Cross-Stitch Application:-

Version 1 -- Image Model

In this assignment, we will be building a model that can be used to manipulate images to produce some interesting effects. We will then use this model in the next two assignments to build a full MVC application.

1 -- About Images

Many applications use color images. A good number of these provide a way to change their appearance in different ways. For example, Instagram has "filters" that convert a picture into something more interesting. They do this by editing the colors of individual dots in the image (called *pixels*). Each pixel has a position in the image *(row, column)* and a color.

Colors on the computer are stored using three numbers: red, green, blue. Any color is a combination of these three *base* colors. To see how various colors can be made using these base colors, check out [this color picker (Links to an external site.)](https://image-color.com/color-picker.html#DA4F4F). You can use the slide bars to pick different colors and observer their red, green, and blue values change (this particular website also shows other color scales that other systems can use). This can provide you with an intuition for how this works.

Each of the three base colors -- red, green, and blue -- is called a *channel* in the image (i.e. a color image has 3 channels, ignoring transparent images). Each channel is usually stored using 8-bits, that is 256 distinct values. This is where the name *24-bit image* comes from (three 8-bit channels)!

2 -- Loading and Saving Images

In this assignment, we will be using a standard representation where an image can be thought of as simply as a 3D array of integers with *rows = height*, *columns = width* and *depth = 3*. With 8-bit channels, each value is between 0 and 255. We are providing [this ImageUtilities class](https://northeastern.instructure.com/courses/63372/files/8852655?wrap=1)[download](https://northeastern.instructure.com/courses/63372/files/8852655/download?download_frd=1)for you to use (but not modify) which contains methods that allow you to read and write actual image files in a variety of standard image formats (jpg, bmp, png, etc) using 3D arrays since you are not allowed to use the image classes in the Java class libraries.

3 -- Filtering Images

A basic operation in many image processing algorithms is filtering. A filter has a "kernel", which is a 2D array of numbers, having odd dimensions (3x3, 5x5, etc.). Given a pixel in the image and a channel, the result of the filter can be computed for that pixel and channel.

As an example, consider a 3x3 kernel being applied for the pixel at (5,4) of the red channel of an image (row 5, column 4). We place the center of the kernel at the particular pixel (e.g. kernel[1][1] overlaps pixel (5,4) in channel 0. This aligns each number in the kernel with a corresponding number in that channel (kernel[0][0] aligns with pixel (4,3), kernel[1][2] aligns with pixel(5,5), etc.). The result of the filter is calculated by multiplying together corresponding numbers in the kernel and the pixels and adding them. If some portions of the kernel do not overlap any pixels, those pixels are not included in the computation.

Many image processing operations can be framed in terms of filtering (i.e. they are a matter of designing an appropriate kernel and filtering the image with it).

3.1 -- Image blur

Blurring an image is probably the simplest example of filtering. We can use a 3x3 filter as follows:

Blurring can be done by applying this filter to every channel of every pixel to produce the output image. Here is an example of blurring an image this way:

The filter can be repeatedly applied to an image to blur it further. The last image above shows the result of applying the same filter to the second image.

3.2 -- Image sharpening

Image sharpening is in some ways, the opposite of blurring. Sharpening accentuates edges (the boundaries between regions of high contrast), thereby giving the image a "sharper" look.

Sharpening is done by using the following filter:

⎡⎣⎢⎢⎢⎢⎢⎢⎢⎢−18−18−18−18−18−18141414−18−1814114−18−18141414−18−18−18−18−18−18⎤⎦⎥⎥⎥⎥⎥⎥⎥⎥[−18−18−18−18−18−18141414−18−1814114−18−18141414−18−18−18−18−18−18]

As with blurring, it is possible to repeatedly sharpen the image.

3.3 -- Clamping values

Often filtering results in an image where some values are beyond their range. For example, using 8 bits per channel means each value in each channel is between 0 and 255. Filtering such an image may cause some values to be outside this range. We must "clamp" these values to avoid overflow and underflow while saving, and to display such images properly. Clamping requires two values: the permissible minimum and maximum. Then each value in an image that is lesser than the minimum is assigned to the minimum, and each value greater than the maximum is assigned to the maximum.

Clamping is implemented as the last step of any filtering operation, to ensure that the resulting image can be properly saved and displayed.

