Development of Computer Graphics and Digital Image Processing Applications on the iPhone

Luciano Godoy Fagundes
Programa de Pós-Graduação em Computação Aplicada
Instituto Nacional de Pesquisas Espaciais
São José dos Campos, São Paulo, Brazil
luciano_fagundes@yahoo.com

Rafael Santos
Lab. Associado de Computação e Matemática Aplicada
Instituto Nacional de Pesquisas Espaciais
São José dos Campos, São Paulo, Brazil
rafael.santos@lac.inpe.br

***Abstract*—The iPhone is one of the most powerful, complete and versatile portable phones on the market. There are presently more than 150.000 applications available for the iPhone, and its users had downloaded more than three billion applications so far. The iPhone have several capabilities that makes it an interesting platform for the development of applications that use image processing, computer graphics and/or pattern recognition algorithms: it is stable, popular, powerful, flexible and of course portable. What can a developer expect from the platform? What, in practical terms, can be done to implement those types of algorithms, and at what price? This survey paper (written as part of a short course presented at Sibgrapi' 2010) shows some concepts and practical issues on the developing of image processing, computer graphics and pattern recognition applications on the iPhone. Code snippets will be provided, and issues such as memory management, capabilities and limitations will be discussed.**

***Keywords*-iPhone; mobile computing; image processing; pattern recognition;**

I. Introduction

A revolution is on its way and you are invited! Computer Science had mostly evolved around a growing number of processors, faster memory and larger disk space. However, nowadays a revolution with much more modest specifications is changing the way people use computers. Mobile devices have become a new platform for computing. This time, the revolution is not centered on bigger computers but it has in its core portability and the end-user experience. Apple brought to the world a set of "pocket computers" called iPhone/iPod Touch/iPad which, besides serving as portable media devices, may also contain different types of sensors and communicate with networks through wi-fi and bluetooth – some devices even allow the user to make phone calls!

On a highly controlled environment, Apple has built a set of rules on how to develop software for its devices and opened their APIs to a world wide community of developers. The extremely intuitive UIs plus a market of around 150.000.000 customers and a world wide competition led developers to quickly evolve beyond the standard Apple UIKit components and start building their own interfaces. In order to build this new class of applications where the user nearly needs no training at all to start using them, software developers have been using technologies that were, until some short time ago, used only in proof-of-concept applications: neural networks for advanced gesture recognition, signal processing techniques for speech recognition, location services for finding the nearest coffee shop, image processing, augmented reality and others.

Specifically for this paper, we will focus on the main capabilities of iPhone/iPod Touch/iPad[1] devices and how those devices can be used for image processing and graphical information presentation. The next sections will describe the devices' capabilities, tools used for Software development, how to deliver your applications to the world wide community of users and some simple examples that shows the basic steps on how to build simple but fully functional image processing applications on the iPhone.

A. Developing for the iPhone and iPad

There are two official lines of development that Apple makes available for their developer community. There is an uncontrolled, unrestricted line that is composed of Apps built on the top of HTML5 and there is a more sensitive and controlled environment composed by the realm of the Native Apps. HTML5 is an open standard (at the moment that this paper is being written it is still not finalized by its committee) and it is fully supported by the Safari Web browser that can be used to run Web Apps on the devices [1]. HTML5 has its own way to interact with the devices and use several of their capabilities. Developers will be limited only by the HTML 5 constraints.

The other side of the story is the realm of Native Apps. Native Apps are deployed on the device themselves and use Apple frameworks to have direct access to the full set of device capabilities. As the Native Apps are the one that can be developed with the widest scope, those are the ones that will be focused from now on.

[1]From this point on, we will focus more on the iPhone. Although most of the concepts are appliable to these three devices, there are some limitations due to the sensors and capabilities of the devices.

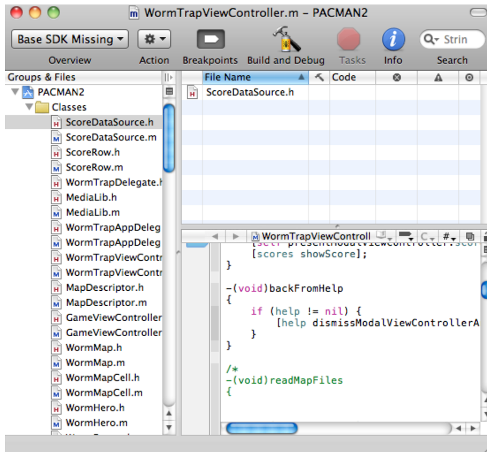

Figure 1: Screenshot of the Xcode IDE.

Figure 2: Screenshot of the Interface Builder GUI.

B. *XCode 3.x and iOS 4 SDK*

In June 2010, Apple has renamed their iPhone OS to iOS. It has been renamed to synchronize the OS with a wider variety of devices. By the end of 2010, iPod Touches, iPads and iPhone should be sharing the same iOS 4 software.

For iOS Native Apps to be developed, developers must agree with rules previously defined by Apple (more about this will be discussed at the “App Store” section). Among those rules, the developers agree that they will develop their Apps on some of the following languages: Objective C, C, Objective C++ and/or C++. Those are a few of the languages officially supported by the Apple development tools. Unfortunately, those tools only work on MacOS devices what requires that developers to own a computer manufactured by Apple in order to work.

Apple offers a set of development tools. They are free of charge and anyone that owns an Apple computer can install them from their MacOS installation CD/DVD. If the bundled software is not available on the CD it is probably because it is outdated and developers would be able to subscribe for free on the iPhone development program (`http://developer.apple.com/iphone`) and have access to the latest version of the development tools.

After installing the development tools, developers will have access to several applications and utilities. The most important for a beginner will be XCode. XCode is the main piece of the development environment (Figure 1). It is a full featured IDE (Integrated Development Environment) that can be used to write your own Objective-C/C++ code. At the current version (3.x), XCode works integrated with Interface Builder (Figure 2) that is a *WYSIWYG* (What You See is What You Get) editor for building UI (User Interfaces) for your Apps with the standard Apple components.

Another important piece of the development environment is the iPhone/iPad Simulator. This enables the developers with a virtual device where most of the features can be tested even if the developer does not own a physical device.

The simulator is very close to the real device but there are a few limitations. From the image processing perspective, the main limitation is the lack of a camera. Developers can use the simulator to get images from the user folder but they cannot acquire images from the camera what specially limits the development of Enhanced Reality Apps. For apps that requires access to the camera, the developer must own a real device.

Another important remark about the development environment is that with a free subscription a developer can have access to all the development tools but the main limitation is that you cannot deploy your Apps on your device. For being able to use the device, a developer must enroll for a paid subscription that by the time this paper has been written costs US$99,00 per year.

C. *The App Store*

For developers, the real revolution came on the shape of the App Store. The App Store has been the first environment that enabled individual developers to distribute their work formally to nearly the entire world with minimum marketing effort.

