Chapter 1: Spark effects

Spark effects define a change to the target over time. The Spark effects ship in the spark.effects package.

While Flex ships with both Spark and Halo effects, Adobe recommends that you use the Spark effects when possible.

For an introduction to effects, see Introduction to effects.

About Spark effects

The Spark effects are divided into categories based on their implementation and target type, as the following table shows:

Type	Description
Property effects	Animates the change of one or more properties of the target.
Transform effects	Animates the change of one or more transform-related properties of the target, such as the scale, rotation, and position. The target can be modified in parallel with other transform effects with no interference among the effects.
Pixel-shader effects	Animates the change from one bitmap image to another, where the bitmap image represents the before and after states of the target.
Filter effects	Applies a filter to the target where the effect modifies the properties of the filter, not properties of the target.
3D effects	A subset of the transform effects that modify the 3D transform properties of the target.

Spark effects can be applied to:

- Any Spark or Halo component.
- Any graphical component in the spark.primitives package, such as Rect, Ellipse, and path.
- Any object that contains the styles or properties modified by the effect.

Spark property effects

The Spark property effects modify one or more properties of a component over time to animate the effect. The Spark property effects are subclasses of the Animate class, and include the following classes:

- Animate Animates any properties of the target.
- Fade Animates a change to the `alpha` property of the target.
- Resize Animates a change to the `height` and `width` properties of the target.
- AnimateColor Animates a change to any color property of the target, such as `color`, or `fontColor`.

The Animate effect

Use the Animate effect directly to animate any properties of the target. You create an instance of the SimpleMotionPath class for each property to animate, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimateProp.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Animate id="scaleUp"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.0" valueTo="1.5"/>
        </s:Animate>
        <s:Animate id="scaleDown"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.5" valueTo="1.0"/>
        </s:Animate>
    </fx:Declarations>

    <s:Button id="myB1"
        label="Scale Button"
        mouseDown="scaleUp.end(); scaleUp.play();"
        mouseUp="scaleDown.end(); scaleDown.play();"/>
</s:Application>
```

The SimpleMotionPath class defines the name of the property, the property's starting value, and the property's ending value.

You can define multiple instance of the SimpleMotionPath class to animate multiple properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimate2Prop.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Animate id="scaleIncrease"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.0" valueTo="1.5"/>
            <s:SimpleMotionPath property="scaleY"
                valueFrom="1.0" valueTo="1.5"/>
        </s:Animate>
        <s:Animate id="scaleDecrease"
            target="{myB1}">
            <s:SimpleMotionPath property="scaleX"
                valueFrom="1.5" valueTo="1.0"/>
            <s:SimpleMotionPath property="scaleY"
                valueFrom="1.5" valueTo="1.0"/>
        </s:Animate>
    </fx:Declarations>

    <s:Button id="myB1"
        label="Scale Button"
        mouseDown="scaleDecrease.end(); scaleIncrease.play();"
        mouseUp="scaleIncrease.end(); scaleDecrease.play();"/>
</s:Application>
```

The Fade, Resize, and AnimateColor effects

The Fade, Resize, and AnimateColor effects modify specific properties of the target. These effects are predefined to modify specific properties. So, you do not need to use the SimpleMotionPath class to specify the property, as you do for the Animate effect.

The following example uses the Resize effect on an Image control:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkResizeEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Declarations>
        <s:Resize id="myResizeEffect"
            target="{myImage}"
            widthBy="10" heightBy="10"/>
    </fx:Declarations>

    <mx:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Resize Me"
        click="myResizeEffect.end();myResizeEffect.play();"/>
</s:Application>
```

This example resizes the Image control by increasing its height and width by 10 pixels when the user clicks the Button control.

The Resize effect defines six properties that you can use to configure it: `heightBy`, `widthBy`, `heightFrom`, `widthFrom`, `heightTo`, and `widthTo`. If omitted, Flex automatically calculates any values for these properties. In the previous example, `heightFrom` and `widthFrom` are set to the current size, and `heightTo` and `widthTo` are set to the final size.

Spark transform effects

The Spark property effects work by modifying one or more properties of the target over the duration of the effect. Using the property effects, you can define two effects that play in parallel but modify the same property of the target. In this situation, the two effects conflict with each other and the results of the effects are undefined.

The Spark transform effects are designed to work together. You can define multiple effects that, while playing in parallel, do not interfere with each other. When executing multiple transform effects in parallel, the effects are combined into a single transformation, rather than playing independently. By combining the transform effects, Flex eliminates the chance that one effect could interfere with another.

The Spark transform effects include the Move, Rotate, and Scale effects.

Applying transform effects

The following example applies the Rotate and Move effects in parallel to an Image control:

```
<?xml version="1.0"?>
<!-- behaviors\SparkXFormRotateMove.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Parallel id="parallelRMForward"
            target="{myImage}">
            <s:Rotate
                angleBy="180"/>
            <s:Move
                xBy="100"
                yBy="100"/>

        </s:Parallel>
        <s:Parallel id="parallelRMBack"
            target="{myImage}">
            <s:Rotate
                angleBy="180"/>
            <s:Move
                xBy="-100"
                yBy="-100"/>
        </s:Parallel>
    </fx:Declarations>

    <s:Button label="Play Effect Forwar"
        x="10" y="10"
        click="parallelRMForward.end();parallelRMForward.play();"/>
    <s:Button label="Play Effect Backwar"
        x="150" y="10"
        click="parallelRMBack.end();parallelRMBack.play();" />
    <mx:Image id="myImage"
        x="10" y="50"
        source="@Embed(source='assets/logo.jpg')"/>
</s:Application>
```

Transform effects operate relative to the *transform center* of the target. By default, the transform center of the target is the upper-left corner of the target, corresponding to coordinates (0,0) in the target coordinate system.

For the Move effect, the change in position is measured by a change of the location of the transform center of the target. The Rotate effect rotates the target around the transform center, and the Scale effect scales the target while the transform center of the target stays stationary.

You do not have to use the upper-left corner of the target as the transform center. For example, instead of rotating the target around the upper-left corner, you rotate the target around its center point.

You can set the coordinates of the transform center either on the target itself, or on the effect class. If you set the transform center on the target, all transform effects use that transform center. Use the UIComponent properties `transformX`, `transformY`, and `transformZ` to define the transform center of the target.

If you set the transform center on the effect, the effect uses that transform center for the duration of the effect. Use the `transformX`, `transformY`, and `transformZ` properties on the transform effect classes to set the transform center for all effect targets. Setting these properties on the effect class overrides the corresponding setting on the target.

You do not have to specify all three properties. For example, if you only want to specify the x location of the transform center, set only the `transformX` property on the target or on the effect class.

You can also set the effect's `autoCenterTransform` property to `true` to configure the effect to play the transform around the center point of the target. If you apply the effect to multiple targets, Flex calculates the transform center independently for each target.

If you apply multiple transform effects in parallel to the same target, set the transform center to the same value for all the effects. For example, set the `autoCenterTransform` property to `true` for one transform effect, set it to the same value for all effects.

The next example modifies the previous example set the `autoCenterTransform` property to `true`. The effects now operate on the center of the target:

```
<?xml version="1.0"?>
<!-- behaviors\SparkXFormRotateMoveAutoCenter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Parallel id="parallelRMForward"
            target="{myImage}">
            <s:Rotate
                angleBy="180"
                autoCenterTransform="true"/>
            <s:Move
                xBy="100"
                yBy="100"
                autoCenterTransform="true"/>
        </s:Parallel>
        <s:Parallel id="parallelRMBack"
            target="{myImage}">

        <s:Rotate
                angleBy="180"
                autoCenterTransform="true"/>
            <s:Move
                xBy="-100"
                yBy="-100"
                autoCenterTransform="true"/>
        </s:Parallel>
    </fx:Declarations>

    <s:Button label="Play Effect Forward"
        x="10" y="10"
        click="parallelRMForward.end();parallelRMForward.play();"/>
    <s:Button label="Play Effect Backward"
        x="150" y="10"
        click="parallelRMBack.end();parallelRMBack.play();" />
    <mx:Image id="myImage"
        x="10" y="50"
        source="@Embed(source='assets/logo.jpg')"/>
</s:Application>
```

Limitations with transform effects

The only limitation of transform effects is that they cannot be played multiple times. Therefore, you cannot use the `repeatCount`, `repeatDelay`, and `repeatBehavior` properties with transform effects.

Spark 3D effects

Most Flex effects manipulate the effect target in the x and y dimensions to create two-dimensional effects. In the two-dimensional coordinate system, the x, y coordinates of 0, 0 corresponds to the upper-left corner of the component's coordinate system. For example, if the Application container takes up your full computer screen, those coordinates correspond to the upper-left corner of the computer screen. Increasing values of x moves to the right of the compute screen, and increasing values of y move down the screen.

The 3D effects add support for the z-axis. The z = 0 coordinate corresponds to the plane of the computer screen. Increasing values of z moves an object into the screen, making the object look farther away from the viewer. Decreasing values of z move the object toward the viewer.

The spark 3D effects are transform effects designed to take advantage of the support for three-dimensional graphics in Flash Player. Flex includes the following 3D effects:

- Move3D

  Moves the effect target in the x, y, and z coordinate system. Moving the target to increasing values in the z direction makes it appear to move back away from the viewer, so the target shrinks. Moving the target to decreasing values in the z direction makes it appear to move toward the viewer, so the target grows.

- Rotate3D

  Rotates the effect target around the x, y, or z-axis. For example, rotating the target around the y-axis rotates the object vertically through the x and z planes, similar to a door opening and closing on vertical hinges. Rotating the target around the z-axis makes the object rotate through the x and y planes, which is the same as a two-dimensional rotation.

- Scale3D

  Scales the effect target in the x, y, and z directions by setting the appropriate scale properties on the target. A scale of 2.0 means the object has been magnified by a factor of 2, and a scale of 0.5 means the object has been reduced by a factor of 2. A scale value of 0.0 is not allowed.

For an introduction to transform effects, see "Spark transform effects" on page 4.

Applying 3D effects

The following example applies the Move3D effect to an image:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3MoveEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Move3D id="moveEffect" target="{targetImg}"
            xBy="100" zBy="100"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>

    <s:Panel title="Move3D Effect Example"
        width="75%" height="75%" >

        <mx:Image id="targetImg"
            horizontalCenter="0"
            verticalCenter="0"
            source="@Embed(source='assets/Nokia_6630.png')"/>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Move3D"
            click="moveEffect.play();"/>
    </s:Panel>
</s:Application>
```

