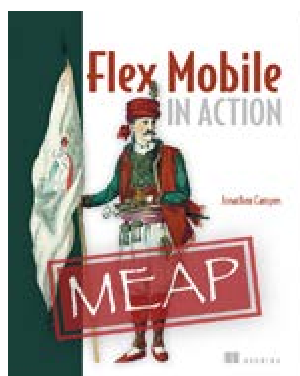
MANNING PUBLICATIONS



Flex Mobile in Action
By Jonathan Campos

With hardware that would make even James Bond jealous we can take stunning pictures with quality that rivals many cameras on the market. In this article based on chapter 4 of Flex Mobile in Action, author Jonathan Campos discusses three ways to get camera data into your mobile applications.

To save 35% on your next purchase use Promotional Code **campos0435** when you check out at www.manning.com/.

You may also be interested in…

Taking the Same Picture Three Different Ways

With hardware that would make even James Bond jealous we can take stunning pictures with quality that rivals many cameras on the market. When taking a picture there are three ways to get camera data into your mobile applications, leading to a variety of questions about which will work for you at the quality you expect.

Do you want to stay inside your application or use the hardware's built in camera user interface to take the picture?

If you want to stay in your application rather than leave, then you will need to use the `getCamera()` method. If you want to have the "native" experience then you should use the options provided by the Camera UI.

Do you need to take a picture or access one that already exists in the device's memory?

If the image needs to be taken then you can use either the `getCamera()` method or the Camera UI. If your picture is in the device's memory, then you will need to pull the image from the Camera Roll.

Should I use a custom camera or the built-in camera?

The quality of the image is based on the camera; however, you may find it easier to use the device's built-in Camera UI. If you use the `getCamera()` method and build your own camera interface, you will have to fully implement any zooming, file storage, or other functions. While it isn't impossible to implement these functions, this may be more effort than you want to attempt.

Table 1 provides a quick breakdown of the various camera options.

Table 1 Camera options

Method	Parameters
CameraUI	Leaves your application
	Uses native camera UI
	Takes a new picture

CameraRoll	Leaves your application
	Uses native camera roll
	Pulls images in your device
`getCamera()`	Stays in your application
	You build the UI
	Memory intensive
	Pulls bitmap data for you to manipulate

In this article, we will look at the three different ways we can access our camera and the images we've taken along with the best practices for mobile performance.

Using the CameraUI the cross-platform way

The first method to access the camera data is to use the CameraUI, which accesses the device's native camera user interface. The following listing shows the basic structure for CameraUIView.mxml. Next, we will break down the various methods to process the camera data.

Listing 1 Basic view structure

```
<?xml version="1.0" encoding="utf-8"?>
<s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
     xmlns:s="library://ns.adobe.com/flex/spark"
     title="Camera UI"
     viewActivate="_onView_ViewActivateHandler(event)">
     <fx:Script>
     <![CDATA[

     . . . imports . . .

     private var _camera:CameraUI;
private var _loader:Loader;

protected function _onView_ViewActivateHandler(event:ViewNavigatorEvent):void
{
       //viewActivate Handler Implementation #1
}

private function _onButton_ClickHandler(event:MouseEvent):void
{
       //button click Handler Implementation #2
}

private function _onCamera_CompleteHandler(event:MediaEvent):void
{
       //camera complete Handler Implementation     #3
}

private function _onMediaPromise_LoadedHandler(event:Event):void
{
       //mediaPromise load Handler Implementation   #4
}

       ]]>
       </fx:Script>

       <fx:Declarations>
       <!-- Place non-visual elements (e.g., services, value objects) here -->
       </fx:Declarations>

       <s:layout>
              <s:VerticalLayout verticalAlign="middle" #5
                     horizontalAlign="center"
                     gap="10"/>
```

```
        </s:layout>

        <s:Button label="Take Picture"        #6
                         click="_onButton_ClickHandler(event)"
                         id="button"/>

        <s:Image id="image"/>  #7

    </s:View>
```

#1 viewActivate event handler
#2 Button click event handler
#3 Camera operation complete handler
#4 mediaPromise load handler
#5 View layout
#6 Button to take picture
#7 Component to load camera image

The current CameraUI simply lays out a button to start the picture capturing process and an image component to display the captured picture.

