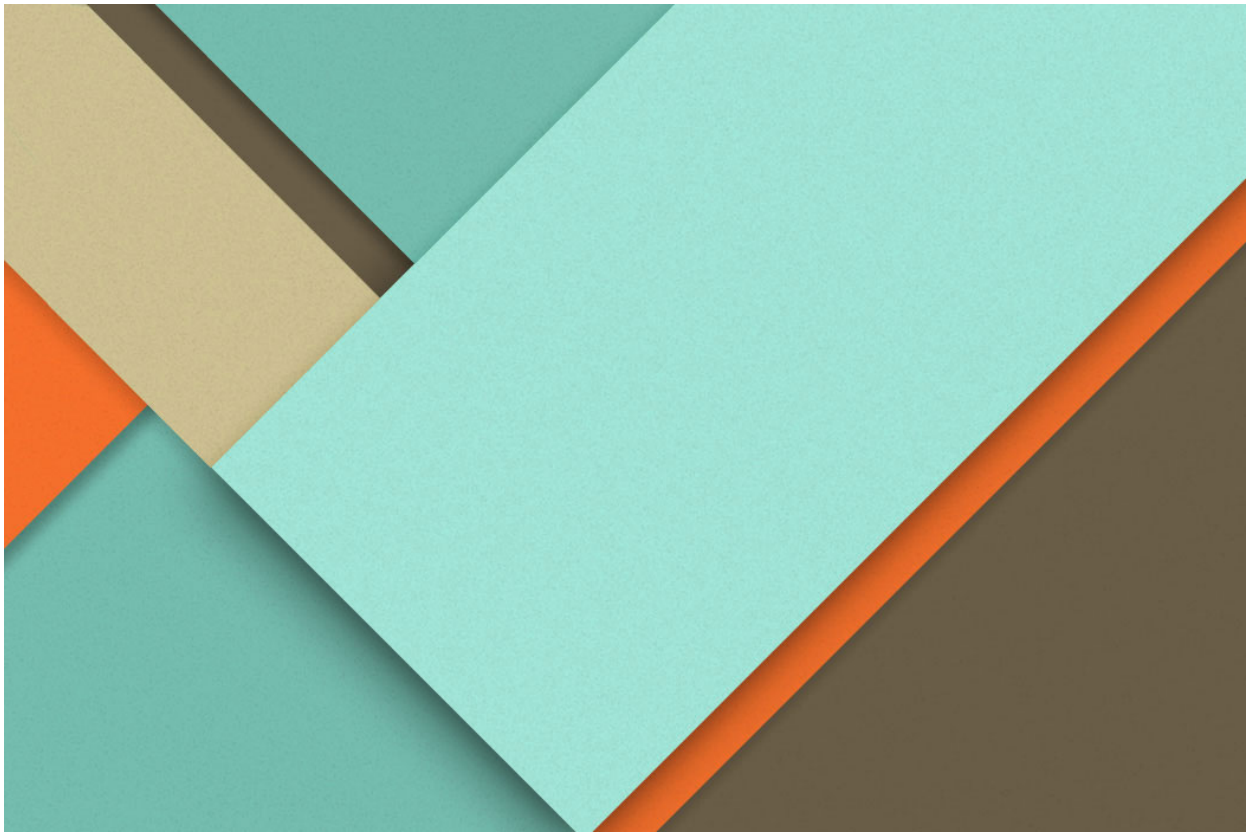

Image Effect Application

02.01.2024

Sai Venkata Sohith Gutta (IMT2022042)
R Lakshman (IMT2022090)
Margasahayam Venkatesh Chirag (IMT2022583)
Siddharth Reddy Maramreddy (IMT2022031)

Setup (for mac)

Backend

C++

1) Go to the Libraries folder in terminal.
2) Run the command ‘make clean’
3) Run the command ‘make’

Java

1) Open pom.xml file using Intellij.
2) Mark the Libraries folder as sources root.
3) Run the file ImageEffectApplication.java

Frontend

1) Go to project folder in terminal.
2) Run the command ‘npm i’
3) Go to ImageEffectFrontend folder in terminal.
4) Run the command ‘npm i’
5) Run the command ‘npm start’

Brightness (by Sai Venkata Sohith Gutta)

In Brightness.cpp, the logic has been implemented wherein anything left to the slider's centre reduces brightness while sliding it the other way increases brightness. This logic has been implemented by just adding a constant to RGB values.

In effectImplemention, brightness.java file is created wherein it implements the SingleValueParameterizableEffect in which it assigns parameter values and handles exceptions and also has a method to add logs.

In PhotoEffectService, a Brightness object is created and methods are used to apply the effect and create a log.

Sepia (by Sai Venkata Sohith Gutta)

In Sepia.cpp, the values of new pixels are calculated by the given formula in code which finally creates a reddish-brown effect.

In effectImplementation, sepia.java file is created wherein it implements PhotoEffect which has a method to add a log.

In PhotoEffectService, a Sepia object is created and methods are called to apply the effect and to create a log.

Gaussian Blur (by Sai Venkata Sohith Gutta)

In GaussianBlur.cpp, Blur effect logic is implemented by first creating a 1-D kernel matrix and then doing convolution along both x and y-axis pixels independently with this 1-D kernel matrix. The resulting pixels have been normalized to create a perfect blur(with no black edges etc).

In effectImplementation, gaussianblur.java file is created wherein it implements SingleValueParameterizableEffect in which it assigns parameter values and handles exceptions and also has a method to add logs.

In PhotoEffectService, a GaussianBlur object is created and methods are used to apply the effect and create a log.

