Part 1 – Grayscale Photomosaic

Functions Defined:

```
//calculate the brightness of a bitmap
double getTileBrightness(Bitmap& tile)
{
    int tile_w = tile.getWidth();
    int tile_h = tile.getHeight();
    double tile_brightness = 0.0;
    for (int y = 0; y < tile_h; y++) {
        for (int x = 0; x < tile_w; x++) {
            unsigned char R, G, B;
            tile.getColor(x, y, R, G, B);
            tile_brightness += 0.299 * R + 0.587 * G + 0.114 * B;
        }
    }
    return tile_brightness / (double)(tile_w * tile_h);
}
```

This function is to get the average brightness among all pixels in the image.

```
//convert a tile to grayscale and resize it to cell_w and cell_h
void processTile(Bitmap& tile, int cell_w, int cell_h, Bitmap& processed_tile){
    int tile_w = tile.getWidth();
    int tile_h = tile.getHeight();
    processed_tile.create(cell_w, cell_h);
    for (int y = 0; y < cell_h; y++) {
        for (int x = 0; x < cell_w; x++) {
            //mapping
            float fx = x * tile_w / cell_w;//f(x)
            float fy = y * tile_h / cell_h;//f(y)
            //find the four surrounding pixels
            int x1 = floor(fx);
            // int x2 = ceil(fx) > tile_w - 1 ? tile_w - 1 : ceil(fx);
            int x2 = ceil(fx);
            int y1 = floor(fy);
            // int y2 = ceil(fy) > tile_h - 1 ? tile_h - 1 : ceil(fy);
            int y2 = ceil(fy);
```

```
            //border case
            if (x2 == 0 && x1 == 0) x2 += 1;
            else if(x1 == x2) x1 -= 1;
            if (y2 == 0 && y1 == 0) y2 += 1;
            else if(y1 == y2) y1 -= 1;
            Color c1, c2, c3, c4;//BL, TL, BR, TR
            tile.getColor(x1, y1, c1.R, c1.G, c1.B);
            tile.getColor(x1, y2, c2.R, c2.G, c2.B);
            tile.getColor(x2, y1, c3.R, c3.G, c3.B);
            tile.getColor(x2, y2, c4.R, c4.G, c4.B);

            int horizontal_distance = abs(x2-x1);
            int vertical_distance = abs(y2-y1);
            unsigned char upper_weighted_R, upper_weighted_G, upper_weighted_B,
lower_weighted_R, lower_weighted_G, lower_weighted_B;
            //upper weighted average
            upper_weighted_R = (c2.R * (x2-fx) + c4.R * (fx-x1))/horizontal_distance;
            upper_weighted_G = (c2.G * (x2-fx) + c4.G * (fx-x1))/horizontal_distance;
            upper_weighted_B = (c2.B * (x2-fx) + c4.B * (fx-x1))/horizontal_distance;
            //lower weighted average
            lower_weighted_R = (c3.R * (x2-fx) + c1.R * (fx-x1))/horizontal_distance;
            lower_weighted_G = (c3.G * (x2-fx) + c1.G * (fx-x1))/horizontal_distance;
            lower_weighted_B = (c3.B * (x2-fx) + c1.B * (fx-x1))/horizontal_distance;

            unsigned char vertical_weighted_R, vertical_weighted_G, vertical_weighted_B;

            //vertical weighted average
            vertical_weighted_R = (upper_weighted_R * (y2-fy) + lower_weighted_R * (fy-
y1))/vertical_distance;
            vertical_weighted_G = (upper_weighted_G * (y2-fy) + lower_weighted_G * (fy-
y1))/vertical_distance;
            vertical_weighted_B = (upper_weighted_B * (y2-fy) + lower_weighted_B * (fy-
y1))/vertical_distance;

            unsigned char gray = vertical_weighted_R * 0.299 + vertical_weighted_G * 0.587 +
vertical_weighted_B * 0.114;
            processed_tile.setColor(x,y,gray,gray,gray);
        }
    }
```

```
}
```

This function is to implement the Bilinear Interpolation to resize an image into the specified size by obtaining four original pixels surrounding the new point and linearly interpolating the values of the pixels to get the pixel value of the new point. It also sets the pixel values to grayscale using the formula Y’ = 0.299 * R + 0.587 * G + 0.114 * B.

```
double getRegionBrightness(int cell_h, int cell_w, int leftx, int lefty, Bitmap& bmp){
    double region_brightness = 0.0;
    for (int i = 0; i < cell_h; i++) {
        for (int j = 0; j < cell_w; j++) {
            unsigned char R, G, B;
            bmp.getColor(j + leftx, i + lefty, R, G, B);
            region_brightness += 0.299 * R
                + 0.587 * G
                + 0.114 * B;
        }
    }
    return region_brightness /= (cell_h * cell_w);
}
```

This function is for obtaining the average brightness of the specified region cell_h * cell_w.

```
int getClosestDistanceIndex(int num_of_tiles, vector<double> tiles_brightness, double
region_brightness){
    int most_similar = 0;
    for (int i = 1; i < num_of_tiles; i++){
        if (abs(tiles_brightness[i] - region_brightness) < abs(tiles_brightness[most_similar]
- region_brightness))
            most_similar = i;
    }
    return most_similar;
}
```

This function is for obtaining the index of most similar tile in the brightness array.

Read and arrange source image and photo tiles:

```
// Parse output and cell shape specified in argv[3]
     int output_w, output_h, cell_w, cell_h;
     sscanf(argv[3], "%d,%d,%d,%d", &output_w, &output_h, &cell_w, &cell_h);
```

The output width, height and cell width, height are stored into `output_w, output_h,` `cell_w, cell_h`, respectively

```
// Read source bitmap from argv[1]

Bitmap source_bitmap(argv[1]);

int source_w = source_bitmap.getWidth();

int source_h = source_bitmap.getHeight();
```

The original bitmap is saved into source_bitmap.

```
// List .bmp files in argv[2] and do preprocessing

vector<string> list_of_paths;

list_files(argv[2], ".bmp", list_of_paths, false);
```

The full path to the photo tile bitmaps are saved into list_of_paths.

Resizing photo by Bilinear Interpolation and Query for photo tiles with nearest brightness values:

```
// Create output bitmap

Bitmap transformed_bitmap(output_w, output_h);

//resize and convert original bmp into grayscale

processTile(source_bitmap, output_w, output_h, transformed_bitmap);
```

The original bitmap will be resized and in grayscale and saved into transformed_bitmap.

```
int num_of_tiles = list_of_paths.size();

    vector<Bitmap> tiles, processed_tiles;

      vector<double> tiles_brightness;

    //resize all tiles and convert them to grayscale

    for (int i = 0; i < num_of_tiles; i++) {

        Bitmap bmp1,bmp2;

            bmp1.create(list_of_paths[i].c_str());//convert a string object to a char*

            bmp2.create(cell_w, cell_h);

            tiles.push_back(bmp1);

            processed_tiles.push_back(bmp2);

            processed_tiles[i].create(cell_w, cell_h);

        processTile(tiles[i], cell_w, cell_h, processed_tiles[i]);

        //average the brightness

            tiles_brightness.push_back(getTileBrightness(processed_tiles[i]));

    }
```

All the tiles will be processed and saved into processed_tiles. All theirs brightness values will be saved into tiles_brightness for efficiency.

