Programming-2 Project

Team Kaalkoot

January 2024

1 Introduction

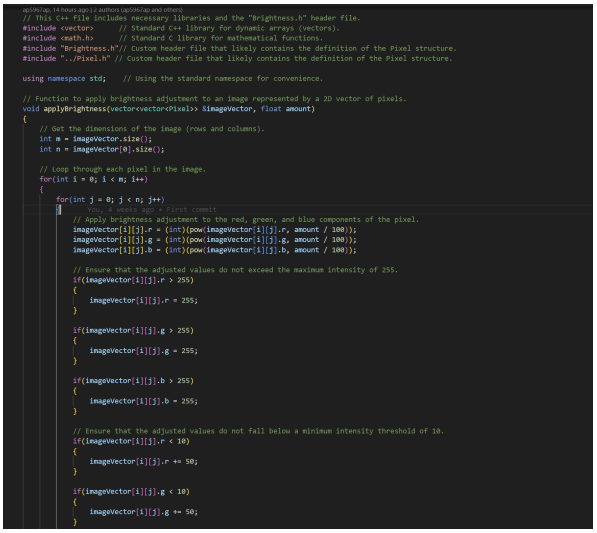
This project is done under Prof. Vivek Yadav as a part of the course offered at IIITB Programming-2 which is all about Object Oriented Programming(OOPS) in Java and C++. This project is a full-fledged web app which implements the very basics of OOPS in the backend. The project is about Image Processing with the image processing portion done in CPP in order to provide speed and the interfaces are in Java to provide better interactivity and abstraction between the backend and the frontend. The backend is written in Spring Boot which is a Java Framework. The frontend portion of the project is in Angular.
The motivation behind this project was to teach practical usage of OOPS in real life. Therefore, the basic frontend structure and backend skeleton were provided to us by the Prof himself. There is a proper folder structure maintained, which separates the call to function and the function itself into two different folders. The main function definition is written under the Library directory, and all the calls to the functions and the logs are maintained under the src directory. The requirements and the Spring Boot .yaml file are also under src directory itself.

2 Code Explanation

2.1 Libraries

This section contains screenshots of the different libraries that were built to process the image efficiently without calling the built-in libraries.

2.1.1 Brightness Library

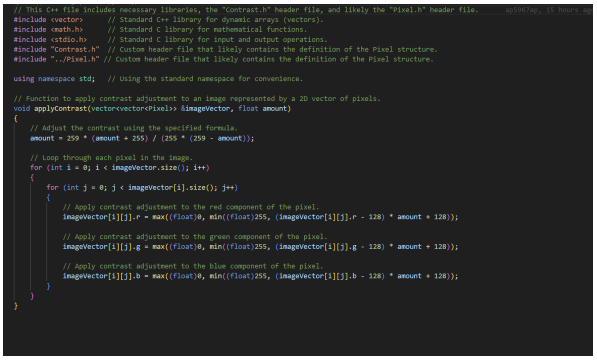


```cpp
// This C++ file includes necessary libraries and the "Brightness.h" header file.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include <math.h>       // Standard C library for mathematical functions.
#include "Brightness.h" // Custom header file that likely contains the definition of the Pixel structure.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to apply brightness adjustment to an image represented by a 2D vector of pixels.
void applyBrightness(vector<vector<Pixel>> &imageVector, float amount)
{
    // Get the dimensions of the image (rows and columns).
    int m = imageVector.size();
    int n = imageVector[0].size();

    // Loop through each pixel in the image.
    for(int i = 0; i < m; i++)
    {
        for(int j = 0; j < n; j++)
        {   You, 4 weeks ago • First commit
            // Apply brightness adjustment to the red, green, and blue components of the pixel.
            imageVector[i][j].r = (int)(pow(imageVector[i][j].r, amount / 100));
            imageVector[i][j].g = (int)(pow(imageVector[i][j].g, amount / 100));
            imageVector[i][j].b = (int)(pow(imageVector[i][j].b, amount / 100));

            // Ensure that the adjusted values do not exceed the maximum intensity of 255.
            if(imageVector[i][j].r > 255)
            {
                imageVector[i][j].r = 255;
            }

            if(imageVector[i][j].g > 255)
            {
                imageVector[i][j].g = 255;
            }

            if(imageVector[i][j].b > 255)
            {
                imageVector[i][j].b = 255;
            }

            // Ensure that the adjusted values do not fall below a minimum intensity threshold of 10.
            if(imageVector[i][j].r < 10)
            {
                imageVector[i][j].r += 50;
            }

            if(imageVector[i][j].g < 10)
            {
                imageVector[i][j].g += 50;
            }
```

