Research About the Color Colorization from Gray Scale Based on C#

COSC 4306 - Computer Graphics I

Yifei Ren

2017 May 29

Table of Content

Abstract 2

Introduction 2

Prior and related work 3

Description of algorithm or system 5

Improvement 7

1. The result and effect of different grayscale conversion algorithm. 8

1.1 Desaturation 9

1.2 Luminosity 10

1.3 Average 11

2. Fixed Difference 12

2.1 Implementation in C# 12

2.2 Improvement 2 result comparison 13

3. Improve color table and usage 15

3.1 Calculation for shortest distance 16

3.2 Uniformity of different sizes 17

3.3 C# implementation 18

3.4 Comparison between before and after improve 3 19

4. Result optimization - Simplest Color Balance 20

4.1 copy the color value from image to three array 20

4.2 Sort the pixel values 21

4.3 Pick the quantiles from the sorted pixels 21

4.4 Affine transform 22

4.5 Implementation of Improvement 4 using C# 22

4.6 Improvement Result 24

Experimental results 25

Conclusions 26

Bibliography 28

Abstract

Colorization is a computer-assisted process of adding color to a monochrome image. The process typically involves segmenting images into regions and tracking these regions across image sequences. Neither of these tasks can be performed reliably in practice; consequently, colorization requires considerable user intervention and remains a tedious, time-consuming, and expensive task.

In this paper. we present a simple automatic colorization method and its improvement.This method is based on a simple premise: pixels in spacetime that have similar intensities should have similar colors.

To formalizing this premise, this method will takes in a grayscale image and a color image with a similar theme as input, and the grayscale image will be colorized and returned as result.

Introduction

Converting a Grayscale image back into a colored image can be considered a hot topic started in the past few years.The accuracy of the colors has become the measurement of success. Comparing color conversions between RGB to grayscale and grayscale to RGB. During the conversion from an RGB to a grayscale, some of the color information will be lost in the process for the color map. Because it is a many to one situation, therefore many different pixels of RGB could map to the same grayscales. The new generated grayscale image may vary, depending on the algorithm used for the grayscale conversion. On the other hand, when converting a grayscale image back into RGB, the situation has become one too many problems. Where one pixel of a grayscale image could have matched to many corresponding RGBs. Grayscale to RGB conversion are normally used for How to find and repair this lost information is a key part of this topic. Once the problems are solved, recovering old black white photos and processing those image with color loss will not be an issue anymore. The methods that are mentioned in this report all have their pros and cons. In this research will be mainly going to focus on the implementation and improvement of an automatic colorization approach.

This fully automated colorization process is to apply a similar color table to the grayscale image. This color table can be generated from another picture (Detail\_1, Detail\_2). The advantages of a fully automated processing it does not require manual participation in the process. However, due to the limitations of the algorithm, the results are sometimes not ideal. For images with more complex colors, this method will not apply 100% correctly. And through the four major improvements the result will be somewhat toward ideal.

Prior and related work

Grayscale – An Introduction

Black and white (or monochrome) photography dates back to the mid-19th century. Despite the eventual introduction of color photography, monochromatic photography remains popular. If anything, the digital revolution has actually increased the popularity of monochromatic photography because any digital camera is capable of taking black-and-white photographs (whereas analog cameras required the use of special monochromatic film). Monochromatic photography is sometimes considered the “sculpture” variety of photographic art. It tends to abstract the subject, allowing the photographer to focus on form and interpretation instead of simply reproducing reality. Because the terminology black-and-white is imprecise – black-and-white photography actually consists of many shades of gray – this article will refer to such images as grayscale.

All grayscale algorithms utilize the same basic three-step process:

1.Get the red, green, and blue values of a pixel

2.Use fancy math to turn those numbers into a single gray value

3.Replace the original red, green, and blue values with the new gray value.

In Order for us to continue, we need to understand how grayscale images are converted and find the relations between a grayscale image to its colored image.There are many methods that are commonly used all around us such as (Detail\_3):

Method 1 – Averaging (aka “quick and dirty”)

Gray = (Red + Green + Blue) / 3

Because it is fast and simple.

Method 2 – Correcting for the human eye ( Luminosity Method )

Gray = (Red \* 0.3 + Green \* 0.59 + Blue \* 0.11) or

Gray = (Red \* 0.2126 + Green \* 0.7152 + Blue \* 0.0722) or

Gray = (Red \* 0.299 + Green \* 0.587 + Blue \* 0.114)

This is largely used in Photoshop and GIMP

Method 3 – Desaturation

Gray = ( Max(Red, Green, Blue) + Min(Red, Green, Blue) ) / 2

Desaturation results in a flatter, softer grayscale image.

