Assignment 6, CSC 101

due March 13, 11:59pm

For this assigment you will implement three programs that transform images.

The first program will read an image file and write a new image revealing a hidden picture.

The second program will read an image file and write a new, "faded" version.

The third program will read an image file and write a new, blurred version of the image.

Download hw6.zip. Unpack *hw6.zip* to create a *hw6* directory that contains three subdirectories (*puzzle*, *fade* and *blur*).

Image File Format

The specific details for the three programs are discussed below. Each will process an image file stored in the *ppm* format. More specifically, your programs will process image files that conform to the P3 format discussed below. On the lab machines, you can view ppm files with either the *eog* program or the *gthumb* program (among others). Details of this format are provided on a separate ppm P3 format page.

Note: if you use a program to create your own ppm files for testing, be sure to remove anything (such as comments) that does not conform to the expected format.

---

puzzle : program 1

Develop this first program in a file named `puzzle.py`. This program will get you started with the core functionality of processing a ppm image file.

You may separate the functionality of your program into multiple files, but `puzzle.py` must contain the `main` function. In addition, to avoid name conflicts with the later programs, the names of any additional files must start with `puzzle`.

Command-line Argument

Your program must take a single command-line argument that specifies the name of the input image file. If this argument is not provided, an appropriate usage message must be printed and the program should terminate.

Image Reading

For this program you can read a pixel, process the pixel, and then output the pixel before moving to the next pixel (in other words, you will not need to store all of the pixels). Recall that, in the ppm P3 format, each pixel is denoted by three integer values representing the pixel's color components. Your program must not make any assumptions about the number of pixels or pixel components per line.

This means that a line may contain many pixels or only a single pixel; likewise, a line may contain only the red and green components of a pixel with the blue component appearing on the next line.

To aid in processing pixels, consider using the `groups_of_3` function from Lab 6.

Image Output

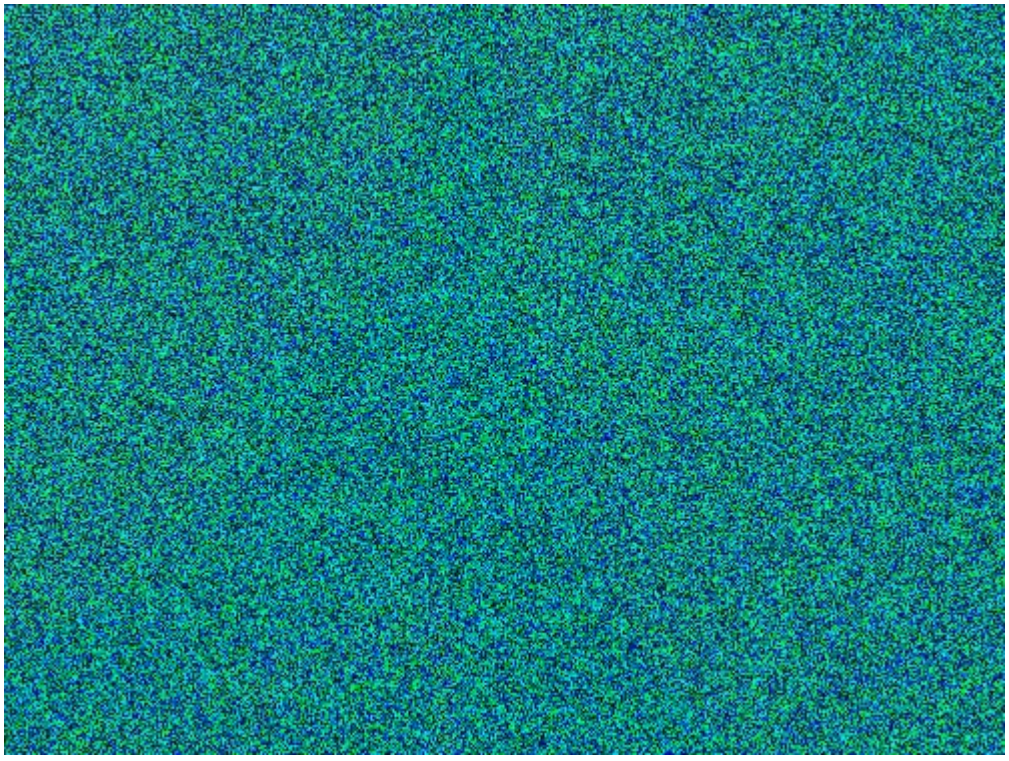
Your program will output the decoded image to a file named *hidden.ppm*. The output must be a valid P3 ppm image file.

