ITK Image IO Interface with Apple iOS

Release 1.00

Boris Shabash[1], Ghassan Hamarneh[1], Zhi Feng Huang[1], and Luis Ibanez[2]

September 12, 2010

[1]School of Computing Science, Simon Fraser University, BC, Canada
[2]The Insight Group

Abstract

We have recently detailed the procedure for building ITK on the iOS[1]. In this work, we contribute `itkiOSImageIO` , the necessary ITK class that provides the input/output (IO) interface with the repository of images stored on iOS Apple devices, such as the iPod touch, iPhone, or iPad. The proposed classes provide the ITK programmer with the facility to read or write iOS images. Along with contributing the source code for these classes, we provide examples of reading and writing different types of gray level and color iOS images, as well as filtering the images using ITK filters.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3216]

Contents

1 Introduction

Mobile computing devices are becoming increasingly important and prevalent[2, 3]. Apple's mobile devices, the iPod touch, iPhone, and iPad are becoming increasingly popular. These devices are now featuring computing power and storage at par with what personal computers featured only some years ago. Furthermore, these mobile devices are providing more stunning displays (e.g. the new Retina display on Apple's iPhone 4 features 326 pixels per inch[1]), with intuitive multi-touch display interface.

Health applications are already benefiting from mobile computing to enable portable health care applications[4]. It is also undeniable how health applications are increasingly relying on the processing, analysis, visualization, and interaction with biomedical image, e.g. magnetic resonance imaging (MRI) or computed tomography)[5].

The Insight ToolKit (ITK)[2] is the de facto library for medical image analysis[6]. ITK is becoming increasingly popular given the large number of classes for basic and advanced image processing, segmentation, and registration algorithms, combined with the continuous growth though user-contributed source code from leading groups around the world. Numerous medical image analysis applications have been developed based on ITK and different software packages now incorporate or interface with ITK functionality, e.g. VolView[3], Analyze[4], MATITK[7], and many others.

We are interested in the marriage of these two parallel developments: ITK for medical image analysis and iOS for mobile computing devices. We have recently detailed the procedure for building ITK on the iOS[1]. That work opened the way to process, analyses, segment, register images. However, what is missing now is the ability to interface with the iOS for reading and writing images.

ITK supports a large number of medical image file formats[5], e.g. DICOM, MetaImage, and NRRD, and the list of supported formats continues to grow through open source user contributions, e.g.[8, 9, 10]. However, the current image file support does not allow images to be read from the iOS because of the way iOS handles images.

While in other operating systems the images are specified by a path to the image file itself, the iOS software development kit (SDK) does not allow the image path to be retrieved. Instead, it uses the `UIImage` class[6] which contains all the image pixel data and meta-data. In other words, all image information can only be retrieved and saved via the `UIImage` object. The files themselves are abstracted even from the programmer and only their data can be manipulated. Based on the `UIImage` class, in this work, we develop `itkiOSImageIO`, the necessary ITK class that provides the interface with the repository of images stored on iOS devices; the iPod touch, iPhone, and the iPad.

Along with this written report, this publication includes supplementary source code for the aforementioned classes. The provided classes bestow on the programmer the ability to easily read or write iOS images (images in the photo or image library of Apple's mobile devices). We further provide examples of reading and writing different types of images (graylevel and color images with and without alpha channels), which act as test suite for our classes as well as template code that programmers can reuse in their software development.

In the remainder of this paper, we discuss the methods that were implemented for both reading and writing

[1] http://www.apple.com/iphone/features/retina-display.html
[2] http://www.itk.org/
[3] http://www.kitware.com/products/volview.html
[4] http://www.analyzedirect.com
[5] http://www.paraview.org/Wiki/ITK_File_Formats
[6] http://developer.apple.com/library/ios/documentation/uikit/reference/UIImage_Class/UIImage_Class.pdf

iOS images (Section 2.1). Code snippets for reading different image type is described in Section 2.2. In Section 2.3, we describe the overall procedure for making use of the `itkiOSImageIO` in an iOS device app. Finally, we conclude and mention some ideas for future work in Section 3.

2 Method

2.1 ITK Image IO Methods and Interface

In order to acquire and handle images from the iOS, we develop a key class which handle iOS image pixel data and related meta-data, `itkiOSImageIO`. `itkiOSImageIO` is basically a class designed to deal with input and output of the `UIImage` class. It extracts the image data as well as properties from the `UIImage` class and makes them accessible to other ITK functions and classes.

