JOURNAL OF OBJECT TECHNOLOGY

. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6. No. 6, July-August 2007

Diffraction Rangefinding in Java

By **Douglas A. Lyon**

Abstract

We describe a prototype of a diffraction range finding camera implemented in, and simulated by, Java. This sensor is able to digitize objects in 3D using only a single video camera, a diffraction grating and a means to position the target. The system is novel because it uses diffraction range finding to digitize objects and because it uses only the Java language for image processing, simulation and control.

Embedded machine vision applications of Java are still rare. Java's support for video digitization and serial port I/O is poor. The lack of I/O has limited Java's application in the area of image processing systems. Our 3D scanner uses Java for the digitization and processing of image sequences. The digitization is enabled by two API's. The first, called *QuickTime for Java* is an Apple Computer product that has been made available for Macintosh and Windows platforms. The second, called *javax.comm*, consists of a specification from Sun, with implementations available for several platforms. The *javax.comm* package enables the control of serial-port based devices. We have written drivers for an x-y table and a frame grabber. Performance has been a primary concern with the use of Java for image sequence processing. We show that Java is both fast enough for 3D reconstructions and for display of wire-frame and smooth shaded models.

1 INTRODUCTION

This paper presents a summary of the 3D digitization sub-project.

Java is portable, strongly typed and object-oriented. Java is typically weak in the area of hardware integration. Further, there is an inherent difficulty in writing Java native method interfaces to take advantage of local hardware [Gordon]. Finally, there are few engineering books available in Java [Lyon 99] [Lyon and Rao].

The rest of this paper is divided into the following sections: Serial Port control in Java, Video Digitization in Java, Range Finding via Diffraction in Java, Image Processing in Java and Computer Graphics in Java.

2 SERIAL PORT CONTROL IN JAVA

The serial port is a common way to interface a computer to a peripheral. Examples of serial-port interfaced hardware includes modems, robots, digital cameras, synthesizers, etc. Many computers have at least two serial ports. For example, Sun workstations have *ttya*

and *ttyb*. Wintel systems have *com1* and *com2*. Macintosh computers have a *modem port* and a *printer port*. Sun has an API specification for the support of serial ports, called the *Java Communications API* available from http://java.sun.com/products/javacomm. The new API has not been updated in some time, however.

However, Sun only supplies implementations for Windows and Solaris operating systems. Such two-platform distributions are limited compared with the number of platforms that can run Java today (e.g., Linux, FreeBSD, MacOS, SGI, etc.).

To bridge the technology gap, several third-parties have supplied implementations for the communications API on various platforms. The third party ports typically support only a subset of the communications API, called the *javax.comm* package. An example of using the *javax.com* package to write "atdt5551234" to the serial port follows:

```
import java.io.*;
import java.util.*;
import javax.comm.*;

public class SimpleWrite {
    static Enumeration portList;
    static CommPortIdentifier portId;
    static String messageString = "atdt5551234\n";
    static SerialPort serialPort;
    static OutputStream outputStream;
    public static void main(String[] args) {
        portList = CommPortIdentifier.getPortIdentifiers();
        while (portList.hasMoreElements()) {
            portId = (CommPortIdentifier) portList.nextElement();
            if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL)
        {
                if (portId.getName().equals("COM1")) {
                //[1]if (portId.getName().equals("/dev/term/a")) {
                //[2]if (portId.getName().equals("modem")) {
                    try {
                        serialPort = (SerialPort)
                            portId.openPort("SimpleWriteApp", 2000);
                    } catch (PortInUseException e) {}
                    try {
                        outputStream = serialPort.getOutputStream();
                    } catch (IOException e) {}
                    try {
                        serialPort.setSerialPortParams(9600,
                            SerialPort.DATABITS_8,
                            SerialPort.STOPBITS_1,
                            SerialPort.PARITY_NONE);
                    } catch (UnsupportedCommOperationException e) {}
                    try {
                        outputStream.write(messageString.getBytes());
                    } catch (IOException e) {}
                }
            }
        }
    }
}
```

To test the *SimpleWrite* class, a modem (that understands the "AT" command set) is connected to the serial port. The modem responds to the command "ATDT5551234" by performing a touch-tone dial to the number "5551234". Modems with a speaker typically

also emit audible tones (even when they are not connected to phone lines). Note that on a Sun the serial port is called *ttya*. On the Mac, the serial port is called *modem*.

