

Erwan Bezie's passion for graphic arts started early with illustration and black & white photography. However, he studied music and was a professional bass player for ten years, touring with jazz and pop bands around the south of France. In 1998 he bought a computer to do some proper music editing, but quickly switched back to graphics as a means of expressing himself. He decided to move to London to study web design and start a new career. After three years of working for corporate clients, he moved back to France and now runs his own web design agency, lebonze.com.

Tim Hawkins is a multimedia developer (and occasional designer) based in the UK, where people pay him to make Flash do things it probably shouldn't. He infrequently updates his website (www.cellpattern.com) with content of wildly varying usefulness, although now people like you know about it there's some chance he'll be more diligent in future. At the time of writing, Tim likes Indian food and tiny cameras, and hates French keyboards and people who steal guitars.



Video masking

lebonze with Tim Hawkins

Video on the Web undoubtedly has a nice, bright future ahead of itself, thanks to faster Internet connections every year. However, the majority of users are still on a dial-up connection, and this is where the ability to manipulate video with ActionScript comes in handy. At last, web designers have the opportunity to remain creative without forcing tens of megabytes of raw video data down a poor viewer's modem connection. Adding ActionScript to video also opens up a whole new world of interactivity. By combining code with your own creativity, your video work will become richer and more noticeable.

The French sociologist Pierre Bourdieu talks about a 'collective intellect' when he mentions the necessity of the world's thinkers to collaborate in tackling the numerous drawbacks of neo-liberalism. Why not then a 'collective intellect' for designers and artists? Admittedly, it's not neo-liberalism that's at the forefront of a designer's mind when faced with a creative task, but the point is that collaboration frequently produces better results than doing things on your own. Environment, cultural influences, gender and age difference, and even the straightforward fact of having more than one mind interpreting the subject at hand, all lead to more finely tuned and advanced results.

When we were asked to participate in this book, we decided to do a joint chapter. Each of us felt that our own skills would complement the other's, and that a collaboration would produce some interesting results.

We met about two years ago through dreamless.org – Joshua Davis' now dead-and-gone forum – when several designers from the London area decided to confront the real world. At first we met for drinks, and then collaborations started to emerge. Meeting like-minded designers in person was, for me, the most enriching of all my online experiences. Dreamless' most characteristic forum was probably '04 – battle arena + digital landfill' where, as the name suggests, designers engaged in graphical combat. Being under pressure to compete with others is a very effective way of discovering and affirming your own style. It is also a great way of assessing your weaknesses, and to begin addressing them. Indeed, anyone can do this on their own as an exercise – just choose a random topic and force yourself to complete a piece on an extremely tight deadline.

Expanding your knowledge is certainly another good source of inspiration in itself. If you haven't done it yet, you might want to learn all about trigonometry and how to use it effectively in your code. Go to your local library. The simple act of browsing through a shelf of books might well trigger a thirst to learn about a whole range of different subjects.

Thinking creatively

When I need to think creatively on a particular quandary, I find that the following exercise helps me to approach the problem from a new angle, and find an innovative solution:

I allow my mind to wander off the creative obstacle completely – in fact, the further the better. I may pick a random word in a dictionary and then find connections to the subject I'm trying to tackle. There's a good chance that the paths along which my mind meanders turn out to be new and surprising directions I'd never have considered otherwise.

Seemingly unrelated disparate material can emerge into a very coherent ensemble. It will also have a certain amount of 'freshness' not found in other creative processes. It's an exercise I would recommend to all designers faced with a creative block – improvisation is an essential source for creativity.

I also find that forcing myself to get out from in front of the computer and seeing the real world renews my innovation, enthusiasm and ideas. If you have a digital camera, get out and start snapping. Take pictures of anything – uniqueness frequently lies in the most obscure images.

I personally find that nature is probably the most inexhaustible of all sources of inspiration. Whether I'm amazed by the colors of the autumn leaves or a weed that's managed to fight its way through a concrete paving slab, whether it's the dappled colors of sunlight through a canopy of trees or the sound of a waterfall, life always offers the most incredible and moving of experiences.

Yet even if you have no shortage of inspiration, a difficult question remains: how do you assess your work's aesthetic value, or, sometimes just as importantly, its market value? What is it that gives your work this value? On the Internet, your name may constantly be competing against millions of others. Does the legitimacy of design and art work only come with popularity?

We often hear people coming out of an art exhibition saying, "I could have easily done exactly the same thing!" This might well be the case, but chances are it wouldn't have actually made it into the gallery without their name or style being previously acknowledged by those critics who define the art world. Though I may have a concept equal to the artists' work hanging in the exhibition hall, getting my piece hung in its place isn't straightforward.

So does this hold true for those of us trying to make a name for ourselves online? The anonymous nature of the Internet can certainly make it equally difficult to get recognition. Furthermore, its vastness makes it hard to know exactly how legitimized you are.

In an ideal world, artwork of any kind should not be quantifiable. The relationship between a visual piece and its viewer should be of a passionate, intense kind, its value derived not from its institutional price, but from how it moves and affects the viewer. The aim of this relationship would not be to satisfy criticism. Rather it would be a relationship where artwork imposes its own and self-sufficient rules onto a game the viewer accepts they are playing. But that's in an ideal world, right?

Well, perhaps not. The nature of the Web – forwarding amongst a peer group, 'word of mouth', linkage, specialized forums open to almost anyone with a passion of the same subject matter – allows an idea to be viewed and appreciated by millions without any kind of institutional patronage. If an idea is *good*, it will be passed on from one person to another in seconds.

Great, you may say, I want to get cracking! But there are still some practical constraints to this. You may have the creative idea of the century, but if a user has to wait ten minutes for your masterpiece to download, chances are they won't hang around to see it, and no one will pass it on. An online user, rather like a television viewer or any other person wishing to be entertained, requires almost instant results. This is where the ActionScript outlined in this chapter can help.

