Flex Tutorial: Bitmaps & Filters

Flex is a powerful framework. The top layer graphics are easy to use and this level accessibility is maintained when delving deeper into the graphics suite. This is a guide to using Bitmaps and Filters in ActionScript 3. Basic knowledge of Actionscript and a rough understanding of Flex interfaces are assumed.

The Bitmap object

From the Language Reference :

The Bitmap class represents display objects that represent bitmap images. These can be images that you load with the flash.display.Loader class, or they can be images that you create with the `Bitmap()` constructor.

Importing images from the file system

Using the Loader class, you can import GIF, JPEG, or PNG files. The Loader class is an event driven system, used to prevent the player from accessing data that isn't available yet. The following code snippet shows how to load a file.

```
private var loader:Loader;
private var bitMap:Bitmap
public function Example(){
      loader = new Loader();
      loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
      loaded);
      var request:URLRequest = new URLRequest("this/url/string");
      loader.load(request);
}
```

First, the Loader class is initialized and the loaded function below is added to the object as an Event Listener. The loader is then fed a URLRequest (`flash.net.URLRequest`) to load.

```
private function loaded(event:Event):void {
      bitMap = Bitmap(loader.content
      loader.contentLoaderInfo.removeEventListener(Event.COMPLETE,
      loaded);
      loader = null;
}
```

The above function is added to the loader as an Event Listener and assigns the loader's content to bitMap, an instance of the Bitmap class. Optionally, you can add a listener for IO_Errors and Progress Events. The first enables handling of verbose error output, the latter allows you to make a progress bar.

Importing images from peripherals

Flex supports generating Bitmaps from camera streams. Using the Camera `(flash.media.Camera)` and Video `(flash.media.Video)` classes, you can use the Bitmap constructor to generate snapshots. The first snippet illustrates how to set up a Camera with the Video class.

```
private var video:Video;
private var videoHolder:UIComponent;

public function insertWebcamVideo():void{
      videoHolder = new UIComponent();
      private camera:Camera = Camera.getCamera();
      video = new Video(camera.width, camera.height);
      video.attachCamera(camera);
      videoHolder.addChild(video);
 }
```

As you can see, the Camera object is taken from the static method `Camera.getCamera()` and passed to the Video instance. The UIComponent videoHolder is only necessary if you want to make the stream visible. You can add the UIComponents to interface objects like Panels, something you cannot do with a video object directly.

```
public function takeSnapshot():Bitmap{
      private var snapshot:BitmapData = new BitmapData(video.width,
      video.height, true);

      snapshot.draw(video);
      private var snapshotbitmap:Bitmap = new Bitmap(snapshot);
      return snapshotbitmap;
}
```

The above function draws a snapshot from the video object into a BitmapData class (explained later) and then feeds it to a Bitmap constructor. This will contain a still image of the camera stream.

Making Bitmaps from scratch

The Bitmap constructor will accept a BitmapData object as an argument, generating a bitmap from that. To learn more about BitmapData, read the last section of this tutorial.

Viewing Bitmaps

Bitmaps cannot be added to interface objects directly. A UIComponent is required as a wrapper. A UIComponent can be created on its own as shown in the video instance, or you can use specialized containers such as Sprite or MovieClip.

Filters

Implementing the built-in filters in Flex is quite straightforward. There are nine filters in the API.

- Bevel filter (BevelFilter class)
- Blur filter (BlurFilter class)
- Drop shadow filter (DropShadowFilter class)
- Glow filter (GlowFilter class)
- Gradient bevel filter (GradientBevelFilter class)
- Gradient glow filter (GradientGlowFilter class)
- Color matrix filter (ColorMatrixFilter class)
- Convolution filter (ConvolutionFilter class)
- Displacement map filter (DisplacementMapFilter class)

Applying Filters

Some, like Bevel, have a basic set of parameters that you can pass as a string or an integer. Others, like the Convolution filter, require a matrix as function parameters. Applying a filter to a bitmap is simple. The snippet below shows a function that applies a blur filter to a Bitmap.

```
function applyFilterToBitmap(bitmap:Bitmap):Bitmap{
      private var blur:BlurFilter = new BlurFilter();
      blur.blurX = 10;
      blur.blurY = 10;
      blur.quality = BitmapFilterQuality.MEDIUM;
      bitmap.filters = [blur];
      return bitmap;
}
```

This blur filter has only three parameters. You can see that you are setting the `filters` array in the Bitmap object bitmap to `[blur]`. You can put multiple filters in this array to chain them. Note that you can also apply filters to other graphics containers, such as a Video object.