Apple has embedded the App Store in all their devices. It basically allows users to browse and search among all the available Apps. Users can access the App Store from their personal computers, their devices or the web. Registered users can buy Apps directly from their devices with a single click.

The upside of the App Store is that it takes care of all the logistics behind selling Apps worldwide. Developers are free to choose in what countries they want their Apps to be available and their prices, they are also responsible for filling in the App description (all that is done on-line at `http://itunesconnect.apple.com`) with textual

information and also providing snapshots of their Apps. This goes automatically to the App Store and it is what the customers will see. The description will be filled for at least one language (English) but it also allows developers to enter localized data that will be used as needed by the App Store from different countries.

The App Store will take care of selling, delivering and charging customers by the Apps. App Store will also open a connection between the end-users and the developers. Developers will have to take care of technical support, customer care, etc. Apple charges developers 30% of the App price in order to keep all this logistics going. For example, on a US$ 0.99 App, the developer would get US$ 0.69.

The downside of the App Store is that it is a very crowded environment. As this is a huge market, tens of thousands of App developers are fighting for the same space. Apple has continuously improved their search mechanisms but it has not been enough to make developers happy. It is still very hard to highlight your App on the middle of the hundreds of thousands of the Apps available today.

Another polemic item of the App Store is its approval process. Differently than HTML 5, Apple reserves to itself the right to not make your App available if they do not consider it appropriate for their devices. The main critics for the approval process is the lack of clear guidelines on what Apple considers appropriate or not. During the WWDC 2010 keynote, Steve Jobs (Apple CEO) has listed the most likely causes for an App to be rejected: 1) The App does not do what you say it does; 2) Use of private API (APIs that exist on iOS but are not publicly available to developers) and 3) App Crashes. For the most of the developers, those three reasons are not deal breakers for building Apps. However, it is important to highlight that there is a risk that one spent time and money to build an App that will be rejected to enter the store. As the only official way to deliver your Apps to the larger world wide customer base is the App Store, it would mean that your App would face an early retirement and the developer would take the hit.

II. Basic Image Processing on the iPhone

From the image processing perspective, the APIs are the same for all iOS devices. All iOS devices share the same low level functionalities. The main difference would be the hardware capabilities themselves; the performance growth goes from iPod Touch to iPhone to iPad (it is important to highlight that iPhone 4 and iPad share the same Apple A4 processor). Among other capabilities, the camera is also a major component for image processing. At this moment, only iPhone devices are enabled with cameras and iPhone 4 devices has two cameras (on the back and front).

In order to share the same APIs among all devices, Apple made the developers in charge of identifying device capabilities and deciding what features should or should not be enabled during runtime. To make sure that the Apps handle the device limitations appropriately, this is one of the tests that Apple runs during their approval process and a misbehavior would imply on a rejection of the App. For example, an App that is supposed to run on iPod, iPad and iPhone should be smart enough to disable the camera capability on iPad and iPod and enable the camera capability on an iPhone. If this checking is not in place, App will not make it to the App Store.

The main example for this paper is an App that let the user select an image from the Photo library, apply some basic image processing techniques and save the modified image back into the Photo library. This basic example should enable developers with the basic tools they need to start building more serious image processing Apps.

A. Some classes for image loading and display

In this section we will present some classes that are used for image processing and representation on the iPhone and similar devices.

UIImageView is part of the UIKit library. This library is from where developers can get the main UI components such as scrollers, buttons, sliders, etc. UIImageView is a component whose purpose is to show images (it contains a reference to an instance of UIImage, which represents a digital image). There are several reasons why developers use UIImageView instead of drawing the images directly to the screen, among them are: autoresizing of the image in several different ways, autorotation to the several device orientations (portrait, landscape, etc), memory management, built-in animations, etc.

Creation of an instance of UIImageView is simple, as shown in the code snippet in listing 1.

Listing 1: Creating an instance of `UIImageView`.

```
// Create an instance of UIImageView with a fixed size
// and position.
UIImageView *imgView =
   [[UIImageView alloc] initWithFrame(10,10,100,100)]];
// Create an instance of UIImage with a URL for a image.
UIImage *img =
   [[UIImage alloc] initWithURL: <<URL of the image>>];
// Associate the image with the UIImageView.
imgView.image = img;
[img release];
```

Comments on the code shown in listing 1 shows the different steps in image creation and association with the GUI component. As the UIImageView instance retains a reference for the UIImage object, the image object can be released.

There are better ways to work with images than loading them statically from the internet. That is why Apple has prepared a framework that allow developers to get pictures from the local Photo library or the built-in camera (iPhone only); that is done with the classes **UIImagePickerController** and **UIImagePickerControllerDelegate**.

Most of the Apple frameworks use this delegate pattern. The App asynchronously requests a functionality to an object and set its delegate. The object handles the functionality

and sends messages to its delegate with updates about the original request.

On this specific case, the component responsible by getting an image from the Photo library or camera is the UIImagePickerController. Its delegate must implement the UIImagePickerControllerDelegate; another protocol that also needs to be implemented is the UINavigationControllerDelegate; this last protocol must be implemented not because of any image loading or processing capabilities, but for allowing the end-user to navigate inside their photo library if needed; the default implementation for this protocol is usually enough for most needs and nothing else needs to be added to the code for handling navigation events.

The code snippet in listing 2 shows how to create a dialog that allows the user to select an image from his/her photo library.

Listing 2: Creating a dialog to load images from the user's photo library.

```
// This method will be called when the load button is
// pressed.
-(void)buttonLoadPressed
  {
  UIImagePickerController *ipc =
     [[UIImagePickerController alloc] init];
  // Get images from the photo library.
  ipc.sourceType =
    UIImagePickerControllerSourceTypePhotoLibrary;
  // This class will deal with the dialog.
  ipc.delegate = self;
  // Show the picker controller.
  [self presentModalViewController:ipc animated:YES];
  }
```

After the user selects an image from its photo library, the method shown in listing 3 is called.

Listing 3: Code called when the user select an image from the photo library.

```
-(void) imagePickerController:
  (UIImagePickerController *)picker
  didFinishPickingMediaWithInfo:(NSDictionary *)info
  {
  // Call a method to display the image selected in the
  // dialog.
  [self displayImage:[info
   objectForKey:@"UIImagePickerControllerOriginalImage"]];
  // Dismiss the dialog and release the image picker.
  [self dismissModalViewControllerAnimated:YES];
  [picker release];
  }
```

The method shown in listing 3 receives a dictionary with several image-related information, including the image itself. From that point on, the App can do whatever needs to be done with the received image.

There are also error conditions that would trigger different messages to the delegate; one of those is the method **imagePickerControllerDidCancel** which is called when the user does not pick an image.

Storing the image back in the iPhone is quite simple: the code snippet shown in listing 4 shows a method that uses the image associated with UIImageView to store it. The method is called when the user clicks on a specific button, but the method that does the image storing is the `UIImageWriteToSavedPhotosAlbum`, which does all the work for us.

Listing 4: Code called when the user click a button to store the image.

```
// This method will be called when the save button is
// pressed.
-(void)buttonSavePressed
  {
  // Save the image on the photos album.
  UIImageWriteToSavedPhotosAlbum([imageView image],
                                 nil,nil,nil);
  }
```

B. Image Representation

Images are represented as instances of the class `UIImage`. Although this class contains several useful methods (e.g. to draw images, to create images from different sources) we will often need to process the images' pixels. To do this we recommend the use of the helper class `ImageHelper` (see [2]) that contains several macros and class methods, including:

- Methods to convert between instances of UIImage and arrays of characters for direct pixel access;
- Macros to get and set pixels in these arrays of characters;
- Methods to get images from views;
- Methods to scale an image and to center it around a view;

Using methods in this class makes manipulation of the pixels easy, as seen in the code snippet shown in listing 5, which gets an image from a view, processes it (converts it to grayscale) and display it.

Listing 5: Code called when the user decides to process an image.

```
// This method will be called with the process button is
// pressed.
-(void)buttonProcessPressed
  {
  // Get the image from the image view. Return if there is
  // no image.
  UIImage *image = [imageView image];
  if (image == nil) return;
  // Get the bytes from the image.
  unsigned char *inbits = (unsigned char *)
    [ImageHelper bitmapFromImage:image];
  // Alloc memory for the bytes for the output image.
  unsigned char *outbits =
     (unsigned char *)malloc(image.size.width*
                             image.size.height*4);
  // The main image processing loop. Scan all image pixels
  // and create output values from the input. In this
  // case we will convert the color image to a gray-level
  // one using the average of the red, green and blue
  // pixels.
  for(int h=0;h<image.size.height;h++)
    for(int w=0;w<image.size.width;w++)
      {
      int r = inbits[redOffset(w,h,image.size.width)];
      int g = inbits[greenOffset(w,h,image.size.width)];
      int b = inbits[blueOffset(w,h,image.size.width)];
      int gray = (r+g+b)/3;
      outbits[redOffset(w,h,image.size.width)] = gray;
      outbits[greenOffset(w,h,image.size.width)] = gray;
      outbits[blueOffset(w,h,image.size.width)] = gray;
      outbits[alphaOffset(w,h,image.size.width)] =
        inbits[alphaOffset(w,h,image.size.width)];
      }
  // Create the output image from its bytes.
  UIImage *output = [ImageHelper imageWithBits:outbits
                                 withSize:image.size];
  // Change the view.
  [self displayImage:output];
  }
```

C. Quartz2D

Quartz 2D is a set of APIs that allow developers to do basic drawing in addition to image manipulation. It provides basic drawing commands such as “draw an ellipse”, “fill a rect”, “draw a rect”, “draw an image”, etc.

Drawing commands are issued to a graphics context, which is represented by an instance of `CGContextRef`, which may be created in several different ways. One way to do this is through the method `CGBitmapContextCreate` which creates a bitmap graphics context. It is relatively simple to wrap code around instances of `CGContextRef` to draw on a bitmap graphics context and then return this context as an instance of `UIImage`, as shown in listing 6, which creates and returns an image containing a pie chart based on some predefined values.

Listing 6: Code to create a pie chart and return it as an UIImage.

```
-(UIImage *)drawPieChart
  {
  // Define an area where the chart will be draw. Use
  // the size of the view on the iPhone.
  CGRect workArea =
    CGRectMake(0,0,self.view.frame.size.width,
                   self.view.frame.size.height);
  // Set the center of the pie chart.
  int pixelsWide = workArea.size.width;
  int pixelsHigh = workArea.size.height;
  CGPoint chartCenter =
     CGPointMake(pixelsWide/2,pixelsHigh/2);
  // Manually creates a a Graphics Context.
  CGContextRef ctx=NULL;
  CGColorSpaceRef colorSpace;
  void* bitmapData;
  int bitmapByteCount;
  int bitmapBytesPerRow;
  // Calculate dimensions of the image.
  bitmapBytesPerRow =(pixelsWide*4); //RGBA
  bitmapByteCount =(bitmapBytesPerRow*pixelsHigh);
  // Allocate image buffer
  bitmapData=malloc(bitmapByteCount);
  if(bitmapData==NULL)
    {
    return NULL;
    }
  // Create an instance of RGB color space.
  colorSpace = CGColorSpaceCreateDeviceRGB();
  // Create a Graphic Context with this buffer -- whatever
  // we draw will be drawn on the buffer.
  ctx =
    CGBitmapContextCreate(bitmapData,
                          pixelsWide,pixelsHigh,
                          8, // Bits per component
                          bitmapBytesPerRow,
                          colorSpace,
                          kCGImageAlphaPremultipliedLast);
  // Returns NULL if context creation fails.
  if (ctx == NULL)
   {
   free(bitmapData);
   return NULL;
   }
  // Release color space because it is no longer needed.
  CGColorSpaceRelease(colorSpace);
  // Clear drawing area.
  CGContextClearRect(ctx, workArea);
  // Create a sample array of values
  NSMutableArray * dataSet = [[NSMutableArray alloc] init];
  [dataSet addObject:[NSNumber numberWithInt:10]];
  [dataSet addObject:[NSNumber numberWithInt:15]];
  [dataSet addObject:[NSNumber numberWithInt:30]];
  [dataSet addObject:[NSNumber numberWithInt:5]];
  [dataSet addObject:[NSNumber numberWithInt:18]];
  // Create an array of colors for each pie slice.
  NSMutableArray * colors = [[NSMutableArray alloc] init];
  [colors addObject:[UIColor blueColor]];
  [colors addObject:[UIColor greenColor]];
  [colors addObject:[UIColor redColor]];
  [colors addObject:[UIColor grayColor]];
  [colors addObject:[UIColor whiteColor]];
  // Sum all values in the array.
  float total = 0;
  for ( NSNumber * iTmp in dataSet )
    {
    total+=[iTmp intValue];
    }
  // Plots the Pie Chart
  float startDegree = 0;
  float endDegree = 0;
  float radius = (workArea.size.width/2)-15; // margin
  // Plot Items
  int item;
  CGContextSetStrokeColorWithColor(ctx,
                          [UIColor blackColor].CGColor);
  for(int i=0;i<dataSet.count;i++)
    {
    item = [(NSNumber*)[dataSet objectAtIndex:i] intValue];
    endDegree += (item*360.0f)/total;
    CGContextSetFillColorWithColor(ctx,
          ((UIColor*)[colors objectAtIndex:i]).CGColor);
    CGContextSetLineWidth(ctx, 5);
    CGContextMoveToPoint(ctx,chartCenter.x,chartCenter.y);
    CGContextAddArc(ctx,chartCenter.x,chartCenter.y,
                    radius,
                    startDegree*M_PI/180.0f,
                    endDegree*M_PI/180.0f,
                    0);
    CGContextFillPath(ctx);
    startDegree = endDegree;
    }
  // Create image to be returned from the
  // graphics context.
  CGImageRef img = CGBitmapContextCreateImage(ctx);
  UIImage* ret = [UIImage imageWithCGImage:img];
  // Free up all remaining memory.
  free(CGBitmapContextGetData(ctx));
  CGContextRelease(ctx);
  CGImageRelease(img);
  return ret;
  }
```

Listing 6 shows how to create a graphics context with an associated bitmap, and demonstrates the use of several drawing/painting methods (`CGContextSetStrokeColorWithColor`, `CGContextSetFillColorWithColor`, `CGContextMoveToPoint`, `CGContextAddArc`, `CGContextFillPath`, etc.). More examples of drawing on a bitmap will be shown later in this article.

D. A basic, complete example

So far we saw some code snippets that demonstrates which classes can be used for basic image reading, storing and processing and for drawing over a bitmap graphics context. To better demonstrate we will present the complete code for a basic iPhone image processing application.

Applications for the iPhone are created with the Xcode IDE – there are wizards to create applications with different styles, i.e. which have different sets of user interface widgets, appropriate for different purposes [2]. For example, one of the styles is “Navigation-based Application”, which organizes code to facilitate the creation of applications based on lists and tables for interaction with the user; another is “Utility Application”, which facilitates the creation of applications with a two-sided view for interaction with the user.

Most of the applications will have rich user interfaces, which can be easily created with the Interface Builder tool (see figure 2). Interface Builder allows the user to select different GUI components and to link code to process data entry and display into these components.

Since Interface Builder is interactive, we will not use it on this example: we will, instead, create the GUI of our basic application programmatically (components will be created through code). To do so, we must create the iPhone application in Xcode as a “View-based Application” – Xcode

will create the bare code to a view (a window to receive user interaction and/or to display information, and where we must add the GUI components) and to an app delegate (roughly speaking, the main part of the application, where the view will be created and displayed).

Our application will be a very simple one, but with code organized in such a way to be easily modified and extended. It will contain a view for displaying an image, and three buttons: one to load a image file from the iPhone's photo library) into the view, one to process the image with a specific algorithm and one to store back the image in the photo library.

Four source code files will be generated by Xcode. Assumming we chose "HelloImages" as a name, these files will be:

- **HelloImagesAppDelegate.h**, shown in listing 7, which is the header file for the application delegate.
- **HelloImagesAppDelegate.m**, shown in listing 8, which is the source code for the application delegate.
- **HelloImagesViewController.h**, shown in listing 9, which is the header file for the view/controller.
- **HelloImagesViewController.m**, shown in listing 10, which is the source code file for the view/controller, and which will contain both the programmatically-constructed GUI and the methods that will do the image processing.

Of course when creating a new view-based application the source code will be empty, please refer to comments on listings 7 to 8 for what must be written to add functionality to the application.

The header file for the application delegate (listing 7) declares that the class will inherit from NSObject and implement the protocol UIApplicationDelegate; and also declare some objects as properties of this class.

Listing 7: `HelloImagesAppDelegate.h`, header file for the application delegate.

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

@class HelloImagesViewController;

// Our interface extends NSObject and implement the
// protocol UIApplicationDelegate.
@interface HelloImagesAppDelegate :
  NSObject <UIApplicationDelegate>
  {
  // Contains its window and the view controller.
  UIWindow *window;
  HelloImagesViewController *viewController;
  }

// Properties of this app delegate.
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet
    HelloImagesViewController *viewController;

@end
```

The source code for the application delegate, shown in listing 8, declares the class itself. It contains the declaration for the method `didFinishLaunchingWithOptions`, which adds the main view; and the method `dealloc` which release resources allocated by this class.

Listing 8: `HelloImagesAppDelegate.m`, source code file for the application delegate.

```
#import "HelloImagesAppDelegate.h"
#import "HelloImagesViewController.h"

@implementation HelloImagesAppDelegate

@synthesize window;
@synthesize viewController;

// This method will be called when the application has
// finished launching -- it will create the main view and
// add it to the app's window.
- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:
        (NSDictionary *)launchOptions
  {
  [window addSubview:viewController.view];
  [window makeKeyAndVisible];
  return YES;
  }

// This method will release resources used in this class.
- (void)dealloc
  {
  [viewController release];
  [window release];
  [super dealloc];
  }

@end
```

The header file for the view/controller (listing 9) contains the declaration for the class (which will inherit from UIViewController) and for the implementation of some protocols. It also must contain the declarations for methods we will define on the class source code.

Listing 9: `HelloImagesViewController.h`, header file for the view/controller.

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

// Our interface extends UIViewController and implement
// the protocols UINavigationControllerDelegate,
// UIImagePickerControllerDelegate and
// UIScrollViewDelegate, used to react to some
// user interactions.
@interface HelloImagesViewController:
     UIViewController <UINavigationControllerDelegate,
                       UIImagePickerControllerDelegate,
                       UIScrollViewDelegate>
  {
  UIScrollView *scrollView;
  UIImageView *imageView;
  }

// We will implement and use this method.
- (void)displayImage:(UIImage *)image;

@end
```

The source code for the view/controller (listing 10) is where we will create the GUI, register events for user interaction and do the image processing.

Listing 10: `HelloImagesViewController.m`, source code file for the view/controller.

```
#import "HelloImagesViewController.h"

@implementation HelloImagesViewController

// Implement loadView to create a view hierarchy
// programmatically, without using a nib.
- (void)loadView
  {
  // Create a view to hold the components.
  UIView *contentView =
    [[UIView alloc] initWithFrame:
      [[UIScreen mainScreen] applicationFrame]];
  contentView.backgroundColor = [UIColor blackColor];

  // Create a UIButton programatically.
  UIButton *load = [UIButton buttonWithType:
                              UIButtonTypeRoundedRect];
  // Set its position on the view.
```

```
  load.frame = CGRectMake(4.0f,4.0f,100.0f,24.0f);
  // Set its title.
  [load setTitle:@"Load" forState:UIControlStateNormal];
  // Tell the application which method to call when the
  // button is pressed.
  [load addTarget:self action:@selector(buttonLoadPressed)
            forControlEvents:UIControlEventTouchUpInside];
  // Add the button to the view.
  [contentView addSubview:load];

  // Create a button labeled "process", which when
  // pressed will call the buttonProcessPressed method.
  UIButton *process = [UIButton buttonWithType:
                              UIButtonTypeRoundedRect];
  process.frame = CGRectMake(108.0f,4.0f,100.0f,24.0f);
  [process setTitle:@"Process"
                        forState:UIControlStateNormal];
  [process addTarget:self
                 action:@selector(buttonProcessPressed)
            forControlEvents:UIControlEventTouchUpInside];
  [contentView addSubview:process];

  // Create a button labeled "save", which when pressed
  // will call the buttonSavePressed method.
  UIButton *save = [UIButton buttonWithType:
                              UIButtonTypeRoundedRect];
  save.frame = CGRectMake(212.0f,4.0f,100.0f,24.0f);
  [save setTitle:@"Save" forState:UIControlStateNormal];
  [save addTarget:self action:@selector(buttonSavePressed)
            forControlEvents:UIControlEventTouchUpInside];
  [contentView addSubview:save];

  // Create a scrollView to hold the image.
  scrollView = [[[UIScrollView alloc]
     initWithFrame:CGRectMake(4.0f,32.0f,312.0f,444.0f)]
     autorelease];
  scrollView.delegate = self;
  // Add it to the view.
  [contentView addSubview:scrollView];
  // Set this applications' view.
  self.view = contentView;
  [contentView release];
  }

// This method will be called whenever the image changes.
-(void)displayImage:(UIImage *)image
  {
  // Recreate the UIImageView instance.
  imageView = [[[UIImageView alloc] initWithImage:image]
    autorelease];
  // Add it to the view.
  [scrollView addSubview:imageView];
  scrollView.contentSize = image.size;
  }

// This method will be called when the load button is
// pressed.
-(void)buttonLoadPressed
  {
  // Create an image pick controller.
  UIImagePickerController *ipc =
    [[UIImagePickerController alloc] init];
  // Get images from the photo library.
  ipc.sourceType =
    UIImagePickerControllerSourceTypePhotoLibrary;
  // This class will deal with the dialog.
  ipc.delegate = self;
  // Show it.
  [self presentModalViewController:ipc animated:YES];
  }

// This method will be called with the process button is
// pressed.
-(void)buttonProcessPressed
  {
  // Get the image from the image view. Return if there is
  // no image.
  UIImage *image = [imageView image];
  if (image == nil) return;
  // Get the bytes from the image.
  unsigned char *inbits = (unsigned char *)
    [ImageHelper bitmapFromImage:image];
  // Alloc memory for the bytes for the output image.
  unsigned char *outbits =
     (unsigned char *)malloc(image.size.width*
                             image.size.height*4);
  // The main image processing loop. Scan all image pixels
  // and create output values from the input. In this
  // case we will convert the color image to a gray-level
  // one using the average of the red, green and blue
  // pixels.
  for(int h=0;h<image.size.height;h++)
    for(int w=0;w<image.size.width;w++)
      {
      int r = inbits[redOffset(w,h,image.size.width)];
      int g = inbits[greenOffset(w,h,image.size.width)];
      int b = inbits[blueOffset(w,h,image.size.width)];
      int gray = (r+g+b)/3;
      outbits[redOffset(w,h,image.size.width)] = gray;
      outbits[greenOffset(w,h,image.size.width)] = gray;
      outbits[blueOffset(w,h,image.size.width)] = gray;
      outbits[alphaOffset(w,h,image.size.width)] =
        inbits[alphaOffset(w,h,image.size.width)];
      }
  // Create the output image from its bytes.
  UIImage *output = [ImageHelper imageWithBits:outbits
                              withSize:image.size];
  // Change the view.
  [self displayImage:output];
  }

// This method will be called when the save button is
// pressed.
-(void)buttonSavePressed
  {
  // Save the image on the photos album.
  UIImageWriteToSavedPhotosAlbum([imageView image],
                              nil,nil,nil);
  }

// This method will be called when we finish the
// selection of an image.
-(void) imagePickerController:
  (UIImagePickerController *)picker
  didFinishPickingMediaWithInfo:(NSDictionary *)info
  {
  [self displayImage:[info objectForKey:
            @"UIImagePickerControllerOriginalImage"]];
  [self dismissModalViewControllerAnimated:YES];
  [picker release];
  }

// This method will be called if the image selection
// dialog is dismissed (cancelled)
-(void) imagePickerControllerDidCancel:
    (UIImagePickerController *)picker
  {
  [self dismissModalViewControllerAnimated:YES];
  [picker release];
  }

// Release the resources used in this class.
- (void)viewDidUnload
  {
  [scrollView release];
  [imageView release];
  [super viewDidUnload];
  }

// Dealloc memory used in this class.
- (void)dealloc
  {
  [super dealloc];
  }

@end
```

The most relevant methods of the class `HelloImagesViewController.m` (listing 10) are:

- **loadView** is called when the view is created, and is where we will programmatically create the GUI for our application. On this method we can see the creation of three instances of UIButton, corresponding to the three buttons on our application. The buttons will be positioned on fixed locations on the application window. Each button will also have an associated selector – a method that will be called when that button is pressed.
- **displayImage** is an auxiliary method that will be called when we need to redisplay an image. It basically recreates the instance of UIImageView that is responsible for showing the image. The UIImageView component will be contained by an instance of UIScrollView which will automatically allow the scrolling of images which are larger than the device's screen.
- **buttonLoadPressed** is the selector method that will be called when the "Load" button is pressed. It will, in turn, create an instance of UIImagePickerController that will allow the user to pick an image from the photo library. Control will be passed to that dialog, which will call yet another method depending on the

Figure 3: Screenshot for the first application: GUI just loaded.

Figure 4: Screenshot for the first application: user pressed the "Load" button.

users' selection (selection of an image or canceling the selection).

- **buttonProcessPressed** is the selector method that will be called when the "Process" button is pressed. It will scan the pixels on the image and, in this example, convert a color image to a gray-level one by averaging the red, green and blue values for each pixel. This method uses the auxiliary class ImageHelper (see [2]).
- **buttonSavePressed** is the selector method that will be called when the "Save" button is pressed. It will store the image associated to the instance of UIImageView into the photo library.
- **imagePickerController** is the method that will be called when the user selects an image from the photo library. It will get that image reference and use it to call the displayImage method.
- **imagePickerControllerDidCancel** is the method that will be called when the user decides to cancel the selection of an image from the photo library.
- **viewDidUnload** will be called when the view is unloaded; it will release the objects used in the class.
- **dealloc** will be called when there is the need to deallocate memory allocated by this class.

These methods define the basic behaviour of our application, which can be seen in the screenshots in figures 3 to 6.

To demonstrate the simplicity of the example, we developed two other versions of the selector method buttonProcessPressed, to illustrate how other image processing algorithms could be used instead of the gray-level conversion. For those other examples, we didn't change the code for the header files or the application delegate source code; only the code for the buttonProcessPressed method was changed.

The second version of the buttonProcessPressed selector method is shown in listing 11. It also scans the image's

Figure 5: Screenshot for the first application: user selected an image from the photo library and it is being shown in the UIImageView component.

pixels one by one, using methods on the ImageHelper class, changing the output pixels using a different combination of the bands on the input image – the RGB values on the output image will be assigned to the GBR bands from the input image (there is another band, alpha or transparency, that will not be changed).

Listing 11: Method `buttonProcessPressed` on `HelloImagesViewController.m`, second version.

```
// This method will be called with the process button is
// pressed.
-(void)buttonProcessPressed1
  {
  // Get the image from the image view. Return if there is
  // no image.
  UIImage *image = [imageView image];
  if (image == nil) return;
  // Get the bytes from the image.
  unsigned char *inbits = (unsigned char *)
    [ImageHelper bitmapFromImage:image];
```

Figure 6: Screenshot for the first application: user pressed the "Process" button which processes the image and displays it.

