用策略设计模式比较颜色

- 构建一个简单的算法，用来识别图像中具有某种颜色的所有像素。

  输入：一幅图像和一个颜色，还要指定一个能接受的颜色的公差。

  返回：一个二值图像，显示具有指定颜色的像素。

- **策略模式**：定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。

使用类

```
#if !defined COLORDETECT
#define COLORDETECT

#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>
#include "opencv2/imgproc/types_c.h"

class ColorDetector
{
private:

    // 允许要探测的颜色与目标颜色的最大差距
    int maxDist;

    // 目标颜色（BGR）
    cv::Vec3b target;

    // 存储Lab色彩空间下的输入图像
    cv::Mat converted;
    bool useLab; // 是否使用Lab色彩空间

    // 存储结果的图像(二值图像)
    cv::Mat result;

public:

#pragma region 构造函数

    // 空构造函数，在此初始化默认参数
    ColorDetector() : maxDist(100), target(0, 0, 0), useLab(false) {}

    // 额外的构造函数，使用Lab色彩空间
    ColorDetector(bool useLab) : maxDist(100), target(0, 0, 0), useLab(useLab)
{}

    // 完整的构造函数
    ColorDetector(uchar blue, uchar green, uchar red, int mxDist = 100, bool
useLab = false) : maxDist(mxDist), useLab(useLab)
    {
        // 设置目标颜色
```

```
        setTargetColor(blue, green, red);
    }

#pragma endregion

    // 图像处理函数，返回一个单通道二值图像
    cv::Mat process(const cv::Mat& image);

    // 计算与目标颜色差距
    int getDistanceToTargetColor(const cv::Vec3b& color) const
    {
        return getColorDistance(color, target);
    }

    // 计算两个颜色之间的城区距离（RGB值差距的绝对值）
    // cv::Vec3b存储三个无符号字符型，即颜色的RGB值
    // color1: 要检测像素的颜色值
    // color2: 目标颜色值
    int getColorDistance(const cv::Vec3b& color1, const cv::Vec3b& color2) const
    {
        // |ΔB|+|ΔG|+|ΔR|
        return abs(color1[0] - color2[0]) +
            abs(color1[1] - color2[1]) +
            abs(color1[2] - color2[2]);
    }

#pragma region 使用CIE L*a*b色彩空间和仿函数
    // 重载运算符（），第二种计算颜色向量间的距离的方法
    cv::Mat operator()(const cv::Mat& image)
    {
        cv::Mat input;

        // 如果需要转换成Lab色彩空间
        if (useLab)
        {
            cv::cvtColor(image, input, CV_BGR2Lab);
        }
        else
        {
            input = image;
        }

        cv::Mat output;
        // 计算输入图像与目标颜色的距离的绝对值
        cv::absdiff(input, cv::Scalar(target), output);
        // 得到的是：每个通道与目标颜色对应通道的差值（|ΔB|，|ΔG|，|ΔR|）

        // 把输出结果图像的通道拆分成3幅单通道图像，分别对应B,G,R
        std::vector<cv::Mat> images;
        cv::split(output, images);

        // 三个通道的差值相加（这里可能出现饱和的情况）
        output = images[0] + images[1] + images[2]; // |ΔB|+|ΔG|+|ΔR|

        // 对图像进行阈值化，成为二值图像
        cv::threshold(
            output,  // 输入图像
            output,  // 输出图像
```

```
            maxDist, // 要探测的颜色与目标颜色的最大差距
            255,     // 要填充的最大值
            cv::THRESH_BINARY_INV); // 阈值化模式
        //                | 0       if output(x, y) > maxDist   大于允许颜色差距的像素都赋值为0（黑色）
        // output(x, y) = {
        //                | 255     otherwise                   小于允许颜色差距的像素都赋值为255（白色）

        // 返回阈值化后的二值图像
        return output;
    }

#pragma endregion

#pragma region Getters and setters

    // Sets the color distance threshold.
    // Threshold must be positive, otherwise distance threshold
    // is set to 0.
    void setColorDistanceThreshold(int distance) {

        if (distance < 0)
            distance = 0;
        maxDist = distance;
    }

    // Gets the color distance threshold
    int getColorDistanceThreshold() const {

        return maxDist;
    }

    // Sets the color to be detected
    // given in BGR color space
    void setTargetColor(uchar blue, uchar green, uchar red) {

        // BGR order
        target = cv::Vec3b(blue, green, red);

        if (useLab) {
            // Temporary 1-pixel image
            cv::Mat tmp(1, 1, CV_8UC3);
            tmp.at<cv::Vec3b>(0, 0) = cv::Vec3b(blue, green, red);

            // Converting the target to Lab color space
            cv::cvtColor(tmp, tmp, CV_BGR2Lab);

            target = tmp.at<cv::Vec3b>(0, 0);
        }
    }

    // Sets the color to be detected
    void setTargetColor(cv::Vec3b color)
    {

        target = color;
    }
```

```
    // Gets the color to be detected
    cv::Vec3b getTargetColor() const {

        return target;
    }
};
#endif
```

```
/// <探测出图像中与目标颜色相近的颜色>

    // 1.创建图像处理器对象
    ColorDetector cdetect;  // 类是自己写的

    // 2.读取输入的图像
    cv::Mat image = cv::imread("boldt.jpg");
    if (image.empty())
        return 0;
    cv::namedWindow("Original Image");
    cv::imshow("Original Image", image);

    // 3.设置输入参数
    cdetect.setTargetColor(230, 190, 130); // 这里颜色 (B = 230, G = 190, R =
130) 表示蓝天的颜色

    // 4.处理图像
    cv::Mat result = cdetect.process(image);  // 类中自己写的函数

    // 5.显示结果
    cv::namedWindow("result");
    cv::imshow("result", result);
    cv::waitKey();
```