The critical methods in `itkiOSImageIO` which implemented are:

- `virtual bool CanReadFile(const char*)` - A method that determines if the IO class can read the image in question specified by the path `const* char`.
- `virtual void ReadImageInformation()` - A method which reads the properties (dimensions, colour scheme, tropicity[7], etc.) of the current image in question. Note that this function does not read the individual pixel data yet.
- `virtual void Read(void* buffer)` - A method which reads the current image data (the actual value at each pixel).
- `virtual bool CanWriteFile(const char*)` - A method that determines if the IO class can write the image in question to the path specified by `const* char`.
- `virtual void WriteImageInformation()` - A method that writes the image information to the current image. Note this does not write the pixel data yet.
- `virtual void Write(const void* buffer)` - A method that writes the image data into the image file. This writes the actual pixel data.
- `virtual ImageIORegion GenerateStreamableReadRegionFromRequestedRegion( const ImageIORegion &requested ) const` - A method that determines the largest streamable region for the image in case several streaming operations are required to stream a large image.

As noted briefly in Section 1, the iOS does not allow the software to access image paths or names, but instead returns an `UIImage` object as the only access to the image. As a result, the methods `CanReadFile` and `CanWriteFile` which accept a `const* char` as their argument must be modified accordingly. In addition, the method `virtual void SetFileName(const* char)` had to be redefined as `virtual void SetFileName(UIImage*)`. With the interface of the IO class set, we detail the implementation of the different methods. The implementation of `CanReadFile` and `CanWriteFile` methods is trivial and isn't discussed in detail. We simply return 'true' as long as the object passed as an argument is an `UIImage` instance and 'false' otherwise. Similarly, the implementation of `virtual ImageIORegion GenerateStreamableReadRegionFromRequestedRegion( const`

[7] Tropcity, according to Apple's iOS SDK terminology, refers to the property of a pixel being isotropic (unity aspect ratio of width to height) or anisotropic (non-unity aspect ratio).

`ImageIORegion &requested) const` is trivial since the images can all be streamed in one chunk. Basically the entire image is returned as the largest streamable region. The following sections present a detailed overview of the core methods responsible for reading and writing.

Implementing Reading Methods

The first nontrivial method to implement is `ReadImageInformation()`. The necessary image properties that need to be read via this method are the dimensions of the image (width and height), the number of bits used to store each color component value, as well as the interpretation of these bits, e.g. 8 bits per pixel for a 256-level grayscale image or 32 bits per pixel for Red, Green, Blue, and Alpha transparency (RGBA) values of color images).

Fortunately, these necessary components are accessible via Objective-C queries to the `CGImage` object, which can be obtained from the `UIImage` object via the method call (or 'message', as it is referred to in Objective-C) `[theUIImagePointer CGImage]`. The dimensions of the image are available via the messages `CGImageGetWidth([UIImagePointer CGImage])` and `CGImageGetHeight([UIImagePointer CGImage])`, whereas the number of bits used to store each color component can be easily found by using the query `size_t numBitsPerComponent = CGImageGetBitsPerComponent(theCGImageRef)`.

For proper interpretation of the bits assigned to pixel, we query the `ColorSpace` using `CGColorSpaceRef theColorSpace = CGImageGetColorSpace(theCGImageRef)`. This method returns an eumeration whose value indicates the number of color space components in the image and whether an alpha value is used. In particular, grayscale images have a single color space component, RGB images have three, whereas RGBA images have four[8].

The method `virtual void Read(void* buffer)` needs the data buffer `void* buffer` to be filled with the pixel data. Unfortunately, the image data is not easily accessible to the programmer. There is no way to directly extract the image data from the `UIImage` object or from the `CGImage` object. Instead, we used a code snippet posted publicly online[9]. In essence, this code creates a new image context in Objective-C (equivalent to setting up a canvas for an artist), then redraws the image in the new context with `buffer` as a pointer to the context data (equivalent to redrawing the image, this time recording how the redrawing was done with a video camera). The following steps expose the details:

1. Obtain the meta-data information required to create the context. This includes which are the number of bits per component, the number of bits per row, the color space information, endianness, the existence of an alpha (transparency) channel or not, and the width and height in pixels of the image.

2. Create the context with the command

```
CGContextRef theContext = CGBitmapContextCreate(buffer,
                                                width,
                                                height,
                                                bitsPerComponent,
                                                bytesPerRow,
                                                colorSpace,
                                                theBitmapInfo)
```

[8] Note that it is possible to have an RGB image (not RGBA) but with four color space components. In this case, the fourth component must be ignored.