In this example, the Move3D effect moves the target by increasing the value of the x and z coordinates by 100. It then reverses, and moves the component back to its original x and z coordinates. To the user, the image moves to the left and shrinks, then moves back to the right and grows to its original size.

The following example uses the Rotate3D effect to rotate the image around the y-axis:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateEffect.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>

    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <mx:Image id="targetImg"
            horizontalCenter="0"
            verticalCenter="0"
            source="@Embed(source='assets/Nokia_6630.png')"/>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Setting the transform center of a 3D effect

By default, the transform center of the target of a 3D transform effect is the upper-left corner of the target component, corresponding to coordinates (0, 0, 0) in the target component's coordinate system. You can set the transform center to a different location. For an introduction to setting the transform center for two-dimensional effects, see "Applying transform effects" on page 4.

If you run the example in the previous section, you notice that the target object rotates around its left edge. That is because the effects, by default, operates around the default transform center of the target object.

Often, you want to rotate the target component around its center point instead of around the default transform center. One option is to set the transform center to the center point of the component by setting the `autoCenterTransform` property to `true`. The following example modifies the example from the previous section to rotate the image around its center point:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateLayoutCenterTransform.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            duration="3000"
            autoCenterTransform="true"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>

    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Alternatively, you can set the `transormX`, `transformY`, and `transformZ` properties on the effect target to define the transform center. In the following example, you set the transform center of the image to the right edge, corresponding to the `transormX` property having a value of 100:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateLayoutRightTransform.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            duration="3000"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>

    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image id="targetImg"
                transformX="100"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Component layout and 3D effects

Effects can modify the layout of the parent container of the effect target. For example, suppose the effect target is in a container that uses vertical layout. You then use the two-dimensional Rotate effect to rotate the target through 360°. As the effect plays, the parent container modifies the layout of its other children to accommodate the rotating child. Therefore, container children can change position during the effect.

Flex has the concept of layering of the children of a container. Children are drawn on the screen in the order in which they are defined in the container. If children overlap, the child defined later in the container appears on top because it is drawn last.

Flex provides a set of built-in layout classes, such as the VerticalLayout class, to control child layout in a container. These layout classes assume that all children are in the z = 0 plane. That means a container always lays out children in the two-dimensional x, y plane.

However, a 3D effect can modify the effect target in the x, y, and z dimensions. If your 3D effect uses the z dimension, the built-in layout classes do not consider it during layout. If you then allow the parent container to update its layout by taking only the x and y dimensions into consideration, your application might not appear correctly. Therefore, you typically disable the parent container from performing layout while the 3D effect plays.

All effects support the `disableLayout` property. When set to `true`, this property disables layout in the parent container of the effect target for the duration of the effect. The default value is `false`. For effects though, you typically do not want to disable layout of the parent container entirely.

All transform effects define the `applyChangesPostLayout` property which, by default, is set to `true` for the 3D effects. This setting lets the 3D effect modify the target component, but the parent container ignores the changes and does not update its layout while the effect plays. Changes to other container children still cause a layout update.

Note: *For the 2D transform effects Move, Rotate, and Scale, the affectLayout property is true by default.*

You can think of the 3D effects as not playing until after the container for the effect target has completed its layout. Because the effect plays post layout, the parent container does not modify its layout for changes to the target component caused by the effect.

