Carleton University
Department of Systems and Computer Engineering
SYSC 1005 - Introduction to Software Development - Fall 2013

Lab 5 - Image Processing, Continued: Developing Filters that Perform Selection

Objectives
To develop Python functions that manipulate digital images by selectively modifying pixels in the images.

Attendance/Demo
To receive credit for this lab, you must demonstrate your work. When you have finished all the exercises, call your instructor or a TA, who will review the functions you wrote. For those who don't finish early, the TAs will ask you to demonstrate whatever functions you've completed, starting at about 30 minutes before the end of the lab period.

If you don't finish all the exercises during today's lab, you must finish them on your own time as "homework".

References
Image processing lecture slides and examples are posted on cuLearn.

Instructions

Getting Started
Step 1: Create a new folder named `Lab 5`.

Step 2: Copy `filters.py` (the module you worked on in Lab 4), `Cimpl.py`, and the three image (JPEG) files from last week's lab to your `Lab 5` folder. Download `if-examples.py` from cuLearn to your `Lab 5` folder. (This file contains the filters that were presented in Monday and Tuesday's lectures.)

Step 3: Launch Wing IDE 10 and open `filters.py`. Don't delete the `negative` and `grayscale` functions you wrote during Lab 4!

Exercise 1 - Solarizing an Image (`if` Statements)
Solarizing is a technique in which the developed negatives of colour photos are re-exposed. Here is a Python function that simulates this effect with a digital image. For each pixel, the value *v* of each of the red, green and blue components is changed to 255 - *v*, but only if *v* is less than 128. Higher intensity components (those with values of 128 or more) are left as-is. Read the code, and notice how three `if` statements are used to determine whether each of the component values should be changed.

```
def solarize(img):
    """
    Solarize the specified image.
    """
    for x, y, col in img:

        # Invert the values of all RGB components less than 128
        # leaving components with higher values unchanged.

        red, green, blue = col

        if red < 128:
            red = 255 - red

        if green < 128:
            green = 255 - green

        if blue < 128:
            blue = 255 - blue

        col = create_color(red, green, blue)
        set_color(img, x, y, col)
```

- Edit filters.py to contain a copy of this function (you can copy and paste the code from if-examples.py).
- Using the Python shell, load an image, call `solarize` to modify it, and display the modified image.
- Edit the function header, adding a parameter named `threshold`:

  ```
  def solarize(img, threshold):
  ```

  Edit the function so that each component is changed only if its value is less than the threshold value.
- Use the Python shell to interactively test your function with threshold values 64, 128 and 192. (Remember to reload the original image before you call the function with a different threshold value.) What effect does increasing the threshold have?
- Predict what the modified image will look like if 0 is passed as the threshold value. Call your function with this argument, and see if your prediction was correct.

- What threshold value will cause `solarize` to create the same effect as your `negative` function? Try it!

Exercise 2 - Converting an Image to Black and White (`if-else` Statements)

Here is a function that makes an image look like a woodcut print (http://en.wikipedia.org/wiki/Woodcut). For each pixel, we determine its brightness by calculating the average of its red, green and blue components. If the pixel's brightness is between 0 and 127, we change its colour to black, by setting the three colour components to 0. If the pixel's brightness is between 128 and 255, we change its colour to white. Read the code, and notice how an if-else statement is used to determine whether the pixel's colour should be changed to black or white.

```
def black_and_white(img):
    """
    Convert the specified image to a black-and-white (two-tone)
    image.
    """

    # Brightness levels range from 0 to 255.
    # Change the colour of each pixel to black or white,
    # depending on whether its brightness is in the lower or
    # upper half of this range.

    black = create_color(0, 0, 0)
    white = create_color(255, 255, 255)

    for x, y, col in img:
        red, green, blue = col

        brightness = (red + green + blue) / 3
        if brightness < 128:
            set_color(img, x, y, black)
        else:   # brightness is between 128 and 255, inclusive
            set_color(img, x, y, white)
```

- Edit filters.py to contain a copy of this function (you can copy and paste the code from if-examples.py).
- Using the Python shell, load an image, call `black_and_white` to modify it, and display the modified image.

Exercise 3 - Extreme Contrast (Using `if-else` Statements)
In filters.py, define a function that is passed an image and maximizes the contrast between pixels (light pixels are made lighter, and dark pixels are made darker). The function header is:

```
def extreme_contrast(img):
```

This function will change the red, green and blue components of each pixel. If the component's value is between 0 and 127 (i.e., it's relatively dark), the component is changed to 0. If the component's value is between 128 and 255 (i.e., it's relatively light), the component is changed to 255. When defining this function, use `if-else` statements. Do not use `if` statements or `if-elif-else` statements.

Exercise 4 - Converting an Image to Black and White and Gray (`if-elif-else` Statements)
Here is a function that is similar to the black-and-white filter from Exercise 2, but which selectively changes the pixels' colours to black, white or medium-gray. For each pixel, an `if-elif-else` statement is used to determine whether the pixel's brightness is between 0 and 84 (the pixel's colour is changed to black), between 85 and 170 (the pixel's colour is changed to medium-gray), or between 171 and 255 (the pixel's colour is changed to white).

```
def black_and_white_and_gray(img):
    """
    Convert the specified image to a black-and-white-and-gray
    (three-shade) image.
    """

    black = create_color(0, 0, 0)
    gray = create_color(128, 128, 128)
    white = create_color(255, 255, 255)

    # Brightness levels range from 0 to 255. Change the colours
    # of pixels whose brightness is in the lower third of this
    # range to black, in the upper third to white, and in the
    # middle third to medium-gray.

    for x, y, col in img:
        red, green, blue = col
        brightness = (red + green + blue) / 3

        if brightness < 85:
            set_color(img, x, y, black)
        elif brightness < 171:  # brightness is between 85 and 170,
                                #inclusive
            set_color(img, x, y, gray)
        else:                    # brightness is between 171 and 255,
                                # inclusive
            set_color(img, x, y, white)
```

- Edit filters.py to contain a copy of this function (you can copy and paste the code from if-examples.py).
- Using the Python shell, load an image, call `black_and_white_and_gray` to modify it, and display the modified image.

Exercise 5 - Sepia Tinting (Using `if-elif-else` Statements)
I'm sure you've seen old black-and-white (actually, grayscale) photos that, over time, have gained a yellowish tint. We can mimic this effect by creating sepia-toned images.

In filters.py, define a function that is passed an image and transforms it by sepia-tinting it.The function header is:

```
    def sepia_tint(img):
```

There are several different ways to sepia-tint an image. One of the simplest approaches involves determining if each pixel's red component lies in a particular range of values, and if it does, modifying the pixel's red and blue components, leaving the green component unchanged.

Here's a description of the algorithm:

First, we convert the image to grayscale, because old photographic prints were grayscale. (Hint: to do this, your function must call your `grayscale` function. Don't replicate the grayscale algorithm in `sepia_tint`.) After this conversion, each pixel is a shade of gray; that is, its red, green and blue components are equal.

Next, we tint each pixel so that it's a bit yellow. In the RGB system, yellow is a mixture of red and green. To make a pixel appear slightly more yellow, we can simply decrease its blue component by a small percentage. If we also increase the red component by the same percentage, the brightness of the tinted pixel is essentially unchanged. The amount by which we change a pixel's red and blue components depends on whether the pixel is a dark gray, a medium gray, or a light gray:

- If the red component is less than 63, the pixel is in a shadowed area (it's a dark gray), so the blue component is decreased by multiplying it by 0.9 and the red component is increased by multiplying it by 1.1;
- If the red component is between 63 and 191 inclusive, the pixel is a medium gray. The blue component is decreased by multiplying it by 0.85 and the red component is increased by multiplying it by 1.15.;
- If the red component is greater than 191, the pixel is in a highlighted area (it's a light gray), so the blue component is decreased by multiplying it by 0.93 and the red component is increased by multiplying it by 1.08.

When defining this function, use `if-elif-else` statements. Do not use `if` statements or `if-else` statements.

Interactively test your sepia-tinting function.

Exercise 6
In `filters.py`, define a *helper function* named `_adjust_component` that is passed the value of a pixel's red, green or blue component. (Yes, the function's name begins with an underscore. A convention followed by Python programmers is to use a leading underscore to indicate that a function is intended for internal use only; in other words, that it should only be called by functions in the same module.) Here is the function header:

```
    def _adjust_component(amount):
```

Note: parameter `amount` is a value of type `int`, not an image or a value of type `Color`.

If `amount` is between 0 and 63, inclusive, the function returns 31. If `amount` is between 64 and 127, inclusive, the function returns 95. If `amount` is between 128 and 191, inclusive, the function returns 159. If `amount` is between 192 and 255, inclusive, the function returns 223. Your function should assume that the integer bound to `amount` will always lie between 0 and 255, inclusive; in other words, it does not need to check if `amount` is negative or greater than 255.

Important:
- the value returned by this function must be an `int`, not a `float` or a `str` (character string);
- this function does not print anything; i.e., it must not contain `print` statements.

Interactively test your function; for example, try:

```
>>> _adjust_component(10)
31
>>> _adjust_component(85)
95
>>> _adjust_component(142)
159
>>> _adjust_component(230)
223
```

You'll need to run more than these four tests (one test for each of the quadrants). It's also a good idea to have test cases that check values at the upper and lower limits of each quadrant. Verify that:

- `_adjust_component(0)` and `_adjust_component(63)` return 31,
- `_adjust_component(64)` and `_adjust_component(127)` return 95,
- `_adjust_component(128)` and `_adjust_component(191)` return 159, and
- `_adjust_component(192)` and `_adjust_component(255)` return 223.

Exercise 7 - Posterizing an Image
Posterizing is a process in which we change an image to have a smaller number of colours than the original. Here is one way to do this:

Recall that a pixel's red, green and blue components have values between 0 and 255, inclusive. Suppose we divide this range into four equal-size quadrants: 0 to 63, 64 to 127, 128 to 191, and 192 to 255. We can use the _adjust_component function you wrote for Exercise 6 to determine if a component lies within a specific quadrant, and if it is, change the component to the value that is in the middle of the quadrant.

In filters.py, define a function that is passed an image and posterizes it. The function header is:

```
    def posterize(img):
```

For each pixel, change the red component to the midpoint of the quadrant in which it lies. Do the same thing for the green and blue components. In other words, for each pixel, your function must call your `_adjust_component` function three times (once for each of the pixel's components).

Interactively test your posterizing function.

Exercise 8 (The and Operator)
Download checkerboard_bw.gif and checkerboard_red_blue.gif from cuLearn to your Lab 5 folder.

Module if-examples.py contains the definition of a function named `swap_black_white` that is passed an image. This function makes all black pixels white and all white pixels black, leaving all other pixels unchanged.

Read the code, and notice how the `and` operator is used in the expressions that determine if the pixel is black (the red, blue and green components are all 0) or if the pixel is white (the red, blue and green components are all 255).

- Load the image stored in checkerboard_bw.gif and display it. Call `swap_black_white`, passing it that image. If the function is correct, the modified image should be a different black-and-white checkerboard.

- Load the image stored in checkerboard_red_blue.gif and display it. Call `swap_black_white`, passing it that image. If the function is correct, the image will be unchanged (it has no black or white pixels).

Exercise 9
In filters.py, define a function that is passed an image and changes each pixel's colour to white, black, or one of three primary colours (red, green and blue). The function header is:

```
    def simplify(img):
```

For every pixel, the function first compares the pixel's red, green and blue components to a high threshold (200) and a low threshold (50). If all three components are above the high threshold, the pixel's colour is changed to white (255, 255, 255). If all three components are below the low threshold, the pixel's colour is changed to black (0, 0, 0).

If the pixel's colour isn't changed to white or black, the function performs more comparisons on the the components. If the red component is greater than both the green component and the blue component, the pixel's colour is changed to red; i.e., the RGB triple (255, 0, 0). If the green component is greater both the red component and the blue component, the pixel's colour is changed to green; i.e., the RGB triple (0, 255, 0). Otherwise, the pixel's colour is changed to blue; i.e., the RGB triple (0, 0, 255). (This handles the case where the blue component is greater than both the red component and the green component, plus all the other cases that can occur; for example, two of a pixel's three components are equal, all three of its components are equal, etc.)

Hint: this function doesn't require many lines of code if you use one or more of Python's Boolean operators: `and`, `or`, `not`.

Wrap-up

1. Remember to have a TA review your solutions to the exercises and give you a grade (Satisfactory, Marginal or Unsatisfactory) before you leave the lab.
2. You'll be adding functions to your `filters` module during Lab 6, so remember to save a copy of your filters.py file (copy it to a flash drive, or email a copy to yourself, or store it on your M: drive - remember, files left on your desktop or in your Documents folder are not guaranteed to be there the next time you log in).
3. Log in to cuLearn, click the Submit Lab 5 link and submit filters.py. After you click the Add submission button, drag the file to the File submissions box. After the icon for the file appears in the box, click the Save changes button. At this point, the submission status for your files is "Draft (not submitted)". You can resubmit the file by clicking the Edit my submission button. After you've finished uploading the file, remember to click the Submit assignment button to change the submission status to "Submitted for grading".

Posted: Tuesday, October 8, 2013