Compose the output image with photo tiles:

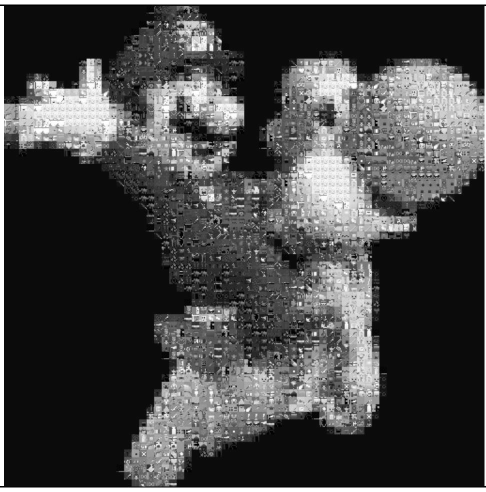
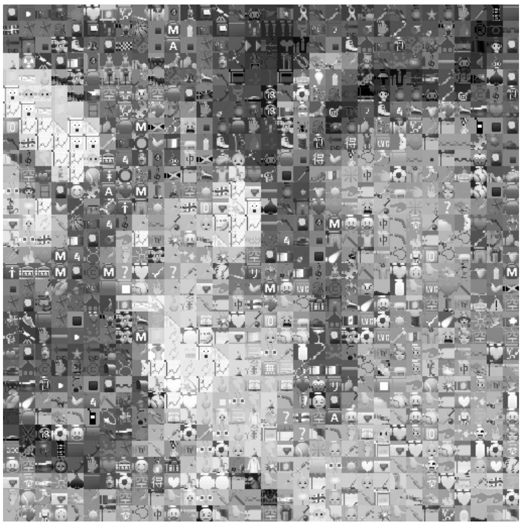
```
    Bitmap output_bitmap(output_w, output_h);

    for (int y = 0; y < output_h; y+=cell_h) {
        for (int x = 0; x < output_w; x+=cell_w) {
            //calculate the region brightness of the transformed bitmap
            double region_brightness = getRegionBrightness(cell_h, cell_w, x, y,
transformed_bitmap);
            int most_similar =
getClosestDistanceIndex(num_of_tiles,tiles_brightness,region_brightness);
            //loop through the tile and copy the tile to one of the regions of the output bmp
            for (int i = 0; i < cell_h; i++) {
                for (int j = 0; j < cell_w; j++) {
                    Color RGB;
                    processed_tiles[most_similar].getColor(j, i, RGB.R, RGB.G, RGB.B);
                    output_bitmap.setColor(x + j , y + i, RGB.R, RGB.G, RGB.B);
                }
            }
        }
    }
    // Save output bitmap to argv[4]
    output_bitmap.save(argv[4]);

    return 0;
```

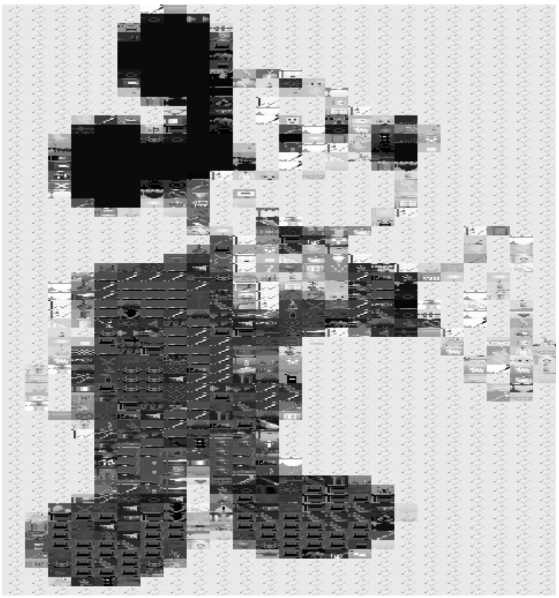
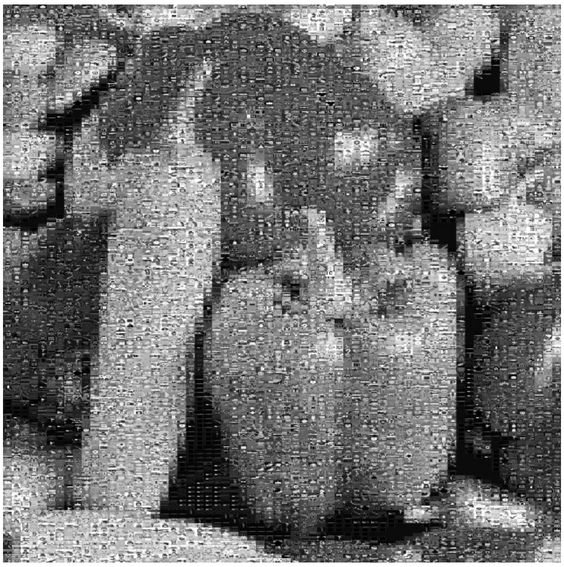
The transformed bitmap is divided into regions with size of cell_h * cell_w. With each region, its brightness and the most similar tile's index are stored into region_brightness and most_similar, respectively. After that, with each region of the output bitmap, theirs pixels will be filled up with the most similar tile. Finally, all the regions will be filled up into the output_bitmap and output the image with the specified name.

Result:

Compile: **cl photomosaic.cpp bmp.cpp list_files.cpp -std:c++17**

Testing:

Input	Output
photomosaic mario.bmp photo_tiles 512,512,8,8 output.bmp	
photomosaic cat.bmp photo_tiles 512,512,16,16 output.bmp	
photomosaic lena.bmp photo_tiles 1024,512,16,8 output.bmp	

<table>
<tr><td>photomosaic micky.bmp
photo_tiles 480,512,20,8
output.bmp</td><td></td></tr>
<tr><td>photomosaic peppers.bmp
photo_tiles_extended
1024,1024,16,8
output.bmp</td><td></td></tr>
</table>

Photomosaic.cpp:

```
#include "stdio.h"
#include <iostream>
#include <vector>
#include <string>
#include "malloc.h"
#include "memory.h"
#include "math.h"
#include "bmp.h"        //  Simple .bmp library
#include "list_files.h" //  Simple list file library
```

```
#define SAFE_FREE(p)  { if(p){ free(p);  (p)=NULL;} }
using namespace std;

double getTileBrightness(Bitmap& tile);
void processTile(Bitmap& tile, int cell_w, int cell_h, Bitmap& processed_tile);
double getRegionBrightness(int cell_h, int cell_w, int leftx, int lefty, Bitmap& bmp);
int getClosestDistanceIndex(int num_of_tiles, vector<double> tiles_brightness, double
region_brightness);

int main(int argc, char** argv){
    // Parse output and cell shape specified in argv[3]
    int output_w, output_h, cell_w, cell_h;
    sscanf(argv[3], "%d,%d,%d,%d", &output_w, &output_h, &cell_w, &cell_h);

    // Read source bitmap from argv[1]
    Bitmap source_bitmap(argv[1]);
    int source_w = source_bitmap.getWidth();
    int source_h = source_bitmap.getHeight();

    // List .bmp files in argv[2] and do preprocessing
    vector<string> list_of_paths;
    list_files(argv[2], ".bmp", list_of_paths, false);

    //create output bitmap
    Bitmap transformed_bitmap(output_w, output_h);
    //resize and convert original bmp into grayscale
    processTile(source_bitmap, output_w, output_h, transformed_bitmap);

    int num_of_tiles = list_of_paths.size();
    vector<Bitmap> tiles, processed_tiles;
    vector<double> tiles_brightness;

    //resize all tiles and convert them to grayscale
    for (int i = 0; i < num_of_tiles; i++) {
        Bitmap bmp1,bmp2;
        bmp1.create(list_of_paths[i].c_str());//convert a string object to a char*
        bmp2.create(cell_w, cell_h);
        tiles.push_back(bmp1);
        processed_tiles.push_back(bmp2);
```

```
        processed_tiles[i].create(cell_w, cell_h);
        processTile(tiles[i], cell_w, cell_h, processed_tiles[i]);

        //average the brightness
        tiles_brightness.push_back(getTileBrightness(processed_tiles[i]));

    }

    Bitmap output_bitmap(output_w, output_h);

    for (int y = 0; y < output_h; y+=cell_h) {
        for (int x = 0; x < output_w; x+=cell_w) {
            //calculate the region brightness of the transformed bitmap
            double region_brightness = getRegionBrightness(cell_h, cell_w, x, y,
transformed_bitmap);
            int most_similar =
getClosestDistanceIndex(num_of_tiles,tiles_brightness,region_brightness);
            //loop through the tile and copy the tile to one of the regions of the output bmp
            for (int i = 0; i < cell_h; i++) {
                for (int j = 0; j < cell_w; j++) {
                    Color RGB;
                    processed_tiles[most_similar].getColor(j, i, RGB.R, RGB.G, RGB.B);
                    output_bitmap.setColor(x + j , y + i, RGB.R, RGB.G, RGB.B);
                }
            }
        }
    }

    //save output bitmap to argv[4]
    output_bitmap.save(argv[4]);

    return 0;
}


//calculate the brightness of a bitmap
double getTileBrightness(Bitmap& tile)
{
    int tile_w = tile.getWidth();
```

```
    int tile_h = tile.getHeight();
    double tile_brightness = 0.0;
    for (int y = 0; y < tile_h; y++) {
        for (int x = 0; x < tile_w; x++) {
            unsigned char R, G, B;
            tile.getColor(x, y, R, G, B);
            tile_brightness += 0.299 * R + 0.587 * G + 0.114 * B;
        }
    }

    return tile_brightness / (double)(tile_w * tile_h);
}

//convert a tile to grayscale and resize it to cell_w and cell_h
void processTile(Bitmap& tile, int cell_w, int cell_h, Bitmap& processed_tile){
    int tile_w = tile.getWidth();
    int tile_h = tile.getHeight();
    processed_tile.create(cell_w, cell_h);

    for (int y = 0; y < cell_h; y++) {
        for (int x = 0; x < cell_w; x++) {
            //mapping
            float fx = x * tile_w / cell_w;//f(x)
            float fy = y * tile_h / cell_h;//f(y)
            //find the four surrounding pixels
            int x1 = floor(fx);
            // int x2 = ceil(fx) > tile_w - 1 ? tile_w - 1 : ceil(fx);
            int x2 = ceil(fx);
            int y1 = floor(fy);
            // int y2 = ceil(fy) > tile_h - 1 ? tile_h - 1 : ceil(fy);
            int y2 = ceil(fy);

            //border case
            if (x2 == 0 && x1 == 0) x2 += 1;
            else if(x1 == x2) x1 -= 1;
            if (y2 == 0 && y1 == 0) y2 += 1;
            else if(y1 == y2) y1 -= 1;

            Color c1, c2, c3, c4;//BL, TL, BR, TR
```

```
            tile.getColor(x1, y1, c1.R, c1.G, c1.B);
            tile.getColor(x1, y2, c2.R, c2.G, c2.B);
            tile.getColor(x2, y1, c3.R, c3.G, c3.B);
            tile.getColor(x2, y2, c4.R, c4.G, c4.B);

            int horizontal_distance = abs(x2-x1);
            int vertical_distance = abs(y2-y1);
            unsigned char upper_weighted_R, upper_weighted_G, upper_weighted_B,
lower_weighted_R, lower_weighted_G, lower_weighted_B;
            //upper weighted average
            upper_weighted_R = (c2.R * (x2-fx) + c4.R * (fx-x1))/horizontal_distance;
            upper_weighted_G = (c2.G * (x2-fx) + c4.G * (fx-x1))/horizontal_distance;
            upper_weighted_B = (c2.B * (x2-fx) + c4.B * (fx-x1))/horizontal_distance;
            //lower weighted average
            lower_weighted_R = (c3.R * (x2-fx) + c1.R * (fx-x1))/horizontal_distance;
            lower_weighted_G = (c3.G * (x2-fx) + c1.G * (fx-x1))/horizontal_distance;
            lower_weighted_B = (c3.B * (x2-fx) + c1.B * (fx-x1))/horizontal_distance;

            unsigned char vertical_weighted_R, vertical_weighted_G, vertical_weighted_B;

            //vertical weighted average
            vertical_weighted_R = (upper_weighted_R * (y2-fy) + lower_weighted_R * (fy-
y1))/vertical_distance;
            vertical_weighted_G = (upper_weighted_G * (y2-fy) + lower_weighted_G * (fy-
y1))/vertical_distance;
            vertical_weighted_B = (upper_weighted_B * (y2-fy) + lower_weighted_B * (fy-
y1))/vertical_distance;

            unsigned char gray = vertical_weighted_R * 0.299 + vertical_weighted_G * 0.587 +
vertical_weighted_B * 0.114;
            processed_tile.setColor(x,y,gray,gray,gray);
        }
    }
}

double getRegionBrightness(int cell_h, int cell_w, int leftx, int lefty, Bitmap& bmp){
    double region_brightness = 0.0;
    for (int i = 0; i < cell_h; i++) {
        for (int j = 0; j < cell_w; j++) {
```

```
            unsigned char R, G, B;
            bmp.getColor(j + leftx, i + lefty, R, G, B);
            region_brightness += 0.299 * R
                + 0.587 * G
                + 0.114 * B;
        }
    }
    return region_brightness /= (cell_h * cell_w);
}

int getClosestDistanceIndex(int num_of_tiles, vector<double> tiles_brightness, double
region_brightness){
    int most_similar = 0;
    for (int i = 1; i < num_of_tiles; i++){
        if (abs(tiles_brightness[i] - region_brightness) < abs(tiles_brightness[most_similar]
- region_brightness))
            most_similar = i;
    }
    return most_similar;
}
```

Part 2 – Enhancement Features

Bicubic Interpolation:

For the Cubic Interpolation

function defined a spline piecewise: $f(x) = ax^3 + bx^2 + cx + d$

$a = 2f(0) - 2f(1) + f'(0) + f'(1)$

$b = -3f(0) + 3f(1) - 2f'(0) - f'(1)$

$c = f'(0)$

$d = f(0)$

Resampling data:

$$\Delta x = \frac{\# \, of origina \, points}{\# \, of \, New \, Points} \, , x_{new} = (n + 0.5) * \Delta x - 0.5, n \in [0, \# \, of \, New \, Points)$$

Since x_{new} may be negative and out of bound, the extrapolation on both ends is required for calculating the first order derivative.

Assume that there are 4 points x1,x2,x3 and x4 on x-axis, where x1<x2<x3<x4, the spline we are going through is between x2 and x3.

$f(r) = converted \, RGB \, values, where \, r = (x_{new} - x2) \in (x2, x3) \, on \, a \, spline$

$Resampled \, data \, R = \{r_1, r_2, \ldots, r_n\} \, where \, n = \, number \, of \, new \, points$

$original \, set \, of \, points \, P = \{p | 0 \leq p \leq n - 1\}, n = number \, of \, original \, points$

After extrapolation, $E = \{e | e \in [-2, -1, n, n + 1]\}$

$set \, of \, points \, being \, extrapolated: A = \{a | P \cup E\}$

$Spline \, S(r)_{x_i, x_j, x_k, x_l} = \{r \epsilon R | \, x_j \leq r \leq x_k, x_i < x_j < x_k < x_l \, , x \in A \,\}$

$= ar^3 + br^2 + cr + d$

Set of Splines = $\{S_{-2,-1,0,1}, S_{-1,0,1,2}, \ldots, S_{n-2,n-1,n,n+1}\}$ = a set of interpolated pixels in a row/column

The function below resamples all the data points and extrapolates on both ends. It then loops through all splines, calculates f(x) for each pixel in the row/column and returns the RGB values for a set of pixels in the corresponding row/col.

```
struct Coordinate{

    double x;

    unsigned char r,g,b;

};

vector<Color> splineCalculation(vector<Coordinate> points, double delta, int length){

    vector<double> resampled_points;

    vector<Coordinate> extrapolated_points;

    vector<Color> RGB;

    //resample all the data points

    for(int i = 0; i < length; i++){
```

```
        double x = (i + 0.5) * delta - 0.5;
        // double x = delta * i;
        resampled_points.push_back(x);
    }
    //extrapolate on both ends
    for(int i = 0; i < points.size(); i++){
        Coordinate new_point;
        new_point.x = (double)points[i].x;
        new_point.r = points[i].r;
        new_point.g = points[i].g;
        new_point.b = points[i].b;
        if(i == 0){
            new_point.x = (double)points[i].x - 2;
            extrapolated_points.push_back(new_point);
            new_point.x += (double)1;
            extrapolated_points.push_back(new_point);
            new_point.x += (double)1;
        }
        extrapolated_points.push_back(new_point);
        if(i == points.size()-1){
            new_point.x += (double)1;
            extrapolated_points.push_back(new_point);
            new_point.x += (double)1;
            extrapolated_points.push_back(new_point);
        }
    }

    vector<Color> output_row;
    int cur_point = 0;
    //loop through all splines
    for(int i = 1; i < extrapolated_points.size()-2; i++){
        Coordinate prev_point = extrapolated_points[i-1];
        Coordinate p1 = extrapolated_points[i];
        Coordinate p2 = extrapolated_points[i+1];
        Coordinate next_point = extrapolated_points[i+2];
        //check if the value exceeds the current upper point of the spline
        while(cur_point < length && resampled_points[cur_point] <= p2.x){
            //calculate f(x)
            double x = resampled_points[cur_point] - p1.x;
```

```
            double f0[3]={(double)p1.r,(double)p1.g,(double)p1.b};

            double f1[3]={(double)p2.r,(double)p2.g,(double)p2.b};

            double df0[3] = {

                (p2.r-prev_point.r)/(p2.x-prev_point.x),

                (p2.g-prev_point.g)/(p2.x-prev_point.x),

                (p2.b-prev_point.b)/(p2.x-prev_point.x)

            };

            double df1[3] = {

                (next_point.r - p1.r)/(next_point.x - p1.x),

                (next_point.g - p1.g)/(next_point.x - p1.x),

                (next_point.b - p1.b)/(next_point.x - p1.x)

            };

            double a[3],b[3],c[3],d[3],fx[3];

            Color RGB;

            for(int j = 0;j < 3; j++){

                a[j] = 2 * f0[j] - 2 * f1[j] + df0[j] + df1[j];

                b[j] = -3 * f0[j] + 3 * f1[j] - 2*df0[j] - df1[j];

                c[j] = df0[j];

                d[j] = f0[j];

                fx[j] = a[j]*x*x*x + b[j] *x*x + c[j]*x + d[j];

            }

            RGB.R=(unsigned char)fx[0];

            RGB.G=(unsigned char)fx[1];

            RGB.B=(unsigned char)fx[2];

            output_row.push_back(RGB);

            cur_point++;

            if(cur_point >= length) break;

        }

    }

    return output_row;//return RGB values for each pixel in a row/col

}
```

After finishing a row, also finish the rest of the rows and save the row-processed bitmap into processed_x_tile.

