Jimmy the body worker

What's in this chapter:

- Introducing Jimmy.
- Drawing dynamic graphics.
- Understanding color mixing.
- Using the Color object to solidly fill.
- Using the Color object and transform to tint and shade.
- Using the Color object to fill and shade components of movie clips.
- The Paint-O-Rama 3000.
- Drawing shapes dynamically.
- Understanding the methods of the drawing API.
- Understanding `curveTo`.

Introducing Jimmy

There was a man named Jimmy. He dreamt of one day becoming a great painter, roaming the banks of the Seine, creating great masterpieces as he went. Then, one day, someone reminded him that it was 2002, not 1802, and that the days of the classic painter were all but gone.

So Jimmy headed off in search of a new, contemporary means of expression. It was then that he came across Rufus, the owner of a state-of-the-art automotive garage. Rufus was sitting on a wooden box outside his garage, looking sad and dejected. Jimmy, being a concerned individual, asked Rufus what was wrong. Rufus explained that his recent auto-body artist had been tragically crushed under a stack of 10,000 duplicate wrenches.

Feeling sad, while at the same time recognizing a terrific opportunity, Jimmy explained that he knew a thing or two about art, and could most definitely perform the duties of an auto-body artist.

Encouraged, Rufus said, “Really? Step into my garage and show me what you can do!”

Dynamic graphics

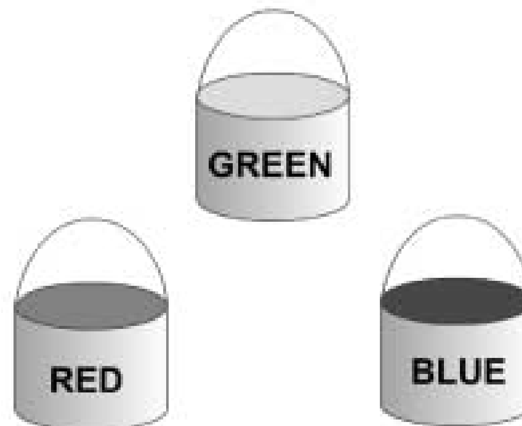
In Flash MX, we can draw and create simple and complex images using the draw tools and the color palette. However, what happens if we want to create images, shapes and colors ‘on the fly’, with ActionScript, instead of pre-drawing them?

That’s where Jimmy comes in.

Jimmy is the master of all things dynamically drawn and colored. Before we get into the specifics of the code, we must first make sure we understand how ‘paint’ is mixed in this garage.

Color mixing

Every color, shade, scale and tint of paint used in this garage is derived by mixing exact amounts of the three ‘primary’ paint colors; red, green and blue.



Most artists know that the primary colors of paint are red, yellow and blue, but in this garage, things are a little different. Because the garage exists in the Flash MX digital world, our colors are based on the mixing colors of light, not paint.

When it's time for Jimmy to order up a color from the paint machine, he must specify what color he wants by specifying the mix of the primary red, green and blue. All the colors in the visible spectrum are available to him using this means. Each primary color ranges from 0 to 255, meaning there are 256 possible values for each primary color. This means that, when they're combined, there is a palette of 16,777,216 possible colors (256 x 256 x 256)

Now, there's one catch: colors must be specified as a neatly combined number, which is the result of the red, green, and blue amounts combined. Something like this:

```
0xRRGGBB
```

What's that? Well, Jimmy walks up to the paint machine, and keys in the eight-digit paint code, starting with "0x" and then followed by the double-digit amounts of R, G and B.

So, if each primary color goes from 0 to 255, then how come there are only two digits specified for each color? Well, that's because our colors must be specified in the 16-based 'hexadecimal' numbering system, where numbers are counted not from 0 to 9 but from 0 to F. This means that where our largest 10-based double-digit number is 99, our largest double-digit hexadecimal number is FF.

Counting from 00 to FF takes some understanding of hexadecimal, and that can take some time to get your head around. Just remember that when counting up from 0, do it like this:

```
00, 01, 02, 03, 04, 05, 06, 07, 08, 09,
0A, 0B, 0C, 0D, 0E, 0F
```

Then, we have:

```
10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
1A, 1B, 1C, 1D, 1E, 1F
```

This pattern will repeat until the last 16, which are:

```
F0, F1, F2, F3, F4, F5, F6, F7, F8, F9,
FA, FB, FC, FD, FE, FF
```

So, in the end, Jimmy will have the following for black:

```
0x000000
```

for white:

```
0xFFFFFF
```

for red:

```
0xFF0000
```

for green:

```
0x00FF00
```

for blue:

```
0x0000FF
```

for yellow:

```
0xFFFF00
```

for cyan (light blue):

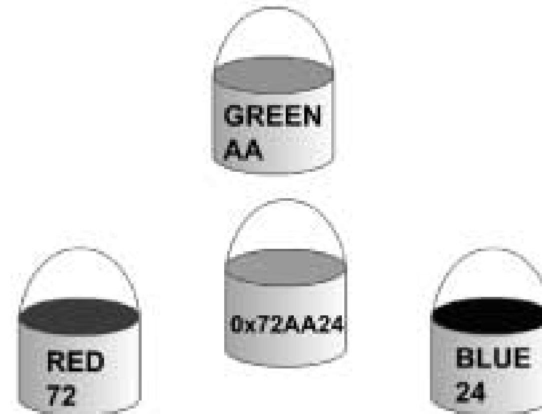
```
0x00FFFF
```

for magenta (purple):

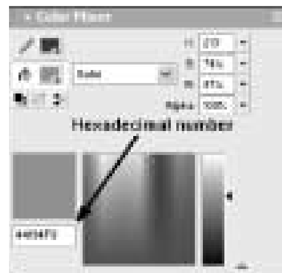
```
0xFF00FF
```

Every other color is an intricate combination of hexadecimal values. For example, a medium olive green is:

```
0x72AA24
```



The best way to figure out these colors is to use the color mixer in the Flash MX design environment to pick your color, and then write down the hexadecimal number that appears in the lower-left hand corner of the window.



