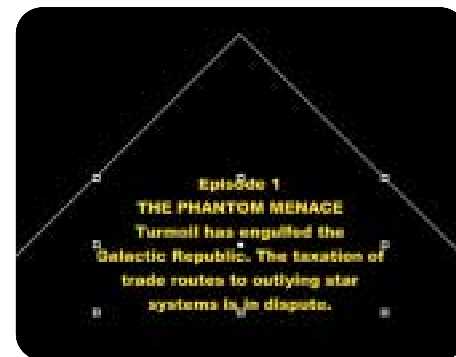
This is what you should see so far:



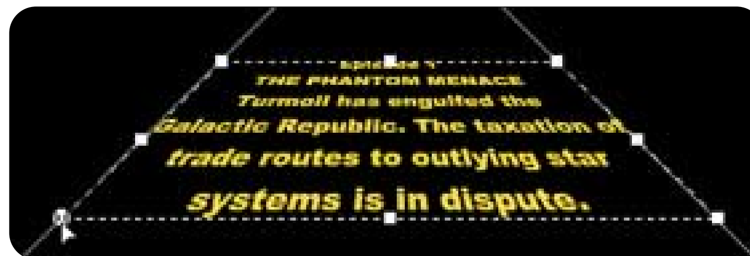
5. Lock the `guide` layer, and select the text field between the perspective lines. Move the text so that both the top corners of the field touch the perspective lines (in the next few diagrams we've removed the text at the top left to keep things simple):

6. With the text still selected, hit CTRL+B once to break it into individual letters, and then again to break it into vector shapes. Now select the **Free Transform** tool (Q) from the toolbar. You'll see a bounding box appear. The two top corner points may no longer be touching the perspective lines, so use the arrow keys to nudge the text up slightly until they are.

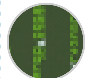




7. Now, here's the important bit. In the Options part of the toolbar, select the **Distort modifier**. Hold down the SHIFT key, and select either of the two bottom corner points of the text bounding box. Drag it onto the corresponding perspective line. The opposite corner will also move towards its perspective line, as long as the SHIFT key is pressed down.



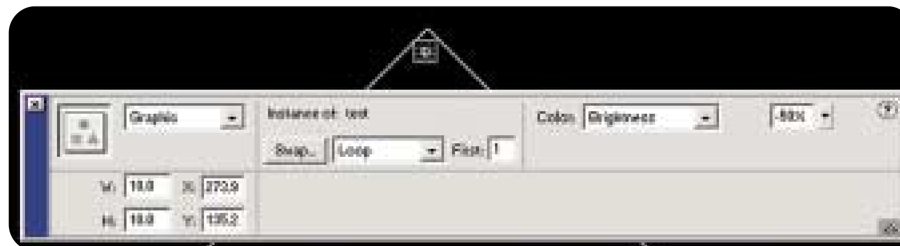
When you release the mouse, you'll end up with the perspective-distorted text shown above.



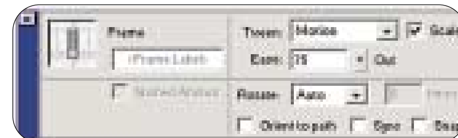
8. Hit F8 to bring up the ***Convert to Symbol*** window, and convert your distorted text into a graphic symbol called `text` (make sure the registration point is in the center of the clip). Move this new symbol off the stage directly below the perspective lines, scaling it so that it is as wide as the stage:

9. Extend the `text` and `guide` layers to frame 180 (F5), and add a keyframe at frame 180 on the `text` layer (F6). This will be the target frame for our tween. In this new keyframe, move the text field to the tip of the pyramid, and change both its height and width to 10 pixels. This will make the text get smaller as it scrolls up, getting smaller at the same rate as our two perspective lines get closer.

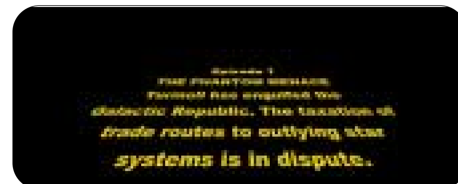
10. Finally, set the Brightness to -50% via the Color drop-down menu in the Property inspector. This will make the text fade as it moves towards our vanishing point.



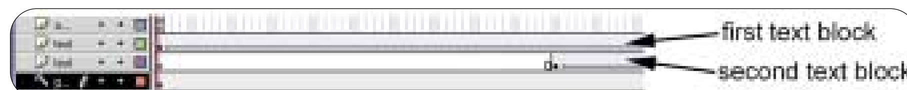
11. All we have to do now is add the tween. Select frame 1 of the `text` layer and, using the Property inspector, select Motion from the Tween drop-down menu. We want the text to slow down as it moves away, so set the Ease value to 75:



Done! Test the movie (CTRL+ENTER) to see your effect.



If you want to add more blocks of text, simply add a new `text` layer that will cause the new block of text to start moving as soon as the first one is fully on the stage. See `text01b.fla` for an example of this.



It is preferable to do this rather than make one large block of text because:

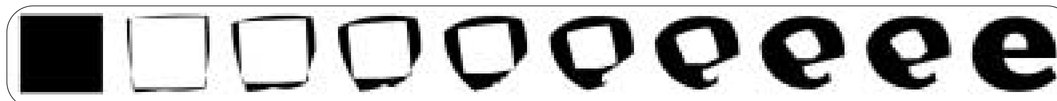
- the final movie will stream better.
- you force Flash to move fewer graphics at any one time, thereby making the animation faster.

As long as you are happy with the fact that you cannot change your text dynamically, or that the text effect cannot react too much in the way of user interaction (other than start and stop on demand), motion tween effects are easy to implement.

Sometimes, a tween-based text example requires nothing more than experimentation and a lot of simple animations. Have a look at `passages.fla` in the download files for a text-based intro that is nothing more than a few tweens with color and position transitions...