In the following example, the application contains three images in an HGroup container. The Rotate3D effect then plays on the middle image to rotate it around the y-axis. Because the `applyChangesPostLayout` property is `true` by default, no layout occurs as the image rotates, and the target image overlaps the image on its left:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateLayout.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleYFrom="0" angleYTo="360"
            duration="3000"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>

    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

For example, set the `transformX` property to 100 on the effect target in the previous example to rotate the middle image so that it overlaps the image on the right. The image on the right appears on top of the middle image because it was drawn last.

You can override the default of disabling layout on the effect target by setting the `applyChangesPostLayout` property of the effect to `false`. In the following example, you rotate the image around the z-axis, with the `applyChangesPostLayout` property set to `false`. A 3D rotation around the z-axis is essentially a 2D rotation in the x and y plane. Because the `applyChangesPostLayout` property is `false`, the parent container updates the layout of the other two images as the effect plays:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DRotateWithLayout.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Rotate3D id="rotateEffect" target="{targetImg}"
            angleZFrom="0" angleZTo="360"
            applyChangesPostLayout="false"
            duration="5000"
            repeatCount="2" repeatBehavior="reverse"
            effectStart="playButton.enabled=false;"
            effectEnd="playButton.enabled=true;"/>
    </fx:Declarations>

    <s:Panel title="Rotate 3D Effect Example"
        width="75%" height="75%" >

        <s:HGroup
            horizontalCenter="0"
            verticalCenter="0">
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>

        <s:Button id="playButton"
            left="5" bottom="5"
            label="Rotate3D"
            click="rotateEffect.play();"/>
    </s:Panel>
</s:Application>
```

Setting the postLayoutTransformOffsets property on a component

You can directly modify the position, rotation, and scale of a component post layout without using a 3D effect. Instead, use the `postLayoutTransformOffsets` property, of type mx.geom:TransformOffsets, of the UIComponent class. By setting the `postLayoutTransformOffsets` property directly, you modify the target without causing the parent container to update its layout.

In the following example, you use the `postLayoutTransformOffsets` property to modify the position and scale of a component. As you modify it, notice that the parent container does not update its layout:

```
<?xml version="1.0"?>
<!-- behaviors\Spark3DOffset.mxml -->
<s:Application
    xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[
            import mx.geom.TransformOffsets;

            // Define an instance of TransformOffsets.
            private var myXForm:TransformOffsets = new TransformOffsets();

            // Initialize the postLayoutTransformOffsets property of the target.
            private function initOffsets():void {
                targetImg.postLayoutTransformOffsets = myXForm;
            }

            // Move the target 20 pixels to the left and
            // increase its x and y scale by 0.1.
            private function nudgeImageLeft():void {
                targetImg.postLayoutTransformOffsets.x =
                    targetImg.postLayoutTransformOffsets.x - 20;
                targetImg.postLayoutTransformOffsets.scaleX =
                    targetImg.postLayoutTransformOffsets.scaleX + 0.1;
                targetImg.postLayoutTransformOffsets.scaleY =
                    targetImg.postLayoutTransformOffsets.scaleY + 0.1;
            }

            // Move the target 20 pixels to the right and
            // decrease its x and y scale by 0.1.
            private function nudgeImageRight():void {
                targetImg.postLayoutTransformOffsets.x =
                    targetImg.postLayoutTransformOffsets.x + 20;
                targetImg.postLayoutTransformOffsets.scaleX =
                    targetImg.postLayoutTransformOffsets.scaleX - 0.1;
                targetImg.postLayoutTransformOffsets.scaleY =
                    targetImg.postLayoutTransformOffsets.scaleY - 0.1;
            }

            // Reset the transform.
            private function resetImage():void {
                targetImg.postLayoutTransformOffsets.x = 0;
                targetImg.postLayoutTransformOffsets.scaleX = 1.0;
                targetImg.postLayoutTransformOffsets.scaleY = 1.0;
            }
        ]]>
    </fx:Script>

    <s:Panel title="Offset Example"
        width="75%" height="75%" >

        <s:HGroup
```

```
            horizontalCenter="0"
            verticalCenter="0">
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
            <mx:Image id="targetImg"
                source="@Embed(source='assets/Nokia_6630.png')"
                creationComplete="initOffsets();"/>
            <mx:Image
                source="@Embed(source='assets/Nokia_6630.png')"/>
        </s:HGroup>