Using Complex Filters

The Blur filter used in the example is one of the simple native type-fed filters. Some filters require other types of data as input. The Gradient filters require an array of colors as one of the arguments. ColorMatrix and Convolution require a matrix as their primary parameter. The following snippet shows how to apply a Convolution filter.

```
function applyFilter(bitmap:Bitmap):Bitmap{{
      var matrix:Array = [0, -1, 0,
                          -1, 4, -1,
                           0, -1, 0];
      var convolution:ConvolutionFilter = new ConvolutionFilter();
      convolution.matrixX = 3;
      convolution.matrixY = 3;
      convolution.matrix = matrix;
      convolution.divisor = 1;
      bitmap.filters = [convolution];
}
```

This instance of the convolution filter creates an edge detection effect. The ‘area of effect’ is three pixels by three pixels, as defined by the matriX and matrixY parameters. The matrix itself is weighted so that the surrounding pixels are strongly averaged into the middle pixel. For other Convolution effects, you can visit the following page.
http://livedocs.adobe.com/flex/3/html/help.html?content=Filtering_Visual_Objects_17.html

Applying filters to BitmapData

Alternatively, you can apply filters directly to the underlying BitmapData objects:

```
function applyFilterToBitmapData(bitmapdata:BitmapData):BitmapData{
      private var blur:BlurFilter = new BlurFilter();
      blur.blurX = 10;
      blur.blurY = 10;
      blur.quality = BitmapFilterQuality.MEDIUM;
      bitmapdata.applyFilter(bitmapdata,new
      Rectangle(0,0,bitmapdata.width,bitmapdata.height),new Point(0,0),
      blur);
      return bitmapdata;
}
```

Doing this will allow you to access the image data post-filter. Getting the BitmapData from a Bitmap will otherwise always get a pre-filter set of pixel data. The last next section will explain more about BitmapData.

The BitmapData object

From the Language Reference :

The BitmapData class lets you work with the data (pixels) of a Bitmap object . You can use the methods of the BitmapData class to create arbitrarily sized transparent or opaque bitmap images and manipulate them in various ways at runtime. This class lets you separate bitmap rendering operations from the internal display updating routines of Flash Player. By manipulating a BitmapData object directly, you can create complex images without incurring the per-frame overhead of constantly redrawing the content from vector data.

BitmapData is a powerful class that allows you access to the array of pixels in a Bitmap. This array either represents a fully opaque bitmap or a transparent bitmap with an alpha channel. The core of the class is a buffer of 32-bit integers, each determining the properties of a single pixel in the bitmap. If you don't use the alpha channel, 24-bit integers are used.

Making a canvas

The constructor for BitmapData will create a black canvas (0x00000000) with an alpha layer. Like so :

```
new BitmapData(width, height);
```

This will create an instance with dimensions of your choosing. If you don't need transparency and need optimal performance, you can add a third argument.

```
new BitmapData(width, height, false);
```

This will create an instance without an alpha channel, using 24-bit integers. You can add a fourth argument, replacing the default black canvas with a fill color of your choosing.

Note that the maximum dimensions for BitmapData are 2880 by 2880 pixels. You might want to build in a check if your width and height arguments aren't hardcoded.

Manipulating bytes

In order to do analysis or generate image data, you need access to the raw bytes. The following snippet is a function to compare two Bitmaps to each other and return the similarity as a percentage of pixel-to-pixel equality. It is a nice example of how to use BitmapData for analysis.

```
public function compareBitmaps(bitmap1:Bitmap,bitmap2:Bitmap):int{
      if(bitmap1.height != bitmap2.height || bitmap1.width !=
      bitmap2.width){
            trace("bitmaps are not of equal size");
            return 0;
      }
      private var score:int=0;
      private var rect:Rectangle = new
      Rectangle(0,0,bitmap1.width,bitmap1.height);
      private var array1:ByteArray =
      bitmap1.bitmapData.getPixels(rect);
      private var array2:ByteArray =
      bitmap2.bitmapData.getPixels(rect);
      array1.position=0;
      array2.position=0;
      while(array1.bytesAvailable){
      if(array1.readUnsignedInt()==array2.readUnsignedInt())score++;
      }
      return (score/(bitmap1.height*bitmap2.width))*100;
}
```