We hope this chapter will give you, as a reader, the **confidence** to pursue your own ideas, however random, the **inspiration** to get out there and collaborate with other designers, and the **tools** to spread your ideas easily and effectively.

The concept

You live in a personal 3D virtual reality invented by your visual cortex. The electrical signals from your two eyes are compared, filtered and processed by this area of your brain, resulting in the world you see around you. This allows you to pick matching furniture, drive fast in traffic, distinguish between your friend and an inflatable doll, and catch a Frisbee (some cortices are evidently better than others).

Compared to real life, video seems pretty easy to process. It's a two-dimensional picture projected onto a screen. So maybe if we make folks' brains work a little harder, we'll attract more viewer attention.

It's possible that by fragmenting an appropriate video clip with a moving mask, we can create the illusion of depth and motion – that the guy in the movie is walking behind the canvas of the screen. In other words, we might be able to encourage a bit of spatial awareness as well as increased object recognition (as the whole image is always partially occluded).

Or maybe not.

Worst-case scenario: we make yet another arty effect to put on a website!



Setting up

Let's start with a new FLA called `lookin.fla`. Make sure the main stage is set to 550 x 400 pixels with a frame rate of 21 fps, rename the first layer as `gradient`, and add a simple color gradient background. Lock the layer, so that none of our other steps risk interfering with its contents.

Next, create a new movie clip symbol called `myMovie`, and select the Linkage section. Check `Export for ActionScript`, and use the default identifier `myMovie`. Import your chosen video (I've used a file called `lookin.flv`, which you can find on the CD) and place it at (0,0) in the movie clip, giving the movie clip a top left-hand registration point.

We'd like to thank our friend Ivan who kindly agreed to have his face chopped up for the sake of the project!

Back in the main timeline, create a new layer called `actions` to house some ActionScript. Select the first frame of this new layer, call up the Actions panel, and get ready to start typing!

Initializing the code

First, we set two variables for the x and y coordinates of the video clip where they can be used by the rest of the code:

```
this.xPos = 100;
this.yPos = 90;
```

Now make a function we will use to start everything going:

```
this.init = function() {
```



Create an empty movie clip called `canvas`, which we'll use to draw the mask on later, in level 10, and put it in the right place on the stage, using the coordinates we stored a second ago:

```

this.createEmptyMovieclip("canvas", 10);
var c = this.canvas;
c._x = this.xPos;
c._y = this.yPos;

```

It can be useful to temporarily alias variables that will get used often, like I've done here by declaring that variable `c` is a reference to `this.canvas`. It can make code tidier to look at, and therefore a bit easier to understand.

Attach the `myMovie` movie clip:

```

this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});

```

We need it to be on level 9 so that it will be under the mask we're going to draw in our `canvas` movie clip. The placing parameters within the curly brackets make sure that `myMovie` is attached in the same position as its masking movie clip.

Now we've put the video movie clip on the stage, we can get its dimensions:

```

this.width = this.mmov._width;
this.height = this.mmov._height;

```

These will be useful later, so storing them somewhere memorable is a good plan.

To finish the initialization, we set the `canvas` as a mask of `mmov`, make an array to store our `block` objects in, and set the `processFrame` function to run every frame:

```

this.mmov.setMask(c);
this.blocks = new Array();
this.onEnterFrame = this.processFrame;
};

```

Now let's look at the functions we need to write.

The `processFrame` function

The first is the `processFrame` function we called earlier:

```

this.processFrame = function() {

```



Every time we enter a new frame, we want to do three things:

- possibly add a new block
- move and rescale all the existing blocks
- remove blocks that have moved out of the visible area, as drawing them consumes CPU power, even though they can't be seen

With just a couple of tweaks, this code can utterly swamp slower PCs and Macs. How processor-intensive it gets depends on the number of blocks on screen, the size of blocks, and the movie frame rate.

Of course, we can change the number of blocks on screen by changing how often a new block is generated and how quickly each one disappears – in other words, how fast it moves across the canvas.

First, set an alias `b` to the `blocks` array because we'll be using it loads:

```
var b = this.blocks;
```

Next, we need to set up some code to generate new blocks on some frames:

```
if (Math.random() < 0.5) {
  var nB = new Block(0, Math.random()*this.height,
    ↪ 20+(Math.random()*40), 20+(Math.random()*60));
  b.push(nB);
}
```

`Math.random` spits out a number between 0 and 1, kind of like the 'random' button on a calculator. Since it's random, it will be less than 0.5 half the time. Therefore, our `if` statement will be true 50% of the time it's called, and on average we will make one new block every two frames.

To add a block, we instantiate an object called `nB` from the `Block` class which we will define in a minute. Don't worry too much about that yet – just notice that we're varying the initial `y` coordinate, the width and the height of it with some randomization.

When it's made, we push it into the `blocks` array ready for use. Simple!

Next, we loop through the entire `blocks` array and alter each item:

```
for (var i=0; i<b.length; i++) {
  b[i].w *= 1.03;
  b[i].h *= 1.01;
  b[i].x += b[i].w * 0.3;
```

It might look good if each block grows as it gets older, so width is increased by 3% each frame, and height by 1%.

Then we increase the `x` coordinate by 30% of the current width. This means that the wider a block is, the faster it goes. So, not only do they get faster as they move across the canvas, but some are quicker to begin with – remember we randomized the initial widths when we make a new block. Why? Well, it might be quite dull if they were all the same!

This loop is a good place to remove the blocks that have gone past the edge of the video clip:

```
if (b[i].x > this.width) {
  b.splice(i, 1);
  i--;
}
}
```

When a block's x coordinate is greater than the canvas width, we delete that block from the array by splicing one value at position *i*. Since the array has just been squashed up, we reduce *i* by one so the loop still catches everything.