Flash will display it with a '#' at the beginning of it, but in ActionScript we must ignore the '#' and instead tack on a '0x' to the beginning. The difference between these two is simple: The '0x' prefix has been used for years in programming languages and is a programmers way of indicating a hexadecimal number, while the # has been used by web designers in HTML to indicate hexidecimally formatted colors. So, for us, the '0x' prefix tells Flash that the number immediately following is to be treated as a hexadecimal number, and not just a normal decimal number. For example, the following ActionScript:

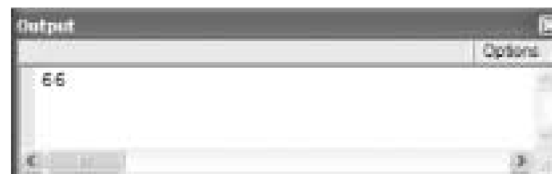
```
a = 42;
trace (a);
```

will produce the following output:

However, the following ActionScript:

```
a = 0x42;
trace (a);
```

will produce the following output:



Notice that Flash traces out the decimal equivalent of the hexadecimal number, '0x42'. As mentioned before, using this combination there are 16,777,216 possible colors using 2-digit hexadecimal numbers. To prove this, look at the output of this code:

```
a = 0xFFFFFF;
trace (a);
```

The highest color, white, corresponds to 0xFFFFFF or 16777215 in decimal. (The number is not 1677721**6**, because that is the *total* number of colors, but they *span* from 0 to 16777215, which totals 1677721**6**).

Jimmy's brushes

We've talked about the paint; we've talked about Jimmy, what's in between? The brushes, of course. Whenever Jimmy wants to paint something, he needs a brush. If Jimmy plans on painting more than one object, he needs more than one brush, because once a brush is dedicated to a particular object, it may not be reused.

What are we talking about here? We're talking about the Color object in Flash MX. The Color object is used to change the RGB values of any movie clip. The Color object is used by first "constructing" it, during which time you specify the movie clip to which it will apply.

Creating or constructing a Color object works with the following ActionScript code:

```
myColor = new Color(myMovieClip);
```

In this code, 'myColor' will be the name of the color object itself, and 'myMovieClip' would be the name of the movie clip to which color effects, changes and transformations will be applied. It could be thought of like this:

```
sixInchBrush1 = new Color(theHood);
```

where we're creating a new brush, and defining its application to be 'painting the hood of the car'. In Flash MX, we can also specify a 'timeline', instead of a movie clip. For example, we could say:

```
myColor = new Color(_level5);
```

or:

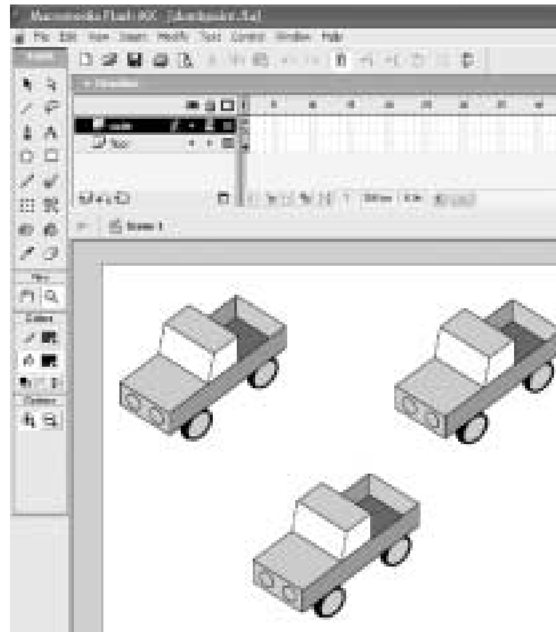
```
myColor = new Color(_root);
```

which would cause Flash to color all objects (graphics, buttons, movie clips) within a particular timeline.

Ok, let's give Jimmy some work - let's see this in action.

Dummy paint

It's Jimmy's first day, so his methods are somewhat crude as he learns the machinery and brushes. Consequently his paint job is a bit ... solid. What do we mean by that? Well, take a look at `dumbpaint.fla`.

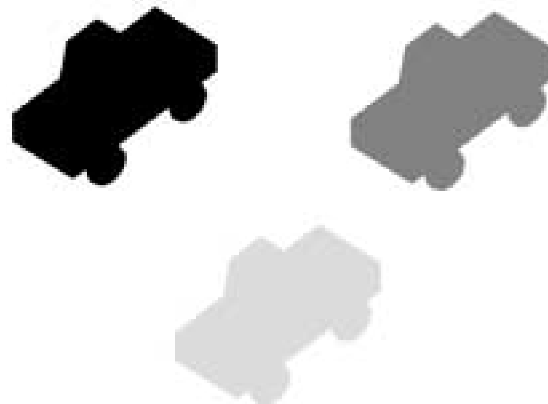


We have three trucks on the shop floor. Jimmy has been given the task of painting one black, one red, and the last one yellow. The trucks have the instance names `truck1`, `truck2` and `truck3`. (As a reminder, the instance name of any movie clip is set in the Property inspector, which is activated by clicking on the movie clip, and pressing CTRL/⌘+F3.) Here's the code by which Jimmy will do his work (attached to frame 1 of the code layer):

```
c1 = new Color(truck1);
c2 = new Color(truck2);
c3 = new Color(truck3);

c1.setRGB(0x000000);
c2.setRGB(0xFF0000);
c3.setRGB(0xFFFF00);
```

First, we're creating three brushes, `c1`, `c2` and `c3`. Notice that each brush is being attached to its own consecutive truck. Once these brushes are created, `c1` will be used to paint `truck1`, `c2` will be used to paint `truck2` and `c3` will be used to paint `truck3`.