Then, looping through each column of the processed rows to calculate the grayscale value for each pixel. The splineCalculation function is used to interpolate between points in the row or column.

```
//convert a tile to grayscale and resize it to cell_w and cell_h

void processTile(Bitmap& tile, int cell_w, int cell_h, Bitmap& processed_tile){
```

```
    int tile_w = tile.getWidth();
    int tile_h = tile.getHeight();
    //process the row first therefore the height will be original height
    Bitmap processed_x_tile(cell_w,tile_h);
    processed_tile.create(cell_w, cell_h);
    double delta_x = (double)tile_w/cell_w;
    //loop through each row
    for (int y = 0; y < tile_h; y++) {
        vector<Coordinate> points;//a row of original points
        for (int x = 0; x < tile_w; x++) {
            Coordinate new_point;
            new_point.x = (double)x;
            tile.getColor(x,y,new_point.r,new_point.g,new_point.b);
            points.push_back(new_point);
        }
        vector<Color> row = splineCalculation(points, delta_x, cell_w);
        for(int x = 0; x < cell_w; x++){
            processed_x_tile.setColor(x,y,row[x].R,row[x].G,row[x].B);
        }
    }
    //loop through each column
    double delta_y = (double)tile_h/cell_h;
    for(int x = 0; x < cell_w; x++){
        vector<Coordinate> points;
        for (int y = 0; y < tile_h; y++){
            Coordinate new_point;
            new_point.x = (double)y;
            processed_x_tile.getColor(x,y,new_point.r,new_point.g,new_point.b);
            points.push_back(new_point);
        }
        vector<Color> col = splineCalculation(points, delta_y, cell_h);
        for (int y = 0; y < cell_h; y++){
            unsigned char gray = col[y].R * 0.299 + col[y].G * 0.587 + col[y].B * 0.114;
            processed_tile.setColor(x,y,gray,gray,gray);
        }
    }
}
```

Reference: https://www.youtube.com/watch?v=syH8ASkotFg

Histogram and Histogram Intersection:

Histogram Calculation: [0,255]= bin1 ⋃ bin2 ⋃ .... ⋃ binN

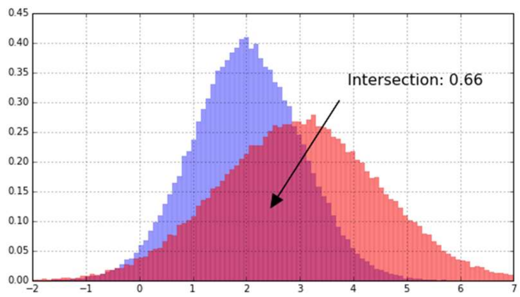
```
//calculate histogram for a bitmap
vector<int> calculateHistogram(Bitmap& bitmap){
    int num_of_bins = 256;
    int width = bitmap.getWidth();
    int height = bitmap.getHeight();
    vector<int> hist(num_of_bins);
     for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            unsigned char r,g,b;
            bitmap.getColor(i, j, r, g, b);
            int bin_idx = ((r+g+b)/3) % num_of_bins;
            ++hist[bin_idx];
        }
     }
    return hist;
}
```

It calculates the histogram of the bitmap by looping through each pixel and getting its color values (r, g, b). It then calculates the bin index for each pixel by taking the average of the three color values and modding it with the number of bins. Increments that bin index in the histogram vector.
number of bins = 256 is the most optimal choice regarding the accuracy of the output bitmap in my application.

Histogram Intersection:

$$\sum_{i=1}^{bins} \min(H_1, H_2) \ where\ H_1\ and\ H_2\ are\ two\ different\ histograms$$

The concept of intersection between 2 histograms can be visualized:



source:https://blog.datadive.net/histogram-intersection-for-change-detection/

```
//computes the intersection of two histograms
double histogramIntersection(vector<int>& hist1, vector<int>& hist2) {
    double intersection = 0.0;
    for (int i = 0; i < hist1.size(); i++) {
        intersection += min(hist1[i], hist2[i]);
    }
    return intersection;
}
```

It computes the intersection of two color histograms by looping through each bin index and taking the minimum value between both histograms for that bin index.

Distance Metrics:$1 - Normalized\ Intersection = 1 - \frac{Intersection}{bins} = overlapped\ area$

```
//computes the distance between two histograms using Intersection
double histogramIntersectionDistance(vector<int>& hist1, vector<int>& hist2) {
    double intersection = histogramIntersection(hist1, hist2);
    double total_bins = hist1.size();
    double distance = 1 - (intersection / total_bins);
    return distance;
}
```

It computes the distance between two histograms using the Intersection.

References:

https://mpatacchiola.github.io/blog/2016/11/12/the-simplest-classifier-histogram-intersection.html
http://amroamroamro.github.io/mexopencv/opencv/histogram_calculation_demo.html
https://blog.datadive.net/histogram-intersection-for-change-detection/

Blending for avoiding using the same tiles in the adjacent cells:

blended_pixel = weight * source_pixel + (1 - weight) * destination_pixel

where **weight** is the blending factor (typically between 0 and 1). A higher weight will result in more contribution from the source_pixel and a lower weight will result in more contribution from the destination_pixel. Since our original goal is to fill all the cells with tiles, I will use 0.5 to make the result in average contribution. The implementation is as below.

```
void blend(Bitmap& b1, Bitmap& b2){
    int width = b2.getWidth();
```

```
    int height = b2.getHeight();
    float weight = 0.5;
    for(int y = 0; y < height; y++){
        for (int x = 0; x < width; x++){
            unsigned char r[2],g[2],b[2];
            b1.getColor(x,y,r[0],g[0],b[0]);
            b2.getColor(x,y,r[1],g[1],b[1]);
            unsigned char blended_r = weight * r[0] + (1 - weight) * r[1];
            unsigned char blended_g = weight * g[0] + (1 - weight) * g[1];
            unsigned char blended_b = weight * b[0] + (1 - weight) * b[1];

            b2.setColor(x,y,blended_r,blended_g,blended_b);
        }
    }
}
```

Reference: https://homepages.inf.ed.ac.uk/rbf/HIPR2/blend.htm

The whole process of the image processing is similar to part 1.

```
    vector<int> used_tiles;
    vector<Bitmap> used_bitmaps;
    for (int y = 0; y < output_h; y+=cell_h) {
        for (int x = 0; x < output_w; x+=cell_w) {
            vector<int> region_hist = getRegionHistogram(cell_h, cell_w, x, y,
transformed_bitmap);//histogram for the region
            int most_similar = getClosestDistanceIndex(num_of_tiles,tiles_hist,region_hist);//most similar
tile's index
            bool same_color = false;
            int used_tiles_size = used_tiles.size();
            Bitmap region(cell_w,cell_h);
            if(used_tiles_size){
                //check left and up
                if((x != 0 && used_tiles[used_tiles_size - 1] == most_similar) || (used_tiles_size >=
output_w / cell_w && used_tiles[used_tiles_size - output_w / cell_w] == most_similar)){
                    same_color = true;
                    for (int i = 0; i < cell_h; i++) {
                        for (int j = 0; j < cell_w; j++) {
                            unsigned char r,g,b;
                            // transformed_bitmap_color.getColor(x + j , y + i, r, g, b);
                            transformed_bitmap.getColor(x + j , y + i, r, g, b);
```

```
                    region.setColor(j,i,r,g,b);
                }
            }
            int idx = (x != 0 && used_tiles[used_tiles_size - 1] == most_similar)? used_tiles_size
- 1 : used_tiles_size - output_w / cell_w;
            blend(used_bitmaps[idx], region);
            // toGrayscale(region);
            // most_similar = -1;//mark as blended cell
            // used_bitmaps.push_back(region);
        }
    }
    //loop through the tile and copy the tile to one of the regions of the output bmp
    for (int i = 0; i < cell_h; i++) {
        for (int j = 0; j < cell_w; j++) {
            Color RGB;
            if(!same_color){
                processed_tiles[most_similar].getColor(j, i, RGB.R, RGB.G, RGB.B);
            }
            else{
                region.getColor(j, i, RGB.R, RGB.G, RGB.B);
            }
                output_bitmap.setColor(x + j , y + i, RGB.R, RGB.G, RGB.B);
        }
    }
    used_tiles.push_back(most_similar);
    Bitmap temp = same_color?region:processed_tiles[most_similar];
    used_bitmaps.push_back(temp);
}
}
```

used_tiles and used_bitmaps to keep track of the tiles that have already been used in the output image and the corresponding regions in the input image.

Then loops through each region in the output image and gets the histogram of the region using the getRegionHistogram function. The most similar tile is found using the getClosestDistanceIndex function.

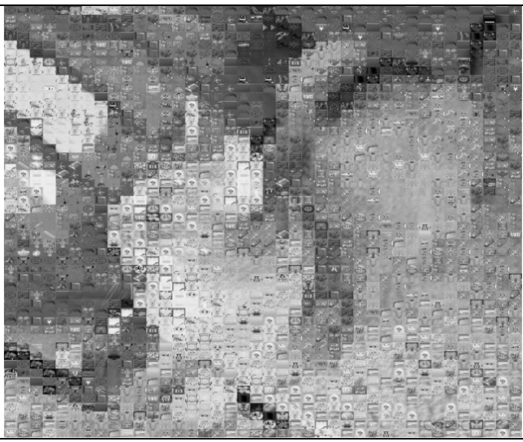
If the tile to the left or above the current region has used the same tile(even if the tile is blended), blends the current region(from resized original bmp) with the adjacent tile. Replace the region by the most similar tile otherwise.

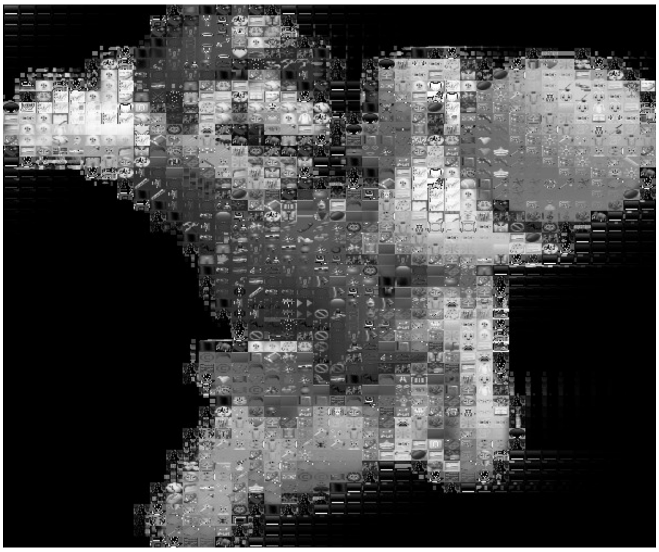
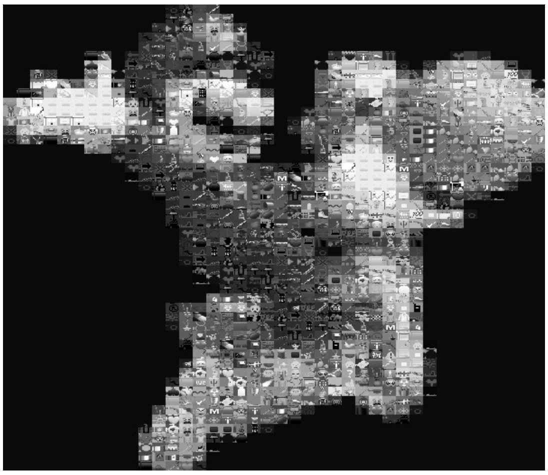
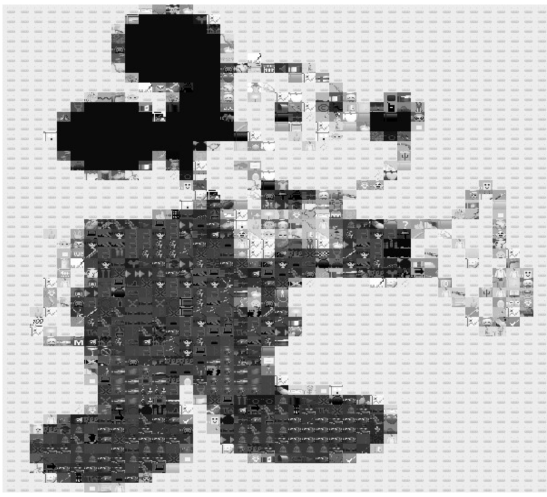
Compile: **cl photomosaic_enhancement.cpp bmp.cpp list_files.cpp -std:c++17**

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.29.30147 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

photomosaic_enhancement.cpp
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\include\ostream(743): warning C4530: C++ exception handler used, but unwind semantics are not enable
d. Specify /EHsc
photomosaic_enhancement.cpp(55): note: see reference to function template instantiation 'std::basic_ostream<char,std::char_traits<char>> &std::operator <<<std::char_traits<char>>(std::basi
c_ostream<char,std::char_traits<char>> &,const char *)' being compiled
bmp.cpp
list_files.cpp
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.29.30133\include\ostream(743): warning C4530: C++ exception handler used, but unwind semantics are not enable
d. Specify /EHsc
list_files.cpp(15): note: see reference to function template instantiation 'std::basic_ostream<char,std::char_traits<char>> &std::operator <<<std::char_traits<char>>(std::basic_ostream<cha
list_files.cpp(15): note: see reference to function template instantiation 'std::basic_ostream<char,std::char_traits<char>> &std::operator <<<std::char_traits<char>>(std::basic_ostream<cha
r,std::char_traits<char>> &,const char *)' being compiled
Generating Code...
Microsoft (R) Incremental Linker Version 14.29.30147.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:photomosaic_enhancement.exe
photomosaic_enhancement.obj
bmp.obj
list_files.obj
```

Result: photomosaic_enhancement lena.bmp photo_tiles 600,600,10,10 output.bmp

```
D:\CUHK\Year 3\Sem2\CSCI3280\Assignment1>photomosaic_enhancement lena.bmp photo_tiles 600,600,10,10 output.bmp
Resizing...
Composing...
Done!!
```

Sample Input	**Sample Output with enhancement**	**Sample Output by photomosaic.cpp with same size input**
photomosaic_enhancement lena.bmp photo_tiles 600,600,10,10 output.bmp		
photomosaic_enhancement cat.bmp photo_tiles 600,500,15,10 output.bmp		

photomosaic_enhancement mario.bmp photo_tiles 600,500,15,10 output.bmp		
photomosaic_enhancement micky.bmp photo_tiles 600,500,15,10 output.bmp		

photomosaic_enhancement.cpp:

