Image editing techniques and algorithms using Qt

This article explains how to edit images at the pixel level using Qt in order to implement a number of visual effects. The effects demonstrated include: grey scale, blur, sharpen, drawing to a frame or metal, and changing the image saturation, brightness, or warmth.

Note: This is an entry in the PureView Imaging Competition 2012Q2

Introduction

In this article we'll discuss different techniques and algorithms to modify an image using Qt, but before we continue you must know some principles to work with images.

- There are 2 main classes to represent an image in Qt: QImage and QPixmap; we can also use QBitmap for monochrome images like masks, and QPicture stores QPainter drawing commands.
  - `QPixmap` is the recommended class to use when we want to draw images on the screen because it's the quickest way to do it. The problem with `QPixmap` is that we cannot access to an individual pixel to read it and modify it.
  - `QImage` is faster than QPixmap in IO operations and gives us access to the individual pixel information. It's the class that we'll use in this article to edit images.
- If you are managing big images, like photos taken with the camera, it's recommended to work with a downsized image to show it on the screen as a preview image unless we want to allow users to zoom in the image. There are two ways to load an image from a file and downsize it:
  - Load the image into a `QImage` or `QPixmap` and resize it later:

```
QImage image("sample.png");
image = image.scaled(width, height);
```

  - Using QImageReader setting the target size before loading the image into a `QImage` - `QImageReader` can't load an image into a `QPixmap` but it's easy to create a `QPixmap` from a `QImage` with the static method `QPixmap::fromImage(QImage img)`. This method is faster and you won't need the memory required to load the full size image:

```
QImageReader imgReader("sample.png");
imgReader.setScaledSize(QSize(width, height));
QImage * image;
imgReader.read(image);
```

- Every image is made by pixels and every pixel is formed by 3 color channels: red, green and blue, and an alpha channel that contains the transparency value (JPG and other image formats don't support transparency). These channels have values between 0 and 255 and black color is formed when these 3 color channels are 0 while white is represented when the value of the 3 channels is 255. To represent a color in this article we'll refer to it as RGB(red, green, blue) being red, green and blue the value of these 3 channels.
- To simplify the code below we are reading pixel by pixel, but reading a full line with the method `QImage::scanLine(int i)` is more efficient. This is an example on how the code would look like reading pixels line by line with the first technique we'll describe: converting an image to grey scale.

```
QImage * MainWindow::greyScale(QImage * origin){
    QImage * newImage = new QImage(origin->width(), origin->height(),
QImage::Format_ARGB32);

    QRgb * line;

    for(int y = 0; y<newImage->height(); y++){
        QRgb * line = (QRgb *)origin->scanLine(y);
```


(C) Copyright Nokia 2014. All rights reserved.

```
        for(int x = 0; x<newImage->width(); x++){
            int average = (qRed(line[x]) + qGreen(line[x]) + qRed(line[x]))/3;
            newImage->setPixel(x,y, qRgb(average, average, average));
        }

    }

    return newImage;
}
```

Grey Scale

First technique we'll see is converting an image to grey scale. We must know that grey tones have similar if not the same value in the 3 color channels, so our strategy will be to modify every pixel and make the RGB channels have the same value. We'll determine this value with the arithmetic mean of the 3 channels so if the original color is RGB(169, 204, 69) the new color will be RGB(147, 147, 147) - *(169+204+69)/3 = 147* -.

```
QImage * MainWindow::greyScale(QImage * origin){
    QImage * newImage = new QImage(origin->width(), origin->height(),
QImage::Format_ARGB32);

    QColor oldColor;

    for(int x = 0; x<newImage->width(); x++){
        for(int y = 0; y<newImage->height(); y++){
            oldColor = QColor(origin->pixel(x,y));
            int average = (oldColor.red()+oldColor.green()+oldColor.blue())/3;
            newImage->setPixel(x,y,qRgb(average,average,average));
        }
    }

    return newImage;
}
```

Original Picture

(C) Copyright Nokia 2014. All rights reserved.

Grey Scale Picture

Brightness

As we said in the introduction of this article, white is represented as RGB(255, 255, 255) while black is RGB(0, 0, 0), so if we want to increase the brightness (the color will be closer to white) we need to increment the value of the 3 channels and decrease it to get a darker image.

