

Shapes created at run-time. We will discuss the Shape/Sprite/MovieClip hierarchy a bit more at the end of this chapter. For now a simple analogy should suffice.

We can think of a Shape or Sprite as a huge transparency slide with a little "+" marking its registration point. We can draw on the slide using the **graphics** property of the object, and we can attach a new Shape or Sprite as a child to a Sprite. The difference between a Shape and a Sprite is that Shapes cannot have children and cannot listen for events, so a Shape is used for simple graphics only. The registration point of a display object denotes the (0,0) point of its local coordinate system. Meanwhile, the **x** and **y** properties of a display object denote the coordinates of that object's registration point in the coordinate system of the object's "parent." This is a little confusing until you get used to it, so we will have more to say about it at the end of the chapter.

Before delving into the code for this application, it will be useful to summarize the graphics properties and methods for display objects that we will be using. We will see examples of most of these in the remainder of this chapter and beyond.

AS ESSENTIALS: THE GRAPHICS PROPERTY OF A DISPLAY OBJECT

If **d** is a display object (Sprite, Shape or MovieClip), it has a **graphics** property with the following properties and methods. All coordinates given are relative to the coordinate system of the object **d**.

- ▶ **d.graphics.lineStyle(s,c)**; makes the "pen" have stroke width **s** pixels and color **c** (as an integer representing RGB values).
- ▶ **d.graphics.moveTo(a,b)**; moves the "pen" off of the "paper" to point (**a, b**).
- ▶ **d.graphics.lineTo(a,b)**; draws a line from the current location of the "pen" to point (**a, b**).
- ▶ **d.graphics.drawCircle(a,b,r)**; draws a circle of radius **r** pixels with center at point (**a, b**).
- ▶ **d.graphics.drawRect(a,b,w,h)**; draws a rectangle **w** pixels wide by **h** pixels high with upper left corner at local coordinates (**a, b**).
- ▶ **d.graphics.drawEllipse(a,b,w,h)**; draws an ellipse **w** pixels wide by **h** pixels high with center at local coordinates (**a, b**).
- ▶ **d.graphics.drawRoundRect(a,b,w,h,ew,eh)**; draws a rounded rectangle **w** pixels wide by **h** pixels high with upper left corner at local coordinates (**a, b**). The numbers **ew** and **eh** determine the width and height of the ellipse that is used for the rounded corners.
- ▶ **d.graphics.beginFill(c,a)**; sets a color of **c** (as an integer representing RGB values) and an alpha (as a number between 0 and 1) value of **a** for the fill. (The default value of **a** is 1, representing

no transparency.)

- ▶ **d.graphics.endFill()**; any closed figure drawn between the **beginFill(...)** and **endFill()** calls will be filled.
- ▶ **d.graphics.clear()**; erases everything drawn in the **graphics** property of **d**.
- ▶ In Flash CS4, there are additional methods available, including the **drawPath** method that we will discuss briefly in **Chapter 10**.

To learn more about graphics in ActionScript, select from the main menu [Help > Flash Help](#), and browse to "Using the drawing API" under the "Programming in ActionScript 3.0" heading.

Open a working version (yours or our **CircleMotion.fla**) of the previous example, save it under a new name like **myCircleSprite.fla**, and delete (select and hit the delete keyboard key) the circle and **mcArrow** from the stage. Replace lines 1-9 of this version with the following code.

```
var degree:Number = 0;
var degChange:Number = 1;

var circleX:Number = 200;
var circleY:Number = 200;
var circleR:Number = 100;

var spBoard:Sprite = new Sprite();
spBoard.x = circleX;
spBoard.y = circleY;
addChild(spBoard);

var shCircle:Shape = new Shape();
shCircle.graphics.lineStyle(2,0);
shCircle.graphics.drawCircle(0,0,circleR);
shCircle.x = 0;
shCircle.y = 0;
spBoard.addChild(shCircle);

var shArrow:Shape = new Shape();
shArrow.graphics.beginFill(0x000000);
shArrow.graphics.moveTo(0,1);
shArrow.graphics.lineTo(circleR-15,1);
shArrow.graphics.lineTo(circleR-15,10);
shArrow.graphics.lineTo(circleR-1,0);
shArrow.graphics.lineTo(circleR-15,-10);
shArrow.graphics.lineTo(circleR-15,-0);
shArrow.graphics.lineTo(0,-1);
shArrow.graphics.lineTo(0,1);
shArrow.graphics.endFill();
shArrow.x = 0;
shArrow.y = 0;
```

