Multimedia Project

videoMixer Report

Nikolaos Poulios

Student No. : 2001527
MSc. Computer Science
ni.poulios@student.vu.nl

2011

Vrije Universiteit Amsterdam

Introduction

Scope of this project was the further development of the videoMixer application. videoMixer was built on the idea to learn and explore Adobe's Flex and Actionscript 3 platform capabilities for rich multimedia web applications development and specially image manipulation filters available on Flash 10. The application allows the user to mix two different sources of audio or video, create loops from parts of the clips and apply image filters on videos in real time.

The latest stage of development included the addition of extra filters for blending two images, shader image filters developed with the PixelBender toolkit, some simple examples of camera interaction demonstrating the methods of color and motion tracking, the ability to search youtube for videos and add them to the current playlists, changes and bug fixes on the previous version and a minor refinement of the graphical user interface.

This documents includes a description of all the major techniques, libraries, classes and components used in the programme and a guide on how and where in the source code are used.

Bitmap and BitmapData

The Bitmap class represents display objects that represent bitmap images. The BitmapData class includes all the pixels of a Bitmap object and allows us to retrieve or manipulate individual pixels or color channels in an image or blend two images together. A Bitmap object is defined by its BitmapData source object, which also determines its size on pixels. Combining the use of a single Bimap object with multiple BitmapData objects allow us to work on a traditional model of multiple signal sources ending on a single monitor with the user choosing the right source.

Using the **BitmapData.draw(source:IBitmapDrawable)**[1] we can store all pixels representing a display object and then feed that as source to a bitmap to display them. To use this model to display a video we have to add a loop function that will redraw the video object to the BitmapData on every video frame.

In the initial version of videoMixer two VideoDisplay objects were placed on a canvas for video playback and a slider was changing their alpha values to crossfade between the two videos. The current version supports video playback through the chromeless youtube video player. In this version the two VideoDisplay objects and the two SWFLoader objects including the youtube players remain invisible. In the **init**() function (**/lib/videoMixer.as**) of the application two BitmapData objects and two Bitmap object are initialized and added to a mx:UIComponent which represents the final output of the two videos. Each video source has two BitmapData of different sizes for the normal and full screen running mode. During the initialization of the application an enter frame event listener is added pointing at the onFrame function:

```
this.addEventListener(Event.ENTER_FRAME,onFrame);
```

[1] see Adobe Flex 3 Language Reference for a full list of possible arguments

In the onFrame function the programme determines the screen size mode and the right video/youtube source object to draw it on one of the corresponding BitmapData classes.

Blend Modes and the Blend Images Filter

BitmapData.draw function also allows us to blend two images together by drawing one BitmapData object into another, defining one of the available blending modes of Actionscript 3. The blend images filter (placed under "Others" on Filters Library and **/modules/BlendFilterInterface** on source code) demonstrates the following blend modes based on logical operations:

- **Add** : Adds the values of the constituent colors of the display object to the colors of its background, applying a ceiling of 0xFF
- **Multiply**: Multiplies the values of the display object constituent colors by the constituent colors of the background color, and normalizes by dividing by 0xFF, resulting in darker colors
- **Difference**: Compares the constituent colors of the display object with the colors of its background, and subtracts the darker of the values of the two constituent colors from the lighter value.
- **Subtract**: Subtracts the values of the constituent colors in the display object from the values of the background color, applying a floor of 0.
- **Screen**: Multiplies the complement (inverse) of the display object color by the complement of the background color, resulting in a bleaching effect.
- **Lighten**: Selects the lighter of the constituent colors of the display object and the colors of the background (the colors with the larger values).
- **Darken**: Selects the darker of the constituent colors of the display object and the colors of the background (the colors with the smaller values).
- **Invert**: Inverts the background.
- **Overlay**: Adjusts the color of each pixel based on the darkness of the background.
- **Hardlight**: Adjusts the color of each pixel based on the darkness of the display object.

The blend filter follows the same operational model as the rest of the filters with a small change as it has to be applied during the BitmapData draw operation. The **BlendFilterInterface** module triggers an event which cause the main application to enable blending during the **onFrame** function and apply the selected Blend Mode.

```
if(this._blend1Enabled) { vid1Data.draw(vid2Data,null,null,_blendMode1); }
```

Camera Interaction

In this section I will describe the use of BitmapData, filters and blend modes on analyzing pixel data from a camera to implement two image tracking methods based on color and motion detection and demonstrate some simple interaction with the application.