To digitize an object in 3D, we use a serial-port interfaced stepper motor to drive an x-y table. This is shown in Figure 1.

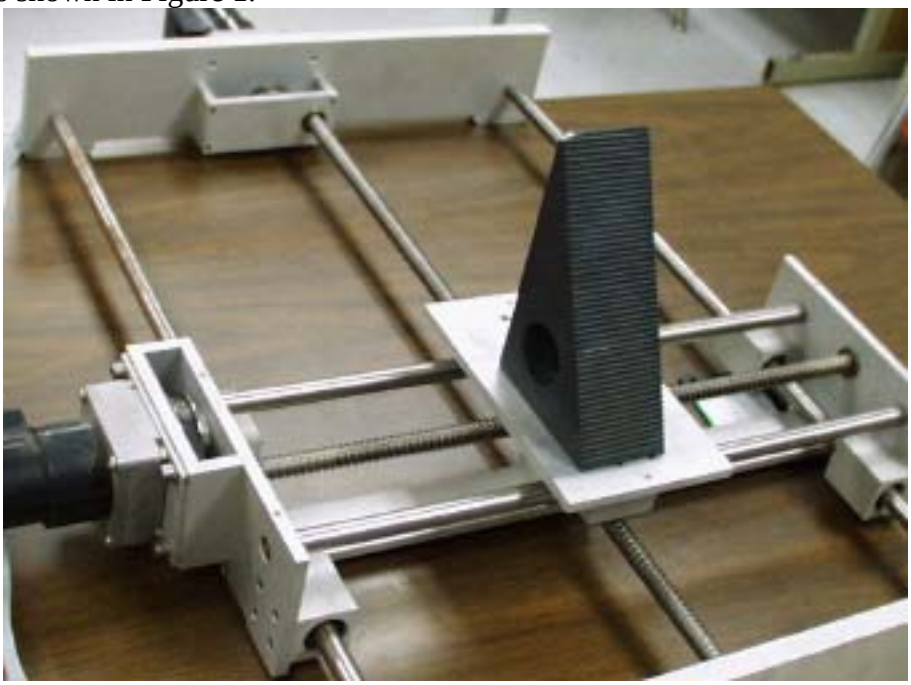


Figure 1. The x-y Table with Calibration Wedge

Figure 1 shows an image of the x-y table with a sample target (a calibration wedge).

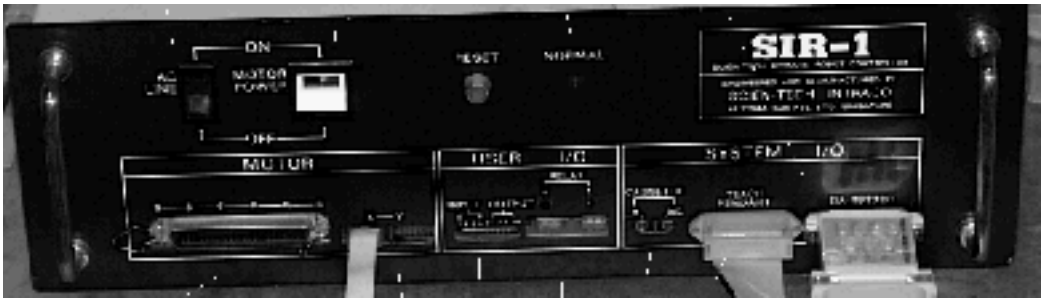


Figure 2. The Serial Port to Stepper Motor Controller

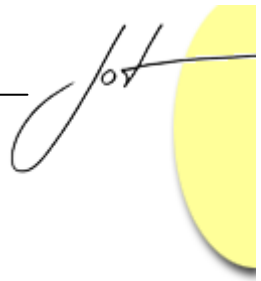
Figure 2. shows the stepper motor controller used to drive the turntable from the serial port. The controller contains driver electronics for 6 stepper motors which are able to drive a full, six-degree of freedom robot arm, as well as the turn-table, and an x-y table. The string "M 0,0,0,0,0,1,0" causes the x-y table's stepper motor to move single step.

