

ActionScript 3.0 Cookbook

Display List

By Joey Lott, Darron Schall and Keith Peters

Publisher: O'Reilly Media, Inc.

Pub Date: 2006-10-11

ISBN: 9780596526955

Table Of Contents

Display List.....	3
Introduction	3
Adding an Item to the Display List	6
Removing an Item from the Display List	10
Moving Objects Forward and Backward	14
Creating Custom Visual Classes	17
Creating Simple Buttons	20
Loading External Images at Runtime	25
Loading and Interacting with External Movies	28
Creating Mouse Interactions	31
Dragging and Dropping Objects with the Mouse	35
Index.....	43

Copyright (©) 2004, 2005, 2006, 2007 O'Reilly Media.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472

The copyrights in individual elements of this work are owned by their respective publishers, authors or others, as the case may be, and the prior written permission of the copyright owner is required for reuse in any form or medium of any individual element.

O'Reilly books may be purchased for educational, business or sales promotional use. Online editions are also available for most titles (<http://safari.oreilly.com/>). For more information, contact our corporate/institutional sales department: (800)998.9938 or corporate@oreilly.com.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. Microsoft, the .NET logo, Virtual C#, Visual Basic, Visual Studio, and Windows are registered trademarks or trademarks of Microsoft Corporation. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of the African crowned crane and the topic of C# is a trademark of O'Reilly Media, Inc.

While reasonable care has been exercised in the preparation of this book, the publisher and the authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Introduction

The rendering model for ActionScript 3.0 and Flash Player 9 is radically different than in previous versions. Traditionally, the *MovieClip* was the focal point of the renderer. Every *.swf* movie contained a root *MovieClip* (commonly referred to as the *Stage*). The root *MovieClip* could contain child *MovieClips*, which could, in turn, contain more child *MovieClips*. The concept of *depths* was used to control the stacking order in which *MovieClips* were drawn (objects on higher depths appear “on top”). Methods such as *createEmptyMovieClip()*, *attachMovie()*, or *duplicateMovieClip()* were used to create *MovieClips*. Anytime a *MovieClip* was created, it was automatically added into the visual hierarchy and consequently drawn by the renderer. *MovieClips* weren’t able to move to different places within the hierarchy; instead, they first had to be destroyed and then recreated before they could be positioned elsewhere in the display.

The new renderer is still hierarchical, but not as rigid, and aims to simplify and optimize the rendering process. The new rendering model centers on the *display list* concept and focuses on the classes available in the *flash.display* package. The display list is a hierarchy that contains all visible objects in the *.swf* movie. Any object not on the display list is not drawn by the renderer. Each *.swf* movie contains exactly one display list, which is comprised of three types of elements:

The stage

The stage is the root of the display list hierarchy. Every movie has a single stage object that contains the entire object hierarchy of everything displaying on the screen. The stage is a container that typically contains only a single child, the main application class of the *.swf* movie. You can access the stage by referring to the stage property on any display object in the display list.

Display object containers

A display object container is an object that is capable of containing child display objects. The stage is a display object container. Other display object containers include *Sprite*, *MovieClip*, and *Shape*. When a display object container is removed from the display list, all its children are removed as well.

Display objects

A display object is a visual element. Some classes function as both display objects and display object containers, such as *MovieClip*, while other classes are only display objects, such as *TextField*. After a display object is created, it won’t appear on-screen until it is added into a display object container.

The hierarchy tree for a display list might look like something in *Figure 6-1*. The stage is at the very top of the hierarchy, with display object containers as branches and display objects as leaves. The items at the top of the diagram are visually underneath the items at the bottom.

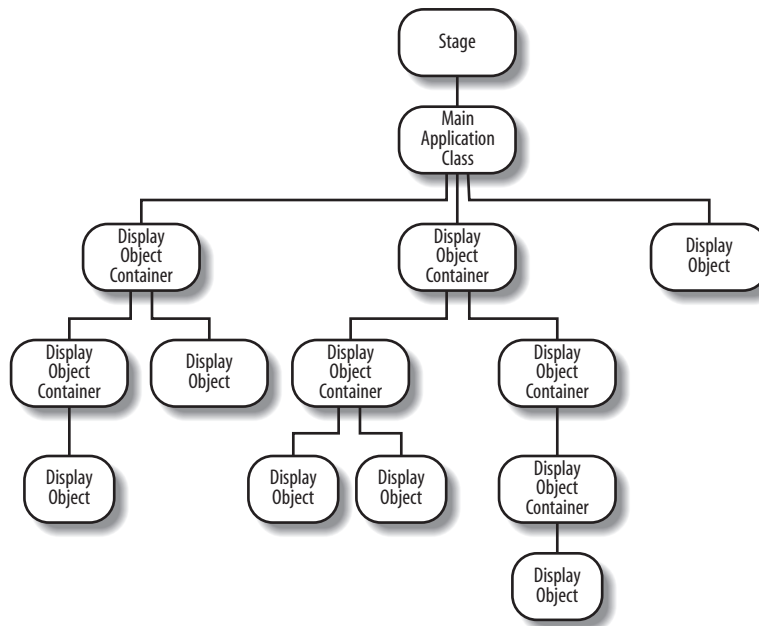


Figure 6-1.
An example display list hierarchy

Transitioning to the display list realizes a number of benefits for programmers over working with previous versions of the Flash Player; these include:

Increased performance

The display list contains multiple visual classes besides just *MovieClip*. Classes such as *Sprite* can be used to reduce the memory requirements when a timeline isn't necessary. Additionally, a *Shape* can be used to draw into rather than relying on a full *MovieClip* instance. By having these lighter-weight classes and using them when possible, precious memory and processor resources can be saved, resulting in improved overall performance.

Easier depth management

The hierarchy of the display functions as depth management under the new display list model. In the previous model, methods such as *getNextHighestDepth()* were used to create *MovieClips* on the correct depth, and *swapDepths()* was used to control the visual stacking order. Depth management was cumbersome and tedious before and often required extremely careful programming. It was a fact of life that just had to be dealt with because it was so intertwined with the language. The new display list model handles depth almost automatically now, making depth management almost a thing of the past.

Less rigid structure

The previous model featured a fairly inflexible and rigid hierarchy. To change the hierarchy, *MovieClips* had to be destroyed and recreated at a new location. This was a time-consuming and expensive operation, and often was painfully slow. The new display list model is much more flexible—entire portions of the hierarchy tree can be moved via the new *reparenting* functionality (discussed in *Recipe 6.1*), without suffering the performance penalties of creating and destroying elements as before.

Easier creation of visual items

The display list rendering model makes creating display objects easier, especially when creating instances of custom visual classes. The previous model required extending *MovieClip*,

combined with using special linkage that associated a library item with the ActionScript class. Then `attachMovie()` had to be used to actually create an instance of the custom class. Under the display list model, you extend one of the many display object classes, but you use the new keyword to create instances of custom visual classes, which is much more intuitive, easier, and cleaner. See *Recipe 6.4* for details.

As noted earlier, the `flash.display` package contains the core classes for the display list model. The old model focused on the `MovieClip` class, but the display list revolves around the `DisplayObject` class and its various subclasses. *Figure 6-2* illustrates the display list class hierarchy.

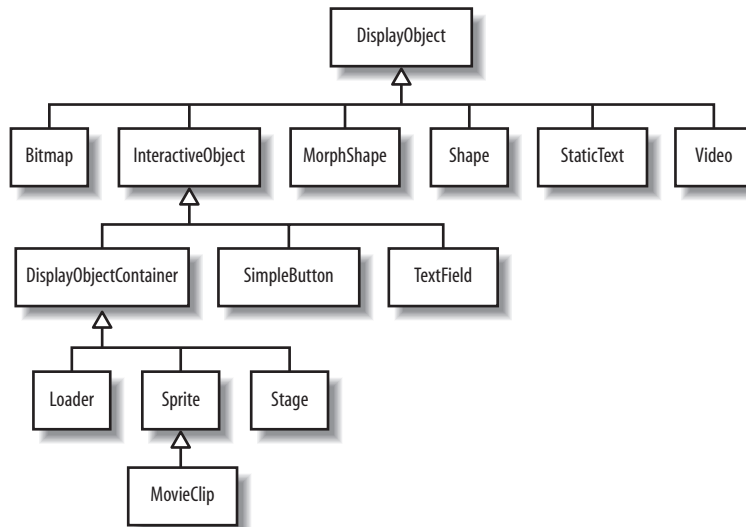


Figure 6-2.
The display list class hierarchy

Each one of the core classes is designed to serve a specific purpose. By having more than just `MovieClip` available, the display list offers more flexibility to programmers than the previous model. The more commonly used display classes are listed in *Table 6-1*.

Table .
Commonly used display classes

Display class	Description
<code>DisplayObject</code>	The base class of all display list classes. <code>DisplayObject</code> defines properties and methods common to all display classes. The <code>DisplayObject</code> class is not meant to be instantiated directly.
<code>Bitmap</code>	The <code>Bitmap</code> class allows for the creation and manipulation of images via the <code>BitmapData</code> methods; it is described in Chapter 8.
<code>Shape</code>	The <code>Shape</code> class contains a <code>graphics</code> property that allows for drawing using lines, fills, circles, rectangles, etc. Chapter 7 has more information.
<code>Sprite</code>	<code>Sprites</code> are similar to shapes, but can contain child display objects such as text and video. A <code>Sprite</code> can be thought of as a <code>MovieClip</code> without a timeline.
<code>MovieClip</code>	<code>MovieClip</code> is the familiar class with a timeline and methods for controlling the playhead. Because <code>MovieClip</code> is a subclass of <code>Sprite</code> , you can draw inside of it, and it can contain child display objects as well.

Display class	Description
<i>Video</i>	The <i>Video</i> class lives in the <i>flash.media</i> package, but is also a subclass of <i>DisplayObject</i> . <i>Video</i> instances are used to play video, as described in Chapter 16.
<i>TextField</i>	The <i>TextField</i> class, found in the <i>flash.text</i> package, allows the creation of dynamic and input text fields. See Chapter 9 for more information.
<i>Loader</i>	<i>Loader</i> instances are used to load in external visual assets, such as other <i>.swf</i> movies or image files.