In this case we'll use a delta parameter to determine how much we'll increase all the channels, so if delta is negative we'll get a dark image. After adding delta we only need to check the new values are between 0 and 255.

```
QImage * MainWindow::brightness(int delta, QImage * origin){
    QImage * newImage = new QImage(origin->width(), origin->height(),
QImage::Format_ARGB32);

    QColor oldColor;
    int r,g,b;

    for(int x=0; x<newImage->width(); x++){
        for(int y=0; y<newImage->height(); y++){
            oldColor = QColor(origin->pixel(x,y));

            r = oldColor.red() + delta;
            g = oldColor.green() + delta;
            b = oldColor.blue() + delta;

            //we check if the new values are between 0 and 255
            r = qBound(0, r, 255);
            g = qBound(0, g, 255);
            b = qBound(0, b, 255);

            newImage->setPixel(x,y, qRgb(r,g,b));
        }
    }

    return newImage;
}
```


(C) Copyright Nokia 2014. All rights reserved.

Original Picture

Brighter Picture with Delta = 30

Warm

When we talk about a warm image that's because colors tend to yellow. We don't have a yellow channel, but however yellow is the result of mixing red and green so to get a warm image we'll increase these two channels leaving blue channel with the same value.

We'll use a delta parameter to determine how much we'll increase red and green channels. With a warmer image we can get a retro effect and for example in images where there's sand we'll get a more vivid image.

```
QImage * MainWindow::warm(int delta, QImage * origin){
    QImage *newImage = new QImage(origin->width(), origin->height(),
QImage::Format_ARGB32);

    QColor oldColor;
    int r,g,b;

    for(int x=0; x<newImage->width(); x++){
        for(int y=0; y<newImage->height(); y++){
            oldColor = QColor(origin->pixel(x,y));

            r = oldColor.red() + delta;
            g = oldColor.green() + delta;
            b = oldColor.blue();

            //we check if the new values are between 0 and 255
            r = qBound(0, r, 255);
```


(C) Copyright Nokia 2014. All rights reserved.

```
            g = qBound(0, g, 255);

            newImage->setPixel(x,y, qRgb(r,g,b));
        }
    }

    return newImage;
}
```

Original Picture

Warm Picture with Delta = 30

Cool

If colors in a warm image tend to yellow, with cool images the most intense color is blue. In our method we'll increase the blue channel with a delta value leaving red and green channels unmodified.

Cool images can be related with future, death or simply cold.

```
QImage * MainWindow::cool(int delta, QImage * origin){
    QImage *newImage = new QImage(origin->width(), origin->height(),
QImage::Format_ARGB32);

    QColor oldColor;
    int r,g,b;

    for(int x=0; x<newImage->width(); x++){
        for(int y=0; y<newImage->height(); y++){
            oldColor = QColor(origin->pixel(x,y));
```


(C) Copyright Nokia 2014. All rights reserved.

```
            r = oldColor.red();
            g = oldColor.green();
            b = oldColor.blue()+delta;

            //we check if the new value is between 0 and 255
            b = qBound(0, b, 255);

            newImage->setPixel(x,y, qRgb(r,g,b));
        }
    }

    return newImage;
}
```

Original Picture

Cool Picture with Delta = 30

Saturation

We've said colors are formed by 3 channels: red, green and blue. Even if that's true, RGB isn't the only way to represent a color and in this case we'll use the HSL format - hue, saturation, lightness. We only need to increase saturation channel to get our saturated image.

Saturated images have more vivid colors and use to be more spectacular, but it's important not to abuse with the saturation or we'll lose the fidelity of the original image.

```
QImage * MainWindow::saturation(int delta, QImage * origin){
    QImage * newImage = new QImage(origin->width(), origin->height(),
```


(C) Copyright Nokia 2014. All rights reserved.

```
QImage::Format_ARGB32);

    QColor oldColor;
    QColor newColor;
    int h,s,l;

    for(int x=0; x<newImage->width(); x++){
        for(int y=0; y<newImage->height(); y++){
            oldColor = QColor(origin->pixel(x,y));

            newColor = oldColor.toHsl();
            h = newColor.hue();
            s = newColor.saturation()+delta;
            l = newColor.lightness();

            //we check if the new value is between 0 and 255
            s = qBound(0, s, 255);

            newColor.setHsl(h, s, l);

            newImage->setPixel(x, y, qRgb(newColor.red(), newColor.green(),
newColor.blue()));
        }
    }

    return newImage;
}
```

Original Picture

Saturated Picture with Delta = 30

(C) Copyright Nokia 2014. All rights reserved.

Blur

This is a bit more complex than what we have made until now. To blur an image we'll use a convolution filter that creates the new color of every pixel according to its original color and the color of the adjacent pixels. There's a matrix called kernel that determines how much influence will have each adjacent pixel in this calculation.