3 VIDEO DIGITIZATION IN JAVA

To perform embedded machine vision tasks in Java, we grab frames from a video source. This feature is mentioned by Sun's Java Media Framework (JMF) API.

Another solution to video capture (available only for MacOS and Windows, as of this writing) is Apples' *Quicktime for Java. This* has been tested with a variety of video cards on both Windows and MacOS platforms. Two books with identical names (*QuickTime for Java*) have been published on the subject [Maremaa][Adamson]. Once the *Quicktime for Java* package is installed on the system, the following code enables the grabbing of a video frame:

```
import java.awt.*;
import quicktime.qd.*;
import quicktime.*;
import quicktime.std.StdQTConstants;
import quicktime.std.sg.*;
import quicktime.app.image.*;
import java.io.*;
public class FrameGrab
      implements
      StdQTConstants {

//the method will grab a frame as a Pict file and print "click"
      public static void main (String args[]) {
         FrameGrab fg = new FrameGrab();
         fg.snapShot();
         System.out.println("click");
      }

      private  SequenceGrabber sg;

//the method will return a JAVA Image if given a Pict instance
      public static Image pictToImage(Pict inPict) {
         ImageData id = null;
         try {
           id.fromPict(inPict);
         } catch (Exception e) {
           System.out.println(e);
         };
         QDRect r = id.getDisplayBounds();
         Image i = null;
         try {
           i =
                 Toolkit.getDefaultToolkit().createImage(new
                 QTImageProducer(
                        id, new
                               Dimension(
                               r.getWidth(), r.getHeight())));

         }
         catch (Exception e) {
           System.out.println(e);
         }
      return i;
         }

//will promt for a filename and call toFile(String fn)
      public void toFile() {
                 FileDialog fd = new FileDialog(
                 new Frame(),
                 "Gimme a pict name",
                 FileDialog.SAVE);
```



```
            fd.show();

            String fn= fd.getDirectory()+fd.getFile();
            toFile(fn);
        }

    //it will save the grabbed Pict object under the name fn
        public void toFile(String fn) {
          File file =
           new File(fn);
          Pict p = getPict();
          try {
           p.writeToFile(file);
          }
          catch (Exception e) {
           System.out.println(e);
          };
        }

    //will returns a Java Image of the grabbed frame
        public Image getImage() {
          return pictToImage(getPict());
        }

    //will return a Pict instance of the grabbedd frame
        public Pict getPict() {
          try {
          sg = new SequenceGrabber();
          }
          catch (Exception e) {
           System.out.println(e);
          };
          int offScreenDepth=8;
          int grabPictFlags =
                seqGrabToMemory |
                  seqGrabPreview |
                  seqGrabRecord |
                  seqGrabPlayDuringRecord    ;
          QDRect bounds=new QDRect(256,256);
          Pict p = null;
          try {
           p = Pict.fromSequenceGrabber(
                  sg,
                  bounds,
                  offScreenDepth,
                  grabPictFlags);
          }
          catch (Exception e) {};

          return p;
        }

    //used internally by main; handles the QTSession and grabs a frame as a
          Pict file
        private void snapShot() {
            try{
            QTSession.open();
            toFile();
          } catch (Exception ee) {
            ee.printStackTrace();
            QTSession.close();
          }
        }

    }
```

An instance of a Java *Image* is returned with the invocation of:

```
FrameGrab fg = new FrameGrab();
Image img = fg.getImage();
```

This greatly simplifies the cross-platform porting of embedded machine vision software and provides a high-level object-oriented interface for the programmer to obtain images on-line. Facilities are also present in the *FrameGrab* class to save the image to a file using the *PICT* file format. We have software for support of other file formats too, including PPM and GIF.

