# 7

# Face Detection

Images are often viewed as static data, something to be displayed to the user rather then interpreted programmatically. Especially on a mobile device, where processing power is limited, interpretation of images is often neglected. However the current generation of high-end mobile devices, such as the iPhone or iPod touch, are quite capable of performing fairly advanced feats of computer vision in real-time.

## The OpenCV Library

The Open Source Computer Vision (OpenCV) Library, see http://opencv.willowgarage.com/wiki/, is a collection of routines intended for real-time computer vision, released under the BSD License, free for both private and commercial use. The library has a number of different possible applications including object recognition and tracking. We're going to spend both this and the next chapter, where we'll go into object recognition in more detail, looking at this library and its capabilities.

### Building the Library

Building the OpenCV library for the iPhone isn't entirely straightforward, as we need different versions of the library compiled to run in the iPhone Simulator, and on the iPhone or iPod touch device itself. We therefore have to go ahead and compile both a statically linked x86 version of the library, and a cross-compile a version for the ARMv6 processor.

Before proceeding, and as of the iPhone 3.1.2 SDK, if you are on Snow Leopard you need to create a symlink between the iPhone SDK and the OS 10.6 developer version of crt1.o as, although the crt1.10.5.o version of the library is present in the SDK, this is missing.

```
% cd
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhon
eSimulator3.1.2.sdk/usr/lib
% sudo ln -s /Developer/SDKs/MacOSX10.6.sdk/usr/lib/crt1.10.6.o
crt1.10.6.o
```

The first thing that you need to do is go ahead and download the latest version of the OpenCV library.

```
% cd ~/Desktop; mkdir opencv
% cd opencv
% bunzip2 ~/Downloads/OpenCV-2.0.0.tar.bz2
% tar -xvf ~/Downloads/OpenCV-2.0.0.tar
```

Once that's done, download the *build.sh* and *cvcalibration.cpp.patch* files from the book's website at http://programmingiphonesensors.com/code/ and move them into the *opencv* directory on your desktop.

```
% cp build.sh ~/Desktop/opencv
% cp cvcalibration.cpp.patch ~/Desktop/opencv
% chmod uog+rx build.sh
```

If you do an `ls` in *~/Desktop/opencv* you should now see something very much like this,

```
% ls
total 29240
drwxr-xr-x   7 aa  staff       238 15 Dec 10:56 ./
drwx------+ 16 aa  staff       544 15 Dec 10:58 ../
drwxr-xr-x@ 42 aa  staff      1428 15 Dec 10:53 OpenCV-2.0.0/
-rwxr-xr-x@  1 aa  staff      3348 15 Dec 10:51 build.sh*
-rw-r--r--   1 aa  staff       455 15 Dec 10:53 cvcalibration.cpp.patch
```

and if that's the case you should be able to go head and build the library.

The build script below, based on work by Yoshimaha Niwa (http://niw.at/) and Daniele Pizzoni (http://ildan.blogspot.com/), has been tested for version 2.0.0 of the library on both Leopard (OS X 10.5) and Snow Leopard (OS X 10.6), and against the iPhone 3.1.2 SDK.

The build script I've provided, see below, will build both the x86 (for the iPhone Simulator) and ARMv6 (for the device) versions of the library and package them together into five multi-architecture static libraries that can be easily included into your own Xcode projects.

```
#!/bin/sh

# version configuration
OPENCV_VERSION=2.0.0
GCC_VERSION=4.2
SDK_VERSION=3.1.2

# directory paths
ROOTDIR=$(pwd)
SRCDIR="${ROOTDIR}/OpenCV-${OPENCV_VERSION}"

I686DIR="$ROOTDIR/i686"
ARMDIR="$ROOTDIR/armv6"

# apply patch
cd ${SRCDIR}
patch -p0 < ../cvcalibration.cpp.patch
cd  ${ROOTDIR}

# configure path
if [ -z "${CONFIGURE}" ]; then
    CONFIGURE=${SRCDIR}/configure
fi
if [ ! -e "${CONFIGURE}" ]; then
    echo "Missing '${CONFIGURE}'."
    exit 1
fi
if [ "$1" = "--configure-help" -o "$1" = "-c" ]; then
    ${CONFIGURE} --help
    exit
fi

# build for simulator
build_simulator() {
   rm -rf $I686DIR
   mkdir -p $I686DIR
   pushd $I686DIR

   SDKNAME=Simulator
   ARCH=i686
   HOST=i686
   PLATFORM=/Developer/Platforms/iPhone${SDKNAME}.platform
   BIN=${PLATFORM}/Developer/usr/bin
   SDK=${PLATFORM}/Developer/SDKs/iPhone${SDKNAME}${SDK_VERSION}.sdk

   PREFIX=`pwd`/`dirname $0`/${ARCH}
   PATH=/bin:/sbin:/usr/bin:/usr/sbin:${BIN}

   ${CONFIGURE} \
    --prefix=${PREFIX} \
    --build=i686-apple-darwin9 \
    --host=${HOST}-apple-darwin9 \
    --target=${HOST}-apple-darwin9 \
    --enable-static \
    --disable-shared \
    --disable-sse \
    --disable-apps \
    --without-python \
    --without-ffmpeg  \
    --without-1394libs \
    --without-v4l \
    --without-imageio \
    --without-quicktime \
    --without-carbon \
    --without-gtk \
    --without-gthread \
    --without-swig \
    --disable-dependency-tracking \
    $* \
    CC=${BIN}/gcc-${GCC_VERSION} \
    CXX=${BIN}/g++-${GCC_VERSION} \
    CFLAGS="-arch ${ARCH} -isysroot ${SDK}" \
    CXXFLAGS="-arch ${ARCH} -isysroot ${SDK}" \
    CPP=${BIN}/cpp \
    CXXCPP=${BIN}/cpp \
    AR=${BIN}/ar

    make || exit 1
    popd
}

build_device() {
   rm -rf $ARMDIR
   mkdir -p $ARMDIR
   pushd $ARMDIR

   SDKNAME=OS
   ARCH=armv6
   HOST=arm
   PLATFORM=/Developer/Platforms/iPhone${SDKNAME}.platform
   BIN=${PLATFORM}/Developer/usr/bin
   SDK=${PLATFORM}/Developer/SDKs/iPhone${SDKNAME}${SDK_VERSION}.sdk

   PREFIX=`pwd`/`dirname $0`/${ARCH}
   PATH=/bin:/sbin:/usr/bin:/usr/sbin:${BIN}

   ${CONFIGURE} \
    --prefix=${PREFIX} \
    --build=i686-apple-darwin9 \
    --host=${HOST}-apple-darwin9 \
    --target=${HOST}-apple-darwin9 \
    --enable-static \
    --disable-shared \
    --disable-sse \
    --disable-apps \
    --without-python \
    --without-ffmpeg  \
    --without-1394libs \
    --without-v4l \
    --without-imageio \
    --without-quicktime \
    --without-carbon \
    --without-gtk \
    --without-gthread \
    --without-swig \
    --disable-dependency-tracking \
    $* \
    CC=${BIN}/gcc-${GCC_VERSION} \
    CXX=${BIN}/g++-${GCC_VERSION} \
    CFLAGS="-arch ${ARCH} -isysroot ${SDK}" \
    CXXFLAGS="-arch ${ARCH} -isysroot ${SDK}" \
    CPP=${BIN}/cpp \
    CXXCPP=${BIN}/cpp \
    AR=${BIN}/ar

    make || exit 1
    popd
}

# build the multi-architecture static libraries
build_static_libs() {
   lipo -create ${I686DIR}/src/.libs/libcv.a \
                ${ARMDIR}/src/.libs/libcv.a \
        -output libcv.a || exit 1

   lipo -create ${I686DIR}/src/.libs/libcxcore.a \
                ${ARMDIR}/src/.libs/libcxcore.a \
        -output libcxcore.a || exit 1

   lipo -create ${I686DIR}/src/.libs/libcvaux.a \
                ${ARMDIR}/src/.libs/libcvaux.a \
        -output libcvaux.a || exit 1

    lipo -create ${I686DIR}/src/.libs/libml.a \
                 ${ARMDIR}/src/.libs/libml.a \
         -output libml${SUFFIX}.a || exit 1

    lipo -create ${I686DIR}/src/.libs/libhighgui.a \
                 ${ARMDIR}/src/.libs/libhighgui.a \
         -output libhighgui.a || exit 1
}

# main loop
cd ${ROOTDIR}
rm -f *.a

build_device
build_simulator
build_static_libs
```

You can start the build process going by running the script. The libraries should build without intervention,

```
% ./build.sh
```

and after some time has passed you should see something that looks very much like this below when you do an `ls` inside your *~/Desktop/opencv* directory.

```
% ls
total 129808
drwxr-xr-x  13 aa  staff       442 15 Dec 11:30 ./
drwx------+ 15 aa  staff       510 15 Dec 11:04 ../
drwxr-xr-x@ 42 aa  staff      1428 15 Dec 10:53 OpenCV-2.0.0/
drwxr-xr-x  19 aa  staff       646 15 Dec 10:56 armv6/
-rw-r--r--@  1 aa  staff      3409 15 Dec 11:30 build.sh
-rw-r--r--   1 aa  staff       455 15 Dec 10:53 cvcalibration.cpp.patch
drwxr-xr-x  19 aa  staff       646 15 Dec 11:03 i686/
-rw-r--r--   1 aa  staff  17693368 15 Dec 11:11 libcv.a
-rw-r--r--   1 aa  staff   9979456 15 Dec 11:11 libcvaux.a
-rw-r--r--   1 aa  staff  17292744 15 Dec 11:11 libcxcore.a
-rw-r--r--   1 aa  staff   1844792 15 Dec 11:30 libhighgui.a
-rw-r--r--   1 aa  staff   4672376 15 Dec 11:11 libml.a
```

Congratulations, you've successfully built the OpenCV library for the iPhone, iPod touch and iPhone Simulator.

## Adding the Library to your Xcode Project

To add the static linked libraries to your own Xcode project you should navigate to the build directory using the Finder and drag-and-drop all five of the *.a* library files into the Frameworks group in your project, remembering to tick the "Copy items into destination group's folder (if needed)" check box in the pop up dialog that appears when you drop the files into Xcode.

Next navigate to the *OpenCV-2.0.0/include* folder inside the library sources tree and drag-and-drop the *opencv* folder inside that directory into your project. Again remember to tick the "Copy items into destination group's folder (if needed)" check box. This folder contains all of the necessary *.h* header files that you'll need to make use of the OpenCV library.

Finally in the Groups & Files panel in Xcode, first double-click on the "Targets" group to open it, and then double-click on your main application target inside that group (for most Xcode projects this will be the entry in the group) to open the Target Info window. Click on the Build tab and scroll down to "Other Linker Flags", and add **`-lstdc++`** and **`-lz`** to this setting.

After doing this to make use of the OpenCV library inside your project you just have to remember to import the library at the relevant place in your own code.

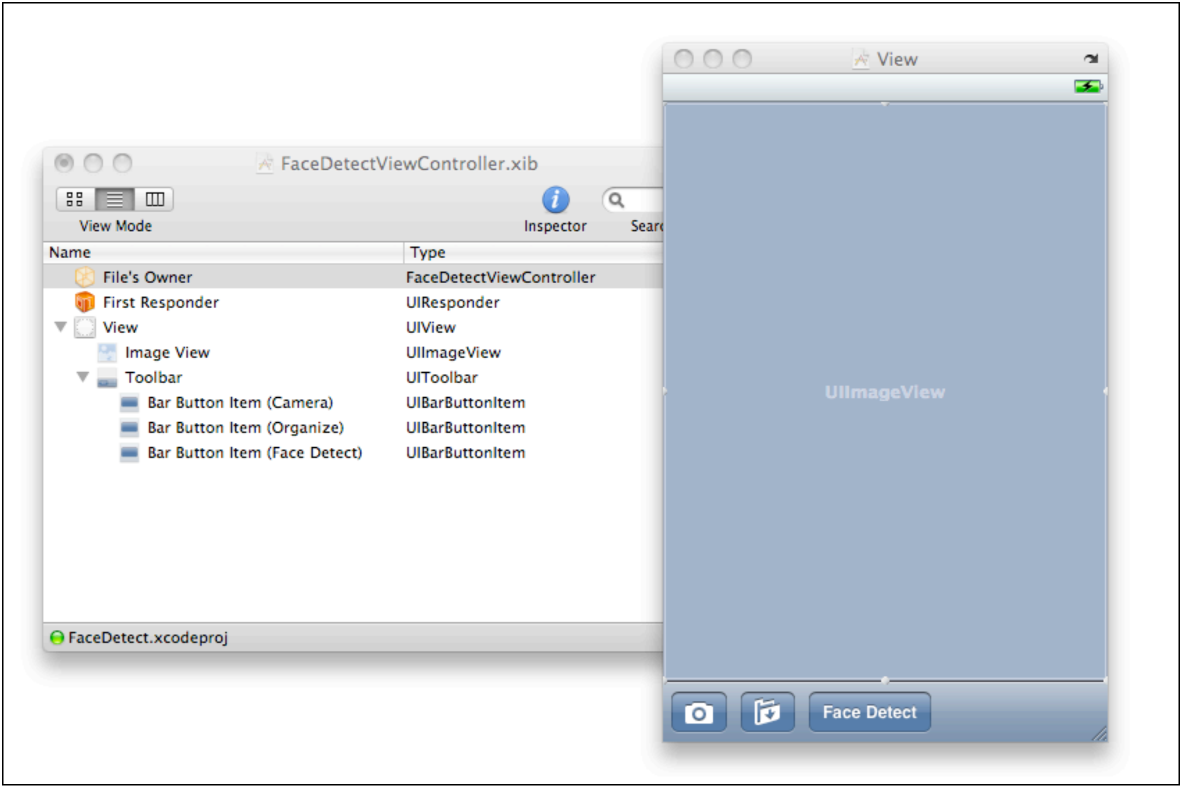
```
#import "opencv/cv.h"
```

# Face Recognition

To show you how to use the OpenCV library I'm going to walkthrough building a face recognition application. The application will either take an image with the camera, or alternatively load one from the Photo Album, and then perform face detection on the image.

Open Xcode and choose Create a new Xcode project in the startup window, and then choose the View-based Application template from the New Project popup window. When prompted, name your new project *FaceDetect*. When it opens, go ahead and add the OpenCV libraries and include files to this project as we discussed above in the proceeding section.

After adding the OpenCV library to our project, the first thing we want to do is build out the user interface. In the Groups & Files dialog panel in Xcode double-click on the *FaceDetectViewController.xib* file to open it in Interface Builder. Drag and drop a `UIToolbar` from the Library window into the View window and position it at the bottom of the bottom of the view. Double click on the `UIBarButtonItem` already embedded in the toolbar to select it and in the Attributes tab of the Inspector window change its Identifier to be "Camera". Now go ahead and drag two more UIBarButtonItem into the toolbar and position them to the right of the original button. Double-click on one of these and change it's Identifier to "Organize", and then double-click on the final button and change its Identifier to "Custom", and its title to "Face Detect". Finally drag and drop a UIImageView from the Library window into the View window and resize it to fill the remainder of the view. Then in the Attributes tab of the Inspector window set the View's Mode to "Aspect Fill". If everything has gone well you should have something very similar to Figure 7-1.



| Name | Type |
|---|---|
| File's Owner | FaceDetectViewController |
| First Responder | UIResponder |
| View | UIView |
| Image View | UIImageView |
| Toolbar | UIToolbar |
| Bar Button Item (Camera) | UIBarButtonItem |
| Bar Button Item (Organize) | UIBarButtonItem |
| Bar Button Item (Face Detect) | UIBarButtonItem |

*Figure 7-1. The Face Detect application user interface*

We're done in Interface Builder for now, although after we've written some code we'll come back and connect our interface elements to our code. But for now save your changes (⌘-S) before closing Interface Builder and returning to Xcode.

At least for this example you don't need to make any changes to the application delegate interface or implementation files. So click on the *FaceDetectViewController.h* interface file in the Groups & Files pane to open it in the Xcode editor.

```
#import <UIKit/UIKit.h>

@interface FaceDetectViewController : UIViewController
      <UIImagePickerControllerDelegate, UINavigationControllerDelegate> {!!CO1!!

    IBOutlet UIImageView *imageView;!!CO2!!
    IBOutlet UIBarButtonItem *cameraButton;!!CO3!!
    UIImagePickerController *pickerController;

}

- (IBAction)getImageFromCamera:(id) sender;
- (IBAction)getImageFromPhotoAlbum:(id) sender;

@end
```

1. Since we're going to make use of the UIImagePickerController class we need to declare our view controller to be both an Image Picker Controller Delegate and a Navigation Controller Delegate.
2. The image view will hold the image returned by the image picker controller, either from the camera or from the photo album depending on whether a camera is available.
3. We will enable or disable the button allowing the application user to take an image with the camera depending on whether the device has a camera present from our viewDidLoad: method.

With that done, click on the corresponding *FaceDetectViewController.m* implementation file in the Groups & Files pane to open it in the editor and uncomment and then edit the viewDidLoad: method to as shown below,

```
- (void)viewDidLoad {
    [super viewDidLoad];

    pickerController = [[UIImagePickerController alloc] init];
    pickerController.allowsEditing = NO;
    pickerController.delegate = self;
    if ( [UIImagePickerController
            isSourceTypeAvailable:UIImagePickerControllerSourceTypeCamera]) {
         cameraButton.enabled = YES;
    } else {
         cameraButton.enabled = NO;
    }

}
```

Here we initialize the image picker controller and determine whether our device has a camera, if so we go ahead and enable out camera button, otherwise we disable it.

Next we need to implement the methods that will be called when we tap on the camera or photo album buttons to retrieve an image, each sets the image picker controller sourceType to the correct source and presents the controller modally. We'll connect both these methods to our user interface later on in the chapter.

```
- (IBAction)getImageFromCamera:(id) sender {
    pickerController.sourceType = UIImagePickerControllerSourceTypeCamera;
    NSArray* mediaTypes = [UIImagePickerController
       availableMediaTypesForSourceType:UIImagePickerControllerSourceTypeCamera];
    pickerController.mediaTypes = mediaTypes;
    [self presentModalViewController:pickerController animated:YES];

}

- (IBAction)getImageFromPhotoAlbum:(id) sender {
    pickerController.sourceType =
       UIImagePickerControllerSourceTypeSavedPhotosAlbum;
    [self presentModalViewController:pickerController animated:YES];
}
```

Next we need to implement the UIImagePickerControllerDelegate methods,

```
- (void)imagePickerController:(UIImagePickerController *)picker
        didFinishPickingMediaWithInfo:(NSDictionary *)info {
    [self dismissModalViewControllerAnimated:YES];
    UIImage *image = [info objectForKey:@"UIImagePickerControllerOriginalImage"];
    imageView.image = image;
}

- (void)imagePickerControllerDidCancel:(UIImagePickerController *)picker {
    [self dismissModalViewControllerAnimated:YES];
}
```

The first of these dismisses the image picker controller and takes the image returned by the controller and passes it to our image view for display. The second, called if the controller is dismissed without an image, simply dismisses the controller.

Finally we have to make sure we release our pickerController and imageView objects in the dealloc: method.

```
- (void)dealloc {
    [pickerController release];
    [imageView release];
    [super dealloc];
}
```

We're done for now with the view controller, so lets go ahead and connect our UI to the code. Make sure you've saved your changes (⌘-S) to the *FaceDetectViewController.m* implementation file, and double-click on the *FaceDetectViewController.xib* file to open it in Interface Builder.

In the *FaceDetectViewController.xib* main window click on File's Owner, and then in the Inspector Window switch to the Connections pane (*Command 2*) and connect the cameraButton outlet to the camera button in the toolbar, and the imageView outlet to the UIImageView. After doing this connect the getImageFromCamera and getImagefromPhotoAlbum received actions to the appropriate buttons on the toolbar. After making these connections you should have something that looks like Figure 7-2.