Flip (by Margasahayam Venkatesh Chirag)

In Flip.cpp, Flip effect logic is implemented by first checking if we want to perform horizontal flip, vertical flip, both or no flip. If we don't want to flip(hf=vf=0), then return the original image. If we want to perform a horizontal flip(hf=1,vf=0), we flip each row for all columns. If we want to perform a vertical flip(vf=1,hf=0), we start traversing the original image from the last row to the top row. If we want to perform horizontal and vertical flips (hf=vf=1), we first perform vertical flips and then we perform horizontal flips (flips' processes have been explained above).

In EffectImplementation, Flip.java file is created wherein it implements SingleValueDiscreteEffect, in which it assigns parameter values and handles exceptions and also has a method to add logs.

In PhotoEffectService, a Flip object is created and methods are called to apply the effect and to create a log.

Colour inversion (by Margasahayam Venkatesh Chirag)

In Invert.cpp, Invert effect logic is implemented. Consider a cell (in the image matrix image) image[i][j]. We invert it by doing RGB(image[i][j])=255-RGB(image[i][j]). This inverts the RGB value at that pixel(i.e., cell in the image matrix).

In EffectImplementation, Inversion.java file is created wherein it implements PhotoEffect, in which it has a method to add logs.

In PhotoEffectService, Inversion object is created and methods are used to apply the effect and create a log.

Grayscale (by Margasahayam Venkatesh Chirag)

In Grayscale.cpp, Grayscale effect logic is implemented. The weight for red is 0.299/1=0.299, the weight for green is 0.587/1=0.587, weight for blue is 0.0.114/1=0.114. Note that 0.299+0.587+0.114=1. The weights 0.299, 0.587 and 0.114 are derived based on the luminance values normalized to the maximum intensity of each channel. So, the formula is:

RGB(image[i][j])= 0.299* R(image[i][j])+ 0.587* G(image[i][j])+0.114*B(image[i][j])

In EffectImplementation, Grayscale.java file is created wherein it implements PhotoEffect, in which it has a method to add logs.

In PhotoEffectService, a Grayscale object is created and methods are used to apply the effect and create a log.

Contrast (by R Lakshman)

In Contrast.cpp, contrast effect logic is implemented. We iterate through each cell and apply the general formula for contrast-> New Pixel Value=(Old Pixel Value−Middle Value)×Contrast Factor+Middle Value, where the middle value is 127.

In EffectImplementation, Contrast.java is created where it implements SingleValueParameterizableEffect in which it assigns parameter values and handles exceptions and also has a method to add logs. Here the parameter value is adjusted so that the contrast factor is from 0.5 to 2.5 (reasonable range).

In PhotoEffectService, a Contrast object is created and methods are called to apply the effect and to create a log.

Sharpen (by R Lakshman)

In sharpen.cpp, sharpen effect logic is implemented. We consider each pixel [i,j] at the centre and its neighbours (from i-1, j-1 to i+1, j+1) to get image matrix in which convolution is applied with a specialised kernel matrix which we get by applying the general sharpening formula (F(x,y) = f(x,y)+k*(f(x,y)-Fblur(x,y)) where we used the box blur kernel and identity kernel in place of f and Fblur respectively. We adjusted the sharpen value (parameter) such that it is now from 0 to 10 (which is a reasonable range for the sharpening factor). We handled 8 edge cases (4 sides and 4 corners) by extension.

In EffectImplementation, Sharpen.java is created where it implements SingleValueParameterizableEffect in which it assigns parameter values and handles exceptions and also has a method to add logs.

In PhotoEffectService, a Sharpen object is created and methods are called to apply the effect and to create a log.

Rotation (by Siddharth Reddy Maramreddy)

In Rotation.cpp, for 90° rotation the matrix is rotated 90° once, for 180° rotation the matrix is rotated 90° twice and for 270° rotation the matrix is rotated 90° thrice. The rotation is in the clockwise direction.

In effectImplementation, Rotation.java file is created wherein it implements SIngleValueDiscreteEffect, in which it assigns parameter values and handles exceptions and also has a method to add logs.

In PhotoEffectService, a Rotation object is created and methods are called to apply the effect and to create a log.

Hue and Saturation (by Siddharth Reddy Maramreddy)

In HueSaturation.cpp, RGB is converted to HSV (Hue Saturation Value) for each pixel using the formulas from the given websites below and then the values of Hue and Saturation are increased. Then the HSV values are converted back to RGB values.

In effectImplementation, HueSaturation.java file is created wherein it implements ParameterizableEffect, in which it assigns parameter values and handles exceptions and also has a method to add logs.

In PhotoEffectService, a HueSaturation object is created and methods are used to apply the effect and create a log.

HSV to RGB
RGB to HSV

Dominant Colours (by Siddharth Reddy Maramreddy)

In DominantColour.cpp, k-means clustering is used to find n (n is set to 10) dominant colours in an image. The output image will display 10 dominant colours of the image.

In effectImplementation,DominantColour.java file is created wherein it implements PhotoEffect, in which it has a method to add logs.

In PhotoEffectService, the DominantColour object is created and methods are used to apply the effect and create a log.

K-Means Clustering Algorithm

Logging Service (by Siddharth Reddy Maramreddy)

addLog

Checks if file exists and creates a new file if it doesn't exist. Then a timestamp with current date and time is obtained. Then an entry is made to the file in the specified format.

getAlllogs

Reads each entry from the file, creates a Log object for each entry and stores it in an ArrayList. Finally, it returns the ArrayList.

getLogsByEffect

Reads each entry from the file and checks its effect field. If the effect matches the required effect name, it creates a Log object for the entry and stores it in an ArrayList. Finally, it returns the ArrayList.

clearLogs

Deletes the file.