The *Kahindu* program (described in [Lyon 99]) contains a large API for image processing. The *Kahindu* program contains over 123 files with features that include wavelet processing, convolutional filtering, transforms, etc. *Kahindu is* supplied with source code and is written in 100% Pure Java. Once the *QuickTime for Java* package is introduced, the code becomes "impure" Java. Thus, there are two versions of the *Kahindu* program, one with *QuickTime for Java* and one without. This is a natural course of events when experimenting with a new API, and probably cannot be helped.

The primary reason for establishing a 100% Pure Java version version of the program is to make clear that it requires no special API's. Once we limit ourselves to an API with a native implementation, the number of platforms that can be supported is greatly reduced.

4 RANGEFINDING VIA DIFFRACTION IN JAVA

We use diffraction range finding to scan a target in 3D. Scanning is performed with a video camera, x-y table, diffraction grating and a source of illumination. The introduction of a diffraction grating into the digitization system enables ranging into concavities because of how light is bent when it passes through the grating. In the far-field, a diffraction grating bends light in accordance with the diffraction equation, namely that the sum of the sines of the angles passing through the grating varies in inverse proportion to the gratings' pitch. Diffraction range finding theory has been discussed at length in [DeWitt and Lyon A].

In order to simplify the design of diffraction range finders, a Java program was developed called *DiffCAD*. This program remains unique, as far as we know, in that it is the only program available for the design of diffraction range finders and first appeared in [Lyon and Rao].

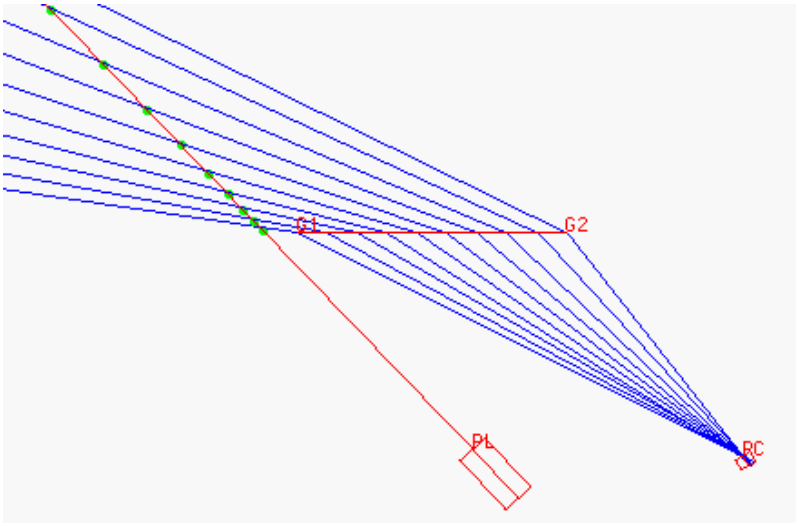


Fig. 3. Output of the *DiffCAD* Program

Fig. 3 shows the output of *DiffCAD*. The center of a pin-hole camera is labeled *PC*. The left and right side of the grating is labeled *G1* and *G2*. The center of the front of the laser is labeled *PL*. The distance between the slits in the diffraction grating is called the *pitch* of the grating. This varies linearly as a function of displacement across the grating.

A strip of light leaves the laser at *PL* and strikes the target at points represented by the green dots. Light is reflected from the green dots through the grating. The light rays are bent in accordance with the geometric model of diffraction and pass through the pin-hole in the camera, *PL*.

