浙江大学
ZHEJIANG UNIVERSITY

数字信号处理综合实验
实验报告

实验名称 基于 C++ 的人脸分割的设计

实验地点 玉泉高分子楼-201

姓 名 _____

学 号 _____

实验日期 2025.12.07

指导老师 屈民军/唐奕

Contents

1 实验目的

(1). 了解数字图像处理技术的应用场合、发展现状和发展前景。

(2). 掌握图像采集、处理和显示过程，搞清数据存储、处理及传输过程。

(3). 了解图像处理原理和算法，掌握人脸分割的算法。

(4). 在给定的顶层设计框架下，完成人脸分割系统的设计与实现。

2 使用软件及其版本

Visual Studio 2022

3 实验背景原理

此处主要介绍有关 YUV 色彩空间的相关知识。

要了解 YUV 色彩空间，首先要了解 RGB 色彩空间。RGB 色彩空间是由红（Red）、绿（Green）、蓝（Blue）三种颜色光的不同强度按不同比例混合而成的颜色空间，是一种加色法颜色空间。RGB 色彩空间广泛应用于显示设备如计算机显示器、电视机和投影仪等。

YUV 色彩空间是一种基于亮度和色度分离的颜色空间。Y 表示亮度分量，U 和 V 表示色度分量。YUV 色彩空间的设计初衷是为了更有效地传输和存储彩色图像和视频信号，同时兼容黑白电视信号。YUV 色彩空间的优点在于它将亮度信息与色度信息分离，使得在传输和存储过程中可以对亮度和色度进行不同程度的压缩，从而节省带宽和存储空间。此外，由于人眼对亮度变化更敏感，对色度变化不太敏感，因此可以对色度分量进行更高程度的压缩，而不会显著影响视觉质量。

本实验选用 YUV422 格式进行图像处理。每个色素信道的采样率为亮度信道采样率的一半，即每两个像素共享一组色度分量 U 和 V。YUV422 又分为平面和打包两种存储方式。平面格式是将所有的 Y 分量存储在一起，接着存储所有的 U 分量，最后存储所有的 V 分量。而打包格式则是将 Y、U、V 分量交错存储，每两个像素对应一组 YUV 数据。二者分别适用图像处理和视频传输场景。

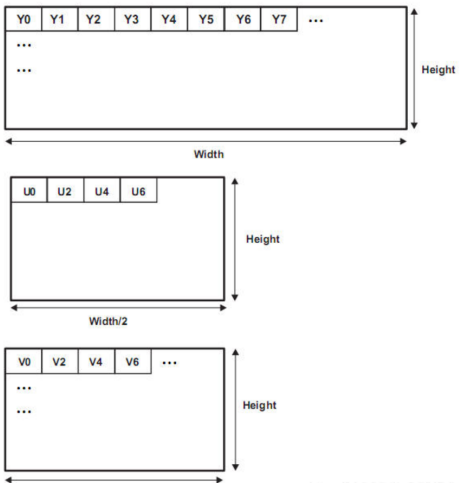

Figure 1: YUV422 平面格式

Figure 2: YUV422 打包格式

4 实验步骤

4.1 系统总体设计

项目解决方案整体由主项目和五个插件项目组成，主项目完成界面设计、插件搜索激活等功能，用来生成可执行文件；插件项目则用来生成动态链接库.DLL 文件，用来实现功能。本实验主要涉及前两个插件的设计与实现，分别为“摄像头视频流图像截取、重采样等处理”和“人脸检测与定位”插件。

本实验要求摄像头输出图像的格式为 YUV422 打包（packed) 或 RGB24 格式。然后将采集到的视频转换 YUV422 平面格式，用于图像处理。

重采样的目的是减少运算量，提高图像处理速度。重采样将产生 1/2W*1/4H 降采样彩色图片 clrBmp_1d8 及 1/4W*1/4H 降采样灰度图片 grayBmp_1d16。

人脸检测和定位将在 1/2W*1/4H 降采样彩色图片 clrBmp_1d8 中进行，检测出脸部区域（矩形），并存放在结构体 rcnFace 中。

4.2 摄像头视频流图像截取、重采样等处理

本插件主要作用是初始化 pImgProcBuf 向量，并为后面插件做一些准备工作，具体如下：

(1). 信号初始化

当输入为第一帧图象时（bNotInited=TRUE）时，初始化一系列信号，由于本实验未涉及眨眼检测等内容，因此有关眼鼻跟踪体的信号初始化均省略。其中包含动态内存相关初始化，见下：

