Using image processing to teach CS1 and CS2

Dr. Kenny Hunt
Computer Science Department
The University of Wisconsin – La Crosse
<hunt@mail.uwlax.edu>

Abstract

The use of digital image processing techniques in undergraduate computer science curriculum has advantages in terms of motivating student interest and immediate, visual feedback of executed code. Although the standard Java distribution includes support for basic image processing operations, including the display of images, the complexity of the package renders it unsuitable for inexperienced programmers. This paper presents an extension to the built-in image processing package that is suitable for use in CS1 and CS2 courses and suggests ways that the package can be used to teach topics in these courses.

Introduction

The increasing use of digital images and image processing techniques makes the study of digital image processing an important sub-discipline of computer science. Image processing is used in a wide array of applications including sports broadcasting, mail delivery, military target acquisition, satellite imaging, robotics, medical imaging and the traditional print industry. Although there is great diversity in practical application of these techniques, fundamental image processing techniques are used throughout each of these fields and many of these techniques are easily accessible to undergraduate programming students [1, 2, 4]. The goal of this paper is not to advocate the development of image processing courses, but to provide practical ways in which image processing techniques can be infused into existing undergraduate curriculum.

A digital image is, at the most abstract level, a two-dimensional array of colored pixels or dots. When these pixels are displayed on a high-resolution monitor and viewed at an appropriate distance, they appear to be a continuous colored image. Each pixel is a certain color which is typically defined, using the red-green-blue (RGB) color model, as a combination of varying amounts of red, green, and blue light. A color image is therefore said to contain three bands, each of which represents the amount of red, green, or blue light in the image. Whereas a color image contains both color and intensity information, a gray-scale image is an image composed of pixels that vary only in intensity, not color. Grayscale images therefore have only a single band.

The total number of pixels in a color image can be quite large. Modern high-end digital cameras take images with resolutions in the range of 4-5 Mega-pixels while even mid-range cameras take images in the 2-3 Mega-pixel range. Since each pixel of a color image typically occupies 4 bytes, an uncompressed digital image will consume in the range of 10 megabytes of memory. Since the memory requirements for images are large,

image-processing software often compresses the image data using standard formats such as GIF, JPEG or PNG.

Implementation

The standard Java distribution, J2SE [3], includes full support for reading and writing digital images to file, viewing digital images as part of a graphical interface, and manipulating digital images at either low or high levels of abstraction. The presentation in this paper uses the Java 2 Platform, Standard Edition version 1.4 throughout.

Java's Built-in Image Classes

J2SE supports both immediate and asynchronous mode image processing operations. The asynchronous mode operations are useful for developing applications like web browsers where images can be displayed when the data is available while not blocking operation of the entire browser as the data is loaded. Immediate mode operations block until image data is loaded and, since threads are not utilized, are suitable for CS1 and CS2 classrooms.

In J2SE the most generic properties of digital images are captured in a high-level abstract Image class but the most useful class for image processing is a subclass known as BufferedImage. A UML class diagram of the BufferedImage is depicted in Figure 1. A BufferedImage contains a Raster, representing a two-dimensional array of pixels, and a ColorModel, which specifies how to interpret the Raster data. A Raster contains a DataBuffer, representing the actual low-level pixel data and a SampleModel, which specifies how to interpret this pixel data. Image processing methods will typically take a BufferedImage object as a parameter and manipulate the image using a WritableRaster (a Raster subclass).

Thorough knowledge of each of these classes is required to perform even the most basic image processing operations. While this is not a substantial overhead for experienced

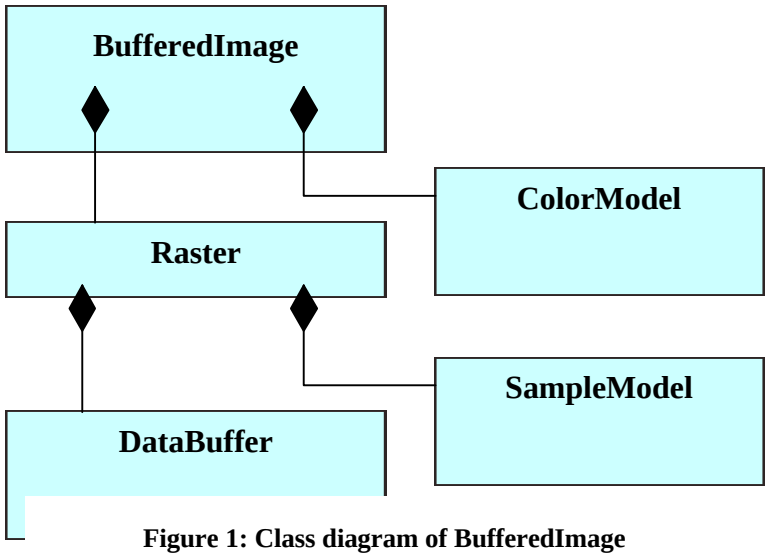


Figure 1: Class diagram of BufferedImage

programmers, it is prohibitive to most undergraduates taking introductory CS1 or CS2 level courses. Even in upper-division courses, it is not tenable to introduce students to the built-in image-processing package only for the purpose of giving domain-specific examples of Huffman encoding or quad-tree decomposition. An EasyBufferedImage class has therefore been developed which hides this complexity and provides convenient methods for creating and manipulating digital images.

Extensions to the Built-in Library

The EasyBufferedImage class is shown in the UML class diagram of Figure 2 below. It is a direct subclass of BufferedImage and is therefore fully compatible with the built-in imaging capabilities. The class contains a number of static methods to create EasyBufferedImage objects by reading from a local file, loading an image via remote URL access, or creating an image directly from pixel data. The only way to construct an EasyBufferedImage is through one of the createImage methods as there is no public constructor available. The export method saves an image in GIF format while the get/set pixel methods allow read/write access to pixel data through direct array manipulation.

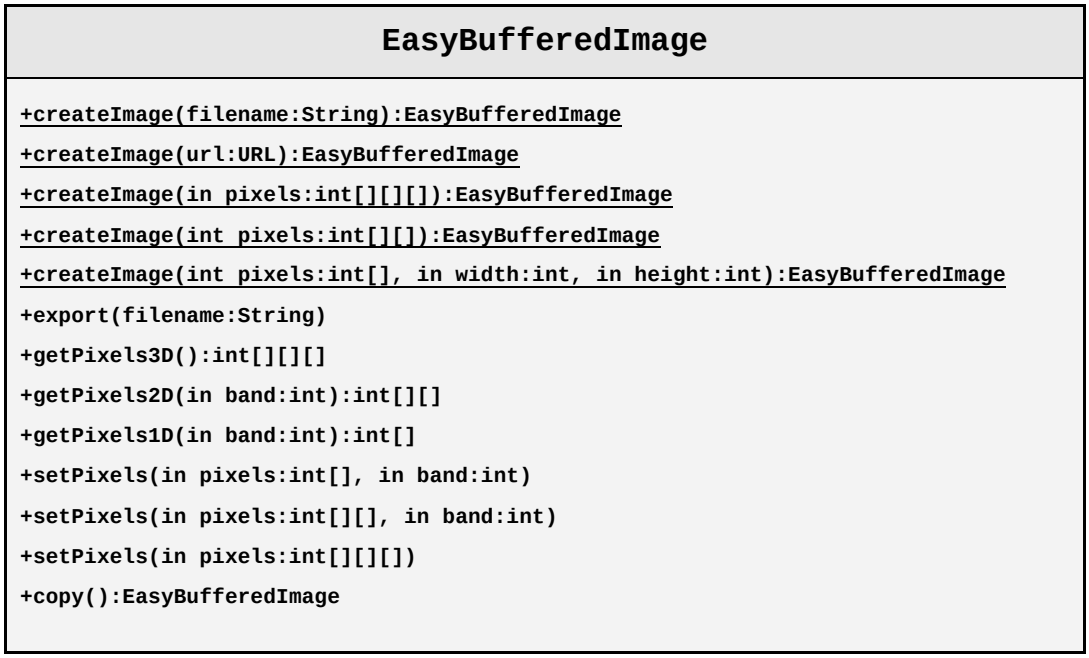


Figure 2. UML overview of the EasyBufferedImage class

For example, to load an image from a local file, a student can write the following code. The file must be in GIF, JPEG, or PNG format and a FileNotFoundException is thrown if the specified file doesn't exist.

```
EasyBufferedImage image = EasyBufferedImage.createImage("myImage.gif");
```

Alternatively, to load an image over the Internet, a student may write the following code. The resource file must again be in an appropriate format and an exception may be thrown if the resource is not accessible.

```
URL url = new URL("http://charity.cs.uwlax.edu/images/image.gif");
EasyBufferedImage image = EasyBufferedImage.createImage(url);
```

Once an image is generated, it is a simple matter to access and modify pixels using the get/set pixels methods. The getPixels method takes one of three forms. The getPixels1D method returns a 1D array of pixel values corresponding to the specified band. The array contains all of the pixel values in the specified band arranged in row-major format. The band value must be GRAY for grayscale images or one of RED, GREEN, or BLUE for color images. Note that the pixel values themselves will always be in the range 0 to 255 inclusive. The getPixels2D method returns a 2D array of pixel values corresponding to the specified band. The number of rows in the array corresponds to the height of the image and the number of columns in the array corresponds to the width of the image. The getPixels3D method returns a 3D array of pixel values. The first two dimensions of the array are identical in meaning to the 2D method while the third dimension corresponds to the number of bands in the image. The number of bands will be 1 for a gray-scale image or 3 for a color image. The setPixels method takes one of three forms, each of which is symmetric to a get method. The set methods will throw an IllegalArgumentException if the pixels array is not compatible with the dimensions of the image being changed.

Display of Images

The swing package supports methods to display images on graphical components. The EasyBufferedImage package provides an ImagePanel component which displays an EasyBufferedImage on the available component area, clipping the image if the component is not large enough to display the entire image and centering the image if the component is larger than the displayed image.

The image-processing equivalent of a "hello world" program is listed below. This program creates a window that displays an image loaded from a file where the file name is specified as a command-line argument.

```
class ImageViewer extends JFrame {
    ImageViewer(EasyBufferedImage image) {
        super("Image Viewer");
        ImagePanel panel = new ImagePanel(image);
        getContentPane().add(panel);
        addWindowListener(new WindowAdapter() {
           public void windowClosing(WindowEvent e) {
                  System.exit(0);
            }});
    }
```

```
    public static void main(String[] args) throws IOException {
        EasyBufferedImage image = EasyBufferedImage.createImage(args[0]);
        ImageViewer viewer = new ImageViewer(image);
        viewer.setSize(image.getWidth(), image.getHeight()+30);
        viewer.show();
    }
}
```

Incorporating Image Processing into CS1, CS2 and Beyond

The EasyBufferedImage and ImageViewer classes can be utilized in introductory programming courses to teach a number of fundamental programming concepts. This section suggests a number of ways in which image processing techniques can be used to illustrate and teach fundamental skills such as array processing and tree traversal.

Array Processing

In the EasyBufferedImage class, pixel data is consistently viewed as an array of one, two, or three dimensions. Students in CS1 are typically introduced to arrays using algorithms such as linear search for the maximum or minimum element or computing the sum of all array elements. Certain digital image point-processing techniques are of comparable complexity and are easily accessible to CS1 students. Inversion, thresholding, and linear mapping are image processing techniques, which can be quickly explained, and produce interesting visual effects when applied to certain digital images. Since the pixel data is provided in three separate formats, there are three code variants for each algorithm.

In addition to these simple point-processing techniques, the idea of representing two-dimensional data as a one-dimensional array also foreshadows questions of layout that are typically covered in a language concepts course. Histogram computation, computing the number of times each pixel value occurs in an image, is another more advanced topic that is easily explained and provides an excellent example of indirection. Each of these topics is expanded in the subsections below.

Inversion

The inverse of a digital image is directly analogous to the traditional film negative. Given an 8-bit pixel with value V, the inverse of that pixel is defined as 255-V. The inverse of an image is then computed by inverting each pixel in each band of the original image. The actual code to perform image inversion is straightforward and serves as an example of processing arrays of various dimensionalities.

Thresholding

Thresholding is a quick way to convert grayscale images into black-and-white images although interesting visual effects can be achieved when this technique is applied to a color image. Given an 8-bit pixel with value V, the thresholded value is either 0 or 255 depending on where V is above or below the center of the 8-bit dynamic range. The output image is generated by thresholding every pixel of every band in the original image.

Figure 3 Thresholding: Original Image (left), threshold = 128 (center), adaptive threshold (right)

$$threshold(V) = \begin{cases} 0 & if \quad V \le 128 \\ 128 & if \quad V > 128 \end{cases}$$

A simple modification of this algorithm is to choose the threshold value to be the average of all pixels in the input image. This is an adaptive strategy that yields superior results for images having poor global contrast. Figure 3 shows examples of each thresholding method.

Linear Mapping

Linear mapping is used to brighten or darken images. Given a pixel with value V the value of the output pixel is α***V +** β where α is a real-valued scaling factor and β is an offset value. An output image can be generated by linearly mapping every pixel in every band of the original image. If $0<\alpha<1$ the image is darkened, if $\alpha=1$ the image is unchanged, and if $\alpha>1$ the image is brightened. Thresholding can be represented as a linear mapping where $\alpha=-1$ and $\beta=255$. Note that this transform may map pixels to invalid 8-bit output values. Values outside of valid ranges must be clamped in order to assure reasonable results.

Floyd-Steinberg Dithering

Although it is common to see gray-scale images in newspapers, books, and magazines, it is not obvious how using only black ink on white paper can create such images. The process of converting a gray-scale image into a black-and-white image is known as dithering and the most common dithering technique is known as Floyd-Steinberg error-diffusion dithering.

This technique is a modified thresholding operation. The algorithm proceeds by moving left-to-right-top-to-bottom over each pixel. Each pixel is thresholded on the value 128 for 8-bit images. An error-diffusion step is then performed where the “error” between the input and output pixel is propagated to neighboring pixels according to a fixed system. The error E at pixel P is the quantity (P-threshold(P)). The error E is distributed such that the pixel to the immediate east of P is incremented by 7E/16, the pixel to the south-west of P is incremented by 3E/16, the pixel to the south of P is incremented by 5E/16 and the

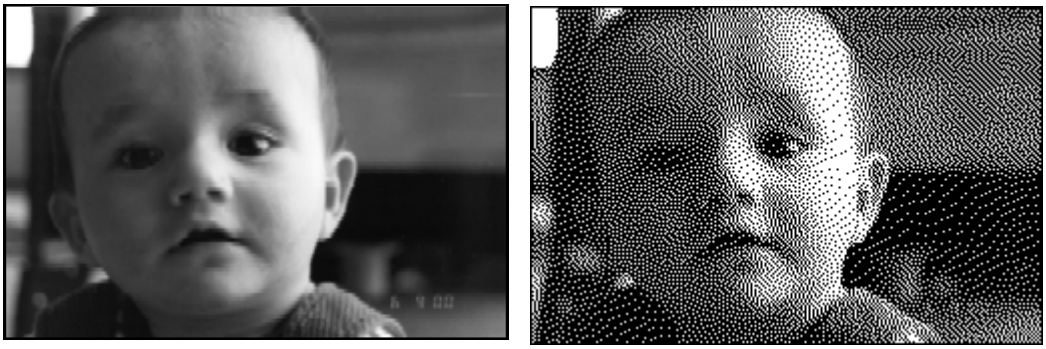


Figure 4: Original image (left), Floyd-Steinberg dithered image (right)

pixel to the south-east of P is incremented by E/16. Once neighboring pixels have been adjusted, the thresholding operation continues in left-right-top-bottom fashion. Figure 4 gives an example of using Floyd-Steinberg diffusion to dither a gray-scale image.