The camera, laser, grating, illumination wavelength, etc., are all manipulated by the user during interaction with the *DiffCAD* program. Response and redraw times are less than a second, even on the slowest of machines. This type of optical modeler allows users to get a good simulation of diffraction range finder performance for a variety of configurations. Such a modeler makes optical design accessible, without having to grapple with cumbersome mathematics.

Fig. 4. Photo of the Bench

An optics bench is provided, with accessories, to enable experimentation with a variety of configurations. A photo of the optical bench, with accessories is shown in Figure 4. The laser is shown on the left, the camera and diffraction grating are shown on the right. One property of interest is that the variable pitch diffraction grating enables a folding-back of the rays.

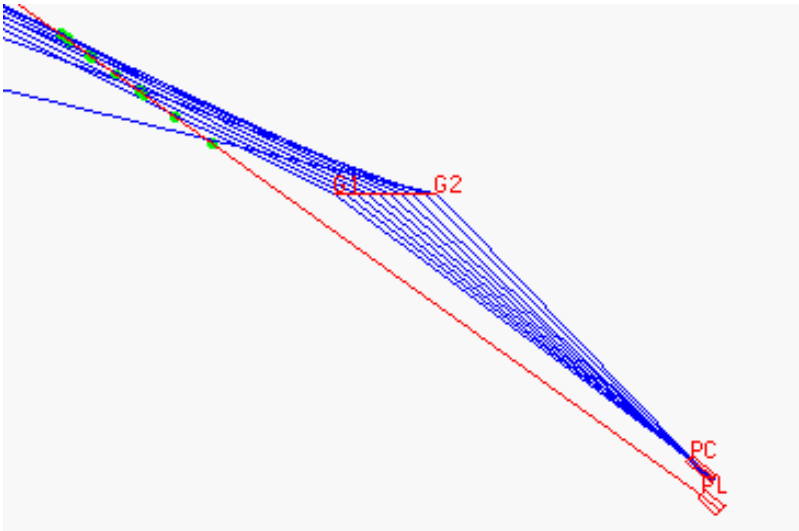


Figure 5. Set-up showing rays folding back

Such a system enables the ranging of a target such that accuracy may increase at a large offset, and then fall off rapidly. This type of change in accuracy is not typical of triangulation range finding systems.

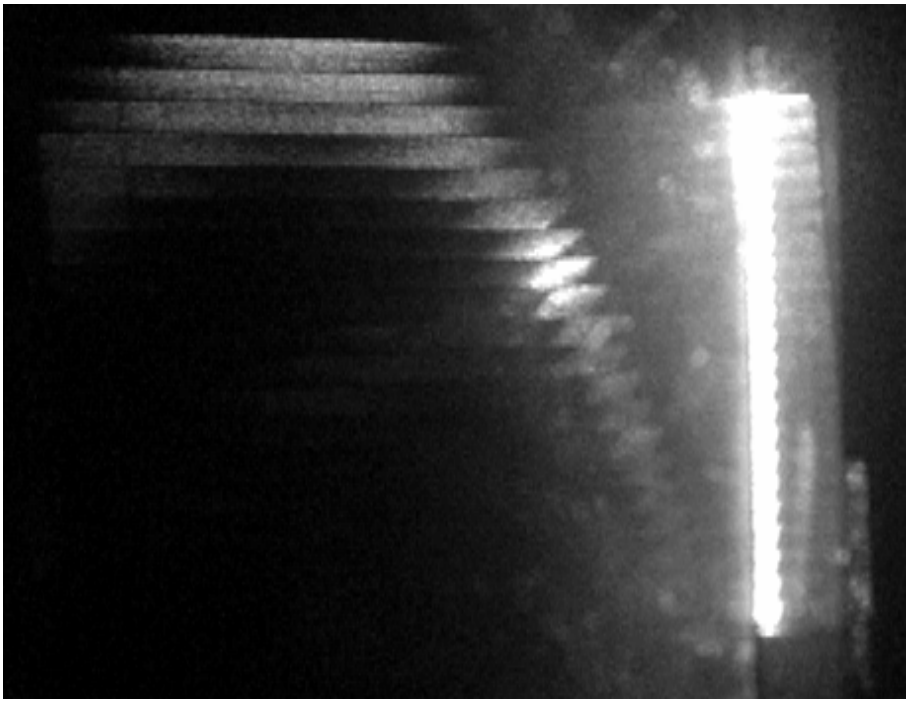


Fig. 6. Accuracy Can Increase with Increased Offset.

