Broadview
www.broadview.com.cn

站在浪潮之巅，零距离接触最前沿的计算机视觉编程技术

全面涵盖OpenCV2、OpenCV3双版本的核心编程技巧

附赠OpenCV2、OpenCV3双版本总计200余个配套示例程序源代码

OpenCV3 编程入门

毛星云　冷雪飞　等编著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

OpenCV3
编程入门

毛星云　冷雪飞　王碧辉　吴松森　编著

電子工業出版社
Publishing House of Electronics Industry

内 容 简 介

OpenCV 在计算机视觉领域扮演着重要的角色。作为一个基于开源发行的跨平台计算机视觉库，OpenCV 实现了图像处理和计算机视觉方面的很多通用算法。本书以当前最新版本的 OpenCV 最常用最核心的组件模块为索引，深入浅出地介绍了 OpenCV2 和 OpenCV3 中的强大功能、性能，以及新特性。书本配套的 OpenCV2 和 OpenCV3 双版本的示例代码包中，含有总计两百多个详细注释的程序源代码与思路说明。读者可以按图索骥，按技术方向进行快速上手和深入学习。

本书要求读者具有基础的 C/C++知识，适合研究计算机视觉以及相关领域的在校学生和老师、初次接触 OpenCV 但有一定 C/C++编程基础的研究人员，以及已有过 OpenCV 1.0 编程经验，想快速了解并上手 OpenCV2、OpenCV3 编程的计算机视觉领域的专业人员。本书也适合于图像处理、计算机视觉领域的业余爱好者、开源项目爱好者做为通向新版 OpenCV 的参考手册之用。

本书配套的【示例程序】、【.exe 可执行文件】、【书内彩图】的下载链接可通过扫描本书封底后勒口的二维码获取。

图书在版编目（CIP）数据

OpenCV3 编程入门 / 毛星云等编著. —北京：电子工业出版社，2015.2

ISBN 978-7-121-25331-7

Ⅰ. ①O… Ⅱ. ①毛… Ⅲ. ①图象处理软件－程序设计 Ⅳ. ①TP391.41

中国版本图书馆 CIP 数据核字（2014）第 310454 号

责任编辑：陈晓猛
印　　刷：北京京科印刷有限公司
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编　100036
开　　本：787×1 092　1/16　印张：29.25　字数：653 千字
版　　次：2015 年 2 月第 1 版
印　　次：2015 年 2 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

计算机视觉是一个近几年来日臻成熟的领域。随着运算性能强劲而又价格实惠的计算设备的不断问世，创建复杂的图像应用从未像今天这般容易。OpenCV在计算机视觉领域扮演着重要的角色，它是一个基于开源发行的跨平台计算机视觉库，实现了图像处理和计算机视觉方面的很多通用算法。自1999年问世以来，OpenCV 已经被计算机视觉领域的学者和开发者视为首选工具，并成为了计算机视觉领域最有力的研究工具之一。

OpenCV 最初由 Intel 的一个小组进行开发。在一系列的 beta 版本后，OpenCV 1.0 正式版本终于在 2006 年 10 月 19 日发布。

2009 年 10 月 1 日，OpenCV 2.0 问世，它带来了全新的 C++接口，将 OpenCV 的能力无限放大。在 2.0 的时代，OpenCV 增加了全新的平台支持，包括 iOS 和 Andriod，通过 CUDA 和 OpenCL 实现了 GPU 加速，为 Python 和 Java 用户提供了接口，基于 Github 和 Buildbot 构建了充满艺术感的持续集成的系统，所以才有了被全世界的很多公司和学校所采用的稳定易用的 OpenCV 2.4.x。

2014 年 8 月 21 日，OpenCV 3.0 Alpha 发布，它带来了全新的项目架构的改变，宣告了计算机视觉新时代的来临。和其他大型项目一样，OpenCV3 抛弃了整体统一架构，使用内核+插件的架构形式，让自身主体更加稳定，而附加的库则可以更加灵活多变，以保持高速的发展与迭代。

本书源自于笔者在 CSDN 上连载的名为“OpenCV 入门教程”的系列博客文章，自 2014 年 2 月 24 日发表第一篇以来，得到了广大 OpenCV 爱好者的广泛关注与支持，累计阅读量突破了 40 多万人次。不少读者强烈希望将这些内容集结成书，并加入更多新的内容。于是，经过笔者半年的笔耕不辍，便有了现在这本书的诞生。

作为一本入门级的 OpenCV 编程教材，本书以详细注释的程序代码为主线，以新版 OpenCV 最核心的 core、HighGUI、improc 和 feature2d 这 4 个组件的相关函数、类和数据结构为出发点，详细讲解了学习新版本 OpenCV 中会遇到了各种问题，并提供了详尽的实战代码作为参考。本书的写作初衷是让更多的使用者能熟练使用采用新版 C++接口的 OpenCV2 或 OpenCV3，了解 OpenCV2 和 OpenCV3 的诸多细节上的区别，以推动新版 OpenCV 在世界范围内的普及。

本书的内容安排

本书分为 4 个部分、11 个章节，现将内容梗概列举如下。

第 1 章　邂逅 OpenCV：介绍 OpenCV 的周边概念，分析 OpenCV 的基本架

构，讲解 OpenCV3 的新特性。本章重点讲解了 OpenCV 的下载、安装与配置过程；在配置完成后，带领大家正式领略 OpenCV 的魅力，讲解了 4 个 OpenCV 图像处理小程序，并指导大家学习如何使用 OpenCV 操作视频和调用摄像头。

第 2 章　启程前的认知准备：进行 OpenCV 官方例程的引导学习与赏析，讲解如何编译 OpenCV 的源代码，并引入了对一些周边概念的认知。

第 3 章　HighGUI 图形用户界面初步：对图像的载入、显示和输出到文件进行详细地分析，讲解 OpenCV 中滑动条的创建和使用，以及如何用鼠标进行交互操作。

第 4 章　OpenCV 数据结构与基本绘图：讲解 OpenCV 中常用的数据结构以及基本的绘图操作。

第 5 章　core 组件进阶：讲解 core 模块的一些进阶知识点，如操作图像中的像素、图像混合、分离颜色通道、调节图像的对比度和亮度、进行离散傅里叶变换，以及输入输出 XML 和 YAML 文件。

第 6 章　图像处理：学习各种利用 OpenCV 进行图像处理的方法，包括属于线性滤波的方框滤波、均值滤波与高斯滤波，属于非线性滤波的中值滤波、双边滤波；两种基本形态学操作——膨胀与腐蚀；5 种高级形态学滤波操作——开运算、闭运算、形态学梯度、顶帽以及黑帽；此外，还有漫水填充算法、图像金字塔、图像缩放、阈值化。

第 7 章　图像变换：讲解多种类型的图像变换方法。包括利用 OpenCV 进行边缘检测所用到的 canny 算子、sobel 算子，Laplace 算子以及 scharr 滤波器；进行图像特征提取的霍夫线变换、霍夫圆变换，重映射和仿射变换以及直方图均衡化。

第 8 章　图像轮廓与图像分割修复：讲解如何查找轮廓并绘制轮廓，如何寻找物体的凸包，使用多边形来包围轮廓，以及计算一个图像的矩。此外还介绍了分水岭算法和图像修补操作的实现方法。

第 9 章　直方图与匹配：讲解图像直方图相关的编程技巧，以及直方图对比、反向投影和模板匹配技术。

第 10 章　角点检测：讲解 Harris 角点检测和 Shi-Tomasi 角点检测，以及一种亚像素角点检测方法。

第 11 章　特征检测与匹配：使用 OpenCV2 讲解并实现了 SURF、SIFT 和 ORB 特征检测方法，并在 FLANN 特征匹配的基础上，进一步实现了利用 Homography 映射来找出已知物体。

适合阅读本书的读者

- 研究计算机视觉以及相关领域的在校学生和老师

本书拥有详实的内容，注释详尽的代码，会是助你通过 OpenCV 来研习计算

机视觉理论、撰写论文、通过毕业设计、完成科研项目的得力工具。同时，本书适合作为大学计算机视觉课程的教学用书。

- 初次接触 OpenCV、有一定 C/C++编程基础的研究人员

作为一本定位为快速入门新版 OpenCV 标准的编程教程，本书需要的仅仅是一些简单的 C/C++编程语言基础。如果你已经具备了这些基础，并对计算机视觉感兴趣，那么本书正是为你所准备的。

- 已经有过 OpenCV 1.0 编程经验，想快速了解并上手 OpenCV2、OpenCV3 编程的计算机视觉领域的专业人员

如果你曾经使用过 OpenCV 1.0，或者研读过 OpenCV 1.0 时代的经典著作《Learning OpenCV》，本书会让你倍感亲切。你会发现新版 OpenCV 带了更多强大和便利的特性，让你事半功倍，如虎添翼。

- 想拥有一本新版 OpenCV 接口工具书的计算机视觉爱好者

本书中将自 OpenCV2 以来（包括 OpenCV3）的常用类和函数进行了详细地讲解，并在附录中提供了“书本核心函数清单”以便检索。你会在书中快速查找到你需要用到的函数、数据结构和类的用法。

- 想拥有海量的详细注释的 OpenCV2、OpenCV3 示例程序代码的 OpenCV 爱好者

本书包含 OpenCV2 版的 95 个书本主线示例程序源代码、21 个附赠示例程序源代码，OpenCV3 版的 95 个书本主线示例程序源代码。OpenCV2、OpenCV3 两版代码提供分开下载。这些程序代码都经过详细而有条理的注释，并提供可以独立运行的.exe文件供快速查看程序效果，方便查看和检索。你会在海量的示例程序中找到你需要的参考代码，从而加速你的研究和学习。

- 图像处理、计算机视觉领域的业余爱好者

海阔凭鱼跃，天高任鸟飞，计算机视觉领域的宝库任你探索。

- 开源项目爱好者

OpenCV 作为一个完全免费并开源代码开发的计算机视觉代码库，有总计上百万行的源代码供你研究学习，本书将是引导你学习它们的良师益友。

本书的示例程序说明

本书的示例程序最初都在 OpenCV 2.4.9（2014 年 4 月 15 日面世）版本下开发，书稿初版也是基于 OpenCV 2.4.9 而写。在书稿写作和修订过程中，恰逢 OpenCV 3.0 Alpha（2014 年 8 月 21 日）和 OpenCV3 Beta（2014 年 11 月 11 日）的发布，所以本书在审校和修订过程中（2014 年 12 月 1 日），决定站在浪潮之巅，以 OpenCV2 为主，加入 OpenCV3 的诸多特性，让这本书可以同时胜任 OpenCV2 和 OpenCV3 两个版本教材的角色。这也是为什么本书会有 OpenCV2 和 OpenCV3

两个独立版本的示例程序的原因。

两个版本、详细注释的 100 多个示例程序源代码是本书的灵魂，现将示例程序的相关情况概括如下。

- 本书包含 OpenCV2 版的 95 个书本主线示例程序源代码、21 个附赠示例程序源代码，以及 OpenCV3 版的 95 个书本主线示例程序源代码。
- OpenCV2、OpenCV3 两版代码提供分开下载。
- OpenCV2 版的示例程序在 Windows7 64 位旗舰版、Visual Stuido 2010 、OpenCV 2.4.9 的环境下开发与测试，理论上支持 OpenCV2 系列的所有版本的编译运行。
- OpenCV3 版的示例程序在 Windows7 64 位旗舰版、Visual Stuido 2010 、OpenCV 3.0 beta 的环境下开发与测试，理论上支持目前已经发布的 OpenCV3 全版本。
- 程序源代码都经过详细而有条理的注释。
- 额外提供可以独立运行的.exe 文件供快速查看程序效果，并方便检索。

本书配套示例代码的下载方式有以下几种。

- 扫描本书封面后勒口的二维码，得到下载地址。
- 在作者博客（http://blog.csdn.net/poem_qianmo）中单击相应的书本维护博文里贴出的下载链接。
- 在电子工业出版社的官方网站（http://www.phei.com.cn）中进行下载。
- 直接用搜索引擎搜索“《OpenCV3 编程入门》书本配套源代码”，找到对应的下载地址进行下载。

致谢

首先需要感谢我的导师，南京航空航天大学冷雪飞教授的知遇之恩，她也亲自参与撰写了本书的部分章节。在攻读硕士学位阶段，如果没有导师的谆谆教诲，我不会和 OpenCV 相遇，也就不会有此书的出版。

感谢我的同门师兄王碧辉与吴松森参与撰写本书的部分章节，他们为本书的完善做出了卓越的贡献。

感谢 OpenCV 开发团队为世界研发出如此强大且稳定、易用的计算机开源视觉库，并持续不断地对其进行维护与更新。

感谢父母将我养育成人，感谢家人们的嘘寒问暖，你们是我最坚强的后盾。

感谢母校南京航空航天大学赐予我一颗不甘平庸、上下求索的心。

感谢南京航空航天大学的戴泉晨老师对本书出版所做出的帮助与支持。

感谢国家自然科学基金青年科学基金项目“新型单定子二自由度超声电机及其驱动的智能云台系统的关键技术研究”（项目批准号：51205193）对本书理论研究方面提供的经费支持。

感谢电子工业出版社博文视点的陈晓猛和丁一琼编辑为本书的出版所做的大量的工作，他们对出版物的专业和严谨的态度给我留下了深刻的印象。

最后，需要感谢我博客上的众多读者们，是你们对这本书的期待和热情的留言让我有了完成这本书的动力和勇气。

交流与勘误

由于编者水平有限，书籍即使经过了多次的校对，也难免会有疏漏之处。希望书本前的你，能够热心地指出书本中错误，以便在这本书下一版印刷的时候，能以一个更完美更严谨的样子，呈现在大家的面前。另外，你要相信你不是一个人在战斗，在作者的博客中，可以找到与自己志同道合的众多喜欢计算机视觉编程技术的爱好者们。我们可以一同交流，共同学习进步。

最后，愿大家在本书的帮助下，都能很好地入门和掌握新版 OpenCV。

愿本书能为新版 OpenCV 在国内的普及以及在世界范围内的发展，献上绵薄之力。

作者博客地址：http://blog.csdn.net/poem_qianmo

作者联系邮箱：happylifemxy@163.com

作者新浪微博：@浅墨_毛星云

浅墨

2014 年 12 月于南京

目　录

第一部分
快速上手 OpenCV

欢迎来到奇妙的 OpenCV 编程世界，作为本书的作者，非常荣幸能与你共同站在浪潮之巅，一起踏上精彩绝伦的图像处理编程之旅，探索和学习这座近几年来蓬勃发展的图像处理与计算机视觉的大宝藏。

正所谓，授之以鱼，不如授之以渔。作为 OpenCV 入门的开篇，此部分旨在让你快速认识和了解新版本 OpenCV，并提供了一些指导大家跳出书本桎梏，依靠官方文档自学的思路，比如对 OpenCV 官方例程的引导与赏析、手把手教你阅读和编译 OpenCV 总计 44 万多行的源代码。这让你不仅能快速入门新版 OpenCV，更能学到在书本之外找到新的、进阶的学习 OpenCV 资源和代码的方法。

通过这部分的学习，相信可以对新版本 OpenCV 有一个全面而立体的认识。

第一部分包含如下三个章节：

- 第 1 章　邂逅 OpenCV
- 第 2 章　启程前的认知准备
- 第 3 章　HighGUI 图形用户界面初步

第 1 章
邂逅 OpenCV

导读

本章中，你将学到：

- 什么是计算机视觉
- 什么是 OpenCV
- 计算机视觉与 OpenCV 的联系
- OpenCV 的起源发展
- OpenCV 的应用领域
- 对 OpenCV、OpenAL、OpenGL 的辨析
- OpenCV 的基本架构分析
- OpenCV 的下载、安装与配置
- 通过几个程序快速上手 OpenCV
- 如何进行 OpenCV 视频操作
- 如何用 OpenCV 调用摄像头

1.1 OpenCV 周边概念认知

1.1.1 图像处理、计算机视觉与 OpenCV

图像处理（Image Processing)是用计算机对图像进行分析，以达到所需结果的技术，又称影像处理。图像处理技术一般包括图像压缩，增强和复原，匹配、描述和识别 3 个部分。图像处理一般指数字图像处理（Digital Image Processing）。其中，数字图像是指用工业相机、摄像机、扫描仪等设备经过拍摄得到的一个大的二维数组。该数组的元素称为像素，其值称为灰度值。而数字图像处理是通过计算机对图像进行去除噪声、增强、复原、分割、提取特征等处理的方法和技术。

计算机视觉（Computer Vision）是一门研究如何使机器“看”的科学，具体地说，就是是指用摄影机和电脑代替人眼对目标进行识别、跟踪和测量等机器视觉，并进一步做图形处理，用电脑处理使之成为更适合人眼观察或传送给仪器检测的图像的一门学科。作为一门科学学科，计算机视觉研究相关的理论和技术，试图建立能够从图像或者多维数据中获取“信息”的人工智能系统。因为感知可以看做是从感官信号中提取信息，所以计算机视觉也可以看做是研究如何使人工系统从图像或多维数据中“感知”的科学。

图像处理和计算机视觉的区别在于：图像处理侧重于“处理”图像——如增强，还原，去噪，分割，等等；而计算机视觉重点在于使用计算机（也许是可移动式的）来模拟人的视觉，因此模拟才是计算机视觉领域的最终目标。

而 OpenCV（Open Source Computer Vision Library），是一个基于开源发行的跨平台计算机视觉库，它实现了图像处理和计算机视觉方面的很多通用算法，已经成为了计算机视觉领域最有力的研究工具之一。

1.1.2 OpenCV 概述

OpenCV 的全称是 Open Source Computer Vision Library，直译就是“开源计算机视觉库”。取代表开源的单词“Open”、“Computer”的首字母“C”以及“Vision”的首字母“V”，组合命名为“OpenCV”。

OpenCV 于 1999 年由 Intel 建立，如今由 Willow Garage 提供支持。它是一个基于开源发行的跨平台计算机视觉库，可以运行在 Linux、Windows、Mac OS、Android、iOS、Maemo、FreeBSD、OpenBSD 等操作系统上。OpenCV 由一系列 C 函数和 C++类构成，轻量且高效。强大的 OpenCV 除了用 C/C++语言进行开发和使用之外，还支持使用 C#、Ch、Ruby 等编程语言，同时提供了对 Python、Ruby、MATLAB 等语言的接口，实现了图像处理和计算机视觉方面的很多通用算法。图 1.1 所示为 OpenCV 的 Logo 图形。

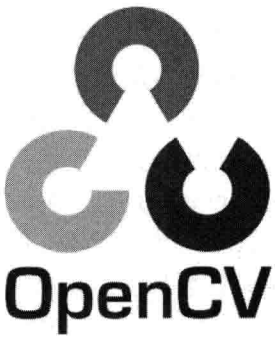


图 1.1 OpenCV Logo

OpenCV 的设计目标是执行速度尽量快，主要关注实时应用。它采用优化的C/C++代码编写，能够充分利用多核处理器的优势，其主要目标是构建一个简单易用的计算机视觉框架，以帮助开发人员更便捷地设计更复杂的计算机视觉相关应用程序。

OpenCV 由一系列 C 函数和 C++类构成，拥有包括 500 多个 C 函数的跨平台的中高层 API。它不依赖于其他的外部库——但如果有需要，也可以使用某些外部库。OpenCV 覆盖了计算机视觉的许多应用领域，如工厂产品检测、医学成像、信息安全、用户界面、摄像机标定、立体视觉和机器人等。因为计算机视觉和机器学习密切相关，所以 OpenCV 还提供 MLL（Machine Learning Library）机器学习库。该机器学习库主要用于统计方面的模式识别和聚类（clustering）。MLL 除了用在视觉相关的任务中，还可以方便地应用于其他机器学习场合。

OpenCV 官方主页：http://opencv.org

OpenCV Github 主页：https://github.com/Itseez/opencv

OpenCV 开发版 Wiki 主页：http://code.opencv.org

1.1.3 起源及发展

OpenCV 项目最早由 Intel 公司于 1999 年启动，旨在促进 CPU 密集型应用。为了达到这一目的，Intel 启动了多个项目，包括实时光线追踪和三维显示墙。

在 Intel 的性能库团队的帮助下，OpenCV 实现了一些核心代码和算法，并发给 Intel 俄罗斯的库团队。因此，OpenCV 的诞生，是在与软件性能库团队的合作下，发源于 Intel 的研究中心，并在俄罗斯得到实现和优化。

在开始时，OpenCV 有以下三大目标，这三大目标也说明了 OpenCV 的初衷。

- 为基本的视觉应用提供开放且优化的源代码，以促进视觉研究的发展，从而有效地避免“闭门造车”。
- 通过提供一个通用的架构来传播视觉知识，开发者可以在这个架构上继续开展工作，所以代码应该是非常易读且可改写的。
- OpenCV 库采用的协议不要求商业产品继续开放代码，这使得可移植的、

性能被优化的代码可以自由获取，可以促进基于视觉的商业应用的发展。

而如今看来，OpenCV 出现的目的是提供一个可普遍适用的计算机视觉库。OpenCV 的第一个预览版本于 2000 年在 IEEE Conference on Computer Vision and Pattern Recognition 公开，并且后续提供了 5 个测试版本。1.0 版本于 2006 年发布。OpenCV 的第二个主要版本是 2009 年 10 月的 OpenCV 2.0。该版本的主要更新包括 C++接口，更容易、更安全的模式，新的函数，以及对现有实现代码的优化（特别是多核心方面）。现在，约定每 6 个月就会有一个官方 OpenCV 版本发布，并且是由某商业公司赞助的独立小组进行开发。在 2012 年 8 月，对 OpenCV 的支持由一个非营利性组织（OpenCV.org）来提供，并保留了一个开发者网站和用户网站。值得一提的是，现在它也集成了对 CUDA 的支持。

2014 年 8 月 21 日，随着 OpenCV3 时代的第一个版本 OpenCV3.0 Alpha 在官网首页现身并提供下载，这个强大的计算机视觉库正式迎来了全新的纪元。

本书的示例程序最初都是以 OpenCV 2.4.9（2014 年 4 月 15 日面世）版本为开发环境的，书稿初版也是基于 OpenCV2.4.9 而写。在书稿写作和修订过程中，恰逢 OpenCV3.0 Alpha（2014 年 8 月 21 日）和 OpenCV3 Beta（2014 年 11 月 11 日）的发布，所以本书在审校和修订过程中（2014 年 11 月 29 日），决定站在浪潮之巅，以 OpenCV3 为主，加入 OpenCV3 的诸多特性，让这本书可以同时胜任 OpenCV2 和 OpenCV3 两个版本教材的角色。这也是为什么本书会有 OpenCV2 和 OpenCV3 两个独立版本的示例程序的原因。其中，OpenCV2 版的示例程序原则上支持 OpenCV 2 系列的所有版本的编译运行；OpenCV3 版的示例程序原则上支持目前已经发布的 OpenCV3 全版本。在书本正文贴出的代码中，以 OpenCV3 代码为主角，且将 OpenCV2 和 OpenCV3 代码有明显区别的地方进行了对比书写，方便大家学习。

1.1.4 应用概述

自从 OpenCV 在 1999 年 1 月发布 beta 版本开始，它就被广泛应用于许多领域、产品和研究成果，具体包括卫星地图和电子地图的拼接、扫描图像对齐、医学图像去噪（消噪或滤波）、图像中的物体分析、安全和入侵检测系统、自动监视和安全系统，以及制造业中的产品质量检测系统、摄像机标定、军事应用、无人飞行器、无人汽车和无人水下机器人。此外，还可以将视觉识别技术用在声谱图上，用 OpenCV 进行声音和音乐识别。

OpenCV 提供的视觉处理算法非常丰富。由于它部分以高效的 C 语言编写，加上其开源的特性，处理得当，不需要添加新的外部支持也可以完整地编译链接生成执行程序，所以很多研究者用它来做算法的移植。OpenCV 的代码经过适当改写可以正常地运行在 DSP 系统和单片机系统中，这种移植在高等院校中经常作为相关专业本科生的毕业设计或者研究生、博士生的课题选题。

OpenCV 可用于解决如下领域的问题：

- 人机交互
- 物体识别
- 图像分区
- 人脸识别
- 动作识别
- 运动跟踪
- 机器人

1.2 OpenCV 基本架构分析

初学 OpenCV 时，先了解一下 OpenCV 的整体模块架构，再重点学习和突破自己感兴趣的部分，就会有得心应手，一览众山小的学习体验。基于这个理念，笔者决定将此节的内容放在本书的首章，让大家高屋建瓴，在学习之初就能把握住新版 OpenCV 的脉络。

在此需要提示一下，若想和笔者一起在计算机中进入到 OpenCV 的文件夹目录，可以先跳到 1.3 节，先学习如何进行 OpenCV 的安装和配置，然后再反过来看本节的内容。至于 OpenCV 组件结构的研究方法，我们不妨管中窥豹，通过 OpenCV 安装路径下 include 目录里面头文件的分类存放，来一窥 OpenCV 这些年迅猛发展起来的庞杂组件架构。

进入到…\opencv\build\include 目录，可以看到有 opencv 和 opencv2 这两个文件夹。显然，opencv 这个文件夹里面包含着旧版的头文件，而 opencv2 这个文件夹里面包含着具有时代意义的新版 OpenCV2 系列的头文件。

在 opencv 这个文件夹里面，也就是…\opencv\build\include\opencv 目录下，可以看到如下各种头文件。这里面大概就是 OpenCV 1.0 最核心的，而且保留下来的内容的头文件，如图 1.2 所示。我们可以把它们整体理解为一个大的组件。

cv.h	2013/12/20 星期五 ...	C/C++ Header	4 KB
cv.hpp	2013/12/20 星期五 ...	C/C++ Header	3 KB
cvaux.h	2013/12/20 星期五 ...	C/C++ Header	3 KB
cvaux.hpp	2013/12/20 星期五 ...	C/C++ Header	3 KB
cvwimage.h	2013/12/20 星期五 ...	C/C++ Header	3 KB
cxcore.h	2013/12/20 星期五 ...	C/C++ Header	3 KB
cxcore.hpp	2013/12/20 星期五 ...	C/C++ Header	3 KB
cxeigen.hpp	2013/12/20 星期五 ...	C/C++ Header	3 KB
cxmisc.h	2013/12/20 星期五 ...	C/C++ Header	1 KB
highgui.h	2013/12/20 星期五 ...	C/C++ Header	3 KB
ml.h	2013/12/20 星期五 ...	C/C++ Header	3 KB

图 1.2 OpenCV 1.0 头文件

下面再来看看我们重点关注的 opencv2 文件夹，在…\opencv\build\include\opencv2 目录下，可以看到类似如图 1.3 所示的文件夹。

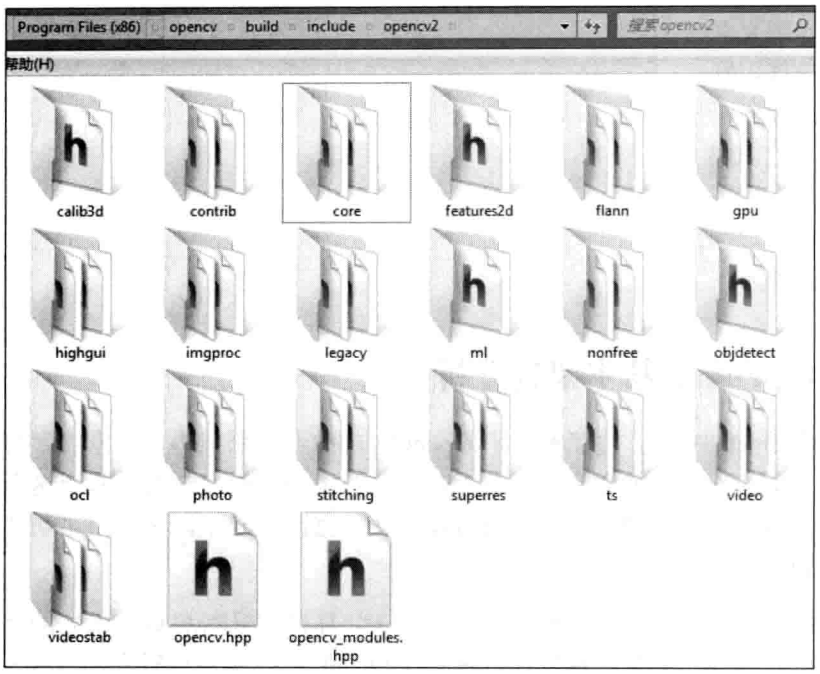


图 1.3 新版 OpenCV 的头文件（以 2.4.9 为例）

若仔细观察，会发现上述文件夹中有个名为 opencv_modules.hpp 的 hpp 文件，里面存放的是 OpenCV2 中与新模块构造相关的说明代码，打开可以发现其定义的是 OpenCV2 所有组件的宏，具体如下。

```
#define HAVE_OPENCV_CALIB3D
#define HAVE_OPENCV_CONTRIB
#define HAVE_OPENCV_CORE
#define HAVE_OPENCV_FEATURES2D
#define HAVE_OPENCV_FLANN
#define HAVE_OPENCV_GPU
#define HAVE_OPENCV_HIGHGUI
#define HAVE_OPENCV_IMGPROC
#define HAVE_OPENCV_LEGACY
#define HAVE_OPENCV_ML
#define HAVE_OPENCV_NONFREE
#define HAVE_OPENCV_OBJDETECT
#define HAVE_OPENCV_OCL
#define HAVE_OPENCV_PHOTO
#define HAVE_OPENCV_STITCHING
#define HAVE_OPENCV_SUPERRES
#define HAVE_OPENCV_TS
#define HAVE_OPENCV_VIDEO
#define HAVE_OPENCV_VIDEOSTAB
```

下面就是 OpenCV 的所有模块，按照宏定义的顺序依次介绍。

（1）【calib3d】——Calibration（校准）和 3D 这两个词的组合缩写。这个模块主要是相机校准和三维重建相关的内容，包括基本的多视角几何算法、单个立体摄像头标定、物体姿态估计、立体相似性算法、3D 信息的重建等。

（2）【contrib】——Contributed/Experimental Stuf 的缩写。该模块包含了一些最近添加的不太稳定的可选功能，不用去多管。新增了新型人脸识别、立体匹配、人工视网膜模型等技术。

（3）【core】——核心功能模块，包含如下内容：

- OpenCV 基本数据结构
- 动态数据结构
- 绘图函数
- 数组操作相关函数
- 辅助功能与系统函数和宏
- 与 OpenGL 的互操作

（4）【imgproc】——Image 和 Process 这两个单词的缩写组合，图像处理模块。包含如下内容：

- 线性和非线性的图像滤波
- 图像的几何变换
- 其他（Miscellaneous）图像转换
- 直方图相关
- 结构分析和形状描述
- 运动分析和对象跟踪
- 特征检测
- 目标检测等内容

（5）【features2d】 ——也就是 Features2D，即 2D 功能框架，包含如下内容：

- 特征检测和描述
- 特征检测器（Feature Detectors）通用接口
- 描述符提取器（Descriptor Extractors）通用接口
- 描述符匹配器（Descriptor Matchers）通用接口
- 通用描述符（Generic Descriptor）匹配器通用接口
- 关键点绘制函数和匹配功能绘制函数

（6）【flann】—— Fast Library for Approximate Nearest Neighbors，高维的近似近邻快速搜索算法库，包含以下两个部分：

- 快速近似最近邻搜索
- 聚类

（7）【gpu】——运用 GPU 加速的计算机视觉模块。

（8）【highgui】——高层 GUI 图形用户界面，包含媒体的输入输出、视频捕捉、图像和视频的编码解码、图形交互界面的接口等内容。

（9）【legacy】——一些已经废弃的代码库，保留下来作为向下兼容，包含如下内容：

- 运动分析
- 期望最大化
- 直方图
- 平面细分（C API）
- 特征检测和描述（Feature Detection and Description）
- 描述符提取器（Descriptor Extractors）的通用接口
- 通用描述符（Generic Descriptor Matchers）的常用接口
- 匹配器

（10）【ml】——Machine Learning，机器学习模块，基本上是统计模型和分类算法，包含如下内容：

- 统计模型（Statistical Models）
- 一般贝叶斯分类器（Normal Bayes Classifier）
- K-近邻（K-Nearest Neighbors）
- 支持向量机（Support Vector Machines）
- 决策树（Decision Trees）
- 提升（Boosting）
- 梯度提高树（Gradient Boosted Trees）
- 随机树（Random Trees）
- 超随机树（Extremely randomized trees）
- 期望最大化（Expectation Maximization）
- 神经网络（Neural Networks）
- MLData

（11）【nonfree】——一些具有专利的算法模块，包含特征检测和 GPU 相关的内容。最好不要商用。

（12）【objdetect】——目标检测模块，包含 Cascade Classification（级联分类）和 Latent SVM 这两个部分。

（13）【ocl】——OpenCL-accelerated Computer Vision，运用 OpenCL 加速的计算机视觉组件模块。

（14）【photo】——Computational Photography，包含图像修复和图像去噪两部分

（15）【stitching】——images stitching，图像拼接模块，包含如下部分：

- 拼接流水线
- 特点寻找和匹配图像
- 估计旋转
- 自动校准
- 图片歪斜
- 接缝估测
- 曝光补偿

- 图片混合

（16）【superres】——SuperResolution，超分辨率技术的相关功能模块。

（17）【ts】——OpenCV 测试相关代码，不用去管。

（18）【video】——视频分析组件，该模块包括运动估计、背景分离、对象跟踪等视频处理相关内容。

（19）【Videostab】——Video stabilization，视频稳定相关的组件，官方文档中没有多做介绍，不用管它。

看到这里，相信大家已经对 OpenCV 的模块架构设计有了一定的认识。OpenCV 其实就是这么多模块作为代码容器组合起来的一个 SDK（Software Development Kit，软件开发工具包）而已，并不繁杂，也不稀奇。

1.3 OpenCV3 带来了什么

2009 年 10 月 01 日，OpenCV 2.0 发布，这标志着革命性的 OpenCV2 时代的来临。OpenCV 2 带来了全新的 C++接口，将 OpenCV 的能力无限放大。在 2.0 的时代，OpenCV 增加了新的平台支持，包括 iOS 和 Andriod，通过 CUDA 和 OpenCL 实现了 GPU 加速，为 Python 和 Java 用户提供了接口，基于 Github 和 Buildbot 构建了充满艺术感的持续集成系统，所以才有了被全世界的很多公司和学校所采用的稳定易用的 OpenCV 2.4.x。

2014 年 8 月 21 日，OpenCV 3.0 Alpha 发布，宣告着 OpenCV3 时代的登场。官方更新日志中提到，OpenCV 在 3.0 中改变了项目架构的方式，所以 3.0 时代不会有像 2.0 时代一样激进的尝试，只会有足够稳定的改进。且不说更多眼花缭乱的新特性，项目架构的改变是 OpenCV3 最为重大的革新之处。让我们在 1.3.1 节中一起进行了解。

1.3.1 项目架构的改变

通过 1.2 节《OpenCV 基本架构分析》的介绍，我们可以发现 OpenCV 是一个相对于整体的项目，各个模块都是以整体的形式构建然后组合在一起的。然而，随着功能的增加，OpenCV 主体集成了各种各样的功能模块，变得越来越臃肿。

而 3.0 的出现，就是为了给日益发福的 OpenCV 减肥，因为 OpenCV3 决定像其他大项目一样，抛弃整体架构，使用内核+插件的架构形式。

在 GitHub 中，除了存放着正式版 OpenCV 的主仓库和新增加的“opencv_extra”仓库以外，OpenCV 3.0 中还添加了一个名为 opencv_contrib 的全新仓库，这个新仓库中有很多让人兴奋的功能：包括脸部识别和文本探测，以及文本识别、新的边缘检测器、充满艺术感的图像修复、深度地图处理、新的光流和追踪算法等。

OpenCV 主仓库的地址：https://github.com/itseez/opencv
opencv_extra 仓库地址：https://github.com/itseez/opencv_extra
opencv_contrib 仓库地址：http://github.com/itseez/opencv_contrib

正式版 opencv 与 opencv_contrib 之间的区别如下：

- 两者都由 OpenCV 官方开发团队持续集成系统维护，虽然目前 opencv_contrib 仓库中的代码测试并没有完成，很多功能不稳定。
- 主体的 opencv 在 GitHub 中由 Itseez 提供，其有着非常稳定的 API 以及少部分的创新。
- opencv_contrib 仓库是大多数实验性代码放置的地方，一些 API 可能会有改变，一直会欢迎广大开发者们贡献新的精彩算法。
- opencv_contrib 中的这些额外模块可以在 CMake 中用 OPENCV_EXTRA_MODULES_PATH=/modules 传递给 CMake 文件，和 OpenCV3 主体中的代码一起编译和运行。
- opencv_contrib 的文档是自动生成的，可以在 http://docs.opencv.org/master/ 中找到，并会在随后的版本中更加完善。

还有一些 OpenCV3 的细节由于比较小众，在此不再赘述，大家可以在 OpenCV3 的官网中查看到更多的更新信息。

1.3.2 将 OpenCV2 代码升级到 OpenCV3 报错时的一些策略

由于 OpenCV3 的主体部分只是在 OpenCV2 的基础上进行小幅度的更改，所以我们会发现不少 OpenCV2 下开发的程序仍然可以在 OpenCV3 中正常编译运行。

然而，有一些版本升级导致的改变则会让我们在 OpenCV2 下开发的程序在 OpenCV3 中报错。本节，让我们将这些知识点聚在一起做一个列举，供大家在把基于 OpenCV2 的代码升级到 OpenCV3 时查阅。

1.【问题一】由于宏名称的变更照成的“未声明的标识符”系列问题

有时候，遇到此类问题加入一句 “#include<cv.h>”便可以让 OpenCV3 也能认识一些“CV_”前缀的宏，将问题解决掉。因为有不少 OpenCV1 的宏依然在 OpenCV3 的 cv.h 头文件中有定义。

症状：在 OpenCV3 的环境下运行 OpenCV2 中写的程序，报“error C2065: ‘CV_WINDOW_AUTOSIZE’：未声明的标识符”系列错误。

分析：OpenCV3 取消了 OpenCV1 中残留的“CV_”式的宏前缀命名规范，对这些 CV_前缀的宏使用了新的命名规范。这里有几种情况要分类讨论。

（1）情况 1：直接去掉“CV_”前缀

在这种情况下，比较典型的有如下一些函数。

1)namedWindow 函数中，例如将 CV_WINDOW_AUTOSIZE 改为 WINDOW_AUTOSIZE。

2)threshold 函数中，例如将 CV_THRESH_BINARY 改为 THRESH_BINARY。

3）line 函数等一系列绘图函数中，CV_FILLED 改为 FILLED。

4）remap 函数中，CV_INTER_LINEAR 改为 INTER_LINEAR。

5）在鼠标操作函数 SetMouseCallback 中，将 CV_EVENT_LBUTTONUP 改为 EVENT_LBUTTONUP 、 CV_EVENT_LBUTTONDOWN 改为 EVENT_LBUTTONDOWN、CV_EVENT_FLAG_LBUTTON 改为 EVENT_FLAG_LBUTTON、CV_EVENT_MOUSEMOVE 改为 EVENT_MOUSEMOVE。

7）HoughCircles 函数中，将 CV_HOUGH_GRADIENT 改成 HOUGH_GRADIENT。

8）inpaint 函数中，将 CV_INPAINT_TELEA 改为 INPAINT_TELEA、CV_INPAINT_NS 写作 INPAINT_NS。

9）matchTemplate 函数中，将 CV_TM_SQDIFF 改为 TM_SQDIFF、CV_TM_SQDIFF_NORMED 改为 TM_SQDIFF_NORMED。

10）在 imwrite 函数相关使用中，将 CV_IMWRITE_PNG_COMPRESSION 改成 IMWRITE_PNG_COMPRESSION。

11）设置摄像头尺寸的时候，将 CV_CAP_PROP_FRAME_WIDTH 改为 CAP_PROP_FRAME_WIDTH 、CV_CAP_PROP_FRAME_HEIGHT 改为 CAP_PROP_FRAME_HEIGHT。

（2）情况 2：需要用新的前缀替换

在这种情况下，比较典型的有如下一些函数。

1）line 函数等一系列绘图函数中，CV_AA 改为 LINE_AA

2）cvtColor 函数中颜色空间转换系的宏，全替换为“COLOR_”前缀，如 CV_BGR2HSV 改为 COLOR_BGR2HSV

3）normalize 函数中，将 CV_MINMAX 改为 NORM_MINMAX。

4）morphologyEx 函数中的宏，全部替换为“MORPH_”前缀，如 CV_MOP_OPEN 改为 MORPH_OPEN。

5）threshold 函数中的宏，全部替换为“THRESH_”前缀，如将 CV_THRESH_BINARY 改 THRESH_BINARY

（3）情况 3：需要在新的命名空间中使用宏

在这种情况下，比较典型的有如下一些函数。

1）TermCriteria 函数中，CV_TERMCRIT_EPS 改为 TermCriteria::EPS、CV_TERMCRIT_ITER 改为 TermCriteria::MAX_ITER

2）CascadeClassifier::detectMultiScale 函数中，CV_HAAR_SCALE_IMAGE 改为 CASCADE_SCALE_IMAGE。

2.【问题二】使用 vector 容器之时提示“error C2065: “vector”: 未声明的标识符”系列错误。

症状：在 OpenCV3 的环境下运行 OpenCV2 中写的程序，使用了 vector 容器，而未包含 STD 命名空间，于是便会报“error C2065: “vector”: 未声明的标识符“系列错误。

分析：OpenCV3 中并没有在头文件中使用标准程序库 std 的命名空间。所以遇到这个错误，在我们写的程序开头加上一句“using namespace std;”使用 C++的标准命名空间即可解决问题。

上述两个就是 OpenCV2 代码升级 OpenCV3 时最常会遇到的问题。接下来，列举一些 OpenCV3 升级时一些其他的改动细节，以结束此节。

3.其他一些细节问题的解决方案

1）features2d. hpp 头文件路径的更改。将#include <opencv2/nonfree/features2d.hpp>

改为#include <opencv2/features2d.hpp>

2）core.hpp 头文件路径更改。将#include <opencv2/core/core.hpp>改为#include <opencv2/core.hpp>

3）用 format 进行格式化输出时，将 format(r,"python")改成 format(r, Formatter::FMT_PYTHON)。

4）定义尺寸时，将 cvSize(-1,-1)改为 Size(-1,-1)。

5）在表示颜色时，将 CV_RGB 改为 Scalar。

在 OpenCV 的官方开发网站的更新日志页面，可以看到历代 OpenCV 官方详细的改动日志：http://code.opencv.org/projects/opencv/wiki/ ChangeLog

1.4 OpenCV 的下载、安装与配置

在本节开头需要说明的是，OpenCV 开发环境的搭建方法从最初的 1.0 版到当前的最新版本，基本上没有大的改变。在笔者初次编写本章内容的时候，OpenCV 的最新版本是 2014 年 4 月 25 日发行的 2.4.9 版，并且此版本的配置过程比较典型，因此本书的开发环境搭建就将这个版本作为主要示例。这套方法可以看作 OpenCV 的万金油配置方法，无论以后 OpenCV 更新到什么版本的，或者是需要配置以前的任何一版，原则上都可以使用本节介绍的配置方法来进行开发环境的搭建。

1.4.1 预准备：下载和安装集成开发环境

既然是进行 OpenCV 编程开发的学习，一个顺手的集成开发环境自然少不了。

本书的内容以及示例程序都是默认以微软的 Visual Studio 作为开发环境的。关于 Visual Studio，目前有众多的版本，从最经典的 Visual Studio 6.0，历经 Visual Studio 2002、Visual Studio 2005、Visual Studio 2010、Visual Studio 2012，直到当前的最新版本 Visual Studio 2015。目前业界使用最多，用起来最顺手也支持率最高的，当属 Visual Studio 2010。本书所有的示例程序都是在 Visual Studio 2010 旗舰版的环境下开发编写的，虽然使用其他版本 Visual Studio 也同样可行，但大家在学习此书时，最好也同样使用 Visual Studio 2010，因为各个版本之间还是有些细微差别的。需要注意，不少旧版本 Visual Studio 可能和新版本的 OpenCV 不兼容，若实在不想使用 Visual Studio 2010，推荐大家使用 Visual Studio 2010 以上的版本，如 Visual Studio 2012、Visual Studio 2013、Visual Studio 2015 等，因为选用低版本的 Visual Studio 可能会在开发过程中出现一些难以解决的问题。

1.4.2 第一步：下载和安装 OpenCV SDK

本书配置 OpenCV 时，将以最为典型的 OpenCV2.4.9 作为参考，只要弄懂该配置过程，以后可以轻松完成当前发行的任意一个 OpenCV 版本的配置任务，而区别仅仅是下文中的第五步——链接库的配置，把对应的 249 改成自己对应的 OpenCV 版本号即可。

OpenCV3 的配置方法。OpenCV3 官方对配置过程已经简化了，将 OpenCV2 中的近 20 个 lib 文件浓缩成了 opencv_ts300.lib、opencv_world300.lib（Release 版本）这两个。所以区别也是在第五步链接库的配置处，将 OpenCV2 中的那一大串 lib 对应地改为 opencv_ts300.lib、opencv_world300.lib（Release 版本）或 opencv_ts300d.lib、opencv_world300d.lib（Debug 版本）即可

在官网 http://opencv.org/上找到 OpenCV Windows 版下载下来，如图 1.4 所示。

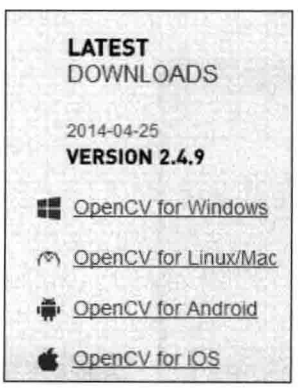


图 1.4 OpenCV 安装包下载区域

下载完后得到文件“opencv 2.x.x.exe”，如图 1.5 所示。下载完成后，便可以开始进行 OpenCV 的安装和配置。与其说是安装，不如叫解压更加合适，因为我们下载的.exe 安装文件就是一个自解压程序而已。双击这个文件后程序会提示我们解压到某个地方，推荐放到…\Program Files\下，比如 D:\Program Files（因为

OpenCV 项目文件打包的时候，根目录就是 opencv，所以我们不需要额外新建一个名为 opencv 的文件夹），然后在弹出的对话框中点击【Extract】按钮。如图 1.6 所示。

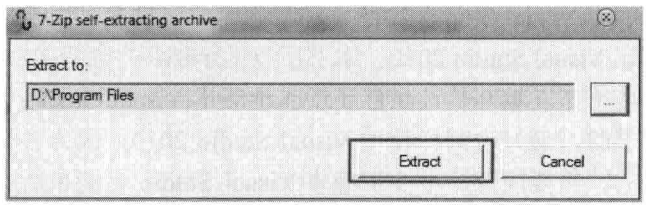


图 1.5 OpenCV 2.4.9 安装程序　　图 1.6 提取 OpenCV 文件

稍后，OpenCV 约几个 GB 的文件就解压到了我们上文指定的目录下，本步骤到此结束。在此进行简单的文件内容的说明：解压完成后，会在指定的路径下生成一个名为 opencv 的文件夹，它包含了两个子文件夹，分别名为 build 和 sources。其中，build 文件夹中是支持 OpenCV 使用的相关文件，而 sources 中为 OpenCV 的源代码及相关文件。如果仅仅是希望将 OpenCV 写的程序在电脑中跑起来并想要尽量节省硬盘空间，就只需要 build 里面的文件内容，sources 文件夹完全可以删掉。但是需要注意的是，OpenCV 官方示例集以及说明文档，也就是 samples 文件夹里面的示例程序和 doc 文件夹中的说明文档，都是存放在 sources 文件夹里面的。所以当然是推荐保留 sources 文件夹，若真要删除，还请三思而后行。

1.4.3 第二步：配置环境变量

配置方法如下。

【计算机】→【（右键）属性】→【高级系统设置】→【高级（标签）】→【环境变量】→（双击）系统变量中的 PATH→在变量值里面添加相应的路径。注意：是“添加”相应的值，和之前已有的值用分号“;”来分隔，而不是删掉之前已有的变量值。如图 1.7 和图 1.8 所示。

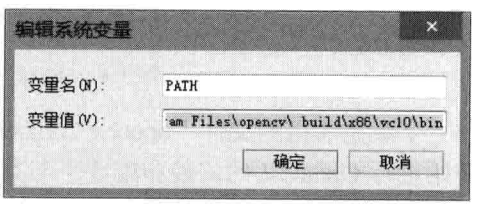


图 1.7 选取 PATH 环境变量　　图 1.8 修改 PATH 环境变量的值

对于 32 位系统，就添加“;... opencv\build\x86\vc10\bin”，和之前的就有的环境变量用英文的分号“;”进行分隔。而对于 64 位系统，可以两个都添加上——“;... opencv\build\x86\vc10\bin”和“... opencv\build\x64\vc10\bin”。这样，到时候才可以在编译器 Win32 和 X64 中来回切换。

例如，笔者的就是 D:\Program Files\opencv\build\x64\vc10\bin;D:\Program Files\opencv\build\x86\vc10\bin 这两个路径。

也有朋友在笔者博客留言道“亲测 64 位系统也只需添加..... opencv\build\x86\vc10\bin 即可”，这是只针对 32 位编译器版本的做法，也是可行的。

变量值实际为 bin 文件夹的路径。D 表示 OpenCV 安装于 D 盘；X64 表示运行系统环境位 64 位系统，若安装于 32 位系统，应为 X86；vc10 表示编译环境为 Microsoft Visual Studio 2010。变量添加完成后最好注销系统，然后才会生效。

1.4.4 第三步：工程包含（include）目录的配置

笔者在写作此节内容之前，发现互联网上相当多的 OpenCV 配置博文都写道“每次新建工程都要重新配置”，其实是不用这样麻烦的。下面且看如何经过一次的配置就可以“一劳永逸”。

首先是在 Visual Studio 里面新建一个控制台应用程序，最好是勾选空项目。考虑到不少看此节的读者们很少接触 Visual Studio，那么在这里将过程详细截取出来，方便大家快速掌握配置方法。

1. 打开 Visual Studio，新建一个项目。可以单击【起始页】中的【新建项目】，如图 1.9 所示。或者依次单击菜单栏中的【文件】→【新建】→【项目】，如图 1.10 所示。

图 1.9 打开新建项目对话框的方法一

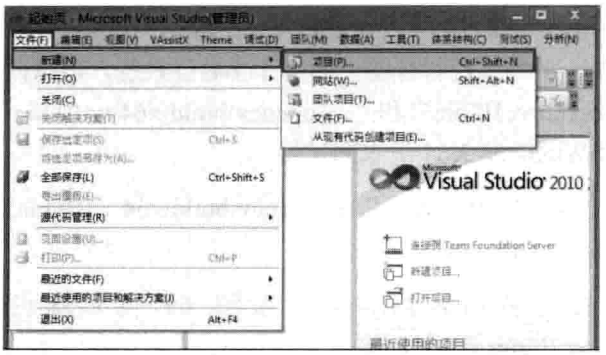


图 1.10 打开新建项目对话框的方法二

然后选择新建【Win32 控制台应用程序】，进行命名，比如 test1，然后选好路径，单击【确定】，如图 1.11 所示。

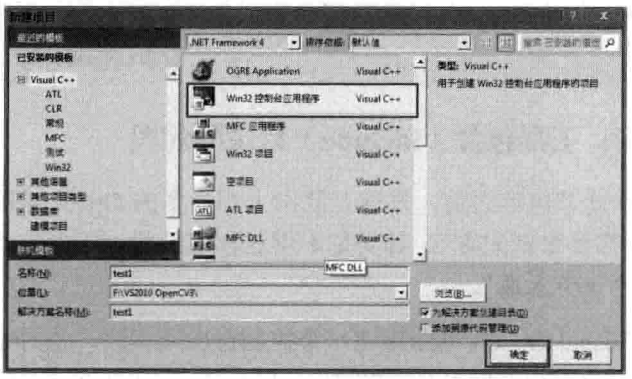


图 1.11 新建 Win32 控制台应用程序

2．进入【Win32 应用程序向导】页面后，单击“下一步”，或者“应用程序设置”，会跳转到同样的应用程序设置页面。如图 1.12 所示。

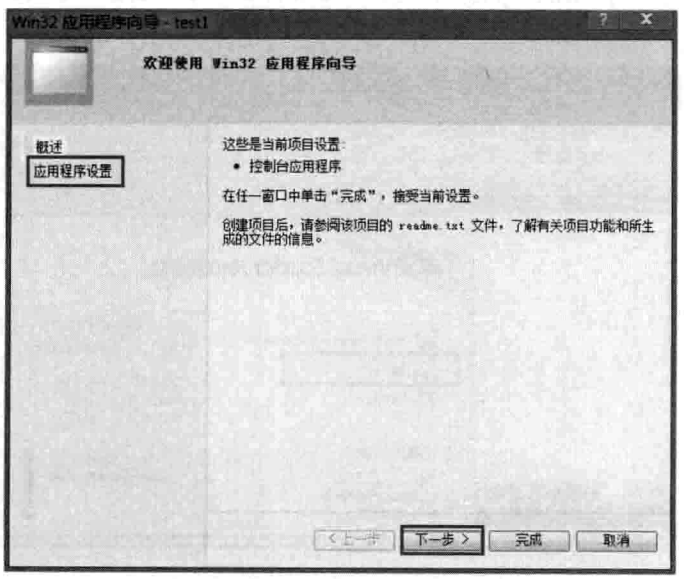


图 1.12 Win32 应用程序向导页面

3．在应用程序设置页面，勾选【空项目】。如图 1.13 所示。

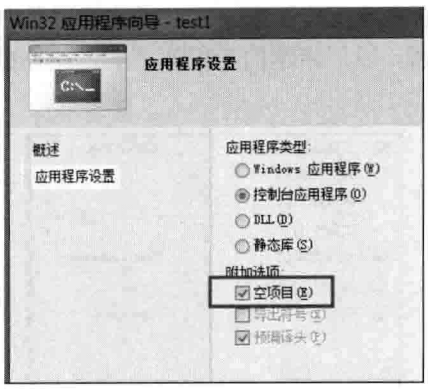


图 1.13　勾选空项目

4．在解决方案资源管理器的【源文件】处右键单击→添加→新建项，准备在工程中新建一个 cpp 源文件。如图 1.14 所示。

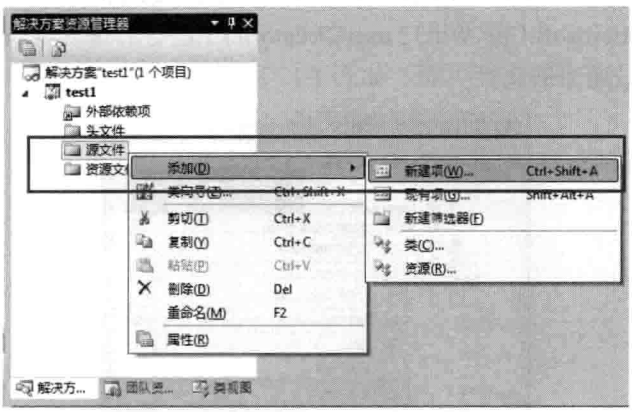


图 1.14　在源文件中添加新建项

5．选定【C++文件（cpp）】，进行命名，比如“main”，然后单击【添加】，一个新的.cpp 文件就添加到了工程中。如图 1.15 所示。

图 1.15　在源文件中添加新建项

6. 在菜单栏里单击【视图】→【属性管理器】(如图 1.16 所示)，就会在 Visual Studio 中多出一个属性管理器工作区来。而在属性管理器中进行一次配置，就相当于进行了通用的配置过程，以后新建的工程就不用再额外进行配置。这就是我们这种配置方法的核心思想。

图 1.16 属性管理器

7. 在新出现的“属性管理器”工作区中，展开【Debug|Win32】文件夹，对文件夹中的【Microsoft.Cpp.Win32.userDirectories】进行右键属性操作，或者双击，即可打开工程最通用的属性页面。如图 1.17 所示。

图 1.17 打开工程最通用的属性页面

8. 打开属性页面后，首先在【通用属性】→【VC++目录】→【包含目录】中添加以下三个目录。如图 1.18、1.19 所示。

- D:\Program Files\opencv\build\include
- D:\Program Files\opencv\build\include\opencv
- D:\Program Files\opencv\build\include\opencv2

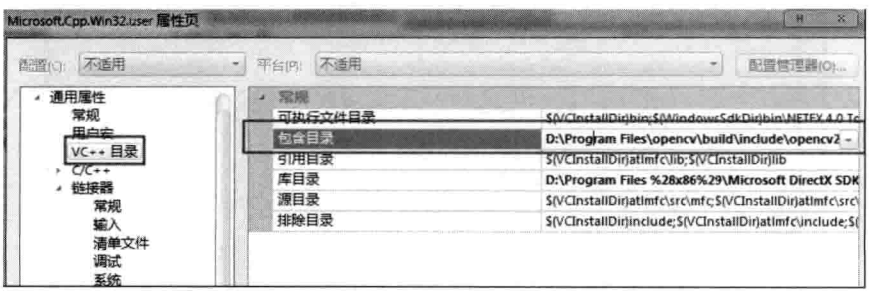


图 1.18 添加包含目录（1）

图 1.19 添加包含目录（2）

当然，这是之前把 OpenCV 解压到 D:\Program Files\下的情况。实际的路径还要看读者自己把 OpenCV 解压到了哪个目录下，根据具体情况来调节。

1.4.5 第四步：工程库（lib）目录的配置

第四步和第三步差不多，在“属性管理器”工作区中，单击【项目】→Debug|Win32→Microsoft.Cpp.Win32.userDirectories（右键属性，或者双击）打开属性页面。然后在【通用属性】→【VC++目录】→【库目录】中，添加 D:\Program Files\opencv\build\x86\vc10\lib 这个路径。如图 1.20 所示。

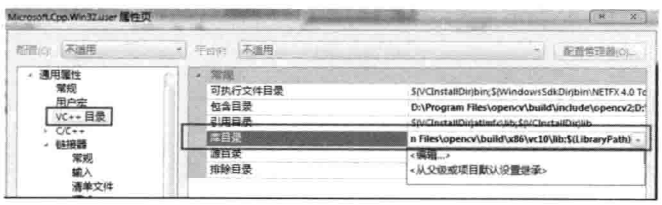


图 1.20 添加库目录

最好不要手动用键盘输入这里给出的路径，而是去预览里面指定出来，这样会准确得多。因为路径不小心输入出错，可能在编译时出现“LINK : fatal error LNK1104: 无法打开文件“opencv_calib3d249.lib”之类的错误提示。

这里选择 x86 还是 x64 是一个常常令人困惑的问题。当然，对于 32 位操作系统，铁定就是选 x86 了。如果是 64 位操作系统，不要直接想当然地选择 x64，正确的理解是这样的：

不管是 32 位还是 64 位操作系统，只需要考虑用 Win32 编译器还是 x64 编译器。其实配置选择什么跟 64 位还是 32 位系统没有直接的关系，而是在于在编译程序时是使用哪个编译器。编译器选的是 Win32，就用 x86；编译器选的是 x64，就用 x64。不过一般情况下，都是用的 Win32 的 x86 编译器。所以，无论 32 还是 64 位操作系统，配置文件最好都选择 x86 版的。

另外，这里的 vc10 表示 vs2010，如果是其他版本的 visual studio，稍微要微调一下。其中：vc8 等同于 Visual Studio 2005，vc9 等同于 Visual Studio 2008，vc10 等同于 Visual Studio 2010，vc11 等同于 Visual Studio 2012，vc12 等同于 Visual Studio 2013，后续版本以此类推。

1.4.6 第五步：链接库的配置

同样在“属性管理器”工作区中单击【项目】→【Debug|Win32】→【Microsoft.Cpp.Win32. userDirectories】(右键单击然后单击属性，或者直接双击)，即可打开属性页面。然后依次单击【通用属性】→【链接器】→【输入】→【附加的依赖项】，如图 1.21 所示。

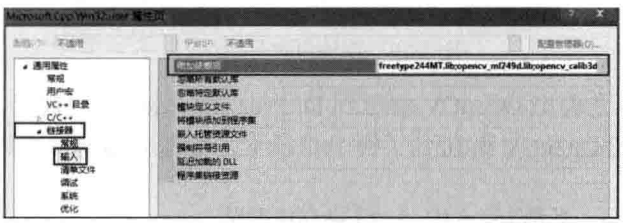


图 1.21 添加附加依赖项

这里给出两个版本的例子，对于其他版本读者可以自行变通，将 248 或 249 改成对应的 OpenCV 版本即可。

对于【OpenCV 2.4.8】，添加以下 248 版本的 lib(lib 顺序是：19 个带 d 的 debug 版 lib 写在前面，19 个不带 d 的 release 版 lib 写在后面，即优先支持 debug 模式的编译运行)。

```
opencv_ml248d.lib
opencv_calib3d248d.lib
opencv_contrib248d.lib
opencv_core248d.lib
opencv_features2d248d.lib
opencv_flann248d.lib
opencv_gpu248d.lib
opencv_highgui248d.lib
opencv_imgproc248d.lib
opencv_legacy248d.lib
opencv_objdetect248d.lib
opencv_ts248d.lib
opencv_video248d.lib
opencv_nonfree248d.lib
opencv_ocl248d.lib
opencv_photo248d.lib
opencv_stitching248d.lib
opencv_superres248d.lib
opencv_videostab248d.lib

opencv_objdetect248.lib
opencv_ts248.lib
```

```
opencv_video248.lib
opencv_nonfree248.lib
opencv_ocl248.lib
opencv_photo248.lib
opencv_stitching248.lib
opencv_superres248.lib
opencv_videostab248.lib
opencv_calib3d248.lib
opencv_contrib248.lib
opencv_core248.lib
opencv_features2d248.lib
opencv_flann248.lib
opencv_gpu248.lib
opencv_highgui248.lib
opencv_imgproc248.lib
opencv_legacy248.lib
opencv_ml248.lib
```

对于【OpenCV 2.4.9】，添加如下 249 版本的 lib（这样的 lib 顺序是：19 个带 d 的 debug 版的 lib 写在前面，19 个不带 d 的 release 版的 lib 写在后面），如图 1.22 所示。

图 1.22 添加附加依赖项

```
opencv_ml249d.lib
opencv_calib3d249d.lib
opencv_contrib249d.lib
opencv_core249d.lib
opencv_features2d249d.lib
opencv_flann249d.lib
opencv_gpu249d.lib
opencv_highgui249d.lib
opencv_imgproc249d.lib
opencv_legacy249d.lib
opencv_objdetect249d.lib
opencv_ts249d.lib
opencv_video249d.lib
opencv_nonfree249d.lib
opencv_ocl249d.lib
```

```
opencv_photo249d.lib
opencv_stitching249d.lib
opencv_superres249d.lib
opencv_videostab249d.lib

opencv_objdetect249.lib
opencv_ts249.lib
opencv_video249.lib
opencv_nonfree249.lib
opencv_ocl249.lib
opencv_photo249.lib
opencv_stitching249.lib
opencv_superres249.lib
opencv_videostab249.lib
opencv_calib3d249.lib
opencv_contrib249.lib
opencv_core249.lib
opencv_features2d249.lib
opencv_flann249.lib
opencv_gpu249.lib
opencv_highgui249.lib
opencv_imgproc249.lib
opencv_legacy249.lib
opencv_ml249.lib
```

上述内容无须手动用键盘输入，请使用 Google 或百度搜索“OpenCV2.4.9 配置”关键字（将 2.4.9 改成自己的 OpenCV 版本），会找到海量的 OpenCV 配置博文，然后将博主在文章中贴出的的这些 lib 复制出来即可。

对于 OpenCV3，将这些 lib 对应地改为 opencv_ts300d.lib、opencv_world300d.lib（Debug 版本）或 opencv_ts300.lib、opencv_world300.lib（Release 版本）即可。且 OpenCV3 似乎终于修复了下文中介绍的 OpenCV2 时代让众多使用者苦恼的，由于带 d 和不带 d 的 lib 的区别而导致报错的问题。测试使用将带 d 的 Debug 版本的 lib 作为【属性】→【链接器】→【输入】→【附加的依赖项】中的链接库，程序在 OpenCV3 中可以同时支持 Debug 和 Release 版本代码的正确运行，不会再像 OpenCV2 一样报字符串异常照成的内存错误。

需要注意的是，所粘贴内容即为之前我们解压的 OpencV 目录 D:\opencv\build\x86\vc10\lib 下所有 lib 库文件的名字。其中的 249 代表 OpenCV 版本为 2.4.9，若是其他版本则在这里要进行相应的更改。比如说 2.4.6 版的 OpenCV，那么这里的 opencv_calib3d249d.lib 就要改成 opencv_calib3d246d.lib。

Debug 文件库名有 d 结尾，Release 没有，如 opencv_ts248d.lib（debug 版本的 lib）和 opencv_ts248.lib（release 版本的 lib）。

按照以上方式进行配置，也许会出现 debug 下可以运行但是 release 下不能运行的情况（因为字符串读取问题引起的诸如图片载入不了、报指针越界、内存错

误等等），这算是 OpenCV 自 2.4.1 以来的一个 bug。这里会优先选择在依赖项中，写在前面的那一类 lib 作为默认可以支持的调试方式。若带 d 的 debug 系的十几个 lib 写在前面，那么就默认支持 debug 模式下可以载入图片、识别字符串等操作。反之，若不带 d 的 release 系的 lib 写在前面，那么就默认支持 release 模式下可以载入图片、识别字符串等操作。

解决方案：上述方法在 debug 下可以正常载入图片（因为写在前面的是众多带 d 的 lib），若想在 release 模式下也支持图片载入等操作，可以在工程的 release 模式下，将不带 d 的 lib 添加到【项目】→【属性】→【配置属性】→【链接器】→【输入】→【附加的依赖项】下即可。注意这样打开的是当前工程的属性页，在这里将 release 版（即不带 d）的相关 lib 添加进去，就表示让当前的这个工程的 release 模式，有了这些 lib 作为依赖。类似错误如图 1.23 所示。

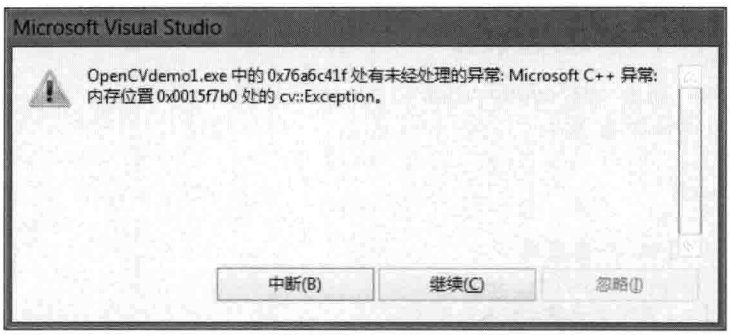


图 1.23 “未经处理的异常”系列错误报错图示

1.4.7 第六步：在 Windows 文件夹下加入 OpenCV 动态链接库

需要注意的是：如果环境变量配置得准确，且配置之后经过重启，就没有进行这步配置的必要了。即做完上面第五步的配置，重启一次，就可以直接跳到第七步进行测试，看是否配置成功。当然，若想不重启而马上查看是否配置成功，就需要进行这步的配置。

在运行基于 OpenCV 的程序的时候，往往会得到与图 1.24 所示界面类似的结果。

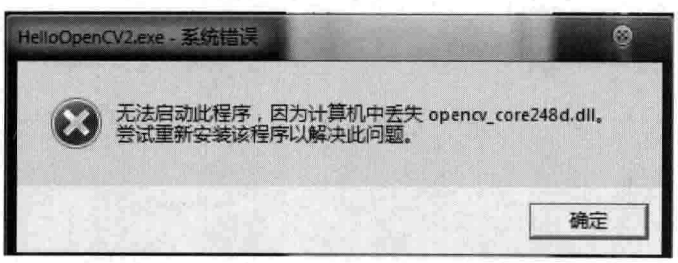


图 1.24 “计算机中丢失 XX.dll”系列错误报错图示

这种问题最简单的方法，是把相关的 dll 都复制到 Windows 操作系统的目录下。如果 Windows 安装在 C 盘，那么 32 位系统的放置路径就是 C:\Windows\System32，而 64 位系统，放置路径为 C:\Windows\SysWOW64。

按照之前我们的 OpenCV 的存放环境，这些 dll 存放在 D:\Program Files\opencv\

build\x86\vc10\bin 目录下。找到这个目录，用【Alt+A】全选，【Alt+C】复制，然后转到 C:\Windows\SysWOW64 下，用【Alt+V】粘贴，这步就完成了。

OpenCV 的配置过程就是如上 6 步，接下来我们进行最终的实例测试来检测是否配置无误。

1.4.8 第七步：最终测试

此次配置的工程文件已在本书配套代码包中给出，是第一个示例程序，名为“HelloOpenCV”。大家若不想手动打出上述代码，或者配置过程中遇到问题想查看配置细节，都可以进行参考。

测试过程为用 OpenCV 载入并显示一张图片到窗口中。我们用 Visual Studio 新建一个空项目控制台应用程序，随意命名，比如叫做“HelloOpenCV”，然后新建一个 cpp 文件，输入如下代码。

```
#include <opencv2/opencv.hpp>
using namespace cv;

int main()
{
    // 【1】读入一张图片
    Mat img=imread("1.jpg");
    // 【2】在窗口中显示载入的图片
    imshow("【载入的图片】",img);
    // 【3】等待 6000 ms 后窗口自动关闭
    waitKey(6000);
}
```

当然，上述代码中带双斜杠“//”的文字是注释，输不输入不影响测试结果。我们放置一张名为 1.jpg 的图片到工程目录中（和 cpp 源文件同一路径），然后单击 Visual Studio 中的【运行】按钮。如果配置成功，就不会报错，从而得到一个控制台窗口和一张图片窗口，分别如图 1.25 和图 1.26 所示。

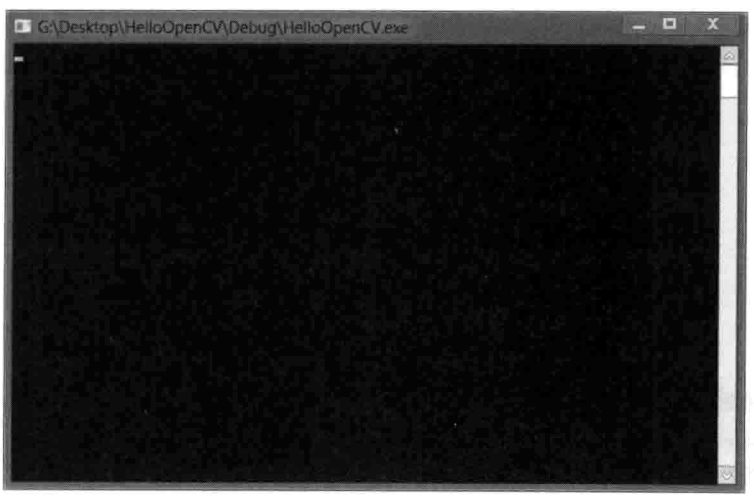


图 1.25 程序运行后得到的控制台窗口

图 1.26 程序运行后得到的图片窗口

1.4.9 可能遇到的问题和解决方案

生活不可能是一帆风顺的，我们的配置过程也是如此。笔者在几次的配置过程中，遇到了如下几种典型问题，相信各位读者也可能会遇到，本小节就在这里集中列举这些问题，希望能对大家有所帮助。

1.【问题 1】找不到 core.h

出现这个问题也许是因为 include 的时候粗心大意了。比如你的版本是 2.4.6，在这个版本下，opencv 根文件夹下面会有个 include，但配置的时候如果包含的是它就错了，正确的应该填 build 文件夹中的那个 include。

2.【问题 2】无法解析的外部命令

这个问题其实上文已经有过解释：不管是 32 位还是 64 位操作系统，只需要考虑用 win32 编译器还是 X64 编译器。

其实配置选择什么跟 64 位还是 32 位系统没有直接的关系，而是在于你在编译程序时使用的是哪个编译器：编译器是 win32，就用 x86；编译器是 X64，就用 X64。不过一般情况下，都是用的 win32 的 X86 编译器。所以，无论 32 还是 64 位操作系统，配置文件最好都选择 x86 版的。

3.【问题 3】形如--error LNK2005:xxx 已经在 msvcrtd.lib(MSVCR90D.dll)中定义

出现这个问题，把静态库不包含就行了。

4.【问题 4】应用程序无法正常启动 0xc000007b

这个是 Lib 包含的问题。可能你同时包含了 X86 和 X64 的，可能包含出错了。而且对于 windows7/ 8 64 位，dll 要放在和 System32 文件夹同级的 SysWOW64 文

件夹中。

5.【问题 5】明明图片路径是对的，却载入不进图片，提示指针越界，有未经处理的异常

类似错误如图 1.22，这表示在内存中图片没有读取成功，导致指针越界异常。这时要再次检查以确保图片的后缀名和路径与代码中的一致。一般情况下出现此问题算是 OpenCV 的一个 bug，是工程属性里面关于带 d 和不带 d 的 lib 文件的附加依赖项的问题。

就算配置成功，若想在 debug 和 release 模式下同时可以运行，还需手动在工程属性里面加上一些 lib。当得到这样的错误时，可以把调试方式改一改，将 debug 和 release 互换，如图 1.27 所示。

图 1.27 更改调试方式

或者打开当前工程的属性页（注意不是通用属性页），debug 或者 release 哪个报错，就把对应的带 d 或不带 d 的 lib 添加到【（当前）工程属性】→【链接器】→【输入】→【附加的依赖项】下即可。

6.【问题 6】无法打开文件“opencv_ml249d.lib”系列的错误

fatal error LNK1104：无法打开文件“opencv_ml249d.lib”是一个常见的错误。这个错误主要是因为包含的库目录中，和包含的附加依赖项不能相互对应照成的。也许是“opencv_ml249d.lib“多加了一个空格，成了”opencv_ml249d．lib“，就会报错。遇到这个问题，检查以下三个方面。

（1）检查第四步”4.工程库（lib）目录的配置“库目录中的路径是否准确。

（2）检查第五步”5.链接库的配置“中“附加依赖项”的格式有没有问题，有没有多空格，版本号 248、249 是否正确，有没有多一个空格或少一个点。

（3）第二步环境变量的配置是否准确。

另外的解决方案是：依次进入【项目】→【属性管理器】→【Debug|Win32-> Microsoft.Cpp. Win32.userDirectories】中的【属性页面】→【链接器】→【常规】，在里面的【附加库目录】中加入相应的 lib 文件目录。

1.5 快速上手 OpenCV 图像处理

上一节中，我们已经配置完成 OpenCV，而本节将带领大家接触几个 OpenCV 图像处理相关的程序，看看 OpenCV 用简洁的代码能实现哪些有趣的图像效果，让大家快速地爱上 OpenCV，提高学习兴趣。

1.5.1 第一个程序：图像显示

本小节介绍的程序即为上一节 OpenCV 配置中的测试用例，为了演示的连贯性而再次在此处展开讲解。

在新版本 OpenCV 中，图像显示过程非常简单，只需用 imread 函数载入到新版本的图像存储数据结构 Mat 类中，然后用 imshow 函数显示即可。这两个函数在第 3 章会有详细讲解。打开 Visual Studio，新建一个控制台项目，然后新建一个 cpp 文件，在其中添加如下代码：

```
#include <opencv2/opencv.hpp>
using namespace cv;  //包含 cv 命名空间

void main()
{
    Mat srcImage = imread("1.jpg"); //载入图像
    imshow("【原始图】",srcImage);//显示图像
    waitKey(0);//等待任意按键按下
}
```

其中，#include <opencv2/opencv.hpp> 一句为 OpenCV 头文件的包含；而 using namespace cv;一句为命名空间的包含，这在第 3 章的第 1 节会有详细的讲解。之后，先载入图像，然后显示图像，并且调用 waitKey 函数等待按键按下，以便让图片窗口一直显示，直到有按键按下。

放置一张以网络上下载的图片，并改名为程序代码中指定的“1.jpg”，然后运行程序，可以得到如图 1.28 所示效果。

图 1.28 运行效果图

1.5.2 第二个程序：图像腐蚀

我们再来看如何用 OpenCV 实现最基本的形态学运算之一——腐蚀，即用图像中的暗色部分“腐蚀”掉图像中的高亮部分。图像形态学操作在后文有详细讲解，这里我们先一起了解一下：同样是新建控制台项目，新建 cpp 源文件，放置名为“1.jpg”的图片到工程目录下（和 cpp 源文件同一目录下）。

之后的示例程序，都默认采取类似的操作——在 Visual Studio 中新建项目，新建源文件，放置名为“1.jpg”的图片到 cpp 源文件同一目录下。特在此说明。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/highgui/highgui.hpp>//OpenCV  highgui 模块头文件
#include <opencv2/imgproc/imgproc.hpp>//OpenCV 图像处理头文件
using namespace cv; //包含 cv 命名空间

int main(  )// 控制台应用程序的入口函数，我们的程序从这里开始
{
    //载入原图
    Mat srcImage = imread("1.jpg");
    //显示原图
    imshow("【原图】腐蚀操作", srcImage);
    //进行腐蚀操作
    Mat element = getStructuringElement(MORPH_RECT, Size(15, 15));
    Mat dstImage;
    erode(srcImage, dstImage, element);
    //显示效果图
    imshow("【效果图】腐蚀操作", dstImage);
    waitKey(0);

    return 0;
}
```

程序首先依然是载入和显示一幅图像，然后定义一个 Mat 类型的变量来获得 getStructuringElement 函数的返回值，而 getStructuringElement 函数的返回值为指定形状和尺寸的结构元素（内核矩阵）。参数准备完毕，接着便可以调用 erode 函数进行图像腐蚀操作，最后调用 imshow 函数进行显示，用 waitKey 函数等待按键按下，以便能让窗口一直显示。

原图和程序运行效果图如图 1.29 和图 1.30 所示。

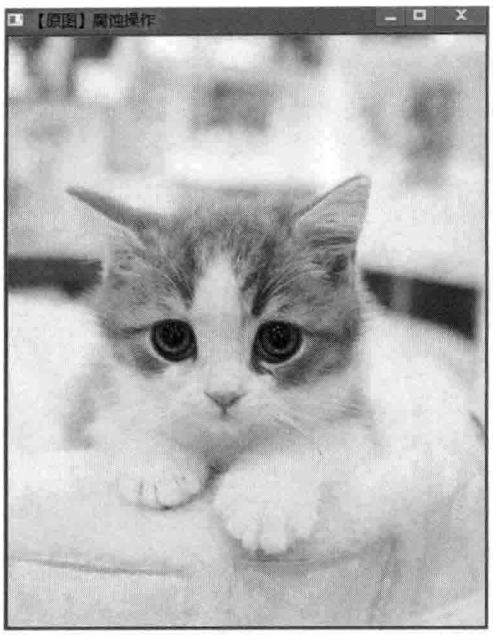


图 1.29 原始图

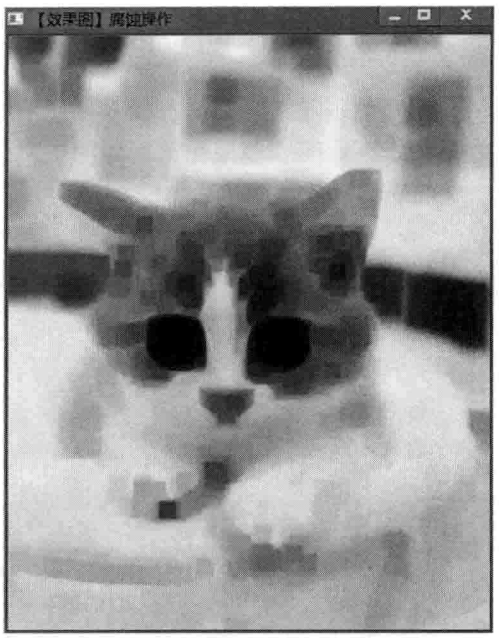


图 1.30 图像腐蚀效果图

1.5.3 第三个程序：图像模糊

接着让我们看看用 OpenCV 对图像进行均值滤波操作，模糊一幅图像的代码如何书写。主要使用进行均值滤波操作的 blur 函数，代码如下：

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;

//----------------------------【main()函数】----------------------------
//  描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------
int main()
{
        //【1】载入原始图
        Mat srcImage=imread("1.jpg");

        //【2】显示原始图
        imshow( "均值滤波【原图】", srcImage );

        //【3】进行均值滤波操作
        Mat dstImage;
        blur( srcImage, dstImage, Size(7, 7));

        //【4】显示效果图
        imshow( "均值滤波【效果图】" ,dstImage );
```

```
    waitKey( 0 );
}
```