Method 4 – Decomposition

Gray = Max(Red, Green, Blue) or

Gray = Mini(Red, Green, Blue) or

a maximum decomposition provides a brighter grayscale image, while a minimum decomposition provides a darker one. This method of grayscale reduction is typically used for artistic effect.

Method 5 – Single color channel

Gray = Red or

Gray = Green or

Gray = Blue

Fastest computational method for grayscale reduction and it is the most used algorithm in digital cameras for taking “grayscale” photos.

The method that we focused on in this report are Method1, Method2, Method3. Luminosity Method is also called Weighted method. (Detail\_4, Detail\_5). This method create a grayscale image that is easy on the eye, and it solves the problem occurs in a Averaging method and. which a Averaging method takes average of the three colors. Since the three different colors have three different wavelength and have their own contribution in the formation of image, so it has to take average according to their contribution, not done it averagely using average method. The averaging methods’ doing is this, 33% of Red,33% of Green and 33% of Blue. It is taking 33% of each, that means, each of the portion has same contribution in the image. But in reality that's not the case. Which result a rather darker image. And for the Desaturation method results in a flatter, softer grayscale image.

According to the equation of Luminosity Method, we considered that Red has contributed 21%, Green has contributed 72% which is greater in all three colors and Blue has contributed 7%.It decrease the contribution of red color, and increase the contribution of the green color, and put blue color contribution in between these two. Since red color has more wavelength of all the three colors, and green is the color that has not only less wavelength than red color but also green is the color, As compared to the result of the other two methods, this image is more brighter and gives a more soothing effect to the eyes.

On the other hand, we do not know which conversion method is applied to create our input grayscale image. Therefore we need to understand and compare each method in order to find a better result.

Furthermore, We need to know how the color table works (Detail\_6). At the same time we also need to know and understand how to create a color matching table. Because our research topic is to convert the grayscale image into a color image. In that case, we do not need a very Complex color table, only a simple comparison table would do the trick.

In this project. We used C#, a programming tool to deal with the pictures, and compare the results, which can lead to better solutions for a fully automated colorization process.

Description of algorithm or system

In this report, we choose to implement and achieve a fully automated algorithm function to restore the color reduction in grayscale image. In order for this method to work, it requires a grayscale and a color table. Therefore we chose to generate a color table from another picture for our benefit. In that case, this algorithm requires a grayscale image that need to be colored and a color image with similar theme to the grayscale image as inputs, and Four steps of process to get our result.

• Generating a new grayscale image using the Luminosity Algorithm ( This method will be discussed and compared later on in the IMPROVE section)

• Sorting each pixel of the new grayscaled image in ascending order, and creating a matching table. At this moment, we used the insert method to implement this table by creating an empty Array of size 256, each index represents a corresponding value of the grayscale (0-255). Then by comparing the generated grayscale image with it original colored image, we could match each corresponding grayscale to its original color, and saving it into the table to generate an one to one corresponding matching table for the colors (lookup Table).

*Color[] table = new Color[256];*

*//scan through the whole image, and saving the colors into the right index*

*for (int y = 0; y < exampleMap.Height; y++)*

*for (int x = 0; x < exampleMap.Width; x++) {*

*table[exampleGrayMap.GetPixel(x, y).R] = exampleMap.GetPixel(x, y);*

*}*

This method is very efficient,because it just needs to traversal all the pixels once. But there are also some downside to it as well. When the different colors in the given color map correspond to the same gray scale value, we will only keep the most recent found color information, so that the previous color information will be lost (will lose part of the color information). This problem will be improved in the subsequent algorithm. ( In Improve section )

• Each gray value from the input grayscale image will be matched with the index of the color table generated from step 2. Once a matching value is found, the RGB value saved in matching index will replace that gray value in the input image. Because we have a one-to-one color table, so we only need to grayscale value from the image to find and get the corresponding color from the color table. However, due to the uncertainty of the table, there may be no gray value corresponding to the color in the lookup table. So we need to find the value of the gray scale value closest to the existing gray scale in the collocation relation in the lookup table. Therefore we need a method that respectively search it ascendingly and descendingly through the index to find the closest existing RBG.

*// to find up closest gray color*

*private static int findUpClosest(int gray, Color[] table) {*

*if (gray < 256 && table[gray].IsEmpty)*

*return findUpClosest(++gray, table);*

*else*

*return gray;*

*}*

*// to find down closest gray color*

*private static int findDownClosest(int gray, Color[] table) {*

*if (gray >= 0 && table[gray].IsEmpty)*

*return findDownClosest(--gray, table);*

*else*

*return gray;*

*}*

Then, by comparing the grayscale that we are looking for with the result of both search, the nearest index will be used and the RBG will be considered as result.

