EECS 1710
LAB 6 :: Generating, Masking and Filtering Images

Prerequisite (labs 0-5) – please ensure that you have setup your EECS account, and can log into the lab machines prior to starting this lab, and you are familiar with submit/websubmit.

NOTE: This lab is worth 3% of your final grade.

STEP 1: Importing and unzipping the starter code

Download and extract the following lab4 starter code:
http://www.eecs.yorku.ca/course_archive/2022-23/F/1710/labs/lab6/lab6.zip

There are three files to submit for this lab. Assuming the above zip file is downloaded and extracted into the **`1710/labs/`** folder in your home directory, you should be able to navigate to the labs folder and submit everything with the following commands:

```
cd
cd 1710/labs/
submit 1710 lab6 lab6/*
```

This will submit all files within the **`lab6`** folder. Alternatively, you may use web submit to submit/delete individual files (see link at the end of this document). To list files submitted, type:

```
submit -l 1710 lab6
```

STEP 2: Exercises

Please note, **the organization of your starter code** is similar to **`lab5`**, in that all the files for your lab are contained within a single sketch folder. The main file for the lab is **`lab6.pde`** (this contains your setup, draw and any event methods like `mousePressed()` that processing recognizes as standard), and all the other methods (stored in additional files within your sketch folder) are invoked from here.

Relevant reference objects/methods you will need for this lab can be found here:

`PImage`	https://processing.org/reference/#image
`Color`	https://processing.org/reference/#color
`pixels[]`	https://processing.org/reference/pixels.html
`lerpColor()`	https://processing.org/reference/lerpColor_.html
`filter()`	https://processing.org/reference/filter_.html

Question 1: Nested loops

In **`Question1.pde`**, as a quick warmup exercise for using nested loops (common when manipulating/generating image pixels). Specifically, we will use the loop counters **`i`** and **`j`** to each increment over `0-imgWidth` and `0-imgHeight` in a nested fashion (a loop within a loop) to print a set of coordinate locations to the console.

(a) Create a method **`showImageCoords(…)`**, that displays the coordinates regularly spaced points on a 2D grid, using an *image* coordinate system.

```
void showImageCoords(int imgWidth, int imgHeight)
```

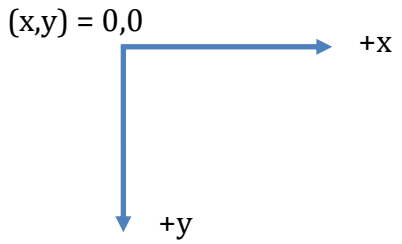
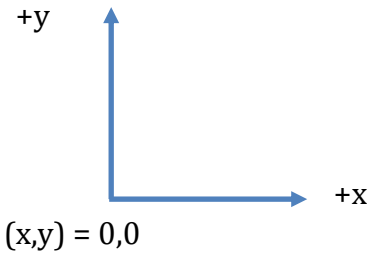


Image co-ordinates



Cartesian co-ordinates

Thus, a grid of size (`imgWidth` x `imgHeight` = 3 x 4) would be created with the call:

```
showImageCoords(3, 4);
```

And would give the following output to the console window:

```
( 0, 0) ( 1, 0) ( 2, 0)
( 0, 1) ( 1, 1) ( 2, 1)
( 0, 2) ( 1, 2) ( 2, 2)
( 0, 3) ( 1, 3) ( 2, 3)
```

(b) Create a method **`showCartesianCoords(…)`**, that displays the coordinates regularly spaced points on a 2D grid, using an *cartesian* coordinate system.

```
void showCartesianCoords(int width, int height)
```

Cartesian coordinates essentially begin in the bottom left corner (see figure above), so a grid of size (width x height = 3 x 4) would be created with the call:

```
showCartesianCoords(3, 4);
```

And would give the following output to the screen:

```
(0, 3) (1, 3) (2, 3)
(0, 2) (1, 2) (2, 2)
(0, 1) (1, 1) (2, 1)
(0, 0) (1, 0) (2, 0)
```

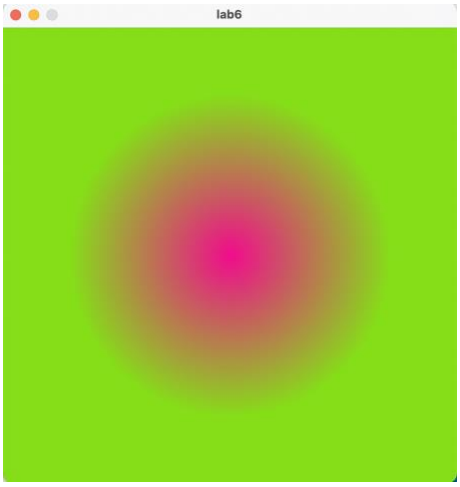
Question 2: Generating and Masking Images

(a) In **`Question2.pde`**, we will first construct a method to generate a **new image** that has an opaque radial gradient that gradually changes between two input colours (one colour begins at the centre of the image, and slowly changes as we move away from the centre to the edges of the image). The method header is given as:

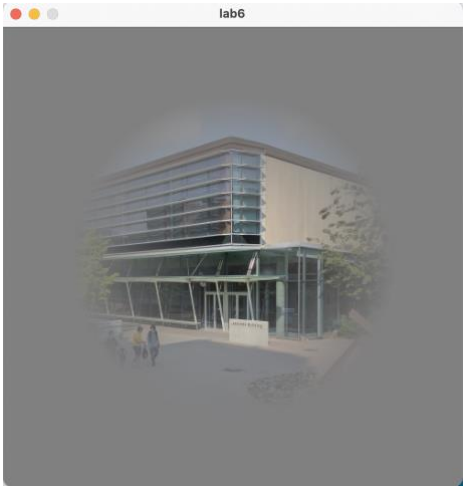
```
PImage radialGradient(int sizeX, int sizeY,  int col1, int col2)
```

where **`sizeX`** and **`sizeY`** represent parameters for the width and height of the image (respectively); **`col1`** is a colour at the centre of the image, and **`col2`** is the colour **`col1`** gradually changes to as we move away from the centre of the image.

The method should produce something like that shown on the left of the figure below – though it depends on the colours passed to the method.



(a) radial gradient (500x500)
col1 = color(241, 12, 141, 255)
col2 = color(134, 222, 25, 255)



(b) radial vignette (on lassonde.jpg)
col1 = color(128,128,128,0)
col2 = color(128,128,128,255)

Note: to achieve the gradient, you can use the **`lerpColor(col1, col2, percentage)`** method, where the `percentage` parameter is used to interpolate between `col1` and `col2` for a pixel based on its distance away from the centre of the image, i.e. as a percentage of the maximum radius (maxRadius) of the image:

$$\textbf{percentage} = (2 \times distance)/maxRadius$$

where:

$$\textbf{maxRadius} = \sqrt{(sizeX/2)^2 + (sizeY/2)^2}$$

$$\textbf{distance} = \sqrt{(centreX - i)^2 + (centreY - j)^2}$$

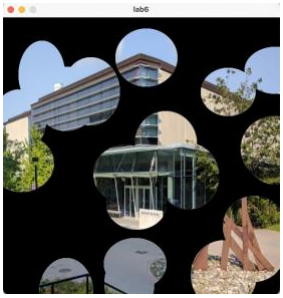
(b) In the second part of this question, use your method to create a *vignette* effect on an image of your choice (see right side of the figure above). In this example, the **`lassonde.jpg`** file is used (you may use this or another image).

The colours **`col1`** and **`col2`** should be chosen such that `col1` is fully transparent (i.e. has an alpha channel of zero), and `col2` is fully opaque (has an alpha channel of 255).

To apply this radial gradient as a vignetting mask:

- set your application window to the same size as the image (you will have to load and check the dimensions of the image first… then go back and modify your `size()` method and re-run your program).
- Set your background colour to the same colour used for col2.
- Paint the image to the application window first
- Now paint the radial gradient image (using the new colours col1 & col2).. the example in the figure uses a grey colour for both, with one fully transparent and the other fully opaque.
- save your composited image (that combines the original image + the vignette overlay) to a new image file (this will save in the sketch folder for submission). You can do this using by running the **`save()`** method to save the image from the application window into an image file. See link for details on usage.

