Visualise My Picture – Project Summary
Winter Term 2015/2016

Jiaqi Weng

October 12, 2017

1 Introduction

This document is a summary about the project 'Visualise My Picture'.

1.1 What is 'Visualise My Picture'

'Visualise My Picture' is a project focus on research of picture visualisation, including how to analyse if the composition is one-third rule or not, show the 3D model and make a 3D presentation of a black and white picture.

1.2 Composition (Rule of Thirds)

The rule of thirds is one of the most important composition rules used by photographers to create high-quality photos. The rule of thirds states that placing important objects along the imagery thirds lines or around their intersections often produces highly aesthetic photos.

In our project, we mainly focus on detecting the rule of thirds [1]. It needs to detect some specific objects, such as face, important object detection, in general, is still a challenging problem. Methods below used in our project.

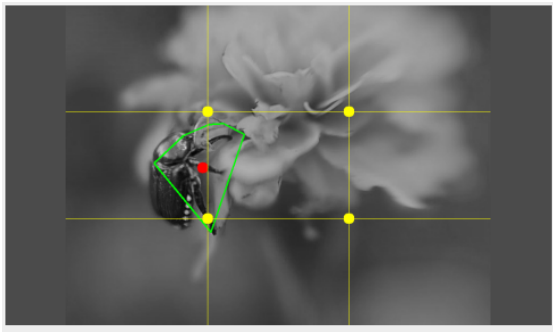

Figure 1: Rule of Thirds

1.2.1 First Gradient

Using first derivative [2] (max value) to detect the notorious changes of the pixel intensity.

1.2.2 Second Gradient

Using second derivative [3] (zero value) to detect the edges in an image.

1.2.3 Saliency map

Use VOCUS [4] to generate independent on-center and off-center intensity maps, i) On-center maps which respond to bright areas surrounded by a dark background, and ii) off-center maps which respond to dark areas surrounded by a bright background. Most attention systems build upon the base of the Theory of Attention using center-surround differences.

Figure 2: 1: Origin, 2: First Gradient, 3: Second Gradient, 4: Saliency Map

1.3 Color Space

A color space [5] is a specific organisation of colors. In combination with physical device profiling, it allows for reproducible representations of color, in both analog and digital representations.

1.3.1 RGB

An RGB [5] color space is any additive color space based on the RGB color model. A particular RGB color space is defined by the three chromaticities of the red, green, and blue additive primaries, and can produce any chromaticity that is the triangle defined by those primary colors.

1.3.2 HSV

HSV [6] is the most common cylindrical-coordinate representation of points in an RGB color model. This representation rearrange the geometry of RGB in an attempt to be more intuitive and perceptually relevant than the cartesian (cube) representation.

1.3.3 Lab

A Lab [7] color space is a color-opponent space with dimension L for lightness and a and b for the color-opponent dimensions, based on nonlinearly compressed (e.g. CIE XYZ color space) coordinates.

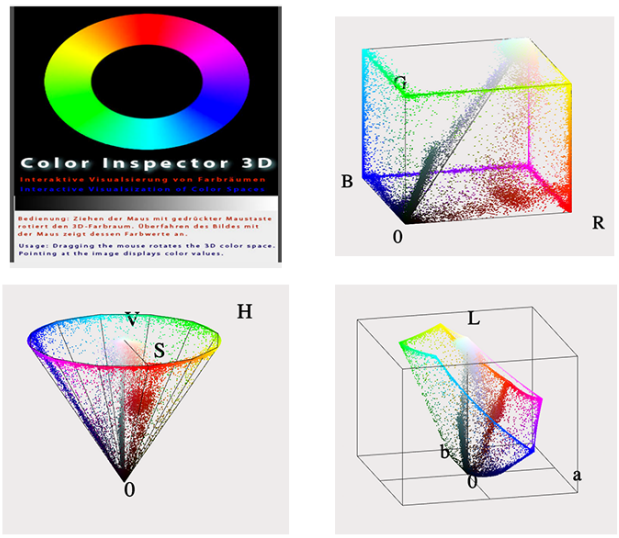

Figure 3: 1: Image, 2: RGB, 3: HSV, 4: Lab

1.4 3D visulization

Using intensity of every pixel from a black and white photo as a Z value, and create a xyz space to show a picture.
From the 3D visualisation of the image, we can see the differences directly from the height of every pixel, and we can define the focus area by calculate the biggest difference in every neighbour pixels.

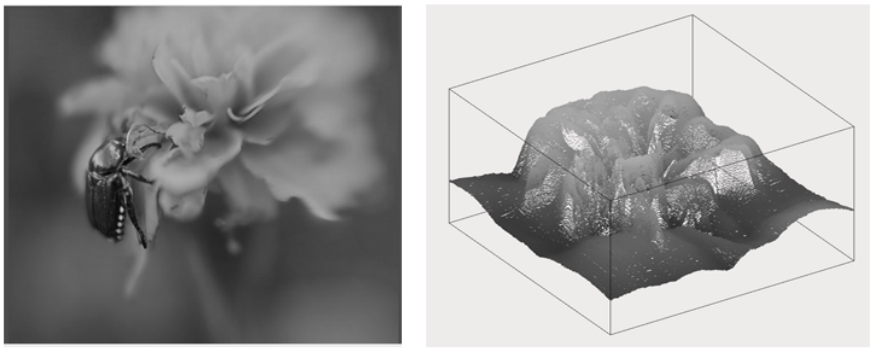

Figure 4: Left: Image, Right: 3D visualisation

2 Implementation

2.1 Composition (Rule of Thirds)

The source files (gradient.h, gradient.cpp) work for first derivative and second derivative, and also have function for draw convex hull and rule of third. And files (staticSaliencyFineGrained.h, staticSaliencyFineGrained.cpp) work for saliency map.

2.1.1 First Gradient [2]

Assuming that the image to be operated is I:
1, We calculate two derivatives:
Horizontal changes: This is computed by convolving I with a kernel G_x with odd size.For example for a kernel size of 3, G_x would be computed as:

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I$$

Vertical changes: This is computed by convolving I with a kernel G_y with odd size.For example for a kernel size of 3, G_y would be computed as:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * I$$

There is two functions Scharr and Sobel to calculate first derivative.

```
Mat grad_x, grad_y;
Mat abs_grad_x, abs_grad_y;
Scharr( src_gray, grad_x, ddepth, 1, 0, scale, delta,
    BORDER_DEFAULT );
//Sobel( src_gray, grad_x, ddepth, 1, 0, 3, scale, delta,
    BORDER_DEFAULT );
convertScaleAbs( grad_x, abs_grad_x );

Scharr( src_gray, grad_y, ddepth, 0, 1, scale, delta,
    BORDER_DEFAULT );
//Sobel( src_gray, grad_y, ddepth, 0, 1, 3, scale, delta,
    BORDER_DEFAULT );
convertScaleAbs( grad_y, abs_grad_y );
```

Functions:

```
void Scharr(InputArray src, OutputArray dst, int ddepth, int dx,
    int dy, double scale=1, double delta=0, int borderType=
    BORDER_DEFAULT )
void Sobel(InputArray src, OutputArray dst, int ddepth, int dx, int
     dy, int ksize=3, double scale=1, double delta=0, int
    borderType=BORDER_DEFAULT )
```

Parameters:
src - input image.
dst - output image of the same size and the same number of channels as src.
ddepth - output image depth (see Sobel() for the list of supported combination of src.depth() and ddepth).
dx - order of the derivative x.
dy - order of the derivative y.
scale - optional scale factor for the computed derivative values; by default, no scaling is applied (see getDerivKernels() for details).
delta - optional delta value that is added to the results prior to storing them in dst.
borderType - pixel extrapolation method (see borderInterpolate() for details).