Our original pixel will be in the center of this matrix and for that reason we'll use matrices with an odd number of rows and columns. Note that we won't modify the pixels that are in the borders of the image because we don't have all the adjacent pixels we need, although we could modify the method to use only the available pixels.

Let's see an example on how we calculate the RGB value for a pixel. These are the matrices that represent the red, green and blue channels of our pixel and its adjacent pixels - our pixel is in the center of the matrix:

```
R =  20 102 99
     150 200 77
     170 210 105
```

```
G = 22 33 40
    17 21 33
     8 15 24
```

```
B = 88 70 55
    90 72 59
    85 69 50
```

And our kernel is:

```
Kernel = 0 2 0
         2 5 2
         0 2 0
```

And this is how we calculate our new value after applying the filter:

```
r = ( (102*2) + (150*2) + (200*5) + (77*2) + (210*2) ) / (2+2+5+2+2) = 159
```

```
g = ( (33*2) + ( 17*2) + (21*5) + (33*2) + (15*2) ) / (2+2+5+2+2) = 23
```

```
b = ( (70*2) + (90*2) + (72*5) + (59*2) + (69*2) ) / (2+2+5+2+2) = 72
```

So we went from **RGB(200, 21, 72)** to **RGB(159, 23, 72)**. You can notice that the biggest change was in the red channel where there's a big difference in the value of the adjacent pixels.

One of the cases when we'll want to use blur with an image is when we have portraits. Blur will help us to hide skin flaws.

```
QImage * MainWindow::blur(QImage * origin){
    QImage * newImage = new QImage(*origin);

    int kernel [5][5]= {{0,0,1,0,0},
                        {0,1,3,1,0},
                        {1,3,7,3,1},
```


(C) Copyright Nokia 2014. All rights reserved.

```
                        {0,1,3,1,0},
                        {0,0,1,0,0}};
    int kernelSize = 5;
    int sumKernel = 27;
    int r,g,b;
    QColor color;

    for(int x=kernelSize/2; x<newImage->width()-(kernelSize/2); x++){
        for(int y=kernelSize/2; y<newImage->height()-(kernelSize/2); y++){

            r = 0;
            g = 0;
            b = 0;

            for(int i = -kernelSize/2; i<= kernelSize/2; i++){
                for(int j = -kernelSize/2; j<= kernelSize/2; j++){
                    color = QColor(origin->pixel(x+i, y+j));
                    r += color.red()*kernel[kernelSize/2+i][kernelSize/2+j];
                    g += color.green()*kernel[kernelSize/2+i][kernelSize/2+j];
                    b += color.blue()*kernel[kernelSize/2+i][kernelSize/2+j];
                }
            }

            r = qBound(0, r/sumKernel, 255);
            g = qBound(0, g/sumKernel, 255);
            b = qBound(0, b/sumKernel, 255);

            newImage->setPixel(x,y, qRgb(r,g,b));

        }
    }
    return newImage;
}
```

Original Picture

(C) Copyright Nokia 2014. All rights reserved.

Blurred Picture

Sharpen

To sharpen an image we'll use again a convolution filter as the one we used to blur it, but with a different kernel. Now, the kernel will have negative values in the adjacent pixels.

Sharpening is useful with blurry images, it helps to improve the level of detail.

```
QImage * MainWindow::sharpen(QImage * origin){
    QImage * newImage = new QImage(* origin);

    int kernel [3][3]= {{0,-1,0},
                        {-1,5,-1},
                        {0,-1,0}};
    int kernelSize = 3;
    int sumKernel = 1;
    int r,g,b;
    QColor color;

    for(int x=kernelSize/2; x<newImage->width()-(kernelSize/2); x++){
        for(int y=kernelSize/2; y<newImage->height()-(kernelSize/2); y++){

            r = 0;
            g = 0;
            b = 0;

            for(int i = -kernelSize/2; i<= kernelSize/2; i++){
                for(int j = -kernelSize/2; j<= kernelSize/2; j++){
                    color = QColor(origin->pixel(x+i, y+j));
                    r += color.red()*kernel[kernelSize/2+i][kernelSize/2+j];
                    g += color.green()*kernel[kernelSize/2+i][kernelSize/2+j];
                    b += color.blue()*kernel[kernelSize/2+i][kernelSize/2+j];
                }
            }

            r = qBound(0, r/sumKernel, 255);
            g = qBound(0, g/sumKernel, 255);
            b = qBound(0, b/sumKernel, 255);

            newImage->setPixel(x,y, qRgb(r,g,b));

        }
```


