“第一次编程作业”实验报告

苏壮 22307140012

简介(Introduction)：

本文中我们将使用 C++语言实现一次**模板匹配(template matching)**。模板匹配是指在一个较大尺寸的图像中寻找一个小尺寸图像(称为模板图像)位置的过程。例如图 1 是一张分辨率为 1920 * 1280 的高清图片，而图 2 则是分辨率为 16 * 16 的小模版。我们会在图 1 中找到图 2 对应的位置，图 3 中的绿颜色方框(bounding box)即为寻找结果。

图 1 原图片 source image, 1920 * 1280

图 2 模板图片 template image, 16 * 16

图 3 匹配结果，1920 * 1280

模板匹配的原理，就是将模板放到原图像上，当作一个滑窗，从左到右，从上到下进行二维遍历，过程中不断计算其与原图像对应区域的匹配度。遍历后即可得到匹配度最高的区域。模板匹配的具体实现方法众多，对于不同测试集，常常采用不同的处理方法。观察图 1~3，不难发现模板图 2 不仅仅是从图 1 中截取下来，而是经过模糊，尺度缩放，降采样，亮度变化等一系列操作后得到的。因此我制定了以下流程：

读取并存储图像文件→处理原图像(模糊、缩放、降采样)→用模板遍历原图像并得到匹配度→输出结果

在下文中，我将会逐一介绍流程中的详细思路。

编译环境(Compilation Environment):

我提供了一个 OBJ.cpp 源文件，你可以在 Windows 上任何支持 C++20 标准的环境中编译它。没有使用任何图像处理库，你不必配置例如 OpenCV 的环境才能运行。而 C++20 是必须的，因为在代码使用了 std::format 这样的特性，除非你找到对应代码段并将之删除或替换。

我自己是在 vscode 中用 cmake 配置项目，Visual Studio Commutity 2022 Release - amd64 工具集(内含 MSVC 编译器)编译运行的。而我推荐你直接在 Visual Studio 2022 中新建一个项目，将源文件添加或将代码复制到项目中,并且在项目-属性-配置属性-常规-C++语言标准中选择 C++20，之后便可正常编译。

另外，我也提供了可执行文件 pj1.exe。你可以在 Windows 上直接运行它，会匹配名为 inpu1.bmp 和 input2.bmp(即图 1 和图 2)的文件，并将结果输出到 output.txt 中。你也可以在终端或命令行中输入 .\pj1.exe 1 或将 1 换成其他 1 到 100 之间的数字，它会在匹配测试集中对应的文件(例如 test001.bmp 和 obj001.bmp)并将统计结果输出到 output.txt，以及一张带有 bounding box 的图片(图片未必输出成功，仅作调试使用)。

读取图像文件(Read the Images):

测试图像的格式是 24 位**位图(bitmap)**。位图并不是常见的图像格式。但它储存数据的方式简单，因此便于学习。如果你感兴趣，可以参考这篇文章 https://zhuanlan.zhihu.com/p/25119530?utm_source=wechat_session&utm_medium=social&utm_oi=1372317944190476288。对于本项目，我们无需了解位图的全部细节。我直接采用了 http://www.kalytta.com/bitmap.h 链接中的代码操作位图，只修改并增加了部分接口，来适配我的需求。OBJ.cpp 中大约前 800 行便是对应的代码。代码数量比较多，因为支持了很多除 24 位以外的位图格式。其中接口设计得不方便，因此我不推荐你在自己项目中使用（你可以去尝试 stb_image 这种成熟的图像库）。接下来我结合代码来讲此项目必需的位图知识。

```
// read and write bitmap files
class CBitmap {
public:
    BITMAP_FILEHEADER m_BitmapFileHeader;
    BITMAP_HEADER m_BitmapHeader;
    RGBA *m_BitmapData;
    unsigned int m_BitmapSize;
    // Masks and bit counts shouldn't exceed 32 Bits
```

图 4 CBitmap 类

代码主要定义了一个名字是 CBitmap 的类，所有读写位图的函数都写在这个类里。.bmp 文件在颜色数据前有两个文件头，记录了一些文件的总体信息，例如 bmp 文件签名，文件大小，位数，宽度和高度。我们对应定义了两个结构体，在读取文件时将这些信息写入结构体中。信息中最值得关心的是文件的宽度(width)和高度(height)。因为在之后的操作中我们会频繁地获取它们，并且如果我们要改变图片尺寸，也需要手动修改宽度和高度。

关于颜色，CBitmap 中定义了一个指向 RGBA 类型的指针 m_BitmapData。RGBA 是自定义的结构体类型，里面四个 uint8_t 类型变量分别存储红绿蓝和透明度。感兴趣的同学可以学习色彩空间的知识。读取图片时会在堆上开辟一段连续的储存空间，m_BitmapData 即指向此空间的首地址。需要说明的是，位图中数据不会以 RGBA 的形式存储，我略去了中间的转化细节，事实上 24 位位图不会存储透明度 alpha，但这不会影响我们的项目。

