Multidimensional Arrays and Image Manipulation

Eric Roberts
CS 106A
February 17, 2010

In Our Last Episode . . .

• In last week’s section, you were asked to assume the existence of a method
  
  ```java
double[] readScoresArray(String filename)
```

that reads score data from the specified file, one value per line, and returns an array of doubles containing those values.

• Before we start on today’s topic, I want to finish this exercise so that you have another code example for working with files.

Multidimensional Arrays

• Because the elements of an array can be of any Java type, those elements can themselves be arrays. Arrays of arrays are called multidimensional arrays.

• In Java, you can create a multidimensional array by using multiple brackets in both the type and the initialization parts of the declaration. For example, you can create array space for a 3 x 3 tic-tac-toe board using the following declaration:
  
  ```java
  char[][] board = new char[3][3];
  ``

• This declaration creates a two-dimensional array of characters that is organized like this:

  ```text
  board[0][0]  board[0][1]  board[0][2]
  board[1][0]  board[1][1]  board[1][2]
  ```

Multidimensional Arrays and Images

• One of the best examples of multidimensional arrays is a Java image, which is logically a two-dimensional array of pixels.

• Consider, for example, the logo for the Java Task Force at the top right. That logo is actually an array of pixels as shown in the expanded diagram at the bottom.

• The `GImage` class allows you to convert the data for the image into a two-dimensional array of pixel values. Once you have this array, you can work with the data to change the image.

Pixel Arrays

• If you have a `GImage` object, you can obtain the underlying pixel array by calling the `getPixelArray`, which returns a two-dimensional array of type `int`.

• For example, if you wanted to get the pixels from the image file `JTFLogo.gif`, you could do so with the following code:

  ```java
  GImage logo = new GImage("JTFLogo.gif");
  int[][] pixels = logo.getPixelArray();
  ```

• The first subscript in a pixel array selects a row in the image, beginning at the top. The height of the image is therefore given by the expression `pixels.length`.

• The second subscript in a pixel array selects an individual pixel within a row, beginning at the left. You can use the expression `pixels[r].length` to determine the width of the image.

Pixel Values

• Each individual element in a pixel array is an `int` in which the 32 bits are interpreted as follows:

  ```text
<table>
<thead>
<tr>
<th>transparency</th>
<th>red</th>
<th>green</th>
<th>blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
  ```

• The first byte of the pixel value specifies the transparency of the color, which is described in more detail on the next slide.

• The next three bytes indicate the amount of red, green, and blue in the pixel, in which each value varies from 0 to 255. Together, these three bytes form the `RGB` value of the color, which is typically expressed using six hexadecimal digits. The color in the example has the `RGB` value `0x996633`, which is a light brown.
Transparency

• The first byte of the pixel value specifies the transparency of the color, which indicates how much of the background shows through. This value is often denoted using the Greek letter alpha (\(\alpha\)).

• Transparency values vary from 0 to 255. The value 0 is used to indicate a completely transparent color in which only the background appears. The value 255 indicates an opaque color that completely obscures the background. The standard color constants all have alpha values of 255.

• Fully transparent colors are particularly useful in images, because they make it possible to display images that do not have rectangular outlines. For example, if the gray pixels in the corners of the JTFLogo.gif image have an alpha value of 0, the background will show through those parts of the logo.

Image Manipulation

• You can use the facilities of the GImage class to manipulate images by executing the following steps:
  1. Read an existing image from a file into a GImage object.
  2. Call GImageArray to get the pixels.
  3. Write the code to manipulate the pixel values in the array.
  4. Call the GImage constructor to create a new image.

• The program on the next slide shows how you can apply this technique to flip an image vertically. The general strategy for inverting the image is simply to reverse the elements of the pixel array, using the same technique as the reverseArray method on an earlier slide.

Selecting Color Components

• If you want to work with the colors of individual pixels inside a pixel array, you can adopt either of two strategies:
  – You can use the bitwise operators described in the text to select or change individual bits in the pixel value.
  – You can use the static methods provided by the GImage class for that purpose.

• Although it is useful to remember that all information is stored as bits, there doesn’t seem to be much point in going into all the details, at least in CS 106A. We will therefore use the second strategy and employ the static methods getRed, getGreen, getBlue, getAlpha, and createRGBPixel.

• The definitions of these methods (along with the underlying implementations, which you are free to ignore) are shown on the next slide.

Static Methods in GImage

```java
/** Returns the alpha component from an RGB value. */
public static int getAlpha(int pixel) {
    return (pixel >> 24) & 0xFF;
}

/** Returns the red component from an RGB value. */
public static int getRed(int pixel) {
    return (pixel >> 16) & 0xFF;
}

/** Returns the green component from an RGB value. */
public static int getGreen(int pixel) {
    return (pixel >> 8) & 0xFF;
}

/** Returns the blue component from an RGB value. */
public static int getBlue(int pixel) {
    return pixel & 0xFF;
}

/** Creates an opaque pixel value from the color components */
public static int createRGBPixel(int r, int g, int b) {
    return createRGBPixel(r, g, b, 0xFF);
}

/** Creates a pixel value from the color components, including alpha */
public static int createRGBPixel(int r, int g, int b, int alpha) {
    return (alpha << 24) | (r << 16) | (g << 8) | b;
}
```

Creating a Grayscale Image

• As an illustration of how to use the bitwise operators to manipulate colors in an image, the text implements a method called createGrayscaleImage that converts a color image into a black-and-white image, as shown in the sample run at the bottom of this slide.

• The code to implement this method appears on the next slide.
The `createGrayscaleImage` Method

```java
private GImage createGrayscaleImage(GImage image) {
    int[][] array = image.getPixelArray();
    int height = array.length;
    for (int i = 0; i < height; i++) {
        int width = array[i].length;
        for (int j = 0; j < width; j++) {
            int pixel = array[i][j];
            int r = GImage.getRed(pixel);
            int g = GImage.getGreen(pixel);
            int b = GImage.getBlue(pixel);
            int xx = computeLuminosity(r, g, b);
            array[i][j] = GImage.createRGBPixel(xx, xx, xx);
        }
    }
    return new GImage(array);
}
```

/* Calculates the luminosity of a pixel using the NTSC formula */
private int computeLuminosity(int r, int g, int b) {
    return GMath.round(0.299 * r + 0.587 * g + 0.114 * b);
}

Steganography

"The more I reflected upon the daring, dashing, and discriminating ingenuity of D--- . . . the more satisfied I became that, to conceal this letter, the Minister had resorted to the comprehensive and sagacious expedient of not attempting to conceal it at all."
— Edgar Allan Poe, "The Purloined Letter," 1845

- In his famous short story, Poe argues that it is often best to hide things in plain sight. That idea is sometimes used in cryptography to encode messages that—even though they are completely public—no one knows that the message is there. This technique is called **steganography**.
- One approach is to hide a message in the least significant bits of an image, where that change isn’t visible to the human eye. For example, you can store an 8-bit ASCII character like this: