Filtering display objects

Example: Filter Workbench

The Filter Workbench provides a user interface to apply different filters to images and other visual content and see the resulting code that can be used to generate the same effect in ActionScript. In addition to providing a tool for experimenting with filters, this application demonstrates the following techniques:

- Creating instances of various filters
- Applying multiple filters to a display object

To get the application files for this sample, see www.adobe.com/go/learn_programmingAS3samples_flash. The Filter Workbench application files can be found in the Samples/FilterWorkbench folder. The application consists of the following files:

File	Description
com/example/programmingas3/filterWorkbench/FilterWorkbenchController.as	Class that provides the main functionality of the application, including switching content to which filters are applied, and applying filters to content.
com/example/programmingas3/filterWorkbench/IFilterFactory.as	Interface defining common methods that are implemented by each of the filter factory classes. This interface defines the common functionality that the FilterWorkbenchController class uses to interact with the individual filter factory classes.

File	Description
in folder com/example/programmingas3/filterWorkbench/: BevelFactory.as BlurFactory.as ColorMatrixFactory.as ConvolutionFactory.as DropShadowFactory.as GlowFactory.as GradientBevelFactory.as GradientGlowFactory.as	Set of classes, each of which implements the IFilterFactory interface. Each of these classes provides the functionality of creating and setting values for a single type of filter. The filter property panels in the application use these factory classes to create instances of their particular filters, which the FilterWorkbenchController class retrieves and applies to the image content.
com/example/programmingas3/filterWorkbench/ColorStringFormatter.as	Utility class that includes a method to convert a numeric color value to hexadecimal String format
com/example/programmingas3/filterWorkbench/GradientColor.as	Class that serves as a value object, combining into a single object the three values (color, alpha, and ratio) that are associated with each color in the GradientBevelFilter and GradientGlowFilter
User interface (Flash)	
FilterWorkbench.fla	The main file defining the application's user interface.
flashapp/FilterWorkbench.as	Class that provides the functionality for the main application's user interface; this class is used as the document class for the application FLA file.
In folder flashapp/filterPanels: BevelPanel.as BlurPanel.as ColorMatrixPanel.as ConvolutionPanel.as DropShadowPanel.as GlowPanel.as GradientBevelPanel.as GradientGlowPanel.as	Set of classes that provide the functionality for each panel that is used to set options for a single filter. For each class, there is also an associated MovieClip symbol in the library of the main application FLA file, whose name matches the name of the class (for example, the symbol "BlurPanel" is linked to the class defined in BlurPanel.as). The components that make up the user interface are positioned and named within those symbols.
flashapp/ImageContainer.as	A display object that serves as a container for the loaded image on the screen
flashapp/ButtonCellRenderer.as	Custom cell renderer used to include a button component in a cell in the DataGrid component

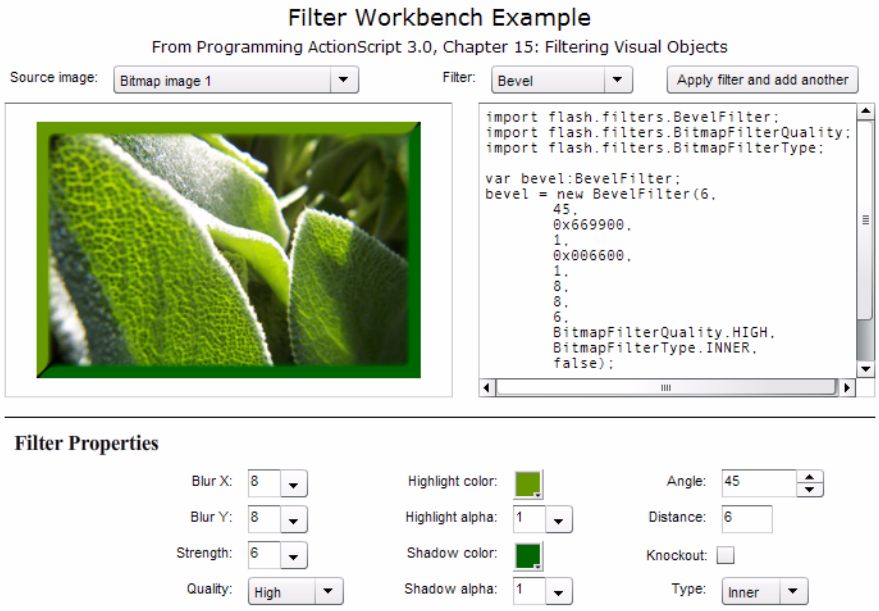
File	Description
Filtered image content	
com/example/programmingas3/filterWorkbench/ImageType.as	This class serves as a value object containing the type and URL of a single image file to which the application can load and apply filters. The class also includes a set of constants representing the actual image files available.
images/sampleAnimation.swf, images/sampleImage1.jpg, images/sampleImage2.jpg	Images and other visual content to which filters are applied in the application.