All functions implementing the camera tracker are placed inside the **/components/CameraTracker.mxml** and **/lib/cameraTrackerListeners.as** files. By the CameraTracker panel the user selects which source to control and select three different modes of interaction:

- **Play previous/next**: the user moves to the previous or next track on the current play list by moving her hand left or right
- **Video fader**: The user can handle the fader between the two videos by moving her hand left or right
- **Color Picking**: When activated an overlay palette appears over the camera monitor allowing the user to apply a certain color matrix filter to the video by moving her hand along the palette

The component communicates with the main application using custom events, handled by functions placed on the /lib/cameraTrackerListeners.as file.

During the initialization of the CameraTracker the camera is attached to a video with size 160x120 pixels and the BitmapData and Bitmap objects that will include the pixels of the video feed are created. When enabled the CameraTracker starts a timer the delay of which defines the sampling rate of the video feed. Before executing one of the two tracking methods described later we apply two basic filters on the image, a **blur filter** which makes tracking easier on both methods and a matrix manipulation to mirror the image in order to follow the movement from the user perspective.

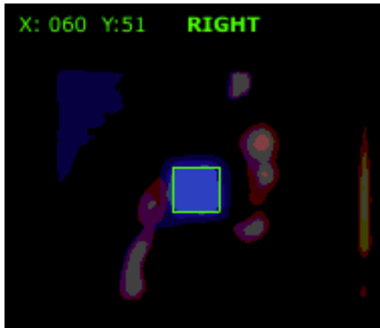
```
_vidTrack.filters = [new BlurFilter(10,10,1)];
```

```
var mirrMatrix:Matrix = new Matrix(-1, 0, 0, 1, _bmpdata.width, 0);
```

On Timer event function:

```
_bmpdata.draw(_vidTrack, mirrMatrix,null,null,null,true);
```

Color Tracking



When color tracking method is selected on the camera tracker panel the user clicks on the image monitor to the color she wants to track. The idea is that the user places a brightly colored item in front of the camera and clicks on the color to start tracking it. The programme gets the color of the object by executing the method:

```
_color=_bmpdata.getPixel(event.localX, event.localY);
```

when the user clicks on the camera monitor.

In order to make color tracking easier we reduce the color palette of the camera feed. The function **makePaletteArrays()** creates an array for each color channel to be applied on the BitmapData channels to floor each pixel's color value according to the defined level of detail. For example if we choose levels =8, we break each channel down to 8 values, for a total of 2,048 colors.

```
private function makePaletteArrays():void
{
        _red = new Array();
        _green = new Array();
        _blue = new Array();
        var levels:int = 8;
        var div:int = 256/levels;
        for(var i:int =0; i<256; i++)
        {
                var value:Number = Math.floor(i/div)*div;
                _red[i] = value << 16;
                _green[i] = value << 8;
                _blue[i] = value;
         }
}
```

These arrays are later applied to the BitmapData at the draw loop by the function:

```
_bmpdata.paletteMap(_bmpdata, _bmpdata.rect, new Point(), _red, _green,
_blue);
```

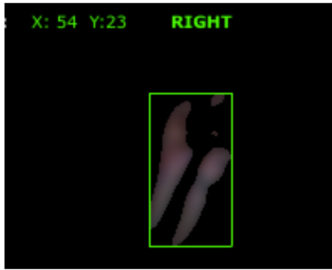
The palleteMap method takes each of the red, green, and blue (and even alpha if you want) channels of a bitmap and maps their values to another array.

The next step in the color tracking is to simply execute the method:

```
_bmpdata.getColorBoundsRect(0xffffff, _color, true);
```

which returns a rectangle defining the area that fully encloses the selected marker color, and draw that rectangle in the camera monitor.

Motion Tracking



Motion tracking detects changes between two consecutive frames two determine that something moved in front of the camera. To perform this we create three BitmapData objects. One object is for storing the current frame, one for the previous frame and a third one in which we combine the other two, using the Difference blend mode.

```
_blendFrame.draw(_oldFrame);
_newFrame.draw(_vidTrack,mirrMatrix);
_oldFrame.draw(_newFrame);
_blendFrame.draw(_newFrame, null,null, BlendMode.DIFFERENCE);
```

Again here we apply an alternative option threshold method against each pixel of the blended frame to reduce the colors

```
_blendFrame.threshold(_blendFrame, _blendFrame.rect, new Point(), "<",
0x00330000, 0xff000000, 0x00ff0000, true);
```

The last part is also the same as the color tracking method, we just call the method:

```
_blendFrame.getColorBoundsRect(0xffffff, 0x000000, false);
```

and draw the returned rectangle on the monitor screen.