```
#include "stdio.h"

#include <iostream>

#include <vector>

#include <string>

#include "malloc.h"

#include "memory.h"

#include "math.h"

#include "bmp.h"        //  Simple .bmp library

#include "list_files.h" //  Simple list file library


#define SAFE_FREE(p)  { if(p){ free(p);  (p)=NULL;} }

using namespace std;
```

```
struct Coordinate{
    double x;
    unsigned char r,g,b;
};


void processTile(Bitmap& tile, int cell_w, int cell_h, Bitmap& processed_tile);
int getClosestDistanceIndex(int num_of_tiles, vector<vector<int>> tiles_hist, vector<int>
region_hist);
vector<Color> splineCalculation(vector<Coordinate> points, double delta, int length);
void toGrayscale(Bitmap& bitmap);
vector<int> calculateHistogram(Bitmap& bitmap);
double histogramIntersection(vector<int>& hist1, vector<int>& hist2);
double histogramIntersectionDistance(vector<int>& hist1, vector<int>& hist2);
vector<int> getRegionHistogram(int cell_h, int cell_w, int leftx, int lefty, Bitmap& bmp);
void blend(Bitmap& b1, Bitmap& b2);

int main(int argc, char** argv){
    // Parse output and cell shape specified in argv[3]
    int output_w, output_h, cell_w, cell_h;
    sscanf(argv[3], "%d,%d,%d,%d", &output_w, &output_h, &cell_w, &cell_h);

    // Read source bitmap from argv[1]
    Bitmap source_bitmap(argv[1]);
    int source_w = source_bitmap.getWidth();
    int source_h = source_bitmap.getHeight();

    // List .bmp files in argv[2] and do preprocessing
    vector<string> list_of_paths;
    list_files(argv[2], ".bmp", list_of_paths, false);

    //create output bitmap
    Bitmap transformed_bitmap(output_w, output_h);
    //resize and convert original bmp into grayscale
    processTile(source_bitmap, output_w, output_h, transformed_bitmap);
    // Bitmap transformed_bitmap_color = transformed_bitmap;
    toGrayscale(transformed_bitmap);
```

```
    int num_of_tiles = list_of_paths.size();
    vector<Bitmap> tiles, processed_tiles;
    vector<double> tiles_brightness;
    vector<vector<int>> tiles_hist;
    cout << "Resizing..." << endl;
    //resize all tiles and convert them to grayscale
    for (int i = 0; i < num_of_tiles; i++) {
        Bitmap bmp1,bmp2;
        bmp1.create(list_of_paths[i].c_str());//convert a string object to a char*
        bmp2.create(cell_w, cell_h);
        tiles.push_back(bmp1);
        processed_tiles.push_back(bmp2);
        processed_tiles[i].create(cell_w, cell_h);
        processTile(tiles[i], cell_w, cell_h, processed_tiles[i]);
        // processed_tiles_color.push_back(processed_tiles[i]);
        toGrayscale(processed_tiles[i]);

        tiles_hist.push_back(calculateHistogram(processed_tiles[i]));

    }

    cout << "Composing..." << endl;
    Bitmap output_bitmap(output_w, output_h);
    vector<int> used_tiles;
    vector<Bitmap> used_bitmaps;
    for (int y = 0; y < output_h; y+=cell_h) {
        for (int x = 0; x < output_w; x+=cell_w) {
            vector<int> region_hist = getRegionHistogram(cell_h, cell_w, x, y,
transformed_bitmap);//histogram for the region
            int most_similar =
getClosestDistanceIndex(num_of_tiles,tiles_hist,region_hist);//most similar tile's index
            bool same_color = false;
            int used_tiles_size = used_tiles.size();
            Bitmap region(cell_w,cell_h);
            if(used_tiles_size){
                //check left and up
                if((x != 0 && used_tiles[used_tiles_size - 1] == most_similar) ||
(used_tiles_size >= output_w / cell_w && used_tiles[used_tiles_size - output_w / cell_w] ==
most_similar)){
```

```
                    same_color = true;
                    for (int i = 0; i < cell_h; i++) {
                        for (int j = 0; j < cell_w; j++) {
                            unsigned char r,g,b;
                            // transformed_bitmap_color.getColor(x + j , y + i, r, g, b);
                            transformed_bitmap.getColor(x + j , y + i, r, g, b);
                            region.setColor(j,i,r,g,b);
                        }
                    }
                    int idx = (x != 0 && used_tiles[used_tiles_size - 1] == most_similar)?
used_tiles_size - 1 : used_tiles_size - output_w / cell_w;
                    blend(used_bitmaps[idx], region);
                    // toGrayscale(region);
                    // most_similar = -1;//mark as blended cell
                    // used_bitmaps.push_back(region);
                }
            }
            //loop through the tile and copy the tile to one of the regions of the output bmp
            for (int i = 0; i < cell_h; i++) {
                for (int j = 0; j < cell_w; j++) {
                    Color RGB;
                    if(!same_color){
                        processed_tiles[most_similar].getColor(j, i, RGB.R, RGB.G, RGB.B);
                    }
                    else{
                        region.getColor(j, i, RGB.R, RGB.G, RGB.B);
                    }
                        output_bitmap.setColor(x + j , y + i, RGB.R, RGB.G, RGB.B);
                }
            }
            used_tiles.push_back(most_similar);
            // Bitmap temp = same_color?region:processed_tiles[most_similar];
            used_bitmaps.push_back(same_color?region:processed_tiles[most_similar]);
        }
    }
    cout << "Done!!";
    //save output bitmap to argv[4]
    output_bitmap.save(argv[4]);
```

```
    return 0;
}



vector<Color> splineCalculation(vector<Coordinate> points, double delta, int length){
    vector<double> resampled_points;
    vector<Coordinate> extrapolated_points;
    vector<Color> RGB;
    //resample all the data points
    for(int i = 0; i < length; i++){
        double x = (i + 0.5) * delta - 0.5;
        // double x = delta * i;
        resampled_points.push_back(x);
    }
    //extrapolate on both ends
    for(int i = 0; i < points.size(); i++){
        Coordinate new_point;
        new_point.x = (double)points[i].x;
        new_point.r = points[i].r;
        new_point.g = points[i].g;
        new_point.b = points[i].b;
        if(i == 0){
            new_point.x = (double)points[i].x - 2;
            extrapolated_points.push_back(new_point);
            new_point.x += (double)1;
            extrapolated_points.push_back(new_point);
            new_point.x += (double)1;
        }
        extrapolated_points.push_back(new_point);
        if(i == points.size()-1){
            new_point.x += (double)1;
            extrapolated_points.push_back(new_point);
            new_point.x += (double)1;
            extrapolated_points.push_back(new_point);
        }
    }