Fig. 6 shows an image from the camera of an illuminated stepped wedge viewed through a variable pitch diffraction grating. This phenomenon, first predicted by *DiffCAD* and shown in Fig. 5, is atypical of most range finders. The phenomenon is the subject of an improvement patent [Ditto and Lyon 1997]. Fig. 5 shows an image, whose peak uses 182 pixels to measure a 2.54 mm step. This gives an accuracy of 13.8 microns with a 200 mm offset.

It has been shown that using an illumination source that is coaxial with respect to the camera enables an increase in occlusion immunity with respect to stereo gram based techniques. Diffraction systems have been used to range into holes of Fibre optic spinnerets. Such systems have holes that are 2.5 mm long and 0.2 mm in diameter [Wei et Al.] [DeWitt and Lyon B].

5 IMAGE PROCESSING IN JAVA

Image processing is an essential part of any embedded machine vision system. As a result, a 100% Java solution has been brought to market in the form of a book [Lyon 99]. Source

code is available from http://www.docjava.com and is called *Kahindu* (named for a region of Kenya that produces AA coffee).

Real-world image processing hardly ever works as simply as text-book image processing. For example, using one type of diffraction element, five orders of diffraction are visible at once, as shown in Figure 7.

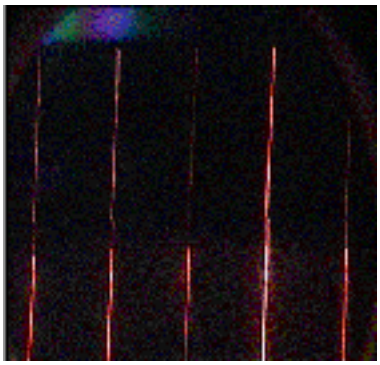


Fig. 7. Five orders of Diffraction

We isolated the brightest order of diffraction and turn it into a single pixel-width wide edge. After several experiments, a process was found that produced Figure 8.

Fig. 8. A Single Pixel-Width Wide Diffraction Order

Fig. 8 shows a single pixel-width wide edge extracted from Fig. 7. The processing that produced Fig. 8 included low-pass filtering, thresholding, skeletonization, median square filtering, more skeletonization and another thresholding. Such ad-hoc image processing is arrived at only through repeated experiment.

6 COMPUTER GRAPHICS IN JAVA

There are several pure Java implementations of 3D graphics packages available. We use a package by Peter Walser called *idx3d* (available at http://www.vis.inf.ethz.ch/students/pwalser).

Embedded Java solutions typically have a small API implementation and probably lack non-core API's, like Java 3D and JAI. It is only with a 100% Java solution that 3D and image processing features can be brought to market on an embedded system with minimal porting effort. Thus, the non-hardware, non-native implementations of an API are better suited for a wide variety of embedded hardware.

The *idx3d* package is written in only 2,500 lines of Java code, and as such provides only a subset of the features present in Java 3D. However, the entire package represents less than 80k bytes of class files and thus has the advantage of a small foot-print. Features supported include height-field rendering, texture mapping and polygon boundary representations. We are currently working on improving the rendering algorithms and expanding the geometric representations.

The *idx3D* package is able to display, rotate, translate and render (using smooth shading, wire-frame or texture mapping). The rate of rendering is a function of the CPU speed and the implementation of the just-in-time compiler. A scan of a 3D resolution target (the calibration wedge) permitted a reconstruction that could be viewed in real-time. Fig. 9 shows a wire frame rendering of a wedge.