[OPTIONAL - creative alternative for part (b)] create a method to generate a unique mask of your choosing, by drawing graphics (black fill on a white background), to the application window. You may use any combination of the primitive shapes (lines, rect, circle, ellipse, etc) or generate them randomly/using a loop. Another approach could be to use a second image and filter it using THRESHOLD and/or INVERT … see the example of using *filter* in the reference manual for here.



← uses random ellipses to create transparent regions

With the black and white image, extract the pixels from the application window and store in a new image using `get()`: https://processing.org/reference/get_.html

Once this is done, traverse your pixels and modify the alpha values of the black (or white) pixels to be transparent, while keeping all the other regions as opaque (or use the `mask()` method directly. Then re-paint the resulting final image. Save this image (this will save to your sketch folder – for submission with the rest of the lab).

Question 3: Instagram Filters!

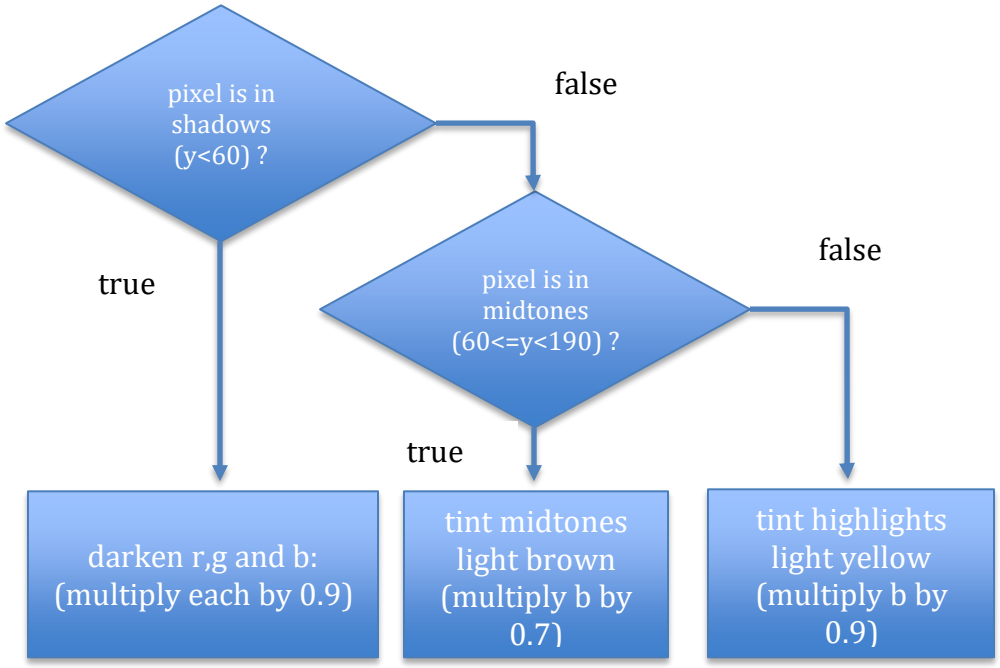
In `Question3.pde`, you will write two methods to manually filter (stylize) an input image. These are similar to the effects (though somewhat simplified here) that you might apply to an instagram photo (or through photoshop edits).

Before we create the methods, **look at the figures at the end of this question** to see some results of running these two filters on two sample images (included in your starter code sketch folder).

(a) The first filter is a method to generate a "sepia" version of your original image. This is a stylised re-colouring of each individual image pixel in the image, to give the image an aged feel, achieved by modifying shadows, midtones and highlights.

```
PImage sepiaImage(PImage image)
```

The algorithm (steps) for converting to sepia can be summarized by the following flowchart (where the diamond represents a conditional test – e.g. if/else test; and "y" refers to the grayscale equivalent of a colour pixel; and "r", "g", "b" refer to the red, green and blue components of a pixel, respectively.