生成灰度图像(Generate Grayscale Image):

在图像处理中，我们经常使用灰度图像进行研究。所以在读取图片后我们要将颜色数据转为灰度。RGB 转为灰度没有标准的公式。我采用的是公式如图 5 所示：

```
uint8_t R8G8B8A82GR(RGBA rgba)
{
    // quick calculation
    return (rgba.Red * 76 + rgba.Green * 150 + rgba.Blue * 30) >> 8;
}
```

图 5 转化灰度函数

这是一个经验公式，并针对 8 位颜色进行了计算加速。你可以在 https://blog.csdn.net/a200800170331/article/details/51564854 中找到更多信息。

```cpp
uint8_t* image_gray_buffer = new uint8_t[image_bmp_copy->GetSize()];
RGBA* image_rgba_buffer = reinterpret_cast<RGBA*>(image_bmp_copy->GetBits());

for (unsigned int i = 0; i < image_bmp_copy->GetSize(); ++i)
    image_gray_buffer[i] = R8G8B8A82GR(rgba: image_rgba_buffer[i]);
```

图 6 储存灰度信息

对于每一个像素，调用上述转化为灰度的方法，最后用一个 uint8_t 类型的指针储存。

矩阵存储与坐标表示(Use Matrix and Coordinates):

如今我们用数组线性储存灰度。然而图像是由二维的**像素(pixel)**矩阵描述的，我们后续的操作用矩阵描述也更加方便。所以我写了一个 matrix 类，接收灰度数据。

```cpp
// correspond to the coordinate system
matrix(uint8_t* data, unsigned int rows, unsigned int cols) : rows(rows), cols(cols)
{
    allocSpace();
    for (unsigned int i = 0; i < rows; ++i)
    {
        for (unsigned int j = 0; j < cols; ++j)
        {
            p[i][j] = data[(rows - i - 1) * cols + j];
        }
    }
}
```

图 7 矩阵类的构造函数

可以看到矩阵元素和数组元素的对应方式有些奇怪，这是因为本项目我们采用的坐标系统以左上角为原点，x 轴正半轴水平向右，y 轴正半轴则竖直向下。而位图原始储存方式则从图片左下角开始。这个转化细节被隐藏在 matrix 类的构造函数中，而在使用时则无需考虑，只需要直到本项目的坐标系规定。同时 matrix 也重载了[ ]运算符，同时和数学中矩阵元素的表示联系在一起。例如 pixel_matrix 是 matrix 类型，则 pixel_matrix[1][2]代表矩阵第 2 行，第 3 列的元素([ ]表示从 0 开始)，同时它的坐标为(x = 2, y = 1).

		pixel_matrix[1][2]		

图 8 矩阵存储的表示方法

图像相关性度量(Image Correlation):

接下来我们要寻找一种评分标准，采取何种统计量计算匹配度。不同统计量之间有准确率和速度之间的差异，你可以在 OpenCV 的文档上看到直观的对比 https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html。我采用的是一种叫做 the normalized cross correlation coefficient(NCC)的统计量，总的来说它需要的计算量最大，因此速度较慢，但准确率相对更高。NCC 的计算方法为：

$$\frac{1}{n\sigma_f\sigma_t}\sum_{x,y}\big(f(x,y)-\mu_f\big)\big(t(x,y)-\mu_t\big)$$

其中，f()，t()在本项目中为原图和模板的像素灰度矩阵，n 为模板像素总数。μ和σ分别是均值和标准差。NCC 的值在-1~1 之间，越接近 1 匹配程度越高。你可以在 https://blog.csdn.net/fb_help/article/details/104162770 中找到更多关于 NCC 的消息。

用 C++计算 NCC 需要多次遍历，因为标准差的求解依赖于均值，不能同时得到，具体操作可参照代码。另外关于遍历的其他细节，例如遍历范围的上下界问题，请同样参考代码，不在此处讲解。

尺度缩放(Scaling):

如果模板只是从原图片中截图得到，那么做到这里已经可以匹配到了。因为模板会和原图中某一部分重合，NCC 值为 1。但是上文提到，图 2 经历了模糊，尺度缩放，降采样，亮度变化等一系列操作。NCC 可以包容亮度变化，而模糊和降采样是尺度缩放的中间步骤。因此，接下来我们会对原图进行尺度缩放到相同比例，再统计 NCC。而具体的缩放程度，需要人为设置一个范围和步长，匹配多次直至得到结果。