Experimenting with ActionScript filters

The Filter Workbench application is designed to help you experiment with various filter effects and generate the relevant ActionScript code for that effect. The application lets you select from three different files containing visual content, including bitmap images and a Flash animation, and apply eight different ActionScript filters to the selected image, either individually or in combination with other filters. The application includes the following filters:

- Bevel (flash.filters.BevelFilter)
- Blur (flash.filters.BlurFilter)
- Color matrix (flash.filters.ColorMatrixFilter)
- Convolution (flash.filters.ConvolutionFilter)
- Drop shadow (flash.filters.DropShadowFilter)
- Glow (flash.filters.GlowFilter)
- Gradient bevel (flash.filters.GradientBevelFilter)
- Gradient glow (flash.filters.GradientGlowFilter)

Once a user has selected an image and a filter to apply to that image, the application displays a panel with controls for setting the specific properties of the selected filter. For example, the following image shows the application with the Bevel filter selected:



As the user adjusts the filter properties, the preview updates in real time. The user can also apply multiple filters by customizing one filter, clicking the Apply button, customizing another filter, clicking the Apply button, and so forth.

There are a few features and limitations in the application's filter panels:

- The color matrix filter includes a set of controls for directly manipulating common image properties including brightness, contrasts, saturation, and hue. In addition, custom color matrix values can be specified.
- The convolution filter, which is only available using ActionScript, includes a set of commonly used convolution matrix values, or custom values can be specified. However, while the ConvolutionFilter class accepts a matrix of any size, the Filter Workbench application uses a fixed 3 x 3 matrix, the most commonly used filter size.

- The displacement map filter, which is only available in ActionScript, is not available in the Filter Workbench application. A displacement map filter requires a map image in addition to the filtered image content. The map image is the primary input that determines the result of the filter, so without the ability to load or create a map image, the ability to experiment with the displacement map filter would be extremely limited.

Creating filter instances

The Filter Workbench application includes a set of classes, one for each of the available filters, which are used by the individual panels to create the filters. When a user selects a filter, the ActionScript code associated with the filter panel creates an instance of the appropriate filter factory class. (These classes are known as *factory classes* because their purpose is to create instances of other objects, much like a real-world factory creates individual products.)

Whenever the user changes a property value on the panel, the panel's code calls the appropriate method in the factory class. Each factory class includes specific methods that the panel uses to create the appropriate filter instance. For example, if the user selects the Blur filter, the application creates a BlurFactory instance. The BlurFactory class includes a `modifyFilter()` method that accepts three parameters: `blurX`, `blurY`, and `quality`, which together are used to create the desired BlurFilter instance:

```
private var _filter:BlurFilter;

public function modifyFilter(blurX:Number = 4, blurY:Number = 4,
  quality:int = 1):void
{
  _filter = new BlurFilter(blurX, blurY, quality);
  dispatchEvent(new Event(Event.CHANGE));
}
```

On the other hand, if the user selects the Convolution filter, that filter allows for much greater flexibility and consequently has a larger set of properties to control. In the ConvolutionFactory class, the following code is called when the user selects a different value on the filter panel:

```
private var _filter:ConvolutionFilter;

public function modifyFilter(matrixX:Number = 0,
                             matrixY:Number = 0,
                             matrix:Array = null,
                             divisor:Number = 1.0,
                             bias:Number = 0.0,
                             preserveAlpha:Boolean = true,
                             clamp:Boolean = true,
                             color:uint = 0,
                             alpha:Number = 0.0):void
```

```
{
  _filter = new ConvolutionFilter(matrixX, matrixY, matrix, divisor, bias,
  preserveAlpha, clamp, color, alpha);
  dispatchEvent(new Event(Event.CHANGE));
}
```

Notice that in each case, when the filter values are changed, the factory object dispatches an `Event.CHANGE` event to notify listeners that the filter's values have changed. The FilterWorkbenchController class, which does the work of actually applying filters to the filtered content, listens for that event to ascertain when it needs to retrieve a new copy of the filter and re-apply it to the filtered content.