Listing 1: 信号初始化

```
1  if (((BUF_STRUCT*)pBuffer)->bNotInited) {
2          ((BUF_STRUCT*)pBuffer)->colorBmp = pBuffer + sizeof(
               BUF_STRUCT);
3          ((BUF_STRUCT*)pBuffer)->grayBmp = ((BUF_STRUCT*)pBuffer)->
               colorBmp;
4          ((BUF_STRUCT*)pBuffer)->clrBmp_1d8 = ((BUF_STRUCT*)pBuffer)->
               grayBmp + w * h * 2;
5          ((BUF_STRUCT*)pBuffer)->grayBmp_1d16 = ((BUF_STRUCT*)pBuffer)
               ->clrBmp_1d8 + w * h / 4;
6          ((BUF_STRUCT*)pBuffer)->TempImage1d8 = ((BUF_STRUCT*)pBuffer)
               ->grayBmp_1d16 + w * h / 16;
7          ((BUF_STRUCT*)pBuffer)->lastImageQueue1d16m8 = ((BUF_STRUCT*)
               pBuffer)->TempImage1d8 + w * h / 8;
8          ((BUF_STRUCT*)pBuffer)->pOtherVars = (OTHER_VARS*)(((
               BUF_STRUCT*)pBuffer)->lastImageQueue1d16m8 + w * h / 2);
9          ((BUF_STRUCT*)pBuffer)->pOtherData = (aBYTE*)((BUF_STRUCT*)
               pBuffer)->pOtherVars + sizeof(OTHER_VARS);
10         for (int i = 0; i <= 7; i++)
11                 ((BUF_STRUCT*)pBuffer)->pImageQueue[i] = ((BUF_STRUCT
                       *)pBuffer)->lastImageQueue1d16m8 + i * (w * h / 16)
                       ;
12
13         ((BUF_STRUCT*)pBuffer)->W = w;
14         ((BUF_STRUCT*)pBuffer)->H = h;
15         ((BUF_STRUCT*)pBuffer)->cur_allocSize = 0;
16         ((BUF_STRUCT*)pBuffer)->allocTimes = 0;
17         ((BUF_STRUCT*)pBuffer)->cur_maxallocsize = 0;
18
19         //省略部分前两个插件中用不到的信号初始化
20
21         ((BUF_STRUCT*)pBuffer)->max_allocSize = w * h * 17 / 16 -
               sizeof(BUF_STRUCT) - sizeof(OTHER_VARS);
22         //动态内存大小计算
23         myHeapAllocInit((BUF_STRUCT*)pBuffer);
24         ((BUF_STRUCT*)pBuffer)->bNotInited = false;
25 }
```

(2). 设置显示图片指针

Listing 2: 图片指针

```
((BUF_STRUCT*)pBuffer)->displayImage = pYBits;
```

(3). 复制彩色图片 colorBmp

Listing 3: 复制彩色图片

```
memcpy(((BUF_STRUCT*)pBuffer)->colorBmp, pYBits, w * h * 2);
```

(4). 重采样产生彩色图片与灰度图片

Listing 4: 重采样

```
ReSample(((BUF_STRUCT*)pBuffer)->colorBmp, w, h, w / 2, h / 4, false
    , false, ((BUF_STRUCT*)pBuffer)->clrBmp_1d8);
ReSample(((BUF_STRUCT*)pBuffer)->grayBmp, w, h, w / 4, h / 4, false,
    true, ((BUF_STRUCT*)pBuffer)->grayBmp_1d16);//重采样处理
```

4.3 人脸检测与定位

4.3.1 光照补偿

由于肤色等色彩信息经常受到光源的颜色，图片采集设备的色彩偏差等因素影响，人脸颜色也会随着光照发生一定程度的变化，因此常对人脸图片进行预处理来补偿光照。可采用直方图均衡法、灰度校正等的方法。

由于实验室光照较为稳定，这一步骤暂时可省略。

4.3.2 肤色建模

对于任意一幅的图片，如果某个像素满足下式给定的条件即认为肤色，否则就是非肤色。肤色建模后，将得到二值图片，肤色的像素值用 255 表示，非肤色用 0 表示。

$$85 \leq U \leq 126$$

$$130 \leq V \leq 165$$

这里我们在动态内存创建大小为 W*H/16 字节的临时空间向量 tempImage，用于存放人脸肤色建模结果。

Listing 5: 肤色建模函数

```
DLL_EXP void SkinColorModeling(aBYTE* pU, aBYTE* pV, aBYTE* tempImg, int
    size) {
        for (int i = 0; i < size; i++) {
                if (pU[i] >= 85 && pU[i] <= 126 && pV[i] >= 130 && pV[i] <=
                    165) {
                        tempImg[i] = 255;
                }
                else {
                        tempImg[i] = 0;
                }
        }
}
```

4.3.3 形态学处理

由于获得的灰度图像存在大量的孤立噪声点、连接间断和区域孔洞，为了去除上述干扰，更好地获得物体形状，需要对二值图片进行形态学处理。

最基础的形态学操作有膨胀和腐蚀。膨胀操作可以填补二值图像中的小孔洞，连接断开的区域，而腐蚀操作则可以去除孤立的噪声点。

由膨胀和腐蚀两种基本操作可以组合出开运算和闭运算。开运算是先腐蚀后膨胀，主要用于去除小的噪声点或者分离相邻的目标区域；闭运算是先膨胀后腐蚀，主要用于填补小的孔洞或连接断裂的目标区域。

这里首先定义了带参数 N 的膨胀和腐蚀函数，然后利用这两个函数实现开运算和闭运算。在设计上首先进行行处理，然后进行列处理，以提高效率。

Listing 6: 腐蚀函数

```
DLL_EXP void erode(aBYTE* src, aBYTE* dst, int w, int h, int N) {
        aBYTE* tempImage1 = myHeapAlloc(w * h);
        memset(tempImage1, 0, w * h);

        int r = (N - 1) / 2;
        int i, j, k;

        for (i = r * w; i < (h - r) * w; i += w) {
                for (j = r; j < w - r; j++) {
                        tempImage1[i + j] = 255;
                        for (k = i - r * w; k <= i + r * w; k += w) {
                                if (src[k + j] == 0) {
```

```
13                                      tempImage1[i + j] = 0;
14                                      break;
15                              }
16                      }
17              }
18      }//行腐蚀
19      for (i = r; i < w - r; i++) {
20              for (j = r * w; j < (h - r) * w; j += w) {
21                      dst[i + j] = 255;
22                      for (k = i - r; k <= i + r; k++) {
23                              if (tempImage1[k + j] == 0) {
24                                      dst[i + j] = 0;
25                                      break;
26                              }
27                      }
28              }
29      }//列腐蚀
30      myHeapFree(tempImage1);
31  }
```

Listing 7: 膨胀函数

```
1   DLL_EXP void dilate(aBYTE* src, aBYTE* dst, int w, int h, int N) {
2       int r = (N - 1) / 2;
3       int i, j, k;
4       aBYTE* tempImage1 = myHeapAlloc((w + 2 * r) * (h + 2 * r));
5       aBYTE* tempImage2 = myHeapAlloc((w + 2 * r) * (h + 2 * r));
6       memset(tempImage1, 0, (w + 2 * r) * (h + 2 * r));
7       memset(tempImage2, 0, (w + 2 * r) * (h + 2 * r));
8
9       for (i = 0; i < h * w; i += w) {
10              for (j = 0; j < w; j++) {
11                      tempImage2[i + r * (w + 2 * r) + j + r] = src[i + j
                            ];
12              }
13      }
14      for (i = 0; i < h * w; i += w) {
15              for (j = 0; j < w; j++) {
16                      tempImage1[i + j] = 0;
```

```
                    for (int k = i - r * w; k <= i + r * w; k += w) {
                        if (tempImage2[k + j] != 0) {
                            tempImage1[i + j] = 255;
                            break;
                        }
                    }
                }
            }//行膨胀
            for (i = 0; i < w; i++) {
                for (j = 0; j < h * w; j += w) {
                    dst[i + j] = 0;
                    for (k = i - r; k <= i + r; k++) {
                        if (tempImage1[k + j] != 0) {
                            dst[i + j] = 255;
                            break;
                        }
                    }
                }
            }//列膨胀
            myHeapFree(tempImage2);
            myHeapFree(tempImage1);
}
```

然后，用 3×3 的矩形结构元素对人脸肤色建模图片 tempImage 做一次开运算，再用 7×7 的矩形结构元素做一次闭运算.