[9] http://stackoverflow.com/questions/448125/how-to-get-pixel-data-from-a-uiimage-cocoa-touch-or-cgimage-core-gr
1262893#1262893

3. Draw the image of interest into it using the command
   `CGContextDrawImage(theContext, CGRectMake(0, 0, width, height), theCGImageRef)`

4. Release the context, since we are only interested in the pointed to the data, `buffer`.

Implementing Writing Methods

To create an image file in ITK, the meta-data is written as the first few blocks of memory using the `WriteImageInformation` method, followed by writing the image pixel data using the `Write` method. However, on the iOS, the whole image writing (data and meta-data) should be implemented via the SDK interface. In particular, the `UIImage` class should be used to write the image, which in turn requires access to, not only the image pixel data, but also the meta data about the image. Therefore, ITK's `WriteImageInformation` does not implement any code; it is merely a place holder or an empty method, which returns as soon as it is called. Instead, the `Write(const void* buffer)` method is responsible for organizing all the pertinent image data (pixel data and meta-data) and saving it into the appropriate place on the iOS. The following steps highlight the image writing procedure on the iOS:

1. All image meta-data is collected. This includes the width and height of the image, the number of bits per component, the number of bits per pixel, the number of bytes per row, the color space of the image, the tropicity, and the ordering of the bits in each pixel.

2. The image is created using the command:
   ```
   CGImageRef theImageRef = CGImageCreate(width,
                                          height,
                                          bitsPerComponent,
                                          bitsPerPixel,
                                          bytesPerRow,
                                          colorSpace,
                                          bitmapInfo,
                                          theDataProvider,
                                          nil,
                                          shouldInterpolate,
                                          theIntent)
   ```
   This creates an image with all the required properties. The last three arguments in this command are not critical but will be explained briefly. `nil` indicates that the image pixel values should not be modified in anyway; they are supposed to be written as they are passed. The `shouldInterpolate` indicates whether an image should be interpolated when displayed on devices with higher resolution than the image data or not. `theIntent` indictates how colors that don't fall into the proposed color space should be handled. Further information can be found at Apple's Developer website[10].

3. A new `UIImage` is created using the `CGImage` from the previous step using the command `UIImage* outputImage = [UIImage imageWithCGImage:(theImageRef)]`.

4. The image is saved into the iOS image library using the command
   ```
   UIImageWriteToSavedPhotosAlbum (outputImage,
                                   NULL,
   ```

[10] http://developer.apple.com/library/mac/documentation/GraphicsImaging/Reference/CGImage/CGImage.pdf

```
NULL,
NULL)
```

Again, the last three arguments are not critical, but in essence they outline that no other action should be taken after saving the photo. More information about this method can be found on the Apple Developer website[11].

2.2 Examples: “Hello iOS Images”

Here, we show how we use the new `itkiOSImageIO` class to read an image, make use of existing ITK classes to filter the image, followed by, again, using the `itkiOSImageIO` class to write the resulting image to the iOS image library. In order to test the functionality and effectiveness of the newly developed `itkiOSImageIO` class, we have created two simple programs that tested different aspects of the IO class. The first set of tests simply tested that the IO class can read and write images, which were downloaded into the iOS (or could be captured by the iPhone built-in camera). The second set introduced a filter into the pipeline to test that the `itkiOSImageIO` class stores and handles the information as required by the different ITK filters. The exact filter used and how the processed image looks like is not of much importance and can be easily substituted by another. What is important is that the pipeline executes without any errors. The filter chosen for this series of tests is `itk::BinaryThresholdImageFilter` which filters (thresholds) an image into a binary image based on the intensity of each pixel. We demonstrate the our results on three types of images: a scalar medical image (a grayscale MRI image), an RGB color image (color cryosection), and finally an RGBA image exhibiting semi-transparent regions.

Medical Grayscale Image

The medical grayscale image used is a standard testing image for ITK. For the reading and writing tests, the image was read using an instance of `itkiOSImageIO` and then written to the saved photo library using the same instance, i.e. a read→write pipeline. The results are shown in Figure 1.

When performing executing the thresholding filter, i.e. a read→filter→write pipeline, the following parameters were set for the thresholding filter:

```
binaryFilter->SetLowerThreshold(0.3*255);
binaryFilter->SetUpperThreshold(0.7*255);
binaryFilter->SetOutsideValue(0);
binaryFilter->SetInsideValue(255);
```

When performing the filtering on the image, there was a need to create a second `itkiOSImageIO` instance which will deal with the writing of the processed image since its properties will be changed. The original image and the expected results are shown in Figure 2.

Medical Color Image

In order to demonstrate the effectiveness of the `itkiOSImageIO` class when dealing with color image as well, we have chosen the color image in Figure 3(a) from the public domain Visible Human Project provided by

[11] http://developer.apple.com/library/ios/documentation/uikit/reference/UIKitFunctionReference/UIKitFunctionReference.pdf

(a) The original image (b) The image copy

Figure 1: Reading and writing a grayscale image using `itkiOSImageIO`. (a) The original MRI image and (b) its written copy.

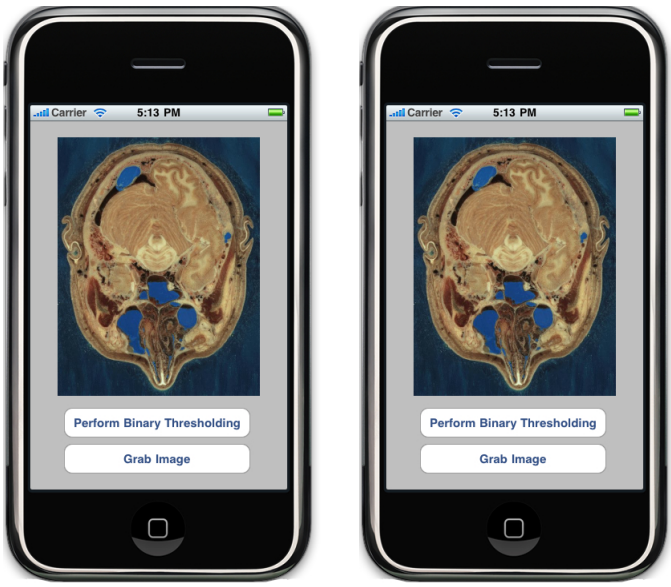
(a) The original image (b) The filtered image

Figure 2: Reading, filtering, and writing a grayscale image using `itkiOSImageIO` and `BinaryThresholdImageFilter`. (a) The grayscale image and (b) its filtered version.

the US National Library of Medicine[12]. When reading and writing the image, the code again ran error-free and the results can be seen in Figure 3(b).

[12] http://www.nlm.nih.gov/research/visible/image/head.jpg

When performing the binary filtering, another modification had to be made. Since the input image is a color image, whereas the input for the filter is a grayscale, we need to convert the color image into a grayscale image. Luckily, ITK implements `itk::RGBToLuminanceImageFilter`, which does the job. So, the pipeline in place was composed of an image reader, an RGB to grayscale image filter, a binary thresholding filter, and finally the writer, i.e. read→filter$_1$→filter$_2$→write. The reader and writer classes both had their own instances of the `itkiOSImageIO` class attributed to them. The result of the filtering pipeline can be seen in Figure 4.



(a) The original image (b) The image copy

Figure 3: Reading and writing a color image using `itkiOSImageIO` . (a) The original image and (b) its written copy.

Color Image with an Alpha Transparency

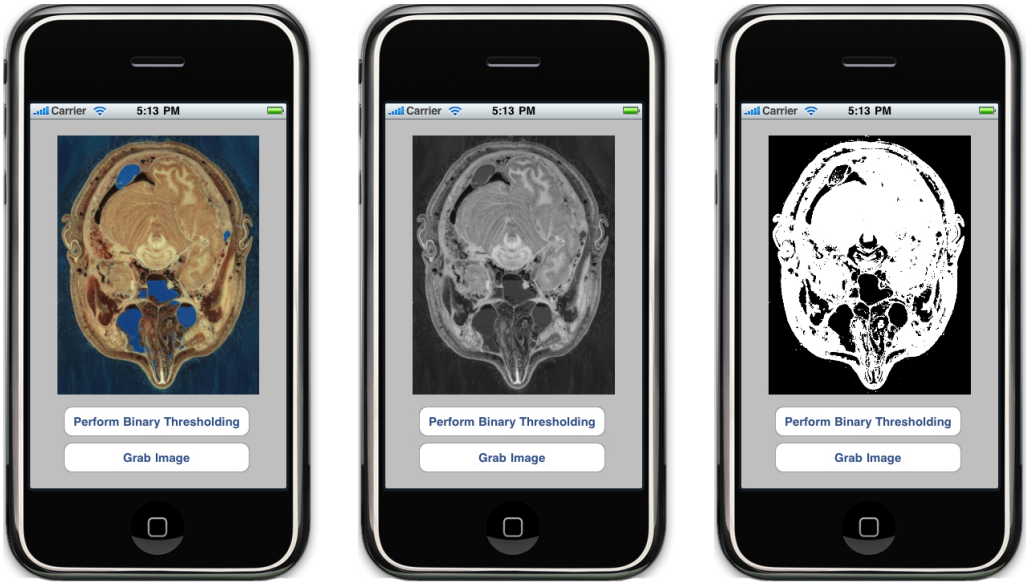
The final test set for the `itkiOSImageIO` class was using an RGBA public domain image with an alpha transparency channel[13] and can be seen in Figure 5(a). The read→write pipeline again ran successfully without any problems and the results can be seen in Figure 5(b). For the filtering test, we again needed to import the `itk::RGBToLuminanceImageFilter` class into the pipeline to transform the RGBA image into a grayscale input image. The results of the filtering pipeline, i.e. read→filter$_1$→filter$_2$→write, can be seen in Figure 6.

2.3 Putting It All Together

In order to reproduce an application which can perform similar filtering actions as described in the previous sections, the following steps are followed.

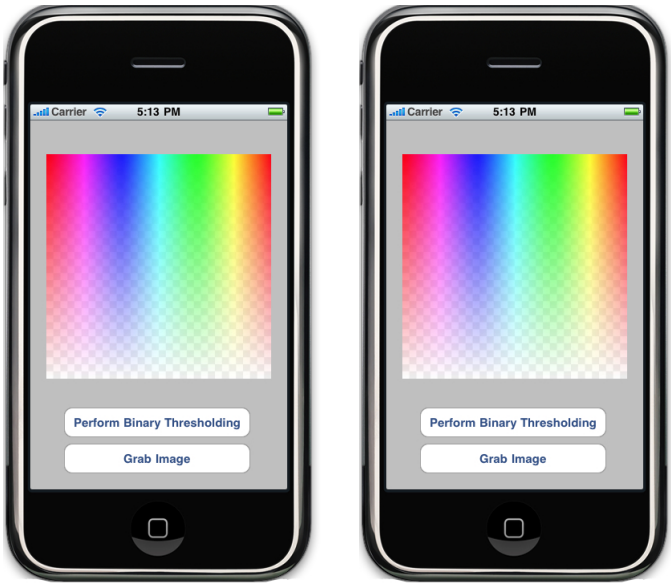
1. **Download ITK, CMake and Xcode**: Download ITK, CMake and Xcode according to the instructions outlined in[1]. Make sure to build all the ITK libraries for the device you are interested in.

[13] http://en.wikipedia.org/wiki/File:Hue_alpha.png



(a) The original image (b) The image converted into grayscale (c) The filtered image

Figure 4: Reading, filtering and writing a color image using `itkiOSImageIO` . (a) The original image, (b) its converted image and (c) its filtered image.



(a) The original image (b) The image copy

Figure 5: Reading and writing an RGBA image with an alpha component using `itkiOSImageIO` . (a) The original image and (b) its written copy.

2. **Create a new iOS app project in Xcode**: Open Xcode and select File→New Project→iOS→Application. Select the application which suits your needs and click 'Choose...' Figure 7.

(a) The original image (b) The image converted into grayscale (c) The filtered image

Figure 6: Reading, filtering and writing an RGBA image with an alpha component using `itkiOSImageIO`

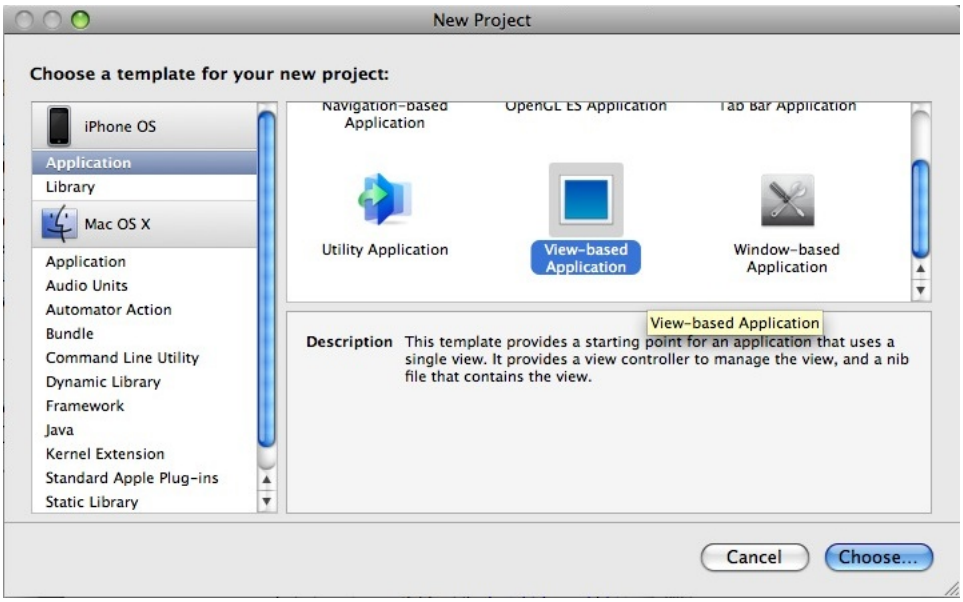


Figure 7: New project screen in Xcode used for selecting the template for the desired application.

