TRIPOD – Computer Vision for Classroom Instruction and Robot Design

Paul Y. Oh
Drexel University, Philadelphia PA, Email: paul@coe.drexel.edu

Summary

TRIPOD is free and open source software for rapidly developing computer vision applications. With such software, robot vision systems can be developed, tested and validated. This paper presents a step-by-step tutorial where live video is thresheld to display the corresponding binary image. This simple exercise serves to illustrate the realization of a robot vision system. A LEGO Mindstorms Vision Command module, which is a Logitech USB camera, is used. Platform development is in ANSI C/C++ on a Windows PC without low-level Windows programming.

Keywords: computer vision, robotics, USB cameras, LEGO, real-time, image processing

1 Introduction

In recent years, the emergence of hardware to rapidly prototype robots and software to test algorithms have made an impact on robotics education. LEGOs, programmable embedded microcontrollers and MATLAB are some examples of tools which students can practice real-world hands-on robot design. Lacking however, are non-proprietary, affordable, easy-to-use and customizable software tools to design robot vision systems. While affordable hardware like USB webcams, firewire cards and lens-on-a-chip boards are ubiquitous, software tools to visually servo robots are not widely available. Although computer vision packages are commercially available, they are often too expensive or have too steep a learning curve to implement effectively in the classroom. As such, despite its importance and significance, students rarely get hands on experience in designing vision systems in robotics and artificial intelligence courses. While there are computer vision courses, often only static image files are used. Robot vision systems often demand visual-servoing where live video is processed in a closed-loop feedback loop that is designed to handle processing delays and robot dynamics [4].

This paper describes a computer vision software package called TRIPOD: **T**emplate for **R**eal-time **I**mage **PrO**cessing **D**evelopment. A screenshot is provided in Figure 1. It was designed for classroom implementation and addresses the need for a flexible platform to rapidly design robot vision systems. To reach the widest possible audience, TRIPOD was designed to use affordable USB Logitech cameras [1] and is programmable in ANSI C/C++. TRIPOD runs on Windows-based PCs with the Microsoft Visual C++ 6.0 compiler since most students can easily find such platforms on campus or configure ones at home. TRIPOD is free open source software and does not require low-level Windows programming knowledge. The software provides a pointer to the frame's pixel data to enable one to focus on image processing. For example, textbook computer vision algorithms [2] like Sobel edge filters and sum-of-square difference trackers have been easily implemented in ANSI C on TRIPOD. To illustrate TRIPOD as a potential classroom tool for robot vision instruction, a tutorial of binarizing live video is given in this paper. Section 2 provides the coding objective while Sections 3 and 4 present

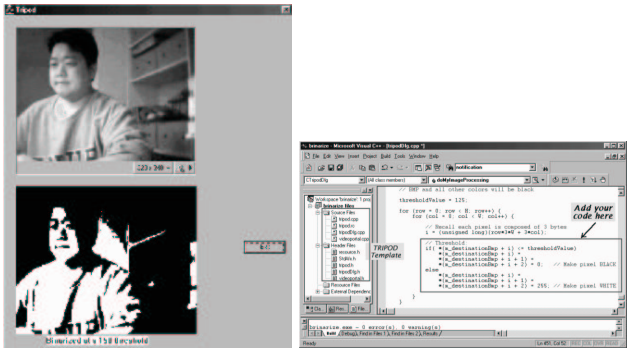

Figure 1: TRIPOD's Windows' interface. Top and bottom viewports respectively display the live camera field-of-view and processing results (left). Development just involves inserting ANSI C/C++ algorithms in the template, indicated by the boxed areas (right).

[1] For example, the Logitech Quickcam or LEGO Vision Command Camera, are color cameras that cost less than 50 US dollars.

step-by-step instructions and code commentary. Section 5 concludes with map of future work.

2 Coding Objective

In computer vision, a "Hello World" type program would be the generation of a binary image. Here, pixels below a pre-defined threshold are made black while the remaining pixels are set white as shown in the bottom viewport of Figure 1. Such binarization is often performed as a pre-processing step for algorithms like edge detection and tracking. Pseudocode for example:

```
for(row = 0; row < H; row++) {
   for(col = 0; col < W; col++) {
      if(grayImage[row, col] < threshold)
         /* make pixel black */
         binaryImage[row, col] = 0;
      else
         /* make pixel white */
         binaryImage[row, col] = 255;
   }
}
```

where an 8-bit grayscale image, `grayImage` with `(W×H)` pixels is read. Here, `W` is the width (number of columns), `H` is height (number of rows) and `threshold` is a pre-defined value ranging from 0 to 255 used to generate the binarized image `binaryImage`.

Given the popularity of LEGO Mindstorms in robotics education and experimentation, TRIPOD was designed to work with the LEGO Vision Command camera. This Mindstorms product is actually a 24-bit color Logitech USB camera but its accompanying image processing software is very limited. To overcome this, TRIPOD interfaces into Logitech's free Quickcam software developers kit (QCSDK) [2]. TRIPOD thus permits one to concentrate on computer vision algorithm in ANSI C/C++ while avoiding low level Windows programming like DirectX.

Step-by-step instructions for generating a binarized view of live video, as shown in Figure 1, follows. This was tested on a Pentium III 500 MHz, 128 MB RAM running Windows 98 and LEGO Mindstorms Vision Command camera. Microsoft Visual C++ 6.0 is used and only minimal MFC knowledge is assumed.

[2] http://developer.logitech.com/sdk

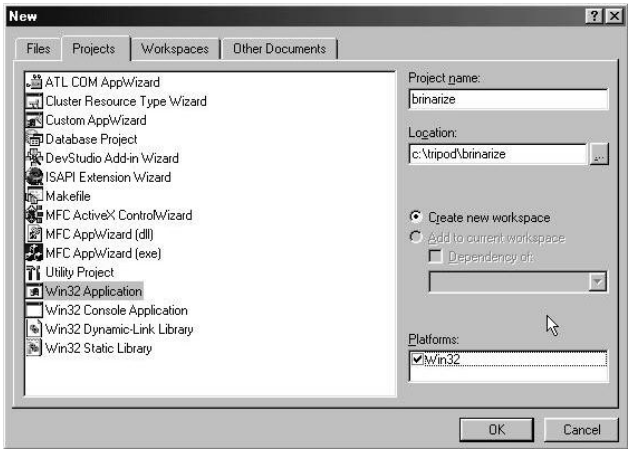

Figure 2: Visual C++ 6.0 screenshot when creating a Win32 application

3 Step-by-Step Instructions

TRIPOD will be used to create a program with two viewports. The top viewport will display live video captured by the camera while the bottom viewport displays a binarized version.

STEP 1: Create Win32 Application

From the menubar choose *File-New* and select *Win32 Application*. For the *Location*, choose `C:\tripod` and type `brinarize` for the Project name. Note: the spelling has an "r" in `brinarize.exe`). The Win32 check box should be checked. When your screen looks like Figure 2, click the OK button.

STEP 2: Create MFC Project

After clicking OK above, Choose *Empty Project* when prompted by the popup box for a project type. When the *Finish* button is clicked VC++ automatically creates the project's structure and makefiles. Click the *FileView* tab (near the screen's bottom) and the Source, Header and Resource folders can be seen. From the menubar choose *File-Save All* and then *Build-Rebuild All*. There should be no compile errors since these folders are currently empty.

STEP 3: Applications folder

Using Windows Explorer, copy the TRIPOD template files from the `C:\tripod` folder to the application project folder, for example, `C:\tripod\brinarize`

`StdAfx.h, resource.h, tripod.cpp,`

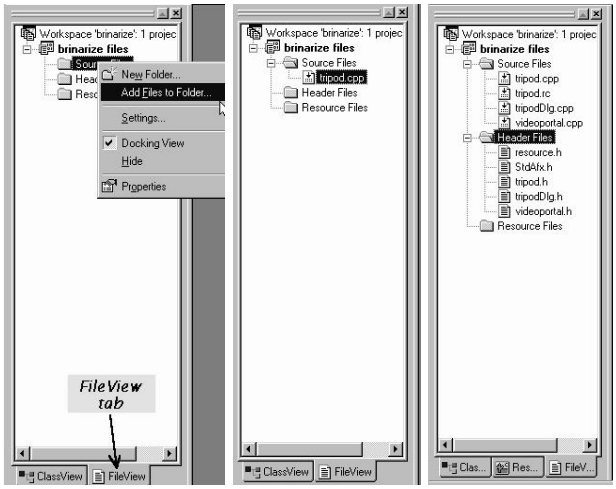

Figure 3: Clicking *FileView* enables all files in the Source, Header and Resource folders to be seen.

```
tripod.h, tripod.rc, tripodDlg.cpp,
tripodDlg.h, videoportal.h,
videoportal.cpp.
```

The *res* folder must be copied as well. In VC++, choose *FILE-Save All*.