At the end, we fire the `drawAll` function to paint the blocks onto the canvas:

```
    this.drawAll();
};
```

The drawAll function

So far, we've been working only with pure data – an array full of `block` objects with properties representing the dimensions they should have. The `drawAll` function will be used to represent our blocks visually, using some methods of the Flash MX drawing API:

```
this.drawAll = function() {
    var c = this.canvas;
    var b = this.blocks;
    c.lineStyle();
    c.clear();
};
```

First, we set `lineStyle` to undefined – lines won't mask anything, so it's a waste of time drawing them. Then the canvas is cleared to wipe away the stuff we drew last time.

Next, we use the `drawBlock` function with the `x`, `y`, `w`, and `h` attributes of each block:

```
for (var i = 0; i < b.length; i++) {
    this.drawBlock(c, b[i].x, b[i].y, b[i].w, b[i].h);
}
};
```

The drawBlock function

This is the `drawBlock` function we just used. It takes five arguments: `cvs` is a reference to the movie clip to draw on, while the others are pretty obvious (`x` and `y` coordinates, width and height).

```
this.drawBlock = function(cvs, x, y, w, h) {
```

Our first line moves the virtual 'pen' to our starting coordinate, ready to draw:

```
    cvs.moveTo(x,y);
```

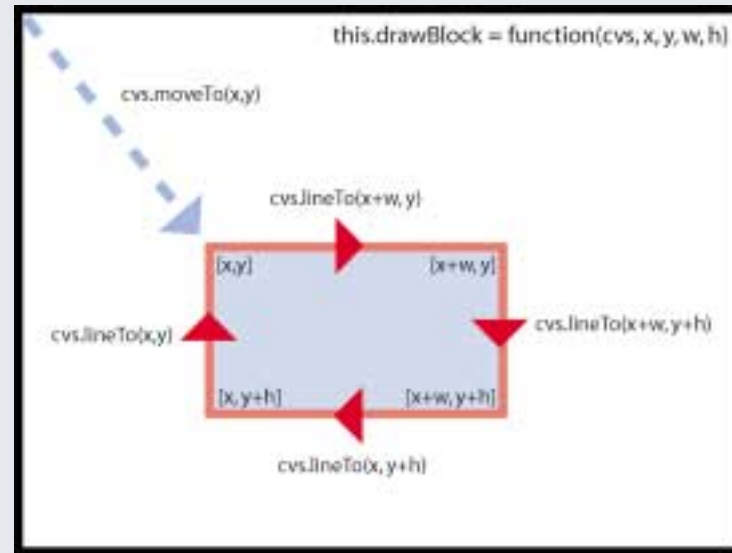
Then we say we want to start making a filled shape:

```
    cvs.beginFill(0xff0000);
```

Since it's a mask, the color is completely irrelevant – but I made it red anyway.

Then use the `lineTo` function to draw a rectangle through the specified points, and end the fill:

```
    cvs.lineTo(x+w, y);
    cvs.lineTo(x+w, y+h);
    cvs.lineTo(x, y+h);
    cvs.lineTo(x,y);
    cvs.endFill();
};
```



The block class

The user-defined `block` class we used earlier is very basic. A class is a template for an object that we can make using `ActionScript`.

Probably the most confusing thing about this is that `Flash` uses `function` to define a class, which means it's hard to tell a class from a method at first glance. Personally, I distinguish between them in my own code by writing traditional methods-that-do-things with syntax like this:

```
this.doSomething = function(arg) { trace(arg) }
```

Meanwhile, I define my classes like this:

```
function BeSomething(arg) { this.aProperty = arg }
```

There's a lot more you can do with classes – if you're interested, look for material on `OOP` in `Flash`.

So here, the `Block` class just takes four arguments and stores them as properties in the constructed object.

```
function Block(x, y, w, h) {  
  this.x = x;  
  this.y = y;  
  this.w = w;  
  this.h = h;  
}
```

Finally, we get everything running by firing the `init` function:

```
this.init();
```

When you've finished, try commenting out the `setMask` line and you'll see how the mask is drawn and how it all works.

Looking back over the script, you'll notice that the code to describe and manipulate the blocks is well separated from the code that draws them onto the screen. We could have done the same thing by attaching, moving, and removing movie clips in the `canvas` clip. However, as you'll see in some of the iterations, this way of doing things allows more flexibility when we want to change the way it looks (and it's arguably more efficient too).

Iterations

Now that we have a basic video effect in place, we're going to spend the rest of the chapter looking at some variations and iterations. Although code is shown for each one, you may want to skip straight to the FLAs on the CD and start playing around with them.

The following iterations tend to involve making changes to the initialization and modification code blocks in `processFrame`. Most of these changes are fairly self-explanatory though, so we won't spend too much time discussing them.

lookin_it1.fl



The human brain is extremely good at picking out faces with very little information. This is the reason for the man on the moon, and why cloud formations seem to have expressions. Let's alter the code so that each new `Block` is more likely to be tall and narrow (than wide and flat), so that we get vertical stripes and make the face easier to spot.

Here's how we can make that change:



```
this.processFrame = function() {
  var b = this.blocks;

  if (Math.random() < 0.5) {
    var nB = new Block(0, Math.random() * this.height,
      ↳ 5 + (Math.random() * 10), 2 + (Math.random() * 60));
    b.push(nB);
  }
  for (var i=0; i<b.length; i++) { // modification
    b[i].w *= 1.01;
    b[i].h *= 1.06;
    b[i].x += b[i].w * 0.8;
  }
}
```

```

        if (b[i].x > this.width) {
            b.splice(i, 1);
            i--;
        }
    }
    this.drawAll();
}

```

I also wanted to have the effect going both ways – the easiest way to achieve this was to put all the code into a movie clip called `effect`, and put two instances of it on the stage, in a new layer called `effects`.