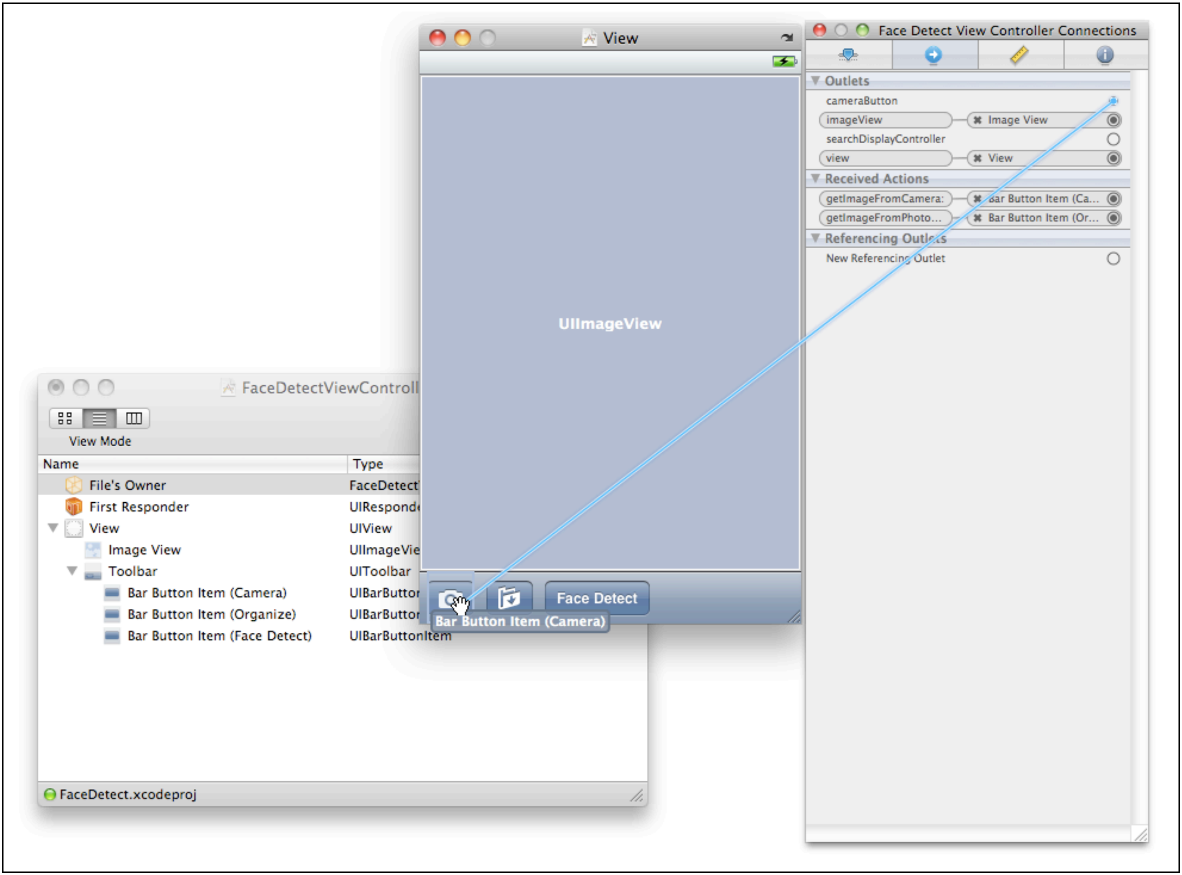


*Figure 7-2. Connecting the Face Detect application's view controller to the UI*

Make sure you've saved your changes (⌘-S) and close Interface Builder and return to Xcode. We can now go ahead and test our code, having reached the point where we should be able to select a picture (either using taken using the camera or from the photo album) and then display it in our image view.

Click on the Build and Run button to build and deploy your application into the iPhone Simulator, you should initially see an interface that resembles the left most of the images in Figure 7-3. Note that, as the application is running inside the iPhone Simulator, the camera button is disabled.

Tap on the middle (Organize) button in the toolbar to open the image picker to your photo album, and select the image you want to load, once you do so the picker should close and the image should be displayed in the image view of the application, see Figure 7-3 again.

If you test the application in the iPhone Simulator you'll notice that there aren't any images in the Saved Photos folder. There is a way around this problem. In the Simulator, tap on the Safari Icon and drag and drop a picture from your computer (you can drag it from the Finder or iPhoto) into the browser. You'll notice that the URL bar displays the *file:* path to the image. Click and hold down the cursor over the image and a dialog will appear allowing you to save the image to the Saved Photos folder.

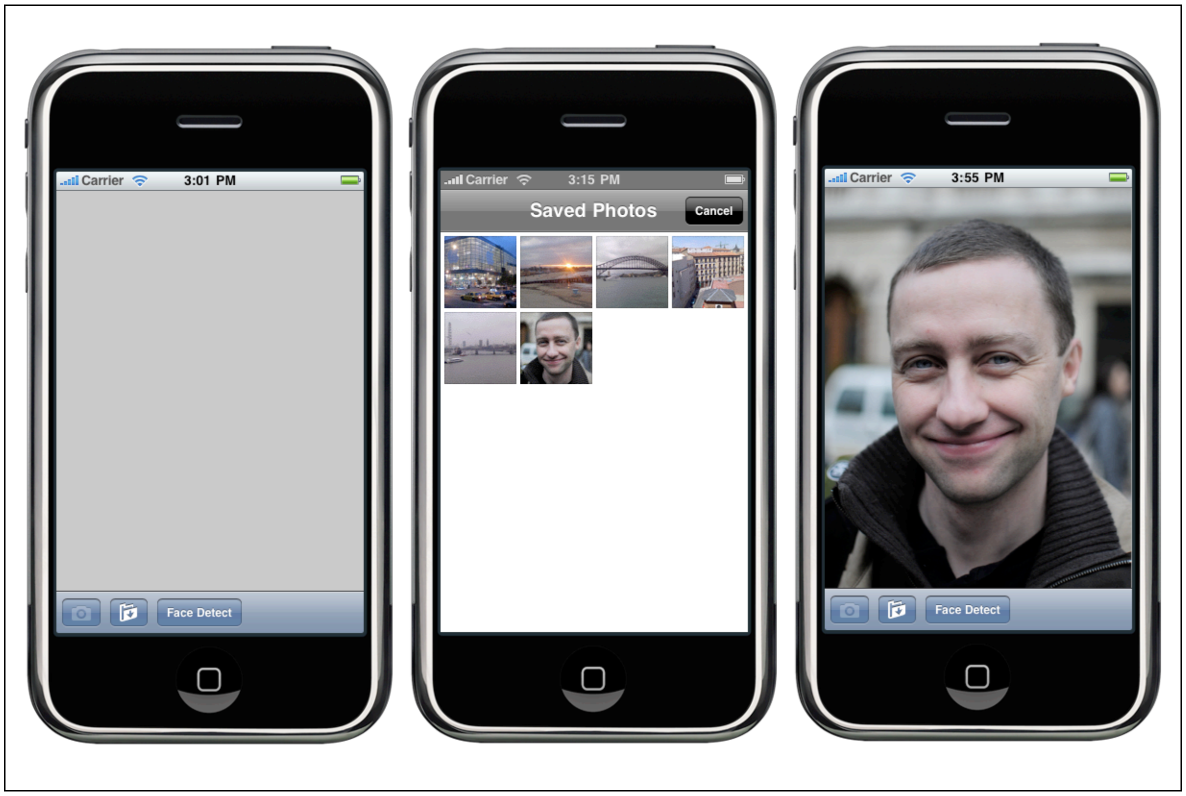


*Figure 7-3. Loading a picture from the Photo Album into the Face Detect application*

If you now change the targeted SDK to your device, using the Overview drop down menu in the Xcode toolbar, and click on the Build and Run button again to build and deploy onto your iPhone or iPod touch when the application starts you should see that the camera button is enabled. Tapping on this button will again open the image picker controller, although this time it should display the camera interface allowing you to take an image. However after you do so, and confirmed that this is the image you wish to use, it should be displayed in the application's image view in the same manner as before.