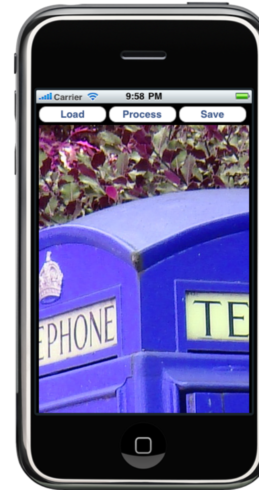

Figure 7: Screenshot for the second application: user pressed the "Process" button which processes the image and displays it.

```
  // Alloc memory for the bytes for the output image.
  unsigned char *outbits =
     (unsigned char *)malloc(image.size.width*
                             image.size.height*4);
  // The main image processing loop. Scan all image pixels
  // and create output values from the input. In this
  // case we change the image's bands order to GBR.
  for(int h=0;h<image.size.height;h++)
   for(int w=0;w<image.size.width;w++)
     {
     outbits[redOffset(w,h,image.size.width)] =
         inbits[greenOffset(w,h,image.size.width)];
     outbits[greenOffset(w,h,image.size.width)] =
         inbits[blueOffset(w,h,image.size.width)];
     outbits[blueOffset(w,h,image.size.width)] =
         inbits[redOffset(w,h,image.size.width)];
     outbits[alphaOffset(w,h,image.size.width)] =
         inbits[alphaOffset(w,h,image.size.width)];
     }
  // Create the output image from its bytes.
  UIImage *output = [ImageHelper imageWithBits:outbits
                                   withSize:image.size];
  // Change the view.
  [self displayImage:output];
  }
```

Results of the application of the code on listing 11 are shown in figure 7.

The third version of the buttonProcessPressed selector method is shown in listing 12. It scans the image's pixels but instead of creating a value for the output pixels based only on the input pixels on the same position it applies the Sobel operator [3] for edge detection. This operator requires the calculation of a value using pixel values around the pixel of interest, being significantly slower than the other image processing examples shown so far. Results for processing the test image on our application with that method are shown in figure 8.

Listing 12: Method `buttonProcessPressed` on `HelloImagesViewController.m`, third version.

```
// This method will be called with the process button is
// pressed.
-(void)buttonProcessPressed1
  {
  // Get the image from the image view. Return if there is
  // no image.
  UIImage *image = [imageView image];
  if (image == nil) return;
  // Get the bytes from the image.
  unsigned char *inbits = (unsigned char *)
    [ImageHelper bitmapFromImage:image];
  // Alloc memory for the bytes for the output image.
  unsigned char *outbits =
     (unsigned char *)malloc(image.size.width*
                             image.size.height*4);
  // The main image processing loop. Scan all image pixels
  // and create output values from the input. In this
  // case we will calculate the Sobel filter using a 3x3
  // neighborhood around the central pixel.
  for(int h=1;h<image.size.height-1;h++)
    for(int w=1;w<image.size.width-1;w++)
      {
      // Calculate the horizontal gradient for each band.
      int gx_r =
        (-1)*inbits[redOffset(w-1,h-1,image.size.width)]+
        (-2)*inbits[redOffset(w,h-1,image.size.width)]+
        (-1)*inbits[redOffset(w+1,h-1,image.size.width)]+
        ( 1)*inbits[redOffset(w-1,h+1,image.size.width)]+
        ( 2)*inbits[redOffset(w,h+1,image.size.width)]+
        ( 1)*inbits[redOffset(w+1,h+1,image.size.width)];
      int gx_g =
        (-1)*inbits[greenOffset(w-1,h-1,image.size.width)]+
        (-2)*inbits[greenOffset(w,h-1,image.size.width)]+
        (-1)*inbits[greenOffset(w+1,h-1,image.size.width)]+
        ( 1)*inbits[greenOffset(w-1,h+1,image.size.width)]+
        ( 2)*inbits[greenOffset(w,h+1,image.size.width)]+
        ( 1)*inbits[greenOffset(w+1,h+1,image.size.width)];
      int gx_b =
        (-1)*inbits[blueOffset(w-1,h-1,image.size.width)]+
        (-2)*inbits[blueOffset(w,h-1,image.size.width)]+
        (-1)*inbits[blueOffset(w+1,h-1,image.size.width)]+
        ( 1)*inbits[blueOffset(w-1,h+1,image.size.width)]+
        ( 2)*inbits[blueOffset(w,h+1,image.size.width)]+
        ( 1)*inbits[blueOffset(w+1,h+1,image.size.width)];
      // Calculate the vertical gradient for each band.
      int gy_r =
        (-1)*inbits[redOffset(w-1,h-1,image.size.width)]+
        (-2)*inbits[redOffset(w-1,h,image.size.width)]+
        (-1)*inbits[redOffset(w-1,h+1,image.size.width)]+
        ( 1)*inbits[redOffset(w+1,h-1,image.size.width)]+
        ( 2)*inbits[redOffset(w+1,h,image.size.width)]+
        ( 1)*inbits[redOffset(w+1,h+1,image.size.width)];
      int gy_g =
        (-1)*inbits[greenOffset(w-1,h-1,image.size.width)]+
        (-2)*inbits[greenOffset(w-1,h,image.size.width)]+
        (-1)*inbits[greenOffset(w-1,h+1,image.size.width)]+
        ( 1)*inbits[greenOffset(w+1,h-1,image.size.width)]+
        ( 2)*inbits[greenOffset(w+1,h,image.size.width)]+
        ( 1)*inbits[greenOffset(w+1,h+1,image.size.width)];
      int gy_b =
        (-1)*inbits[blueOffset(w-1,h-1,image.size.width)]+
        (-2)*inbits[blueOffset(w-1,h,image.size.width)]+
        (-1)*inbits[blueOffset(w-1,h+1,image.size.width)]+
        ( 1)*inbits[blueOffset(w+1,h-1,image.size.width)]+
        ( 2)*inbits[blueOffset(w+1,h,image.size.width)]+
        ( 1)*inbits[blueOffset(w+1,h+1,image.size.width)];
      // Calculate (and normalize) the gradient magnitude
      // for each band.
      int edge_r = sqrt(gx_r*gx_r+gy_r*gy_r)/5.635;
      int edge_g = sqrt(gx_g*gx_g+gy_g*gy_g)/5.635;
      int edge_b = sqrt(gx_b*gx_b+gy_b*gy_b)/5.635;
      // Use the edge values as the output pixel values.
      outbits[redOffset(w,h,image.size.width)] = edge_r;
```

Figure 8: Screenshot for the third application: user pressed the "Process" button which processes the image and displays it.

```
      outbits[greenOffset(w,h,image.size.width)] = edge_g;
      outbits[blueOffset(w,h,image.size.width)] = edge_b;
      // Keep the alpha channel unchanged.
      outbits[alphaOffset(w,h,image.size.width)] =
         inbits[alphaOffset(w,h,image.size.width)];
      }
   // Create the output image from its bytes.
   UIImage *output = [ImageHelper imageWithBits:outbits
                                  withSize:image.size];
   // Change the view.
   [self displayImage:output];
   }
```

III. Drawing on an iPhone

As in many other platforms, when we want do draw something on the screen of the iPhone we must use a graphics context – an abstraction that can receive drawing commands which will be mapped to a more concrete implementation, such as a screen or a printable page. As mentioned in section II-C there are several ways to create a graphics context for the iPhone, we will present another example that creates and uses a graphics context that can be converted into an image for displaying.