Listing 8: 形态学处理

```
//开运算
erode(tempImage, tempImage, w / 4, h / 4, 3);
dilate(tempImage, tempImage, w / 4, h / 4, 3);
//闭运算
dilate(tempImage, tempImage, w / 4, h / 4, 7);
erode(tempImage, tempImage, w / 4, h / 4, 7);
```

4.3.4 连通域标记

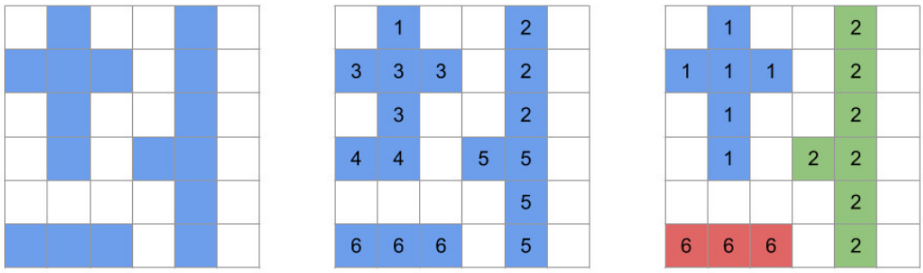

Figure 3: 连通域标记算法示意图

连通域标记是一种图像处理技术，用于在二值图像中标记和区分不同的连通区域。连通域标记的目标是将图像中的像素根据连通性分组，每个连通区域分配一个唯一的标签。连通域标记又可以分为四连通域标记和八连通域标记。四连通域标记只考虑像素的上下左右四个方向的邻居，而八连通域标记则考虑上下左右以及四个对角线方向的邻居。在本实验中，采用四连通域标记算法。具体流程如下：

(1). 连通域临时标记

- 按照从左到右，从上到下顺序扫描所有“白点”像素。
- 如果此点左面和上面两个像素均为“黑点”，则暂时判定为一个新区域，分配一个新的标号 L。
- 如果此点的左边或上边有且只有一个“白点”，则复制这个“白点”的标号值。
- 如果此点的左边或上边两个均为“白点”，且两点标号相同，则复制该标号即可。如果两点标号不同（分别用 Left、Up 表示，实际上 Left、Up 为两个等价标号），则复制左边白点的标号值，并将等价关系记录在等价表中。

由于图像是逐行扫描的，无法在第一次扫描时立即解决所有标签的冲突，因此需要一个等价表来记录这些关系，并在后续阶段中合并等价的标签，这里使用数组 LK 来存储等价表。

Listing 9: 连通域临时标记

```
int* tempImage = (int*)calloc(w * h, sizeof(int));
int* LK = (int*)calloc(1024, sizeof(int));

int i, j;
for (i = 0; i < 1024; i++) {
        LK[i] = i;
}
```

```
8   //连通区域临时标记
9   int index = 1, left = 0, up = 0, Lmin, Lmax;
10  for (i = 0; i < h * w; i += w) {
11          for (j = 0; j < w; j++) {
12                  if (src[i + j] == 0) continue;
13
14                  if (i == 0) {
15                          if (j == 0) {
16                                  tempImage[i + j] = index++;
17                          }
18                          else {
19                                  if (src[i + j - 1] == 255) {
20                                          tempImage[i + j] = tempImage[i +
                                                j - 1];
21                                  }
22                                  else {
23                                          tempImage[i + j] = index++;
24                                  }
25                          }
26                  }
27                  else if (j == 0) {
28                          if (src[i + j - w] == 0) {
29                                  tempImage[i + j] = index++;
30                          }
31                          else {
32                                  tempImage[i + j] = tempImage[i + j - w];
33                          }
34                  }
35                  else {
36                          if (src[i + j - 1] == 0 && src[i + j - w] == 0)
                                {
37                                  tempImage[i + j] = index++;
38                          }
39                          else if (src[i + j - 1] == 255 && src[i + j - w
                               ] == 255) {
40                                  tempImage[i + j] = tempImage[i + j - 1];
41                                  if (tempImage[i + j - w] > tempImage[i +
                                        j - 1]) {
```

```
42                                  Lmax = tempImage[i + j - w];
43                                  Lmin = tempImage[i + j - 1];
44                              }
45                              else {
46                                  Lmin = tempImage[i + j - w];
47                                  Lmax = tempImage[i + j - 1];
48                              }
49                              if (Lmax == Lmin || (left == tempImage[i
                                    + j - 1] && up == tempImage[i + j -
                                    w])) continue;
50                              while (LK[Lmax] != Lmax) {
51                                  Lmax = LK[Lmax];
52                              }
53                              while (LK[Lmin] != Lmin) {
54                                  Lmin = LK[Lmin];
55                              }
56                              if (Lmax > Lmin) {
57                                  LK[Lmax] = Lmin;
58                              }
59                              else {
60                                  LK[Lmin] = Lmax;
61                              }
62                              left = tempImage[i + j - 1];
63                              up = tempImage[i + j - w];
64                          }
65                          else {
66                              if (src[i + j - 1] == 255) {
67                                  tempImage[i + j] = tempImage[i +
                                        j - 1];
68                              }
69                              else {
70                                  tempImage[i + j] = tempImage[i +
                                        j - w];
71                              }
72                          }
73                      }
74                  }
75              }
```