高斯滤波(Gaussian Filtering)：

高斯滤波或者高斯模糊是降采样前的必经步骤，它可以滤去图片信号中的高频部分，避免降采样后图像形成锯齿。高斯模糊利用了卷积计算，使用的卷积核叫高斯核。高斯核是一个正方形大小的滤波核，其中每个元素的计算都是基于下面的高斯方程：

$$G_{(x,y)}=\frac{1}{2\pi\sigma^2}e^{-\frac{x^2+y^2}{2\sigma^2}}$$

其中，σ是标准差(一般取值为 1)，x 和 y 分别对应了当前位置到卷积核中心的整数距离。在本项目中，我们使用了 5 * 5 的高斯核，并将之拆分为两个一维高斯核。

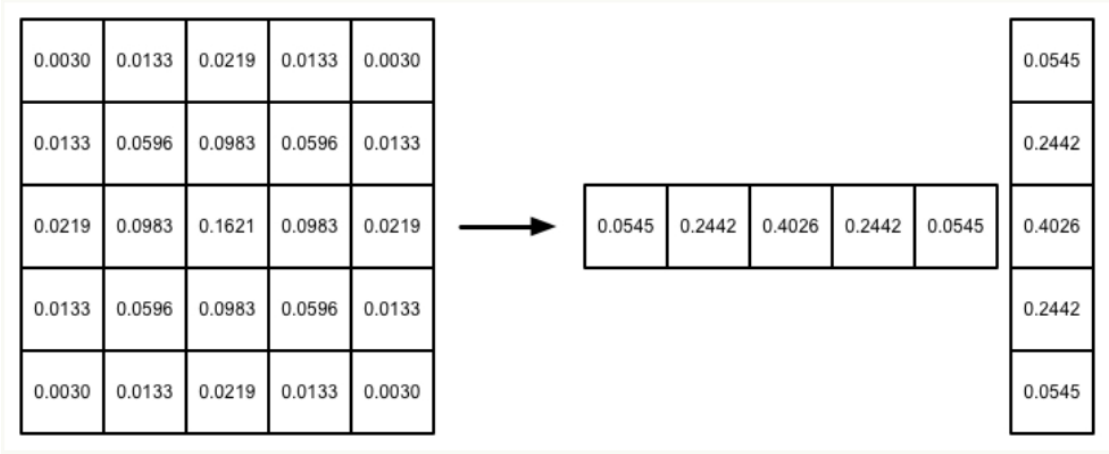

图 9 一个 5*5 大小的高斯核。左图显示了标准方差为 1 的高斯核的权重分布，我们可以把这个二维高斯核拆分成两个一维的高斯核(右图) 图片来源《Unity Shader 入门精要》冯乐乐著

代码中 void GaussianFilter(CBitmap* bmp)函数有详细的实现过程。由于一维高斯核的对称性，我们实际只需要存储 3 个数就可以实现高斯模糊。一次高斯模糊的结果通常不明显，这时可以多进行几次得到更好的效果。

最近邻插值(Nearest Interpolation):

通常的降采样实现方法是在滤波后，保留原图片偶数或奇数行和列，但本项目需要各向异性缩放，所以用这种方法并不方便。代码中函数 void DownSample(CBitmap* bmp)提供了其实现，虽然我们不会使用它。

我们最终采取的是插值方法中的最近邻插值，它在能保证一定精度的同时速度最快。用最近邻插值缩放的思想是对于缩放后图片的像素点，利用缩放比例找到缩放前的像素点，直接采用其颜色。在代码实现时，需注意取整和边界处理。经测试，最近邻插值可以满足本项目的需求。如果你需要，也可以自行实现**双线性插值(Bilinear)**等其他插值方法。

输出结果(Output the Results):

现在终于能够统计 NCC 来得到匹配结果了！我们定义一个矩阵来记录个像素点的 NCC 值。理论上，只需要得到 NCC 最大的值即是匹配结果。但也可以统计高于一个阈值的所有像素点。

衡量一个图像匹配算法的量有准确率和运行时间，这是很直观的，好的方法会运行得又快又准。运行时间可以通过 C++标准库函数得到，而本项目采用**准确率(Accuracy)**和**交并比(IoU)**来衡量准确程度。准确率用匹配结果和**真实数据(Ground Truth)**的重叠面积除以模板面积得到，交并比是指重叠面积与匹配结果和真实数据的面积和减去重叠面积(即所谓“并”面积)的比值求得。当然也可以调用代码提供的 DrawRectangle 函数画出 bounding box。你也可以自己实现生

成 NCC 灰度图的函数，将统计结果输出到一张图中。

结语(Conclusion):

至此我们便完成了一次模板匹配。模板匹配还有很多其他的方法，例如基于灰度的匹配，基于边缘检测的匹配。另外，本项目测试集(应该)没有对模板进行旋转和切变，这两种变换以及缩放变换都是**线性变换(Linear Transformation)**，可以在我们的代码上直接应用变换矩阵加以解决。