In the example on this section we will create an image containing three histograms calculated from the pixels of another image, selected by the user. The three histograms represents the distribution of the pixels of the original image in the red, green and blue channel.

In order to create the image with the histogram we will declare another method that can be used in the same code framework shown before. The first thing we need to do is declare the method on the header file for the view/-controller (file HelloImagesViewController.h), by adding the declaration `-(UIImage *)createHistograms;`. The declaration of the method itself is shown in listing 13.

Listing 13: Method `createHistograms` which will calculate and draw the RGB histograms of an image.

```
// This method will scan the pixels of an image and
// create three histograms, one for each of the R,G,B
// bands on the image. The histograms will be plotted in
// the graphical context. The graphical context is
// represented as a bitmap, which will be converted to an
// UIImage and returned.
-(UIImage *)createHistograms
  {
  // Declare some constants for the histogram plot.
  int margin = 10;
  float binW = 2;
  int histogramW = (256*binW);
  int histogramH = 140;
  // Define an area where the graphics will be drawn.
  CGRect workArea =
    CGRectMake(0,0,
               histogramW+2*margin,
               2*margin+2*margin+3*histogramH);
  // Manually creates a graphics context.
  int pixelsWide = workArea.size.width;
  int pixelsHigh = workArea.size.height;
  CGContextRef ctx = NULL;
  CGColorSpaceRef colorSpace;
  void* bitmapData;
  int bitmapByteCount;
  int bitmapBytesPerRow;
  bitmapBytesPerRow =(pixelsWide*4); // RGBA
  bitmapByteCount =(bitmapBytesPerRow*pixelsHigh);
  // Allocate an image buffer for the graphics context.
  bitmapData = malloc(bitmapByteCount);
  if(bitmapData == NULL)
    {
    return NULL;
    }
  // Create an instance of RGB color space.
  colorSpace = CGColorSpaceCreateDeviceRGB();
  // Create a Graphic Context with this buffer -- whatever
  // we draw will be drawn on the buffer.
  ctx =
    CGBitmapContextCreate(bitmapData,
                          pixelsWide,pixelsHigh,
                          8, // Bits per component
                          bitmapBytesPerRow,
                          colorSpace,
                          kCGImageAlphaPremultipliedLast);
  // Returns NULL if context creation fails.
  if (ctx == NULL)
   {
   free(bitmapData);
   return NULL;
   }
  // Release color space because it is no longer needed.
  CGColorSpaceRelease(colorSpace);
  // Clear drawing area (paints it using a dark gray
  // color).
  CGContextSetRGBFillColor(ctx,0.3,0.3,0.35,1);
  CGContextFillRect(ctx,workArea);
  // Use a thick white line to draw the histograms.
  CGContextSetStrokeColorWithColor(ctx,
     [UIColor whiteColor].CGColor);
  CGContextSetLineWidth(ctx,3.0);
  // The red histogram.
  CGContextMoveToPoint(ctx,margin,pixelsHigh-margin);
  CGContextAddLineToPoint(ctx,margin,
                          pixelsHigh-(margin+histogramH));
  CGContextAddLineToPoint(ctx,margin+histogramW,
                          pixelsHigh-(margin+histogramH));
  CGContextAddLineToPoint(ctx,margin+histogramW,
                                       pixelsHigh-margin);
  // The green histogram.
  CGContextMoveToPoint(ctx,margin,
                        pixelsHigh-(margin*2+histogramH));
  CGContextAddLineToPoint(ctx,margin,
                      pixelsHigh-(margin*2+histogramH*2));
  CGContextAddLineToPoint(ctx,margin+histogramW,
                      pixelsHigh-(margin*2+histogramH*2));
  CGContextAddLineToPoint(ctx,margin+histogramW,
                        pixelsHigh-(margin*2+histogramH));
  // The blue histogram.
  CGContextMoveToPoint(ctx,margin,
                      pixelsHigh-(margin*3+histogramH*2));
  CGContextAddLineToPoint(ctx,margin,
                      pixelsHigh-(margin*3+histogramH*3));
  CGContextAddLineToPoint(ctx,margin+histogramW,
                      pixelsHigh-(margin*3+histogramH*3));
  CGContextAddLineToPoint(ctx,margin+histogramW,
                      pixelsHigh-(margin*3+histogramH*2));
  // Draw it all.
  CGContextStrokePath(ctx);
  // Get the image data and calculate the histograms.
  UIImage *image = [imageView image];
  if (image != nil)
    {
    unsigned char *inbits =
      (unsigned char *)[ImageHelper bitmapFromImage:image];
    // Allocate memory for the three histograms, clear
    // the memory to avoid nasty surprises.
    int *redHisto = malloc(256 * sizeof(int));
    memset(redHisto,0,256*sizeof(int));
    int *greenHisto = malloc(256 * sizeof(int));
    memset(greenHisto,0,256*sizeof(int));
    int *blueHisto = malloc(256 * sizeof(int));
```

```
    memset(blueHisto,0,256*sizeof(int));
    // Scan the image pixels to fill the histograms.
    for(int h=0;h<image.size.height;h++)
      for(int w=0;w<image.size.width;w++)
        {
        redHisto[
           inbits[redOffset(w,h,image.size.width)]]++;
        greenHisto[
           inbits[greenOffset(w,h,image.size.width)]]++;
        blueHisto[
           inbits[blueOffset(w,h,image.size.width)]]++;
        }
    // Let's get the largest value to normalize the
    // histogram height.
    int max = -1;
    for(int b=0;b<256;b++)
      {
      max = MAX(max,redHisto[b]);
      max = MAX(max,greenHisto[b]);
      max = MAX(max,blueHisto[b]);
      }
    // Select a thick line to draw the histogram bars.
    CGContextSetLineWidth(ctx,2.0);
    // Draw the red histogram bars.
    CGContextSetFillColorWithColor(ctx,
        [UIColor redColor].CGColor);
    for(int b=0;b<256;b++)
      {
      CGContextFillRect(ctx,
                  CGRectMake(margin+b*binW,
                  pixelsHigh-(margin+histogramH),binW,
                  (histogramH*(1.0*redHisto[b]/max))));
      }
    // Draw the green histogram bars.
    CGContextSetFillColorWithColor(ctx,
        [UIColor greenColor].CGColor);
    for(int b=0;b<256;b++)
      {
      CGContextFillRect(ctx,
                  CGRectMake(margin+b*binW,
                  pixelsHigh-(2*margin+2*histogramH),binW,
                  (histogramH*(1.0*greenHisto[b]/max))));
      }
    // Draw the blue histogram bars.
    CGContextSetFillColorWithColor(ctx,
        [UIColor blueColor].CGColor);
    for(int b=0;b<256;b++)
      {
      CGContextFillRect(ctx,
                  CGRectMake(margin+b*binW,
                  pixelsHigh-(3*margin+3*histogramH),binW,
                  (histogramH*(1.0*blueHisto[b]/max))));
      }
    }
  // Create image from the graphics context.
  CGImageRef img = CGBitmapContextCreateImage(ctx);
  UIImage *ret = [UIImage imageWithCGImage:img];
  // Free up all remaining memory.
  free(CGBitmapContextGetData(ctx));
  CGContextRelease(ctx);
  CGImageRelease(img);
  free(redHisto); free(greenHisto); free(blueHisto);
  return ret;
  }
```

