Project 5, CPE101
Due: 11/22/19

For this project, you will implement three programs that transform images. **No unittest required** for this project.

1) The first program will read an image file and write a new image revealing a hidden picture (puzzle.py).
2) The second program will read an image file and write a new, "faded" version (fade.py).
3) The third program will read an image file and write a new, blurred version of the image (blur.py).
4) Your main program can call all these three programs (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. Note that all values written to P3 files (except the P3 header) **must be integers**. On the lab machines, you can view ppm files with either the IrfanView, *eog* program, or the *gthumb* program (among others).

Image File Format

The P3 (.ppm) format is a text-based format (meaning that it is readable text) that defines an image as a sequence of pixels beginning with the top-left pixel and stored in row order (i.e. every pixel in a row is stored in left-to-right order and before any pixel in the next row).

A file conforming to the P3 format begins with header information. The header consists of the characters P3, the integer width of the image (in pixels), the integer height of the image, and the maximum value for a color component (we will use 255 for this value). Immediately following the header is the color information for each pixel. A pixel's color is represented by three integers denoting the red, green, and blue components (in that order). The following example shows how a file would look if it contained an image with one blue pixel, one red pixel, and one green pixel.

```
P3
3 1
255
0
0
255
255
0
0
0
255
0
...
```

The top-left pixel is said to be at location <0, 0> and the bottom-right at location `<width - 1, height - 1>` .

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. `puzzle.py` which contains function for decoding the image file.

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 to the terminal and the program should terminate. This message is as follows:

```
Usage: python3 puzzle.py image_file.ppm
```

If, however, the image argument is provided but cannot be opened (e.g. does not exist), your program must print to the terminal:

```
Unable to open <image>
```

In this case, replace `<image>` with the name of the file passed in as an argument. You must use a try/except statement to perform this step.

Image Reading

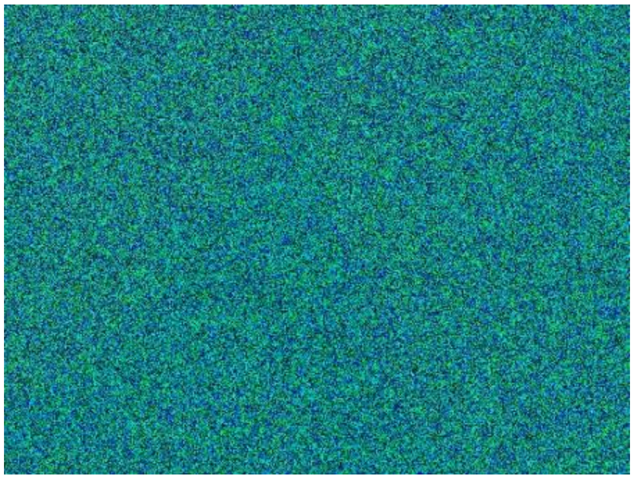
For this program you must read a pixel, process the pixel, and then output the pixel before moving to the next pixel (in other words, you must not store all of the pixels at once). Recall that, in the ppm P3 format, each pixel is denoted by three integer values representing the pixel's color components. 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.

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. Shown below is the hidden image. 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.

Command-line Arguments

Your program must take four command-line arguments (in the following order).

1) The first specifies the name of the input image file.
2) The second specifies *row* (y-coordinate)
3) The third specifies *column* (x-coordinate) positions.
4) The last specifies the fade *radius*.

These last three arguments are expected to be integers. If these arguments are not provided, an appropriate usage message must be printed to the terminal and the program should terminate. This message is as follows:

```
Usage: python3 fade.py <image> <row> <column> <radius>
```

Your program will output the faded image to file name faded.ppm, which must be valid ppm file.

If, however, the image argument is provided but cannot be opened (e.g. does not exist), your program must print to the terminal:

```
Unable to open <image>
```

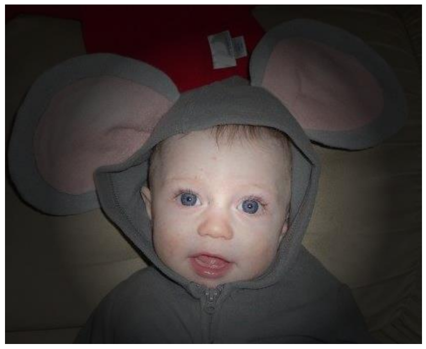
In this case, replace `<image>` with the name of the file passed in as an argument. You must use a try/except statement to perform this step.