Simple gesture tracking

The camera tracker "play previous/next" mode works on a simple algorithm that every time compares a x and y coordinate of the center of tracked rectangle with its previous position. The algorithm takes a set of initial coordinates and then compares each position with each previous one. If the x position is always greater than its previous it means that the object keeps moving right, otherwise it moves to the left. Every time there is a change towards the other direction the algorithm resets and the current position becomes the current initial position. In the algorithm the distance between the current and the initial position is measured and if this is greater than 50 pixels a left or right gesture is recognized. The algorithm also sets a maximum limit on the deviation along the y axis to 40 pixels so that a gesture is made along a more or less straight line in order to avoid a lot of accidental gesture triggers. The algorithm is implemented on the **gestureTracking(posx:int, posy:int)** function in the file **/components/CameraTracker.mxml** file.

Additional Filters

This section provides a description on all the new filters added to the programme and a guide on how to use PixelBender toolkit to write shaders for Flash 10 and import them to an actionscript application.

All interfaces for filters are implemented as mxml modules and placed on the **/modules** directory. Each category of filter communicates with the main application through custom event classes placed on the **/lib/FilterEvents** directory, carrying possible filter parameters, and handled by the corresponding functions in **/lib/FilterListeners** directory.

ASCII Art filter

This filter uses the Asciify class **(com.oaxoa.fx.Asciify** file)developed by Pierluigi Pesenti which transforms images to ASCII characters. Asciify takes a display object as a parameter and maps an area of pixels (depending on the defined pixelSize) to a character according to the color of the pixels. The final string formed by this method is displayed on a Text object on screen with the same size as the image.

Asciify produces a nice image effect but requires a lot of calculations and uses only ActionScript classes reducing dramatically the performance and responsiveness of the application when applied to a video. My research on how to create more advanced image filters for flash and looking alternatives for Asciify lead me to use Adobe's PixelBender toolkit which I describe in the following section.

The PixelBender

Pixel Bender is a toolkit provided by Adobe to write, test and compile shaders that can be used in other Adobe's products like Photoshop, After Effects and Flash. The advantage of pixel shaders over Actionscript filters is that pixel shaders perform calculations to determine the value of a single pixel in parallel for all pixels of an object. Shaders written in PixelBender are compiled to be very optimized and run in a separate process from the Flash Player. For that reason shaders are a lot more faster than Actionscript filters.

In order to create a pixel shader for flash you have to write a filter kernel in PixelBender which uses a language based on the C programming language and has its own data types, compile it and export it as byte code which can be imported to Actionscript. Before describing the examples of sharers included in the application, I will describe how the application imports a shader to be used as an image filter.

The **/shaders/kernel codes/** directory provides the kernel codes of all filters used by the application and in the shaders directory there are all the filters byrecode .pbj files that are used to import a shader into Actionscript. The directory also includes a class named **ShaderProxy.as.** The ShaderProxy class extends the proxy and event dispatcher classes of Actionscript and acts as an interface between Flash and the actual shader. The ShaderProxy takes as argument an embedded bytecode file class or the url to the corresponding .pbj file. After loading the bytecode through the embedded class or a URLLoader, the ShaderProxy creates a new shader based on the imported byte code:

```
_shader:Shader = new Shader(data:ByteArray)
```

The rest of the methods of ShaderProxy are used to pass parameters on the shader using the set and get Parameter method and dispatch internal events. The method **getShader()** returns the loaded shader. In order to use a shader as a filter, a shader filter must be created:

```
import flash.filters.ShaderFilter;
import shaders.ShaderProxy;
import flash.display.Shader;

private function applyShader():void

{

    //set parameters
   if(_shaderProxy.percent != paramSlider.value)
       _shaderProxy.percent = paramSlider.value;
  var shader:Shader = _shaderProxy.getShader();
//create filter with shader
_shaderFilter = new ShaderFilter(shader);
filterEvnt = new ShaderFilterEvent(_shaderFilter,"ShaderFilterEvent");
dispatchEvent(filterEvnt);
}
```

From that point the shader filter created can be added to an filter array just as any other actionscript filter.