The original image and the sepia version should both be shown by painting them to the application window (make sure it is big enough to show both images – one under the other). HINT: use `imageMode(CORNER)`, with `image()` to place them.

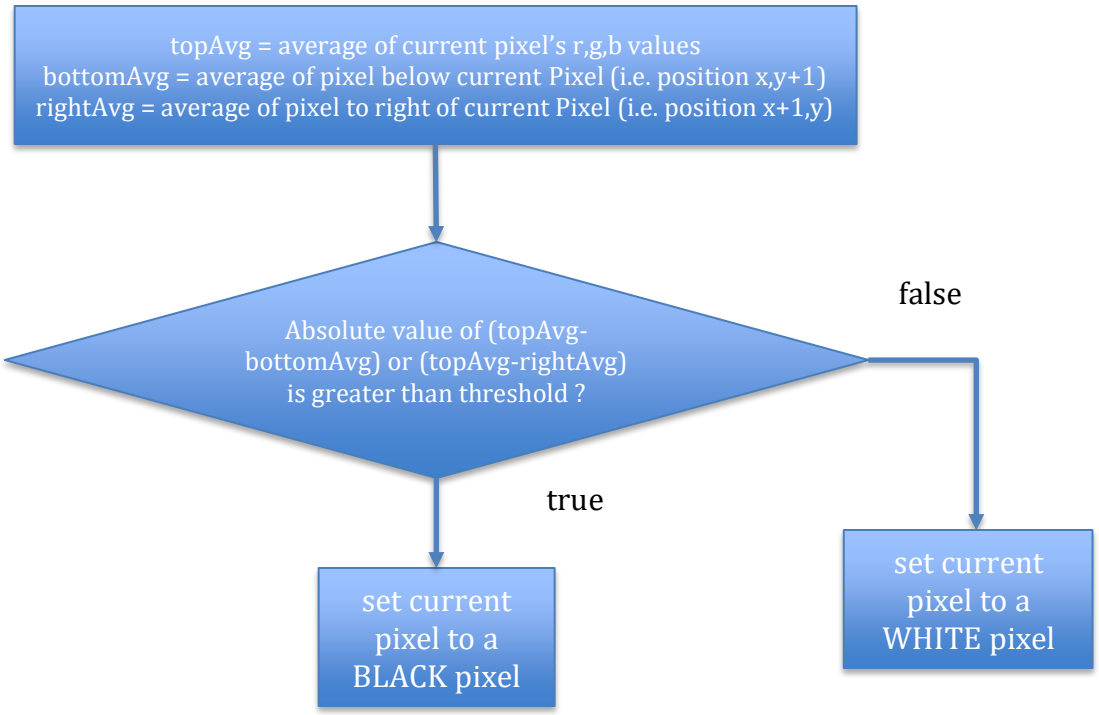
It is recommended to make a copy of **`PImage`** inside the method, and work with that. To compute **`y`**, you can either use the formula from lab 1, or (an easier approach), create a second copy of the **`PImage`** and filter it using the **`filter()`** method with the **GRAY** filter. The pixel indices can be used to refer both to the original image and the filtered gray image to get these values: https://processing.org/reference/filter_.html