Figure 1: Brightness Library

The brightness backend is based on Luminosity Model. It takes the amount by which the brightness is to be increased and then the brightness is increased by the 100th power of the amount. The maximum and the lowest value of the brightness values of r,g and b are adjusted.

2.1.2 Contrast Library



```cpp
// This C++ file includes necessary libraries, the "Contrast.h" header file, and likely the "Pixel.h" header file.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include <math.h>       // Standard C library for mathematical functions.
#include <stdio.h>      // Standard C library for input and output operations.
#include "Contrast.h"  // Custom header file that likely contains the definition of the Pixel structure.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;   // Using the standard namespace for convenience.

// Function to apply contrast adjustment to an image represented by a 2D vector of pixels.
void applyContrast(vector<vector<Pixel>> &imageVector, float amount)
{
    // Adjust the contrast using the specified formula.
    amount = 259 * (amount + 255) / (255 * (259 - amount));

    // Loop through each pixel in the image.
    for (int i = 0; i < imageVector.size(); i++)
    {
        for (int j = 0; j < imageVector[i].size(); j++)
        {
            // Apply contrast adjustment to the red component of the pixel.
            imageVector[i][j].r = max((float)0, min((float)255, (imageVector[i][j].r - 128) * amount + 128));

            // Apply contrast adjustment to the green component of the pixel.
            imageVector[i][j].g = max((float)0, min((float)255, (imageVector[i][j].g - 128) * amount + 128));

            // Apply contrast adjustment to the blue component of the pixel.
            imageVector[i][j].b = max((float)0, min((float)255, (imageVector[i][j].b - 128) * amount + 128));
        }
    }
}
```

Figure 2: Contrast Library

The Contrast is simply implemented by adjusting the value of amount and then adding it to the current r,b and b values.

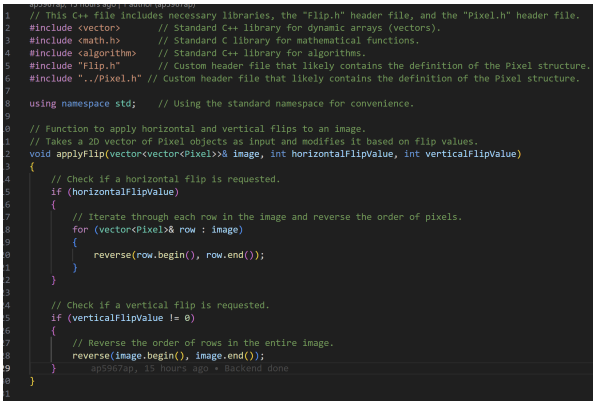
2.1.3 Dominant Color Library

```cpp
// This C++ file includes necessary libraries, the "DominantColour.h" header file, and the "Pixel.h" header file.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include <map>          // Standard C++ library for associative containers (map).
#include <math.h>       // Standard C library for mathematical functions.
#include <algorithm>    // Standard C++ library for algorithms.
#include "DominantColour.h" // Custom header file that likely contains the definition of the Pixel structure.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to apply the dominant color of an image to all pixels in the image.
void applyDominantColour(vector<vector<Pixel>>& image)
{
    // Create a vector to count occurrences of each color in the image.
    vector<int> colorCount(255 << 16 | 255 << 8 | 255, 0);

    // Iterate through each pixel in the image and update the color count vector.
    for (int i = 0; i < image.size(); i++)
    {
        for (int j = 0; j < image[i].size(); j++)
        {
            int colorKey = (image[i][j].r << 16) | (image[i][j].g << 8) | image[i][j].b;
            colorCount[colorKey]++;
        }
    }

    // Find the color with the maximum occurrence in the image.
    int dominantColor = max_element(colorCount.begin(), colorCount.end()) - colorCount.begin();

    // Create a Pixel object representing the dominant color.
    Pixel dominantPixel;
    dominantPixel.r = (dominantColor >> 16) & 0xFF;
    dominantPixel.g = (dominantColor >> 8) & 0xFF;
    dominantPixel.b = dominantColor & 0xFF;

    // Update all pixels in the image to have the dominant color.
    for (int i = 0; i < image.size(); i++)
    {
        for (int j = 0; j < image[i].size(); j++)
        {
            image[i][j] = dominantPixel;
        }
    }
}
```