If you've worked with Flash in the past, transitioning to the display list model is going to take some time. Old habits are hard to break, and display list programming has a lot of depth. However, once you get into the swing of things you'll see that the time it takes to learn the new model is well worth it. The display list dramatically changes Flash display programming for the better.

Adding an Item to the Display List

Problem

You want to add a new display object to the display list so it appears on-screen.

Solution

Use the *addChild()* and *addChildAt()* methods from the *DisplayObjectContainer* class.

Discussion

The Flash Player is composed of two main pieces that function together to form a cohesive unit, the ActionScript Virtual Machine (AVM) and the Rendering Engine. The AVM is responsible for executing ActionScript code, and the Rendering Engine is what draws objects on-screen. Because the Flash Player is composed of these two main pieces, drawing an object on the screen is a two-step process:

1. The display object needs to be created in the ActionScript engine.
2. The display object is then created in the rendering engine and drawn on-screen.

The first step is done by using the `new` operator to create an instance of the display object. Any object that is going to be added to the display list must be either a direct or indirect subclass of *DisplayObject*, such as *Sprite*, *MovieClip*, *TextField*, or a custom class you create (according to *Recipe 6.4*). To create a *TextField* instance you would use the following code:

```
var hello:TextField = new TextField();
```

The preceding line of code creates a *TextField* display object in the AVM, but the object is not drawn on the screen yet because the object doesn't exist in the Rendering Engine. To create the object in the Rendering Engine, the object needs to be added to the display list hierarchy. This can be done by calling the *addChild()* or *addChildAt()* method from a *DisplayObjectContainer* instance that is itself already on the display list hierarchy.

The *addChild()* method takes a single parameter—the display object that the container should add as a child. The following code is a complete example that demonstrates how to create an object in the AVM and then create the object in the Rendering Engine by adding it to the display list:

```

package {
    import flash.display.DisplayObjectContainer;
    import flash.display.Sprite;
    import flash.text.TextField;

    public class DisplayListExample extends Sprite {
        public function DisplayListExample() {
            // Create a display object in the actionscript engine
            var hello:TextField = new TextField();
            hello.text = "hello";

            // Create the display object in the rendering engine
            // by adding it to the display list so that the
            // text field is drawn on the screen
            addChild( hello );
        }
    }
}

```

Here the *DisplayListExample* class is the main application class for the *.swf* movie and it extends the *Sprite* class. Because of the class hierarchy described in *Figure 6-2*, the *DisplayListExample* is therefore an indirect subclass of *DisplayObjectContainer* and is capable of having multiple *DisplayObject* instances as children. This allows for the use of the *addChild()* method to add a display object as a child in the container.

A *TextField* display object is created in the *DisplayListExample* constructor, which creates the object inside the AVM. At this point, the object won't appear on-screen because the Rendering Engine doesn't know about it yet. It is only after the object is added to the display list—via the *addChild()* method call—that the *TextField* is displayed.



The *addChild()* and *addChildAt()* methods only add display objects as children of display object containers. They do not necessarily add display objects to the display list. Children are added to the display list only if the container they are being added to is on the display list as well.

The following code snippet demonstrates how the *addChild()* method doesn't guarantee that a display object is added to the display list. A container is created with some text inside of it, but because the container is not on the display list, the text is not visible:

```

// Create a text field to display some text
var hello:TextField = new TextField();
hello.text = "hello";

// Create a container to hold the TextField
var container:Sprite = new Sprite();
// Add the TextField as a child of the container
container.addChild( hello );

```

To make the text display on-screen, the text container needs to be added to the display list. This is accomplished by referencing a display object container already on the display list, such as *root* or *stage*, calling *addChild()*, and passing in the text container display object. Both *root* and *stage* are properties of the *DisplayObject* class:

```
// Cast the special root reference as a container and add the
// container that holds the text so it appears on-screen
DisplayObjectContainer( root ).addChild( container );
```

Display object containers are capable of holding multiple children. The container keeps a list of children internally, and the order of the children in the list determines the visual stacking order on-screen. Each child has a specific position in the list as specified by an integer index value, much like an array. Position 0 is the very bottom of the list and is drawn underneath the child at position 1, which is, in turn, drawn underneath the child at position 2, etc. This is similar to the depth concept you may be familiar with if you have prior Flash experience, but it's easier to manage. There are no gaps between position numbers. That is, there can never be children at position 0 and position 2 with an opening at position 1.

When a new child display object is added via the *addChild()* method, it is drawn visually on top of all of the other children in the container because *addChild()* places the child at the front of the children list, giving it the next highest position index. To add a child and specify where it belongs in the visual stacking order at the same time, use the *addChildAt()* method.

The *addChildAt()* method takes two parameters: the child display object to add, and the position in the stacking order that the child should use. Specifying a position of 0 causes the child to be added to the very bottom of the list and makes the child appear (visually) underneath all of the other children. If there was previously a child at the position specified, all of the children at and above the position index are shifted forward by one to allow the child to be inserted. Specifying an invalid position value, such as a negative value or a number greater than the number of children in the container, generates a *RangeError* and causes the child to not be added.

The following example creates three different colored circles. The red and blue circles are added with the *addChild()* method, making the blue circle appear on top because it was added after the red circle. After the two calls to *addChild()*, the red circle is at position 0 and the blue circle is at position 1. The green circle is then inserted between the two with the *addChildAt()* method, specifying position 1 as the location in the list. The blue circle, previously at position 1, is shifted to position 2 and the green circle is inserted at position 1 in its place. The final result is the red circle at position 0 being drawn underneath the green circle at position 1, and the green circle being drawn underneath blue circle at position 2.

```
package {
    import flash.display.*;
    public class CircleExample extends Sprite {
        public function CircleExample() {
            // Create three different colored circles and
            // change their coordinates so they are staggered
            // and aren't all located at (0,0).
            var red:Shape = createCircle( 0xFF0000, 10 );
            red.x = 10;
            red.y = 20;
            var green:Shape = createCircle( 0x00FF00, 10 );
            green.x = 15;
            green.y = 25;
            var blue:Shape = createCircle( 0x0000FF, 10 );
            blue.x = 20;
            blue.y = 20;

            // First add the red circle, then add the blue circle (so blue
            // is drawn on top of red)
```

```

addChild( red );
addChild( blue );

// Place the green circle between the red and blue circles
addChildAt( green, 1 );
}

// Helper function to create a circle shape with a given color
// and radius
public function createCircle( color:uint, radius:Number ):Shape {
    var shape:Shape = new Shape();
    shape.graphics.beginFill( color );
    shape.graphics.drawCircle( 0, 0, radius );
    shape.graphics.endFill();
    return shape;
}
}
}

```

So far we've only talked about adding new items to the display list, but what happens when *addChild()* is used on a child that is already on the display list, as a child of another container? This is the concept of *reparenting*. The child is removed from the container that it currently resides in and is placed in the container that it is being added to.



When you reparent a display object, it is not necessary to remove it first. The *addChild()* method takes care of that for you.

The following example shows reparenting in action. A container is created to display red, green, and blue circles that are all added as children, and the container is added to the display list. Another container is created and added to the display list as well, and then the red circle is moved from the first container to the second. Because the second container is visually above the first container in the display list, all children in the second container appear on top of the children in the first container. This makes the red circle display on top of the blue and green ones. The red circle was reparented from the first container to the second simply by calling the *addChild()* method.

```

package {
    import flash.display.*;
    public class DisplayListExample extends Sprite {
        public function DisplayListExample() {
            // Create three different colored circles and
            // change their coordinates so they are staggered
            // and aren't all located at (0,0).
            var red:Shape = createCircle( 0xFF0000, 10 );
            red.x = 10;
            red.y = 20;
            var green:Shape = createCircle( 0x00FF00, 10 );
            green.x = 15;
            green.y = 25;
            var blue:Shape = createCircle( 0x0000FF, 10 );
            blue.x = 20;
            blue.y = 20;
        }
    }
}

```

```

// Create a container to hold the three circles, and add the
// circles to the container
var container1:Sprite = new Sprite();
container1.addChild( red );
container1.addChild( green );
container1.addChild( blue );

// Add the container to the display list
addChild( container1 );

// Create a second container and add it the display list
var container2:Sprite = new Sprite();
addChild( container2 );

// Reparent the red circle from container 1 to container 2,
// which has the net effect of the red circle being drawn
// on top of the green and blue ones.
container2.addChild( red );
}

// Helper function to create a circle shape with a given color
// and radius
public function createCircle( color:uint, radius:Number ):Shape {
    var shape:Shape = new Shape();
    shape.graphics.beginFill( color );
    shape.graphics.drawCircle( 0, 0, radius );
    shape.graphics.endFill();
    return shape;
}
}
}
}

```

See Also

Recipes 6.2 and 6.4

Removing an Item from the Display List

Problem

You want to remove an item from the display list and consequently remove it from the screen.

Solution

Use the *removeChild()* and *removeChildAt()* methods from the *DisplayObjectContainer* class.

Discussion

Recipe 6.1 demonstrates how to add display objects to the display list using the *addChild()* and *addChildAt()* methods. To achieve the opposite effect and remove a child via one of these methods, use either the *removeChild()* or *removeChildAt()* method.

The *removeChild()* method takes a single parameter, which is a reference to the display object that should be removed from the container. If an object is supposed to be removed and it isn't a child of the container, an *ArgumentError* is thrown:

```
package {
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.events.MouseEvent;

    public class RemoveChildExample extends Sprite {

        // Create a local variable to store a reference
        // to the TextField so that we can remove it later
        private var _label:TextField;

        public function RemoveChildExample() {
            _label = new TextField();
            _label.text = "Some Text";

            // Add the hello TextField to the display list
            addChild( _label );

            // When the mouse is clicked anywhere on the stage,
            // remove the label
            stage.addEventListener( MouseEvent.CLICK, removeLabel );
        }

        // Removes the label from this container's display list
        public function removeLabel( event:MouseEvent ):void {
            removeChild( _label );
        }
    }
}
```

The preceding code example creates a local variable *label* that stores a reference to the *TextField* within the class itself. This is a necessary step because the *removeChild()* method must be passed a reference to the display object to remove, so *label* is used to store the reference for later. If *label* were not available, extra work would be required to get a reference to the *TextField* to remove it, or the *removeChildAt()* method could be used instead.