Do not forget to write the required header information to your output file.

Solving the Puzzle

The puzzle image (shown below) hides a real image behind a mess of random pixels. In reality, the image is hidden in the red components of the pixels. Decode the image by increasing the value of the red component by multiplying it by 10 (do not allow the resulting value to pass the maximum value of 255). In addition, set the green and blue components equal to the new red value.

Download the puzzle image. Shown below is the image in png format. It should be obvious once you have properly decoded the puzzle image. If you are unsure, just ask.



fade : program 2

This program is a relatively minor modification of the previous program.

Develop your program in a file named `fade.py`. The required functionality of your program is described below.

You may separate the functionality of your program into multiple files, but `fade.py` must contain the `main` function. In addition, to avoid name conflicts with the other programs, the names of any

additional files must start with `fade`.

Command-line Arguments

Your program must take four command-line arguments (in the following order). The first specifies the name of the input image file. The second and third specify *row* (y-coordinate) and *col* (x-coordinate) positions. The last specifies a *radius*. These last three arguments are expected to be integers.

Image Reading

For this program you can read a pixel, process the pixel, and then output the pixel before moving to the next pixel (in other words, you will not need to store all of the pixels). Again, your program must not make any assumptions about the number of pixels or pixel components per line. This means that a line may contain many pixels or only a single pixel; likewise, a line may contain only the red and green components of a pixel with the blue component appearing on the next line.

Image Output

Your program will output the faded image to a file named *faded.ppm*. The output must be a valid P3 ppm image file.

Do not forget to write the required header information to your output file.

Fading

This program will transform pixel values based on their distance from a specified point (this point may fall outside of the image). The *row* and *col* coordinates specified on the command-line give the point and the *radius* is used to control the fading.

For each pixel, compute the distance from the pixel location to the specified point. Scale the color components of the pixel according to **(radius - distance) / radius** (multiply the pixel components by this value). **Do not**, however, use a scale value below 20% (i.e., if the result of the scaling equation is less than 0.2, use 0.2; this prevents very dark borders around the image).

Two examples are shown below. A link to the ppm file is provided, but the images shown in the browser are in png format.

mouse.ppm input

Faded version with point at row=230, col=255 and radius 255.

lights.ppm input

Faded version at point <160, 200> with radius 250.

blur : program 3

This program is an extension of the type of processing done in the previous. In particular, this program will require storing the pixels of the image while processing.

Develop your program in a file named `blur.py`. The required functionality of your program is described below.

You may separate the functionality of your program into multiple files, but `blur.py` must contain the

`main` function. In addition, to avoid name conflicts with the other programs, the names of any additional files must start with `blur`.

Command-line Arguments

Your program must take two command-line arguments (in the following order). The first specifies the name of the input image file. The second argument is optional. If provided, the second argument is an integer value specifying the "neighbor reach" to use in the blurring calculation (discussed below). This will allow a user to specify the amount of blurring. Your program will use this value to determine which neighbor pixels to consider in the averaging calculation.

If a second command-line argument is not present (i.e., the blur factor is not specified), then use a default value of 4.

Image Reading

Unlike the previous programs, your program for this part will need to store the image while processing.

Your program will read the pixel information from the input image file into a two-dimensional list (you can use a one-dimensional list, if you truly prefer, but the calculations are a bit more direct on a two-dimensional list) that stores the information for each pixel.

As before, your program must not make any assumptions about the number of pixels or pixel components per line. This means that a line may contain many pixels or only a single pixel; likewise, a line may contain only the red and green components of a pixel with the blue component appearing on the next line.

Image Output

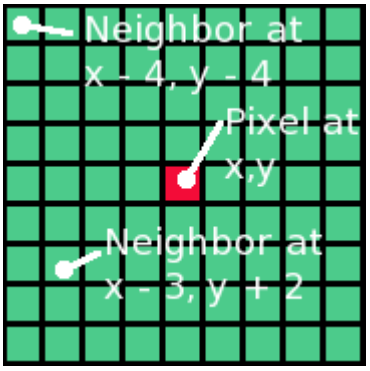
Your program will output the blurred image to a file named *blurred.ppm*. The output must be a valid P3 ppm image file.

Do not forget to write the required header information to your output file.

Blurring

Once your program has read the contents of the image file, it will blur the image. This will be done by computing, for each pixel, the average of nearby pixels (more precisely, the averages over each color component of the nearby pixels).