        <s:HGroup left="5" bottom="5">
            <s:Button id="nudgeLeftButton"
                label="Nudge Left"
                click="nudgeImageLeft();"/>
             <s:Button id="nudgeRightButton"
                label="Nudge Right"
                click="nudgeImageRight();"/>
             <s:Button id="resetButton"
                label="Reset"
                click="resetImage();"/>
        </s:HGroup>
    </s:Panel>
</s:Application>
```

Notice in this example that as you nudge the image to the left, it overlaps the image to its left. As you nudge the image to the right, it moves behind the image on the right. This overlap is because the image on the right is defined later in the container, and is therefore drawn last on the screen.

Setting the center of the projection

The 3D effects work by mapping a three-dimensional image onto a two-dimensional representation for display on a computer screen. The *projection point* defines the center of the field of view, and controls how the target is projected from three dimensions onto the screen.

By default, when you apply a 3D effect, the effect automatically sets the projection point to the center of the target. You can set the `autoCenterProjection` property of the effect to `false` to disable this default. You then use the `projectionX` and `projectionY` properties to explicitly set the projection point. These properties specify the offset of the projection point from the (0, 0) coordinate of the target.

Spark pixel-shader effects

The Spark pixel-shader effects apply an animation to a target that has a before and after bitmap representation, rather than by animating a change to the value of a property. The effect works by capturing a before bitmap image of the component, capturing an after bitmap image of the component, and then applying the animation between the two images.

While the use of pixel-shader effects might seem obvious when working with Image components, you can use them with any component.

The bitmaps are represented by an instance of the flash.display.BitmapData class. The animation from the initial to the final bitmap is defined by a pixel-shader program created by using Adobe® Pixel Bender™ Toolkit. A pixel-shader program used with the pixel-shader effects operates on two bitmap inputs to animate from one to the other.

The Spark pixel-shader effects, CrossFade and Wipe, use their own internal pixel-shaders. However, you can use a custom pixel-shader defined by using the Pixel Bender Toolkit. For an example of a custom pixel-shader, see "Creating a custom pixel shader" on page 19.

You can use pixel-shader effects on their own, just as you can use the Spark effects such as Fade and Resize. However, since they are designed to by applied to a component with a before and after bitmap, they are often used as part of a transition.

Using a pixel-shader effect in a transition

View states let you change the appearance of an application, typically in response to a user action. Transitions define how a change of view state looks as it occurs on the screen. You define a transition by using the effect classes, in combination with several effects designed explicitly for handling transitions. For more information on view states, see Using View States. For more information on transitions, see Using Transitions.

When using a pixel-shader effect with a transition, you use one view state to represent the before image of the target, and another view state to represent the after image. The following example uses the Wipe effect as part of a transition that changes the font color of a button control from black to red:

```
<?xml version="1.0"?>
<!-- behaviors\SparkWipeEffect.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <!-- Define two view states.-->
    <s:states>
        <s:State name="default"/>
        <s:State name="red"/>
    </s:states>

    <!-- Define the transition that applies the Wipe effect
        whenever the view state changes.-->
    <s:transitions>
        <s:Transition fromState="*" toState="*">
            <s:Sequence target="{myB}">
                <s:Wipe id="wipeEffect"
                    direction="right"
                    duration="1000"/>
            </s:Sequence>
        </s:Transition>
    </s:transitions>

    <s:Button id="myB" label="Wipe"
        color="black" color.red="red"/>

    <!-- Define two buttons to change the view state. -->
    <s:Button label="Set Default State"
        click="currentState='default'"/>
    <s:Button label="Set Red State"
        click="currentState='red'"/>
</s:Application>
```

The transition automatically captures the bitmap of the button in the initial and final states, then plays the Wipe effect on the button to animate the change.

Applying a pixel-shader effect

To use a pixel-shader effect outside a transition, create the bitmaps that define the initial state of the target and the final state of the target, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkShaderBitmap.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">
    <s:layout>
        <s:VerticalLayout/>
    </s:layout>

    <fx:Script>
        <![CDATA[

        import mx.core.BitmapAsset;

        [Embed(source='assets/logo.jpg')]
        [Bindable]
        public var beforeImage:Class;
        [Bindable]
        public var beforeImageBitmap:BitmapAsset = new beforeImage();

        [Embed(source='assets/flexLogo.jpg')]
        [Bindable]
        public var afterImage:Class;
        [Bindable]
        public var afterImageBitmap:BitmapAsset = new afterImage();

        ]]>
    </fx:Script>

    <fx:Declarations>
```

```
        <s:CrossFade id="forwardCF"
            target="{myImage}"
            bitmapFrom="{beforeImageBitmap.bitmapData}"
            bitmapTo="{afterImageBitmap.bitmapData}"
            effectEnd="myImage.source=afterImage;"/>

        <s:CrossFade id="backwardCF"
            target="{myImage}"
            bitmapFrom="{afterImageBitmap.bitmapData}"
            bitmapTo="{beforeImageBitmap.bitmapData}"
            effectEnd="myImage.source=beforeImage;"/>
    </fx:Declarations>

    <mx:Image id="myImage"
        source="{beforeImage}"/>
    <s:Button id="fwd" label="Forward"
        click="forwardCF.end();forwardCF.play();"/>
    <s:Button id="bwd" label="Backward"
        click="backwardCF.end();backwardCF.play();"/>
</s:Application>
```

In this example, you embed two JPEG images, logo.jpg and flexLogo.jpg. When you embed JPEG files into Flex, they are represented as instances of the mx.core.BitmapAsset class. Use the `bitmapData` property of the BitmapAsset class to access the BitmapData object that contains the actual image data. For more information on embedding JPEG files, see Embedding JPEG, GIF, and PNG images.