Histogram Computation

The histogram of an image is a count of the number of times each pixel value actually occurs in an image. The code fragment listed below illustrates how to compute the histogram of a gray-scale image. The histogram counts are indexed by pixel values and hence, histogram[V] contains the number of times that pixel value V occurs in the image. Since each pixel value V is itself stored in an array, this serves as an excellent practical example of the power of indirection.

```
int[][] pixels = image.getPixels2D(EasyBufferedImage.GRAY);
int[] histogram = new int[256];
for(int i=0; i<pixels.length; i++) {
    for(int j=0; j<pixels[0].length; j++) {
      histogram[pixels[i][j]]++;
    }
}
```

Tree Traversal

Image compression techniques often sacrifice image quality for decreased storage requirements. Hierarchical compression is one such method for compressing image data. Given a rectangular region R of an image, the entire region can be approximated by the average value of its pixels if most of the pixels in the region are close to the average. If region R does not contain pixels that are close to the average, then the region is decomposed into four sub-regions by slicing down the center and across the middle and then representing each sub-region in a similar fashion. The result is a quad–tree decomposition of the image data where entire regions of the image are represented with a single pixel. Developing code for this project is appropriate for teaching tree traversal and construction at the CS2 level. A more formal algorithmic description is given below.

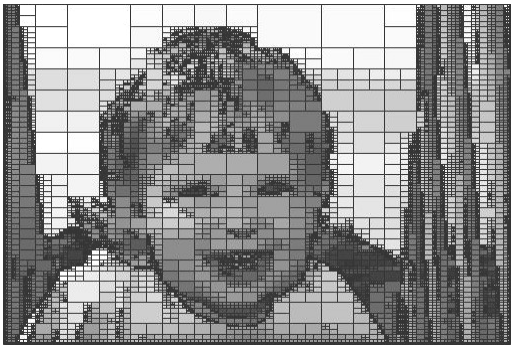
Note that the error of region R is typically the root-mean-squared error and that the tolerance value dictates the amount of acceptable deviation within a region. High tolerance values correspond to low quality images with high compression ratios.

```
algorithm TreeNode compress(Region region, Image image, double tolerance)
  avg = AveragePixelValue(image, region)
  error = RootMeanSquareError(image, region, avg)
  if error > tolerance then
        TreeNode t = new TreeNode()
        for each subregion Rs of region
           t.addChild(compress(Rs, image, tolerance))
        return t
  else
        return new TreeNode(region, avg)
```

Figure 5 shows an example of using hierarchical compression on a gray-scale portrait of a child. The tolerance value was set at 10 and the images were generated using the QuadViewer application that is distributed with the EasyBufferedImage package.

(a)

(b)



(c)

Figure 5. Example of hierarchical image compression. Original Image (a). Hierarchically compressed image (b). Hierarchically compressed image with grid lines indicating sub-regions (c).

Conclusion

This paper has described a Java package that gives introductory-level and advanced programmers access to image processing theory and techniques. Suggestions were given for incorporating this package into a typical undergraduate CS1 and CS2 level curriculum. In addition, the package is freely available by request to hunt@mail.uwlax.edu.

References

[1] Efford, N. *Digital Image Processing in Java*, Addison-Wesley Longman Publishing Co., Inc, Boston, Ma, 2000

[2] Gonzalez, R., and Woods, R. *Digital Image Processing*, Addison-Wesley Longman Publishing Co., Inc, Boston, Ma, 1992

[3] *Java2 Platform Standard Edition 1.4 Release 2002*. Sun Microsystems. http://java.sun.com/j2se.

[4] Marks, J., Freeman, W., and Leitner, H. *Teaching applied computing without programming: a case-based introductory course for general education* in Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education 2001, Charlotte, North Carolina, United States.