STEP 4: Include TRIPOD files

In VC++, click the *FileView* tab and expand `brinarize files` to see folders named Source Files, Headers Files and Resources. Click the *Source Files* folder once and then right click and choose *Add Files*. Figure 3 should result.

Browse to `C:\tripod\brinarize` and add `tripod.cpp`. Expand the *Source Files* folder and one should see `tripod.cpp` listed as shown in middle image Figure 3. Repeat the above, adding the following files to the *Source Files* folder:

```
tripod.rc, tripodDlg.cpp, videoportal.cpp
```

Next, add the following files to the *Header Files* folder:

```
StdAfx.h, tripod.h, resource.h,
tripodDlg.h, videoportal.h
```

Once all these files have been added, the workspace tree should look like the left image in Figure 3.

STEP 5: Include QCSDK and MFC Shared DLLs

QCSDK include files need to be added to the project. From the menubar click *Project-Settings*. Next, click on the root directory `brinarize` and then the *C/C++* tab. Under the *Category* combo pulldown box choose *Preprocessor*. In the *Additional Include Directories* edit box, type `\QCSDK1\inc`. This results in Figure 4 (left).

Next, click the *General* tab and under the *Microsoft Foundations Class* pulldown menu, choose *Use MFC in a shared DLL* as shown in the Figure 4 (right).

Finish off by clicking OK. Next, save all work by clicking *File-Save All*. Next compile the project by choosing *Build-Rebuild All*.

STEP 6: Add Image Processing Code

The TRIPOD source, header and resource files used in the previous steps grab the color image frame, converts the red, green and blue pixels into a grayscale value, and stores the frame pixels into a malloc'ed row-column vector. All that remains is to add image processing routines. The added code (see Appendix Section 6) goes in the `tripodDlg.cpp` file, under the `CTripodDlg::doMyImageProcessing` function.

STEP 7: Save, Compile and Execute

Once image processing algorithms have been implemented choose *File-Save All* and compile by choosing *Build-Rebuild All*. Upon successful compile, choose *Build-Execute brinarize.exe*. The application should launch, successfully thresholding and displaying real-time binarized images as was shown in Figure 1.

4 Code Commentary

TRIPOD files and classes are structured so that image processing algorithms can be written in ANSI C/C++ and inserted in `CTripodDlg::doMyImageProcessing` (copy of which is in Appendix Section 6). This is possible by providing pointers to pixel data arranged in row-column vector format that is refreshed at frame rate.

4.1 Destination and Source Bitmaps

The variables `m_destinationBmp` and `sourceBmp` relate to pixel data as follows. ANSI C/C++ programmers will recognize that in `doMyImageProcessing`, the code nested between the two `for` loops is ANSI C. `m_destinationBmp` is a pointer to an array of pixels and `*(m_destinationBmp + i)` is the value of the

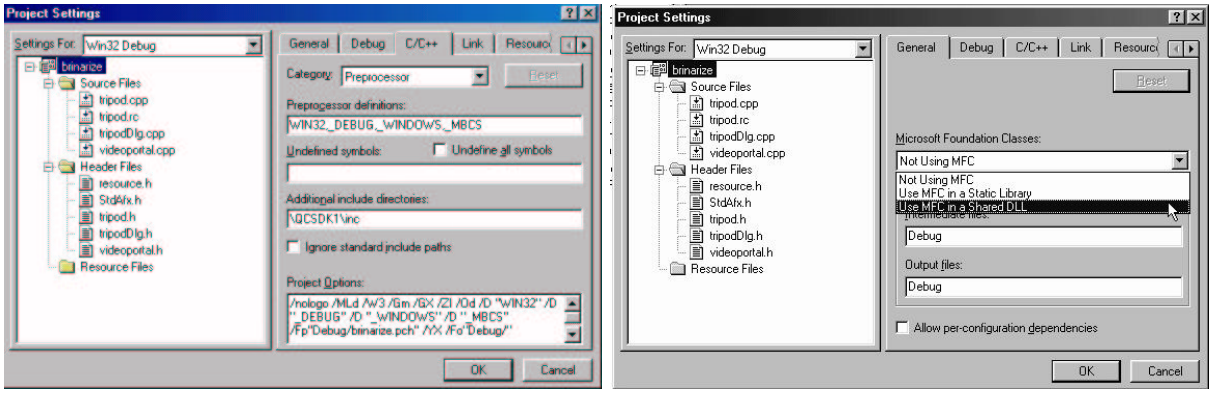

Figure 4: Screenshot after including QCSDK and MFC Shared DLLs

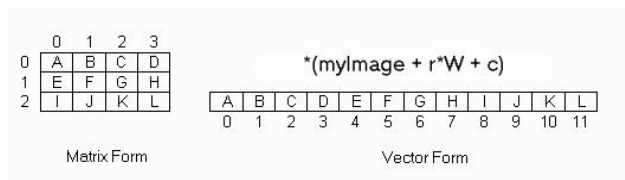

Figure 5: Image data represented as a matrix (left) and row-column vector right

`i`'th pixel. The two `for` loops read, process and write every pixel in the image. After cycling through the array, a final `m_destinationBmp` results and can be displayed. `doMyImageProcessing` and displaying `m_destinationBmp` runs in real-time (30 frames/sec) if the nested code is not computationally intensive, like simple threshold or centroid calculations.

`m_destinationBmp` points to a 24-bit grayscale bitmap. It is 320 pixels wide by 240 pixels high. It is malloc'ed and created in the function `grayScaleTheFrameData`. In this function, `sourceBmp` points to the actual pixel data in the 24-bit RGB color image captured by the camera. Being RGB, each pixel in `sourceBmp` is represented by three bytes (red, green and blue).

The reason for creating `m_destinationBmp` is that often, computer vision developers use grayscale images to reduce computation cost. If you need color data, then just use `sourceBmp`.

4.2 Row-Column Vector Format

An image is an arranged set of pixels. A 2-dimensional array like `myImage[r,c]` where `r` and `c` are the pixel's row and column positions respectively, is an intuitive arrangement as illustrated in Figure 5 (left). For example, `myImage` is a (3×4) image having three rows and four columns. `myImage[2,1]`, which refers to the pixel at row 2 column 1, has a pixel intensity value J.

An alternative arrangement, often encountered in computer vision, is the *row-column* format which uses a 1-dimensional vector and shown in Figure 5 (right). A particular pixel is referenced by:

```
(myImage + r*W + c)
```

where `myImage` is the starting address of the pixels, `r` and `c` are the pixel's row and column positions respectively , and `W` is the total number of columns in the image (width in pixels). To access the pixel's value, one uses the ANSI C de-referencing operator:

```
*(myImage + r*W + c)
```

For example for `r`=2, `c`=1 and `W`=4, then `(myImage + r*C + c)` yields `(myImage + 9)`. In vector form `myImage[9]`, which is the same as `*(myImage + 9)`, has the pixel value J.

The row-column format has several advantages over 2D arrays. First, memory for an array must be allocated before run-time. This forces a programmer to size an array according to the largest possible image the program might encounter. As such, small images requiring smaller arrays would lead to wasted memory. Furthermore, passing an array between functions forces copying it on the stack which again wastes memory and takes time. Pointers are more computationally efficient and memory can be `malloc`'ed at run-time. Second, once image pixels are arranged in

row-column format, you can access a particular pixel with a single variable, as well as take advantage of pointer arithmetic like `*(pointToImage++)`. Arrays take two variables and do not have similar arithmetic operators. For these two reasons row-column formats are used in computer vision, especially when more computationally intensive and time-consuming image processing is involved.

4.3 24-bit Bitmap Images

A 24-bit image uses three bytes to specify a single pixel. Often these bytes are the pixel's red, green and blue (RGB) contributions. RGB is also known as the Truecolor format since 16 million different colors are possible with 24-bits. As mentioned above, `m_destinationBmp` and `sourceBmp` are 24-bit grayscale and Truecolor images respectively. `m_destinationBmp` makes all three bytes of a single pixel equal in intensity value. The intensity is a gray value computed from the amount of red, green and blue in the pixel. As such `*(m_destinationBmp + i)`, `*(m_destinationBmp + i + 1)`, and `*(m_destinationBmp + i + 2)` are made equal (see the function `grayScaleTheFrameData` for details). Referring to Appendix Section 6, thresholding sets these three bytes to either black or white.