4 -- Color transformations

Filtering modifies the value of a pixel depending on the values of its neighbors. Filtering is applied separately to each channel. In contrast, a color transformation modifies the color of a pixel based on its own color. Consider a pixel with color (r,g,b)(r,g,b). A color transformation results in the new color of this pixel to be (r′,g′,b′)(r′,g′,b′) such that each of them are dependent only on the values (r,g,b)(r,g,b).

A simple example of a color transformation is a linear color transformation. A linear color transformation is simply a color transformation in which the final red, green and blue values of a pixel are linear combinations of its initial red, green and blue values. For example, if the initial color is (r,g,b)(r,g,b), then the final red value being  0.3r+0.4g+0.6b0.3r+0.4g+0.6b is a linear color transformation.

Linear color transformations can be succinctly represented in matrix form as follows:

⎡⎣⎢r′g′b′⎤⎦⎥=⎡⎣⎢a11a21a31a12a22a32a13a23a33⎤⎦⎥∗⎡⎣⎢rgb⎤⎦⎥[r′g′b′]=[a11a12a13a21a22a23a31a32a33]∗[rgb]

where aijaij are numbers. Using the above notation, the final values will be:

r′=a11r+a12g+a13br′=a11r+a12g+a13b

g′=a21r+a22g+a23bg′=a21r+a22g+a23b

b′=a31r+a32g+a33bb′=a31r+a32g+a33b

Similar to filtering, clamping may be necessary after a color transformation to save and display an image properly.

There are many image processing operations that can be expressed as color transformations on individual pixels (i.e. implementing them is a matter of using the correct numbers aijaij on all pixels of the image).

4.1 -- Greyscale

A simple operation is to convert a color image into a greyscale image. A greyscale image is composed only of shades of grey (if the red, green and blue values are the same, it is a shade of grey). There are many ways to convert a color image into greyscale. A simple way is to use the following color transformation: r′=g′=b′=0.2126r+0.7152g+0.0722br′=g′=b′=0.2126r+0.7152g+0.0722b. This is a standard formula to compute the "luma" for high-definition television [read more about it here (Links to an external site.)](https://en.wikipedia.org/wiki/Rec._709)). This can be expressed as a color transformation in matrix form as:

⎡⎣⎢r′g′b′⎤⎦⎥=⎡⎣⎢0.21260.21260.21260.71520.71520.71520.07220.07220.0722⎤⎦⎥∗⎡⎣⎢rgb⎤⎦⎥[r′g′b′]=[0.21260.71520.07220.21260.71520.07220.21260.71520.0722]∗[rgb]

4.2 -- Sepia tone

Photographs taken in the 19th and early 20th century had a characteristic reddish brown tone. This is referred to as a sepia tone (see the [origin of this term (Links to an external site.)](https://en.wikipedia.org/wiki/Sepia_(color))). Most image processing programs allow an operation to convert a normal color image into a sepia-toned image. This conversion can be done using the following color transformation:

⎡⎣⎢r′g′b′⎤⎦⎥=⎡⎣⎢0.3930.3490.2720.7690.6860.5340.1890.1680.131⎤⎦⎥∗⎡⎣⎢rgb⎤⎦⎥[r′g′b′]=[0.3930.7690.1890.3490.6860.1680.2720.5340.131]∗[rgb]

5 -- Reducing color density

One of the ways in which we want to transform the colors in an image is to reduce the number of colors in the image. In our current representation, each channel is represented by 256 values that vary between 0 and 255. This means that there are over 16 million (256 \* 256 \* 256 = 16,777,216) possible colors!.  But what if do not have that many colors at your disposal?

What you need is some sort of mapping, that maps pixels with 16 million possible colors, to say 8 possible colors. One way to do this would be to calculate which 8 colors are most similar to the pixel's color and use this similarity for mapping. This works, but does not really maintain the *essence* of the image.

EDITTED FOR CLARIFICATION:

We have modified this part of the description because of the confusion and also because the original images had a bug in them (yes, even I have bugs in my code sometimes :). There has been two possible ways to interpret 8 colors.  If we take 8 colors to mean that we have 8 possible values per channel, then we have 8 \* 8 \* 8 = 512 colors.  However, if we mean 8 possible colors in total, we would really have only 2 possible values per channel giving you 2 \* 2 \* 2 = 8 total colors.  You can choose to interpret this either way in your implementation.