When the view is activated, the first thing we need to do is determine if our device even has access to a camera. If the device doesn't have access to the camera, we won't want to allow the user to even take a picture.

```
protected function _onView_ViewActivateHandler(event:ViewNavigatorEvent):void
{
   button.enabled = CameraUI.isSupported;
}
```

With the knowledge that our device has access to a camera, we need to respond to the button click if a user chooses to take a picture.

```
private function _onButton_ClickHandler(event:MouseEvent):void
{
   if(!_camera)
   {
        _camera = new CameraUI();
        _camera.addEventListener(MediaEvent.COMPLETE, _onCamera_CompleteHandler);
   }
   _camera.launch(MediaType.IMAGE);
   //_camera.launch(MediaType.VIDEO);
}
```

With the `CameraUI` class instance, we will use the `launch()` method to access the CameraUI. The `launch()` method takes a parameter that says whether you are intending to capture an image or video. Once the `launch()` method is called, we will leave our application and go to the native camera user interface. Before this happens, we need to add an event listener for the `COMPLETE` event to get data from our capture process.

When the user finishes capturing the picture or video, the complete event returns a `MediaPromise` object. This is very important because what is returned isn't an actual image but the information of how to load the media file. If you just want access to the media file's metadata and file location, you can stop now; but, if you want to access the actual media data, then we need to load the media file.

```
private function _onCamera_CompleteHandler(event:MediaEvent):void
{
   var mediaPromise:MediaPromise = event.data;

   if(!_loader)
   {
        _loader = new Loader();
        _loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
   _onMediaPromise_LoadedHandler);
   }

        _loader.loadFilePromise(mediaPromise);
   }
```

Finally, with the media loaded we can show the picture using the image component.

```
   private function _onMediaPromise_LoadedHandler(event:Event):void
   {
        var loadedInfo:LoaderInfo = event.target as LoaderInfo;
        image.source = loadedInfo.loader;
   }
```

With our media loaded and shown to the user, our CameraUI method is complete. Some developers may stop at the media complete handler and that would work on an Android device. On an iOS device, this isn't enough to show your image because the MediaPromise isn't actually loaded. Since we want to make sure our code works across all devices, we will need to use the entire workflow to show our captured media.

CAMERA PERMISSIONS

At the end of this article, we will focus on the permissions required to access the camera on QNX and Android devices; iOS devices require no special permissions for the camera.

Using the getCamera() method

The second method to access the camera is the getCamera() method, which was originally used to access webcams and built-in computer cameras. A benefit to using this function is that you can reuse the code even for the desktop or web applications. The following listing shows the base view for the getCamera() method. For this view, we will include a window to show the camera's data and give the user the ability to switch between the front and back camera.

Listing 2 GetCameraView.mxml

```
   <?xml version="1.0" encoding="utf-8"?>
   <s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="getCamera()"
        viewActivate="_onView_ViewActivateHandler(event)">
        <fx:Script>
        <![CDATA[
        . . . imports . . .

protected function _onView_ViewActivateHandler(event:ViewNavigatorEvent):void
{
        //viewActivate Handler Implementation        #1
}

protected function _onButtonBar_ChangeHandler(event:IndexChangeEvent):void
{
        //buttonbar change Handler Implementation            #2
}

private function _getCamera():void
{
        //buttonbar change Handler Implementation            #3
}

private var _video:Video;
private var _videoHolder:UIComponent;

private function _attachCamera(camera:Camera):void
{
        //buttonbar change Handler Implementation            #4
}

        ]]>
        </fx:Script>

        <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
        </fx:Declarations>

        <s:layout>
                <s:VerticalLayout verticalAlign="middle" #5
                        horizontalAlign="center"
                        gap="10"/>
        </s:layout>

        <s:Group width="320" height="240"            #6
                         id="videoDisplay"/>
```

```
    <s:ButtonBar id="buttonBar" requireSelection="true"         #7
                        change="_onButtonBar_ChangeHandler(event)">
            <s:dataProvider>
                    <s:ArrayList>
                            <fx:Object label="{CameraPosition.FRONT}"/>
                            <fx:Object label="{CameraPosition.BACK}"/>
                    </s:ArrayList>
            </s:dataProvider>
    </s:ButtonBar>

</s:View>
```

#1 viewActivate Handler
#2 ButtonBar change handler
#3 Get the selected camera
#4 Attaches camera to view
#5 View layout
#6 Video display holder
#7 Camera position select bar

The GetCameraView lays out a video display container and a ButtonBar to select which camera to show in the video display—either front or back camera—in a vertical layout.

When the view activates, the first thing we do is determine if the Camera is supported on the device. If the device supports a camera, then we activate the ButtonBar and get the currently selected camera for the video display.

```
protected function _onView_ViewActivateHandler(event:ViewNavigatorEvent):void
{
        buttonBar.enabled = Camera.isSupported;
        if(Camera.isSupported)
                _getCamera();
}
```

When the user selects a different camera position using the ButtonBar, we select the proper camera using the `_getCamera()` method.

```
protected function _onButtonBar_ChangeHandler(event:IndexChangeEvent):void
{
        _getCamera();
}
```

The important part of this view is the code to connect the camera data to the view. With this data shown on screen, we can easily copy the image from the display and store it to a file (see listing 3).

