Eric Roberts
CS 106A

Handout #42
May 11, 2012

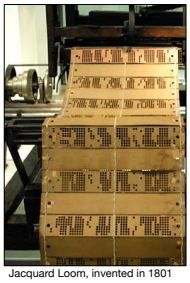
Image Manipulation

Image Manipulation

Eric Roberts
CS 106A
May 11, 2012

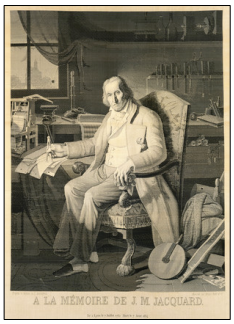
Pictures and Pixels

- The focus of today's lecture is on how to represent images using two-dimensional arrays of tiny dots called ***pixels***.
- The idea of representing an image as a two-dimensional array of dots is much older than modern computing and has several antecedents:
  - Halftone pictures in newspapers
  - Pointillist art
  - Mechanical weaving
- Of these examples, the most amazing connections to modern computing are in weaving, which contributed substantially to early computing.



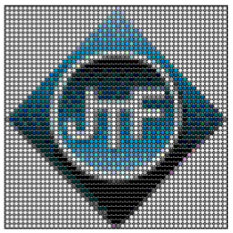
Jacquard Loom, invented in 1801

Weavings on the Jacquard Loom



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

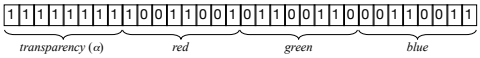
- If you have a **`GImage`** object, you can obtain the underlying pixel array by calling the **`getPixelArray`**, which returns a two-dimensional array of type **`int`**.
- For example, if you wanted to get the pixels from the image file **`JTFLogo.png`**, you could do so with the following code:

```
GImage logo = new GImage("JTFLogo.png");
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[0].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:



- The first byte of the pixel value specifies the ***transparency*** of the color, which is described in more detail on a later 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, as in the following examples:

Combining Colors of Light

Transparency

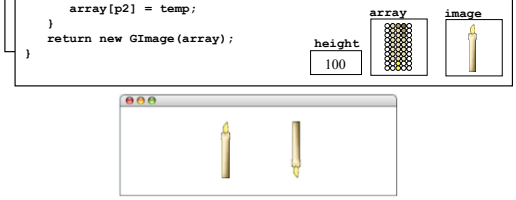
- The first byte of the pixel value specifies the transparency of the color, which indicates how much of the background shows through. This value is often denoted using the Greek letter alpha (α).
- Transparency values vary from 0 to 255. The value 0 is used to indicate a completely transparent color 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.png`** 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 **`getPixelArray`** 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.

The `FlipVertical` Program

```
public void run() {
private GImage flipVertical(GImage image) {
   int[][] array = image.getPixelArray();
   int height = array.length;
   for (int p1 = 0; p1 < height / 2; p1++) {
      int p2 = height - p1 - 1;
      int[] temp = array[p1];
      array[p1] = array[p2];
      array[p2] = temp;
   }
   return new GImage(array);
}
```

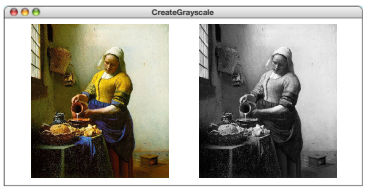


Selecting Color Components

- 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.

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

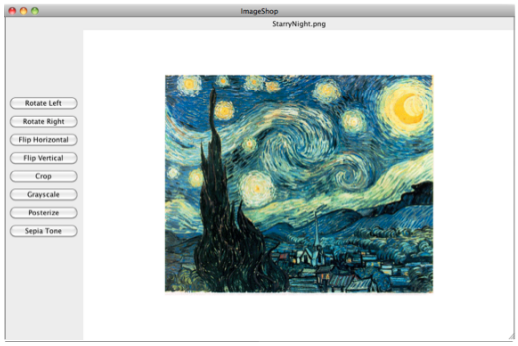
```
/* Creates a grayscale version of the original image */
private GImage createGrayscaleImage(GImage image) {
   int[][] array = image.getPixelArray();
   int height = array.length;
   int width = array[0].length;
   for (int i = 0; i < height; i++) {
      for (int j = 0; j < width; j++) {
         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);
}
```

Exercise: Creating an Image

How might you write a program to create the image used to illustrate combining colors:

The ImageShop Application



Exercise: Posterize

Write a **`posterize`** method that takes a **`GImage`** and returns a new **`GImage`** in which every red, blue, and green component is either 0 or 255, as illustrated in the following conversion: