

APPENDIX A: WEBCAMS AND MICROPHONES

Ever since Flash 6, users have been able to make use (to a limited extent) of the cameras and microphones connected to their computer. The original purpose of these features was use with the Flash Media Server, a multiuser platform for streaming media between users. Because Adobe has not added much functionality to these components since their creation, little more can be done with them today than when they were first released. In fact, one might wonder why they're being brought up in a book about game development at all.

There are actually more applications for these devices than you might think. More and more computer users are purchasing machines with built-in cameras and microphones (including close to every Mac for the past several years), making content that uses them more relevant to the general public. Even though the feature set is limited, each device has some unique methods to it that can be used as a controller, or in conjunction with interface elements. We'll start out by looking at microphones first.

Testing 1, 2, 3: The Microphone Object

When not connected to FMS (Flash Media Server), the microphone can really only perform one function: it can route back out to your speakers. This in and of itself is not useful except perhaps for testing; you cannot perform any operations on the sound data going out, and if your speakers are very loud, you'll experience some echo feedback. However, you can read the activity level of the microphone, while it is being processed. The activity level is anywhere from 0 to 100, representing quiet to loud, respectively. Let's look at an example of how to set up a microphone for input and use the activity level for a real purpose. You can follow along with the `MicrophoneExample.fla` file in the Appendices folder.

The `MicrophoneExample` file uses a single document class for the functionality we'll explore. The class extends `MovieClip` because it is purely a wrapper for the document.

```
public class MicrophoneExample extends MovieClip {  
  
    private var _mic:Microphone;  
    public var microphoneLevel:MovieClip;
```

This class will keep two persistent variables. One is a reference to the `Microphone` object, and the other is a vertical bar inside a

clip called *microphoneLevel*. This clip will ultimately reflect the activity level of the microphone. Now, we'll take a look at the class constructor.

```
public function MicrophoneExample() {
    _mic = Microphone.getMicrophone();
    _mic.setLoopBack(true);
    _mic.soundTransform = new SoundTransform(0);
    _mic.addEventListener(ActivityEvent.ACTIVITY, micActivity,
        false, 0, true);
}
```

Microphone instances aren't created using the *new* keyword; Flash has a list of all available microphones for the computer and will return the default device by calling *getMicrophone*. Once the instance is created, the microphone is activated by calling *setLoopBack* and passing it a parameter of *true*. Because we're not using FMS with this file, Flash needs a place to route the audio from the microphone to make it available to us. Until this method is called again with a parameter of *false*, any sound the microphone picks up will be passed through to the speakers. Because, for this example, we don't want the sound to come back through, we set the *soundTransform* of the object to a volume of 0. The microphone will still reflect its current activity level in code, but we won't hear anything. Finally, we add a listener for any *ActivityEvents* that the microphone generates.

```
private function micActivity(e:ActivityEvent):void {
    if (e.activating) {
        micUpdate(null);
        addEventListener(Event.ENTER_FRAME, micUpdate, false,
            0, true);
    } else {
        removeEventListener(Event.ENTER_FRAME, micUpdate);
    }
}
```

ActivityEvent objects are a derivative of normal events, with an additional property that tells whether the device dispatching the event is in the process of activation or deactivation. In this case, if the microphone registers anything more than a negligible amount of sound data, it will dispatch this event with the activating property set to *true*. If the microphone has gone a specific amount of time (which can be set per *Microphone* object) without detecting any audio, it will dispatch this same event with activating set to *false*. When the microphone becomes active, we attach a frame listener that will be called the *micUpdate* method. Likewise, when the microphone goes dormant, we'll remove the listener to clean up after ourselves.

```
private function micUpdate(e:Event):void {
    microphoneLevel.scaleY = Math.max(.01, _mic.
        activityLevel/100);
}
```

Although it is the code at the heart of this exercise, the *micUpdate* method is only one line. It sets the *scaleY* value of the *microphoneLevel* clip to the value of the microphone's activity level (with a minimum value of 0.01 to prevent it from disappearing altogether). This is the extent of the functionality we will include. When this SWF is exported, Flash should prompt you to allow it the access to your microphone (assuming you have one). After agreeing, you should see a bar on screen that raises and lowers with the amount of sound it picks up. Here is the class we just created in its entirety.

```
package {

    import flash.display.MovieClip;
    import flash.media.Microphone;
    import flash.media.SoundTransform;
    import flash.events.Event;
    import flash.events.ActivityEvent;

    public class MicrophoneExample extends MovieClip {

        private var _mic:Microphone;

        public var microphoneLevel:MovieClip;

        public function MicrophoneExample() {
            _mic = Microphone.getMicrophone();
            _mic.setLoopBack(true);
            _mic.soundTransform = new SoundTransform(0);
            _mic.addEventListener(ActivityEvent.ACTIVITY,
                micActivity, false, 0, true);
        }

        private function micActivity(e:ActivityEvent):void {
            if (e.activating) {
                micUpdate(null);
                addEventListener(Event.ENTER_FRAME,
                    micUpdate, false, 0, true);
            } else {
                removeEventListener(Event.ENTER_FRAME,
                    micUpdate);
            }
        }
    }
}
```

```
private function micUpdate(e:Event):void {
    microphoneLevel.scaleY = Math.max(.01, _mic.
        activityLevel/100);
}
}
```