Listing 3 GetCameraView.mxml _getCamera() and _attachCamera() methods

```
private function _getCamera():void
{
        var names:Array = Camera.names; #1
        var i:int = -1;
        var n:int = names.length;

        while(++i<n)
        {
                var cam:Camera = Camera.getCamera( names[i] as String);
                if(cam && cam.position == buttonBar.selectedItem.label) #2
                {
                        _attachCamera(cam);
                        return;
                }
        }
        _attachCamera(Camera.getCamera()); #3
}

private var _video:Video;
private var _videoHolder:UIComponent;

private function _attachCamera(camera:Camera):void
{
        //setup camera
camera.setMode(400, 300, 15, false); #4
```

```
    camera.setMotionLevel(0);

    if(!_video)
    {
         _video = new Video(320, 240); #5

         var m:Matrix = new Matrix(); #6
         m.rotate(Math.PI/2);
         _video.transform.matrix = m;
    }

    _video.attachCamera(camera); #7

    if(!_videoHolder)   #8
    {
         _videoHolder = new UIComponent();
         _videoHolder.x = 0;
         _videoHolder.y = 0;
         _videoHolder.x = ((videoDisplay.width/2) - (_video.width/2)) + _video.width;
         _videoHolder.y = ((videoDisplay.height/2) - (_video.height/2)) - 50;
         _videoHolder.addChild(_video);
         videoDisplay.addElement(_videoHolder);
    }
}
```

#1 Gets the list of available cameras
#2 Pulls the selected camera
#3 Attaches a camera to the view
#4 Sets the camera size and refresh rate
#5 Video to display the camera data
#6 Rotates the image right side up
#7 Attaches camera to video
#8 Adds the video to the display list

The `_getCamera()` and `_attachCamera()` methods work synchronously, one after the other. `_getCamera()` uses the list of available cameras and pulls the camera based on the selected item in the ButtonBar. With the proper camera selected, we use the `_attachCamera()` method to first set the camera's properties. With the camera set up properly we need to show it to the user. Here, we get into two tricks needed to show the camera's data to the user.

The first trick is to use the `Video` class to show the camera's data. One issue with mobile phone camera's is that the image is automatically rotated. To show the image in the right orientation, we need to transform the image, rotating it, to adjust the image's orientation.

The second trick is needed to add the `Video` class to the video display container. Flex containers require an `IVisualElement` child—the base visual interface for Flex components. The `Video` class doesn't implement the IVisualElement interface, so we need to add the `Video` class to a UIComponent,

WARNING!

If you find the performance of the `getCamera()` method on your device is slow, don't be surprised. Currently the process to show data from the camera is labor and memory intensive as it continuously redraws the bitmap data to the screen. If this is an issue on your device you may prefer to use the CameraUI method.

We've now created the beginnings of our own custom camera user interface. There are many more ways to customize and add features but this serves as a good starting point for our applications. Next, we will look at pulling images from the CameraRoll.

Accessing the device's camera roll

The third way to access data from the camera is to pull data from the camera roll. While it is also possible to use the file system to access camera data this is a central place to access images and video taken from the device's camera without having to search through the file system for images. The following listing shows the basic layout for the CameraRollView.mxml we will use to access your device's camera roll.

Listing 4 CameraRollView.mxml

```
   <?xml version="1.0" encoding="utf-8"?>
   <s:View xmlns:fx="http://ns.adobe.com/mxml/2009"
        xmlns:s="library://ns.adobe.com/flex/spark"
        title="Camera Roll"
        viewActivate="_onView_ViewActivateHandler(event)">
        <fx:Script>
        <![CDATA[
        import spark.events.ViewNavigatorEvent;

        private var _cameraRoll:CameraRoll;
        private var _loader:Loader;

protected function _onView_ViewActivateHandler(event:ViewNavigatorEvent):void
{
   //viewActivate handler method             #1
}

private function _onButton_ClickHandler(event:MouseEvent):void
{
   //button click handler method             #2
}

private function _onCameraRoll_SelectHandler(event:MediaEvent):void
{
   //camera roll select handler method       #3
}

private function _onMediaPromise_LoadedHandler(event:Event):void
{
   //media promise handler method            #4
}

        ]]>
   </fx:Script>

        <fx:Declarations>
        <!-- Place non-visual elements (e.g., services, value objects) here -->
        </fx:Declarations>

        <s:layout>
                <s:VerticalLayout verticalAlign="middle" #5
                        horizontalAlign="center"
                        gap="10"/>
        </s:layout>

        <s:Button label="Get Picture"         #6
                         click="_onButton_ClickHandler(event)"
                         id="button"/>

        <s:Image id="image"/>  #7

   </s:View>
```

#1 viewActivate handler
#2 Button click handler
#3 Camera roll select handler
#4 Media promise loader handler
#5 View Layout
#6 "Get Picture" Button
#7 Image Component

The CameraRollView contains a button to initiate the camera roll selection code and an image to display the camera roll data when it is returned.