Figure 3: Dominant Color Library

This library counts the number of different r,g and b values and stores them in a vector. Then the combination of maximum red, green and blue gives the dominant color.

2.1.4 Flip Library

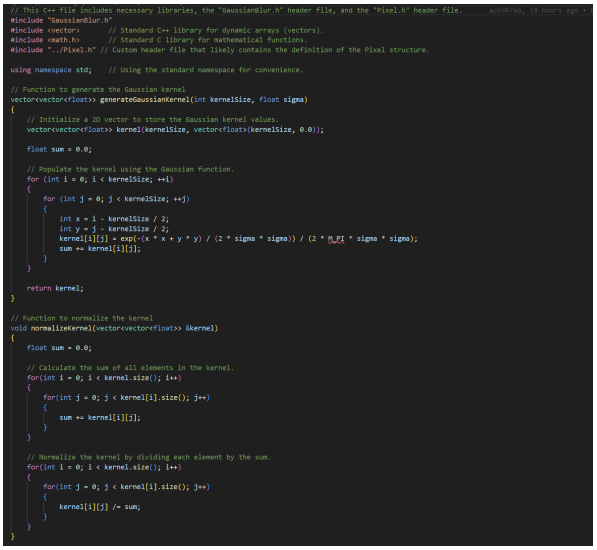


```cpp
// This C++ file includes necessary libraries, the "Flip.h" header file, and the "Pixel.h" header file.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include <math.h>       // Standard C library for mathematical functions.
#include <algorithm>    // Standard C++ library for algorithms.
#include "Flip.h"       // Custom header file that likely contains the definition of the Pixel structure.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to apply horizontal and vertical flips to an image.
// Takes a 2D vector of Pixel objects as input and modifies it based on flip values.
void applyFlip(vector<vector<Pixel>>& image, int horizontalFlipValue, int verticalFlipValue)
{
    // Check if a horizontal flip is requested.
    if (horizontalFlipValue)
    {
        // Iterate through each row in the image and reverse the order of pixels.
        for (vector<Pixel>& row : image)
        {
            reverse(row.begin(), row.end());
        }
    }

    // Check if a vertical flip is requested.
    if (verticalFlipValue != 0)
    {
        // Reverse the order of rows in the entire image.
        reverse(image.begin(), image.end());
    }      ap5967ap, 15 hours ago • Backend done
}
```

Figure 4: Flip Library

Matrix inversion is used as the logic behind this library.