Shape tweening text

Although shape tweening of text is possible, and used widely by beginners for text transitions, it is actually very difficult to do well. This shape tween, which changes from a square to the letter 'e', shows how most shape tweens end up:

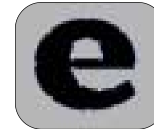


There are a few problems here.

Firstly, shape tweening doesn't like enclosed areas such as the eye of the 'e' here. This confuses the tween at the start, and it makes the square disappear at frame 2, creating a large initial transition.

Secondly, many fonts (especially free public domain fonts) may look cool, but are created without well-spaced

curve points. You'll usually find lots of odd points that don't really need to be there. This occurs because the application usually used to create such fonts (Macromedia Fontographer) has a cool feature that allows you to mix two fonts to get a third. This is a fast way to get a font like the one you are aiming for (if you want to do it quickly or cheaply), but doesn't usually result in a pretty distribution of points around the character outline.



Looking at our 'e' (break a character apart and select the shape outline with the Sub-selection tool to get the same picture), we see that there are lots of bunched-up points at the two corners of the 'e' and the right-most outer corner point:

This has the unfortunate effect of giving these three corner points more weighting in the tween than they should have, and this throws the tween.

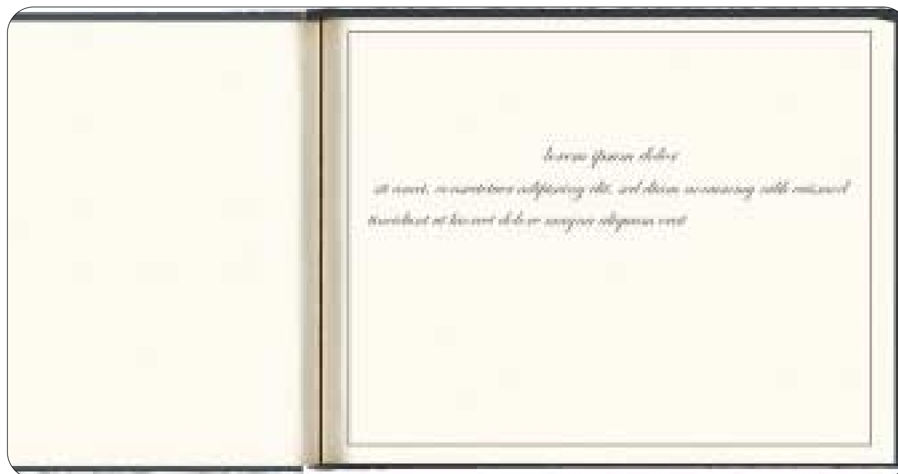
You *can* fix some of these problems by using shape hints, but you usually still end up with a mess of a tween that only starts to look like the target shape a few frames from the end (or means hours of painstaking work getting the tween right). Unfortunately, there aren't many ways around this and we wouldn't recommend using shape tweens with most text unless you are happy with the 'messy jumble that doesn't look like anything for ages' type of tween shown above.

However, here are a few tips for the brave who want to go up against us and give it a go:

- Don't use serif fonts (such as Times).
- Use simple fonts with few curves, and check that the font is well built with no bunching of points before using it.
- Don't use fonts that include enclosed spaces (which is a pretty big constraint!).
- Don't use characters that are not one continuous shape, such as i and j, unless the start of the tween is also two shapes (otherwise this throws the tween almost as much as the enclosed space problem noted above).

Timeline masking

One of the simplest masking tricks is emulating handwritten text, which we'll learn how to do in a moment. Before we do so, take a look at `text02b.swf` to see the effect in action.



Notice here how the finished effect is in a context; the opening book animation makes the handwriting effect seem more realistic. This often occurs, and many effects only really start to look cool when they have supporting animated effects around them that add realism and/or a context.

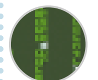
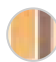
The basic animation effect is actually very easy (this exercise is `text02a.fla` in the download files).

1. Start a new movie, rename the default layer `text`, and create some text on it. In the example file, we broke the text up to make sure that even those without the font installed will see it, but to keep the bandwidth down (and retain the ability to change the text by using a dynamic text field), you may consider using embedded fonts. Our font is called Edwardian Script Italic if you want to look for it on the web.

2. Lock the current layer. Add another layer above `text`, naming it `mask`. Create a black rectangle with the Rectangle tool (no stroke) that will completely cover the text, and place it to the left of the text as shown:

3. If you are using italic text (the effect seems to work better when you do), zoom into the front edge of the rectangle and slant it as shown, so that its taper is just less than the slant of the text:

4. Select the rectangle, and hit F8 to open the Convert to Symbol window. Convert the rectangle to a graphic symbol, and call it `line mask`.

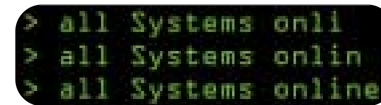



5. Add a 60-frame motion tween to the `mask` layer, with the rectangle moving right to completely cover the text at frame 60. Extend the `text` layer to frame 60 (F5):

6. Finally, turn the `mask` layer into a mask layer by right-clicking on the layer title and selecting Mask from the pop-up menu that appears.

The layers will now change as shown above, and when you test the movie, you'll see the text appear gradually as if it were being handwritten. In the fuller final version (`text02b.fla`), you can see that this basic effect is taken further by making the mask an animated movie clip.

You can also use this effect to create a typewriter effect, although you would then choose a computer-like font and make the tween much quicker (to hide the fact that it is not really typing a character at a time).