程序代码非常好理解，载入并显示原始图后，调用一次 blur 函数，最后显示效果图。原始图和效果图分别如图 1.31 和 1.32 所示。

图 1.31 原始图

图 1.32 均值滤波效果图

1.5.4 第四个程序：canny 边缘检测

接着我们来看看如何用 OpenCV 进行 canny 边缘检测。载入图像，并将其转成灰度图，再用 blur 函数进行图像模糊以降噪，然后用 canny 函数进行边缘检测，最后进行显示。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include<opencv2/imgproc/imgproc.hpp>
using namespace cv;

//-----------------------------【main()函数】--------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //【0】载入原始图
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材
```

```
图
    imshow("【原始图】Canny 边缘检测", srcImage);      //显示原始图
    Mat dstImage,edge,grayImage;  //参数定义

    //【1】创建与 src 同类型和大小的矩阵(dst)
    dstImage.create( srcImage.size(), srcImage.type() );

    //【2】将原图像转换为灰度图像
    //此句代码的 OpenCV2 版为:
    //cvtColor( srcImage, grayImage, CV_BGR2GRAY );
    //此句代码的 OpenCV3 版为:
    cvtColor( srcImage, grayImage, COLOR_BGR2GRAY );

    //【3】先使用 3x3 内核来降噪
    blur( grayImage, edge, Size(3,3) );

    //【4】运行 Canny 算子
    Canny( edge, edge, 3, 9,3 );

    //【5】显示效果图
    imshow("【效果图】Canny 边缘检测", edge);

    waitKey(0);

    return 0;
}
```

原始图和效果图分别如图 1.33 和 1.34 所示。

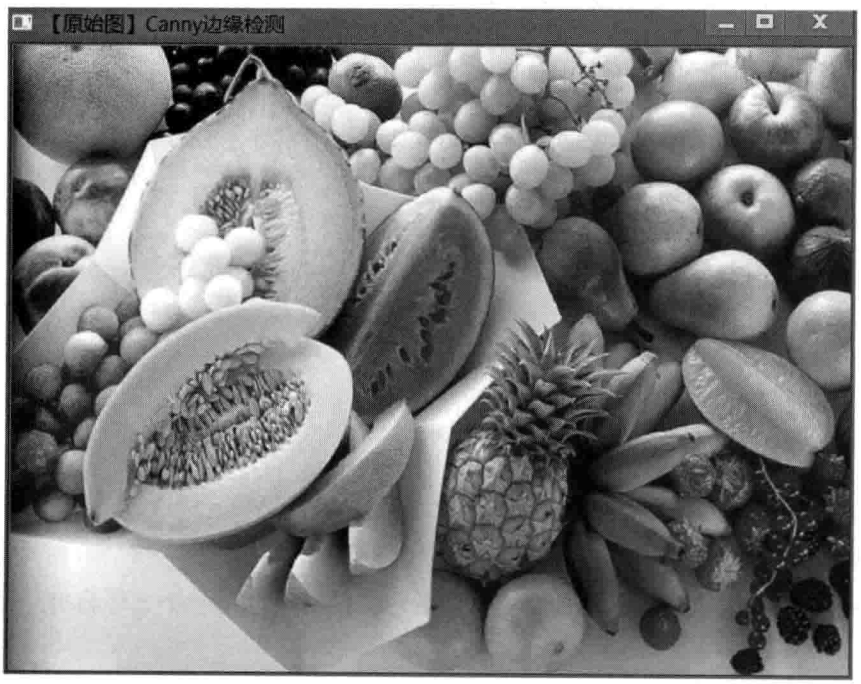


图 1.33 原始图

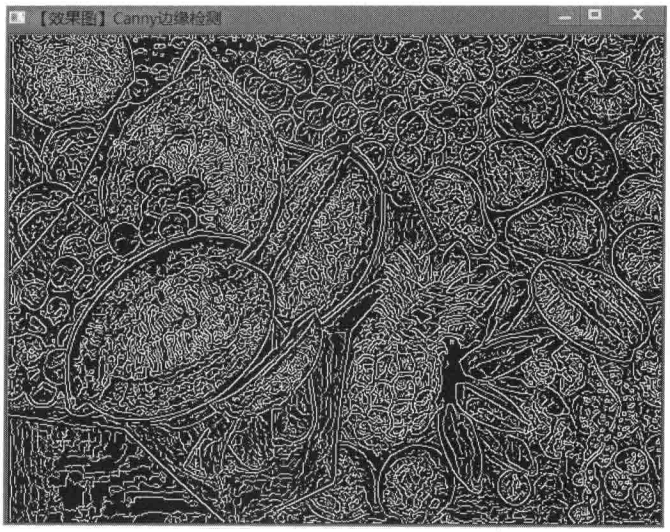


图 1.34 边缘检测效果图

1.6 OpenCV 视频操作基础

本节我们将学习如何利用 OpenCV 中的 VideoCapture 类，来对视频进行读取显示，以及调用摄像头。

VideoCapture 是 OpenCV 2.X 中新增的一个类，对应于之前 C 语言版本的 CvCapture 结构体。它提供了从摄像机或视频文件捕获视频的 C++接口，作用是从视频文件或从摄像头捕获视频并显示出来。

1.6.1 读取并播放视频

通过对 VideoCapture 类的分析，可以发现利用它读入视频的方法一般有如下两种。比如读入的视频为工程路径下名为“1.avi”的视频文件，那么这两种写法分别如下。

（1）先实例化再初始化：

```
VideoCapture capture;
capture.open("1.avi");
```

（2）在实例化的同时进行初始化：

```
VideoCapture capture("1.avi");
```

这两种写法的区别就如我们定义一个 int 类型的变量一样：“int a;a=1;”为先定义再初始化；“int a=1;”为在定义时初始化。

视频读入到 VideoCapture 类对象之后，紧接着可以用一个循环将每一帧显示出来，相关代码如下：

```
//循环显示每一帧
while(1)
```

```
{
    Mat frame;  //定义一个 Mat 变量，用于存储每一帧的图像
    capture>>frame;  //读取当前帧
    imshow("读取视频",frame);  //显示当前帧
    waitKey(30);  //延时 30ms
}
```

这段代码中，首先定义了一个 Mat 变量，用于存储每一帧的图像，接着读取当前帧到 Mat 变量中，然后调用 imshow 显示当前的这一帧图像，并用 waitKey 延时 30 毫秒，开始下一次循环。

将本节讲解的内容串联起来，就成了这一节颇具代表性且代码相当精简的视频的载入示例程序。在这里贴出其源码给大家参考：

```
#include <opencv2\opencv.hpp>
using namespace cv;

int main()
{
    //【1】读入视频
    VideoCapture capture("1.avi");

    //【2】循环显示每一帧
    while(1)
    {
        Mat frame;  //定义一个 Mat 变量，用于存储每一帧的图像
        capture>>frame;  //读取当前帧
        imshow("读取视频",frame);  //显示当前帧
        waitKey(30);  //延时 30ms
    }
    return 0;
}
```

在工程目录下，笔者准备了一个名为“1.avi”的视频文件放置于工程目录之下，程序运行截图如图 1.35 所示。

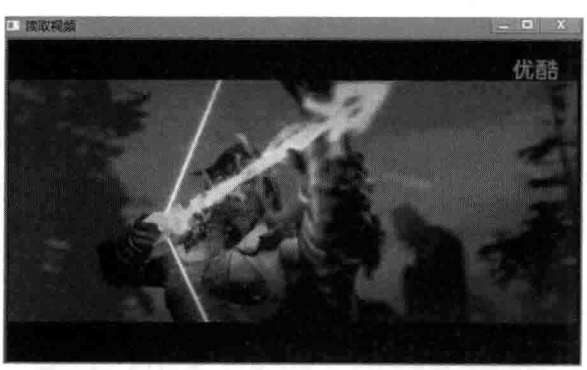


图 1.35 播放视频

1.6.2 调用摄像头采集图像

上一小节讲到了视频文件的读入，而如果我们要调用摄像头进行视频采集的

话，将代码 VideoCapture capture（“1.avi”）；中的“1.avi”换为 0 就可以了，表示调用摄像头而不是从文件中读取视频。对应于上文讲到的两种写法，即：

（1）先实例化再初始化

```
VideoCapture capture;
capture.open(0);
```

（2）在实例化的同时进行初始化

```
VideoCapture capture(0);
```

通过两种方式中的一种读入视频之后，循环将每一帧显示出来代码和之前 1.5.1 节讲到的内容完全相同，可以发现，利用 VideoCapture 类调用摄像头采集视频和从文件中读入视频的区别仅仅是在 VideoCapture 类对象初始化时指定的内容的区别，即，是指定文件名如“1.avi”，还是填一个 0 表示调用摄像头而已。接下来让我们一起看看用 OpenCV 调用摄像头的精简示例程序，并在代码中熟悉整个调用过程。

```
#include <opencv2/opencv.hpp>
using namespace cv;

int main()
{
    //【1】从摄像头读入视频
    VideoCapture capture(0);

    //【2】循环显示每一帧
    while(1)
    {
        Mat frame;  //定义一个 Mat 变量，用于存储每一帧的图像
        capture>>frame;  //读取当前帧
        imshow("读取视频",frame);  //显示当前帧
        waitKey(30);  //延时 30ms
    }
    return 0;
}
```

摄像头采集到的视频如图 1.36 所示。

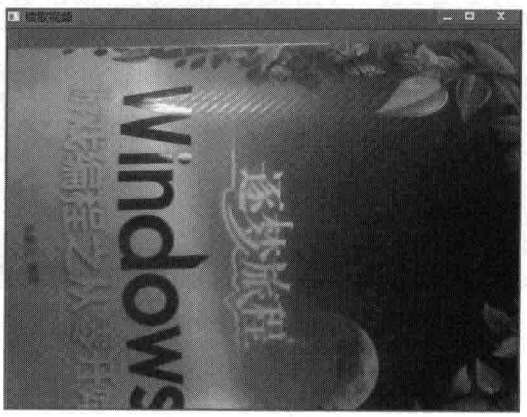


图 1.36 摄像头采集到的视频

另外，我们还可以用上面的摄像头调用示例程序来配合 canny 边缘检测，得到 canny 边缘检测并高斯模糊后的摄像头采集视频，源码如下：

```
#include "opencv2/opencv.hpp"
using namespace cv;

int main()
{
   //从摄像头读入视频
   VideoCapture capture(0);
   Mat edges;

   //循环显示每一帧
   while(1)
   {
        //【1】读入图像
        Mat frame;//定义一个 Mat 变量，用于存储每一帧的图像
        capture >> frame; //读取当前帧

        //【2】将原图像转换为灰度图像
        cvtColor(frame, edges, CV_BGR2GRAY);//转化 BGR 彩色图为灰度图

        //【3】使用 3x3 内核来降噪（2x3+1=7）
        blur(edges, edges, Size(7,7));//进行模糊

        //【4】进行 canny 边缘检测并显示
        Canny(edges, edges, 0, 30, 3);
        imshow("被 canny 后的视频", edges);//显示经过处理后的当前帧
        if(waitKey(30) >= 0) break;//延时 30ms
   }
   return 0;
}
```

运行效果图如图 1.37 所示。

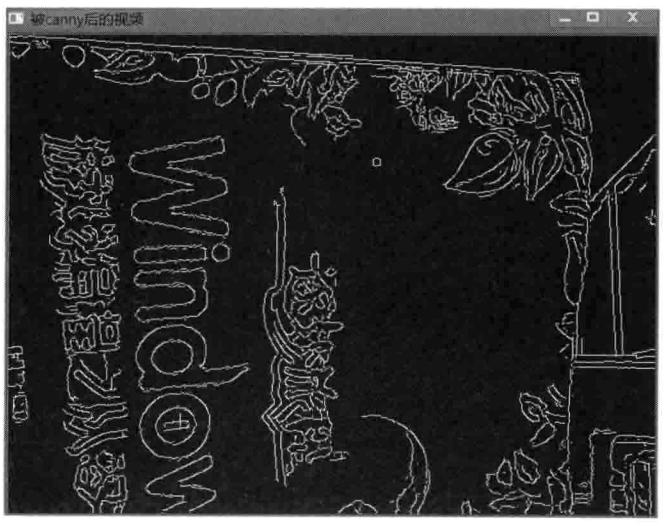


图 1.37 经过 canny 边缘检测后的视频

1.7 本章小结

相信通过本章的学习，大家已经对 OpenCV 有了一些初步的认知。本章中，我们不仅介绍了 OpenCV 的周边概念，还分析了其基本架构。然后重点讲解了 OpenCV 的下载、安装与配置过程。配置完成后，我们带领大家正式开始领略 OpenCV 的魅力，接触了四个 OpenCV 图像处理小程序。而章节最后，我们还学习了如何使用 OpenCV 操作视频和调用摄像头。内容可谓非常丰富，相信认真学完此章，会对接下来 OpenCV 的深入学习打下很好的基础。

本章示例程序清单

示例程序序号	程序说明	对应章节
1	OpenCV 环境配置的测试用例	1.3.8
2	快速上手 OpenCV 的第一个程序：图像显示	1.4.1
3	快速上手 OpenCV 的第二个程序：图像腐蚀	1.4.2
4	快速上手 OpenCV 的第三个程序：blur 图像模糊	1.4.3
5	快速上手 OpenCV 的第四个程序：canny 边缘检测	1.4.4
6	读取并播放视频	1.5.1
7	调用摄像头采集图像	1.5.2

可以根据每章小结或者书中提供的示例程序的序号，在本书配套示例程序包中快速查询到需要查看的工程源代码或者可运行.exe 程序。

第 2 章
启程前的认知准备

导读

本章中，你将学到：

- OpenCV 官方例程的引导学习与赏析
- 如何编译 OpenCV 的源代码
- 对“opencv.hpp”头文件的认知
- 本书的命名规范约定
- argc 与 argv 参数解惑
- 格式输出函数 printf()用法小议

本章内容可以进行跳跃性的阅读，阅读顺序由读者自行选择。其中 2.5、2.6 节内容为 C 语言基础薄弱的读者所准备，有一定的 C 语言基础的读者对这两节可以略读。

2.1 OpenCV 官方例程引导与赏析

OpenCV 作为一个在全球使用人数众多的计算机视觉库，其实官方已经准备了大量的示例程序，供广大初学者学习。而市面上绝大多数的 OpenCV 书籍教程和网络博文的第一手知识来源，就是这些官方提供的技术文档和示例程序。

本节旨在带领大家粗略了解 OpenCV 官方提供的示例程序，看看学好 OpenCV 之后，可以写出哪些炫酷的程序来。

经过第 1 章的学习，我们已经安装、下载并配置好了 OpenCV。在 OpenCV 安装目录下，可以找到 OpenCV 官方提供的示例代码。具体位于…\opencv\sources\samples\cpp 目录下。

如图 2.1 所示，若将 OpenCV 安装于 D:\Program Files(x86)\路径，则源码的存放目录应为 D:\Program Files (x86)\opencv\sources\samples\cpp。

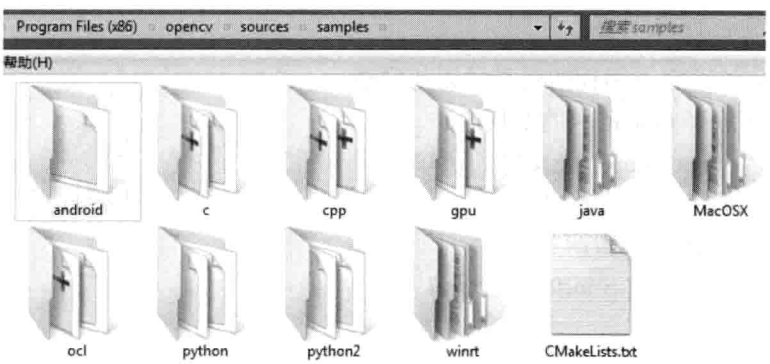


图 2.1 OpenCV 官方示例程序文件夹

通过观察文件名可以发现，OpenCV 官方提供了 Android、C、CPP、GPU、Java、Mac OS X、OCL、Python、Python2、Win RT 等众多版本的示例程序。不难理解，名为 c 的文件夹中存放着 OpenCV1.0 等旧版本的示例程序，而名为 cpp 的文件夹中存放着 OpenCV 2.X 等新版本的示例程序。

本书旨在对新版本 OpenCV 的讲解，自然主要专注于 cpp 文件夹内的众多示例程序。打开该文件夹，可以发现 100 余个 C++版本的 OpenCV 官方示例程序。在…\opencv\sources\samples\cpp\tutorial_code 路径下，存放着和官方教程配套的示例程序。其内容按 OpenCV 各组件模块而分类，非常适合学习，大家可以按需查询，分类学习，各个击破。

接下来，笔者将带着大家在此 100 余个示例程序中选择比较有特点的几个，得出运行效果。为了方便大家学习，以下的几个示例都和本书的其他主线示例程序一样，在配套代码包中提供了可以直接运行的执行文件与源工程。请在配套示例程序包中的第 2 章中找到第 8 到第 13 这 6 个配套示例程序（它们分别对应了 2.1.1 到 2.1.6 这 5 节的内容，2.1.5 节分为了两个工程），对其进行编译运行并查看结果。

2.1.1 彩色目标跟踪：Camshift

本小节讲解的例程为彩色目标跟踪，程序的用法是根据鼠标框选区域的色度光谱来进行摄像头读入的视频目标的跟踪。其主要采用 CamShift 算法，全称是“Continuously Adaptive Mean-SHIFT”，是对 MeanShift 算法的改进，被称为连续自适应的 MeanShift 算法。打开本书配套的第 8 个示例程序（或者是在…\opencv\sources\samples\cpp 目录下找到名为 camshiftdemo.cpp 的文件，自行修改部分代码。实际路径会因为 OpenCV 版本的不同略有差异），编译并运行，用鼠标在窗口中框选要跟踪的区域，便可以得到如图 2.2、2.3、2.4 所示运行截图。

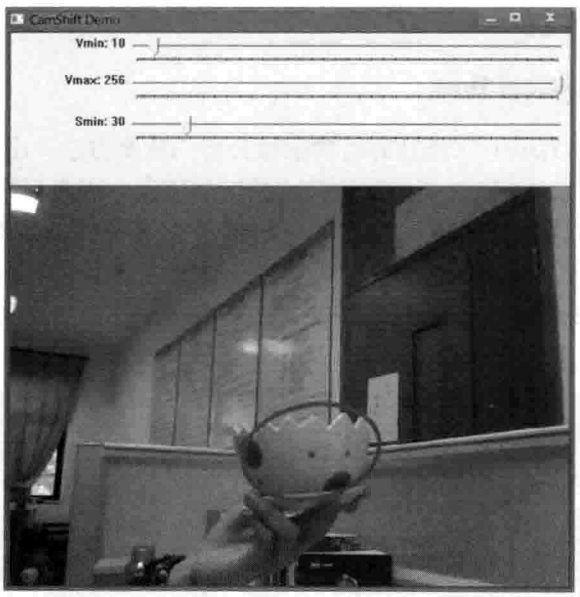


图 2.2 彩色目标追踪运行截图（1）

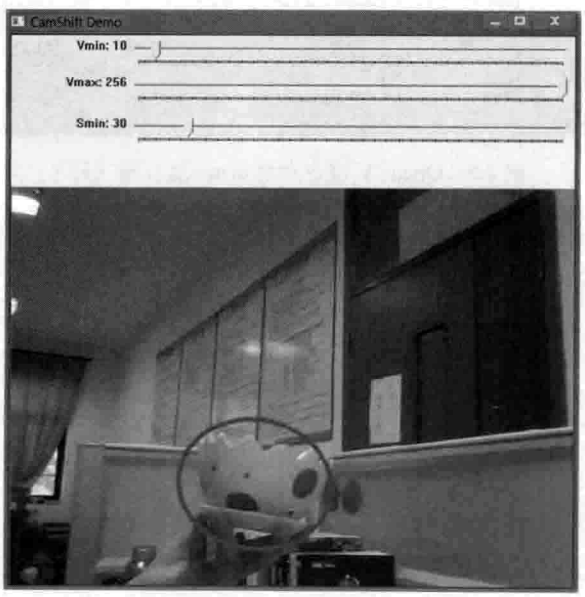


图 2.3 彩色目标追踪运行截图（2）

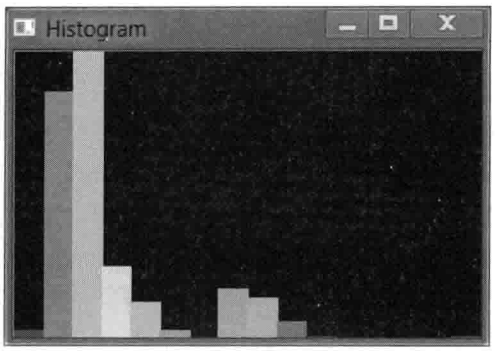


图 2.4 对应的直方图

2.1.2 光流：optical flow

光流（optical flow）法是目前运动图像分析的重要方法，由 Gibso 于 1950 年首先提出。光流用来指定时变图像中模式的运动速度，因为当物体在运动时，在图像上对应点的亮度模式也在运动。这种图像亮度模式的表观运动（apparent motion）就是光流。光流表达了图像的变化，由于它包含了目标运动的信息，因此可被观察者用来确定目标的运动情况。图 2.5 和 2.6 即为 OpenCV 官方为我们准备的示例程序的运行效果图。

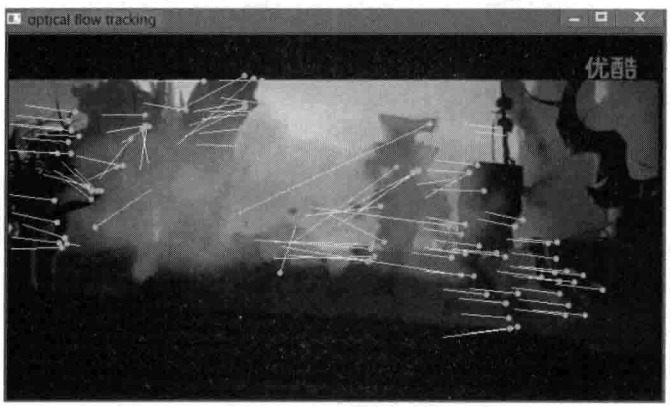


图 2.5 OpenCV 官方光流 demo 运行图（1）

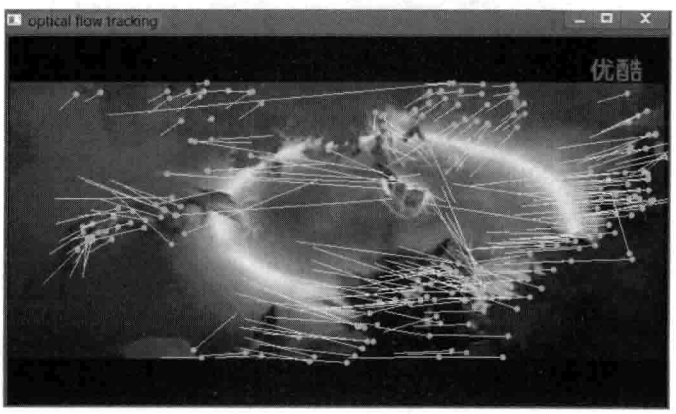


图 2.6 OpenCV 官方光流 demo 运行图（2）

2.1.3 点追踪：lkdemo

在…\opencv\sources\samples\cpp 目录下（实际路径会因为 OpenCV 版本的不同略有差异）的 lkdemo.cpp 文件中，存放着这样一个精彩的例程。程序运行后，会自动启用摄像头，这时按键盘上的“r”键来启动自动点追踪，便可以看到如图 2.7 所示的效果图 1。而我们在摄像头中移动物体，可以看到物体上的点随着物体一同移动（如图 2.8 所示），富含科技感且妙趣横生。

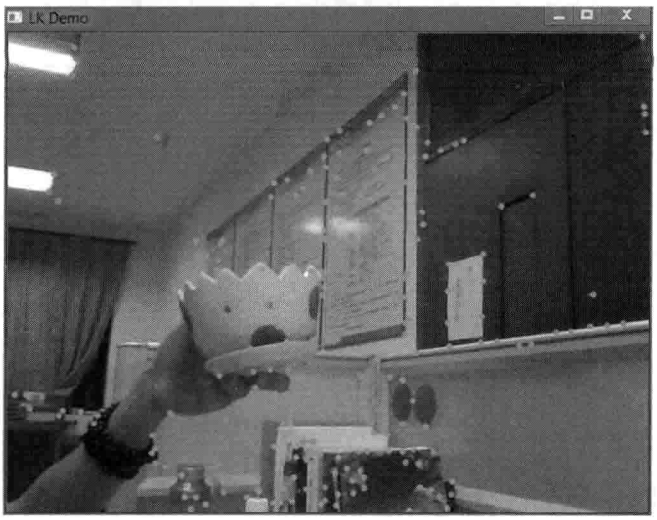


图 2.7 点追踪程序运行图（1）

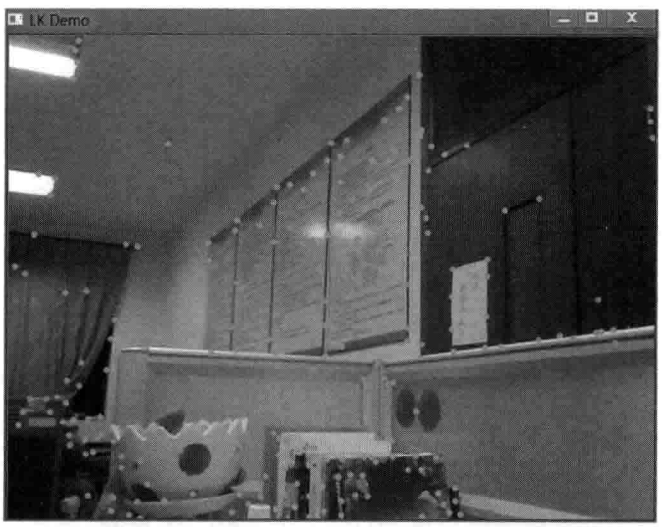


图 2.8 点追踪程序运行图（2）

2.1.4 人脸识别：objectDetection

人脸识别是图像处理与 OpenCV 非常重要的应用之一，OpenCV 官方专门有教程和代码讲解其实现方法。此示例程序就是使用 objdetect 模块检测摄像头视频流中的人脸，位于…\opencv\sources\samples\cpp\tutorial_code\objectDetection 路径

之下。需要额外注意的是，需要将“…\opencv\sources\data\haarcascades”路径下的“haarcascade_eye_tree_eyeglasses.xml”和“haarcascade_frontalface_alt.xml”文件复制到和源文件同一目录中，才能正确运行。运行程序，将自己的脸对准摄像头，或者放置一张照片对准摄像头任其捕获，便可以发现程序准确地识别出了人脸，并用彩色的圆将脸圈出。如图 2.9 所示。

图 2.9 人脸识别效果图

2.1.5 支持向量机引导

在 OpenCV 的机器学习模块中，官方为我们准备了两个示例程序。第一个程序是使用 CvSVM::train 函数训练一个 SVM 分类器，如图 2.10 所示。

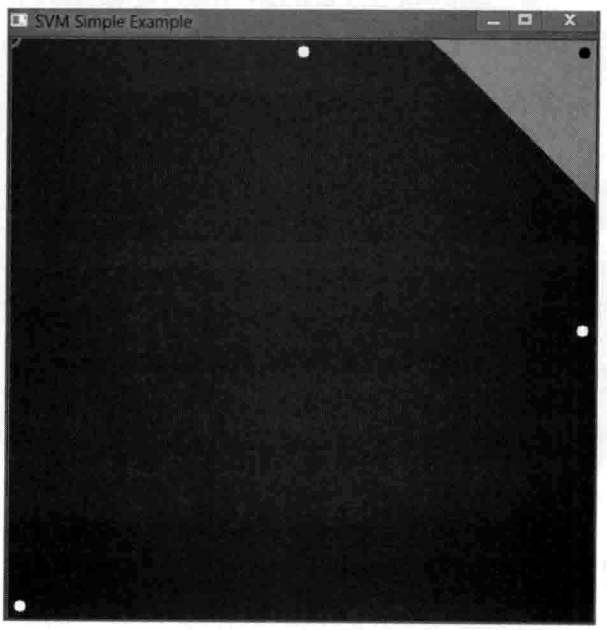


图 2.10 训练 SVM 分类器程序截图

第二个示例程序主要用于讲解在训练数据线性不可分时，如何定义支持向量

机的最优化问题。如图 2.11 所示。

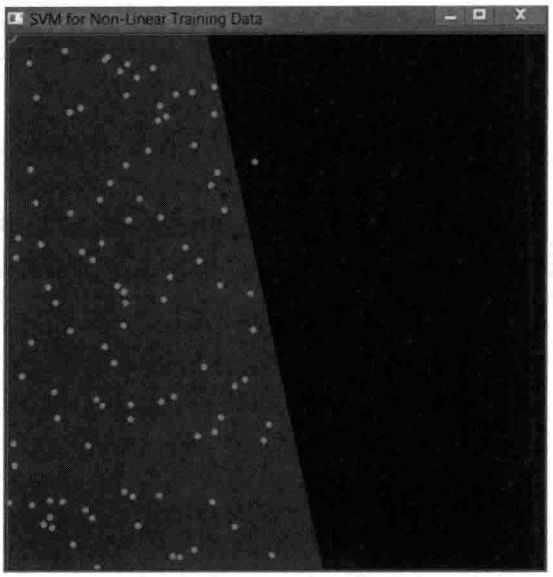


图 2.11 SVM 对线性不可分数据的处理程序截图

以上就是对 OpenCV 官方提供的示例程序的引导和概览。

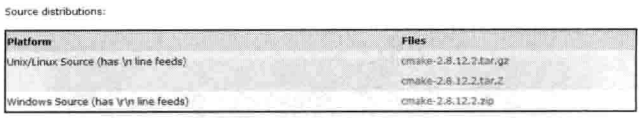
2.2 开源的魅力：编译 OpenCV 源代码

本节，我们将一起探讨如何通过已经安装的 OpenCV，选择不同的编译器类型，生成高度还原的 OpenCV 开发时的解决方案工程文件，并欣赏 OpenCV 新版本中总计六十六多万行的精妙源代码。我们可以对源代码进行再次编译，得到二进制文件，或者修改原版官方的 OpenCV 代码，并编译后为自己所用，从而为深入理解 OpenCV 的开源魅力迈出坚实的一步。

2.2.1 下载安装 CMake

想要在 Windows 平台下生成 OpenCV 的解决方案，需要一个名为 CMake 的开源软件。note：CMake，是“crossplatform make”的缩写，它是一个跨平台的安装（编译）工具，可以用简单的语句来描述所有平台的安装（编译过程）。他能够输出各种各样的 makefile 或者 project 文件，能测试编译器所支持的 C++特性，类似 UNIX 下的 automake。只是 CMake 的组态档取名为 CmakeLists.txt。Cmake 并不直接建构出最终的软件，而是产生标准的建构档（如 Unix 的 Makefile 或 Windows Visual C++的 projects/workspaces），然后再依一般的建构方式使用。这使得熟悉某个集成开发环境（IDE）的开发者可以用标准的方式建构他的软件，这种可以使用各平台的原生建构系统的能力是 CMake 和 SCons 等其他类似系统的区别之处。

CMake 可以在官网：http://www.cmake.org/上下载到。打开此链接，首先转到其下载页面，如图 2.12 所示。下载页面的 Source distributions 处可以下载到 CMake 软件的源码，对这款开源软件感兴趣的读者不妨研究一下。

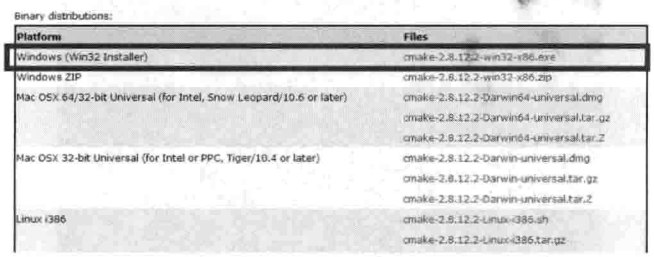


Source distributions:

Platform	Files
Unix/Linux Source (has \n line feeds)	cmake-2.8.12.2.tar.gz
	cmake-2.8.12.2.tar.Z
Windows Source (has \r\n line feeds)	cmake-2.8.12.2.zip

图 2.12 CMake 源码下载页面

在 Binary distributions 处可以下载到 CMake 的执行文件（如图 2.13），我们选择 Windows（Win32 Installer）版的进行下载。



Binary distributions:

Platform	Files
Windows (Win32 Installer)	cmake-2.8.12.2-win32-x86.exe
Windows ZIP	cmake-2.8.12.2-win32-x86.zip
Mac OSX 64/32-bit Universal (for Intel, Snow Leopard/10.6 or later)	cmake-2.8.12.2-Darwin64-universal.dmg
	cmake-2.8.12.2-Darwin64-universal.tar.gz
	cmake-2.8.12.2-Darwin64-universal.tar.Z
Mac OSX 32-bit Universal (for Intel or PPC, Tiger/10.4 or later)	cmake-2.8.12.2-Darwin-universal.dmg
	cmake-2.8.12.2-Darwin-universal.tar.gz
	cmake-2.8.12.2-Darwin-universal.tar.Z
Linux i386	cmake-2.8.12.2-Linux-i386.sh
	cmake-2.8.12.2-Linux-i386.tar.gz

图 2.13 CMake 软件下载页面

下载完成之后便是安装 CMake，这一步很简单，在此不再赘言。安装完成后，如果没有生成桌面快捷方式，则在安装路径下（如 D:\Program Files(x86)\CMake 2.8\bin）处找到【cmake-gui.exe】运行。

2.2.2 使用 CMake 生成 OpenCV 源代码工程的解决方案

这一步是全文的核心内容，为了使讲解条理清晰，我们进行分步介绍。

1.【第一步】运行 cmake-gui

上文已经讲到，如果没有生成桌面快捷方式，在安装路径下（如 D:\Program Files(x86)\CMake 2.8\bin 处）找到【cmake-gui.exe】运行。

2.【第二步】指定 OpenCV 的安装路径

如图，单击方框内的“Browse Source”按钮，在弹出的对话框中指定 OpenCV 安装时源代码的存储路径。图中以 OpenCV 安装在 D:\Program Files 下为例，在此选择路径：D:\Program Files\opencv\sources。

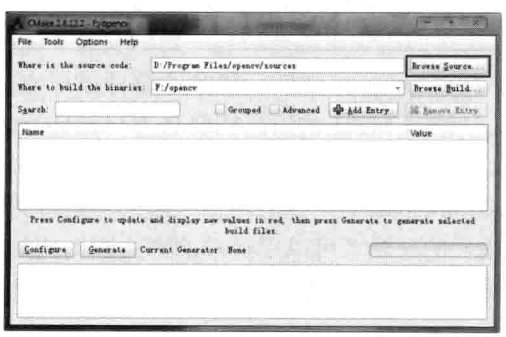


图 2.14 指定 OpenCV 的安装路径

我们可以发现，此路径下必须会有一个名为 CMakeLists.txt 的文件，这就是

给 CMake 留下的配置文件。CMake 可以根据这个配置文件，通过选择不同的编译器，来生成不同的解决方案——VisualStudio 的编译器对应的就是生成 Visual Studio 版的 sln 解决方案。

3.【第三步】指定解决方案的存放路径

单击对话框中的 “Browse Build”按钮，在弹出的对话框中指定存放生成的 opencv 解决方案的路径。比如 F:/opencv。

4.【第四步】第一次 Configure

路径都设置好后，单击如图 2.15 所示的【Configure】按钮，进行第一次配置过程。

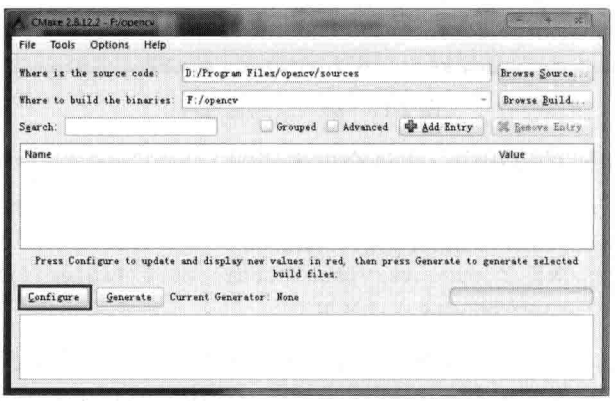


图 2.15 第一次 Configure

然后会弹出如图 2.16 所示进行编译器选择的对话框。

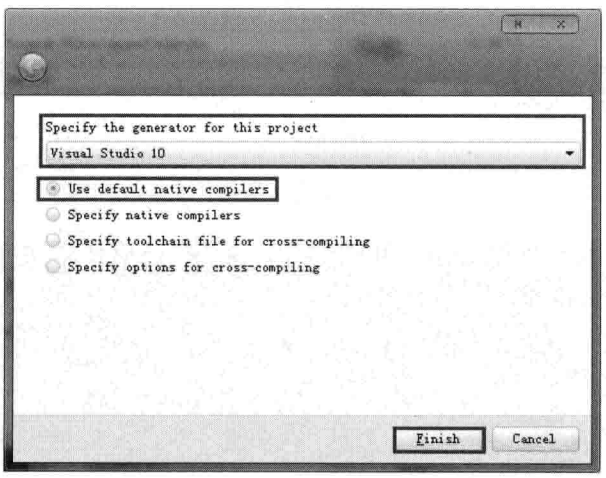


图 2.16 编译器选择

先选定“Use default native compilers”，然后可以发现下拉列表中提供了几十种编译器供选择。因为我们安装了 Visual Studio，这里会默认选择对应版本的 Visual Studio 编译器，比如 Visual Studio 10（即之后会生成对应 VS2010 的 sln 解决方案）。

确认无误后，单击“finish”按钮。于是，CMake 开始第一次源代码配置过程，如图 2.17 所示。

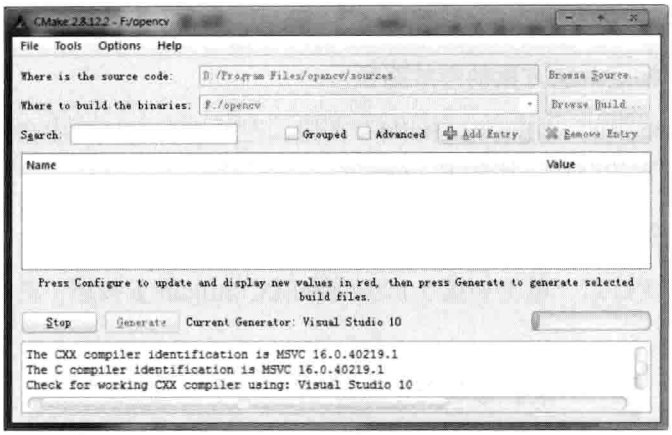


图 2.17 第一次配置过程

在 CMake 处理的过程中，也许会出现诸如

```
“Could not copy from: D:/Program Files(x86)/CMake 2.8/share/cmake
2.8/Templates/CMakeVSMacros2.vsmacros
to: C:/Users/浅墨/Documents/VisualStudio 2010/Projects/VSMacros80/
CMakeMacros/CMakeVSMacros2.vsmacros”
```

的红色字样警告，因为这是系统用户的路径名有中文字符 “浅墨”。CMake 不识别中文路径，只要我们在上面的第三步中设置生成的路径中没有中文就行了。就算有中文而造成了这个错误，也对我们这次的生成无碍，不用去管它。

看到进度条读到尽头，出现 Configuring done 字样，第一次的源码配置就完成了，如图 2.18 所示。

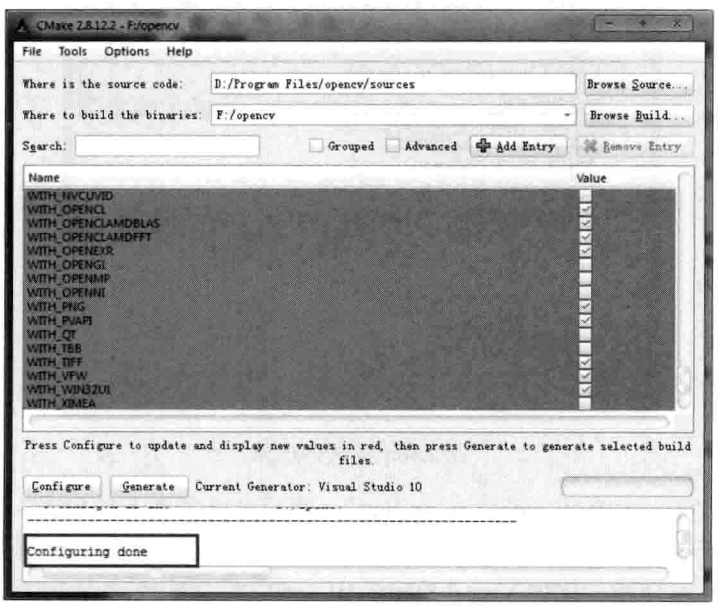


图 2.18 第一次配置完成

5.【第五步】第二次 Configure

第一次配置完成之后，还需要进行第二次配置，于是再次单击“Configure”按钮。这次的配置很快，几秒钟就会再次出现“Configuring done”字样，并且选中部分也都正常了。如图 2.19 所示。

图 2.19 第二次配置完成

6.【第六步】单击 Generate，成功生成项目

这样，就只需要单击 Generate 按钮，来生成最终的解决方案了。如图 2.20 所示。

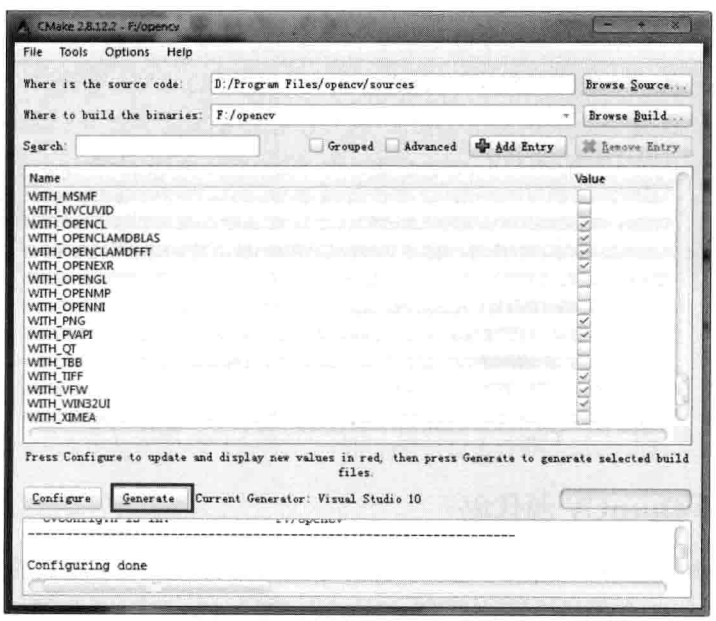


图 2.20 单击生成【Generate】按钮

因为之前已经有过两次的 configure 过程，所以生成解决方案也将会非常快。如图 2.21 所示。

图 2.21 成功生成解决方案

出现 Generating done 字样，就表示大功告成，可以去之前指定的路径（F:\opencv）下找寻生成的解决方案了。

注意：2.4.8、2.4.9 的 OpenCV 用 CMake 生成的工程只有 3M 多，相比之前测试 2.4.6 版本的 OpenCV 有 3 个多 G 的工程，笔者一开始都以为生成出错了。但是点开 sln 工程，发现里面依然可以看到源代码。这是因为从 2.4.7 版本起，OpenCV 源代码就直接包含在了 opencv 的安装路径下，我们生成的 sln 工程，也只是链接到了 opencv 安装路径下的源文件而已，工程本身并不大。工程文件如图 2.22 所示。

图 2.22 OpenCV 源代码工程的 Visual Stuido 解决方案文件

2.2.3 编译 OpenCV 源代码

打开刚刚生成的 “OpenCV.sln” 解决方案，可以看到一个庞大的工程——这是一个包含了 67 个项目的解决方案（对 OpenCV 2.4.9 而言），如图 2.23 所示。

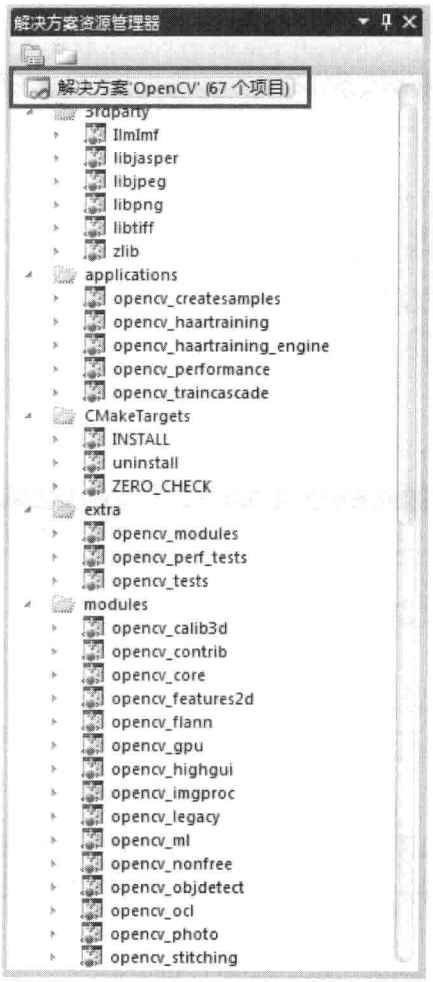


图 2.23 OpenCV 源代码的资源管理器视图

大家这时在解决方案资源管理界面，可以随意点进去一个感兴趣的项目，比如 opencv_core，再在 opencv_core/Src/matrix.cpp 查看其中某个文件的源代码。笔者截的这张图（图 2.24）是大家以后都会很熟悉的 Mat 类型的某个构造函数的源代码。

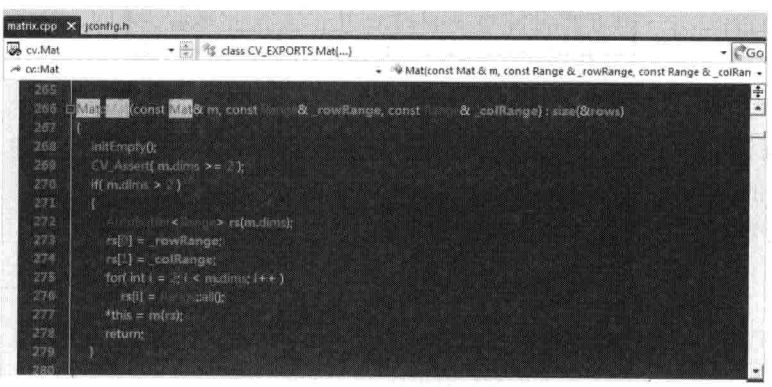


图 2.24 Mat 类型某构造函数的源代码

对源代码进行欣赏之后，我们可以按【F5】或者使用其他操作来启动调试，

编译过程如图 2.25 所示。

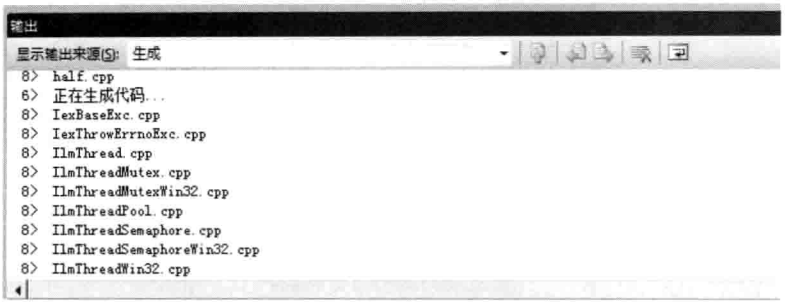


图 2.25 OpenCV 源代码编译过程

编译结果如图 2.26 所示。

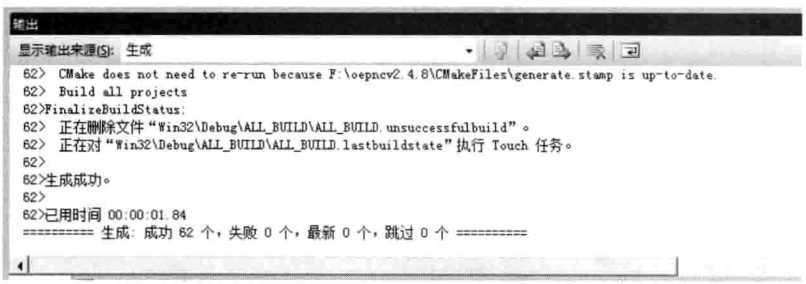


图 2.26 OpenCV 源代码编译结果

在实验过程中，编译总共用时约 5 分钟，而具体时间和机器配置息息相关。从截图 2.26 中可以发现，这次编译成功了 62 个项目，失败了 0 个，即全部编译生成成功。

另外，编译完成会得到如图 2.27 所示的警告。

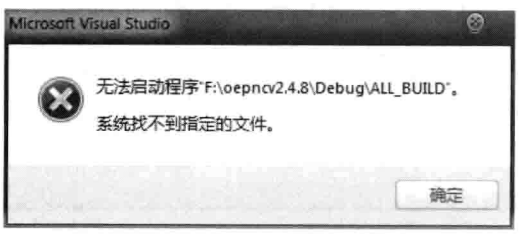


图 2.27 “系统找不到指定文件”错误提示对话框

若编译完成时出现这样的对话框，无须紧张，因为这是正常的。原因是 OpenCV 的源代码工程默认将“ALL_BUILD”这个项目设为了启动项，编译成功后，就会默认运行它。

图 2.28 被默认设置为启动项的“ALL_BUILD”项目

而“ALL_BUILD”是一个项目生成周边的杂项，不是 exe 执行文件，本身不可以运行，所以自然就会弹出这样的错误提示。

若想让此报错的对话框不显示，指定一个另外的启动项即可。我们可以在解决方案资源管理器里，对需要设为启动项的那个项目右键点击，在弹出的菜单项中单击【设为启动项（J）】进行设定。如下图：

图 2.29 设置新的启动项目

编译 OpenCV，其实就是得到了一些二进制的生成文件，如 dll、lib 和 exe。因为是在 debug 下编译的，所以在工程目录的 bin\debug 下会生成 OpenCV 对应版本的依赖库（700 多 MB，有点大），可以供以后调用此次编译的 OpenCV 时使用。

2.3 “opencv.hpp”头文件认知

在任意一个 OpenCV 程序中，通过转到定义，我们可以发现”#include <opencv2/ opencv.hpp>”一句中的头文件定义类似如下：

```
#include <opencv2/opencv.hpp>
#ifndef __OPENCV_ALL_HPP__
#define __OPENCV_ALL_HPP__

#include "opencv2/core/core_c.h"
#include "opencv2/core/core.hpp"
#include "opencv2/flann/miniflann.hpp"
#include "opencv2/imgproc/imgproc_c.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/photo/photo.hpp"
#include "opencv2/video/video.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/calib3d/calib3d.hpp"
```

```
#include "opencv2/ml/ml.hpp"
#include "opencv2/highgui/highgui_c.h"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/contrib/contrib.hpp"

#endif
```

通过观察代码可知，opencv.hpp 中已经包含了 OpenCV 各模块的头文件，如高层 GUI 图形用户界面模块头文件“highgui.hpp”、图像处理模块头文件“imgproc.hpp”、2D 特征模块头文件“features2d.hpp”等。

所以，我们在编写 core、objdetect、imgproc、photo、video、features2d、objdetect、calib3d、ml、highgui、contrib 模块的应用程序时，原则上仅写上一句”#include <opencv2/opencv.hpp>”即可，这样可以精简优化代码。

而本书为了便于演示和帮助大家理解，在编写特定模块的程序时，默认情形下还是包含了相应模块的头文件。

2.4 命名规范约定

使用一套成熟的命名规则，不仅可以让我们规范行事，还可以让别人在阅读我们书写的代码时，可以更好更快地理解我们的思路，从而增强了代码的可读性，方便各程序员之间相互交流代码。

《代码大全（第二版）》书中第 277 页，表 11-3 展示的一套成熟的命名规则，笔者对其细节部分进行细微地改进，得到如表 2.1 所示命名约定。

表 2.1 命名规则约定

描述	实例
类名混合使用大小写，首字母大写	ClassName
类型定义，包括枚举和 typedef，混合使用大小写，首字母大写	TypeName
枚举类型除了混合使用大小写外，总以复数形式表示	EnumeratedTypes
局部变量混合使用大小写，且首字母小写，其名字应该与底层数据类型无关，而且应该反映该变量所代表的事物	localVariable
子程序参数的格式混合使用大小写，且每个单词首字母大写，其名字应该与底层数据类型无关，而且应该反映该变量所代表的事物	RoutineParameter
对类的多个子程序可见（且只对该类可见）的成员变量名用 m_前缀	m_ClassVariable
局部变量名用 g_前缀	g_GlobalVariable
具名常量全部大写	CONSTANT
宏全部大写，单词间用分隔符“_”隔开	SCREEN_WIDTH
枚举类型成员名用能反映其基础类型的、单数形式的前缀。例如，Color_Red，Color_Blue	Base_EnumeratedType

说到命名规则，当然少不了匈牙利命名法。匈牙利命名法是编程时的一种命名规范，其基本原则是：变量名=属性+类型+对象描述，其中每一对象的名称都要求有明确含义，可以取对象名字全称或名字的一部分。命名要基于容易记忆容易理解的原则，并要求保证名字的连贯性。据说这种命名法是一位叫 Charles Simonyi 的匈牙利程序员发明的，后来他在微软工作了几年，于是这种命名法就通过微软的各种产品和文档资料向世界传播开了。现在，大部分程序员不管使用什么软件进行开发，或多或少都沿用了这种命名法。匈牙利命名法的出发点是把变量名按“属性+类型+对象描述“的顺序组合起来，以使程序员命名变量时对变量的类型和其他属性有直观的了解。下面是匈牙利变量命名法的一些规范。

匈牙利命名法中常用的小写字母的缀如表 2.2 所示。

表 2.2 变量命名规范

前缀写法	类型	描述	实例
ch	char	8 位字符	chGrade
ch	TCHAR	如果_UNICODE 定义，则为 16 位字符	chName
b	BOOL	布尔值	bEnable
n	int	整型（其大小依赖于操作系统）	nLength
n	UINT	无符号值（其大小依赖于操作系统）	nHeight
w	WORD	16 位无符号值	wPos
l	LONG	32 位有符号整型	lOffset
dw	DWORD	32 位无符号整型	dwRange
p	*	指针	pDoc
lp	FAR*	远指针	lpszName
lpsz	LPSTR	32 位字符串指针	lpszName
lpsz	LPCSTR	32 位常量字符串指针	lpszName
lpsz	LPCTSTR	如果_UNICODE 定义，则为 32 位常量字符串指针	lpszName
h	handle	Windows 对象句柄	hWnd
lpfn	callback	指向 CALLBACK 函数的远指针	LpfnName

关键字字母组合如表 2.3 所示。

表 2.3 关键字字母组合

描述内容	使用的关键字母组合
最大值	Max
最小值	Min
初始化	Init
临时变量	T（或 Temp）
源对象	Src
目的对象	Dst

虽然命名规范摆在这里，但我们在实际应用中切记不要迂腐，不要墨守成规，该变通的时候可以做适当的变通。且一旦确定了一套合适且成熟命名规范后，最好不要中途更改。

2.5 argc 与 argv 参数解惑

2.5.1 初识 main 函数中的 argc 和 argv

在与 OpenCV 打交道时，我们常会在相关的示例程序中见到 argc 和 argv 这两个参数，如 OpenCV 的官方示例程序 Samples 中、OpenCV 经典教材《Learning OpenCV》中，等等。这常常会让众多初学者疑惑，不清楚它们是何用途。本节内容正为解此惑而写。

argc 和 argc 中的 arg 指的是“参数”（例如：arguments，argument counter 和 argument vector）。其中，argc 为整数，用来统计运行程序时送给 main 函数的命令行参数的个数；而* argv[ ]：为字符串数组，用来存放指向字符串参数的指针数组，每一个元素指向一个参数。

Argc，argc 这两个参数一般在用命令行编译程序时有用。在初学 C++时，往往要弱化 argc 和 argv 的用法，main 函数常常不带参数，如下。

```
int main()
{

}
```

而在 opencv 的官方示例程序中，main 函数的写法常常会带上两个形参，一般为 argc 和 argv，并且在函数体内部会使用到这两个形参，如下：

```
int main( int argc, char** argv )
{
const char* imagename = argc > 1 ? argv[1] : "lena.jpg";
……
}
```

其实，带形参的 main 函数，如 main（int argc，char *argv[ ]，char **env），

是 UNIX、Linux 以及 Mac OS 操作系统中 C/C++ 的 main 函数的标准写法，并且是血统最纯正的 main 函数的写法。可能是由于外国的专家们更习惯使用 UNIX、Linux 以及 Mac OS 等操作系统，所以我们接触到由他们开发和维护的 OpenCV 这款开源视觉库的时候，自然会发现代码中常有 argc 和 argv 的出现。

2.5.2 argc、argv 的具体含义

argc 和 argv 这两个参数一般在用命令行编译程序时有用。

主函数 main 中变量（int argc，char *argv[ ]）的含义

有些编译器允许将 main()的返回类型声明为 void，这就已不再是合法的 C++了。

其实，main（int argc, char *argv[ ]，char **env）才是 UNIX 和 Linux 中的标准写法。其中，第一个参数，int 类型的 argc，为整型，用来统计程序运行时发送给 main 函数的命令行参数的个数，在 Vsiual Studio 中默认值为 1。第二个参数，char* 类型的 argv[ ]:，为字符串数组，用来存放指向的字符串参数的指针数组，每一个元素指向一个参数。各成员含义如下：

- argv[0]指向程序运行的全路径名
- argv[1]指向在 DOS 命令行中执行程序名后的第一个字符串
- argv[2]指向执行程序名后的第二个字符串
- argv[3]指向执行程序名后的第三个字符串
- argv[argc]为 NULL

需要指出，argv[1]对应于【项目属性】→【配置属性】→【调试】→【命令参数】中的值。记住双引号也要带上，比如读取名为 1.jpg 的图片，如图 2.30 所示，就要在命令参数中填字符串“1.jpg”。

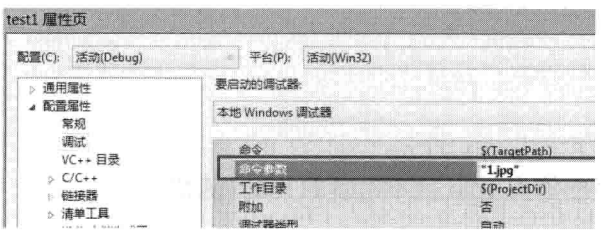


图 2.30 属性页中的命令参数（1）

而如果有多个字符串，则用空格隔开。比如要读入两张名称分别为 1.jpg 和 2.jpg 的图片，在命令参数中填“1.jpg”“2.jpg”（“1.jpg”和“2.jpg”之间用空格隔开）即可，如图 2.31 所示。

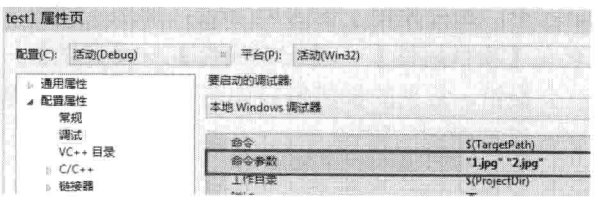


图 2.31 属性页中的命令参数（2）

第三个参数，char **类型的 env，为字符串数组。env[ ]的每一个元素都包含 ENVVAR=value 形式的字符串。其中 ENVVAR 为环境变量，value 为 ENVVAR 的对应值。在 OpenCV 中很少使用它。

argc、argv 和 env 是在 main()函数之前被赋值的。其实，main()函数严格意义上并不是真正的程序入口点函数，往往入口点还与操作系统有关。而在 Windows 的控制台应用程序中，将 main()函数作为程序入口点，并且很少使用 argc、argv 等命令行参数。

2.5.3 Visual Studio 中 main 函数的几种写法说明

需要注意的是，在如今各版本的 Visual Stuidio 编译器中，main()函数带参数 argc 和 argv 或不带，也就是说，无论我们是否在函数体中使用 argc 和 argv，返回值为 void 或不为 void，都是合法的。

即至少有如下 3 种写法合法。

1.【写法一】返回值为整型带参的 main 函数

```
int main( int argc, char** argv )
{
   //函数体内使用或不使用 argc 和 argv 都可行
   ……
   return 1;
}
```

2.【写法二】返回值为整型不带参的 main 函数

```
int main( int argc, char** argv )
{
   //函数体内使用了 argc 或 argv
   ……
   return 1;

}
```

3.【写法三】返回值为 void 且不带参的 main 函数

```
int main()
{
   ……
   return 1;

}
```

在 Visual Studio 中，如果使用了 argv 或 argc，即上文代码中的第一种写法，且在使用之前没有在【项目属性】→【配置属性】→【调试】→【命令参数】中指定参数的值，就会报错，常见的报错窗口如图 2.32 所示，这是研究 OpenCV 官方提供的示例程序时经常碰到的错误。

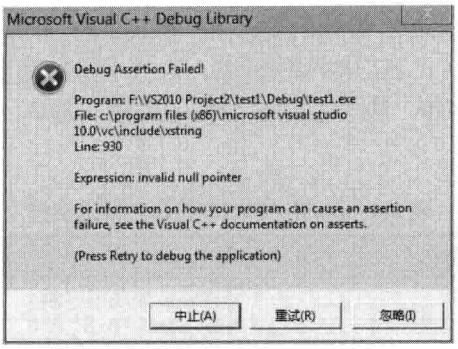


图 2.32 命令参数缺失时常见的报错窗口

想要解决此问题并编译通过，除了上文讲过的在项目属性页中填上命令参数外，最简单的做法就是在不影响原始程序的基础上，将和 argv 或 argc 有关的代码进行替换或注释。

比如将“Mat srcImage=imread（argv[1]，1）；//读取字符串名为 argv[1]的图片”替换为“Mat srcImage=imread（“1.jpg”,1）；//工程目录下有一张名为“1.jpg”的图片”。

2.5.4 总结

讲解至此，读者应该对 argc 和 argv 有了比较透彻的认识。简单来说

- int argc 表示命令行字串的个数
- char *argv[]表示命令行参数的字符串

由于我们使用的开发环境为 Visual Studio，往后若在 OpenCV 相关代码中遇到这两个参数，可以在代码中将其用路径字符串替换，或者在项目属性页中给其赋值，而对程序整体影响不大的部分就将其注释掉。

2.6 格式输出函数 printf()简析

2.6.1 格式输出：printf()函数

相信学习过 C 语言的读者们都知道，printf 函数并非 OpenCV 中的函数，而是标准的 C 语言函数，包含在 studio.h 之中。只不过 OpenCV 对其也有包含，我们只需包含头文件如 opencv.hpp，就可以使用它。

printf 函数是我们经常会用到的格式输出函数，其关键字最末一个字母 f 即为“格式”（format）之意。其功能是按用户指定的格式，把指定的数据显示到窗口中。printf 函数调用的一般形式如下：

```
int printf(const char *format, ...);
```

即：

```
int printf("格式控制字符串", 输出表列);
```

printf 函数的一个比较有特殊的用法是“格式字符串”，其用于指定输出格式，可由格式字符串和非格式字符串两种组成。格式字符串是以%开头的字符串，在%后面跟有各种格式字符，以说明输出数据的类型、形式、长度、小数位数等。如表 2.4 所示。

表 2.4 printf“格式字符串”候选字符

格式字符串	作用
%d	将整数转成十进制
%f	将整数转成浮点数
%u	十进制无符号整数
%o	将整数转成八进制
%c	将整数转成对应的 ASCII 字符
%s	将整数转成字符串
%x	整数转成小写十六进制
%X	整数转成大写十六进制
%p	输出地址符
%%	输出百分比符号，不进行转换

而上述表中没有，或不带%的“非格式字符”，则以原样输出，且格式字符串和各输出项在数量和类型上应该一一对应。

除了格式字符串，printf 还有一些特殊规定的字符，用法如表 2.5 所示。

表 2.5 printf 中特殊规定的字符

规定字符	作用
\n	换行操作
\f	清屏并换页
\r	回车
\t	Tab 符
\xhh	用 16 进表示的 ASCII 码，其中每个 h 可以用 0～f 中的一个代替

2.6.2 示例程序：printf 函数的用法示例

以上讲解的内容已经足够应付本书将要讲解的代码里关于 printf 的使用了，现在，让我们看一个示例，以结束 printf 函数的讲解，向下一节进发。

```
#include <opencv2/opencv.hpp>
using namespace cv;

void main()
{
    int a=66,b=68;
    printf("\n\t%d %d\n",a,b);//输出十进制整型
```

```
    printf("\n\t%06d,%06d\n",a,b);//输出 6 位十进制整型
    printf("\n\t%c,%c\n",a,b);//按字符型输出
    printf("\n\t 结果为：a=%d,b=%d",a,b);//可以配合其他内容一同输出
    getchar();//等待读入任意字符而结束，在此用于保持窗口显示，直到任意按键按下
}
```

运行结果如图 2.33 所示。

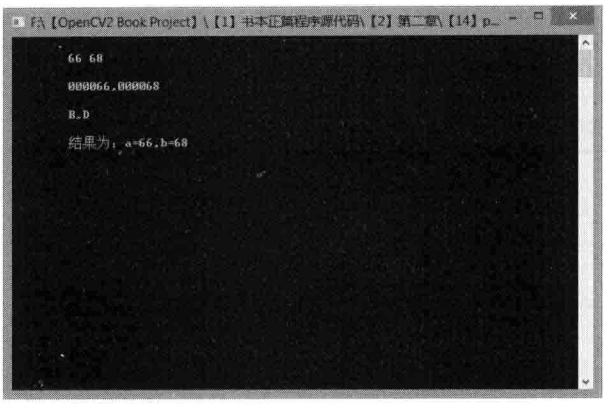


图 2.33 printf 输出结果

2.7 智能显示当前使用的 OpenCV 版本

如果大家已经查看了书本配套示例代码中的第 8 到第 13 的程序源码或者是后面章节中稍微复杂一点的示例程序，就会发现它们能根据你当前使用的 OpenCV 版本智能检测并输出相应的版本号。其实现非常简单，原理是 OpenCV 为我们提供了一个名为“CV_VERSION”的用于标识当前 OpenCV 版本的宏。于是，我们使用 cout 或者 printf 等文字输出函数以此宏作为输出对象，可以智能地显示当前的 OpenCV 版本号。

可以书写如下代码进行 OpenCV 版本的智能检测，并将版本号输出到控制台窗口中。

```
printf("\t 当前使用的 OpenCV 版本为  OpenCV "CV_VERSION);
```

2.8 本章小结

在本章中，我们学习了很多非常实用的知识，使大家可以对 OpenCV 有一个宏观的认知，也给大家提供了一些脱离书本依靠官方文档自学的思路，比如对 OpenCV 官方例程的引导与赏析，和如何编译 OpenCV 总计四十四万多行的源代码。此外，还有一些命名规范的约定和相关 C 语言基础函数的复习。本章内容授之以鱼，更授之以渔，相信通过本章内容的学习，可以对 OpenCV 有一个更加全面的认识。

本章示例程序清单

示例程序序号	程序说明	对应章节
8	官方例程引导、赏析之彩色目标跟踪：Camshift	2.1.1
9	官方例程引导、赏析之光流：optical flow	2.1.2

续表

示例程序序号	程序说明	对应章节
10	官方例程引导、赏析之点追踪：lkdemo	2.1.3
11	官方例程引导、赏析之人脸识别：objectDetection	2.1.4
12	官方例程引导、赏析之支持向量机：支持向量机引导	2.1.5
13	官方例程引导、赏析之支持向量机：处理线性不可分数据	2.1.5
14	printf 函数的用法示例	2.6.2

第 3 章
HighGUI 图形用户界面初步

导读

在 OpenCV 架构分析一节中我们讲过，HighGUI 模块为高层 GUI 图形用户界面模块，包含媒体的输入输出、视频捕捉、图像和视频的编码解码、图形交互界面的接口等内容。而在 1.5 节中讲到的 VideoCapture 视频类，就是出自此 HighGUI 模块。本章旨在为大家展开讲解 OpenCV 中最常用到的一些交互操作，包括图像的载入、显示和输出，为程序添加滑动条，以及鼠标操作等常用内容。

本章中，你将学到：

- 图像的载入、显示和输出到文件的详细分析
- 滑动条（Trackbar）的创建和使用
- OpenCV 中的鼠标操作

3.1 图像的载入、显示和输出到文件

学习过以往版本OpenCV的读者应该都清楚，对于OpenCV1.0时代的基于C语言接口而建的图像存储格式IplImage*，如果在退出前忘记release掉的话，会造成内存泄露，而且用起来十分繁琐。我们在debug程序的时候，往往很大一部分时间会去纠结手动释放内存相关的问题。虽然对于小型的程序来说，手动管理内存不是什么难题，但一旦开发的项目日益庞大，代码量达到一定的规模，我们便会开始越来越多地纠缠于内存管理的问题，而不能把全部精力用于解决核心开发目标。因为不合适的图像存储数据结构而疲于维护日益庞大的项目，就有些舍本逐末的感觉了。

自踏入2.0版本的时代以来，OpenCV采用了Mat类作为数据结构进行图像存取。这一改进使OpenCV变得和几乎零门槛入门的Matlab一样，很容易上手和用于实际开发。新版OpenCV中甚至有些函数名称都和Matlab中的一样，比如大家所熟知的imread、imwrite、imshow等函数。这对于广大图像处理和计算机视觉领域的研究者们来说，的确是一件可喜可贺的事情。而这一小节中，我们主要来详细讲解OpenCV2、OpenCV3入门最基本的问题，即图像的载入、显示和输出。

3.1.1 OpenCV的命名空间

OpenCV中的C++类和函数都是定义在命名空间cv之内的，有两种方法可以访问：第一种，是在代码开头的适当位置加上usingnamespace cv；这句代码，规定程序位于此命名空间之内；另外一种，是在使用OpenCV的每一个类和函数时，都加入cv::命名空间。不过这种情况会很繁琐，每用一个OpenCV的类或者函数，都要多敲四下键盘写出cv::。所以，推荐大家在代码开头的适当位置，加上using namespace cv;这句。

比如在写简单的OpenCV程序的时候，以下三句可以作为标配：

```
#include <opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
using namespace cv;
```

3.1.2 Mat类简析

Mat类是用于保存图像以及其他矩阵数据的数据结构，默认情况下其尺寸为0。我们也可以指定其初始尺寸，比如定义一个Mat类对象，就要写cv::Mat pic（320,640,cv::Scalar(100)）；

Mat类型作为OpenCV2、OpenCV3新纪元的重要代表，在稍后的章节中，笔者会花长篇幅详细讲解它，现在我们只要理解它是对应于OpenCV1.0时代的IplImage，主要用来存放图像的数据结构就行了。对于本节，我们需要用到关于Mat的其实就简单的这样一句代码：

```
Mat srcImage= imread("dota.jpg");
```

这表示从工程目录下把一幅名为 dota.jpg 的 jpg 类型的图像载入到 Mat 类型的 srcImage 变量中。对于这里的 imread 函数，用于将图片读入 Mat 类型中，会在下文进行详细剖析。而 Mat 类，也会在第 4 章中进行更加全面的讲解。

3.1.3 图像的载入与显示概述

在新版本的 OpenCV2 中，最简单的图像载入和显示只需要两句代码，非常便捷。这两句代码分别对应了两个函数，它们分别是 imread()以及 imshow()。

3.1.4 图像的载入：imread()函数

首先来看 imread 函数，其用于读取文件中的图片到 OpenCV 中。可以在 OpenCV 官方文档中查到它的原型，如下。

```
Mat imread(const string& filename, intflags=1 );
```

（1）第一个参数，const string&类型的 filename，填我们需要载入的图片路径名。在 Windows 操作系统下，OpenCV 的 imread 函数支持如下类型的图像载入。

- Windows 位图：*.bmp, *.dib
- JPEG 文件：*.jpeg, *.jpg, *.jpe
- JPEG 2000 文件：*.jp2
- PNG 图片：*.png
- 便携文件格式：*.pbm, *.pgm, *.ppm
- Sun rasters 光栅文件：*.sr, *.ras
- TIFF 文件：*.tiff, *.tif

（2）第二个参数，int 类型的 flags，为载入标识，它指定一个加载图像的颜色类型。可以看到它自带默认值 1，所以有时候这个参数在调用时可以忽略。在看了下面的讲解之后，我们就会发现，如果在调用时忽略这个参数，就表示载入三通道的彩色图像。这个参数可以在 OpenCV 中标识图像格式的枚举体中取值。通过转到定义，我们可以在 higui_c.h 中发现这个枚举的定义是这样的：

```
enum
{
/* 8bit, color or not */
   CV_LOAD_IMAGE_UNCHANGED  =-1,
/* 8bit, gray */
   CV_LOAD_IMAGE_GRAYSCALE  =0,
/* ?, color */
   CV_LOAD_IMAGE_COLOR      =1,
/* any depth, ? */
   CV_LOAD_IMAGE_ANYDEPTH   =2,
/* ?, any color */
   CV_LOAD_IMAGE_ANYCOLOR   =4
};
```

对常用标识符相应的解释：

- CV_LOAD_IMAGE_UNCHANGED——等价取值为-1，这个标识在新版本中已被废置，忽略。
- CV_LOAD_IMAGE_GRAYSCALE——等价取值为 0，如果取这个标识的话，始终将图像转换成灰度再返回。
- CV_LOAD_IMAGE_COLOR——等价取值为 1，如果取这个标识，总是转换图像到彩色再返回。
- CV_LOAD_IMAGE_ANYDEPTH——等价取值为 2，如果取这个标识，且载入的图像的深度为 16 位或者 32 位，就返回对应深度的图像，否则，就转换为 8 位图像再返回。

需要说明的是，如果输入有冲突的标志，将采用较小的数字值。比如 CV_LOAD_IMAGE_COLOR | CV_LOAD_IMAGE_ANYCOLOR 将载入三通道图。而如果想要载入最真实无损的源图像，可以选择 CV_LOAD_IMAGE_ANYDEPTH | CV_LOAD_IMAGE_ ANYCOLOR。

OpenCV3 中此处列举的 CV_LOAD_IMAGE_UNCHANGED、CV_LOAD_IMAGE_GRAYSCALE 、 CV_LOAD_IMAGE_COLOR 、 CV_LOAD_IMAGE_ ANYDEPTH、CV_LOAD_IMAGE_ANYCOLOR 等宏已经失效，但在官方文档和源代码中暂时没有发现新的可以替换它们的宏，所以请暂时用与其等价的-1、0、1、2、4 代替。相信后续 OpenCV 版本中一定会进行修复。

因为 flags 是 int 型的变量，若我们不在这个枚举体中取固定的值，可以这样进行：

- flags >0 返回一个 3 通道的彩色图像；
- flags =0 返回灰度图像；
- flags <0 返回包含 Alpha 通道的加载图像。

输出的图像默认情况下不返回 Alpha 通道。如果想要载入 Alpha 通道，这里就需要取负值。比如，将 flags 取 1999 也是可以的，和取 1 的效果一样，它们同样表示返回一个 3 通道的彩色图像。
另外，需要额外注意，若以彩色模式载入图像，解码后的图像会以 BGR 的通道顺序进行存储，即蓝、绿、红的顺序，而不是通常的 RGB 的顺序。

经过上面详细的讲解，我们一起看几个载入示例，以便多方位地掌握 imread 函数的用法。

```
Mat image0=imread("1.jpg",2 | 4);//载入无损的源图像
Mat image1=imread("1.jpg",0);//载入灰度图
Mat image2=imread("1.jpg",199);//载入 3 通道的彩色图像
```

3.1.5 图像的显示：imshow()函数

imshow()函数用于在指定的窗口中显示一幅图像，函数原型如下。

```
void imshow(const string& winname, InputArray mat);
```

- 第一个参数：const string&类型的 winname，填需要显示的窗口标识名称。
- 第二个参数：InputArray 类型的 mat，填需要显示的图像。

imshow 函数用于在指定的窗口中显示图像。如果窗口是用 CV_WINDOW_AUTOSIZE（默认值）标志创建的，那么显示图像原始大小。否则，将图像进行缩放以适合窗口。而 imshow 函数缩放图像，取决于图像的深度，具体如下。

- 如果载入的图像是 8 位无符号类型(8-bit unsigned)，就显示图像本来的样子。
- 如果图像是 16 位无符号类型(16-bit unsigned)或 32 位整型(32-bit integer)，便用像素值除以 256。也就是说，值的范围是[0,255 x 256]映射到[0, 255]。
- 如果图像是 32 位浮点型（32-bit floating-point），像素值便要乘以 255。也就是说，该值的范围是[0, 1]映射到[0, 255]。

还有一点，在窗口创建的时候，如果设定了支持 OpenGL(WINDOW_OPENGL)，那么 imshow 还支持 ogl::Buffer、ogl::Texture2D 以及 gpu::GpuMat 作为输入。

关于 imwrite 和 imshow 函数最精简的示例程序，可以参考 1.3.8 节“最终的测试”或 1.4.1 节“第一个程序：图像显示”中的代码。

3.1.6 关于 InputArray 类型

对于这里的 InputArray 类型，通过对 InputArray 转到定义，我们可以在 core.hpp 中查到一个 typedef 声明，如下：

```
typedef const _InputArray& InputArray;
```

这其实一个类型声明引用，就是说_InputArray 和 InputArray 是一个意思，因此我们就来做最后一步：对_InputArray 进行转到定义，可以在 core.hpp 头文件中发现 InputArray 的真身。InputArray 的源代码略显冗长，不在这里贴出，若读者自己去实践并通过转到定义的方法找到 InputArray 的真身，便可以看到_InputArray 类的里面首先定义了一个枚举，然后是各类的模板类型和一些方法。而很多时候，遇到函数原型中的 InputArray/OutputArray 类型，我们把它简单地当做 Mat 类型即可。

3.1.7 创建窗口：namedWindow()函数

namedWindow 函数用于创建一个窗口。若是简单地进行图片显示，可以略去 namedWindow 函数的调用，即先调用 imread 读入图片，然后用 imshow 直接指定出窗口名进行显示即可。但需要在显示窗口之前就用到窗口名时，比如我们后面会马上讲到滑动条的使用，要指定滑动条依附到某个窗口上，就需要 namedWindow 函数先创建出窗口，显式地规定窗口名称了。

namedWindow 的函数原型如下：

```
void namedWindow(const string& winname,int flags=WINDOW_AUTOSIZE );
```

(1) 第一个参数，const string&型的 name，填写被用作窗口的标识符的窗口名称。

（2）第二个参数，int 类型的 flags，窗口的标识，可以填如下几种值。

- WINDOW_NORMAL，设置这个值，用户可以改变窗口的大小（没有限制）。OpenCV2 中它还可以写为 CV_WINDOW_NORMAL。
- WINDOW_AUTOSIZE，设置这个值，窗口大小会自动调整以适应所显示的图像，并且用户不能手动改变窗口大小。OpenCV2 中它还可以写为 CV_WINDOW_AUTOSIZE。
- WINDOW_OPENGL，设置这个值，窗口创建的时候会支持 OpenGL。OpenCV2 中它还可以写为 CV_WINDOW_OPENGL。

首先需要注意的是，namedWindow 函数有默认值 WINDOW_AUTOSIZE，所以，一般情况下，这个函数我们填一个变量就行了。namedWindow 函数的作用是通过指定的名字，创建一个可以作为图像和进度条的容器窗口。如果具有相同名称的窗口已经存在，则函数不做任何事情。我们可以调用 destroyWindow()或者 destroyAllWindows()函数来关闭窗口，并取消之前分配的与窗口相关的所有内存空间。

但是事实上，对于代码量不大的简单程序来说，我们完全没有必要手动调用上述的 destroyWindow()或者 destroyAllWindows()函数，因为在退出时，所有的资源和应用程序的窗口会被操作系统自动关闭。

3.1.8 输出图像到文件：imwrite()函数

在 OpenCV 中，输出图像到文件一般采用 imwrite 函数，它的声明如下。

```
bool imwrite(const string& filename,InputArray img, const vector<int
>& params=vector<int>() );
```

（1）第一个参数，const string&类型的 filename，填需要写入的文件名。注意要带上后缀，如“123.jpg”。

（2）第二个参数，InputArray 类型的 img，一般填一个 Mat 类型的图像数据。

（3）第三个参数，const vector<int>&类型的 params，表示为特定格式保存的参数编码。它有默认值 vector<int>()，所以一般情况下不需要填写。而如果要填写的话，有下面这些需要了解的地方：

- 对于 JPEG 格式的图片，这个参数表示从 0 到 100 的图片质量（CV_IMWRITE_JPEG_ QUALITY），默认值是 95。
- 对于 PNG 格式的图片，这个参数表示压缩级别（CV_IMWRITE_PNG_ COMPRESSION）从 0 到 9。较高的值意味着更小的尺寸和更长的压缩时间，默认值是 3。
- 对于 PPM，PGM，或 PBM 格式的图片，这个参数表示一个二进制格式标志（CV_IMWRITE_PXM_BINARY），取值为 0 或 1，默认值是 1。

imwrite 函数用于将图像保存到指定的文件。图像格式是基于文件扩展名的，可保存的扩展名和 imread 中可以读取的图像扩展名一致。

下面是一个示例程序，讲解 imwrite 函数的用法——在 OpenCV 中生成一幅 png 图片，并写入到当前工程目录下。

```
#include<opencv2/opencv.hpp>
#include <vector>

using namespace cv;
using namespace std;

void createAlphaMat(Mat &mat)
{
    for(int i = 0; i < mat.rows; ++i) {
        for(int j = 0; j < mat.cols; ++j) {
         Vec4b&rgba = mat.at<Vec4b>(i, j);
         rgba[0]= UCHAR_MAX;
         rgba[1]= saturate_cast<uchar>((float (mat.cols - j)) /
((float)mat.cols) *UCHAR_MAX);
         rgba[2]= saturate_cast<uchar>((float (mat.rows - i)) /
((float)mat.rows) *UCHAR_MAX);
         rgba[3]= saturate_cast<uchar>(0.5 * (rgba[1] + rgba[2]));
        }
    }
}

int main()
{
    //创建带 Alpha 通道的 Mat
    Mat mat(480, 640, CV_8UC4);
    createAlphaMat(mat);

    vector<int>compression_params;
    /此句代码的 OpenCV2 版为：
    //compression_params.push_back(CV_IMWRITE_PNG_COMPRESSION);
    //此句代码的 OpenCV3 版为：
    compression_params.push_back(IMWRITE_PNG_COMPRESSION);
    compression_params.push_back(9);

    try{
        imwrite("透明 Alpha 值图.png", mat, compression_params);
        imshow("生成的 PNG 图",mat);
        fprintf(stdout,"PNG 图片文件的 alpha 数据保存完毕~\n 可以在工程目录下
查看由 imwrite 函数生成的图片\n");
        waitKey(0);
    }
    catch(runtime_error& ex) {
        fprintf(stderr,"图像转换成 PNG 格式发生错误：%s\n", ex.what());
        return 1;
    }

    return 0;
}
```

程序运行截图如图 3.1 所示。运行完毕，我们可以在工程目录下发现一张生成的名为“透明 Alpha 值图.png”的图片文件。

图 3.1 由 imwrite 生成的图

3.1.9 综合示例程序：图像的载入、显示与输出

本小节将给出一个综合示例，演示如何载入图像，进行简单的图像混合，显示图像，并且输出混合后的图像到 jpg 格式的文件中。

出于注重演示效果的原因，程序中图像混合的具体细节我们放到稍后的篇幅中再讲，现在先给大家看看混合的效果和源码。以下就是本节的综合示例程序，经过详细注释的代码非常简单明了。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
using namespace cv;


int main()
{
    //----------------------【一、图像的载入和显示】----------------------
    //  描述：以下三行代码用于完成图像的载入和显示
    //-------------------------------------------------------------------

    Mat girl=imread("girl.jpg"); //载入图像到 Mat
    namedWindow("【1】动漫图"); //创建一个名为 "【1】动漫图"的窗口
    imshow("【1】动漫图",girl);//显示名为 "【1】动漫图"的窗口

    //----------------------【二、初级图像混合】----------------------
    //  描述：二、初级图像混合
    //-------------------------------------------------------------------
    //载入图片
    Mat image= imread("dota.jpg",199);
    Mat logo= imread("dota_logo.jpg");
```

```
    //载入后先显示
    namedWindow("【2】原画图");
    imshow("【2】原画图",image);

    namedWindow("【3】logo 图");
    imshow("【3】logo 图",logo);

    // 定义一个 Mat 类型，用于存放，图像的 ROI
    Mat imageROI;
    //方法一
    imageROI= image(Rect(800,350,logo.cols,logo.rows));
    //方法二
    //imageROI=
image(Range(350,350+logo.rows),Range(800,800+logo.cols));

    // 将 logo 加到原图上
    addWeighted(imageROI,0.5,logo,0.3,0.,imageROI);

    //显示结果
    namedWindow("【4】原画+logo 图");
    imshow("【4】原画+logo 图",image);

    //-------------------------【三、图像的输出】-------------------------
    //  描述：将一个 Mat 图像输出到图像文件
    //----------------------------------------------------------------
    //输出一张 jpg 图片到工程目录下
    imwrite("由 imwrite 生成的图片.jpg",image);

    waitKey();

    return 0;
}
```

运行这个程序，会弹出 4 个在 OpenCV 中创建的窗口。

下面是运行截图。首先是图像载入和显示的演示，我们载入了一张动漫人物图，如图 3.2 所示。

图 3.2 动漫显示图

接着是载入一张 dota2 原画（图 3.3）和 dota2 的 logo（图 3.4），为图像融合做准备。

图 3.3 dota2 原画图窗口

图 3.4 logo 图

最终，经过处理，得到 dota2 原画+logo 的融合，并输出一张名为“由 imwrite 生成的图片.jpg”的图片到工程目录下。如图 3.5 所示。

图 3.5 融合后的效果图

3.2 滑动条的创建和使用

滑动条（Trackbar）是 OpenCV 动态调节参数特别好用的一种工具，它依附于窗口而存在。

由于 OpenCV 中并没有实现按钮的功能，所以很多时候，我们还可以用仅含 0-1 的滑动条来实现按钮的按下、弹起效果。

3.2.1 创建滑动条：createTrackbar()函数

createTrackbar 函数用于创建一个可以调整数值的滑动条（常常也被称作轨迹条），并将滑动条附加到指定的窗口上，使用起来很方便。需要记住，它往往会和一个回调函数配合起来使用。先看下它的函数原型，如下。

```
C++: int createTrackbar(conststring& trackbarname, conststring& winn
ame,
int* value, int count, TrackbarCallback onChange=0,void* userdata=0);
```

- 第一个参数，const string&类型的 trackbarname，轨迹条的名字，用来代表我们创建的轨迹条。
- 第二个参数，const string&类型的 winname，窗口的名字，表示这个轨迹条会依附到哪个窗口上，即对应 namedWindow()创建窗口时填的某一个窗口名。
- 第三个参数，int* 类型的 value，一个指向整型的指针，表示滑块的位置。在创建时，滑块的初始位置就是该变量当前的值。
- 第四个参数，int 类型的 count，表示滑块可以达到的最大位置的值。滑块最小位置的值始终为 0。
- 第五个参数，TrackbarCallback 类型的 onChange，它有默认值 0。这是一个指向回调函数的指针，每次滑块位置改变时，这个函数都会进行回调。并且这个函数的原型必须为 void XXXX(int，void*);，其中第一个参数是轨迹条的位置，第二个参数是用户数据（看下面的第六个参数）。如果回调是 NULL 指针，则表示没有回调函数的调用，仅第三个参数 value 有变化。
- 第六个参数，void*类型的 userdata，也有默认值 0。这个参数是用户传给回调函数的数据，用来处理轨迹条事件。如果使用的第三个参数 value 实参是全局变量的话，完全可以不去管这个 userdata 参数。

createTrackbar 函数为我们创建了一个具有特定名称和范围的轨迹条（Trackbar，或者说是滑块范围控制工具），指定一个和轨迹条位置同步的变量，而且要指定回调函数 onChange（第五个参数），在轨迹条位置改变的时候来调用这个回调函数，并且，创建的轨迹条显示在指定的 winname（第二个参数）所代表的窗口上。

至于回调函数，就是一个通过函数指针调用的函数。如果我们把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，就称其为回调函数。回调函数不由该函数的实现方直接调用，而是在特定的事件

或条件发生时由另外的一方调用，用于对该事件或条件进行响应。

在函数讲解之后，给大家一个createTrackbar函数使用的小例子作为参照。

```
//创建轨迹条
  createTrackbar("对比度：", "【效果图窗口】",&g_nContrastValue,
        300,on_Change);// g_nContrastValue为全局的整型变量，on_Change为
回调函数的函数名（在C/C++中，函数名为指向函数地址的指针）
```

接着，我们一起来欣赏一个完整的使用示例，它演示了如何用轨迹条来控制两幅图像的Alpha混合。

```
#include <opencv2/opencv.hpp>
#include "opencv2/highgui/highgui.hpp"
using namespace cv;

#define WINDOW_NAME "【线性混合示例】"        //为窗口标题定义的宏

//-------------------------【全局变量声明部分】-------------------------
//          描述：全局变量声明
//-----------------------------------------------------------------------
const int g_nMaxAlphaValue = 100;//Alpha值的最大值
int g_nAlphaValueSlider;//滑动条对应的变量
double g_dAlphaValue;
double g_dBetaValue;

//声明存储图像的变量
Mat g_srcImage1;
Mat g_srcImage2;
Mat g_dstImage;


//-------------------------【on_Trackbar()函数】-------------------------
//          描述：响应滑动条的回调函数
//-----------------------------------------------------------------------
void on_Trackbar( int, void* )
{
    //求出当前alpha值相对于最大值的比例
    g_dAlphaValue = (double) g_nAlphaValueSlider/g_nMaxAlphaValue ;
    //则beta值为1减去alpha值
    g_dBetaValue = ( 1.0 - g_dAlphaValue );

    //根据alpha和beta值进行线性混合
    addWeighted( g_srcImage1, g_dAlphaValue, g_srcImage2, g_dBetaValue,
0.0, g_dstImage);

    //显示效果图
    imshow( WINDOW_NAME, g_dstImage );
}

//-----------------------------【main()函数】--------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
```

```
int main( int argc, char** argv )
{
    //加载图像 (两图像的尺寸需相同)
    g_srcImage1 = imread("1.jpg");
    g_srcImage2 = imread("2.jpg");
    if( !g_srcImage1.data ) { printf("读取第一幅图片错误，请确定目录下是否有 imread 函数指定图片存在~！ \n"); return -1; }
    if( !g_srcImage2.data ) { printf("读取第二幅图片错误，请确定目录下是否有 imread 函数指定图片存在~！ \n"); return -1; }

    //设置滑动条初值为 70
    g_nAlphaValueSlider = 70;

    //创建窗体
    namedWindow(WINDOW_NAME, 1);

    //在创建的窗体中创建一个滑动条控件
    char TrackbarName[50];
    sprintf( TrackbarName, "透明值 %d", g_nMaxAlphaValue );

    createTrackbar( TrackbarName, WINDOW_NAME, &g_nAlphaValueSlider, g_nMaxAlphaValue, on_Trackbar );

    //结果在回调函数中显示
    on_Trackbar( g_nAlphaValueSlider, 0 );

    //按任意键退出
    waitKey(0);

    return 0;
}
```

运行此程序，我们可以通过调节滑动条的位置，来得到不同的混合效果。如图 3.6、图 3.7、图 3.8 所示。

图 3.6 透明值为 70 时的效果图

图 3.7 透明值为 100 时的效果图

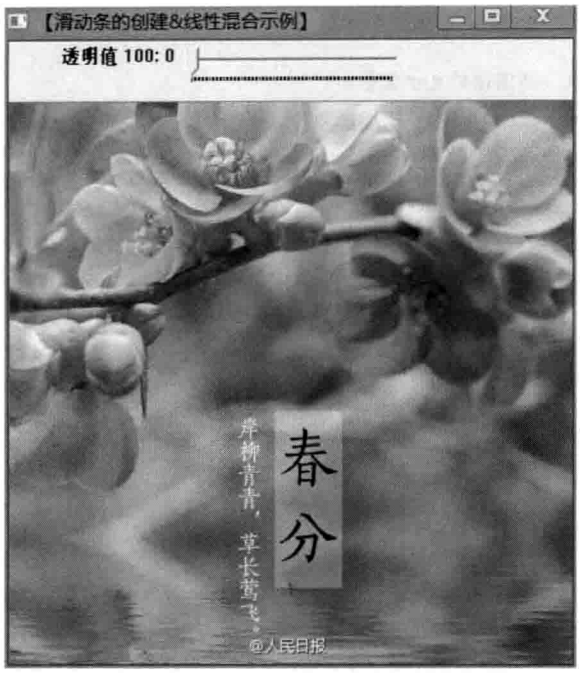


图 3.8 透明值为 0 时的效果图

3.2.2 获取当前轨迹条的位置：getTrackbarPos()函数

本小节介绍一个配合 createTrackbar 使用的函数——getTrackbarPos()，它用于获取当前轨迹条的位置。

下面这个函数用于获取当前轨迹条的位置并返回。

```
C++: int getTrackbarPos(conststring& trackbarname, conststring& winn
ame);
```

- 第一个参数，const string&类型的 trackbarname，表示轨迹条的名字。
- 第二个参数，const string&类型的 winname，表示轨迹条的父窗口的名称。

3.3 鼠标操作

OpenCV 中的鼠标操作和滑动条的消息映射方式很类似，都是通过一个中介函数配合一个回调函数来实现的。创建和指定滑动条回调函数的函数为 createTrackbar，而指定鼠标操作消息回调函数的函数为 SetMouseCallback。下面一起来了解一下它。

```
SetMouseCallback 函数的作用是为指定的窗口设置鼠标回调函数，原型如下。
C++: void setMouseCallback(conststring& winname, MouseCallback onMouse,
void* userdata=0 )
```

- 第一个参数，const string&类型的 winname，窗口的名字。
- 第二个参数，MouseCallback 类型的 onMouse，指定窗口里每次鼠标时间发

生的时候，被调用的函数指针。这个函数的原型的大概形式为 void Foo(int event, int x, int y, int flags, void* param)。其中 event 是 EVENT_+变量之一，x 和 y 是鼠标指针在图像坐标系（需要注意，不是窗口坐标系）中的坐标值，flags 是 EVENT_FLAG 的组合，param 是用户定义的传递到 SetMouseCallback 函数调用的参数。如 EVENT_MOUSEMOVE 为鼠标移动消息、EVENT_LBUTTONDOWN 为鼠标左键按下消息等。

在 OpenCV2 中，上述“EVENT_”之前可以加上“CV_”前缀。

- 第三个参数，void*类型的 userdata，用户定义的传递到回调函数的参数，有默认值 0。

下面看一个详细注释的示例程序，在实战中了解此函数的用法以及如何在 OpenCV 中使用鼠标进行交互。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------
#include <opencv2/opencv.hpp>
using namespace cv;

#define WINDOW_NAME "【程序窗口】"        //为窗口标题定义的宏


//-------------------------【全局函数声明部分】-------------------------
//      描述：全局函数的声明
//-------------------------------------------------------------------
void on_MouseHandle(int event, int x, int y, int flags, void* param);
void DrawRectangle( cv::Mat& img, cv::Rect box );
void ShowHelpText();

//-------------------------【全局变量声明部分】-------------------------
//      描述：全局变量的声明
//-------------------------------------------------------------------
Rect g_rectangle;
bool g_bDrawingBox = false;//是否进行绘制
RNG g_rng(12345);


//-------------------------【main()函数】-------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------
int main( int argc, char** argv )
{

    //【1】准备参数
    g_rectangle = Rect(-1,-1,0,0);
    Mat srcImage(600, 800,CV_8UC3), tempImage;
    srcImage.copyTo(tempImage);
```

```
    g_rectangle = Rect(-1,-1,0,0);
    srcImage = Scalar::all(0);

    //【2】设置鼠标操作回调函数
    namedWindow( WINDOW_NAME );
    setMouseCallback(WINDOW_NAME,on_MouseHandle,(void*)&srcImage);

    //【3】程序主循环，当进行绘制的标识符为真时，进行绘制
    while(1)
    {
        srcImage.copyTo(tempImage);//复制源图到临时变量
        if( g_bDrawingBox ) DrawRectangle( tempImage, g_rectangle );//
当进行绘制的标识符为真，则进行绘制
        imshow( WINDOW_NAME, tempImage );
        if( waitKey( 10 ) == 27 ) break;//按下ESC键，程序退出
    }
    return 0;
}


//-----------------------【on_MouseHandle()函数】-----------------------
//      描述：鼠标回调函数，根据不同的鼠标事件进行不同的操作
//-----------------------------------------------------------------------
void on_MouseHandle(int event, int x, int y, int flags, void* param)
{

    Mat& image = *(cv::Mat*) param;
    switch( event)
    {
        //鼠标移动消息
        case EVENT_MOUSEMOVE:
         {
             if( g_bDrawingBox )//如果是否进行绘制的标识符为真，则记录下长和
宽到RECT型变量中
             {
                 g_rectangle.width = x-g_rectangle.x;
                 g_rectangle.height = y-g_rectangle.y;
             }
         }
              break;

        //左键按下消息
        case EVENT_LBUTTONDOWN:
         {
             g_bDrawingBox = true;
             g_rectangle =Rect( x, y, 0, 0 );//记录起始点
         }
             break;

        //左键抬起消息
        case EVENT_LBUTTONUP:
         {
```

```
            g_bDrawingBox = false;//置标识符为 false
            //对宽和高小于 0 的处理
            if( g_rectangle.width < 0 )
            {
                g_rectangle.x += g_rectangle.width;
                g_rectangle.width *= -1;
            }

            if( g_rectangle.height < 0 )
            {
                g_rectangle.y += g_rectangle.height;
                g_rectangle.height *= -1;
            }
            //调用函数进行绘制
            DrawRectangle( image, g_rectangle );
        }
            break;

    }
}


//---------------------【DrawRectangle()函数】----------------------
//      描述：自定义的矩形绘制函数
//-------------------------------------------------------------------
void DrawRectangle( cv::Mat& img, cv::Rect box )
{
    rectangle(img,box.tl(),box.br(),Scalar(g_rng.uniform(0,255),
g_rng.uniform(0,255), g_rng.uniform(0,255)));//随机颜色
}
```

首先一起看看程序运行效果。我们可以通过鼠标左键的按下和松开来在黑色的窗口中绘制出一个一个彩色的矩形。如图 3.9、图 3.10 所示。

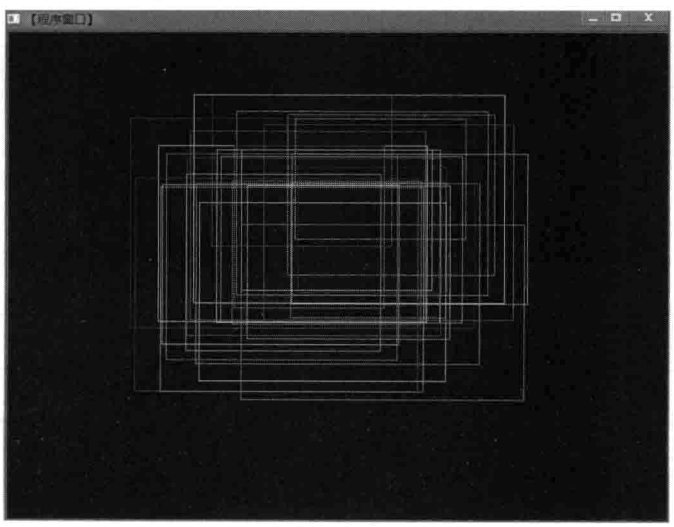


图 3.9 运行截图（1）

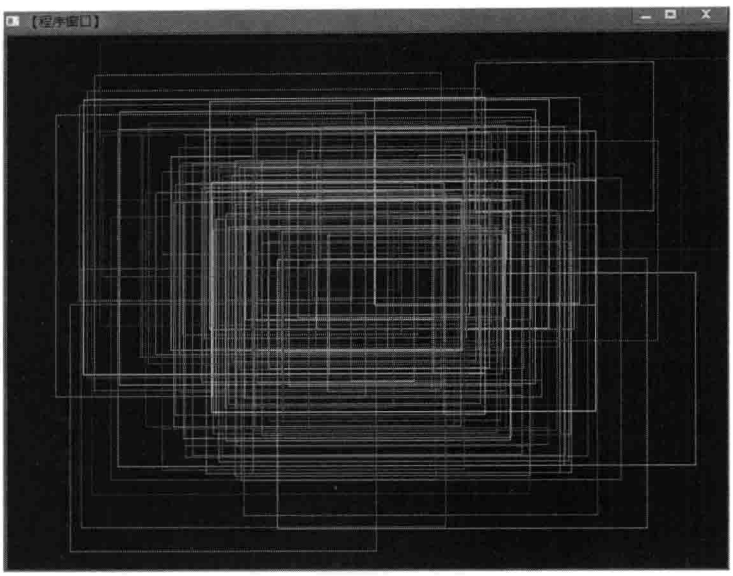


图 3.10 运行截图（2）

上述示例程序中的 on_MouseHandle 就是我们的鼠标消息回调函数。其中用一个 switch 语句指出了各种类型的鼠标消息，如鼠标移动消息 EVENT_MOUSEMOVE、左键按下消息 EVENT_LBUTTONDOWN、左键抬起消息 EVENT_LBUTTONUP，并对其进行了处理，以得到随机颜色的矩形绘制的功能。

3.4 本章小结

本章中，我们学习了 OpenCV 的高层 GUI 图形用户界面模块 highgui 中最重要的几个方面，分别是图像的载入、显示与输出图像到文件，以及如何使用滑动条如何进行鼠标操作。

本章核心函数清单

函数名称	用途	讲解章节
imread	用于读取文件中的图片到 OpenCV 中	3.1.4
imshow	在指定的窗口中显示一幅图像	3.1.5
namedWindow	用于创建一个窗口	3.1.7
imwrite	输出图像到文件	3.1.8
createTrackbar	用于创建一个可以调整数值的轨迹条	3.2.1
getTrackbarPos	用于获取轨迹条的当前位置	3.2.2
SetMouseCallback	为指定的窗口设置鼠标回调函数	3.3

本章示例程序清单

示例程序序号	程序说明	对应章节
15	用 imwrite 函数生成 png 透明图	3.1.8
16	综合示例程序：图像的载入、显示与输出	3.1.9
17	为程序界面添加滑动条	3.2.1
18	鼠标操作	3.3

第二部分
初探 core 组件

OpenCV 中的 core 组件是核心功能模块，第二部分包含如下内容：

- 基本数据结构
- 动态数据结构
- 绘图函数
- 数组操作相关函数
- 辅助功能与系统函数和宏
- 与 OpenGL 的互操作

第 4 章
OpenCV 数据结构与基本绘图

导读

本章中，你将学到：

- 基础图像容器 Mat 的用法
- OpenCV 中的多种格式化输出方法
- 常用的数据结构
- 基本绘图操作

4.1 基础图像容器 Mat

4.1.1 数字图像存储概述

我们可以通过各种各样的方法从现实世界获取到数字图像，如借助相机、扫描仪、计算机摄像头或磁共振成像等。通常由显示屏上看到的都是真实而漂亮的图像，但是这些图像在转化到我们的数字设备中时，记录的却是图像中的每个点的数值。

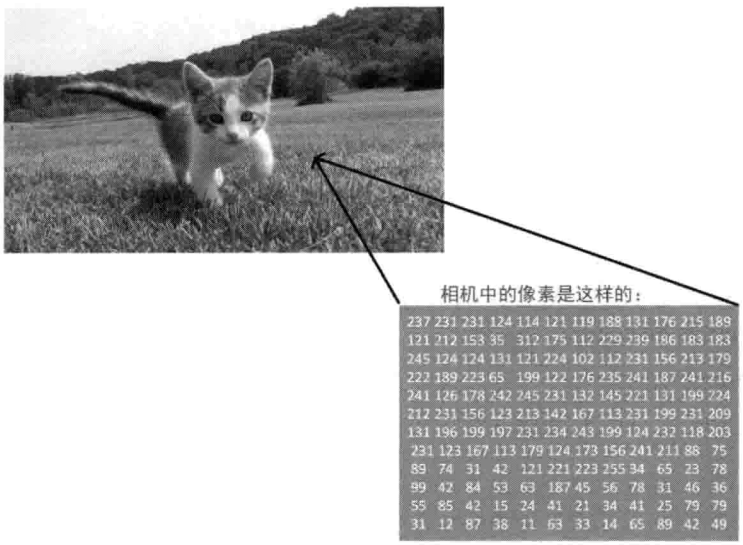


图 4.1 图像的存储图示

比如在图 4.1 中你可以看到草坪的颜色是一个包含众多强度值的像素点矩阵。可以这样说，矩阵就是图像在数码设备中的表现形式。OpenCV 作为一个计算机视觉库，其主要的工作是处理和操作并进一步了解这些形式和信息。因此，理解 OpenCV 是如何存储和处理图像是非常有必要的。那么，就让我们来进一步了解和学习。

4.1.2 Mat 结构的使用

自 2001 年以来，OpenCV 的函数库一直是基于 C 接口构建的，因此在最初的几个 OpenCV 版本中，一般使用名为 IplImage 的 C 语言结构体在内存中存储图像。时至今日，这仍出现在大多数的旧版教程和教学材料中，如最经典的 OpenCV 教程《Learning OpenCV》。

对于 OpenCV1.X 时代的基于 C 语言接口而建的图像存储格式 IplImage*，如果在退出前忘记 release 掉的话，就会照成内存泄露，而且用起来有些不便，我们在调试的时候，往往要花费很多时间在手动释放内存的问题上。虽然对于小型的程序来说，手动管理内存不是问题，但一旦需要书写和维护的代码越来越庞大，

我们便会开始越来越多地纠缠于内存管理的问题，而不是着力解决最终的开发目标。这，就有些舍本逐末的感觉了。

幸运的是，C++出现了，并且带来了类的概念，这使我们有了另外一个选择：自动的内存管理（非严格意义上的）。这对于广大图像处理领域的研究者来说，的确是一件可喜可贺的事情。也就是说，OpenCV 在 2.0 版本中引入了一个新的 C++接口，利用自动内存管理给出了解决问题的新方法。使用此方法，我们不需要再纠结在管理内存的问题，而且代码会变得干净而简洁。

但 C++接口唯一的不足是：当前许多嵌入式开发系统只支持 C 语言。所以，当开发目标不是仅能使用 C 语言作为开发语言时，便没有必要使用旧的 C 语言接口了，除非你真的很有自信。

从 OpenCV 踏入 2.0 时代，使用 Mat 类数据结构作为主打之后，OpenCV 变得越发像需要很少编程涵养的 Matlab 那样，上手很方便。甚至有些函数名称都和 Matlab 一样，比如大家所熟知的 imread、imwrite、imshow 等函数。

关于 Mat 类，首先我们要知道的是：

（1）不必再手动为其开辟空间。

（2）不必再在不需要时立即将空间释放。

这里指的是手动开辟空间并非必须，但它依旧是存在的——大多数 OpenCV 函数仍会手动地为输出数据开辟空间。当传递一个已经存在的 Mat 对象时，开辟好的矩阵空间会被重用。也就是说，我们每次都使用大小正好的内存来完成任务。

总而言之，Mat 是一个类，由两个数据部分组成：矩阵头（包含矩阵尺寸、存储方法、存储地址等信息）和一个指向存储所有像素值的矩阵（根据所选存储方法的不同，矩阵可以是不同的维数）的指针。矩阵头的尺寸是常数值，但矩阵本身的尺寸会依图像的不同而不同，通常比矩阵头的尺寸大数个数量级。因此，当在程序中传递图像并创建副本时，大的开销是由矩阵造成的，而不是信息头。OpenCV 是一个图像处理库，囊括了大量的图像处理函数，为了解决问题通常要使用库中的多个函数，因此在函数中传递图像是常有的事。同时不要忘了我们正在讨论的是计算量很大的图像处理算法，因此，除非万不得已，不应该进行大图像的复制，因为这会降低程序的运行速度。

为了解决此问题，OpenCV 使用了引用计数机制。其思路是让每个 Mat 对象有自己的信息头，但共享同一个矩阵。这通过让矩阵指针指向同一地址而实现。而拷贝构造函数则只复制信息头和矩阵指针，而不复制矩阵。

来看下面这段代码。

```
Mat A, C; // 仅创建信息头部分
A = imread("1.jpg", CV_LOAD_IMAGE_COLOR); // 这里为矩阵开辟内存
Mat B(A); //使用拷贝构造函数
C = A; //赋值运算符
```

以上代码中的所有 Mat 对象最终都指向同一个也是唯一一个数据矩阵。虽然它们的信息头不同，但通过任何一个对象所做的改变也会影响其他对象。实际上，不同的对象只是访问相同数据的不同途径而已。这里还要提及一个比较棒的功能：我们可以创建只引用部分数据的信息头。比如想要创建一个感兴趣区域（ROI），只需要创建包含边界信息的信息头：

```
Mat D (A, Rect(10, 10, 100, 100) ); // 使用矩形界定
Mat E = A(Range:all(), Range(1,3)); // 用行和列来界定
```

现在你也许会问：如果矩阵属于多个 Mat 对象，那么当不再需要它时，谁来负责清理呢？

简单的回答是：最后一个使用它的对象。通过引用计数机制来实现。我们无论什么时候复制一个 Mat 对象的信息头，都会增加矩阵的引用次数。反之，当一个头被释放之后，这个计数被减一；当计数值为零，矩阵会被清理。但某些时候你仍会想复制矩阵本身（不只是信息头和矩阵指针），这时可以使用函数 clone() 或者 copyTo()。

```
Mat F = A.clone();
Mat G;
A.copyTo(G);
```

现在改变 F 或者 G 就不会影响 Mat 信息头所指向的矩阵。本小节可总结为如下 4 个要点。

- OpenCV 函数中输出图像的内存分配是自动完成的(如果不特别指定的话)。
- 使用 OpenCV 的 C++接口时不需要考虑内存释放问题。
- 赋值运算符和拷贝构造函数（构造函数）只复制信息头。
- 使用函数 clone()或者 copyTo()来复制一幅图像的矩阵。

4.1.3 像素值的存储方法

本节我们将讲解如何存储像素值。存储像素值需要指定颜色空间和数据类型。其中，颜色空间是指针对一个给定的颜色，如何组合颜色元素以对其编码。最简单的颜色空间要属灰度级空间，只处理黑色和白色，对它们进行组合便可以产生不同程度的灰色。

对于彩色方式则有更多种类的颜色空间，但不论哪种方式都是把颜色分成三个或者四个基元素，通过组合基元素可以产生所有的颜色。RGB 颜色空间是最常用的一种颜色空间，这归功于它也是人眼内部构成颜色的方式。它的基色是红色、绿色和蓝色，有时为了表示透明颜色也会加入第四个元素 alpha（A）。

颜色系统有很多，它们各有优势，具体如下。

- RGB 是最常见的，这是因为人眼采用相似的工作机制，它也被显示设备所采用
- HSV 和 HLS 把颜色分解成色调、饱和度和亮度/明度。这是描述颜色更自然的方式，比如可以通过抛弃最后一个元素，使算法对输入图像的光照条

件不敏感

- YCrCb 在 JPEG 图像格式中广泛使用
- CIE L*a*b*是一种在感知上均匀的颜色空间，它适合用来度量两个颜色之间的距离

每个组成元素都有其自己的定义域，而定义域取决于其数据类型，如何存储一个元素决定了我们在其定义域上能够控制的精度。最小的数据类型是 char，占一个字节或者 8 位，可以是有符号型（0 到 255 之间）或无符号型（-127 到+127 之间）。尽管使用三个 char 型元素已经可以表示 1600 万种可能的颜色（使用 RGB 颜色空间），但若使用 float（4 字节，32 位）或 double（8 字节，64 位）则能给出更加精细的颜色分辨能力。但同时也要切记，增加元素的尺寸也会增加图像所占的内存空间。

4.1.4 显式创建 Mat 对象的七种方法

在之前的章节中，我们已经讲解了如何使用函数 imwrite()函数将一个矩阵写入图像文件中。但是作为 debug，更加方便的方式是看实际值，我们可以通过 Mat 的运算符“<<”来实现。但要记住，Mat 的运算符“<<”只对二维矩阵有效。

Mat 不但是一个非常有用的图像容器类，同时也是一个通用的矩阵类，我们也可以用它来创建和操作多维矩阵。

创建一个 Mat 对象有多种方法，列举如下。

1.【方法一】使用 Mat()构造函数

最常用的方法是直接使用 Mat()构造函数，这种方法简单明了，示范代码如下。

```
Mat M(2,2, CV_8UC3, Scalar(0,0,255));
cout << "M = " << endl << " " << M << endl << endl;
```

上述代码的运行结果如图 4.2 所示。

```
M =
 [0, 0, 255, 0, 0, 255;
  0, 0, 255, 0, 0, 255]
```

图 4.2 方法一示例代码运行结果

对于二维多通道图像，首先要定义其尺寸，即行数和列数。然后，需要指定存储元素的数据类型以及每个矩阵点的通道数。为此，依据下面的规则有多种定义：

```
CV_[The number of bits per item][Signed or Unsigned][Type Prefix]C[The
channel number]
```

即：

CV_[位数][带符号与否][类型前缀]C[通道数]

比如 CV_8UC3 表示使用 8 位的 unsigned char 型，每个像素由三个元素组成三通道。而预先定义的通道数可以多达四个。另外，Scalar 是个 short 型的向量，

能使用指定的定制化值来初始化矩阵，它还可以用于表示颜色，后文有详细讲解。当然，若需要更多通道数，可以使用大写的宏并把通道数放在小括号中，如方法二中的代码所示。

2.【方法二】在 C\C++中通过构造函数进行初始化

这种方法为在 C\C++ 中通过构造函数进行初始化，示范代码如下。

```
int sz[3] = {2,2,2};
Mat L(3,sz, CV_8UC, Scalar::all(0));
```

上面的例子演示了如何创建一个超过两维的矩阵：指定维数，然后传递一个指向一个数组的指针，这个数组包含每个维度的尺寸；后续的两个参数与方法一中的相同。

3.【方法三】为已存在的 IplImage 指针创建信息头

方法三是为已存在的 IplImage 指针创建信息头，示范代码如下。

```
IplImage* img = cvLoadImage("1.jpg", 1);
Mat mtx(img); // 转换 IplImage*-> Mat
```

4.【方法四】利用 Create()函数

方法四是利用 Mat 类中的 Create()成员函数进行 Mat 类的初始化操作，示范代码如下。

```
M.create(4,4, CV_8UC(2));
cout << "M = "<< endl << " "  << M << endl << endl;
```

上述代码的运行结果如图 4.3 所示。

```
M =
 [205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205;
  205, 205, 205, 205, 205, 205, 205, 205]
```

图 4.3 方法四示例代码运行结果

需要注意的是，此创建方法不能为矩阵设初值，只是在改变尺寸时重新为矩阵数据开辟内存而已。

5.【方法五】采用 Matlab 式的初始化方式

方法五采用 Matlab 形式的初始化方式：zeros()，ones()，eyes()。使用以下方式指定尺寸和数据类型：

```
Mat E = Mat::eye(4, 4, CV_64F);
cout << "E = " << endl << " " << E << endl << endl;

Mat O = Mat::ones(2, 2, CV_32F);
cout << "O = " << endl << " " << O << endl << endl;

Mat Z = Mat::zeros(3,3, CV_8UC1);
cout << "Z = " << endl << " " << Z << endl << endl;
```

上述代码的运行结果如图 4.4 所示。

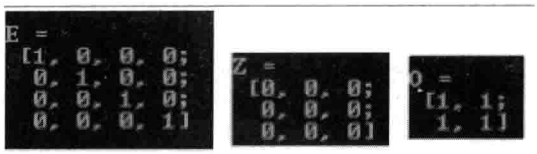


图 4.4 方法五示例代码运行结果

6.【方法六】对小矩阵使用逗号分隔式初始化函数

方法六为对小矩阵使用逗号分隔式初始化函数，示范代码如下。

```
Mat C = (Mat_<double>(3,3) << 0, -1, 0, -1, 5, -1, 0, -1, 0);
cout << "C = " << endl << " " << C << endl << endl;
```

上述代码的运行结果如图 4.5 所示。

图 4.5 方法六示例代码运行结果

7.【方法七】为已存在的对象创建新信息头

方法七为使用成员函数 clone()或者 copyTo()为一个已存在的 Mat 对象创建一个新的信息头，示范代码如下。

```
Mat RowClone = C.row(1).clone();
cout << "RowClone = " << endl << " " << RowClone << endl << endl;
```

上述代码的运行结果如图 4.6 所示。

图 4.6 方法七示例代码运行结果

4.1.5 OpenCV 中的格式化输出方法

在上一个例子中我们可以看到有默认的格式选项，同时，OpenCV 也提供了风格各异的格式化输出方法，本小节将对这些方法一一进行演示和列举。

首先是下面代码中将要使用的r矩阵的定义。需要注意，我们可以通过用randu()函数产生的随机值来填充矩阵，需要给定一个上限和下限来确保随机值在期望的范围内。

```
Mat r = Mat(10, 3, CV_8UC3);
randu(r, Scalar::all(0), Scalar::all(255));
```

初始化完 r 矩阵，下面便开始对输出风格的讲解。

1.【风格一】OpenCV 默认风格

风格一为 OpenCV 默认风格的输出方法，如下。

```
cout << "r (OpenCV 默认风格) = " << r << ";" << endl << endl;
```

上述代码的运行结果如图 4.7 所示。

```
r (OpenCV默认风格) = [91, 2, 79, 179, 52, 205, 236, 8, 181;
 239, 26, 248, 207, 218, 45, 183, 158, 101;
 102, 18, 118, 68, 210, 139, 198, 207, 211;
 181, 162, 197, 191, 196, 40, 7, 243, 230;
 45, 6, 48, 173, 242, 125, 175, 90, 63;
 90, 22, 112, 221, 167, 224, 113, 208, 123;
 214, 35, 229, 6, 143, 138, 98, 81, 118;
 187, 167, 140, 218, 178, 23, 43, 133, 154;
 150, 76, 101, 8, 38, 238, 84, 47, 7;
```

图 4.7 OpenCV 默认风格运行结果

2.【风格二】Python 风格

风格二为 Python 风格的输出方法，如下。

```
//此句代码的 OpenCV2 版为：
cout << "r (Python 风格) = " << format(r,"python") << ";" << endl << endl;

//此句代码的 OpenCV3 版为：
cout << "r (Python 风格) = " << format(r, Formatter::FMT_PYTHON) << ";"
<< endl << endl;
```

上述代码的运行结果如图 4.8 所示。

```
r (Python风格) = [[[91, 2, 79], [179, 52, 205], [236, 8, 181]],
 [[239, 26, 248], [207, 218, 45], [183, 158, 101]],
 [[102, 18, 118], [68, 210, 139], [198, 207, 211]],
 [[181, 162, 197], [191, 196, 40], [7, 243, 230]],
 [[45, 6, 48], [173, 242, 125], [175, 90, 63]],
 [[90, 22, 112], [221, 167, 224], [113, 208, 123]],
 [[214, 35, 229], [6, 143, 138], [98, 81, 118]],
 [[187, 167, 140], [218, 178, 23], [43, 133, 154]],
 [[150, 76, 101], [8, 38, 238], [84, 47, 7]],
 [[117, 246, 163], [237, 69, 129], [60, 101, 41]]];
```

图 4.8 Python 风格运行结果

3.【风格三】逗号分隔风格 (Comma separated values，CSV)

风格三为逗号分隔风格的输出方法，如下。

```
//此句代码的 OpenCV2 版为：
cout <<"r(逗号分隔风格) = " << format(r,"csv") << ";" << endl << endl;

//此句代码的 OpenCV3 版为：
cout << "r (逗号分隔风格) = " << format(r, Formatter::FMT_CSV   )<< ";"
<< endl<< endl;
```

上述代码的运行结果如图 4.9 所示。

```
r (逗号分隔风格) = 91, 2, 79, 179, 52, 205, 236, 8, 181
  239, 26, 248, 207, 218, 45, 183, 158, 101
  102, 18, 118, 68, 210, 139, 198, 207, 211
  181, 162, 197, 191, 196, 40, 7, 243, 230
  45, 6, 48, 173, 242, 125, 175, 90, 63
  90, 22, 112, 221, 167, 224, 113, 208, 123
  214, 35, 229, 6, 143, 138, 98, 81, 118
  187, 167, 140, 218, 178, 23, 43, 133, 154
  150, 76, 101, 8, 38, 238, 84, 47, 7
  117, 246, 163, 237, 69, 129, 60, 101, 41
;
```

图 4.9 逗号分隔风格运行结果

4.【风格四】Numpy 风格

风格四为 Numpy 风格的输出方法，如下。

```
//此句代码的 OpenCV2 版为:
cout << "r (Numpy 风格) = " << format(r,"numpy") << ";" << endl << endl;

//此句代码的 OpenCV3 版为:
cout << "r (Numpy 风格) = " <<  format(r, Formatter::FMT_NUMPY )<< ";"
<< endl << endl;
```

上述代码的运行结果如图 4.10 所示。

```
r (Numpy风格) = array([[[91, 2, 79], [179, 52, 205], [236, 8, 181]],
  [[239, 26, 248], [207, 218, 45], [183, 158, 101]],
  [[102, 18, 118], [68, 210, 139], [198, 207, 211]],
  [[181, 162, 197], [191, 196, 40], [7, 243, 230]],
  [[45, 6, 48], [173, 242, 125], [175, 90, 63]],
  [[90, 22, 112], [221, 167, 224], [113, 208, 123]],
  [[214, 35, 229], [6, 143, 138], [98, 81, 118]],
  [[187, 167, 140], [218, 178, 23], [43, 133, 154]],
  [[150, 76, 101], [8, 38, 238], [84, 47, 7]],
  [[117, 246, 163], [237, 69, 129], [60, 101, 41]]], type='uint8');
```

图 4.10 Numpy 风格运行结果

5.【风格五】C 语言风格

风格五为 C 语言风格的输出方法，如下。

```
//此句代码的 OpenCV2 版为:
cout << "r (C 语言风格) = " << format(r,"C") << ";" << endl << endl;

//此句代码的 OpenCV3 版为:
    cout << "r (C 语言风格) = " <<  format(r, Formatter::FMT_C     ) << ";"
<< endl << endl;
```

上述代码的运行结果如图 4.11 所示。

```
r (C语言风格) = {91, 2, 79, 179, 52, 205, 236, 8, 181,
  239, 26, 248, 207, 218, 45, 183, 158, 101,
  102, 18, 118, 68, 210, 139, 198, 207, 211,
  181, 162, 197, 191, 196, 40, 7, 243, 230,
  45, 6, 48, 173, 242, 125, 175, 90, 63,
  90, 22, 112, 221, 167, 224, 113, 208, 123,
  214, 35, 229, 6, 143, 138, 98, 81, 118,
  187, 167, 140, 218, 178, 23, 43, 133, 154,
  150, 76, 101, 8, 38, 238, 84, 47, 7,
  117, 246, 163, 237, 69, 129, 60, 101, 41};
```

图 4.11 C 语言风格运行结果

4.1.6 输出其他常用数据结构

之前我们讲解了如何输出 Mat 类型，其实，OpenCV 同样支持使用运算符“<<”来打印其他常用的 OpenCV 数据结构，本小节会通过代码对其中典型的几种进行讲解。

1. 定义和输出二维点

首先看看二维点的定义和输出方法：

```
Point2f p(6, 2);
cout << "【二维点】p = " << p << ";\n" << endl;
```

上述代码的运行结果如图 4.12 所示。

```
【2维点】p = [6, 2];
```

图 4.12 输出二维点

2. 定义和输出三维点

以下是三维点的定义和输出方法：

```
Point3f p3f(8, 2, 0);
cout << "【三维点】p3f = " << p3f << ";\n" << endl;
```

上述代码的运行结果如图 4.13 所示。

```
【三维点】p3f = [8, 2, 0];
```

图 4.13 输出三维点

3. 定义和输出基于 Mat 的 std::vector

接着是基于 Mat 类的 std::vector 的定义和输出方法：

```
vector<float> v;
v.push_back(3);
v.push_back(5);
v.push_back(7);

cout << "【基于 Mat 的 vector】shortvec = " << Mat(v) << ";\n"<<endl;
```

上述代码的运行结果如图 4.14 所示。

```
【基于Mat的vector】shortvec = [3;
 5;
 7];
```

图 4.14 输出基于 Mat 的 vector

4. 定义和输出 std::vector 点

最后看看如何定义和输出存放着点的 vector 容器，以存放二维点 Point2f 为例：

```
vector<Point2f> points(20);
for (size_t i = 0; i < points.size(); ++i)
```

```
points[i] = Point2f((float)(i * 5), (float)(i % 7));

cout << "【二维点向量】points = " << points<<";";
```

上述代码的运行结果如图 4.15 所示。

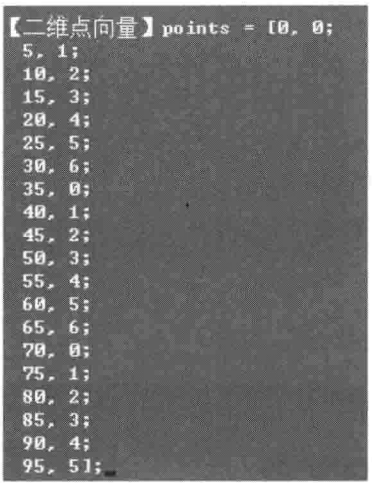


图 4.15 输出 std::vector 点

4.1.7 示例程序：基础图像容器 Mat 类的使用

本小节介绍的这些代码片段都被整理放到了一个短小的示例程序中，大家可以在书本配套示例程序包中找到此程序并运行。

运行截图之一如图 4.16 所示。

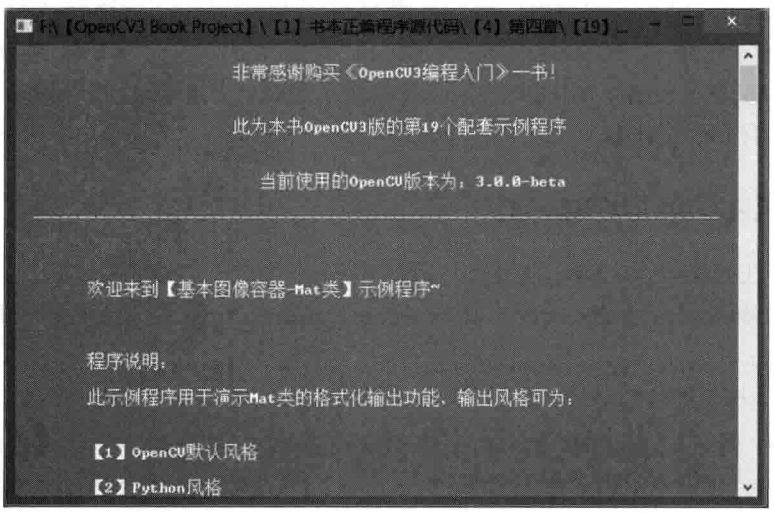


图 4.16 示例程序运行截图

4.2 常用数据结构和函数

本节我们将简单讲解 OpenCV 中常用的数据结构与函数。

4.2.1 点的表示：Point 类

Point 类数据结构表示了二维坐标系下的点，即由其图像坐标 x 和 y 指定的 2D 点。用法如下：

```
Point point;
point.x = 10;
point.y = 8;
```

或者

```
Point point = Point(10, 8);
```

另外，在 OpenCV 中有如下定义：

```
typedef Point_<int> Point2i;
typedef Point2i Point;
typedef Point_<float> Point2f;
```

所以，Point_<int>、Point2i、Point 互相等价，Point_<float>、Point2f 互相等价。

4.2.2 颜色的表示：Scalar 类

Scalar()表示具有 4 个元素的数组，在 OpenCV 中被大量用于传递像素值，如 RGB 颜色值。而 RGB 颜色值为三个参数，其实对于 Scalar 函数来说，如果用不到第四个参数，则不需要写出来；若只写三个参数，OpenCV 会认为我们就想表示三个参数。

来看个例子。如果给出以下颜色参数表达式：

```
Scalar( a, b, c )
```

那么定义的 RGB 颜色值：红色分量为 c，绿色分量为 b，蓝色分量为 a。

Scalar 类的源头为 Scalar_类，而 Scalar_类是 Vec4x 的一个变种，我们常用的 Scalar 其实就是 Scalar_<double>。这就解释了为什么很多函数的参数输入可以是 Mat，也可以是 Scalar。

4.2.3 尺寸的表示：Size 类

通过在代码中对 Size 类进行“转到定义”操作，我们可以在……\opencv\sources\modules\core\include\opencv2\core\core.hpp 路径下，找到 Size 类相关的源代码：

```
typedef Size_<int> Size2i;
typedef Size2i Size;
```

其中，Size_ 是个模板类，在这里 Size_<int>表示其类体内部的模板所代表的类型为 int。那这两句代码的意思，就是首先给已知的数据类型 Size_<int>起个新名字，叫 Size2i。然后又给已知的数据类型 Size2i 起个新名字，叫 Size。所以，连起来就是，Size_<int>、Size2i、Size 这三个类型名等价。

然后我们追根溯源，找到 Size_模板类的定义：

```
template<typename _Tp> class Size_
{
```

```
public:
  typedef _Tp value_type;

  //不同的构造函数定义
  Size_();
  Size_(_Tp _width, _Tp _height);
  Size_(const Size_& sz);
  Size_(const CvSize& sz);
  Size_(const CvSize2D32f& sz);
  Size_(const Point_<_Tp>& pt);

  Size_& operator = (const Size_& sz);
  //区域(width*height)
  _Tp area() const;

  //转化为另一种数据类型
   template<typename _Tp2> operator Size_<_Tp2>() const;

  //转换为旧式的 OpenCV 类型
  operator CvSize() const;
  operator CvSize2D32f() const;

  _Tp width, height; //宽度和高度，常用属性
};
```

可以看到 Size_模板类的内部又是重载了一些构造函数，其中，我们使用频率最高的是下面这个构造函数：

```
Size_(_Tp _width, _Tp _height);
```

另外，代码末尾定义了模板类型的宽度和高度：

```
_Tp width, height; //宽度和高度
```

于是我们可以用 *XXX*.width 和 *XXX*.height 来分别表示其宽度和高度。

下面给出一个示例，方便大家理解。

```
Size(5, 5);//构造出的 Size 宽度和高度都为 5，即 XXX.width 和 XXX.height 都为 5
```

4.2.4 矩形的表示：Rect 类

Rect 类的成员变量有 x、y、width、height，分别为左上角点的坐标和矩形的宽和高。常用的成员函数有：Size()返回值为 Size；area()返回矩形的面积；contains(Point)判断点是否在矩形内；inside(Rect)函数判断矩形是否在该矩形内；tl()返回左上角点坐标；br()返回右下角点坐标。值得注意的是，如果想求两个矩形的交集和并集，可以用如下格式：

```
Rect rect = rect1 & rect2;
Rect rect = rect1 | rect2;
```

如果想让矩形进行平移操作和缩放操作，甚至可以这样写：

```
Rect rectShift = rect + point;
```

```
Rect rectScale = rect + size;
```

4.2.5 颜色空间转换：cvtColor()函数

cvtColor()函数是 OpenCV 里的颜色空间转换函数，可以实现 RGB 颜色向 HSV、HSI 等颜色空间的转换，也可以转换为灰度图像。

原型如下：

```
C++: void cvtColor(InputArray src, OutputArray dst, int code, int
dstCn=0)
```

第一个参数为输入图像，第二个参数为输出图像，第三个参数为颜色空间转换的标识符（具体见表 4.1），第四个参数为目标图像的通道数，若该参数是 0，表示目标图像取源图像的通道数。下面是一个调用示例：

```
//此句代码的 OpenCV2 版为：
cvtColor(srcImage,dstImage, CV_GRAY2BGR);//转换原始图为灰度图
//此句代码的 OpenCV3 版为：
cvtColor(srcImage,dstImage, COLOR_GRAY2BGR);//转换原始图为灰度图
```

而随着 OpenCV 版本的升级，cvtColor()函数对于颜色空间种类的支持也是越来越多。其标识符列举如表 4.1 所示。

表 4.1 cvtColor()函数标识符（OpenCV2 版）

转换类型	标识符列举
RGB <--> BGR	CV_BGR2BGRA 、 CV_RGB2BGRA 、 CV_BGRA2RGBA 、 CV_BGR2BGRA、CV_BGRA2BGR
RGB <--> 5X5	CV_BGR5652RGBA、CV_BGR2RGB555 等
RGB <---> Gray	CV_RGB2GRAY 、 CV_GRAY2RGB 、 CV_RGBA2GRAY 、 CV_GRAY2RGBA
RGB <--> CIE XYZ	CV_BGR2XYZ 、 CV_RGB2XYZ 、 CV_XYZ2BGR 、 CV_XYZ2RGB
RGB <--> YCrCb（YUV） JPEG	CV_RGB2YCrCb 、 CV_RGB2YCrCb 、 CV_YCrCb2BGR 、 CV_YCrCb2RGB、CV_RGB2YUV（可将 YCrCb 用 YUV 替代）
RGB <--> HSV	CV_BGR2HSV 、 CV_RGB2HSV 、 CV_HSV2BGR 、 CV_HSV2RGB
RGB <--> HLS	CV_BGR2HLS、CV_RGB2HLS、CV_HLS2BGR、CV_HLS2RGB
RGB <--> CIE L*a*b*	CV_BGR2Lab、CV_RGB2Lab、CV_Lab2BGR、CV_Lab2RGB
RGB <--> CIE L*u*v	CV_BGR2Luv、CV_RGB2Luv、CV_Luv2BGR、CV_Luv2RGB
RGB <--> Bayer	CV_BayerBG2BGR、CV_BayerGB2BGR、CV_BayerRG2BGR、CV_BayerGR2BGR、CV_BayerBG2RGB、CV_BayerGB2RGB、CV_BayerRG2RGB、CV_BayerGR2RGB（在 CCD 和 CMOS 上常用的 Bayer 模式）
YUV420 <--> RGB	CV_YUV420sp2BGR 、 CV_YUV420sp2RGB 、 CV_YUV420i2BGR、CV_YUV420i2RGB

表 4.2 cvtColor()函数标识符（OpenCV3 版）

转换类型	标识符列举
RGB <--> BGR	COLOR_BGR2BGRA 、 COLOR_RGB2BGRA 、 COLOR_BGRA2RGBA 、 COLOR_BGR2BGRA 、 COLOR_BGRA2BGR
RGB <--> 5X5	COLOR _BGR5652RGBA、COLOR _BGR2RGB555 等
RGB <---> Gray	COLOR_RGB2GRAY 、 COLOR_GRAY2RGB 、 COLOR_RGBA2GRAY、COLOR_GRAY2RGBA
RGB <--> CIE XYZ	COLOR_BGR2XYZ、COLOR_RGB2XYZ、COLOR_XYZ2BGR、COLOR_XYZ2RGB
RGB <--> YCrCb（YUV） JPEG	COLOR_RGB2YCrCb 、 COLOR_RGB2YCrCb 、 COLOR_YCrCb2BGR 、 COLOR_YCrCb2RGB 、 COLOR_RGB2YUV（可将 YCrCb 用 YUV 替代）
RGB <--> HSV	COLOR_BGR2HSV、COLOR_RGB2HSV、COLOR_HSV2BGR、COLOR_HSV2RGB
RGB <--> HLS	COLOR_BGR2HLS、COLOR_RGB2HLS、COLOR_HLS2BGR、COLOR_HLS2RGB
RGB <--> CIE L*a*b*	COLOR_BGR2Lab、COLOR_RGB2Lab、COLOR_Lab2BGR、COLOR_Lab2RGB
RGB <--> CIE L*u*v	COLOR_BGR2Luv、COLOR_RGB2Luv、COLOR_Luv2BGR、COLOR_Luv2RGB
RGB <--> Bayer	COLOR_BayerBG2BGR、COLOR _BayerGB2BGR、COLOR _BayerRG2BGR 、 COLOR _BayerGR2BGR 、 COLOR _BayerBG2RGB 、 COLOR _BayerGB2RGB 、 COLOR _BayerRG2RGB、COLOR _BayerGR2RGB（在 CCD 和 CMOS 上常用的 Bayer 模式）
YUV420 <--> RGB	COLOR_YUV420sp2BGR 、 COLOR_YUV420sp2RGB 、 COLOR_YUV420i2BGR、COLOR_YUV420i2RGB

即对于颜色空间转换，OpenCV2 的 CV_前缀的宏命名规范，被 OpenCV3 中 COLOR_式的宏命名前缀所取代。另外，在这里需要再次提醒大家的是，OpenCV 默认的图片通道存储顺序是 BGR，即蓝绿红，而不是 RGB。

本节最后，附上进行颜色空间转换的最简化版代码，大家可以选择上表中列举的宏替换掉 cvtColor 函数中名为 COLOR_BGR2Lab 的宏，进行 cvtColor 函数的测试。

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
using namespace cv;
```

```
void main( )
{
    //【1】载入图片
    Mat srcImage=imread("1.jpg",1),dstImage;
    //【2】转换颜色空间
    cvtColor(srcImage,dstImage, COLOR_BGR2Lab);
    //【3】显示效果图
    imshow("效果图",dstImage);
    //【4】保持窗口显示
    waitKey();
}
```

4.2.6 其他常用的知识点

本小节我们列举一下 OpenCV 的 Core 模块中其他常用的知识点，如下。

- Matx 是个轻量级的 Mat，必须在使用前规定好大小，比如一个 2*3 的 float 型的 Matx，可以声明为 Matx23f。
- Vec 是 Matx 的一个派生类，是一个一维的 Matx，跟 vector 很相似。在 OpenCV 源码中有如下定义。

```
template<typename_Tp, int n> class Vec : public Matx<_Tp, n, 1> {...};
typedef Vec<uchar, 2> Vec2b;
```

- Range 类其实就是为了使 OpenCV 的使用更像 MATLAB 而产生的。比如 Range::all()其实就是 MATLAB 里的符号。而 Range(a, b)其实就是 MATLAB 中的 a：b，注意这里的 a 和 b 都应为整型。
- OpenCV 中防止内存溢出的函数有 alignPtr、alignSize、allocate、deallocate、fastMalloc、fastFree 等。
- <math.h>里的一些函数使用起来很方便，有计算向量角度的函数 fastAtan2、计算立方根的函数 cubeRoot、向上取整函数 cvCeil、向下取整函数 cvFloor、四舍五入函数 cvRound 等。还有一些类似 MATLAB 里面的函数，比如 cvIsInf 判断自变量是否无穷大，cvIsNaN 判断自变量是否不是一个数。
- 显示文字相关的函数有 getTextSize、cvInitFont、putText。
- 作图相关的函数有 circle、clipLine、ellipse、ellipse2Poly、line、rectangle、polylines、类 LineIterator。
- 填充相关的函数有 fillConvexPoly、fillPoly。
- OpenCV 中 RNG()函数的作用为初始化随机数状态的生成器。

其他数据结构相关的知识由于使用较少在此不再过多说明，在需要用到的时候读者可以配合 OpenCV 文档，并查阅 OpenCV 的源码进行学习。

4.3 基本图形的绘制

本节我们将学习如何用 Point 在图像中定义 2D 点、如何使用 Scalar 表示颜色值。涉及到的绘制函数如下：

- 用于绘制直线的 line 函数；
- 用于绘制椭圆的 ellipse 函数；
- 用于绘制矩形的 rectangle 函数；
- 用于绘制圆的 circle 函数；
- 用于绘制填充的多边形的 fillPoly 函数。

让我们通过一个程序实例的学习来掌握 OpenCV 中各种绘制函数的用法。此程序的原型为 OpenCV 官方的示例程序，主要的脉络是定义了几个自定义的绘制函数，然后调用这些自定义的函数绘制出了两幅图——一幅化学原子示例图和一幅组合图。

在此，我们主要分析一下程序中的 4 个自定义函数的写法和用意。

需要注意，程序的源文件开头有如下的宏定义：

```
#define WINDOW_WIDTH 600//定义窗口大小的宏
```

4.3.1 DrawEllipse()函数的写法

如下为 DrawEllipse()函数的代码。

```
//-----------------------【DrawEllipse()函数】-----------------------
//          描述：自定义的绘制函数，实现了绘制不同角度、相同尺寸的椭圆
//-------------------------------------------------------------------
void DrawEllipse( Mat img, double angle )
{
  int thickness = 2;
  int lineType = 8;

  ellipse( img,
       Point( WINDOW_WIDTH/2, WINDOW_WIDTH/2 ),
       Size( WINDOW_WIDTH/4, WINDOW_WIDTH/16 ),
       angle,
       0,
       360,
       Scalar( 255, 129, 0 ),
       thickness,
       lineType );
}
```

此函数的写法解析如下。

函数 DrawEllipse 调用了 OpenCV 中的 ellipse 函数，将椭圆画到图像 img 上，椭圆中心为点(WINDOW_WIDTH/2.0, WINDOW_WIDTH/2.0)，并且大小位于矩形(WINDOW_WIDTH/4.0, WINDOW_WIDTH/16.0)内。椭圆旋转角度为 angle，扩展的弧度从 0 度到 360 度。图形颜色为 Scalar(255,129,0)代表的蓝色，线宽(thickness)为 2，线型(lineType)为 8（8 联通线型）。

线型的含义详见第 8 章 8.1.2 节的表 8.3。

4.3.2 DrawFilledCircle()函数的写法

```
//----------------------【DrawFilledCircle()函数】----------------------
//          描述：自定义的绘制函数，实现了实心圆的绘制
//-----------------------------------------------------------------------
void DrawFilledCircle( Mat img, Point center )
{
    int thickness = -1;
    int lineType = 8;

    circle( img,
        center,
        WINDOW_WIDTH/32,
        Scalar( 0, 0, 255 ),
        thickness,
        lineType );
}
```

此函数的写法解析如下。

函数 DrawFilledCircle()调用了 OpenCV 中的 circle 函数，将圆画到图像 img 上，圆心由点 center 定义，圆的半径为 WINDOW_WIDTH/32，圆的颜色为 Scalar(0,0,255)，按 BGR 的格式为红色，线粗定义为 thickness = -1，因此绘制的圆是实心的。

4.3.3 DrawPolygon()函数的写法

```
//-------------------------【DrawPolygon()函数】-------------------------
//          描述：自定义的绘制函数，实现了凹多边形的绘制
//-----------------------------------------------------------------------
void DrawPolygon( Mat img )
{
    int lineType = 8;

    //创建一些点
    Point rookPoints[1][20];
    rookPoints[0][0]  = Point(    WINDOW_WIDTH/4,   7*WINDOW_WIDTH/8 );
rookPoints[0][1]  = Point(  3*WINDOW_WIDTH/4,   7*WINDOW_WIDTH/8 );
    rookPoints[0][2]  = Point(  3*WINDOW_WIDTH/4,
13*WINDOW_WIDTH/16 );
    rookPoints[0][3]  = Point( 11*WINDOW_WIDTH/16,
13*WINDOW_WIDTH/16 );
    rookPoints[0][4]  = Point( 19*WINDOW_WIDTH/32,  3*WINDOW_WIDTH/8 );
    rookPoints[0][5]  = Point(  3*WINDOW_WIDTH/4,   3*WINDOW_WIDTH/8 );
    rookPoints[0][6]  = Point(  3*WINDOW_WIDTH/4,     WINDOW_WIDTH/8 );
    rookPoints[0][7]  = Point( 26*WINDOW_WIDTH/40,    WINDOW_WIDTH/8 );
    rookPoints[0][8]  = Point( 26*WINDOW_WIDTH/40,    WINDOW_WIDTH/4 );
    rookPoints[0][9]  = Point( 22*WINDOW_WIDTH/40,    WINDOW_WIDTH/4 );
    rookPoints[0][10] = Point( 22*WINDOW_WIDTH/40,    WINDOW_WIDTH/8 );
    rookPoints[0][11] = Point( 18*WINDOW_WIDTH/40,    WINDOW_WIDTH/8 );
    rookPoints[0][12] = Point( 18*WINDOW_WIDTH/40,    WINDOW_WIDTH/4 );
```

```
    rookPoints[0][13] = Point( 14*WINDOW_WIDTH/40,   WINDOW_WIDTH/4 );
    rookPoints[0][14] = Point( 14*WINDOW_WIDTH/40,   WINDOW_WIDTH/8 );
    rookPoints[0][15] = Point(   WINDOW_WIDTH/4,     WINDOW_WIDTH/8 );
    rookPoints[0][16] = Point(   WINDOW_WIDTH/4,   3*WINDOW_WIDTH/8 );
    rookPoints[0][17] = Point( 13*WINDOW_WIDTH/32,  3*WINDOW_WIDTH/8 );
    rookPoints[0][18] = Point(  5*WINDOW_WIDTH/16,
13*WINDOW_WIDTH/16 );
    rookPoints[0][19] = Point(   WINDOW_WIDTH/4,
13*WINDOW_WIDTH/16 );

    const Point* ppt[1] = { rookPoints[0] };
    int npt[] = { 20 };

    fillPoly( img,
        ppt,
        npt,
            1,
        Scalar( 255, 255, 255 ),
        lineType );
}
```

此函数的写法解析如下。

函数 DrawPolygon()调用了 OpenCV 中的 fillPoly 函数，用于将多边形画到图像 img 上，其中多边形的顶点集为 ppt，要绘制的多边形顶点数目为 npt，要绘制的多边形数量仅为 1，多边形的颜色定义为白色 Scalar(255,255,255)。

4.3.4 DrawLine()函数的写法

```
//---------------------------【DrawLine()函数】---------------------------
//          描述：自定义的绘制函数，实现了线的绘制
//------------------------------------------------------------------------
void DrawLine( Mat img, Point start, Point end )
{
  int thickness = 2;
  int lineType = 8;
  line( img,
    start,
    end,
    Scalar( 0, 0, 0 ),
    thickness,
    lineType );
}
```

此函数的写法解析如下。

DrawLin()函数调用了 OpenCV 中的 line 函数，用于在图像 img 上画一条从点 start 到点 end 的直线段，线的颜色为 Scalar(0,0,0)代表的黑色，线的粗细 thickness 为 2，且此线为 8 联通（lineType = 8）。

以上为 4 个自定义函数的写法和分析。 接下来，让我们一起看看 main 函数的写法。

4.3.5 main 函数的写法

main 函数的写法非常简单，先创建空白的 Mat 图像，然后调用函数绘制化学中的原子示例图，接着绘制组合图，最后显示绘制出的图像。

```
//----------------------【头文件、命名空间包含部分】----------------------

//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;

//----------------------------【宏定义部分】----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【绘制图 1】"        //为窗口标题定义的宏
#define WINDOW_NAME2 "【绘制图 2】"        //为窗口标题定义的宏
#define WINDOW_WIDTH 600//定义窗口大小的宏
//----------------------------【main()函数】----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main( void )
{

    // 创建空白的 Mat 图像
    Mat atomImage = Mat::zeros( WINDOW_WIDTH, WINDOW_WIDTH, CV_8UC3 );
    Mat rookImage = Mat::zeros( WINDOW_WIDTH, WINDOW_WIDTH, CV_8UC3 );

    // --------------------<1>绘制化学中的原子示例图--------------------

    //【1.1】先绘制出椭圆
    DrawEllipse( atomImage, 90 );
    DrawEllipse( atomImage, 0 );
    DrawEllipse( atomImage, 45 );
    DrawEllipse( atomImage, -45 );

    //【1.2】再绘制圆心
    DrawFilledCircle( atomImage, Point( WINDOW_WIDTH/2,
WINDOW_WIDTH/2) );

    // ------------------------<2>绘制组合图------------------------
    //【2.1】先绘制出椭圆
    DrawPolygon( rookImage );

    // 【2.2】绘制矩形
    rectangle( rookImage,
        Point( 0, 7*WINDOW_WIDTH/8 ),
        Point( WINDOW_WIDTH, WINDOW_WIDTH),
        Scalar( 0, 255, 255 ),
        -1,
```

```
        8 );

    // 【2.3】绘制一些线段
    DrawLine( rookImage, Point( 0, 15*WINDOW_WIDTH/16 ),
Point( WINDOW_WIDTH, 15*WINDOW_WIDTH/16 ) );
    DrawLine( rookImage, Point( WINDOW_WIDTH/4, 7*WINDOW_WIDTH/8 ),
Point( WINDOW_WIDTH/4, WINDOW_WIDTH ) );
    DrawLine( rookImage, Point( WINDOW_WIDTH/2, 7*WINDOW_WIDTH/8 ),
Point( WINDOW_WIDTH/2, WINDOW_WIDTH ) );
    DrawLine( rookImage, Point( 3*WINDOW_WIDTH/4, 7*WINDOW_WIDTH/8 ),
Point( 3*WINDOW_WIDTH/4, WINDOW_WIDTH ) );

    //-----------------------<3>显示绘制出的图像-----------------------
    imshow( WINDOW_NAME1, atomImage );
    moveWindow( WINDOW_NAME1, 0, 200 );
    imshow( WINDOW_NAME2, rookImage );
    moveWindow( WINDOW_NAME2, WINDOW_WIDTH, 200 );

    waitKey( 0 );
    return(0);
}
```

运行上述代码组合起来的程序，便可以绘制出如图4.17和图4.18所示的图片。

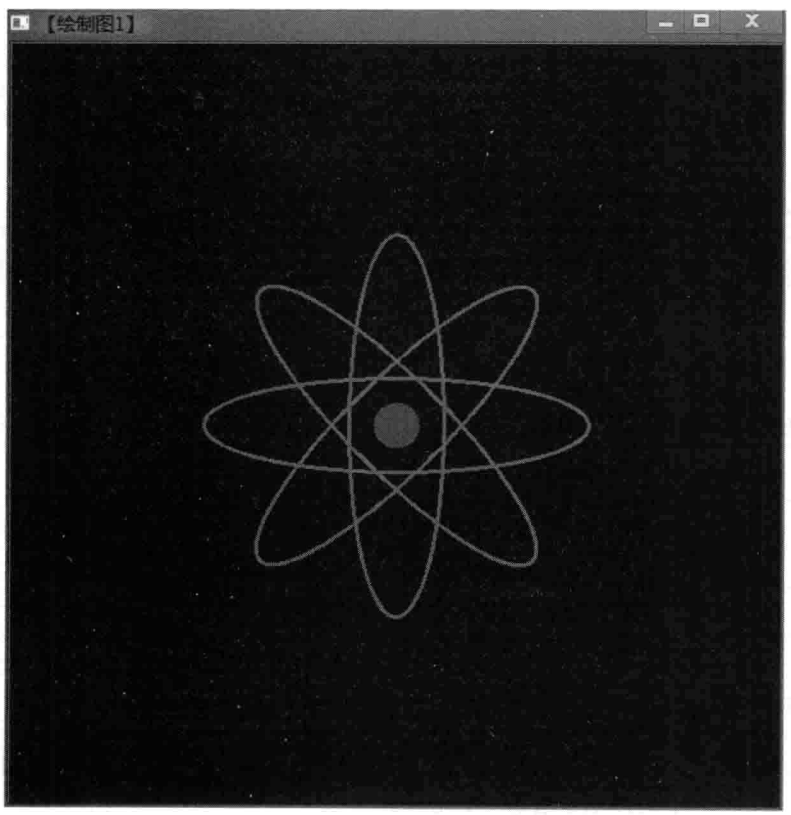


图 4.17 绘制出的化学原子示例图

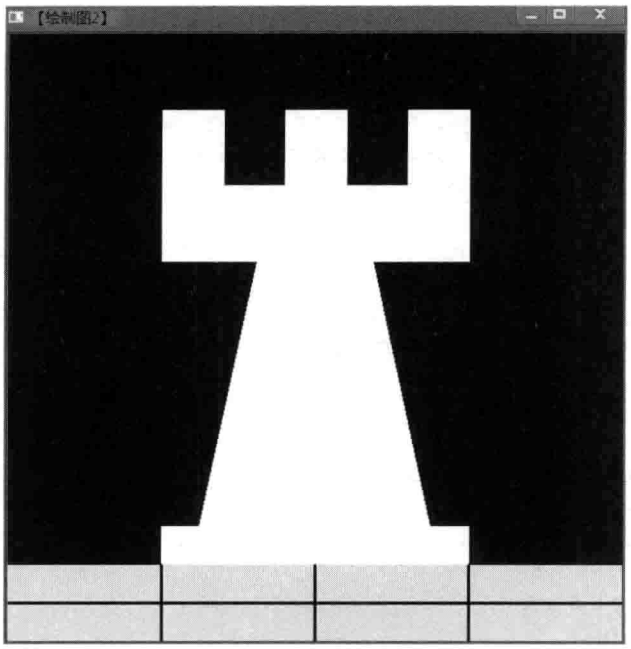


图 4.18 绘制出的组合图

4.4 本章小结

本章我们主要学习了经常会遇到的各种数据结构，主要是基础图像容器 Mat 的用法。

本章核心函数/类清单

函数名称	说明	对应讲解章节
Mat::Mat()	Mat 类的构造函数	4.1.4
Mat::Create()	Mat 类的成员函数，可用于 Mat 类的初始化操作	4.1.4
Point 类	用于表示点的数据结构	4.2.1
Scalar 类	用于表示颜色的数据结构	4.2.2
Size 类	用于表示尺寸的数据结构	4.2.3
Rect 类	用于表示矩形的数据结构	4.2.4
CvtColor()	用于颜色空间转换	4.2.5

本章示例程序清单

示例程序序号	程序说明	对应章节
19	基础图像容器 Mat 类的使用	4.1.7
20	基本图形的绘制	4.3

第 5 章
core 组件进阶

导读

本章中，你将学到：

- 如何操作图像中的像素
- 设置感兴趣区域（ROI）
- 如何进行图像混合
- 如何分离颜色通道
- 如何进行多通道图像混合
- 如何调整图像的对比度和亮度值
- 如何对图像进行离散傅里叶变换
- 如何输入输出 XML 和 YAML 文件

5.1 访问图像中的像素

5.1.1 图像在内存之中的存储方式

在之前的章节中，我们已经了解到图像矩阵的大小取决于所用的颜色模型，确切地说，取决于所用通道数。如果是灰度图像，矩阵就会如图 5.1 所示。

	Column 0	Column 1	Column ...	Column m
Row 0	0,0	0,1	...	0, m
Row 1	1,0	1,1	...	1, m
Row ...	...,0	...,1	...	..., m
Row n	n,0	n,1	n,...	n, m

