

ASSIGNMENT OVERVIEW

In this assignment you'll be writing a series of small programs that take a digital image, examine the pixels that make up that image one by one, and alter those pixels based on a series of rules to produce different kinds of new images.

This assignment is worth 50 points and should be uploaded to the course server by 23:59:59 on the date given in class.

BACKGROUND

Digital images are composed of *picture elements*, or "pixels," arranged in a series of columns and rows. In a high-resolution image displayed on a high-resolution monitor, you may have some difficulty even seeing the image as a series of individual points of light. As the resolution decreases, the pixels become more distinct, and the image itself becomes less so.



You may be familiar with the idea of accessing data in a series of columns and rows by using *nested loops* to go through the data. In this assignment you'll take an image and go through it pixel-by-pixel, across all the rows and columns of the photo, and **get** information about the color of that pixel, and (if we're editing the photo) **set** the color of the pixel based on rules we'll write into our code.

ASSIGNMENT SPECIFICATION

Write programs as indicated that will import an image and process it accordingly.

1. **ImageNormal.pyde**
This Processing program takes an image, imports it for use in the program, and then displays it on screen.
2. **ImageMagenta.pyde**
Takes an image and removes the green hues from each pixel before displaying it again.
3. **ImageDarker.pyde**
Takes an image and darkens the image based on a system that you develop.
4. **ImageGray.pyde**
Takes the red, green, and blue values for a pixel and uses them to create an average (gray) color for that pixel.
5. **ImageColorize.pyde**
Takes an image and makes one or two of the color channels (red, green, and blue) *more* of that color (red, green, and blue).
6. **ImageBlurry.pyde**
Takes an image and makes the photo more blurry.
7. **ImageSharpen.pyde**
Takes an image and tries to sharpen the image by finding edges.

DELIVERABLES

A zipped folder, **PhotoProcessing.zip**, containing the Processing program files as specified above along with the original and processed image files used with each of your programs.

To submit your assignment for grading, copy your file to your directory in **/home/studentID/forInstructor/** at *crashwhite.polytechnic.org* before the deadline given in class.

ASSIGNMENT NOTES

- Check the *Getting Started* section below for specific discussion of Processing syntax that will be useful to you in writing your programs.
- Check the *Getting Started* section below also for specific reference files that demonstrate specific strategies that will be useful to you in writing your programs:
 - Using nested loops with get and set
 - Using a 1-dimensional array to manipulate a 2-dimensional image
 - Creating a new image from an original and saving it to your computer
- The documentation for Processing provides additional documentation and examples on how you can work with images. See the Processing tutorial at <https://processing.org/tutorials/pixels/> for additional information as needed.

GETTING STARTED

These three examples will introduce you to some of the basic syntax that you can use in manipulating digital images.

1. Basic Digital Image Analysis

The general strategy when analyzing a photo is that one sets up a nested loop to run through the entire image, pixel by pixel. In the body of those two loops, one typically uses a **get** command to get the color of a pixel from the image, and then uses a **set** command to alter the color of that pixel based on whatever function is being performed.

Take a look at the commented code snippet here to identify how this process works.

```
"""
* ImageProcessingBasics is a demonstration of using the Processing to process
photo images.
* In this program, we are using Processing in "static" mode, without the setup()
and draw()
* methods.
* @author Richard White
* @version 2017-08-24
"""

img = loadImage("periodE.jpg")      # load an image from the drive into a reference
                                    # This image needs to be located in the same
                                    # directory as this program.
Size(1600, 916)                     # Establish size of window (same size as image)
image(img, 0, 0)                     # displays an image at upperleft 0,0

for row in range(height):
    for col in range(width):
        # Here's the classic way to get a pixel's color
        c = img.get(col,row)         # Gets the color of the given pixel
        r = red(c)                   # Pulls out the red value (0-255)
        g = green(c)                 # Pulls out the green value (0-255)
        b = blue(c)                  # Pulls out the blue value (0-255)
        # print("This pixel has colors: " + str(r) + ", " + str(g) + ", " + str(b))

        '''
        If you want to change the color of the pixel, you use the .set() method.
        Take the red, green, and blue values and alter them according to
        whatever function you are trying to implement, and then "set" the
        specified pixel to the new color.
        '''
        newR = r
        newG = g
        newB = b                     # Note that we're setting blue value to 0.
                                    # Can you predict what this image is going
                                    # to look like?

        img.set(col,row,color(newR, newG, newB))

# Now that the .set() method has altered pixels in the image, we need to
# display this new version of the image on the screen.
image(img, 0, 0)                     # displays the edited image at upper left 0,0
                                    # (used with first strategy)
img.save("altered.png")
```

You should have a very clear understanding of how the nested loops address each pixel in the image: **row** is set to 0, then **col** is set to 0, and the pixel at (0,0) is examined. The **col** loop is incremented to 1, and pixel (1,0) is examined, then (2,0), (3,0), and so on until all of the columns have been traversed. The **col** loop is finished at that point, so the **row** loop is incremented to 1, the **col** loop starts again, and we begin working with pixels (1,0), (1,1), (1,2), and so on.

2. A Two-Dimensional Data Structure in One Dimension

The `set` command in Processing allows one to access a pixel in an image by referring to its *column* and *row*, or *x* and *y* "coordinates." You might be interested to know that this abstract concept of a two-dimensional grid of values is actually stored in the computer as a one-dimensional array.

So, this grid:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14

is actually stored in a sequence of memory locations in the computer like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----

This raises an interesting question, though. How does the 2-d cell at row 1, column 2—the value of 7—correspond to cell location #7 in the one-d array? How can we, given the *x*- and *y*-coordinates from the 2-d table, arrive at the correct index in the one-d sequence?

If we can figure out a way to convert the *x-y* location to the one-d index, we can use that with our pixels and save a lot of time in running our programs. It turns out that accessing using `get(x, y)` and `set(x, y)` is much less efficient than just using the single array of pixels, conveniently called `pixels[]`.

Take a look at the program on the next page, and study carefully the comments there.

```

"""
* ImageProcessingBlueBlocker is a demonstration of using the 1-d array "pixels."
* @author Richard White
* @version 2017-08-24
"""

img = loadImage("periodE.jpg")      # load an image from the drive into a reference
size(1600, 916)                     # Establish size of window (same size as image)

image(img, 0, 0)                     # displays an image at upperleft 0,0
loadPixels()                         # sets up access to pixels in pixels array
                                    # You have to load pixels before you can use
                                    # the "pixels" array.

'''
Note that pixel array is a single-dimensional array of the
two-dimensional image:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] =

    0, 1, 2, 3, 4
    5, 6, 7, 8, 9
    10,11,12,13,14

So index = (row * width) + column
'''

for row in range(height):
    for col in range(width):
        '''
        From the documentation:
        Getting the color of a single pixel with get(x, y) is easy,
        but not as fast as grabbing the data directly from pixels[].
        The equivalent statement to get(x, y) using pixels[] is
        pixels[y*width+x]. See the reference for pixels[] for more
        information.
        '''
        index = row * width + col      # identify the index for this pixel
        old_red = red(pixels[index])   # get the RGB colors of this pixel
        old_green = green(pixels[index])
        old_blue = blue(pixels[index])
        new_red = old_red               # alter them as desired
        new_green = old_green
        new_blue = 0

        # Now we reset the value of pixels[index] to a new color.
        pixels[index] = color(new_red, new_green, new_blue)

# After going through the entire picture, update the image on screen.
updatePixels()

```

3. Saving an Image

Updating an image on the screen is fine, but you may want to save an image to the disk. There are two ways to do this: saving an image that you've altered to a new name, or creating a new image based on the old one and saving that. Take a look at the code here to see how that works.

Notice that we're transitioning to using Processing's `setup()` and `draw()` functions. Note also that this is our first effort to display an image *before* and *after* editing.

```
"""
* ImageProcessingBrighten is a demo of using Processing to process photo images.
* This example demonstrates how to create a new, separate image based on
* an old image, as well as how to save an image onto disk.
"""

# global variables to be used both in functions
brightenFactor = 1.5
img = loadImage('periodE.jpg')
img2 = createImage(1600, 916, RGB) # creates an empty image

def setup():
    global img, img2
    size(1600, 916) # Establish size of window (same size as image)
    img = loadImage('periodE.jpg')
    image(img, 0, 0) # displays the image at upperleft 0,0
    noLoop() # execute draw() method only once (otherwise,
             # draw() method repeats infinitely, which is
             # desired for interactive programs

def delay(delayTime):
    time = millis()
    while(millis() - time <= delayTime):
        pass

def draw():
    global img, img2
    delay(2000) # delays the program for 2 seconds so that we
               # can see original image from setup() method

    img.loadPixels() # sets up access to pixels in pixels array
    img2.loadPixels()

    for row in range(height):
        for col in range(width):
            index = row * width + col
            r = red(img.pixels[index])
            g = green(img.pixels[index])
            b = blue(img.pixels[index])
            r = min(r * brightenFactor, 255) # increases red by factor, max 255
            g = min(g * brightenFactor, 255)
            b = min(b * brightenFactor, 255)
            img2.pixels[index] = color(r, g, b) # sets the color in the new image

    img2.updatePixels() # updates the pixels in the new image
    image(img2, 0, 0) # displays new image in the window

    img.save("originalclass.jpg") # saves the original version (under a new name)
    img2.save("newclass.jpg") # save the new version (under a new name)
    img2.save("newclass.jpg") # save the new version
```

QUESTIONS FOR YOU TO CONSIDER (NOT HAND IN)

1. How well does the "blur" program work? Are there other ways of producing a blur effect? What is a "Gaussian" blur?
2. How well does your "sharpen" program work?
3. Take a look at the video at https://www.youtube.com/watch?v=LbF_56SxrGk. (Enhance HD). Is it possible for a program to identify elements in a photo that aren't initially visible, as demonstrated in the video?