Do this for yourself, and name the instances `movie1` and `movie2`. Place one at (0, 0) and the other at (470, 0). Now add this code on the main timeline:

```
movie2._xscale*=-1;
```

Multiplying the `_xscale` of a clip by -1 effectively flips it horizontally.

lookin_it2.fla



This one uses two masked videos, playing slightly out of time with each other. Again, the simplest way to achieve this was with two instances of the whole effect.

Within the initialization function, I added an extra line of code, which makes the video start playing at a random frame:

```

this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});
this.width = this.mmov._width;
this.height = this.mmov._height;
this.mmov.gotoAndPlay(Math.ceil(Math.random() * this.mmov._totalframes));
this.mmov.setMask(c);
this.blocks = new Array();
this.onEnterFrame = this.processFrame;
};

```

This means that the two instances of the `mymovie` movie clip will start at different frames, so they will not play entirely in sync.

The processFrame function has been changed again to alter the appearance of the blocks:

```

this.processFrame = function() {
    var b = this.blocks;

    if (Math.random()<0.200000) {
        var nB = new Block(0, Math.random()*this.height,
            ↪ 2+Math.random()*1, 60+Math.random()*20);
        b.push(nB);
    }

    for (var i = 0; i<b.length; i++) {
        b[i].w = b[i].w*1.300000;
        b[i].y = b[i].y*0.960000;
        b[i].x = b[i].x+b[i].w*0.200000;
        if (b[i].x>this.width) {
            b.splice(i, 1);
            i--;
        }
    }

    this.drawAll();
};

```

There's now just a one in five chance that the `if` condition will be met, so blocks will be generated less frequently.

The drawing code has been tweaked to give a slight impression of depth:

```

this.drawBlock = function(cvs, x, y, w, h) {
    cvs.moveTo(x, y);
    cvs.beginFill(16711680);
    cvs.lineTo(x+w, y);
    cvs.lineTo(x+w, y+h+10);
    cvs.lineTo(x, y+h-5);
    cvs.lineTo(x, y);
    cvs.endFill();
};

```

It's a simple example of the flexibility gained by separating the logic (keeping track of the blocks) from the presentation (drawing them out) mentioned a few pages back. All we've done is alter how the blocks are represented on screen. As far as Flash is concerned, it's working with exactly the same `Block` objects as it was in the main effect.

In fact, it's more common to find application programmers talking about presentation layers and such – but it's a handy paradigm to keep in mind even for simpler coding tasks.

lookin_it3 fla

This one uses two different videos and a white background – so it doesn't need the gradient layer we put in the original lookin fla. The building looks far away, but it's actually appearing in front of the face, creating an optical illusion.

The video lookin.flv was embedded into a movie clip called movie2, with a Linkage identifier of movie2. The new video, building.flv, was embedded into a movie clip called movie3, which I gave a Linkage identifier of movie3.

We also have two new movie clip symbols: movie2container and movie3container. These are both variations on the mymovie symbol, with very similar code in each one.

First, let's look at the `init` function for movie2container:

```

this.init = function() {
    this.createEmptyMovieClip("canvas", 10);
    var c = this.canvas;
    c._x = this.xPos;
    c._y = this.yPos;
    this.attachMovie("movie2", "mmov", 9, {_x:this.xPos, _y:this.yPos});
    this.mmov.setMask(c);
    this.blocks = new Array();
    this.onEnterFrame = this.processFrame;
};

```

And here's the equivalent for movie3container:

```

this.init = function() {
    this.createEmptyMovieClip("canvas", 10);
    var c = this.canvas;
    c._x = this.xPos;
    c._y = this.yPos;
    this.attachMovie("movie3", "mmov", 9, {_x:this.xPos, _y:this.yPos});
    this.mmov.setMask(c);
    this.blocks = new Array();
    this.onEnterFrame = this.processFrame;
};

```

The only difference here is the Linkage identifier that each one uses to attach its video-containing movie clip.

The other changes in the code from the lookin.fla ActionScript occur in the processFrame function.

For movie2container:

```

this.processFrame = function() {
    var b = this.blocks;
    if (Math.random()>0.8) {
        b.push(new block(0, 50+Math.random()*250, 10, 600));
    }
    for (var i = 0; i<b.length; i++) {
        b[i].x += b[i].w*0.3;
        b[i].w *= 1.01;
        b[i].h *= 1.01;
        if (b[i].x>300) {
            b.splice(i, 1);
            i--;
        }
    }
    this.draw();
};

```

And for movie3container:

```

this.processFrame = function() {
    var b = this.blocks;

    if (Math.random()>0.4) {
        b.push(new block(0, Math.random()*250,
            ↪      20+Math.random()*10, 20+Math.random()*10));
    }

    for (var i = 0; i<b.length; i++) {
        b[i].x += b[i].w*0.3;
        b[i].w *= 1.01;
        b[i].h *= 1.01;
        if (b[i].x>300) {
            b.splice(i, 1);
            i--;
        }
    }

    this.draw();
};

```

As you can see, the only difference between the two is in the parameters passed into the new block. This means that the blocks drawn on each container will be different shapes.

lookin_it4 fla

This is what happens if we don't always `clear` the canvas each frame before we redraw the blocks in their new position – they leave trails:



```

this.drawAll = function() {
  var c = this.canvas;
  var b = this.blocks;

  c.lineStyle();

  if (Math.random() < 0.2) {
    c.clear();
  }

  for (var i=0; i<b.length; i++) {
    this.drawBlock(c, b[i].x, b[i].y, b[i].w, b[i].h);
  }
}

```

Basically, on average the old blocks will be cleared one in every five frames.

We've actually got three instances of `effect` on the stage here, and a line of code we used in `lookin_it2 fla` is included within the `init` function to make them start playing at a random frame of the video:

```
this.mmov.gotoAndPlay(Math.ceil(Math.random() * this.mmov._totalframes));
```

We're also changing the height of each frame here, which along with the trails gives a kind of 'torn strips' look:

```

this.processFrame = function() {
  var b = this.blocks;
  if (Math.random()<0.2) {
    var nB = new Block(0, Math.random()*this.height,
      ↪      2+(Math.random()*1), 15+(Math.random()*20));
    b.push(nB);
  }
}

```

We're only drawing a new block approximately one in every five frames, as they are hanging around for longer.