图 5.1 灰度图像模型的矩阵排列

而对多通道图像来说，矩阵中的列会包含多个子列，其子列个数与通道数相等。例如，如图 5.2 所示 RGB 颜色模型的矩阵。

	Column 0			Column 1			Column ...			Column m		
Row 0	0,0	0,0	0,0	0,1	0,1	0,1	...	...	...	0, m	0, m	0, m
Row 1	1,0	1,0	1,0	1,1	1,1	1,1	...	...	...	1, m	1, m	1, m
Row ...	...,0	...,0	...,0	...,1	...,1	...,1	...	...	...	..., m	..., m	..., m
Row n	n,0	n,0	n,0	n,1	n,1	n,1	n,...	n,...	n,...	n, m	n, m	n, m

图 5.2 RGB 颜色模型的矩阵排列

可以看到，OpenCV 中子列的通道顺序是反过来的——BGR 而不是 RGB。很多情况下，因为内存足够大，可实现连续存储，因此，图像中的各行就能一行一行地连接起来，形成一个长行。连续存储有助于提升图像扫描速度，我们可以使用 isContinuous()来判断矩阵是否是连续存储的。相关示例会在接下来的内容中提供。

5.1.2 颜色空间缩减

我们知道，若矩阵元素存储的是单通道像素，使用 C 或 C++的无符号字符类型，那么像素可有 256 个不同值。但若是三通道图像，这种存储格式的颜色数就太多了（确切地说，有一千六百多万种）。用如此之多的颜色来进行处理，可能会对我们的算法性能造成严重影响。

其实，仅用这些颜色中具有代表性的很小的部分，就足以达到同样的效果。

如此，颜色空间缩减（color space reduction）便可以派上用场了，它在很多应用中可以大大降低运算复杂度。颜色空间缩减的做法是：将现有颜色空间值除以某个输入值，以获得较少的颜色数。也就是“做减法”，比如颜色值 0 到 9 可取为新值 0，10 到 19 可取为 10，以此类推。

如 uchar 类型的三通道图像，每个通道取值可以是 0～255，于是就有 256×256 个不同的值。我们可以定义：

0～9 范围的像素值为 0；

10～19 范围的像素值为 10；

20～29 范围的像素值为 20。

这样的操作将颜色取值降低为 26×26×26 种情况。这个操作可以用一个简单的公式来实现。因为 C++中 int 类型除法操作会自动截余。例如 Iold=14；Inew=(Iold/10)*10=(14/10)* 10=1*10=10;

uchar（无符号字符，即 0 到 255 之间取值的数）类型的值除以 int 值，结果仍是 char。因为结果是 char 类型的，所以求出来小数也要向下取整。利用这一点，刚才提到在 uchar 定义域中进行的颜色缩减运算就可以表达为下面的形式：

$$I_{new}=\left(\frac{I_{old}}{10}\right)\times 10$$

因为 C++中 int 类型除法操作会自动截余。比如：

```
Iold=14; Inew=(Iold/10)*10=(14/10)*10=1*10=10;
```

在处理图像像素时，每个像素需要进行一遍上述计算，也需要一定的时间花销。但我们注意到其实只有 0～255 种像素，即只有 256 种情况。进一步可以把 256 种计算好的结果提前存在表中 table 中，这样每种情况不需计算，直接从 table 中取结果即可。

```
int divideWith=10;
uchar table[256];
for (int i = 0; i < 256; ++i)
  table[i] = divideWith* (i/divideWith);
```

于是 table[i]存放的是值为 i 的像素减小颜色空间的结果，这样也就可以理解上述方法中的操作：

```
p[j] = table[p[j]];
```

这样，简单的颜色空间缩减算法就可由下面两步组成：

（1）遍历图像矩阵的每一个像素；

（2）对像素应用上述公式。

值得注意的是，我们这里用到了除法和乘法运算，而这两种运算又特别费时，所以，应尽可能用代价较低的加、减、赋值等运算来替换它们。此外，还应注意到，上述运算的输入仅能在某个有限范围内取值，如 uchar 类型可取 256 个值。

由此可知，对于较大的图像，有效的方法是预先计算所有可能的值，然后需要这些值的时候，利用查找表直接赋值即可。查找表是一维或多维数组，存储了不同输入值所对应的输出值，其优势在于只需读取、无须计算。

5.1.3 LUT 函数：Look up table 操作

对于上文提到的 Look up table 操作，OpenCV 官方文档中强烈推荐我们使用

一个原型为operationsOnArrays:LUT()<lut>的函数来进行。它用于批量进行图像元素查找、扫描与操作图像。其使用方法如下：

```
//首先我们建立一个mat型用于查表
Mat lookUpTable(1, 256, CV_8U);
uchar* p = lookUpTable.data;
for( int i = 0; i < 256; ++i)
    p[i] = table[i];

//然后我们调用函数 (I 是输入 J 是输出):
for (int i = 0; i < times; ++i)
    LUT(I, lookUpTable, J);
```

5.1.4 计时函数

另外有个问题是如何计时。可以利用这两个简便的计时函数——getTickCount()和getTickFrequency()。

- getTickCount()函数返回CPU自某个事件（如启动电脑）以来走过的时钟周期数
- getTickFrequency()函数返回CPU一秒钟所走的时钟周期数。这样，我们就能轻松地以秒为单位对某运算计时。

这两个函数组合起来使用的示例如下。

```
double time0 = static_cast<double>(getTickCount());//记录起始时间
// 进行图像处理操作……
time0 = ((double)getTickCount() - time0)/getTickFrequency();
cout<<"此方法运行时间为： "<<time0<<"秒"<<endl;  //输出运行时间
```

5.1.5 访问图像中像素的三类方法

任何图像处理算法，都是从操作每个像素开始的。即使我们不会使用OpenCV提供的各种图像处理函数，只要了解了图像处理算法的基本原理，也可以写出具有相同功能的程序。在OpenCV中，提供了三种访问每个像素的方法。

- 方法一　指针访问：C操作符[ ]；
- 方法二　迭代器iterator；
- 方法三　动态地址计算。

这三种方法在访问速度上略有差异。debug模式下，这种差异非常明显，不过在release模式下，这种差异就不太明显了。我们通过一组程序来说明这几种方法。程序的目的是减少图像中颜色的数量，比如原来的图像是是256种颜色，我们希望将它变成64种颜色，那只需要将原来的颜色除以4（整除）以后再乘以4就可以了。

将要使用的主程序代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
```

```
//-----------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
using namespace std;
using namespace cv;

//-------------------------【全局函数声明部分】-------------------------
//          描述：全局函数声明
//-----------------------------------------------------------------------
void colorReduce(Mat& inputImage, Mat& outputImage, int div);

//-------------------------【main()函数】-------------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main()
{
    //【1】创建原始图并显示
    Mat srcImage = imread("1.jpg");
    imshow("原始图像",srcImage);

    //【2】按原始图的参数规格来创建创建效果图
    Mat dstImage;
    dstImage.create(srcImage.rows,srcImage.cols,srcImage.type());//效
果图的大小、类型与原图片相同

    //【3】记录起始时间
    double time0 = static_cast<double>(getTickCount());

    //【4】调用颜色空间缩减函数
    colorReduce(srcImage,dstImage,32);

    //【5】计算运行时间并输出
    time0 = ((double)getTickCount() - time0)/getTickFrequency();
    cout<<"此方法运行时间为： "<<time0<<"秒"<<endl;  //输出运行时间

    //【6】显示效果图
    imshow("效果图",dstImage);
    waitKey(0);
}
```

主程序中调用 colorReduce 函数来完成减少颜色的工作，而我们根据访问每个像素三类方法，写出了三个版本的 colorReduce 函数。下面，我们将分别对其进行简单地讲解。

1.【方法一】用指针访问像素

用指针访问像素的这种方法利用的是 C 语言中的操作符[ ]。这种方法最快，但是略有点抽象。实验条件下单次运行时间为 0.00665378。范例代码如下。

```
//-------------------------【colorReduce()函数】------------------------
//          描述：使用【指针访问：C 操作符[ ]】方法版的颜色空间缩减函数
```

```
//-----------------------------------------------------------------------
void colorReduce(Mat& inputImage, Mat& outputImage, int div)
{
    //参数准备
    outputImage = inputImage.clone();  //复制实参到临时变量
    int rowNumber = outputImage.rows;  //行数
    int colNumber = outputImage.cols*outputImage.channels();  //列数×
通道数=每一行元素的个数

    //双重循环，遍历所有的像素值
    for(int i = 0;i < rowNumber;i++)  //行循环
    {
        uchar* data = outputImage.ptr<uchar>(i);  //获取第 i 行的首地址
        for(int j = 0;j < colNumber;j++)   //列循环
        {
         // ---------【开始处理每个像素】-------------
         data[j] = data[j]/div*div + div/2;
         // ----------【处理结束】---------------------
        }  //行处理结束
    }
}
```

下面对上述代码进行讲解。

Mat类有若干成员函数可以获取图像的属性。公有成员变量cols和rows给出了图像的宽和高，而成员函数channels()用于返回图像的通道数。灰度图的通道数为1，彩色图的通道数为3。

每行的像素值由以下语句得到：

```
int colNumber = outputImage.cols*outputImage.channels();  //列数×通道数=
每一行元素的个数
```

为了简化指针运算，Mat类提供了ptr函数可以得到图像任意行的首地址。ptr是一个模板函数，它返回第 *i* 行的首地址：

```
uchar* data = outputImage.ptr<uchar>(i);  //获取第 i 行的首地址
```

而双层循环内部的那句处理像素的代码，我们可以等效地使用指针运算从一列移动到下一列。所以，也可以这样来写：

```
*data++=*data/div*div+div/2;
```

2.【方法二】用迭代器操作像素

第二种方法为用迭代器操作像素，这种方法与STL库的用法类似。

在迭代法中，我们所需要做的仅仅是获得图像矩阵的begin和end，然后增加迭代直至从begin到end。将*操作符添加在迭代指针前，即可访问当前指向的内容。

相比用指针直接访问可能出现越界问题，迭代器绝对是非常安全的方法。

```
//-----------------------【colorReduce()函数】-----------------------
//          描述：使用【迭代器】方法版本的颜色空间缩减函数
//-----------------------------------------------------------------------
```

```
void colorReduce(Mat& inputImage, Mat& outputImage, int div)
{
    //参数准备
    outputImage = inputImage.clone();  //复制实参到临时变量
    //获取迭代器
    Mat_<Vec3b>::iterator it = outputImage.begin<Vec3b>();  //初始位置
的迭代器
    Mat_<Vec3b>::iterator itend = outputImage.end<Vec3b>();  //终止位置
的迭代器

    //存取彩色图像像素
    for(;it != itend;++it)
    {
        //----------------------【开始处理每个像素】----------------------
        (*it)[0] = (*it)[0]/div*div + div/2;
        (*it)[1] = (*it)[1]/div*div + div/2;
        (*it)[2] = (*it)[2]/div*div + div/2;
        //-----------------------【处理结束】-----------------------
    }
}
```

实验条件下单次运行时间为 0.242588。

不熟悉面向对象编程中迭代器的概念的读者，可以阅读与 STL 中迭代器相关的入门书籍和文字。用关键字“STL 迭代器”进行搜索可以找到各种相关的博文和资料。

3.【方法三】动态地址计算

第三种方法为用动态地址计算来操作像素。下面是使用动态地址运算配合 at 方法的 colorReduce 函数的代码。这种方法简洁明了，符合大家对像素的直观认识。实验条件下单次运行时间约为 0.334131。

```
//-----------------------【colorReduce()函数】-----------------------
//          描述：使用【动态地址运算配合 at】方法版本的颜色空间缩减函数
//-------------------------------------------------------------------
void colorReduce(Mat& inputImage, Mat& outputImage, int div)
{
    //参数准备
    outputImage = inputImage.clone();  //复制实参到临时变量
    int rowNumber = outputImage.rows;  //行数
    int colNumber = outputImage.cols;  //列数

    //存取彩色图像像素
    for(int i = 0;i < rowNumber;i++)
    {
        for(int j = 0;j < colNumber;j++)
        {
         //----------------------【开始处理每个像素】----------------------
         outputImage.at<Vec3b>(i,j)[0] =
outputImage.at<Vec3b>(i,j)[0]/div*div + div/2;  //蓝色通道
         outputImage.at<Vec3b>(i,j)[1] =
```

```
outputImage.at<Vec3b>(i,j)[1]/div*div + div/2;  //绿色通道
        outputImage.at<Vec3b>(i,j)[2] =
outputImage.at<Vec3b>(i,j)[2]/div*div + div/2;  //红色通道
        //------------------------【处理结束】------------------------
      }  // 行处理结束
   }
}
```

让我们讲解一下上述的代码。

Mat类中的cols和rows给出了图像的宽和高。而成员函数at（int y，int x）可以用来存取图像元素，但是必须在编译期知道图像的数据类型。需要注意的是，我们一定要确保指定的数据类型要和矩阵中的数据类型相符合，因为at方法本身不会对任何数据类型进行转换。

对于彩色图像，每个像素由三个部分构成：蓝色通道、绿色通道和红色通道（BGR）。因此，对于一个包含彩色图像的Mat，会返回一个由三个8位数组成的向量。OpenCV将此类型的向量定义为Vec3b，即由三个unsigned char组成的向量。这也解释了为什么存取彩色图像像素的代码可以写出如下形式：

```
image.at<Vec3b>(j,i)[channel]=value;
```

其中，索引值channel标明了颜色通道号。

另外需要再次提醒大家的是，OpenCV中的彩色图像不是以RGB的顺序存放的，而是BGR，所以程序中的outputImage.at<Vec3b>(i,j)[0]代表的是该点的B分量。同理还有(*it)[0]。

5.1.6 示例程序

本节至此结束。在这里说明一下本节相关的配套示例程序。本节共有四个配套的示例程序，按示例程序序号来排列为：

【21】用指针访问像素

【22】用迭代器访问像素

【23】用动态地址计算配合at访问像素

【24】遍历图像像素的14种方法

其中，第21~23分别为5.1.5节中讲到三种方法的示例程序，而第24个示例程序“遍历图像像素的14种方法”为来自一本国外OpenCV2书籍的配套示例程序，其用14个函数分别封装好了14种像素存取方法。比较典型，于是将其注释后提供给大家学习。

5.2 ROI区域图像叠加&图像混合

在本节中，我们将一起学习在OpenCV中如何定义感兴趣区域ROI，如何使用addWeighted函数进行图像混合操作，以及如何将ROI和addWeighted函数结

合起来使用，对指定区域进行图像混合操作。

5.2.1 感兴趣区域：ROI

在图像处理领域，我们常常需要设置感兴趣区域（ROI，region of interest），来专注或者简化工作过程。也就是从图像中选择的一个图像区域，这个区域是图像分析所关注的重点。我们圈定这个区域，以便进行进一步处理。而且，使用 ROI 指定想读入的目标，可以减少处理时间，增加精度，给图像处理来带不小的便利。

定义 ROI 区域有两种方法：第一种是使用表示矩形区域的 Rect。它指定矩形的左上角坐标（构造函数的前两个参数）和矩形的长宽（构造函数的后两个参数）以定义一个矩形区域。

其中，image 为已经载入好的图片。

```
//定义一个 Mat 类型并给其设定 ROI 区域
Mat imageROI;
//方法一
imageROI=image(Rect(500,250,logo.cols,logo.rows));
```

另一种定义 ROI 的方式是指定感兴趣行或列的范围（Range）。Range 是指从起始索引到终止索引（不包括终止索引）的一连段连续序列。cRange 可以用来定义 Range。如果使用 Range 来定义 ROI，那么前例中定义 ROI 的代码可以重写为：

```
//方法二
imageROI=image(Range(250,250+logoImage.rows),Range(200,200+logoImage
.cols));
```

下面我们来看一个实例，显示如何利用 ROI 将一幅图加到另一幅图的指定位置。大家如果需要复制以下函数中的代码直接运行，可以自己建一个基于 console 的程序，然后把函数体中的内容复制到 main 函数中，然后找两幅大小合适的图片，加入到工程目录下，并和代码中读取的文件名一致即可。

在下面的代码中，我们通过一个图像掩膜（mask），直接将插入处的像素设置为 logo 图像的像素值，这样效果会很逼真。

```
//----------------------【ROI_AddImage()函数】----------------------
// 函数名：ROI_AddImage()
//     描述：利用感兴趣区域 ROI 实现图像叠加
//-----------------------------------------------------------------
bool ROI_AddImage()
{

    //【1】读入图像
    Mat srcImage1= imread("dota_pa.jpg");
    Mat logoImage= imread("dota_logo.jpg");
    if(!srcImage1.data ) { printf("读取 srcImage1 错误~！ \n"); return
false; }
    if(!logoImage.data ) { printf("读取 logoImage 错误~！ \n"); return
false; }
```

```
    //【2】定义一个 Mat 类型并给其设定 ROI 区域
    Mat imageROI=
srcImage1(Rect(200,250,logoImage.cols,logoImage.rows));

    //【3】加载掩模（必须是灰度图）
    Mat mask= imread("dota_logo.jpg",0);

    //【4】将掩膜复制到 ROI
    logoImage.copyTo(imageROI,mask);

    //【5】显示结果
    namedWindow("<1>利用 ROI 实现图像叠加示例窗口");
    imshow("<1>利用 ROI 实现图像叠加示例窗口",srcImage1);

    return true;
}
```

这个函数首先是载入了两张 jpg 图片到 srcImage1 和 logoImage 中，然后定义了一个 Mat 类型的 imageROI，并使用 Rect 设置其感兴趣区域为 srcImage1 中的一块区域，将 imageROI 和 srcImage1 关联起来。接着定义了一个 Mat 类型的的 mask 并读入 dota_logo.jpg，顺势使用 Mat:: copyTo 把 mask 中的内容复制到 imageROI 中，于是就得到了最终的效果图。namedWindow 和 imshow 配合使用，显示出最终的结果。

运行结果如图 5.3 所示。

图 5.3 运行结果

图 5.3 中白色的 dota2 logo，就是通过操作之后加上去的图像。

5.2.2 线性混合操作

线性混合操作是一种典型的二元（两个输入）的像素操作，它的理论公式如下：

$$g(x)=(1-a)f_a(x)+af_3(x)$$

我们通过在范围 0 到 1 之间改变 alpha 值，来对两幅图像（f0(x)和 f1(x)）或

两段视频（同样为（f0(x)和 f1(x)）产生时间上的画面叠化（cross-dissolve）效果，就像幻灯片放映和电影制作中的那样，也就是在幻灯片翻页时设置的前后页缓慢过渡叠加效果，以及电影情节过渡时经常出现的画面叠加效果。

实现方面，主要运用了 OpenCV 中 addWeighted 函数，下面来一起全面地了解它。

5.2.3 计算数组加权和：addWeighted()函数

这个函数的作用是计算两个数组（图像阵列）的加权和。原型如下：

```
void (InputArray src1, double alpha, InputArray src2, double beta, double
gamma, OutputArray dst, int dtype=-1);
```

- 第一个参数，InputArray 类型的 src1，表示需要加权的第一个数组，常常填一个 Mat；
- 第二个参数，double 类型的 alpha，表示第一个数组的权重；
- 第三个参数，InputArray 类型的 src2，表示第二个数组，它需要和第一个数组拥有相同的尺寸和通道数；
- 第四个参数，double 类型的 beta，表示第二个数组的权重值；
- 第五个参数，double 类型的 gamma，一个加到权重总和上的标量值。其含义通过接下来列出的式子自然会理解；
- 第六个参数，OutputArray 类型的 dst，输出的数组，它和输入的两个数组拥有相同的尺寸和通道数；
- 第七个参数，int 类型的 dtype，输出阵列的可选深度，有默认值-1。当两个输入数组具有相同的深度时，这个参数设置为-1（默认值），即等同于 src1.depth()。

下面的数学公式表示：用 addWeighted 函数计算以下两个数组（src1 和 src2）的加权和，得到结果输出给第四个参数，也就是 addWeighted 函数的作用的矩阵表达式。

```
dst = src1[I]*alpha+ src2[I]*beta + gamma;
```

其中 I 是多维数组元素的索引值。而且，在遇到多通道数组的时候，每个通道都需要独立地进行处理。另外需要注意的是，当输出数组的深度为 CV_32S 时，这个函数就不适用了，这时候就会内存溢出或者算出的结果压根不对。

理论和函数的讲解就是上面这些，接着我们来看代码实例，以融会贯通。

```
//-----------------------【LinearBlending()函数】----------------------
// 函数名：LinearBlending()
// 描述：利用 cv::addWeighted()函数实现图像线性混合
//-----------------------------------------------------------------------
bool LinearBlending()
{
      //【0】定义一些局部变量
      double alphaValue = 0.5;
      double betaValue;
```

```
    Mat srcImage2, srcImage3, dstImage;

    //【1】读取图像 ( 两幅图片需为同样的类型和尺寸 )
    srcImage2= imread("mogu.jpg");
    srcImage3= imread("rain.jpg");

    if(!srcImage2.data ) { printf("读取 srcImage2 错误~! \n"); return
false; }
    if(!srcImage3.data ) { printf("读取 srcImage3 错误~! \n"); return
false; }

    //【2】做图像混合加权操作
    betaValue= ( 1.0 - alphaValue );
    addWeighted(srcImage2, alphaValue, srcImage3, betaValue, 0.0,
dstImage);

    //【3】创建并显示原图窗口
    namedWindow("<2>线性混合示例窗口【原图】", 1);
    imshow("<2>线性混合示例窗口【原图】", srcImage2 );

    namedWindow("<3>线性混合示例窗口【效果图】", 1);
    imshow("<3>线性混合示例窗口【效果图】", dstImage );

    return true;

}
```

下面对以上代码进行解析。

(0) 首先当然是定义一些局部变量、alpha值、beta值，以及三个Mat类型的变量

```
//【0】定义一些局部变量
    double alphaValue = 0.5;
    double betaValue;
    Mat srcImage2, srcImage3, dstImage;
```

在这里我们设置alpha值为0.5。

(1) 读取两幅图像并作错误处理

这步很简单，直接看代码。

```
//读取图像 ( 两幅图片需为同样的类型和尺寸 )
    srcImage2= imread("mogu.jpg");
    srcImage3= imread("rain.jpg");
if(!srcImage2.data ) { printf("读取 srcImage2 错误~! \n"); return false; }
    if(!srcImage3.data ) { printf("读取 srcImage3 错误~! \n"); return
false; }
```

在这里需要注意的是，因为我们是对 srcImage1 和 srcImage2 求和，所以它们必须要有相同的尺寸（宽度和高度）和类型，不然多余的部分没有对应的“伴”，肯定会出问题。

（2）进行图像混合加权操作

载入图像后，我们就可以来生成混合图像，也就是之前公式中的 g(x)。为此目的，使用函数 addWeighted 可以很方便地实现。代码其实很简单，就是这样：

```
//【2】进行图像混合加权操作
  betaValue = ( 1.0 - alphaValue );
  addWeighted( srcImage2, alphaValue, srcImage3,betaValue, 0.0,
dstImage);
```

其中对应于 addWeighted 的参数中的 beta 值为 1-alpha，gamma 值为 0。

（3）创建显示窗口，显示图像

```
// 【3】创建并显示原图窗口
      namedWindow("<2>线性混合示例窗口【原图】", 1);
      imshow("<2>线性混合示例窗口【原图】", srcImage2 );

      namedWindow("<3>线性混合示例窗口【效果图】", 1);
      imshow("<3>线性混合示例窗口【效果图】", dstImage );
```

接着来看一下运行效果图，首先是原图（图 5.4）。

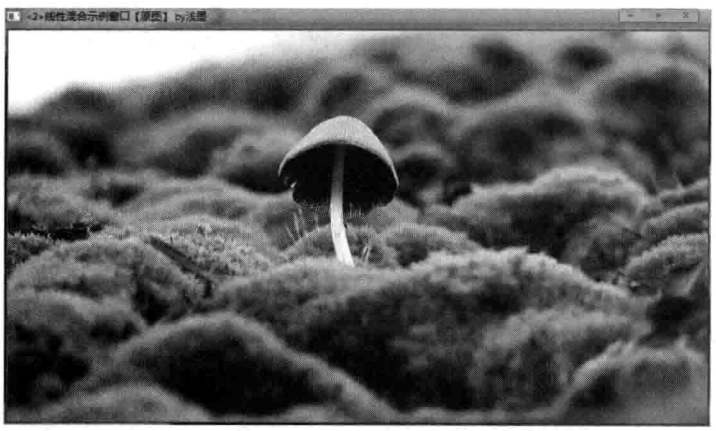


图 5.4 原始图

然后是效果图，如图 5.5 所示。

图 5.5 线性混合效果图

5.2.4 综合示例：初级图像混合

在前文介绍的设定感兴趣区域ROI和使用addWeighted函数进行图像线性混合的基础上，我们还可以将二者结合起来使用，也就是先指定ROI，再用addWeighted函数对指定的ROI区域的图像进行混合操作。我们将其封装在了一个名为ROI_LinearBlending的函数中，方便大家分块学习。代码如下。

```
//-----------------------【ROI_LinearBlending()】----------------------
// 函数名：ROI_LinearBlending()
// 描述：线性混合实现函数，指定区域线性图像混合.利用cv::addWeighted()函数结合定义
//                    感兴趣区域ROI，实现自定义区域的线性混合
//-----------------------------------------------------------------------
bool ROI_LinearBlending()
{

    //【1】读取图像
    Mat srcImage4= imread("dota_pa.jpg",1);
    Mat logoImage= imread("dota_logo.jpg");

    if(!srcImage4.data ) { printf("读取srcImage4错误~！ \n"); return false; }
    if(!logoImage.data ) { printf("读取logoImage错误~！ \n"); return false; }

    //【2】定义一个Mat类型并给其设定ROI区域
    Mat imageROI;
        //方法一

imageROI=srcImage4(Rect(200,250,logoImage.cols,logoImage.rows));
    //方法二

//imageROI=srcImage4(Range(250,250+logoImage.rows),Range(200,200+logoImage.cols));

    //【3】将logo加到原图上
    addWeighted(imageROI,0.5,logoImage,0.3,0.,imageROI);

    //【4】显示结果
    namedWindow("<4>区域线性图像混合示例窗口");
    imshow("<4>区域线性图像混合示例窗口",srcImage4);

    return true;
}
```

下面放出详细注释的本节示例源代码，示例代码都将封装在了不同的函数中，具体如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-----------------------------------------------------------------------
#include <opencv2/opencv.hpp>
```

```
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;


//-------------------------【全局函数声明部分】-------------------------
//     描述：全局函数声明
//-----------------------------------------------------------------------
bool ROI_AddImage();
bool LinearBlending();
bool ROI_LinearBlending();

//-------------------------【main()函数】-------------------------
//     描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
      system("color 5E");

      if(ROI_AddImage()&& LinearBlending()&&ROI_LinearBlending())
      {
            cout<<endl<<"运行成功，得出了你需要的图像~！ : )";
      }

      waitKey(0);
      return 0;
}

//-------------------------【ROI_AddImage()函数】-------------------------
// 函数名：ROI_AddImage()
//     描述：利用感兴趣区域ROI实现图像叠加
//-----------------------------------------------------------------------
bool ROI_AddImage()
{

      //【1】读入图像
      Mat srcImage1= imread("dota_pa.jpg");
      Mat logoImage= imread("dota_logo.jpg");
      if(!srcImage1.data ) { printf("读取srcImage1错误~！ \n"); return
false; }
      if(!logoImage.data ) { printf("读取logoImage错误~！ \n"); return
false; }

      //【2】定义一个Mat类型并给其设定ROI区域
      Mat imageROI=
srcImage1(Rect(200,250,logoImage.cols,logoImage.rows));

      //【3】加载掩模（必须是灰度图）
      Mat mask= imread("dota_logo.jpg",0);

      //【4】将掩膜复制到ROI
```

```
    logoImage.copyTo(imageROI,mask);

    //【5】显示结果
    namedWindow("<1>利用ROI实现图像叠加示例窗口");
    imshow("<1>利用ROI实现图像叠加示例窗口",srcImage1);

    return true;
}


//----------------------【LinearBlending()函数】----------------------
// 函数名: LinearBlending()
// 描述: 利用cv::addWeighted()函数实现图像线性混合
//-------------------------------------------------------------------
bool LinearBlending()
{
    //【0】定义一些局部变量
    double alphaValue = 0.5;
    double betaValue;
    Mat srcImage2, srcImage3, dstImage;

    //【1】读取图像 ( 两幅图片需为同样的类型和尺寸 )
    srcImage2= imread("mogu.jpg");
    srcImage3= imread("rain.jpg");

    if(!srcImage2.data ) { printf("读取srcImage2错误~! \n"); return
false; }
    if(!srcImage3.data ) { printf("读取srcImage3错误~! \n"); return
false; }

    //【2】进行图像混合加权操作
    betaValue= ( 1.0 - alphaValue );
    addWeighted(srcImage2, alphaValue, srcImage3, betaValue, 0.0,
dstImage);

    //【3】创建并显示原图窗口
    namedWindow("<2>线性混合示例窗口【原图】", 1);
    imshow("<2>线性混合示例窗口【原图】", srcImage2 );

    namedWindow("<3>线性混合示例窗口【效果图】", 1);
    imshow("<3>线性混合示例窗口【效果图】", dstImage );

    return true;

}

//----------------------【ROI_LinearBlending()】----------------------
// 函数名: ROI_LinearBlending()
// 描述:线性混合实现函数,指定区域线性图像混合.利用cv::addWeighted()函数结合定
义
//                    感兴趣区域ROI, 实现自定义区域的线性混合
//-------------------------------------------------------------------
```

```
bool ROI_LinearBlending()
{

    //【1】读取图像
    Mat srcImage4= imread("dota_pa.jpg",1);
    Mat logoImage= imread("dota_logo.jpg");

    if(!srcImage4.data ) { printf("读取 srcImage4 错误~！ \n"); return
false; }
    if(!logoImage.data ) { printf("读取 logoImage 错误~！ \n"); return
false; }

    //【2】定义一个 Mat 类型并给其设定 ROI 区域
    Mat imageROI;
        //方法一

imageROI=srcImage4(Rect(200,250,logoImage.cols,logoImage.rows));
    //方法二

//imageROI=srcImage4(Range(250,250+logoImage.rows),Range(200,200+log
oImage.cols));

    //【3】将 logo 加到原图上
    addWeighted(imageROI,0.5,logoImage,0.3,0.,imageROI);

    //【4】显示结果
    namedWindow("<4>区域线性图像混合示例窗口");
    imshow("<4>区域线性图像混合示例窗口",srcImage4);

    return true;
}
```

最后看一下整体的运行效果，如图 5.6、图 5.7、图 5.8、图 5.9 所示。

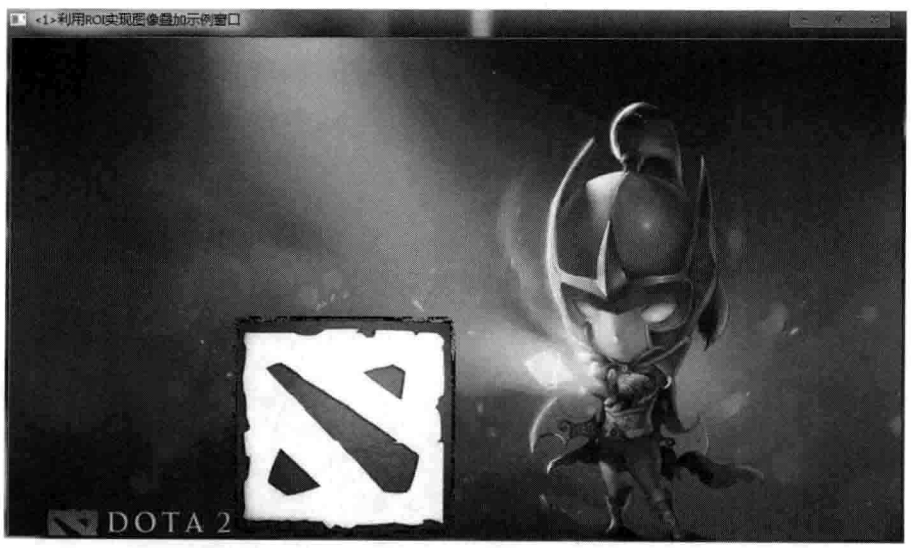


图 5.6 图像叠加效果图

图 5.7 区域线性混合效果图

图 5.8 线性混合原始图

图 5.9 线性混合效果图

5.3 分离颜色通道、多通道图像混合

上节中我们讲解了如何使用 addWeighted 函数进行图像混合操作，以及如何将 ROI 和 addWeighted 函数结合起来，对指定区域进行图像混合操作。

而为了更好地观察一些图像材料的特征，有时需要对 RGB 三个颜色通道的分量进行分别显示和调整。通过 OpenCV 的 split 和 merge 方法可以很方便地达到目的。

这一节，我们会详细介绍这两个互为“冤家”的函数。首先来看看进行通道分离的 split 函数。

5.3.1 通道分离：split()函数

split 函数用于将一个多通道数组分离成几个单通道数组。这里的 array 按语境翻译为数组或者阵列。

这个 split 函数的 C++版本有两个原型，分别是：

- C++: void split(const Mat& src, Mat*mvbegin);
- C++: void split(InputArray m,OutputArrayOfArrays mv);

变量介绍如下：

- 第一个参数，InputArray 类型的 m 或者 const Mat&类型的 src，填我们需要进行分离的多通道数组。
- 第二个参数，OutputArrayOfArrays 类型的 mv，填函数的输出数组或者输出的 vector 容器。

split 函数分割多通道数组转换成独立的单通道数组，公式如下：

$$mv[c](I) = src(I)_c$$

最后我们一起看一个示例。

```
Mat srcImage;
Mat imageROI;
vector<Mat> channels;
srcImage= cv::imread("dota.jpg");

// 把一个 3 通道图像转换成 3 个单通道图像
split(srcImage,channels);//分离色彩通道
      imageROI=channels.at(0);

addWeighted(imageROI(Rect(385,250,logoImage.cols,logoImage.rows)),1.
0,

logoImage,0.5,0.,imageROI(Rect(385,250,logoImage.cols,logoImage.rows
)));

      merge(channels,srcImage4);
```

```
namedWindow("sample");
imshow("sample",srcImage);
```

将一个多通道数组分离成几个单通道数组的 split()函数的内容大概就是以上这些了，下面我们来看一下和它关系密切的 merge()函数。

5.3.2 通道合并：merge()函数

merge()函数是 split()函数的逆向操作——将多个数组合并成一个多通道的数组。它通过组合一些给定的单通道数组，将这些孤立的单通道数组合并成一个多通道的数组，从而创建出一个由多个单通道阵列组成的多通道阵列。它有两个基于 C++的函数原型如下。

- C++: void merge(const Mat* mv, size_tcount, OutputArray dst)
- C++: void merge(InputArrayOfArrays mv,OutputArray dst)

变量介绍如下。

- 第一个参数，mv。填需要被合并的输入矩阵或 vector 容器的阵列，这个 mv 参数中所有的矩阵必须有着一样的尺寸和深度。
- 第二个参数，count。当 mv 为一个空白的 C 数组时，代表输入矩阵的个数，这个参数显然必须大于 1。
- 第三个参数，dst。即输出矩阵，和 mv[0]拥有一样的尺寸和深度，并且通道的数量是矩阵阵列中的通道的总数。

函数解析如下。

merge 函数的功能是将一些数组合并成一个多通道的数组。关于组合的细节，输出矩阵中的每个元素都将是输出数组的串接。其中，第 i 个输入数组的元素被视为 mv[i]。C 一般用其中的 Mat::at()方法对某个通道进行存取，也就是这样用：channels.at(0)。

这里的 Mat::at()方法返回一个引用到指定的数组元素。注意是引用，相当于两者等价，也就是修改其中一个，另一个也会随之改变。

依然是一个示例，如下。

```
vector<Mat> channels;
Mat imageBlueChannel;
Mat imageGreenChannel;
Mat imageRedChannel;
srcImage4= imread("dota.jpg");
// 把一个 3 通道图像转换成 3 个单通道图像
split(srcImage4,channels);//分离色彩通道
imageBlueChannel = channels.at(0);
imageGreenChannel = channels.at(1);
imageRedChannel = channels.at(2);
```

上面的代码先做了相关的类型声明，然后把载入的 3 通道图像转换成 3 个单

通道图像，放到 vector<Mat>类型的 channels 中，接着进行引用赋值。

根据 OpenCV 的 BGR 色彩空间(Bule、Green、Red，蓝绿红)，其中 channels.at(0)就表示引用取出 channels 中的蓝色分量，channels.at(1)就表示引用取出 channels 中的绿色分量，channels.at(2)就表示引用取出 channels 中的红色分量。

一对做相反操作的 split()函数和 merge()函数的用法就是这些。另外提一点，如果我们需要从多通道数组中提取出特定的单通道数组，或者说实现一些复杂的通道组合，可以使用 mixChannels()函数。

5.3.3 示例程序：多通道图像混合

在本小节展示的示例程序中，我们把多通道图像混合的实现代码封装在了名为 MultiChannelBlending()的函数中。详细注释的代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//  描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <iostream>
using namespace cv;
using namespace std;


//-------------------------【全局函数声明部分】-------------------------
//  描述：全局函数声明
//-------------------------------------------------------------------------
bool  ROI_AddImage();
bool  LinearBlending();
bool  ROI_LinearBlending();

//-------------------------【main()函数】-------------------------
//  描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    system("color 5E");

    if(ROI_AddImage()&& LinearBlending()&&ROI_LinearBlending())
    {
        cout<<endl<<"运行成功,得出了你需要的图像~！ : )";
    }

    waitKey(0);
    return 0;
}

//----------------------【ROI_AddImage()函数】----------------------
//  函数名：ROI_AddImage()
//  描述：利用感兴趣区域 ROI 实现图像叠加
//-------------------------------------------------------------------------
```

```
bool  ROI_AddImage()
{

    // 【1】读入图像
    Mat srcImage1= imread("dota_pa.jpg");
    Mat logoImage= imread("dota_logo.jpg");
    if( !srcImage1.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定图片存在~！ \n"); return false; }
    if( !logoImage.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定图片存在~！ \n"); return false; }

    // 【2】定义一个 Mat 类型并给其设定 ROI 区域
    Mat imageROI=
srcImage1(Rect(200,250,logoImage.cols,logoImage.rows));

    // 【3】加载掩膜（必须是灰度图）
    Mat mask= imread("dota_logo.jpg",0);

    //【4】将掩膜复制到 ROI
    logoImage.copyTo(imageROI,mask);

    // 【5】显示结果
    namedWindow("<1>利用 ROI 实现图像叠加示例窗口");
    imshow("<1>利用 ROI 实现图像叠加示例窗口",srcImage1);

    return true;
}


//-----------------------【LinearBlending()函数】----------------------
// 函数名：LinearBlending()
// 描述：利用 cv::addWeighted()函数实现图像线性混合
//-----------------------------------------------------------------------
bool  LinearBlending()
{
    //【0】定义一些局部变量
    double alphaValue = 0.5;
    double betaValue;
    Mat srcImage2, srcImage3, dstImage;

    // 【1】读取图像 （两幅图片需为同样的类型和尺寸）
    srcImage2 = imread("mogu.jpg");
    srcImage3 = imread("rain.jpg");

    if( !srcImage2.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定图片存在~！ \n"); return false; }
    if( !srcImage3.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定图片存在~！ \n"); return false; }

    // 【2】进行图像混合加权操作
    betaValue = ( 1.0 - alphaValue );
    addWeighted( srcImage2, alphaValue, srcImage3, betaValue, 0.0,
```

```
dstImage);

    // 【3】创建并显示原图窗口
    namedWindow("<2>线性混合示例窗口【原图】", 1);
    imshow( "<2>线性混合示例窗口【原图】", srcImage2 );

    namedWindow("<3>线性混合示例窗口【效果图】", 1);
    imshow( "<3>线性混合示例窗口【效果图】", dstImage );

    return true;

}

//----------------------【ROI_LinearBlending()】----------------------
// 函数名: ROI_LinearBlending()
// 描述:线性混合实现函数,指定区域线性图像混合.利用 cv::addWeighted()函数结合定
义
//         感兴趣区域 ROI, 实现自定义区域的线性混合
//-------------------------------------------------------------------
bool  ROI_LinearBlending()
{

    //【1】读取图像
    Mat srcImage4= imread("dota_pa.jpg",1);
    Mat logoImage= imread("dota_logo.jpg");

    if( !srcImage4.data ) { printf("读取图片错误, 请确定目录下是否有 imread
函数指定图片存在~! \n"); return false; }
    if( !logoImage.data ) { printf("读取图片错误, 请确定目录下是否有 imread
函数指定图片存在~! \n"); return false; }

    //【2】定义一个 Mat 类型并给其设定 ROI 区域
    Mat imageROI;
    //方法一
    imageROI= srcImage4(Rect(200,250,logoImage.cols,logoImage.rows));
    //方法二
    //imageROI=
srcImage4(Range(250,250+logoImage.rows),Range(200,200+logoImage.cols
));

    //【3】将 logo 加到原图上
    addWeighted(imageROI,0.5,logoImage,0.3,0.,imageROI);

    //【4】显示结果
    namedWindow("<4>区域线性图像混合示例窗口");
    imshow("<4>区域线性图像混合示例窗口",srcImage4);

    return true;
}
```

可以发现，其实多通道混合的实现函数中的代码大体可分成三部分，分别对蓝绿红三个通道进行处理，唯一不同的地方在于取通道分量时取的是 channels.at(0)、

channels.at(1)还是 channels.at(2)。

运行截图如图 5.10、图 5.11、图 5.12 所示，具体区别表现在箭头指向处。

图 5.10 游戏原画+logo 蓝色通道

图 5.11 游戏原画+logo 绿色通道

图 5.12 游戏原画+logo 红色通道

5.4 图像对比度、亮度值调整

本节我们将学习如何用 OpenCV 进行图像对比度和亮度值的动态调整。

5.4.1 理论依据

首先了解一下算子的概念。一般的图像处理算子都是一个函数，它接受一个或多个输入图像，并产生输出图像。下面是算子的一般形式。

$$g(x)=h(f(x)) \text{ 或者 } g(x)=h(f_0(x)\cdots\cdots f_n(x))$$

本节所讲解的图像亮度和对比度的调整操作，其实属于图像处理变换中比较简单的一种——点操作（pointoperators）。点操作有一个特点：仅仅根据输入像素值（有时可加上某些全局信息或参数），来计算相应的输出像素值。这类算子包括亮度（brightness）和对比度（contrast）调整、颜色校正（colorcorrection）和变换（transformations）。

两种最常用的点操作（或者说点算子）是乘上一个常数（对应对比度的调节）以及加上一个常数（对应亮度值的调节）。公式如下：

$$g(x)=a*f(x)+b$$

看到这个式子，我们关于图像亮度和对比度调整的策略就比较好理解了。

其中：

- 参数 $f(x)$表示源图像像素。
- 参数 $g(x)$表示输出图像像素。
- 参数 a（需要满足 $a>0$）被称为增益（gain），常常被用来控制图像的对比度。
- 参数 b 通常被称为偏置（bias），常常被用来控制图像的亮度。

而更近一步，我们这样改写这个式子：

$$g(i,j)=a*f(i,j)+b$$

其中，i 和 j 表示像素位于第 i 行和第 j 列，这个式子可以用来作为我们在 OpenCV 中控制图像的亮度和对比度的理论公式。

5.4.2 访问图片中的像素

访问图片中的像素有很多种方式，在本书 5.1 节“访问图像中的像素”中已有过比较系统的讲解。

而为了执行如下运算：

$$g(i,j)=a*f(i,j)+b$$

我们需要访问图像的每一个像素。因为是对 GBR 图像进行运算，每个像素有三个值（G、B、R），所以我们必须分别访问它们（OpenCV 中的图像存储模式为

GBR)。以下是访问像素的代码片段，使用了三个for循环。

```
//三个 for 循环，执行运算 new_image(i,j) =a*image(i,j) + b
for(int y = 0; y < image.rows; y++ )
{
      for(int x = 0; x < image.cols; x++ )
      {
            for(int c = 0; c < 3; c++ )
            {
                  new_image.at<Vec3b>(y,x)[c]=
saturate_cast<uchar>( (g_nContrastValue*0.01)*(image.at<Vec3b>(y,x)[
c] ) + g_nBrightValue );
            }
      }
}
```

让我们分三个方面进行讲解。

- 为了访问图像的每一个像素，使用这样的语法：image.at<Vec3b>(y,x)[c]。

其中，*y*是像素所在的行，*x*是像素所在的列，*c*是R、G、B（对应0、1、2）其中之一。

- 因为运算结果可能会超出像素取值范围（溢出），还可能是非整数（如果是浮点数的话），所以要用saturate_cast对结果进行转换，以确保它为有效值。
- 这里的*a*也就是对比度，一般为了观察的效果，它的取值为0.0到3.0的浮点值，但是轨迹条一般取值都会取整数，因此在这里我们可以将其代表对比度值的nContrastValue参数设为0到300之间的整型，在最后的式子中乘以一个0.01，这样就完成了轨迹条中300个不同取值的变化。这就是为什么在式子中，会有saturate_cast<uchar>((g_nContrastValue*0.01)*(image.at<Vec3b>(y,x)[c])+g_nBrightValue)中的g_nContrastValue*0.01。

5.4.3 示例程序：图像对比度、亮度值调整

本小节依然是一个详细注释的配套示例程序，把本节前文介绍的知识点以代码为载体，展现给大家。

此示例程序用两个轨迹条分别控制对比度和亮度值，详细注释的示例程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
using namespace std;
using namespace cv;


//-------------------------【全局函数声明部分】-------------------------
```

```
//      描述：全局函数声明
//-----------------------------------------------------------------------
static void on_ContrastAndBright(int, void *);
static void ShowHelpText();

//-------------------------【全局变量声明部分】--------------------------
//      描述：全局变量声明
//-----------------------------------------------------------------------
int g_nContrastValue; //对比度值
int g_nBrightValue;  //亮度值
Mat g_srcImage,g_dstImage;
//-------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
    //【1】读取输入图像
    g_srcImage = imread( "1.jpg");
        if( !g_srcImage.data ) { printf("读取图片错误，请确定目录下是否有
imread函数指定图片存在~！ "); return false; }
    g_dstImage = Mat::zeros( g_srcImage.size(), g_srcImage.type() );

    //【2】设定对比度和亮度的初值
    g_nContrastValue=80;
    g_nBrightValue=80;

    //【3】创建效果图窗口
    namedWindow("【效果图窗口】", 1);

    //【4】创建轨迹条
    createTrackbar("对比度：", "【效果图窗口】",&g_nContrastValue,
300,on_ContrastAndBright );
    createTrackbar("亮　度：", "【效果图窗口】",&g_nBrightValue,
200,on_ContrastAndBright );

    //【5】进行回调函数初始化
    on_ContrastAndBright(g_nContrastValue,0);
    on_ContrastAndBright(g_nBrightValue,0);

    //【6】按下"q"键时，程序退出
    while(char(waitKey(1)) != 'q') {}
    return 0;
}


//-------------------【on_ContrastAndBright()函数】-------------------
// 描述：改变图像对比度和亮度值的回调函数
//-----------------------------------------------------------------------
static void on_ContrastAndBright(int, void *)
{
    // 创建窗口
    namedWindow("【原始图窗口】", 1);
```

```
    // 三个 for 循环，执行运算 g_dstImage(i,j) = a*g_srcImage(i,j) + b
    for( int y = 0; y < g_srcImage.rows; y++ )
    {
        for( int x = 0; x < g_srcImage.cols; x++ )
        {
          for( int c = 0; c < 3; c++ )
          {
              g_dstImage.at<Vec3b>(y,x)[c] =
saturate_cast<uchar>( (g_nContrastValue*0.01)*( g_srcImage.at<Vec3b>
(y,x)[c] ) + g_nBrightValue );
          }
        }
    }

    // 显示图像
    imshow("【原始图窗口】", g_srcImage);
    imshow("【效果图窗口】", g_dstImage);
}
```

讲解一下上述代码中的 saturate_cast 模板函数，其用于溢出保护，大致原理如下。

```
if(data<0)
      data=0;
else if(data>255)
   data=255;
```

最后看一下程序的运行截图。运行这个程序会得到两个图片显示窗口：第一个为原图窗口（图 5.13），第二个为效果图窗口（图 5.14）。在效果图窗口中可以调节两个轨迹条，来改变当前图片的对比度和亮度。

图 5.13 原始图

图 5.14 可调节对比度和亮度的效果图

5.5 离散傅里叶变换

离散傅里叶变换（Discrete Fourier Transform，缩写为 DFT），是指傅里叶变换在时域和频域上都呈现离散的形式，将时域信号的采样变换为在离散时间傅里叶变换（DTFT）频域的采样。在形式上，变换两端（时域和频域上）的序列是有限长的，而实际上这两组序列都应当被认为是离散周期信号的主值序列。即使对有限长的离散信号做 DFT，也应当对其经过周期延拓成为周期信号再进行变换。在实际应用中，通常采用快速傅里叶变换来高效计算 DFT。

5.5.1 离散傅里叶变换的原理

简单来说，对一张图像使用傅里叶变换就是将它分解成正弦和余弦两部分，也就是将图像从空间域（spatial domain）转换到频域（frequency domain）。

这一转换的理论基础为：任一函数都可以表示成无数个正弦和余弦函数的和的形式。傅里叶变换就是一个用来将函数分解的工具。

二维图像的傅里叶变换可以用以下数学公式表达。

$$F(k,l)=\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}f(i,j)\mathrm{e}^{-\mathrm{i}2\pi\left(\frac{ki}{N}+\frac{lj}{N}\right)}$$

$$\mathrm{e}^{\mathrm{i}x}=\cos x+\mathrm{i}\sin x$$

式中 f 是空间域（spatial domain）值，F 是频域（frequency domain）值。转换之后的频域值是复数，因此，显示傅里叶变换之后的结果需要使用实数图像（real image）加虚数图像（complex image），或者幅度图像（magitude image）加相位图像（phase image）的形式。在实际的图像处理过程中，仅仅使用了幅度图像，因为幅度图像包含了原图像的几乎所有我们需要的几何信息。然而，如果想通过修改幅度图像或者相位图像的方法来间接修改原空间图像，需要使用逆傅里叶变换得到修改后的空间图像，这样就必须同时保留幅度图像和相位图像了。

在此示例中，我们将展示如何计算以及显示傅里叶变换后的幅度图像。由于数字图像的离散性，像素值的取值范围也是有限的。比如在一张灰度图像中，像素灰度值一般在 0 到 255 之间。 因此，我们这里讨论的也仅仅是离散傅里叶变换（DFT）。如果需要得到图像中的几何结构信息，那你就要用到它了。

在频域里面，对于一幅图像，高频部分代表了图像的细节、纹理信息；低频部分代表了图像的轮廓信息。如果对一幅精细的图像使用低通滤波器，那么滤波后的结果就只剩下轮廓了。这与信号处理的基本思想是相通的。如果图像受到的噪声恰好位于某个特定的“频率”范围内，则可以通过滤波器来恢复原来的图像。傅里叶变换在图像处理中可以做到图像增强与图像去噪、图像分割之边缘检测、图像特征提取、图像压缩等。

5.5.2 dft()函数详解

dft 函数的作用是对一维或二维浮点数数组进行正向或反向离散傅里叶变换。

```
C++: void dft(InputArray src, OutputArray dst, int flags=0, int
nonzeroRows=0)
```

- 第一个参数，InputArray 类型的 src。输入矩阵，可以为实数或者虚数。
- 第二个参数，OutputArray 类型的 dst。函数调用后的运算结果存在这里，其尺寸和类型取决于标识符，也就是第三个参数 flags。
- 第三个参数，int 类型的 flags。转换的标识符，有默认值 0，取值可以为表 6.1 中标识符的结合。

表 6.1 dft 标识符取值列表

标识符名称	意义
DFT_INVERSE	用一维或二维逆变换代替默认的正向变换
DFT_SCALE	缩放比例标识符，输出的结果都会以 1/*N* 进行缩放，通常会结合 DFT_INVERSE 一起使用
DFT_ROWS	对输入矩阵的每行进行正向或反向的变换，此标识符可以在处理多种矢量的时候用于减小资源开销，这些处理常常是三维或高维变换等复杂操作
DFT_COMPLEX_OUTPUT	进行一维或二维实数数组正变换。这样的结果虽然是复数阵列，但拥有复数的共轭对称性（CCS），所以可以被写成一个拥有同样尺寸的实数阵列
DFT_REAL_OUTPUT	进行一维或二维复数数组反变换。这样的结果通常是一个大小相同的复矩阵。如果输入的矩阵有复数的共轭对称性（比如是一个带有 DFT_COMPLEX_OUTPUT 标识符的正变换结果），便会输出实矩阵

- 第四个参数，int 类型的 nonzeroRows，有默认值 0。当此参数设为非零时（最好是取值为想要处理的那一行的值，比如 C.rows），函数会假设只有输入矩阵的第一个非零行包含非零元素（没有设置 DFT_INVERSE 标识符），或只有输出矩阵的第一个非零行包含非零元素（设置了 DFT_INVERSE 标识符）。这样的话，函数就可对其他行进行更高效的处理，以节省时间开销。这项技术尤其是在采用 DFT 计算矩阵卷积时非常有效。

讲解完函数的参数含义，下面我们看一个用 dft 函数计算两个二维实矩阵卷积的示例核心片段。

```
void convolveDFT(InputArray A, InputArray B, OutputArray C)
{
    //【1】初始化输出矩阵
    C.create(abs(A.rows - B.rows)+1, abs(A.cols - B.cols)+1, A.type());
Size dftSize;

//【2】计算 DFT 变换的尺寸
dftSize.width = getOptimalDFTSize(A.cols + B.cols - 1);
```

```
    dftSize.height = getOptimalDFTSize(A.rows + B.rows - 1);

    //【3】分配临时缓冲区并初始化置零
    Mat tempA(dftSize, A.type(), Scalar::all(0));
    Mat tempB(dftSize, B.type(), Scalar::all(0));

//【4】分别复制A和B到tempA和tempB的左上角
Mat roiA(tempA, Rect(0,0,A.cols,A.rows));
    A.copyTo(roiA);
    Mat roiB(tempB, Rect(0,0,B.cols,B.rows));
    B.copyTo(roiB);

//【5】就地操作(in-place),进行快速傅里叶变换,并将nonzeroRows参数置为非零,
以进行更加快速地处理
    dft(tempA, tempA, 0, A.rows);
    dft(tempB, tempB, 0, B.rows);

//【6】将得到的频谱相乘,结果存放于tempA中
    mulSpectrums(tempA, tempB, tempA);

//【7】将结果变换为频域,且尽管结果行(result rows)都为非零,我们只需要其中的
C.rows的第一行,所以采用nonzeroRows == C.rows
    dft(tempA, tempA, DFT_INVERSE + DFT_SCALE, C.rows);

    //【8】将结果复制到C中
    tempA(Rect(0, 0, C.cols, C.rows)).copyTo(C);

    //所有的临时缓冲区将被自动释放,所以无须收尾操作
}
```

此示例中的注释已经十分详尽。其中出现了新的函数 MulSpectrums，它的作用是计算两个傅里叶频谱的每个元素的乘法，前两个参数为输入的参加乘法运算的两个矩阵，第三个参数为得到的乘法结果矩阵。

由于 5.5.8 节会放出一个使用了不少新函数的示例程序，下面我们先将这些“新面孔”各个击破。

5.5.3 返回 DFT 最优尺寸大小：getOptimalDFTSize()函数

getOptimalDFTSize 函数返回给定向量尺寸的傅里叶最优尺寸大小。为了提高离散傅里叶变换的运行速度，需要扩充图像，而具体扩充多少，就由这个函数来计算得到。

```
C++: int getOptimalDFTSize(int vecsize)
```

此函数的唯一一个参数为 int 类型的 vecsize，向量尺寸，即图片的 rows、cols。

5.5.4 扩充图像边界：copyMakeBorder()函数

copyMakeBorder 函数的作用是扩充图像边界。

```
C++: void copyMakeBorder(InputArray src, OutputArray dst, int top, int
```

```
bottom, int left, int right, int borderType, const Scalar&
value=Scalar() )
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，即这个参数用于存放函数调用后的输出结果，需和源图片有一样的尺寸和类型，且 size 应该为 Size（src.cols+left+right, src.rows+top+bottom）。
- 接下来的 4 个参数分别为 int 类型的 top、bottom、left、right，分别表示在源图像的四个方向上扩充多少像素，例如 top=2，bottom=2，left=2，right=2 就意味着在源图像的上下左右各扩充两个像素宽度的边界。
- 第七个参数，borderType 类型的，边界类型，常见取值为 BORDER_CONSTANT，可参考 borderInterpolate()得到更多的细节。
- 第八个参数，const Scalar&类型的 value，有默认值 Scalar()，可以理解为默认值为 0。当 borderType 取值为 BORDER_CONSTANT 时，这个参数表示边界值。

5.5.5 计算二维矢量的幅值：magnitude()函数

magnitude()函数用于计算二维矢量的幅值。

```
C++: void magnitude(InputArray x, InputArray y, OutputArray magnitude)
```

- 第一个参数，InputArray 类型的 x，表示矢量的浮点型 X 坐标值，也就是实部。
- 第一个参数，InputArray 类型的 y，表示矢量的浮点型 Y 坐标值，也就是虚部。
- 第三次参数，OutputArray 类型的 magnitude，输出的幅值，它和第一个参数 x 有着同样的尺寸和类型。

下式可以表示 magnitude()函数的原理：

$$dst(I)=\sqrt{x(I)^2+y(I)^2}$$

5.5.6 计算自然对数：log()函数

log()函数的功能是计算每个数组元素绝对值的自然对数。

```
C++: void log(InputArray src, OutputArray dst)
```

第一个参数为输入图像，第二个参数为得到的对数值。其原理如下。

$$\mathrm{dst(I)}=\begin{cases}\log|\mathrm{src(I)}| & \text{if src(I)}\neq 0\\ \mathrm{C} & \text{oterwise}\end{cases}$$

5.5.7 矩阵归一化：normalize()函数

normalize()的作用是进行矩阵归一化。

```
C++: void normalize(InputArray src, OutputArray dst, double alpha=1,
double beta=0, int norm_type=NORM_L2, int dtype=-1, InputArray
mask=noArray() )
```

- 第一个参数，InputArray 类型的 src。输入图像，即源图像，填 Mat 类的对象即可。
- 第二个参数，OutputArray 类型的 dst。函数调用后的运算结果存在这里，和源图片有一样的尺寸和类型。
- 第三个参数，double 类型的 alpha。归一化后的最大值，有默认值 1。
- 第四个参数，double 类型的 beta。归一化后的最大值，有默认值 0。
- 第五个参数，int 类型的 norm_type。归一化类型，有 NORM_INF、NORM_L1、NORM_L2 和 NORM_MINMAX 等参数可选，有默认值 NORM_L2。
- 第六个参数，int 类型的 dtype，有默认值-1。当此参数取负值时，输出矩阵和 src 有同样的类型，否则，它和 src 有同样的通道数，且此时图像深度为 CV_MAT_DEPTH（dtype）。
- 第七个参数，InputArray 类型的 mask，可选的操作掩膜，有默认值 noArray()。

5.5.8 示例程序：离散傅里叶变换

这节中我们将学习一个以 dft()函数为核心，对图像求傅里叶变换的，有详细注释的示例程序。

在此示例中，将展示如何计算以及显示傅里叶变换后的幅度图像。由于数字图像的离散性，像素值的取值范围也是有限的。比如在一张灰度图像中，像素灰度值一般在 0 到 255 之间。因此，我们这里讨论的也仅仅是离散傅里叶变换（DFT）。如果需要得到图像中的几何结构信息，那么就要用到离散傅里叶变换了。下面的步骤将以输入图像为单通道的灰度图像 I 为例，进行分步说明。

1.【第一步】载入原始图像

我们在这一步以灰度模式读取原始图像，进行是否读取成功的检测，并显示出读取到的图像。代码如下。

```
    //【1】以灰度模式读取原始图像并显示
    Mat srcImage = imread("1.jpg", 0);
    if(!srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread 函
数指定图片存在~！ \n"); return false; }
    imshow("原始图像" , srcImage);
```

2.【第二步】将图像扩大到合适的尺寸

离散傅里叶变换的运行速度与图片的尺寸有很大关系。当图像的尺寸是 2、3、5 的整数倍时，计算速度最快。因此，为了达到快速计算的目的，经常通过添凑新的边缘像素的方法获取最佳图像尺寸。函数 getOptimalDFTSize()用于返回最佳尺寸，而函数 copyMakeBorder()用于填充边缘像素，这一步代码如下。

```
//【2】将输入图像延扩到最佳的尺寸，边界用 0 补充
int m = getOptimalDFTSize( I.rows );
```

```
int n = getOptimalDFTSize( I.cols );
//将添加的像素初始化为0.
Mat padded;
copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT,
Scalar::all(0));
```

3.【第三步】为傅里叶变换的结果（实部和虚部）分配存储空间

傅里叶变换的结果是复数，这就是说对于每个原图像值，结果会有两个图像值。此外，频域值范围远远超过空间值范围， 因此至少要将频域储存在 float 格式中。所以我们将输入图像转换成浮点类型，并多加一个额外通道来储存复数部分。

```
    //【3】为傅里叶变换的结果(实部和虚部)分配存储空间
    //将planes数组组合合并成一个多通道的数组complexI
    Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(),
CV_32F)};
    Mat complexI;
    merge(planes, 2, complexI);
```

4.【第四步】进行离散傅里叶变换

这里的离散傅里叶变换为图像就地计算模式（in-place，输入输出为同一图像）。

```
//【4】进行就地离散傅里叶变换
    dft(complexI, complexI);
```

5.【第五步】将复数转换为幅值

复数包含实数部分（Re）和虚数部分（imaginary – Im）。离散傅里叶变换的结果是复数，对应的幅度可以表示为：

$$M = \sqrt{\mathrm{Re}(DET(I))^2 + \mathrm{Im}(DFT(I))^2}$$

那么，转化为 OpenCV 代码，就是下面这样的。

```
    //【5】将复数转换为幅值，即=> log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
    split(complexI, planes); // 将多通道数组complexI分离成几个单通道数组,
planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
    magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
    Mat magnitudeImage = planes[0];
```

6.【第六步】进行对数尺度（logarithmic scale）缩放.

傅里叶变换的幅度值范围大到不适合在屏幕上显示。高值在屏幕上显示为白点，而低值为黑点，高低值的变化无法有效分辨。为了在屏幕上凸显出高低变化的连续性，我们可以用对数尺度来替换线性尺度，公式如下。

$$M_1=\log(1+M)$$

而写成 OpenCV 代码，就是下面这样的。

```
    //【6】进行对数尺度(logarithmic scale)缩放
    magnitudeImage += Scalar::all(1);
```

```
log(magnitudeImage, magnitudeImage);//求自然对数
```

7.【第七步】剪切和重分布幅度图象限

因为在第二步中延扩了图像，那现在是时候将新添加的像素剔除了。为了方便显示，也可以重新分布幅度图象限位置（注：将第五步得到的幅度图从中间划开，得到 4 张 1/4 子图像，将每张子图像看成幅度图的一个象限，重新分布，即将 4 个角点重叠到图片中心）。这样的话原点(0,0)就位移到图像中心了。OpenCV 代码如下。

```
//【7】剪切和重分布幅度图象限
//若有奇数行或奇数列，进行频谱裁剪
magnitudeImage = magnitudeImage(Rect(0, 0, magnitudeImage.cols & -2,
magnitudeImage.rows & -2));
//重新排列傅里叶图像中的象限，使得原点位于图像中心
int cx = magnitudeImage.cols/2;
int cy = magnitudeImage.rows/2;
Mat q0(magnitudeImage, Rect(0, 0, cx, cy));   // ROI 区域的左上
Mat q1(magnitudeImage, Rect(cx, 0, cx, cy));  // ROI 区域的右上
Mat q2(magnitudeImage, Rect(0, cy, cx, cy));  // ROI 区域的左下
Mat q3(magnitudeImage, Rect(cx, cy, cx, cy)); // ROI 区域的右下
//交换象限（左上与右下进行交换）
Mat tmp;
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);
//交换象限（右上与左下进行交换）
q1.copyTo(tmp);
q2.copyTo(q1);
tmp.copyTo(q2);
```

8.【第八步】归一化

这一步仍然是为了显示。现在有了重分布后的幅度图，但是幅度值仍然超过可显示范围[0,1]。我们使用 normalize()函数将幅度归一化到可显示范围。

```
//【8】归一化，用 0 到 1 之间的浮点值将矩阵变换为可视的图像格式
//此句代码的 OpenCV2 版为：
//normalize(magnitudeImage, magnitudeImage, 0, 1, CV_MINMAX);
//此句代码的 OpenCV3 版为：
normalize(magnitudeImage, magnitudeImage, 0, 1, NORM_MINMAX);
```

9.【第九步】显示效果图

处理完成，最后进行显示即可。

```
//【9】显示效果图
imshow("频谱幅值", magnitudeImage);
```

将上述代码组合到一起，便得到了程序的完整源代码。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
```

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
using namespace cv;


//-----------------------------【main()函数】--------------------------
//        描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//---------------------------------------------------------------------
int main( )
{

    //【1】以灰度模式读取原始图像并显示
    Mat srcImage = imread("1.jpg", 0);
    if(!srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread 函
数指定图片存在~！ \n"); return false; }
    imshow("原始图像" , srcImage);

    ShowHelpText();

    //【2】将输入图像延扩到最佳的尺寸，边界用 0 补充
    int m = getOptimalDFTSize( srcImage.rows );
    int n = getOptimalDFTSize( srcImage.cols );
    //将添加的像素初始化为 0.
    Mat padded;
    copyMakeBorder(srcImage, padded, 0, m - srcImage.rows, 0, n -
srcImage.cols, BORDER_CONSTANT, Scalar::all(0));

    //【3】为傅里叶变换的结果(实部和虚部)分配存储空间。
    //将 planes 数组组合合并成一个多通道的数组 complexI
    Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(),
CV_32F)};
    Mat complexI;
    merge(planes, 2, complexI);

    //【4】进行就地离散傅里叶变换
    dft(complexI, complexI);

    //【5】将复数转换为幅值，即=> log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
    split(complexI, planes); // 将多通道数组 complexI 分离成几个单通道数组,
planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
    magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
    Mat magnitudeImage = planes[0];

    //【6】进行对数尺度(logarithmic scale)缩放
    magnitudeImage += Scalar::all(1);
    log(magnitudeImage, magnitudeImage);//求自然对数

    //【7】剪切和重分布幅度图象限
    //若有奇数行或奇数列，进行频谱裁剪
    magnitudeImage = magnitudeImage(Rect(0, 0, magnitudeImage.cols & -2,
```

```
magnitudeImage.rows & -2));
    //重新排列傅里叶图像中的象限，使得原点位于图像中心
    int cx = magnitudeImage.cols/2;
    int cy = magnitudeImage.rows/2;
    Mat q0(magnitudeImage, Rect(0, 0, cx, cy));   // ROI 区域的左上
    Mat q1(magnitudeImage, Rect(cx, 0, cx, cy));  // ROI 区域的右上
    Mat q2(magnitudeImage, Rect(0, cy, cx, cy));  // ROI 区域的左下
    Mat q3(magnitudeImage, Rect(cx, cy, cx, cy)); // ROI 区域的右下
    //交换象限（左上与右下进行交换）
    Mat tmp;
    q0.copyTo(tmp);
    q3.copyTo(q0);
    tmp.copyTo(q3);
    //交换象限（右上与左下进行交换）
    q1.copyTo(tmp);
    q2.copyTo(q1);
    tmp.copyTo(q2);

    //【8】归一化，用 0 到 1 之间的浮点值将矩阵变换为可视的图像格式
    //此句代码的 OpenCV2 版为：
    //normalize(magnitudeImage, magnitudeImage, 0, 1, CV_MINMAX);
    //此句代码的 OpenCV3 版为：
    normalize(magnitudeImage, magnitudeImage, 0, 1, NORM_MINMAX);

    //【9】显示效果图
    imshow("频谱幅值", magnitudeImage);
    waitKey();

    return 0;
}
```

程序注释已经足够详尽，其中新出现的函数都在上文中有过介绍，所以在这里就不花篇幅进一步展开讲解了。让我们一起看一下运行截图（图 5.15、图 5.16），以结束本节的学习。

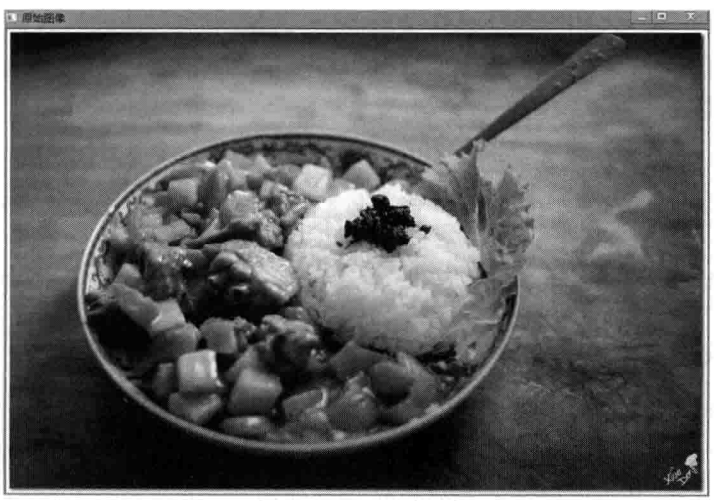


图 5.15 原始图

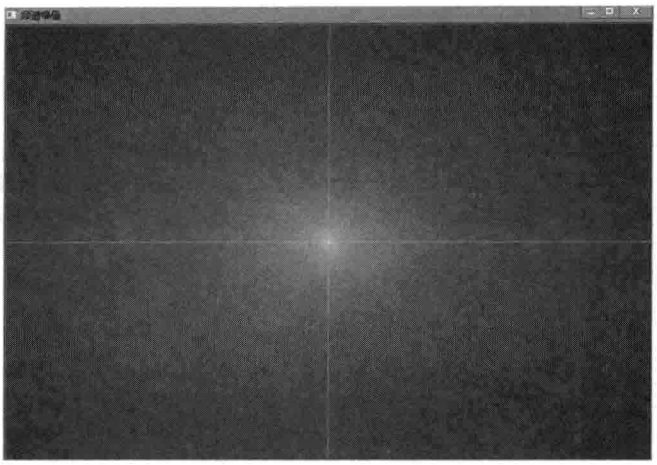


图 5.16 离散傅里叶变换效果图

5.6 输入输出 XML 和 YAML 文件

5.6.1 XML 和 YAML 文件简介

本节我们将一起认识 XML 和 YAML 这两种文件类型。

所谓 XML，即 eXtensible Markup Language，翻译成中文为“可扩展标识语言”。首先，XML 是一种元标记语言。所谓“元标记”，就是开发者可以根据自身需要定义自己的标记，比如可以定义标记<book>、<name>。任何满足 XML 命名规则的名称都可以标记，这就向不同的应用程序打开了的大门。此外，XML 是一种语义/结构化语言，它描述了文档的结构和语义。

YAML 是“YAML Ain't a Markup Language”（译为“YAML 不是一种置标语言”）的递回缩写。在开发的这种语言时，YAML 的原意是：“Yet Another Markup Language”（仍是一种置标语言），但为了强调这种语言以数据为中心，而不是以置标语言为重点，而用返璞词进行重新命名。YAML 是一个可读性高，用来表达资料序列的格式。它参考了其他多种语言，包括：XML、C 语言、Python、Perl，以及电子邮件格式 RFC2822。

.yml 和.yaml 同为 YAML 格式的后缀名

总之，YAML 试图用一种比 XML 更敏捷的方式，来完成 XML 所完成的任务。

5.6.2 FileStorage 类操作文件的使用引导

XML 和 YAML 是使用非常广泛的文件格式，可以利用 XML 或者 YAML 格式的文件存储和还原各式各样的数据结构。当然，它们还可以存储和载入任意复杂的数据结构，其中就包括了 OpenCV 相关周边的数据结构，以及各种原始数据类型，如整数和浮点数字和文本字符串。

我们一般使用如下过程来写入或者读取数据到 XML 或 YAML 文件中。

（1）实例化一个 FileStorage 类的对象，用默认带参数的构造函数完成初始化，或者用 FileStorage::open()成员函数辅助初始化。

（2）使用流操作符<<进行文件写入操作，或者>>进行文件读取操作，类似 C++中的文件输入输出流。

（3）使用 FileStorage::release()函数析构掉 FileStorage 类对象，同时关闭文件。

下面分别对这三个步骤进行实例讲解。

1. 【第一步】XML、YAML 文件的打开

（1）准备文件写操作

FileStorage 是 OpenCV 中 XML 和 YAML 文件的存储类，封装了所有相关的信息。它是 OpenCV 从文件中读数据或向文件中写数据时必须要使用的一个类。

此类的构造函数为 FileStorage::FileStorage，有两个重载，如下。

```
C++: FileStorage::FileStorage()
C++: FileStorage::FileStorage(const string& source, int flags, const
string& encoding=string())
```

构造函数在实际使用中，方法一般有两种。

1）对于第二种带参数的构造函数，进行写操作范例如下。

```
FileStorage fs("abc.xml", FileStorage::WRITE);
```

2）对于第一种不带参数的构造函数，可以使用其成员函数 FileStorage::open 进行数据的写操作，范例如下。

```
FileStorage fs;
fs.open("abc.xml", FileStorage:: WRITE);
```

（2）准备文件读操作

上面讲到的都是以 FileStorage:: WRITE 为标识符的写操作，而读操作，采用 FileStorage::READ 标识符即可，相关示例代码如下。

1）第一种方式

```
FileStorage fs("abc.xml", FileStorage:: READ);
```

2）第二种方式

```
FileStorage fs;
fs.open("abc.xml", FileStorage:: READ);
```

另外需要注意的是，上面的这些操作示例是对 XML 文件为例子作演示的，而对 YAML 文件，操作方法是类似的，就是将 XML 文件换为 YAML 文件即可。

2. 【第二步】进行文件读写操作

（1）文本和数字的输入和输出

定义好 FileStorage 类对象之后，写入文件可以使用“<<”运算符，例如：

```
fs << "iterationNr" << 100;
```

而读取文件，使用“>>”运算符，例如：

```
int itNr;
fs["iterationNr"] >> itNr;
itNr = (int) fs["iterationNr"];
```

（2）OpenCV 数据结构的输入和输出

关于 OpenCV 数据结构的输入和输出，和基本的 C++形式相同，范例如下。

```
//数据结构的初始化
Mat R = Mat_<uchar >::eye (3, 3),
Mat T = Mat_<double>::zeros(3, 1);
// 向 Mat 中写入数据
fs << "R" << R;
fs << "T" << T;
// 从 Mat 中读取数据
fs["R"] >> R;
fs["T"] >> T;
```

3.【第三步】vector（arrays）和 maps 的输入和输出

对于 vector 结构的输入和输出，要注意在第一个元素前加上“[”，在最后一个元素前加上“]”。例如：

```
fs << "strings" << "["; / /开始读入 string 文本序列
fs << "image1.jpg" << "Awesomeness" << "baboon.jpg";
fs << "]"; //关闭序列
```

而对于 map 结构的操作，使用的符号是“{”和“}”，例如：

```
fs << "Mapping"; //开始读入 Mapping 文本
fs << "{" << "One" << 1;
fs << "Two" << 2 << "}";
```

读取这些结构的时候，会用到 FileNode 和 FileNodeIterator 数据结构。对 FileStorage 类的“[”、“]”操作符会返回 FileNode 数据类型；对于一连串的 node，可以使用 FileNodeIterator 结构，例如：

```
FileNode n = fs["strings"]; //读取字符串序列以得到节点
if (n.type() != FileNode::SEQ)
{
cerr << "发生错误！字符串不是一个序列！ " << endl;
return 1;
}

FileNodeIterator it = n.begin(), it_end = n.end(); // 遍历节点
for (; it != it_end; ++it)
cout << (string)*it << endl;
```

4.【第四步】文件关闭

需要注意的是，文件关闭操作会在 FileStorage 类销毁时自动进行，但我们也

可显式调用其析构函数 FileStorage::release()实现。FileStorage::release()函数会析构掉 FileStorage 类对象，同时关闭文件。

调用过程非常简单，如下。

```
fs.release();
```

下面将通过实例来帮助大家将以上知识融会贯通。

5.6.3 示例程序：XML 和 YAML 文件的写入

让我们先看一个关于 XML 或 YAML 文件的写入实例，示例代码如下：

```
#include "opencv2/opencv.hpp"
#include <time.h>

using namespace cv;

int main()
{
    //初始化
    FileStorage fs("test.yaml", FileStorage::WRITE);

    //开始文件写入
    fs << "frameCount" << 5;
    time_t rawtime; time(&rawtime);
    fs << "calibrationDate" << asctime(localtime(&rawtime));
    Mat cameraMatrix = (Mat_<double>(3,3) << 1000, 0, 320, 0, 1000, 240,
0, 0, 1);
    Mat distCoeffs = (Mat_<double>(5,1) << 0.1, 0.01, -0.001, 0, 0);
    fs << "cameraMatrix" << cameraMatrix << "distCoeffs" << distCoeffs;
    fs << "features" << "[";
    for( int i = 0; i < 3; i++ )
    {
        int x = rand() % 640;
        int y = rand() % 480;
        uchar lbp = rand() % 256;

        fs << "{:" << "x" << x << "y" << y << "lbp" << "[:";
        for( int j = 0; j < 8; j++ )
         fs << ((lbp >> j) & 1);
        fs << "]" << "}";
    }
    fs << "]";
    fs.release();

    printf("文件读写完毕，请在工程目录下查看生成的文件～");
    getchar();

    return 0;
}
```

运行此程序，会在工程目录下程序一个名为“test.yaml”的文件，用记事本或

者其他文本编辑器比如notepad++打开它，可以发现写入的YAML文档内容如下。

```
%YAML:1.0
frameCount: 5
calibrationDate: "Wed Jun 25 10:20:17 2014\n"
cameraMatrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 1000., 0., 320., 0., 1000., 240., 0., 0., 1. ]
distCoeffs: !!opencv-matrix
   rows: 5
   cols: 1
   dt: d
   data: [ 1.0000000000000001e-001, 1.0000000000000000e-002,
      -1.0000000000000000e-003, 0., 0. ]
features:
   - { x:41, y:227, lbp:[ 0, 1, 1, 1, 1, 1, 0, 1 ] }
   - { x:260, y:449, lbp:[ 0, 0, 1, 1, 0, 1, 1, 0 ] }
   - { x:598, y:78, lbp:[ 0, 1, 0, 0, 1, 0, 1, 0 ] }
```

如果我们修改上面代码的FileStorage fs（“test.yaml”，FileStorage::WRITE）这一句代码，将其中的“test .yaml”的后缀换为xml、yml、txt甚至doc，都是可以得到运行结果的，大家不妨尝试一下。

5.6.4 示例程序：XML和YAML文件的读取

接着，我们一起看读操作应该如何进行，完整的源代码如下。

```
#include "opencv2/opencv.hpp"
#include <time.h>

using namespace cv;
using namespace std;

int main()
{
    //改变console字体颜色
    system("color 6F");

    //初始化
    FileStorage fs2("test.yaml", FileStorage::READ);

    // 第一种方法，对FileNode操作
    int frameCount = (int)fs2["frameCount"];

    std::string date;
    // 第二种方法，使用FileNode运算符>>
    fs2["calibrationDate"] >> date;

    Mat cameraMatrix2, distCoeffs2;
    fs2["cameraMatrix"] >> cameraMatrix2;
```

```
    fs2["distCoeffs"] >> distCoeffs2;

    cout << "frameCount: " << frameCount << endl
        << "calibration date: " << date << endl
        << "camera matrix: " << cameraMatrix2 << endl
        << "distortion coeffs: " << distCoeffs2 << endl;

    FileNode features = fs2["features"];
    FileNodeIterator it = features.begin(), it_end = features.end();
    int idx = 0;
    std::vector<uchar> lbpval;

    //使用 FileNodeIterator 遍历序列
    for( ; it != it_end; ++it, idx++ )
    {
        cout << "feature #" << idx << ": ";
        cout << "x=" << (int)(*it)["x"] << ", y=" << (int)(*it)["y"] <<
", lbp: (";
        // 我们也可以使用 filenode > > std::vector 操作符来很容易地读数值阵列
        (*it)["lbp"] >> lbpval;
        for( int i = 0; i < (int)lbpval.size(); i++ )
         cout << " " << (int)lbpval[i];
        cout << ")" << endl;
    }
    fs2.release();

    //程序结束，输出一些帮助文字
    printf("\n 文件读取完毕，请输入任意键结束程序～");
    getchar();

    return 0;
}
```

此程序需要工程目录下有指定的文件存在，如我们将前文刚刚生成的 test.yaml 复制到工程目录下，运行此程序，便可以得出正确的结果。运行截图如图 5.17 所示。

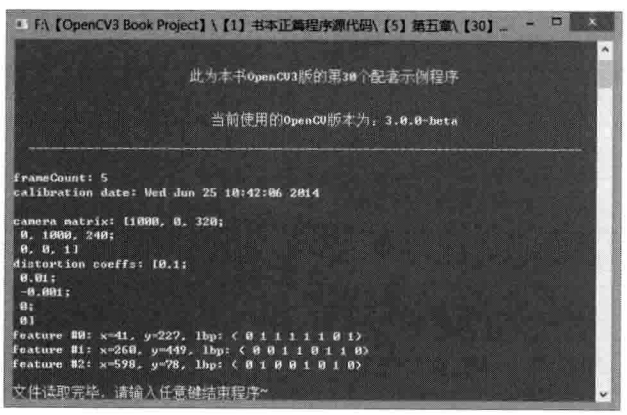


图 5.17 运行输出截图

5.7 本章小结

本节中，我们学习了 core 模块的一些进阶知识点——操作图像中的像素、图像混合、分离颜色通道、调节图像的对比度和亮度、进行离散傅里叶变换，以及输入输出 XML 和 YAML 文件。

本章核心函数清单

函数名称	说明	对应讲解章节
addWeighted	计算两个数组（图像阵列）的加权和	5.2.3
split	将一个多通道数组分离成几个单通道数组	5.3.1
merge	将多个数组组合合并成一个多通道的数组	5.3.2
dft	对一维或二维浮点数数组进行正向或反向离散傅里叶变换	5.5.2
getOptimalDFTSize	返回给定向量尺寸的傅里叶最优尺寸大小	5.5.3
copyMakeBorder	扩充图像边界	5.5.4
magnitude	计算二维矢量的幅值	5.5.5
log	计算每个数组元素绝对值的自然对数	5.5.6
normalize	进行矩阵归一化	5.5.7
FileStorage 类	进行文件操作的类	5.6.2

本章示例程序清单

示例程序序号	程序说明	对应章节
21	操作图像中像素的方法一：用指针访问像素	5.1.5、5.1.6
22	操作图像中像素的方法二：用迭代器操作像素	5.1.5、5.1.6
23	操作图像中像素的方法三：动态地址计算	5.1.5、5.1.6
24	遍历图像中像素的 14 种方法	5.1.6
25	初级图像混合	5.2.4
26	多通道图像混合	5.3.3
27	图像对比度、亮度值调整	5.4.3
28	离散傅里叶变换	5.5.8
29	XML 和 YAML 文件的写入	5.6.3
30	XML 和 YAML 文件的读取	5.6.4

第三部分
掌握 imgproc 组件

imgproc 组件是 Image 和 Process 这两个单词的缩写组合，即图像处理模块，这个模块包含了如下内容：

- 线性和非线性的图像滤波
- 图像的几何变换
- 其他（Miscellaneous）图像转换
- 直方图相关
- 结构分析和形状描述
- 运动分析和对象跟踪
- 特征检测
- 目标检测等内容

第三部分包含如下四个章节：

第 6 章
图像处理

导读

到这里，我们已经了解了 OpenCV 库的结构，知道了用来表示图像的基本数据结构，也熟悉了运行程序并将结果显示到屏幕上的 Highgui 接口。现在，我们可以利用这些控制图像结构的基本方法，来学习更加复杂的图像处理方法。

所谓的学习高级的处理方法，即把图像以“图像”的方式来处理，而不是作为只由颜色值或灰度图组成的数组。而我们在提到“图像处理”时，就是表示使用图像结构中所定义的高层处理方法来完成特殊任务，而这些任务，就是图形和视觉范畴的任务。

本章中，你将学到：

- 三种线性滤波：方框滤波、均值滤波、高斯滤波
- 两种非线性滤波：中值滤波、双边滤波
- 7 种图像处理形态学：腐蚀、膨胀，开运算、闭运算、形态学梯度、顶帽、黑帽
- 漫水填充
- 图像缩放
- 图像金字塔
- 阈值化

6.1 线性滤波：方框滤波、均值滤波、高斯滤波

6.1.1 平滑处理

平滑处理（smoothing）也称模糊处理（bluring），是一种简单且使用频率很高的图像处理方法。平滑处理的用途有很多，最常见的是用来减少图像上的噪点或者失真。在涉及到降低图像分辨率时，平滑处理是非常好用的方法。

6.1.2 图像滤波与滤波器

图像滤波，指在尽量保留图像细节特征的条件下对目标图像的噪声进行抑制，是图像预处理中不可缺少的操作，其处理效果的好坏将直接影响到后续图像处理和分析的有效性和可靠性。

消除图像中的噪声成分叫作图像的平滑化或滤波操作。信号或图像的能量大部分集中在幅度谱的低频和中频段，而在较高频段，有用的信息经常被噪声淹没。因此一个能降低高频成分幅度的滤波器就能够减弱噪声的影响。

图像滤波的目的有两个：一个是抽出对象的特征作为图像识别的特征模式；另一个是为适应图像处理的要求，消除图像数字化时所混入的噪声。

而对滤波处理的要求也有两条：一是不能损坏图像的轮廓及边缘等重要信息；二是使图像清晰视觉效果好。

平滑滤波是低频增强的空间域滤波技术。它的目的有两类：一类是模糊；另一类是消除噪音。

空间域的平滑滤波一般采用简单平均法进行，就是求邻近像元点的平均亮度值。邻域的大小与平滑的效果直接相关，邻域越大平滑的效果越好，但邻域过大，平滑也会使边缘信息损失的越大，从而使输出的图像变得模糊，因此需合理选择邻域的大小。

关于滤波器，一种形象的比喻是：可以把滤波器想象成一个包含加权系数的窗口，当使用这个滤波器平滑处理图像时，就把这个窗口放到图像之上，透过这个窗口来看我们得到的图像。

滤波器的种类有很多， 在新版本的 OpenCV 中，提供了如下 5 种常用的图像平滑处理操作方法，它们分别被封装在单独的函数中，使用起来非常方便。

- 方框滤波——BoxBlur 函数
- 均值滤波（邻域平均滤波）——Blur 函数
- 高斯滤波——GaussianBlur 函数
- 中值滤波——medianBlur 函数
- 双边滤波——bilateralFilter 函数

本节要讲解的是作为线性滤波的方框滤波、均值滤波和高斯滤波。其他两种

非线性滤波操作——中值滤波和双边滤波，我们留待下节讲解。

6.1.3 线性滤波器的简介

线性滤波器：线性滤波器经常用于剔除输入信号中不想要的频率或者从许多频率中选择一个想要的频率。

几种常见的线性滤波器如下。

- 低通滤波器：允许低频率通过；
- 高通滤波器：允许高频率通过；
- 带通滤波器：允许一定范围频率通过；
- 带阻滤波器：阻止一定范围频率通过并且允许其他频率通过；
- 全通滤波器：允许所有频率通过，仅仅改变相位关系；
- 陷波滤波器（Band-Stop Filter）：阻止一个狭窄频率范围通过，是一种特殊带阻滤波器。

6.1.4 滤波和模糊

关于滤波和模糊，大家往往在初次接触的时候会弄混淆："一会儿说滤波，一会儿又说模糊，似乎不太清楚。"

没关系，在这里，我们就来分析一下，为大家扫清障碍。

上文已经提到过，滤波是将信号中特定波段频率滤除的操作，是抑制和防止干扰的一项重要措施。

为了方便说明，就拿我们经常用的高斯滤波来作例子吧。滤波可分低通滤波和高通滤波两种：高斯滤波是指用高斯函数作为滤波函数的滤波操作，至于是不是模糊，要看是高斯低通还是高斯高通，低通就是模糊，高通就是锐化。

其实说白了是很简单的：

- 高斯滤波是指用高斯函数作为滤波函数的滤波操作；
- 高斯模糊就是高斯低通滤波。

6.1.5 邻域算子与线性邻域滤波

邻域算子（局部算子）是利用给定像素周围的像素值的决定此像素的最终输出值的一种算子。而线性邻域滤波就是一种常用的邻域算子，像素的输出值取决于输入像素的加权和，具体过程如图 6.1 所示。

邻域算子除了用于局部色调调整以外，还可以用于图像滤波，以实现图像的平滑和锐化，图像边缘增强或者图像噪声的去除。本节我们介绍的主角是线性邻域滤波算子，即用不同的权重去结合一个小邻域内的像素，来得到应有的处理效果。

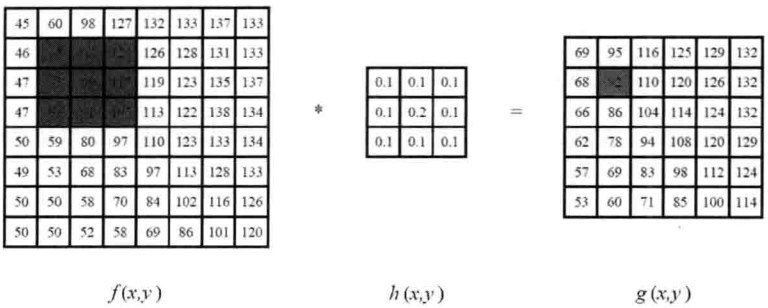


图 6.1 邻域滤波（卷积）

图注：邻域滤波（卷积）——左边图像与中间图像的卷积产生右边图像。目标图像中蓝色标记的像素是利用原图像中红色标记的像素计算得到的。

线性滤波处理的输出像素值 $g(i,j)$是输入像素值 $f(i+k,j+I)$的加权和，如下

$$g(i,j)=\sum_{k,I} f(i+k,j+I)h(k,I)$$

其中的 $h(k,I)$，我们称其为“核”，是滤波器的加权系数，即滤波器的“滤波系数”。

上面的式子可以简单写作：

$$g=f\otimes h$$

其中 f 表示输入像素值，h 表示加权系数“核”，g 表示输出像素值。

在新版本的 OpenCV 中，提供了如下三种常用的线性滤波操作，它们分别被封装在单独的函数中，使用起来非常方便。

- 方框滤波——boxblur 函数
- 均值滤波——blur 函数
- 高斯滤波——GaussianBlur 函数

下面我们来对它们进行一一介绍。

6.1.6 方框滤波（box Filter）

方框滤波（box Filter）被封装在一个名为 boxblur 的函数中，即 boxblur 函数的作用是使用方框滤波器（box filter）来模糊一张图片，从 src 输入，从 dst 输出。

函数原型如下。

```
C++: void boxFilter(InputArray src,OutputArray dst, int ddepth, Size
ksize, Point anchor=Point(-1,-1), boolnormalize=true, int
borderType=BORDER_DEFAULT )
```

参数详解如下。

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对

象即可。该函数对通道是独立处理的，且可以处理任意通道数的图片。但需要注意，待处理的图片深度应该为 CV_8U、CV_16U、CV_16S、CV_32F 以及 CV_64F 之一。

- 第二个参数，OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。
- 第三个参数，int 类型的 ddepth，输出图像的深度，-1 代表使用原图深度，即 src.depth()。
- 第四个参数，Size 类型（对 Size 类型稍后有讲解）的 ksize，内核的大小。一般用 Size(w,h)来表示内核的大小，其中 w 为像素宽度，h 为像素高度。Size（3,3）就表示 3x3 的核大小，Size（5,5）就表示 5x5 的核大小。
- 第五个参数，Point 类型的 anchor，表示锚点（即被平滑的那个点）。注意它有默认值 Point(-1,-1)。如果这个点坐标是负值的话，就表示取核的中心为锚点，所以默认值 Point(-1,-1)表示这个锚点在核的中心。
- 第六个参数，bool 类型的 normalize，默认值为 true，一个标识符，表示内核是否被其区域归一化（normalized）了。
- 第七个参数，int 类型的 borderType，用于推断图像外部像素的某种边界模式。有默认值 BORDER_DEFAULT，我们一般不去管它。

BoxFilter()函数方框滤波所用的核表示如下。

$$K = a\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

其中：

$$a\begin{cases} \dfrac{1}{\text{ksize.width*ksize.height}} & \text{当normalize = ture时} \\ 1 & \text{当normalize = false时} \end{cases}$$

上式中 f 表示原图，h 表示核，g 表示目标图，当 normalize=true 的时候，方框滤波就变成了我们熟悉的均值滤波。也就是说，均值滤波是方框滤波归一化（normalized）后的特殊情况。其中，归一化就是把要处理的量都缩放到一个范围内，比如（0,1），以便统一处理和直观量化。而非归一化（Unnormalized）的方框滤波用于计算每个像素邻域内的积分特性，比如密集光流算法（dense optical flow algorithms）中用到的图像倒数的协方差矩阵（covariance matrices of image derivatives）。

如果我们要在可变的窗口中计算像素总和，可以使用 integral()函数。

6.1.7 均值滤波

均值滤波，是最简单的一种滤波操作，输出图像的每一个像素是核窗口内输入图像对应像素的平均值（所有像素加权系数相等），其实说白了它就是归一化后

的方框滤波。我们在下文进行源码剖析时会发现，blur 函数内部中其实就是调用了一下 boxFilter。

下面开始讲均值滤波的内容吧。

1. 均值滤波的理论简析

均值滤波是典型的线性滤波算法，主要方法为邻域平均法，即用一片图像区域的各个像素的均值来代替原图像中的各个像素值。一般需要在图像上对目标像素给出一个模板（内核），该模板包括了其周围的临近像素（比如以目标像素为中心的周围 8（3x3-1）个像素，构成一个滤波模板，即去掉目标像素本身）。再用模板中的全体像素的平均值来代替原来像素值。即对待处理的当前像素点（x,y），选择一个模板，该模板由其近邻的若干像素组成，求模板中所有像素的均值，再把该均值赋予当前像素点（x,y），作为处理后图像在该点上的灰度点 g（x,y），即 g（x,y）=1/m ∑f（x,y），其中 *m* 为该模板中包含当前像素在内的像素总个数。

2. 均值滤波的缺陷

均值滤波本身存在着固有的缺陷，即它不能很好地保护图像细节，在图像去噪的同时也破坏了图像的细节部分，从而使图像变得模糊，不能很好地去除噪声点。

3. 在 OpenCV 中使用均值滤波——blur 函数

blur 函数的作用是：对输入的图像 src 进行均值滤波后用 dst 输出。

blur 函数在 OpenCV 官方文档中，给出的其核是这样的：

$$K=\frac{1}{\text{ksize.width*ksize.height}}\begin{bmatrix}1 & 1 & 1 & \cdots & 1 & 1\\ 1 & 1 & 1 & \cdots & 1 & 1\\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots\\ 1 & 1 & 1 & \cdots & 1 & 1\end{bmatrix}$$

这个内核一看就明了，就是在求均值，即 blur 函数封装的就是均值滤波。

blur 函数的原型如下。

```
C++: void blur(InputArray src, OutputArraydst, Size ksize, Point
anchor=Point(-1,-1), int borderType=BORDER_DEFAULT )
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。该函数对通道是独立处理的，且可以处理任意通道数的图片。但需要注意的是，待处理的图片深度应该为 CV_8U、CV_16U、CV_16S、CV_32F 以及 CV_64F 之一。
- 第二个参数，OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。比如可以用 Mat::Clone，以源图片为模板，来初始化得到如假包换的目标图。
- 第三个参数，Size 类型（对 Size 类型稍后有讲解）的 ksize，内核的大小。

一般写作 Size(w,h)来表示内核的大小（其中，w 为像素宽度，h 为像素高度）。Size(3,3)就表示 3×3 的核大小，Size(5,5)就表示 5×5 的核大小

- 第四个参数，Point 类型的 anchor，表示锚点（即被平滑的那个点），注意它有默认值 Point(-1,-1)。如果这个点坐标是负值，就表示取核的中心为锚点，所以默认值 Point(-1,-1)表示这个锚点在核的中心。
- 第五个参数，int 类型的 borderType，用于推断图像外部像素的某种边界模式。有默认值 BORDER_DEFAULT，我们一般不去管它。

6.1.8 高斯滤波

1. 高斯滤波的理论简析

高斯滤波是一种线性平滑滤波，可以消除高斯噪声，广泛应用于图像处理的减噪过程。通俗地讲，高斯滤波就是对整幅图像进行加权平均的过程，每一个像素点的值，都由其本身和邻域内的其他像素值经过加权平均后得到。高斯滤波的具体操作是：用一个模板（或称卷积、掩模）扫描图像中的每一个像素，用模板确定的邻域内像素的加权平均灰度值去替代模板中心像素点的值。

大家常说高斯滤波是最有用的滤波操作，虽然它用起来效率往往不是最高的。高斯模糊技术生成的图像，其视觉效果就像是经过一个半透明屏幕在观察图像，这与镜头焦外成像效果散景以及普通照明阴影中的效果都明显不同。高斯平滑也用于计算机视觉算法中的预先处理阶段，以增强图像在不同比例大小下的图像效果（参见尺度空间表示以及尺度空间实现）。从数学的角度来看，图像的高斯模糊过程就是图像与正态分布做卷积。由于正态分布又叫作高斯分布，所以这项技术就叫作高斯模糊。

图像与圆形方框模糊做卷积将会生成更加精确的焦外成像效果。由于高斯函数的傅里叶变换是另外一个高斯函数，所以高斯模糊对于图像来说就是一个低通滤波操作。

高斯滤波器是一类根据高斯函数的形状来选择权值的线性平滑滤波器。高斯平滑滤波器对于抑制服从正态分布的噪声非常有效。一维零均值高斯函数如下。

$$G(x)=\exp(-x^2/(2\text{sigma}^2))$$

其中，高斯分布参数 Sigma 决定了高斯函数的宽度。对于图像处理来说，常用二维零均值离散高斯函数作平滑滤波器。

二维高斯函数如下。

$$G_0(x,y)=Ae^{\frac{-(x-u_x)^2}{2\sigma_x^2}+\frac{-(y-u_y)^2}{2\sigma_y^2}}$$

2. 高斯滤波：GaussianBlur 函数

GaussianBlur 函数的作用是用高斯滤波器来模糊一张图片，对输入的图像 src

进行高斯滤波后用 dst 输出。它将源图像和指定的高斯核函数做卷积运算，并且支持就地过滤（In-placefiltering）。

```
C++: void GaussianBlur(InputArray src,OutputArray dst, Size ksize,
double sigmaX, double sigmaY=0, intborderType=BORDER_DEFAULT )
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。它可以是单独的任意通道数的图片，但需要注意的是，其图片深度应该为 CV_8U、CV_16U、CV_16S、CV_32F 以及 CV_64F 之一。
- 第二个参数，OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。比如可以用 Mat::Clone，以源图片为模板，来初始化得到如假包换的目标图。
- 第三个参数，Size 类型的 ksize 高斯内核的大小。其中 ksize.width 和 ksize.height 可以不同，但它们都必须为正数和奇数，或者是零，这都由 sigma 计算而来。
- 第四个参数，double 类型的 sigmaX，表示高斯核函数在 X 方向的的标准偏差。
- 第五个参数，double 类型的 sigmaY，表示高斯核函数在 Y 方向的的标准偏差。若 sigmaY 为零，就将它设为 sigmaX；如果 sigmaX 和 sigmaY 都是 0，那么就由 ksize.width 和 ksize.height 计算出来。

为了结果的正确性着想，最好是把第三个参数 Size、第四个参数 sigmaX 和第五个参数 sigmaY 全部指定到。

- 第六个参数，int 类型的 borderType，用于推断图像外部像素的某种边界模式。有默认值 BORDER_DEFAULT，我们一般不去管它。

6.1.9 线性滤波相关 OpenCV 源码剖析

在这一部分中，笔者将带领大家领略 OpenCV 的开源魅力，对 OpenCV 中节讲解到的线性滤波函数——boxFilter、blur 和 GaussianBlur 函数以及周边所涉及到的源码进行适当的剖析。

通过本章的学习，我们可以对 OpenCV 有更加深刻的理解，成为一个高端大气的 OpenCV 使用者。

1. OpenCV 中 boxFilter 函数源码解析

可以在 OpenCV 的安装路径\sources\modules\imgproc\src 下的 smooth.cpp 源文件中找到 boxFilter 函数的源代码，如下。

```
void cv::boxFilter( InputArray _src,OutputArray _dst, int ddepth,
                Size ksize, Point anchor,
                bool normalize, int borderType)
{
  Mat src = _src.getMat();//复制源图的形参 Mat 数据到临时变量，用于稍后的操
作
```

```
int sdepth =src.depth(), cn = src.channels();//定义 int 型临时变量，代表
源图深度的 sdepth，源图通道的引用 cn

//处理 ddepth 小于零的情况
   if( ddepth < 0 )
      ddepth = sdepth;
   _dst.create( src.size(), CV_MAKETYPE(ddepth, cn) );//初始化目标图
Mat dst =_dst.getMat();//复制目标图的形参 Mat 数据到临时变量，用于稍后的操作

//处理 borderType 不为 BORDER_CONSTANT 且 normalize 为真的情况
   if( borderType != BORDER_CONSTANT && normalize )    {
      if( src.rows == 1 )
         ksize.height = 1;
      if( src.cols == 1 )
         ksize.width = 1;
}

//若之前有过 HAVE_TEGRA_OPTIMIZATION 优化选项的定义，则执行宏体中的 tegra 优化
版函数并返回
#ifdef HAVE_TEGRA_OPTIMIZATION
   if ( tegra::box(src, dst, ksize, anchor, normalize, borderType) )
      return;
#endif

      //调用 FilterEngine 滤波引擎，正式开始滤波操作
   Ptr<FilterEngine> f = createBoxFilter( src.type(), dst.type(),
                     ksize, anchor,normalize, borderType );
   f->apply( src, dst );
}
```

其中，Ptr 是用来动态分配的对象的智能指针模板类。可以发现，函数的内部代码思路是很清晰的：先复制源图的形参 Mat 数据到临时变量，定义一些临时变量，再处理 ddepth 小于零的情况，接着处理 borderType 不为 BORDER_CONSTANT 且 normalize 为真的情况，最终调用 FilterEngine 滤波引擎创建一个 BoxFilter，正式开始滤波操作。

这里的 FilterEngine 是 OpenCV 图像滤波功能的核心引擎，我们有必要详细剖析看其源代码。

2. FilterEngine 类解析：OpenCV 图像滤波核心引擎

FilterEngine 类是 OpenCV 关于图像滤波的主力军类，是 OpenCV 图像滤波功能的核心引擎。各种滤波函数如 blur、GaussianBlur，其实是就是在函数末尾处定义了一个 Ptr<FilterEngine>类型的 f，然后 f->apply(src，dst)了一下而已。

这个类可以把几乎所有的滤波操作施加到图像上，它包含了所有必要的中间缓存器。有很多和滤波相关的 create 系函数的返回值直接就是 Ptr<FilterEngine>。比如

- cv::createSeparableLinearFilter()

- cv::createLinearFilter(),cv::createGaussianFilter(), cv::createDerivFilter()
- cv::createBoxFilter()
- cv::createMorphologyFilter()

下面给出其中一个函数的原型。

```
Ptr<FilterEngine>createLinearFilter(int srcType, int dstType,
InputArray kernel, Point_anchor=Point(-1,-1), double delta=0, int
rowBorderType=BORDER_DEFAULT, intcolumnBorderType=-1, const Scalar&
borderValue=Scalar() )
```

上面提到过，其中的 Ptr 是用来动态分配的对象的智能指针模板类，而尖括号里的模板参数就是 FilterEngine。

使用 FilterEngine 类可以分块处理大量的图像，构建复杂的管线，其中就包含一些进行滤波阶段。如果我们需要使用预先定义好的的滤波操作，有 cv::filter2D()、cv::erode()和 cv::dilate()可以选择，它们不依赖于 FilterEngine，在自己函数体内部就实现了 FilterEngine 提供的功能；不像其他诸如我们今天讲的 blur 系列函数，依赖于 FilterEngine 引擎。

我们看下其类声明经过详细注释的源码，如下。

```
class CV_EXPORTS FilterEngine
{
public:
   //默认构造函数
   FilterEngine();
   //完整的构造函数。 _filter2D 、_rowFilter 和 _columnFilter 之一，必须为非
空
   FilterEngine(const Ptr<BaseFilter>& _filter2D,
                constPtr<BaseRowFilter>& _rowFilter,
               constPtr<BaseColumnFilter>& _columnFilter,
                int srcType, int dstType, intbufType,
                int_rowBorderType=BORDER_REPLICATE,
                int _columnBorderType=-1,
                const Scalar&_borderValue=Scalar());
   //默认析构函数
   virtual ~FilterEngine();
   //重新初始化引擎。释放之前滤波器申请的内存
   void init(const Ptr<BaseFilter>& _filter2D,
             constPtr<BaseRowFilter>& _rowFilter,
             constPtr<BaseColumnFilter>& _columnFilter,
             int srcType, int dstType, intbufType,
             int_rowBorderType=BORDER_REPLICATE, int
_columnBorderType=-1,
             const Scalar&_borderValue=Scalar());
   //开始对指定了 ROI 区域和尺寸的图片进行滤波操作
   virtual int start(Size wholeSize, Rect roi, int maxBufRows=-1);
    //开始对指定了 ROI 区域的图片进行滤波操作
   virtual int start(const Mat& src, const Rect&srcRoi=Rect(0,0,-1,-1),
                     bool isolated=false, intmaxBufRows=-1);
   //处理图像的下一个 srcCount 行（函数的第三个参数）
```

```
    virtual int proceed(const uchar* src, int srcStep, int srcCount,
                        uchar* dst, intdstStep);
    //对图像指定的 ROI 区域进行滤波操作，若 srcRoi=(0,0,-1,-1)，则对整个图像进行
滤波操作
    virtual void apply( const Mat& src, Mat& dst,
                        const Rect&srcRoi=Rect(0,0,-1,-1),
                        Point dstOfs=Point(0,0),
                        bool isolated=false);

    //如果滤波器可分离，则返回 true
boolisSeparable() const { return (const BaseFilter*)filter2D == 0; }

    //返回输入和输出行数
    int remainingInputRows() const;
intremainingOutputRows() const;

    //一些成员参数定义
    int srcType, dstType, bufType;
    Size ksize;
    Point anchor;
    int maxWidth;
    Size wholeSize;
    Rect roi;
    int dx1, dx2;
    int rowBorderType, columnBorderType;
    vector<int> borderTab;
    int borderElemSize;
    vector<uchar> ringBuf;
    vector<uchar> srcRow;
    vector<uchar> constBorderValue;
    vector<uchar> constBorderRow;
    int bufStep, startY, startY0, endY, rowCount, dstY;
    vector<uchar*> rows;

    Ptr<BaseFilter> filter2D;
    Ptr<BaseRowFilter> rowFilter;
    Ptr<BaseColumnFilter> columnFilter;
};
```

3. OpenCV 中 blur 函数源码剖析

我们可以在 OpenCV 的安装路径\sources\modules\imgproc\src 下的 smooth.cpp 源文件中找到 blur 的源代码，下面一起来看 OpenCV 中 blur 函数定义的真面目。

```
void cv::blur(InputArray src, OutputArray dst,
        Size ksize, Point anchor, int borderType )
{
//调用 boxFilter 函数进行处理
    boxFilter( src, dst, -1, ksize, anchor, true, borderType );
}
```

可以发现，在 blur 函数内部调用了一个 boxFilter 函数，且第 6 个参数为 true，也就是我们上文所说的 normalize=true，即均值滤波是均一化后的方框滤波。

6.1.10 OpenCV 中 GaussianBlur 函数源码剖析

下面我们看一下 OpenCV 中 GaussianBlur 函数源代码：

```
void cv::GaussianBlur( InputArray _src,OutputArray _dst, Size ksize,
                double sigma1, doublesigma2,
                int borderType )
{
//复制形参 Mat 数据到临时变量，用于稍后的操作
   Mat src = _src.getMat();
   _dst.create( src.size(), src.type() );
Mat dst =_dst.getMat();

//处理边界选项不为 BORDER_CONSTANT 时的情况
   if( borderType != BORDER_CONSTANT )
    {
      if( src.rows == 1 )
         ksize.height = 1;
      if( src.cols == 1 )
         ksize.width = 1;
}

      //若 ksize 长宽都为 1，将源图复制给目标图
   if( ksize.width == 1 && ksize.height == 1 )
    {
      src.copyTo(dst);
      return;
}

//若之前有过 HAVE_TEGRA_OPTIMIZATION 优化选项的定义，则执行宏体中的 tegra 优化
版函数并返回
#ifdef HAVE_TEGRA_OPTIMIZATION
   if(sigma1 == 0 && sigma2 == 0 && tegra::gaussian(src,dst, ksize,
borderType))
      return;
#endif

//如果 HAVE_IPP&& (IPP_VERSION_MAJOR >= 7 为真，则执行宏体中语句
#if defined HAVE_IPP &&(IPP_VERSION_MAJOR >= 7)
   if(src.type() == CV_32FC1 && sigma1 == sigma2 &&ksize.width ==
ksize.height && sigma1 != 0.0 )
    {
      IppiSize roi = {src.cols, src.rows};
      int bufSize = 0;
      ippiFilterGaussGetBufferSize_32f_C1R(roi, ksize.width,
&bufSize);
      AutoBuffer<uchar> buf(bufSize+128);
      if( ippiFilterGaussBorder_32f_C1R((const Ipp32f
```

```
*)src.data,(int)src.step,
                                        (Ipp32f *)dst.data, (int)dst.step,
                                        roi,ksize.width, (Ipp32f)sigma1,
                                        (IppiBorderType)borderType, 0.0,
                                        alignPtr(&buf[0],32)) >= 0 )
            return;
        }
#endif

    //调用滤波引擎，正式进行高斯滤波操作
    Ptr<FilterEngine> f = createGaussianFilter( src.type(), ksize,sigma1,
sigma2, borderType );
    f->apply( src, dst );
}
```

通过本节的源码解析，相信大家应该对 OpenCV 中的线性滤波有了比较详细的认识，已经跃跃欲试，想看这个几个函数用起来可以得出什么效果了。

6.1.11 线性滤波核心 API 函数

本节的内容就是为了大家能快速上手 boxFilter、blur 和 GaussianBlur 这三个函数所准备的。下面进行详细讲解。

1．方框滤波：boxFilter 函数

boxFilter 的函数作用是使用方框滤波（box filter）来模糊一张图片，由 src 输入，dst 输出。

函数原型如下。

```
C++: void boxFilter(InputArray src,OutputArray dst, int ddepth, Size
ksize, Point anchor=Point(-1,-1), boolnormalize=true, int
borderType=BORDER_DEFAULT )
```

参数详解如下。

- 第一个参数：InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。该函数对通道是独立处理的，且可以处理任意通道数的图片。但需要注意，待处理的图片深度应该是 CV_8U、CV_16U、CV_16S、CV_32F、CV_64F 之一。
- 第二个参数：OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。
- 第三个参数：int 类型的 ddepth，输出图像的深度。“-1”代表使用原图深度，即 src.depth()。
- 第四个参数：Size 类型的 ksize，内核的大小。一般用 Size(w,h)的写法来表示内核的大小（其中，w 为像素宽度，h 为像素高度）。例如，Size(3,3)表示 3x3 的核大小；Size(5,5)就表示 5x5 的核大小。
- 第五个参数：Point 类型的 anchor，表示锚点（即被平滑的那个点），注意它有默认值 Point(-1,-1)。如果这个点坐标是负值的话，就表示取核的中心

为锚点，所以默认值 Point(-1,-1)表示这个锚点在核的中心。

- 第六个参数：bool 类型的 normalize，默认值为 true，一个标识符，表示内核是否被其区域归一化（normalized）了。
- 第七个参数：int 类型的 borderType，用于推断图像外部像素的某种边界模式。有默认值 BORDER_DEFAULT，我们一般不去管它。

调用代码示范如下。

```
//载入原图
Mat image=imread("2.jpg");
//进行均值滤波操作
Mat out;
boxFilter(image, out, -1,Size(5, 5));
```

用上面三句核心代码架起来的完整程序代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include"opencv2/highgui/highgui.hpp"
#include"opencv2/imgproc/imgproc.hpp"

using namespace cv;

//----------------------------【main()函数】--------------------------
//     描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
      //载入原图
      Mat image=imread("2.jpg");

      //创建窗口
      namedWindow("均值滤波【原图】" );
      namedWindow("均值滤波【效果图】");

      //显示原图
      imshow("均值滤波【原图】", image );

      //进行滤波操作
      Mat out;
      boxFilter(image, out, -1,Size(5, 5));

      //显示效果图
      imshow("均值滤波【效果图】" ,out );

      waitKey(0 );
}
```

此程序运行效果图（内核大小为 Size(5,5)）如图 6.2 所示。

图 6.2 运行效果图（内核大小 Size(5, 5)）

2. 均值滤波：blur 函数

blur 的作用是对输入的图像 src 进行均值滤波后用 dst 输出。

函数原型如下。

```
C++: void blur(InputArray src, OutputArraydst, Size ksize, Point
anchor=Point(-1,-1), int borderType=BORDER_DEFAULT )
```

参数详解如下。

- 第一个参数：InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。该函数对通道是独立处理的，且可以处理任意通道数的图片。但需要注意，待处理的图片深度应该为 CV_8U、CV_16U、CV_16S、CV_32F、CV_64F 之一。
- 第二个参数：OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。比如可以用 Mat::Clone，以源图片为模板，来初始化得到如假包换的目标图。
- 第三个参数：Size 类型（对 Size 类型稍后有讲解）的 ksize，内核的大小。一般用 Size(w,h)的写法来表示内核的大小（其中 w 为像素宽度，h 为像素高度）。例如，Size(3,3)表示 3x3 的核大小，Size(5,5)就表示 5x5 的核大小。
- 第四个参数：Point 类型的 anchor，表示锚点（即被平滑的那个点），注意它有默认值 Point(-1,-1)。如果这个点坐标是负值的话，就表示取核的中心为锚点，所以默认值 Point(-1,-1)表示这个锚点在核的中心。
- 第五个参数：int 类型的 borderType，用于推断图像外部像素的某种边界模式。有默认值 BORDER_DEFAULT，我们一般不去管它。

调用代码示范如下。

```
//载入原图
  Mat image=imread("1.jpg");
  //进行均值滤波操作
  Mat out;
  blur(image, out, Size(7, 7));
```

用上面三句核心代码架起来的完整程序代码如下。

```
#include "opencv2/core/core.hpp"
#include"opencv2/highgui/highgui.hpp"
#include"opencv2/imgproc/imgproc.hpp"
using namespace cv;

int main()
{
      //载入原图
      Mat image=imread("1.jpg");

      //创建窗口
      namedWindow("均值滤波【原图】" );
      namedWindow("均值滤波【效果图】");

      //显示原图
      imshow("均值滤波【原图】", image );

      //进行滤波操作
      Mat out;
      blur(image, out, Size(7, 7));

      //显示效果图
      imshow("均值滤波【效果图】" ,out );

      waitKey(0 );
}
```

此程序运行效果（内核大小 Size(7,7)）如图 6.3 所示。

图 6.3 运行效果图（内核大小 Size(7,7)）

3. 高斯滤波：GaussianBlur 函数

GaussianBlur 函数的作用是用高斯滤波器来模糊一张图片，对输入的图像 src 进行高斯滤波后用 dst 输出。

函数原型如下。

```
C++: void GaussianBlur(InputArray src,OutputArray dst, Size ksize,
double sigmaX, double sigmaY=0, intborderType=BORDER_DEFAULT )
```

参数详解如下。

- 第一个参数：InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。它可以是单独的任意通道数的图片。但需要注意，图片深度应该为 CV_8U、CV_16U、CV_16S、CV_32F、CV_64F 之一。
- 第二个参数：OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。比如可以用 Mat::Clone，以源图片为模板，来初始化得到如假包换的目标图。
- 第三个参数：Size 类型的 ksize 高斯内核的大小。其中 ksize.width 和 ksize.height 可以不同，但它们都必须为正数和奇数，也可以为零。它们都是由 sigma 计算而来的。
- 第四个参数：double 类型的 sigmaX，表示高斯核函数在 X 方向的的标准偏差。
- 第五个参数：double 类型的 sigmaY，表示高斯核函数在 Y 方向的的标准偏差。若 sigmaY 为零，就将它设为 sigmaX，如果 sigmaX 和 sigmaY 都是 0，那么就由 ksize.width 和 ksize.height 计算出来。为了结果的正确性着想，最好是把第三个参数 Size、第四个参数 sigmaX 和第五个参数 sigmaY 全部指定到。
- 第六个参数：int 类型的 borderType，用于推断图像外部像素的某种边界模式。注意它有默认值 BORDER_DEFAULT。

调用示例如下。

```
//载入原图
Mat image=imread("1.jpg");
//进行滤波操作
Mat out;
GaussianBlur( image, out, Size( 5, 5 ), 0, 0 );
```

用上面三句核心代码架起来的完整程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include"opencv2/highgui/highgui.hpp"
#include"opencv2/imgproc/imgproc.hpp"
using namespace cv;

//--------------------------【main()函数】--------------------------
```

```
//    描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
    //载入原图
    Mat image=imread("1.jpg");

    //创建窗口
    namedWindow("高斯滤波【原图】" );
    namedWindow("高斯滤波【效果图】");

    //显示原图
    imshow("高斯滤波【原图】", image );

    //进行均值滤波操作
    Mat out;
    GaussianBlur(image, out, Size( 3, 3 ), 0, 0 );

    //显示效果图
    imshow("均值滤波【效果图】" ,out );

    waitKey(0 );
}
```

此程序运行效果（内核大小 Size(5,5)）如图 6.4 所示。

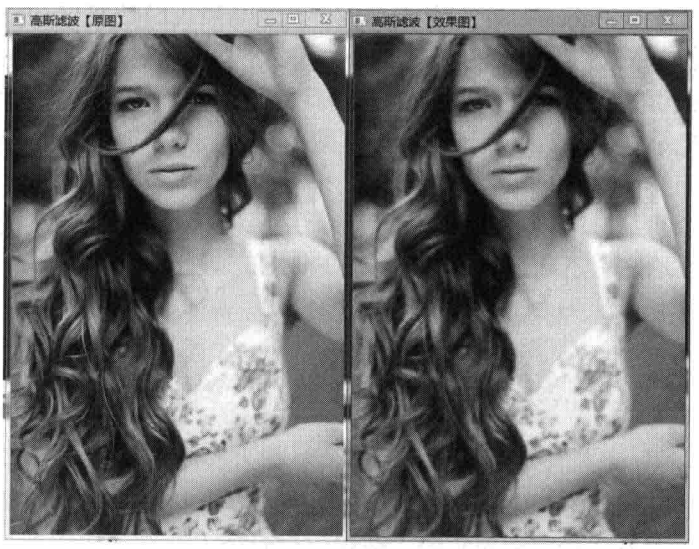


图 6.4 运行效果图（内核大小 Size(5,5)）

6.1.12 图像线性滤波综合示例

本节我们将学习一个综合性示例程序，把前文介绍的知识点以代码为载体，展现给大家。

这个示例程序中可以用滑动条来控制三种线性滤波的核参数值。通过滑动滚动条，就可以控制图像在三种线性滤波下的模糊度，有一定的可玩性。详细注释

的代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
using namespace std;
using namespace cv;


//-------------------------【全局变量声明部分】--------------------------
//     描述：全局变量声明
//-------------------------------------------------------------------------
Mat g_srcImage,g_dstImage1,g_dstImage2,g_dstImage3;//存储图片的 Mat 类型
int g_nBoxFilterValue=3;  //方框滤波参数值
int g_nMeanBlurValue=3;  //均值滤波参数值
int g_nGaussianBlurValue=3;  //高斯滤波参数值


//-------------------------【全局函数声明部分】--------------------------
//     描述：全局函数声明
//-------------------------------------------------------------------------
//轨迹条的回调函数
static void on_BoxFilter(int, void *);           //方框滤波
static void on_MeanBlur(int, void *);           //均值滤波
static void on_GaussianBlur(int, void *);                  //高斯滤波


//-----------------------------【main()函数】--------------------------
//     描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
      //改变 console 字体颜色
      system("color5E");

      //载入原图
      g_srcImage= imread( "1.jpg", 1 );
      if(!g_srcImage.data ) { printf("读取 srcImage 错误~! \n"); return
false; }

      //复制原图到三个 Mat 类型中
      g_dstImage1= g_srcImage.clone( );
      g_dstImage2= g_srcImage.clone( );
      g_dstImage3= g_srcImage.clone( );

      //显示原图
      namedWindow("【<0>原图窗口】", 1);
```

```
    imshow("【<0>原图窗口】",g_srcImage);


    //=================【<1>方框滤波】==================
    //创建窗口
    namedWindow("【<1>方框滤波】", 1);
    //创建轨迹条
    createTrackbar("内核值:", "【<1>方框滤波】",&g_nBoxFilterValue,
40,on_BoxFilter );
    on_MeanBlur(g_nBoxFilterValue,0);
    imshow("【<1>方框滤波】", g_dstImage1);
    //=================================================

    //=================【<2>均值滤波】==================
    //创建窗口
    namedWindow("【<2>均值滤波】", 1);
    //创建轨迹条
    createTrackbar("内核值:", "【<2>均值滤波】",&g_nMeanBlurValue,
40,on_MeanBlur );
    on_MeanBlur(g_nMeanBlurValue,0);
    //=================================================

    //=================【<3>高斯滤波】====================
    //创建窗口
    namedWindow("【<3>高斯滤波】", 1);
    //创建轨迹条
    createTrackbar("内核值:", "【<3>高斯滤波】",&g_nGaussianBlurValue,
40,on_GaussianBlur );
    on_GaussianBlur(g_nGaussianBlurValue,0);
    //=================================================


    //输出一些帮助信息
    cout<<endl<<"\t,请调整滚动条观察图像效果~\n\n"
        <<"\t按下"q"键时,程序退出~!\n" ;

    //按下"q"键时,程序退出
    while(char(waitKey(1))!= 'q') {}

    return 0;
}


//-----------------------【on_BoxFilter()函数】----------------------
//    描述:方框滤波操作的回调函数
//-------------------------------------------------------------------
static void on_BoxFilter(int, void *)
{
    //方框滤波操作
    boxFilter(g_srcImage, g_dstImage1, -1,Size( g_nBoxFilterValue+1,
g_nBoxFilterValue+1));
    //显示窗口
```

```
        imshow("【<1>方框滤波】", g_dstImage1);
}


//-------------------------【on_MeanBlur()函数】-------------------------
//      描述：均值滤波操作的回调函数
//-----------------------------------------------------------------------
static void on_MeanBlur(int, void *)
{
        //均值滤波操作
        blur(g_srcImage, g_dstImage2, Size( g_nMeanBlurValue+1,
g_nMeanBlurValue+1),Point(-1,-1));
        //显示窗口
        imshow("【<2>均值滤波】", g_dstImage2);
}


//-----------------------【on_GaussianBlur()函数】-----------------------
//      描述：高斯滤波操作的回调函数
//-----------------------------------------------------------------------
static void on_GaussianBlur(int, void *)
{
        //高斯滤波操作
        GaussianBlur(g_srcImage, g_dstImage3,
Size( g_nGaussianBlurValue*2+1,g_nGaussianBlurValue*2+1 ), 0, 0);
        //显示窗口
        imshow("【<3>高斯滤波】", g_dstImage3);
}
```

代码注释已经十分详细了，最后一起看一下程序运行得到的效果，如图 6.6、6.7、6.8 所示。原图如图 6.5 所示。

图 6.5 原始图

图 6.6 方框滤波效果图

图 6.7 均值滤波效果图

图 6.8 高斯滤波效果图

6.2 非线性滤波：中值滤波、双边滤波

正如我们在 6.1 节中讲到的，线性滤波可以实现很多种不同的图像变换。而非线性滤波，如中值滤波器和双边滤波器，有时可以达到更好的实现效果。

6.2.1 非线性滤波概述

在 6.1 节中，我们所考虑的滤波器都是线性的，即两个信号之和的响应和它们各自响应之和相等。换句话说，每个像素的输出值是一些输入像素的加权和。线性滤波器易于构造，并且易于从频率响应角度来进行分析。

然而，在很多情况下，使用邻域像素的非线性滤波会得到更好的效果。比如在噪声是散粒噪声而不是高斯噪声，即图像偶尔会出现很大的值的时候，用高斯滤波器对图像进行模糊的话，噪声像素是不会被去除的，它们只是转换为更为柔和但仍然可见的散粒。这就到了中值滤波登场的时候了。

6.2.2 中值滤波

中值滤波（Median filter）是一种典型的非线性滤波技术，基本思想是用像素点邻域灰度值的中值来代替该像素点的灰度值，该方法在去除脉冲噪声、椒盐噪声的同时又能保留图像的边缘细节。

中值滤波是基于排序统计理论的一种能有效抑制噪声的非线性信号处理技

术，其基本原理是把数字图像或数字序列中一点的值用该点的一个邻域中各点值的中值代替，让周围的像素值接近真实值，从而消除孤立的噪声点。这对于斑点噪声（speckle noise）和椒盐噪声（salt-and-pepper noise）来说尤其有用，因为它不依赖于邻域内那些与典型值差别很大的值。中值滤波器在处理连续图像窗函数时与线性滤波器的工作方式类似，但滤波过程却不再是加权运算。

中值滤波在一定的条件下可以克服常见线性滤波器，如最小均方滤波、方框滤波器、均值滤波等带来的图像细节模糊，而且对滤除脉冲干扰及图像扫描噪声非常有效，也常用于保护边缘信息。保存边缘的特性使它在不希望出现边缘模糊的场合也很有用，是非常经典的平滑噪声处理方法。

- 中值滤波与均值滤波器比较

优势：在均值滤波器中，由于噪声成分被放入平均计算中，所以输出受到了噪声的影响。但是在中值滤波器中，由于噪声成分很难选上，所以几乎不会影响到输出。因此同样用 3×3 区域进行处理，中值滤波消除的噪声能力更胜一筹。中值滤波无论是在消除噪声还是保存边缘方面都是一个不错的方法。

劣势：中值滤波花费的时间是均值滤波的 5 倍以上。

顾名思义，中值滤波选择每个像素的邻域像素中的中值作为输出，或者说中值滤波将每一像素点的灰度值设置为该点某邻域窗口内的所有像素点灰度值的中值。

例如，取 3×3 的函数窗，计算以点[i,j]为中心的函数窗像素中值，具体步骤如下。

（1）按强度值大小排列像素点。

（2）选择排序像素集的中间值作为点[i,j]的新值。

一般采用奇数点的邻域来计算中值，但像素点数为偶数时，中值就取排序像素中间两点的平均值。采用大小不同邻域的中值滤波器的结果如图 6.9 所示。

图 6.9 中值滤波效果图

中值滤波在一定条件下，可以克服线性滤波器（如均值滤波等）所带来的图像细节模糊，对滤除脉冲干扰即图像扫描噪声最为有效，而且在实际运算过程中并不需要图像的统计特性，也给计算带来不少方便。但是对一些细节（特别是细、尖顶等）多的图像不太适合。

6.2.3 双边滤波

双边滤波（Bilateral filter）是一种非线性的滤波方法，是结合图像的空间邻近度和像素值相似度的一种折中处理，同时考虑空域信息和灰度相似性，达到保边去噪的目的，具有简单、非迭代、局部的特点。

双边滤波器的好处是可以做边缘保存（edge preserving）。以往常用维纳滤波或者高斯滤波去降噪，但二者都会较明显地模糊边缘，对于高频细节的保护效果并不明显。双边滤波器顾名思义，比高斯滤波多了一个高斯方差 sigma－d，它是基于空间分布的高斯滤波函数，所以在边缘附近，离得较远的像素不会对边缘上的像素值影响太多，这样就保证了边缘附近像素值的保存。但是，由于保存了过多的高频信息，对于彩色图像里的高频噪声，双边滤波器不能够干净地滤掉，只能对于低频信息进行较好地滤波。

在双边滤波器中，输出像素的值依赖于邻域像素值的加权值组合，公式如下。

$$g(i,j)=\frac{\sum_{k,l} f(k,l)\omega(i,j,k,l)}{\sum_{k,l}\omega(i,j,k,l)}$$

而加权系数 w(i,j,k,l)取决于定义域核和值域核的乘积。

其中定义域核表示如下。

$$d(i,j,k,l)=\exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}\right)$$

值域核表示如下。

$$r(i,j,k,l)=\exp\left(-\frac{\|f(i,j)-f(k,l)\|^2}{2\sigma_r^2}\right)$$

定义域滤波和值域滤波如图 6.10 所示。

	2	1	0	1	2
2	0.1	0.3	0.4	0.3	0.1
1	0.3	0.6	0.8	0.6	0.3
0	0.4	0.8	1.0	0.8	0.4
1	0.3	0.6	0.8	0.6	0.3
2	0.1	0.3	0.4	0.3	0.1

定义域滤波

0.0	0.0	0.0	0.0	0.2
0.0	0.0	0.0	0.4	0.8
0.0	0.0	1.0	0.8	0.4
0.0	0.2	0.8	0.8	1.0
0.2	0.4	1.0	0.8	0.4

值域滤波

图 6.10 定义域滤波和值域滤波

两者相乘后，就会产生依赖于数据的双边滤波权重函数，如下。

$$\omega(i,j,k,l)=\exp\left(-\frac{(i-k)^2+(j-l)^2}{2\sigma_d^2}-\frac{\|f(i,j)-f(k,l)\|^2}{2\sigma_r^2}\right)$$

6.2.4 非线性滤波相关核心 API 函数

1. 中值滤波：medianBlur 函数

medianBlur 函数使用中值滤波器来平滑（模糊）处理一张图片，从 src 输入，结果从 dst 输出。对于多通道图片，它对每一个通道都单独进行处理，并且支持就地操作（In-placeoperation）。函数原型如下。

```
C++: void medianBlur(InputArray src,OutputArray dst, int ksize)
```

参数详解如下。

- 第一个参数，InputArray 类型的 src，函数的输入参数，填 1、3 或者 4 通道的 Mat 类型的图像。当 ksize 为 3 或者 5 的时候，图像深度需为 CV_8U、CV_16U、CV_32F 其中之一，而对于较大孔径尺寸的图片，它只能是 CV_8U。
- 第二个参数：OutputArray 类型的 dst，即目标图像，函数的输出参数，需要和源图片有一样的尺寸和类型。我们可以用 Mat::Clone，以源图片为模板，来初始化得到如假包换的目标图。
- 第三个参数：int 类型的 ksize，孔径的线性尺寸（aperture linear size），注意这个参数必须是大于 1 的奇数，比如：3、5、7、9……

调用范例如下。

```
//载入原图
Mat image=imread("1.jpg");
//进行中值滤波操作
Mat out;
medianBlur( image, out, 7);
```

用上面三句核心代码架起来的完整程序代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include"opencv2/highgui/highgui.hpp"
#include"opencv2/imgproc/imgproc.hpp"
using namespace cv;

//-----------------------------【main()函数】--------------------------
//     描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
      //载入原图
```

```
    Mat image=imread("1.jpg");

    //创建窗口
    namedWindow("中值滤波【原图】" );
    namedWindow("中值滤波【效果图】");

    //显示原图
    imshow("中值滤波【原图】", image );

    //进行中值滤波操作
    Mat out;
    medianBlur( image, out, 7);

    //显示效果图
    imshow("中值滤波【效果图】" ,out );

    waitKey(0 );
}
```

运行效果图（孔径的线性尺寸为 7）如图 6.11 所示。

图 6.11 运行效果图（孔径线性尺寸为 7）

2．双边滤波：bilateralFilter 函数

此函数的作用是用双边滤波器来模糊处理一张图片，由 src 输入图片，结果于 dst 输出。函数原型如下。

```
C++: void bilateralFilter(InputArray src, OutputArraydst, int d, double
```

```
sigmaColor, double sigmaSpace, int borderType=BORDER_DEFAULT)
```

参数详解如下。

- 第一个参数：InputArray 类型的 src，输入图像，即源图像，需要为 8 位或者浮点型单通道、三通道的图像。
- 第二个参数：OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。
- 第三个参数：int 类型的 d，表示在过滤过程中每个像素邻域的直径。如果这个值被设为非正数，那么 OpenCV 会从第五个参数 sigmaSpace 来计算出它。
- 第四个参数：double 类型的 sigmaColor，颜色空间滤波器的 sigma 值。这个参数的值越大，就表明该像素邻域内有越宽广的颜色会被混合到一起，产生较大的半相等颜色区域。
- 第五个参数：double 类型的 sigmaSpace，坐标空间中滤波器的 sigma 值，坐标空间的标注方差。它的数值越大，意味着越远的像素会相互影响，从而使更大的区域中足够相似的颜色获取相同的颜色。当 d>0 时，d 指定了邻域大小且与 sigmaSpace 无关。否则，d 正比于 sigmaSpace。
- 第六个参数：int 类型的 borderType，用于推断图像外部像素的某种边界模式。注意它有默认值 BORDER_DEFAULT。

调用代码示范如下。

```
//载入原图
Mat image=imread("1.jpg");
//进行双边滤波操作
Mat out;
bilateralFilter( image, out, 25, 25*2, 25/2 );
```

用一个完整的示例程序来说明 bilateralFilter 函数的用法，具体如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include"opencv2/highgui/highgui.hpp"
#include"opencv2/imgproc/imgproc.hpp"
using namespace cv;

//---------------------------【main()函数】---------------------------
//     描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
      //载入原图
      Mat image=imread("1.jpg");

      //创建窗口
      namedWindow("双边滤波【原图】" );
```

```
    namedWindow("双边滤波【效果图】");

    //显示原图
    imshow("双边滤波【原图】", image );

    //进行双边滤波操作
    Mat out;
    bilateralFilter( image, out, 25, 25*2, 25/2 );

    //显示效果图
    imshow("双边滤波【效果图】" ,out );

    waitKey(0);
}
```

运行效果图如图 6.12 所示。

图 6.12 双边滤波运行效果图

6.2.5 OpenCV 中的 5 种图像滤波综合示例

这一小节会将前文介绍的知识点以代码为载体，展现给大家。

在下面这个示例程序中，可以用滑动条来控制我们学习到的各种滤波（方框滤波、均值滤波、高斯滤波、中值滤波、双边滤波）的参数值，通过滑动滚动条，就可以控制图像在各种平滑处理下的模糊度，有一定的可玩性。详细注释的完整代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
```

```
//          描述：包含程序所依赖的头文件和命名空间包含
//-----------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include <iostream>
using namespace std;
using namespace cv;


//-------------------------【全局变量声明部分】-------------------------
//          描述：全局变量声明
//-----------------------------------------------------------------------
Mat
g_srcImage,g_dstImage1,g_dstImage2,g_dstImage3,g_dstImage4,g_dstImag
e5;
int g_nBoxFilterValue=6;  //方框滤波内核值
int g_nMeanBlurValue=10;  //均值滤波内核值
int g_nGaussianBlurValue=6;  //高斯滤波内核值
int g_nMedianBlurValue=10;  //中值滤波参数值
int g_nBilateralFilterValue=10;  //双边滤波参数值


//-------------------------【全局函数声明部分】-------------------------
//          描述：全局函数声明
//-----------------------------------------------------------------------
//轨迹条回调函数
static void on_BoxFilter(int, void *);           //方框滤波
static void on_MeanBlur(int, void *);           //均值块滤波器
static void on_GaussianBlur(int, void *);        //高斯滤波器
static void on_MedianBlur(int, void *);         //中值滤波器
static void on_BilateralFilter(int, void*);        //双边滤波器


//-------------------------【main()函数】-------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main( )
{
      system("color 5E");

      //载入原图
      g_srcImage= imread( "1.jpg", 1 );
      if(!g_srcImage.data ) { printf("读取srcImage错误~！ \n"); return
false; }

      //复制原图到4个Mat类型中
      g_dstImage1= g_srcImage.clone( );
      g_dstImage2= g_srcImage.clone( );
      g_dstImage3= g_srcImage.clone( );
      g_dstImage4= g_srcImage.clone( );
```

```
    g_dstImage5= g_srcImage.clone( );

    //显示原图
    namedWindow("【<0>原图窗口】", 1);
    imshow("【<0>原图窗口】",g_srcImage);


    //=================【<1>方框滤波】=========================
    //创建窗口
    namedWindow("【<1>方框滤波】", 1);
    //创建轨迹条
    createTrackbar("内核值：", "【<1>方框滤波】",&g_nBoxFilterValue,
50,on_BoxFilter );
    on_MeanBlur(g_nBoxFilterValue,0);
    imshow("【<1>方框滤波】", g_dstImage1);
    //=====================================================


    //=================【<2>均值滤波】=========================
    //创建窗口
    namedWindow("【<2>均值滤波】", 1);
    //创建轨迹条
    createTrackbar("内核值：", "【<2>均值滤波】",&g_nMeanBlurValue,
50,on_MeanBlur );
    on_MeanBlur(g_nMeanBlurValue,0);
    //=====================================================


    //=================【<3>高斯滤波】=========================
    //创建窗口
    namedWindow("【<3>高斯滤波】", 1);
    //创建轨迹条
    createTrackbar("内核值：", "【<3>高斯滤波】",&g_nGaussianBlurValue,
50,on_GaussianBlur );
    on_GaussianBlur(g_nGaussianBlurValue,0);
    //=====================================================


    //=================【<4>中值滤波】=========================
    //创建窗口
    namedWindow("【<4>中值滤波】", 1);
    //创建轨迹条
    createTrackbar("参数值：", "【<4>中值滤波】",&g_nMedianBlurValue,
50,on_MedianBlur );
    on_MedianBlur(g_nMedianBlurValue,0);
    //=====================================================


    //=================【<5>双边滤波】=========================
    //创建窗口
    namedWindow("【<5>双边滤波】", 1);
    //创建轨迹条
```

```
        createTrackbar("参数值: ", "【<5>双边滤波】
",&g_nBilateralFilterValue, 50,on_BilateralFilter);
        on_BilateralFilter(g_nBilateralFilterValue,0);
        //=======================================================
        while(char(waitKey(1))!= 'q') {}

        return 0;
}

//-----------------------【on_BoxFilter()函数】-----------------------
//          描述: 方框滤波操作的回调函数
//--------------------------------------------------------------------
static void on_BoxFilter(int, void *)
{
        //方框滤波操作
        boxFilter(g_srcImage, g_dstImage1, -1,Size( g_nBoxFilterValue+1,
g_nBoxFilterValue+1));
        //显示窗口
        imshow("【<1>方框滤波】", g_dstImage1);
}

//-----------------------【on_MeanBlur()函数】-----------------------
//          描述: 均值滤波操作的回调函数
//--------------------------------------------------------------------
static void on_MeanBlur(int, void *)
{
        blur(g_srcImage, g_dstImage2, Size( g_nMeanBlurValue+1,
g_nMeanBlurValue+1),Point(-1,-1));
        imshow("【<2>均值滤波】", g_dstImage2);

}

//-----------------------【on_GaussianBlur()函数】-----------------------
//          描述: 高斯滤波操作的回调函数
//--------------------------------------------------------------------
static void on_GaussianBlur(int, void *)
{
        GaussianBlur(g_srcImage, g_dstImage3,
Size( g_nGaussianBlurValue*2+1,g_nGaussianBlurValue*2+1 ), 0, 0);
        imshow("【<3>高斯滤波】", g_dstImage3);
}


//-----------------------【on_MedianBlur()函数】-----------------------
//          描述: 中值滤波操作的回调函数
//--------------------------------------------------------------------
static void on_MedianBlur(int, void *)
{
        medianBlur( g_srcImage, g_dstImage4, g_nMedianBlurValue*2+1 );
        imshow("【<4>中值滤波】", g_dstImage4);
}
```

```
//--------------------【on_BilateralFilter()函数】--------------------
//          描述：双边滤波操作的回调函数
//-----------------------------------------------------------------
static void on_BilateralFilter(int, void *)
{
      bilateralFilter( g_srcImage, g_dstImage5,
g_nBilateralFilterValue,
g_nBilateralFilterValue*2,g_nBilateralFilterValue/2 );
      imshow("【<5>双边滤波】", g_dstImage5);
}
```

下面是程序运行的原图和效果图示例，如图 6.13 至 6.18 所示。

图 6.13 原始图

图 6.14 方框滤波效果图

图 6.15 均值滤波效果图

图 6.16 高斯滤波效果图

图 6.17 中值滤波效果图

图 6.18 双边滤波效果图

观察各种截图可以发现，方框滤波和均值滤波效果很相似，中值滤波对原图颠覆很大，而双边滤波和原图差别不大，要仔细看才看得出效果。

6.3 形态学滤波（1）：腐蚀与膨胀

本小节中，我们将一起探究图像处理中最基本的形态学运算——膨胀与腐蚀。

6.3.1 形态学概述

形态学（morphology）一词通常表示生物学的一个分支，该分支主要研究动植物的形态和结构。而我们图像处理中的形态学，往往指的是数学形态学。下面一起来了解数学形态学的概念。

数学形态学（Mathematical morphology）是一门建立在格论和拓扑学基础之上的图像分析学科，是数学形态学图像处理的基本理论。其基本的运算包括：二值腐蚀和膨胀、二值开闭运算、骨架抽取、极限腐蚀、击中击不中变换、形态学梯度、Top-hat 变换、颗粒分析、流域变换、灰值腐蚀和膨胀、灰值开闭运算、灰值形态学梯度等。

简单来讲，形态学操作就是基于形状的一系列图像处理操作。OpenCV 为进行图像的形态学变换提供了快捷、方便的函数。最基本的形态学操作有两种，分别是：膨胀（dilate）与腐蚀（erode）。

膨胀与腐蚀能实现多种多样的功能，主要如下。

- 消除噪声；
- 分割（isolate）出独立的图像元素，在图像中连接（join）相邻的元素；
- 寻找图像中的明显的极大值区域或极小值区域；
- 求出图像的梯度。

在这里给出下文会用于对比膨胀与腐蚀运算的“浅墨”字样毛笔字原图，如

图 6.19 所示。

图 6.19 “浅墨”字样毛笔字原图

在进行腐蚀和膨胀的讲解之前，首先提醒大家注意，腐蚀和膨胀是对白色部分（高亮部分）而言的，不是黑色部分。膨胀是图像中的高亮部分进行膨胀，类似于“领域扩张”，效果图拥有比原图更大的高亮区域；腐蚀是原图中的高亮部分被腐蚀，类似于“领域被蚕食”，效果图拥有比原图更小的高亮区域。

6.3.2 膨胀

膨胀（dilate）就是求局部最大值的操作。从数学角度来说，膨胀或者腐蚀操作就是将图像（或图像的一部分区域，称之为 A）与核（称之为 B）进行卷积。

核可以是任何形状和大小，它拥有一个单独定义出来的参考点，我们称其为锚点（anchorpoint）。多数情况下，核是一个小的，中间带有参考点和实心正方形或者圆盘。其实，可以把核视为模板或者掩码。

而膨胀就是求局部最大值的操作。核 B 与图形卷积，即计算核 B 覆盖的区域的像素点的最大值，并把这个最大值赋值给参考点指定的像素。这样就会使图像中的高亮区域逐渐增长，如图 6.20 所示。这就是膨胀操作的初衷。

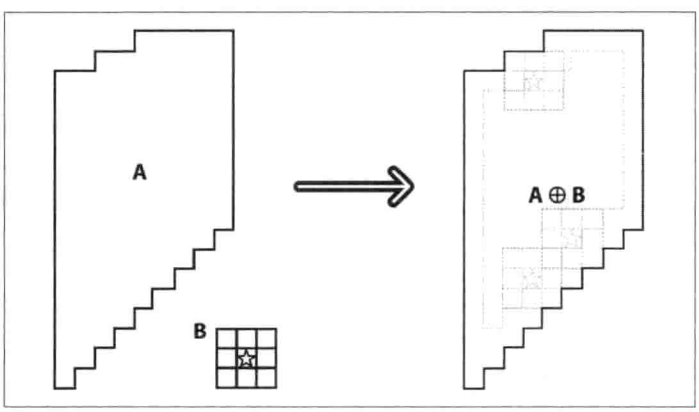


图 6.20 膨胀操作示例图

膨胀的数学表达式如下。

$$\mathrm{dst}(x,y)=\max_{(x',y'):\ \mathrm{element}(x',y')\neq 0}\mathrm{src}(x+x',y+y')$$

图 6.21 和图 6.22 分别是毛笔字和照片膨胀的效果示例。

图 6.21 膨胀效果图（毛笔字）

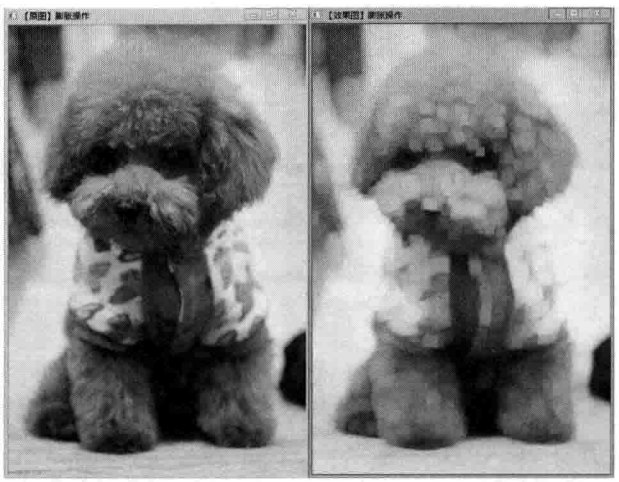


图 6.22 膨胀效果图（照片）

6.3.3 腐蚀

大家应该知道，膨胀和腐蚀（erode）是相反的一对操作，所以腐蚀就是求局部最小值的操作。

我们一般都会把腐蚀和膨胀进行对比理解和学习。下文就可以看到，两者的函数原型也是基本一样的。腐蚀操作示例如图 6.23 所示。

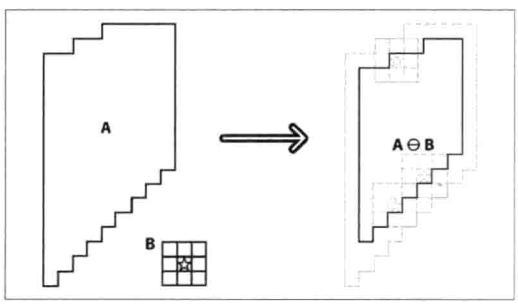


图 6.23 腐蚀操作示例图

腐蚀的数学表达式如下。

$$\mathrm{dst}(x,y)=\min_{(x',y'):\ \mathrm{element}(x',y')\neq 0}\mathrm{src}(x+x',y+y')$$

图 6.24、图 6.25 分别给出了毛笔字和照片的腐蚀效果示例。

图 6.24 腐蚀效果图（毛笔字）

图 6.25 腐蚀效果图（照片）

6.3.4 相关 OpenCV 源码分析溯源

在…\opencv\sources\modules\imgproc\src\ morph.cpp 路径中，我们可以发现 erode（腐蚀）函数和 dilate（膨胀）函数的源码，如下。

```
void cv::erode( InputArray src, OutputArraydst, InputArray kernel,
                Point anchor, int iterations,
                int borderType, constScalar& borderValue )
{
//调用 morphOp 函数，并设定标识符为 MORPH_ERODE
   morphOp( MORPH_ERODE, src, dst, kernel, anchor, iterations,
borderType,borderValue );
}

void cv::dilate( InputArray src,OutputArray dst, InputArray kernel,
                 Point anchor, int iterations,
                 int borderType, constScalar& borderValue )
{
//调用 morphOp 函数，并设定标识符为 MORPH_DILATE
   morphOp( MORPH_DILATE, src, dst, kernel, anchor, iterations,
borderType,borderValue );
}
```

可以发现，erode 和 dilate 这两个函数内部就是调用了一下 morphOp，只是它

们调用 morphOp 时，第一个参数标识符不同：一个为 MORPH_ERODE（腐蚀），一个为 MORPH_DILATE（膨胀）。

morphOp 函数的源码在...\opencv\sources\modules\imgproc\src\morph.cpp 中的第 1286 行，有兴趣的朋友们可以自行研究，这里就不再展开分析了。

6.3.5 相关核心 API 函数讲解

1. 膨胀：dilate 函数

dilate 函数使用像素邻域内的局部极大运算符来膨胀一张图片，从 src 输入，由 dst 输出。支持就地（in-place）操作。

函数原型如下。

```
C++: void dilate(
    InputArray src,
    OutputArray dst,
    InputArray kernel,
    Point anchor=Point(-1,-1),
    int iterations=1,
    int borderType=BORDER_CONSTANT,
    const Scalar& borderValue=morphologyDefaultBorderValue()
);
```

参数详解如下。

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。图像通道的数量可以是任意的，但图像深度应为 CV_8U、CV_16U、CV_16S、CV_32F 或 CV_64F 其中之一。
- 第二个参数，OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。
- 第三个参数，InputArray 类型的 kernel，膨胀操作的核。当为 NULL 时，表示的是使用参考点位于中心 3×3 的核。

我们一般使用函数 getStructuringElement 配合这个参数的使用。getStructuringElement 函数会返回指定形状和尺寸的结构元素（内核矩阵）。其中，getStructuringElement 函数的第一个参数表示内核的形状，有如下三种形状可以选择。

- 矩形: MORPH_RECT;
- 交叉形: MORPH_CROSS;
- 椭圆形: MORPH_ELLIPSE。

而 getStructuringElement 函数的第二和第三个参数分别是内核的尺寸以及锚点的位置。

一般在调用 erode 以及 dilate 函数之前，先定义一个 Mat 类型的变量来获得 getStructuringElement 函数的返回值。对于锚点的位置，有默认值 Point(-1,-1)，表

示锚点位于中心。此外，需要注意，十字形的 element 形状唯一依赖于锚点的位置，而在其他情况下，锚点只是影响了形态学运算结果的偏移。

getStructuringElement 函数相关的调用示例代码如下。

```
int g_nStructElementSize = 3; //结构元素(内核矩阵)的尺寸

//获取自定义核
Mat element = getStructuringElement(MORPH_RECT,
    Size(2*g_nStructElementSize+1,2*g_nStructElementSize+1),
    Point( g_nStructElementSize, g_nStructElementSize ));
```

调用之后，我们可以在接下来调用 erode 或 dilate 函数时，在第三个参数填保存了 getStructuringElement 返回值的 Mat 类型变量。对应于上面的示例，就是 element 变量。

- 第四个参数，Point 类型的 anchor，锚的位置，其有默认值(-1,-1)，表示锚位于中心。
- 第五个参数，int 类型的 iterations，迭代使用 erode()函数的次数，默认值为 1。
- 第六个参数，int 类型的 borderType，用于推断图像外部像素的某种边界模式。注意它有默认值 BORDER_DEFAULT。
- 第七个参数，const Scalar&类型的 borderValue，当边界为常数时的边界值，有默认值 morphologyDefaultBorderValue()，一般不用去管它。需要用到它时，可以看官方文档中的 createMorphologyFilter()函数，以得到更详细的解释。

使用 erode 函数，一般只需要填前面的三个参数，后面的四个参数都有默认值，而且往往会结合 getStructuringElement 一起使用。

下面给出一个调用范例。

```
    //载入原图
Mat image = imread("1.jpg");
//获取自定义核
    Mat element = getStructuringElement(MORPH_RECT, Size(15, 15));
    Mat out;
    //进行膨胀操作
    dilate(image, out, element);
```

用上面核心代码架起来的完整程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include <iostream>

//-------------------------【命名空间声明部分】-------------------------
```

```
//     描述：包含程序所使用的命名空间
//-----------------------------------------------------------------------
using namespace std;
using namespace cv;

//-----------------------------【main()函数】--------------------------
//     描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main( )
{

      //载入原图
      Mat image = imread("1.jpg");

      //创建窗口
      namedWindow("【原图】膨胀操作");
      namedWindow("【效果图】膨胀操作");

      //显示原图
      imshow("【原图】膨胀操作", image);

//获取自定义核
      Mat element = getStructuringElement(MORPH_RECT, Size(15, 15));
      Mat out;
//进行膨胀操作
      dilate(image,out, element);

      //显示效果图
      imshow("【效果图】膨胀操作", out);

      waitKey(0);

      return 0;
}
```

此程序运行截图见前文中图 6.22。

2. 腐蚀：erode 函数

erode 函数使用像素邻域内的局部极小运算符来腐蚀一张图片，从 src 输入，由 dst 输出。支持就地（in-place）操作。

看一下函数原型，如下。

```
C++: void erode(
    InputArray src,
    OutputArray dst,
    InputArray kernel,
    Point anchor=Point(-1,-1),
    int iterations=1,
    int borderType=BORDER_CONSTANT,
    const Scalar& borderValue=morphologyDefaultBorderValue()
);
```

参数详解如下。

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。图像通道的数量可以是任意的，但图像深度应为 CV_8U、CV_16U、CV_16S、CV_32F 或 CV_64F 其中之一。
- 第二个参数，OutputArray 类型的 dst，即目标图像，需要和源图片有一样的尺寸和类型。
- 第三个参数，InputArray 类型的 kernel，腐蚀操作的内核。为 NULL 时，表示的是使用参考点位于中心 3x3 的核。一般使用函数 getStructuringElement 配合这个参数的使用。getStructuringElement 函数会返回指定形状和尺寸的结构元素（内核矩阵，具体看上文中 dilate 函数的第三个参数讲解部分。
- 第四个参数，Point 类型的 anchor，锚的位置。其有默认值(-1,-1)，表示锚位于单位（element）的中心，一般不用管它。
- 第五个参数，int 类型的 iterations，迭代使用 erode()函数的次数，默认值为 1。
- 第六个参数，int 类型的 borderType，用于推断图像外部像素的某种边界模式。注意它有默认值 BORDER_DEFAULT。
- 第七个参数，const Scalar&类型的 borderValue，当边界为常数时的边界值，有默认值 morphologyDefaultBorderValue()，一般不用去管它。需要用到它时，可以看官方文档中的 createMorphologyFilter()函数以得到更详细的解释。

同样的，使用 erode 函数，一般只需要填前面的三个参数，后面的四个参数都有默认值。而且往往结合 getStructuringElement 一起使用。

调用范例如下。

```
//载入原图
Mat image = imread("1.jpg");
//获取自定义核
Mat element = getStructuringElement(MORPH_RECT, Size(15, 15));
Mat out;
//进行腐蚀操作
erode(image,out, element);
```

用上面核心代码架起来的完整程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//     描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/core/core.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include <iostream>
using namespace std;
using namespace cv;
```

```
//-----------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
      //载入原图
      Matimage = imread("1.jpg");

       //创建窗口
      namedWindow("【原图】腐蚀操作");
      namedWindow("【效果图】腐蚀操作");

      //显示原图
      imshow("【原图】腐蚀操作", image);


//获取自定义核
      Mat element = getStructuringElement(MORPH_RECT, Size(15, 15));
      Mat out;

//进行腐蚀操作
      erode(image,out, element);

      //显示效果图
      imshow("【效果图】腐蚀操作", out);

      waitKey(0);

      return 0;
}
```

此程序运行截图见前文中图 6.25。

6.3.6 综合示例：腐蚀与膨胀

此示例程序中的效果图窗口中有两个滑动条第一个滑动条“腐蚀/膨胀”用于在腐蚀/膨胀之间进行切换；第二个滚动条”内核尺寸”用于调节形态学操作时的内核尺寸，以得到效果不同的图像，有一定的可玩性。详细注释的示例程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
#include <iostream>
using namespace std;
using namespace cv;
```

```
//-------------------------【全局变量声明部分】-------------------------
//          描述：全局变量声明
//-----------------------------------------------------------------------
Mat g_srcImage, g_dstImage;//原始图和效果图
int g_nTrackbarNumer = 0;//0表示腐蚀erode, 1表示膨胀dilate
int g_nStructElementSize = 3; //结构元素(内核矩阵)的尺寸


//-------------------------【全局函数声明部分】-------------------------
//          描述：全局函数声明
//-----------------------------------------------------------------------
void Process();//膨胀和腐蚀的处理函数
void on_TrackbarNumChange(int, void *);//回调函数
void on_ElementSizeChange(int, void *);//回调函数


//-------------------------【main()函数】-------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
      //改变console字体颜色
      system("color5E");

      //载入原图
      g_srcImage= imread("1.jpg");
      if(!g_srcImage.data ) { printf("读取srcImage错误~! \n"); return
false; }

      //显示原始图
      namedWindow("【原始图】");
      imshow("【原始图】", g_srcImage);

      //进行初次腐蚀操作并显示效果图
      namedWindow("【效果图】");
      //获取自定义核
      Matelement = getStructuringElement(MORPH_RECT,
Size(2*g_nStructElementSize+1,2*g_nStructElementSize+1),Point( g_nSt
ructElementSize, g_nStructElementSize ));
      erode(g_srcImage,g_dstImage, element);
      imshow("【效果图】", g_dstImage);

      //创建轨迹条
      createTrackbar("腐蚀/膨胀", "【效果图】", &g_nTrackbarNumer, 1,
on_TrackbarNumChange);
      createTrackbar("内核尺寸", "【效果图】",&g_nStructElementSize, 21,
on_ElementSizeChange);


      //轮询获取按键信息，若下q键，程序退出
      while(char(waitKey(1))!= 'q') {}
```

```
    return 0;
}

//-------------------------【Process()函数】-------------------------
//          描述：进行自定义的腐蚀和膨胀操作
//-------------------------------------------------------------------
void Process()
{
    //获取自定义核
    Mat element = getStructuringElement(MORPH_RECT,
Size(2*g_nStructElementSize+1,2*g_nStructElementSize+1),Point( g_nSt
ructElementSize, g_nStructElementSize ));

    //进行腐蚀或膨胀操作
    if(g_nTrackbarNumer== 0) {
        erode(g_srcImage,g_dstImage, element);
    }
    else{
        dilate(g_srcImage,g_dstImage, element);
    }

    //显示效果图
    imshow("【效果图】", g_dstImage);
}


//-------------------【on_TrackbarNumChange()函数】-------------------
//          描述：腐蚀和膨胀之间切换开关的回调函数
//-------------------------------------------------------------------
void on_TrackbarNumChange(int, void *)
{
    //腐蚀和膨胀之间效果已经切换，回调函数体内需调用一次Process函数，使改变
后的效果立即生效并显示出来
    Process();
}


//-------------------【on_ElementSizeChange()函数】-------------------
//          描述：腐蚀和膨胀操作内核改变时的回调函数
//-------------------------------------------------------------------
void on_ElementSizeChange(int, void *)
{
    //内核尺寸已改变，回调函数体内需调用一次Process函数，使改变后的效果立即
生效并显示出来
    Process();
}
```