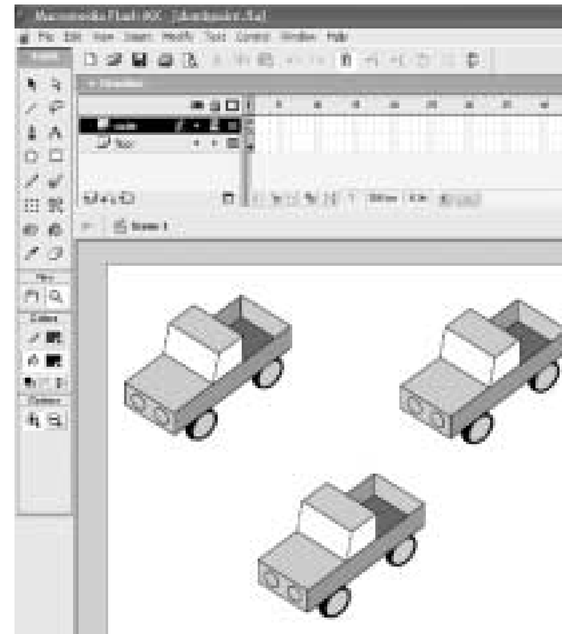
Next, we use the `setRGB` method of the Color object. The `setRGB` method does one simple thing: it turns all solid areas in a movie clip into the color specified by `setRGB`. The effect? Let's put Jimmy to work by pressing CTRL/⌘+ENTER:



Those are some well-painted vehicles. One is solid black, the other is solid red, and the last one is solid yellow. However, the art of auto-body painting requires a greater attention to detail. And, we want our trucks to look a little less like solid cutout shapes, and more like painted trucks.

Smarty paints

Let's look at `smartypaint.fla`. From the outset, `smartypaint.fla` looks like `dumbpaint.fla`; three trucks on the stage, with actions on frame 1 of the code layer.



However, this time the actions are substantially different. Rather than use the `setRGB` method of the `Color` object, we're going to be using the `setTransform` method. The difference between `setRGB` and `setTransform` is simple: `setRGB` completely changes all solid areas to a specific color, while `setTransform` *modifies* the color of all solid areas. Essentially, `setTransform` will take a movie clip and increase or decrease its RGB values so that we're still able to see its details and lines, but on the whole it will change color.

The `setTransform` method works like this:

```
myColor.setTransform(transformObject);
```

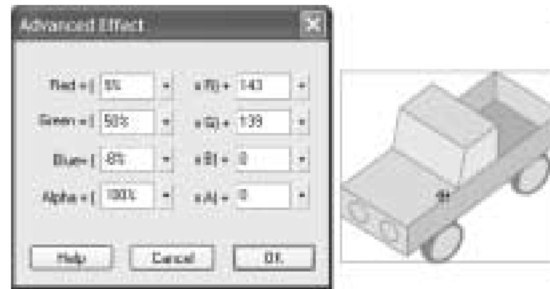
Notice the '`transformObject`'? That's not a color. That's an object, which contains several values that will be used to modify, or "transform" the colors of a movie clip. The transform object is created like so:

```
transObj = new Object();

transObj.ra = 100;
transObj.ga = 100;
transObj.ba = 100;
transObj.aa = 100;
```

What's that? Those are percentage values – percentages of each color to maintain in the transformed movie clip.

1. Before we go on, let's try something. Click on a movie clip (any one of them will do) and open up the properties panel with CTRL/⌘+F3. At the far right portion of the panel, there's a drop-down menu next to the word 'Color'. From this menu, select Advanced, like so:

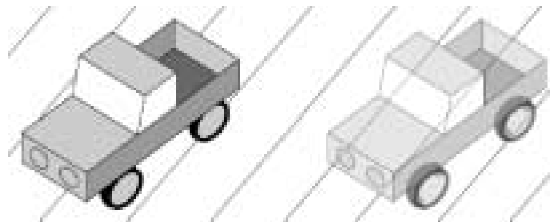
2. A button that is labeled 'Settings...' will appear. Click on this button now. We will then be presented with the 'Advanced Effect' box:



3. In the left column, we can see that there is a column of Red, Green, Blue and Alpha percentages. In the right column, we can see that there is a column of Red, Green, Blue and Alpha offsets. The percentages work by scaling the RGB/A values of any movie clip by a certain percentage, while the offsets work by merely *adding to* or *subtracting from* the RGB/A values of any movie clip.

This dialog box has exactly the same effect as `setTransform`, only `setTransform` does it with ActionScript. For example, if `ra` was set to 100, but `ga` and `ba` were set to 0, then we'd be saying 'keep all red values in the movie clip while discarding green and blue'. Since every color is made up of a combination of red, green and blue, the effect this would have would be to tint the entire movie clip into nice shades of red. It is also possible to set any of these values to a number greater than 100, which will have the effect of brightening up a particular R, G or B component of the colors. This is just like our left column of the Advanced Effect box.

The '`aa`' value is to specify the amount of alpha to maintain. The alpha value refers to the semitransparency of the movie clip. Look at this:



The left truck has an alpha value of 100%, while the right truck has an alpha value of 50%.

Lets look at the code attached to frame 1 of the code layer:

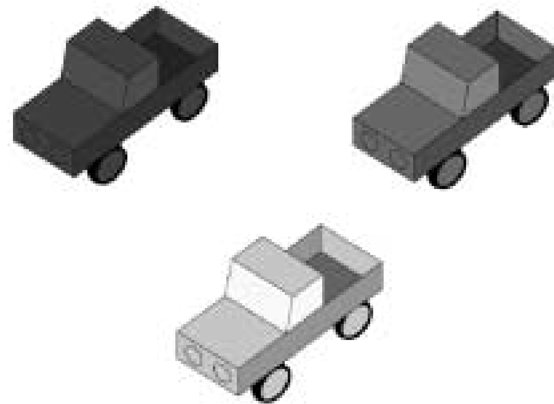
```
c1 = new Color(truck1);
c1obj = new Object();
c1obj.ra = 30;
c1obj.ga = 30;
c1obj.ba = 30;
c1.setTransform(c1obj);

c2 = new Color(truck2);
c2obj = new Object();
c2obj.ra = 100;
c2obj.ga = 0;
c2obj.ba = 0;
c2.setTransform(c2obj);
```

```
c3 = new Color(truck3);
c3obj = new Object();
c3obj.ra = 100;
c3obj.ga = 100;
c3obj.ba = 0;
c3.setTransform(c3obj);
```

We're creating three Color objects, each of which are attached to a truck. However, since we're using the `setTransform` method, this time we're creating three generic objects: `c1obj`, `c2obj` and `c3obj`.