(C) Copyright Nokia 2014. All rights reserved.

```
    }
    return newImage;
}
```

Original Picture

Sharpened Picture

Drawing a Frame

Drawing a frame is quite simple, we only need to draw the frame over our original image. In our method we suppose we have a picture with the same size of our frame, but we could need to resize the frame or even use 4 images for the 4 borders to keep the thickness of the frame when we resize it.

```
QImage * MainWindow::drawFrame(QImage * origin){
    QImage * newImage = new QImage(* origin);
    QPainter painter;

    painter.begin(newImage);

    painter.drawImage(0,0, QImage(":images/frame.png"));

    painter.end();

    return newImage;
}
```


(C) Copyright Nokia 2014. All rights reserved.

Original Picture

Frame

Framed Picture

Drawing on Metal

This is a sample on how we can combine several techniques to get an effect. These are the steps we follow to get this effect:

1. Adjust the brightness of the image to get a darker picture.
2. Convert the result to grey scale.
3. We draw the grey picture over a metal texture applying an opacity of 50%

```
QImage * MainWindow::metal(QImage * origin){
    QImage * newImage = new QImage(":images/metal.png");
    QImage * darkImage = brightness(-100, origin);
    QImage * greyImage = greyScale(darkImage);
```


(C) Copyright Nokia 2014. All rights reserved.

```
    QPainter painter;

    painter.begin(newImage);

    painter.setOpacity(0.5);
    painter.drawImage(0, 0, * greyImage);

    painter.end();

    delete greyImage;
    delete darkImage;

    return newImage;
}
```

Original Picture

Metal Texture

(C) Copyright Nokia 2014. All rights reserved.

Final Picture

Blurred Frame

To finalize, we'll see another combined effect. This time we want to blur the exterior part of the picture so the focus is in the center.

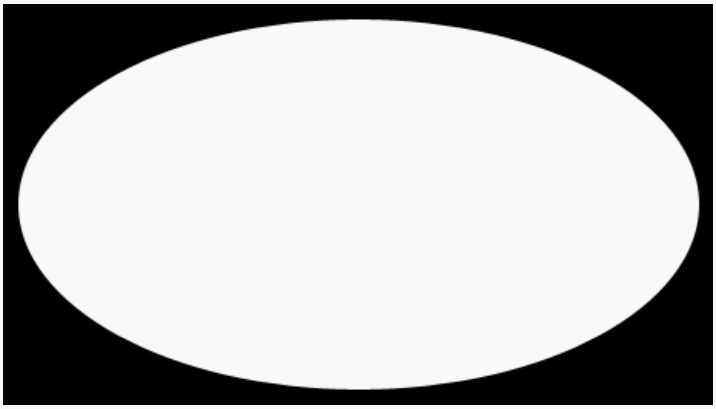
We'll use an image that serves as mask to determine what part of the image will be blurred. These are the steps we take:

1. We get a full blurred image from our original picture.
2. Using one of QPainter's composition modes we get a blurred frame with the shape of our mask. You can learn more about QPainter's composition modes in QPainter documentation.
3. We draw our blurred frame over the original image.

```
QImage * MainWindow::blurFrame(QImage * origin){
    QImage * newImage = new QImage(* origin);
    QImage * blurredImage = blur(newImage);
    QImage * mask = new QImage(":images/mask.png");
    QPainter painter;

    //Using the composition mode SourceAtop we get a blurred frame stored in QImage mask
    painter.begin(mask);

    painter.setCompositionMode(QPainter::CompositionMode_SourceAtop);
    painter.drawImage(0, 0, * blurredImage);

    painter.end();

    //With our new frame we simply draw it over the original image
    painter.begin(newImage);

    painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
    painter.drawImage(0, 0, * mask);

    painter.end();

    delete mask;
    delete blurredImage;

    return newImage;
}
```


(C) Copyright Nokia 2014. All rights reserved.

Original Picture



Mask

Blurred Frame

Final Picture

(C) Copyright Nokia 2014. All rights reserved.

Sample App

You can download the source code of a sample app with all the methods we have seen in this article. In this app it's included 3 sample images with the same size, 462x260, to test the different methods. You simply select one of these pictures and touch one of the buttons to apply the desired effect.

Download: File:Image Editor.zip

Image editor screenshot

Summary

This article should serve you as an entry point to image editing, but possibilities are endless. You can modify these methods, combine them, use other techniques... imagination is your only limit.

(C) Copyright Nokia 2014. All rights reserved.