*private static int findClosest(int gray, Color[] table) {*

*int up = findUpClosest(gray, table);*

*int down = findDownClosest(gray, table);*

*if (up >= 256) return down;*

*else if (down < 0) return up;*

*else if (up - gray < down - gray) return up;*

*else return down;*

*}*

• Once every grayscale value of the input image has found it corresponding index in the color table. Then a simple linear nested for loop to replace each grayscale to the corresponding color will generate a colored image of the input image as result.

*private static Bitmap gray2color\_easy(Bitmap grayImage, Color[] colorTable) {*

*//create a blank bitmap the same size as grayImage*

*Bitmap newBitmap = new Bitmap(grayImage.Width, grayImage.Height);*

*// change each pixle to color base on color table*

*for (int y = 0; y < grayImage.Height; y++)*

*for (int x = 0; x < grayImage.Width; x++) {*

*newBitmap.SetPixel(x, y,*

*colorTable[findClosest(grayImage.GetPixel(x, y).R, colorTable)]);*

*}*

*// return*

*return newBitmap;*

*}*

Improvement

This above method can be improved from a number of aspects, different improvements may improve the quality of image generation, or for a particular type of picture will have a better effect. Due to our one to one mapping, we still have the problem of data loss, because we are replacing the previous found value with the most recent found value. If one gray value could represent multiple RBG, then only the most recent found RBG will be considered in the result. Therefore the result may vary depending on the algorithm for grayscale conversion.

For the preparation of each experiment, two or more color images with similar theme are selected and set as originals. And the originals will be used as comparison with the results. By using other imaging tools such as Photoshop to randomly generate grayscale images of our original images.

Next, the grayscale image will be cross processed using our colorization method with the originals, and saved as experiment result (gray1 process with Original2 generating a colored image of Original1 & gray2 process with Original1 generating a colored image of Original2). Then by compare the experiment result with it original image, we could see the quality this re-colorization method. Image differencing is often used to see the difference, the darker the comparison is the more similar the two image is. (Detail\_7)

1. The result and effect of different grayscale conversion algorithm.

Example of different grayscale conversion algorithm. (using different tools), we can generate two different grayscale images as result:

(image\_1) (gray\_1 ) (gray\_2 )

Applying gray\_1, and gray\_2 and image\_2 to our algorithm, and we got the following result:

(image\_2) (colorization result of gray\_1) (colorization result of gray\_2)

Image differencing colorization result of gray\_1 and colorization result of gray\_2 with image\_1:

(differencing\_1) (differencing\_2)

From the result, differencing\_2 is more close to a perfect colorization, as result we can conclude that gray\_2’s conversion used a similar algorithm as we did.

Considering that we are going to process images that are converted by different algorithms, and we will not be able to know which algorithm it was used to convert the image. Therefore we are providing users with different options to process their color tables. From the results, users would be about to choose the best image that they think is more realistic.

Therefore we take the 3 most used algorithm into count. (Detail\_8, Detail\_9 )

Grayscale processing will directly affect the results of the color table, so the grayscale that is the most similar to the with the original is the best processing method. But There is no way for us to automatically determine its gray processing, therefore we allow the user to determine which method has the result of their choice, by giving user the power to customizing the algorithm settings.

1.1 Desaturation

This is the simplest approach for grayscale processing, by finding the average value of maximum and minimum.

Lightness = (max(R,G,B) + min(R,G,B)) / 2

First, by using simple code to find the Max and Min value in RGB

*// find the value is Max*

*private static int MAX(int a,int b,int c) {*

*int r = a;*

*if (r < b) r = b;*

*if (r < c) r = c;*

*return r;*

*}*

*// find the value is Min*

*private static int MIN(int a, int b, int c) {*

*int r = a;*

*if (r > b) r = b;*

*if (r > c) r = c;*

*return r;*

*}*

Then, by using the Lightness algorithm to convert all pixel into gray value, and generating a new grayscale image.

*for (int y = 0; y < colorImage.Height; y++)*

*for (int x = 0; x < colorImage.Width; x++) {*

*int r = colorImage.GetPixel(x, y).R;*

*int g = colorImage.GetPixel(x, y).G;*

*int b = colorImage.GetPixel(x, y).B;*

*int gray = (MAX(r, g, b) + MIN(r, g, b)) / 2;*

*newBitmap.SetPixel(x, y, Color.FromArgb(gray, gray, gray));*

*}*

1.2 Luminosity

According to the equation of Luminosity Method, we considered that Red has contributed 21%, Green has contributed 72% which is greater in all three colors and Blue has contributed 7%.It decrease the contribution of red color, and increase the contribution of the green color, and put blue color contribution in between these two. Since red color has more wavelength of all the three colors, and green is the color that has not only less wavelength than red color but also green is the color, the result image is more brighter and gives a more soothing effect to the eyes.