Desaturate filter

The above code example is the code used on the **DesaturateFilterInterface** module. It is a simple filter that desaturates an image taking a single parameter called percent. Bellow it's the filters kernel code in PixelBender:

```
<languageVersion : 1.0;>
kernel Desaturate
<   namespace : "info.psylus";
    vendor : "Psylus";
    version : 1;
    description : "Desaturates an image by a specified amount."; >

{  input image4 src;
    output pixel4 dst;
    const float3

greyValues = float3(0.3, 0.59, 0.11);
 parameter float percent
  < minValue:      0.0;
     maxValue:      1.0;
     defaultValue:  1.0;
    description:    "Percent image should be desaturated.";
>;
void evaluatePixel() {
     pixel4 px = sampleNearest(src,outCoord());
     float distanceToDesaturatedValues = dot(px.rgb, greyValues);
     float3 fullDesaturate = float3(distanceToDesaturatedValues);
      float3 noDesaturate = px.rgb;
      px.rgb= mix(noDesaturate, fullDesaturate, percent);
      dst = px;    } }
```

In the code above, typical elements of a shader code is in Bold. image4 and pixel4 are PixelBender data types that define an input image containing 4 channels (red,green,blue and aplha) and an output destination pixel containing 4 values. Parameters are defined by defining their name, minimum, maximum and default values. The function evaluatePixel is where all the calculations to determine a pixel value go. The function sampleNearest(src.outCoord()); gets the nearest pixel value to a x, y location in an image.

In this particual example of a desaturate filter the filter calculates the distance of the sampled pixel values and the fully desaturated values and the mix function retruns a linear interpolation between current and fully desaturated values:
noDesaturate*(1.0-percent)+fullDesaturate*percent)

Invert RGB

The Invert RGB filters is one of the example filter of the PixelShader toolkit. Its evaluatePixel function calculates the inverse value for every color channel.

```
void     evaluatePixel()     {
       float4 inputColor = sampleNearest(src, outCoord());
       dst.rgb = float3(1.0, 1.0, 1.0) - inputColor.rgb;
       dst.a = inputColor.a;  }
```

Pixelate

The Pixelate filter is also one of the example files in the PixelBender toolkit. It takes a dimension parameter. Inside the evaluatePixel function the location of the evaluated pixel truncates to the value at the top right corner of the square defined by distance. Then the pixel located at the resulted location is sampled and its value is passed as the color value of the evaluated pixel.

```
 evaluatePixel()     {
          float dimAsFloat = float(dimension);
          float2 sc = floor(outCoord() / float2(dimAsFloat, dimAsFloat));
          sc *= dimAsFloat;
         outputPixel = sampleNearest(inputImage, sc);     }
```

Stamp Filter

The Stamp filter produces a nice comic like effect by using two different colors for the foreground and background of an image. Besides the two colors the filter takes a threshold parameter against which the luminance of every pixel is compared in order to determine if they belong to the foreground or background of the image.

```
 evaluatePixel()     {
   float2 coord = outCoord();
   pixel4 px = sample(source, coord);
   float numLevels = 2.0;
   px = floor(px*numLevels)/numLevels;
   float luminance = px.r * 0.3086 + px.g * 0.6094 +  px.b * 0.0820;
   if (luminance <= levelsThreshold) {
         px = backgroundColor;
   } else {
            px = foregroundColor;
   }
   result = px;
}
```

Chroma Key

Using chroma key filter or "greenbox technique" we can select a color to exclude from the picture and a threshold on the range of color values that the filter is applied.

```
evaluatePixel()  {
  float4 color = sampleNearest(src,outCoord());
  float dist = distance(color, keyColor);
  if (dist <= levelsThreshold) {
            dst = float4(0.0, 0.0, 0.0, 0.0);
   } else {
        dst = color;         }
}
```

Light Bright

The light bright effect draws lighter areas of an image as pegs of color, producing a nice kind of LED wall effect.

```
evaluatePixel()     {
         float2 coord = outCoord();
        pixel4 px = sample(source, coord);
        float numLevels = 4.0;
        px = floor(px*numLevels)/numLevels;
        int modX = int(mod(coord.x, 5.0));
        int modY = int(mod(coord.y, 5.0));
        if ((modX == 2 && (modY > 0 && modY < 4)) ||
             (modY == 2 && (modX > 0 && modX < 4)) ) {
            float luminance = px.r * 0.3086 + px.g * 0.6094 + px.b * 0.0820;
             if (luminance <= levelsThreshold) {
               px = backgroundColor;
        } else {
                px.rgb = mix(px.rgb, float3(1.0, 1.0, 1.0), float3(0.2, 0.2, 0.2));
       }
       } else {
                     px = backgroundColor;
        }
             result = px;
    }
```