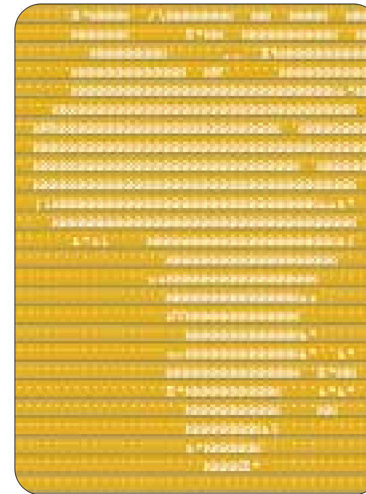


There are better ways to create such effects via scripting though, and we'll illustrate this in the next section.

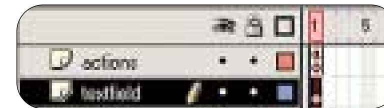
Simple scripted text effects

You can go a long way with text effects by just adding a little bit of scripting. You can add basic effects such as scrolling or a typewriter with very little code, and in a much more efficient way than tweening. This is due to the fact that these effects rely on a simple but repetitive set of instructions, something that code is very good at implementing.

Scroll and typewriter effects are a little underwhelming, unless you either use them in a novel way or embed them in a larger effect. Have a look at `main.swf`, which is part of my current site (you can see how the interface was built up in *Foundation ActionScript MX*, or simply just download the source FLA for the interface from the downloads page for this book from the friends of ED site). If you navigate to the home>futuremedia>contact page, you'll see a scrolling map:



Closer inspection reveals that it is actually some ASCII art; a world map built up from text. The map is simply a single large scrolling text field! Here's how the basic effect works.



See the example movie `text03a.fla`. We have two layers called `actions` and `textfield`. The `textfield` layer has a dynamic text field on it called `tape_txt`.

The code on the `actions` layer looks like this:

```
function tickerTape(chars) {
    displaytext = chars;
    displayLength = chars.length;
    this.onEnterFrame = tickerControl;
}
function tickerControl() {
    displayText = displayText.substr(1, displayLength-1)+displayText.sub
    ➥str(0, 1);
    tape_txt.text = displayText;
}
tickerTape("  hello world");
```

The effect will constantly scroll the text you pass to the `tickertape` function. This function stores the value of this text as `displayText`, and its length as `displayLength`. Finally, it sets up an `onEnterFrame` event as the function `tickerControl`, and it is this function that actually performs the animation. All it does is to use the `string.substr` (string substring) function to constantly take the first character in the current display and move it to the end. So, for example, if the text were 'cat dog', we would get the sequence:

```
cat dog
at dog c
t dog ca
dog cat
```

Tip: This effect works best with a proportional font, such as _typewriter.

The map effect we looked at initially actually scrolls in the other direction, and `text03b.fla` shows how to alter the code to achieve this. All that changes is our substring splicing in the first line of `tickercontrol`, which is now:

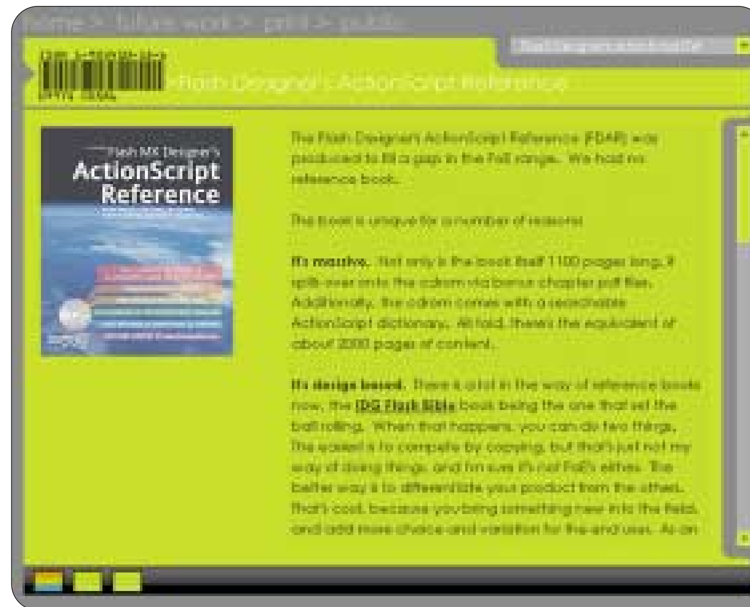
```
displayText = displayText.substr(-1, 1)+displayText.substr(0, displayLength-
➥1);
```

