CSCI 1300 Notes on Image Processing

Acknowledgement: These notes were originally inspired by examples developed by Mark Guzdial of Georgia Tech (see http://coweb.cc.gatech.edu/csl/83)

##These notes are provided for your interest. They include questions to guide you on the way, if you want to write programs that do things to images. If you choose to use these notes for extra work in the class, you can include answers to these questions in your report of your extra work.

Representing Things Seen in the World

We are used to representing visible things in the world using *pictures*. This just means that many operations we might do on things in the world, like recognizing shapes or faces, can be done by looking at pictures instead.

We can obtain a number of advantages if we find a way of representing pictures using *computational* stuff, that is numbers, that we know we can manipulate in programs.

If a picture is represented in computational stuff,

- we can *transport* it to the other side of the world, amazingly quickly and cheaply.

- we can *store* it, in a tiny space, and at practically no cost, and *retrieve* it later.

- we can *automate* many changes to the picture, such as making it brighter or darker, or increasing or decreasing contrast.

These advantages, together with the fact that pictures represented in computational stuff can be captured by a purely electronic camera, with no processing of chemicals required, explains the popularity of digital photography.

Representing pictures with numbers: Pixels

Pictures can be approximated by dividing them into a large number of small, closely spaced dots, called *pixels* (short for *picture elements*). Human vision works in such a way that if the dots are small enough, we can't see them: we see only the shapes the dots make up. (This fact is the basis of many ways of representing pictures, including most forms of printing.)

Then we have to control the color and brightness of the dots. Another useful fact about human vision is that *all colors can be created by mixing just three colors*. While there are many combinations of three colors that will work, it's usual to use red, green, and blue. A common way to manage this mixing of colors is to break each pixel down into three even smaller dots, one red, one green, and one blue, and then vary how bright each of these smaller dots is.

The color screen on your computer displays pictures in just this way. A picture is made up of many pixels. For a color picture, each pixel is divided into three smaller dots, one red, one green, and one blue. The brightness of each of these smaller dots is controlled by a number between 0 and 255, with 0 giving minimum brightness, and 255 the maximum. These three numbers are called an *RGB triple*, after the initials of the colors. We'll call the three numbers the *red value*, the *green value*, and the *blue value* of the pixel. The general term we'll use for these three numbers is *color values*.

In our Python code, the three color values, r, g, and b, for a pixel are combined into one number, using this formula:

color=256*256*r + 256*g + b

This allows a single number to represent a pixel of any color.

This formula has the important property that, given the number for any color, one can calculate the values of r, g, and b that make up that color. For example, color%256 is b, for any color, and there are similar formulae for determining the r and g values. Note: functions that perform these calculations are included in the sample Python program that accompanies these notes.

Because pixels have to be small, we need a lot of them to make a picture. A small picture may require 100x100 pixels, or 10,000. A bigger picture may take 800x600 pixels, or 480,000. As you probably know, current digital cameras all take pictures much bigger than that.

You may wonder how specifying just the values for red, green, and blue can give us all the colors we need. How about yellow, for example? Isn't green a mixture of yellow and blue? And what about white? Remarkably, for reasons determined by the mechanisms of color vision in humans, mixtures of three colors are sufficient to produce the appearance of any color, and red, green, and blue are a workable set of three colors. Yellow is a combination of red and green, and green is NOT a mixture of yellow and blue. White consists of an equal mixture of all three colors. (Combining yellow and blue to make green works with *paint*, but not with *light;* to understand how this happens you can read about *subtractive color mixing* in Wikipedia.)

Here is a table showing the colors of pixels with various color values:

red value	green value	blue value	color
255	0	0	bright red
128	0	0	dull red
255	255	255	white
128	128	128	grey
0	0	0	black
255	255	0	bright yellow
255	0	255	???
0	255	255	???
255	200	200	pink
200	200	255	???

You can fill in the ??? in the table by working on one of the (purely optional) challenge questions below. Here are two *general rules*:

*any pixel whose three color values are equal will be "colorless", somewhere in the range from white to black.
*the overall brightness of a pixel is determined by the average value of its color values

You can read more about RGB, and its relationship to human vision, in Wikipedia.

Operations on pictures

We can find operations on collections of pixels that correspond to operations on pictures. Here are some examples:

If we want a brighter picture, we take the pixels that represent the original picture, and we multiply the color values in them by some number bigger than 1. The modified pixels represent a brighter version of the original picture.

If we want a picture that is redder, we take the pixels that represent the original picture, and we multiply only the red value in each pixel by a number greater than 1. The modified pixels represent a redder version of the original picture.

If we want a monochrome (black and white) version of a picture, we take the pixels in the original picture, and we replace all three color values by the average of the three original color values.

If we want a version of a picture with increased contrast, and we have the pixels that represent the original picture, we want to make the bright pixels brighter and the dark pixels darker. We can do this by multiplying the color values of all pixels whose color values average more than 128 by a number greater than one, and multiplying the color values of all pixels whose color values average less than 128 by a number less than one.

In performing any of these operations we have to bear in mind that color values have to be whole numbers, so we have to be sure to convert the result of the multiplication to an int. Also, color values can't be bigger than 255, so we need to look out for products that are too big when we increase the brightness of pixels.