2.1.5 Gaussian Blur Library

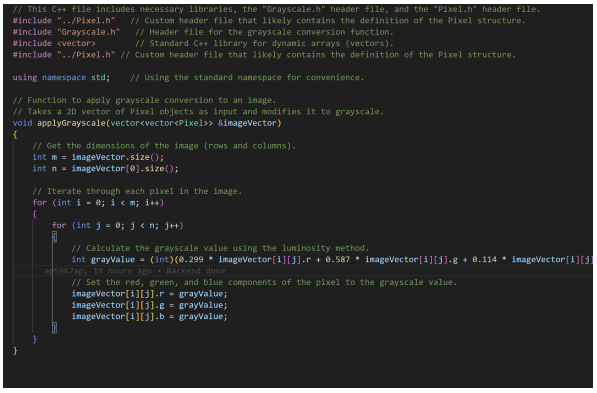


```cpp
// This C++ file includes necessary libraries, the "GaussianBlur.h" header file, and the "Pixel.h" header file.
#include "GaussianBlur.h"
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include <math.h>       // Standard C library for mathematical functions.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to generate the Gaussian kernel
vector<vector<float>> generateGaussianKernel(int kernelSize, float sigma)
{
    // Initialize a 2D vector to store the Gaussian kernel values.
    vector<vector<float>> kernel(kernelSize, vector<float>(kernelSize, 0.0));

    float sum = 0.0;

    // Populate the kernel using the Gaussian function.
    for (int i = 0; i < kernelSize; ++i)
    {
        for (int j = 0; j < kernelSize; ++j)
        {
            int x = i - kernelSize / 2;
            int y = j - kernelSize / 2;
            kernel[i][j] = exp(-(x * x + y * y) / (2 * sigma * sigma)) / (2 * M_PI * sigma * sigma);
            sum += kernel[i][j];
        }
    }

    return kernel;
}

// Function to normalize the kernel
void normalizeKernel(vector<vector<float>> &kernel)
{
    float sum = 0.0;

    // Calculate the sum of all elements in the kernel.
    for(int i = 0; i < kernel.size(); i++)
    {
        for(int j = 0; j < kernel[i].size(); j++)
        {
            sum += kernel[i][j];
        }
    }

    // Normalize the kernel by dividing each element by the sum.
    for(int i = 0; i < kernel.size(); i++)
    {
        for(int j = 0; j < kernel[i].size(); j++)
        {
            kernel[i][j] /= sum;
        }
    }
}
```

Figure 5: Gaussian Blur Library

A kernel moves around the image matrix according to the entered amount, distorting the values of r,g and b of the pixel at that particular matrix point. The logic works by blurring the rectangular portion of a particular width according to the entered amount around the centeral portion of the image.

2.1.6 Grayscale Library

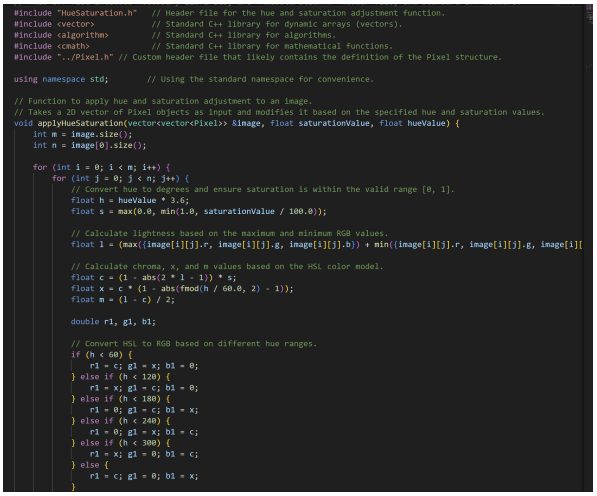


```cpp
// This C++ file includes necessary libraries, the "Grayscale.h" header file, and the "Pixel.h" header file.
#include "../Pixel.h"   // Custom header file that likely contains the definition of the Pixel structure.
#include "Grayscale.h"   // Header file for the grayscale conversion function.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to apply grayscale conversion to an image.
// Takes a 2D vector of Pixel objects as input and modifies it to grayscale.
void applyGrayscale(vector<vector<Pixel>> &imageVector)
{
    // Get the dimensions of the image (rows and columns).
    int m = imageVector.size();
    int n = imageVector[0].size();

    // Iterate through each pixel in the image.
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            // Calculate the grayscale value using the luminosity method.
            int grayValue = (int)(0.299 * imageVector[i][j].r + 0.587 * imageVector[i][j].g + 0.114 * imageVector[i][j]
    ag5967ag, 19 hours ago • Backend done
            // Set the red, green, and blue components of the pixel to the grayscale value.
            imageVector[i][j].r = grayValue;
            imageVector[i][j].g = grayValue;
            imageVector[i][j].b = grayValue;
        }
    }
}
```

Figure 6: Grayscale Library

The logic is standard Grayscale formula.