The only difference between this effect and the final map is that we are doing this to a text field that contains many text strings. In fact, we are using an array of strings that are fed into a loop. You can see the ASCII map if you look at the definition of this array from a distance (or, as below, view the code with a small font):

```
// define map array...
map = new Array();
map[0] = "````````````````````````````oxsi````````````````````````````````````````````````````MM````````MMMMMMMMMM````````E%``````````````````````````````````";
map[1] = "````````````````````````oxMMMM&*si``MMMMMMil``````````````````````````````=-``````E*````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMil```````````````````````````";
map[2] = "``````MMMMMMMMMMMMMMMMMMMMMMMMM&*MMMMMM``M#MM`````````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````````````";
map[3] = "````MMMMMMMMMMMMMMMMMMMMMMMMMMMM&$MM``siMM``````````````&*MM````````MMMM&*MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````````````";
map[4] = "``MMMMMM&*MMMMMMMMMMMMMMMMMMMM``````MMMM``````````````````````````MMMME%MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``&*MME*``````````````";
map[5] = "``E*````````MMMMMMMMMMMMMMMMMM``````MMMMMM````````````````````E*ij``E*MMijMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````MMMM````````````````";
map[6] = "````````````MMMMMMMMMMMMMMMMMMMM``MMMMMMMMMM``````````````````MMMM``RgMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMME*``````MM````````````````";
map[7] = "````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMvx``````````````````ozMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````````````````````";
map[8] = "````````````MMMMMMMMMMMMMMMMMMMMMMMMMM```MM`````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMF6````````````````````";
map[9] = "``````````MMMMMMMMMMMMMMMMMMMMMMMMMM&$ox````````````````````````MMMMMMME*MMMMMMozE*MMMMMM``MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM`E*```````````````````";
map[10] = "``````````MMMMMMMMMMMMMMMMMMMMMMMM``````````````````````````E*MMMM``/\\MMMMMMM``MM``MMMM``MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM```Rg````````````````````";
map[11] = "````````srMMMMMMMMMMMMMMMMMMMMMM````````````````````````````MMMMMM`````E*MM``MMMMMMMMMM``MMMMMMMMMMMMMMMMMMMMMMMMMMMMMME*``MM```MM```````````````````";
map[12] = "````````+-MMMMMMMMMMMMMMMMMMMM````````````````````````````````MMMMMMMM`````,,``E%MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM```E*E%MM```````````````````";
map[13] = "``````````MMMMMMMMMMMMMMMMMM````````````````````````````````MMMMMMMMMMMM``MW`````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM```MM````````````````````";
map[14] = "``````````ilMMMMMMMozsi``MM````````````````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMM&*MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````````````````````````";
map[15] = "``````````ilsrMMMMsr`````MM``````````````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````````````````````````";
map[16] = "``````````````MMMM````````sr````````````````````````````MWMMMMMMMMMMMMMMMMMMMMMM``MMMMMMMMMM```MMMMMMMMMMMMMMMMMMMMMMMM``ji`````````````````````````";
map[17] = "``````````````MMMM```MM```Rgsr``````````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM`````MMMMMMMM``ilMMMMMM^^``````````````````````````````";
map[18] = "````````````````MMMMMM``````````````````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMM``MMMMMM`````````MMMM`````MMMMMM``````MM````````````````````````";
map[19] = "````````````````````&*MMoz``````````````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM```````````MMMM```````MMMMMM`````@h```````````````````````";
map[20] = "```````````````````````````siMM``si````````````````````jiMMMMMMMMMMMMMMMMMMMMMMMMMMox&*`````````````MMMM`````````MMMM`````M#```````````````````````";
map[21] = "```````````````````````oz&*MMMMMMMM``````````````````````MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM``````````````@h```````MW``````````MME%`````````````````````";
map[22] = "```````````````````````````MMMMMMMMMMMM````````````````````&*si```MMMMMMMMMMMMMMMMMMM&$`````````````````````BX``MM```MME*`````````````````````````";
map[23] = "`````````````````````````MMMMMMMMMMMMMMMMM``````````````````````````MMMMMMMMMMMMMMMMMM`````````````````````````MMMM`MMMM+-`````````````````````````";
map[24] = "`````````````````````````MMMMMMMMMMMMMMMMMM````````````````````````uoMMMMMMMMMMMMMMM`````````````````````````````MM``MMMMff+-```````E%``````````````";
map[25] = "`````````````````````````MMMMMMMMMMMMMMMMMMMMMM`````````````````````MMMMMMMMMMMMMMoz`````````````````````````````MM````oiMM```E%MMMM`````````````";
map[26] = "`````````````````````````MMMMMMMMMMMMMMMMMMMMMM&*```````````````````NVMMMMMMMMMMMM`````````````````````````````````MWRg````````````MMMMMM``````````";
map[27] = "```````````````````````````MMMMMMMMMMMMMMMMMMMM```````````````````````MMMMMMMMMMMM&*``````````````````````````````````````````````````````:^````````";
map[28] = "```````````````````````````MMMMMMMMMMMMMMMMMMMM``````````````````````oxMMMMMMMMMMMM&*``&*```````````````````````````````````````MMMM``MM`````````````";
map[29] = "`````````````````````````````MMMMMMMMMMMMMMMMM#`````````````````````MMMMMMMMMMMMMM``E*MM```````````````````````````````````MMMMMMMM``MM`````````````";
map[30] = "```````````````````````````````MMMMMMMMMMMMMM&*`````````````````````E*MMMMMMMMMM```&*&*`````````````````````````````````E%MMMMMMMMMMMMM````````````";
map[31] = "```````````````````````````````MMMMMMMMMMMMMMM````````````````````````MMMMMMMMMM```MM`````````````````````````````````MMMMMMMMMMMMMMMMMM```````````";
map[32] = "```````````````````````````````MMMMMMMMMM&$```````````````````````````MMMMMMMM&$`````````````````````````````````````E%MMMMMMMMMMMMMMMMMM```````````";
map[33] = "```````````````````````````````MMMMMMMMMMvx```````````````````````````&*MMMMMMM``````````````````````````````````````E%MMMMMMMMMMMMMMMMMM```````````";
map[34] = "```````````````````````````````MMMMMMMMMM```````````````````````````````MMMME*```````````````````````````````````````ijMMMMMMMMMMMMMMMMMM```````````";
map[35] = "```````````````````````````````MMMMMMMMsr````````````````````````````````````````````````````````````````````````````MM````````MMMMMMMM`````````````";
map[36] = "```````````````````````````````MMMMMMMM``````````````````````````````````````````````````````````````````````````````````````````MMMM`````````MM```";
map[37] = "```````````````````````````````&*MMMM````````````````````````````````````````````````````````````````````````````````````````````E*``````````E*````";
map[38] = "`````````````````````````````````MMMM`````````````````````````````````````````````````````````````````````````````````````````````````````&*MM``````";
map[39] = "`````````````````````````````````MMMM```````````````````````````````````````````````````````````````````````````````````````````````````````````````";
map[40] = "`````````````````````````````````&$MM```````````````````````````````````````````````````````````````````````````````````````````````````````````````";
map[41] = "```````````````````````````````````E*```````````````````````````````````````````````````````````````````````````````````````````````````````````````";
map[42] = "ISN'T THE WORLD WIDE WEB WONDERFUL? ...";
```

Time for another one...