Bitmaps, the default image format of the Windows operating system, can be saved to a disk file and typically have a .BMP filename extension. Bitmaps can also exist in memory and be loaded, reloaded, displayed and resized. There are two caveats to using bitmaps. First, pixels are stored from left-to-right but bottom-to-top; when a bitmap is viewed, pixels towards the bottom are stored closer to the image's starting address. Second, a pixel's color components are stored in reverse order; the first, second and third bytes are the amounts of blue, green and red consecutively. Again, the `grayScaleTheFrameData` function can be referenced to see this reverse-ordering of color.

4.4 Code Operation

The flowchart in Figure 6 shows `brinarize.exe`'s function calling sequence. A Window's application begins with a call to `OnInitDialog`. Code here initializes the sizes for the two videoportals. A call to `allocateDib` allocates memory to display both the image captured by the camera and the image resulting from `doMyImageProcessing`, like binarizing.

The Logitech SDK defines a variable flag called `NOTIFICATIONMSG_VIDEOHOOK` and goes

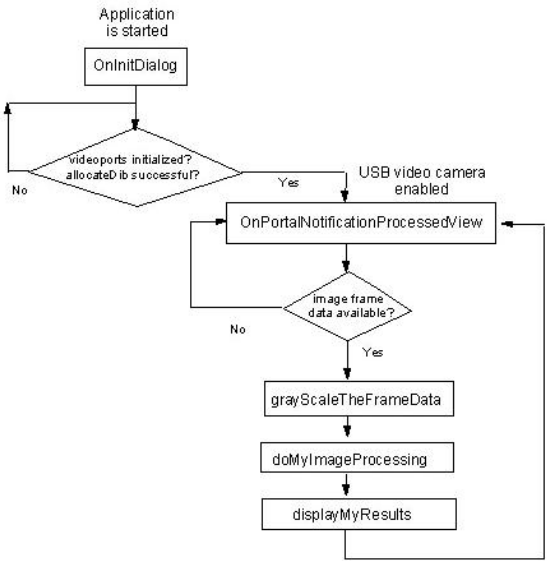

Figure 6: The loop in the flowchart above executes `doMyImageProcessing` on every frame.

true whenever the camera acquires a new image frame. After `OnInitDialog`, the code in `OnPortalNotificationProcessedview` checks for this flag and executes. Code here then assigns the pointer `lpBitmapPixelData` to the frame's pixel data, grayscales the color image, executes any computer vision algorithm stored in `doMyImageProcessing`. The image processing results are then displayed through `displayMyResults` which uses the MFC function `StretchDIBits` to stretch a device-independent bitmap image to fit the videoportal's display window. If `doMyImageProcessing` is not computationally time-consuming, `OnPortalNotificationProcessedview` will execute at 30 frames/sec.

5 Conclusions

This paper presented TRIPOD, a free and open source software, for developing computer vision applications. Creating TRIPOD was motivated by two reasons. First, there is a dire need for non-proprietary, customizable and affordable software tools to rapidly design robot vision systems. Second, such tools would serve well in the classroom by providing hands-on experience in real-world robot vision design. As such, TRIPOD was designed to work with USB cameras like the LEGO Vision Command. A detailed step-by-step tutorial was presented to illustrate the development process. Beyond binarized images, algorithms like de-

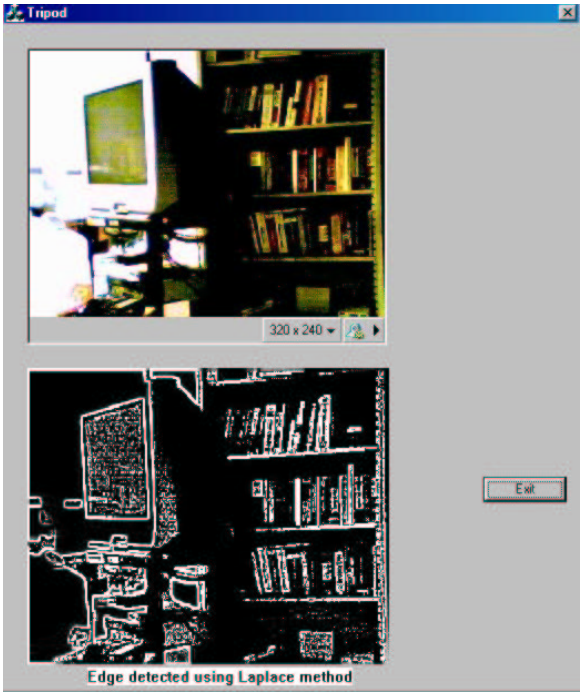

Figure 7: TRIPOD Edge detection application.

tecting edges (see Figure 7 and colors, tracking regions and counting objects have also been tested and implemented on TRIPOD [3] [3]. Beta versions of TRIPOD have been tested by several university robot labs including Drexel, Columbia, Harvard, Texas A&M, Brown, USC, CMU and Rutgers, with positive feedback.