In the case when you do not have a reference to the display object you want to remove, you can use the *removeChildAt()* method. Similar to the *addChildAt()* method, the *removeChildAt()* method takes a single parameter—the index in the container's list of child display objects to remove. The possible values for the index can range from 0 to *numChildren* - 1. If an invalid index is specified, such as a negative value or a value greater than the number of children in the container, a *RangeError* is thrown and no child is removed. Adopting the previous code snippet to use *removeChildAt()* instead yields the following:

```
package {
    import flash.display.Sprite;
    import flash.text.TextField;
    import flash.events.MouseEvent;

    public class DisplayListExample extends Sprite {
```


In the second case, removing the very last child from the container does not cause any children to adjust their positions. Only children with a higher position value than what is removed are shifted down by one. The last child has the highest position value within the container; therefore, no other children need to have their positions adjusted.

The first approach is the one we recommend using, and has been implemented in the *ascb.util.DisplayObjectUtilities* class:

```
package ascb.util {
    import flash.display.*;
    public class DisplayObjectUtilities {
        // Remove all of the children in a container
        public static function removeAllChildren(
            container: DisplayObjectContainer ):void {

            // Because the numChildren value changes after every time
            // you remove a child, save the original value so you can
            // count correctly
            var count:int = container.numChildren;

            // Loop over the children in the container and remove them
            for ( var i:int = 0; i < count; i++ ) {
                container.removeChildAt( 0 );
            }
        }
    }
}
```

Using the *DisplayObjectUtilities.removeAllChildren()* method is relatively straight- forward, as shown here:

```
package {
    import flash.display.*;
    import ascb.util.DisplayObjectUtilities;

    public class DisplayListExample extends Sprite {

        public function DisplayListExample() {

            // Add some empty sprites
            addChild( new Sprite() );
            addChild( new Sprite() );

            // Remove all children from this container
            DisplayObjectUtilities.removeAllChildren( this );

            // Demonstrate that all children have been removed - displays: 0
            trace( numChildren );
        }
    }
}
```

See Also

Recipe 6.1

Moving Objects Forward and Backward

Problem

You want to change the order in which objects are drawn on-screen, moving them either in front of or behind other display objects.

Solution

Use the `setChildIndex()` method of the `DisplayObjectContainer` class to change the position of a particular item. Use the `getChildIndex()` and `getChildAt()` methods to query siblings of the item so the item can be positioned properly relative to them.

Discussion

Recipes 6.1 and 6.2 introduced how the display list model deals with the visual stacking order (depth). Essentially, every `DisplayObjectContainer` instance has a list of children, and the order of the children in this list determines the order in which child display objects are drawn inside of the container. The children are each given a position index, ranging from 0 to `numChildren - 1`, much like an array. The child at position 0 is drawn on the bottom, underneath the child at position 1, etc. There are no empty position values in the list; if there are three children, the children will always have index values of 0, 1, and 2 (and not, say, 0, 1, and 6).

The `setChildIndex()` method is provided by `DisplayObjectContainer` to reorder the children inside the container. It takes two parameters: a reference to the child to be moved and the child's new position in the container. The index position specified must be a valid value. Negative values or values too large will generate a `RangeError` and the function won't execute properly.

The following example creates three colored circles, with the blue one being drawn on top. The `setChildIndex()` method is used to move the blue circle underneath the two other circles, changing its position from 2 to 0 in the container. The positions of the other children are adjusted accordingly; red is moved to 1 and green is moved to 2:

```
package {
    import flash.display.*;
    public class SetChildIndexExample extends Sprite {
        public function SetChildIndexExample() {
            // Create three different colored circles and
            // change their coordinates so they are staggered
            // and aren't all located at (0,0).
            var red:Shape = createCircle( 0xFF0000, 10 );
            red.x = 10;
            red.y = 20;
            var green:Shape = createCircle( 0x00FF00, 10 );
            green.x = 15;
            green.y = 25;
            var blue:Shape = createCircle( 0x0000FF, 10 );
            blue.x = 20;
```

```

    blue.y = 20;

    // Add the circles, red has index 0, green 1, and blue 2
    addChild( red );
    addChild( green );
    addChild( blue );

    // Move the blue circle underneath the others by placing
    // it at the very bottom of the list, at index 0
    setChildIndex( blue, 0 );
}

// Helper function to create a circle shape with a given color
// and radius
public function createCircle( color:uint, radius:Number ):Shape {
    var shape:Shape = new Shape( );
    shape.graphics.beginFill( color );
    shape.graphics.drawCircle( 0, 0, radius );
    shape.graphics.endFill( );
    return shape;
}
}
}
}

```

One of the requirements for *setChildIndex()* is that you know the index value you want to give to a specific child. When you're sending a child to the back, you use 0 as the index. When you want to bring a child to the very front, you specify `numChildren - 1` as the index. But what if you want to move a child underneath another child?

For example, suppose you have two circles—one green and one blue—and you don't know their positions ahead of time. You want to move the blue circle behind the green one, but *setChildIndex()* requires an integer value for the new position. There are no *setChildAbove* or *setChildBelow* methods, so the solution is to use the *getChildIndex()* method to retrieve the index of a child, and then use that index to change the position of the other child. The *getChildIndex()* method takes a display object as a parameter and returns the index of the display object in the container. If the display object passed in is not a child of the container, an *ArgumentError* is thrown.

The following example creates two circles—one green and one blue—and uses *getChildIndex()* on the green circle so the blue circle can be moved beneath it. By setting the blue circle to the index that the green circle has, the blue circle takes over the position and the green circle moves to the next higher position because blue had a higher position initially:

```

package {
    import flash.display.*;
    public class GetChildIndexExample extends Sprite {
        public function GetChildIndexExample() {
            // Create two different sized circles
            var green:Shape = createCircle( 0x00FF00, 10 );
            green.x = 25;
            green.y = 25;
            var blue:Shape = createCircle( 0x0000FF, 20 );
            blue.x = 25;
            blue.y = 25;

```

```

// Add the circles to this container
addChild( green );
addChild( blue );

// Move the blue circle underneath the green circle. First
// the index of the green circle is retrieved, and then the
// blue circle is set to that index.
setChildIndex( blue, getChildIndex( green ) );
}

// Helper function to create a circle shape with a given color
// and radius
public function createCircle( color:uint, radius:Number ):Shape {
    var shape:Shape = new Shape();
    shape.graphics.beginFill( color );
    shape.graphics.drawCircle( 0, 0, radius );
    shape.graphics.endFill();
    return shape;
}
}
}

```

When a child is moved to an index lower than the one it currently has, all children from the target index up to the one just before the child index will have their indexes increased by 1 and the child is assigned to the target index. When a child is moved to a higher index, all children from the one just above the child index up to and including the target index are moved down by 1, and the child is assigned the target index value.

In general, if object *a* is above object *b*, the following code to moves *a* directly below *b*:

```
setChildIndex( a, getChildIndex( b ) );
```

Conversely, if object *a* is below object *b*, the preceding code moves *a* directly above *b*.

So far, we've always been moving around children that we've had a reference to. For example, the *blue* variable referenced the display object for the blue circle, and we were able to use this variable to change the index of the blue circle. What happens when you don't have a reference to the object you want to move, and the *blue* variable doesn't exist? The *setChildIndex()* method requires a reference to the object as its first parameter, so you'll need to get the reference somehow if it isn't available with a regular variable. The solution is to use the *getChildAt()* method.

The *getChildAt()* method takes a single argument, an index in the container's children list, and returns a reference to the display object located at that index. If the specified index isn't a valid index in the list, a *RangeError* is thrown.

The following example creates several circles of various colors and sizes and places them at various locations on the screen. Every time the mouse is pressed, the child at the very bottom is placed on top of all of the others:

```

package {
    import flash.display.*;
    import flash.events.*;
    public class GetChildAtExample extends Sprite {
        public function GetChildAtExample() {
            // Define a list of colors to use
            var color:Array = [ 0xFF0000, 0x990000, 0x660000, 0x00FF00,

```

```

        0x009900, 0x006600, 0x0000FF, 0x000099,
        0x000066, 0xCCCCCC ];
// Create 10 circles and line them up diagonally
for ( var i:int = 0; i < 10; i++ ) {
    var circle:Shape = createCircle( color[i], 10 );
    circle.x = i;
    circle.y = i + 10; // the + 10 adds padding from the top

    addChild( circle );
}

stage.addEventListener( MouseEvent.CLICK, updateDisplay );
}

// Move the circle at the bottom to the very top
public function updateDisplay( event:MouseEvent ):void {
    // getChildAt(0) returns the display object on the
    // very bottom, which then gets moved to the top
    // by specifying index numChildren - 1 in setChildIndex
    setChildIndex( getChildAt(0), numChildren - 1 );
}

// Helper function to create a circle shape with a given color
// and radius
public function createCircle( color:uint, radius:Number ):Shape {
    var shape:Shape = new Shape();
    shape.graphics.beginFill( color );
    shape.graphics.drawCircle( 0, 0, radius );
    shape.graphics.endFill();
    return shape;
}
}
}
}

```

See Also

Recipes [6.1](#) and [6.2](#)

Creating Custom Visual Classes

Problem

You want to create a new type of *DisplayObject*.

Solution

Create a new class that extends *DisplayObject* or one of its subclasses so it can be added into a display object container via *addChild()* or *addChildAt()*.

Discussion

Among the benefits of moving toward the display list model is the ease of creating new visual classes. In the past, it was possible to extend *MovieClip* to create custom visuals, but there always had to be a *MovieClip* symbol in the library linked to the ActionScript class to create an on-screen instance via *attachMovie()*. Creating a custom visual could never be done entirely in ActionScript. With the display list model, the process has been simplified, allowing you to do everything in pure ActionScript code in a much more intuitive manner.

In the display list model, as discussed in the introduction of this chapter, there are many more display classes available besides just *MovieClip*. Before you create your custom visual, you need to decide which type it is going to be. If you're just creating a custom shape, you'll want to extend the *Shape* class. If you're creating a custom button, you'll probably want to extend *SimpleButton*. If you want to create a container to hold other display objects, *Sprite* is a good choice if you don't require the use of a timeline. If you need a timeline, you'll need to subclass *MovieClip*.

All of the available display object classes are tailored for specific purposes. It's best to decide what purpose your own visual class is going to serve, and then choose the appropriate parent class based on that. By choosing the parent class carefully you optimize size and resource overhead. For example, a simple *Circle* class doesn't need to subclass *MovieClip* because it doesn't need the timeline. The *Shape* class is the better choice in this case because it's the most lightweight option that appropriately fits the concept of a circle.

Once the base class has been decided, all you need to do is write the code for the class. Let's follow through with the circle example and create a new *Circle* class that extends the *Shape* display object. In a new ActionScript file named *Circle.as*, enter the following code:

```
package {
    import flash.display.Shape;

    /* The Circle class is a custom visual class */
    public class Circle extends Shape {

        // Local variables to store the circle properties
        private var _color:uint;
        private var _radius:Number;

        /*
         * Constructor: called when a Circle is created. The default
         * color is black, and the default radius is 10.
         */
        public function Circle( color:uint = 0x000000, radius:Number = 10 ) {
            // Save the color and radius values
            _color = color;
            _radius = radius;

            // When the circle is created, automatically draw it
            draw();
        }

        /*
         * Draws the circle based on the color and radius values
         */
    }
}
```

```

private function draw():void {
    graphics.beginFill( _color );
    graphics.drawCircle( 0, 0, _radius );
    graphics.endFill( );
}
}
}

```

The preceding code defines a new *Circle* display object. When a *Circle* instance is created, you can specify both a color and a radius in the constructor. Methods from the Drawing API (discussed in *Recipe 7.3*) are used to create the body of the circle with the `graphics` property, which is inherited from the superclass *Shape*.



It is always a good idea to separate all drawing code into a separate *draw()* method. The constructor for *Circle* does not draw the circle directly, but it calls the *draw()* method to create the visual elements.

All that is left to do is create new instances of our custom *Circle* class and add them to the display list with *addChild()* or *addChildAt()* so they appear on-screen. To create new instances of the class, use the `new` keyword. The following code example creates a few *Circle* instances and displays them on the screen:

```

package {
    import flash.display.Sprite;
    public class UsingCircleExample extends Sprite {
        public function UsingCircleExample() {
            // Create some circles with the Circle class and
            // change their coordinates so they are staggered
            // and aren't all located at (0,0).
            var red:Circle = new Circle( 0xFF0000, 10 );
            red.x = 10;
            red.y = 20;
            var green:Circle = new Circle( 0x00FF00, 10 );
            green.x = 15;
            green.y = 25;
            var blue:Circle = new Circle( 0x0000FF, 10 );
            blue.x = 20;
            blue.y = 20;

            // Add the circles to the display list
            addChild( red );
            addChild( green );
            addChild( blue );
        }
    }
}

```

See Also

Recipes *6.1* and *7.3*

Creating Simple Buttons

Problem

You want to create an interactive button that enables a user to click and perform an action, such as submitting a form or calculating a total.

Solution

Create an instance of the *SimpleButton* class and create display objects for `upState`, `downState`, `overState`, and `hitTestState`. Alternatively, create a subclass of *SimpleButton* that describes your desired button behavior.

Use the `click` event to invoke a method whenever the user presses the button.

Discussion

The display list model provides an easy way to create buttons through the *SimpleButton* class. The *SimpleButton* class allows a user to interact with the display object using their mouse, and makes it easy for you to define that interaction through various button states. The possible button states, listed here, are available as properties of the *SimpleButton* class:

upState

A display object for the default “up” state of the button. The “up” state is shown whenever the mouse is not over the button.

overState

A display object that determines what the button looks like when the mouse moves over the button. When the mouse leaves the button area, the button moves back to the “up” state.

downState

A display object that’s shown when the button is pressed (or clicked) “down”. When the button is in the “over” state, the “down” state displays when the user presses the left mouse button.

hitTestState

A display object that defines a button’s bounds. When the mouse moves inside of the button’s hit area, the button enters the “over” state. The `hitTestState` is typically set to the same display object as the `upState`. The `hitTestState` is never actually displayed on-screen; it is only used for mouse tracking purposes.

A button’s state is handled by the *SimpleButton* class, and is based on movement of the user’s mouse. You don’t have control over setting the internal button state (up, down, or over). Rather, you can only control which display object should appear when the button is in a particular state. By setting the state properties to different display objects, you can provide feedback to the user as they interact with the button using their mouse.

The following example creates a new *SimpleButton* instance and defines button states using the four state properties defined earlier. Because each state property of the button needs to be set to a *DisplayObject* instance, the helper method `createCircle()` is used to create different colored circle shapes to be used for the various button states:

```
package {  
    import flash.display.*;
```

```

import flash.events.*;

public class SimpleButtonDemo extends Sprite {
    public function SimpleButtonDemo() {
        // Create a simple button and configure its location
        var button:SimpleButton = new SimpleButton();
        button.x = 20;
        button.y = 20;

        // Create the different states of the button, using the
        // helper method to create different colors circles
        button.upState = createCircle( 0x00FF00, 15 );
        button.overState = createCircle( 0xFFFFFF, 16 );
        button.downState = createCircle( 0xCCCCCC, 15 );
        button.hitTestState = button.upState;

        // Add an event listener for the click event to be notified
        // when the user clicks the mouse on the button
        button.addEventListener( MouseEvent.CLICK, handleClick );

        // Finally, add the button to the display list
        addChild( button );
    }

    // Helper function to create a circle shape with a given color
    // and radius
    private function createCircle( color:uint, radius:Number ):Shape {
        var circle:Shape = new Shape();
        circle.graphics.lineStyle( 1, 0x000000 );
        circle.graphics.beginFill( color );
        circle.graphics.drawCircle( 0, 0, radius );
        circle.graphics.endFill();
        return circle;
    }

    // Event handler invoked whenever the user presses the button
    private function handleClick( event:MouseEvent ):void {
        trace( "Mouse clicked on the button" );
    }
}

```

After running the preceding code block, a green circle appears in the movie. When you move your mouse over the green circle, a slightly bigger white circle appears as a rollover. When you click the white circle, it turns into a slightly smaller gray circle. This visual effect is created by the *SimpleButton* instance changing its state based on the actions of your mouse, switching between the display objects defined in the four state properties.

To listen for events from the `button` instance, the *addEventListener()* method is used as described in *Recipe 1.5*. The click event, specified with `MouseEvent.CLICK`, is handled in the preceding code by the *handleClick()* method. Anytime the user clicks the button instance, the *handleClick()* method is invoked, allowing certain actions to take place. In this simple example, a short message (“Mouse clicked on the button”) is displayed to the console.

The `hitTestState` property is perhaps the most interesting of the button's state properties. You'll notice that the preceding code sets the `hitTestState` to be the same display object that defines the `upState`. It is typical to do this because buttons should be activated when the user's mouse is within the bounds of the `upState` display object.



Although the `hitTestState` is never visible, failure to set the `hitTestState` to a display object results in a button that can't be interacted with. Always remember to set the `hitTestState` of your *SimpleButton*, even if you simply set it to the same value as `upState`.

The `hitTestState` can be set to any display when you'd like to control the active bounds of a button. To create a larger hit area for the button, try modifying the previous code segment to set the `hitTestState` via this line:

```
button.hitTestState = createCircle( 0x000000, 50 );
```

When running this example, you'll notice that the button displays the white "over" circle before the mouse even enters the area of the green circle, contrary to previous behavior. This is because the hit area was increased to a circle of radius 50, giving a larger target area for the user's mouse. You might also notice that black (0x000000) was specified as the color for the hit area circle. This was done on purpose to reinforce the fact that the hit area display object is never visible.

An alternate approach to creating a *SimpleButton* and setting the four display states for every button is to create a subclass of *SimpleButton* that defines your button's visual style and creates instances of that instead. *Recipe 6.4* describes how to create new visual classes. Following this technique, you can create your own version of a *SimpleButton*, making it easier to add multiple buttons to your movie.

The following code creates a new *RectangleButton* class. The *RectangleButton* class defines the behavior for a special type of *SimpleButton* that draws a green rectangle with some text on top of it:

```
package {
    import flash.display.*
    import flash.text.*;
    import flash.filters.DropShadowFilter;

    public class RectangleButton extends SimpleButton {
        // The text to appear on the button
        private var _text:String;
        // Save the width and height of the rectangle
        private var _width:Number;
        private var _height:Number;

        public function RectangleButton( text:String, width:Number, height:Number ) {
            // Save the values to use them to create the button states
            _text = text;
            _width = width;
            _height = height;

            // Create the button states based on width, height, and text value
            upState = createUpState( );
            overState = createOverState( );
            downState = createDownState( );
        }
    }
}
```

```

    hitTestState = upState;
}

// Create the display object for the button's up state
private function createUpState():Sprite {
    var sprite:Sprite = new Sprite();

    var background:Shape = createdColoredRectangle( 0x33FF66 );
    var textField:TextField = createTextField( false );

    sprite.addChild( background );
    sprite.addChild( textField );

    return sprite;
}

// Create the display object for the button's up state
private function createOverState():Sprite {
    var sprite:Sprite = new Sprite();

    var background:Shape = createdColoredRectangle( 0x70FF94 );
    var textField:TextField = createTextField( false );

    sprite.addChild( background );
    sprite.addChild( textField );

    return sprite;
}

// Create the display object for the button's down state
private function createDownState():Sprite {
    var sprite:Sprite = new Sprite();

    var background:Shape = createdColoredRectangle( 0xCCCCCC );
    var textField:TextField = createTextField( true );

    sprite.addChild( background );
    sprite.addChild( textField );

    return sprite;
}

// Create a rounded rectangle with a specific fill color
private function createdColoredRectangle( color:uint ):Shape {
    var rect:Shape = new Shape();
    rect.graphics.lineStyle( 1, 0x000000 );
    rect.graphics.beginFill( color );
    rect.graphics.drawRoundRect( 0, 0, _width, _height, 15 );
    rect.graphics.endFill();
    rect.filters = [ new DropShadowFilter( 2 ) ];
    return rect;
}

// Create the text field to display the text of the button

```

```

private function createTextField( downState:Boolean ):TextField {
    var textField:TextField = new TextField();
    textField.text = _text;
    textField.width = _width;

    // Center the text horizontally
    var format:TextFormat = new TextFormat();
    format.align = TextFormatAlign.CENTER;
    textField.setTextFormat( format );

    // Center the text vertically
    textField.y = ( _height - textField.textHeight ) / 2;
    textField.y -= 2; // Subtract 2 pixels to adjust for offset

    // The down state places the text down and to the right
    // further than the other states
    if ( downState ) {
        textField.x += 1;
        textField.y += 1;
    }

    return textField;
}
}
}
}

```

Because all of the button drawing is encapsulated into its own reusable class, creating new button instances is much easier. Instead of having to create a *SimpleButton* and define the button states by hand for each instance, you can simply create a new *RectangleButton* instance and add that to the display list.