Now we've tested our code, and have our image, lets go back into Xcode and implement the actual face detection functionality. To support this we're going to go ahead and implement a `Utility` class that you can reuse in your own code to encapsulate our dealings with the OpenCV library.

The utility code shown below is based on previous work by Yoshimasa Niwa, and is released under the terms of the MIT license. The original code can be obtained from http://github.com/niw/iphone_opencv_test.

Right click on the Classes group in the Groups & Files pane in Xcode and select Add→New Group, and name the group utilities. The right click on the new Utilities group and select Add→New File... choosing Objective-C class, a subclass of `NSObject`, in the New File dialog. Name the new class *Utilities.m* when prompted.

After doing so, click on the newly created *Utilities.m* interface file to open it in the Xcode editor and make the changes highlighted below,

```
#import <Foundation/Foundation.h>
#import "opencv/cv.h"

@interface Utilities : NSObject {

}

+ (IplImage *)CreateIplImageFromUIImage:(UIImage *)image;
+ (UIImage *)CreateUIImageFromIplImage:(IplImage *)image;
+ (UIImage *)opencvFaceDetect:(UIImage *)originalImage;

@end
```

As the OpenCV library uses the `IplImage` class to hold image data we're going to need methods to convert to and from the `IplImage` class and the iPhone's own `UIImage` class. You can see from the method names we're going to implement methods to create an `IplImage` from a `UIImage`, and a `UIImage` from an `IplImage`. We're also going to go ahead and tackle the meat of the problem and implement a method that will take a `UIImage` and carry out face detection using the OpenCV library.

After saving your changes to the interface file, click on the corresponding *Utilities.m* implementation file and add the two methods that follow.

```
+ (IplImage *)CreateIplImageFromUIImage:(UIImage *)image {
    CGImageRef imageRef = image.CGImage;

    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    IplImage *iplimage = cvCreateImage(cvSize(image.size.width, image.size.height),
                                       IPL_DEPTH_8U, 4);
    CGContextRef contextRef =
        CGBitmapContextCreate(iplimage->imageData, iplimage->width, iplimage->height,
                              iplimage->depth,iplimage->widthStep, colorSpace,
                              kCGImageAlphaPremultipliedLast|kCGBitmapByteOrderDefault);
    CGContextDrawImage(contextRef,
                       CGRectMake(0, 0, image.size.width, image.size.height),
                       imageRef);
    CGContextRelease(contextRef);
    CGColorSpaceRelease(colorSpace);

    IplImage *ret = cvCreateImage(cvGetSize(iplimage), IPL_DEPTH_8U, 3);
    cvCvtColor(iplimage, ret, CV_RGBA2BGR);
    cvReleaseImage(&mp;iplimage);

    return ret;
}

+ (UIImage *)CreateUIImageFromIplImage:(IplImage *)image {
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    NSData *data = [NSData dataWithBytes:image->imageData length:image->imageSize];
    CGDataProviderRef provider = CGDataProviderCreateWithCFData((CFDataRef)data);
    CGImageRef imageRef = CGImageCreate(image->width, image->height,
                                        image->depth, image->depth * image->nChannels,
                                        image->widthStep, colorSpace,
                                        kCGImageAlphaNone|kCGBitmapByteOrderDefault,
                                        provider, NULL, false, kCGRenderingIntentDefault);
    UIImage *ret = [UIImage imageWithCGImage:imageRef];
    CGImageRelease(imageRef);
    CGDataProviderRelease(provider);
    CGColorSpaceRelease(colorSpace);
    return ret;
}
```

Here we made use of the underlying Core Graphics framework to convert between UIImage from an IplImage.

Next we need to implement the method that we'll use to carry out the actual face detection inside the application. We're going to make use of the CvHaarClassifierCascade object, a "fast-object-detector" which can discover objects such as faces in an image. To do so we're going to load an XML file containing a cascade of Haar classifiers.

## Haar Classifiers and OpenCV

Haar-like features are image features used for object recognition. These features use the change in contrast values between adjacent groups of pixels. Two (or three) adjacent groups with a relative contrast variance form a Haar-like feature.

Haar classifiers is a object detection technique first developed by Paul Viola and Michael Jones which uses these Haar-like features in a cascade, allowing only the images with the highest probability of containing the object to be analyzed for all of the distinguishing Haar-like features.

To detect objects using these cascades, you must train the classifiers using two sets of images; one set containing images that do not contain the object, and another set containing images that contain one (or more) instances of the object.

Fortunately the OpenCV library ships with a number of such pre-trained classifier files which can be found in the data/haarcascades/ directory inside the OpenCV source bundle.

More information about Haar Classifiers and their use inside the OpenCV library can be found in Chapter 13 of *Learning OpenCV* (http://oreilly.com/catalog/9780596516130) by Gary Rost Bradski and Adrian Kaehler (O'Reilly).

You can find the Haar classifier cascade files in the inside the OpenCV source bundle in the *data/haarcascades/* directory. There are several different cascade files to choose from, all of which perform slightly differently. However for recognition of (frontal) faces in images the *haarcascade_frontalface_alt2.xml* is probably the best choice. Drag-and-drop this file into the Resources group in your project, remembering to tick the "Copy items into destination group's folder (if needed)" check box in the pop up dialog that appears when you drop the files into Xcode. This will make it available to your code.

Once you have added the cascade file to your project, open the *Utilities.m* implementation file and add the following method. Here we use Haar classifiers to detect a face (or a sequence of faces) in a UIImage passed to the routine, and then proceed to draw a box around any detected faces.

```
+ (UIImage *) opencvFaceDetect:(UIImage *)originalImage  {
    cvSetErrMode(CV_ErrModeParent);

    IplImage *image = [self CreateIplImageFromUIImage:originalImage];

    // Scaling down
    IplImage *small_image = cvCreateImage(cvSize(image->width/2,image->height/2),
                                          IPL_DEPTH_8U, 3);
    cvPyrDown(image, small_image, CV_GAUSSIAN_5x5);
    int scale = 2;

    // Load XML
    NSString *path = [[NSBundle mainBundle]
```

```
                      pathForResource:@"haarcascade_frontalface_alt2"
                               ofType:@"xml"];
    CvHaarClassifierCascade* cascade = CvHaarClassifierCascade*)cvLoad(
           [path cStringUsingEncoding:NSASCIIStringEncoding], NULL, NULL, NULL);
    CvMemStorage* storage = cvCreateMemStorage(0);

    // Detect faces and draw rectangle on them
    CvSeq* faces = cvHaarDetectObjects( small_image, cascade, storage, 1.2f, 2,
                                    CV_HAAR_DO_CANNY_PRUNING, cvSize(20, 20));
    cvReleaseImage(&small_image);

    // Create canvas to show the results
    CGImageRef imageRef = originalImage.CGImage;
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGContextRef contextRef =
       CGBitmapContextCreate(NULL, originalImage.size.width, originalImage.size.height,
                             8, originalImage.size.width * 4, colorSpace,
                             kCGImageAlphaPremultipliedLast|kCGBitmapByteOrderDefault);
    CGContextDrawImage(contextRef, CGRectMake(0, 0, originalImage.size.width,
                       originalImage.size.height), imageRef);

    CGContextSetLineWidth(contextRef, 4);
    CGContextSetRGBStrokeColor(contextRef, 0.0, 0.0, 1.0, 0.5);

    // Draw results on the iamge
    for(int i = 0; i < faces->total; i++) {
         NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

         // Calc the rect of faces
         CvRect cvrect = *(CvRect*)cvGetSeqElem(faces, i);
         CGRect face_rect = CGContextConvertRectToDeviceSpace(contextRef,
                                    CGRectMake(cvrect.x * scale, cvrect.y * scale,
                                     cvrect.width * scale, cvrect.height * scale));
         CGContextStrokeRect(contextRef, face_rect);

         [pool release];
    }

    UIImage *returnImage =
                  [UIImage imageWithCGImage:CGBitmapContextCreateImage(contextRef)];
    CGContextRelease(contextRef);
    CGColorSpaceRelease(colorSpace);

    cvReleaseMemStorage(&storage);
    cvReleaseHaarClassifierCascade(&cascade);

    return returnImage;
}
```

Now we've implemented the underlying face detection routine, we can go ahead and hook it up to our user interface. Make sure you have saved your changes (⌘-S) and click on the *FaceDectectViewController.h* interface file to open it in the Xcode editor. Add the following method to the class definition.

```
- (IBAction)faceDetect:(id) sender;
```

Save your changes and click on the corresponding implementation file, *FaceDectectViewController.m*, and add the following method. The method passes the current image displayed in the image view to our face detection method. We then take the image returned by the method and display it in the image view.

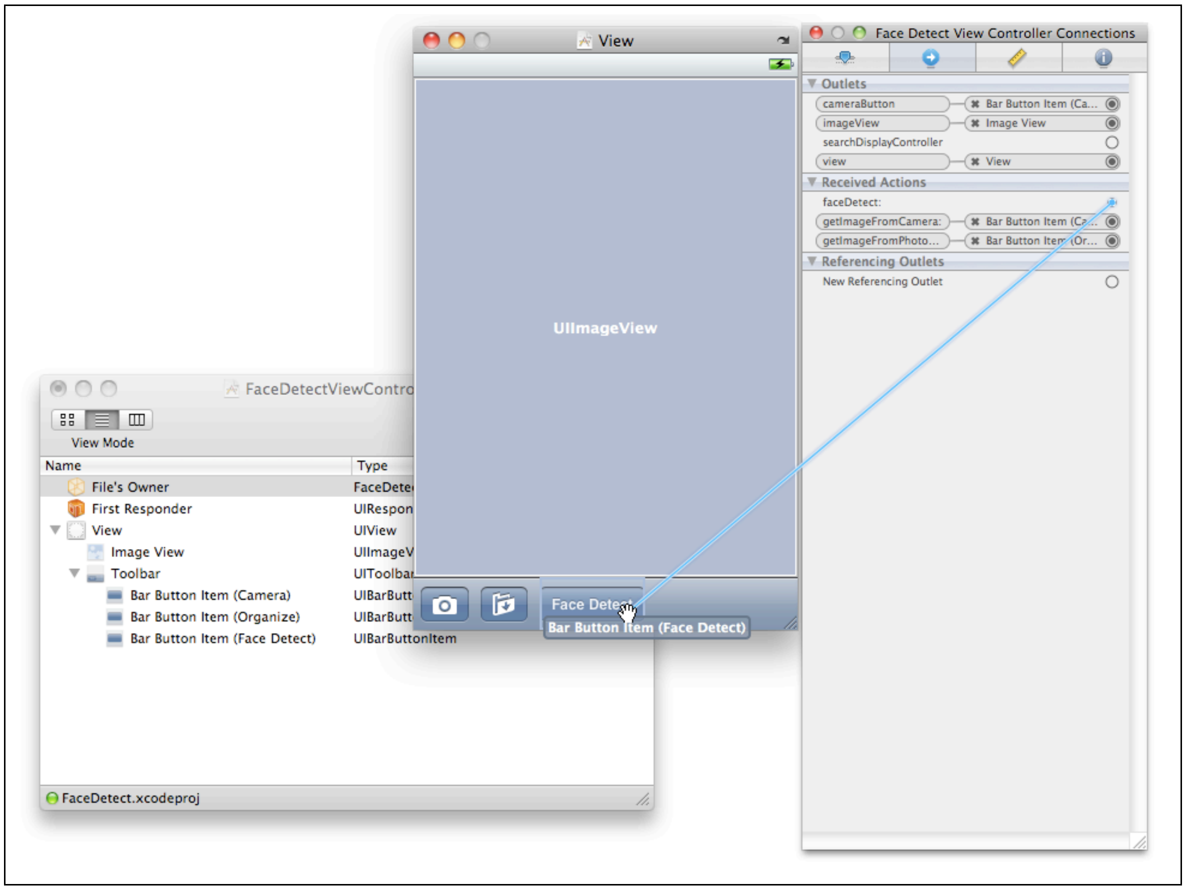
```
- (IBAction)faceDetect:(id) sender {
    UIImage *processedImage = [Utilities opencvFaceDetect:imageView.image];
    imageView.image = processedImage;
}
```

Also, since we've made use of our Utilities class, remember to add the follow import at the top of the *FaceDectectViewController.m* implementation file before saving your changes.

```
#import "Utilities.h"
```

Before we can test our code we need to go back into Interface Builder and connect the `faceDetect:` received action in our view controller to the Face Detect button in our user interface.

Double click on the *FaceDetectViewController.xib* file to open it in Interface Builder and click on File's Owner in the NIB window. Then in the Connections tab of the Inspector Window connect the `faceDetect:` and button, see Figure 7-4.



| Name | Type |
|---|---|
| File's Owner | FaceDete |
| First Responder | UIRespon |
| View | UIView |
| Image View | UIImageV |
| Toolbar | UIToolbar |
| Bar Button Item (Camera) | UIBarButt |
| Bar Button Item (Organize) | UIBarButt |
| Bar Button Item (Face Detect) | UIBarButtonItem |

*Figure 7-4. Connecting the `faceDetect:` outlet*

Save your changes (⌘-S) and close Interface Builder. Back in Xcode click on the Build and Run button in the Xcode toolbar. Once the application has loaded click on the Organize button to load an image (containing a frontally viewed face) into our application from the Photo Album and tap on the Face Detect button in our interface. If all goes well after a brief pause you should see a blue square appear around the face, see Figure 7-5.

