Still Image Colorization

Objective of the project

Colorization represents a computer-assisted process of adding color to grey-scale still images or monochrome movies. Since the mapping from grey-scale pixel to color pixel is not unique, this process generally requires some user interactions to help confine the color selection. The goal of the project is to implement a colorization algorithm similar to [1, 2, 3] and a user-friendly interface with Visual C++ and OpenCV library. After user scribbles the color of some selected pixels, the program will automatically propagate the color to the remaining regions to generate a comfortable color image with good subjective quality.

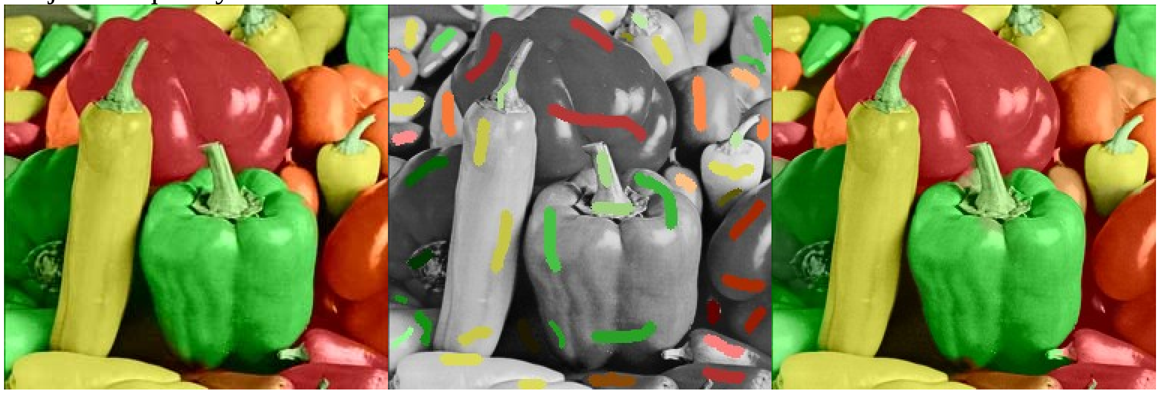


Figure 1. Illustration of colorization process

Problem Formulation

I use the YCbCr color space to colorize the gray image. The term *color* and *chrominance* represent Cb or Cr component in the following discussion. A colorization algorithm blending the color close to a pixel with iterative dynamic programming is designed and implemented. First the intrinsic distance is introduced to define the contiguity of two pixels. The intrinsic distance between two adjacent pixels is defined as the absolute value of the intensity difference. The intrinsic distance of two arbitrary pixels is defined as the minimum accumulative sum of intrinsic distances of a curve linking these two pixels. Each pixel with user specified color propagates its color to all the adjacent pixels, and each pixel maintains a list of top 3 colors with the minimum intrinsic distances. The resultant color is the weighted sum of these 3 colors. The color propagate process is an iterative dynamic programming procedure on the 8-connect pixel array. The more strict descriptions are presented as follows.

Denote the gray intensity of a pixel s as $Y(s)$, the chrominance of s is denoted as $Chroma(s)$, which represents Cb or Cr. The intrinsic distance of two adjacent pixels s and t is defined as:

$$d(s,t) = abs(Y(s) - Y(t))$$

For an 8-connect curve $C = \{p_1, p_2, ..., p_m)$, the intrinsic distance of C is defined as

$$d(C) = \sum_{i=1}^{m-1} d(p_i, p_{i+1})$$

The intrinsic distance of two arbitrary pixels *s* and *t* is defined as the minimum intrinsic distance of a curve, which link *s* and *t* :

$$d(s,t) \doteq \min d(C), \text{where } p_1 = s \text{ and } p_m = t$$

Suppose users have specified *N* different colors and the set of pixels with the same color are denoted as $\Omega_n, n = 1,...,N$, the intrinsic distance of pixel *s* to Ω_n is defined as

$$d(s,\Omega_n) = \min d(s,t), \forall t \in \Omega_n$$

$\forall s \notin \Omega_n, n = 1,...,N$, the final chrominance is the weighted sum of the top *K*=3 chrominance with the minimum intrinsic distances to *s*.

$$Chroma(s) = \frac{\sum_{i=1}^{3} w(d(s,\Omega_{ki}))Chroma(k_i)}{\sum_{i=1}^{3} w(d(s,\Omega_{ki}))}$$

where the weight function $w(d) = d^{-r}, r = 4$. The factor *r* can control the smoothness of the chrominance transition.