Either way, the technique for preserving the essence of the image breaks down an image that has many colors into an image that is made of dots from just a few colors is known as *dithering.*A popular technique to dither an image is the [Floyd-Steinberg algorithm (Links to an external site.)](https://en.wikipedia.org/wiki/Floyd%E2%80%93Steinberg_dithering).

To apply this algorithm to an image:

for each position (r,c) in image (traversing row-wise) :

old-color = red-component of pixel (r,c) //or green, or blue

new-color = find-closest-palette-color(old-color)

error = old-color - new-color

set red component pixel(r,c) to new-color

add (7/16 \* error) to pixel on the right (r,c+1)

add (3/16 \* error) to pixel on the next-row-left (r+1,c-1)

add (5/16 \* error) to pixel below in next row (r+1,c)

add (1/16 \* error) to pixel on the next-row-right (r+1,c+1)

Applying this algorithm to different images then becomes a function of how to *find the closest palette color*. When dealing with grayscale images, this is simply choosing between 0 and 255 (which is closer). For color images, we can perform a simple rounding:

find-closest-palette-color(old-color) = round(old-pixel / max-number-of-colors-in-channel)

As with other color transformations, clamping may be necessary after dithering to save and display an image properly.

Version 2 -- Cross-Stitch Controller

[Cross-stitch (Links to an external site.)](https://en.wikipedia.org/wiki/Cross-stitch) is a very old form of counted thread embroidery which has been found all over the world since the middle ages. It is one of the easiest forms of hand embroidery to learn. It gets it's name from the X-shaped stitches that are done on an even and open weave fabric. A pattern consists of a chart that tells you everything you need to know about where to stitch and what color to use:

(Image from [https://stitchedmodern.com/blogs/news/a-beginners-guide-to-cross-stitch# (Links to an external site.)](https://stitchedmodern.com/blogs/news/a-beginners-guide-to-cross-stitch))

In this example, the pattern uses both a color and a symbol to tell you which color thread (called *floss*) to use. In this project, we will extend the image model that we developed in the last project by adding to two algorithms for breaking down an image into "chunks", and then use one of these to develop a controller for creating cross-stitch patterns.

1 -- Image Mosaic

One such image "chunking" approach that you may have seen is an effect that gives a mosaic effect. [Mosaics (Links to an external site.)](https://en.wikipedia.org/wiki/Mosaic) are a popular art form where small pieces irregularly shaped pieces of color stone, glass, or ceramic are used to construct an image (check out [Mosaic Park in Vancouver, Canada (Links to an external site.)](https://www.insidevancouver.ca/2010/08/27/hidden-art-in-the-ground-mosaic-park/)).

To simulate mosaics on a computer, an image can be "broken down" into such shapes, by choosing a set of points in the image (called *seeds*). Each pixel in the image is then paired to the seed that is closest to it (by distance). This creates a cluster of pixels for each seed. The color of each pixel in the image is replaced with the color of its seed pixel.

The seeds can be chosen in a number of ways. The simplest method (that you will implement) is to choose the seeds randomly (i.e., choose random pixel locations from the image).

2 -- Image Pixelation

Another way to chunk an image is by chunking the image into regular squares across the rows and columns. This chunking method produces what many think of as an equivalent to *pixelating* the image. The big difference between this effect and resizing the image is that the resulting image is the same size as the original image, and the color of each pixel in the original image is replaced with the color of the pixel at the center of its square creating *super-pixels*

To do this we decide how many super-pixels we want to have across the width of the image and then create evenly sized squares across. Each super-pixel should be no more than one pixel different in width and/or height of another super-pixel in the image.

3 -- Pattern Generation

Generated chunked images makes it possible to convert an image that has many pixels into one that has fewer pixels without actually changing the number of colors that the image uses. This comes in handy when we want to create a cross-stitch pattern from an image.

To create a cross-stitch patterns for an image, start by pixelating the image and then map each super-pixel in the image to a floss color by calculating the "closest" color of available floss. Use this [DMC floss to RBG value conversion chart (Links to an external site.)](http://my.crazyartzone.com/dmc.asp) to do this.