The method createHistograms, shown in listing 13, implements the following steps:

- Creates a bitmap with an area large enough to contain all the three histograms. Some constants allow the easy manipulation of scales and margins, which affect the size of this area. Since the image this method created will be displayed within an instance of UIScrollView it may be larger than the physical dimensions of the screen.
- Creates a graphics context based on that bitmap, meaning that when we use drawing functions for primitives such as lines and rectangles, they will be renderized on that bitmap.
- Clears the bitmap with a dark gray color and draws the basic shapes for the histograms with thick white lines.
- Creates the data structures (arrays) to hold the three histograms and clear them.
- Reads the image associated with the UIImageView component and populates the histogram arrays.
- Draws each histogram with the data on the corresponding array.
- Frees memory allocated in the method.
- Return an instance of UIImage which was created with the bitmap associated to the graphics context.

Figure 9: Screenshot of the histogram application (histogram corresponds to image shown in figure 5).

With that method that creates an UIImage from the data on another UIImage we can modify the selector method associated with the "Process" button. The modification is shown in the code snippet in listing 14.

Listing 14: Selector method for the "Process" button.

```
// This method will be called with the process button is
// pressed.
-(void)buttonProcessPressed
  {
  // Calls createHistogram that will draw on an image
  // for us.
  UIImage *chart = [self createHistograms];
  [self displayImage:chart];
  }
```

Figure 9 shows the screenshot for the application. Part of the histograms is not shown because the plotting area is larger than the device screen, but it can be scrolled to display the other regions.

An important last note on using graphic contexts to draw on the iPhone: the Quartz graphics environment origin (0.0,0.0) is located in the lower left corner, differently from other environment and platforms.

IV. Comments on Performance and Limitations

Differently from the traditional desktop, iPhones and similar devices are not upgradable (or even user-serviceable). As such, developers need to deal with their capabilities and limitations on their own. There are at least a few issues that developers must be aware before starting developing applications for such devices.

A. *Memory and storage management*

The memory and storage capacity have been increasing in each new generation of devices. However, developing for those devices can be challenging, specially if your application may also be required to run on earlier hardware generations. In general, as available memory and storage can change from device to device, Apple built their frameworks to make easier for developers to be prepared for bottleneck situations.

As this can change from time to time, these lines should not be seen as a final recommendation; developers should always stay in touch with Apple to be aware of the latest best-practices and guidelines. Here are a few best practices that one may want to follow:

- As this paper is written, the recommendation from Apple regarding storage usage is that each application should do its best to avoid using more than 100MB of storage space. As far as this team knows, there is not physical or logical lock to reenforce this limit but that is what can be found at the Apple website in order to make sure users can take better advantage of their device storage.
- Memorywise, Apple has no logical size limits, however, as memory is a lot more critical than storage space, the recommendation is that the application uses as little memory as possible. iOS has a memory policy that may be more or less restrictive depending on the device. In a nutshell, the way it works is that the operating system continuously monitor the running applications; if the device starts running low on memory, iOS will start sending "Memory warning" events to all applications. This should be the trigger for the application to release as much memory as possible (clear images, release hidden UI components, etc).
  If, even with the "Memory warning" alarms the device still is low on memory, iOS will start killing the running applications. The current iOS policy for killing applications starts from the highest memory users to the lowest. This mean that it is always possible for your application to be killed but if you keep a very small memory footprint, it is more likely that it will be one of the last applications to be terminated.

B. *CPU and pseudo-multithreading*

iOS itself is not a fully multithreaded OS. In fact, Apple has implemented a "user-facing" multithreaded OS only in its newest version, iOS4. For users, applications will feel like if they were running in parallel when, in fact, they are not.

In order to build this "user-facing" multithreaded OS, Apple has enhanced their frameworks with hooks that tell the developers about the several applications' states. The main best-practice that Apple recommends for multitasking is that applications should continuously save their states. This means that applications should be smart enough to keep their current state safe before they enter into the background and they will have to reload this state right after the user decides to bring the application back to the foreground mode.

The reason why Apple recommends that you continuously save your state (as in checkpoints of a game) is because when the user shifts from one app to the other or just closes an application, iOS will give just a few seconds to the application shutdown itself; if the application does not shutdown on time (and this time may vary from application to application) iOS will kill the application and the state will be lost. This hard cut is necessary to provide a nice user experience. The end-user would not feel like using a multithreaded environment it he/she had to wait a very long time to switch from application to application.

The bottom line here is: do not expect to show the users a window asking if they really want to close the application or if they want to save their current work. On an iOS device, users expect that those implementation details are all handled by the application. They know that they can close their application at any time they want and their work will be there for them when they get back.

From the hardware perspective, this paper will not make available any performance comparisons or benchmark information. The reason for that is because Apple has not released, up to date, any official information about it. There are informal reviews that can be found over the web but it is a team decision to not use unofficial data that could compromise the accuracy of this paper.

V. CONCLUSION

This paper presented the basics on software development for the iPhone and related platforms, with demonstration on code that does some basic image processing and drawing tasks. Due to the lack of space, more techniques and sample code were not shown but will be available from the authors at the Sibgrapi 2010 tutorial.

Since most of the fun of the iPhone is just its user experience developers must learn to follow a strict set of guidelines to avoid creating unresponsive applications that may be terminated by the operating system at any time. While the platform is not suitable for large-scale or processing-intensive applications, there are several possibilities of using it as a remote device for image processing, computer graphics and pattern recognition applications.

REFERENCES

[1] R. Wagner, Ed., *"Safari WebKit Development for iPhone OS 3.0"*. Wiley Publishing Inc., 2010.

[2] E. Sadun, Ed., *"The iPhone Developer's Cookbook – Building Applications with the iPhone 3.0 SDK"*, 2nd ed. Addison-Wesley, 2010.

[3] R. C. Gonzalez and P. Wintz, *Digital Image Processing*. Reading, Massachusetts: Addison-Wesley, 1987.