If you also look at the **home>future work>print>public** page in `main.swf`, you'll see a typewriter effect at work. The barcode data and title will slowly appear, as shown below right, after you select an option form the drop-down menu.



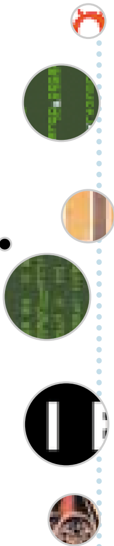
The effect works by using four separate typewriter effects running in four text fields, one each for the ISBN number, barcode (which is actually using a font that creates barcodes called *3 of 9 barcode*), barcode number, and book title. To see how the basic effect works, have a look at `text04.fla`.

We have the same timeline setup as the last example, (`text03a.fla` and `text03b.fla`). The `textfield` layer consists of a dynamic text field called `typewriter_txt`, and this is set to show multiline text.

The script on the `actions` layer looks like this:

```
function typewriter(chars) {
    this.onEnterFrame = writeText;
    this.count = 1;
    this.message = chars;
    this.maxLength = this.message.length;
}
function writeText() {
    this.typewriter_txt.text = this.message.substr(0, this.count);
    this.count++;
    if (this.count>this.maxlength) {
        this.onEnterFrame = undefined;
    }
}
typewriter("hello everybody!  \nHow are you?");
```

The \n is a control code that represents 'new line'.



The typewriter will type any string you set as the argument for the `typewriter` function. This does the following:

- Creates an `onEnterFrame` event handler, and equates it to the function `writeText`.
- Sets up a variable `count`, which is used to specify how many characters have been typed so far.
- Sets up a variable `message`, which is the text you want to type.
- Sets up a variable `maxLength`, which is the length of `message`.

The `writeText` function acts as the `onEnterFrame` event handler, and this will type one character per frame until all the characters are typed. It works like this:

- Line 1 sets the text to be displayed as what is already there as the first `count` characters. Since `count` goes up by 1 every frame (via line 2), this will look like an extra character has been added to the end of the text every frame.
- The `if` statement compares the current `count` value with the length of the text to be printed. If it is greater, then all of the text has now been printed, and the event handler undefines itself (which stops it from running again).

Notice that you cannot attach event handlers to text fields. Although the text field is a lot more like a movie clip in Flash MX, it is still not as flexible because of this, and the event handler is actually attached to the current timeline (in this case, it is the timeline pointed to be `this`, which is `_root`).

If you want to see the effect as per the *Futuremedia* web site, have a look at `typewriter.fla` in the Main folder of the download files. (Note that the movie will not render properly if you do not have the 3 of 9 barcode, OCR Extended, and Century Gothic fonts installed.) Also notice that because I have a number of different typewriters running at the same time (all of which will finish typing at different times), I have had to attach several dummy movie clips to the current timeline. I am hanging the event handlers on these for the four individual typewriter effects driving the four text fields.

```
// create four clips to hang the onEnterFrame scripts
// onto
this.createEmptyMovieClip("isbn", 1);
this.createEmptyMovieClip("barcode", 2);
this.createEmptyMovieClip("barcodeNumber", 3);
this.createEmptyMovieClip("book", 4);
```

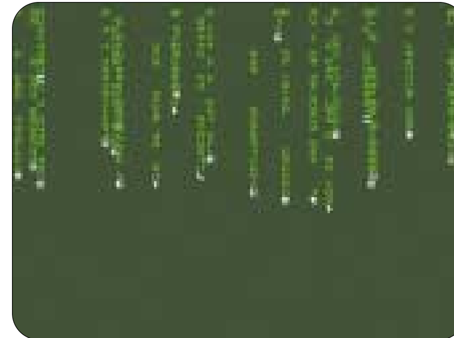
Advanced scripted text effects

You can start to create much more advanced effects once you start to use the ability to create text fields dynamically, and combine this with event driven animation. These effects can be very eye-catching, making an otherwise boring block of text move on screen in a funky way that screams "read me!" When overused though, such effects may start to look a little too much like overkill, so go easy.

Here's a smorgasbord of effects to finish with...

The Matrix waterfall effect

Have a look at `waterfall.swf`. It's the Matrix text effect. To reproduce it, you first need to know how the actual movie effect is put together. It consists of **katakana** (a Japanese text) but with a twist; the characters are all displayed backwards. A few light colored characters (which we will call the **heads**) start at the top of the screen and drop down, leaving behind a trail of green characters. These trails disappear after a while.



Okay, let's have a look at how we've done this in Flash.

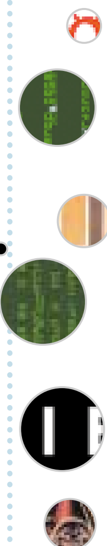
To do the 'how the hell do we display katakana backwards?' is easy in Flash; simply display katakana in a text field that has been scaled by -100% in the x direction. You can find several katakana text fonts on the web (searching Google with 'katakana ttf' is a good start).

The 'how do we make the characters drop and leave trails' deal is quite a bit harder:

- We need to make the heads drop by property-based animation, controlled by an event attached to each head.
- As they drop, we need to duplicate a series of green copies of the heads, and these will form a trail. They need to have a different event script attached to them that makes them disappear after a short time.

The text fields used in the example file (`text05.fla`) are inside a movie clip called `mc.katakana` (it's the only symbol in the Library).

Why embed it in a movie clip? Well, remember that you can't attach event scripts to text fields, and this is something we will need to do per character in our waterfall. Although there are several ways round this problem, the easiest for the busy designer who doesn't want to get bogged down with ActionScript for its own sake is the way we have chosen.