A pixel's blurred color will be determined by the colors of the pixels within a specified "neighbor reach". The default "neighbor reach" is 4 (but this can be modified by a command-line argument as discussed previously). This means that a pixel's blurred color will be determined by the colors of the pixels within four pixels to the left or right and within four pixels above or below (this will form a square around the pixel). The following diagram is meant to help illustrate.



In this diagram, we are trying to compute the color for the pixel in the center (the red element). Its neighbors (within a reach of 4) are all of the green elements. The pixels outside of this 9x9 square are not considered in the blurring calculation for this pixel. To compute the color for the center pixel, average the red, green, and blue components (independently) of every pixel in the 9x9 square (including the pixel itself; the red element in this diagram).

Some pixels will not have the full complement of neighbors (such as those on or near the edge of the image). In these cases, just average the existing neighbors (i.e., those within the bounds of the image).

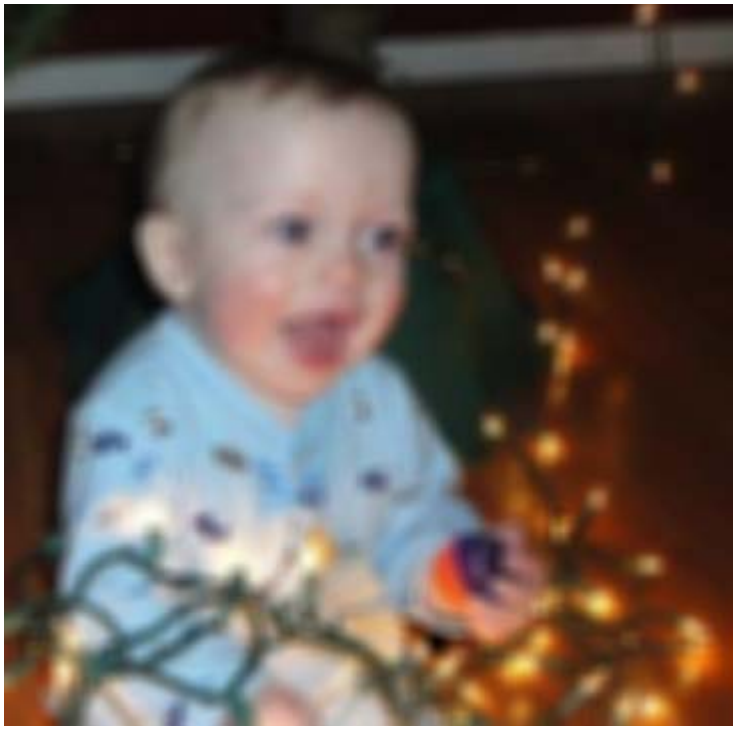
Two examples are shown below. A link to the ppm file is provided, but the images shown in the browser are in png format.

mouse.ppm input

Blurred version (reach of 4).

lights.ppm input

Blurred version (reach of 4).



Handin

You must submit your solution on unix1.csc.calpoly.edu (or on unix2, unix3, or unix4) by 11:59pm on the due date.

The handin command will vary by section.

- Those in Aaron Keen's sections will submit to the **akeen** user.

  At the prompt, type **handin akeen 101hw6 puzzle/*.py fade/*.py blur/*.py**

- Those in Julie Workman's sections will submit to the **grader-ph** user.

  At the prompt, type **handin grader-ph 101hw6 puzzle/*.py fade/*.py blur/*.py**

- Those in Paul Hatalsky's sections will submit to the **grader-ph** user.

  At the prompt, type **handin grader-ph 101hw6 puzzle/*.py fade/*.py blur/*.py**

- Those in John Campbell's sections will submit to the **jcampb25** user.

  At the prompt, type **handin jcampb25 hw6 puzzle/*.py fade/*.py blur/*.py**

Be sure to submit all files that are necessary to compile your program (including your files from the previous assignment(s)).

Note that you can resubmit your files as often as you'd like prior to the deadline. Each subsequent submission will replace files of the same name.

Grading

The grading breakdown for this assignment is as follows.

- **Clean Execution**: 5% — Program runs without crashing (and the submitted source demonstrates a legitimate attempt at a solution).
- **Style**: 5% — Consistent formatting including the use of braces, parentheses, and indentation. Each line of code below maximum length (i.e., less than 80 characters).
- **Decomposition**: 15% — Functional decomposition restricts each function to a single task. Duplicate functionality has been abstracted to a single function (i.e., duplicate code is removed).
- **Functionality**: 75% — Required functionality has been implemented.