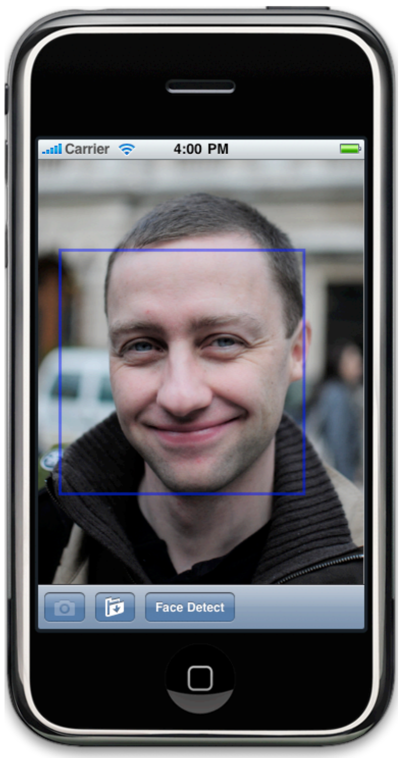


*Figure 7-5. The working Face Detect application running in the iPhone Simulator, since the simulator does not have a camera the Camera Button is disabled*

If everything has gone to plan, congratulations, you've just successfully implemented a face detection application on the iPhone. However if your image was re-displayed rotated (see Figure 7-6) then continue reading. We'll solve this problem shortly.

So far we've only deployed the application into the Simulator, lets go ahead and deploy onto our device hardware. As we did earlier in the chapter, change the targeted SDK to your device, using the Overview drop down menu in the Xcode toolbar, and click on the Build and Run button again to build and deploy onto your iPhone or iPod touch. When the application starts you should again see that the camera button is enabled. Tap on this button and take an image with the camera.

Once the image has loaded into our image view, tap on the Face Detect button. Unlike when we ran the application in the Simulator there will be a significant delay before the face detection method returns. Typically an iPhone 3GS will return in just under 2 seconds, while an iPhone 3G may take as long as 5 seconds to process the image. However there is a problem, see Figure 7-6.

When we attempt to use the `FaceDetect:` method on images taken with the camera, rather than those loaded from the Photo Album the images are rotated, and predictably at this point the Haar classifiers fail to identify a face in the image. What's going on here is that the image orientation metadata hasn't survived the conversion between the `UIImage` and `IpImage`.

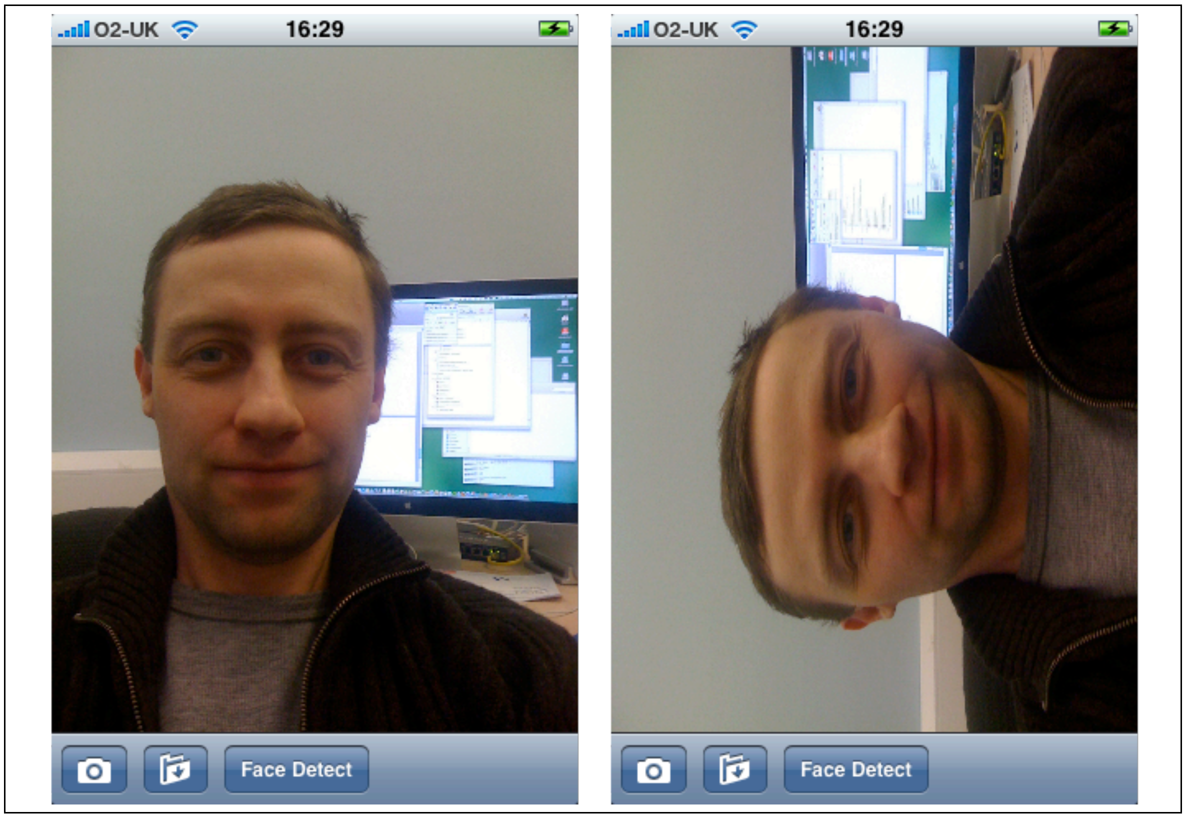


*Figure 7-6. Images taken with the camera are rotated*

The solution is to add an additional image manipulation method to our `Utilities` class to make sure the image we pass into the Haar classifiers is correctly scaled and rotated. Click on the *Utilites.h* interface file to open it in the Xcode editor and add the following method prototype.

```
+ (UIImage *)scaleAndRotateImage:(UIImage *)image;
```