In the first example, we're setting the `ra`, `rg` and `rb` values of the transform to 30, 30 and 30. This will be our 'black' paint. We're equally setting all red, green and blue values to 30%. The second transform is using 100 for ra, but 0 and 0 for `ga` and `ba`. This will cause all green and blue to be removed from our truck, and only shades of red will remain. The third transform retains 100% of the red and green components but discards the blue component. The effect is coloring the image yellow.

Since the original truck movie clip is grayscale, coloring it will make the truck become evenly painted using the transform object. Think of this as like a light spray paint, rather than the thick solid color applied by something like `setRGB`. Let's try running this movie with CTRL/⌘+ENTER:



Now that looks a bit nicer! Those trucks have been painted with the artistic skill of Cézanne himself.

Now, though we're not going to use it here, we can also use the color offset value (the right hand column in the advanced color box) in the `setTransform` function. The only difference is in our object, we specify `rb`, `gb`, `bb` and `ab`, like so:

```
c1 = new Color(truck1);
c1obj = new Object();
c1obj.rb = -40;
c1obj.gb = 20;
c1obj.bb = 0;
c1obj.ab = 0;
c1.setTransform(c1obj);
```

In this example, the value 40 would be subtracted from any red component of the colors in the movie clip, and the value 20 would be added to any green component. Blue would be unaffected, because since

bb is 0 then nothing is being added to or subtracted from the blue value. Alpha would also be unaffected.

From now on however, we're going to stick to using the percentages (ra, ga, ba, aa) with setTransform because they have the clearest and most intuitive affect on the colors. The offsets can get a bit "strange" looking at times, and consequently, can become confusing.

Now, there's one more level we can take this to. Let's ask Jimmy to paint the entire vehicle in an assorted array of colors.

Component paint

Let's look at our truck as something broken down into these eight basic component movie clips:

bedfloor – The floor of the bed of the truck.

bedinside – The inside walls of the bed of the truck.

grill – The front grill of the truck.

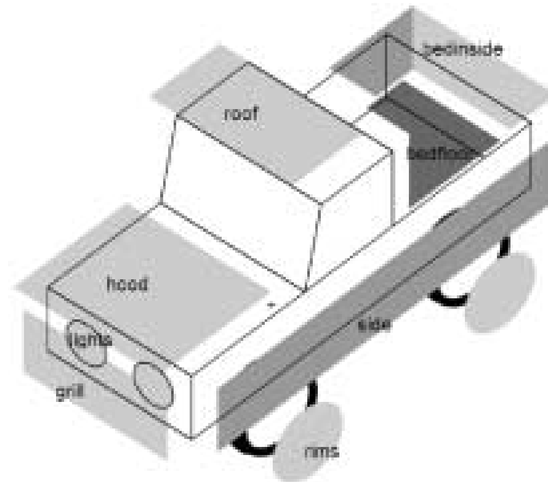
hood – The hood.

lights – The headlights of the truck.

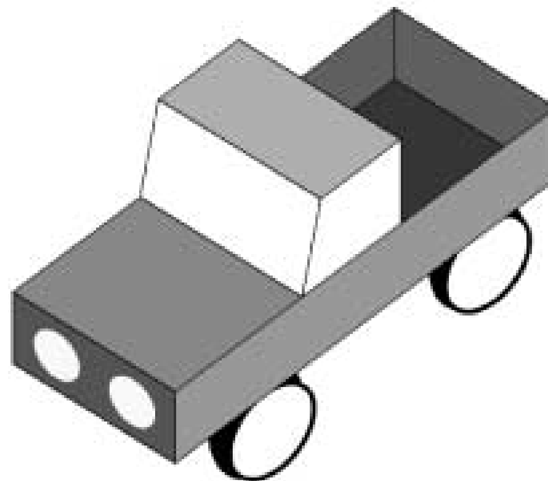
rims – The rims of the wheels.

roof – The roof above the cab.

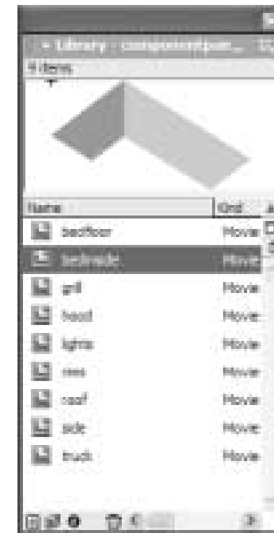
side – The side running the length of the truck.



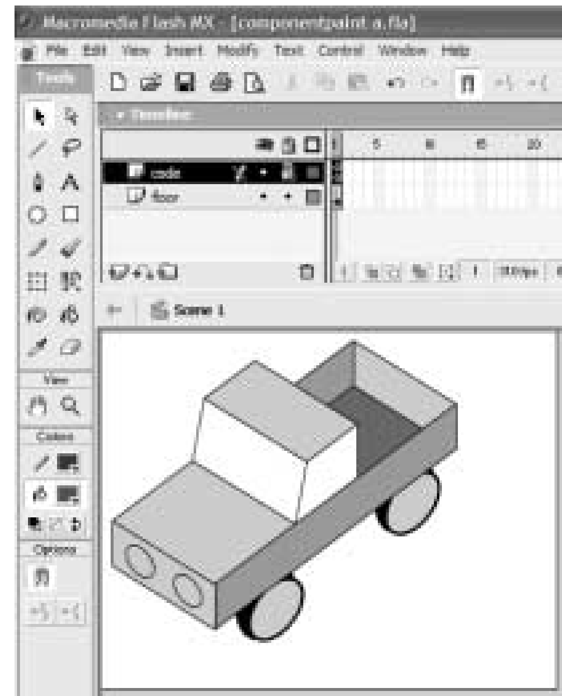
Since we must create one Color object per area colored, we must create eight Color objects. There are two ways to approach this. First, we could use the setRGB method to solidly paint each of the truck's components, and create something like this:



