YOUR EARLY ASSIGNMENT HELP | ASSIGNMENT FIVE

YEAH! Hours: ImageShop

Garrick Fernandez <gfaerr@stanford.edu>
Robbie Jones <rmjones@stanford.edu>
Due Mon. Feb. 26th at 11:00am PST

Questions or feedback? tinyurl.com/yeah-hours
Loaded image VanGogh-StarryNight.png.
Why ImageShop, why?

An introduction to bigger programs! Think about the given interface and how your code communicates with it.

Boost familiarity with GImage and its methods.

Practice using 2D arrays and manipulating them.

Translating between the pixels in a GImage and a 2D array of integers!

tinyurl.com/yeah-hours
Skills For Approaching Problems

- Quickly *intuit* what a problem is asking for.
- Learn how to **draft and design** good code.
- Pull bits and snippets from our **coding toolbox**.
- Anticipate **edge cases** and **test** for errors.

[ tinyurl.com/yeah-hours ]
A Primer: ImageShop

their code in ImageShopProgram.java

our code (!) in ImageShopAlgorithms.java

How do these two interact?
A Primer: ImageShop

photo files

meme1.jpg
meme2.jpg
...

image algorithms

CLIENT >

user input
(click button)

< DEVELOPER

tinyurl.com/yeah-hours
A Primer: ImageShop

photo files

meme1.jpg
meme2.jpg
...

image algorithms

user input
(click button)

CLIENT >

< DEVELOPER

display output

tinyurl.com/yeah-hours
A Primer: ImageShop

We’re in charge of the algorithms!

CLIENT >

The interface is covered! Yay!

< DEVELOPER

Photo files

User input (click button)

Display output

We’re in charge of the algorithms!
Breaking Up the Problem

How do we break up the problem into approachable milestones?

tinyurl.com/yeah-hours
Breaking Up the Problem

First, let’s figure out how to operate on 2D arrays and figure out what that means for our image.

Then, let’s understand how GIImages work and how to use them and convert between them and 2D arrays.

Last, let’s fill out our algorithms one by one!
1D and 2D Arrays

Slides courtesy of Nick Troccoli. Thanks Nick :)
Arrays: the basics

- An array stores an **ordered sequence** of multiple objects.
  - Can access objects by index using [].
- All stored objects have the same type, which you choose.
- Can store *any* type, even primitive types (**int**, **boolean**, **double**, etc.).
- Fixed size; cannot grow or shrink once created.
Arrays: how to make

• To create a new array, specify the type of the array and the size:

\[
\text{Type[]} \ arr = \text{new Type[size]}
\]

ex:

\[
\text{int[]} \ numList = \text{new int[6]};
\]

• All elements are initialized to the type’s default value (0 for numeric types, false for booleans, null for objects).
Arrays: accessing

• To access an element of the array, use the square brackets to choose the index:

\[ arr[index] \]

ex. (using the above array):

\[ \text{int num} = \text{numList}[2]; // 314 \]
Arrays: accessing

To loop over every element in an array, just use a `for` loop:

```java
for (int i = 0; i < arr.length; i++) {
    Type elem = arr[i];
    // use elem...
}
```
Arrays: accessing

```
// prints out every element in numList
for (int i = 0; i < numList.length; i++) {
    int elem = numList[i];
    println(elem);
}
```
Arrays: persistence

• Like with objects, if an array is passed as a parameter to a method and that method changes its elements, those changes will persist when that method is done.

```java
private void doSomething(int[] nums) {
    nums[0] = 2;
}

// elsewhere...
int[] arr = new int[5];
doSomething(arr); // now arr[0] = 2!
```
2D Arrays are grids!

- You can think of a 2D array as a grid. 
  \( \text{arr[row][col]} \) selects the element in the grid at position (row, col).

```
int num = numGrid[1][2]; // 12
```

- You can also think of a 2D array as an array of arrays. 
  \( \text{arr[row]} \) selects the 1D array consisting of the columns in row row.

```
int[] secondRow = numGrid[1];
int num = secondRow[2]; // 12
```
2D Arrays

• To create a new 2D array, specify the type using *two* pairs of brackets, and specify both the width and height:

\[
\text{Type}[][] \ arr = \text{new \ Type}[\text{rows}][\text{cols}];
\]

ex:

\[
\text{int}[][] \ \text{numGrid} = \text{new \ int}[2][3];
\]
2D Arrays

• The height of a 2D array is the length of the array itself. The width is the length of any of the element arrays.

ex.

```java
int gridHeight = numGrid.length; // 2
int gridWidth = numGrid[0].length; // 3
```
2D Arrays

• You can iterate over a 2D array just like with a 1D array, but you need 2 loops instead of 1.

Ex. What does the grid below look like?

```java
int[][] arr = new int[4][5];
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[0].length; col++) {
        arr[row][col] = row + col;
    }
}
```
```java
int[][] arr = new int[4][5];
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[0].length; col++) {
        arr[row][col] = row + col;
    }
}
```
int[][] arr = new int[4][5];
for (int row = 0; row < arr.length; row++) {
    for (int col = 0; col < arr[0].length; col++) {
        arr[row][col] = row + col;
    }
}

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,0</td>
<td>0,1</td>
<td>0,2</td>
<td>0,3</td>
<td>0,4</td>
</tr>
<tr>
<td>1</td>
<td>1,0</td>
<td>1,1</td>
<td>1,2</td>
<td>1,3</td>
<td>1,4</td>
</tr>
<tr>
<td>2</td>
<td>2,0</td>
<td>2,1</td>
<td>2,2</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>3</td>
<td>3,0</td>
<td>..</td>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>
GImages are 2D arrays of pixels!

Think of each pixel as an int representing the RGB value at that pixel

Thousands of pixels/ints comprise a single image!

#ff0000 means:
(255, 0, 0)
Red: 255
Green: 0
Blue: 0
How to use a pixel

A pixel contains all three integers (red, green, blue) as a single int. Sometimes, we need to be able to extract the individual values from it!

```java
// Extract the red, green, and blue values individually
int red = GImage.getRed(pixel);
int green = GImage.getGreen(pixel);
int blue = GImage.getBlue(pixel);
red = 0; // Get rid of the red, for example
// Create a new pixel
int pixel = GImage.createRGBPixel(red, green, blue);
```

Notice that these methods are called on the GImage class itself, not on an object of the class!
Breaking Up the Problem

First, let’s figure out how to operate on 2D arrays and figure out what that means for our image.

Then, let’s understand how GImages work and how to use them and convert between them and 2D arrays.

Last, let’s fill out our algorithms one by one!
From GImages to 2D arrays

If GImages are 2D arrays of pixels/ints, how do we convert from one into the other?

There are some helpful methods to convert between the two!

```java
// Turn a GImage into a 2D array of pixels
int[][][] pixels = img.getPixelArray();

// Create a new img given a 2D array of pixels
GImage img = new GImage(pixels);
```
Breaking Up the Problem

First, let’s figure out how to operate on 2D arrays and figure out what that means for our image.

Then, let’s understand how GImages work and how to use them and convert between them and 2D arrays.

Last, let’s fill out our algorithms one by one!
Negative

\[ K \rightarrow 255 - K \]

**smiley-face.png** → **smiley-face-negative.png**

tinyurl.com/yeah-hours
Negative

The basic ideas used here will be prevalent throughout the assignment

- Looping through entire image
- Grabbing (and possibly manipulating) pixel at each location
- Returning new image
Lights, Camera, **Green Screen**!

- **Main idea:** make all green pixels transparent (alpha = 0)
- **Problem:** not all pixels are *exactly* green
- **Solution:** treat a pixel as green if its green color component is at least 2x as large as the maximum of its red and blue components.
Lights, Camera, Green Screen!

Helpful methods (again!) from the book / Java documentation:

```java
int red = GImage.getRed(PIXEL);
int green = GImage.getGreen(PIXEL);
int blue = GImage.getBlue(PIXEL);
int pixel = GImage.createRGBPixel(RED, GREEN, BLUE, ALPHA);
int maxNum = Math.max(NUM1, NUM2);
```
Flip Horizontal, Rotate Left, and Rotate Right
Flip Horizontal, Rotate Left, Rotate Right

• Tip: draw out pictures of pixel movement

Ex. Rotate Left:

Notice: pixel’s **column #** in new image is its **row #** in original image. How do we calculate the new **row #**?
(Lost in) Translation

smiley-face.png  \(dx=100, dy=50\)  smiley-face-translate-100-50.png

smiley-face.png  \(dx=-45, dy=-91\)  smiley-face-translate-negative-45-91.png
(Lost in) Translation

Big idea: pixels need to “roll over” by a certain \((dx, dy)\) to a new location

Problem: pixels can’t go off the edge of the image

Question: how do we make pixels wrap around to the other side of the image?
Simpler Example

before translate by \((dx=2, \ dy=-1)\)  
\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & A & B & C & D & E & F \\
1 & G & H & I & J & K & L \\
2 & M & N & O & P & Q & R \\
3 & S & T & U & V & W & X \\
\end{array}
\]

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
0 & K & L & G & H & I & J \\
1 & Q & R & M & N & O & P \\
2 & W & X & S & T & U & V \\
3 & E & F & A & B & C & D \\
\end{array}
\]  

NOTE: \(dx\) can be greater than width of image!  
(\text{and similar for} \(dy\))

6 \rightarrow 0
7 \rightarrow 1

Notice the wraparound!
There are other really cool types of blurs, including Gaussian blur!
Blur

For every pixel p at position \((r,c)\), we take the average of all the RGB values over the nine positions from \(r-1\) to \(r+1\) and \(c-1\) to \(c+1\).

Can we imagine the pseudocode?

[tinyurl.com/yeah-hours](tinyurl.com/yeah-hours)
Blur

For every pixel $p$ at position $(r,c)$, we take the average of all the RGB values over the nine positions from $r-1$ to $r+1$ and $c-1$ to $c+1$.

Can we imagine the pseudocode?

```plaintext
// for each pixel in the image:
//     for each neighbor in range, including that pixel
//         tally up the total RGB
//     the new pixel is the avg. RGB (divide by # neighbors)
```
Blur

For every pixel $p$ at position $(r,c)$, we take the average of all the RGB values over the nine positions from $r-1$ to $r+1$ and $c-1$ to $c+1$.

Can we imagine the pseudocode?

```plaintext
// for each pixel in the image:
//     for each neighbor in range, including that pixel
//         tally up the total RGB
//     the new pixel is the avg. RGB (divide by # neighbors)
```

What's this gonna look like?
Images, Arrays, And Double-For loops

Convert our image to an int[][][], and use a double for loop to go over each pixel.

```java
// converting a GImage to an array of ints
int[][][] original = img.getPixelArray();

// some helpful array bits...you could also check the docs!
int rows = original.length;
int cols = original[0].length;
// If a row number is in bounds, it will be 0 to rows-1...
```
Equalize
Histogram Equalization
Histogram Equalization

Big idea: the colors here are all shades of grey that are “close” to one another.

Instead, we want to spread them out, so there are more “extreme” values or greater contrast.
Histogram Equalization

• Main idea: spread out *luminosity values* as much as possible
• **Step 1:** compute the luminosity histogram.
Histogram Equalization

- Main idea: spread out *luminosity values* as much as possible
- **Step 1**: compute the luminosity histogram.

Notice how the distribution of colors is “bunched up”. Intuitively, for the image to have more contrast, we want to spread the distribution out!

No black or white! Just grey
Histogram Equalization

• **Step 1:** compute the luminosity histogram.

\[
\text{histogram}[i] = \text{# of pixels with luminosity } i
\]

```c
int luminosity = computeLuminosity(red, green, blue) // helpful!
```
Histogram Equalization

• **Step 2**: use the luminosity histogram to compute the cumulative luminosity histogram
Histogram Equalization

• **Step 2:** use the luminosity histogram to compute the cumulative luminosity histogram

How will this help spread out the distribution? Notice how pixels appear “late” in the cumulative histogram and the number of pixels rises sharply before the end.
Step 2: use the luminosity histogram to compute the cumulative luminosity histogram

How will this help spread out the distribution? Notice how pixels appear “late” in the cumulative histogram and the number of pixels rises sharply before the end.

This is because the pixels were clumped together in the first place!

We want to spread the “buildup” of pixels more over all the values.
**Histogram Equalization**

- **Step 2**: use the luminosity histogram to compute the cumulative luminosity histogram.

\[
\text{cumulativeHistogram}[i] = \text{# of pixels with luminosity} \leq i
\]

Careful not to confuse this with the previous histogram!
Histogram Equalization

- **Step 3**: use the cumulative luminosity histogram to compute new luminosities for each pixel.

\[
\text{new pixel luminosity} = \frac{255 \times \text{cumulative histogram}[L]}{\text{total \# pixels}}
\]

where \( L \) is the original luminosity for the pixel. Change each pixel to have red, green, and blue values equal to its new luminosity.
Histogram Equalization

- **Step 3:** use the cumulative luminosity histogram to compute a new luminosity for each pixel given its old luminosity.

<table>
<thead>
<tr>
<th>Old L</th>
<th>New L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>≈ 16</td>
</tr>
<tr>
<td>1</td>
<td>≈ 110</td>
</tr>
<tr>
<td>2</td>
<td>≈ 251</td>
</tr>
<tr>
<td>3</td>
<td>255</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>255</td>
<td>255</td>
</tr>
</tbody>
</table>
How Does this help?

Say there are two cumulative pixels here...
Histogram Equalization

\[ 255 \times \frac{\text{# pixels with luminosity} \leq \text{this pixel's luminosity}}{\text{# total pixels}} \]

This will be 255 times a number close to 0.

Say there are two cumulative pixels here...
Histogram Equalization

255 \times \frac{\text{# pixels with luminosity} \leq \text{this pixel’s luminosity}}{\text{# total pixels}}

Now the new pixel will be darker to compensate!

Say there are two cumulative pixels here...

This will be a small number.
Histogram Equalization

Say there are almost all the cumulative pixels in the image here...
Histogram Equalization

\[
255 \times \frac{\text{# pixels with luminosity} \leq \text{this pixel's luminosity}}{\text{# total pixels}}
\]

Then we have 255 times a number close to 1!

Say there are almost all the cumulative pixels in the image here...
Histogram Equalization

255 \times \frac{\text{# pixels with luminosity} \leq \text{this pixel's luminosity}}{\text{# total pixels}}

Say there are almost all the cumulative pixels in the image here...

Now the new pixel will be lighter (close to 255) to compensate!

This will be 255 times a number close to 1!
Histogram Equalization

Summary:
1. Compute the histogram for the original image.
2. Compute the cumulative histogram from the image histogram.
3. Change each original pixel to have red, green, and blue values equal to its new luminosity from the given formula.

Hint: decompose into helper methods! (e.g. computing the histogram and computing the cumulative histogram).
Tips and Tricks

Read read read the documentation! It’s on the course website.

Check the given functions in the handout. You may have some of your work cut out for you! Read the handout too (no ivars allowed!)

Think about algorithms conceptually, and then move up to pseudocode, and then implement them for real! Then, most importantly, test them!

The Compare To Image option in the program will be great for testing.

Run the demo JAR from the course website!

tinyurl.com/yeah-hours
Documentation is your friend. As with any good friend, use ‘em.

tinyurl.com/yeah-hours
Tips and Tricks: The Pixel Problem

Back to blur...do we see any problems with it?

```cpp
// for each pixel in the image:
//   for each neighbor in range, including that pixel
//       tally up the total RGB
//   the new pixel is the avg. RGB (divide by # neighbors)
```

tinyurl.com/yeah-hours
Hmmm...

// for each pixel in the image:
// for each neighbor in range, including that pixel
// tally up the total RGB
// the new pixel is the avg. RGB (divide by #

Let’s look at this step as it repeats.
Hmmm...

We blur over the pixels in range of this pixel p1...

// for each pixel in the image:
//   for each neighbor in range, including that pixel
//     tally up the total RGB
//   the new pixel is the avg. RGB (divide by #...
Hmmm...

// for each pixel in the image:
// for each neighbor in range, including that pixel
// tally up the total RGB
// the new pixel is the avg. RGB (divide by #

We move to the next pixel p2...

Let's look at this step as it repeats.
Hmmm...

// for each pixel in the image:
//   for each neighbor in range, including that pixel
//     tally up the total RGB
//   the new pixel is the avg. RGB (divide by #

Oops! We’re using a blurred pixel to calculate the next pixel’s value.

Let’s look at this step as it repeats.
Tips and Tricks: The Pixel Problem

Back to blur...there’s something about it!

How do we put the blurred pixel p prime in p’s place without overwriting p? We don’t want to select p again and blur an already blurred pixel...

```// for each pixel in the image:
// for each neighbor in range, including that pixel
// tally up the total RGB
// the new pixel is the avg. RGB (divide by # neighbors)```
Tips and Tricks: Temps

Intuition: to avoid double-scattering, we can create a temporary int[][][] to map our new pixels to.

```c
// for each pixel in the image:
//     for each neighbor in range, including that pixel
//         tally up the total RGB
//     the new pixel is the avg. RGB (divide by #
```

Let's look at this step as it repeats.
Tips and Tricks: Temps

Temporary arrays are useful when you don’t want data in a new image you’re making to bleed over & influence said image!

// for each pixel in the image:
//   for each neighbor in range, including
//     tally up the total RGB
//     the new pixel is the avg. RGB (divide by #

Nothing in temp is being used to create the new image.

Let’s look at this step as it repeats.

tinyurl.com/yeah-hours
Good Luck!

tinyurl.com/yeah-hours