The following example shows how to create three different rectangular buttons using this new instance:

```

package {
    import flash.display.*;
    public class SimpleButtonDemo extends Sprite {
        public function SimpleButtonDemo() {

            // Create three rectangular buttons with different text and
            // different sizes, and place them at various locations within
            // the movie

            var button1:RectangleButton = new RectangleButton( "Button 1", 60, 100 );
            button1.x = 20;
            button1.y = 20;

            var button2:RectangleButton = new RectangleButton( "Button 2", 80, 30 );
            button2.x = 90;
            button2.y = 20;

            var button3:RectangleButton = new RectangleButton( "Button 3", 100, 40 );
            button3.x = 100;
            button3.y = 60;

```

```

        // Add the buttons to the display list so they appear on-screen
        addChild( button1 );
        addChild( button2 );
        addChild( button3 );
    }
}
}

```

See Also

Recipes 1.5, 6.1, 6.4, and 6.8

Loading External Images at Runtime

Problem

You want to load an external image into a movie while it plays.

Solution

Use the new *Loader* class to load an image (*.jpg*, progressive *.jpg*, *.png*, or *.gif*) and display it on-screen.

Discussion

Recipe 9.17 demonstrates how to embed external assets into a movie at compile time via the [Embed] metadata tag. To load external images or movies at runtime during the playback of a *.swf*, the *Loader* class needs to be used.

The *flash.display.Loader* class is very similar to the *flash.net.URLLoader* class discussed in *Recipe 19.3*. One of the key differences is that *Loader* instances are able to load external images and movies and display them on-screen, whereas *URLLoader* instances are useful for transferring data.

There are three fundamental steps for loading external content:

1. Create an instance of the *Loader* class.
2. Add the *Loader* instance to the display list.
3. Call the *load()* method to pull in an external asset.

The *load()* method of the *Loader* class is responsible for downloading the image or *.swf* file. It takes a single *URLRequest* object as a parameter that specifies the URL of the asset to download and display.

The following is a small example of using a *Loader* instance to download an image named *image.jpg* at runtime. The code in the *LoaderExample* constructor has been commented to coincide with the three basic loading steps previously outlined:

```

package {
    import flash.display.*;
    import flash.net.URLRequest;
    public class LoaderExample extends Sprite {
        public function LoaderExample() {

```

```

    // 1. Create an instance of the Loader class
    var loader:Loader = new Loader( );
    // 2. Add the Loader instance to the display list
    addChild( loader );
    // 3. Call the load( ) method to pull in an external asset
    loader.load( new URLRequest( "image.jpg" ) );
  }
}
}

```

When running this code, the Flash Player looks for *image.jpg* in the same directory that the *.swf* movie is being served from because the *URLRequest* object uses a relative URL. Either a relative or absolute URL can be used to point to the location of the target to load, but the actual loading of the asset is governed by Flash Player's security sandbox, as discussed in *Recipe 3.12*. As soon as the asset has downloaded, it is automatically added as a child of the *Loader* instance.

When loading external assets, it's possible that something could go wrong during the loading process. For instance, perhaps the URL is pointing to the incorrect location due to a spelling mistake, or there's a security sandbox violation that won't allow the asset to be loaded. Or, it's possible that the asset is large and is going to take a long time download. Rather than just having an empty screen while the asset downloads, you'd like to show a preloader to inform the user of the download progress.

In these situations, you should add event listeners to the *contentLoaderInfo* property of the *Loader* instance to be able to respond to the different events as they occur. The *contentLoaderInfo* property is an instance of the *flash.display.LoaderInfo* class, designed to provide information about the target being loaded. The following is a list of useful events dispatched by instances of the *LoaderInfo* class and what those events mean:

open

Generated when the asset has started downloading.

progress

Generated when progress has been made while downloading the asset.

complete

Generated when the asset has finished downloading.

init

Generated when the properties and methods of a loaded external *.swf* are available.

httpStatus

Generated when the status code for a failed HTTP request is detected when attempting to load the asset.

ioError

Generated when a fatal error occurs that results in an aborted download, such as not being able to find the asset.

securityError

Generated when data you're trying to load resides outside of the security sandbox.

unload

Generated when either the *unload()* method is called to remove the loaded content or the *load()* method is called again to replace content that already has been loaded.

The following example demonstrates listening for the various download progress related events when loading an image:

```
package {
    import flash.display.*;
    import flash.text.*;
    import flash.net.URLRequest;
    import flash.events.*;

    public class LoaderExample extends Sprite {
        public function LoaderExample() {
            // Create the loader and add it to the display list
            var loader:Loader = new Loader();
            addChild( loader );

            // Add the event handlers to check for progress
            loader.contentLoaderInfo.addEventListener( Event.OPEN, handleOpen );
            loader.contentLoaderInfo.addEventListener( ProgressEvent.PROGRESS, handleProgress );
            loader.contentLoaderInfo.addEventListener( Event.COMPLETE, handleComplete );

            // Load in the external image
            loader.load( new URLRequest( "image.jpg" ) );
        }

        private function handleOpen( event:Event ):void {
            trace( "open" );
        }

        private function handleProgress( event:ProgressEvent ):void {
            var percent:Number = event.bytesLoaded / event.bytesTotal * 100;
            trace( "progress, percent = " + percent );
        }

        private function handleComplete( event:Event ):void {
            trace( "complete" );
        }
    }
}
```

When running the preceding code, you'll see the open message appear in the console window followed by one or more progress messages displaying the current percent loaded, followed by the complete message signaling that the download finished successfully.

By placing code in the event handlers for these events, you can show the progress of a download as it's being loaded. For instance, the *handleOpen()* method would be in charge of creating the preloader and adding it to the display list. The *handleProgress()* method would update the percentage value of the preloader, such as setting the text of a *TextField* instance to the percent value. Finally, the *handleComplete()* method would perform "clean up" and remove the preloader, since the asset is fully downloaded. Focusing on those methods, the code might look something like this:

```
private function handleOpen( event:Event ):void {
    // Create a simple text-based preloader and add it to the
    // display list
    _loaderStatus = new TextField();
```

```

addChild( loaderStatus );
_loaderStatus.text = "Loading: 0%";
}

private function handleProgress( event:ProgressEvent ):void {
    // Update the loading % to inform the user of progress
    var percent:Number = event.bytesLoaded / event.bytesTotal * 100;
    _loaderStatus.text = "Loading: " + percent + "%";
}

private function handleComplete( event:Event ):void {
    // Clean up - preloader is no longer necessary
    removeChild( loaderStatus );
    _loaderStatus = null;
}

```

The preceding code snippet assumes that there is a `_loaderStatus` variable as part of the class, with type *TextField*:

```
private var _loaderStatus:TextField;
```

Instead of messages appearing via the `trace()` statement as before, modifying the event handlers allows the loading information to be presented inside of the movie itself. This allows users to see the information directly and provides a better experience when loading large external assets.

See Also

Recipes 3.12, 6.7, 9.17, and 19.3

Loading and Interacting with External Movies

Problem

You want to load, and be able to interact with, an external *.swf* movie into your own movie.

Solution

Use the new *Loader* class to load the *.swf* file, and then access the *.swf* file via the content property of the *Loader* instance.

Discussion

Recipe 6.6 demonstrates how to load external images via the *Loader* class. Loading external *.swf* movies uses the same technique—by calling the `load()` method on a *Loader* instance and passing a URL to a *.swf* instead of an image, the *.swf* is loaded into the movie. If the *Loader* is in the main display hierarchy, the *.swf* also appears on-screen.

This recipe involves creating two separate *.swf* files, *ExternalMovie.swf* and *LoaderExample.swf*. The first movie, *ExternalMovie.swf*, will be loaded at runtime into the second movie, *LoaderExample.swf*. The code for *ExternalMovie.swf* is as follows:

```
package {
    import flash.display.Sprite;

```

```

import flash.display.Shape;
public class ExternalMovie extends Sprite {
    private var _color:uint = 0x000000;
    private var _circle:Shape;

    public function ExternalMovie() {
        updateDisplay();
    }

    private function updateDisplay():void {
        // If the circle hasn't been created yet, create it
        // and make it visible by adding it to the display list
        if ( _circle == null ) {
            _circle = new Shape();
            addChild( _circle );
        }

        // Clear any previously drawn content and draw
        // a new circle with the fill color
        _circle.graphics.clear();
        _circle.graphics.beginFill( _color );
        _circle.graphics.drawCircle( 100, 100, 40 );
    }

    // Changes the color of the circle
    public function setColor( color:uint ):void {
        _color = color;
        updateDisplay();
    }

    // Gets the current circle color value
    public function getColor():uint {
        return _color;
    }
}
}

```

The code for *ExternalMovie.swf* is nothing out of the ordinary—a black circle is created when the movie is executed. The main thing to notice about the code is that there are two public methods for accessing and modifying the color of the circle, *getColor()* and *setColor()*. Whenever the *setColor()* method is invoked, the circle is redrawn with the updated color value.