Then there are a few changes to the `w`, `h`, `x` and `y` parameters:

```
for (var i=0; i<b.length; i++) {
  b[i].w *= 1.1;
  b[i].y += (Math.random() < 0.8) ? 1 : -1;
  b[i].x += b[i].w * 0.3;
  b[i].h -= 2;
  if (b[i].x > this.width + 50) {
    b.splice(i, 1);
    i--;
  }
}
this.drawAll();
}
```

The main change here is that `y` will increase by one 80% of the time, and the rest of the time it'll decrease.

lookin_it5 fla



Here, two instances of the clip are overlaid, and the top one is given an alpha value of 50%.

First, we change the `x` and `y` coordinates of the video to (0,0):

```
this.xPos = 0;
this.yPos = 0;
```

As we're going to be starting our blocks from the extreme left-hand side of the stage, we want the video to be visible from there.

Again, we add the extra line of code into the `init` function, so that each instance of `effect` starts at a random point on its respective timeline:

```
this.mmov.gotoAndPlay(Math.ceil(Math.random() * this.mmov._totalframes));
```

Then we have some changes to the `processFrame` function:

```
this.processFrame = function() {
  var b = this.blocks;
  if (Math.random() < 0.4) {
    var nB = new Block(-50, Math.random() * this.height,
      ↪ 15 + (Math.random() * 15), 15 + (Math.random() * 50));
    b.push(nB);
  }
}
```

We now have a 40% chance of a new block being generated each frame.

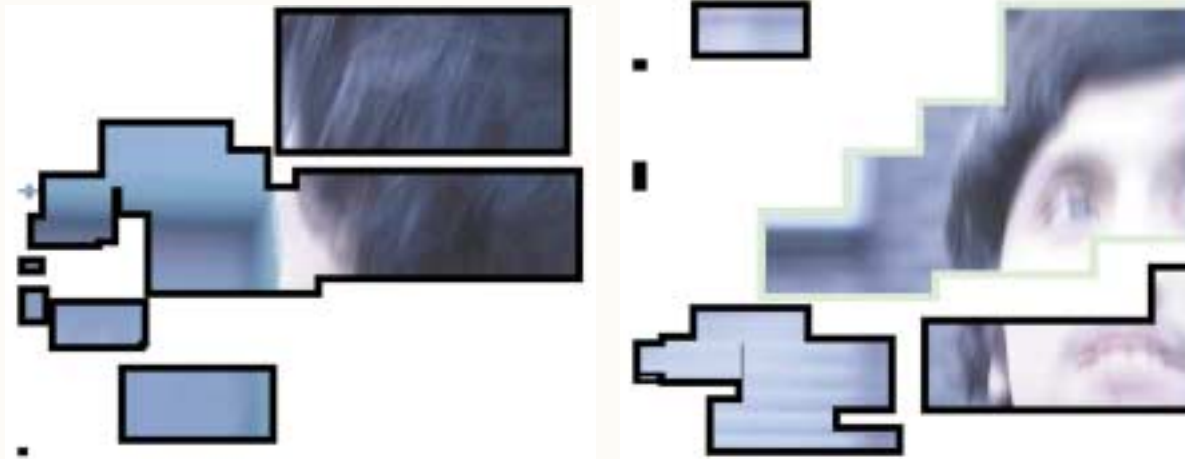
Starting the `block x` value at -50 prevents the new blocks from seeming to 'jump' into existence on the left-hand side, and contributes to a smoother progression.

The `h` property of each block is drastically decreased in each frame, so it quickly attains a negative value. The block is still drawn without any problems though, and we get the effect of blocks enlarging upwards rather than downwards (remember that Flash's `y` axis is, for some reason, reversed).

Again, there are some changes to the values of the `block` parameters:

```
for (var i=0; i<b.length; i++) {
  b[i].w *= 1.1;
  b[i].y += (Math.random() < 0.2) ? 1 : -1;
  b[i].x += b[i].w * 0.15;
  b[i].h -= 10;
  if (b[i].x > this.width + 50) {
    b.splice(i, 1);
    i--;
  }
}
this.drawAll();
};
```

lookin_it6.fla



In this version, what appear to be outlines are really just the same blocks drawn slightly bigger on a secondary background canvas (bg). This is a useful trick in many situations, especially for making shadows and outlines for groups of objects without getting in between the objects themselves. (An offset and setting `bg._alpha` to a lower value would create a drop shadow.)

First make some alterations to the `drawAll` function:

```

this.drawAll = function () {
    var c = this.canvas;
    var bg = this.bg;
    var b = this.blocks;
    c.clear();
    bg.clear();
    for (var i=0; i < b.length; i++) {
        this.drawBlock(c, b[i].x, b[i].y, b[i].w, b[i].h);
        this.drawBlock(bg, b[i].x-10, b[i].y+10, b[i].w+20, b[i].h-20);
    }
};

```

As you can see, we just re-use the `drawBlock` function – this time asking it to use the `bg` movie clip as a canvas and draw to coordinates 10 pixels outside those it's drawing to in the mask.

Because now it looks messy where we draw outside the borders of the movie clip (it doesn't matter when it's a mask as it isn't visible), we apply yet another mask to the background layer when we initialize the clip:

```

this.init = function () {
    this.createEmptyMovieClip("canvas", 10);
    this.createEmptyMovieClip("bg", 7);
    this.createEmptyMovieClip("bgMask", 8);
    var c = this.canvas;
    this.bg._x = this.bgMask._x = c._x = this.xPos;
    this.bg._y = this.bgMask._y = c._y = this.yPos;
    this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});
    this.width = this.mmov._width;
    this.height = this.mmov._height;
    this.mmov.setMask(c);
    this.drawBlock(this.bgMask, this.xPos - 10, this.yPos - 10,
        this.width + 20, this.height + 20);
    this.bg.setMask(this.bgMask);
    this.blocks = new Array();
    this.onEnterFrame = this.processFrame;
};

```

We use the `drawBlock` function to draw a simple block, just larger than the stage, on the `bgmask` clip.