Here's the script in its entirety, from frame 1 on the main timeline:

```
function dropStart() {
     if (Math.random()>0.9) {
          this._visible = true;
          this.onEnterFrame = drop;
     }
}
function drop() {
     // create a trail character at the current position
     trailName = "trail"+trail;
     trail++;
     attachMovie("katakana", trailName, trail);
     _root[trailName]._x = this._x;
     _root[trailName]._y = this._y;
     _root[trailName].kata_txt.text = this.kata_txt.text;
     _root[trailName].fader_color = new Color(_root[trailName]);
     _root[trailName].fader_color.setRGB(0x008000);
     _root[trailName].onEnterFrame = fadeTrail;
     //
     //Move the head character of the trail down by one
     //character
     this.kata_txt.text = String.fromCharCode(Math.floor(Math.ran
     ➥dom()*26)+97);
     this._y += 10;
     // If the head character has reached the bottom, restart
     //it at the top
     if (this._y>400) {
          this._x = Math.round(Math.random()*550);
          this._y = 0;
     }
}
function fadeTrail() {
     if (Math.random()>0.95) {
          this.removeMovieClip();
     }
     if (trail>1000) {
          trail = 100;
     }
}
/*
MAIN PROGRAM INITIALIZATION
*/
trail = 100;
for (i=0; i<30; i++) {
     newName = "char"+i;
     attachMovie("katakana", newName, i);
     _root[newName]._x = Math.round(Math.random()*550);
     _root[newName]._visible = false;
     _root[newName].onEnterFrame = dropStart;
}
```

This may look a bit complicated, but it makes a lot more sense if you follow it through while looking at what waterfall.swf does when you run it.

The first thing that happens is that the section marked MAIN PROGRAM INITIALIZATION runs. This copies 30 of our movie clips to the top line of the screen, and makes them disappear by setting their _visible property to false. It also attaches the dropStart function to each as an onEnterFrame script. If you could see everything on the screen at this point, you would see something like this, with all the text sitting at the top of the screen.

dropStart doesn't actually do much. It just keeps generating a random number between 0 and 1 every frame, until it sees one that is over 0.9. As soon as that happens, the if statement in dropStart does two things:

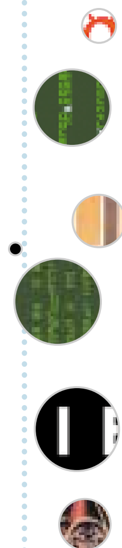
- It makes the text character visible by changing _visible to true.
- The onEnterFrame for the movie clip is changed to one that starts the animation process, drop.

This raw description does hide the purpose of dropStart somewhat, which is actually very simple: to make each trail start at a slightly different time from all the others. Think of all the characters sitting at the top rolling dice. If they hit a six, they can leave home and start on the board.

Anyway, once drop is called by the onEnterFrame, the head character starts to move down with the following lines:

```
//Move the head character of the trail down by one
//character
this.kata_txt.text = String.fromCharCode(Math.floor(Math.random()*26)+97);
this._y += 10;
// If the head character has reached the bottom, restart
//it at the top
if (this._y>400) {
     this._x = Math.round(Math.random()*550);
     this._y = 0;
}
```

This simply changes the character being displayed every frame, and also moves it down the screen. If it actually hits the bottom of the screen (_y>400), it is simply moved back up to the top of the screen and starts again.



The other thing that the head does is to create trails, the darker green characters. These are created via the following code chunk:

```
// create a trail character at the current position
trailName = "trail"+trail;
trail++;
attachMovie("katakana", trailName, trail);
_root[trailName]._x = this._x;
_root[trailName]._y = this._y;
_root[trailName].kata_txt.text = this.kata_txt.text;
_root[trailName].fader_color = new Color(_root[trailName]);
_root[trailName].fader_color.setRGB(0x008000);
_root[trailName].onEnterFrame = fadeTrail;
```

This simply attaches a new trail movie clip at the current position, and colors it a darker green than the head, using a `Color` object. The trails don't actually move, but they have an `onEnterFrame` event handler called `fadeTrail` attached to them. This is actually very similar to `dropStart`, in that it waits a random amount of time (based on how long it takes to generate a random number greater than 0.95) before making the trail disappear:

```
function fadeTrail() {
     if (Math.random()>0.95) {
          this.removeMovieClip();
     }
     if (trail>1000) {
          trail = 100;
     }
}
```

Although this function is called `fadeTrail`, it's obvious that the trail isn't really faded, it's just that pieces of it disappear over time, and this is the basis of the effect.

If the effect runs too slow for your machine, change the 30 in the for loop in the main program initialization (the code at the end of the listing).

If you've seen the film, you might be thinking 'wow, I thought the effect was more complicated than that!' Well, so did I until I actually looked at it carefully. The difference is that the film uses very small text and, from a distance, it looks as if the trail characters are fading rather than bits of the trail just simply disappearing. I was so disappointed with this revelation that I decided to change it so that the text really did fade, and you can see this in the file `waterfall2.swf`. You can also look at the code that creates this in `text05b.fla`.

Transition effects

There are a few Flash text effect generation programs out there that remove all the stress of coding up these funky text effects. Some of you may have entered this chapter with this thought at the back of your mind: 'why bother when I can just use one of those?' Well, sure, you can do that, but there is something else to bear in mind. The code to make any of them is very similar. Once you've cracked the general method, you can change it to make any number of effects, simply by playing about with the code. The beauty of this, of course, is that:

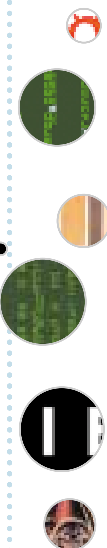
- If you write your own code you can optimize it.
- If you write your own code, it's much harder for anyone else to replicate it and make it 'So Yesterday' before today has even finished.
- If you do it yourself, you can say that 'I did it myself'.