转换颜色表示法

- RGB 色彩空间的基础是对加色法三原色（红、绿、蓝）的应用
- 选用这三种颜色作为三原色，是因为将它们组合后可以产生色域很宽的各种颜色，与人类视觉系统对应。这通常是 **数字成像中默认的色彩空间**
- 红绿蓝三个通道还要做归一化处理，当三种颜色强度相同时就会取得灰度，即从黑色(0, 0, 0)到白色(255, 255, 255)
- RGB 色彩空间的缺点：

  不是感知均匀的色彩空间。两种具有一定差距的颜色可能看起来非常接近，而另外两种具有同样差距的颜色看起来却差别很大 。

  为解决这个问题，引入了一些具有感知均匀特性的颜色表示法
- 将图像从一个色彩空间转换到另一个色彩空间时：

  1.每个输入像素上做一个线性或非线性的转换，以得到输出像素。

  2.输出图像的像素类型与输入图像是一致的。
- 色彩空间

YCrCb	CV_BGR2YCrCb	JPEG 压缩中使用的色彩空间
CIE L*a*b*	CV_BGR2Lab	使用同样的转换公式，但对色度通道则使用不同的表示法。
CIE L*u*v*	CV_BGR2Luv	
CIE XYZ	CV_BGR2XYZ	用与设备无关的方式表示任何可见颜色
HSV和HLS		人们用这种方式来描述的颜色会更加自然

CIE L*a*b*色彩空间** - CV_BGR2Lab

- **L 通道**：表示每个像素的亮度，范围是0~100；在使用8 位图像时，它的范围就会调整为0~255。
- **a通道和b通道**：表示色度组件，这些通道包含了像素的颜色信息。它们的值的范围是-127~127；对于8 位图像，每个值会加上128，将范围调整为0~255。
- 注意，进行8位颜色转换时会产生舍入误差，因此转换过程并不是完全可逆的。