If you comment out the last two new lines, where you set the mask, you'll see why this extra mask is necessary. Once the black blocks are not hidden by the video, you can see they are solid shapes, and the illusion of outlined blocks of video is lost.

And once again, we've tweaked the block parameters inside the `processFrame` function:

```

this.processFrame = function() {
  var b = this.blocks;
  if (Math.random() < 0.400000) {
    var nB = new Block(-50, Math.random()*this.height,
      ↪      15+Math.random()*15, 15+Math.random()*50);
    b.push(nB);
  }
  for (var i = 0; i < b.length; i++) {
    b[i].w = b[i].w*1.100000;
    b[i].y = b[i].y+3;
    b[i].x = b[i].x+b[i].w*0.100000;
    b[i].h = b[i].h-6;
    if (b[i].x > this.width+50) {
      b.splice(i, 1);
      i--;
    }
  }
  this.drawAll();
};

```

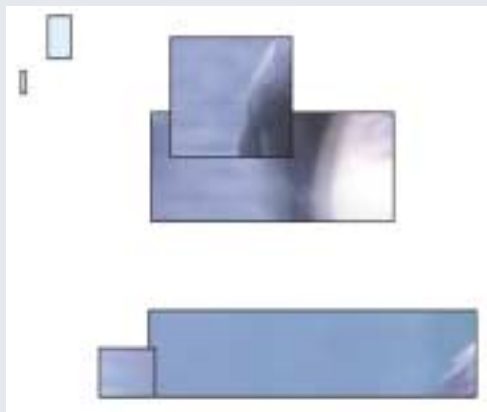
Oh, one final thing to remember. When we originally wrote the `drawBlock` function, it didn't matter what color we drew in, as the drawing would be used as a mask. But now we want to see the blocks, so unless you want your outlines to be red (which I think changes the mood slightly) amend the `beginFill` method within the `drawBlock` function:

```

this.drawBlock = function(cvs, x, y, w, h) {
  cvs.moveTo(x, y);
  cvs.beginFill(0);

```

lookin_it7 fla



In this version, based on `lookin_it6 fla`, the borders are slimmed down to two pixels and three instances of the effect are overlaid, with the line of code added to randomize the frame that the video starts playing in each of them, so they play out of sync.

To achieve the almost insectoid block-lurching movement, x motion for each frame is partly based on the `sin` of the y position (which itself is somewhat randomized). Block height is increased in a similar way, but to a lesser extent.

First, the changes to the init function:

```

this.init = function() {
  this.createEmptyMovieClip("canvas", 10);
  this.createEmptyMovieClip("bg", 7);
  this.createEmptyMovieClip("bgMask", 8);
  var c = this.canvas;
  this.bg._x = this.bgMask._x=c._x=this.xPos;
  this.bg._y = this.bgMask._y=c._y=this.yPos;
  this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});
  this.width = this.mmov._width;
  this.height = this.mmov._height;
  this.mmov.gotoAndPlay(Math.ceil(Math.random() * this.mmov._totalframes));
  this.mmov.setMask(c);
  this.drawBlock(this.bgMask, this.xPos-2, this.yPos-2,
    ↳ this.width+4, this.height+4);
  this.bg.setMask(this.bgMask);
  this.blocks = new Array();
  this.onEnterFrame = this.processFrame;
};

```

Next, the usual tweaks to the block shape:

```

this.processFrame = function () {
  var b = this.blocks;
  if (Math.random() < 0.06) {
    var nB = new Block(-15, Math.random() * this.height,
      ↳ 1 + Math.random() * 15, 1 + Math.random() * 10);
    b.push(nB);
  }
  for (var i=0; i<b.length; i++) {
    b[i].w *= 1.1;
    b[i].y -= 6 * Math.random();
    b[i].x += 6+ Math.sin(b[i].y *0.1) * 6;
    b[i].h += 4+ Math.sin(b[i].y *0.1) * 2;
    if (b[i].x > this.width + 2) {
      b.splice(i, 1);
      i--;
    }
  }
  this.drawAll();
}

```

The parameters passed to the drawBlock function for bg are changed so that the blocks drawn here will only be slightly bigger than those drawn on the canvas movie clip:

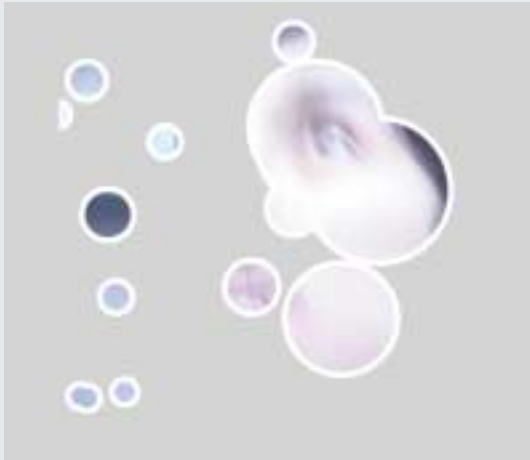
```

for (var i = 0; i<b.length; i++) {
  this.drawBlock(c, b[i].x, b[i].y, b[i].w, b[i].h);
  this.drawBlock(bg, b[i].x-2, b[i].y-2, b[i].w+4, b[i].h+4);
}
};

```

lookin_it8.fla

In this iteration, based on the last one, we only have one instance of the effect on the stage. By replacing the `drawBlock` function with a `drawCircle` one (in this case, one swiped from Casper Shuirink and modified slightly – thanks!) and doing a couple of other tweaks, we have a stylish bubble effect.