Indeed there are other computer vision packages for Windows. The Intel OpenCV [1] and Microsoft's Vision SDK are powerful packages but have steep learning curves. Commercially available packages also exist but often are proprietary and involve run-time licenses. As such, they do not lend themselves to classroom implementation. Matlab's Image Processing Toolbox, while excellent for computer vision courses, works only with static images and thus ill-suited for teaching visual-servoing design. As such, TRIPOD reaches a wider audience; USB cameras and ANSI C/C++ programming give students affordable hands-on instruction on computer vision and visual-servoing development. The author is currently developing a Linux version called TRIPODx that will leverage the Bt8x8 chipset.

[3]Such algorithms and TRIPOD are downloadable from the author at http://www.boondog.com

References

[1] Bradski, G., "The OpenCV Library – An open-source library for processing image data," *Dr. Dobb's Journal*, pp. 120-125, November 2000

[2] Myler, H.R., The Pocket Handbook of Image Processing Algorithms in C, Prentice Hall, 1993.

[3] Oh, P.Y., Green, W.E., "A Kite and Teleoperated Vision System for Acquiring Aerial Images", *IEEE Int Conf Robotics and Automation (ICRA)*, pp. 1404-09, Taiwan, September 2003.

[4] Oh, P.Y., Allen, P.K., "Visual Servoing by Partitioning Degrees of Freedom," *IEEE Transactions on Robotics and Automation*, V17, N1, pp. 1-17, February 2001

6 Appendix

Source code for the `doMyImageProcessing` function. Pixel processing, thresholding for example, occurs in the the nested `for` loops.

```
void CTripodDlg::doMyImageProcessing(LPBITMAPINFOHEADER lpThisBitmapInfoHeader)
{
 // doMyImageProcessing:  This is where you'd write your own image processing code
 // Task: Read a pixel's grayscale value and process accordingly

 unsigned int    W, H;    // Width and Height of current frame [pixels]
 unsigned int    row, col; // Pixel's row and col positions
 unsigned long   i;    // Dummy variable for row-column vector

 BYTE thresholdValue;  // Value to threshold grayvalue

 char str[80]; // To print message
 CDC  *pDC;    // Device context need to print message

 W = lpThisBitmapInfoHeader->biWidth;  // biWidth: number of columns
 H = lpThisBitmapInfoHeader->biHeight; // biHeight: number of rows

 // In this example, the grayscale image (stored in m_destinationBmp) is
 // thresholded to create a binary image.  A threshold value close to 255
 // means that only colors close to white will remain white in binarized
 // BMP and all other colors will be black

 thresholdValue = 150;

 for (row = 0; row < H; row++) {
   for (col = 0; col < W; col++) {

   // Recall each pixel is composed of 3 bytes
   i = (unsigned long)(row*3*W + 3*col);

   // Add your code to operate on each pixel. For example
   // *(m_destinationBmp + i) refers to the ith pixel in the destinationBmp
   // Since destinationBmp is a 24-bit grayscale image, you must also apply
   // the same operation to *((m_destinationBmp + i + 1) and
   // *((m_destinationBmp + i + 2).

   //  Threshold: if a pixel's grayValue is less than thresholdValue
   if( *(m_destinationBmp + i) <= thresholdValue)
      *(m_destinationBmp + i) =
      *(m_destinationBmp + i + 1) =
      *(m_destinationBmp + i + 2) = 0; // Make pixel BLACK
   else
      *(m_destinationBmp + i) =
      *(m_destinationBmp + i + 1) =
      *(m_destinationBmp + i + 2) = 255; // Make pixel WHITE
 }
}

 // To print message at (row, column) = (75, 580). Comment if not needed
 pDC = GetDC();
 sprintf(str, "Binarized at a %d threshold", thresholdValue);
 pDC->TextOut(75, 580, str);
 ReleaseDC(pDC);
```