By declaring these methods as public, the methods are able to be called from a movie that loads the *ExternalMovie.swf* in at runtime. In contrast, the private *updateDisplay()* method won't be available to the loading movie. See *Recipe 1.13* for more information about the visibility modifiers for methods.

Now that *ExternalMovie.swf* is created, a new *.swf* needs to be created to load the external movie. This is done with *LoaderExample.swf*, which has the following code:

```

package {
    import flash.display.*;
    import flash.net.URLRequest;

```

```

import flash.events.Event;

public class LoaderExample extends Sprite {

    private var _loader:Loader;

    public function LoaderExample() {
        // Create the Loader and add it to the display list
        _loader = new Loader();
        addChild( _loader );

        // Add the event handler to interact with the loaded movie
        _loader.contentLoaderInfo.addEventListener( Event.INIT, handleInit );

        // Load the external movie
        _loader.load( new URLRequest( "ExternalMovie.swf" ) );
    }

    // Event handler called when the externally loaded movie is
    // ready to be interacted with
    private function handleInit( event:Event ):void {
        // Typed as * here because the type is not known at compile-time.
        var movie:* = _loader.content;

        // Calls a method in the external movie to get data out
        // Displays: 0
        trace( movie.getColor() );

        // Calls a method in the external movie to set data.
        // Sets the color in the external movie, which draws
        // a circle with the new color, in this case red
        movie.setColor( 0xFF0000 );
    }
}

```

The code for *LoaderExample.swf* is more interesting in that it communicates with the loaded movie. There are two main aspects in the preceding code:

1. Listening for the `init` event
2. Accessing the loaded movie via the content property

The `init` event is fired when the loaded movie has initialized enough that its methods and properties are available to be interacted with. The movie can be controlled only *after* the `init` event has been fired from the loader. Attempting to interact with a loaded movie before it has initialized will generate runtime errors.

To control the loaded movie, you'll first need to get a reference to it. This is done via the content property of the *Loader* class. In the preceding code, the loader variable refers to the *Loader* that pulled in the external *.swf* file, so you can access the movie via `loader.content`. If the loader variable weren't available, the `event.target.content` path could be used instead to get to the contents of the *Loader*. This is because `event.target` refers to the instance that generated the event, which is the same instance that the loader variable refers to.

The content property is read-only, and returns an object of type *DisplayObject*. In the *LoaderExample.swf* code, you'll notice that instead of typing the movie variable as a *DisplayObject*, the same type as what the content property returns, the * type was used. This is necessary because trying to call the *getColor()* or *setColor()* methods on the movie reference generates compile-time errors if movie is typed as a *DisplayObject*.

The movie being loaded, *ExternalMovie.swf*, has two public methods available for interaction. These methods are not part of the *DisplayObject* class; therefore, trying to call one of the methods from a variable of type *DisplayObject* is an error. The * type allows you to call any method that you'd like on the loaded movie. If the method does not exist in the loaded movie, a *ReferenceError* is thrown during execution.

The *getColor()* method returns the color of the circle in *ExternalMovie.swf* to *LoaderExample.swf*. The *LoaderExample.swf* reports the color as 0, which is the same as 0x000000, or the color black. The *setColor()* method allows *LoaderExample.swf* to change the color of the circle drawn by *ExternalMovie.swf*. In this case, the color of the circle is set to red, and you can see that the *ExternalMovie.swf* updates the display after the new circle color value is set.



It is only possible to interact with *.swf* files of Version 9 and above using this technique. When loading Version 8 and below *.swf* files, this technique won't work because ActionScript 3.0 code runs independently of ActionScript 1.0 and 2.0. Communication with these *.swf* files is not trivial and involves using *LocalConnection* as a workaround to send and receive messages. See *Chapter 19* for details.

See Also

Recipes *1.13* and *6.6*

Creating Mouse Interactions

Problem

You want users to interact with your movie using their mouse.

Solution

Use the various mouse events to listen for mouse interactions on display objects of type *InteractiveObject*. Use the read-only *mouseX* and *mouseY* properties from *DisplayObject* to examine the mouse location relative to a display object, or the *localX* and *localY* properties from the *MouseEvent* passed to a mouse event handler.

Discussion

Basic mouse interaction can be created with the *SimpleButton* class, as described in *Recipe 6.5*. The *SimpleButton* class provides an easy way to create a clickable button with different button visual states: up, over, and down.

However, there are times when buttons just don't provide enough interactivity. By listening to the various mouse events, you can create interesting interactive experiences. For instance, consider that you want to track the mouse cursor to create an interactive drawing program, drawing lines

on-screen based on the user's mouse movement. Or, consider that you have a maze that a user must navigate their mouse through without colliding with the walls to find the exit. Or, perhaps the user's mouse movement needs to control the direction of a golf club, and the mouse button is used to swing.

These situations require use of the special *InteractiveObject* display object, which provides the ability to respond to the user's mouse. If you go back to the introduction for this chapter, you'll recall that the *InteractiveObject* class is a base class fairly high in the display object class hierarchy. Because of this, the *Sprite*, *Loader*, *TextField*, and *MovieClip* classes are all examples of the *InteractiveObject* class since they fall underneath *InteractiveObject* in the hierarchy, and you may already be familiar with their use.

Instances of the *InteractiveObject* dispatch the necessary events specific to mouse interaction. The following is a list of more useful mouse events:

click

Generated when the user presses and releases the mouse button over the interactive display object.

doubleClick

Generated when the user presses and releases the mouse button twice in rapid succession over the interactive display object.

mouseDown

Generated when the user presses the mouse button over the interactive display object.

mouseUp

Generated when the user releases the mouse button over the interactive display object.

mouseOver

Generated when the user moves the mouse pointer from outside of the bounds of interactive display object to inside of them.

mouseMove

Generated when the user moves the mouse pointer while the pointer is inside the bounds of the interactive display object.

mouseOut

Generated when the user moves the mouse pointer from inside the bounds of an interactive display object to outside of them.

mouseWheel

Generated when the user rotates the mouse wheel while the mouse pointer is over the interactive display object.

Using these events is simply a matter of calling *addEventListener()* on the *InteractiveObject* and defining an event handler to handle the *MouseEvent* passed to it.

The following code snippet creates a *Sprite*, draws a red circle inside of it, and outputs a message to the console whenever the mouse moves over the circle:

```
package {
    import flash.display.Sprite;
    import flash.events.*;
    import flash.geom.Point;
```

```

public class InteractiveMouseDemo extends Sprite {
    public function InteractiveMouseDemo( ) {
        var circle:Sprite = new Sprite( );
        circle.x = 10;
        circle.y = 10;
        circle.graphics.beginFill( 0xFF0000 );
        circle.graphics.drawCircle( 0, 0, 5 );
        circle.graphics.endFill( );

        circle.addEventListener( MouseEvent.CLICK, handleClick );

        addChild( circle );
    }

    // Event handle to capture the click event over the circle
    private function handleClick( event:MouseEvent ):void {
        trace( "mouse click" );
    }
}

```

In this example, notice that the message appears only when the mouse is moved while the pointer is over the circle. The circle defines the bounds for the *Sprite* in this case.



Mouse events are generated from a particular interactive display object only when the pointer is within the bounds of that object.

Another common use of mouse events stems from wanting to inspect the location of the mouse pointer to create mouse interactivity. For example, to draw a line with the mouse, the mouse location needs to be known so the line can be plotted accurately. There are two ways to determine the location of the mouse pointer:

- Using the `mouseX` and `mouseY` properties available on any *DisplayObject* instance.
- Using the `localX` and `localY` properties available from the *MouseEvent* instance passed to the mouse event handler.

The `mouseX` and `mouseY` properties can be inspected to determine the location of the mouse cursor relative to the top-left corner of the *DisplayObject*. Both of the properties are read-only; it is not possible to set the location of the mouse cursor, only to examine the location.

So, imagine that a rectangle is at `x` location 20 and `y` location 50 and the user moves the mouse pointer to `x` location 25 and `y` location 60. The `mouseX` property of the rectangle returns 5 and `mouseY` of the rectangle reports 10 because from the rectangle's perspective, the mouse is 5 pixels in from the left and 10 pixels down from the top.

The `localX` and `localY` properties of the *MouseEvent* are also relative. In the *MouseEvent* case, the properties are relative to interactive display object that dispatched the event. Therefore, consider that a rectangle reports `mouseX` of 10 and dispatches a `mouseMove` event. The event's `localX` property is also 10.

To get the global position of the mouse from local coordinates, use the `localToGlobal()` method of the *DisplayObject* class. The `localToGlobal()` method takes *flash.geom.Point* as a parameter that specifies the local coordinates, and returns a new *Point* with the coordinates converted to the global

space. The following code snippet focuses on the event handler and demonstrates how to convert localX and localY to global coordinates:

```
// Event handler to respond to a mouseMove event
private function handleMouseMove( event:MouseEvent ):void {
    /* Displays:
    local x: 3.95
    local y: 3.45
    */
    trace( "local x: " + event.localX );
    trace( "local y: " + event.localY );

    // Create the point that localToGlobal should convert
    var localPoint:Point = new Point( event.localX, event.localY );
    // Convert from the local coordinates of the display object that
    // dispatched the event to the global stage coordinates
    var globalPoint:Point = event.target.localToGlobal( localPoint );

    /* Displays:
    global x: 13.95
    global y: 13.45
    */
    trace( "global x: " + globalPoint.x );
    trace( "global y: " + globalPoint.y );
}
```

A complete working example of creating interactivity through handling the various mouse events can be demonstrated by the simple drawing program that follows. Whenever the mouse is pressed, the drawing starts. As the user moves the mouse around the screen, a line is drawn that follows the movement of the mouse pointer. When the user releases the mouse button, the drawing stops:

```
package {
    import flash.display.Sprite;
    import flash.events.MouseEvent;
    public class DrawingDemo extends Sprite {

        // Flag to indicate whether the mouse is in draw mode
        private var _drawing:Boolean;

        public function DrawingDemo() {
            // Configure the line style
            graphics.lineStyle( 2, 0xFFCC33 );

            // Drawing is false until the user presses the mouse
            _drawing = false;

            // Add the mouse listeners on the stage object to be
            // notified of any mouse event that happens while the
            // mouse is over the entire movie
            stage.addEventListener( MouseEvent.CLICK, startDrawing );
            stage.addEventListener( MouseEvent.CLICK, draw );
            stage.addEventListener( MouseEvent.CLICK, stopDrawing );
        }

        public function startDrawing( event:MouseEvent ):void {
```

```

        // Move to the current mouse position to be ready for drawing
        graphics.moveTo( mouseX, mouseY );
        _drawing = true;
    }

    public function draw( event:MouseEvent ):void {
        if ( _drawing ) {
            // Draw a line from the last mouse position to the
            // current one
            graphics.lineTo( mouseX, mouseY );
        }
    }

    public function stopDrawing( event:MouseEvent ):void {
        _drawing = false;
    }
}
}

```

See Also

Recipes 6.5, 6.9, and Chapter 7

Dragging and Dropping Objects with the Mouse

Problem

You want to provide a drag-and-drop-style interface.