First, let's change the parameters passed to `drawBlock` within the `init` function to make the outlines look a bit more substantial:



```

this.init = function() {
    this.createEmptyMovieClip("canvas", 10);
    this.createEmptyMovieClip("bg", 7);
    this.createEmptyMovieClip("bgMask", 8);
    var c = this.canvas;
    this.bg._x = this.bgMask._x=c._x=this.xPos;
    this.bg._y = this.bgMask._y=c._y=this.yPos;
    this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});
    this.width = this.mmov._width;
    this.height = this.mmov._height;
    this.mmov.gotoAndPlay(Math.ceil(Math.random() * this.mmov._totalframes));
    this.mmov.setMask(c);
    this.drawBlock(this.bgMask, this.xPos-5, this.yPos-5,
        ↳ this.width+10, this.height+10);
    this.bg.setMask(this.bgMask);
    this.blocks = new Array();
    this.onEnterFrame = this.processFrame;
};

```

Now some alterations to the `processFrame` function:

```

this.processFrame = function() {
    var b = this.blocks;
    if (Math.random()<0.14) {
        var nB = new Block(-15, Math.random()*this.height,
            ↳ 15+Math.random()*25, 1+Math.random()*10);
        b.push(nB);
    }
}

```

```

for (var i = 0; i<b.length; i++) {
  b[i].w *= 1.02;
  b[i].y -= 2*Math.random();
  b[i].x += 5+Math.sin(b[i].y*0.1)*5;
  if (b[i].x-b[i].w/2>this.width) {
    b.splice(i, 1);
    i--;
  }
}
this.drawAll();
};

```

Now let's look at the drawAll function:

```

this.drawAll = function () {
  var c = this.canvas;
  var bg = this.bg;
  var b = this.blocks;
  c.strokeStyle();
  c.clear();
  bg.clear();
  for (var i = 0; i < b.length; i++) {
    this.drawCircle(c, b[i].x, b[i].y, b[i].w*0.5);
    this.drawCircle(bg, b[i].x, b[i].y, b[i].w*0.5 + 8);
  }
};

```

The same block-moving code as before is used – we're just ignoring the h property of each and passing the w property divided by two when the drawCircle function needs a radius parameter.

Even though we are now going to draw circles rather than blocks, don't forget that the drawBlock function is still needed for the large mask.

And now for the last, vital extra function: drawCircle.

```

this.drawCircle = function(cvs, x, y, r){
  cvs.moveTo(x+r, y);
  cvs.beginPath(0xfffff);
  cvs.curveTo(r+x, -0.4142*r+y, 0.7071*r+x, -0.7071*r+y);
  cvs.curveTo(0.4142*r+x, -r+y, x, -r+y);
  cvs.curveTo(-0.4142*r+x, -r+y, -0.7071*r+x, -0.7071*r+y);
  cvs.curveTo(-r+x, -0.4142*r+y, -r+x, y);
  cvs.curveTo(-r+x, 0.4142*r+y, -0.7071*r+x, 0.7071*r+y);
  cvs.curveTo(-0.4142*r+x, r+y, x, r+y);
  cvs.curveTo(0.4142*r+x, r+y, 0.7071*r+x, 0.7071*r+y);
  cvs.curveTo(r+x, 0.4142*r+y, r+x, y);
  cvs.endFill();
} // (Casper Schuirink)

```

Don't worry too much about this. While there are much more elegant-looking ways of drawing a circle, sometimes the performance gains of hard-coding numbers are a good enough excuse for ugly code!

lookin_it9 fla

A bit of a departure... Since we already had circles happening, here's a strange rippling distortion effect made with only a few modifications to the code from lookin_it8 fla.



We need to make some changes to the initialization function:

```

this.init = function() {
  this.createEmptyMovieClip("canvas", 10);
  this.createEmptyMovieClip("bg", 7);
  this.createEmptyMovieClip("bgMask", 8);
  var c = this.canvas;
  this.bg._x = this.bgMask._x=c._x=this.xPos;
  this.bg._y = this.bgMask._y=c._y=this.yPos;
  this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});
  this.attachMovie("myMovie", "bgmov", 1, {_x:this.xPos, _y:this.yPos});
  this.attachMovie("myMovie", "mmov2", 2, {_x:this.xPos, _y:this.yPos});
  this.width = this.mmov._width;
  this.height = this.mmov._height;
  this.mmov.setMask(c);
  this.mmov2.setMask(bg);
  this.mmov.gotoAndPlay(4);
  this.mmov2.gotoAndPlay(2);
  this.blocks = new Array();
  this.onEnterFrame = this.processFrame;
};

```



We're overlaying three movies here from within the ActionScript - one remains unmasked (bgmov), and the other two (mmov and mmov2) are masked by canvas (as usual) and bg, which is where our circles were being drawn anyway.

mmov and mmov2 are started on frames 2 and 4 respectively, and these slight offsets from bgmov are what make the visual effect work.



lookin_it10 fla

Once again, we're building on the last example here, `lookin_it9 fla`. Although you can't tell from the image, in this version the blocks move down the screen now instead of from left to right.