// We make a Sprite called **spBoard** on which to place the other elements. Changing the **x** and **y** properties of **spBoard** will move it along with all of its children.

// A Shape cannot have children added to it, so we use the Sprite class for **spBoard** and the Shape class for **shCircle** and **shArrow**.

// The block of code for **shArrow** does nothing more than draw a filled arrow shape and add it as a child of **spBoard**.

// Everything drawn for **shArrow** with the **moveTo/lineTo** methods will be filled since these lines fall between **beginFill** and **endFill** statements.

// Graph paper is useful for organizing the drawing of a complicated shape in the code.

```
spBoard.addChild(shArrow);
```

Note that we now refer to the “arrow” on the stage as **shArrow** to remind us that it is a Shape object. Consequently, we will need to change the references to **mcArrow** to be references to **shArrow** in all remaining functions. If you click in the **Actions** panel and select from the main menu **Edit > Find and Replace** (shortcut Ctrl+f), you can make this change quickly.

See **CircleMotionSprite.fla** in **Chapter3** of the download folder http://flashandmath.com/appletsbook_files for the complete source code for this example.

If you test the movie at this point, you will see that it behaves just like the previous version. Interesting, sure, but why should we bother with the extra code if we don’t have to? The next example illustrates the power of drawing our display objects at run time.

FILLING GRAPHICS FOR SHADED AREA

In our final example, we illustrate an advantage to updating the drawing at run time. The basic interface remains the same, but now the exploration focuses on the area of a circular sector in terms of the angle sweeping out the sector, as shown in **Figure 6**. The dynamically changing, filled shape is not a native ActionScript method. Instead, the secret here is to draw a filled shape (just as we did for our arrow) that gives the illusion of filling the circular sector. In reality, at each step we are drawing part of a filled 360-sided polygon — the speed in which this can be cleared and redrawn gives the perfect effect of interactive motion!

For a finished version of this applet, see <http://www.flashandmath.com/appletsbook/CircleFillSprite.html>

Save a copy of **CircleMotionSprite.fla** with a new name like **myCircleFill.fla**, and replace the block of code that defines the **shArrow** Shape with the the following code:

```
var shFill:Shape = new Shape();
```

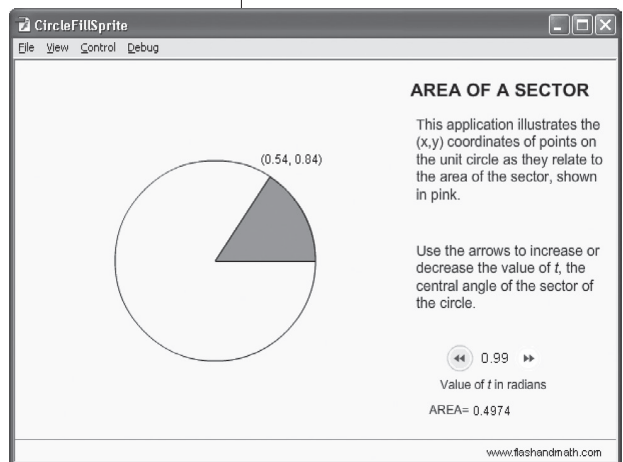


Figure 6. Filled Area

// The Shape **shFill** will eventually be a circular sector, so we pin it to **spBoard** with its registration point (0,0) at the center of the circle that we previously drew on **spBoard**.

```

shFill.graphics.lineStyle(1,0x000000);
shFill.graphics.moveTo(0,0);
shFill.graphics.lineTo(circleR,0);
shFill.x = 0;
shFill.y = 0;
board.addChild(shFill);

```

Now we will also change the function **updateArrow** to do more than just rotate the radial line. At each step, we clear all the graphics from **shFill** and then redraw an entirely new picture composed of an appropriate number of sides of a 360-sided polygon inscribed in our circle. This will give the appearance of filling the circular region.