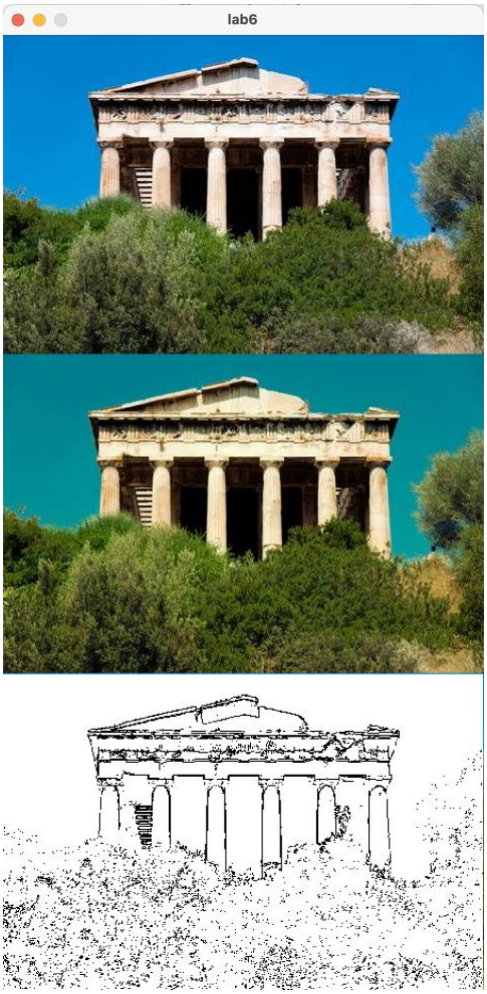
To increase the sepia effect, you can experiment with manipulating the factors (0.9, 0.7, 0.9) used to multiply shadow, midtone and highlight pixels respectively.

(b) Create a method to generate a simplified "edge image" version of your original image. This image will create a new version of the original image as a black and white image, showing black pixels for the "edges" (locations where there is a high colour contrast/change) and white everywhere else. The sensitivity to generating edges is controlled by a parameter (**`threshold`**) which can be modified to strengthen the edges present on a given image.

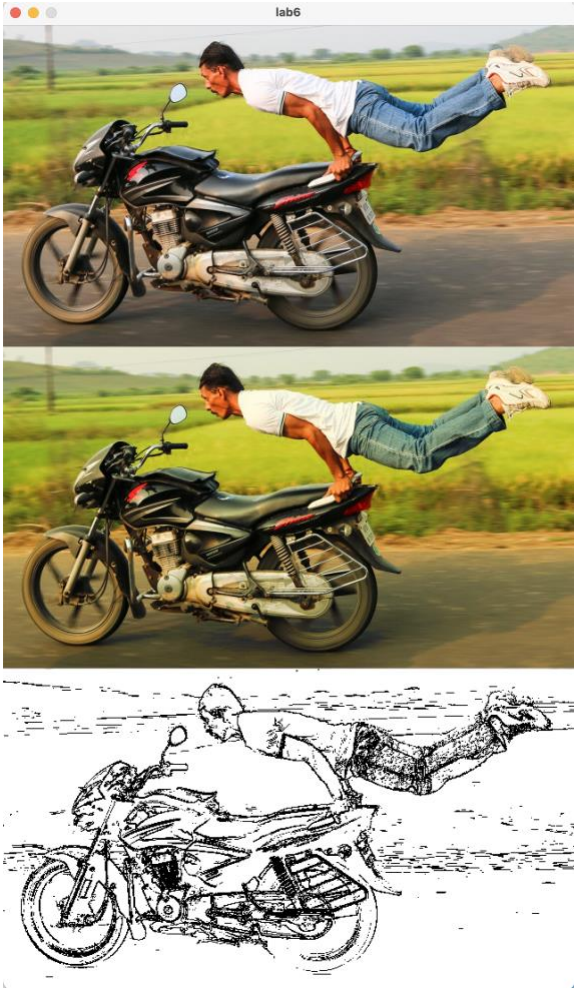
```
PImage edgeImage(PImage image, int threshold)
```

The algorithm for generating a simple edge version of an image, is given in the flowchart below, where topAvg, bottomAvg and rightAvg are averages of the current pixel, and pixels nearby (just below, and just to the right of the current pixel):





(a) *top*: "temple.jpg" (original)
middle: sepiaImage
bottom: edgeImage, threshold=50



(b) *top*: "yogaBike.jpg"
middle: sepiaImage
bottom: edgeImage, threshold=22

STEP 3: SUBMISSION (Deadline 5:00pm Tue Nov 22th, 2022)

NOTE: ***REMEMBER,*** ***you can choose to use the web-submit function (see Lab 0 for walkthrough). You will need to find and upload your lab6 *.pde files independently if doing it this way.***

Web-submit can also be used to check your submission from the terminal, and can be found at the link: https://webapp.eecs.yorku.ca/submit/