2.1.7 Hue and Saturation Library

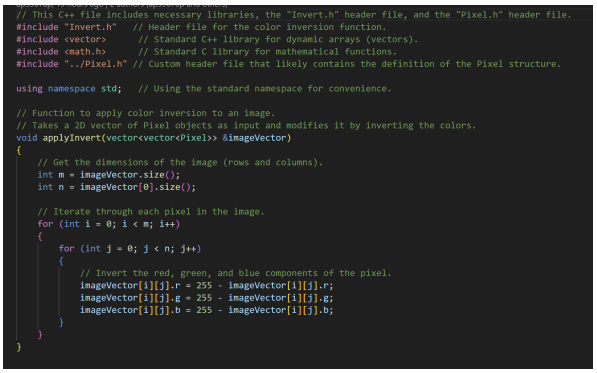


```cpp
#include "HueSaturation.h"   // Header file for the hue and saturation adjustment function.
#include <vector>            // Standard C++ library for dynamic arrays (vectors).
#include <algorithm>         // Standard C++ library for algorithms.
#include <cmath>             // Standard C++ library for mathematical functions.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;        // Using the standard namespace for convenience.

// Function to apply hue and saturation adjustment to an image.
// Takes a 2D vector of Pixel objects as input and modifies it based on the specified hue and saturation values.
void applyHueSaturation(vector<vector<Pixel>> &image, float saturationValue, float hueValue) {
    int m = image.size();
    int n = image[0].size();

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            // Convert hue to degrees and ensure saturation is within the valid range [0, 1].
            float h = hueValue * 3.6;
            float s = max(0.0, min(1.0, saturationValue / 100.0));

            // Calculate lightness based on the maximum and minimum RGB values.
            float l = (max({image[i][j].r, image[i][j].g, image[i][j].b}) + min({image[i][j].r, image[i][j].g, image[i][

            // Calculate chroma, x, and m values based on the HSL color model.
            float c = (1 - abs(2 * l - 1)) * s;
            float x = c * (1 - abs(fmod(h / 60.0, 2) - 1));
            float m = (l - c) / 2;

            double r1, g1, b1;

            // Convert HSL to RGB based on different hue ranges.
            if (h < 60) {
                r1 = c; g1 = x; b1 = 0;
            } else if (h < 120) {
                r1 = x; g1 = c; b1 = 0;
            } else if (h < 180) {
                r1 = 0; g1 = c; b1 = x;
            } else if (h < 240) {
                r1 = 0; g1 = x; b1 = c;
            } else if (h < 300) {
                r1 = x; g1 = 0; b1 = c;
            } else {
                r1 = c; g1 = 0; b1 = x;
            }
```

Figure 7: Hue Saturation Library

The logic is standard HSL to RGB conversion formula where L value is (max(r,g,b)+min(r,g,b)/2 of that pixel and H and S values are provided by the user.