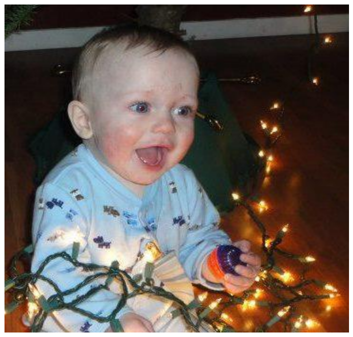
Fading

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 *column* 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 (multiply) the color components of the pixel by **(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.

Original image


Faded version row=230, col=255, radius=255


Original image

Faded version row=160, col=200, 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`.

Command-line Arguments

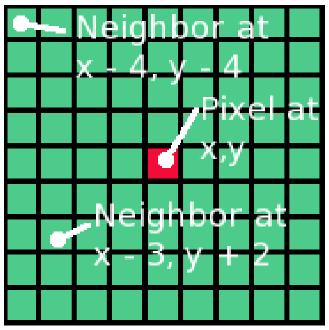
Your program must take two command-line arguments (in the following order). The first specifies the name of the input image file. The second argument is optional. If provided, the second argument is an integer value specifying the "neighbor reach" to use in the blurring calculation (discussed below). 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, however, the image argument is provided but cannot be opened (e.g. does not exist), your program must print to the terminal:

```
Unable to open <image>
```

In this case, replace `<image>` with the name of the file passed in as an argument. You must use a try/except statement to perform this step.

Blurring



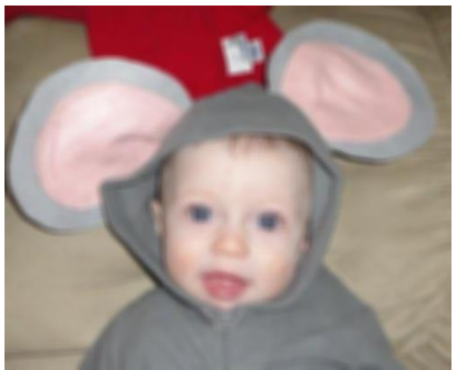
Once your program has read the contents of the image file, it will blur the image. This will be done by computing, for each pixel, the average of nearby pixels (more precisely, the averages over each color component of the nearby pixels). A pixel's blurred color will be determined by the colors of the 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 diagram is meant to help illustrate.

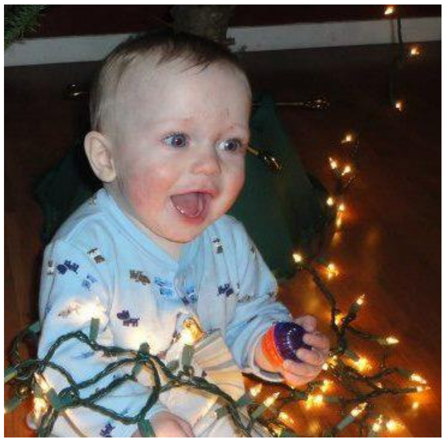
Note: Even for default neighbor reach you need to pass the reach as parameter.
python3 blur.py <image_file.ppm> <reach>

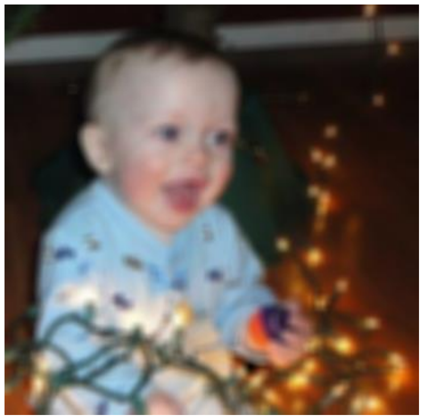
In this diagram, we are trying to compute the color for the pixel in the center (the red element). Its neighbors (within a reach of 4) are all of the green elements. The pixels outside of this 9x9 square are not considered in the blurring calculation for this pixel. To compute the color for the center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

Original Image


Blurred version (reach of 4).


Original Image


Blurred version (reach of 4).

Submission

Submit all you py files in the PolyLearn.

1) puzzle.py runs without parameter
2) fade.py
3) blur.py
4) pixelImage.py (your main program)

Note:

1) The main program (pixelImage.py) in project5 need to import puzzle.py, fade.py, and blur.py

2) Based on number of parameters you access appropriate file to call the functions. (If it is just image file it calls function from puzzle.py, if it has image and reach it calls function from blur.py and if it has 4 parameters it is accessing fade.py functions.)

3) Assume we always pass the reach as a parameter for blurring the image

Due: Friday 11/22 for 100%

Saturday 11/23 for 80%

Rubric:

Implementation:

1) **pixelImage: 10 points**
2) **puzzle: 10 points**
3) **fade: 10 points**
4) **blur: 15 points**

Output Image:

5) **puzzle: 15 points**
6) **fade: 15 points**
7) **blur: 15 points**
8) **purpose statement/signature 10 points**