The FilterWorkbenchController class doesn't need to know specific details of each filter factory class—it just needs to know that the filter has changed and to be able to access a copy of the filter. To support this, the application includes an interface, IFilterFactory, that defines the behavior a filter factory class needs to provide so the application's FilterWorkbenchController instance can do its job. The IFilterFactory defines the `getFilter`() method that's used in the FilterWorkbenchController class:

```
function getFilter():BitmapFilter;
```

Notice that the `getFilter()` interface method definition specifies that it returns a BitmapFilter instance rather than a specific type of filter. The BitmapFilter class does not define a specific type of filter. Rather, BitmapFilter is the base class on which all the filter classes are built. Each filter factory class defines a specific implementation of the `getFilter()` method in which it returns a reference to the filter object it has built. For example, here is an abbreviated version of the ConvolutionFactory class's source code:

```
public class ConvolutionFactory extends EventDispatcher implements
  IFilterFactory
{
  // ------- Private vars -------
  private var _filter:ConvolutionFilter;
  ...
  // ------- IFilterFactory implementation -------
  public function getFilter():BitmapFilter
  {
    return _filter;
  }
  ...
}
```

In the ConvolutionFactory class's implementation of the `getFilter()` method, it returns a ConvolutionFilter instance, although any object that calls `getFilter()` doesn't necessarily know that—according to the definition of the `getFilter()` method that ConvolutionFactory follows, it must return any BitmapFilter instance, which could be an instance of any of the ActionScript filter classes.

Applying filters to display objects

As explained in the previous section, the Filter Workbench application uses an instance of the FilterWorkbenchController class (hereafter referred to as the "controller instance"), which performs the actual task of applying filters to the selected visual object. Before the controller instance can apply a filter, it first needs to know what image or visual content the filter should be applied to. When the user selects an image, the application calls the `setFilterTarget()` method in the FilterWorkbenchController class, passing in one of the constants defined in the ImageType class:

```
public function setFilterTarget(targetType:ImageType):void
{
  ...
  _loader = new Loader();
  ...
  _loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
  targetLoadComplete);
  ...
}
```

Using that information the controller instance loads the designated file, storing it in an instance variable named `_currentTarget` once it loads:

```
private var _currentTarget:DisplayObject;

private function targetLoadComplete(event:Event):void
{
  ...
  _currentTarget = _loader.content;
  ...
}
```

When the user selects a filter, the application calls the controller instance's `setFilter()` method, giving the controller a reference to the relevant filter factory object, which it stores in an instance variable named `_filterFactory`.

```
private var _filterFactory:IFilterFactory;

public function setFilter(factory:IFilterFactory):void
{
  ...

  _filterFactory = factory;
  _filterFactory.addEventListener(Event.CHANGE, filterChange);
}
```

Notice that, as described previously, the controller instance doesn't know the specific data type of the filter factory instance that it is given; it only knows that the object implements the IFilterFactory instance, meaning it has a `getFilter()` method and it dispatches a `change` (`Event.CHANGE`) event when the filter changes.

When the user changes a filter's properties in the filter's panel, the controller instance finds out that the filter has changed through the filter factory's `change` event, which calls the controller instance's `filterChange()` method. That method, in turn, calls the `applyTemporaryFilter()` method:

```
private function filterChange(event:Event):void
{
  applyTemporaryFilter();
}

private function applyTemporaryFilter():void
{
  var currentFilter:BitmapFilter = _filterFactory.getFilter();

  // Add the current filter to the set temporarily
  _currentFilters.push(currentFilter);

  // Refresh the filter set of the filter target
  _currentTarget.filters = _currentFilters;

  // Remove the current filter from the set
  // (This doesn't remove it from the filter target, since
  // the target uses a copy of the filters array internally.)
  _currentFilters.pop();
}
```

The work of applying the filter to the display object occurs within the `applyTemporaryFilter()` method. First, the controller retrieves a reference to the filter object by calling the filter factory's `getFilter()` method.

```
var currentFilter:BitmapFilter = _filterFactory.getFilter();
```

The controller instance has an Array instance variable named `_currentFilters`, in which it stores all the filters that have been applied to the display object. The next step is to add the newly updated filter to that array:

```
_currentFilters.push(currentFilter);
```

Next, the code assigns the array of filters to the display object's `filters` property, which actually applies the filters to the image:

```
_currentTarget.filters = _currentFilters;
```

Finally, since this most recently added filter is still the "working" filter, it shouldn't be permanently applied to the display object, so it is removed from the `_currentFilters` array:

```
_currentFilters.pop();
```

Removing this filter from the array doesn't affect the filtered display object, because a display object makes a copy of the filters array when it is assigned to the `filters` property, and it uses that internal array rather than the original one. For this reason, any changes that are made to the array of filters don't affect the display object until the array is assigned to the display object's `filters` property again.