The killer is that you already know the general code to do these effects. We used it in the Matrix waterfall. Although you might have been thinking 'well, that's nice Sham, but does it have an application?' When we talked about the waterfall, the thing is that playing around with it gave us the code-based insight to crack other problems. We learnt:

- To split the text effect into a number of text fields.
- To control each text field via a separate event handler, using an intermediate 'starter' event handler if you want to stagger the animation starting points.
- If you want to stop an animation, simply delete the event handler by making it `undefined`.

Text without hassle

Okay. Let's go with that and try to replicate one of the most famous text effects of the Golden Age of Flash, the ContentWithoutClutter text transition from thevoid's site (www.thevoid.co.uk).



When reviewing the original effect for the *New Masters of Flash* book, I remember that this effect was created using an individual tween per character of text, which meant that it took almost an hour to set up (and it was difficult to change if your text had a spelling mistake in it, or you just wanted to try another sentence!). Because of this, I set myself a limit for this example; I had to get it done in under an hour to show that writing a script can work out quicker to create, as well as being more flexible when updating the movie later on.

Have a look at the finished ActionScript-based effect `voidEffect.swf`.

Let's look at how the effect is constructed. Open up the source file `text06.fla` and open the Actions panel for frame 1 of the main timeline. The first job is placing the movie clips on the stage. This is done via the `transition` function:

```
function transition(chars, startX, startY) {
    // initialize the text format object
    mFormat = new TextFormat();
    mFormat.font = "Arial Black";
    mFormat.size = "32";
    // Assign the initial value of startX to a
    // new variable
    startText = startX;
    // Create a text field per character
    for (i=0; i<chars.length; i++) {
        // create an empty clip
        name = "char"+i;
        this.attachMovie("character", name, depthCount);
        // put it at the appropriate place
        this[name]._x = startX+startText;
        this[name]._y = startY;
        // write the character into the clip text field
        this[name].field.text = chars.substr(i, 1);
        // get this text's width and update the spacing
        mSize = mFormat.getTextExtent(this[name].field.text);
        startX += mSize.width;
        // update depth
        depthCount++;
        // set alpha to zero and resize in preparation
        // for the effect, and then add the event handler
        // to trigger the start of the animation
        this[name]._alpha = 0;
        this[name]._xscale = this[name]._yscale=1000;
        this[name].timing = (startX-startText)/10;
        this[name].onEnterFrame = starter;
    }
}
```

A subtle point to note here is that the text font used (Arial Black) is non-proportional. If I tried to space the characters using fixed character spacing, I would end up with something like this:

The i and l are much narrower than the w, and this causes problems. Instead, we have to space the text fields depending on what character we are displaying in it. The thing that comes to our rescue is the `TextFormat` object. This has a method called `getTextExtent()` that tells us how big the text field has to be in order to fully display a given text string. We can use this to give us our variable spacing depending on whether we are displaying an i or w, or any other character. This is why I have to define a `TextFormat` object at the beginning of the function, even though the text fields themselves have already been set up to display 32 point Arial Black text manually.

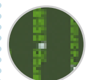
At the end of this `transition` function, the text scales up to 1000% and the `_alpha` property of all the text is set to 0%, making all the text invisible. If you could see it, the text would look like this at the end of the `transition` function:

What we have to do is make each of these oversized but invisible characters scale back to 100%, and get back to 100% `_alpha`. We also want the transition to be staggered, depending on the how far along the character is within the text, with the left-most character starting its transition first.

This is what the `starter` function does. It looks at the distance of the current character from the beginning of the sentence, and waits a short period before setting off the animation script (the `animate` function). This gives us the cool ripple effect in the final piece, where the transition moves down the text.

```
function starter() {
    // Stagger the start of the effect depending
    // on the position of this character in the text
    this.timing--;
    if (this.timing<0) {
        this.onEnterFrame = animate;
    }
}
```

Finally, the actual animation is handled by `animate`, which comes above the above function. This has to increase the `_alpha` from 0 to 100%, while at the same time decreasing the scale from 1000% to 100%. So, we have to go from 0 to 100 `_alpha` at the same time as moving down by 900 in scale. To keep things simple (and because of the fact that I had just exceeded my one hour limit when I stated writing this function) I increased the `_alpha` by 10 at the same time as decreasing the scale by 90 per frame. This means that `_alpha`, `_xscale`, and `_yscale` will get to their respective target points at the same frame, so I only have to check for any one of them.



```
function animate() {
    // Scale and fade in until the text is
    //back to its original alpha
    this._xscale -= 90;
    this._yscale -= 90;
    this._alpha += 10;
    if (this._alpha>90) {
        this.onEnterFrame = undefined;
    }
}
```

Therefore, the function only checks for `_alpha` being greater than 90 before it stops the animation for this character, by making the `onEnterFrame` undefined.

Finally, we simply call the `transition` function with our text string and the position we want it to start at. We also have to set up a variable that handles depth (the movie clips that make up our text start at a depth of 100).

```
depthCount = 100;
transition("text effects without hassle", 10, 200);
```

Looking at the code, you'll see that the function structure is *very* similar to the matrix effect, yet the two effects look completely different and actually do two completely different things. The core code for many advanced text effects is very similar in structure, and knowing this fact makes it easy for you to create your own. What are you waiting for?

Oh, and I was ten minutes out for my self-imposed one hour limit. Never mind...

No-one can hear you scream

As always, there is a reason why I showed you this last effect. There is a very important bit of code here – the ability to take a text string and:

- separate the string into individual characters.
- place each character into a separate text field, with each text field inside its own movie clip 'wrapper', allowing us to add event driven code to each one.
- space the text fields out so that they stick to the original character spacing (or kerning) of the font.

We can use this attribute to change the effect, using our core code engine. For example, we could combine the masked text effect with our core code so that the masking occurs on each individual character:

Each character appears to be behind its own shutter. Have a look at `text6b.fla`, where you'll see that each of the `mc.character` movie clips now has its own timeline-based mask added. The final effect looks almost as if each character is behind a slat, and the effect was used on a certain scary sci-fi movie some time ago...