2.1.8 Invert Library

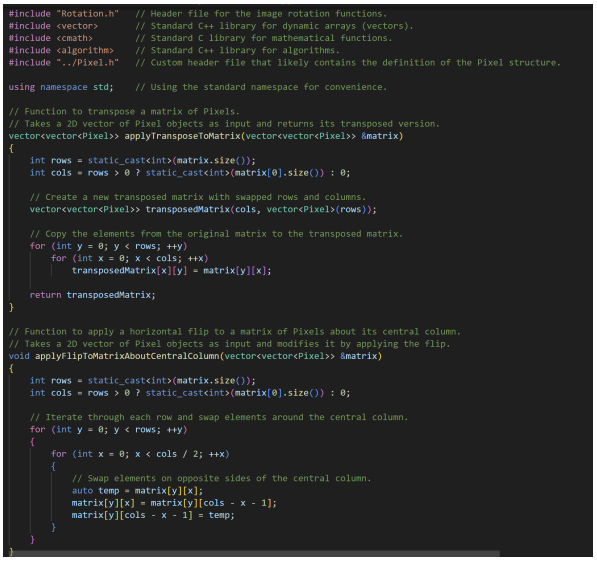


```cpp
// This C++ file includes necessary libraries, the "Invert.h" header file, and the "Pixel.h" header file.
#include "Invert.h"   // Header file for the color inversion function.
#include <vector>     // Standard C++ library for dynamic arrays (vectors).
#include <math.h>     // Standard C library for mathematical functions.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;   // Using the standard namespace for convenience.

// Function to apply color inversion to an image.
// Takes a 2D vector of Pixel objects as input and modifies it by inverting the colors.
void applyInvert(vector<vector<Pixel>> &imageVector)
{
    // Get the dimensions of the image (rows and columns).
    int m = imageVector.size();
    int n = imageVector[0].size();

    // Iterate through each pixel in the image.
    for (int i = 0; i < m; i++)
    {
        for (int j = 0; j < n; j++)
        {
            // Invert the red, green, and blue components of the pixel.
            imageVector[i][j].r = 255 - imageVector[i][j].r;
            imageVector[i][j].g = 255 - imageVector[i][j].g;
            imageVector[i][j].b = 255 - imageVector[i][j].b;
        }
    }
}
```

Figure 8: Invert Library

The logic is simple. 255- RGB value of the pixel.

2.1.9 Rotation Library

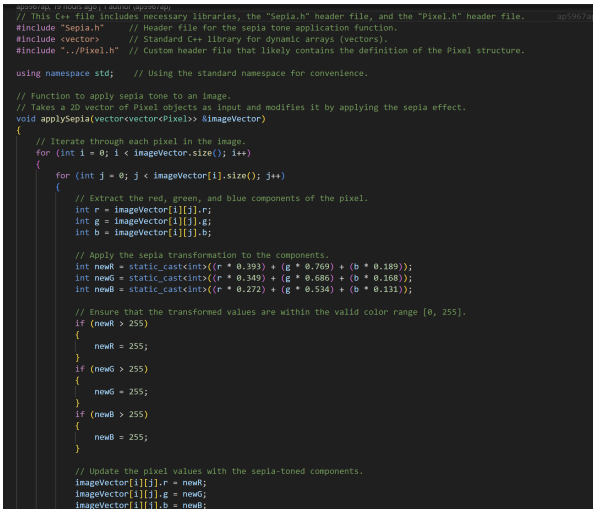


```cpp
#include "Rotation.h"   // Header file for the image rotation functions.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include <cmath>        // Standard C library for mathematical functions.
#include <algorithm>    // Standard C++ library for algorithms.
#include "../Pixel.h"   // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to transpose a matrix of Pixels.
// Takes a 2D vector of Pixel objects as input and returns its transposed version.
vector<vector<Pixel>> applyTransposeToMatrix(vector<vector<Pixel>> &matrix)
{
    int rows = static_cast<int>(matrix.size());
    int cols = rows > 0 ? static_cast<int>(matrix[0].size()) : 0;

    // Create a new transposed matrix with swapped rows and columns.
    vector<vector<Pixel>> transposedMatrix(cols, vector<Pixel>(rows));

    // Copy the elements from the original matrix to the transposed matrix.
    for (int y = 0; y < rows; ++y)
        for (int x = 0; x < cols; ++x)
            transposedMatrix[x][y] = matrix[y][x];

    return transposedMatrix;
}

// Function to apply a horizontal flip to a matrix of Pixels about its central column.
// Takes a 2D vector of Pixel objects as input and modifies it by applying the flip.
void applyFlipToMatrixAboutCentralColumn(vector<vector<Pixel>> &matrix)
{
    int rows = static_cast<int>(matrix.size());
    int cols = rows > 0 ? static_cast<int>(matrix[0].size()) : 0;

    // Iterate through each row and swap elements around the central column.
    for (int y = 0; y < rows; ++y)
    {
        for (int x = 0; x < cols / 2; ++x)
        {
            // Swap elements on opposite sides of the central column.
            auto temp = matrix[y][x];
            matrix[y][x] = matrix[y][cols - x - 1];
            matrix[y][cols - x - 1] = temp;
        }
    }
}
```

Figure 9: Rotation Library

The logic is combination of matrix transpose, matrix inversion and vector reverse in order to achieve 90, 180, 270, 360 or 0 rotation.