ASCII Shader (ASCII Art revised)

This filter is the shader version of the previously described ASCII Art filter. It was written by Richard Zurad. The filter take as parameters an input fontmap image, and two parameters character count and size for the detail of the filter. The fontmap image is an image of 256 8x8 pixel cells of ASCII characters in rows of 16 cells ordered from left to right, top to bottom by order of brightness. The filter maps an area of pixels to a character from the bitmap according to their luminance.

```
void evaluatePixel() {
     float sizef = float(size);
     float charCountf = float(charCount);
     float2 offset2 = mod(outCoord(), sizef);
     pixel4 mosaicPixel4 = sampleNearest(src, outCoord() - offset2);
     float luma = 0.2126 * mosaicPixel4.r + 0.7152 * mosaicPixel4.g + 0.0722 *
mosaicPixel4.b;
     float range = (1.0 / (charCountf - 1.0));
     float fontOffset = sizef * floor(luma / range);
     float fontmapsize = (sizef * floor(sqrt(charCountf)));
     float yRow = floor(fontOffset / fontmapsize);
     offset2.y = offset2.y + (sizef * yRow);
     offset2.x = offset2.x + (fontOffset - (fontmapsize * yRow));
     pixel4 charPixel4 = sample(text, offset2);
    dst.rgb = mosaicPixel4.rgb * charPixel4.rgb;
    dst.a = mosaicPixel4.a;     }
```

YouTube API

The youtube search panel is implemented at the **/components/ YouTubeSearchComponent.mxml** file. It makes use of the as3corelib library by Mike Chamber which contains classes for MD5 and SHA 1 hashing, Image encoders, and JSON serialization and the youtube actionscript api classes developed by Martin Legris.

The panel provides a search field, a data grid to view the search result and a swfLoader including the youtube chromeless player for previewing the results. The user has to drag and drop a clip from the data grid to the corresponding play list combo box. When the application is loaded it reads a predefined playlist of local files in XML format. Each file has the bellow format:

```
<clip>
	<title> Clip Title </title>
	<file> Path to file</file>
	<type>video/audio/youtube</type>
</clip>
```

When the user drops a clip on a combobox a new XML node is created including the title of the clip, its unique id in you tube as path and youtube as type. The combo box drop event handler and all the function related to the youtube players and their playback control in the main application are located in the **/lib/youTubeFunctions.as** file. The **/lib/YouTubeInterface.as** class takes an swfLoader object as constructor parameter, automates the loading of the youTube player. The application includes two instances of the YouTubeInterface class providing methods for playback control and size parameters for each of the two youtube players included.

The Sandbox violation

As mentioned before the application uses the Bitmap and BitmapData classes to display videos and apply filters on them. When a youtube player is activated the **onFrame** function draws the corresponding swfLoader on the deck's BitmapData object. Unfortunately Flash 10 includes a new security policy in which calls like the BitmapData.draw() method on a swfLoader which contains content from a different domain produce a security sandbox violation and they are blocked. YouTube does not allow to get BitmapData from its player. The application works normaly when running under the Flex Builder IDE producing only the sandbox violation message on the debugger console but when deployed to a remote server, youtube playback does not work at all. I tried a workaround using a local php proxy script for youtube requests and crossdomain.xml policy without any success. It should be noted that vimeo online video provider allows such calls through their crossdomain policy but their api provides an interface to laod videos of a user or by a specific group/channel but not a service to search for their whole video library.

References

Chapter 5: Alternate Input:The Camera and Microphone - AdvancED ActionScript 3.0 Animation. Keith Peters. friendsodED. ISBN: 978-1-4302-1608-7

Chapter 5: Pixel Bender and Shaders - Foundation ActionScript 3.0 Image Effects. Todd Yard. friendsofED. ISBN: 978-1-4302-1872-2

AsciiFy – AS3 Ascii Art Class : http://blog.oaxoa.com/2008/03/04/asciify-actionscript-3-as3-ascii-art-class/

AsciiMi: A Pixel Bender Ascii Art Shader : http://asciimii.greyboxware.com/

YouTube ActionScript 3.0 Player API Reference :

http://code.google.com/apis/youtube/flash_api_reference.html

Class for youtube player Basic code taken by Tour de Flex YouTube API Sample: http://www.adobe.com/devnet-archive/flex/tourdeflex/web/ -docIndex=0;illustIndex=3;sampleId=19810

libraries used in this example: http://code.google.com/p/as3-youtube-data-api/by Martin Legris