运行截图分别如图 6.26、图 6.27、图 6.28 所示。

图 6.26 原始图

图 6.27 膨胀效果图

图 6.28 腐蚀效果图

由于篇幅所限，截图就先展示这些，更多的运行效果请大家自行运行书本配套示例程序进行学习和观赏。

6.4 形态学滤波（2）：开运算、闭运算、形态学梯度、顶帽、黑帽

上一节中，我们重点了解了腐蚀和膨胀这两种最基本的形态学操作，而运用这两个基本操作，可以实现更高级的形态学变换。

所以，本节的主角是 OpenCV 中的 morphologyEx 函数，它利用基本的膨胀和腐蚀技术，来执行更加高级的形态学变换，如开闭运算、形态学梯度、“顶帽”、“黑帽”等。

首先，我们需要知道，形态学的高级形态，往往都是建立在腐蚀和膨胀这两个基本操作之上的。而关于腐蚀和膨胀，概念和细节以及相关代码请参考上一小节。对膨胀和腐蚀心中有数了，接下来的高级形态学操作，应该就不难理解。

6.4.1 开运算

开运算（Opening Operation），其实就是先腐蚀后膨胀的过程。其数学表达式如下：

```
dst=open(src,element)=dilate(erode(src,element))
```

开运算可以用来消除小物体，在纤细点处分离物体，并且在平滑较大物体的边界的同时不明显改变其面积。原始图请参考图 6.19，而效果图见图 6.30 和 6.31。

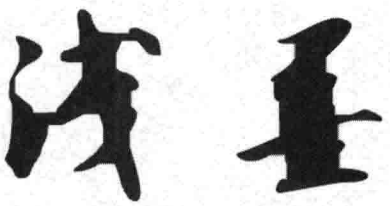


图 6.30 开运算效果图

图 6.31 图片开运算效果图

6.4.2 闭运算

先膨胀后腐蚀的过程称为闭运算（Closing Operation），其数学表达式如下：

```
dst=clese(src,element)= erode(dilate(src,element))
```

闭运算能够排除小型黑洞（黑色区域）。效果图如图 6.32 和图 6.33 所示。

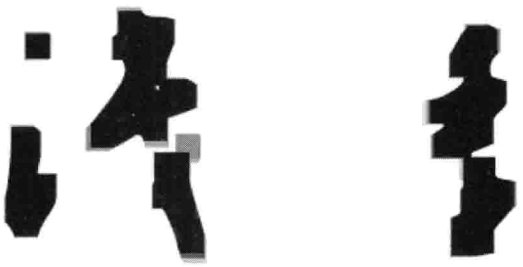


图 6.32 闭运算效果图

图 6.33 图片闭运算效果图

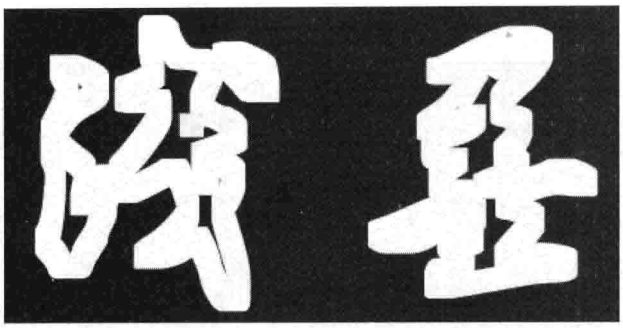
6.4.3 形态学梯度

形态学梯度（Morphological Gradient）是膨胀图与腐蚀图之差，数学表达式如下：

```
dst=morph-grad(src,element)= dilate(src,element)- erode(src,element)
```

对二值图像进行这一操作可以将团块（blob）的边缘突出出来。我们可以用

形态学梯度来保留物体的边缘轮廓，如图 6.34 和图 6.35 所示。



6.34 形态学梯度效果图

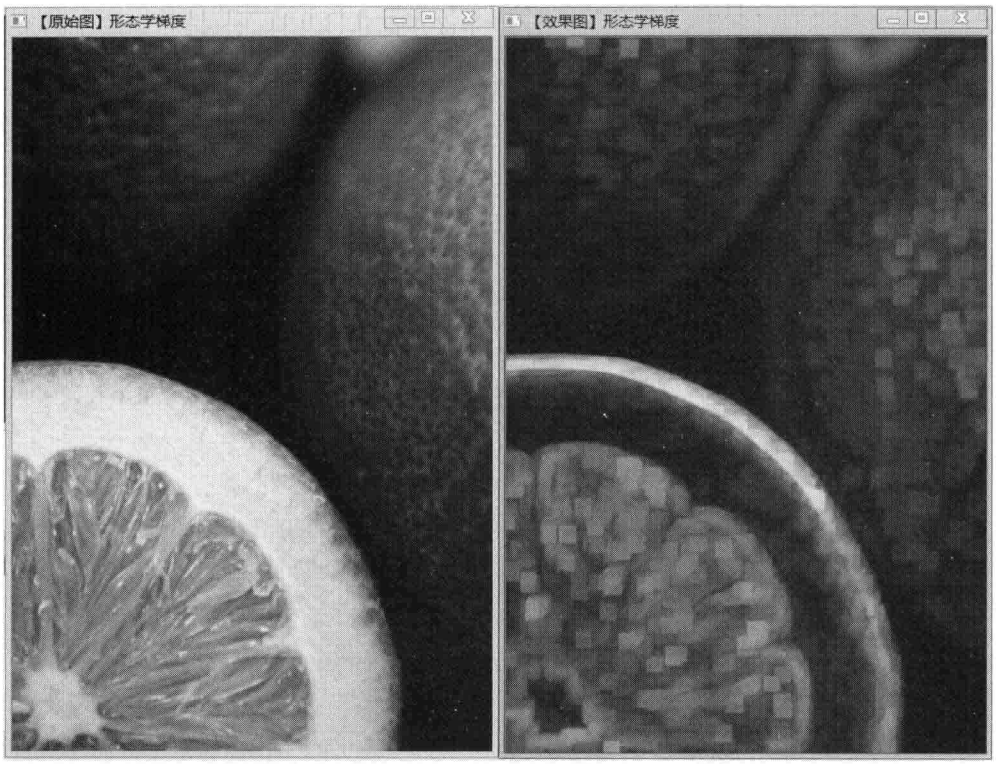


图 6.35 图片形态学梯度效果图

6.4.4 顶帽

顶帽运算（Top Hat）又常常被译为”礼帽“运算，是原图像与上文刚刚介绍的“开运算”的结果图之差，数学表达式如下：

```
dst=tophat(src,element)=src-open(src,element)
```

因为开运算带来的结果是放大了裂缝或者局部低亮度的区域。因此，从原图中减去开运算后的图，得到的效果图突出了比原图轮廓周围的区域更明亮的区域，且这一操作与选择的核的大小相关。

顶帽运算往往用来分离比邻近点亮一些的斑块。在一幅图像具有大幅的背景，

而微小物品比较有规律的情况下，可以使用顶帽运算进行背景提取。

如图 6.36、图 6.37 所示。

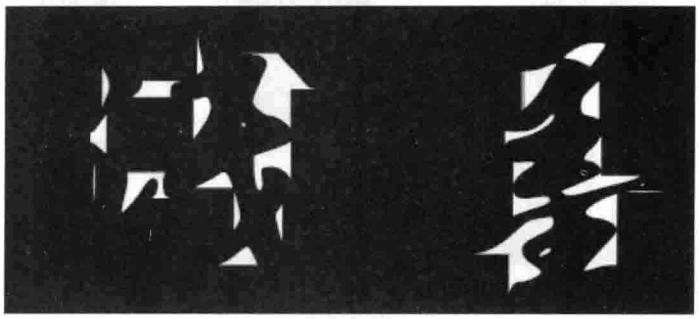


图 6.36 顶帽效果图

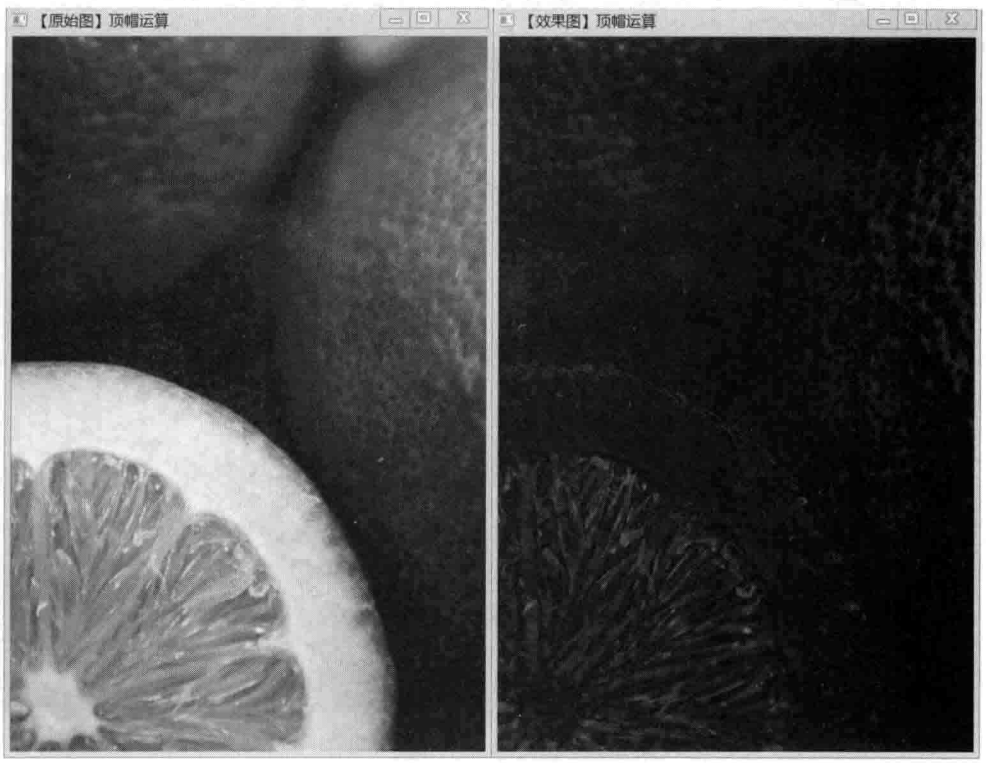


图 6.37 图片顶帽效果图

6.4.5 黑帽

黑帽（Black Hat）运算是闭运算的结果图与原图像之差。数学表达式为：

```
dst=blackhat (src,element)=close(src,element)- src
```

黑帽运算后的效果图突出了比原图轮廓周围的区域更暗的区域，且这一操作和选择的核的大小相关。

所以，黑帽运算用来分离比邻近点暗一些的斑块，效果图有着非常完美的轮廓。示例如图 6.38 和图 6.39 所示。

图 6.38 黑帽效果图

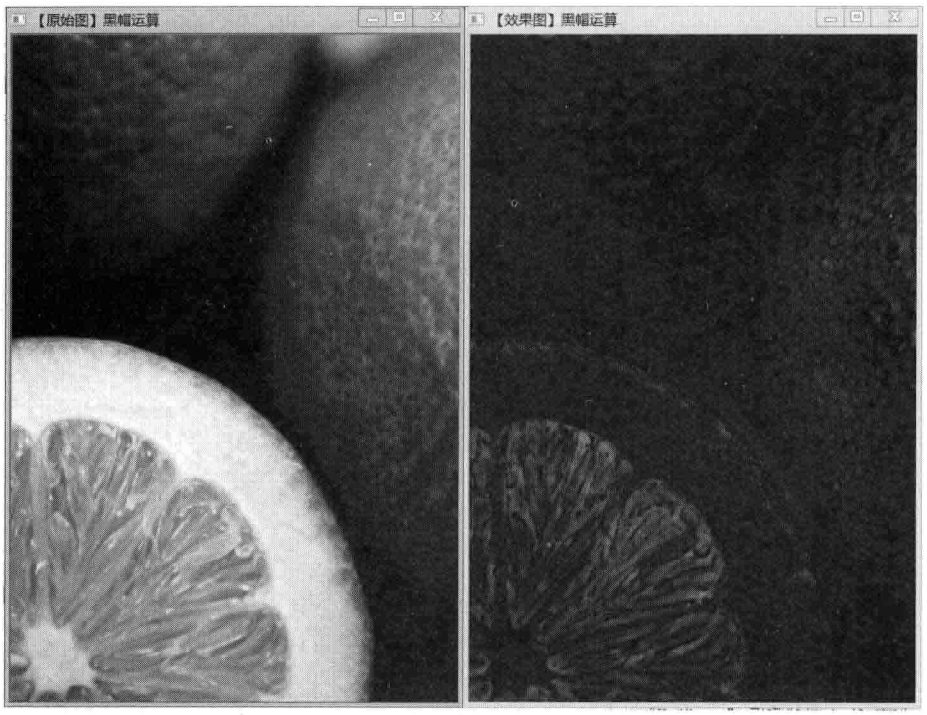


图 6.39 照片黑帽效果图

6.4.6 形态学滤波 OpenCV 源码分析溯源

本节的主角是 OpenCV 中的 morphologyEx 函数，它利用基本的膨胀和腐蚀技术，来执行更加高级的形态学变换，如开闭运算、形态学梯度、“顶帽”、“黑帽”等。这一节我们来一起看一下 morphologyEx 函数的源代码。

```
void cv::morphologyEx( InputArray _src,OutputArray _dst, int op,
                       InputArray kernel, Pointanchor, int iterations,
                       int borderType, constScalar& borderValue )
{
//复制 Mat 数据到临时变量
   Mat src = _src.getMat(), temp;
   _dst.create(src.size(), src.type());
   Mat dst = _dst.getMat();

//一个大 switch，根据不同的标识符取不同的操作
   switch( op )
```

```
    {
   case MORPH_ERODE:
      erode( src, dst, kernel, anchor, iterations, borderType,
borderValue );
      break;
   case MORPH_DILATE:
      dilate( src, dst, kernel, anchor, iterations, borderType,
borderValue );
      break;
   case MORPH_OPEN:
      erode( src, dst, kernel, anchor, iterations, borderType,
borderValue );
      dilate( dst, dst, kernel, anchor, iterations, borderType,
borderValue );
      break;
   case CV_MOP_CLOSE:
      dilate( src, dst, kernel, anchor, iterations, borderType,
borderValue );
      erode( dst, dst, kernel, anchor, iterations, borderType,
borderValue );
      break;
   case CV_MOP_GRADIENT:
      erode( src, temp, kernel, anchor, iterations, borderType,
borderValue );
      dilate( src, dst, kernel, anchor, iterations, borderType,
borderValue );
      dst -= temp;
      break;
   case CV_MOP_TOPHAT:
      if( src.data != dst.data )
         temp = dst;
      erode( src, temp, kernel, anchor, iterations, borderType,
borderValue );
       dilate( temp, temp, kernel, anchor,iterations, borderType,
borderValue );
      dst = src - temp;
      break;
   case CV_MOP_BLACKHAT:
      if( src.data != dst.data )
         temp = dst;
      dilate( src, temp, kernel, anchor, iterations, borderType,
borderValue);
      erode( temp, temp, kernel, anchor, iterations, borderType,
borderValue);
      dst = temp - src;
      break;
   default:
      CV_Error( CV_StsBadArg, "unknown morphological operation" );
    }
}
```

看上面的源码可以发现，morphologyEx 函数其实就是内部的一个大 switch 而

已，根据不同的标识符取不同的操作。比如开运算 MORPH_OPEN，按我们上文中讲解的数学表达式，就是先腐蚀后膨胀，即依次调用 erode 和 dilate 函数，代码非常简明干净。

6.4.7 核心 API 函数：morphologyEx()

上面已经讲到，morphologyEx 函数利用基本的膨胀和腐蚀技术，来执行更加高级形态学变换，如开闭运算、形态学梯度、“顶帽”、“黑帽”等。下面我们来详细地讲解它的参数意义和使用方法。

```
C++: void morphologyEx(
InputArray src,
OutputArray dst,
int op,
InputArraykernel,
Pointanchor=Point(-1,-1),
intiterations=1,
intborderType=BORDER_CONSTANT,
constScalar& borderValue=morphologyDefaultBorderValue() );
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。图像位深应该为以下 5 种之一：CV_8U、CV_16U、CV_16S、CV_32F 和 CV_64F。
- 第二个参数，OutputArray 类型的 dst，即目标图像，函数的输出参数，需要和源图片有一样的尺寸和类型。
- 第三个参数，int 类型的 op，表示形态学运算的类型，可以是如表 6.2 中任意之一的标识符。

表 6.2 morphologyEx 函数可取标识符列举

标识符	含义
MORPH_OPEN	开运算
MORPH_CLOSE	闭运算
MORPH_GRADIENT	形态学梯度
MORPH_TOPHAT	顶帽
MORPH_BLACKHAT	黑帽
MORPH_ERODE	腐蚀
MORPH_DILATE	膨胀

对于 OpenCV2，另有 CV 版本的标识符也可选择，如 CV_MOP_CLOSE、CV_MOP_GRADIENT、CV_MOP_TOPHAT、CV_MOP_BLACKHAT，这是 OpenCV1.0 系列版本遗留下来的标识符，在 OpenCV2 中和上面的“MORPH_OPEN”具有一样的效果，但已经在 OpenCV3 中废止。

- 第四个参数，InputArray 类型的 kernel，形态学运算的内核。若为 NULL，

表示的是使用参考点位于中心3×3的核。一般使用函数 getStructuringElement 配合这个参数的使用。getStructuringElement 函数会返回指定形状和尺寸的结构元素（内核矩阵）。关于 getStructuringElement 我们之前有讲到过，这里为了大家参阅方便，再写一遍。

getStructuringElement 函数的第一个参数表示内核的形状，我们可以选择如下三种形状之一：

- 矩形——MORPH_RECT
- 交叉形——MORPH_CROSS
- 椭圆形——MORPH_ELLIPSE

而 getStructuringElement 函数的第二和第三个参数分别是内核的尺寸以及锚点的位置。

一般在调用 erode 以及 dilate 函数之前，要先定义一个 Mat 类型的变量来获得 getStructuringElement 函数的返回值。对于锚点的位置，有默认值 Point(-1,-1)，表示锚点位于中心。另外需要注意：十字形的 element 形状唯一依赖于锚点的位置。而在其他情况下，锚点只是影响形态学运算结果的偏移。

getStructuringElement 函数相关的调用示例代码如下。

```
int g_nStructElementSize = 3; //结构元素(内核矩阵)的尺寸

//获取自定义核
Mat element =getStructuringElement(MORPH_RECT,
      Size(2*g_nStructElementSize+1,2*g_nStructElementSize+1),
      Point(g_nStructElementSize, g_nStructElementSize ));
```

之后，便可以在调用 erode、dilate 或 morphologyEx 函数时，由 kernel 参数填保存 getStructuringElement 返回值的 Mat 类型变量。对应于上面的示例，就是填 element 变量。

- 第五个参数，Point 类型的 anchor，锚的位置，其有默认值(-1,-1)，表示锚位于中心。
- 第六个参数，int 类型的 iterations，迭代使用函数的次数，默认值为 1。
- 第七个参数，int 类型的 borderType，用于推断图像外部像素的某种边界模式。注意它有默认值 BORDER_ CONSTANT。
- 第八个参数，const Scalar&类型的 borderValue，当边界为常数时的边界值，有默认值 morphologyDefaultBorderValue()，一般不用去管它。需要用到它时，可以看官方文档中的 createMorphologyFilter()函数得到更详细的解释。其中的这些操作都可以进行就地（in-place）操作，且对于多通道图像，每一个通道都单独进行操作。

6.4.8 各形态学操作使用范例一览

核心函数讲解完毕，下面便开始讲解使用范例。为了方便大家需要的时候随

时取用，这里提供了利用 morphologyEx 函数实现的几乎全部的形态学操作：开运算、闭运算、形态学梯度、顶帽、黑帽、腐蚀、膨胀的效果实现简化版完整代码。其实说白了，这些代码基本上内容一致，就是改一下 morphologyEx 里面的第三个标识符参数而已。核都是选的 MORPH_RECT（矩形元素结构）。另外，通过观察源代码发现，最基本的腐蚀和膨胀操作也可以用 morphologyEx 函数来实现，它们由 morphologyEx 函数源码中 switch 的前两个 case 来实现（虽然在 case 体内就是简单地各自调用了一下 erode 和 dilation 函数，但还是有写出来的必要）。所以在这里，我们也用 morphologyEx 再重新来实现一遍它们。以下便为实现代码（以形态学梯度为例）。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
 using namespace cv;

//---------------------------【main()函数】---------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
      //载入原始图
      Mat image = imread("1.jpg");  //工程目录下应该有一张名为1.jpg的素材
图
      //创建窗口
      namedWindow("【原始图】");
      namedWindow("【效果图】");
      //显示原始图
      imshow("【原始图】", image);
      //定义核
      Mat element = getStructuringElement(MORPH_RECT, Size(15, 15));
      //进行形态学操作
      morphologyEx(image,image, MORPH_ GRADIENT , element);
      //显示效果图
      imshow("【效果图】", image);

      waitKey(0);

      return 0;
}
```

程序运行截图见图 6.35。

上述代码是以形态学梯度为例的程序，而若要选取其他形态学的操作，只需将 morphologyEx 函数中的第三个参数 MORPH_ GRADIENT 替换成表 6.2 中相应的标识符即可。但是方面大家方便查看和使用，在配套示例程序代码包中，还是为大家分别准备了开运算、闭运算、形态学梯度、顶帽、黑帽、腐蚀、膨胀的效

果的实现程序。具体程序在本书的配套代码包中的序号为 41 到 47，读者可以进行快速查阅。

6.4.9 综合示例：形态学滤波

这个综合示例程序中，一共会出现 4 个显示图像的窗口，包括原始图一个，开/闭运算为一个，腐蚀/膨胀为一个，顶帽/黑帽运算为一个。它们分别使用滚动条，来控制得到的形态学效果，且迭代值为 10 的时候为中间点。另外，还可以通过键盘按键 1、2、3 以及空格键来调节成不同的元素结构（矩形、椭圆、十字形）。说明页面如图 6.40 所示。

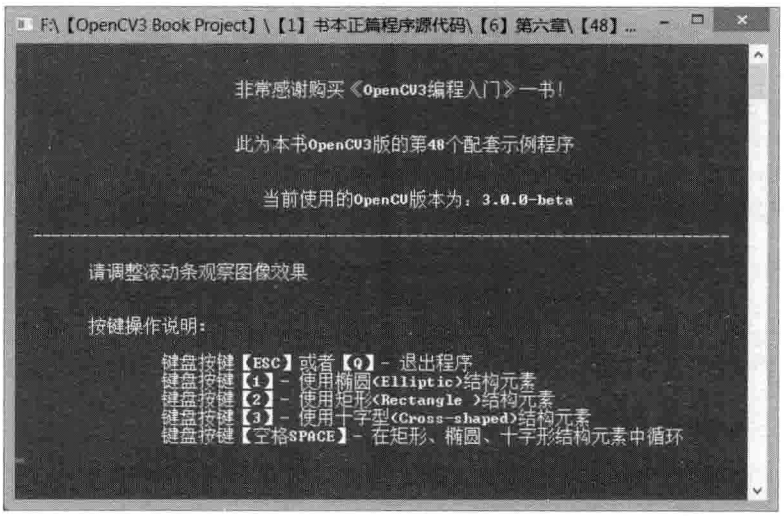


图 6.40 程序说明文字页面

详细注释的程序代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace std;
using namespace cv;


//-------------------------【全局变量声明部分】--------------------------
//      描述：全局变量声明
//-------------------------------------------------------------------------
Mat g_srcImage, g_dstImage;//原始图和效果图
int g_nElementShape = MORPH_RECT;//元素结构的形状

//变量接收的 TrackBar 位置参数
int g_nMaxIterationNum = 10;
int g_nOpenCloseNum = 0;
int g_nErodeDilateNum = 0;
```

```
int g_nTopBlackHatNum = 0;


//-----------------------【全局函数声明部分】-----------------------
//      描述：全局函数声明
//-------------------------------------------------------------------
static void on_OpenClose(int, void*);//回调函数
static void on_ErodeDilate(int, void*);//回调函数
static void on_TopBlackHat(int, void*);//回调函数
static void ShowHelpText();//帮助文字显示


//-----------------------【main()函数】-----------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------
int main()
{
    //载入原图
    g_srcImage = imread("1.jpg");//工程目录下需要有一张名为1.jpg的素材图
    if( !g_srcImage.data ) { printf("读取srcImage错误~！ \n"); return
false; }

    //显示原始图
    namedWindow("【原始图】");
    imshow("【原始图】", g_srcImage);

    //创建三个窗口
    namedWindow("【开运算/闭运算】",1);
    namedWindow("【腐蚀/膨胀】",1);
    namedWindow("【顶帽/黑帽】",1);

    //参数赋值
    g_nOpenCloseNum=9;
    g_nErodeDilateNum=9;
    g_nTopBlackHatNum=2;

    //分别为三个窗口创建滚动条
    createTrackbar("迭代值", "【开运算/闭运算】
    ",&g_nOpenCloseNum,g_nMaxIterationNum*2+1,on_OpenClose);
    createTrackbar("迭代值", "【腐蚀/膨胀】
    ",&g_nErodeDilateNum,g_nMaxIterationNum*2+1,on_ErodeDilate);
    createTrackbar("迭代值", "【顶帽/黑帽】
    ",&g_nTopBlackHatNum,g_nMaxIterationNum*2+1,on_TopBlackHat);

    //轮询获取按键信息
    while(1)
    {
        int c;

        //执行回调函数
        on_OpenClose(g_nOpenCloseNum, 0);
```

```
        on_ErodeDilate(g_nErodeDilateNum, 0);
        on_TopBlackHat(g_nTopBlackHatNum,0);

        //获取按键
        c = waitKey(0);

        //按下键盘按键Q或者ESC，程序退出
        if( (char)c == 'q'||(char)c == 27 )
            break;
        //按下键盘按键1，使用椭圆(Elliptic)结构元素结构元素MORPH_ELLIPSE
        if( (char)c == 49 )//键盘按键1的ASII码为49
            g_nElementShape = MORPH_ELLIPSE;
        //按下键盘按键2，使用矩形(Rectangle)结构元素MORPH_RECT
        else if( (char)c == 50 )//键盘按键2的ASII码为50
            g_nElementShape = MORPH_RECT;
        //按下键盘按键3，使用十字形(Cross-shaped)结构元素MORPH_CROSS
        else if( (char)c == 51 )//键盘按键3的ASII码为51
            g_nElementShape = MORPH_CROSS;
        //按下键盘按键space，在矩形、椭圆、十字形结构元素中循环
        else if( (char)c == ' ' )
            g_nElementShape = (g_nElementShape + 1) % 3;
    }

    return 0;
}


//----------------------【on_OpenClose()函数】----------------------
//      描述:【开运算/闭运算】窗口的回调函数
//-----------------------------------------------------------------
static void on_OpenClose(int, void*)
{
    //偏移量的定义
    int offset = g_nOpenCloseNum - g_nMaxIterationNum;//偏移量
    int Absolute_offset = offset > 0 ? offset : -offset;//偏移量绝对值
    //自定义核
    Mat element = getStructuringElement(g_nElementShape,
Size(Absolute_offset*2+1, Absolute_offset*2+1), Point(Absolute_offset,
Absolute_offset) );
    //进行操作
    if( offset < 0 )
        //此句代码的OpenCV2版为:
        //morphologyEx(g_srcImage, g_dstImage, CV_MOP_OPEN, element);
        //此句代码的OpenCV3版为:
        morphologyEx(g_srcImage, g_dstImage, MORPH_OPEN, element);
    else
        //此句代码的OpenCV2版为:
        //morphologyEx(g_srcImage, g_dstImage, CV_MOP_CLOSE, element);
        //此句代码的OpenCV3版为:
        morphologyEx(g_srcImage, g_dstImage, MORPH_CLOSE, element);
    //显示图像
    imshow("【开运算/闭运算】",g_dstImage);
```

```
}


//----------------------【on_ErodeDilate()函数】----------------------
//      描述：【腐蚀/膨胀】窗口的回调函数
//-----------------------------------------------------------------------
static void on_ErodeDilate(int, void*)
{
    //偏移量的定义
    int offset = g_nErodeDilateNum - g_nMaxIterationNum;    //偏移量
    int Absolute_offset = offset > 0 ? offset : -offset;//偏移量绝对值
    //自定义核
    Mat element = getStructuringElement(g_nElementShape,
Size(Absolute_offset*2+1, Absolute_offset*2+1), Point(Absolute_offset,
Absolute_offset) );
    //进行操作
    if( offset < 0 )
        erode(g_srcImage, g_dstImage, element);
    else
        dilate(g_srcImage, g_dstImage, element);
    //显示图像
    imshow("【腐蚀/膨胀】",g_dstImage);
}


//----------------------【on_TopBlackHat()函数】----------------------
//      描述：【顶帽运算/黑帽运算】窗口的回调函数
//-----------------------------------------------------------------------
static void on_TopBlackHat(int, void*)
{
    //偏移量的定义
    int offset = g_nTopBlackHatNum - g_nMaxIterationNum;//偏移量
    int Absolute_offset = offset > 0 ? offset : -offset;//偏移量绝对值
    //自定义核
    Mat element = getStructuringElement(g_nElementShape,
Size(Absolute_offset*2+1, Absolute_offset*2+1), Point(Absolute_offset,
Absolute_offset) );
    //进行操作
    if( offset < 0 )
        morphologyEx(g_srcImage, g_dstImage, MORPH_TOPHAT , element);
    else
        morphologyEx(g_srcImage, g_dstImage, MORPH_BLACKHAT, element);
    //显示图像
    imshow("【顶帽/黑帽】",g_dstImage);
}
```

以上便是经过详细注释的程序代码，相信大家通过上文的学习，可以很好地掌握这些代码的含义和用法。下面放出此程序的运行效果图（图片素材为电影《美国队长2》的海报）。原图为6.41，效果图为6.42、6.43、6.44、6.45、6.46、6.47。

图 6.41 原始图

图 6.42 腐蚀效果图

图 6.43 膨胀效果图

图 6.44 开运算效果图

图 6.45 闭运算效果图

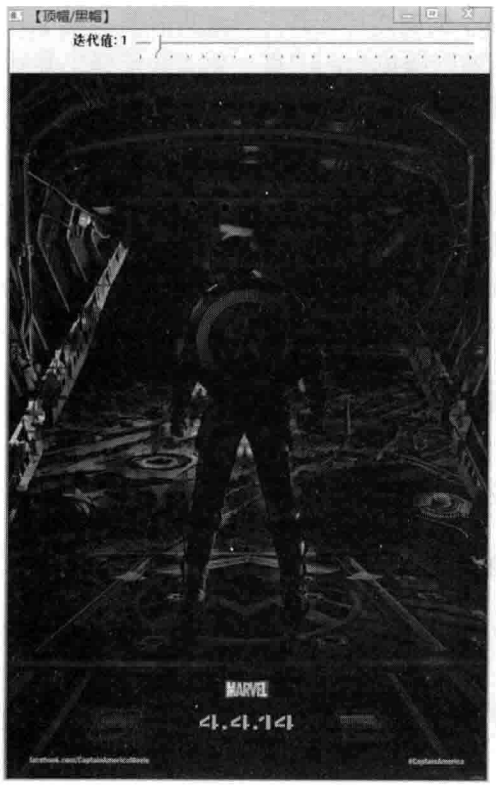


图 6.46 顶帽运算效果图

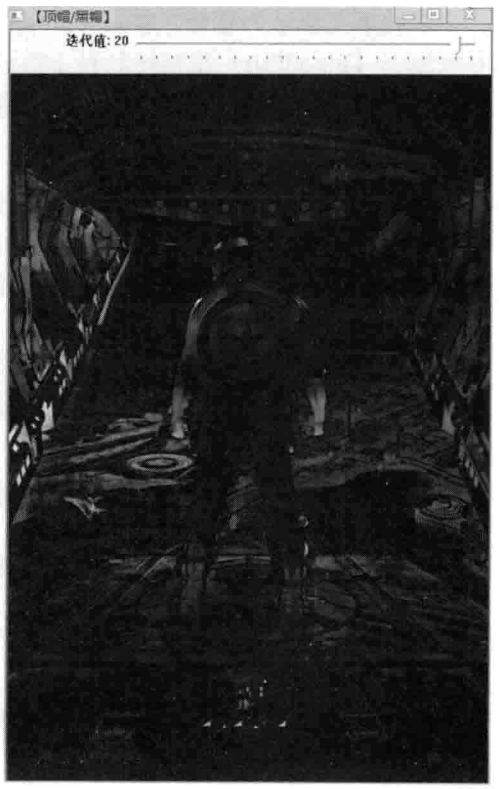


图 6.47 黑帽运算效果图

6.5 漫水填充

本节我们将一起探讨 OpenCV 填充算法中漫水填充算法相关的知识点，并了解 OpenCV 中实现漫水填充算法的两个版本的 floodFill 函数的使用方法。

6.5.1 漫水填充的定义

漫水填充法是一种用特定的颜色填充连通区域，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。

另外，floodfill 官方译作“漫水填充”，但是很多朋友们总是喜欢说成“水漫填充”，因为受从小到大深入骨髓的“水漫金山”这个成语的影响。

6.5.2 漫水填充法的基本思想

所谓漫水填充，简单来说，就是自动选中了和种子点相连的区域，接着将该区域替换成指定的颜色，这是个非常有用的功能，经常用来标记或者分离图像的一部分进行处理或分析。漫水填充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，或者只处理掩码指定的像素点。

以此填充算法为基础，类似 PhotoShop 的魔术棒选择工具就很容易实现了。漫水填充（FloodFill）是查找和种子点连通的颜色相同的点，魔术棒选择工具则是查找和种子点连通的颜色相近的点，把和初始种子像素颜色相近的点压进栈做为新种子。

在 OpenCV 中，漫水填充是填充算法中最通用的方法。且在 OpenCV 2.X 中，使用 C++重写过的 FloodFill 函数有两个版本：一个不带掩膜 mask 的版本，和一个带 mask 的版本。这个掩膜 mask，就是用于进一步控制哪些区域将被填充颜色（比如说当对同一图像进行多次填充时）。这两个版本的 FloodFill，都必须在图像中选择一个种子点，然后把临近区域所有相似点填充上同样的颜色，不同的是，不一定将所有的邻近像素点都染上同一颜色，漫水填充操作的结果总是某个连续的区域。当邻近像素点位于给定的范围（从 loDiff 到 upDiff）内或在原始 seedPoint 像素值范围内时，FloodFill 函数就会为这个点涂上颜色。

6.5.3 实现漫水填充算法：floodFill 函数

在 OpenCV 中，漫水填充算法由 floodFill 函数实现，其作用是用我们指定的颜色从种子点开始填充一个连接域。连通性由像素值的接近程度来衡量。OpenCV2.X 有两个 C++重写版本的 floodFill，具体如下。

第一个版本的 floodFill：

```
int floodFill(InputOutputArray image, Point seedPoint, Scalar newVal,
```

```
Rect* rect=0, Scalar loDiff=Scalar(), Scalar upDiff=Scalar(), int
flags=4 )
```

第二个版本的 floodFill：

```
int floodFill(InputOutputArray image, InputOutputArray mask, Point
seedPoint,Scalar newVal, Rect* rect=0, Scalar loDiff=Scalar(), Scalar
upDiff=Scalar(), int flags=4 )
```

下面是参数详解。这两个版本除了第二个参数外，其他的参数都是共用的。

（1）第一个参数，InputOutputArray 类型的 image，输入/输出 1 通道或 3 通道，8 位或浮点图像，具体参数由之后的参数指明。

（2）第二个参数，InputOutputArray 类型的 mask，这是第二个版本的 floodFill 独享的参数，表示操作掩模。它应该为单通道，8 位，长和宽上都比输入图像 image 大两个像素点的图像。第二个版本的 floodFill 需要使用以及更新掩膜，所以对于这个 mask 参数，我们一定要将其准备好并填在此处。需要注意的是，漫水填充不会填充掩膜 mask 的非零像素区域。例如，一个边缘检测算子的输出可以用来作为掩膜，以防止填充到边缘。同样的，也可以在多次的函数调用中使用同一个掩膜，以保证填充的区域不会重叠。另外需要注意的是，掩膜 mask 会比需填充的图像大，所以 mask 中与输入图像(x,y)像素点相对应的点的坐标为(x+1,y+1)。

（3）第三个参数，Point 类型的 seedPoint，漫水填充算法的起始点。

（4）第四个参数，Scalar 类型的 newVal，像素点被染色的值，即在重绘区域像素的新值。

（5）第五个参数，Rect*类型的 rect，有默认值 0，一个可选的参数，用于设置 floodFill 函数将要重绘区域的最小边界矩形区域。

（6）第六个参数，Scalar 类型的 loDiff，有默认值 Scalar()，表示当前观察像素值与其部件邻域像素值或者待加入该部件的种子像素之间的亮度或颜色之负差（lower brightness/color difference）的最大值。

（7）第七个参数，Scalar 类型的 upDiff，有默认值 Scalar()，表示当前观察像素值与其部件邻域像素值或者待加入该部件的种子像素之间的亮度或颜色之正差（lower brightness/color difference）的最大值。

（8）第八个参数，int 类型的 flags，操作标志符，此参数包含三个部分，比较复杂，我们一起详细看看。

- 低八位（第 0～7 位）用于控制算法的连通性，可取 4（4 为默认值）或者 8。如果设为 4，表示填充算法只考虑当前像素水平方向和垂直方向的相邻点；如果设为 8，除上述相邻点外，还会包含对角线方向的相邻点。
- 高八位部分（16～23 位）可以为 0 或者如下两种选项标识符的组合。
  - ◆ FLOODFILL_FIXED_RANGE：如果设置为这个标识符，就会考虑当前像素与种子像素之间的差，否则就考虑当前像素与其相邻像素的差。也就是说，这个范围是浮动的。

- ◆ FLOODFILL_MASK_ONLY - 如果设置为这个标识符，函数不会去填充改变原始图像（也就是忽略第三个参数 newVal）,而是去填充掩模图像（mask）。这个标识符只对第二个版本的 floodFill 有用，因第一个版本里面压根就没有 mask 参数。

- 中间八位部分，上面关于高八位 FLOODFILL_MASK_ONLY 标识符中已经说得很明显，需要输入符合要求的掩码。Floodfill 的 flags 参数的中间八位的值就是用于指定填充掩码图像的值的。但如果 flags 中间八位的值为 0，则掩码会用 1 来填充。

而所有 flags 可以用 or 操作符连接起来，即“|”。例如，如果想用 8 邻域填充，并填充固定像素值范围，填充掩码而不是填充源图像，以及设填充值为 38，那么输入的参数是下面这样：

```
flags=8 | FLOODFILL_MASK_ONLY | FLOODFILL_FIXED_RANGE | (38<<8)
```

接着，来看一个关于 Floodfill 的简单的调用范例。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;

//---------------------------【main()函数】---------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------

int main()
{
    Mat src = imread("1.jpg");
    imshow("【原始图】",src);
    Rect ccomp;
    floodFill(src, Point(50,300), Scalar(155, 255,55), &ccomp,
Scalar(20, 20, 20),Scalar(20, 20, 20));
    imshow("【效果图】",src);
    waitKey(0);
    return 0;
}
```

接着我们看看此程序的运行截图。原图如图 6.48 所示，效果图如图 6.49 所示。

6.5.4 综合示例：漫水填充

本次的综合示例为 OpenCV 文档中自带的一个程序。作者对其做了适当的修改并详细注释，供大家消化理解。

操作说明如图 6.50 所示。

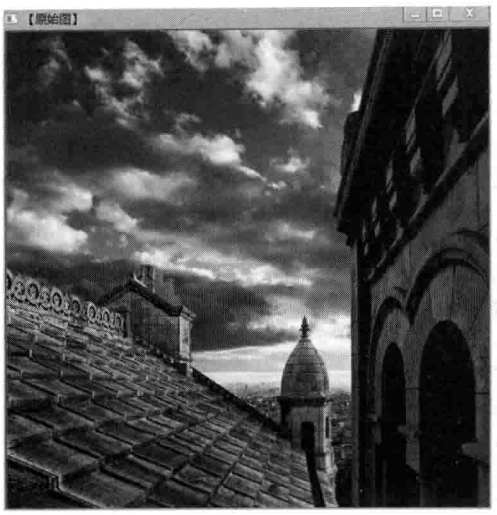


图 6.48 原始图

图 6.49 漫水填充效果图

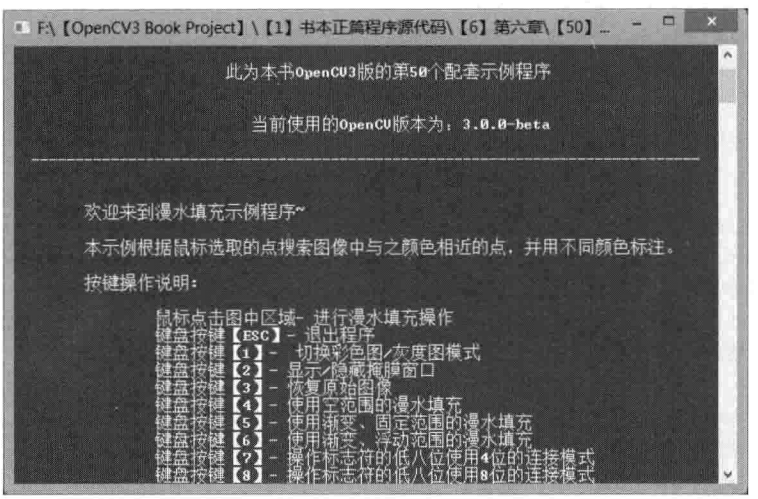


图 6.50 程序操作说明

可以看到，此程序有不少按键功能。我们用鼠标对窗口中的图形进行多次单击，就可以得到类似 PhotoShop 中魔棒的效果。当然，就这短短的两百来行代码写出来的东西，体验还是比不上 PS 的魔棒工具的。程序详细注释的源码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
using namespace cv;
using namespace std;


//-------------------------【全局变量声明部分】-------------------------
```

```
//      描述：全局变量声明
//-----------------------------------------------------------------------
Mat g_srcImage, g_dstImage, g_grayImage, g_maskImage;//定义原始图、目标图、灰度图、掩模图
int g_nFillMode = 1;//漫水填充的模式
int g_nLowDifference = 20, g_nUpDifference = 20;//负差最大值、正差最大值
int g_nConnectivity = 4;//表示 floodFill 函数标识符低八位的连通值
int g_bIsColor = true;//是否为彩色图的标识符布尔值
bool g_bUseMask = false;//是否显示掩膜窗口的布尔值
int g_nNewMaskVal = 255;//新的重新绘制的像素值

//-------------------------【onMouse()函数】-------------------------
//      描述：鼠标消息 onMouse 回调函数
//-----------------------------------------------------------------------
static void onMouse( int event, int x, int y, int, void* )
{
    // 若鼠标左键没有按下，便返回
    //此句代码的 OpenCV2 版为：
    //if( event != CV_EVENT_LBUTTONDOWN )
    //此句代码的 OpenCV3 版为：
    if( event != EVENT_LBUTTONDOWN )
        return;

    //-------------【<1>调用 floodFill 函数之前的参数准备部分】-------------
    Point seed = Point(x,y);
    int LowDifference = g_nFillMode == 0 ? 0 : g_nLowDifference;//空范围的漫水填充，此值设为 0，否则设为全局的 g_nLowDifference
    int UpDifference = g_nFillMode == 0 ? 0 : g_nUpDifference;//空范围的漫水填充，此值设为 0，否则设为全局的 g_nUpDifference

    //标识符的 0~7 位为 g_nConnectivity，8~15 位为 g_nNewMaskVal 左移 8 位的值，16~23 位为 CV_FLOODFILL_FIXED_RANGE 或者 0。
    //此句代码的 OpenCV2 版为：
    //int flags = g_nConnectivity + (g_nNewMaskVal << 8) +(g_nFillMode == 1 ? CV_FLOODFILL_FIXED_RANGE : 0);
    //此句代码的 OpenCV3 版为：
    int flags = g_nConnectivity + (g_nNewMaskVal << 8) +(g_nFillMode == 1 ? FLOODFILL_FIXED_RANGE : 0);


    //随机生成 bgr 值
    int b = (unsigned)theRNG() & 255;//随机返回一个 0～255 之间的值
    int g = (unsigned)theRNG() & 255;//随机返回一个 0～255 之间的值
    int r = (unsigned)theRNG() & 255;//随机返回一个 0～255 之间的值
    Rect ccomp;//定义重绘区域的最小边界矩形区域

    Scalar newVal = g_bIsColor ? Scalar(b, g, r) : Scalar(r*0.299 + g*0.587 + b*0.114);//在重绘区域像素的新值，若是彩色图模式，取 Scalar(b, g, r)；若是灰度图模式，取 Scalar(r*0.299 + g*0.587 + b*0.114)

    Mat dst = g_bIsColor ? g_dstImage : g_grayImage;//目标图的赋值
    int area;
```

```
    //-------------------【<2>正式调用 floodFill 函数】-------------------
    if( g_bUseMask )
    {
        //此句代码的 OpenCV2 版为:
        //threshold(g_maskImage, g_maskImage, 1, 128,
CV_THRESH_BINARY);
        //此句代码的 OpenCV3 版为:
        threshold(g_maskImage, g_maskImage, 1, 128, THRESH_BINARY);

        area = floodFill(dst, g_maskImage, seed, newVal, &ccomp,
Scalar(LowDifference, LowDifference, LowDifference),
            Scalar(UpDifference, UpDifference, UpDifference), flags);
        imshow( "mask", g_maskImage );
    }
    else
    {
        area = floodFill(dst, seed, newVal, &ccomp, Scalar(LowDifference,
LowDifference, LowDifference),
            Scalar(UpDifference, UpDifference, UpDifference), flags);
    }

    imshow("效果图", dst);
    cout << area << " 个像素被重绘\n";
}


//-----------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main( int argc, char** argv )
{
    //载入原图
    g_srcImage = imread("1.jpg", 1);

    if( !g_srcImage.data ) { printf("读取图片 image0 错误~！ \n"); return
false; }

    g_srcImage.copyTo(g_dstImage);//复制源图到目标图
    cvtColor(g_srcImage, g_grayImage, COLOR_BGR2GRAY);//转换三通道的
image0 到灰度图
    g_maskImage.create(g_srcImage.rows+2, g_srcImage.cols+2,
CV_8UC1);//利用 image0 的尺寸来初始化掩膜 mask

        //此句代码的 OpenCV2 版为:
    //namedWindow( "效果图",CV_WINDOW_AUTOSIZE );
    //此句代码的 OpenCV2 版为:
    namedWindow( "效果图",WINDOW_AUTOSIZE );

    //创建 Trackbar
    createTrackbar( "负差最大值", "效果图", &g_nLowDifference, 255, 0 );
    createTrackbar( "正差最大值" ,"效果图", &g_nUpDifference, 255, 0 );
```

```
    //鼠标回调函数
    setMouseCallback( "效果图", onMouse, 0 );

    //循环轮询按键
    while(1)
    {
        //先显示效果图
        imshow("效果图", g_bIsColor ? g_dstImage : g_grayImage);

        //获取键盘按键
        int c = waitKey(0);
        //判断ESC是否按下，若按下便退出
        if( (c & 255) == 27 )
        {
            cout << "程序退出.....\n";
            break;
        }

        //根据按键的不同，进行各种操作
        switch( (char)c )
        {
        //如果键盘"1"被按下，效果图在在灰度图，彩色图之间互换
        case '1':
            if( g_bIsColor )//若原来为彩色，转为灰度图，并且将掩膜mask所有元
素设置为0
            {
                cout << "键盘"1"被按下，切换彩色/灰度模式，当前操作为将【彩色模
式】切换为【灰度模式】\n";
                cvtColor(g_srcImage, g_grayImage, COLOR_BGR2GRAY);
                g_maskImage = Scalar::all(0);   //将mask所有元素设置为0
                g_bIsColor = false; //将标识符置为false，表示当前图像不为彩
色，而是灰度
            }
            else//若原来为灰度图，便将原来的彩图image0再次复制给image，并且将
掩膜mask所有元素设置为0
            {
                cout << "键盘"1"被按下，切换彩色/灰度模式，当前操作为将【彩色模
式】切换为【灰度模式】\n";
                g_srcImage.copyTo(g_dstImage);
                g_maskImage = Scalar::all(0);
                g_bIsColor = true;//将标识符置为true，表示当前图像模式为彩色
            }
            break;
        //如果键盘按键"2"被按下，显示/隐藏掩膜窗口
        case '2':
            if( g_bUseMask )
            {
                destroyWindow( "mask" );
                g_bUseMask = false;
            }
            else
```

```
            {
                namedWindow( "mask", 0 );
                g_maskImage = Scalar::all(0);
                imshow("mask", g_maskImage);
                g_bUseMask = true;
            }
            break;
        //如果键盘按键"3"被按下，恢复原始图像
        case '3':
            cout << "按键"3"被按下，恢复原始图像\n";
            g_srcImage.copyTo(g_dstImage);
            cvtColor(g_dstImage, g_grayImage, COLOR_BGR2GRAY);
            g_maskImage = Scalar::all(0);
            break;
        //如果键盘按键"4"被按下，使用空范围的漫水填充
        case '4':
            cout << "按键"4"被按下，使用空范围的漫水填充\n";
            g_nFillMode = 0;
            break;
        //如果键盘按键"5"被按下，使用渐变、固定范围的漫水填充
        case '5':
            cout << "按键"5"被按下，使用渐变、固定范围的漫水填充\n";
            g_nFillMode = 1;
            break;
        //如果键盘按键"6"被按下，使用渐变、浮动范围的漫水填充
        case '6':
            cout << "按键"6"被按下，使用渐变、浮动范围的漫水填充\n";
            g_nFillMode = 2;
            break;
        //如果键盘按键"7"被按下，操作标志符的低八位使用 4 位的连接模式
        case '7':
            cout << "按键"7"被按下，操作标志符的低八位使用 4 位的连接模式\n";
            g_nConnectivity = 4;
            break;
        //如果键盘按键"8"被按下，操作标志符的低八位使用 8 位的连接模式
        case '8':
            cout << "按键"8"被按下，操作标志符的低八位使用 8 位的连接模式\n";
            g_nConnectivity = 8;
            break;
        }
    }

    return 0;
}
```

我们在窗口图片上单击鼠标，就可以给其中不同的区域随机着色。程序功能非常丰富，有鼠标操作和键盘 8 个按键的操作，还可以调滑动条。图 6.52、6.53 所示是上述这两百来行代码勾勒出来的程序的一些运行截图，原图如图 6.51 所示。更多的运行效果请大家自行运行书本配套示例程序进行学习和观赏。

图 6.51 原始图

图 6.52 运行截图（1）

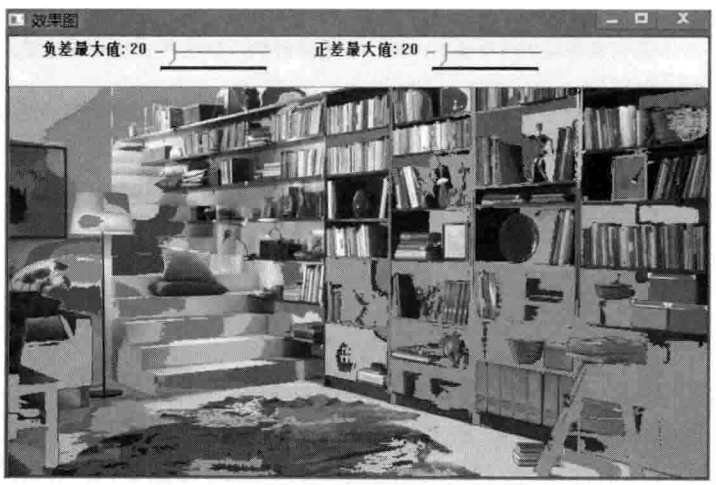


图 6.53 运行截图（2）

随着鼠标的单击，程序会记下我们的以下操作，如图 6.54 所示。

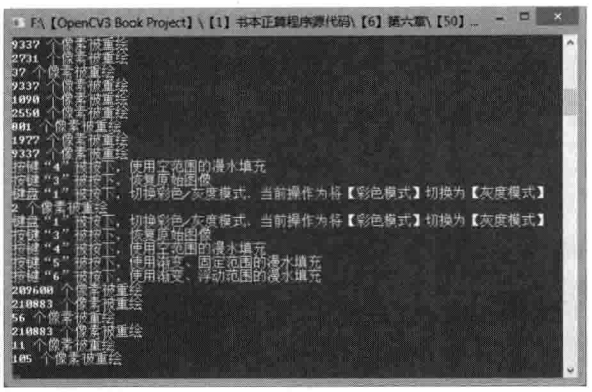


图 6.54 操作记录说明

6.6 图像金字塔与图片尺寸缩放

本节中，我们将一起探讨图像金字塔的一些基本概念，学习如何使用 OpenCV 函数 pyrUp 和 pyrDown 对图像进行向上和向下采样，以及了解专门用于缩放图像尺寸的 resize 函数的用法。

6.6.1 引言

我们经常会将某种尺寸的图像转换为其他尺寸的图像，如果要放大或者缩小图片的尺寸，笼统来说，可以使用 OpenCV 提供的如下两种方法。

- resize 函数。这是最直接的方式
- pyrUp()、pyrDown()函数。即图像金字塔相关的两个函数，对图像进行向上采样和向下采样的操作。

pyrUp、pyrDown 其实和专门用作放大缩小图像尺寸的 resize 在功能上差不多，披着图像金字塔的皮，说白了还是在对图像进行放大和缩小操作。另外需要指出的是，pyrUp、pyrDown 在 OpenCV 的 imgproc 模块中的 Image Filtering 子模块里，而 resize 在 imgproc 模块的 Geometric Image Transformations 子模块里。

本节，我们将先介绍图像金字塔的原理，接着介绍 resize 函数，然后是 pyrUp 和 pyrDown 函数，最后是一个综合示例程序。

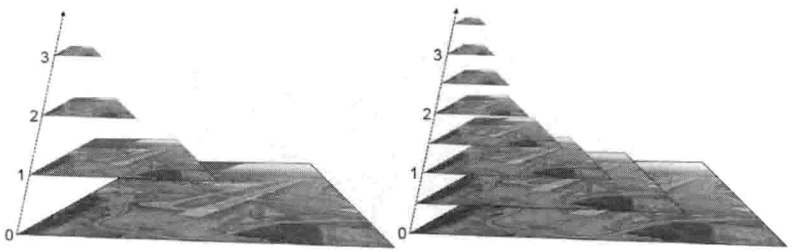
6.6.2 关于图像金字塔

图像金字塔是图像中多尺度表达的一种，最主要用于图像的分割，是一种以多分辨率来解释图像的有效但概念简单的结构。

图像金字塔最初用于机器视觉和图像压缩，一幅图像的金字塔是一系列以金字塔形状排列的，分辨率逐步降低且来源于同一张原始图的图像集合。其通过梯次向下采样获得，直到达到某个终止条件才停止采样。

金字塔的底部是待处理图像的高分辨率表示，而顶部是低分辨率的近似。

我们将一层一层的图像比喻成金字塔，层级越高，则图像越小，分辨率越低。如图 6.55、6.56 所示。



6.55 图像金字塔演示图（1）

6.56 图像金字塔演示图（2）

一般情况下有两种类型的图像金字塔常常出现在文献和以及实际运用中。它们分别是:

- 高斯金字塔（Gaussianpyramid）——用来向下采样，主要的图像金字塔。
- 拉普拉斯金字塔（Laplacianpyramid）——用来从金字塔低层图像重建上层未采样图像，在数字图像处理中也即是预测残差，可以对图像进行最大程度的还原，配合高斯金字塔一起使用。

两者的简要区别在于：高斯金字塔用来向下降采样图像，而拉普拉斯金字塔则用来从金字塔底层图像中向上采样，重建一个图像。

要从金字塔第 i 层生成第 i+1 层（我们将第 i+1 层表示为 G_{i+1}），我们先要用高斯核对 G_i 进行卷积，然后删除所有偶数行和偶数列，新得到图像面积会变为源图像的四分之一。按上述过程对输入图像 G_0 执行操作就可产生出整个金字塔。

当图像向金字塔的上层移动时，尺寸和分辨率会降低。OpenCV 中，从金字塔中上一级图像生成下一级图像的可以用 PryDown，而通过 PryUp 将现有的图像在每个维度都放大两遍。

图像金字塔中的向上和向下采样分别通过 OpenCV 的函数 pyrUp 和 pyrDown 实现。

概括起来就是：

- 对图像向上采样——pyrUp 函数；
- 对图像向下采样——pyrDown 函数。

这里的向下与向上采样，是针对图像的尺寸而言的（和金字塔的方向相反），向上就是图像尺寸加倍，向下就是图像尺寸减半。而如果按图 6.55 和图 6.56 中演示的金字塔方向来理解，金字塔向上图像其实在缩小，这样刚好是反过来了。

但需要注意的是，PryUp 和 PryDown 不是互逆的，即 PryUp 不是降采样的逆操作。这种情况下，图像首先在每个维度上扩大为原来的两倍，新增的行（偶数行）以 0 填充。然后给指定的滤波器进行卷积（实际上是一个在每个维度都扩大为原来两倍的过滤器）去估计“丢失”像素的近似值。

PryDown()是一个会丢失信息的函数。为了恢复原来更高的分辨率的图像，我们要获得由降采样操作丢失的信息，这些数据就和拉普拉斯金字塔有关系了。

6.6.3 高斯金字塔

高斯金字塔是通过高斯平滑和亚采样获得一些列下采样图像，也就是说第 K 层高斯金字塔通过平滑、亚采样就可以获得 K+1 层高斯图像。高斯金字塔包含了一系列低通滤波器，其截止频率从上一层到下一层以因子 2 逐渐增加，所以高斯金字塔可以跨越很大的频率范围。金字塔的图像如图 6.57 所示。

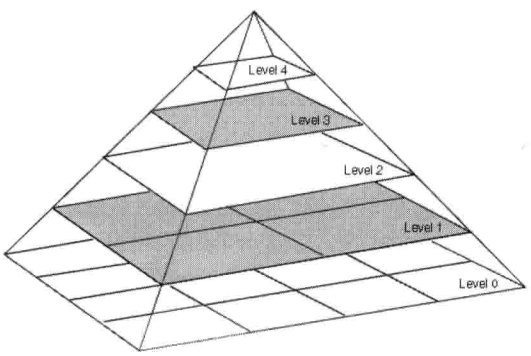


图 6.57 图像金字塔演示图（3）

另外，每一层都按从下到上的次序编号，层级 G_{i+1} (表示为 G_{i+1} 尺寸小于第 i 层 G_i)。

1. 对图像的向下取样

为了获取层级为 G_{i+1} 的金字塔图像，我们采用如下方法:

（1）对图像 G_i 进行高斯内核卷积；

（2）将所有偶数行和列去除。

得到的图像即为 G_{i+1} 的图像。显而易见，结果图像只有原图的四分之一。通过对输入图像 G_i（原始图像）不停迭代以上步骤就会得到整个金字塔。同时我们也可以看到，向下取样会逐渐丢失图像的信息。

以上就是对图像的向下取样操作，即缩小图像。

2．对图像的向上取样

如果想放大图像，则需要通过向上取样操作得到，具体做法如下。

（1）将图像在每个方向扩大为原来的两倍，新增的行和列以 0 填充。

（2）使用先前同样的内核（乘以 4）与放大后的图像卷积，获得 “新增像素” 的近似值。

得到的图像即为放大后的图像，但是与原来的图像相比会发觉比较模糊，因为在缩放的过程中已经丢失了一些信息。如果想在缩小和放大整个过程中减少信息的丢失，这些数据就形成了拉普拉斯金字塔。

接下来一起看一下拉普拉斯金字塔的概念。

6.6.4 拉普拉斯金字塔

下式是拉普拉斯金字塔第 i 层的数学定义：

$$L_i = G_i - \mathrm{UP}(G_{i+1}) \otimes g_{5\times5}$$

式中的 G_i 表示第 i 层的图像。而 UP()操作是将源图像中位置为(x,y)的像素映射到目标图像的(2x+1,2y+1)位置，即在进行向上取样。符号 ⊗ 表示卷积，g_{5x5} 为 5x5 的高斯内核。

我们下文将要介绍的 pryUp，就是在进行上面这个式子的运算。

因此，我们可以直接用 OpenCV 进行拉普拉斯运算：L_i=G_i-PyrUp(G_{i+1})

也就是说，拉普拉斯金字塔是通过源图像减去先缩小后再放大的图像的一系列图像构成的。

整个拉普拉斯金字塔运算过程可以通过图 6.58 来概括.

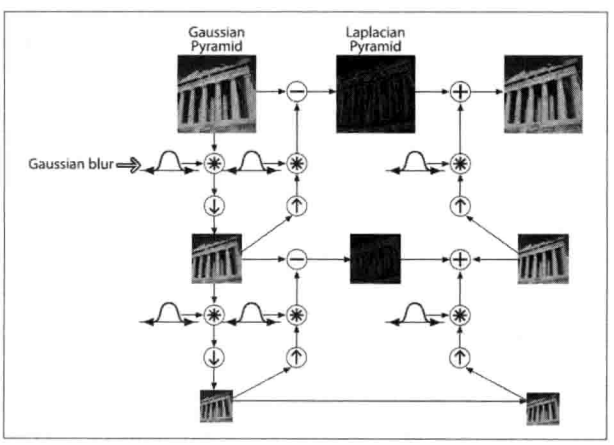


图 6.58 拉普拉斯金字塔运算过程

所以，我们可以将拉普拉斯金字塔理解为高斯金字塔的逆形式。

另外再提一点，关于图像金字塔非常重要的一个应用就是图像分割。图像分割的话，先要建立一个图像金字塔，然后对 G_i 和 G_{i+1} 的像素直接依照对应的关系，建立起“父与子”关系。而快速初始分割可以先在金字塔高层的低分辨率图像上完成，然后逐层对分割加以优化。

6.6.5 尺寸调整：resize()函数

resize()为 OpenCV 中专门用来调整图像大小的函数。

此函数将源图像精确地转换为指定尺寸的目标图像。如果源图像中设置了 ROI（Region Of Interest ，感兴趣区域），那么 resize()函数会对源图像的 ROI 区域进行调整图像尺寸的操作，来输出到目标图像中。若目标图像中已经设置了 ROI 区域，不难理解 resize()将会对源图像进行尺寸调整并填充到目标图像的 ROI 中。

很多时候，我们并不用考虑第二个参数 dst 的初始图像尺寸和类型（即直接定义一个 Mat 类型，不用对其初始化），因为其尺寸和类型可以由 src、dsize、fx 和 fy 这几个参数来确定。

看一下它的函数原型：

```
C++: void resize(InputArray src,OutputArray dst, Size dsize, double fx=0,
double fy=0, int interpolation=INTER_LINEAR )
```

（1）第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。

（2）第二个参数，OutputArray 类型的 dst，输出图像，当其非零时，有着 dsize（第三个参数）的尺寸，或者由 src.size()计算出来。

（3）第三个参数，Size 类型的 dsize，输出图像的大小。如果它等于零，由下式进行计算：

dsize=Size(round(fx*src.cols)，round(fy*src.rows))

其中，dsize、fx、fy 都不能为 0。

（4）第四个参数，double 类型的 fx，沿水平轴的缩放系数，有默认值 0，且当其等于 0 时，由下式进行计算：

(double)dsize.width/src.cols

（5）第五个参数，double 类型的 fy，沿垂直轴的缩放系数，有默认值 0，且当其等于 0 时，由下式进行计算：

(double)dsize.height/src.rows

（6）第六个参数，int 类型的 interpolation，用于指定插值方式，默认为 INTER_LINEAR（线性插值）。

可选的插值方式如下：

- INTER_NEAREST——最近邻插值

- INTER_LINEAR——线性插值（默认值）
- INTER_AREA——区域插值（利用像素区域关系的重采样插值）
- INTER_CUBIC——三次样条插值（超过 4×4 像素邻域内的双三次插值）
- INTER_LANCZOS4——Lanczos 插值（超过 8×8 像素邻域的 Lanczos 插值）

若要缩小图像，一般情况下最好用 CV_INTER_AREA 来插值；而若要放大图像，一般情况下最好用 CV_INTER_CUBIC（效率不高，慢，不推荐使用）或 CV_INTER_LINEAR（效率较高，速度较快，推荐使用）。

关于插值，我们看几张图就能更好地理解。先看原图（图 6.59）。

图 6.59 原始图

当进行 6 次图像缩小接着 6 次图像放大操作后，两种不同的插值方式得到的效果图如图 6.60、图 6.61 所示。

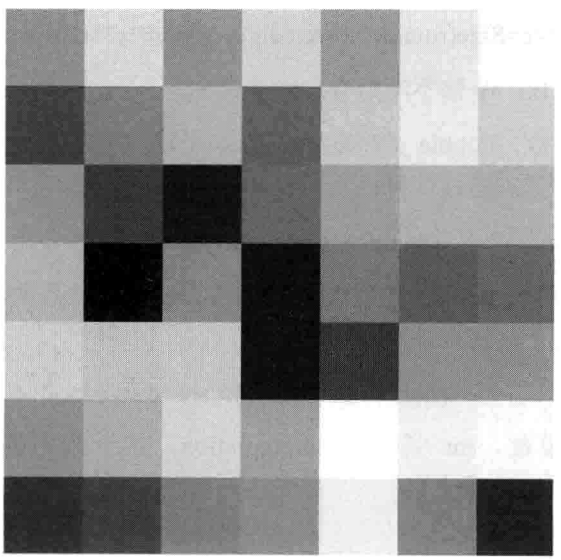


图 6.60 插值效果图（1）

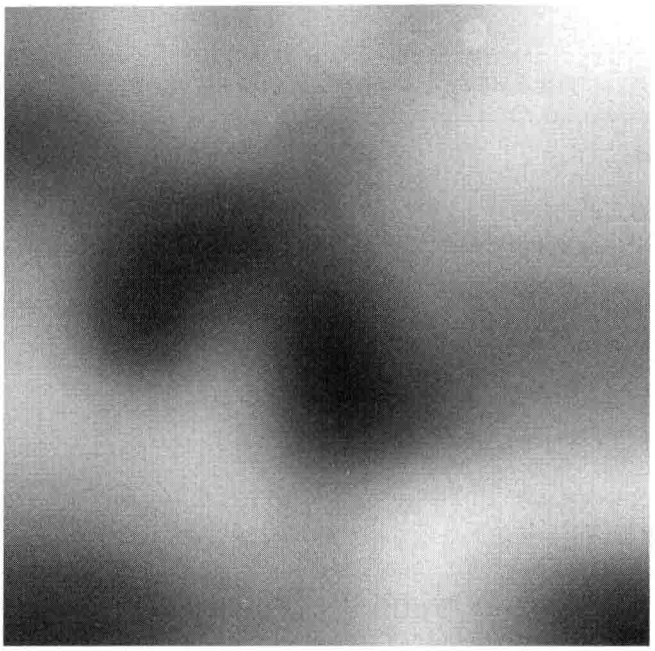


图 6.61 插值效果图（2）

效果很明显，第一张全是一个个的像素，非常影响美观。另外一张却有雾化的朦胧美感，所以插值方式的选择，对经过多次放大缩小的图片最终得到的效果是有很大影响的。

接着我们来看两种 resize 的调用范例。

（1）方式一

调用范例：

```
Mat dstImage=Mat::zeros(512 ,512, CV_8UC3 );//新建一张 512x512 尺寸的图片
Mat srcImage=imread("1.jpg");
//显式指定 dsize=dstImage.size(),那么 fx 和 fy 会其计算出来，不用额外指定。
resize(srcImage, dstImage, dstImage.size());
```

（2）方式二

调用范例：

```
Mat dstImage;
Mat srcImage=imread("1.jpg")
 //指定 fx 和 fy，让函数计算出目标图像的大小。
resize(srcImage, dstImage, Size(), 0.5, 0.5);
```

接着我们看看完整的示例程序。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;
```

```
//-----------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
    //载入原始图
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为1.jpg的素材
图
    Mat tmpImage,dstImage1,dstImage2;//临时变量和目标图的定义
    tmpImage=srcImage;//将原始图赋给临时变量

    //显示原始图
    imshow("【原始图】", srcImage);

    //进行尺寸调整操作
    resize(tmpImage,dstImage1,Size(tmpImage.cols/2,
tmpImage.rows/2 ),(0,0),(0,0),3);
    resize(tmpImage,dstImage2,Size(tmpImage.cols*2,
tmpImage.rows*2 ),(0,0),(0,0),3);

    //显示效果图
    imshow("【效果图】之一", dstImage1);
    imshow("【效果图】之二", dstImage2);

    waitKey(0);
    return 0;
}
```

程序运行截图如图 6.62 所示。

图 6.62 程序运行截图

6.6.6 图像金字塔相关 API 函数

图像金字塔相关 API 函数主要是 pyrUp、pyrDown 这一对，下面分别对其进行讲解。

1. 向上采样：pyrUp()函数

pyrUp()函数的作用是向上采样并模糊一张图像，说白了就是放大一张图片。

```
C++: void pyrUp(InputArray src, OutputArraydst, const Size&
dstsize=Size(), int borderType=BORDER_DEFAULT )
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。
- 第二个参数，OutputArray 类型的 dst，输出图像，和源图片有一样的尺寸和类型。
- 第三个参数，const Size&类型的 dstsize，输出图像的大小;有默认值 Size()，即默认情况下，由 Size（src.cols*2，src.rows*2）来进行计算，且一直需要满足下列条件：

$$|\text{dstsize.width}-src.cols*2| \leqslant (\text{dstsize.width mod2})$$

$$|\text{dstsize.height}-src.rows*2| \leqslant (\text{dstsize.heiht mod2})$$

- 第四个参数，int 类型的 borderType，边界模式，一般不用去管它。

pyrUp 函数执行高斯金字塔的采样操作，其实它也可以用于拉普拉斯金字塔的。

首先，它通过插入可为零的行与列，对源图像进行向上取样操作，然后将结果与 pyrDown()乘以 4 的内核做卷积。

下面直接看完整的示例程序。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;


//---------------------------【main()函数】---------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------
int main()
{
    //载入原始图
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材
图
    Mat tmpImage,dstImage;//临时变量和目标图的定义
    tmpImage=srcImage;//将原始图赋给临时变量

    //显示原始图
    imshow("【原始图】", srcImage);
    //进行向上取样操作
    pyrUp(tmpImage, dstImage, Size(tmpImage.cols*2,
tmpImage.rows*2 ) );
    //显示效果图
```

```
    imshow("【效果图】", dstImage);

    waitKey(0);

    return 0;
}
```

程序运行截图如图 6.63 和图 6.64 所示。

图 6.63 原始图

图 6.64 上取样效果图

2. 采样：pyrDown()函数

pyrDown()函数的作用是向下采样并模糊一张图片，说白了就是缩小一张图片。

```
C++: void pyrDown(InputArray src,OutputArray dst, const Size&
dstsize=Size(), int borderType=BORDER_DEFAULT)
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。
- 第二个参数，OutputArray 类型的 dst，输出图像，和源图片有一样的尺寸和类型。
- 第三个参数，const Size&类型的 dstsize，输出图像的大小;有默认值 Size()，即默认情况下，由 Size Size((src.cols+1)/2，(src.rows+1)/2)来进行计算，且一直需要满足下列条件：

$$|\text{dstsize.width}*2-src.cols| \leqslant 2$$

$$|\text{dstsize.heigth}*2-src.rows| \leqslant 2$$

该 pyrDown 函数执行了高斯金字塔建造的向下采样的步骤。首先，它将源图像与如下内核做卷积运算：

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}$$

接着，它通过对图像的偶数行和列做插值来进行向下采样操作。

依然是看看完整的示例程序，如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;

//----------------------------【main()函数】----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //载入原始图
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为1.jpg的素材
图
    Mat tmpImage,dstImage;//临时变量和目标图的定义
    tmpImage=srcImage;//将原始图赋给临时变量

    //显示原始图
    imshow("【原始图】", srcImage);
    //进行向下取样操作
    pyrDown( tmpImage, dstImage, Size( tmpImage.cols/2,
tmpImage.rows/2 ) );
    //显示效果图
    imshow("【效果图】", dstImage);

    waitKey(0);

    return 0;
}
```

程序运行截图如图 6.65 和图 6.66 所示。

图 6.65 原始图

图 6.66 向下取样效果图

6.6.7 综合示例：图像金字塔与图片尺寸缩放

这个示例程序中，分别演示了用 resize，pryUp，pryDown 来让源图像进行放大缩小的操作，分别用键盘按键 1、2、3、4、A、D、W、S 来控制图片的放大与缩小。如图 6.67 所示。

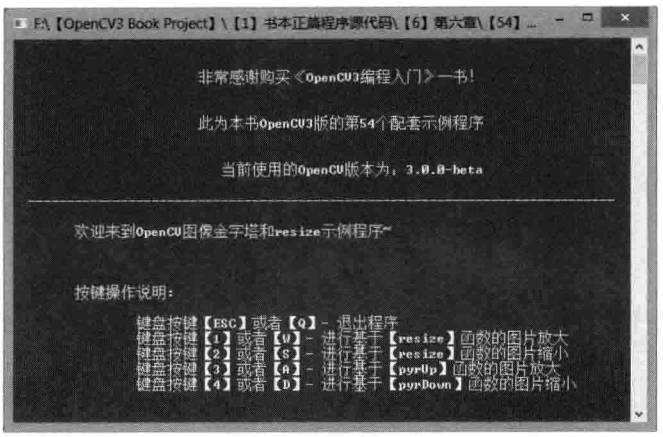


图 6.67 程序操作说明

详细注释的代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace std;
using namespace cv;


//-----------------------------【宏定义部分】-----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME "【程序窗口】"        //为窗口标题定义的宏



//-------------------------【全局变量声明部分】-------------------------
//      描述：全局变量声明
//-------------------------------------------------------------------------
Mat g_srcImage, g_dstImage, g_tmpImage;


//-----------------------------【main()函数】-----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
```

```
    //载入原图
    g_srcImage = imread("1.jpg");//工程目录下需要有一张名为 1.jpg 的测试图像，
且其尺寸需被 2 的 N 次方整除，N 为可以缩放的次数
    if( !g_srcImage.data ) { printf("读取 srcImage 错误～！ \n"); return
false; }

    // 创建显示窗口
    namedWindow( WINDOW_NAME, WINDOW_AUTOSIZE );
    imshow(WINDOW_NAME, g_srcImage);

    //参数赋值
    g_tmpImage = g_srcImage;
    g_dstImage = g_tmpImage;

     int key =0;

    //轮询获取按键信息
    while(1)
    {
        key=waitKey(9) ;//读取键值到 key 变量中

        //根据 key 变量的值，进行不同的操作
        switch(key)
        {
         //====================【程序退出相关键值处理】====================
        case 27://按键 ESC
            return 0;
            break;

        case 'q'://按键 Q
            return 0;
            break;

         //====================【图片放大相关键值处理】====================
        case 'a'://按键 A 按下，调用 pyrUp 函数
            pyrUp( g_tmpImage, g_dstImage, Size( g_tmpImage.cols*2,
g_tmpImage.rows*2 ) );
            printf( ">检测到按键【A】被按下，开始进行基于【pyrUp】函数的图片放
大：图片尺寸×2 \n" );
            break;

        case 'w'://按键 W 按下，调用 resize 函数
            resize(g_tmpImage,g_dstImage,Size( g_tmpImage.cols*2,
g_tmpImage.rows*2 ));
            printf( ">检测到按键【W】被按下，开始进行基于【resize】函数的图片放
大：图片尺寸×2 \n" );
            break;

        case '1'://按键 1 按下，调用 resize 函数
            resize(g_tmpImage,g_dstImage,Size( g_tmpImage.cols*2,
g_tmpImage.rows*2 ));
```

```
            printf( ">检测到按键【1】被按下，开始进行基于【resize】函数的图片放
大：图片尺寸×2 \n" );
            break;

        case '3': //按键3按下，调用pyrUp函数
            pyrUp( g_tmpImage, g_dstImage, Size( g_tmpImage.cols*2,
g_tmpImage.rows*2 ));
            printf( ">检测到按键【3】被按下，开始进行基于【pyrUp】函数的图片放
大：图片尺寸×2 \n" );
            break;
        //=====================【图片缩小相关键值处理】====================
        case 'd': //按键D按下，调用pyrDown函数
            pyrDown( g_tmpImage, g_dstImage, Size( g_tmpImage.cols/2,
g_tmpImage.rows/2 ));
            printf( ">检测到按键【D】被按下，开始进行基于【pyrDown】函数的图片
缩小：图片尺寸/2\n" );
            break;

        case  's' : //按键S按下，调用resize函数
            resize(g_tmpImage,g_dstImage,Size( g_tmpImage.cols/2,
g_tmpImage.rows/2 ));
            printf( ">检测到按键【S】被按下，开始进行基于【resize】函数的图片缩
小：图片尺寸/2\n" );
            break;

        case '2'://按键2按下，调用resize函数
            resize(g_tmpImage,g_dstImage,Size( g_tmpImage.cols/2,
g_tmpImage.rows/2 ),(0,0),(0,0),2);
            printf( ">检测到按键【2】被按下，开始进行基于【resize】函数的图片缩
小：图片尺寸/2\n" );
            break;

        case '4': //按键4按下，调用pyrDown函数
            pyrDown( g_tmpImage, g_dstImage, Size( g_tmpImage.cols/2,
g_tmpImage.rows/2 ) );
            printf( ">检测到按键【4】被按下，开始进行基于【pyrDown】函数的图片
缩小：图片尺寸/2\n" );
            break;
        }