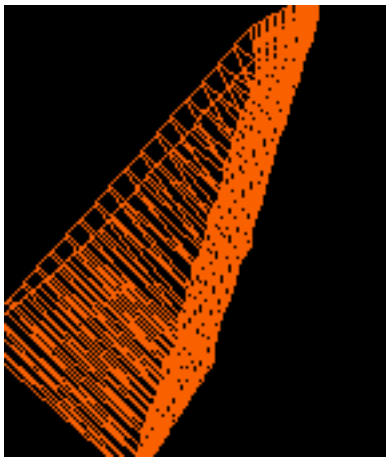


Fig. 9. A Wireframe Rendering of the Calibration Wedge

A simple smooth shading is also available. Fig. 10 shows a smooth shaded version of the wedge, as viewed from two angles.

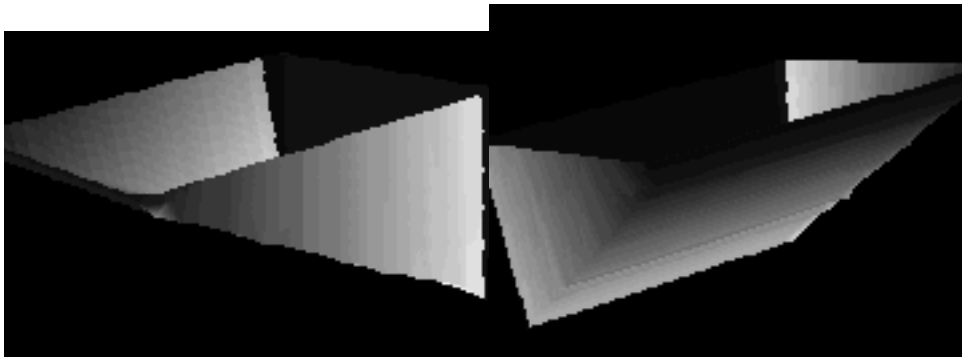


Fig. 9. A Wireframe Rendering of the Calibration Wedge

A simple smooth shading is also available. Fig. 10 shows a smooth shaded version of the wedge, as viewed from two angles.

Fig. 11. Keyboad to be scanned.

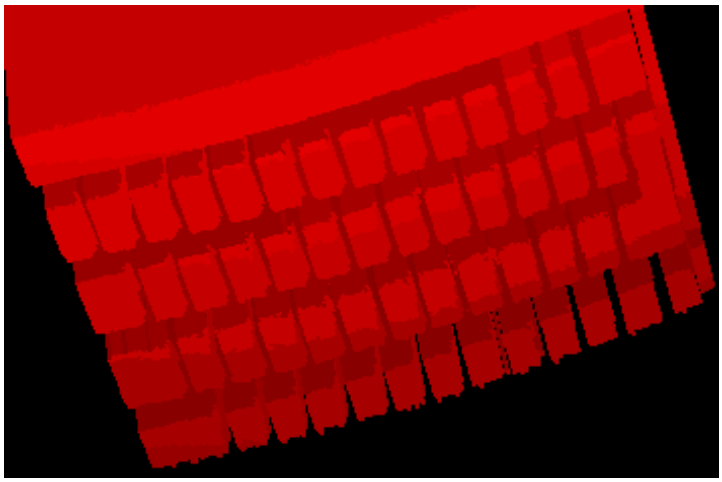
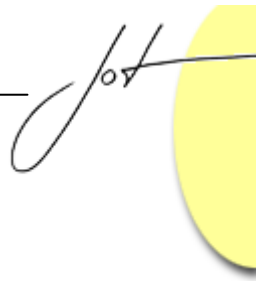


Fig. 12. Scanned in Keyboard with Smooth Shading

Fig. 12 shows the keyboard rendered using smooth shading. The classic problem with this type of scanning is to determine what a good edge is in the face of specular noise from the surface of the image. Such sub-problems are addressed using a variety of techniques (surface smoothing, noise models, etc.) and are beyond the scope of this paper.