The following is the illustration of intrinsic distance:

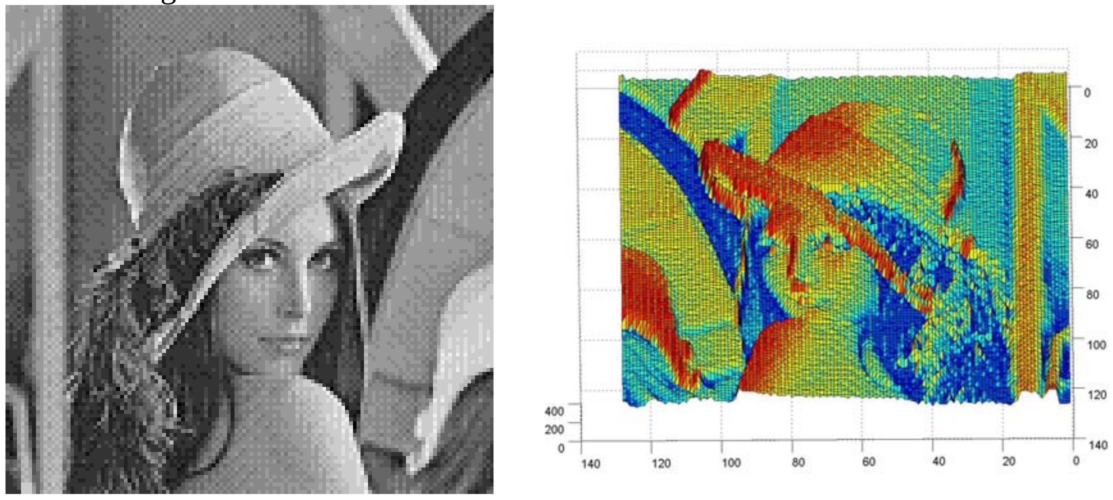


Figure 2. Illustration of intrinsic distance

Implementation Methods

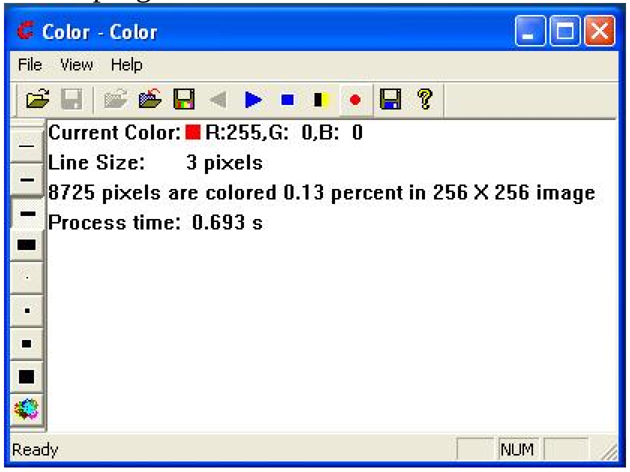
The minimum intrinsic distance of every pixel to $\Omega_n, n = 1,...,N$ is found by an ***iterative dynamic programming*** algorithm. Each pixel maintains the top 3 colors with the minimum intrinsic distances and propagate these 3 color to its 8-connect neighbor pixels.

Active pixel set(APS) is defined as the set of pixels ready to propagate their color to adjacent pixels. The initial set is $\Omega_n, n = 1,...,N$ with zero intrinsic distance. When a pixel *t* is propagating its neighbor pixels $p_1 - p_8$, if the 3 minimum intrinsic distances of p_i is changed, this pixel p_i is added into the APS. After the pixel *t* already propagates to all its 8 neighbors, it is removed from APS. This process is performed iteratively until the APS is empty. Note that a pixel may be added to the APS more than once. Since the intrinsic distance exists for each pixel, the convergence of this procedure is guaranteed. The pseudo codes for the procedure is list below:

```
Initialization:
$APS = \Omega_n, n = 1, ..., N$
while (APS is not empty)
{
    for each pixel t in APS
    {
        propagates its 3 intrinsic distances to its 8 neighbor pixels $\{p_1, ..., p_8\}$
        if ( $p_i$ 's list of 3minimum intrinsic distances is modified)
        {
            add $p_i$ to APS
        }
    }
    remove t from APS
}
```

System

The project is built with Microsoft Visual C++ 6.0 and the OpenCV 4.0 library. Users can scribble color on the gray-scale images with different line width and erase the color. One highlight of the program is to support *Undo* and *Redo* functions. If the users are not satisfied with the operations they can exactly reverse the functions. This is implemented with *command* design pattern. The color used can be picked up from a color image or selected from system color palette, and user-defined color can be saved for future use. Other features include saving and loading the color mask image and save the resultant image. The interface of the program is as follows:



```
Current Color: 🟥 R:255,G:  0,B:  0
Line Size:      3 pixels
8725 pixels are colored 0.13 percent in 256 X 256 image
Process time:  0.693 s
```

Figure 3. System interface

Experimental Result

The colorization system is built and tested on a Pentium IV 3GHz desktop. Various images are tested. Here 4 representative images are shown in Figure 4. The percentage of pixels user specify the color and the processing time are list in the Table 1.

Image	Image resolution	# of colored pixels	Percentage(%)	Processing time (s)
Shirley	333*440	4396	3.0	2.03
Lily	400*300	3355	2.8	2.74
Sunset	400*300	3568	3.0	2.25
Arch	480*321	5668	3.7	2.35

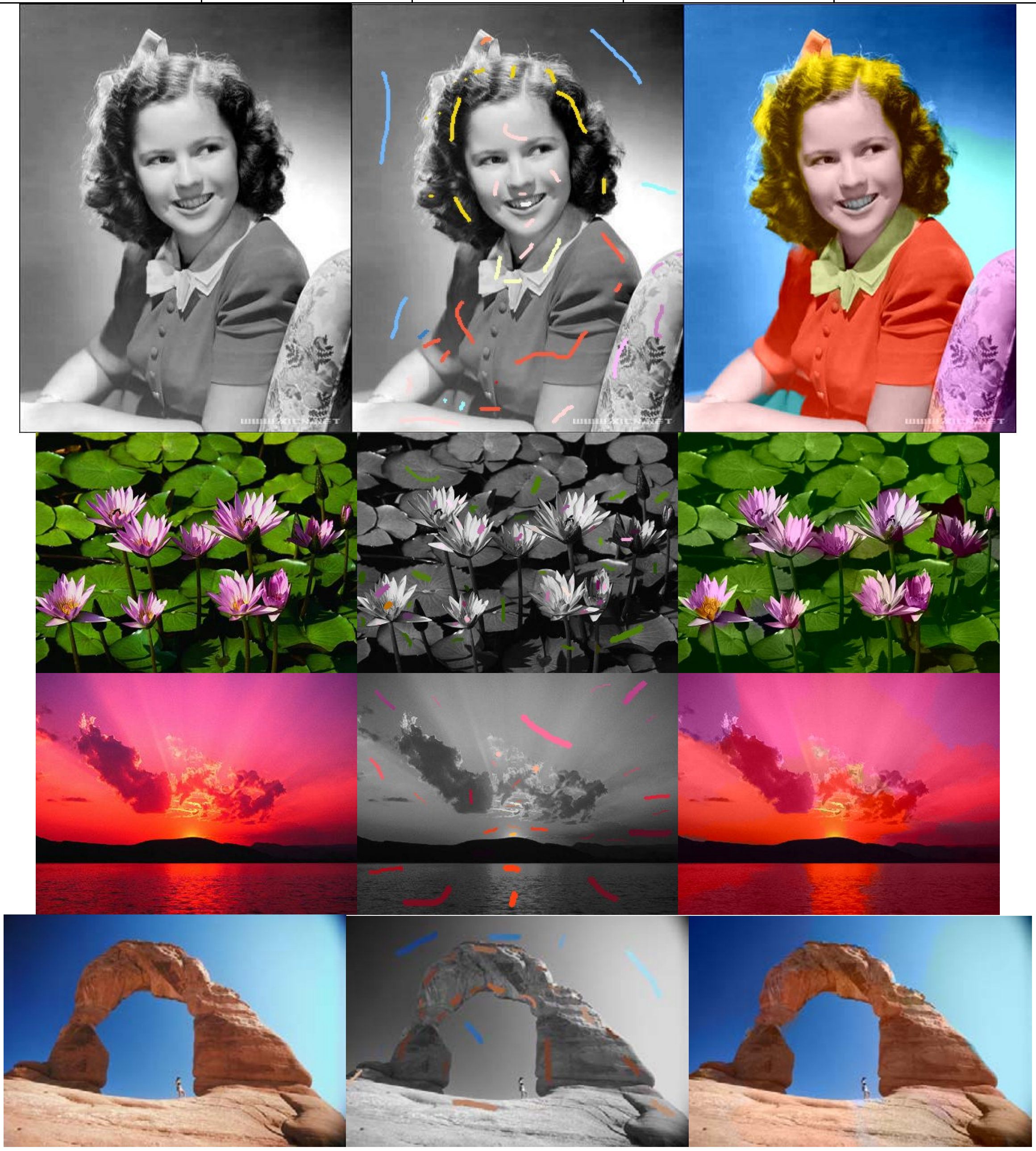


Figure 4. Test results

Conclusions

An efficient still image colorization algorithm by blending color with the intrinsic distances is implemented by iterative dynamic programming. The idea of color blending is borrowed from the references while the iterative dynamic programming method is designed with appropriate data structures in this project. Experiments demonstrate that the algorithm can achieve very good subjective quality for a diversity of images with only 3-4% pixels color specified by users.

Reference:

[1] Anat Levin, Dani Lischinski and Yair Weiss. "Colorization using Optimization". ACM Transactions on Graphics, Aug 2004, Siggraph'04, pp.689-693.
[2] Liron Yatziv and Guillermo Sapiro. "Fast Image and Video Colorization using Chrominance Blending".
[3] Daniel Sykora, Jan Burianek and Jiri Zara. "Unsupervised Colorization of Black-and-White Cartoons", Proceedings of NPAR 2004. Annecy, ACM SIGGRAPH, p.121-127.

Appendix:

Core source codes:

```
#define MAX_BLEND_COLOR 3   //Max number of color to be blended
#define MAX_DISTANCE 65535    //Max intrinsic distance of pixels
/////////////////////////////////////////////////////////////////////////////////////
// Pixels representation for blending colorization algorithm
/////////////////////////////////////////////////////////////////////////////////////
class CBlendPixel
{
    int m_x;    // x-coordinate of the pixel
    int m_y;    // y-coordinate of the pixel
   COLORREF m_c[MAX_BLEND_COLOR];  // maintained chrominance
    float    m_d[MAX_BLEND_COLOR];          // maintained intrinsic distance
};
typedef set<CBlendPixel*> BlendPixelSet;
```

```
/////////////////////////////////////////////////////////////////////////////////////
// Colorization Algorithm
// Blending the color with the intrinsic distance as the weights
/////////////////////////////////////////////////////////////////////////////////////
class CBlendColorize : public CColorize
{
public:
      BOOL Colorize(IplImage* pColorImage,const IplImage* pGreyImage, const IplImage* pLayerImage);

      CBlendColorize();
       virtual ~CBlendColorize();

protected:
              void BlendColor();
              void PropagateColor();
              void GenerateColoredSet();
```

```
        BOOL ModifyPixel(CBlendPixel& testPixel,const CBlendPixel & curPixel);
        BOOL Internal8(int x,int y);
        inline double WeightDistance(double d);

        void AllocateArray();
        void ReleaseArray();
        int  m_iColored;

        BlendPixelSet m_ActiveSet;
        CBlendPixel*** m_PixelArray;
};
```

```
// Iteratively propagate the active colored pixels to their adjacent pixels until the equilibrium
void CBlendColorize::PropagateColor()
{
        CString str;

        BlendPixelSet::iterator it =m_ActiveSet.begin();
        BlendPixelSet::iterator it_temp;
        while (!m_ActiveSet.empty())
        {
                it =m_ActiveSet.begin();
                while (it!=m_ActiveSet.end())
                {
                        CBlendPixel * pPixel = *it;
                        for(int y=pPixel->m_y-1;y<=pPixel->m_y+1;y++)
                                for(int x=pPixel->m_x-1;x<=pPixel->m_x+1;x++)
                                {
                                        if (x==pPixel->m_x||y==pPixel->m_y)
                                        if (x>=0&&x<=m_width-1&&y>=0&&y<=m_height-1)
                                        {
                                                CBlendPixel * testPixel = m_PixelArray[y][x];
                                                if (ModifyPixel(*testPixel,*pPixel))
                                                        m_ActiveSet.insert(testPixel);
                                        }
                                }
                        it_temp = it; it++;
                        m_ActiveSet.erase(it_temp);
                }
        }
}
```

```
//Modify the color list of curPixel according to testPixel
BOOL CBlendColorize::ModifyPixel(CBlendPixel& testPixel,const CBlendPixel & curPixel)
{

        if (cvGetReal2D(m_pY,testPixel.m_y,testPixel.m_x)==1)
                return FALSE;
        double y1 = cvGetReal2D(m_pGreyImage,curPixel.m_y,curPixel.m_x);
        double y2 = cvGetReal2D(m_pGreyImage,testPixel.m_y,testPixel.m_x);
        double d = abs(y1-y2);

        BOOL result = FALSE;
        for(int i=0;i<MAX_BLEND_COLOR;i++)
        {
                if (curPixel.m_d[i]<MAX_DISTANCE)
```

```
                    {
                      int j=0;
                      while (((curPixel.m_d[i]+d)>=testPixel.m_d[j])&&(j<=MAX_BLEND_COLOR))
                      {
                                if (curPixel.m_c[i]==testPixel.m_c[j])
                                {
                                          j=MAX_BLEND_COLOR;
                                          break;
                                }
                                j++;
                      }
                      if (j<MAX_BLEND_COLOR)
                      {
                                if (curPixel.m_c[i]!=testPixel.m_c[j])
                                for(int k=MAX_BLEND_COLOR-1;k>j;k--)
                                {
                                          if (testPixel.m_c[k-1]!=curPixel.m_c[i])
                                          {
                                                  testPixel.m_c[k]=testPixel.m_c[k-1];
                                                  testPixel.m_d[k]=testPixel.m_d[k-1];
                                          }
                                }
                                testPixel.m_c[j]=curPixel.m_c[i];
                                testPixel.m_d[j]=curPixel.m_d[i]+d;
                                result = TRUE;
                      }
                       else
                                break;
                    }
          }
          return result;
}
```

```
//////////////////////////////////////////////////////////////////////////////
// Command Pattern to support Undo and Redo functions
//////////////////////////////////////////////////////////////////////////////
class CCommand
{
public:

          virtual BOOL Execute(IplImage *pImage,WPARAM wParam, LPARAM lParam);
          virtual BOOL UnExecute(IplImage *pImage);
          virtual BOOL ReExecute(IplImage* pImage);

          CCommand();
          virtual ~CCommand();
protected:
          BOOL m_bExecuted;
};
```