使用迭代器

```
// 使用CIE L*a*b色彩空间来进行颜色的距离计算
    ColorDetector colordetector(
        230, 190, 130,  // 要探测的目标颜色
        45,             // 允许的最大差距
        true);          // 使用CIE L*a*b色彩空间
```

```
cv::Mat ColorDetector::process(const cv::Mat& image)
{
    // 创建输出图像，与输入图像的尺寸相同，不过是单通道灰度图像（二值图像）
    result.create(image.size(), CV_8U);

    // 创建遍历输入图像的迭代器（没有初始化，因为后边还得判断用那个色彩空间）
    cv::Mat_<cv::Vec3b>::const_iterator it;
    cv::Mat_<cv::Vec3b>::const_iterator itend;

    // 输出图像的迭代器
    cv::Mat_<uchar>::iterator itout = result.begin<uchar>();

#pragma region 转换色彩空间

    // 判断是否需要用CLE L*a*b色彩空间
    if (useLab)
        // 将输入图像（默认RGB色彩空间）转换为CLE L*a*b色彩空间（简称Lab色彩空间）
        cv::cvtColor(image, converted, CV_BGR2Lab);
#pragma endregion

    // get the iterators
    cv::Mat_<cv::Vec3b>::const_iterator it = image.begin<cv::Vec3b>();
    cv::Mat_<cv::Vec3b>::const_iterator itend = image.end<cv::Vec3b>();
    cv::Mat_<uchar>::iterator itout = result.begin<uchar>();

    // 初始化输入图像的迭代器
    if (useLab)  // 如果使用Lab色彩空间
    {
        it = converted.begin<cv::Vec3b>();
        itend = converted.end<cv::Vec3b>();
    }
    else
```

```
    {
        it = image.begin<cv::Vec3b>();
        itend = image.end<cv::Vec3b>();
    }

    // 遍历输入图像
    for (; it != itend; ++it, ++itout)
    {
        /// <比较输入图像中每个像素的颜色与目标颜色之间的差距>

          // 如果输入图像中当前像素的颜色与目标颜色的差距<可允许的最大差距
        if (getDistanceToTargetColor(*it) < maxDist) // getDistanceToTargetColor
方法来计算与目标颜色的差距
        {
            // 输出图像中当前像素的颜色赋值为255（白色）
            *itout = 255; // 表示该像素的颜色与目标颜色接近
        }
        else
        {
            // 输出图像中当前像素的颜色赋值为255（白色）
            *itout = 0;  // 表示该像素的颜色与目标颜色差距较大
        }
    }

    // 返回输出图像（二值图像）
    return result;
}
```

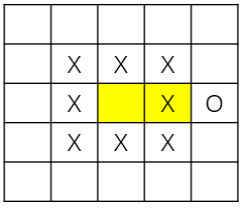
使用仿函数

```
//处理图像并显示二值结果
result = colordetector(image);  // 调用仿函数
```

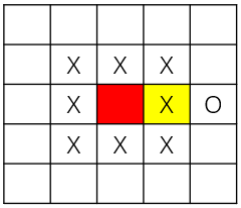
floodFill函数

漫水填充法是一种用特定的颜色填充联通区域，通过设置可连通像素的上下限以及连通方式来达到不同的填充效果的方法。漫水填充经常被用来标记或分离图像的一部分以便对其进行进一步处理或分析，也可以用来从输入图像获取掩码区域，掩码会加速处理过程，或只处理掩码指定的像素点，操作的结果总是某个连续的区域。

- **浮动范围**：就是先用种子点通过上下阈值判断出其邻域内的一个点，再用该判断出的点作为新种子，去继续判断新种子邻域内的点。



- **固定范围**：就是将与种子点相连通区域内的点都与种子点进行判断。



```
// 测试 floodFill 函数
    cv::floodFill(
        image,                          // 输入/输出图像
        cv::Point(100, 50),             // 起始点/种子点（天空的位置，起始点的颜色为目标颜色）
        cv::Scalar(255, 255, 255),  // 填充颜色（白色，与目标颜色相近的颜色都被填充为白色）
        (cv::Rect*)0,                   // 填充区域的边界矩形（这里设置为无边界）
        cv::Scalar(35, 35, 35),     // 当前选定像素与其连通区中相邻像素中的一个像素，或者与加入该连通区的一个种子像素，二者之间的最大下行差异值
    cv::Scalar(35, 35, 35),     // 当前选定像素与其连通区中相邻像素中的一个像素，或者与加入该连通区的一个种子像素，二者之间的最大上行差异值
    cv::FLOODFILL_FIXED_RANGE); // 固定范围模式，即所有像素都与起始点像素比较
```

用GrabCut算法分割图像

- 如果要从静态图像中提取前景物体（例如从图像中剪切一个物体，并粘贴到另一幅图像），最好采用GrabCut 算法。
- cv::grabCut 函数的用法：只需要输入一幅图像，并对一些像素做上“属于背景”或“属于前景”的标记即可。根据这个局部标记，算法将计算出整幅图像的前景/背景分割线。

```
int main()
{
    /// <GrabCut分割算法>

    // 读入输入图像
    cv::Mat image = cv::imread("boldt.jpg");
    if (!image.data)
        return 0;

    // 显示图像
    cv::namedWindow("Original Image");
    cv::imshow("Original Image", image);

    // 定义一个带边框的矩形，矩形外部的像素会被标记为背景
    cv::Rect rectangle(50, 25, 210, 180);

    // 分割结果（四种可能的值）
    cv::Mat result; // segmentation (4 possible values)

    // 模型（内部使用）
    cv::Mat bgModel, fgModel;


    // GrabCut 分割算法
    cv::grabCut(
        image,                  // 输入图像
        result,                 // 分割结果，具有四种可能的值
        rectangle,              // 包含前景的矩形
        bgModel, fgModel,       // 模型
        5,                      // 迭代次数
        cv::GC_INIT_WITH_RECT); // 使用带边框的矩形模型

    /*  分割结果图像result，具有四种可能的值
        cv::GC_BGD:     该值表示明确属于背景的像素（本例中矩形之外的像素）
```

```
        cv::GC_FGD:      该值表示明确属于前景的像素（本例无）
        cv::GC_PR_BGD:   该值表示可能属于背景的像素
        cv::GC_PR_FGD:   该值表示可能属于前景的像素（本例中矩形之内像素的初始值）
    */

    // 取得标记为“可能属于前景”的像素
    /*
        if result(x, y) == cv::GC_PR_FGD(可能属于前景的像素)
            result(x, y) == 255
        else
            result(x, y) == 0
    */
    cv::compare(result, cv::GC_PR_FGD, result, cv::CMP_EQ); // result既是输入又是输出，第二个result成为二值图像

#pragma region Test

    cv::namedWindow("Test");
    cv::imshow("Test", result); // 掩码图像

#pragma endregion

    // 创建输出图像
    cv::Mat foreground(image.size(), CV_8UC3, cv::Scalar(255, 255, 255));

    // 只有前景部分被复制（result掩码图像中像素值不为0的部分），不复制背景像素
    image.copyTo(foreground, result); // result相当于掩码（其中值为255的是前景，值为0的是背景）

    // 显示画了矩形的输入图像
    cv::rectangle(image, rectangle, cv::Scalar(255, 255, 255), 1);
    cv::namedWindow("Image with rectangle");
    cv::imshow("Image with rectangle", image);

    // 显示分割结果
    cv::namedWindow("Foreground object");
    cv::imshow("Foreground object", foreground);

    cv::waitKey();

    return 0;
}
```

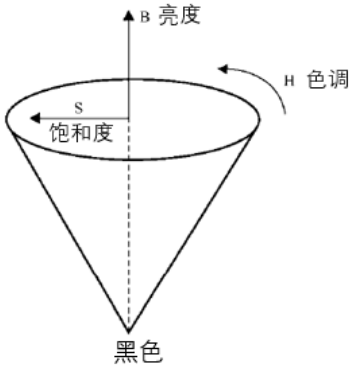
用色调、饱和度和亮度表示颜色

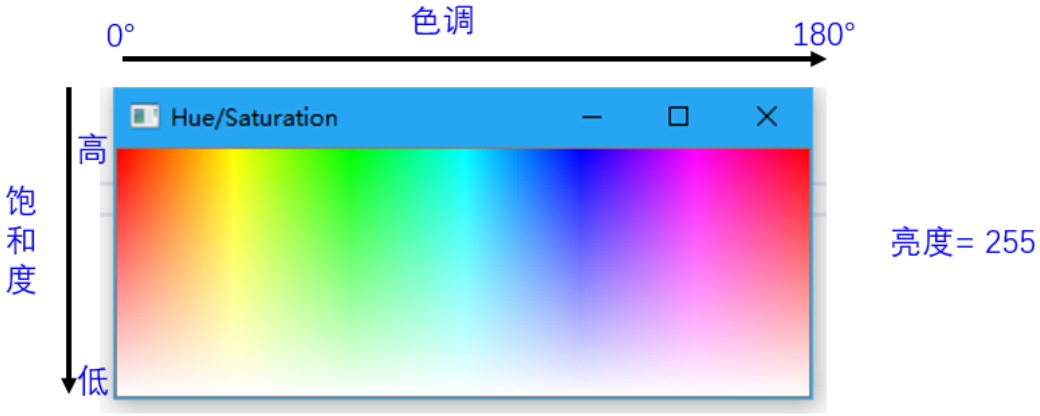
RGB 是一种被广泛接受的色彩空间。虽然它被视为一种在电子成像系统中采集和显示颜色的有效方法，但它其实并不直观，也并不符合人类对于颜色的感知方式。之所以要引入色调/饱和度/亮度的色彩空间概念，是因为人们喜欢凭直觉分辨各种颜色，而它与这种方式吻合。实际上，人类更喜欢用色彩、彩度、亮度等直观的属性来描述颜色，而大多数直觉色彩空间正是基于这三个属性。

