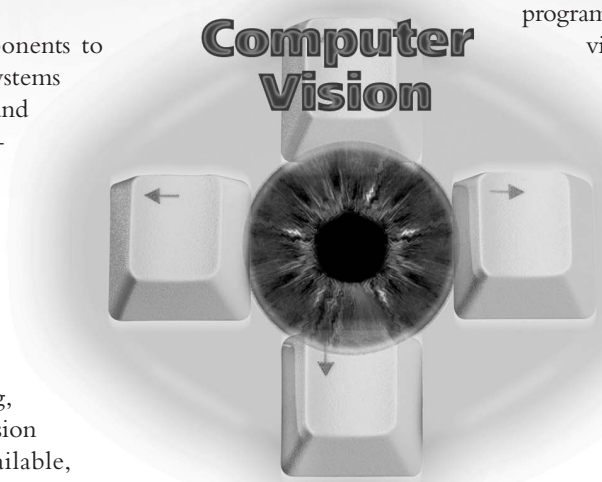
A Template for Real-Time Image Processing Development

Windows-Based PCs and Webcams for Computer Vision Research and Education

BY PAUL Y. OH

Construction kits, like LEGO, and embedded micros can be found in many hands-on robotics curricula today. Such components enable students to exercise fundamental concepts like system integration, mechanism synthesis, and real-time programming. When affordable, widely available, and portable, such components promote learning because students have easier access to hardware. This motivates students and advances the study of robotics [3].

In recent years, the components to construct computer vision systems have also become affordable and widely available. Video hardware like USB Webcams, firewire cards, and television framegrabbers can have a positive impact on computer vision instruction and research. Such components could facilitate lessons on image processing, visual servoing, and sensor fusion [5]. Lacking are widely available, nonproprietary, affordable, easy-to-use, and customizable software packages. Although computer vision packages like OpenCV [1] or toolkits like the MATLAB Image Acquisition Toolbox are available, they often have steep learning curves or are somewhat expensive for students. Indeed, there are excellent Linux-based software for computer vision. Unfortunately, students not majoring in computer science or computer engineering are often more familiar with Windows software and have easier access to Windows-based computers.



IRIS: ©1998 CORBIS CORPORATION, COMPUTER KEYS © DIGITAL VISION

This article describes the computer vision software package called TRIPOD: Template for Real-time Image Processing Development (available at: http://www.mem.drexel.edu/pauloh.html), and Figure 1 is a screenshot. It was designed for classroom implementation or self-instruction. To reach a wide audience, affordable USB Logitech cameras (The Logitech Quickcam and LEGO Vision Command Camera are color cameras costing less than US$50), Windows PC, Microsoft Visual C++ 6.0 compiler, and ANSI C/C++ are used. TRIPOD 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 [4], [6] like Sobel edge filters and sum-of-square difference trackers have been easily implemented in ANSI C on TRIPOD.

This article provides a tutorial to binarize live video and demonstrate TRIPOD's capabilities. A live 20-min demonstration of this tutorial was presented in the 2004 AAAI Spring Symposium on Hands-on Robotics Education [3]. Over 50 TRIPOD CDs were distributed to symposium participants, and comments were positive. TRIPOD was also used to instruct computer vision in high school through a National Science Foundation-sponsored research experience for teachers grant (NSF EEC Grant 0227700 Research Experience for Teachers in Areas of Innovative and Novel Technologies in Philadelphia).

The following section provides the coding objective and then step-by-step instructions and code commentary. To

1070-9932/05/$20.00©2005 IEEE

Figure 1. *TRIPOD's Windows interface. Top and bottom viewports display the live camera field-of-view and processing results, respectively.*

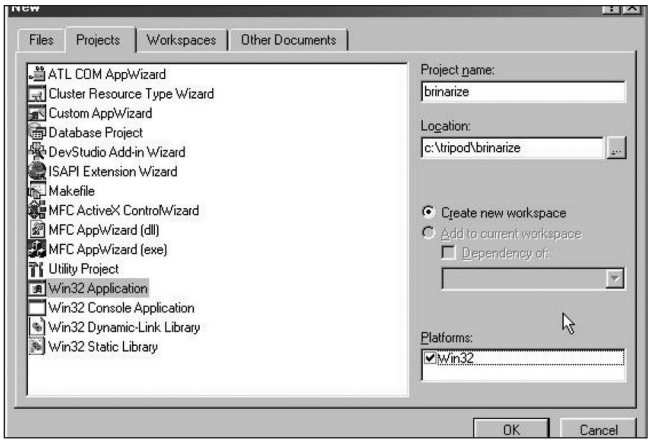


Figure 2. *Visual C++ 6.0 screenshot when creating a Win32 application.*

conclude, applications developed with TRIPOD, like visual-servoing a robotic blimp and laser rangefinding, are briefly presented.

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 preprocessing step for algorithms like edge detection and tracking. For example, pseudocode to binarize an image can be the following:

```
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-b grayscale image, `grayImage` is an 8-b, (`W` × `H`) grayscale image. Here, `W` is the width (number of columns), `H` is height (number of rows) and `threshold` is a predefined value, ranging from 0–255, 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-b 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) (http://www.logitech.com/pub/developer/quickcam/qcsdk.exe or http://www.mem.drexel.edu/pauloh.html). TRIPOD thus permits you to concentrate on writing the computer vision algorithms in ANSI C/C++ and avoid dealing with low-level Windows programming details 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 or XP and LEGO Mindstorms Vision Command camera. Microsoft Visual C++ 6.0 is used and only minimal MFC knowledge is assumed.

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 menu bar 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*, 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,
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 you should see `tripod.cpp` listed as shown in Figure 3(b). 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 Figure 3(c).

Step 5: Include QCSDK and MFC shared DLLs

The Quickcam SDK 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(a).

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

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, convert the red, green, and blue (RGB) pixels into a grayscale value, and store the frame pixels into a malloced row-column vector. All that remains is to add image processing routines. The added code (see Appendix) 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 shown in Figure 1.

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` (a copy of which is in the Appendix). This is possible by providing pointers to pixel data arranged in row-column vector format that is refreshed at frame rate.

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 `i`th B of the pixel. As will be discussed in the following, Windows uses 24-b images where each pixel is composed of three bytes (RGB; i.e., red, green, blue).

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/s) if the nested code is not computationally intensive, like simple threshold or centroid calculations.

The variable `m_destinationBmp` points to a 24-b 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-b RGB color image captured by the camera. Being RGB, each pixel in `sourceBmp` is represented by three bytes (red, green, blue).

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

Row-Column Vector Format

An image is an arranged set of pixels. A two-dimensional (2-D) 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(a). 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 one-dimensional (1-D) vector and is shown in Figure 5(b). 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 dereferencing 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 2-D arrays. First, memory for an array must be allocated before runtime. 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 run through the `malloc` routine at runtime. 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.

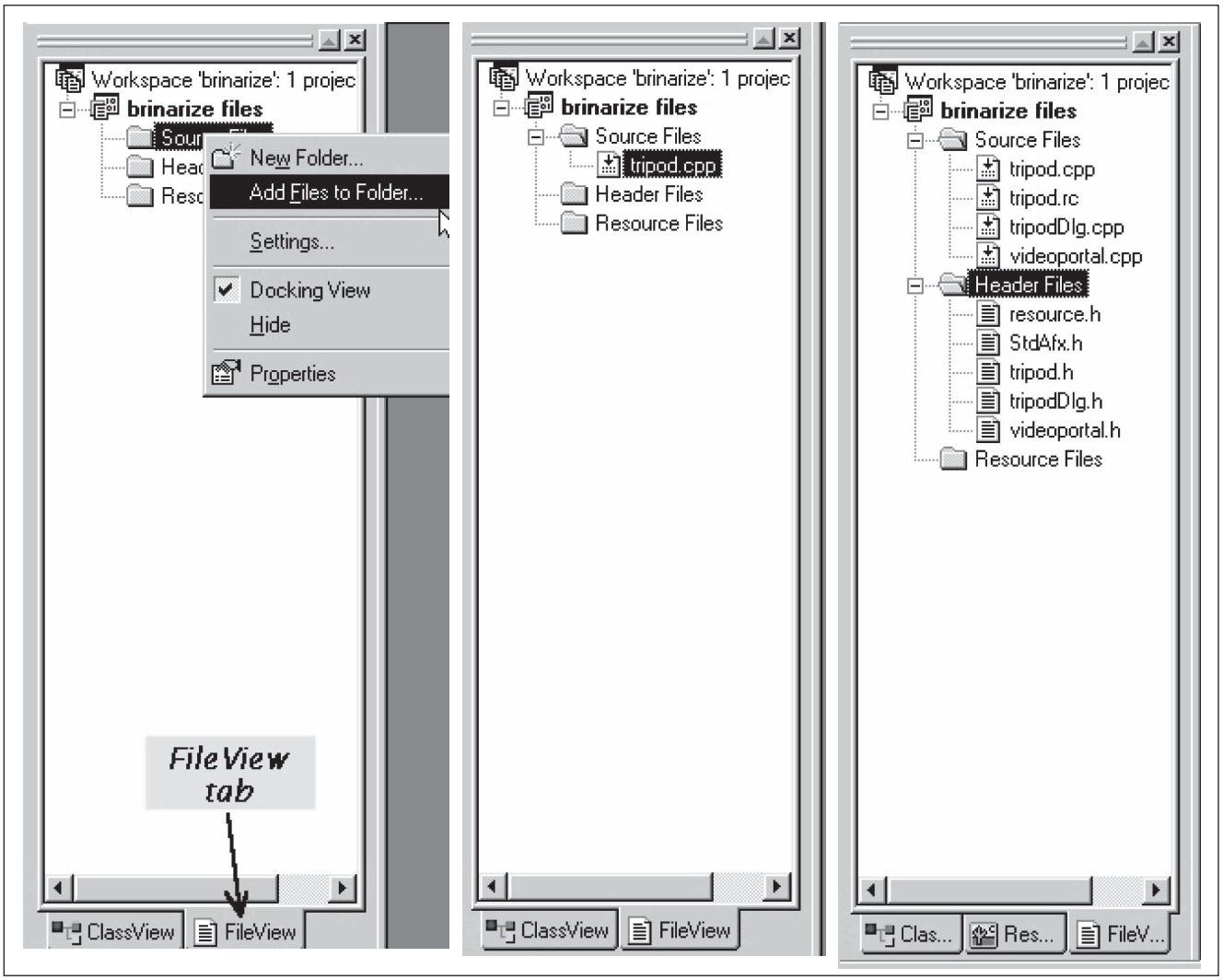


Figure 3. *Clicking* `FileView` *enables all files in the Source, Header, and Resource folders to be seen.*

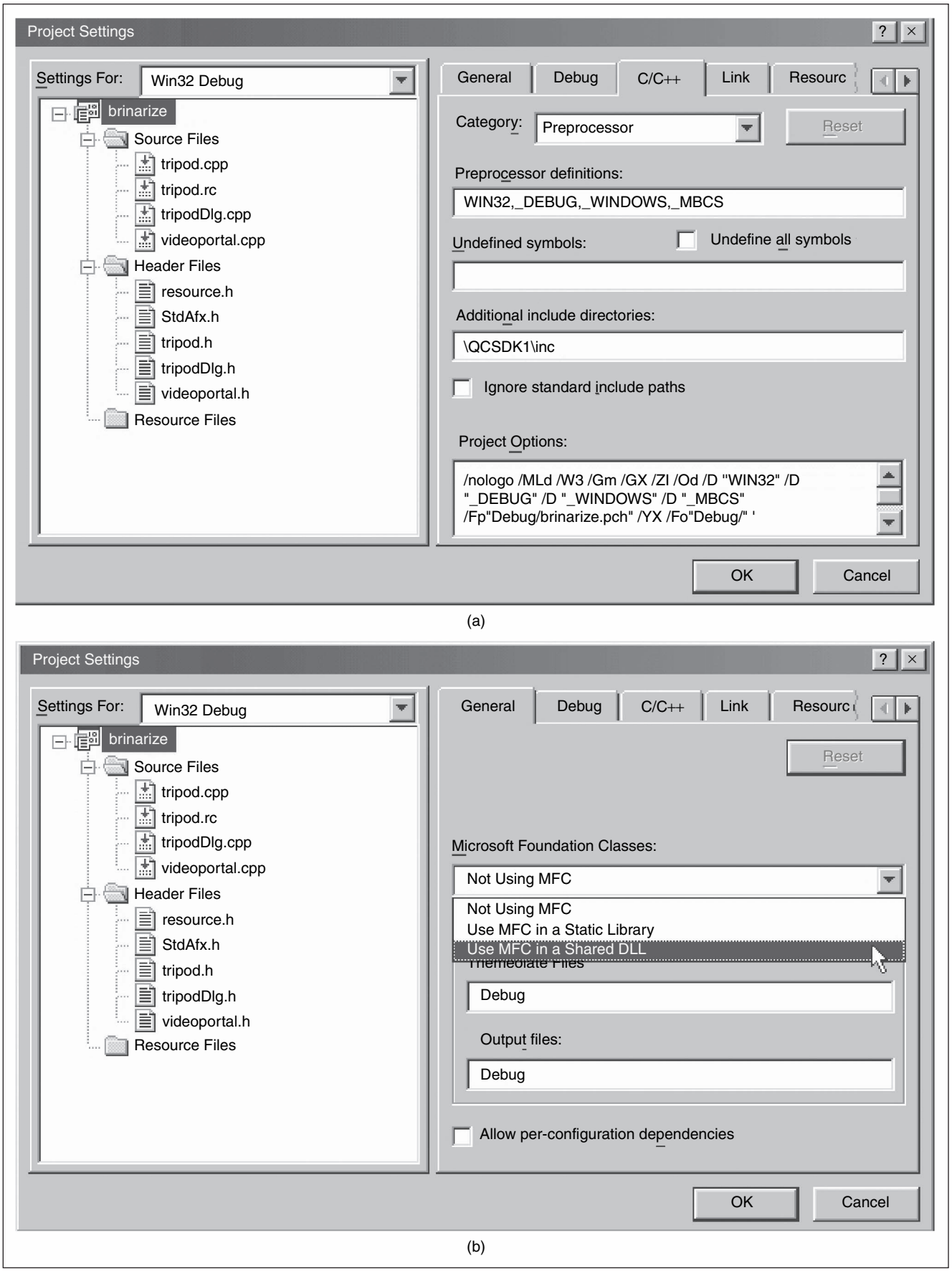


Figure 4. *Screenshot after including (a) QCSDK and (b) MFC Shared DLLs.*

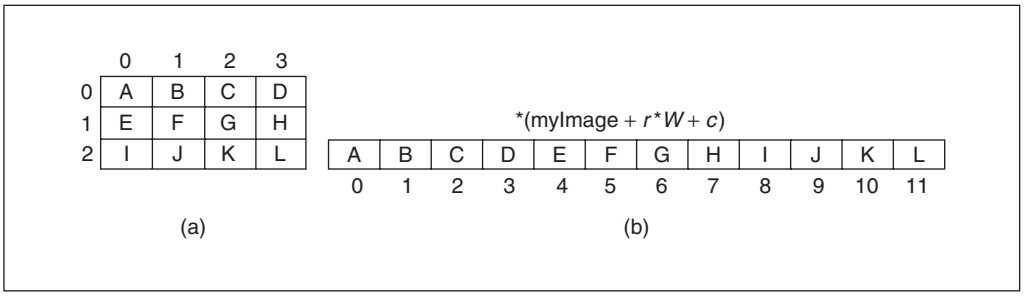


Figure 5. *Image data represented as (a) a matrix and (b) row-column vector.*

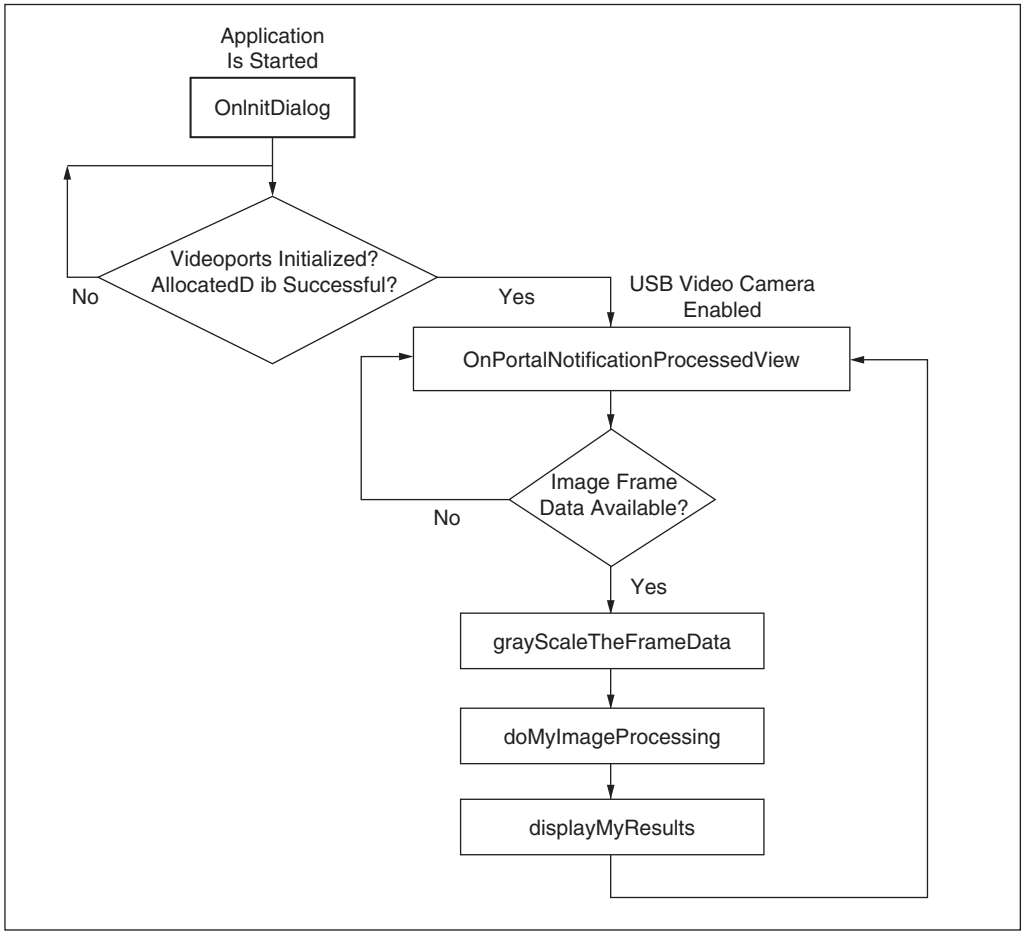


Figure 6. *The loop in the flowchart above executes* `doMyImageProcessing` *on every frame.*

24-b Bitmap Images

A 24-b image uses 3 B to specify a single pixel. Often, these bytes are the pixel's RGB contributions. RGB is also known as the *Truecolor format* since 16 million different colors are possible with 24 b. As mentioned previously, `m_destinationBmp` and `sourceBmp` are 24-b grayscale and Truecolor images, respectively. The variable `m_destinationBmp` makes all 3 B of a single pixel equal in intensity value. The intensity is a gray value computed from the amount of RGB 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 the "Appendix," 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 *and* 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.

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 true whenever the camera acquires a new image frame. After `OnInitDialog`, the code in `OnPortalNotificationProcessedview` checks for this flag and executes. Here, the code then assigns the pointer `lpBitmapPixelData` to the frame's pixel data, grayscales the color image, and 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/s.

Beyond binarized images, edge detection, laser rangefinder (Figure 7), binocular vision, multicamera views, tracking, and counting objects have been tested and implemented with TRIPOD.

Applications

The author's lab uses TRIPOD to quickly prototype code for tasks like vision-based tracking, navigation, and ranging. Two or more USB cameras can be used to develop applications involving stereo vision or multiple fields-of-view. This section highlights the lab's successes in visually servoing a robotic blimp and creating a low-cost laser rangefinder.

Visually Servoed Robotic Blimp

The Drexel University Autonomous Systems Lab (DASL) has been developing sensor suites for aerial robots. TRIPOD was used to help characterize computer vision for aircraft navigation. Figure 8 depicts a 5-ft airship that features two thrust motors to translate or pitch and one rudder motor for yawing. The goal was to visually servo the blimp by following a 20-ft line painted on the ground.

TRIPOD, which currently only works with USB Logitech cameras, was used to prototype visual-servoing algorithms. The code was then ported to third-party software (for

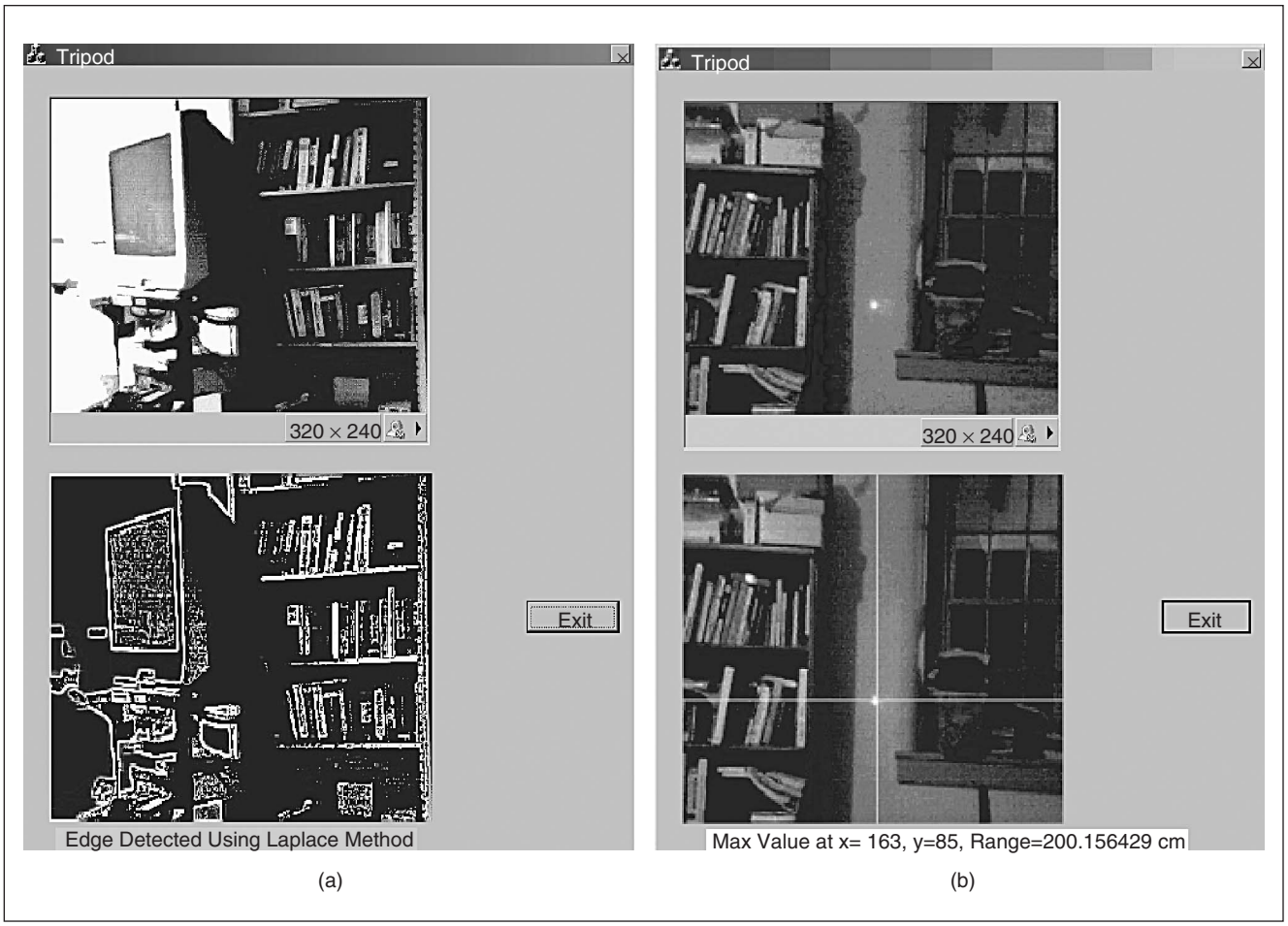


Figure 7. *TRIPOD screenshots in an office: (a) detecting edges and (b) acquiring range measurements.*

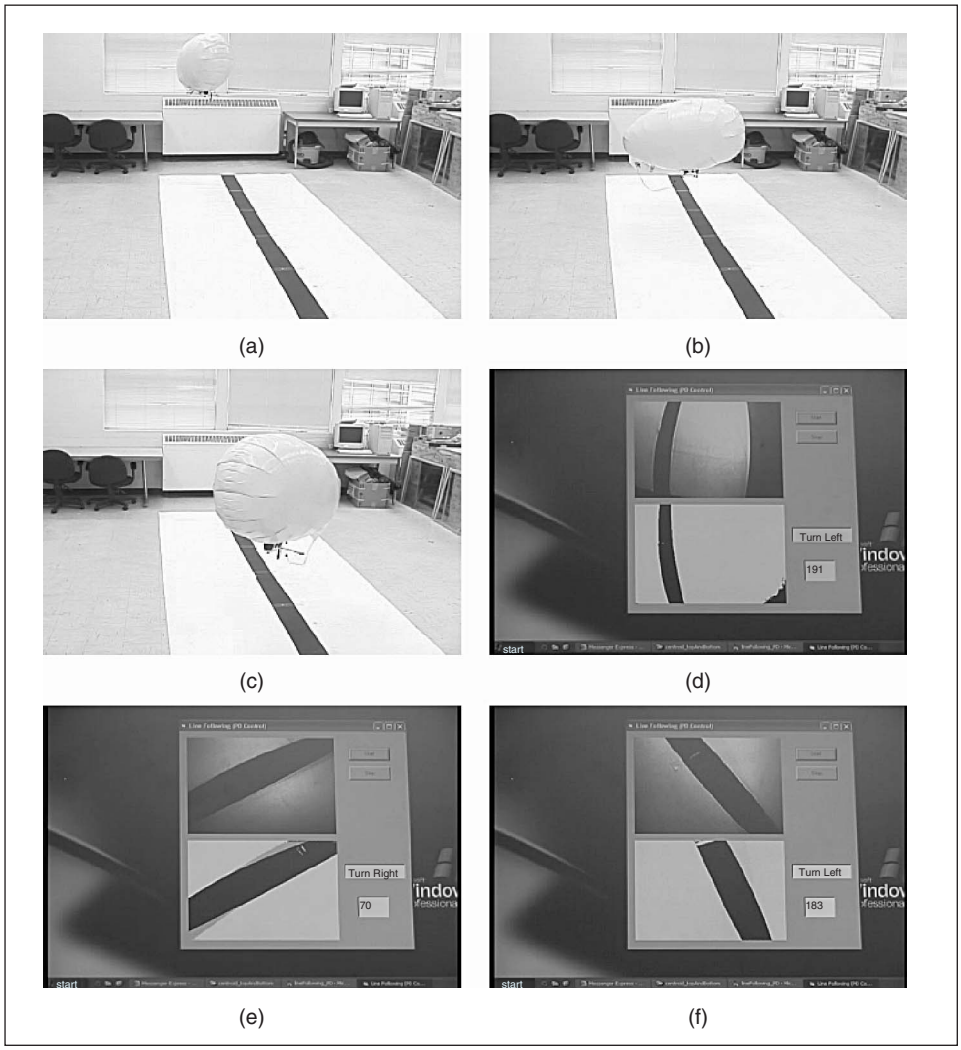


Figure 8. *(a) A robotic blimp is released. (b) Yaw and thrust is visually servoed by tracking a black line on the floor. (c) It reaches its goal location. (d)–(f) Corresponding raw and processed images also displaying commands to yaw left or right.*

example, an ActiveX control like Video OCX http://www.videoocx.de) for the blimp's wireless camera.

Thresholding live video, line orientation was estimated by measuring the centroid of the top and bottom halves of the image. Originally, both bang-bang and proportional controllers were tested; however, the blimp's large inertias yielded poor performance. Blimp dynamics [2] were incorporated, and a proportional-derivative controller was designed to successfully follow the line, as shown in Figure 8.

Laser Range-Finding

A Webcam and off-the-shelf laser pointer were combined to develop a rangefinder programmed under TRIPOD as shown in Figure 9(a). Here, the Webcam's optical and the laser's projection axes are made parallel. The distance between the axes h is configured thus defining a baseline. The brightest pixel in the image plane u corresponds to the target's reflection of the laser beam and hence the camera-to-target range D can be calculated. Figure 9(b) depicts the setup and notation from which one has

$$D = \frac{h}{\tan\theta}, \tag{1}$$

where θ is calculated with the horizontal pixel location u, radians per pixel pitch r_p, and radian offset r_o, as

$$\theta = u\, r_p + r_o. \tag{2}$$

The net effect is that the closer u is to the image center, the further the target is. The parameters r_p and r_o are determined by calibrating with known target distances. A linear relationship and a flat focal plane were assumed in (2), which worked reasonably well. Indoor tests with targets less than 200 cm yielded measurement errors of 0.78–7%. The Logitech Webcam has significant lens distortion. Measurements can be improved using calibration algorithms [7], Kalman filtering, or similar estimator.

Appendix

Source code for the `doMyImageProcessing` function. Pixel processing, thresholding for example, occurs in 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 increments 3 bytes in each column of image data
  i = (unsigned long)(row*3*W + 3*col);

  // Add your code to operate on each pixel (recall there are 3-bytes per
  // pixel. For example *(m_destinationBmp + i) refers to the ith byte
  // in destinationBmp.

  // Note: destinationBmp is a 24-bit grayscale image.  It is generated
  // by the grayScaleTheFrameData function found in tripodDlg.cpp.
  // 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);
```