(2). 整理等价表

连通域初步标号后，等价关系还存在不是最低等价标号，因此我们采取从等价表地址 1 开始，依次检查各个等价标号时否为最低等价标号，符合式 $LK[i] = LK[LK[i]]$ 即最低等价标号，若不是最低等价标号，采用追踪方法，置换最低等价标号。

Listing 10: 整理等价表

```
int temp;
for (i = 1; i < index; i++) {
        if (LK[i] != LK[LK[i]]) {
                temp = LK[i];
                while (LK[temp] != temp) {
                        temp = LK[temp];
                }
                LK[i] = temp;
        }
}
```

(3). 连通区域重新编号

整理等价表后，需要根据临时标号与最终标号的对应关系，合并标签并对连通区域进行重新编号。从而保证将图像中所有的连通区域分配唯一的标签，并确保这些标签是连续的、规范化的。

Listing 11: 连通区域重新编号

```
int type = 1;
for (i = 1; i < index; i++) {
        if (i == LK[i]) {
                LK[i] = type++;
        }
        else {
                LK[i] = LK[LK[i]];
        }
}
```

4.3.5 人脸区域定位及显示

标记连通域后，统计各连通域的像素总数，找出最大连通域（脸部），该连通域像素值设为 255，删去最大连通域以外的内容（非脸部的连通域置 0），图片 tempImage 重新成为二值图。

Listing 12: 找出最大连通域

```
int* pixelSum = (int*)calloc(type, sizeof(int));
//统计各连通域像素数
for (i = 0; i < h * w; i += w) {
	for (j = 0; j < w; j++) {
		if (tempImage[i + j] != 0) {
			if (LK[tempImage[i + j]] <= 255) {
				dst[i + j] = LK[tempImage[i + j]];
				if (dst[i + j] < type) {
					pixelSum[LK[tempImage[i + j]]]++;
				}
			}

		}
	}
}
//找到最大连通域
int maxIdx = 1;
for (i = 1; i < type; i++) {
	if (pixelSum[i] > pixelSum[maxIdx]) {
		maxIdx = i;
	}
}
//最大连通域标记为白色
for (i = 0; i < h * w; i += w) {
	for (j = 0; j < w; j++) {
		if (dst[i + j] == maxIdx) {
			dst[i + j] = 255;
		}
		else {
			dst[i + j] = 0;
		}
	}
}
```

计算脸部区域矩形范围存于 rcnFace 结构体、统计脸部区域肤色像素总数存于 nFacePixelNum 结构体。将两值化图片 tempImage 扩展到降采样彩色图片中的灰度部分，注意在扩展时需要将左边坐标、宽度、像素总数都放大一倍来适应降采样图片的尺寸。最后

在视频中标记框选出人脸的位置。

Listing 13: 人脸定位函数

```
DLL_EXP void faceLoc(aBYTE* src, aRect* face, int w, int h, BYTE* pBuffer ) {
    int left=0, top=0, width=0, height=0;
    int flag = 0;
    int i, j;
    //找顶部与高度
    for (i = 0; i < w * h; i += w) {
        for (j = 0; j < w; j++) {
            if (src[i + j] == 255) {
                if (flag == 0) {
                    top = i / w;
                    flag = 1;
                }
                break;
            }
        }
        if ((j == w && flag == 1) || (i == (w * (h - 1)))) {
            height = i / w - top;
            break;
        }
    }
    //找左边界和宽度
    flag = 0;
    for (i = 0; i < w; i++) {
        for (j = 0; j < w * h; j += w) {
            if (src[i + j] == 255) {
                if (flag == 0) {
                    left = i;
                    flag = 1;
                }
                break;
            }
        }
        if ((j == w * h && flag == 1) || (i == w - 1)) {
            width = i - left;
            break;
```

```
36                  }
37          }
38          ShowDebugMessage("left=%d, width=%d, top=%d, height=%d", left,
                width, top, height);
39          //目标区域边界信息存到缓冲区，左边界、宽度、像素总数放大1倍
40          ((BUF_STRUCT*)pBuffer)->rcnFace.left = left * 2;
41          ((BUF_STRUCT*)pBuffer)->rcnFace.top = top;
42          ((BUF_STRUCT*)pBuffer)->rcnFace.width = width * 2;
43          ((BUF_STRUCT*)pBuffer)->rcnFace.height = height;
44          ((BUF_STRUCT*)pBuffer)->nFacePixelNum = width * height * 2;
45          //将图像复制到缓冲区
46          for (i = 0; i < w * h; i += w) {
47                  for (j = 0; j < w; j++) {
48                          ((BUF_STRUCT*)pBuffer)->clrBmp_1d8[(i + j) * 2] =
                               src[i + j];
49                          ((BUF_STRUCT*)pBuffer)->clrBmp_1d8[(i + j) * 2 + 1]
                               = src[i + j];
50                  }
51          }
52          //在视频中标记出人脸位置
53          face->left = ((BUF_STRUCT*)pBuffer)->rcnFace.left * 2;
54          face->width = ((BUF_STRUCT*)pBuffer)->rcnFace.width * 2;
55          face->top = ((BUF_STRUCT*)pBuffer)->rcnFace.top * 4;
56          face->height = ((BUF_STRUCT*)pBuffer)->rcnFace.height * 4;
57  }
```

5 实验结果和分析

5.1 摄像头视频流图片截取、重采样等处理插件

如图，左上角为十六分之一灰度采样图，右上角为八分之一彩色采样图

Figure 4: 重采样结果

5.2 人脸检测与定位插件

如图，人脸检测成功，在区域内用矩形框标出人脸位置。左上角为形态学处理结果，右下角为肤色建模结果。

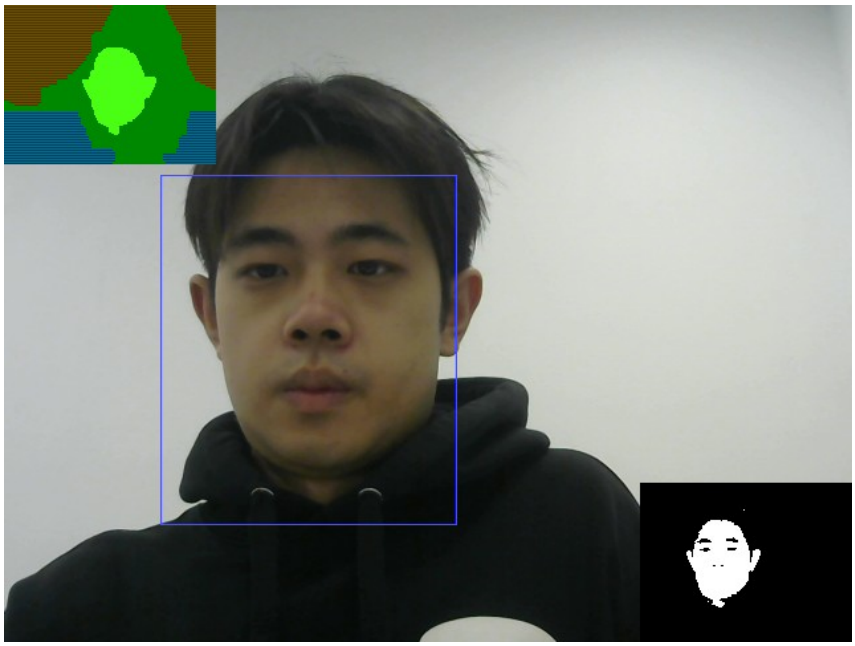

Figure 5: 人脸检测定位结果

6 心得体会

在实验中，我主要完成了基于 C++ 的人脸分割系统的设计与实现。实验内容涵盖了从图像采集、重采样到人脸检测与定位的完整流程。我学习了许多图像处理的基本理论，例如 YUV 色彩空间、形态学处理、连通域标记等。这些知识在实验中得到了充分的应用。通过本次实验，我不仅加深了对数字图像处理技术的理解，还提升了实际编程能力。面对实验中遇到的各种挑战，如图像噪声处理、算法优化等，我学会了如何分析问题并寻找解决方案。

让我印象比较深刻的是，对于膨胀和腐蚀等形态学操作的实现。这些操作对计算效率要求较高，尤其是在实时处理视频流时，算法的效率直接影响系统的性能。在实现过程中，我学习了如何通过行列分步处理来优化算法效率，从而提高整体处理速度。

这次实验是一次宝贵的学习经历，我将以此为起点，继续深入学习和探索，为未来的专业发展打下坚实的基础。