    vector<Color> output_row;
```

```
    int cur_point = 0;
    //loop through all splines
    for(int i = 1; i < extrapolated_points.size()-2; i++){
        Coordinate prev_point = extrapolated_points[i-1];
        Coordinate p1 = extrapolated_points[i];
        Coordinate p2 = extrapolated_points[i+1];
        Coordinate next_point = extrapolated_points[i+2];
        //check if the value exceeds the current upper point of the spline
        while(cur_point < length && resampled_points[cur_point] <= p2.x){
            //calculate f(x)
            double x = resampled_points[cur_point] - p1.x;
            double f0[3]={(double)p1.r,(double)p1.g,(double)p1.b};
            double f1[3]={(double)p2.r,(double)p2.g,(double)p2.b};
            double df0[3] = {
                (p2.r-prev_point.r)/(p2.x-prev_point.x),
                (p2.g-prev_point.g)/(p2.x-prev_point.x),
                (p2.b-prev_point.b)/(p2.x-prev_point.x)
            };
            double df1[3] = {
                (next_point.r - p1.r)/(next_point.x - p1.x),
                (next_point.g - p1.g)/(next_point.x - p1.x),
                (next_point.b - p1.b)/(next_point.x - p1.x)
            };
            double a[3],b[3],c[3],d[3],fx[3];
            Color RGB;
            for(int j = 0;j < 3; j++){
                a[j] = 2 * f0[j] - 2 * f1[j] + df0[j] + df1[j];
                b[j] = -3 * f0[j] + 3 * f1[j] - 2*df0[j] - df1[j];
                c[j] = df0[j];
                d[j] = f0[j];
                fx[j] = a[j]*x*x*x + b[j] *x*x + c[j]*x + d[j];
            }
            RGB.R=(unsigned char)fx[0];
            RGB.G=(unsigned char)fx[1];
            RGB.B=(unsigned char)fx[2];
            output_row.push_back(RGB);
            cur_point++;
            if(cur_point >= length) break;
        }
```

```
    }
    return output_row;//return RGB values for each pixel in a row/col
}

//convert a tile to grayscale and resize it to cell_w and cell_h
void processTile(Bitmap& tile, int cell_w, int cell_h, Bitmap& processed_tile){
    int tile_w = tile.getWidth();
    int tile_h = tile.getHeight();
    //process the row first therefore the height will be original height
    Bitmap processed_x_tile(cell_w,tile_h);
    processed_tile.create(cell_w, cell_h);

    double delta_x = (double)tile_w/cell_w;
    //loop through each row
    for (int y = 0; y < tile_h; y++) {
        vector<Coordinate> points;//a row of original points
        for (int x = 0; x < tile_w; x++) {
            Coordinate new_point;
            new_point.x = (double)x;
            tile.getColor(x,y,new_point.r,new_point.g,new_point.b);
            points.push_back(new_point);
        }
        vector<Color> row = splineCalculation(points, delta_x, cell_w);
        for(int x = 0; x < cell_w; x++){
            processed_x_tile.setColor(x,y,row[x].R,row[x].G,row[x].B);
        }
    }
    //loop through each column
    double delta_y = (double)tile_h/cell_h;
    for(int x = 0; x < cell_w; x++){
        vector<Coordinate> points;
        for (int y = 0; y < tile_h; y++){
            Coordinate new_point;
            new_point.x = (double)y;
            processed_x_tile.getColor(x,y,new_point.r,new_point.g,new_point.b);
            points.push_back(new_point);
        }
        vector<Color> col = splineCalculation(points, delta_y, cell_h);
        for (int y = 0; y < cell_h; y++){
```

```
            // unsigned char gray = col[y].R * 0.299 + col[y].G * 0.587 + col[y].B * 0.114;
            processed_tile.setColor(x,y,col[y].R,col[y].G,col[y].B);
        }
    }
}

int getClosestDistanceIndex(int num_of_tiles, vector<vector<int>> tiles_hist, vector<int>
region_hist){
    int most_similar = 0;
    for (int i = 1; i < num_of_tiles; i++){
        if (histogramIntersectionDistance(tiles_hist[most_similar],region_hist) >
        histogramIntersectionDistance(tiles_hist[i],region_hist))
            most_similar = i;
    }
    return most_similar;
}

void toGrayscale(Bitmap& bitmap){
    int width = bitmap.getWidth();
    int height = bitmap.getHeight();

    for (int i = 0; i < height; i++){
        for (int j = 0; j < width; j++){
            Color c;
            bitmap.getColor(j , i, c.R, c.G, c.B);
            unsigned char gray = c.R * 0.299 + c.G * 0.587 + c.B * 0.114;
            bitmap.setColor(j,i,gray,gray,gray);
        }
    }

}

//calculate histogram for a bitmap
vector<int> calculateHistogram(Bitmap& bitmap){
    int num_of_bins = 256;
    int width = bitmap.getWidth();
    int height = bitmap.getHeight();
    vector<int> hist(num_of_bins);
     for (int i = 0; i < width; i++) {
```

```
        for (int j = 0; j < height; j++) {
            unsigned char r,g,b;
            bitmap.getColor(i, j, r, g, b);
            int bin_idx = ((r+g+b)/3) % num_of_bins;
            ++hist[bin_idx];
        }
     }
    return hist;
}

//computes the intersection of two histograms
double histogramIntersection(vector<int>& hist1, vector<int>& hist2) {
    double intersection = 0.0;
    for (int i = 0; i < hist1.size(); i++) {
        intersection += min(hist1[i], hist2[i]);
    }
    return intersection;
}

//computes the distance between two histograms using Intersection
double histogramIntersectionDistance(vector<int>& hist1, vector<int>& hist2) {
    double intersection = histogramIntersection(hist1, hist2);
    double total_bins = hist1.size();
    double distance = 1 - (intersection / total_bins);
    return distance;
}

vector<int> getRegionHistogram(int cell_h, int cell_w, int leftx, int lefty, Bitmap& bmp){
    double region_brightness = 0.0;
    Bitmap region(cell_w,cell_h);
    for (int i = 0; i < cell_h; i++) {
        for (int j = 0; j < cell_w; j++) {
            unsigned char R, G, B;
            bmp.getColor(j + leftx, i + lefty, R, G, B);
            region.setColor(j,i,R,G,B);
        }
    }
    return calculateHistogram(region);
}
```

```
void blend(Bitmap& b1, Bitmap& b2){
    int width = b2.getWidth();
    int height = b2.getHeight();
    float weight = 0.5;
    for(int y = 0; y < height; y++){
        for (int x = 0; x < width; x++){
            unsigned char r[2],g[2],b[2];
            b1.getColor(x,y,r[0],g[0],b[0]);
            b2.getColor(x,y,r[1],g[1],b[1]);
            unsigned char blended_r = weight * r[0] + (1 - weight) * r[1];
            unsigned char blended_g = weight * g[0] + (1 - weight) * g[1];
            unsigned char blended_b = weight * b[0] + (1 - weight) * b[1];

            b2.setColor(x,y,blended_r,blended_g,blended_b);
        }
    }
}
```