Representing Collections of Pixels in Python

We can't draw a picture from just a collection of pixels. We need to know how the pixels are arranged: if we have 100 pixels, they could form a 10x10 square picture, or a 25x4 rectangular picture, or ....

Further, we need to know which pixel is in the upper left corner... is it the first one, or the last one, or neither?

We'll resolve this issue by putting our pixels in a list of lists. If our image is 300 pixels wide and 300 pixels high, there will be 300 lists of pixels, each representing a column, and each column will contain 300 pixels.. If a variable named picture holds such a list of lists of pixels, picture[x] will give us a list of pixels, those making up the column in position x, and picture[x][y] will be the pixel in column x and row y.

Working with images in Python

There is an example program accompanying these notes, imageproc2.py. You can download this program and put it in the folder you use for your Python programs on your virtual machine. The program uses a Python package called Pygame to do a number of things with images, including:

- filling an image with a solid color of your choice
- reading an image from a file
- modifying an image
- writing a modified image to a file

The main() function in the program includes all of these operations, but all but the first is commented out. By changing what is commented you can explore these various operations on images. You'll also be able to modify the code to do other things. Here is a tour of what the example program can do.

Experimenting with colors. If you run the example program as it is, it will draw a solid block of yellow. That's because the drawColor() function draws a solid block of color, and the makeColor() function creates a color from three color values, 255, 255, and 0 in the example. If you look in the table above you'll see that that should produce a bright yellow, which it does.

Challenge: Modify the program to determine what colors should be filled in for the ??? in the table above.

Reading an image file. Also accompanying these notes is a small image file, enginctr.jpg. Download this file, and put it in the same

folder with the example program. Now if you comment out the call to drawColor() in main(), and remove the comment symbol from the line that calls readImageFromFile(), and run the program, you should see the image.

Modifying an image. Keep the call to readImageFromFile(), and remove the comment symbol from the line that calls modifyImage(). Now if you run the program you will see the same image as before, only green. This happens because the code for the modifyImage() function takes each pixel in the image, finds its green value, creates a new color that has that green value and 0 red and 0 blue, and resets the pixel to that new color. So the pixels in the image are strictly green, with the amount of green depending on how much green there was in that position in the original image.

Saving an image to a file. You probably don't really want that green image, but if you remove the comment symbol from the line that calls saveImageToFile(), and run the program, the program will create the green image, as before, and save it to file called enginctrgreen.jpg.

Doing Your Own Image Processing

By modifying the code in imageproc2.py you can experiment with many ways of modifying images. Here are some things you can try. You can use enginctr.jpg in your experiments, or you can experiment on small jpg files you find yourself. If you are using a Mac you can use the preview program to take a screen shot of any part of an image, being sure not to make it too big, and save it as jpg. I'm sure there is a way to do this on Windows, too. Note: Pygame will probably be able to work with other kinds of images, too, such as png, though I have not tested that.

If you want to work with larger images, you'll want to change this line in the function setUpImage():

```
window = pygame.display.set_mode((300, 300), 0, 32)
```

As it is, this lets you handle images up to 300 by 300 pixels; increase those numbers if you want.

Here are some things you can try.

Easy challenges.

What happens if you switch color values, so that r becomes g (say), g becomes b, and b becomes r? Use the getR(), getG(), and getB() functions on each pixel to get the color values, then use makeColor() to create a new color from these color values in a different order, and put the result back in the pixel.

What happens if you replace each color value with 255 minus the original color value?

What happens if you replace each color value with half of it? Remember to use int() to turn the half into a whole number.

Harder challenges.

What happens if you multiply each color value by a number greater than one? You'll need to use if statements, or some other technique, to make sure the color values don't get bigger than 255. Here's one way:

```
color=pixArray[x][y]
r=getR(color)
newR=2*r
if newR>255:
        newR=255
…similarly for g and b
pixArray[x][y]=makeColor(newR,newG,newB)
```

(Note: It would be good to write a function for increasing a color value, including making sure it's not too big, and then using that to create a function for increasing all the color values in a color. That will clean up the messy code above.)

Convert a color image to black and white. Do this by calculating the average of the color values for each pixel, making a new color that has that average for all of its color values, and putting that new color back in the pixel. Recall from the earlier discussion that any color like this, all of whose color values are equal, will be somewhere on the scale between black and white.

Increase the contrast of an image by increasing the color values of all pixels whose average color value is greater than some amount, and decreasing the color values of all pixels whose average color value is less than some amount.

Convert an image into a “sketch”. Loop over all the pixels, calculating its average color value. If that average is different from the average for the previous pixel (by more than some set percentage) replace that pixel by a black one. If it isn’t, replace it by a white one. This has the effect of placing black lines where the brightness of the image changes, which in turn outlines some of the shapes in the image. You can also experiment with detecting differences in color rather than just in brightness.

About writing your own programs.

You’ll find it easier to make a copy of the imageproc2.py program, and modify that, than to write your own Python code from scratch. That’s because the example code includes various things you have to do to use Pygame that I haven’t explained, like calling pygame.display.update(), and deleting pixArray after using it. But feel free to experiment, and to read the Pygame documentation online. If you do read the Pygame documentation, you’ll see that it does a huge number of things beyond what we have discussed here, like drawing shapes, doing animations, and much more. Feel free to dig in.