        //经过操作后，显示变化后的图
        imshow( WINDOW_NAME, g_dstImage );

        //将g_dstImage赋给g_tmpImage，方便下一次循环
        g_tmpImage = g_dstImage;
    }

    return 0;
}
```

我们用键盘按键1、2、3、4、A、D、W、S来控制图片的放大与缩小，得到大小各异的图片窗口，并被记录在控制台窗口中。原图和效果图如图6.68和图6.69

所示。

图 6.68 原始图

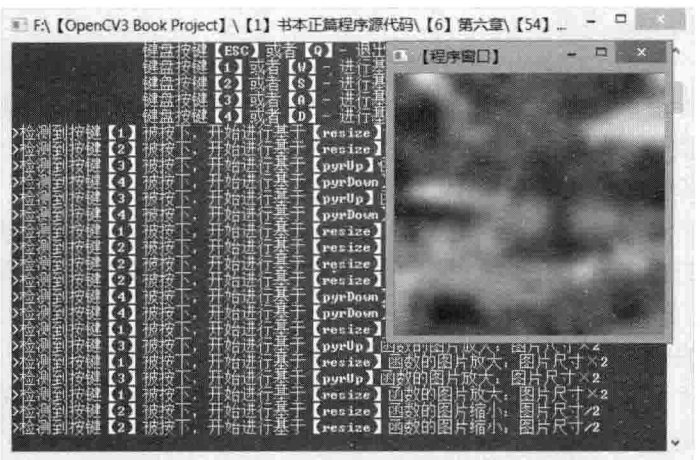


图 6.69 经过多次按键后的效果图

6.7 阈值化

在对各种图形进行处理操作的过程中，我们常常需要对图像中的像素做出取舍与决策，直接剔除一些低于或者高于一定值的像素。

阈值可以被视作最简单的图像分割方法。比如，从一副图像中利用阈值分割出我们需要的物体部分（当然这里的物体可以是一部分或者整体）。这样的图像分割方法基于图像中物体与背景之间的灰度差异，而且此分割属于像素级的分割。为了从一副图像中提取出我们需要的部分，应该用图像中的每一个像素点的灰度值与选取的阈值进行比较，并作出相应的判断。注意：阈值的选取依赖于具体的问题。即物体在不同的图像中有可能会有不同的灰度值。

一旦找到了需要分割的物体的像素点，可以对这些像素点设定一些特定的值来表示。例如，可以将该物体的像素点的灰度值设定为“0”（黑色），其他的像素点的灰度值为“255”（白色）。当然像素点的灰度值可以任意，但最好设定的两种颜色对比度较强，以方便观察结果。

在 OpenCV 2.X 中，Threshold()函数（基本阈值操作）和 adaptiveThreshold()函数（自适应阈值操作）可以完成这样的要求。它们的基本思想是：给定一个数组和一个阈值，然后根据数组中的每个元素的值是高于还是低于阈值而进行一些处理。下面，我们将对这两个函数分别进行剖析。

6.7.1 固定阈值操作：Threshold()函数

函数 Threshold()对单通道数组应用固定阈值操作。该函数的典型应用是对灰度图像进行阈值操作得到二值图像，（compare()函数也可以达到此目的）或者是去掉噪声，例如过滤很小或很大象素值的图像点。

```
C++: double threshold(InputArray src,OutputArray dst, double thresh,
double maxval, int type)
```

- 第一个参数，InputArray 类型的 src，输入数组，填单通道，8 或 32 位浮点类型的 Mat 即可。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，即这个参数用于存放输出结果，且和第一个参数中的 Mat 变量有一样的尺寸和类型。
- 第三个参数，double 类型的 thresh，阈值的具体值。
- 第四个参数，double 类型的 maxval，当第五个参数阈值类型 type 取 CV_THRESH_ BINARY 或 CV_THRESH_BINARY_INV 时阈值类型时的最大值（对应地，OpenCV2 中可以为 CV_THRESH_ BINARY 和 CV_THRESH_BINARY_INV）。
- 第五个参数，int 类型的 type，阈值类型。threshold()函数支持的对图像取阈值的方法由其确定，具体用法如图 6.70。

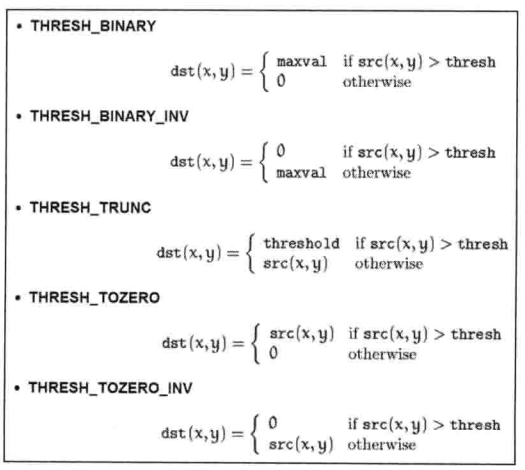


图 6.70 Threshold()函数中阈值类型选项对应的操作

上述标识符依次取值分别为 0，1，2，3，4。

而针对上述公式，一一对应的图形化的阈值描述如图 6.71 所示（THRESH_BINARY 对应二进制阈值，依次类推）。

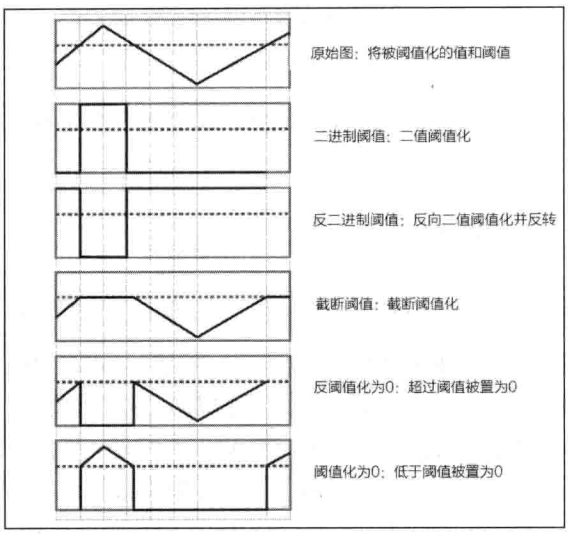


图 6.71 Threshold()函数中不同阈值类型的操作结果

6.7.2 自适应阈值操作：adaptiveThreshold()函数

adaptiveThreshold()函数的作用是对矩阵采用自适应阈值操作，支持就地操作。函数原型如下。

```
C++: void adaptiveThreshold(InputArray src, OutputArray dst, double
maxValue, int adaptiveMethod, int thresholdType, int blockSize, double
C)
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，且需为 8 位单通道浮点型图像。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，需和源图片有一样的尺寸和类型。
- 第三个参数，double 类型的 maxValue，给像素赋的满足条件的非零值。具体看下面的讲解。
- 第四个参数，int 类型的 adaptiveMethod，用于指定要使用的自适应阈值算法，可取值为 ADAPTIVE_ THRESH_MEAN_C 或 ADAPTIVE_THRESH_ GAUSSIAN_C。
- 第五个参数，int 类型的 thresholdType，阈值类型。取值必须为 THRESH_BINARY、THRESH_BINARY_INV 其中之一。
- 第六个参数，int 类型的 blockSize，用于计算阈值大小的一个像素的邻域尺寸，取值为 3、5、7 等。
- 第七个参数，double 类型的 C，减去平均或加权平均值后的常数值。通常其为正数，但少数情况下也可以为零或负数。

adaptiveThreshold()函数根据如下公式，将一幅灰度图变换为一幅二值图。

当第五个参数“阈值类型”thresholdType 取值为 THRESH_BINARY 时，公式如下。

$$dst(x,y)=\begin{cases}maxValue & if\ src(x,y)>T(x,y)\\ 0 & otherwise\end{cases}$$

当第五个参数“阈值类型”thresholdType 取值为 THRESH_BINARY_INV 时，公式为：

$$dst(x,y)=\begin{cases}0 & if\ src(x,y)>T(x,y)\\ maxValue & otherwise\end{cases}$$

而其中的 T(x,y)为分别计算每个单独像素的阈值，取值如下。

- 对于 ADAPTIVE_THRESH_MEAN_C 方法，阈值 T(x,y)为 blockSize×blockSize 邻域内（x，y）减去第七个参数 C 的平均值。
- 对于 ADAPTIVE_THRESH_GAUSSIAN_C 方法，阈值 T(x,y)为 blockSize×blockSize 邻域内（x,y）减去第七个参数 C 与高斯窗交叉相关((cross-correlation with a Gaussian window))的加权总和。

6.7.3 示例程序：基本阈值操作

讲解完这个函数，让我们看一个调用示例程序，这个示例程序演示了基本阈值操作的方方面面。此程序可以通过按键，在不同的阈值模式之间切换，操作说明如图 6.72 所示。

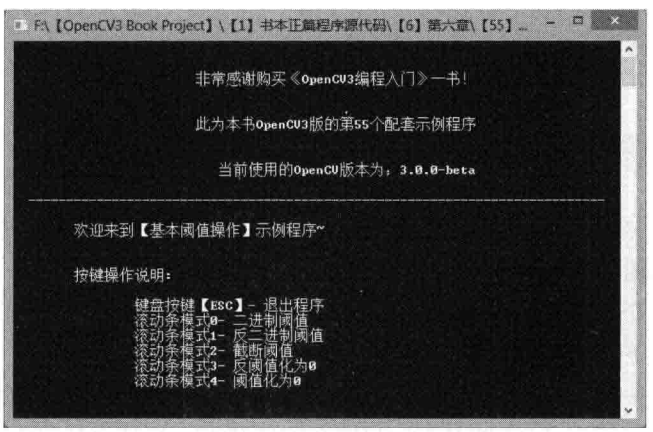


图 6.72 程序操作说明

其经过详细注释的示例程序如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
```

```
#include <iostream>
using namespace cv;
using namespace std;

//-----------------------------【宏定义部分】-----------------------------
//      描述：定义一些辅助宏
//------------------------------------------------------------------------
#define WINDOW_NAME "【程序窗口】"        //为窗口标题定义的宏


//-------------------------【全局变量声明部分】-------------------------
//      描述：全局变量的声明
//------------------------------------------------------------------------
int g_nThresholdValue = 100;
int g_nThresholdType = 3;
Mat g_srcImage, g_grayImage, g_dstImage;

//-------------------------【全局函数声明部分】-------------------------
//      描述：全局函数的声明
//------------------------------------------------------------------------
static void ShowHelpText();//输出帮助文字
void on_Threshold( int, void* );//回调函数


//-----------------------------【main()函数】-----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//------------------------------------------------------------------------
int main()
{
    //【1】读入源图片
    g_srcImage = imread("1.jpg");
    if(!g_srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定的图片存在~！ \n"); return false; }

    //【2】存留一份原图的灰度图
    cvtColor( g_srcImage, g_grayImage, COLOR_RGB2GRAY );

    //【3】创建窗口并显示原始图
    namedWindow( WINDOW_NAME, WINDOW_AUTOSIZE );

    //【4】创建滑动条来控制阈值
    createTrackbar( "模式",
                WINDOW_NAME, &g_nThresholdType,
                4, on_Threshold );

    createTrackbar( "参数值",
                WINDOW_NAME, &g_nThresholdValue,
                255, on_Threshold );

    //【5】初始化自定义的阈值回调函数
    on_Threshold( 0, 0 );
```

```
    // 【6】轮询等待用户按键，如果ESC键按下则退出程序
    while(1)
     {
         int key;
         key = waitKey( 20 );
         if( (char)key == 27 ){ break; }
     }

}

//-----------------------【on_Threshold()函数】-----------------------
//      描述：自定义的阈值回调函数
//-----------------------------------------------------------------------
void on_Threshold( int, void* )
{
    //调用阈值函数

threshold(g_grayImage,g_dstImage,g_nThresholdValue,255,g_nThresholdT
ype);

  //更新效果图
  imshow( WINDOW_NAME, g_dstImage );
}
```

运行此程序，我们得到带有滑动条的窗口，通过调节滑动条，可以得到不同的阈值效果。原始图如图 6.73 所示，效果图如图 6.74～6.78 所示。

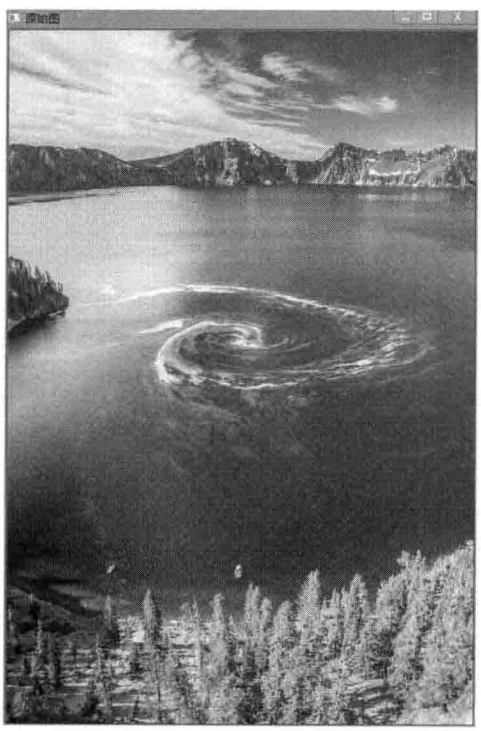


图 6.73 原始图

图 6.74 模式 0（二进制阈值）效果图

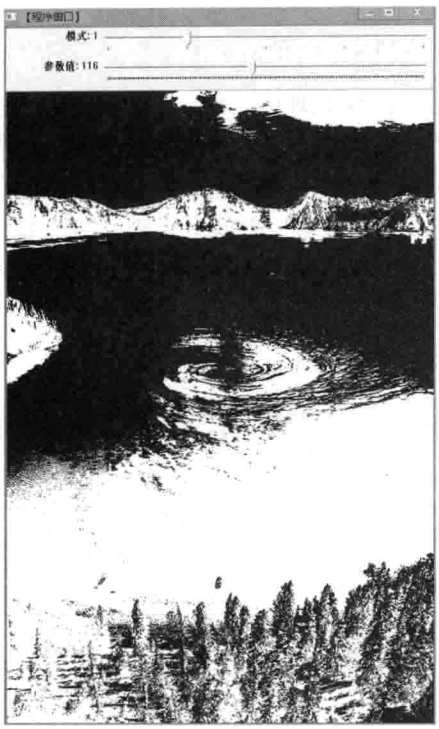


图 6.75 模式 1（反二进制阈值）效果图

图 6.76 模式 2（截断阈值）效果图

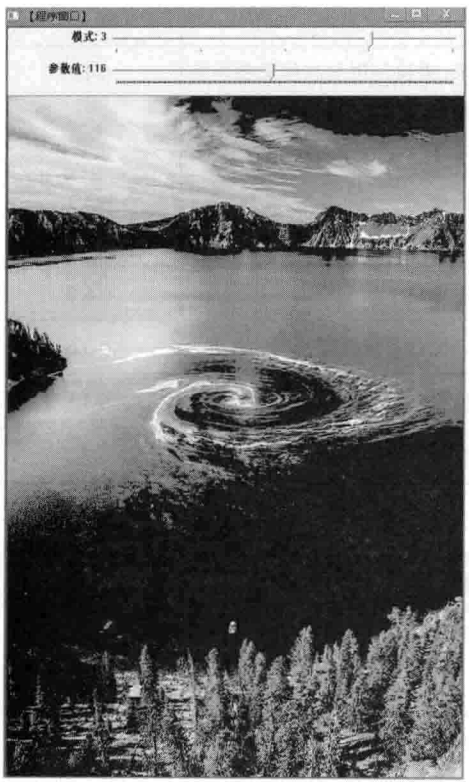


图 6.77 模式 3（ 反阈值化为 0）效果图

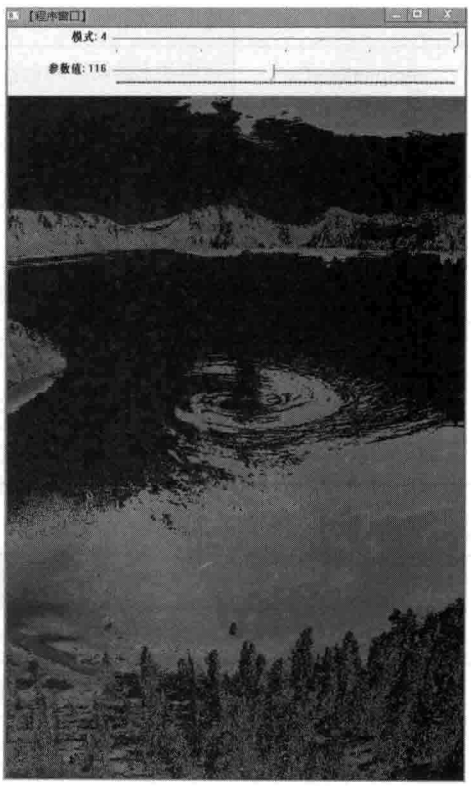


图 6.78 模式 4（阈值化为 0）效果图

6.8 本章小结

本章中我们学习了各种利用 OpenCV 进行图像处理的方法。包括属于线性滤波的方框滤波、均值滤波与高斯滤波，属于非线性滤波的中值滤波、双边滤波；两种基本形态学操作——膨胀与腐蚀；5 种高级形态学滤波操作——开运算、闭运算、形态学梯度、顶帽以及黑帽；还有漫水填充算法、图像金字塔、图像缩放、阈值化。涉及到的内容可谓非常丰富。相信学完此章的内容，你对 OpenCV 的了解已经上了一个层次。

本章核心函数清单

函数名称	说明	对应讲解章节
boxFilter	使用方框滤波来模糊一张图片	6.1.11
blur	对输入的图像进行均值滤波操作	6.1.11
GaussianBlur	用高斯滤波器来模糊一张图片	6.1.11
medianBlur	使用中值滤波器来模糊一张图片	6.2.4
bilateralFilter	用双边滤波器来模糊处理一张图片	6.2.4
dilate	使用像素邻域内的局部极大运算符来膨胀一张图片	6.3.5
erode	使用像素邻域内的局部极小运算符来腐蚀一张图片	6.3.5
morphologyEx	利用基本的膨胀和腐蚀技术，来执行更加高级形态学变换，如开闭运算、形态学梯度、顶帽、黑帽等，也可以实现最基本的图像膨胀和腐蚀	6.4.7
floodFill	用指定的颜色从种子点开始填充一个连接域，实现漫水填充算法	6.5.3
pyrUp	向上采样并模糊一张图片，说白了就是放大一张图片	6.6.6
pyrDown	向下采样并模糊一张图片，说白了就是缩小一张图片	6.6.6
Threshold	对单通道数组应用固定阈值操作	6.7.1
adaptiveThreshold	对矩阵采用自适应阈值操作	6.7.2

本章示例程序清单

示例程序序号	程序说明	对应章节
31	方框滤波：boxFilter 函数的使用	6.1.11
32	均值滤波：blur 函数的使用	6.1.11
33	高斯滤波：GaussianBlur 函数的使用	6.1.11
34	综合示例：图像线性滤波	6.1.12
35	中值滤波：medianBlur 函数的使用	6.2.4

续表

示例程序序号	程序说明	对应章节
36	双边滤波：bilateralFilter 函数的使用	6.2.4
37	综合示例：图像滤波	6.2.5
38	膨胀：dilate 函数的使用	6.3.5
39	腐蚀：erode 函数的使用	6.3.5
40	综合示例：腐蚀与膨胀	6.3.6
41	用 morphologyEx()函数实现形态学膨胀	6.4.8
42	用 morphologyEx()函数实现形态学腐蚀	6.4.8
43	用 morphologyEx()函数实现形态学开运算	6.4.8
44	用 morphologyEx()函数实现形态学闭运算	6.4.8
45	用 morphologyEx()函数实现形态学梯度	6.4.8
46	用 morphologyEx()函数实现形态学“顶帽”	6.4.8
47	用 morphologyEx()函数实现形态学“黑帽”	6.4.8
48	综合示例：形态学滤波	6.4.9
49	漫水填充算法：floodFill 函数	6.5.3
50	综合示例：漫水填充	6.5.4
51	尺寸调整：resize()函数的使用	6.6.5
52	向上采样图像金字塔：pyrUp()函数的使用	6.6.6
53	向下采样图像金字塔：pyrDown()函数的使用	6.6.6
54	综合示例：图像金字塔与图片尺寸缩放	6.6.7
55	示例程序：基本阈值操作	6.7.3

第 7 章
图像变换

导读

上一章中我们讲解了许多类型的图像处理方法。而迄今为止，所介绍的大多数操作都用于增强、修改或者“处理”图像，使之成为类似但全新的图像。

本章关注的重点是图像变换（image transfrom），即将一幅图像转变成图像数据的另一种表现形式。变换最常见的例子就是傅里叶变换（Fourier transform），即将图像转换成源图像数据的另一种表示形式。这类操作的结果仍然保存为 OpenCV 图像结构的形式，但是新图像的每个单独像素表示原始输出图像的频谱分量，而不是通常所考虑的空间分量。

其实，计算机视觉中经常会用到许多有用的变换。OpenCV 提供了一套完整的实现工具和方法，可以帮助我们实现各种图像变换操作。

本章中，你将学到：

- 基于 OpenCV 的边缘检测
- 霍夫变换
- 重映射
- 仿射变换
- 直方图均衡化

7.1 基于 OpenCV 的边缘检测

本节中，我们将一起学习 OpenCV 中边缘检测的各种算子和滤波器——Canny 算子、Sobel 算子、Laplacian 算子以及 Scharr 滤波器。

7.1.1 边缘检测的一般步骤

在具体介绍之前，先来一起看看边缘检测的一般步骤。

1.【第一步】滤波

边缘检测的算法主要是基于图像强度的一阶和二阶导数，但导数通常对噪声很敏感，因此必须采用滤波器来改善与噪声有关的边缘检测器的性能。常见的滤波方法主要有高斯滤波，即采用离散化的高斯函数产生一组归一化的高斯核，然后基于高斯核函数对图像灰度矩阵的每一点进行加权求和。

2.【第二步】增强

增强边缘的基础是确定图像各点邻域强度的变化值。增强算法可以将图像灰度点邻域强度值有显著变化的点凸显出来。在具体编程实现时，可通过计算梯度幅值来确定。

3.【第三步】检测

经过增强的图像，往往邻域中有很多点的梯度值比较大，而在特定的应用中，这些点并不是要找的边缘点，所以应该采用某种方法来对这些点进行取舍。实际工程中，常用的方法是通过阈值化方法来检测。

另外，需要注意，下文中讲到的 Laplacian 算子、sobel 算子和 Scharr 算子都是带方向的，所以，示例中我们分别写了 X 方向、Y 方向和最终合成的的效果图。

7.1.2 canny 算子

1. canny 算子简介

Canny 边缘检测算子是 John F.Canny 于 1986 年开发出来的一个多级边缘检测算法。更为重要的是，Canny 创立了边缘检测计算理论（Computational theory ofedge detection），解释了这项技术是如何工作的。Canny 边缘检测算法以 Canny 的名字命名，被很多人推崇为当今最优的边缘检测的算法。

其中，Canny 的目标是找到一个最优的边缘检测算法，让我们看一下最优边缘检测的三个主要评价标准。

- 低错误率：标识出尽可能多的实际边缘，同时尽可能地减少噪声产生的误报。
- 高定位性：标识出的边缘要与图像中的实际边缘尽可能接近。

- 最小响应：图像中的边缘只能标识一次，并且可能存在的图像噪声不应标识为边缘。

为了满足这些要求，Canny 使用了变分法，这是一种寻找满足特定功能的函数的方法。最优检测用 4 个指数函数项的和表示，但是它非常近似于高斯函数的一阶导数。

2. Canny 边缘检测的步骤

（1）【第一步】消除噪声

一般情况下，使用高斯平滑滤波器卷积降噪。以下显示了一个 size = 5 的高斯内核示例：

$$K=\frac{1}{139}\begin{bmatrix}2 & 4 & 5 & 4 & 2\\ 4 & 9 & 12 & 9 & 4\\ 5 & 12 & 15 & 12 & 5\\ 4 & 9 & 12 & 9 & 4\\ 2 & 4 & 5 & 4 & 2\end{bmatrix}$$

（2）【第二步】计算梯度幅值和方向

此处，按照 Sobel 滤波器的步骤来操作。

① 运用一对卷积阵列（分别作用于 x 和 y 方向）

$$G_r=\begin{bmatrix}-1 & 0 & +1\\ -2 & 0 & +2\\ -1 & 0 & +1\end{bmatrix}\quad G_y=\begin{bmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ +1 & +2 & +1\end{bmatrix}$$

② 使用下列公式计算梯度幅值和方向

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

$$\theta=\arctan\left(\frac{G_y}{G_x}\right)$$

而梯度方向一般取这 4 个可能的角度之一——0 度，45 度，90 度，135 度。

（3）【第三步】非极大值抑制

这一步排除非边缘像素，仅仅保留了一些细线条（候选边缘）。

（4）【第四步】滞后阈值

这是最后一步，Canny 使用了滞后阈值，滞后阈值需要两个阈值（高阈值和低阈值）：

① 若某一像素位置的幅值超过高阈值，该像素被保留为边缘像素。

② 若某一像素位置的幅值小于低阈值，该像素被排除。

③ 若某一像素位置的幅值在两个阈值之间，该像素仅仅在连接到一个高于高阈值的像素时被保留。

对于 Canny 函数的使用，推荐的高低阈值比在 2:1 到 3:1 之间。

3. Canny 边缘检测：Canny()函数

Canny 函数利用 Canny 算子来进行图像的边缘检测操作。

```
C++: void Canny(InputArray image,OutputArray edges, double threshold1,
double threshold2, int apertureSize=3,bool L2gradient=false )
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像，填 Mat 类的对象即可，且需为单通道 8 位图像。
- 第二个参数，OutputArray 类型的 edges，输出的边缘图，需要和源图片有一样的尺寸和类型。
- 第三个参数，double 类型的 threshold1，第一个滞后性阈值。
- 第四个参数，double 类型的 threshold2，第二个滞后性阈值。
- 第五个参数，int 类型的 apertureSize，表示应用 Sobel 算子的孔径大小，其有默认值 3。
- 第六个参数，bool 类型的 L2gradient，一个计算图像梯度幅值的标识，有默认值 false。

需要注意的是，这个函数阈值 1 和阈值 2 两者中较小的值用于边缘连接，而较大的值用来控制强边缘的初始段，推荐的高低阈值比在 2:1 到 3:1 之间。

调用示例如下。

```
//载入原始图
    Mat src = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材图
    Canny(src, src, 3, 9,3 );
    imshow("【效果图】Canny 边缘检测", src);
```

4. 示例程序：Canny 边缘检测

OpenCV 中调用 Canny 函数的实例代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
using namespace cv;


//----------------------------【main()函数】----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
   //载入原始图
```

```
    Mat src = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材图
    Mat src1=src.clone();

    //显示原始图
    imshow("【原始图】Canny 边缘检测", src);

    //-------------------------------------------------------------------
//  一、最简单的 canny 用法，拿到原图后直接用。
//  注意：此方法在 OpenCV2 中可用，在 OpenCV3 中已失效
//-------------------------------------------------------------------
//Canny( srcImage, srcImage, 150, 100,3 );
//imshow("【效果图】Canny 边缘检测", srcImage);



//-------------------------------------------------------------------
    //  二、高阶的 canny 用法，转成灰度图，降噪，用 canny，最后将得到的边缘作为掩
码，拷贝原图到效果图上，得到彩色的边缘图

//-------------------------------------------------------------------
    Mat dst,edge,gray;

    // 【1】创建与 src 同类型和大小的矩阵(dst)
    dst.create( src1.size(), src1.type() );

    // 【2】将原图像转换为灰度图像
    cvtColor( src1, gray, COLOR_BGR2GRAY );

    // 【3】先用使用 3x3 内核来降噪
    blur( gray, edge, Size(3,3) );

    // 【4】运行 Canny 算子
    Canny( edge, edge, 3, 9,3 );

    //【5】将 g_dstImage 内的所有元素设置为 0
    dst = Scalar::all(0);

    //【6】使用 Canny 算子输出的边缘图 g_cannyDetectedEdges 作为掩码，来将原图
g_srcImage 拷到目标图 g_dstImage 中
    src1.copyTo( dst, edge);

    //【7】显示效果图
    imshow("【效果图】Canny 边缘检测 2", dst);


    waitKey(0);

    return 0;
}
```

原始图和运行效果图如图 7.1、7.2、7.3 所示。

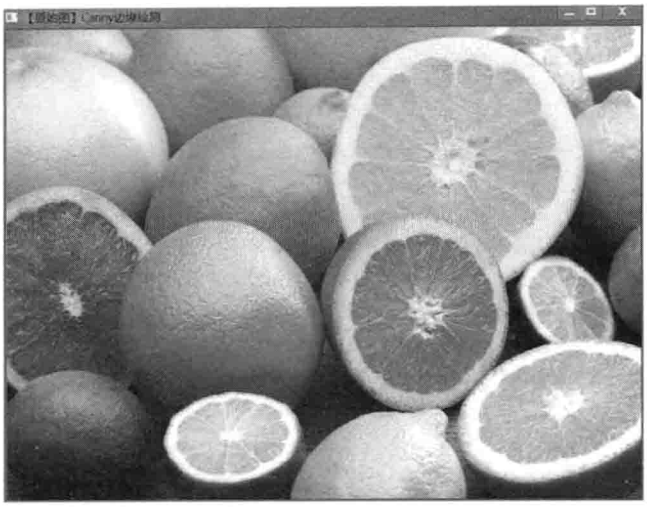


图 7.1 原始图

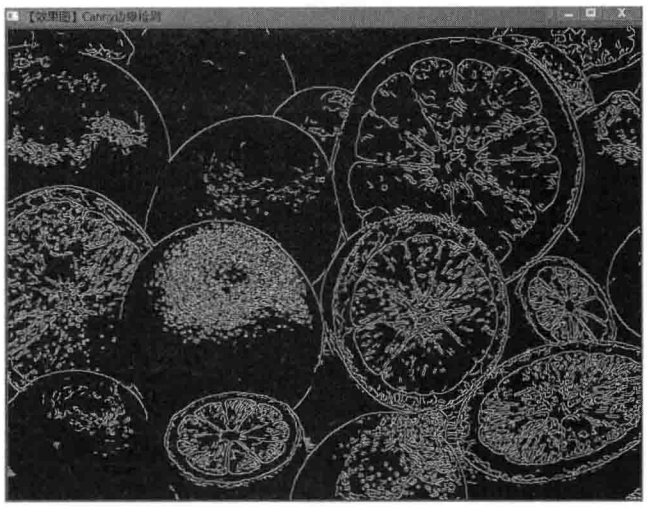


图 7.2 灰度 canny 检测图

图 7.3 彩色 canny 检测图

7.1.3 sobel 算子

1. sobel 算子的基本概念

Sobel 算子是一个主要用于边缘检测的离散微分算子（discrete differentiation operator）。它结合了高斯平滑和微分求导，用来计算图像灰度函数的近似梯度。在图像的任何一点使用此算子，都将会产生对应的梯度矢量或是其法矢量。

2. sobel 算子的计算过程

我们假设被作用图像为 I 然后进行如下操作。

（1）分别在 x 和 y 两个方向求导。

① 水平变化：将 I 与一个奇数大小的内核 G_x 进行卷积。比如，当内核大小为 3 时，G_x 的计算结果为：

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

② 垂直变化：将：I 与一个奇数大小的内核进行卷积。比如，当内核大小为 3 时，计算结果为：

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

（2）在图像的每一点，结合以上两个结果求出近似梯度:

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

另外有时，也可用下面更简单的公式代替:

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

3. 使用 Sobel 算子：Sobel()函数

Sobel 函数使用扩展的 Sobel 算子，来计算一阶、二阶、三阶或混合图像差分。

```
C++: 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 );
```

（1）第一个参数，InputArray 类型的 src，为输入图像，填 Mat 类型即可。

（2）第二个参数，OutputArray 类型的 dst，即目标图像，函数的输出参数，需要和源图片有一样的尺寸和类型。

（3）第三个参数，int 类型的 ddepth，输出图像的深度，支持如下 src.depth() 和 ddepth 的组合：

- 若 src.depth() = CV_8U, 取 ddepth =-1/CV_16S/CV_32F/CV_64F
- 若 src.depth() = CV_16U/CV_16S, 取 ddepth =-1/CV_32F/CV_64F
- 若 src.depth() = CV_32F, 取 ddepth =-1/CV_32F/CV_64F
- 若 src.depth() = CV_64F, 取 ddepth = -1/CV_64F

（4）第四个参数，int 类型 dx，x 方向上的差分阶数。

（5）第五个参数，int 类型 dy，y 方向上的差分阶数。

（6）第六个参数，int 类型 ksize，有默认值 3，表示 Sobel 核的大小;必须取 1、3、5 或 7。

（7）第七个参数，double 类型的 scale，计算导数值时可选的缩放因子，默认值是 1，表示默认情况下是没有应用缩放的。可以在文档中查阅 getDerivKernels 的相关介绍，来得到这个参数的更多信息。

（8）第八个参数，double 类型的 delta，表示在结果存入目标图（第二个参数 dst）之前可选的 delta 值，有默认值 0。

（9）第九个参数，int 类型的 borderType，边界模式，默认值为 BORDER_DEFAULT。这个参数可以在官方文档中 borderInterpolate 处得到更详细的信息。

一般情况下，都是用 ksize × ksize 内核来计算导数的。然而，有一种特殊情况——当 ksize 为 1 时，往往会使用 3 x 1 或者 1 x 3 的内核。且这种情况下，并没有进行高斯平滑操作。

一些补充说明如下。

① 当内核大小为 3 时，Sobel 内核可能产生比较明显的误差（毕竟，Sobel 算子只是求取了导数的近似值而已）。为解决这一问题，OpenCV 提供了 Scharr 函数，但该函数仅作用于大小为 3 的内核。该函数的运算与 Sobel 函数一样快，但结果却更加精确，其内核是这样的：

$$G_x=\begin{bmatrix}-3 & 0 & +3\\ -10 & 0 & +10\\ -3 & 0 & +3\end{bmatrix}\quad G_y=\begin{bmatrix}-3 & -10 & -3\\ 0 & 0 & 0\\ +3 & +10 & +3\end{bmatrix}$$

② 因为 Sobel 算子结合了高斯平滑和分化（differentiation），因此结果会具有更多的抗噪性。大多数情况下，我们使用 sobel 函数时，取【xorder = 1，yorder = 0，ksize = 3】来计算图像 X 方向的导数，【xorder = 0，yorder = 1，ksize = 3】来计算图像 y 方向的导数。

计算图像 X 方向的导数，取【xorder= 1，yorder = 0，ksize = 3】情况对应的内核：

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 3 \end{bmatrix}$$

而计算图像 Y 方向的导数，取【xorder= 0，yorder = 1，ksize = 3】对应的内核：

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

4. 示例程序：Sobel 算子的使用

调用 Sobel 函数的实例代码如下。这里只是教大家如何使用 Sobel 函数，就没有先用一句 cvtColor 将原图转化为灰度图，而是直接用彩色图操作。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
using namespace cv;

//---------------------------【main()函数】---------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //【0】创建 grad_x 和 grad_y 矩阵
    Mat grad_x, grad_y;
    Mat abs_grad_x, abs_grad_y,dst;

    //【1】载入原始图
    Mat src = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材图

    //【2】显示原始图
    imshow("【原始图】sobel 边缘检测", src);

    //【3】求 X 方向梯度
    Sobel( src, grad_x, CV_16S, 1, 0, 3, 1, 1, BORDER_DEFAULT );
    convertScaleAbs( grad_x, abs_grad_x );
    imshow("【效果图】 X 方向 Sobel", abs_grad_x);

    //【4】求 Y 方向梯度
    Sobel( src, grad_y, CV_16S, 0, 1, 3, 1, 1, BORDER_DEFAULT );
    convertScaleAbs( grad_y, abs_grad_y );
    imshow("【效果图】Y 方向 Sobel", abs_grad_y);

    //【5】合并梯度(近似)
    addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, dst );
```

```
    imshow("【效果图】整体方向 Sobel", dst);

    waitKey(0);
    return 0;
}
```

运行截图如图 7.4 和图 7.5 所示。

图 7.4 原始图和 X 方向上的 sobel

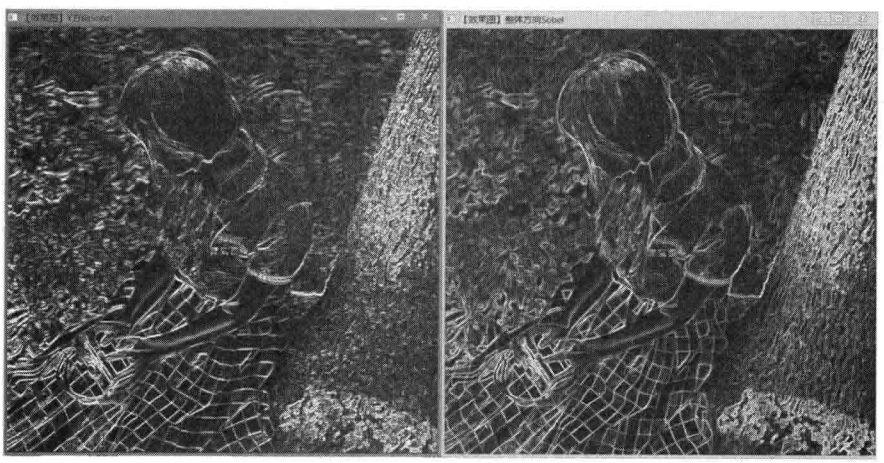


图 7.5 Y 方向上的 sobel 和整体方向上的 sobel

7.1.4 Laplacian 算子

1. Laplacian 算子简介

Laplacian 算子是 n 维欧几里德空间中的一个二阶微分算子，定义为梯度 grad 的散度 div。因此如果 f 是二阶可微的实函数，则 f 的拉普拉斯算子定义如下。

(1) f 的拉普拉斯算子也是笛卡儿坐标系 xi 中的所有非混合二阶偏导数求和。

(2) 作为一个二阶微分算子，拉普拉斯算子把 C 函数映射到 C 函数。对于 k ≥2，表达式(1)（或(2)）定义了一个算子 Δ：C(R)→C(R)；或更一般地，对于任

何开集 Ω，定义了一个算子 Δ：C(Ω) → C(Ω)。

根据图像处理的原理可知，二阶导数可以用来进行检测边缘。因为图像是“二维”，需要在两个方向进行求导。使用 Laplacian 算子将会使求导过程变得简单。

Laplacian 算子的定义:

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

需要说明的是，由于 Laplacian 使用了图像梯度，它内部的代码其实是调用了 Sobel 算子的。

让一幅图像减去它的 Laplacian 算子可以增强对比度。

2. 计算拉普拉斯变换：Laplacian()函数

Laplacian 函数可以计算出图像经过拉普拉斯变换后的结果。

```
C++: void Laplacian(InputArray src,OutputArray dst, int ddepth, int
ksize=1, double scale=1, double delta=0,
intborderType=BORDER_DEFAULT );
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像，填 Mat 类的对象即可，且需为单通道 8 位图像。
- 第二个参数，OutputArray 类型的 edges，输出的边缘图，需要和源图片有一样的尺寸和通道数。
- 第三个参数，int 类型的 ddept，目标图像的深度。
- 第四个参数，int 类型的 ksize，用于计算二阶导数的滤波器的孔径尺寸，大小必须为正奇数，且有默认值 1。
- 第五个参数，double 类型的 scale，计算拉普拉斯值的时候可选的比例因子，有默认值 1。
- 第六个参数，double 类型的 delta，表示在结果存入目标图（第二个参数 dst）之前可选的 delta 值，有默认值 0。
- 第七个参数，int 类型的 borderType，边界模式，默认值为 BORDER_DEFAULT。这个参数可以在官方文档中 borderInterpolate()处得到更详细的信息。

Laplacian()函数其实主要是利用 sobel 算子的运算。它通过加上 sobel 算子运算出的图像 x 方向和 y 方向上的导数，来得到载入图像的拉普拉斯变换结果。

其中，sobel 算子（ksize>1）如下：

$$\mathrm{dst}=\Delta\mathrm{src}=\frac{\partial^2 \mathrm{src}}{\partial x^2}+\frac{\partial^2 \mathrm{src}}{\partial y^2}$$

而当 ksize=1 时，Laplacian()函数采用以下 3x3 的孔径：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

3. 示例程序：Laplacian 算子的使用

让我们看一看调用实例。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
using namespace cv;

//-----------------------------【main()函数】---------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //【0】变量的定义
    Mat src,src_gray,dst, abs_dst;

    //【1】载入原始图
    src = imread("1.jpg");  //工程目录下应该有一张名为1.jpg的素材图

    //【2】显示原始图
    imshow("【原始图】图像Laplace变换", src);

    //【3】使用高斯滤波消除噪声
    GaussianBlur( src, src, Size(3,3), 0, 0, BORDER_DEFAULT );

    //【4】转换为灰度图
    cvtColor( src, src_gray, COLOR_RGB2GRAY );

    //【5】使用Laplace函数
    Laplacian( src_gray, dst, CV_16S, 3, 1, 0, BORDER_DEFAULT );

    //【6】计算绝对值，并将结果转换成8位
    convertScaleAbs( dst, abs_dst );

    //【7】显示效果图
    imshow( "【效果图】图像Laplace变换", abs_dst );

    waitKey(0);

    return 0;
}
```

运行截图如图 7.6 所示。

图 7.6 示例效果图

7.1.5 scharr 滤波器

我们一般直接称 scharr 为滤波器，而不是算子。上文已经讲到，它在 OpenCV 中主要是配合 Sobel 算子的运算而存在的。下面让我们直接来看看其函数讲解。

1. 计算图像差分：Scharr()函数

使用 Scharr 滤波器运算符计算 x 或 y 方向的图像差分。其实它的参数变量和 Sobel 基本上是一样的，除了没有 ksize 核的大小。

```
C++: void Scharr(
InputArray src, //源图
 OutputArray dst, //目标图
 int ddepth,//图像深度
 int dx,// x方向上的差分阶数
 int dy,//y方向上的差分阶数
 double scale=1,//缩放因子
 double delta=0,// delta值
 intborderType=BORDER_DEFAULT )// 边界模式
```

（1）第一个参数，InputArray 类型的 src，为输入图像，填 Mat 类型即可。

（2）第二个参数，OutputArray 类型的 dst，即目标图像，函数的输出参数，需要和源图片有一样的尺寸和类型。

（3）第三个参数，int 类型的 ddepth，输出图像的深度，支持如下 src.depth() 和 ddepth 的组合：

- 若 src.depth() = CV_8U，取 ddepth =-1/CV_16S/CV_32F/CV_64F
- 若 src.depth() = CV_16U/CV_16S，取 ddepth =-1/CV_32F/CV_64F

- 若 src.depth() = CV_32F，取 ddepth =-1/CV_32F/CV_64F
- 若 src.depth() = CV_64F，取 ddepth = -1/CV_64F

（4）第四个参数，int 类型 dx，x 方向上的差分阶数。

（5）第五个参数，int 类型 dy，y 方向上的差分阶数。

（6）第六个参数，double 类型的 scale，计算导数值时可选的缩放因子，默认值是 1，表示默认情况下是没有应用缩放的。我们可以在文档中查阅 getDerivKernels 的相关介绍，来得到这个参数的更多信息。

（7）第七个参数，double 类型的 delta，表示在结果存入目标图（第二个参数 dst）之前可选的 delta 值，有默认值 0。

（8）第八个参数，int 类型的 borderType，边界模式，默认值为 BORDER_DEFAULT。这个参数可以在官方文档中 borderInterpolate 处得到更详细的信息。

不难理解，如下两者是等价的，即：

```
Scharr(src, dst, ddepth, dx, dy, scale,delta, borderType);
```

与

```
Sobel(src, dst, ddepth, dx, dy, CV_SCHARR,scale, delta, borderType);
```

2. 示例程序：Scharr 滤波器

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include<opencv2/highgui/highgui.hpp>
#include<opencv2/imgproc/imgproc.hpp>
using namespace cv;

//---------------------------【main()函数】---------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //【0】创建 grad_x 和 grad_y 矩阵
    Mat grad_x, grad_y;
    Mat abs_grad_x, abs_grad_y,dst;

    //【1】载入原始图
    Mat src = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材图

    //【2】显示原始图
    imshow("【原始图】Scharr 滤波器", src);

    //【3】求 X 方向梯度
    Scharr( src, grad_x, CV_16S, 1, 0, 1, 0, BORDER_DEFAULT );
    convertScaleAbs( grad_x, abs_grad_x );
    imshow("【效果图】 X 方向 Scharr", abs_grad_x);
```

```
    //【4】求 Y 方向梯度
    Scharr( src, grad_y, CV_16S, 0, 1, 1, 0, BORDER_DEFAULT );
    convertScaleAbs( grad_y, abs_grad_y );
    imshow("【效果图】Y 方向 Scharr", abs_grad_y);

    //【5】合并梯度(近似)
    addWeighted( abs_grad_x, 0.5, abs_grad_y, 0.5, 0, dst );

    //【6】显示效果图
    imshow("【效果图】合并梯度后 Scharr", dst);

    waitKey(0);
    return 0;
}
```

原图如图 7.7 所示，运行效果图如图 7.8、图 7.9、图 7.10 所示。

图 7.7 原始图

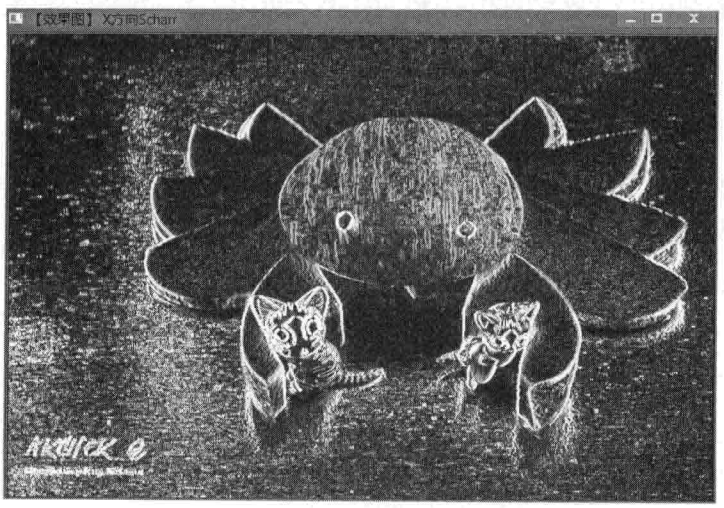


图 7.8 X 方向上的 Scharr

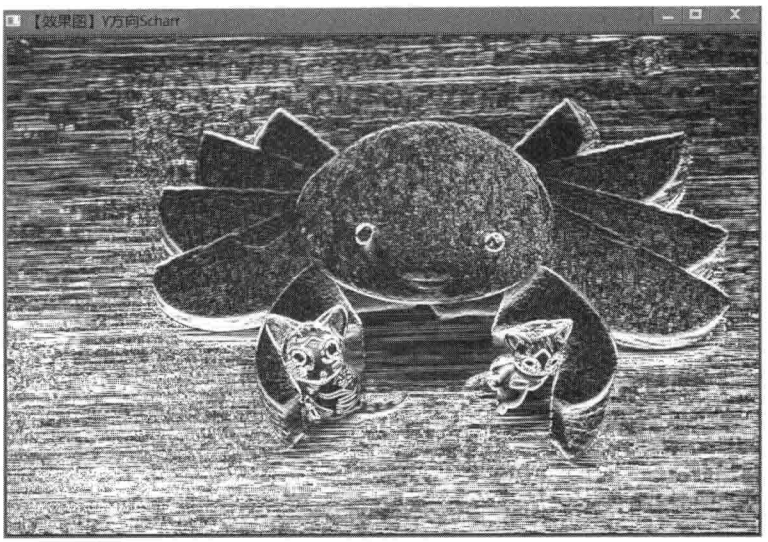


图 7.9 Y 方向上的 Scharr

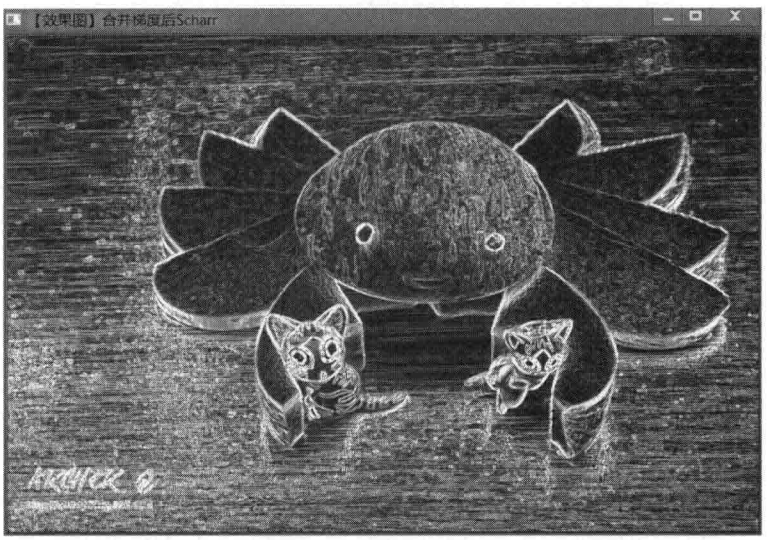


图 7.10 合并梯度后的 Scharr

7.1.6 综合示例：边缘检测

本节依然是配给大家一个详细注释的配套示例程序，把这节中介绍的知识点以代码为载体，更形象地展现出来。

这个示例程序中，分别演示了 canny 边缘检测、sobel 边缘检测、scharr 滤波器的使用，经过详细注释的的代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;
```

```
//-------------------------【全局变量声明部分】-------------------------
//      描述：全局变量声明
//-----------------------------------------------------------------------
//原图，原图的灰度版，目标图
Mat g_srcImage, g_srcGrayImage,g_dstImage;

//Canny 边缘检测相关变量
Mat g_cannyDetectedEdges;
int g_cannyLowThreshold=1;//TrackBar 位置参数

//Sobel 边缘检测相关变量
Mat g_sobelGradient_X, g_sobelGradient_Y;
Mat g_sobelAbsGradient_X, g_sobelAbsGradient_Y;
int g_sobelKernelSize=1;//TrackBar 位置参数

//Scharr 滤波器相关变量
Mat g_scharrGradient_X, g_scharrGradient_Y;
Mat g_scharrAbsGradient_X, g_scharrAbsGradient_Y;


//-------------------------【全局函数声明部分】-------------------------
//      描述：全局函数声明
//-----------------------------------------------------------------------
static void on_Canny(int, void*);//Canny 边缘检测窗口滚动条的回调函数
static void on_Sobel(int, void*);//Sobel 边缘检测窗口滚动条的回调函数
void Scharr();//封装了 Scharr 边缘检测相关代码的函数


//-------------------------【main()函数】-------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main( int argc, char** argv )
{
    //改变 console 字体颜色
    system("color 2F");

    //载入原图
    g_srcImage = imread("1.jpg");
    if( !g_srcImage.data ) { printf("读取 srcImage 错误~！ \n"); return
false; }

    //显示原始图
    namedWindow("【原始图】");
    imshow("【原始图】", g_srcImage);

    // 创建与 src 同类型和大小的矩阵(dst)
    g_dstImage.create( g_srcImage.size(), g_srcImage.type() );

    // 将原图像转换为灰度图像
    cvtColor( g_srcImage, g_srcGrayImage, COLOR_BGR2GRAY );
```

```
    // 创建显示窗口
    namedWindow( "【效果图】Canny 边缘检测", WINDOW_AUTOSIZE );
    namedWindow( "【效果图】Sobel 边缘检测", WINDOW_AUTOSIZE );

    // 创建 trackbar
    createTrackbar( "参数值: ", "【效果图】Canny 边缘检测",
&g_cannyLowThreshold, 120, on_Canny );
    createTrackbar("参数值:", "【效果图】Sobel 边缘检测", &g_sobelKernelSize,
3, on_Sobel );

    // 调用回调函数
    on_Canny(0, 0);
    on_Sobel(0, 0);

    //调用封装了 Scharr 边缘检测代码的函数
    Scharr();

    //轮询获取按键信息，若按下 Q，程序退出
    while((char(waitKey(1)) != 'q')) {}

    return 0;
}

//-------------------------【on_Canny()函数】-------------------------
//      描述：Canny 边缘检测窗口滚动条的回调函数
//-----------------------------------------------------------------
void on_Canny(int, void*)
{
    // 先使用 3x3 内核来降噪
    blur( g_srcGrayImage, g_cannyDetectedEdges, Size(3,3) );

    // 运行我们的 Canny 算子
    Canny( g_cannyDetectedEdges, g_cannyDetectedEdges,
g_cannyLowThreshold, g_cannyLowThreshold*3, 3 );

    //先将 g_dstImage 内的所有元素设置为 0
    g_dstImage = Scalar::all(0);

    //使用 Canny 算子输出的边缘图 g_cannyDetectedEdges 作为掩码，来将原图
g_srcImage 拷到目标图 g_dstImage 中
    g_srcImage.copyTo( g_dstImage, g_cannyDetectedEdges);

    //显示效果图
    imshow( "【效果图】Canny 边缘检测", g_dstImage );
}

//-------------------------【on_Sobel()函数】-------------------------
//      描述：Sobel 边缘检测窗口滚动条的回调函数
//-----------------------------------------------------------------
void on_Sobel(int, void*)
{
```

```
    // 求 X 方向梯度
    Sobel( g_srcImage, g_sobelGradient_X, CV_16S, 1, 0,
(2*g_sobelKernelSize+1), 1, 1, BORDER_DEFAULT );
    convertScaleAbs( g_sobelGradient_X, g_sobelAbsGradient_X );//计算
绝对值，并将结果转换成 8 位

    // 求 Y 方向梯度
    Sobel( g_srcImage, g_sobelGradient_Y, CV_16S, 0, 1,
(2*g_sobelKernelSize+1), 1, 1, BORDER_DEFAULT );
    convertScaleAbs( g_sobelGradient_Y, g_sobelAbsGradient_Y );//计算
绝对值，并将结果转换成 8 位

    // 合并梯度
    addWeighted( g_sobelAbsGradient_X, 0.5, g_sobelAbsGradient_Y, 0.5,
0, g_dstImage );

    //显示效果图
    imshow("【效果图】Sobel 边缘检测", g_dstImage);

}


//-----------------------------【Scharr()函数】----------------------------
//      描述：封装了 Scharr 边缘检测相关代码的函数
//-----------------------------------------------------------------------
void Scharr()
{
    // 求 X 方向梯度
    Scharr( g_srcImage, g_scharrGradient_X, CV_16S, 1, 0, 1, 0,
BORDER_DEFAULT );
    convertScaleAbs( g_scharrGradient_X, g_scharrAbsGradient_X );//计
算绝对值，并将结果转换成 8 位

    // 求 Y 方向梯度
    Scharr( g_srcImage, g_scharrGradient_Y, CV_16S, 0, 1, 1, 0,
BORDER_DEFAULT );
    convertScaleAbs( g_scharrGradient_Y, g_scharrAbsGradient_Y );//计
算绝对值，并将结果转换成 8 位

    // 合并梯度
    addWeighted( g_scharrAbsGradient_X, 0.5, g_scharrAbsGradient_Y, 0.5,
0, g_dstImage );

    //显示效果图
    imshow("【效果图】Scharr 滤波器", g_dstImage);
}
```

下面放出一些程序运行效果图。如图 7.11～7.15 所示。

图 7.11 原始图

图 7.12 canny 边缘检测效果图（1）

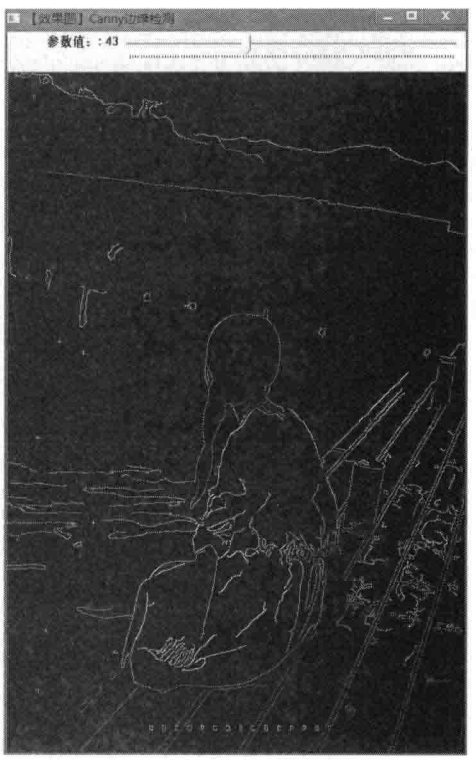


图 7.13 canny 边缘检测效果图（2）

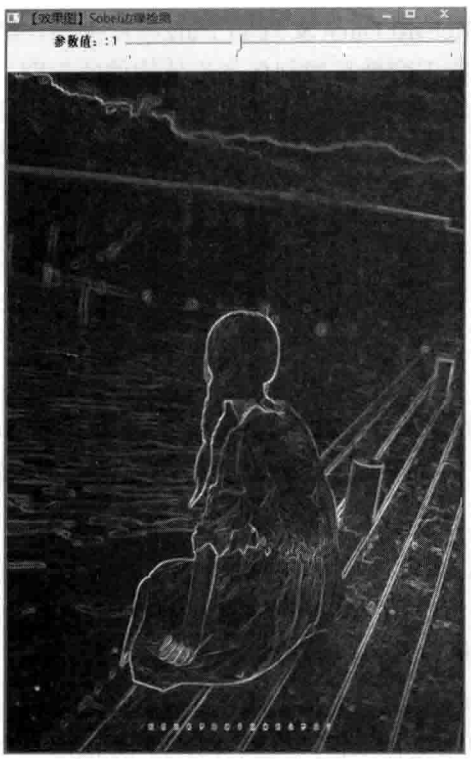


图 7.14 Sobel 边缘检测效果图

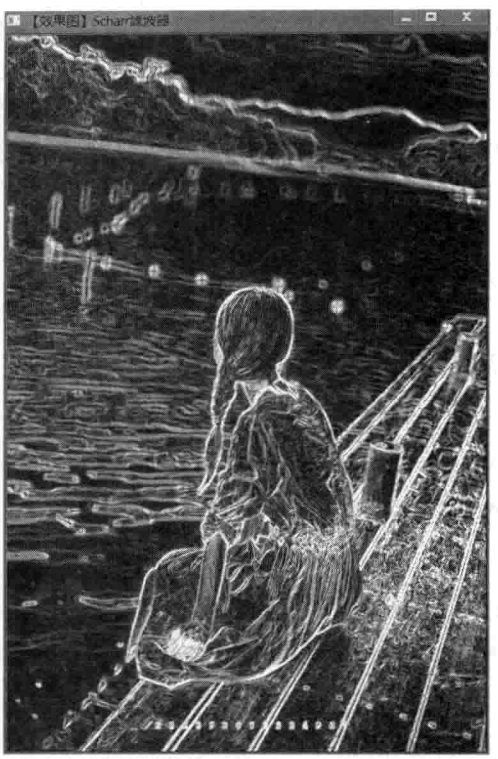


图 7.15 Scharr 滤波器效果图

7.2 霍夫变换

本节中，我们将一起探讨 OpenCV 中霍夫变换相关的知识点，并了解了 OpenCV 中实现霍夫线变换的 HoughLines、HoughLinesP 函数的使用方法，以及实现霍夫圆变换的 HoughCircles 函数的使用方法。

在图像处理和计算机视觉领域中，如何从当前的图像中提取所需要的特征信息是图像识别的关键所在。在许多应用场合中需要快速准确地检测出直线或者圆。其中一种非常有效的解决问题的方法是霍夫（Hough）变换，其为图像处理中从图像中识别几何形状的基本方法之一，应用很广泛，也有很多改进算法。最基本的霍夫变换是从黑白图像中检测直线（线段）。本节就将介绍 OpenCV 中霍夫变换的使用方法和相关知识。

7.2.1 霍夫变换概述

霍夫变换（Hough Transform）是图像处理中的一种特征提取技术，该过程在一个参数空间中通过计算累计结果的局部最大值得到一个符合该特定形状的集合作为霍夫变换结果。霍夫变换于 1962 年由 PaulHough 首次提出，最初的 Hough 变换是设计用来检测直线和曲线的。起初的方法要求知道物体边界线的解析方程，但不需要有关区域位置的先验知识。这种方法的一个突出优点是分割结果的 Robustness，即对数据的不完全或噪声不是非常敏感。然而，要获得描述边界的解

析表达常常是不可能的。后于1972年由Richard Duda & Peter Hart推广使用，经典霍夫变换用来检测图像中的直线，后来霍夫变换扩展到任意形状物体的识别，多为圆和椭圆。霍夫变换运用两个坐标空间之间的变换将在一个空间中具有相同形状的曲线或直线映射到另一个坐标空间的一个点上形成峰值，从而把检测任意形状的问题转化为统计峰值问题。

霍夫变换在OpenCV中分为霍夫线变换和霍夫圆变换两种，下面将分别进行介绍。

7.2.2 OpenCV中的霍夫线变换

我们知道，霍夫线变换是一种用来寻找直线的方法. 在使用霍夫线变换之前，首先要对图像进行边缘检测的处理，即霍夫线变换的直接输入只能是边缘二值图像。

OpenCV支持三种不同的霍夫线变换，它们分别是：标准霍夫变换（Standard Hough Transform，SHT）、多尺度霍夫变换（Multi-Scale Hough Transform，MSHT）和累计概率霍夫变换（Progressive Probabilistic Hough Transform，PPHT）。

其中，多尺度霍夫变换（MSHT）为经典霍夫变换（SHT）在多尺度下的一个变种。而累计概率霍夫变换（PPHT）算法是标准霍夫变换（SHT）算法的一个改进，它在一定的范围内进行霍夫变换，计算单独线段的方向以及范围，从而减少计算量，缩短计算时间。之所以称PPHT为“概率”的，是因为并不将累加器平面内的所有可能的点累加，而只是累加其中的一部分，该想法是如果峰值如果足够高，只用一小部分时间去寻找它就够了。按照猜想，可以实质性地减少计算时间。

在OpenCV中，可以用HoughLines函数来调用标准霍夫变换（SHT）和多尺度霍夫变换（MSHT）。

而HoughLinesP函数用于调用累计概率霍夫变换PPHT。累计概率霍夫变换执行效率很高，所有相比于HoughLines函数，我们更倾向于使用HoughLinesP函数。

总结一下，OpenCV中的霍夫线变换有如下三种：

- 标准霍夫变换（StandardHough Transform，SHT），由HoughLines函数调用。
- 多尺度霍夫变换（Multi-ScaleHough Transform，MSHT），由HoughLines函数调用。
- 累计概率霍夫变换（ProgressiveProbabilistic Hough Transform，PPHT），由HoughLinesP函数调用。

7.2.3 霍夫线变换的原理

（1）众所周知，一条直线在图像二维空间可由两个变量表示，有以下两种情况。如图7.16所示。

① 在笛卡尔坐标系：可由参数斜率和截距（m,b）表示。

② 在极坐标系：可由参数极径和极角（r, θ）表示。

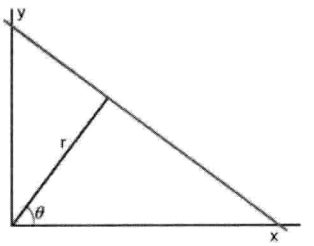


图 7.16 一条直线的两种表示方法

对于霍夫变换，我们将采用第二种方式极坐标系来表示直线．因此，直线的表达式可为:

$$y=\left(-\frac{\cos\theta}{\sin\theta}\right)x+\left(\frac{r}{\sin\theta}\right)$$

化简便可得到:

$$r=x\cos\theta+y\sin\theta$$

（2）一般来说对于点（x_0,y_0），可以将通过这个点的一族直线统一定义为:

$$r_\theta=x_0\cdot\cos\theta+y_0\cdot\sin\theta$$

这就意味着每一对（r_θ, θ）代表一条通过点（x_0,y_0）的直线。

（3）如果对于一个给定点（x_0,y_0），我们在极坐标对极径极角平面绘出所有通过它的直线，将得到一条正弦曲线．例如，对于给定点 x_0= 8 和 y_0= 6 可以绘出如 7.17 所示平面图。

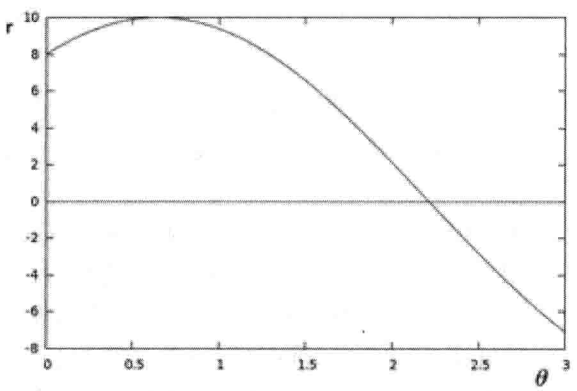


图 7.17 绘制出的正弦曲线

只绘出满足下列条件的点 $r>0$ 和．$0<\theta<2\pi$

（4）我们可以对图像中所有的点进行上述操作．如果两个不同点进行上述操作后得到的曲线在平面 $\theta-r$ 相交，这就意味着它们通过同一条直线。例如，接上面的例子继续对点 x_1=9，y_1=4 和点 x_2=12，y_2=3 绘图，得到图 7.18。

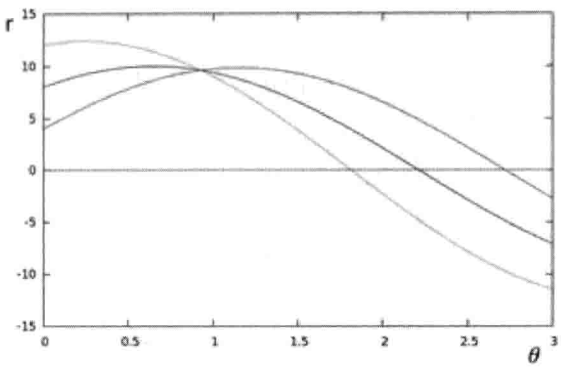


图 7.18 绘制出的曲线

这三条曲线在平面相交于点（0.925,9.6），坐标表示的是参数对 θ —r 或者是说点（x_0,y_0），点（x_1,y_1）和点（x_2,y_2）组成的平面内的的直线。

（5）以上的说明表明，一般来说，一条直线能够通过在平面 θ —r 寻找交于一点的曲线数量来检测。而越多曲线交于一点也就意味着这个交点表示的直线由更多的点组成。一般来说我们可以通过设置直线上点的阈值来定义多少条曲线交于一点，这样才认为检测到了一条直线。

（6）这就是霍夫线变换要做的。它追踪图像中每个点对应曲线间的交点．如果交于一点的曲线的数量超过了阈值，那么可以认为这个交点所代表的参数对（θ，r_θ）在原图像中为一条直线。

7.2.4 标准霍夫变换：HoughLines()函数

此函数可以找出采用标准霍夫变换的二值图像线条。在 OpenCV 中，我们可以用其来调用标准霍夫变换 SHT 和多尺度霍夫变换 MSHT 的 OpenCV 内建算法。

```
C++: void HoughLines(InputArray image, OutputArray lines, double rho,
double theta, int threshold, double srn=0, double stn=0 )
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像。需为 8 位的单通道二进制图像，可以将任意的源图载入进来，并由函数修改成此格式后，再填在这里。
- 第二个参数，InputArray 类型的 lines，经过调用 HoughLines 函数后储存了霍夫线变换检测到线条的输出矢量。每一条线由具有两个元素的矢量（ρ, θ）表示，其中，ρ 是离坐标原点（0,0）（也就是图像的左上角）的距离，θ 是弧度线条旋转角度（0 度表示垂直线，π/2 度表示水平线）。
- 第三个参数，double 类型的 rho，以像素为单位的距离精度。另一种表述方式是直线搜索时的进步尺寸的单位半径。（Latex 中/rho 即表示 ρ）
- 第四个参数，double 类型的 theta，以弧度为单位的角度精度。另一种表述方式是直线搜索时的进步尺寸的单位角度。
- 第五个参数，int 类型的 threshold，累加平面的阈值参数，即识别某部分为图中的一条直线时它在累加平面中必须达到的值。大于阈值 threshold 的线

段才可以被检测通过并返回到结果中。

- 第六个参数，double 类型的 srn，有默认值 0。对于多尺度的霍夫变换，这是第三个参数进步尺寸 rho 的除数距离。粗略的累加器进步尺寸直接是第三个参数 rho，而精确的累加器进步尺寸为 rho/srn。
- 第七个参数，double 类型的 stn，有默认值 0，对于多尺度霍夫变换，srn 表示第四个参数进步尺寸的单位角度 theta 的除数距离。且如果 srn 和 stn 同时为 0，就表示使用经典的霍夫变换。否则，这两个参数应该都为正数。

学完函数解析后，看一个以 HoughLines 为核心的示例程序，就可以全方位了解 HoughLines 函数的使用方法。

```
//---------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;
using namespace std;

//---------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------
int main()
{
    //【1】载入原始图和 Mat 变量定义
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材
图
    Mat midImage,dstImage;//临时变量和目标图的定义

    //【2】进行边缘检测和转化为灰度图
    Canny(srcImage, midImage, 50, 200, 3);//进行一此 canny 边缘检测
    cvtColor(midImage,dstImage, CV_GRAY2BGR);//转化边缘检测后的图为灰度图

    //【3】进行霍夫线变换
    vector<Vec2f> lines;//定义一个矢量结构 lines 用于存放得到的线段矢量集合
    HoughLines(midImage, lines, 1, CV_PI/180, 150, 0, 0 );

    //【4】依次在图中绘制出每条线段
    for( size_t i = 0; i < lines.size(); i++ )
    {
        float rho = lines[i][0], theta = lines[i][1];
        Point pt1, pt2;
        double a = cos(theta), b = sin(theta);
        double x0 = a*rho, y0 = b*rho;
        pt1.x = cvRound(x0 + 1000*(-b));
        pt1.y = cvRound(y0 + 1000*(a));
        pt2.x = cvRound(x0 - 1000*(-b));
        pt2.y = cvRound(y0 - 1000*(a));
        //此句代码的 OpenCV2 版为：
        //line( dstImage, pt1, pt2, Scalar(55,100,195), 1, CV_AA);
```

```
        //此句代码的 OpenCV3 版为:
        line( dstImage, pt1, pt2, Scalar(55,100,195), 1, LINE_AA);
    }

    //【5】显示原始图
    imshow("【原始图】", srcImage);

    //【6】边缘检测后的图
    imshow("【边缘检测后的图】", midImage);

    //【7】显示效果图
    imshow("【效果图】", dstImage);

    waitKey(0);

    return 0;
}
```

运行截图如图 7.19 所示。

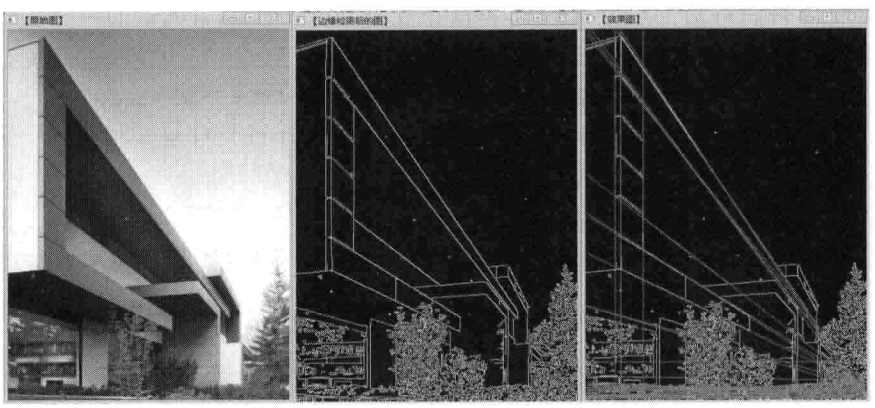


图 7.19 标准霍夫线变换效果图

可以通过调节 line(dstImage, pt1, pt2, Scalar(55,100,195), 1, CV_AA); 一句 Scalar(55,100,195)参数中 G、B、R 颜色值的数值，得到图中想要的线条颜色。

7.2.5 累计概率霍夫变换：HoughLinesP()函数

此函数在 HoughLines 的基础上，在末尾加了一个代表 Probabilistic（概率）的 P，表明它可以采用累计概率霍夫变换（PPHT）来找出二值图像中的直线。

```
C++: void HoughLinesP(InputArray image, OutputArray lines, double rho,
double theta, int threshold, double minLineLength=0, double
maxLineGap=0 )
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像。需为 8 位的单通道二进制图像，可以将任意的源图载入进来后由函数修改成此格式后，再填在这里。

- 第二个参数，InputArray 类型的 lines，经过调用 HoughLinesP 函数后存储了检测到的线条的输出矢量，每一条线由具有 4 个元素的矢量(x_1,y_1, x_2, y_2）表示，其中，(x_1, y_1)和(x_2, y_2) 是是每个检测到的线段的结束点。
- 第三个参数，double 类型的 rho，以像素为单位的距离精度。另一种表述方式是直线搜索时的进步尺寸的单位半径。
- 第四个参数，double 类型的 theta，以弧度为单位的角度精度。另一种表述方式是直线搜索时的进步尺寸的单位角度。
- 第五个参数，int 类型的 threshold，累加平面的阈值参数，即识别某部分为图中的一条直线时它在累加平面中必须达到的值。大于阈值 threshold 的线段才可以被检测通过并返回到结果中。
- 第六个参数，double 类型的 minLineLength，有默认值 0，表示最低线段的长度，比这个设定参数短的线段就不能被显现出来。
- 第七个参数，double 类型的 maxLineGap，有默认值 0，允许将同一行点与点之间连接起来的最大的距离。

对于此函数，依然是为大家准备了示例程序，如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;
using namespace std;

//-----------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //【1】载入原始图和 Mat 变量定义
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材
图
    Mat midImage,dstImage;//临时变量和目标图的定义

    //【2】进行边缘检测和转化为灰度图
    Canny(srcImage, midImage, 50, 200, 3);//进行一此 canny 边缘检测
    cvtColor(midImage,dstImage, COLOR_GRAY2BGR );//转化边缘检测后的图为灰
度图

    //【3】进行霍夫线变换
    vector<Vec4i> lines;//定义一个矢量结构 lines 用于存放得到的线段矢量集合
    HoughLinesP(midImage, lines, 1, CV_PI/180, 80, 50, 10 );

    //【4】依次在图中绘制出每条线段
    for( size_t i = 0; i < lines.size(); i++ )
    {
        Vec4i l = lines[i];
```

```
    //此句代码的 OpenCV2 版为:
        //line( dstImage, Point(l[0], l[1]), Point(l[2], l[3]),
     Scalar(186,88,255), 1, CV_AA);
        //此句代码的 OpenCV3 版为:
    line( dstImage, Point(l[0], l[1]), Point(l[2], l[3]),
Scalar(186,88,255), 1, LINE_AA);
    }

    //【5】显示原始图
    imshow("【原始图】", srcImage);

    //【6】边缘检测后的图
    imshow("【边缘检测后的图】", midImage);

    //【7】显示效果图
    imshow("【效果图】", dstImage);

    waitKey(0);

    return 0;
}
```

运行截图如图 7.20 所示。

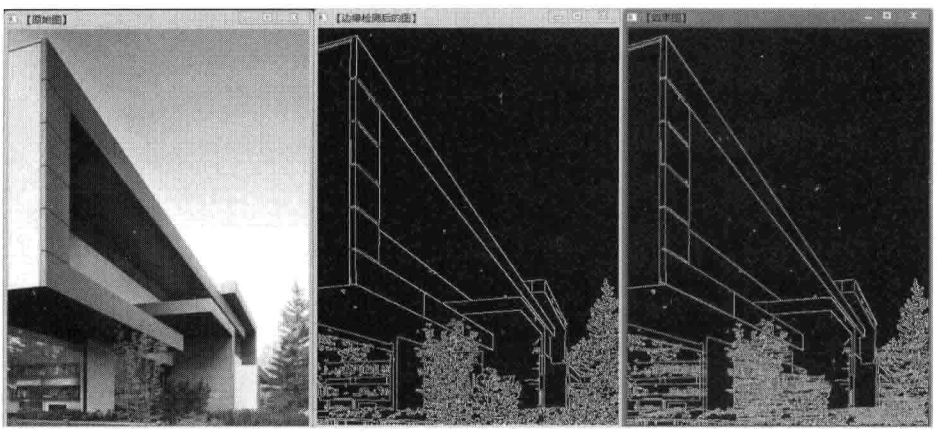


图 7.20 累计概率霍夫变换效果图

7.2.6 霍夫圆变换

霍夫圆变换的基本原理和上面讲的霍夫线变化大体上是很类似的，只是点对应的二维极径极角空间被三维的圆心点 x、y 和半径 r 空间取代。说“大体上类似”的原因是，如果完全用相同的方法的话，累加平面会被三维的累加容器所代替——在这三维中，一维是 x，一维是 y，另外一维是圆的半径 r。这就意味着需要大量的内存而且执行效率会很低，速度会很慢。

对直线来说，一条直线能由参数极径极角（r, θ）表示. 而对圆来说，我们需要三个参数来表示一个圆，也就是：

$$C:(x_{center},y_{center},r)$$

这里的（x_{center},y_{center}）表示圆心的位置（下图中球心的点），而 r 表示半径。这样我们就能唯一的定义一个圆了，见下图。

7.21 原始图

7.22 霍夫圆变换效果图

在 OpenCV 中，我们常常通过一个叫做“霍夫梯度法”的方法来解决圆变换的问题。

7.2.7 霍夫梯度法的原理

霍夫梯度法的原理是这样的：

（1）首先对图像应用边缘检测，比如用 canny 边缘检测。

（2）然后，对边缘图像中的每一个非零点，考虑其局部梯度，即用 Sobel()函数计算 x 和 y 方向的 Sobel 一阶导数得到梯度。

（3）利用得到的梯度，由斜率指定的直线上的每一个点都在累加器中被累加，这里的斜率是从一个指定的最小值到指定的最大值的距离。

（4）同时，标记边缘图像中每一个非 0 像素的位置。

（5）然后从二维累加器中这些点中选择候选的中心，这些中心都大于给定阈值并且大于其所有近邻。这些候选的中心按照累加值降序排列，以便于最支持像素的中心首先出现。

（6）接下来对每一个中心，考虑所有的非 0 像素。

（7）这些像素按照其与中心的距离排序。从到最大半径的最小距离算起，选择非 0 像素最支持的一条半径。

（8）如果一个中心收到边缘图像非 0 像素最充分的支持，并且到前期被选择的中心有足够的距离，那么它就会被保留下来。

这个实现可以使算法执行起来更高效，或许更加重要的是，能够帮助解决三维累加器中会产生许多噪声并且使得结果不稳定的稀疏分布问题。

人无完人，金无足赤。同样，这个算法也并不是十全十美的，还有许多需要指出的缺点。

7.2.8 霍夫梯度法的缺点

（1）在霍夫梯度法中，我们使用 Sobel 导数来计算局部梯度，那么随之而来的假设是，它可以视作等同于一条局部切线，这并不是一个数值稳定的做法。在大多数情况下，这样做会得到正确的结果，但或许会在输出中产生一些噪声。

（2）在边缘图像中的整个非 0 像素集被看做每个中心的候选部分。因此，如果把累加器的阈值设置偏低，算法将要消耗比较长的时间。此外，因为每一个中心只选择一个圆，如果有同心圆，就只能选择其中的一个。

（3）因为中心是按照其关联的累加器值的升序排列的，并且如果新的中心过于接近之前已经接受的中心的话，就不会被保留下来。且当有许多同心圆或者是近似的同心圆时，霍夫梯度法的倾向是保留最大的一个圆。可以说这是一种比较极端的做法，因为在这里默认 Sobel 导数会产生噪声，若是对于无穷分辨率的平滑图像而言的话，这才是必须的。

7.2.9 霍夫圆变换：HoughCircles()函数

HoughCircles 函数可以利用霍夫变换算法检测出灰度图中的圆。它相比之前的 HoughLines 和 HoughLinesP，比较明显的一个区别是不需要源图是二值的，而 HoughLines 和 HoughLinesP 都需要源图为二值图像。