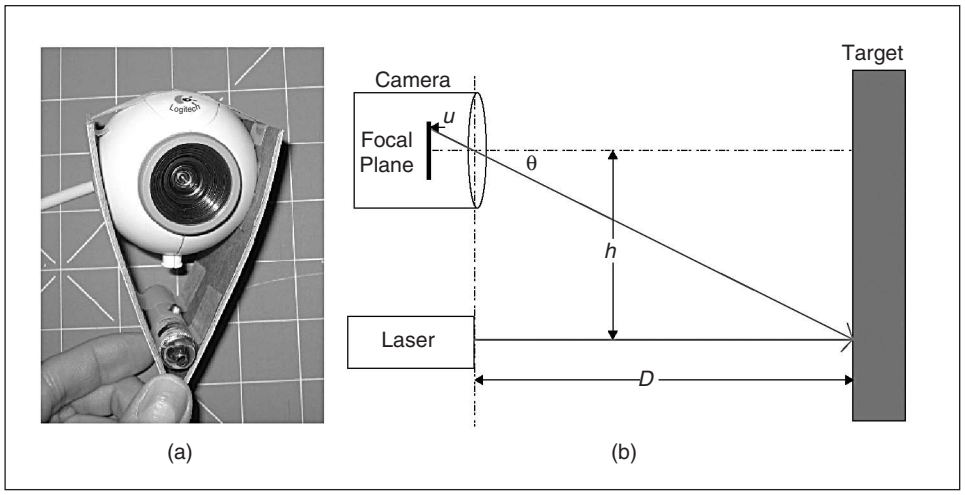


Figure 9. *(a) Webcam and laser pointer combined into a handheld unit. (b) The camera-to-target distance D can be calculated given the pixel location u, baseline h, and angle θ.*

Conclusions

A free and open-source software package called TRIPOD was presented in this article. A step-by-step tutorial was given, and applications like laser rangefinding and visual servoing were showcased. The motivation for TRIPOD stemmed from observations that affordable, widely available, and portable components, like LEGO, have made a strong impact on robotics. Suitable software, when combined with affordable video hardware, can similarly affect computer vision; a USB camera and freeware can give students and developers a hands-on tool to prototype computer vision code in or out of the lab. Towards this goal and to reach a large audience, TRIPOD was designed to work with USB Logitech cameras on Windows 98 and XP computers and requires only programming in ANSI C/C++. Beta versions of TRIPOD have received positive feedback by several university robot labs, including Drexel, Columbia, Harvard, Texas A&M, Brown, Carnegie Mellon, and Rutgers universities and the University of Southern California, and were showcased at the 2004 Robotics Education AAAI Spring Symposium at Stanford University.

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 runtime licenses. As such, they do not lend themselves to use outside research and teaching labs. The MATLAB Image Acquisition Toolbox appeared on the market in 2002 and is an excellent and useful package. The student version is affordable, but runtime performance is slow. Speed can be enhanced using a MATLAB C compiler, but this then requires more programming knowledge. TRIPOD only requires ANSI C/C++ knowledge because its template shields the programming from low-level Windows programming details.

Keywords

Computer vision, robotics, USB cameras, LEGO, real-time, image processing.

References

[1] G. Bradski, "The openCV library—An open-source library for processing image data," *Dr. Dobb's J.*, vol. 25, no. 11, pp. 120–125, Nov. 2000.

[2] W.E. Green, K.W. Sevcik, and P.Y. Oh, "A competition to identify key challenges for unmanned aerial robots in near-earth environments," in *Proc. IEEE Int. Conf. Advanced Robotics (ICAR)*, Seattle, WA, July 2005, pp. 309–315.

[3] L. Greenwald, Z. Dodds, A. Howard, S. Tejada, and J. Weinberg, "Accessible hands-on artificial intelligence and robotics education," AAAI, Stanford, CA, Tech. Rep. SS-04-01, 2004.

[4] H.R. Myler, *The Pocket Handbook of Image Processing Algorithms in C*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[5] P.Y. Oh, "TRIPOD—Computer vision for classroom instruction and robot design," AAAI, Stanford, CA, Tech. Rep. SS-04-01, pp. 42–47, 2004.

[6] E. Trucco and A. Verri, *Introductory Techniques for 3D Computer Vision*. Englewood Cliffs, NJ: Prentice-Hall, 1998.

[7] R.Y. Tsai, "An efficient and accurate camera calibration technique for 3D machine vision," in *Proc. IEEE Computer Vision Pattern Recognition*, Miami, FL, 1986, pp. 364–374.

Paul Y. Oh received a B.Eng. degree (Honors), in 1989, from McGill University, an M.Sc. degree, in 1992, from Seoul National University, and a Ph.D. degree, in 1999, from Columbia University, New York, all in mechanical engineering. His research interests include visual servoing, mechatronics, and aerial robotics. He was a Summer Faculty Fellow at the NASA Jet Propulsion Lab in 2002 and the Naval Research Lab in 2003. He is currently an assistant professor in the Mechanical Engineering and Mechanics department at Drexel University, Philadelphia, PA. In 2004, he received a National Science Foundation CAREER Award for controlling micro air vehicles in near-Earth environments. He chairs the IEEE Technical Committee on Aerial Robotics for the Robotics and Automation Society and is a Member of the IEEE, ASME, and SAE.

Address for Correspondence: Paul Y. Oh, Drexel University, 3141 Chestnut Street, Room 2-115 Mechanical Engineering, Philadelphia, PA 19104 USA. E-mail: paul@coe.drexel.edu.