Save your changes and open the corresponding *Utilites.h* implementation file and add the method body.

```
+ (UIImage *)scaleAndRotateImage:(UIImage *)image {
      static int kMaxResolution = 640;

      CGImageRef imgRef = image.CGImage;
      CGFloat width = CGImageGetWidth(imgRef);
      CGFloat height = CGImageGetHeight(imgRef);

      CGAffineTransform transform = CGAffineTransformIdentity;
      CGRect bounds = CGRectMake(0, 0, width, height);
      if (width > kMaxResolution || height > kMaxResolution) {
      CGFloat ratio = width/height;
      if (ratio > 1) {
      bounds.size.width = kMaxResolution;
      bounds.size.height = bounds.size.width / ratio;
      } else {
      bounds.size.height = kMaxResolution;
      bounds.size.width = bounds.size.height * ratio;
      }
      }

      CGFloat scaleRatio = bounds.size.width / width;
      CGSize imageSize = CGSizeMake(CGImageGetWidth(imgRef), CGImageGetHeight(imgRef));
      CGFloat boundHeight;
```

```
UIImageOrientation orient = image.imageOrientation;
switch(orient) {
case UIImageOrientationUp:
transform = CGAffineTransformIdentity;
break;
case UIImageOrientationUpMirrored:
transform = CGAffineTransformMakeTranslation(imageSize.width, 0.0);
transform = CGAffineTransformScale(transform, -1.0, 1.0);
break;
case UIImageOrientationDown:
transform = CGAffineTransformMakeTranslation(imageSize.width, imageSize.height);
transform = CGAffineTransformRotate(transform, M_PI);
break;
case UIImageOrientationDownMirrored:
transform = CGAffineTransformMakeTranslation(0.0, imageSize.height);
transform = CGAffineTransformScale(transform, 1.0, -1.0);
break;
case UIImageOrientationLeftMirrored:
boundHeight = bounds.size.height;
bounds.size.height = bounds.size.width;
bounds.size.width = boundHeight;
transform = CGAffineTransformMakeTranslation(imageSize.height, imageSize.width);
transform = CGAffineTransformScale(transform, -1.0, 1.0);
transform = CGAffineTransformRotate(transform, 3.0 * M_PI / 2.0);
break;
case UIImageOrientationLeft:
boundHeight = bounds.size.height;
bounds.size.height = bounds.size.width;
bounds.size.width = boundHeight;
transform = CGAffineTransformMakeTranslation(0.0, imageSize.width);
transform = CGAffineTransformRotate(transform, 3.0 * M_PI / 2.0);
break;
case UIImageOrientationRightMirrored:
boundHeight = bounds.size.height;
bounds.size.height = bounds.size.width;
bounds.size.width = boundHeight;
transform = CGAffineTransformMakeScale(-1.0, 1.0);
transform = CGAffineTransformRotate(transform, M_PI / 2.0);
break;
case UIImageOrientationRight:
boundHeight = bounds.size.height;
bounds.size.height = bounds.size.width;
bounds.size.width = boundHeight;
transform = CGAffineTransformMakeTranslation(imageSize.height, 0.0);
transform = CGAffineTransformRotate(transform, M_PI / 2.0);
break;
default:
[NSException raise:NSInternalInconsistencyException
format:@"Invalid image orientation"];
}

UIGraphicsBeginImageContext(bounds.size);
CGContextRef context = UIGraphicsGetCurrentContext();
if (orient == UIImageOrientationRight || orient == UIImageOrientationLeft) {
CGContextScaleCTM(context, -scaleRatio, scaleRatio);
CGContextTranslateCTM(context, -height, 0);
} else {
CGContextScaleCTM(context, scaleRatio, -scaleRatio);
CGContextTranslateCTM(context, 0, -height);
}
CGContextConcatCTM(context, transform);
CGContextDrawImage(UIGraphicsGetCurrentContext(),
CGRectMake(0, 0, width, height), imgRef);
```

```
        UIImage *returnImage = UIGraphicsGetImageFromCurrentImageContext();
        UIGraphicsEndImageContext();

        return returnImage;
        }
```

As you can see, after ascertaining the image orientation we make use of the Core Graphics framework to transform the image. Make sure you've saved your changes after adding the method,

Finally in the *FaceDetectViewController.m* file we're going to go ahead and modify the `imagePickerController:didFinishPickingMediaWithInfo:` delegate method to make use of our new utility method

```
- (void)imagePickerController:(UIImagePickerController *)picker
didFinishPickingMediaWithInfo:(NSDictionary *)info {
    [self dismissModalViewControllerAnimated:YES];
    UIImage *image = [info objectForKey:@"UIImagePickerControllerOriginalImage"];
    image = [Utilities scaleAndRotateImage:image];
    imageView.image = image;
}
```

to make sure that all images are correctly scaled and rotated before being passed to the image view controller.

If you now save your changes (⌘-S) and click on the Build and Run button to compile and (re-)deploy your application onto your device you should see something similar to Figure 7-7 this time around.

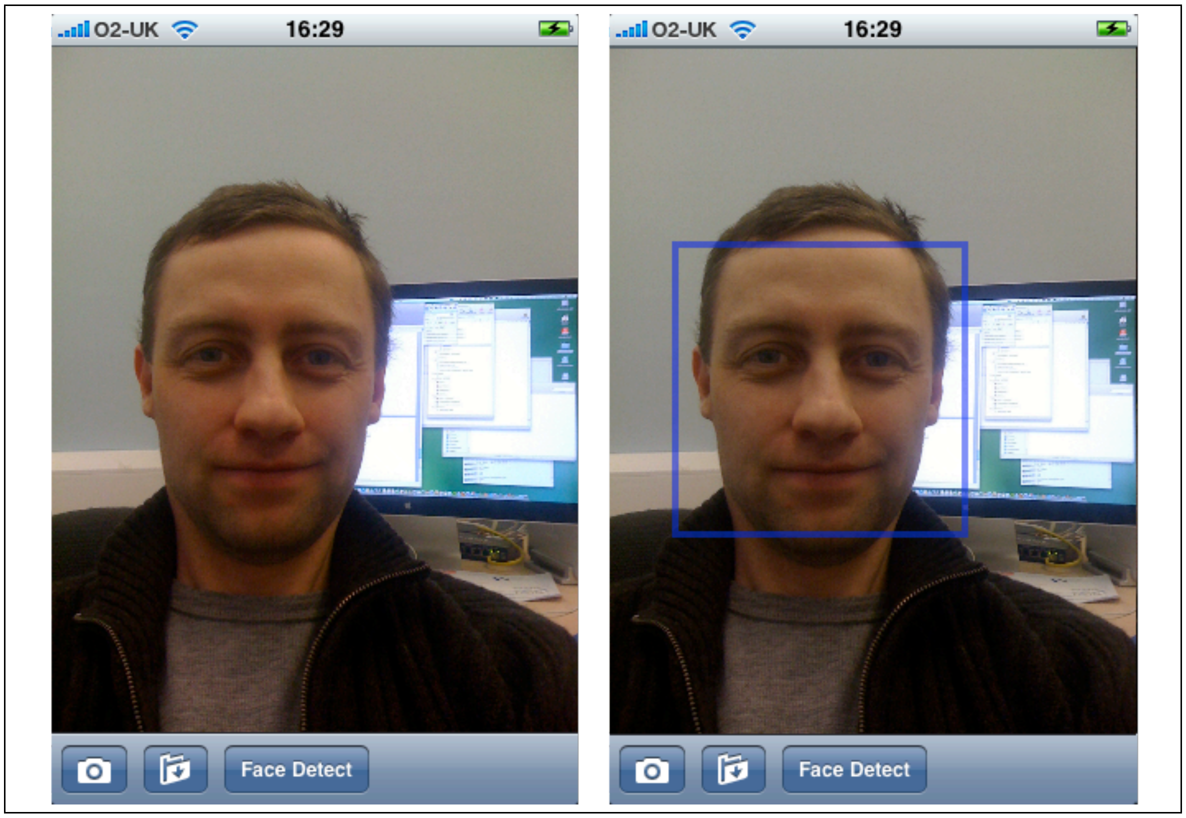


*Figure 7-7. Camera images are now correctly rotated*

You now have a working face detection application on your iPhone.

## Further Information

I’m going to discuss using the OpenCV library and computer vision in more detail in the next chapter, however if you want to learn more about computer vision I’d recommend *Learning OpenCV* (http://oreilly.com/catalog/9780596516130), by Gary Rost Bradski and Adrian Kaehler (O’Reilly), which provides a working guide to the library as well as a good background in the field of computer vision.