The first CrossFade effect, forwardCF, uses logo.jpg file to define the before state and the flexLogo.jpg to define the final state of the effect. When it plays, the effect animates the transition from the logo.jpg file to the linelogo.jpg file. After the effect plays, it loads flexLogo.jpg into the Image control. The backwardCF effect does the opposite.

The target of the effect is the Image control. When the effect plays, the following occurs:

1. The effect target, the Image control, is hidden.
2. The bitmap for the initial state displays.
3. The pixel-shader plays to animate the change from the initial state to the final state.
4. When the effect ends, hide the bitmap for the final state and make the Image control visible.

Creating a custom pixel shader

You can use the Pixel Bender Toolkit to create a custom pixel shader, and then pass it to the AnimateTransitionShader effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkCustomPBTransform.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

     <fx:Script>
          <![CDATA[
            // Embed the .pbj file.
            [Embed(source="assets/twist.pbj", mimeType="application/octet-stream")]
            private static var CustomShaderClass:Class;
            private static var customShaderCode:ByteArray = new CustomShaderClass();
          ]]>
     </fx:Script>

     <fx:Declarations>
        <!-- Use the custom pixel shader with an effect. -->
        <s:AnimateTransitionShader  id="shadeAnim"
            shaderByteCode="{customShaderCode}"
            target="{btn2}"
            repeatCount="2"
            repeatBehavior="reverse"/>
     </fx:Declarations>
     <mx:Button id="btn2" label="Click Me"
          click="shadeAnim.play();">
     </mx:Button>
</s:Application>
```

A custom pixel shader is represented by a .pbj file created by using the Pixel Bender Toolkit. To use the .pbj file in your application, you embed it, and then assign it to a property of type ByteArray. You then specify the ByteArray object as the value of the `shaderByteCode` property of the AnimateTransitionShader effect.

The AnimateTransitionShader effect also supports the `shaderProperties` property that lets you pass properties directly to the pixel shader.

The .pbj file has the following requirements:

- Define three image4 inputs. The first input is unused but must be defined and referenced in the code. If the input is not referenced, the compiler in the Pixel Bender Toolkit removes it as part of optimizing the code.

  One image4 input must be called `from`, and one named `to`. These inputs represent the initial and final states of the target.

- Define one parameter named `progress` of type float. This parameter represents the current fraction of the animation completed from 0.0, animating started, to 1.0, animation completed.

The following example shows the source .pbk file for the twist.pbj file used in the previous example:

```
<languageVersion : 1.0;>
kernel FxTwist
<   namespace : "flex";
    vendor : "Adobe";
    version : 1;
    description : "Twisty Effect";
>
{
    input image4 src0;
    input image4 from;
    input image4 to;
    output pixel4 dst;

    parameter float progress<
        minValue: 0.00;
        maxValue: 1.00;
        defaultValue: 0.0;
    >;

    parameter float width<
        minValue: 0.0;
        maxValue: 1024.0;
        defaultValue: 180.0;
    >;

    parameter float height<
        minValue: 0.00;
        maxValue: 1024.0;
        defaultValue: 275.0;
    >;

    void
    evaluatePixel()
    {

        // Common initialization
        float2 outCoord = outCoord();
        pixel4 color1 = sampleNearest(src0, outCoord);

        const float _height = 2.0;
        const float scale = 1.0 + _height;
        float start = progress * scale - _height;
        float end   = start + _height;


        float yfrac = outCoord.y / height;
        float angle = (yfrac - start) / (end - start);
        float _width = cos(angle * 3.141592653589);

        if (yfrac < start) {
            dst = sampleLinear(to, outCoord());
        }

        else if (yfrac > end) {
            dst = sampleLinear(from, outCoord());
        }
        else {
```