To make this work, we will use a loop control structure in ActionScript. Let's see how this works before applying it:

ACTIONSCRIPT: THE FOR LOOP

This is the syntax for a loop that starts with index value (**i**) of 0 and increments the index by one (via **i++**;) until the test (**i<10**) fails. For each value of **i**, everything within the { } delimiters is executed.

```

for (i=0; i<10; i++) {
    what to do;
}

```

Experiment. In a new ActionScript 3 file, place the following code in the **Actions** panel:

```

var i:int;
var s:Number = 0;
for (i=1; i<11; i++) {
    s = s + i;
    trace(i,s);
}

```

In the **Output** panel, you will see the values of **i** and **s** at each iteration of the loop. Notice the first and last values and how those are reflected in the syntax of your loop.

Another common loop structure is the while loop that has a somewhat simpler form:

```

while (something is true) {
    do stuff;
}

```

Experiment. In a new ActionScript 3 file, place the following code in the **Actions** panel.

```

var s:Number = 49;
while (s > 0) {
    trace(s%2);
    s = Math.floor(s/2);
}

```

Since with each iteration, the value of **s** is cut in half, eventually the condition "**s>0**" will no longer be true, and the loop stops. Did you recognize the binary digits for the number 49 in the **Output** panel?

```

function updateArrow(t:Number):void {
    var radianAngle:Number = t*Math.PI/180.0;
    var i:int;

    shFill.graphics.clear();
    shFill.graphics.lineStyle(1,0x000000);
    shFill.graphics.moveTo(0,0);
    shFill.graphics.beginFill(0xFF00FF,0.7);
    for (i=0; i<=t; i++) {
        shFill.graphics.lineTo(
            circleR*Math.cos(i*Math.PI/180),
            -circleR*Math.sin(i*Math.PI/180) );
    }
    shFill.graphics.lineTo(0,0);
    shFill.graphics.endFill();

    txtCoords.x = circleX +
        (40+circleR)*Math.cos(radianAngle)-40;
    txtCoords.y = circleY -
        (20+circleR)*Math.sin(radianAngle)-10;

    txtCoords.text = "(" +
        Math.cos(radianAngle).toFixed(2) + ", " +
        Math.sin(radianAngle).toFixed(2) + ")";

    txtDegrees.text = radianAngle.toFixed(2);
    txtArea.text = (radianAngle/2).toFixed(4);
}

```

// The loop draws tiny lines between points on the circle one separated from each other by one degree.

// The final `lineTo` outside of the loop takes the "pen" back to its starting point. Since the drawing is between `beginFill` and `endFill`, we get the filled shape.

// The remaining lines position the `txtCoords` textbox, create a string showing the coordinates to two decimal places to put this into this textbox, and put the current radian value of `t` and the area of the shaded region into the appropriate textboxes.

See **CircleFillSprite.fla** in **Chapter3** of the download folder http://flashandmath.com/appletsbook_files for the complete source code for this example.

MORE ON THE DISPLAY LIST AND COORDINATE SYSTEMS

We close this chapter with a brief discussion about the Display List and the relative coordinate systems that come from the parent/child relationship. We will not be working through a source code file in this section, but instead we will try some experiments with a particular interactive applet that illustrates some of the important points. We will discuss the snippets of code relevant to some of the behavior that we observe. You should examine the HTML page below before reading on:

For the applet under discussion, see <http://www.flashandmath.com/appletsbook/ParentChild.html>