3. **Create your app**: Fill the app content in whatever way fits your needs. You will need at least one `UIImageView` to display the images[14]. The next step will demonstrate how to integrate the image reading, processing and writing so make sure to take those components into consideration when designing your app.

[14] A good tutorial on how to build an iOS app can be found at http://developer.apple.com/library/ios/#documentation/iPhone/Conceptual/iPhone101/Articles/00_Introduction.html

4. **Add the proper configurations for the Xcode project**: The Xcode project should be compiled under the 'iPhone Device' or 'iPhone Simulator' environment depending on your preference. Also, make sure to include the right header search paths to find the .h files required for the app. For brevity purposes we strongly recommend reading our previous publication[1] in order to get a good overview on setting the right configuration in the Xcode project.

5. **Use the following code to create a simple app for reading and writing images**: This code can be used to read and write an image. Its purpose is mainly to verify functionality of the `itkiOSImageIO` class.

```
#include "itkiOSImageIO.h"
//Make sure to set the right header search path for these files
#include "ItkImage.h"
#include "itkImportImageFilter.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkRGBPixel.h"
#include "itkRGBAPixel.h"
#include "itkRescaleIntensityImageFilter.h"
#include "itkBinaryThresholdImageFilter.h"
#include "itkRGBToLuminanceImageFilter.h"


//Define all types required for the process.
//Modify this code to suit your types
typedef itk::RGBAPixel <unsigned int> RGBAPixelType;
typedef unsigned char GrayscalePixelType;
typedef itk::Image <RGBAPixelType,2> RGBAImageType;
typedef itk::Image <GrayscalePixelType,2> GrayscaleImageType;
typedef itk::ImageFileReader <RGBAImageType> RGBAReaderType;
typedef itk::ImageFileWriter <RGBAImageType> RGBAWriterType;
typedef itk::ImageFileReader <GrayscaleImageType> GrayscaleReaderType;
typedef itk::ImageFileWriter <GrayscaleImageType> GrayscaleWriterType;


//Instantiate the different objects
RGBAReaderType::Pointer RGBAReader = RGBAReaderType::New();
RGBAWriterType::Pointer RGBAReader = RGBAWriterType::New();
GrayscaleReaderType::Pointer grayReader = GrayscaleReaderType::New();
GrayscaleWriterType::Pointer grayWriter = GrayscaleWriterType::New();


//Create two image IO classes
itk::itkiOSImageIO::Pointer imageIO1 = itk::itkiOSImageIO::New();
itk::itkiOSImageIO::Pointer imageIO2 = itk::itkiOSImageIO::New();


UIImage* inputImage; //Make this image into whatever image you wish
CGImageRef theImageRef= [inputImageImage CGImage];


//Need to obtain the image color space to know how to read it
```

```
CGColorSpaceRef theColourSpace = CGImageGetColorSpace(theImageRef);
size_t numColourSpaceComponents = CGColorSpaceGetNumberOfComponents(theColourSpace);

if (numColourSpaceComponents == 1)//Grayscale image
{
      //Manually set the correct imageIO class
      grayReader->SetImageIO(imageIO1);
      imageIO1->SetFileName(inputImage);

      //This function call is important since the method SetFileName in the
      //reader class must be called for it to run
      grayReader->SetFileName("UIImage");

      grayReader->Update();
      grayWriter->SetInput(grayReader->GetOutput());
      grayWriter->SetImageIO(imageIO2);

      UIImage* outputImage;

      imageIO2->SetFileName(outputImage);
      grayWriter->SetFileName("UIImage");

      grayWriter->Update();
}//end if (numColourSpaceComponents == 1)
else if (numColourSpaceComponents == 3)
// If image has 3 components, it may or may not have an alpha channel
{
      RGBAReader->SetImageIO(imageIO1);
      imageIO1->SetFileName(image.image);
      RGBAReader->SetFileName("UIImage");
      RGBAReader->Update();

      RGBAWriter->SetInput(RGBAReader->GetOutput());
      RGBAWriter->SetImageIO(imageIO2);

      UIImage* outputImage;

      imageIO2->SetFileName(outputImage);
      RGBAWriter->SetFileName("UIImage");

      RGBAWriter->Update();
}//end else if (numColourSpaceComponents == 3)
```

6. **Use the following code to create a more advanced app for reading, filtering and writing**: The following code creates more advanced apps which perform the binary filtering discussed before. You

can use as many filters as you like, but make sure that the input and output types between them are compatible.

```
#include "itkiOSImageIO.h"
//Make sure to set the right header search path for these files
#include "ItkImage.h"
#include "itkImportImageFilter.h"
#include "itkImageFileReader.h"
#include "itkImageFileWriter.h"
#include "itkRGBPixel.h"
#include "itkRGBAPixel.h"
#include "itkRescaleIntensityImageFilter.h"
#include "itkBinaryThresholdImageFilter.h"
#include "itkRGBToLuminanceImageFilter.h"

//Define all the types required for the process.
//Modify this code to suit your types
typedef itk::RGBAPixel <unsigned int> RGBAPixelType;
typedef unsigned char GrayscalePixelType;
typedef itk::Image <RGBAPixelType,2> RGBAImageType;
typedef itk::Image <GrayscalePixelType, 2 > GrayscaleImageType;
typedef itk::ImageFileReader <RGBAImageType> RGBAReaderType;
typedef itk::ImageFileReader <GrayscaleImageType> GrayscaleReaderType;
typedef itk::ImageFileWriter <GrayscaleImageType> GrayscaleWriterType;
typedef itk::RGBToLuminanceImageFilter<RGBAImageType,GrayscaleImageType>
RGBAtoGrayscaleFilterType;
typedef itk::BinaryThresholdImageFilter <GrayscaleImageType,
GrayscaleImageType> BinaryThresholdFilterType;

//Instantiate the different objects
RGBAReaderType::Pointer RGBAReader = RGBAReaderType::New();
RGBAWriterType::Pointer RGBAReader = RGBAWriterType::New();
GrayscaleReaderType::Pointer grayReader = GrayscaleReaderType::New();
GrayscaleWriterType::Pointer grayWriter = GrayscaleWriterType::New();
RGBAtoGrayscaleFilterType::Pointer RGBA2GrayFilter = RGBAtoGrayscaleFilterType::New();
BinaryThresholdFilterType::Pointer binaryFilter = BinaryThresholdFilterType::New();

//Create two image IO classes
itk::itkiOSImageIO::Pointer imageIO1 = itk::itkiOSImageIO::New();
itk::itkiOSImageIO::Pointer imageIO2 = itk::itkiOSImageIO::New();

UIImage* inputImage; //Make this image into whatever image you wish
CGImageRef theImageRef= [inputImageImage CGImage];

//Need to obtain the image color space to know how to read it
CGColorSpaceRef theColourSpace = CGImageGetColorSpace(theImageRef);
size_t numColourSpaceComponents = CGColorSpaceGetNumberOfComponents(theColourSpace);
```

```
if (numColourSpaceComponents == 1)//Grayscale image
{
      //Manually set the correct imageIO class
      grayReader->SetImageIO(imageIO1);
      imageIO1->SetFileName(inputImage);

      //This function call is important since the method SetFileName in the
      //reader class must be called for it to run
      grayReader->SetFileName("UIImage");

      grayReader->Update();

      binaryFilter->SetInput(grayReader->GetOutput());
      binaryFilter->SetOutsideValue(0);
      binaryFilter->SetInsideValue(255);
      binaryFilter->SetLowerThreshold(0.3*255);
      binaryFilter->SetUpperThreshold(0.7*255);

      binaryFilter->Update();

      grayWriter->SetInput(grayReader->GetOutput());
      grayWriter->SetImageIO(imageIO2);

      UIImage* outputImage;

      imageIO2->SetFileName(outputImage);
      grayWriter->SetFileName("UIImage");

      grayWriter->Update();
}//end if (numColourSpaceComponents == 1)
else if (numColourSpaceComponents == 3)
// If image has 3 components, it may or may not have an alpha channel
{
      RGBAReader->SetImageIO(imageIO1);
      imageIO1->SetFileName(image.image);
      RGBAReader->SetFileName("UIImage");
      RGBAReader->Update();

      RGBA2GrayFilter->SetInput(RGBAReader->GetOutput());
      RGBA2GrayFilter->Update();

      binaryFilter->SetInput(RGBA2GrayFilter->GetOutput());
```

```
        binaryFilter->SetOutsideValue(0);
        binaryFilter->SetInsideValue(255);

        binaryFilter->SetLowerThreshold(0.3*255);
        binaryFilter->SetUpperThreshold(0.7*255);

        binaryFilter->Update();

        grayWriter->SetInput(binaryFilter->GetOutput());
        grayWriter->SetImageIO(imageIO2);

        UIImage* outputImage;

        imageIO2->SetFileName(outputImage);
        grayWriter->SetFileName("UIImage");

        grayWriter->Update();
    }//end else if (numColourSpaceComponents == 3)
```

3 Conclusions and Future Work

We have developed `itkiOSImageIO` , the necessary ITK class to interface with images on iOS devices. This work has been motivated by the importance of medical image analysis for health applications, which is increasingly being performed using the ITK library, and the ubiquitous mobile devices, in particular Apple's iOS devices (iPod touch, iPhone, and iPad) with faster processing, larger storage, and exquisite multi-touch displays. The sample code and the `itkiOSImageIO` class provided here should facilitate the development of more sophisticated medical image analysis applications on iOS devices, as well as other applications that may rely on non-medical image data

We foresee the following future extensions of our work. The first is facilitating reading, writing, and processing of 3D or higher dimensional (not only 2D) medical images using ITK on iOS devices. However, the iOS `UIImage` object does not natively support 3D images; it only supports the following 2D formats: TIFF, JPEG, GIF, PNG, DIB, .ico, .cur and .xbm[15]. Nevertheless, there exists several iOS apps that work with and visualize 3D medical images, e.g. ImageVis3D Mobile[16] and OsiriX for the iPhone[17]. These 3D images are handled by the app itself and not via the iOS photo library. It remains to be seen what is the most suitable approach for working with high dimensional images using ITK on the iOS. One option is to read a series of 2D images to compose a 3D image in memory by using the `itk::ImageSeriesReader`, just the same way that it is commonly done for composing a 3D dataset from a set of DICOM slices.

The second foreseeable development is supporting C++ libraries other than ITK, mainly the Visualization ToolKit (VTK), ITK's visualization homologue. In the application presented here, we simply used the `UIImageView` object to display the images before and after processing. VTK, however, offers much greater

[15] http://developer.apple.com/iPhone/library/documentation/UIKit/Reference/UIImageClass/Reference/Reference.html

[16] http://en.wikipedia.org/wiki/ImageVis3D_Mobile

[17] http://www.osirix-viewer.com/iphone/

image visualization capabilities, and works especially well with images and spatial objects produced by ITK.

Another direction for further development is related to ITK's pluggable object factories. ITK handles reading and writing of images using pluggable object factories[18]. This factory mechanism allows the ITK `ImageFileReader` and `ImageFileWriter` functions to determine at run-time the file format (typically based on the file extensions) and invoke the proper image IO code accordingly. Implementing this mechanism for the iOS is left for future work.

Finally, for medical images in particular, the physical dimensions of each pixel are quite important (e.g. the pixel resolution measured in mm/pixel along the horizontal direction and similarly along the vertical direction). Unfortunately, when reading an image, iOS assumes only isotropic pixels without any physical dimensions assigned to it. Nevertheless, the programmer may set the pixel size of the image using `itkiOSImageIOInstance->SetSpacing(0,xSpacing)` for the horizontal pixel spacing and `itkiOSImageIOInstance->SetSpacing(1,ySpacing)` for the vertical spacing. Further exploration of working with physical units of pixel resolutions, image offsets, direction cosines, etc., on the iOS, remains an important future goal.

References

[1] B. Shabash, G. Hamarneh, Z. F. Huang, and L. Ibanez, "ITK on the iOS," *Insight Journal*, vol. July-December, pp. 1–9, 2010. (document), 1, 1, 4

[2] G. D. Abowd and E. D. Mynatt, "Charting past, present, and future research in ubiquitous computing," *ACM Trans. Comput.-Hum. Interact.*, vol. 7, no. 1, pp. 29–58, 2000. 1

[3] D. Taniar, *Mobile Computing: Concepts, Methodologies, Tools, and Applications*. Information Science Reference, 2008. 1

[4] R. Istepanian, S. Laxminarayan, and C. S. Pattichis, Eds., *M-Health: Emerging Mobile Health Systems (Topics in Biomedical Engineering. International Book Series)*. Springer, 2005. 1

[5] J. M. Fitzpatrick and M. Sonka, Eds., *Handbook of Medical Imaging, Volume 2. Medical Image Processing and Analysis*. SPIE Publications, 2009. 1

[6] T. Too, Ed., *Insight into Images: Principles and Practice for Segmentation, Registration, and Image Analysis*. AK Peters, 2004. 1

[7] V. Chu and G. Hamarneh, "MATITK: Extending MATLAB with ITK," *Insight Journal*, vol. Aug-Dec, pp. 1–8, 2005. 1

[8] M. Malaterre, "Zeiss (LSM) file format support," *Insight Journal*, vol. August-December, 2005. 1

[9] K. Mosaliganti, L. Ibanez, and S. Megason, "Support for Streaming the JPEG2000 File Format," *Insight Journal*, vol. July-December, pp. 1–3, 2010. 1

[10] L. Baghdadi, "MINC2.0 IO Support for ITK," *Insight Journal*, vol. August-December, pp. 1–2, 2005. 1

[18] Refer to Chapter 7: Reading and Writing Images, in the ITK Software guide, available from http://www.itk.org/ItkSoftwareGuide.pdf)