We've made `mmov` and `mmov2` 50% transparent, so they blend into the background more – and seem to reach full opacity where they overlap. We've also used the `Color` object to transform the background movie into a pale blue.

```

this.init = function() {
  this.createEmptyMovieClip("canvas", 10);
  this.createEmptyMovieClip("bg", 7);
  this.createEmptyMovieClip("bgMask", 8);
  var c = this.canvas;
  this.bg._x = this.bgMask._x=c._x=this.xPos;
  this.bg._y = this.bgMask._y=c._y=this.yPos;

  this.attachMovie("myMovie", "mmov", 9,
    ↳ {_x:this.xPos, _y:this.yPos, _alpha:50});
  this.attachMovie("myMovie", "bgmov", 1,
    ↳ {_x:this.xPos, _y:this.yPos, _alpha:50});
  this.attachMovie("myMovie", "mmov2", 2,
    ↳ {_x:this.xPos, _y:this.yPos, _alpha:50});
  (new Color(this.bgmov)).setTransform({ra:'70', rb:'20', ga:'30', gb:'150',
    ↳ ba:'20', bb:'200', aa:'10', ab:'100'});

  this.width = this.mmov._width;
  this.height = this.mmov._height;
  this.mmov.setMask(c);
  this.mmov2.setMask(bg);
  this.mmov.gotoAndPlay(4);
  this.mmov2.gotoAndPlay(2);
  this.blocks = new Array();
  this.onEnterFrame = this.processFrame;
};

```

Next, some changes to the `processFrame` function:

```

this.processFrame = function() {
  var b = this.blocks;
  if (Math.random()<0.09) {
    var nB = new Block(Math.random()*this.width, -50,
      ↳ 15+Math.random()*25, 20+Math.random()*10);
    b.push(nB);
  }
  for (var i = 0; i<b.length; i++) {
    b[i].w *= 1.001;
    b[i].x += 2*Math.random();
    b[i].y += 4;
    b[i].h *= 1.05;
  }
};

```



```

    if (b[i].y>this.height) {
        b.splice(i, 1);
        i--;
    }
}
this.drawAll();
};

```

We start the blocks at a random position along the x axis and 50 pixels above the video - the reverse of what we've been doing before.

Then in each frame, *y* is incremented by 4, so blocks move down the screen at a constant rate.

Also, obviously, we've switched back to blocks again, because those circles were just way too exciting:

```

this.drawAll = function() {
    var c = this.canvas;
    var bg = this.bg;
    var b = this.blocks;
    c.lineStyle();
    c.clear();
    bg.clear();
    for (var i = 0; i<b.length; i++) {
        this.drawBlock(c, b[i].x, b[i].y, b[i].w, b[i].h);
        this.drawBlock(bg, b[i].x-10, b[i].y-40, b[i].w+20, b[i].h+20);
    }
};

```

lookin_it11 fla



Finally, we've a bit of a bizarre variation, continuing from `lookin_it10 fla`. In the `init` function, we got rid of the background and made `mmov` fully opaque again:

```

this.init = function() {
    this.createEmptyMovieClip("canvas", 10);
    this.createEmptyMovieClip("bg", 7);
};

```

```

var c = this.canvas;
this.bg._x = c._x=this.xPos;
this.bg._y = c._y=this.yPos;
this.attachMovie("myMovie", "mmov", 9, {_x:this.xPos, _y:this.yPos});
this.attachMovie("myMovie", "mmov2", 2,
    ↳ {_x:this.xPos, _y:this.yPos, _alpha:50});
(new Color(this.bgmov)).setTransform({ra:'70', rb:'20', ga:'30', gb:'50',
    ↳ ba:'20', bb:'20', aa:'10', ab:'100'});
this.width = this.mmov._width;
this.height = this.mmov._height;
this.mmov.setMask(c);
this.mmov2.setMask(bg);
this.mmov.gotoAndPlay(4);
this.mmov2.gotoAndPlay(2);
this.blocks = new Array();
this.count = 0;
this.onEnterFrame = this.processFrame;
};

```

Now some changes to the processFrame function:

```

this.processFrame = function() {
    var b = this.blocks;
    if (Math.random()<0.15) {
        var nB = new Block(Math.random()*this.width, -50,
            ↳ 25+Math.random()*50, 30+Math.random()*50);
        b.push(nB);
    }
    for (var i = 0; i<b.length; i++) {
        b[i].x += 5+10*Math.sin(i+this.count*0.12);
        b[i].y += 11+12*Math.cos(i+this.count*0.4);
        if (b[i].y>this.height+100) {
            b.splice(i, 1);
            i--;
        }
    }
    this.count++;
    this.drawAll();
};

```

We increased the block-generation frequency and made the blocks bigger, because we also removed the code that makes them grow over time.

As well as that, we made use of the `Math.sin` and `Math.cos` functions with a counter variable to make the blocks move around in erratic little ellipses on their way from the top of the screen to the bottom.

The only change in `drawAll` is to make the blocks masking `mmov2` a bit bigger:

```

for (var i = 0; i<b.length; i++) {
    this.drawBlock(c, b[i].x, b[i].y, b[i].w, b[i].h);
    this.drawBlock(bg, b[i].x-10, b[i].y-10, b[i].w+20, b[i].h+50);
}
};

```

Now, the change to the `drawBlock` function is where it gets interesting:

```

this.drawBlock = function(cvs, x, y, w, h) {
  var m = x+(w*0.5);
  var n = y+(h*0.5);
  var z = n*Math.sin(this.count*0.1);
  x *= 0.5;
  y *= Math.sin((this.count+x)*0.03);
  cvs.moveTo(x, y);
  cvs.beginFill(0xffffffff);
  cvs.curveTo(m+z, n+z, x+w, y);
  cvs.lineTo(x+w, y+h);
  cvs.curveTo(m-z, n-z, x, y+h);
  cvs.lineTo(x, y);
  cvs.endFill();
};

```

We create three new variables from the standard ones that get fed into it.

`m` is halfway across the `x` axis, `n` is halfway down the `y` axis, and `z` is a pretty meaningless number based on `n` and a sine wave. We also change the values of `x` and `y`, which really distorts the output.

Then we replace the top and bottom `lineTo` commands with `curveTo`, using combinations of our new variables as control points. Now the shapes seem to bounce around somewhat.

I think the most important thing to take away from all these iterations is a sense of how easily a piece of code can be tweaked to produce quite different results – and maybe how spending a bit more time fiddling can yield something you wouldn't have thought of to begin with. So, if you haven't started playing with these FLAs already, open them up and get tweaking.