```
        if (_width > 0.0) {

            float xstep  = (65536.0 / _width);
            float xval   = 0.0;
            float startx = (width - width * _width) / 2.0;
            float xcount = startx;

            float sampleWidth = (width * 65536.0) / (xstep);

            if (outCoord.x < startx)
                dst = float4(0.0,0.0,0.0,0.0);
            else if (outCoord.x < (startx + sampleWidth)) {
                float perc = (outCoord.x - startx) / sampleWidth;
                dst = sampleLinear(to, float2(width * perc, outCoord.y));
            }
            else
                dst = float4(0.0,0.0,0.0,0.0);
        }

        else if (_width < 0.0) {
            float xstep  = (65536.0 / -_width);
            float xval   = 0.0;
            float startx = (width + width * _width) / 2.0;
            float xcount = startx;
            float sampleWidth = (width * 65536.0) / (xstep);

            if (outCoord.x < startx)
                dst = float4(0.0,0.0,0.0,0.0);
            else if (outCoord.x < (startx + sampleWidth)) {
                float perc = (outCoord.x - startx) / sampleWidth;
                dst = sampleLinear(from, float2( width * perc, outCoord.y));
            }
            else
                dst = float4(0.0,0.0,0.0,0.0);
        }

        }

    }
}
```

You can also look at the source code for the CrossFade.pbk file in the frameworks\projects\flex4\src\spark\effects directory of the Flex source code.

Spark filter effects

Filters let you modify the visual appearance of your components not by modifying properties of the component but by applying a visual filter to the component. The filters are defined in the spark.filters. package. Common filters include the DropShadowFilter, GlowFilter, BlurFilter, and ShaderFilter.

Filters define a static change to the component where the properties of the filter are fixed unless you explicitly modify them at runtime. You can use the AnimateFilter effect to animate a filter at runtime. Unlike effects that animate properties of the target, the AnimateFilter effect animates properties of the filter applied to the target.

For example, if you apply the DropShadowFilter to a component, it has a static value for the filter properties such as `color`, `distance`, and `angle`. If you use the AnimateFilter effect to apply the filter to the target, you can animate these properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkAnimateDropShadowFilter.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:DropShadowFilter id="myDSF"
            color="0x0000FF"
            distance="5"
            angle="315"/>

        <s:AnimateFilter id="myFilter"
            target="{myB2}"
            repeatCount="0"
            duration="500"
            repeatBehavior="reverse"
            bitmapFilter="{new spark.filters.DropShadowFilter()}">
            <s:SimpleMotionPath property="color" valueFrom="0" valueTo="0x0000FF"/>
            <s:SimpleMotionPath property="distance" valueFrom="0" valueTo="10"/>
            <s:SimpleMotionPath property="angle" valueFrom="270" valueTo="360"/>
        </s:AnimateFilter>
    </fx:Declarations>

    <s:Button id="myB1"
        x="50" y="50"
        label="Show a DropShadowFilter"
        filters="{[myDSF]}"/>

    <s:Button id="myB2"
        x="50" y="95"
        label="Animate a DropShadowFilter"
        click="myFilter.end();myFilter.play();"/>
</s:Application>
```

In this example, the first button uses a static DropShadowFilter. The second button uses the AnimateFilter effect to animate the filter. Because the effect sets the `repeatCount` to 0, the effect plays continuously.

To use the AnimateFilter effect, you specify the filter to animate, and then use the SimpleMotionPath class to specify the properties of the filter to animate. In this example, the effect animate the `color`, `distance`, and `angle` properties.

Motion paths and keyframes

The Spark effect classes are designed to make it easy to specify the property to animate, and the starting and ending values of the property for the animation. These classes hide much of the underlying effects implementation to simplify the use of effects.

However, you might want to create an effect that animates a property over a set of values, rather than over just a starting and ending value. To create these effect, use keyframes and motion paths.

A *keyframe* defines the value of a property at a specific time during the effect. For example, you can create three keyframes that define the value of a property at the beginning of the effect, at the midpoint of the effect, and at the end of the effect. The effect animates the property change on the target from keyframe to keyframe over the effect duration.

If your effect has just two keyframes, use the Animate effect. The Animate effect takes a starting an ending value for the property, corresponding to two keyframes. For more information, see "The Animate effect" on page 1.

The collection of keyframes for an effect is called the effect's motion path. A *motion path* can define any number of keyframes. The effect then calculates the value of the property by interpolating between the values specified by two keyframes.

Applying an effect using keyframes and motion paths

Use the Keyframe class to define a keyframe. You typically define a keyframe by specifying the value of the property, and the time during the duration of the effect when the property has that value.