Luminosity = 0.21 × R + 0.72 × G + 0.07 × B

Because each color is weighted differently, we can use the color matrix to quickly change the value of all pixels. So according to the weight, we can get the following matrix:

0.21 0.21 0.21 0 0

0.72 0.72 0.72 0 0

0.07 0.07 0.07 0 0

0 0 0 1 0

0 0 0 0 1

By multiplying each pixel by the above matrix operation, we can quickly get the required gray value.

*private static Bitmap color2gray\_Luminosity(Bitmap colorImage) {*

*//create a blank bitmap the same size as colorImage*

*Bitmap newBitmap = new Bitmap(colorImage.Width, colorImage.Height);*

*//get a graphics object from the new image*

*Graphics g = Graphics.FromImage(newBitmap);*

*//create the grayscale ColorMatrix*

*ColorMatrix colorMatrix = new ColorMatrix(*

*new float[][]{*

*new float[] {0.21f, 0.21f, 0.21f, 0, 0},*

*new float[] {0.72f, 0.72f, 0.72f, 0, 0},*

*new float[] {0.07f, 0.07f, 0.07f, 0, 0},*

*new float[] {0, 0, 0, 1, 0},*

*new float[] {0, 0, 0, 0, 1}});*

*//create some image attributes*

*ImageAttributes attributes = new ImageAttributes();*

*//set the color matrix attribute*

*attributes.SetColorMatrix(colorMatrix);*

*//draw the colorImage image on the new image*

*//using the grayscale color matrix*

*g.DrawImage(colorImage, new Rectangle(0, 0, colorImage.Width, colorImage.Height),*

*0, 0, colorImage.Width, colorImage.Height, GraphicsUnit.Pixel, attributes);*

*//dispose the Graphics object*

*g.Dispose();*

*return newBitmap;*

*}*

1.3 Average

Averaging method takes average of the three colors. 33% of Red,33% of Green and 33% of Blue. It is taking 33% of each, that means, each of the portion has same contribution in the image. Which result a rather darker image.

Average Brightness = (R + G + B) / 3

And we can create a color matirx as follow：

1/3 1/3 1/3 0 0

1/3 1/3 1/3 0 0

1/3 1/3 1/3 0 0

0 0 0 1 0

0 0 0 0 1

By multiplying each pixel by the above matrix operation, we can quickly get the required gray value.

*private static Bitmap color2gray\_Average(Bitmap colorImage) {*

*//create a blank bitmap the same size as colorImage*

*Bitmap newBitmap = new Bitmap(colorImage.Width, colorImage.Height);*

*//get a graphics object from the new image*

*Graphics g = Graphics.FromImage(newBitmap);*

*//create the grayscale ColorMatrix*

*ColorMatrix colorMatrix = new ColorMatrix(*

*new float[][]{*

*new float[] { 1f / 3f, 1f / 3f, 1f / 3f, 0, 0},*

*new float[] { 1f / 3f, 1f / 3f, 1f / 3f, 0, 0},*

*new float[] { 1f / 3f, 1f / 3f, 1f / 3f, 0, 0},*

*new float[] {0, 0, 0, 1, 0},*

*new float[] {0, 0, 0, 0, 1} });*

*//create some image attributes*

*ImageAttributes attributes = new ImageAttributes();*

*//set the color matrix attribute*

*attributes.SetColorMatrix(colorMatrix);*

*//draw the colorImage image on the new image*

*//using the grayscale color matrix*

*g.DrawImage(colorImage, new Rectangle(0, 0, colorImage.Width, colorImage.Height),*

*0, 0, colorImage.Width, colorImage.Height, GraphicsUnit.Pixel, attributes);*

*//dispose the Graphics object*

*g.Dispose();*

*return newBitmap;*

*}*

2. Fixed Difference

This is an improvement for result effect that is not satisfying. Especially when the generated color table is not rich enough. From our algorithm, if a corresponding color was not found for a grayscale, we call function *findClosest(int gray, Color[] table).* But for this improvement, instead of filling the gap with the closest color we found, we modify that color with the index difference to generate a new color. The new RGB be will old\_RGB subtract the difference (difference = p2 - p1). This improvement is based on a simple guess: if the gray value is not too far off, the RGB won’t be too far away.

Eg. if the grayscale 100 has no corresponding color saved at color table index[100], and we found a closest color at index[99] with RGB (15 125 87). The new RGB will become (16 126 88).

2.1 Implementation in C#

First, save the current gray value for every pixel:

*int gray = grayImage.GetPixel(x, y).R;*

Second, find the closest gray value that has a color:

*int closest = findClosest(gray, colorTable);*

And we can simply find the difference between the index:

*int dif = closest - gray;*

Then we subtract the RGB values with the difference to generate a new predicted RGB that we expect somewhat closer to the real color:

*int r = colorTable[closest].R - dif;*

*int g = colorTable[closest].G - dif;*

*int b = colorTable[closest].B - dif;*

The complete code to calculate the difference :

*private static Bitmap gray2color\_difference(Bitmap grayImage, Color[] colorTable) {*

*//create a blank bitmap the same size as grayImage*

*Bitmap newBitmap = new Bitmap(grayImage.Width, grayImage.Height);*

*// change each pixle to color base on color table*

*for (int y = 0; y < grayImage.Height; y++)*

*for (int x = 0; x < grayImage.Width; x++) {*

*int gray = grayImage.GetPixel(x, y).R;*

*int closest = findClosest(gray, colorTable);*

*int dif = closest - gray;*

*// get fixed rgb value*

*int r = colorTable[closest].R - dif;*

*int g = colorTable[closest].G - dif;*

*int b = colorTable[closest].B - dif;*

*// fixed it to avilible value*

*r = Math.Max(r, 0);*

*r = Math.Min(r, 255);*

*g = Math.Max(g, 0);*

*g = Math.Min(g, 255);*

*b = Math.Max(b, 0);*

*b = Math.Min(b, 255);*

*// set color*

*Color p = Color.FromArgb(r, g, b);*

*newBitmap.SetPixel(x, y, p);*

*}*

*// return*

*return newBitmap;*

*}*

2.2 Improvement 2 result comparison

This improvement mainly to focus the problem with color table that are not rich enough. Usually it is because the reference image’s color is too monotonous. And the procedure is list as follow:

First, we convert image\_3 into gray\_3

(image\_3) (gray\_3)

Then, trying to colorize gray\_3 using image\_4 as reference image and compare the improved result with the non-improved result. Image\_4 has a lack of darker colors, which will affect the result. Therefore the result will not be too ideal.

(reference image\_3)

Before improving： After improving:

(colorization result of gray\_3.1) (colorization result of gray\_3.2)

Therefore we can see that colorization result of gray\_3.2 is more toward realistic.

By doing the Image differencing test we can see:

Before improving: After improving:

(colorization result of gray\_3.1) (colorization result of gray\_3.2)

Although both colorization result are not ideal, but from the Image differencing test, colorization result of gray\_3.2 is somewhat closer to image\_3;

By doing another image difference between the first set of image differencing, as result we can see how much we have improved:

The color areas in this image are the areas that have darker colors in the original image, which are not provided by the reference image. Therefore, it is another step towards repairing color lost.

3. Improve color table and usage

In the above algorithm, there is a big drawback. We have only been using a one-to-one color table. This means that if there is a variety of RBG in a picture, that are pointed to the same grayscale, then some color information will be ignored during the process. So for images with more complex colors, the above colorization method will not be ideal. Therefore we implemented this improvement model, by redesigning a one to many color table. Then use a reference image with similar theme to find the similarity between both images, and use the similarity to complete the color reproduction of the grayscale image.

Image example:

Both image share a same theme → Sky, tree, grass.

Similarity between both image → Blue sky, Green tree, Green grass

→ Follows a top to bottom order

(a1) (a2)

Because both image shares a similar structure. Therefore we could assume that a pixel in image (a1) will have the same color in image (a2) at the same coordinate, or similar colors around that same coordinate. If so, by using a color image as reference that shares the same structure as the grayscale image to colorize that grayscale image, would give a more accurate result.

Eg. (b1) Eg.(b2)

Considering both (b1) and (b2) are grayscale images. (b1) is the image we need to colorize. (b2) is the a grayscale image we generated from the reference image that has a similar structure as (b1). For finding the corresponding color for the coordinate (5,3) in (b1), we will need to find the closest pixel in (b2) that matches the gray value in (b1), in this case is (7,1), then use the corresponding RGB from the color image of (b2) to color in the pixel on (b1).

3.1 Calculation for shortest distance

There exist many different ways for searching. Because we are working on a grid plane, the faster way to approach this problem is to use (c1) - by searching outward and counting the grids from the first match to the starting point to determine the shortest distance. On the other hand from the degree of human eye recognition, we choose (c2) as our method for our searching algorithm. - once a match has been found, using mathematical equation to find the distance. The downside to this method is it going to take longer than (c1).

(c1) (c2)

Calculation (c2)：(Detail\_10 in the section of Mathematics)

D = √( ( x2 - x1 )^2 + ( y2 - y1 )^2 )

In our case, the actual D value is not needed, only a boolean value the comparison between two distance is required , and the following equation is reached as result:

D1 < D2 => D1^2 < D2^2

By using ^2, Square Root (√) could be avoided in distance comparison.

3.2 Uniformity of different sizes

It requires two coordinates to find a distance. One coordinate will be at our starting point (the coordinate we are trying to color), the other coordinate will be where the cursor is at. But when the two images have a different size，we will get a discrepancy between the distance we found and and the distance we are looking for. Therefore, we need to unify the size of both images before any calculations.

All coordinates will be computed as a type of *float.* The Maximum height and width will be set to 1, and we can reach an equation as follow:

xf = x / width

Yf =y / herght

One the sizes are unified, then we can find the relationships between coordinates using the following function: (xGray & yGray → x & y coordinate in gray image that need to be colorized , xExample & yExamole → x & y coordinate in the reference image)

xGray / widthGray = xExample / widthExample

yGray / heightGray = yExample / heightExample

⇓

xExample = xGray \* widthExample / widthGray

yExample = yGray \* heightExample / heightGray

3.3 C# implementation

After improving the color table and before calculating the distance, we still need to find and save all colors to its corresponding index in the color table (all colors with the same grayscale will be saved in the same index) and filling in the empty index with function *findCloses* and modify it with previous improvement.

Due to the method we chose previously, the most straightforward method is to search through all the pixel, until the first matching pexels is found and compute its distance from the pixel that we are looking for, and we set that distance as the radius to minimize the search area for the next possible matching point. The search ends when there is no possible matching point in the search area or the distance reach zero. Then return the last point it has found as result.

*for (int y = 0; y < grayImage.Height; y++)*

*for (int x = 0; x < grayImage.Width; x++) {*

*// find the difference*

*int gray = grayImage.GetPixel(x, y).R;*

*int closest = findClosest(gray, colorTable);*

*int dif = closest - gray;*

*// find the nearest point*

*int ox = x \* exampleGray.Width / grayImage.Width,*

*oy = y \* exampleGray.Height / grayImage.Height;*

*int nx = x, ny = y;*

*int xs = 0, xe = exampleGray.Width, ys = 0, ye = exampleGray.Height;*

*int d = xe \* xe + ye \* ye; // set d to bigest*

*for (int yy = ys; yy < ye; yy++) {*

*int dy = (yy - oy) \* (yy - oy);*

*for (int xx = xs; xx < xe; xx++) {*

*if (exampleGray.GetPixel(xx, yy).R == closest) {// find color*

*int dx = (xx - ox) \* (xx - ox);*

*if (dx + dy < d) {// Distance is less than now*

*d = dx + dy;*

*nx = xx;*

*ny = yy;*

*// Narrow the search*

*int area = quickSqrt(d);*

*// if (oy - area > ys) ys = oy - area; // cannot to be used*

*if (oy + area < ye) ye = oy + area;*

*if (ox - area > xs) xs = ox - area;*

*if (ox + area < xe) xe = ox + area;*

*}*

*}*

*}*

*}*

*// get fixed rgb value*

*int r = exampleColor.GetPixel(nx, ny).R - dif;*

*int g = exampleColor.GetPixel(nx, ny).G - dif;*

*int b = exampleColor.GetPixel(nx, ny).B - dif;*

*// fixed it to avilible value*

*r = Math.Max(r, 0);*

*r = Math.Min(r, 255);*

*g = Math.Max(g, 0);*

*g = Math.Min(g, 255);*

*b = Math.Max(b, 0);*

*b = Math.Min(b, 255);*

*// set color*

*Color p = Color.FromArgb(r, g, b);*

*newBitmap.SetPixel(x, y, p);*

*}*

3.4 Comparison between before and after improve 3

Before:

We can see that the sky has been colored into green instead of blue. It is because the gray value of the sky and the grass are somewhat similar. Due to the one to one color table, blue colors are lost during the process.

After:

From this improvement we can see that this result is somewhat more realistic.

Image difference result:

(original) (result from Before) (result After)

From this comparison, the result from before improve, it give a more realistic color for the rocky stairs. some part of the stairs in the result after improvement are colored blue, due to the lake in reference image. On the other hand the problem of color lost is solved by this improvement. We can see that the blue sky is somewhat more realistic.

4. Result optimization - Simplest Color Balance

(Detail\_11, Detail\_12，Detail\_13，Detail\_14)

In the process of coloring the grayscale, we rely entirely on the color table. And the table is generated based on the reference image. because we are using two different image, the color distribution, color balance are not the same. In result, the result image will not be 100% ideal. And the color balance can be very good to adjust the color of the picture in the correct rendering, the result will looks more realistic, three-dimensional, while making up the lack of color in the color table.

There are lots of Color balancing calculation methods are for different needs. And we choose Simplest Color Balance for most of the photos / pictures that can be used to fix color rendering problems. This method is also widely used by Photoshop and other popular image processing software. This algorithm treats the brightest part of the picture as white, and the darkest part is black, so that all the values ​​of the RGB three color channels are stretched to the entire area of ​​[0-255] Covering the entire area. Often in order to avoid the abnormal situation, we will be bright pixels and dark pixels in accordance with a certain percentage has been saturated white or black. Usually this percentage we set to 1%.