- 色调（hue）表示主色，我们使用的颜色名称（例如绿色、黄色和红色）就对应了不同的色调值
- 饱和度（saturation）表示颜色的鲜艳程度，柔和的颜色饱和度较低，而彩虹的颜色饱和度就很高
- 亮度（brightness）是一个主观的属性，表示某种颜色的光亮程度

HSV/HSB色彩空间

- 色调通常用0~360 的角度来表示，其中红色是0 度
- 中轴线的距离表示饱和度，是一个0~1的值
- 高度表示亮度
- 圆锥体的顶点表示黑色，它的色调和饱和度是没有意义的






```
void detectHScolor(
    const cv::Mat& image,           // 输入图像
    double minHue, double maxHue,   // 色调区间
    double minSat, double maxSat,   // 饱和度区间
    cv::Mat& mask) {                // 输出掩码（二值图像）

    // 从RGB色彩空间转换成HSV色彩空间
    cv::Mat hsv;
    cv::cvtColor(image, hsv, CV_BGR2HSV);

    // 将HSV色彩空间的三个通道分割进三个图像中
    std::vector<cv::Mat> channels;
    cv::split(hsv, channels);

    // channels[0] 是色调H
    // channels[1] 是饱和度S
    // channels[2] 是亮度V

    /// <筛选出色调 minHue < Hue < maxHue 的像素>

    // 筛选出色调 < minHue的像素
    cv::Mat mask1;
    cv::threshold(channels[0], mask1, maxHue, 255, cv::THRESH_BINARY_INV);
    // 筛选出色调 > minHue的像素
```

```
    cv::Mat mask2;
    cv::threshold(channels[0], mask2, minHue, 255, cv::THRESH_BINARY);
    // 合并色调掩码
    cv::Mat hueMask;
    if (minHue < maxHue)
        hueMask = mask1 & mask2;
    /*
        255 & 255 = 255
        255 &  0  = 0
        0   &  0  = 0
    */
    else // 如果区间穿越0度中轴线
        hueMask = mask1 | mask2;
    /*
        255 | 255 = 255
        255 |  0  = 255
        0   |  0  = 0
    */

    // 筛选出饱和度 < maxSat的像素
    cv::threshold(channels[1], mask1, maxSat, 255, cv::THRESH_BINARY_INV);
    // 筛选出饱和度 > maxSat的像素
    cv::threshold(channels[1], mask2, minSat, 255, cv::THRESH_BINARY);
    // 合并饱和度掩码
    cv::Mat satMask; // saturation mask
    satMask = mask1 & mask2;

    // 组合掩码（二值图像）
    mask = hueMask & satMask;
}

int main()
{
    // 读入原始图像（RGB色彩空间）
    cv::Mat image = cv::imread("boldt.jpg");
    if (!image.data)
        return 0;

    // 显示原始图像
    cv::namedWindow("Original image");
    cv::imshow("Original image", image);

#pragma region 用色调，饱和度和亮度表示颜色

    // 从RGB色彩空间转换成HSV色彩空间
    cv::Mat hsv;
    cv::cvtColor(image, hsv, CV_BGR2HSV);

    // 把三个通道分割进3幅图像中
    std::vector<cv::Mat> channels;
    cv::split(hsv, channels);
    // channels[0] 是色调H
    // channels[1] 是饱和度S
    // channels[2] 是亮度V

    // 显示色调通道图
    cv::namedWindow("Hue");
    cv::imshow("Hue", channels[0]);
```

```
    // 显示饱和度通道图
    cv::namedWindow("Saturation");
    cv::imshow("Saturation", channels[1]);

    // 显示亮度通道图
    cv::namedWindow("Value");
    cv::imshow("Value", channels[2]);

    cv::waitKey();
    cv::destroyWindow("Hue");
    cv::destroyWindow("Saturation");
    cv::destroyWindow("Value");

#pragma endregion

#pragma region HSV颜色测试

    // 创建一个原图像的亮度通道的副本