The MotionPath class contains a Vector of Keyframe objects. The MotionPath class also defines the name of the property that is modified by the effect, as the following example shows:

```
<?xml version="1.0"?>
<!-- behaviors\SparkKeyFrame.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <fx:Vector id="kf" type="spark.effects.animation.MotionPath">
            <s:MotionPath property="scaleX">
                <s:Keyframe time="250" value="0.5"/>
                <s:Keyframe time="500" value="1.0"/>
                <s:Keyframe time="750" value="0.5"/>
                <s:Keyframe time="1000" value="1.0"/>
                <s:Keyframe time="1250" value="0.5"/>
                <s:Keyframe time="1500" value="1.0"/>
            </s:MotionPath>
            <s:MotionPath property="scaleY">
                <s:Keyframe time="250" value="0.5"/>
                <s:Keyframe time="500" value="1.0"/>
                <s:Keyframe time="750" value="0.5"/>
                <s:Keyframe time="1000" value="1.0"/>
                <s:Keyframe time="1250" value="0.5"/>
                <s:Keyframe time="1500" value="1.0"/>
            </s:MotionPath>
        </fx:Vector>

        <s:Animate id="shrinkEffect"
            motionPaths="{kf}"
            target="{myImage}"/>
    </fx:Declarations>

    <mx:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"
        click="shrinkEffect.end();shrinkEffect.play();"/>
</s:Application>
```

Each MotionPath object defines the property of the target to animate, and contains six Keyframe objects. The Keyframe objects define the values of the `scaleX` and `scaleY` properties of the target at different times during the effect. The Animate class takes as the value of the `motionPaths` property a Vector of MotionPath objects.

In most situations, you do not have to work with keyframes and motion paths; the Spark effect classes handles the creating and use of keyframes internally. For example, the Animate class can take as the value of the `motionPaths` property a Vector of SimpleMotionPath objects. Each SimpleMotionPath object defines the property to animate, the starting property value, and the ending property value.

The SimpleMotionPath class is a subclass of the MotionPath class. When you create an instance of the SimpleMotionPath class, it creates two instances of the Keyframe class to represent the beginning and ending values of the property on the target. The effect then animates the property change on the target between the values specified by the two Keyframe objects.

Handling Spark effect events

Spark effects support all the general events, such as `effectStart`, `effectStop`, and `effectEnd`. For more information on handling these events, see Handling effect events.

In addition, the Spark effect classes support the following additional events:

- `effectRepeat` For any effect that is repeated more than once, dispatched when the effect begins a new repetition. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_UPDATE`.
- `effectUpdate` Dispatched every time the effect updates the target. The `type` property of the event object for this event is set to `EffectEvent.EFFECT_UPDATE`.

Using Spark easing classes

You can change the acceleration of an effect animation by using an easing class with an effect. With easing, you can create a more realistic rate of acceleration and deceleration. You can also use an easing class to create a bounce effect or control other types of motion.

Flex supplies the Spark easing classes in the spark.effects.easing package. This package includes classes for the most common types of easing, including Bounce, Linear, and Sine easing. For more information on using these classes, see the *Adobe Flex Language Reference*.

Easing class	Description
Linear	Defines three phases of animation: acceleration, uniform motion, and deceleration. Acceleration and deceleration occur at a constant rate. Use the `easeinFraction` property to specify the percentage of the total animation duration for acceleration. Use the `easeOutFraction` property to specify the percentage of the total animation duration for deceleration.
Power	Defines easing as consisting of two phases: the acceleration, or ease in phase, followed by the deceleration, or ease out phase. The rate of acceleration and deceleration is based on the `exponent` property. The higher the value of the `exponent` property, the greater the acceleration and deceleration. Use the `easeInFraction` property to specify the percentage of an animation accelerating.
Sine	Defines easing as consisting of two phases: the acceleration, or ease in phase, followed by the deceleration, or ease out phase. Use the `easeInFraction` property to specify the percentage of an animation accelerating.

The following example uses the Sine and Power easing classes with the Move effect:

```
<?xml version="1.0"?>
<!-- behaviorExamples\SparkResizeEasing.mxml -->
<s:Application xmlns:fx="http://ns.adobe.com/mxml/2009"
    xmlns:mx="library://ns.adobe.com/flex/halo"
    xmlns:s="library://ns.adobe.com/flex/spark">

    <fx:Declarations>
        <s:Sine id="sineEasing"
            easeInFraction="0.3"/>
        <s:Power id="powerEasing"
            exponent="4"/>
        <s:Move id="moveRight"
            target="{myImage}"
            xBy="500"
            duration="2000"
            easer="{powerEasing}"/>
        <s:Move id="moveLeft"
            target="{myImage}"
            xBy="-500"
            duration="2000"
            easer="{sineEasing}"/>
    </fx:Declarations>

    <mx:Image id="myImage"
        source="@Embed(source='assets/logo.jpg')"/>
    <s:Button label="Move Right"
         x="0" y="100"
        click="moveRight.end();moveRight.play();"/>
    <s:Button label="Move Left"
         x="0" y="125"
        click="moveLeft.end();moveLeft.play();"/>
</s:Application>
```

The Sine effect specifies a short acceleration phase for the move effect, followed by a longer deceleration. The Power effect specifies a rapid acceleration for the move.