Applications

As I mentioned earlier, this is, obviously, very simplistic functionality, but there a number of potential applications in games. For instance, in a game in which a character slowly falls down the screen, the activity level could be used as a boost to keep the character afloat. Because it does not discriminate between the tones it receives, you could tell the player to blow into the microphone to produce the noise level required. Another possibility is a timing game in which you must create a loud noise in conjunction with some rhythmic activity on the screen. Along those same lines, you could make the player create softer or louder sounds to navigate their character through a series of obstacles.

Considerations

For the purposes of the earlier example, we applied no smoothing whatsoever to the data being received from the microphone. As a result, the bar jumped rather violently from one position to the next. To account for this and make the data less erratic when displaying it on screen, we can take a *sampling* of the data over time and average it. This will smooth out any sudden peaks or drops. Here is a quick example of how the previous class could be modified to do this. First, we'd need to add to properties to the class.

```
private var _micLevels:Vector.<Number> = new Vector.<Number>();
private const _sampleSize:int = 5;
```

The first is a `Vector` (typed Array) that will hold the sample values we pull from the microphone. The second is a constant defining how many samples the `Vector` should contain before pushing older values out. We'll need to now make some additions to the `micUpdate` method and include a new method to average the values.

```
private function micUpdate(e:Event):void {
    _micLevels.push(Math.max(.01, _mic.activityLevel/100));
    if (_micLevels.length > _sampleSize) _micLevels.shift();
    microphoneLevel.scaleY = getAverage(_micLevels);
}

private function getAverage(values:Vector.<Number>):Number {
    var avg:Number = 0;
```

```

    for each (var value:Number in values) avg += value;
    return avg/values.length;
}

```

Now instead of assigning the microphone level directly to the clip's *scaleY* value, we push it into the Vector and remove any samples over the max of 5. We then assign the scale value with the new *getAverage* method. This method simply adds all the values of the Vector and returns the average. Now, when the example is exported, you'll notice the activity bar increases and decreases much more smoothly without the twitchiness it had before. You can make it even smoother by increasing the sample size, but you'll start losing accuracy for very short sounds. If you were measuring the activity level over a longer duration of time, like the example I gave earlier of having someone blow continuously into the microphone, a larger sample size would work fine. For shorter, more punctuated sounds, keeping the sample size low will get you closer to the true values.

Where do you go from here? Come up with your own unique application for how this simple functionality can play into a game mechanic. Flash applications that make use of the microphone are still pretty few and far between, so a fun game will stand out from the crowd if it makes savvy use of the device. As we explore the Camera object next, you'll notice some similarities in syntax and implementation.

Lights, Camera Object, ActionScript!

The ability to grab a live feed from a connected camera is a very nice feature of Flash, even in its limited state. To see the image from a camera, it must be attached to a Video object. Like the Microphone class, all Camera objects have an *activityLevel*, which represents the amount of motion in the video feed. When the image is very still, the value is close to zero. When the entire image is changing every frame, the activity level value approaches 100. Now, we'll look at how to set up a Camera object and display the feed on the Stage. The associated file, for this example, is *CameraExample.fla* in the Appendices folder.

```

public class CameraExample extends MovieClip {

    private var _cam:Camera;
    private var _video:Video;

    public var activityLevel:MovieClip;
}

```

We'll need to store references to both the Camera object we've created, as well as the Video object, which will be added to the Stage. Like the Microphone example, there is a clip on the Stage

already called *activityLevel* that will be scaled to represent the motion rate of the Camera.

```
public function CameraExample() {
    _cam = Camera.getCamera();
    _cam.setMode(stage.stageWidth, stage.stageHeight, 30);
    _cam.addEventListener(ActivityEvent.ACTIVITY, cameraActivity,
        false, 0, true);
    _cam.addEventListener(StatusEvent.STATUS, cameraStatus,
        false, 0, true);
    _video = new Video(stage.stageWidth, stage.stageHeight);
    _video.attachCamera(_cam);
}
```

As with Microphone objects, Camera instances are returned by the *getCamera* method. To define the dimensions and frame rate of the video feed, we call the *setMode* method of the object and use the values of the Stage dimensions. Next, we create two listeners. One is for ActivityEvents, similar to the Microphone. The other is for StatusEvents, which represent changes in the camera's operating state. In this case, we're interested in knowing when the camera is activated after the user grants permission to Flash to access the device. Finally, we create a new Video instance with the same dimensions as the camera and attach it.

```
private function cameraStatus(e:StatusEvent):void {
    if (e.code == "Camera.Unmuted") {
        if (!_video.stage) addChildAt(_video, 0);
    }
}
```