4.1 copy the color value from image to three array

Since we can not change the original color value of the image at the time of statistics and calculation, we need to put these values ​​separately and save them into the corresponding array for later calculations and processing.

*int size = image.Width \* image.Height;*

*int[] redset = new int[size];*

*int[] greenset = new int[size];*

*int[] blueset = new int[size];*

*int i = 0;*

*for (int y = 0; y < image.Height; y++)*

*for (int x = 0; x < image.Width; x++) {*

*Color c = image.GetPixel(x, y);*

*redset[i] = c.R;*

*greenset[i] = c.G;*

*blueset[i] = c.B;*

*i++;*

*}*

4.2 Sort the pixel values

The three channels of RGB values will be saved to three different arrays, in order to facilitate the statistics and further calculations, we need to sort it from small to large.

*Array.Sort(redset);*

*Array.Sort(greenset);*

*Array.Sort(blueset);*

Using build in quick sort function in C#, each arrays are sorted by ascending order.

4.3 Pick the quantiles from the sorted pixels

Next we need to find the maximum and minimum values ​​in each color channel. However, due to the avoidance of abnormal values, we need to adjust a certain percentage (1%) on the entire channel list. Since we need to handle both the most bright parts and the most dark parts, therefore we will be adjusting the colors on both end of the channel by a half percentage (0.5%).

*float half\_percent = 1f / 200.0f;*

Through the size and percentage of the channel size, we can get the corrected minimum and maximum position (index):

*int mini = (int)Math.Floor((size - 1) \* half\_percent);*

*int maxi = (int)Math.Ceiling((size - 1) \* (1 - half\_percent));*

According to the minimum and maximum position of the index, we can respectively get the minimum value and the maximum value of the three color channels:

*int bottomR = redset[mini],*

*bottomG = greenset[mini],*

*bottomB = blueset[mini];*

*int topR = redset[maxi],*

*topG = greenset[maxi],*

*topB = blueset[maxi];*

4.4 Affine transform

Applying the Secondary optimization the adjust all pixels.

First, by obtain the original color.

*Color c = image.GetPixel(x, y);*

*int r = c.R,*

*g = c.G,*

*b = c.B;*

Adjust the colors on both end of the channel by a half percentage (0.5%) with the new calculated maximum and minimum value.

*r = Math.Max(r, bottomR);*

*g = Math.Max(g, bottomG);*

*b = Math.Max(b, bottomB);*

*r = Math.Min(r, topR);*

*g = Math.Min(g, topG);*

*b = Math.Min(b, topB);*

Apply an affine transform ax + b to each channel, computing a and b so that the maximal value

in the channel becomes 255 and the minimal value 0.

*r = (r - bottomR) \* 255 / (topR - bottomR);*

*g = (g - bottomG) \* 255 / (topG - bottomG);*

*b = (b - bottomB) \* 255 / (topB - bottomB);*

Lastly, Apply the new generated colors to the image:

*result.SetPixel(x, y, Color.FromArgb(r, g, b));*

4.5 Implementation of Improvement 4 using C#

*private static Bitmap autoColorBalance(Bitmap image) {*

*// copy the color value from image*

*int size = image.Width \* image.Height;*

*int[] redset = new int[size];*

*int[] greenset = new int[size];*

*int[] blueset = new int[size];*

*int i = 0;*

*for (int y = 0; y < image.Height; y++)*

*for (int x = 0; x < image.Width; x++) {*

*Color c = image.GetPixel(x, y);*

*redset[i] = c.R;*

*greenset[i] = c.G;*

*blueset[i] = c.B;*

*i++;*

*}*

*// Sort the pixel values*

*Array.Sort(redset);*

*Array.Sort(greenset);*

*Array.Sort(blueset);*

*// Pick the quantiles from the sorted pixels*

*float half\_percent = 1f / 200.0f;*

*int mini = (int)Math.Floor((size - 1) \* half\_percent);*

*int maxi = (int)Math.Ceiling((size - 1) \* (1 - half\_percent));*

*int bottomR = redset[mini],*

*bottomG = greenset[mini],*

*bottomB = blueset[mini];*

*int topR = redset[maxi],*

*topG = greenset[maxi],*

*topB = blueset[maxi];*

*// Affine transform*

*Bitmap result = new Bitmap(image.Width, image.Height);*

*for (int y = 0; y < image.Height; y++)*

*for (int x = 0; x < image.Width; x++) {*

*// get pixles*

*Color c = image.GetPixel(x, y);*

*int r = c.R,*

*g = c.G,*

*b = c.B;*

*// Saturate the pixels*

*r = Math.Max(r, bottomR);*

*g = Math.Max(g, bottomG);*

*b = Math.Max(b, bottomB);*

*r = Math.Min(r, topR);*

*g = Math.Min(g, topG);*

*b = Math.Min(b, topB);*

*// transform*

*r = (r - bottomR) \* 255 / (topR - bottomR);*

*g = (g - bottomG) \* 255 / (topG - bottomG);*

*b = (b - bottomB) \* 255 / (topB - bottomB);*

*// make color back to bitmap*

*result.SetPixel(x, y, Color.FromArgb(r, g, b));*

*}*

*// return*

*return result;*

*}*

4.6 Improvement Result

Since the color table we generated may not have a direct relationship with the ideal color table, by using color balance, we could somewhat make the color more toward ideal.

Applying improvement 4 on result generated from improvement 2

(gray\_3) (reference image) (colorization result of gray\_3.2)

Reuslt ：

In the details of this result, we can clearly see that the result is much better after color balance processing.

Result of Image differece with original color image：

(old comparison result from improvement 2) (new comparison result)

As we can conclude that the result of this improve is more ideal than the result we generated previously. And can provide automatic repair lack of color or insufficient color.

Experimental results

We have tried a lot of examples for each way. As the experimental results are too large, here we only list some of the experimental results to illustrate our experimental results (in each case we only cited an example). According to our previous algorithm, we have three gray-scale processing methods (used to generate color matching table), three color methods, and a color balance options. After arranging the combination, we can restore a gray scale and have 18 results.

Original image:

Input: Reference:

Without simple color balance

Easy

Difference

Nearest Distance

Lightness

Luminosity

Average

With simple color balance

Easy

Difference

Nearest Distance

Lightness

Luminosity

Average

Conclusions

For conclusions, this fully automated colorization process successfully re-colorize a grayscale image, and returns an image with reasonable colors as result. Through the experiments, the smaller the image is the better quality the result will be. However the efficiency of this program is way worse than we expected. During this research and study, we have encountered many other problems that we can not solve.

Efficiency is one of the major problem we come across and determine the best choice in an one to many situation. Otherwise, Due to the loss of color information while turning a color image into a grayscale image, the re-colorization problem cannot be solved at this moment. According to the HSV (Detail\_15) color division, the grayscale image only retains lightness (V) (Detail\_16) in HSV, therefore H and S will need to supplemented. Furthermore, this will occur a special case: if there are two adjacent but completely different colors have either similar or same brightness, then their display in the grayscale image will be similar as well. Based on the algorithms discussed previously, these two colors will be similar after re-colorization, which won’t be in line with the original image. The following is an example of this particular case.

In the original image, the top of the magnet has two different colors: blue and red. Because it is hard to see the bound between the top and the body of the magnet in the grayscale image, the computer will treat the top and the body of the magnet as an entirety and dye the whole magnet gray. Even if it is possible to determine a clear bound, the grayscale value of the top of the magnet will be converted to the same value, which makes it difficult to determine which one is red and which one is blue.

In cases like this one, it is difficult for the computer to determine whether the original colors have a big difference or not. If the two colors have the same grayscale value, there won’t be a bound between these two colors in the grayscale image. Therefore, these particular cases cannot be solved at this moment. It is difficult for the naturally performed pictures such as photos and natural images to accurately control the color of each area as human paintings such as cartoon and animation. Therefore, color difference determines these problems will or will not occur in re-colorization.

For future improvement, solving the one to many problem is the first step toward a successful colorization. Furthermore if Automatic Artificial intelligence could determine the object in the grayscale image, then coloring the ideal color will a piece of cake.

Bibliography

Detail\_1 - http://paper.ijcsns.org/07\_book/200704/20070402.pdf

Detail\_2 - https://www.codeproject.com/Articles/31120/Image-Transformation-Grayscale-to-Color

Detail\_3 - http://www.tannerhelland.com/3643/grayscale-image-algorithm-vb6/

Detail\_4 - https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

Detail\_5 - https://www.tutorialspoint.com/dip/grayscale\_to\_rgb\_conversion.htm

Detail\_6 - https://en.wikipedia.org/wiki/Color%E2%80%93color\_diagram

Detail\_7 - https://en.wikipedia.org/wiki/Image\_differencing

Detail\_8 - https://docs.gimp.org/2.6/en/gimp-tool-desaturate.html

Detail\_9 - https://en.wikipedia.org/wiki/Grayscale

Detail\_10 - https://en.wikipedia.org/wiki/Distance

Detail\_11 - https://en.wikipedia.org/wiki/Color\_balance

Detail\_12 - http://web.stanford.edu/~sujason/ColorBalancing/simplestcb.html

Detail\_13 - http://www.morethantechnical.com/2015/01/14/simplest-color-balance-with-opencv-wcode/

Detail\_14 - http://www.ipol.im/pub/art/2011/llmps-scb/article.pdf

Detail\_15 - https://en.wikipedia.org/wiki/HSL\_and\_HSV

Detail\_16 - https://en.wikipedia.org/wiki/Lightness