7 CONCLUSION

We described a prototype of a diffraction range finding camera implemented in, and simulated by, Java. This is the first such prototype, as far as we know. The system is novel both because it uses diffraction range finding to digitize objects and because it uses only the Java language for image processing, simulation and control.

Writing device drivers in Java is eased by API's for driving the serial port and video digitizer. With support for video digitization and serial port I/O a wide variety of image processing systems become possible. As the API's are deployed on more platforms, the embedded control software should become more portable.

The use of Java to support the development of this prototype has come with a small performance cost. The advantage is portability and object-orientation in a structured and strongly-typed programming language.

While we have used a 100% Java implementation of the 3D graphics, there are native method implementations and hardware accelerators (like Java 3D) which are faster. Even so, we have seen that Java is both fast enough for 3D reconstructions and for display of wire-frame and smooth shaded models.

The source code for the programs described in this article are available from http://www.docjava.com. A more detailed exposition may be found in [Lyon and Rao] and [Lyon 99].

LITERATURE CITED

[Adamson] Chris Adamson, *QuickTime for Java*, O'Reilly Media, Inc. 2005.

[DeWitt and Lyon A] "Rangefinding Method Using Diffraction Gratings", *Applied Optics*, by Thomas D. DeWitt and Douglas A. Lyon, May, 10, 1995, Vol 34, No. 14, pp. 2510-2521.

[DeWitt and Lyon B] "Three Dimensional Microscope using Diffraction Grating", Optcon, SPIE - International Society for Optical Engineering, by Thomas D. DeWitt and Douglas A. Lyon, Philadelphia, PA, October 24, 1995, 2599B-35.

[Ditto and Lyon 1997] "Variable pitch grating for Diffraction Range Finding", By Tom Ditto and Douglas Lyon, Number 60/034,112 International Patent Pending (PCT) December 30, 1997.

[Gordon] *Essential JNI* Java Native Interface, by Rob Gordon, Prentice Hall, Upper Saddle River, NJ, 1998.

[Lyon and Rao] *Java Digital Signal Processing*, by D. Lyon and H. Rao, M&T Books, NY, NY. 1998. Available from http://www.docjava.com.

[Lyon 99] *Image Processing in Java*, by D. Lyon. Prentice Hall, Upper Saddle River, NJ. Due in Jan. 1999. Available from http://www.docjava.com.

[Maremaa] Tom Maremaa and William Stewart, *QuickTime for Java*, Morgan Kaufmann, 1999.

[Wei et Al.] 1998. "Intensity and Gradient-based Stereo Matching using Hierarchical Gaussian Basis Functions" by Guo-Qing Wei, Wilfried Brauer and Gerd Hirzinger, *IEEE PAMI* Vol. 20, No. 11, November 1998. pps. 1143-1160.

ACKNOWLEDGMENTS

This project was made possible, in part, by a Instrumentation Laboratory Improvement grant, DUE-9451520, from the National Science Foundation and by a Larsen Professor grant from the Larsen Fund.

Thanks are also due to Raul Mihali, who assisted with the images in this article.

Thanks are due to Tom Ditto, for numerous suggestions and discussions about diffraction range finders.

About the author

Douglas A. Lyon (M'89-SM'00) received the Ph.D., M.S. and B.S. degrees in computer and systems engineering from Rensselaer Polytechnic Institute (1991, 1985 and 1983). Dr. Lyon has worked at AT&T Bell Laboratories at Murray Hill, NJ and the Jet Propulsion Laboratory at the California Institute of Technology, Pasadena, CA. He is currently the Chairman of the Computer Engineering Department at Fairfield University, in Fairfield CT, a senior member of the IEEE and President of DocJava, Inc., a consulting firm in Connecticut. Dr. Lyon has authored or co-authored three books (Java, Digital Signal Processing, Image Processing in Java and Java for Programmers). He has authored over 30 journal publications. Email: lyon@docjava.com. Web: http://www.DocJava.com.