To ensure that the camera is initialized correctly, the *cameraStatus* method adds the Video object to the Stage (underneath the bar showing activity), when a status of Camera.Unmuted is sent. This message is delivered, when a user agrees to let Flash have access to their camera.

```
private function cameraActivity(e:ActivityEvent):void {
    //CAMERA WAS ACTIVATED
    if (e.activating) {
        addEventListener(Event.ENTER_FRAME, updateCamera,
            false, 0, true);
    } else {
        removeEventListener(Event.ENTER_FRAME, updateCamera);
    }
}
```

When the camera is activated by motion, it creates a frame listener that will call the *updateCamera* method.

```
public function updateCamera(e:Event):void {
    activityLevel.scaleY = Math.max(0, _cam.activityLevel/100);
}
```

Once again, just like the Microphone example, the camera's activity level is translated to the *scaleY* property of the clip on the Stage. When this SWF is exported, it will show a full-Stage video feed from your camera (if you have one attached). Here is the class as one whole piece.

```
package {

    import flash.display.MovieClip;
    import flash.events.ActivityEvent;
    import flash.events.Event;
    import flash.events.StatusEvent;
    import flash.media.Camera;
    import flash.media.Video;

    public class CameraExample extends MovieClip {

        private var _cam:Camera;
        private var _video:Video;

        public var activityLevel:MovieClip;

        public function CameraExample() {
            _cam = Camera.getCamera();
            _cam.setMode(stage.stageWidth, stage
                .stageHeight, 30);
            _cam.addEventListener(ActivityEvent.ACTIVITY,
                cameraActivity, false, 0, true);
            _cam.addEventListener(StatusEvent.STATUS,
                cameraStatus, false, 0, true);
            _video = new Video(stage.stageWidth, stage
                .stageHeight);
            _video.attachCamera(_cam);
        }

        private function cameraStatus(e:StatusEvent):void {
            if (e.code == "Camera.Unmuted") {
                if (!_video.stage) addChildAt(_video, 0);
            }
        }

        private function cameraActivity(e:ActivityEvent):
            void {
```

```
//CAMERA WAS ACTIVATED
if (e.activating) {
    addEventListener(Event.ENTER_FRAME,
        updateCamera, false, 0, true);
} else {
    removeEventListener(Event.ENTER_FRAME,
        updateCamera);
}

}

public function updateCamera(e:Event):void {
    activityLevel.scaleY = Math.max(0, _cam.
        activityLevel/100);
}
}
}
```

Because the video feed is linked to a `DisplayObject`, it can be captured and manipulated into `BitmapData`. To change this functionality, we can simply add some code to the existing example. A new file exists for this change in the Appendices folder: `CameraBitmapData-Example.fla`. The two major changes are to *cameraStatus* and *updateCamera* methods.

```
private var _currentImage:BitmapData;
private var _stageImage:Bitmap;

private function cameraStatus(e:StatusEvent):void {
    if (e.code == "Camera.Unmuted") {
        _stageImage = new Bitmap();
        addChildAt(_stageImage, 0);
    }
}
```

Two additional properties have been created in this example, each storing the related information about the image being captured from the camera. When the camera is activated, it instantiates and adds a new `Bitmap` object to the Stage.

```
private function updateCamera(e:Event):void {
    activityLevel.scaleY = Math.max(0, _cam.activityLevel/100);
    if (_currentImage) _currentImage.dispose();
    _currentImage = new BitmapData(_video.width, _video.height);
    _currentImage.draw(_video);
    _stageImage.bitmapData = _currentImage;
}
```

The *updateCamera* method has some additional functionality as well. It checks to find if any residual image data exists and disposes

of it to free up memory, and then draws a new bitmap from the video feed. On viewing this example next to the last one, there would appear to be no difference in the output, except that because *updateCamera* is only called when the camera detects significant motion, the image will freeze when still for too long. This can be adjusted by either setting the motion threshold for the camera much lower, or by running the function every frame regardless of whether the camera is detecting motion. Regardless, it is very powerful to now have live bitmap data that can be used for any number of different tasks and manipulated using any of the filters available in the display package.

Applications

Where the Microphone class relies on changes in sound to be of any real use in game development, the Camera class's reaction to motion can be put to similar use. A game could require that a player is alternately active and inactive in their motions to manipulate gameplay. Additionally, any puzzle games that make use of static imagery could use a live camera image to offer a different experience.

Conclusion

Outside of their use with FMS, microphones and cameras can be valuable input devices that can complement or even replace the keyboard and mouse in certain instances. The more developers make use of these tools, the more attention Adobe will hopefully focus in the future on making them even more robust and useful.