The bodywork of this particular truck was ordered specially by a circus. This is found in the movie `componentpaint-a.fla`. When loading this, look at the Library:



Now, loading the movie up, we can see the stage looks like this:

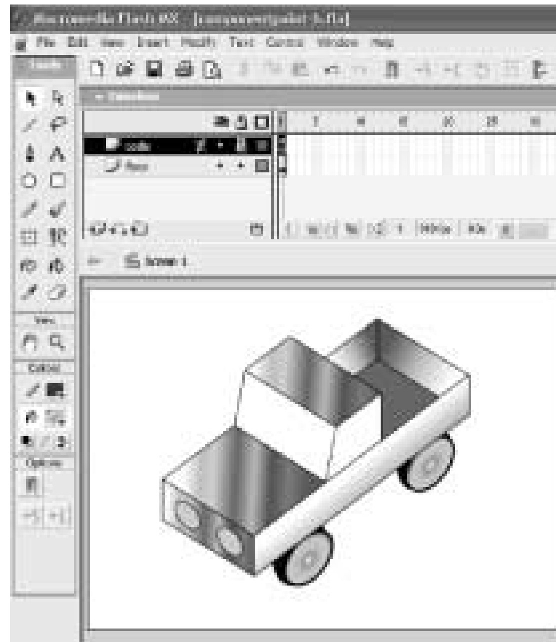


There's one truck on the shop floor, and it has an instance name of simply, 'truck'. All of the other components are contained within it, so that they would be referred to as `truck.hood`, `truck.roof`, `truck.rims`, etc. In the case of our multicolored truck, the code attached to frame 1 of the code layer looks like this:

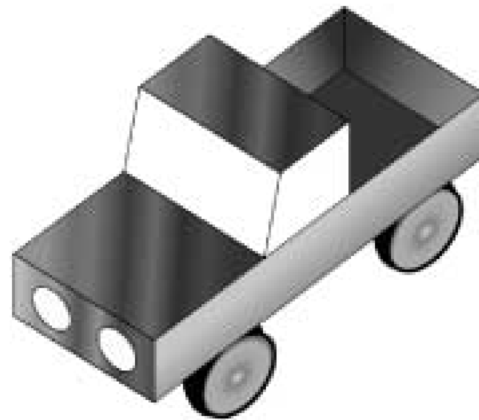
```
cbedfloor = new Color(truck.bedfloor);
cbedinside = new
Color(truck.bedinside);
cgrill = new Color(truck.grill);
chood = new Color(truck.hood);
clights = new Color(truck.lights);
crims = new Color(truck.rims);
croof = new Color(truck.roof);
cside = new Color(truck.side);

cbedfloor.setRGB(0x4F2700);
cbedinside.setRGB(0x944901);
cgrill.setRGB(0x0033FF);
chood.setRGB(0xFF3300);
clights.setRGB(0xFFFF33);
crims.setRGB(0xFFFFFF);
croof.setRGB(0x00CC00);
cside.setRGB(0xFF6600);
```

We're creating eight Color objects, one for each component, and we're pointing them at their respective truck body part. Then, we're simply using the `setRGB` method of each Color object and coloring each body part individually. The colors chosen are deliberately disparate to illustrate the effect.

Now, the second method we can use to color our truck is to use the `setTransform` method. Look at the movie `componentpaint-b.fla`. Notice that our truck has some more detail now:



Each of the components has been painted to include the use of gradients. This is where the advantage of `setTransform` becomes apparent. We're going to let Jimmy paint each part of the truck, but because we're using `setTransform`, the shading of the gradient underneath will be preserved. While `setRGB` will completely paint our truck in a strange sort of lifeless solid color, the `setTransform` method will allow us to do this:



The vehicle is straight out of the 60s, and Jimmy's artistic vision has been realized. Of course, the code to perform this (attached to frame 1 of the code layer) is quite straightforward. Our Color objects are created in the same way as the previous example. However, rather than use `setRGB`, we use the `setTransform` methods, like so:

```
cbedfloort = new Object();
cbedfloort.ra = 100;
cbedfloort.ga = 0;
cbedfloort.ba = 0;
cbedfloor.setTransform(cbedfloort);

cbedinsidet = new Object();
cbedinsidet.ra = 100;
cbedinsidet.ga = 0;
cbedinsidet.ba = 50;
cbedinside.setTransform(cbedinsidet);

cgrillt = new Object();
cgrillt.ra = 100;
cgrillt.ga = 60;
cgrillt.ba = 150;
cgrill.setTransform(cgrillt);
```

```
choodt = new Object();
choodt.ra = 100;
choodt.ga = 0;
choodt.ba = 100;
chood.setTransform(choodt);

clightst = new Object();
clightst.ra = 200;
clightst.ga = 200;
clightst.ba = 100;
clights.setTransform(clightst);

crimst = new Object();
crimst.ra = 10;
crimst.ga = 100;
crimst.ba = 0;
crims.setTransform(crimst);

crooft = new Object();
crooft.ra = 100;
crooft.ga = 0;
crooft.ba = 60;
croof.setTransform(crooft);

csidet = new Object();
csidet.ra = 20;
csidet.ga = 90;
csidet.ba = 30;
cside.setTransform(csidet);
```

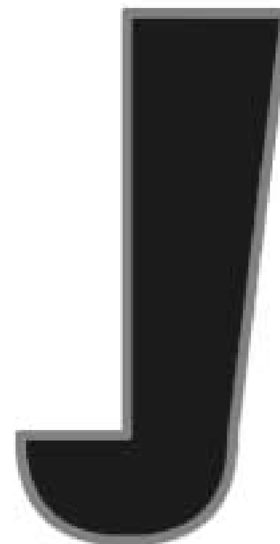
Each body part has its own Color and `transform` object, and they are being applied to it using the `setTransform` method.