The first if() statement is a check to prevent two Bitmaps with different dimensions to be compared to each other. It could have scaled the bitmap2 to the size of bitmap1 and count the difference in dimensions as part of the score, but bailing out was an arbitrary choice. In any case, it's less ambiguous this way.

The interesting part is in the ByteArray initialization. BitmapData has a `getPixels()` method that uses a Rectangle (`flash.geom.Rectangle`) object to pull a given part of the pixel array from BitmapData.

```
Rectangle(0,0,bitmap1.width,bitmap1.height);
```

In the line above, a rectangle with its pointer at the source (0,0) and the full size of the bitmap (width,height) was used to pull the entire image from the array. The rest of the code is a simple byte comparison in a while loop. Note that the pixels are in an Unsigned Int format (32-bit). For this purpose, it doesn't really matter since the images are only checked pixel by pixel for similarity, but if you wish to use the data afterwards, getting them in proper format is important. If you're not using the alpha channel, remember to read 24-bit sequences. You can use `readMultiByte()` in ByteArray for that purpose.

Using Manipulation Methods

The BitmapData class provides many methods for altering your image. You can fill rectangles with solid colors, clone images, and copy isolated channels. You can generate Perlin noise, scroll the image a few rows and/or columns, and so on. Most functions are self-explanatory or explained in the language reference. Explaining them all here would be superfluous.

An interesting example however, that is not explained well in the reference, is the threshold function. It tests pixel values in an image against a specified threshold and sets pixels that pass the test to new color values. It also returns an integer representing the amount of pixels that pass the test. Interestingly, that last fact is not documented in the language reference.

```
var testcolor:uint = 0x000000FF;
var color:uint = 0x00000001;
var maskColor:uint = 0xFFFFFFFF;
var pixelCount:int=newBitmapData.threshold(oldBitmapData, rect,
pt, ">", testcolor, color, maskColor, false);
```

The above code snippet illustrates the use of the threshold method to generate a binary image –not supported by BitmapData-. A binary image consists of only a single channel, with pixels either ‘on’, or ‘off’.

The threshold value is the minimum color you want to pass the test. Note that it would have been smarter to use a 3 channel 24-bit instance of BitmapData in the first place. Stripping down a 3 channel instance has the added benefit of allowing you to use the maskColor to isolate channels. In the last two lines, the threshold function is called from a BitmapData instance called newBitmapData. The arguments are as follows:

`oldBitmapData` is the source Bitmap. All the pixels that pass the text are set to color.
`rect` is a Rectangle object describing the dimensions on which to operate.
`pt` is the point within the destination image that corresponds to the upper-left corner of the source rectangle.
`">"` is a comparison operator. It follows standard practices. <, <=, >, >=, ==, !=
`testcolor, color, and maskColor` are evident, they are the color being tested against (in this case, blue), the color to replace the pixels that are larger than blue with, and the maskColor. maskColor is used to isolate colors. In this case, anything that has more color than fully transparent blue passes the test.

Of course, the resulting newBitmapData will still have 4 channels; the integers representing the pixels will be either 0 or 0x00000001. Generating a ByteArray from this is trivial, however.

Using these methods or direct byte manipulation, you could create your own filters. Unfortunately though, you cannot extend the BitmapFilter class so you’ll have to do this by hand. That’s not very practical, and therefore beyond the scope of this guide.

Further reading:
Adobe Bitmap and BitmapData primer :
http://livedocs.adobe.com/flex/3/html/help.html?content=Working_with_Bitmaps_03.html
Actionscript 3 Language Reference Bitmap entry :
http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/display/Bitmap.html
Flex 3 Language Reference Loader class :
http://livedocs.adobe.com/flex/3/langref/flash/display/Loader.html
Flex 3 Language Reference BitmapData class :
http://livedocs.adobe.com/flex/3/langref/flash/display/BitmapData.html