2.1.10 Sepia Library



```cpp
// This C++ file includes necessary libraries, the "Sepia.h" header file, and the "Pixel.h" header file.
#include "Sepia.h"      // Header file for the sepia tone application function.
#include <vector>       // Standard C++ library for dynamic arrays (vectors).
#include "../Pixel.h"   // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to apply sepia tone to an image.
// Takes a 2D vector of Pixel objects as input and modifies it by applying the sepia effect.
void applySepia(vector<vector<Pixel>> &imageVector)
{
    // Iterate through each pixel in the image.
    for (int i = 0; i < imageVector.size(); i++)
    {
        for (int j = 0; j < imageVector[i].size(); j++)
        {
            // Extract the red, green, and blue components of the pixel.
            int r = imageVector[i][j].r;
            int g = imageVector[i][j].g;
            int b = imageVector[i][j].b;

            // Apply the sepia transformation to the components.
            int newR = static_cast<int>((r * 0.393) + (g * 0.769) + (b * 0.189));
            int newG = static_cast<int>((r * 0.349) + (g * 0.686) + (b * 0.168));
            int newB = static_cast<int>((r * 0.272) + (g * 0.534) + (b * 0.131));

            // Ensure that the transformed values are within the valid color range [0, 255].
            if (newR > 255)
            {
                newR = 255;
            }
            if (newG > 255)
            {
                newG = 255;
            }
            if (newB > 255)
            {
                newB = 255;
            }

            // Update the pixel values with the sepia-toned components.
            imageVector[i][j].r = newR;
            imageVector[i][j].g = newG;
            imageVector[i][j].b = newB;
```

Figure 10: Sepia Library

The logic used is standard sepia formula used to provide Sepia effect.

2.1.11 Image Sharpening Library

```cpp
#include <algorithm>    // Standard C++ library for algorithms.
#include "../Pixel.h" // Custom header file that likely contains the definition of the Pixel structure.

using namespace std;    // Using the standard namespace for convenience.

// Function to apply a sharpening filter to an image.
// Takes a 2D vector of Pixel objects and a sharpening strength value as input.
void applySharpen(vector<vector<Pixel>> &image, float value)
{
    // 3x3 convolution kernel for sharpening.
    const int kernel[3][3] = {
      {-1, -1, -1},
      {-1,  9, -1},
      {-1, -1, -1}
    };

    int height = image.size();
    int width = image[0].size();

    value = value / 45.0;  // Normalize the sharpening strength value.

    // Create a new 3D vector to store the intermediate results after convolution.
    vector<vector<vector<int>>> newImage(height, vector<vector<int>>(width, vector<int>(3, 0)));

    // Apply convolution using the sharpening kernel.
    for (int i = 1; i < height - 1; ++i) {
        for (int j = 1; j < width - 1; ++j) {
            for (int k = 0; k < 3; ++k) {
                int sum = 0;
                for (int m = -1; m <= 1; ++m) {
                    for (int n = -1; n <= 1; ++n) {
                        // Convolution operation for each color channel (r, g, b).
                        if (k == 0)
                            sum += image[i + m][j + n].r * kernel[m + 1][n + 1];
                        else if (k == 1)
                            sum += image[i + m][j + n].g * kernel[m + 1][n + 1];
                        else if (k == 2)
                            sum += image[i + m][j + n].b * kernel[m + 1][n + 1];
                    }
                }
                newImage[i][j][k] = sum;
            }
        }
    }
```

Figure 11: Image Sharpening Library

This uses kernel convolution with the image matrix and a standard colour conversion logic in order to set the border parameters.

3 Authors

1. Varnit Mittal (IMT2022025)
2. Aditya Priyadarshi (IMT2022075)
3. Hemang Seth (IMT2022098)
4. Tanish Pathania (IMT2022049)
5. Siddharth Vikram (IMT2022534)