Armed with these powerful paint tools, Jimmy is able to create some truly stunning and somewhat rare color combinations on the trucks in the garage.

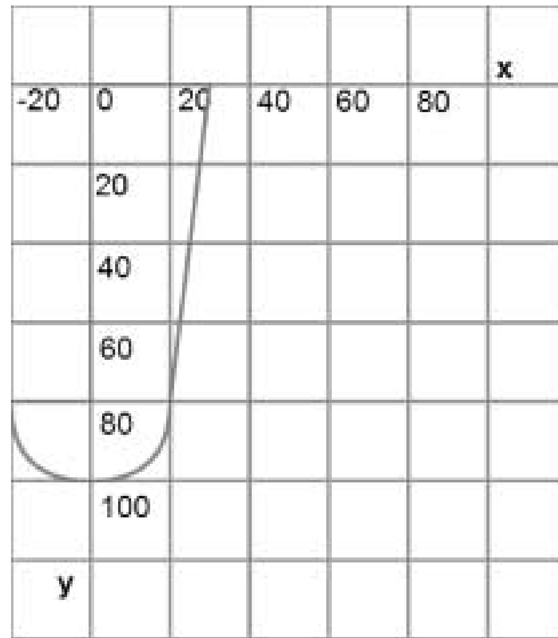
Dynamic drawing

At his disposal, Jimmy has the new state of the art Paint-O-Rama 3000. It's an amazing new auto-body tool that allows him to paint perfect logos and images on the bodies of his cars. At its core, the Paint-O-Rama 3000 has the ability to draw lines, and to draw solid filled areas.

Let's imagine that the surface of the hood of the car is a grid. Upon this grid, Jimmy would like to paint his new logo, the Jimmy "J", like so:



Jimmy still has a thing or two to learn about logo design, but alas, it will have to suffice. The first thing to do is look at our grid, and imagine how we must draw upon it.

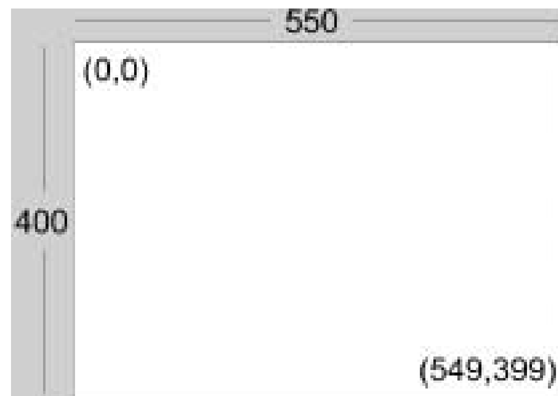


Here we can see the J mapped on the grid surface. The grid is made up of x (horizontal) and y (vertical) coordinates, where x goes from left to right, and y goes from top to bottom. The top-left corner of the J logo starts at x=0 and y=0, or simply "(0,0)". In fact, the drawing instructions that Jimmy would give to the Paint-O-Rama 3000 for this image, are like so:

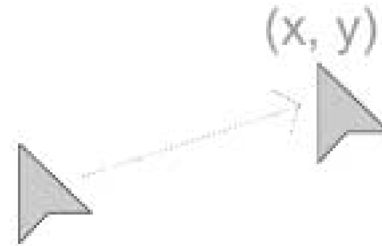
1. *Begin at (0, 0)*
2. *Draw a line out to (30, 0) - The top of the J.*
3. *Draw a straight line down and left to (20, 80) – the diagonal*
4. *right edge.*
5. *Draw a curve down and left to (0, 100), with the point at (20, 100) being where the curve will 'pull' towards.*
6. *Draw a curve up and left to (-20, 80), with the point at (-20, 100) being where the curve will 'pull' towards.*
7. *Draw a straight line right to (0, 80).*
8. *Draw a straight line back up to (0, 0).*
9. *Fill in the enclosed J.*

Once these instructions have been fed into the Paint-O-Rama 3000, it will get to work drawing the lines and then filling in the shape to produce a perfect Jimmy "J" every time.

In ActionScript we have our own version of the Paint-O-Rama 3000, but it is known by the slightly less catchy name 'Drawing API (Application Program Interface)'. Our Paint-O-Rama 3000 uses the stage as its canvas, rather than the hood of a car. The stage is typically (by default) 550 pixels wide by 400 pixels high, and position (0,0) is in the upper left hand corner. Here is the typical stage:

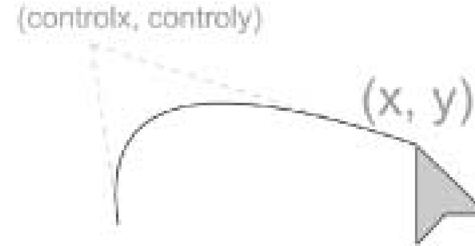


The drawing API has several important functions that we can make use of:



- `moveTo(x, y)` – This moves the 'drawing pen' to a specific location on the stage.

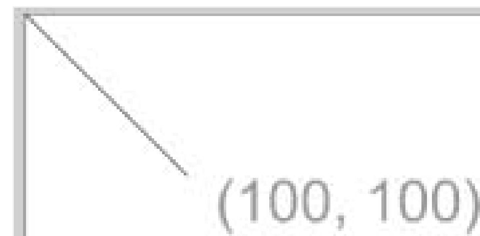
- `lineTo(x, y)` – This draws a line from the pen's current position, to the position (x, y), using the current line style.



- `curveTo(controlx, controly, x, y)` – This draws a curved line from the previous pen position to (x, y), while using the point at (controlx, controly) to influence the path of the curve.
- `lineStyle(thickness, color, alpha)` – This sets the line style of the current drawing pen. In this method we specify the thickness, color and alpha (transparency) value of the line. The thickness is the same as we would specify in the Flash design environment when drawing lines by hand. If we specify a thickness of 0, then Flash will make the line into a hairline thickness. If we want no line at all, then we must specify 'undefined' for the thickness.
- `beginFill(color, alpha)` – This is used to tell the drawing API that any subsequent lines will be used to define an area that we'd like to use to define a solid, filled area.
- `endFill()` – This is used to tell the drawing API to fill in the area we have just defined.