Solution

Use *startDrop()*, *stopDrag()* and `dropTarget` from the *Sprite* class to implement drag-and-drop behavior. Alternatively, extend the *ascb.display.DraggableSprite* class for visually smoother dragging behavior using the *drag()* and *drop()* methods.

Discussion

Creating drag-and-drop behavior is not as difficult as you might think. The *Sprite* class includes methods specifically for the purpose of drag and drop, namely *startDrag()* and *stopDrag()*.

The *startDrag()* method can be called on any *Sprite* instance to have it follow the mouse around the screen, creating the dragging effect. To stop dragging, call the *stopDrag()* method on the *Sprite* instance. After the drag operation is complete, you can examine the `dropTarget` property of the *Sprite* to determine the object that the *Sprite* was dropped on. The value of `dropTarget` is useful for determining if a drop operation is valid (such as dropping a folder icon on a trashcan to delete it).

When calling *startDrag()*, you don't have to specify any parameters; however, the method accepts up to two parameters. The parameters are:

lockCenter

When true the center of the *Sprite* is locked to the mouse position regardless of where the user pressed the mouse. When false the *Sprite* follows the mouse from the location where the user first clicked. The default value is false.

bounds

The *Rectangle* region where you want to constrain dragging. The *Sprite* is not capable of being dragged outside of this region. The default value is null, meaning there is no area constraint.

The following code example uses these methods to set up a simple drag-and-drop behavior. There are three rectangles on the left capable of being dragged: red, green, and blue. The rectangle on the right is white, and serves as the target area where you drop the color rectangles. Dragging and dropping a colored rectangle onto the white rectangle colorizes the white rectangle the same color of the rectangle that was dropped onto it:

```
package {
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.events.MouseEvent;
    import flash.geom.Point;
    import flash.filters.DropShadowFilter;

    public class ColorDrop extends Sprite {

        private var _red:Sprite;
        private var _green:Sprite;
        private var _blue:Sprite;
        private var _white:Sprite;

        // Saves the starting coordinates of a dragging Sprite so
        // it can be placed back
        private var startingLocation:Point;

        // Create the rectangles that comprise the interface
        // and wire the mouse events to make them interactive
        public function ColorDrop() {
            createRectangles();
            addEventListeners();
        }

        private function createRectangles():void {
            _red = new Sprite();
            _red.graphics.beginFill( 0xFF0000 );
            _red.graphics.drawRect( 0, 10, 10, 10 );
            _red.graphics.endFill();

            _green = new Sprite()
            _green.graphics.beginFill( 0x00FF00 );
            _green.graphics.drawRect( 0, 30, 10, 10 );
            _green.graphics.endFill();

            _blue = new Sprite();
            _blue.graphics.beginFill( 0x0000FF );
            _blue.graphics.drawRect( 0, 50, 10, 10 );
            _blue.graphics.endFill();
        }
    }
}
```

```

    _white = new Sprite();
    _white.graphics.beginFill( 0xFFFFFFFF );
    _white.graphics.drawRect( 20, 10, 50, 50 );
    _white.graphics.endFill();

    addChild( _red );
    addChild( _green );
    addChild( _blue );
    addChild( _white );
}

private function addEventListeners():void {
    _red.addEventListener( MouseEvent.CLICK, pickup );
    _red.addEventListener( MouseEvent.CLICK, place );

    _green.addEventListener( MouseEvent.CLICK, pickup );
    _green.addEventListener( MouseEvent.CLICK, place );

    _blue.addEventListener( MouseEvent.CLICK, pickup );
    _blue.addEventListener( MouseEvent.CLICK, place );
}

public function pickup( event:MouseEvent ):void {
    // Save the original location so you can put the target back
    startingLocation = new Point();
    startingLocation.x = event.target.x;
    startingLocation.y = event.target.y;

    // Start dragging the Sprite that was clicked on and apply
    // a drop shadow filter to give it depth
    event.target.startDrag();
    event.target.filters = [ new DropShadowFilter() ];

    // Bring the target to front of the display list so
    // it appears on top of everything else
    setChildIndex( DisplayObject(event.target), numChildren - 1 );
}

public function place( event:MouseEvent ):void {
    // Stop dragging the Sprite around and remove the depth
    // effect (i.e., the drop shadow) from the filter
    event.target.stopDrag();
    event.target.filters = null;

    // Check to see if the Sprite was dropped over the white
    // rectangle, and if so, update the color
    if ( event.target.dropTarget == _white ) {
        // Determine which color was dropped, and apply that color
        // to the white rectangle
        var color:uint;
        switch ( event.target ) {
            case _red: color = 0xFF0000; break;
            case _green: color = 0x00FF00; break;
        }
    }
}

```

```

        case _blue: color = 0x0000FF; break;
    }

    _white.graphics.clear( );
    _white.graphics.beginFill( color );
    _white.graphics.drawRect( 20, 10, 50, 50 );
    _white.graphics.endFill( );
}

// Place the dragging Sprite back to its original location
event.target.x = startingLocation.x;
event.target.y = startingLocation.y;
}

}
}

```

Breaking down this code a bit, all of the rectangles are added to the display list and then the appropriate `mouseDown` and `mouseUp` listeners are defined. Every time a `mouseDown` is received over one of the colored rectangles, the pickup process starts.

First, the original location of the rectangle is saved. This allows the rectangle's location to be restored after the drop operation. Next, the `startDrag()` method is called on the rectangle to start dragging it around the screen. After that, a `DropShadowFilter` is applied to provide depth during the drag and make it appear as if the rectangle were held above the others in the display list. Finally, the rectangle is moved to the front of the display list via `setChildIndex()` so that it is drawn on top of all of the others as it follows the mouse.

When the `mouseUp` event is detected, the drop operation commences via the `place()` method. First, the rectangle has `stopDrag()` called on it to stop the mouse follow behavior, and the filters are removed to reverse the depth effect. Next, the rectangle's `dropTarget` property is examined to determine if it was dropped over the white rectangle. If the white rectangle is indeed the `dropTarget`, the white rectangle is given the same color as the rectangle that was dropped onto it. Finally, because the rectangle is out of position now from following the mouse around, the original starting location is restored.

The previous code works alright, but there are two small problems with it: the `dropTarget` property isn't always reliable and the dragging is choppy.

The `dropTarget` property continually changes during movement after `startDrag()` is issued. This is good because it allows for feedback to be provided as the object is moved over different possible drop targets; you can indicate whether a drop is currently allowed based on whatever `dropTarget` currently is. However, `dropTarget` only changes when the pointer passes over a *new* display object, and *not* when the pointer leaves a display object. This presents a problem when you move over an object and then leave that object without moving over a new one. In such a case, the `dropTarget` property still points to the last object that the mouse moved over, even though the mouse may have moved outside of that object without ever moving over a new object. This means that the mouse is not guaranteed to actually be over the display object that `dropTarget` refers to.

To see this effect in action:

1. Pick up the red rectangle.

2. Move it over the white rectangle.
3. Move the mouse further to the right so the red rectangle is outside the area of the white rectangle.
4. With the red rectangle *outside* the white rectangle, release it.

You can see that the white rectangle is colored red because the `dropTarget` is still referring to the white rectangle, even though the red rectangle is dropped outside of the white rectangle bounds.

To fix this behavior, use the `hitTestPoint()` method to determine if the mouse location is within the bounds of the `dropTarget` display object. The `hitTestPoint()` method takes an *x* and *y* location and returns a true or false value, indicating if the location falls within the bounds of the object. An optional third *Boolean* parameter can be used to specify how the hit test area is calculated. Specifying `false` as the third parameter will use the bounding box rectangle of the object, whereas `true` uses the actual shape of the object itself. The default value is `false`.

Inside of the `place()` method that tests if the colored rectangle was dropped correctly, add a call to `hitTestPoint()` inside the conditional checking the `dropTarget`. This makes sure the mouse cursor still is within the bounds of the white rectangle before allowing the drop:

```
if ( event.target.dropTarget == _white
    && _white.hitTestPoint( _white.mouseX, _white.mouseY ) ) {
```

Another problem with the code is the choppy screen updating during mouse movement. This is because mouse events happen independently of the rendering process. The movie's frame rate determines how often the screen is updated, so if the mouse changes the display, the updated display won't appear until the screen is normally refreshed (as specified by the frame rate).

To combat this problem, the `MouseEvent` class includes the method `updateAfterEvent()`. Typically called when the `mouseMove` event is handled, `updateAfterEvent()` notifies the Flash Player that the screen has changed and instructs it to redraw. This avoids the delay that occurs when waiting for the frame rate to update the screen normally after mouse movement.

Unfortunately, `updateAfterEvent()` does not play nice with `startDrag()`. Even if a `mouseMove` event handler is added for the sole purposes of calling `updateAfterEvent()` to handle the rendering updates, calling `updateAfterEvent()` has no effect. Another problem with `startDrag()` is that you are able to drag only one *Sprite* at a time. Although this isn't necessarily a major problem, it is rather limiting.