    cv::Mat tmp(channels[2].clone()); // 保存源图像的亮度通道，因为后边要修改，但也原来的值也要用

    /// <固定亮度的图像>

    // 修改原图像亮度通道内所有像素的值为255
    channels[2] = 255;
    // 重新合并通道
    cv::merge(channels, hsv);
    // 转换回BGR色彩空间
    cv::Mat newImage;
    cv::cvtColor(hsv, newImage, CV_HSV2BGR);
    // 显示结果
    cv::namedWindow("Fixed Value Image");
    cv::imshow("Fixed Value Image", newImage);

    /// <固定饱和度的图像>

    // 还原亮度通道
    channels[2] = tmp;
    // 修改原图像饱和度通道内所有像素的值为255
    channels[1] = 255;
    // 重新合并通道
    cv::merge(channels, hsv);
    // 转换回BGR色彩空间
    cv::cvtColor(hsv, newImage, CV_HSV2BGR);
    // 显示结果
    cv::namedWindow("Fixed saturation");
    cv::imshow("Fixed saturation", newImage);

    /// <固定亮度和饱和度的图像>
    // 修改原图像饱和度通道内所有像素的值为255
    channels[1] = 255;
    // 修改原图像亮度通道内所有像素的值为255
    channels[2] = 255;
    // 重新合并通道
    cv::merge(channels, hsv);
    // 转换回BGR色彩空间
    cv::cvtColor(hsv, newImage, CV_HSV2BGR);
```

```
    // 显示结果
    cv::namedWindow("Fixed saturation/value");
    cv::imshow("Fixed saturation/value", newImage);

    cv::waitKey();
    cv::destroyAllWindows();

#pragma endregion

#pragma region 色调/饱和度组合

    // 人为生成一幅图像，用来说明各种色调/饱和度组合
    cv::Mat hs(128, 360, CV_8UC3);

    for (int h = 0; h < 360; h++)            // 列对应色调
    {
        for (int s = 0; s < 128; s++)        // 行对应饱和度
        {
            hs.at<cv::Vec3b>(s, h)[0] = h / 2;          // 色调通道：所有色调角度
            hs.at<cv::Vec3b>(s, h)[1] = 255 - s * 2;    // 饱和度通道：饱和度从高到
低
            hs.at<cv::Vec3b>(s, h)[2] = 255;            // 亮度通道：常数
        }
    }

    // 从HSV色彩空间转换成RGB色彩空间
    cv::cvtColor(hs, newImage, CV_HSV2BGR);

    // 显示图像
    cv::namedWindow("Hue/Saturation");
    cv::imshow("Hue/Saturation", newImage);
    cv::waitKey();
    cv::destroyWindow("Hue/Saturation");

#pragma endregion

#pragma region 肤色检测

    // 读入原始图像
    image = cv::imread("girl.jpg");
    if (!image.data)
        return 0;

    // 显示原始图像
    cv::namedWindow("Original image");
    cv::imshow("Original image", image);

    /// <检测肤色>
    cv::Mat mask;

    // 调用我们自己定义的肤色检测函数
    detectHScolor(
        image,      // 输入图像
        160, 10,    // 色调为320°~20°
        25, 166,    // 饱和度为0.1~0.65
        mask);      // 输入的二值掩码图像

    /*
```

```
        8位图像：
        色调范围：0~180°，因此OpenCV会把角度除以2，以适合单字节的存储范围
        饱和度范围：0~255°
    */

    /// <显示使用掩码后的图像>

    // 创建一个黑色3通道彩色图像
    cv::Mat detected(image.size(), CV_8UC3, cv::Scalar(0, 0, 0));
    // mask是检测到的肤色掩码，这里只复制原始图像中掩码不为0的部分
    image.copyTo(detected, mask);

    // 显示结果
    cv::imshow("Detection result", detected);
    cv::waitKey();

#pragma endregion

    return 0;
}
```