How
These are the basic drawing functions. To draw a solid, 1 point black line from (0, 0) to (100, 100) we would do this:

```
_root.lineStyle (1, 0);
_root.moveTo(0, 0);
_root.lineTo (100, 100);
```



This line is being drawing straight on the `_root` timeline. Generally, we would want to create an empty 'container' movie clip to contain our drawing API creations. We do this using the `createEmptyMovieClip` method. It works something like this:

```
source.createEmptyMovieClip
(instanceName, depthLevel);
```

Once we've created an empty movie clip, we can then move it around the screen freely using its _x and _y properties. To create the simple line using a dynamically created movie clip, we would do this:

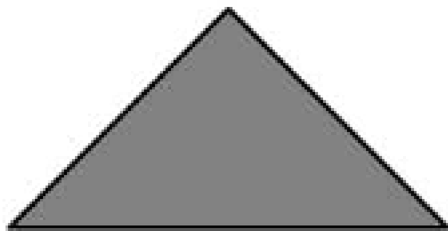
```
_root.createEmptyMovieClip ("myLine", 0);
myLine.lineStyle (1, 0);
myLine.moveTo(0, 0);
myLine.lineTo (100, 100);
```

Now, if we wanted to, we could move the whole line movie clip to a new location:

```
myLine._x = 50;
myLine._y = 50;
```

...and then the line would, on screen, extend from (50, 50) to (150, 150), because it is still 100 x 100, but it now begins at (50, 50) on screen.

Let's say that we wanted to program the drawing API to create a simple solid shape, like a triangle. Open up `triangle.fla`, and look at the output when it's run (CTRL/⌘+ENTER):



There is nothing in this movie except for some code on frame 1 of the code layer. Let's look at that code here:

```
_root.createEmptyMovieClip("triangle", 0);

triangle.lineStyle(1, 0);
triangle.beginFill(0xFF0000);
triangle.moveTo(50, 0);
triangle.lineTo(100, 50);
triangle.lineTo(0, 50);
triangle.lineTo(50,0);
triangle.endFill();

triangle._x = 100;
triangle._y = 100;
```

It's as if we're looking at Jimmy's own handwriting! First, we're creating a nice empty movie clip, with the instance name 'triangle', on depth level 0. Then, within it, we're using the drawing API to set the line to solid black, 1 point, and then beginning a fill of solid red (`0xFF0000`). Finally, we're drawing a triangle that has it's top point at (50, 0) and then its two lower points at (100, 50) and (0, 50). This triangle is 100 wide, and 50 high.

Once all the lines are drawn, we're telling the drawing API to fill in the shape with `endFill()`. Finally, we're moving the triangle movie clip to position (100, 100). Remember, the positions of the drawn lines within the triangle movie clip are relative to the internal coordinate system of the triangle movie clip itself. So, if the tip of the triangle is at (50, 0), but the entire triangle is moved to (100, 100), the tip will physically be sitting at (150, 100), but internally, it will still be at (50, 0).

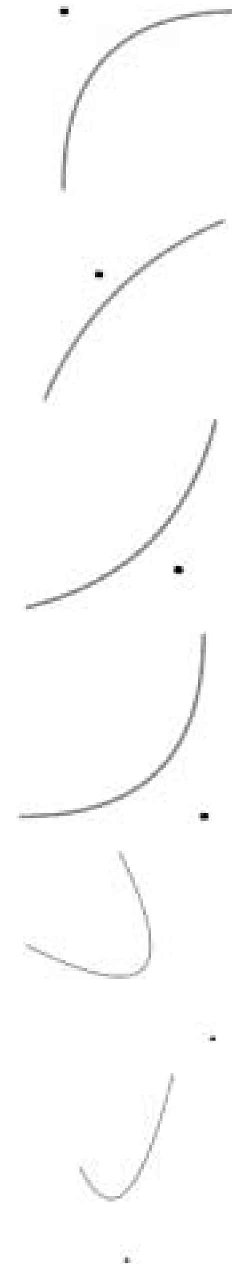
Think of it like this; if Jimmy paints a 100 x 50 centimeter triangle on the hood of a car, at position (100, 100) it will be fixed on the hood of the car permanently in that location. Jimmy can, however, pick up the hood and move it elsewhere in the garage. The painted image will move with the hood, meaning that its location relative to the hood does not change, but relative to the shop floor the triangle is moving around.

The Logo

Now, let's take a look at `jimmyslogo.fla`. When this movie is loaded up, we'll see that it's empty – there's nothing on the stage. That's because everything is being created and drawn dynamically when the program is "run", or "at run-time" The only thing to be seen in this movie is the code layer.

Think back to Jimmy's logo, and the way the Paint-O-Rama 3000 was told what to do. The drawing API code is quite similar:

```
_root.createEmptyMovieClip ("jimmysLogo", 1);

jimmysLogo.lineStyle(2, 0xFF0000);
jimmysLogo.beginFill(0x330000);
jimmysLogo.moveTo(0, 0);
jimmysLogo.lineTo(30, 0);
jimmysLogo.lineTo(20, 80);
jimmysLogo.curveTo(20, 100, 0, 100);
jimmysLogo.curveTo(-20, 100, -20, 80);
jimmysLogo.lineTo(0, 80);
jimmysLogo.lineTo(0,0);
jimmysLogo.endFill();

jimmysLogo._x = 110;
jimmysLogo._y = 110;
```



We're creating a movie clip with the instance name 'jimmyslogo', on depth level 1. After this, we're using the drawing API to set the line to 2 point, bright red, and the fill to `0x330000`, which is a dark red color. Then, we're drawing our outline, by following the same list of instructions given to the Paint-O-Rama 3000. Once the logo is drawn, we're using the `endFill` method to fill the logo in, and then moving the whole position of the logo movie clip to (110, 110).