2, At each point of the image we calculate an approximation of the gradient in that point by combining both results above:

$$G = \sqrt{G_x^2 + G_y^2}$$

Although sometimes the following simpler equation is used:

$$G = |G_x| + |G_y|$$

Use opencv function addWeighted to approximate the gradient by adding both directional gradients.

```
addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad );
```

2.1.2 Second Gradient [3]

1, The Laplacian operator is defined by:

$$Laplace(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

2, The Laplacian operator is implemented in OpenCV by the function Laplacian. In fact, since the Laplacian uses the gradient of images, it calls internally the Sobel operator to perform its computation.

```
void Laplacian(InputArray src, OutputArray dst, int ddepth, int
    ksize=1, double scale=1, double delta=0, int borderType=
    BORDER_DEFAULT )
```

Parameters:
src - Source image.
dst - Destination image of the same size and the same number of channels as src .
ddepth - Desired depth of the destination image.
ksize - Aperture size used to compute the second-derivative filters. See getDerivKernels() for details. The size must be positive and odd.
scale - Optional scale factor for the computed Laplacian values. By default, no scaling is applied. See getDerivKernels() for details.
delta - Optional delta value that is added to the results prior to storing them in dst.
borderType - Pixel extrapolation method. See borderInterpolate() for details.

2.1.3 Saliency map [4]

1, The original color image is converted into grayscale. Then, a Gaussian image pyramid is created. This is achieved by applying a 3x3 Gaussian filter to the grayscale image, and after that, scaling it down by a factor of two on each axis.

```
cv::cvtColor(srcArg, gray, cv::COLOR_BGR2GRAY);

// smooth pixels at least twice, as done by Frintrop and Itti
cv::GaussianBlur( gray, gray, cv::Size( 3, 3 ), 0, 0 );
cv::GaussianBlur( gray, gray, cv::Size( 3, 3 ), 0, 0 );
```

2, Calculates on-center and off-center differences in the three images that represent scales s $\in \{12, 24, 28, 48, 56, 112\}$ respectively.
we define centre and surround by,

$$surround(x, y, s) = \frac{rectSum(x - s, y - s, x + s, y + s) - i(x, y)}{(2s + 1)^2 - 1}$$

$$center(x, y) = i(x, y)$$

and on-center and off-center differences map is defined,

$$Int_{on,s}(x, y) = max\{center(x, y) - surround(x, y, s), 0\}$$

$$Int_{off,s}(x, y) = max\{surround(x, y, s) - center(x, y), 0\}$$

```
for(y = 0; y < gray.rows; y++)
{
    for(x = 0; x < gray.cols; x++)
    {
        point.x = x;
        point.y = y;
        value = getMean(integralImage, point, neighborhood, gray.at
    <uchar>(y, x));

        meanOn = gray.at<uchar>(y, x) - value;
        meanOff = value - gray.at<uchar>(y, x);

        if(meanOn > 0)
            intensityScaledOn.at<uchar>(y, x) = (uchar)meanOn;
        else
            intensityScaledOn.at<uchar>(y, x) = 0;

        if(meanOff > 0)
            intensityScaledOff.at<uchar>(y, x) = (uchar)meanOff;
        else
            intensityScaledOff.at<uchar>(y, x) = 0;
    }
}
```

3, An on-center intensity map is calculated. This is done summing the six on-center intensity submaps pixel by pixel. An off-center intensity map is generated the same

way, using the off-center submaps.

$$Int_{on} = \sum_{s} Int_{on,s}$$

$$Int_{off} = \sum_{s} Int_{off,s}$$

```
for(i=0;i<numScales;i++)
{
    for(y=0;y<height;y++)
        for(x=0;x<width;x++)
        {
               currValOn = intensityScaledOn[i].at<uchar>(y, x);
               if(currValOn > maxValOn)
                   maxValOn = currValOn;

               currValOff = intensityScaledOff[i].at<uchar>(y, x);
               if(currValOff > maxValOff)
                   maxValOff = currValOff;

              mixedValuesOn.at<unsigned short>(y, x) += currValOn;
              mixedValuesOff.at<unsigned short>(y, x) += currValOff;
        }
}
```

4, Sum up both maps.

```
for(y=0;y<height;y++)
    for(x=0;x<width;x++)
    {
        intensity.at<uchar>(y, x) =
         (uchar) (255. * (float)
      (intensityOn.at<uchar>(y, x) + intensityOff.at<uchar>(y, x))
         / (float)maxVal);
    }
```

2.1.4 Convex Hull [8]

1, Scan every pixel of the photo, add the white pixel to the vertex.

```
std::vector<cv::Point> points;
cv::Mat_<uchar>::iterator it = src.begin<uchar>();
cv::Mat_<uchar>::iterator end = src.end<uchar>();
for (; it != end; ++it)
  if (*it)
    points.push_back(it.pos());
```

2, Use opencv function convexHull to find the hull object.

```
vector<vector<Point> >hull(1);
convexHull( Mat(points), hull[0], false );
```

3, Draw the hull result on the black and white photo.

```
Mat _gray;
cvtColor(image.clone(), _gray, COLOR_RGB2GRAY );
cvtColor(_gray, _gray, COLOR_GRAY2BGR);
Mat drawing = _gray;
drawContours( drawing, hull, 0, color, 2, 8, vector<Vec4i>(), 0,
    Point() );
```

4, Lastly, draw the centre point of the hull object.

```
Moments mu=moments( hull[0], false );
Point2f center = Point2f(mu.m10/mu.m00, mu.m01/mu.m00);
circle( drawing, center, 10, Scalar(0,0,255), -1, 8, 0 );
```

2.2 Color Space [5]

The source files (colorspacewidget.h, colorspacewidget.cpp) work for 3D color space visualisation, including RGB, HSV and Lab.
Firstly, use opencv function "calcHist" to get the histogram of the image.

```
int histSize[3] = {256, 256, 256};
float range[2] = {0, 256};
const float * ranges[3] = {range, range, range};
int channels[3] = {0, 1, 2};

calcHist(&_image, 1, channels, Mat(), _hist, 3, histSize, ranges);
```

Function:

```
void calcHist(const Mat* images, int nimages, const int* channels,
    InputArray mask, OutputArray hist, int dims, const int*
    histSize, const float** ranges, bool uniform=true, bool
    accumulate=false )
```

Parameters:
images - Source arrays.
nimages - Number of source images.
channels - List of the dims channels used to compute the histogram.
mask - Optional mask.
hist - Output histogram, which is a dense or sparse dims -dimensional array.
dims - Histogram dimensionality that must be positive and not greater than CV_MAX_DIMS (equal to 32 in the current OpenCV version).
histSize - Array of histogram sizes in each dimension.
ranges - Array of the dims arrays of the histogram bin boundaries in each dimension.
uniform - Flag indicating whether the histogram is uniform or not (see above).
accumulate - Accumulation flag.

Then, draw 3D color space model by OpenGL.

2.2.1 RGB [5]

1, Draw a Cube.
2, Define the black, red, green and blue point.
3, Iterate the histogram map and convert every color pixel value rgb to 3D RGB space coordinates xyz.

```
glBegin(GL_POINTS);
for (int i=0; i<histSize[0]; i++) {
    for (int j=0; j<histSize[1]; j++) {
        for (int k=0; k<histSize[2]; k++) {
            if(hist.at<double>(k, i, j) > 0)
            {
                drawRGBPixel(j,i,k);
            }
        }
    }
}
glEnd();
```

4, Draw all the pixel in the cube.

```
void drawRGBPixel(float r, float g, float b)
{
  float x = -1.0f + 2*r/255;
  float y = -1.0f + 2*g/255;
  float z = 1.0f-2*b/255;

  glColor3f(r/255, g/255, b/255);
  glVertex3f(x,y,z);
}
```

2.2.2 HSV [9]

1, Draw a Cone.
2, Draw the original H, S and V point.
3, Iterate the histogram map and convert the rgb value to hsv value.

```
hsvColor ColorSpaceWidget::rgb2hsv(float r, float g, float b);
```

```
glBegin(GL_POINTS);
for (int i=0; i<histSize[0]; i++) {
    for (int j=0; j<histSize[1]; j++) {
        for (int k=0; k<histSize[2]; k++) {
            if(hist.at<double>(k, i, j) > 0)
            {
                hsvColor hsv = rgb2hsv(j,i,k);
                drawHSVPixel(j, i, k, hsv.h, hsv.s, hsv.v);
            }
        }
```

```
    }
}
glEnd();
```

4, Transfer hsv value to the 3D HSV space coordinates xyz and draw every color point.

```
void ColorSpaceWidget::drawHSVPixel(float r, float g, float b,
    float h, float s, float v)
{
    float x = s * sin(h * PI / 180.0);
    float y = 2 * (v/255) - 1;
    float z = s * cos(h * PI / 180.0);

    glColor3f(r/255, g/255, b/255);
    glVertex3f(x,y,z);
}
```

2.2.3 Lab [7]

1, Draw a Cube.
2, Draw the original L, a and b point.
3, Iterate the histogram map and convert the rgb value to Lab value.

```
labColor ColorSpaceWidget::rgb2lab(float R, float G, float B);
```

```
glBegin(GL_POINTS);
for (int i=0; i<histSize[0]; i++) {
    for (int j=0; j<histSize[1]; j++) {
        for (int k=0; k<histSize[2]; k++) {
            if(hist.at<double>(k, i, j) > 0)
            {
                    labColor lab = rgb2lab(j,i,k);
                    drawLABPixel(j, i, k, lab.l, lab.a, lab.b);
            }
        }
    }
}
glEnd();
```

4, Transfer Lab value to the 3D Lab space coordinates xyz and draw every color point.

```
void ColorSpaceWidget::drawLABPixel(float r, float g, float b,
    float l, float a, float bb)
{
    float x = a/128;
    float y = l/100-1.0f;
```

```
    float z = -bb/128;

    glColor3f(r/255, g/255, b/255);
    glVertex3f(x,y,z);
}
```

2.3 3D visulization

The source files (image3dwidget.h, image3dwidget.cpp) work for 3D white and black picture visualisation.
As color space working, implement by OpenGL.
1, Draw a Cube as a coordinates space.
2, Loop the image, read the intensity of every pixel as z coordinate value.

```
glBegin(GL_POINTS);
for(int i=0; i< _src.rows; i++)
    for(int j=0; j < _src.cols; j++)
        drawPixel(i,j, (int)_src.at<uchar>(i,j));
glEnd();
```

3, transfer the position and intensity value to space xyz value and draw the image.

```
void Image3dWidget::drawPixel(int x, int y, int z)
{
    glColor3f((float)z/255.0f, (float)z/255.0f, (float)z/255.0f);
    float xx = 2* (float)x/(float)_src.rows - 1.0f;
    float yy = 2* (float)y/(float)_src.cols - 1.0f;
    float zz = 2* (float)z/255.0f-1.0f;

    glVertex3f(xx, zz, yy);
}
```

3 Usage

3.1 Composition

Use the source files (staticSaliencyFineGrained.h, staticSaliencyFineGrained.cpp, gradient.h, gradient.cpp), initialise objects.

```
#include "gradient.h"
#include "staticSaliencyFineGrained.h"
...
...
GradientHelper gradhlp;
StaticSaliencyFineGrained saliencyGenerator;
```

Then, call the gradient and saliency map functions.

```
//first gradient
    _grad = gradhlp.FirstGradient(_image);
....
//second gradient
    _grad = gradhlp.SecondGradient(_image);
....
//saliency map
    saliencyGenerator.computeSaliencyImpl(_gray, _grad);
```

Functions:

```
    Mat FirstGradient(Mat image);
...
    Mat SecondGradient(Mat image);
...
    bool computeSaliencyImpl( cv::Mat image, cv::Mat &saliencyMap )
    ;
```

Parameters:
image - input image
saliencyMap - output saliency map

3.2 Color Space

Use the source files (colorspacewidget.h, colorspacewidget.cpp), initialise widget in ui frame.

```
...
<widget class="ColorSpaceWidget" name="colorspace" native="true"/>
...
<customwidget>
   <class>ColorSpaceWidget</class>
   <extends>QWidget</extends>
   <header>colorspacewidget.h</header>
   <container>1</container>
</customwidget>
```

Then, use function SetType (0: RGB, 1: HSV, 2: Lab), and updateImageMap to input the image matrix, function update to refresh the widget.

```
ui->colorspace->SetType(ui->comboBox->currentIndex());
ui->colorspace->updateImageMap(_image);
ui->colorspace->update();
```

3.3 3D visulization

Use the source files (Image3dWidget.h, Image3dWidget.cpp), initialise widget in ui frame.

```
...
<widget class="Image3dWidget" name="image3d" native="true"/>
...
<customwidget>
   <class>Image3dWidget</class>
   <extends>QWidget</extends>
   <header>image3dwidget.h</header>
   <container>1</container>
</customwidget>
```

Then, use function updateImageMap to input the image matrix and function update to refresh the widget.

```
ui->image3d->updateImageMap(_image);
ui->image3d->update();
```

4 Suggestion

4.1 Composition

We need to define the threshold value to find the focus part of the photo.
Currently, the solution is simple. We select a correct value for one photo, and leave it to every other photo, in most time it seems ok, but there are also many cases could be different result.
In my mind, There could be an idea to improve the threshold value definition.
1, Initialise the threshold value to max, there is no white pixel.
2, Decrease the threshold value by 1, and we get very few white points.
3, Draw the hull of these points, and get the focus area.
4, Return to step 2, if the points we get is far away from focus area, we stop the decrease.

4.2 Color Space

There is a problem in current solution, the code of OpenGL like,

```
glBegin(GL_POINTS);
for (int i=0; i<histSize[0]; i++) {
    for (int j=0; j<histSize[1]; j++) {
        for (int k=0; k<histSize[2]; k++) {
            if(hist.at<double>(k, i, j) > 0)
            {
                drawRGBPixel(j,i,k);
            }
       }
    }
}
glEnd();
```

Many loops between glBegin and glEnd will lead the graph slow. To improve this, we need to use vertex buffer to save all the points before rendering the graph.

4.3 3D Visualisation

After convert a image to 3D space, we can easily find the intensity differences between the pixels, but we need to find a solution to define these differences and catch the biggest differences area in the image, and verify whether it is focus area of the photo.

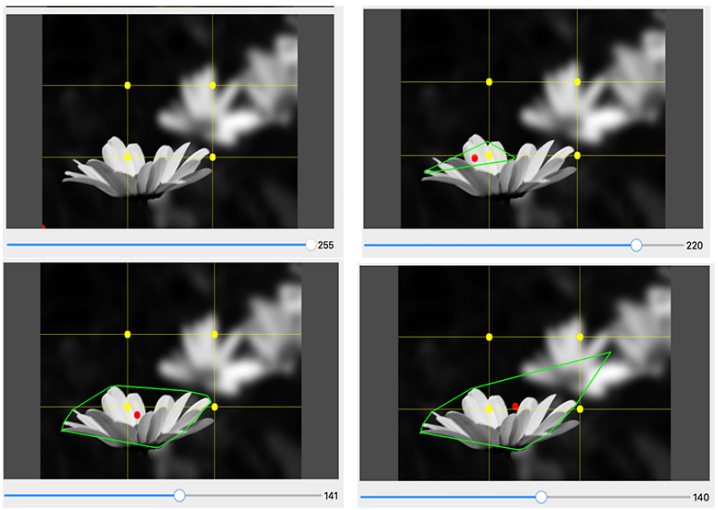

Figure 5: Process of deciding threshold value, 1: 255, 2: 220, 3: 141, 4: 140

References

[1] Y. N. Long Mai, Hoang Le and F. Liu, "Rule of Thirds Detection from Photograph," 2011.

[2] "Sobel Derivatives." http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/sobel_derivatives/sobel_derivatives.html.

[3] "Laplace Operator." http://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/laplace_operator/laplace_operator.html.

[4] A. S. Sebastian Montabone, "Human Detection Using a Mobile Platform and Novel Features Derived From a Visual Saliency Mechanism," 2009.

[5] P. B. A. Barsky, "Light, Color, Perception, and Color Space Theory." https://inst.eecs.berkeley.edu/~cs39j/sp02/handouts/color_slides.pdf.

[6] D. A. Kerr, "The HSV and HSL Color Models and the Infamous Hexcones."

[7] "Lab color space." https://en.wikipedia.org/wiki/Lab_color_space.

[8] "Contour Features." http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_imgproc/py_contours/py_contour_features/py_contour_features.html.

[9] B. Robot, "Converting from RGB to HSV." http://coecsl.ece.illinois.edu/ge423/spring05/group8/finalproject/hsv_writeup.pdf.