As with the previous two camera methods, the first thing we need to do is to determine if our device has access to a camera roll.

```
protected function _onView_ViewActivateHandler(event:ViewNavigatorEvent):void
{
   button.enabled = CameraRoll.supportsBrowseForImage;
}
```

After we are sure that our device supports the camera roll, we then need to respond to a user initiating the camera roll process. To start the process, we need to use the CameraRoll and listen for the `SELECT` event. The `SELECT` event is fired when a user selects a specific piece of media in the camera roll. When we are ready we call the `browseForImage()` method to access the Camera Roll UI. As soon as the Camera Roll UI is launched, our application loses focus and we need to respond to the "select" handler to retrieve any selected media.

```
private function _onButton_ClickHandler(event:MouseEvent):void
{
   if(!_cameraRoll)
   {
        _cameraRoll = new CameraRoll();
        _cameraRoll.addEventListener(MediaEvent.SELECT, _onCameraRoll_SelectHandler);
   }
   _cameraRoll.browseForImage();
}
```

When a user selects their media our application, will have focus again and, as in the CameraUI example, we will need to load the returned MediaPromise object.

```
   private function _onCameraRoll_SelectHandler(event:MediaEvent):void
   {
        var mediaPromise:MediaPromise = event.data;

        if(!_loader)
        {
               _loader = new Loader();
               _loader.contentLoaderInfo.addEventListener(Event.COMPLETE,
   _onMediaPromise_LoadedHandler);
        }
        _loader.loadFilePromise(mediaPromise);
   }
```

Once the MediaPromise file successfully loads the media data our final load handler is called and we can set the data to the image component and show the selected media to our user.

```
   private function _onMediaPromise_LoadedHandler(event:Event):void
   {
        var loadedInfo:LoaderInfo = event.target as LoaderInfo;
        image.source = loadedInfo.loader;
   }
```

With our Camera Roll data returned and displayed to the user, we can move forward and look at the permissions required to access the device's camera.

Camera permissions

As we know each platform includes its own permissions section, either in the application descriptor file or the blackberry-tablet.xml. We will look at the various platforms requirement. For the iOS platform, there are no special permissions to include the camera permissions.

Android

For the Android platform, we need to go to the application descriptor file titled DeviceApp-app.xml and find the `<android/>` tag. Within the manifest tag, we need to add a single permission tag to enable camera access in Android devices.

```
   <android>
   <manifestAdditions><![CDATA[
   <manifest android:installLocation="auto">
   . . .
   <uses-permission android:name="android.permission.CAMERA"/>
   . . .
   </manifest>
   ]]></manifestAdditions>
   </android>
```

This one tag is all that is required to access the camera on Android devices.

QNX (BlackBerry)

For QNX devices we need to go to the blackberry-tablet.xml file and add a few `<action/>` tags within the main `<qnx/>` tag.

```
<qnx>
     <action>use_camera</action>
     <action>access_shared</action>
</qnx>
```

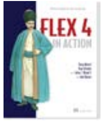
To use the CameraUI or the `getCamera()` method, we need add both the `use_camera` and the `access_shared` action. If we want to access just the CameraRoll, then we will only need to use the `access_shared` action in our QNX tag.

Adding the camera permissions completes the three different ways you can access data from a mobile device's built-in camera.

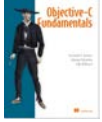
Summary

Each device type has its own permissions. We showed you the required permission tags for the device's camera. A key takeaway is that you make sure to load the MediaPromise file from your Camera for cross-device functionality.

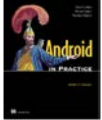
Here are some other Manning titles you might be interested in:



Flex 4 in Action
Tariq Ahmed, Dan Orlando, John C. Bland II, and Joel Hooks



Objective-C Fundamentals
Christopher K. Fairbairn, Johannes Fahrenkrug, and Collin Ruffenach



Android in Practice
Charlie Collins, Michael D. Galpin, and Matthias Kaeppler

Last updated: November 14, 2011