To address these issues, a new custom visual class, named `DraggableSprite`, was created as part of the *ActionScript 3.0 Cookbook* Library (found at <http://www.rightactionscript.com/ascb>); it can be found in the `ascb.display` package.

The `DraggableSprite` class inherits from `Sprite`, and adds two aptly named methods: `drag()` and `drop()`. The `drag()` method takes the same parameters and is used the same way as `startDrag()`. The `drop()` method behaves the same as `stopDrag()`.

The primary difference is that the drag-and-drop functionality available in `DraggableSprite` is implemented by custom mouse tracking code, versus having the Flash Player track the mouse internally. Because of this, both negative aspects of `startDrag()` are overcome. Multiple `DraggableSprite` instances are able to move with the mouse at the same time, and the rendering delay issue is eliminated because `updateAfterEvent()` works as expected.

However, when switching to *DraggableSprite*, the `dropTarget` property is no longer applicable. Instead, you have to use the `getObjectsUnderPoint()` method to return the objects beneath the mouse and determine if a drop is valid based on the information returned.

The `getObjectsUnderPoint()` method returns an array of display objects that are children of the container the method was called on. The item at the end of the array, in position `length - 1`, is the top-most item (the object directly underneath the mouse). The item at position 0 is the very bottom item underneath the mouse. By testing to see if the white rectangle is in the list of objects under the mouse location at the time of the drop, you can determine if the drop was valid or not.

The following code is the same drag-and-drop behavior as before, but updated to use *DraggableSprite* instead of *Sprite*:

```
package {
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.events.MouseEvent;
    import flash.geom.Point;
    import flash.filters.DropShadowFilter;

    import ascb.display.DraggableSprite;

    public class ColorDrop extends Sprite {

        private var _red: DraggableSprite;
        private var _green: DraggableSprite;
        private var _blue: DraggableSprite;
        private var _white: Sprite;

        // Saves the starting coordinates of a dragging Sprite so
        // it can be placed back
        private var startingLocation: Point;

        // Create the rectangles that comprise the interface
        // and wire the mouse events to make them interactive
        public function ColorDrop() {
            createRectangles();
            addEventListeners();
        }

        private function createRectangles(): void {
            _red = new DraggableSprite();
            _red.graphics.beginFill( 0xFF0000 );
            _red.graphics.drawRect( 0, 10, 10, 10 );
            _red.graphics.endFill();

            _green = new DraggableSprite()
            _green.graphics.beginFill( 0x00FF00 );
            _green.graphics.drawRect( 0, 30, 10, 10 );
            _green.graphics.endFill();

            _blue = new DraggableSprite();
            _blue.graphics.beginFill( 0x0000FF );
            _blue.graphics.drawRect( 0, 50, 10, 10 );
        }
    }
}
```

```

    _blue.graphics.endFill( );

    _white = new DraggableSprite( );
    _white.graphics.beginFill( 0xFFFFFF );
    _white.graphics.drawRect( 20, 10, 50, 50 );
    _white.graphics.endFill( );

    addChild( _red );
    addChild( _green );
    addChild( _blue );
    addChild( _white );
}

private function addEventListeners():void {
    _red.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _red.addEventListener( MouseEvent.MOUSE_UP, place );

    _green.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _green.addEventListener( MouseEvent.MOUSE_UP, place );

    _blue.addEventListener( MouseEvent.MOUSE_DOWN, pickup );
    _blue.addEventListener( MouseEvent.MOUSE_UP, place );
}

public function pickup( event:MouseEvent ):void {
    // Save the original location so you can put the target back
    startingLocation = new Point( );
    startingLocation.x = event.target.x;
    startingLocation.y = event.target.y;

    // Start dragging the Sprite that was clicked on and apply
    // a drop shadow filter to give it depth
    event.target.drag( );
    event.target.filters = [ new DropShadowFilter( ) ];

    // Bring the target to front of the display list so
    // that it appears on top of everything else
    setChildIndex( DisplayObject( event.target ), numChildren - 1 );
}

public function place( event:MouseEvent ):void {
    // Stop dragging the Sprite around and remove the depth
    // effect from the filter
    event.target.drop( );
    event.target.filters = null;

    // Get a list of objects inside this container that are
    // underneath the mouse
    var dropTargets:Array = getObjectsUnderPoint( new Point( mouseX, mouseY ) );

    // The display object at position length - 1 is the top-most object,
    // which is the rectangle that is currently being moved by the mouse.
    // If the white rectangle is the one immediately beneath that, the
    // drop is valid

```

```

if ( dropTargets[ dropTargets.length - 2 ] == _white ) {
    // Determine which color was dropped, and apply that color
    // to the white rectangle
    var color:uint;
    switch ( event.target ) {
        case _red: color = 0xFF0000; break;
        case _green: color = 0x00FF00; break;
        case _blue: color = 0x0000FF; break;
    }

    _white.graphics.clear();
    _white.graphics.beginFill( color );
    _white.graphics.drawRect( 20, 10, 50, 50 );
    _white.graphics.endFill();
}

// Place the dragging Sprite back to its original location
event.target.x = startingLocation.x;
event.target.y = startingLocation.y;
}
}
}

```

See Also

Recipes *6.4* and *6.8*

Index

.
.gif files, 25
.jpg files, 25
.png files, 25

A

ActionScript Virtual Machine (AVM), 6
addChild() method, 17
 adding item to display lists, 6
addChildAt() method, 17
 adding item to display lists, 6
addEventListener() method, 32
ArgumentError, 11
ascb.util.DisplayObjectUtilities class, 13
attachMovie() method, 3, 18
AVM (ActionScript Virtual Machine), 6

B

Bitmap class, 5
Booleans
 dragging/dropping objects and, 39
bounds parameter (startDrag() method), 36
button instances, 21
buttons, creating, 20

C

click event (InteractiveObject), 20, 32
complete event (LoaderInfo class), 26
content property (Loader class), 30
contentLoaderInfo property, 26
createCircle() method, 20
createEmptyMovieClip() method, 3

D

display lists, 3
 adding items to, 6
 objects, moving forward/backward, 14
 removing items from, 10
display object containers, 3
DisplayObjectContainer class, 6
 moving objects forward/backward, 14
 removing items from display lists, 10
DisplayObject class, 5
 adding items to display list, 6
 custom visual classes, creating, 17
 external movies and, 31
 mouse interactions, creating, 31
 simple buttons, creating, 20
DisplayObjectUtilities.removeAllChildren() method, 13
doubleClick event (InteractiveObject), 32
downState property (SimpleButton class), 20
drag() method, 39
DraggableSprite custom visual class, 39
dragging objects with the mouse, 35
Drawing API, 19

drop() method, 39
dropping objects with the mouse, 35
dropTarget property (Sprite class), 35
duplicateMovieClip() method, 3

E

external images
 runtime, loading, 25
external movies, loading/interacting with, 28

F

flash.display package, 3
flash.display.Loader class, 25
flash.net.URLLoader class, 25

G

getChildAt() method, 14
getChildIndex() method, 14
getColor() method, 31
getNextHighestDepth() method, 4
getObjectsUnderPoint() method, 40

H

handleClick() method, 21
handleComplete() method, 27
handleOpen() method, 27
handleProgress() method, 27
hierarchy tree (display lists), 3
hitTestPoint() method, 39
hitTestState property (SimpleButton class), 20, 22
httpStatus event (LoaderInfo class), 26

I

images
 runtime, loading, 25
init event (LoaderInfo class), 26
 external movies and, 30
InteractiveObject class, 31
InteractiveObject display object, 32
ioError event (LoaderInfo class), 26

L

load() method, 25, 28
Loader class, 6, 25
 external movies, loading/interacting with, 28
 mouse interactions and, 32
LocalConnection class, 31
localToGlobal() method, 33
localX property (MouseEvent class), 33
localY property (MouseEvent class), 33
lockCenter parameter (startDrag() method), 35

M

- mouse events
 - dragging/dropping objects with, 35
 - interactions, creating, 31
- mouseDown event (InteractiveObject), 32
- MouseEvent class, 31
 - dragging dropping objects, 39
- MouseEvent.CLICK, 21
- mouseMove event (InteractiveObject), 32
- mouseOut event (InteractiveObject), 32
- mouseUp event (InteractiveObject), 32
- mouseWheel event (InteractiveObject), 32
- mouseX property (DisplayObject class), 31, 33
- mouseY property (DisplayObject class), 31, 33
- MovieClip class, 3, 5
 - adding items to display list, 6
 - custom visual classes and, 18
 - mouse interactions and, 32
- movies
 - external, loading/interacting, 28

N

- new operator, 6
- numChildren property, 12

O

- objects
 - dragging/dropping with the mouse, 35
 - moving forward and backward, 14
- open event (LoaderInfo class), 26
- overState property (SimpleButton class), 20

P

- place() method, 39
- progress event
 - LoaderInfo class, 26
- progressive .jpg, 25

R

- RangeErrors, 8, 11, 16
- Rectangle region, 36
- removeChild() method, 10
- removeChildAt() method, 10
- Rendering Engine, 6
- reparenting, 9

S

- securityError event (LoaderInfo class), 26
- setChildIndex() method, 14
 - dragging/dropping objects, 38
- setColor() method, 31
- Shape class, 3, 5
 - custom visual classes, 18
- simple buttons, creating, interactive buttons, creating, 20
- SimpleButton class, 20
 - mouse interactions and, 31
- Sprite class, 3, 5
 - adding items to display list, 6
 - custom visual classes, 18
 - dragging/dropping objects with the mouse, 35
 - mouse interactions and, 32
- stage, 3
- startDrag() method, 39
- startDrop() method, 35
- stopDrag() method, 35
- swapDepths(), 4

T

- TextField class, 3, 6
 - adding items to display list, 6
 - loading external content and, 27
 - mouse interactions and, 32
 - removing items from display lists, 11
- trace()
 - loading external images and, 28

U

- unload event (LoaderInfo class), 26
- updateAfterEvent() method, 39
- upState property (SimpleButton class), 20
- URLLoader object
 - external content, loading, 25

V

- Video class, 6
- visual class, creating custom, 17

\

- \\, 25