3.1 -- Calculating Closest Color

The [distance between two colors (Links to an external site.)](https://en.wikipedia.org/wiki/Color_difference#cite_note-euc-1) can be measured in a variety of ways, but not all of them are created equal. For example, using Euclidean distance is one way but it often suffers from color distortion similar to the way grayscale did because of the way that humans perceive color. To this end, we will a better approximation called the *redmean*:

  where

and R, G, and B represent the values between 0-255 for each channel respectively.

3.2 -- The Pattern Specification

The result of creating a image pattern is text. Each pattern should have 3 parts:

1. The *size* of the pattern expressed a the width and height
2. An *image map* that consists of an n x m grid of symbols where each symbol represents the single floss color that should be used to stitch that super-pixel.
3. A *legend* that lists the DMC floss used in the pattern in numeric order along with the symbol used to represent that floss color in the image map.

We provide [this example pattern](https://northeastern.instructure.com/courses/63372/files/8439151?wrap=1)[download](https://northeastern.instructure.com/courses/63372/files/8439151/download?download_frd=1)as an example of the pattern for the image at the very top of this project description. Your patterns do not need to look exactly like this but it does need to have all three parts.

4 -- Batch Controller

Until now the *driver* of your program was a meaningless main method that did its best to demonstrate how to use your model. In this assignment we want to create a *controller* that will respond to user commands. One way to do this is to prompt the user for input from the keyboard and process each command one at a time. While this works, a better way to deal with user input would be to allow the user to load an image, apply various effects to it, and save the result by placing the input in a *batch file*. For example:

load goat.png

dither 2

save goat-dither.png

load goat.png

blur

save goat-blur.png

load goat.png

pixelate 50

pattern

save goat-pattern.txt

...

Version 3 -- Cross Stitch Application

In this final project, you will build a *view* for your cross-stitch pattern generator featuring a graphical user interface. In this application, the user will be ale to load, process, and save images and use these images to generate cross stitch patterns. In addition to a program that the user can interact with, they should also be able to use batch scripting from the previous project.

1 -- The View

A view is the part of the program that interacts with the user. Views can be non-interactive as well as interactive. In this project, you will build an interactive view with a graphical user interface. While the choices and behavior are up to you, your graphical user interface should have the following characteristics and obey the following constraints:

1. You must use Java Swing to build your graphical user interface. Besides the features from lecture, the provided code example illustrates some other features of Swing that you may find useful. You should also checkout the visual guides that are linked the Resources for Module 11 page.
2. The user interface must expose all the required image-processing and cross-stitch features of the program via a [menu (Links to an external site.)](https://docs.oracle.com/javase/tutorial/uiswing/components/menu.html). This includes blur, sharpen, greyscale, sepia, dithering, mosaics, pixelation, and the ability to create cross-stitch patterns, as well as load images and save both images and patterns. A subset of the operations may also be exposed in other ways (e.g., toolbars, etc.)
3. The user should see the image that is being processed on the screen. If the image is bigger than the area allocated to it in your graphical user interface, then the user should be able to scroll the image.
4. The user interface must also be able to display cross-stitch patterns using the DMC floss colors (rather than the original image colors) to the screen.
5. The user interface should provide the ability to exchange one color for another in a cross-stitch pattern by clicking on the color in a displayed pattern and allowing the user to select a different color from the DMC color options.
6. The user interface should also provide the ability to pick one color in a cross-stitch pattern that will then be removed from the pattern completely. The pixels of that color would be replaced with a blank pixel (one in which there is not stitch -- e.g. the "." in the example from Project 4)
7. The user should be able to specify the image to be loaded and saved. That is, the program cannot assume a hard-coded file or folder to load and save. Check out the [JFileChooser (Links to an external site.)](https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/JFileChooser.html) for this.
8. The user interface must provide a way for a user to interactively create and execute a batch-script as part of the program (in the same format you designed in the earlier iteration).
9. When the user specifies an operation, its result should be visible to the user.

Each user interaction or user input must be reasonably user-friendly (e.g. making the user wait for something without a message is poor UI design). We do not expect snazzy, sophisticated user-friendly programs. Our standard is: can a user unfamiliar with your code and technical documentation operate the program correctly without reading your code and technical documentation?

**Hint:** You might also want to check out the [JOptionPane (Links to an external site.)](https://docs.oracle.com/en/java/javase/11/docs/api/java.desktop/javax/swing/JOptionPane.html) class for making easy pop-up messages.