The `curveTo` method calls for some brief explanation. Basically, `curveTo` uses a method known as a 'bezier' curve, which creates curves based on the tangent line between the control point and the end points; a very complex series of calculations. Don't worry!! Luckily for us, the drawing API does all the work, and we simply need to specify two points: the control point and the destination point.

The control point acts by 'influencing' or 'pulling' the curve towards it. The farther out the control point is, the larger the curve will be. In these images, the control point is the small dot, and its influence is shown on the curve.

As we can see, choosing the position for the control point involves a bit of trial and error, but it can be done with practice. Take some time to have a go and get the feel of it.

We can also play with this in the Flash MX design environment itself because this is how the curve-drawing interface works: We draw a line, and then we pull it into curves. Flash keeps track of an invisible control point that we cannot see. Here, we can see a curve being manually drawn in Flash MX:

Summary

Once Jimmy masters the use of the paintbrush, and the Paint-O-Rama 3000, the sky is the limit. Sure he may never be painting the ceiling of the Sistine Chapel, but as chrome is his canvas, he'll be creating the artwork that we see every day on the road (and on the Web). To finish it all off, here's a neat little program that Jimmy wrote for the Paint-O-Rama 3000, for the days when he's feeling lazy, and looking for a neat design. This is found in `randompaint.fla`.

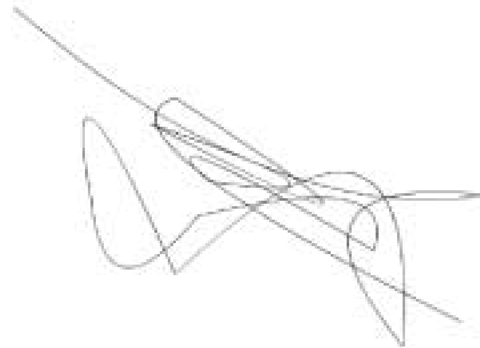
This code is attached to frame 1 of the Actions layer (the only layer).

```
_root.createEmptyMovieClip("painting", 1);

painting.lineStyle(0, 0);
painting.beginFill(0x000099, 50);
for (i = 0; i < 20; i++)
{
  controlx = Math.random() * 550;
  controly = Math.random() * 400;

  pointx = Math.random() * 550;
  pointy = Math.random() * 400;
  painting.curveTo(controlx, controly,
  ➥ pointx, pointy);
}
painting.endFill();
```

Any guesses as to what this does? Here's some examples:

That's right – never the same image twice. The `Math.random` function is being used to come up with four random numbers; one each for 'controlx', 'controly', 'pointx' and 'pointy'. For more on `Math.random`, see **Chapter 8**.

The power is endless. Remove the lines `beginFill` and `endFill`, and we can get some random celebrity's autograph every time (from `randomautograph.fla`).



List of Decimal numbers from 0 to 255 with their hexadecimal equivalents (0 to FF).

Dec	Hex						
0	0	47	2F	94	5E	141	8D
1	1	48	30	95	5F	142	8E
2	2	49	31	96	60	143	8F
3	3	50	32	97	61	144	90
4	4	51	33	98	62	145	91
5	5	52	34	99	63	146	92
6	6	53	35	100	64	147	93
7	7	54	36	101	65	148	94
8	8	55	37	102	66	149	95
9	9	56	38	103	67	150	96
10	A	57	39	104	68	151	97
11	B	58	3A	105	69	152	98
12	C	59	3B	106	6A	153	99
13	D	60	3C	107	6B	154	9A
14	E	61	3D	108	6C	155	9B
15	F	62	3E	109	6D	156	9C
16	10	63	3F	110	6E	157	9D
17	11	64	40	111	6F	158	9E
18	12	65	41	112	70	159	9F
19	13	66	42	113	71	160	A0
20	14	67	43	114	72	161	A1
21	15	68	44	115	73	162	A2
22	16	69	45	116	74	163	A3
23	17	70	46	117	75	164	A4
24	18	71	47	118	76	165	A5
25	19	72	48	119	77	166	A6
26	1A	73	49	120	78	167	A7
27	1B	74	4A	121	79	168	A8
28	1C	75	4B	122	7A	169	A9
29	1D	76	4C	123	7B	170	AA
30	1E	77	4D	124	7C	171	AB
31	1F	78	4E	125	7D	172	AC
32	20	79	4F	126	7E	173	AD
33	21	80	50	127	7F	174	AE
34	22	81	51	128	80	175	AF
35	23	82	52	129	81	176	B0
36	24	83	53	130	82	177	B1
37	25	84	54	131	83	178	B2
38	26	85	55	132	84	179	B3
39	27	86	56	133	85	180	B4
40	28	87	57	134	86	181	B5
41	29	88	58	135	87	182	B6
42	2A	89	59	136	88	183	B7
43	2B	90	5A	137	89	184	B8
44	2C	91	5B	138	8A	185	B9
45	2D	92	5C	139	8B	186	BA
46	2E	93	5D	140	8C	187	BB

Dec	Hex
188	BC
189	BD
190	BE
191	BF
192	C0
193	C1
194	C2
195	C3
196	C4
197	C5
198	C6
199	C7
200	C8
201	C9
202	CA
203	CB
204	CC
205	CD
206	CE
207	CF
208	D0
209	D1
210	D2
211	D3
212	D4
213	D5
214	D6
215	D7
216	D8
217	D9
218	DA
219	DB
220	DC
221	DD
222	DE
223	DF
224	E0
225	E1
226	E2
227	E3
228	E4
229	E5
230	E6
231	E7
232	E8
233	E9
234	EA
235	EB
236	EC
237	ED
238	EE
239	EF
240	F0
241	F1
242	F2
243	F3
244	F4
245	F5
246	F6
247	F7
248	F8
249	F9
250	FA
251	FB
252	FC
253	FD
254	FE
255	FF