> *Notice that we can add a timeline here, because I have not gone down the route of creating the movie clip and embedding text fields via* `createEmptyMovieClip()` *and* `createTextField()`. *Not only does my method make life easier, and allows us to stick to the effect rather than trying to be clever with the code, it also gives me a timeline to add tweens to during author time, something that cannot happen with movie clips created during runtime.*

Interactive effects

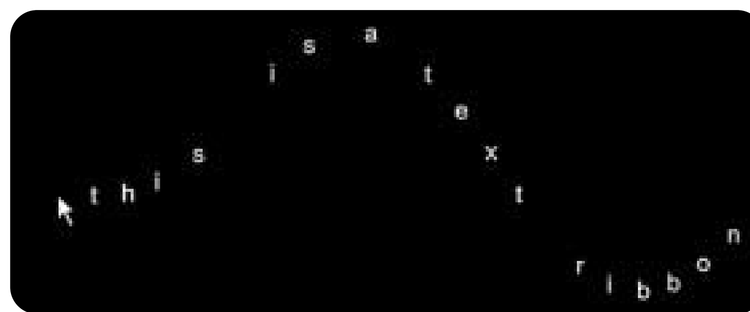
Flash is nothing if not interactive, given that its main purpose is to build mouse-controlled web site interfaces and associated content. The step from timeline-based to ActionScript-based animations may have been a big step, but the step from dynamic to interactive effects is actually a very small one.

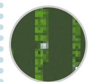
In the previous section, we created animations that:

- set up an event-driven script (the `onEnterFrame`) that animates our characters until they reach a target position (or more correctly, a *target state*, because as well as position, we can also control other properties such as size and transparency, as we did with the ContentWithoutClutter effect).
- when the target state is reached, stop animating.

The simplest interactive effects do exactly the same thing, except for two things. Firstly, the target state is based on something the user is controlling and, in many cases, this is nothing more complicated than the mouse position. Secondly, the animation is never stopped.

One of the most well-known Flash interactive text effects is the 'text ribbon' effect. This is where letters follow the mouse position as if they were printed on a ribbon that is attached to the mouse cursor. Have a look at `ribbon.swf` for the final effect and `text07.fla` for the source code.





The way this works is actually very easy:

- The first character follows the mouse.
- The second character follows the first character.
- The third character follows the second character.

In essence, each character is following the character preceding it in the sentence, with the first character following the mouse.

This code listing creates the 'one text field embedded in a movie clip' for all characters, which is very similar to the effects we've already looked at. The only thing it does differently is that in the `transition` function, it assigns a different `onEnterFrame` depending on whether the current character is the first letter (`i=0`) or not:

```
if (i == 0) {
     this[name].onEnterFrame = follower;
} else {
     this[name].onEnterFrame = mover;
     this[name].before = "char"+(i-1);
}
```

If it is the first character, we use the `follower` event script, and this simply follows the mouse:

```
function follower() {
     this._x = _xmouse+20;
     this._y = _ymouse;
}
```

Tip: You could also just make the first character follow the mouse via `startDrag()`*, but this means that it may become hidden. The* `follower` *function moves the character 20 pixels to the right to make sure this doesn't happen.*

If it is not the first character, the event script will use the `mover` function as the animation script. It also sets a variable called `before` for every character, which identifies the character previous to this one (so, for example, `char9` would have `char8` as its `before`).

`mover` is actually a simple inertia effect. It tries to get to the position of the `before` character with a slight bit of inertia to add some subtlety to the effect (set `inertia` to `1` to see it without any inertia).

```
function mover() {
     this.targetX = _root[this.before]._x;
     this.targetY = _root[this.before]._y;
     this._x -= -10+(this._x-this.targetX)/inertia;
     this._y -= (this._y-this.targetY)/inertia;
}
```

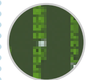
This effect is a little overdone and has been ridiculed because of it in some quarters. Surely we can find a use for it so that it becomes a little more than just eye candy? Well...

You could also use it as a Flashy alternative to help text. By changing the text on certain user events (usually rollovers), you can make the ribbon text change contextually. This could finally make useful, as well as pretty, text.

The second file text07b.fla shows this. The effect changes as soon as you click the mouse. All this does is delete all existing text if it has been used before to create a ribbon (i is not undefined). This code occurs at the top of the transition function:

```
if (i != undefined) {
    for (j=0; j<i; j++) {
        _root["char"+j].removeMovieClip();
        }
    }
```

Text effects get a little bit of stick for being just so much eye candy, but as we have seen, it's not the effect but how you use it that makes the difference!



About the Author
Adam Phillips

I was born in farming country in New South Wales, Australia, where I stayed for far too long. It wasn't until the age of 22 that I ran off to the city to join Disney. Here, I concentrated my efforts in the Effects Animation department, where I clawed my way up then decided to step down to spend some time with my new friend Flash 5. Still working for Disney, I live in Sydney with my wife-to-be Jeanette, and our cat Lucy. Here, I use all my free time to maintain www.oohbitey.com, animate, draw, to write as much as possible, and play lots of online games.

I'd like to thank a number of people who've helped and encouraged me on my way through Flash.

First, my favorite person ever: Jeanette, for liking me a lot, and for bringing me coffee and chocolate while I work.

The Phillips': mum and dad; for the stories and the years at Brewarra; my brother Brett for competitive hosting prices; my sister Juanita for being sistery; and my daughter Kaela for character inspiration.

My good friends and workmates, who either got me into Flash or helped me out in lots of ways: Bernard Derriman, Kevin Peaty, Stephen Deane, Alexs Stadermann, and Dan Forster.

My Internet friends for the encouragement and feedback: Croc, MishY, Cojack, Xaphan, Phantom, .m0rt, Fusion, and Dan Britton at friends of ED.