```
C++: void HoughCircles(InputArray image,OutputArray circles, int method,
double dp, double minDist, double param1=100,double param2=100, int
minRadius=0, int maxRadius=0 )
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像，需为 8 位的灰度单通道图像。
- 第二个参数，InputArray 类型的 circles，经过调用 HoughCircles 函数后此参数存储了检测到的圆的输出矢量，每个矢量由包含了 3 个元素的浮点矢量(x,y,radius)表示。
- 第三个参数，int 类型的 method，即使用的检测方法，目前 OpenCV 中就霍夫梯度法一种可以使用，它的标识符为 HOUGH_GRADIENT (OpenCV2 中可写作 CV_HOUGH_GRADIENT)，在此参数处填这个标识符即可。
- 第四个参数，double 类型的 dp，用来检测圆心的累加器图像的分辨率于输入图像之比的倒数，且此参数允许创建一个比输入图像分辨率低的累加器。例如，如果 dp= 1 时，累加器和输入图像具有相同的分辨率。如果 dp=2，累加器便有输入图像一半那么大的宽度和高度。
- 第五个参数，double 类型的 minDist，为霍夫变换检测到的圆的圆心之间的最小距离，即让算法能明显区分的两个不同圆之间的最小距离。这个参数如果太小的话，多个相邻的圆可能被错误地检测成了一个重合的圆。反之，这个参数设置太大，某些圆就不能被检测出来。
- 第六个参数，double 类型的 param1，有默认值 100。它是第三个参数 method 设置的检测方法的对应的参数。对当前唯一的方法霍夫梯度法

CV_HOUGH_GRADIENT，它表示传递给 canny 边缘检测算子的高阈值，而低阈值为高阈值的一半。

- 第七个参数，double 类型的 param2，也有默认值 100。它是第三个参数 method 设置的检测方法的对应的参数。对当前唯一的方法霍夫梯度法 CV_HOUGH_GRADIENT，它表示在检测阶段圆心的累加器阈值。它越小，就越可以检测到更多根本不存在的圆，而它越大的话，能通过检测的圆就更加接近完美的圆形了。
- 第八个参数，int 类型的 minRadius，有默认值 0，表示圆半径的最小值。
- 第九个参数，int 类型的 maxRadius，也有默认值 0，表示圆半径的最大值。

需要注意的是，使用此函数可以很容易地检测出圆的圆心，但是它可能找不到合适的圆半径。我们可以通过第八个参数 minRadius 和第九个参数 maxRadius 指定最小和最大的圆半径，来辅助圆检测的效果。或者，可以直接忽略返回半径，因为它们都有着默认值 0，只用 HoughCircles 函数检测出来的圆心，然后用额外的一些步骤来进一步确定半径。

依然是为大家准备了基于此函数的示例程序。

```
//----------------------【头文件、命名空间包含部分】----------------------
//         描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;
using namespace std;

//---------------------------【main()函数】---------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-------------------------------------------------------------------------
int main()
{
    //【1】载入原始图、Mat 变量定义
    Mat srcImage = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的素材图
    Mat midImage,dstImage;//临时变量和目标图的定义

    //【2】显示原始图
    imshow("【原始图】", srcImage);

    //【3】转为灰度图并进行图像平滑
    cvtColor(srcImage,midImage, COLOR_BGR2GRAY);//转化边缘检测后的图为灰度图
    GaussianBlur( midImage, midImage, Size(9, 9), 2, 2 );

    //【4】进行霍夫圆变换
    vector<Vec3f> circles;
    HoughCircles( midImage, circles, HOUGH_GRADIENT,1.5, 10, 200, 100, 0, 0 );
```

```
    //【5】依次在图中绘制出圆
    for( size_t i = 0; i < circles.size(); i++ )
    {
        //参数定义
        Point center(cvRound(circles[i][0]), cvRound(circles[i][1]));
        int radius = cvRound(circles[i][2]);
        //绘制圆心
        circle( srcImage, center, 3, Scalar(0,255,0), -1, 8, 0 );
        //绘制圆轮廓
        circle( srcImage, center, radius, Scalar(155,50,255), 3, 8, 0 );
    }

    //【6】显示效果图
    imshow("【效果图】", srcImage);

    waitKey(0);

    return 0;
}
```

程序运行截图见图 7.21、7.22。

7.2.10 综合示例：霍夫变换

这次的综合示例，我们在 HoughLinesP 函数的基础上，为其添加了用于控制其第五个参数阈值 threshold 的滚动条，因此可以通过调节滚动条来改变阈值，从而动态地控制霍夫线变换检测的线条多少。

详细注释的程序源代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace std;
using namespace cv;


//-------------------------【全局变量声明部分】-------------------------
//      描述：全局变量声明
//-------------------------------------------------------------------------
Mat g_srcImage, g_dstImage,g_midImage;//原始图、中间图和效果图
vector<Vec4i> g_lines;//定义一个矢量结构 g_lines 用于存放得到的线段矢量集合
//变量接收的 TrackBar 位置参数
int g_nthreshold=100;

//-------------------------【全局函数声明部分】-------------------------
//      描述：全局函数声明
//-------------------------------------------------------------------------
```

```
static void on_HoughLines(int, void*);//回调函数

//-----------------------------【main()函数】--------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始
//-----------------------------------------------------------------------
int main()
{
    //改变 console 字体颜色
    system("color 3F");

    //载入原始图和 Mat 变量定义
    Mat g_srcImage = imread("1.jpg");  //工程目录下应该有一张名为 1.jpg 的
素材图

    //显示原始图
    imshow("【原始图】", g_srcImage);

    //创建滚动条
    namedWindow("【效果图】",1);
    createTrackbar("值", "【效果图】",&g_nthreshold,200,on_HoughLines);

    //进行边缘检测和转化为灰度图
    Canny(g_srcImage, g_midImage, 50, 200, 3);//进行一次 canny 边缘检测
    cvtColor(g_midImage,g_dstImage, COLOR_GRAY2BGR);//转化边缘检测后的图
为灰度图

    //调用一次回调函数，调用一次 HoughLinesP 函数
    on_HoughLines(g_nthreshold,0);
    HoughLinesP(g_midImage, g_lines, 1, CV_PI/180, 80, 50, 10 );

    //显示效果图
    imshow("【效果图】", g_dstImage);

    waitKey(0);

    return 0;

}


//-----------------------【on_HoughLines()函数】----------------------
//      描述：【顶帽运算/黑帽运算】窗口的回调函数
//-----------------------------------------------------------------------
static void on_HoughLines(int, void*)
{
    //定义局部变量储存全局变量
     Mat dstImage=g_dstImage.clone();
     Mat midImage=g_midImage.clone();

     //调用 HoughLinesP 函数
     vector<Vec4i> mylines;
    HoughLinesP(midImage, mylines, 1, CV_PI/180, g_nthreshold+1, 50,
```

```
10 );

    //循环遍历绘制每一条线段
    for( size_t i = 0; i < mylines.size(); i++ )
    {
        Vec4i l = mylines[i];
        line( dstImage, Point(l[0], l[1]), Point(l[2], l[3]),
Scalar(23,180,55), 1, LINE_AA);
    }
    //显示图像
    imshow("【效果图】",dstImage);
}
```

放一些运行截图。如图 7.23～7.26 所示。

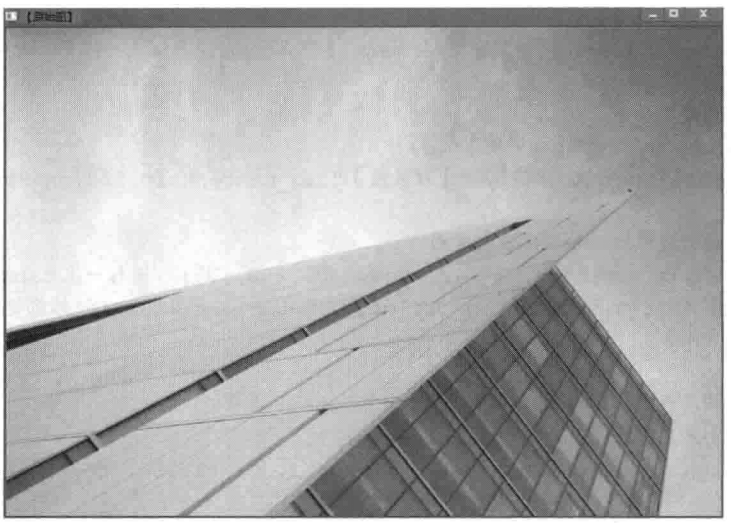


图 7.23 原始图

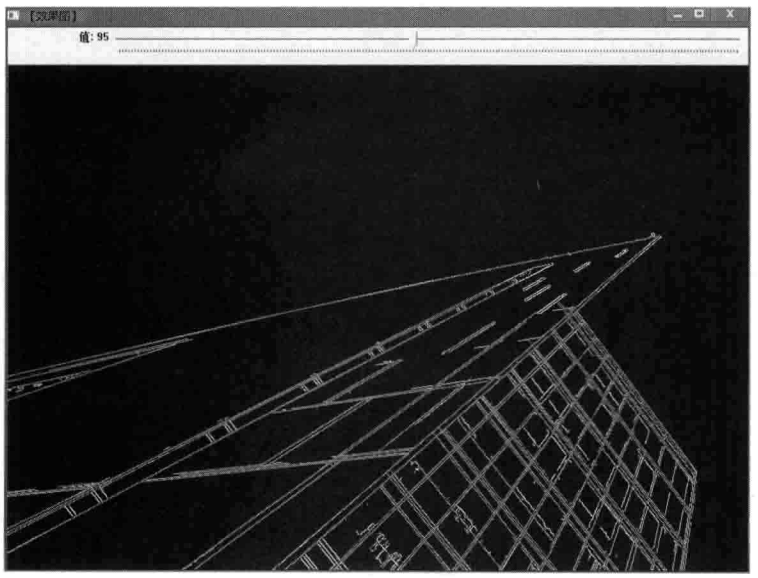


图 7.24 阈值为 95 时的程序截图

图 7.25 阈值为 35 时的程序截图

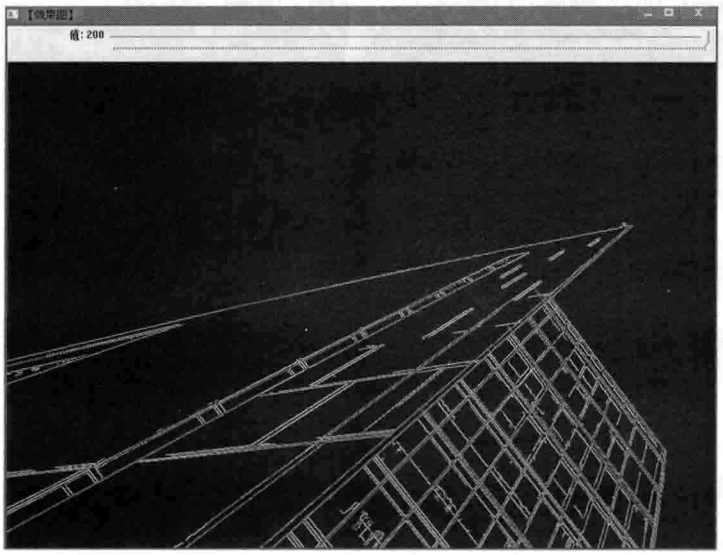


图 7.26 阈值为 200 时的程序截图

7.3 重映射

本节中，我们主要一起了解重映射的概念，以及 OpenCV 中相关的实现函数 remap()。

7.3.1 重映射的概念

重映射，就是把一幅图像中某位置的像素放置到另一个图片指定位置的过程。为了完成映射过程，需要获得一些插值为非整数像素的坐标，因为源图像与目标图像的像素坐标不是一一对应的。一般情况下，我们通过重映射来表达每个像素的位置（x,y），像这样：

$$g(x,y) = f(h(x,y))$$

在这里，g()是目标图像，f()是源图像，而 h(x,y)是作用于(x,y)的映射方法函数。

来看个例子。若有一幅图像 I，对其按照下面的条件作重映射:

$$h(x,y) = (I.cols - x, y)$$

图像会按照 x 轴方向发生翻转。那么，源图像和效果图分别如图 7.27 和 7.28 所示。

图 7.27 原始图

图 7.28 经过重映射的效果图

在 OpenCV 中，可以使用函数 remap()来实现简单重映射，下面我们就一起来看看这个函数。

7.3.2 实现重映射：remap()函数

remap()函数会根据指定的映射形式，将源图像进行重映射几何变换，基于的公式如下：

$$dst(x,y)=src(map_x(x,y),map_y(x,y))$$

需要注意，此函数不支持就地（in-place）操作。看看其原型和参数。

```
C++: void remap(InputArray src, OutputArraydst, InputArray map1,
InputArray map2, int interpolation, intborderMode=BORDER_CONSTANT,
const Scalar& borderValue=Scalar())
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，且需为单通道 8 位或者浮点型图像。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，即这个参数用于存放函数调用后的输出结果，需和源图片有一样的尺寸和类型。
- 第三个参数，InputArray 类型的 map1，它有两种可能的表示对象。
  - 表示点（x，y）的第一个映射。
  - 表示 CV_16SC2、CV_32FC1 或 CV_32FC2 类型的 X 值。
- 第四个参数，InputArray 类型的 map2，同样，它也有两种可能的表示对象，而且它会根据 map1 来确定表示那种对象。
  - 若 map1 表示点（x，y）时。这个参数不代表任何值。

  - 表示 CV_16UC1，CV_32FC1 类型的 Y 值（第二个值）。
- 第五个参数，int 类型的 interpolation，插值方式，之前的 resize()函数中有讲到，需要注意，resize()函数中提到的 INTER_AREA 插值方式在这里是不支持的，所以可选的插值方式如下（需要注意，这些宏相应的 OpenCV2 版为在它们的宏名称前面加上“CV_”前缀，比如“INTER_LINEAR”的 OpenCV2 版为“CV_INTER_LINEAR”）：
  - INTER_NEAREST——最近邻插值
  - INTER_LINEAR——双线性插值（默认值）
  - INTER_CUBIC——双三次样条插值（逾 4×4 像素邻域内的双三次插值）
  - INTER_LANCZOS4——Lanczos 插值（逾 8×8 像素邻域的 Lanczos 插值）
- 第六个参数，int 类型的 borderMode，边界模式，有默认值 BORDER_CONSTANT，表示目标图像中“离群点（outliers）”的像素值不会被此函数修改。
- 第七个参数，const Scalar&类型的 borderValue，当有常数边界时使用的值，其有默认值 Scalar()，即默认值为 0。

7.3.3 基础示例程序：基本重映射

下面将贴出精简后的以 remap 函数为核心的示例程序，方便大家快速掌握 remap 函数的使用方法。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;

//-----------------------------【main()函数】-----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main( )
{
    //【0】变量定义
    Mat srcImage, dstImage;
    Mat map_x, map_y;

    //【1】载入原始图
    srcImage = imread( "1.jpg", 1 );
    if(!srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread 函数指定的图片存在~！ \n"); return false; }
    imshow("原始图",srcImage);

    //【2】创建和原始图一样的效果图，x 重映射图，y 重映射图
    dstImage.create( srcImage.size(), srcImage.type() );
    map_x.create( srcImage.size(), CV_32FC1 );
```

```
    map_y.create( srcImage.size(), CV_32FC1 );

    //【3】双层循环，遍历每一个像素点，改变map_x & map_y的值
    for( int j = 0; j < srcImage.rows;j++)
    {
        for( int i = 0; i < srcImage.cols;i++)
        {
            //改变map_x & map_y的值.
            map_x.at<float>(j,i) = static_cast<float>(i);
            map_y.at<float>(j,i) = static_cast<float>(srcImage.rows -
j);
        }
    }

    //【4】进行重映射操作
    //此句代码的OpenCV2版为:
    //remap( srcImage, dstImage, map_x, map_y, CV_INTER_LINEAR,
BORDER_CONSTANT, Scalar(0,0, 0) );
    //此句代码的OpenCV3版为:
    remap( srcImage, dstImage, map_x, map_y, INTER_LINEAR,
BORDER_CONSTANT, Scalar(0,0, 0) );
    //【5】显示效果图
    imshow( "【程序窗口】", dstImage );
    waitKey();

    return 0;
}
```

程序运行截图如图7.29和图7.30所示。

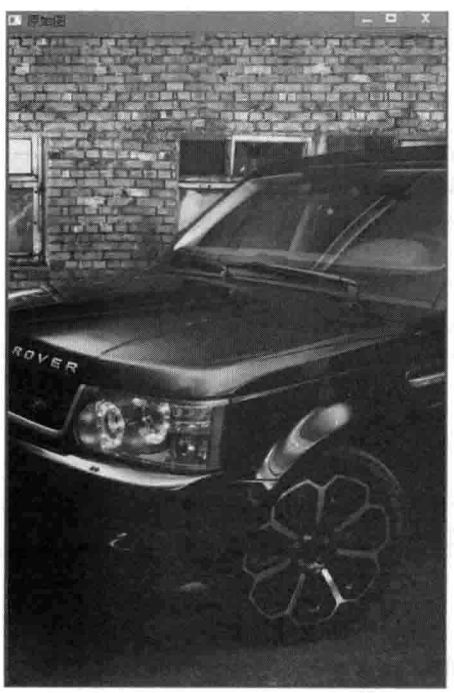


图7.29 原始图

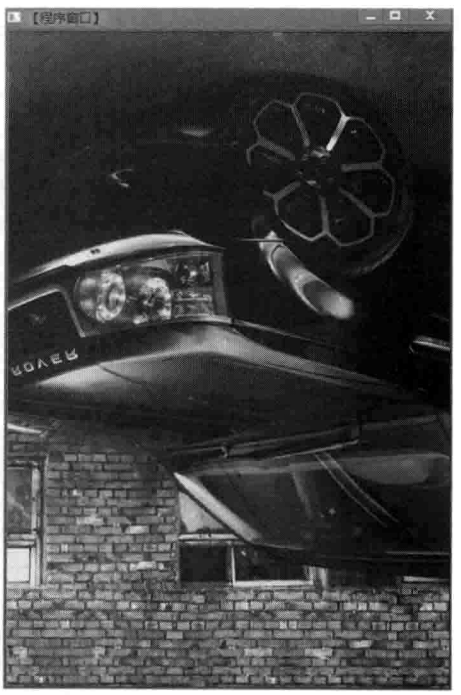


图7.30 效果图

7.3.4 综合示例程序：实现多种重映射

先放出以 remap 为核心的综合示例程序，可以用按键控制四种不同的映射模式。如图 7.31 所示。

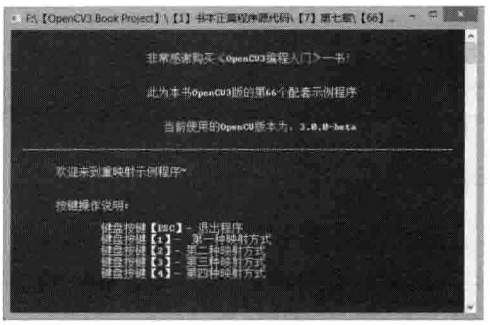


图 7.31 程序按键说明

这个程序详细注释的源代码如下。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//-----------------------------【宏定义部分】-----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME "【程序窗口】"        //为窗口标题定义的宏


//-----------------------【全局变量声明部分】-----------------------
//         描述：全局变量的声明
//-------------------------------------------------------------------
Mat g_srcImage, g_dstImage;
Mat g_map_x, g_map_y;


//-----------------------【全局函数声明部分】-----------------------
//         描述：全局函数的声明
//-------------------------------------------------------------------
int update_map( int key);
static void ShowHelpText();//输出帮助文字

//-----------------------【main()函数】-----------------------
//         描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------
int main( int argc, char** argv )
{
```

```
    //改变 console 字体颜色
    system("color 2F");

    //显示帮助文字
    ShowHelpText();

    //【1】载入原始图
    g_srcImage = imread( "1.jpg", 1 );
    if(!g_srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定的图片存在~！ \n"); return false; }
    imshow("原始图",g_srcImage);

    //【2】创建和原始图一样的效果图，x 重映射图，y 重映射图
    g_dstImage.create( g_srcImage.size(), g_srcImage.type() );
    g_map_x.create( g_srcImage.size(), CV_32FC1 );
    g_map_y.create( g_srcImage.size(), CV_32FC1 );

    //【3】创建窗口并显示
    namedWindow( WINDOW_NAME, WINDOW_AUTOSIZE );
    imshow(WINDOW_NAME,g_srcImage);

    //【4】轮询按键，更新 map_x 和 map_y 的值，进行重映射操作并显示效果图
    while( 1 )
    {
        //获取键盘按键
        int key = waitKey(0);

        //判断 ESC 是否按下，若按下便退出
        if( (key & 255) == 27 )
        {
            cout << "程序退出.....\n";
            break;
        }

       //根据按下的键盘按键来更新 map_x & map_y 的值. 然后调用 remap()进行重映射
        update_map(key);
        //此句代码的 OpenCV2 版为：
        //remap( g_srcImage, g_dstImage, g_map_x, g_map_y,
CV_INTER_LINEAR, BORDER_CONSTANT, Scalar(0,0, 0) );
        //此句代码的 OpenCV3 版为：
        remap( g_srcImage, g_dstImage, g_map_x, g_map_y, INTER_LINEAR,
BORDER_CONSTANT, Scalar(0,0, 0) );
        //显示效果图
        imshow( WINDOW_NAME, g_dstImage );
    }
    return 0;
}

//-----------------------【update_map()函数】-----------------------
//          描述：根据按键来更新 map_x 与 map_x 的值
//-------------------------------------------------------------------
int update_map( int key )
```

```
{
    //双层循环，遍历每一个像素点
    for( int j = 0; j < g_srcImage.rows;j++)
    {
        for( int i = 0; i < g_srcImage.cols;i++)
        {
            switch(key)
            {
            case '1': // 键盘【1】键按下，进行第一种重映射操作
                if( i > g_srcImage.cols*0.25 && i < g_srcImage.cols*0.75
&& j > g_srcImage.rows*0.25 && j < g_srcImage.rows*0.75)
                {
                    g_map_x.at<float>(j,i) = static_cast<float>(2*( i -
g_srcImage.cols*0.25 ) + 0.5);
                    g_map_y.at<float>(j,i) = static_cast<float>(2*( j -
g_srcImage.rows*0.25 ) + 0.5);
                }
                else
                {
                    g_map_x.at<float>(j,i) = 0;
                    g_map_y.at<float>(j,i) = 0;
                }
                break;
            case '2':// 键盘【2】键按下，进行第二种重映射操作
                g_map_x.at<float>(j,i) = static_cast<float>(i);
                g_map_y.at<float>(j,i) =
static_cast<float>(g_srcImage.rows - j);
                break;
            case '3':// 键盘【3】键按下，进行第三种重映射操作
                g_map_x.at<float>(j,i) =
static_cast<float>(g_srcImage.cols - i);
                g_map_y.at<float>(j,i) = static_cast<float>(j);
                break;
            case '4':// 键盘【4】键按下，进行第四种重映射操作
                g_map_x.at<float>(j,i) =
static_cast<float>(g_srcImage.cols - i);
                g_map_y.at<float>(j,i) =
static_cast<float>(g_srcImage.rows - j);
                break;
            }
        }
    }
    return 1;
}

//-----------------------【ShowHelpText()函数】-----------------------
//      描述：输出一些帮助信息
//----------------------------------------------------------------------
static void ShowHelpText()
{
    //输出一些帮助信息
    printf("\n\n\n\t 欢迎来到重映射示例程序～\n\n");
```

```
    printf("\t 当前使用的 OpenCV 版本为 OpenCV "CV_VERSION);
    printf( "\n\n\t 按键操作说明: \n\n"
        "\t\t 键盘按键【ESC】- 退出程序\n"
        "\t\t 键盘按键【1】- 第一种映射方式\n"
        "\t\t 键盘按键【2】- 第二种映射方式\n"
        "\t\t 键盘按键【3】- 第三种映射方式\n"
        "\t\t 键盘按键【4】- 第四种映射方式\n" );
}
```

程序的运行效果图如图 7.32～7.36 所示。

图 7.32 原始图

图 7.33 第一种重映射

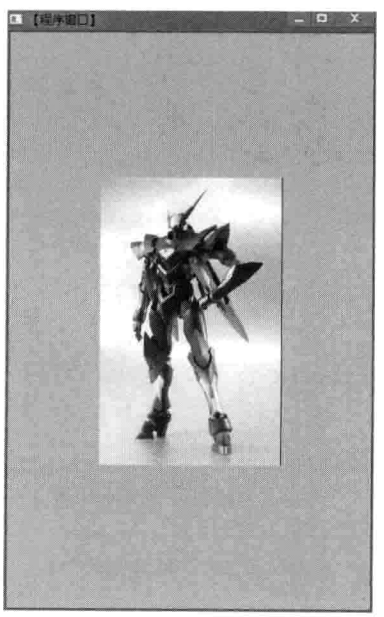


图 7.34 第二种重映射

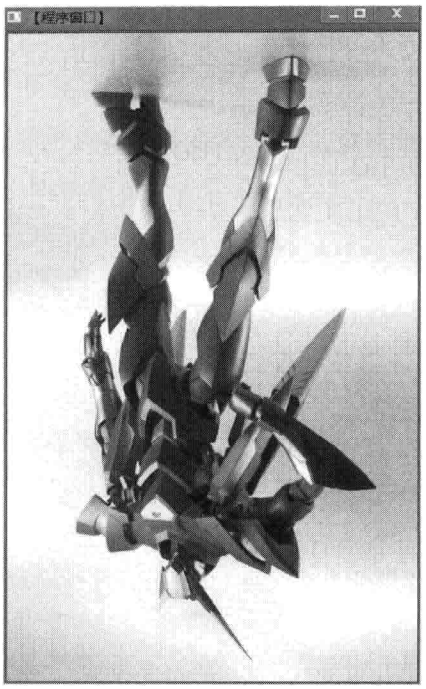


图 7.35 第三种重映射

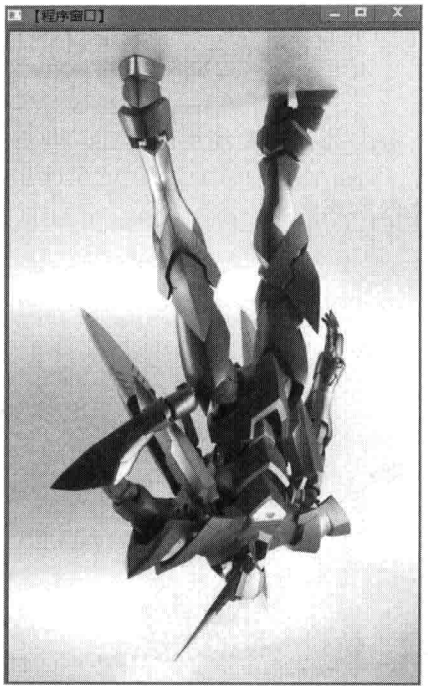


图 7.36 第四种重映射

7.4 仿射变换

本节中，我们将一起了解仿射变换的概念，以及 OpenCV 中相关的实现函数 warpAffine 和 getRotationMatrix2D。

7.4.1 认识仿射变换

仿射变换（Affine Transformation 或 Affine Map），又称仿射映射，是指在几何中，一个向量空间进行一次线性变换并接上一个平移，变换为另一个向量空间的过程。它保持了二维图形的“平直性”（直线经过变换之后依然是直线）和“平行性”（二维图形之间的相对位置关系保持不变，平行线依然是平行线，且直线上点的位置顺序不变）。

一个任意的仿射变换都能表示为乘以一个矩阵（线性变换）接着再加上一个向量（平移）的形式。

那么，我们能够用仿射变换来表示如下三种常见的变换形式：

- 旋转，rotation (线性变换)
- 平移，translation(向量加)
- 缩放，scale(线性变换)

进行更深层次的理解，仿射变换代表的是两幅图之间的一种映射关系。

而我们通常使用 2 x 3 的矩阵来表示仿射变换。

$$A=\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}_{2\times 2} \quad B=\begin{bmatrix} b_{00} \\ b_{10} \end{bmatrix}_{2\times 1} \quad M=[A \quad B]=\begin{bmatrix} a_{00} & a_{01} & b_{00} \\ a_{10} & a_{11} & b_{10} \end{bmatrix}_{2\times 3}$$

考虑到我们要使用矩阵 A 和 B 对二维向量 $X=\begin{bmatrix} x \\ y \end{bmatrix}$ 做变换，所以也能表示为下列形式。

$$T=A\cdot\begin{bmatrix} x \\ y \end{bmatrix}+B$$

或者

$$T=M\cdot[x,y,1]^T$$

即：

$$T=\begin{bmatrix} a_{00}x+a_{01}y+b_{00} \\ a_{10}x+a_{11}y+b_{10} \end{bmatrix}_{2\times 2}$$

7.4.2 仿射变换的求法

我们知道，仿射变换表示的就是两幅图片之间的一种联系，关于这种联系的信息大致可从以下两种场景获得。

- 已知 X 和 T，而且已知它们是有联系的。接下来的工作就是求出矩阵 M。
- 已知 M 和 X，想求得 T。只要应用算式 $T=M\cdot X$ 即可。对于这种联系的信息可以用矩阵 M 清晰地表达 (即给出明确的 2×3 矩阵)，也可以用两幅图片点之间几何关系来表达。

形象地说明一下，因为矩阵 M 联系着两幅图片，就以其表示两图中各三点直接的联系为例。如图 7.37 所示。

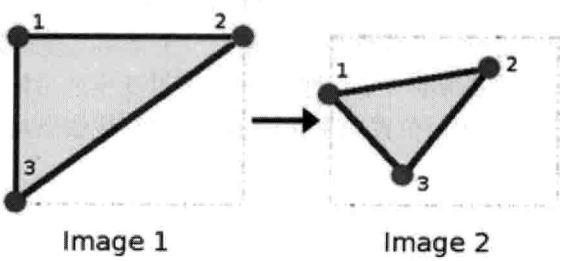


图 7.37 示例图

图中，点 1、2 和 3（在 Image1 中形成一个三角形）与 Image2 中的三个点是一一映射的关系，且它们仍然形成三角形，但形状已经和之前不一样了。我们能通过这样两组三点求出仿射变换（可以选择自己喜欢的点），接着就可以把仿射变换应用到图像中去。

OpenCV 仿射变换相关的函数一般涉及到 warpAffine 和 getRotationMatrix2D 这两个函数：

- 使用 OpenCV 函数 warpAffine 来实现一些简单的重映射。
- 使用 OpenCV 函数 getRotationMatrix2D 来获得旋转矩阵。

下面分别对其进行讲解。

7.4.3 进行仿射变换：warpAffine()函数

warpAffine 函数的作用是依据以下公式子，对图像做仿射变换。

$$dst(x,y)=src(M_{11}x+M_{12}y+M_{13},\ M_{21}x+M_{22}y+M_{23})$$

函数原型如下。

```
C++: void warpAffine(InputArray src,OutputArray dst, InputArray M, Size
dsize, int flags=INTER_LINEAR, intborderMode=BORDER_CONSTANT, const
Scalar& borderValue=Scalar())
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，需和源图片有一样的尺寸和类型。
- 第三个参数，InputArray 类型的 M，2×3 的变换矩阵。
- 第四个参数，Size 类型的 dsize，表示输出图像的尺寸。
- 第五个参数，int 类型的 flags，插值方法的标识符。此参数有默认值 INTER_LINEAR(线性插值)，可选的插值方式如表 7.1 所示。

表 7.1 warpAffine 函数可选的插值方式

标识符	含义
INTER_NEAREST	最近邻插值
INTER_LINEAR	线性插值（默认值）
INTER_AREA	区域插值
INTER_CUBIC	三次样条插值
INTER_LANCZOS4	Lanczos 插值
CV_WARP_FILL_OUTLIERS	填充所有输出图像的象素。如果部分象素落在输入图像的边界外，那么它们的值设定为 fillval
CV_WARP_INVERSE_MAP	表示 M 为输出图像到输入图像的反变换。因此可以直接用来做象素插值。否则, warpAffine 函数从 M 矩阵得到反变换

- 第六个参数，int 类型的 borderMode，边界像素模式，默认值为 BORDER_CONSTANT。
- 第七个参数，const Scalar&类型的 borderValue，在恒定的边界情况下取的值，默认值为 Scalar()，即 0。

另外提一点，WarpAffine 函数与一个叫做 cvGetQuadrangleSubPix()的函数类

似，但是不完全相同。WarpAffine 要求输入和输出图像具有同样的数据类型，有更大的资源开销（因此对小图像不太合适）而且输出图像的部分可以保留不变。而 cvGetQuadrangleSubPix 可以精确地从 8 位图像中提取四边形到浮点数缓存区中，具有比较小的系统开销，而且总是全部改变输出图像的内容。

7.4.4 计算二维旋转变换矩阵：getRotationMatrix2D()函数

getRotationMatrix2D()函数用于计算二维旋转变换矩阵。变换会将旋转中心映射到它自身。

```
C++: Mat getRotationMatrix2D(Point2fcenter, double angle, double scale)
```

- 第一个参数，Point2f 类型的 center，表示源图像的旋转中心。
- 第二个参数，double 类型的 angle，旋转角度。角度为正值表示向逆时针旋转（坐标原点是左上角）。
- 第三个参数，double 类型的 scale，缩放系数。

此函数计算以下矩阵：

$$\begin{bmatrix} \alpha\beta(1-\alpha)\cdot \text{center.x} - \beta\text{center.y} \\ -\beta\alpha\beta\cdot \text{center.x} + (1-\alpha)\cdot \text{center.y} \end{bmatrix}$$

其中：

$$\alpha = \text{scale}\cdot\cos\text{angle},$$
$$\beta = \text{scale}\cdot\sin\text{angle}$$

7.4.5 示例程序：仿射变换

学习完上面的讲解和函数实现，下面是一个以 warpAffine 和 getRotationMatrix2D 函数为核心的对图像进行仿射变换的示例程序。

```
//---------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//---------------------------【宏定义部分】-----------------------------
//      描述：定义一些辅助宏
//-------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图窗口】"                    //为窗口标题定义的宏
#define WINDOW_NAME2 "【经过Warp后的图像】"          //为窗口标题定义的宏
#define WINDOW_NAME3 "【经过Warp和Rotate后的图像】"          //为窗口标题定义的宏
```

```
//-------------------------【全局函数声明部分】-------------------------
//      描述：全局函数的声明
//-----------------------------------------------------------------------
static void ShowHelpText();


//-------------------------【main()函数】-------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main( )
{
    //【0】改变 console 字体颜色
    system("color 1A");

    //【0】显示欢迎和帮助文字
    ShowHelpText();

    //【1】参数准备
    //定义两组点，代表两个三角形
    Point2f srcTriangle[3];
    Point2f dstTriangle[3];
    //定义一些 Mat 变量
    Mat rotMat( 2, 3, CV_32FC1 );
    Mat warpMat( 2, 3, CV_32FC1 );
    Mat srcImage, dstImage_warp, dstImage_warp_rotate;

    //【2】加载源图像并作一些初始化
    srcImage = imread( "1.jpg", 1 );
    if(!srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread 函
数指定的图片存在～！ \n"); return false; }
    // 设置目标图像的大小和类型与源图像一致
    dstImage_warp = Mat::zeros( srcImage.rows, srcImage.cols,
srcImage.type() );

    //【3】设置源图像和目标图像上的三组点以计算仿射变换
    srcTriangle[0] = Point2f( 0,0 );
    srcTriangle[1] = Point2f( static_cast<float>(srcImage.cols - 1),
0 );
    srcTriangle[2] = Point2f( 0, static_cast<float>(srcImage.rows -
1 ));

    dstTriangle[0] = Point2f( static_cast<float>(srcImage.cols*0.0),
static_cast<float>(srcImage.rows*0.33));
    dstTriangle[1] = Point2f( static_cast<float>(srcImage.cols*0.65),
static_cast<float>(srcImage.rows*0.35));
    dstTriangle[2] = Point2f( static_cast<float>(srcImage.cols*0.15),
static_cast<float>(srcImage.rows*0.6));

    //【4】求得仿射变换
    warpMat = getAffineTransform( srcTriangle, dstTriangle );

    //【5】对源图像应用刚刚求得的仿射变换
```

```
    warpAffine( srcImage, dstImage_warp, warpMat,
dstImage_warp.size() );

    //【6】对图像进行缩放后再旋转
    // 计算绕图像中点顺时针旋转 50 度缩放因子为 0.6 的旋转矩阵
    Point center = Point( dstImage_warp.cols/2, dstImage_warp.rows/2 );
    double angle = -30.0;
    double scale = 0.8;
    // 通过上面的旋转细节信息求得旋转矩阵
    rotMat = getRotationMatrix2D( center, angle, scale );
    // 旋转已缩放后的图像
    warpAffine( dstImage_warp, dstImage_warp_rotate, rotMat,
dstImage_warp.size() );


    //【7】显示结果
    imshow( WINDOW_NAME1, srcImage );
    imshow( WINDOW_NAME2, dstImage_warp );
    imshow( WINDOW_NAME3, dstImage_warp_rotate );

    // 等待用户按任意按键退出程序
    waitKey(0);

    return 0;
}


//-----------------------【ShowHelpText()函数】----------------------
//      描述：输出一些帮助信息
//----------------------------------------------------------------------
static void ShowHelpText()
{
    //输出一些帮助信息
    printf(    "\n\n\n\t 欢迎来到【仿射变换】示例程序～\n\n");
    printf("\t 当前使用的 OpenCV 版本为 OpenCV "CV_VERSION); );
}
```

程序的运行截图如图 7.38、7.39、7.40 所示。

图 7.38 原始图

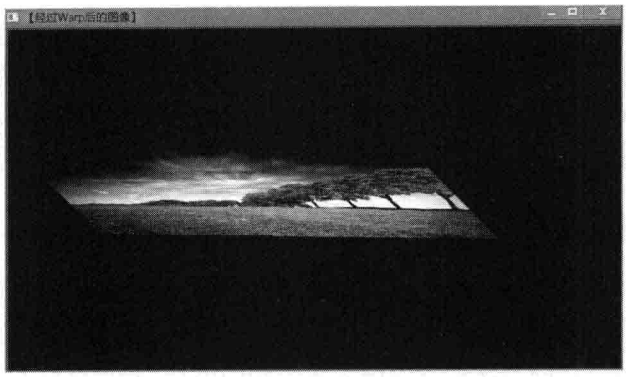


图 7.39 经过 Warp 后的图像

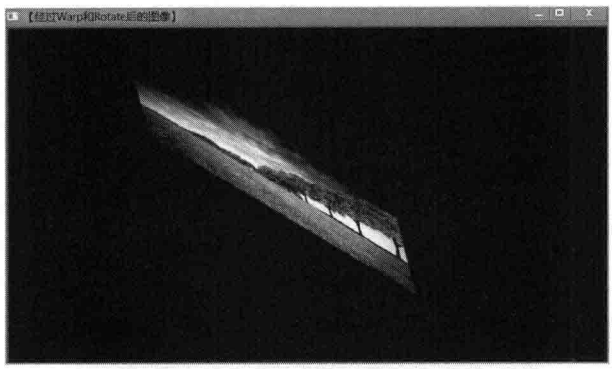


图 7.40 经过 Warp 和 Rotate 后的图像

7.5 直方图均衡化

很多时候，我们用相机拍摄的照片的效果往往会不尽人意。这时，可以对这些图像进行一些处理，来扩大图像的动态范围。这种情况下最常用到的技术就是直方图均衡化。未经均衡化的图片范例如图 7.41、7.42 所示。

图 7.41 未经均衡化的图像

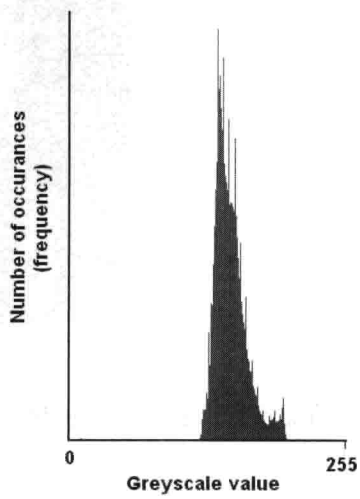


图 7.42 未经均衡化的图像相应的直方图

在图 7.41 中，我们可以看到，左边的图像比较淡，因为其数值范围变化比较小，可以在这幅图的直方图（图 7.42）中明显地看到。因为处理的是 8 位图像，其亮度值是从 0 到 255，但直方图值显示的实际亮度却集中在亮度范围的中间区域。为了解决这个问题，就可以用到直方图均衡化技术，先来看看其概念。

7.5.1 直方图均衡化的概念和特点

直方图均衡化是灰度变换的一个重要应用，它高效且易于实现，广泛应用于图像增强处理中。图像的像素灰度变化是随机的，直方图的图形高低不齐，直方图均衡化就是用一定的算法使直方图大致平和的方法。均衡化效果示例如图 7.43、7.44 所示。

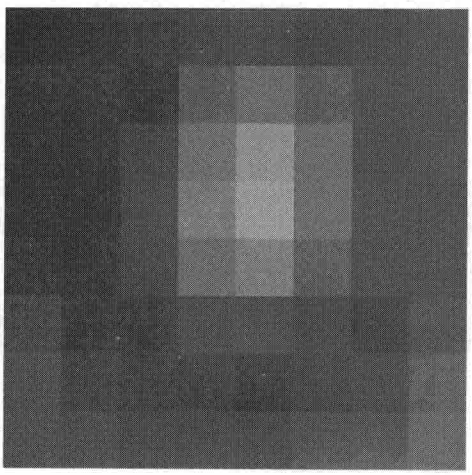


图 7.43 原始图

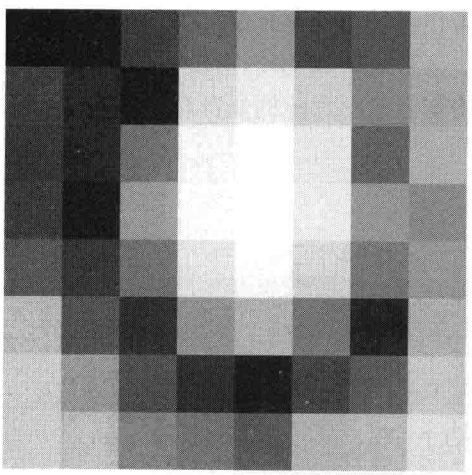


图 7.44 均衡化后图像

简而言之，直方图均衡化是通过拉伸像素强度分布范围来增强图像对比度的一种方法。

均衡化处理后的图像只能是近似均匀分布。均衡化图像的动态范围扩大了，

但其本质是扩大了量化间隔，而量化级别反而减少了，因此，原来灰度不同的象素经处理后可能变的相同，形成了一片相同灰度的区域，各区域之间有明显的边界，从而出现了伪轮廓。

在原始图像对比度本来就很高的情况下，如果再均衡化则灰度调和，对比度会降低。在泛白缓和的图像中，均衡化会合并一些象素灰度，从而增大对比度。均衡化后的图片如果再对其均衡化，则图像不会有任何变化。如图 7.45、7.46 所示。

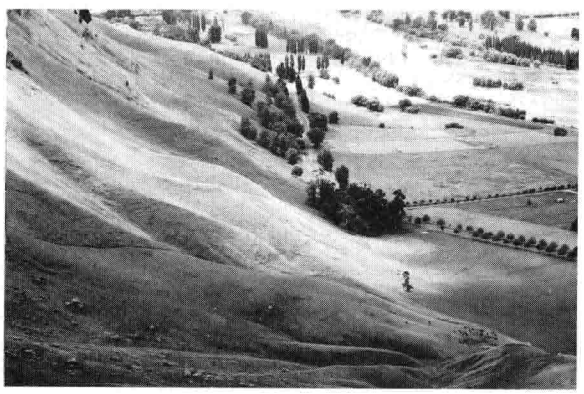


图 7.45 经过均衡化的同一幅图像

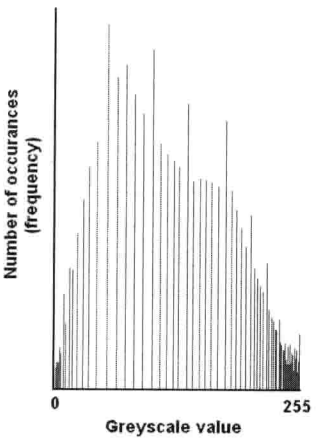


图 7.46 经过均衡化的图像相应的直方图

通过图 7.46 可以发现，经过均衡化的图像，其频谱更加舒展，有效地利用了 0～255 的空间，图像表现力更加出色。

7.5.2 实现直方图均衡化：equalizeHist()函数

在 OpenCV 中，直方图均衡化的功能实现由 equalizeHist 函数完成。我们一起看看它的函数描述。

```
C++: void equalizeHist(InputArray src, OutputArray dst)
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，需为 8 位单通道的图像。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，需和源图片有一样的尺寸和类型。

采用如下步骤对输入图像进行直方图均衡化。

1）计算输入图像的直方图 H。

2）进行直方图归一化，直方图的组距的和为 255。

3）计算直方图积分：

$$H'(i) = \sum_{0 \leqslant j \leqslant i} H(j)$$

4）以H'作为查询表进行图像变换：

$$dst(x,y)=H'(src(x,y))$$

言而言之，由 equalizeHist()函数实现的灰度直方图均衡化算法，就是把直方图的每个灰度级进行归一化处理，求每种灰度的累积分布，得到一个映射的灰度映射表，然后根据相应的灰度值来修正原图中的每个像素。

7.5.3 示例程序：直方图均衡化

这一节将给大家演示的示例程序简明扼要而“字字珠玑”，非常直观地演示出了如何用 equalizeHist()函数来进行图像的直方图均衡化，详细注释的代码如下。

```
//-----------------------【头文件、命名空间包含部分】-----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;


//-----------------------------【main()函数】-----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    // 【1】加载源图像
    Mat srcImage, dstImage;
    srcImage = imread( "1.jpg", 1 );
   if(!srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread 函
数指定图片存在~！ \n"); return false; }

    // 【2】转为灰度图并显示出来
    cvtColor( srcImage, srcImage, COLOR_BGR2GRAY );
    imshow( "原始图", srcImage );

    // 【3】进行直方图均衡化
    equalizeHist( srcImage, dstImage );

    // 【4】显示结果
    imshow( "经过直方图均衡化后的图", dstImage );

    // 等待用户按键退出程序
    waitKey(0);
    return 0;

}
```

看完详细注释的代码，一起看看程序运行后得到的窗口。如图 7.47、7.48 所示。

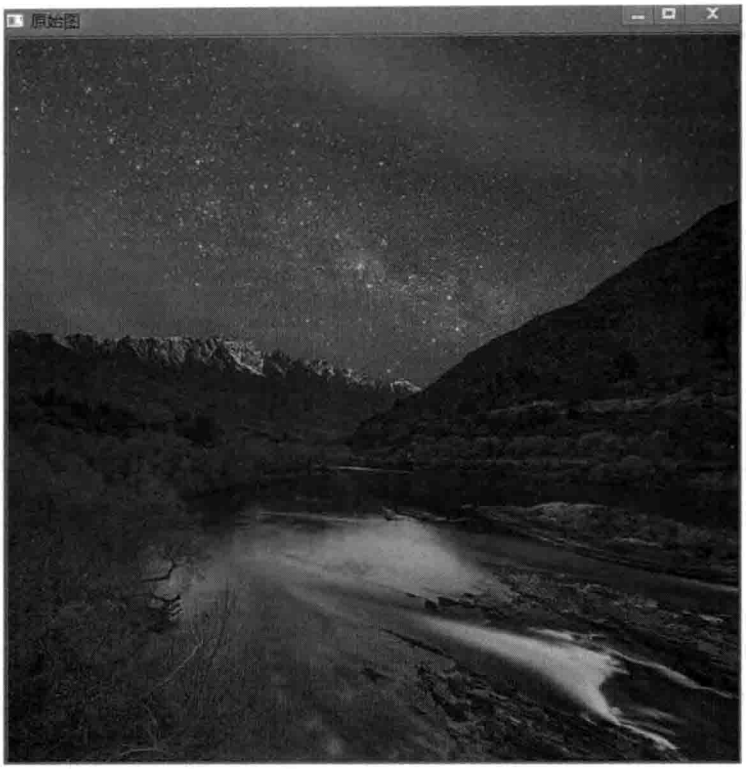


图 7.47 灰度素材图

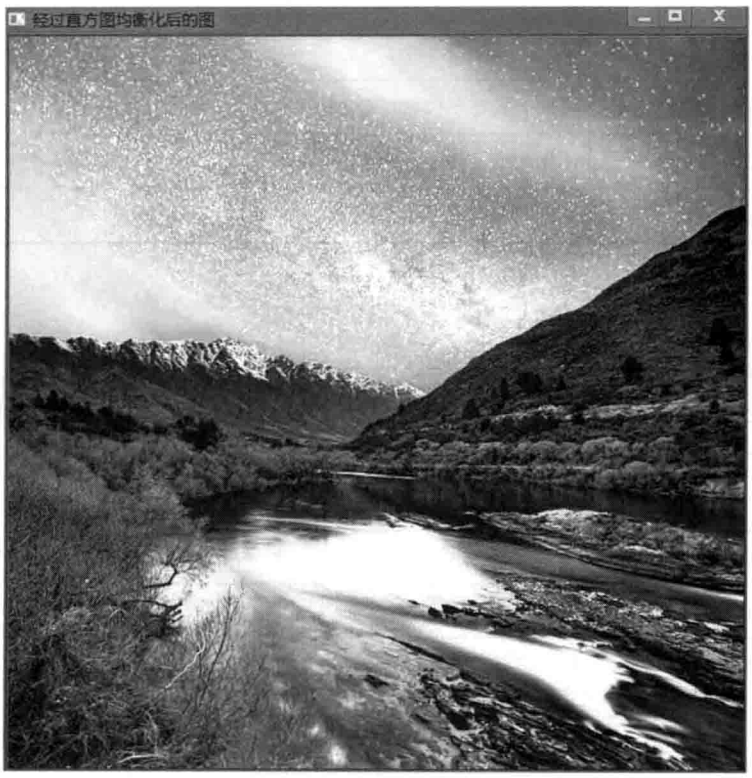


图 7.48 均衡化后的效果图

7.6 本章小结

在这章中我们学习了很多类型的图像变换方法。包括利用 OpenCV 进行边缘检测所用到的 canny 算子、sobel 算子，Laplace 算子以及 scharr 滤波器；进行图像特征提取的霍夫线变换、霍夫圆变换，重映射和仿射变换以及直方图均衡化。

本章核心函数清单

函数名称	说明	对应讲解章节
Canny	利用 Canny 算子来进行图像的边缘检测	7.1.2
Sobel	使用扩展的 Sobel 算子，来计算一阶、二阶、三阶或混合图像差分。	7.1.3
Laplacian	计算出图像经过拉普拉斯变换后的结果。	7.1.4
Scharr	使用 Scharr 滤波器运算符计算 x 或 y 方向的图像差分。	7.1.5
HoughLines	找出采用标准霍夫变换的二值图像线条	7.2.4
HoughLinesP	采用累计概率霍夫变换（PPHT）来找出二值图像中的直线	7.2.5
HoughCircles	利用霍夫变换算法检测出灰度图中的圆	7.2.8
remap	根据指定的映射形式，将源图像进行重映射几何变换	7.3.2
warpAffine	依据公式对图像做仿射变换	7.4.3
getRotationMatrix2D	计算二维旋转变换矩阵	7.4.4
equalizeHist	实现图像的直方图均衡化	7.5.2

本章示例程序清单

示例程序序号	程序说明	对应章节
56	Canny 边缘检测	7.1.2
57	Sobel 算子的使用	7.1.3
58	Laplacian 算子的使用	7.1.4
59	Scharr 滤波器	7.1.5
60	综合示例：边缘检测	7.1.6
61	标准霍夫变换：HoughLines()函数的使用	7.2.4
62	累计概率霍夫变换：HoughLinesP()函数	7.2.5

续表

示例程序序号	程序说明	对应章节
63	霍夫圆变换：HoughCircles()函数	7.2.8
64	综合示例：霍夫变换	7.2.9
65	实现重映射：remap()函数	7.3.3
66	综合示例程序：实现多种重映射	7.3.4
67	仿射变换	7.4.5
68	直方图均衡化	7.5.3

第 8 章
图像轮廓与图像分割修复

导读

虽然 Canny 之类的边缘检测算法可以根据像素之间的差异，检测出轮廓边界的像素，但是它并没有将轮廓作为一个整体。所以，下一步便是把这些边缘像素组装成轮廓。

在第 6 章介绍图像处理时，我们已经讨论了几个图像分割的算法，如图像形态学、阈值化以及金字塔分割法。而本章将进一步介绍分水岭算法，以及如何进行图像修补。

本章你将学到：

- 如何查找并绘制轮廓
- 如何寻找物体的凸包
- 如何使用多边形逼近物体
- 认识图像的矩
- 如何利用 OpenCV 进行图像修补

8.1 查找并绘制轮廓

一个轮廓一般对应一系列的点，也就是图像中的一条曲线。其表示方法可能根据不同的情况而有所不同。在 OpenCV 中，可以用 findContours()函数从二值图像中查找轮廓

8.1.1 寻找轮廓：findContours()函数

findContours()函数用于在二值图像中寻找轮廓。

```
C++: void findContours(InputOutputArray image, OutputArrayOfArrays
contours, OutputArray hierarchy, int mode, int method, Point
offset=Point())
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像，填 Mat 类的对象即可，且需为 8 位单通道图像。图像的非零像素被视为 1，0 像素值被保留为 0，所以图像为二进制。我们可以使用 compare()、inrange()、threshold()、adaptivethreshold()、canny()等函数由灰度图或彩色图创建二进制图像。此函数会在提取图像轮廓的同时修改图像的内容。
- 第二个参数，OutputArrayOfArrays 类型的 contours、检测到的轮廓、函数调用后的运算结果存在这里。每个轮廓存储为一个点向量，即用 point 类型的 vector 表示。
- 第三个参数，OutputArray 类型的 hierarchy，可选的输出向量，包含图像的拓扑信息。其作为轮廓数量的表示，包含了许多元素。每个轮廓 contours[i]对应 4 个 hierarchy 元素 hierarchy[i][0]~hierarchy[i][3]，分别表示后一个轮廓、前一个轮廓、父轮廓、内嵌轮廓的索引编号。如果没有对应项，对应的 hierarchy[i]值设置为负数。
- 第四个参数，int 类型的 mode，轮廓检索模式，取值如表 8.1 所示。

表 8.1 findContours 函数可选的轮廓检索模式

标识符	含义
RETR_EXTERNAL	表示只检测最外层轮廓。对所有轮廓，设置 hierarchy[i][2]=hierarchy[i][3]=-1
RETR_LIST	提取所有轮廓，并且放置在 list 中。检测的轮廓不建立等级关系
RETR_CCOMP	提取所有轮廓，并且将其组织为双层结构(two-level hierarchy: 顶层为连通域的外围边界，次层为孔的内层边界
RETR_TREE	提取所有轮廓，并重新建立网状的轮廓结构

- 第五个参数，int 类型的 method，为轮廓的近似办法，取值如表 8.2 所示。

表 8.2 findContours 函数可选的轮廓近似办法

标识符	含义
CHAIN_APPROX_NONE	获取每个轮廓的每个像素，相邻的两个点的像素位置差不超过 1，即 max（abs（x1-x2），abs（y2-y1））==1
CHAIN_APPROX_SIMPLE	压缩水平方向，垂直方向，对角线方向的元素，只保留该方向的终点坐标，例如一个矩形轮廓只需 4 个点来保存轮廓信息
CHAIN_APPROX_TC89_L1，CHAIN_APPROX_TC89_KCOS	使用 Teh-Chinl 链逼近算法中的一个

同样地，在表 8.1 和 8.2 中列出的宏之前加上"CV_"前缀，便是 OpenCV2 中可以使用的宏。如"RETR_CCOMP"宏的 OpenCV2 版为"CV_RETR_CCOMP"。

- 第六个参数，Point 类型的 offset，每个轮廓点的可选偏移量，有默认值 Point()。对 ROI 图像中找出的轮廓，并要在整个图像中进行分析时，这个参数便可排上用场。

findContours 经常与 drawContours 配合使用—使用用 findContours()函数检测到图像的轮廓后，便可以用 drawContours()函数将检测到的轮廓绘制出来。接下来，让我们一起看看 drawContours()函数的用法。

下面是一个调用小示例。

```
vector<vector<Point>> contours;
findContours(image,
contours, //轮廓数组
CV_RETR_EXTERNAL, //获取外轮廓
CV_CHAIN_APPROX_NONE); // 获取每个轮廓的每个像素
```

8.1.2 绘制轮廓：drawContours()函数

drawContours()函数用于在图像中绘制外部或内部轮廓。

```
C++: void drawContours(InputOutputArray image, InputArrayOfArrays
contours, int contourIdx, const Scalar& color, int thickness=1, int
lineType=8, InputArray hierarchy=noArray(), int maxLevel=INT_MAX, Point
offset=Point() )
```

- 第一个参数，InputArray 类型的 image，目标图像，填 Mat 类的对象即可。
- 第二个参数，InputArrayOfArrays 类型的 contours，所有的输入轮廓。每个轮廓存储为一个点向量，即用 point 类型的 vector 表示。
- 第三个参数，int 类型的 contourIdx，轮廓绘制的指示变量。如果其为负值，则绘制所有轮廓。
- 第四个参数，const Scalar&类型的 color，轮廓的颜色。
- 第五个参数，int thickness，轮廓线条的粗细度，有默认值 1。如果其为负值（如 thickness= cv_filled），便会绘制在轮廓的内部。可选为 FILLED 宏

（OpenCV2 版为 CV_FILLED）。

- 第六个参数，int 类型的 lineType，线条的类型，有默认值 8。取值类型如表 8.3 所示。

表 8.3 可选线性

lineType 线性	含义
8（默认值）	8 连通线型
4	4 连通线型
LINE_AA(OpenCV2 版为 CV_AA)	抗锯齿线型

- 第七个参数，InputArray 类型的 hierarchy，可选的层次结构信息，有默认值 noArray()。
- 第八个参数，int 类型的 maxLevel，表示用于绘制轮廓的最大等级，有默认值 INT_MAX
- 第九个参数，Point 类型的 offset，可选的轮廓偏移参数，用指定的偏移量 offset=（dx，dy）偏移需要绘制的轮廓，有默认值 Point()。

下面是一个调用小示例。

```
//在白色图像上绘制黑色轮廓
Mat result(image.size(),CV_8U,cv::Scalar(255));
drawContours(result,contours,
-1, // 绘制所有轮廓
Scalar(0), // 绘制颜色取黑色
3); // 轮廓线的线宽为 3
```

8.1.3 基础示例程序：轮廓查找

讲解完这两个参数，让我们看一个小型的完整示例程序，看看在 OpenCV 实际运用中，这两个函数的组合用法。此程序详细注释的代码如下。

```
#include <opencv2/opencv.hpp>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;

int main( int argc, char** argv )
{
    // 【1】载入原始图，且必须以二值图模式载入
    Mat srcImage=imread("1.jpg", 0);
    imshow("原始图",srcImage);

    //【2】初始化结果图
    Mat dstImage = Mat::zeros(srcImage.rows, srcImage.cols, CV_8UC3);

    //【3】srcImage 取大于阈值 119 的那部分
    srcImage = srcImage > 119;
    imshow( "取阈值后的原始图", srcImage );
```

```
    //【4】定义轮廓和层次结构
    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;

    //【5】查找轮廓
    //此句代码的 OpenCV2 版为：
    //findContours( srcImage, contours, hierarchy,CV_RETR_CCOMP,
CV_CHAIN_APPROX_SIMPLE );
    //此句代码的 OpenCV3 版为：
    findContours( srcImage, contours, hierarchy,RETR_CCOMP,
CHAIN_APPROX_SIMPLE );

    // 【6】遍历所有顶层的轮廓，以随机颜色绘制出每个连接组件颜色
    int index = 0;
    for( ; index >= 0; index = hierarchy[index][0] )
    {
        Scalar color( rand()&255, rand()&255, rand()&255 );
        //此句代码的 OpenCV2 版为：
        //drawContours( dstImage, contours, index, color, CV_FILLED, 8,
hierarchy );
        //此句代码的 OpenCV3 版为：
        drawContours( dstImage, contours, index, color, FILLED, 8,
hierarchy ); }

    //【7】显示最后的轮廓图
    imshow( "轮廓图", dstImage );

    waitKey(0);

}
```

程序运行截图如图 8.1、8.2、8.3 所示。

图 8.1 原始图

图 8.2 取阈值后的原始图

图 8.3 轮廓图

8.1.4 综合示例程序：查找并绘制轮廓

除了上述这个精简版的示例程序，还为大家准备了一个更加复杂一些的关于查找并绘制轮廓的综合示例程序。此程序利用了图像平滑技术（blur()函数）和边缘检测技术（canny()函数），根据滑动条的调节，可以动态地检测出图形的轮廓。

程序经过详细注释的代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;


//----------------------------【宏定义部分】----------------------------
//      描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图窗口】"                    //为窗口标题定义的宏
#define WINDOW_NAME2 "【轮廓图】"                        //为窗口标题定义的宏
```

```
//-----------------------【全局变量声明部分】-----------------------
//      描述：全局变量的声明
//-------------------------------------------------------------------
Mat g_srcImage;
Mat g_grayImage;
int g_nThresh = 80;
int g_nThresh_max = 255;
RNG g_rng(12345);
Mat g_cannyMat_output;
vector<vector<Point>> g_vContours;
vector<Vec4i> g_vHierarchy;


//-----------------------【全局函数声明部分】-----------------------
//      描述：全局函数的声明
//-------------------------------------------------------------------
static void ShowHelpText( );
void on_ThreshChange(int, void* );


//-----------------------------【main()函数】-----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------
int main( int argc, char** argv )
{
    //【0】改变 console 字体颜色
    system("color 1F");

    //【0】显示欢迎和帮助文字
    ShowHelpText( );

    // 加载源图像
    g_srcImage = imread( "1.jpg", 1 );
    if(!g_srcImage.data )
    {
printf("读取图片错误，请确定目录下是否有 imread 函数指定的图片存在~！ \n");
return false;
    }

    // 转成灰度并模糊化降噪
    cvtColor( g_srcImage, g_grayImage, COLOR_BGR2GRAY );
    blur( g_grayImage, g_grayImage, Size(3,3) );

    // 创建窗口
    namedWindow( WINDOW_NAME1, WINDOW_AUTOSIZE );
    imshow( WINDOW_NAME1, g_srcImage );

    //创建滚动条并初始化
    createTrackbar( "canny 阈值", WINDOW_NAME1, &g_nThresh,
g_nThresh_max, on_ThreshChange );
    on_ThreshChange( 0, 0 );
```

```
    waitKey(0);
    return(0);
}

//----------------------【on_ThreshChange()函数】----------------------
//      描述：回调函数
//-----------------------------------------------------------------------
void on_ThreshChange(int, void* )
{

    // 用Canny算子检测边缘
    Canny( g_grayImage, g_cannyMat_output, g_nThresh, g_nThresh*2, 3 );

    // 寻找轮廓
    findContours( g_cannyMat_output, g_vContours, g_vHierarchy,
RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0) );

    // 绘出轮廓
    Mat drawing = Mat::zeros( g_cannyMat_output.size(), CV_8UC3 );
    for( int i = 0; i< g_vContours.size(); i++ )
    {
        Scalar color = Scalar( g_rng.uniform(0, 255),
g_rng.uniform(0,255), g_rng.uniform(0,255) );//任意值
        drawContours(drawing, g_vContours, i, color, 2, 8, g_vHierarchy,
0, Point() );
    }

    // 显示效果图
    imshow( WINDOW_NAME2, drawing );
}


//-----------------------【ShowHelpText()函数】-----------------------
//      描述：输出一些帮助信息
//----------------------------------------------------------------------
static void ShowHelpText()
{
    //输出欢迎信息和OpenCV版本
    printf("\n\n\t\t\t非常感谢购买《OpenCV3编程入门》一书！\n");
    printf("\n\n\t\t\t此为本书OpenCV3版的第70个配套示例程序\n");
    printf("\n\n\t\t\t    当前使用的OpenCV版本为：" CV_VERSION );
    printf("\n\n----------------------------------------------\n");

    //输出一些帮助信息
    printf(   "\n\n\t欢迎来到【在图形中寻找轮廓】示例程序~\n\n");
    printf(   "\n\n\t按键操作说明：\n\n"
        "\t\t键盘按键任意键- 退出程序\n\n"
        "\t\t滑动滚动条-改变阈值\n" );
}
```

可以在原始图窗口中通过滑动条调节阈值，程序便会根据不同的阈值得到不同的轮廓图。运行效果图如图 8.4～8.8 所示。

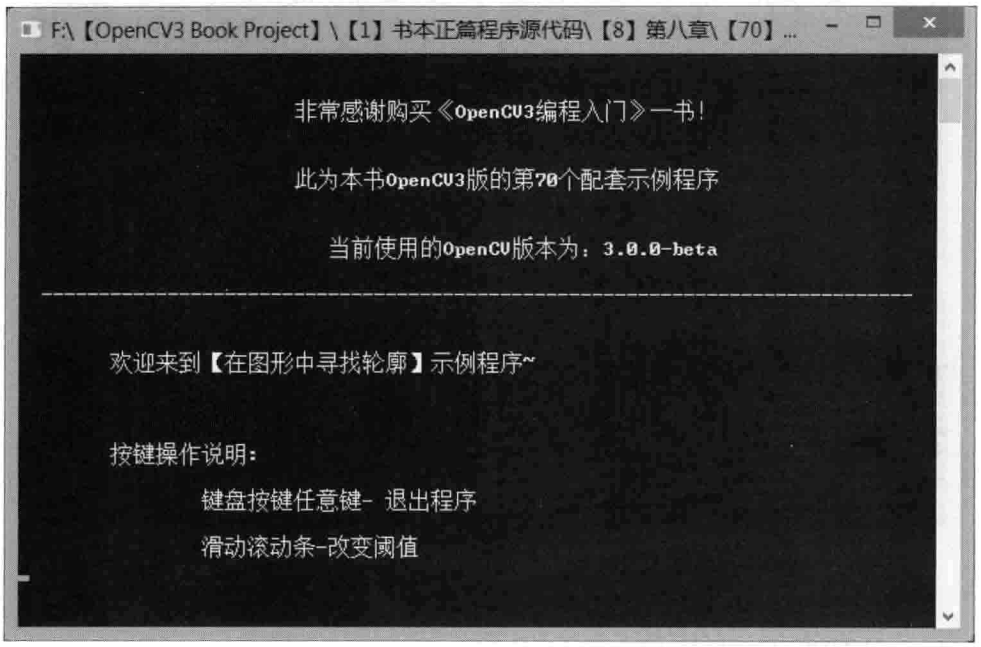


图 8.4 运行帮助文字截图

图 8.5 原始图窗口

图 8.6 阈值为 0 时的轮廓图

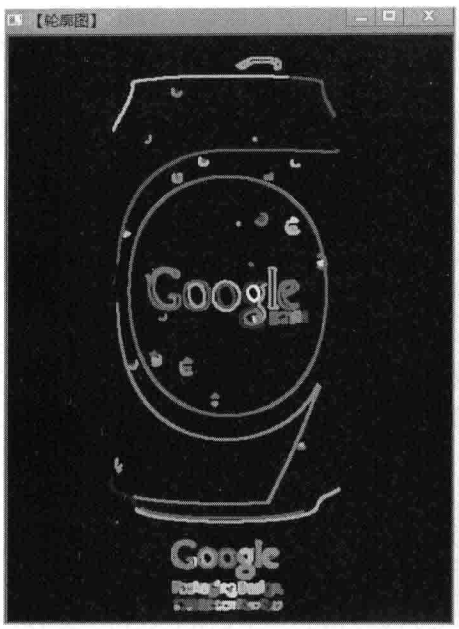


图 8.7 阈值为 80 时的轮廓图

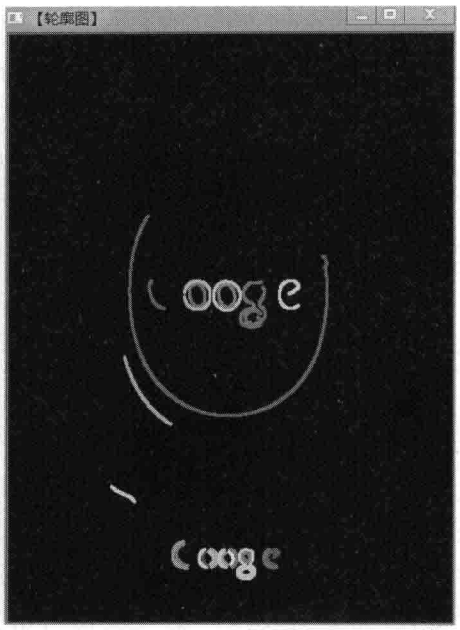


图 8.8 阈值为 187 时的轮廓图

8.2 寻找物体的凸包

8.2.1 凸包

凸包（Convex Hull）是一个计算几何（图形学）中常见的概念。简单来说，给定二维平面上的点集，凸包就是将最外层的点连接起来构成的凸多边型，它是能包含点集中所有点的。理解物体形状或轮廓的一种比较有用的方法便是计算一个物体的凸包，然后计算其凸缺陷（convexity defects）。很多复杂物体的特性能很好地被这种缺陷表现出来。

如图 8.9 所示，我们用人手图来举例说明凸缺陷这一概念。手周围深色的线描画出了凸包，A 到 H 被标出的区域是凸包的各个“缺陷”。正如看到的，这些凸度缺陷提供了手以及手状态的特征表现的方法。

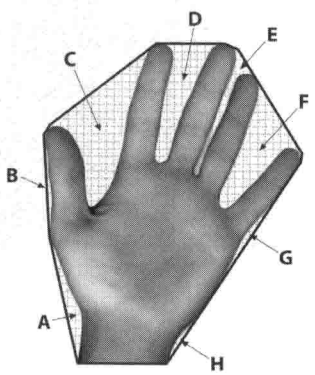


图 8.9 手掌凸缺陷示例

新版 OpenCV 中，convexHull 函数用于寻找图像点集中的凸包，我们一起来看一下这个函数。

8.2.2 寻找凸包：convexHull()函数

上文已经提到过，convexHull()函数用于寻找图像点集中的凸包，其原型声明如下。

```
C++: void convexHull(InputArray points, OutputArray hull, bool
clockwise=false, bool returnPoints=true )
```

- 第一个参数，InputArray 类型的 points，输入的二维点集，可以填 Mat 类型或者 std::vector。
- 第二个参数，OutputArray 类型的 hull，输出参数，函数调用后找到的凸包。
- 第三个参数，bool 类型的 clockwise，操作方向标识符。当此标识符为真时，输出的凸包为顺时针方向。否则，就为逆时针方向。并且是假定坐标系的 x 轴指向右，y 轴指向上方。
- 第四个参数，bool 类型的 returnPoints，操作标志符，默认值 true。当标志符为真时，函数返回各凸包的各个点。否则，它返回凸包各点的指数。当输出数组是 std::vector 时，此标志被忽略。

8.2.3 基础示例程序：凸包检测基础

为了理解凸包检测的运用方法，下面放出一个完整的示例程序。程序中会首先随机生成 3～103 个坐标值随机的彩色点，然后利用 convexHull，对由这些点链接起来的图形求凸包。

操作说明如图 8.10 所示。

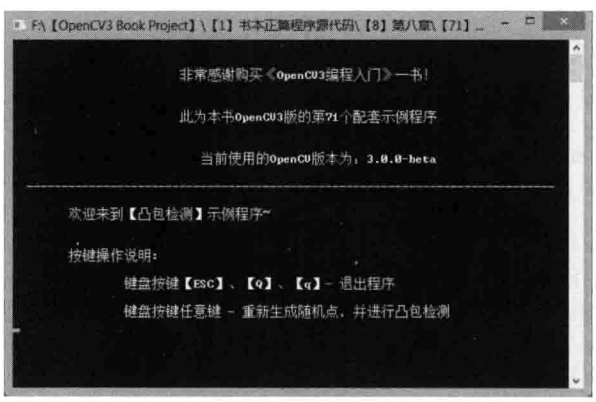


图 8.10 操作说明

此程序详细注释的程序源代码如下。

```
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
using namespace cv;
```

```
//-----------------------------【main()函数】--------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main()
{
    //初始化变量和随机值
    Mat image(600, 600, CV_8UC3);
    RNG& rng = theRNG();

    //循环，按下ESC,Q,q键程序退出，否则有键按下便一直更新
    while(1)
    {
        //参数初始化
        char key;//键值
        int count = (unsigned)rng%100 + 3;//随机生成点的数量
        vector<Point> points; //点值

        //随机生成点坐标
        for(int i = 0; i < count; i++ )
        {
         Point point;
         point.x = rng.uniform(image.cols/4, image.cols*3/4);
         point.y = rng.uniform(image.rows/4, image.rows*3/4);

         points.push_back(point);
        }

        //检测凸包
        vector<int> hull;
        convexHull(Mat(points), hull, true);

        //绘制出随机颜色的点
        image = Scalar::all(0);
        for(int i = 0; i < count; i++ )
         circle(image, points[i], 3, Scalar(rng.uniform(0, 255),
rng.uniform(0, 255), rng.uniform(0, 255)), FILLED, LINE_AA);

        //准备参数
        int hullcount = (int)hull.size();//凸包的边数
        Point point0 = points[hull[hullcount-1]];//连接凸包边的坐标点

        //绘制凸包的边
        for(int i = 0; i < hullcount; i++ )
        {
         Point point = points[hull[i]];
         line(image, point0, point, Scalar(255, 255, 255), 2, LINE_AA);
         point0 = point;
        }

        //显示效果图
```

```
        imshow("凸包检测示例", image);

        //按下 ESC,Q,或者 q, 程序退出
        key = (char)waitKey();
        if( key == 27 || key == 'q' || key == 'Q' )
         break;
    }

    return 0;
}
```

程序的运行截图如图 8.11、8.12 所示。

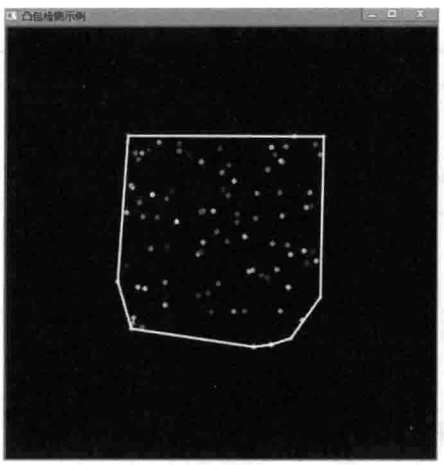


图 8.11 运行截图（1）

图 8.12 运行截图（2）

8.2.4 综合示例程序：寻找和绘制物体的凸包

这一节的综合示例程序，依然是结合滑动条，通过滑动条控制阈值，来得到不同的凸包检测效果图。程序详细注释的源代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//----------------------------【宏定义部分】-----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图窗口】"                    //为窗口标题定义
的宏
#define WINDOW_NAME2 "【效果图窗口】"                    //为窗口标题定义
的宏

//---------------------------【全局变量声明部分】-------------------------
```

```
//  描述：全局变量的声明
//-----------------------------------------------------------------------
Mat g_srcImage; Mat g_grayImage;
int g_nThresh = 50;
int g_maxThresh = 255;
RNG g_rng(12345);
Mat srcImage_copy = g_srcImage.clone();
Mat g_thresholdImage_output;
vector<vector<Point> > g_vContours;
vector<Vec4i> g_vHierarchy;

//-------------------------【全局函数声明部分】-------------------------
//   描述：全局函数的声明
//-----------------------------------------------------------------------
static void ShowHelpText();
void on_ThreshChange(int, void* );

//---------------------------【main()函数】---------------------------
//   描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main()
{
    // 加载源图像
    g_srcImage = imread( "1.jpg", 1 );

    // 将原图转换成灰度图并进行模糊降噪
    cvtColor( g_srcImage, g_grayImage, COLOR_BGR2GRAY );
    blur( g_grayImage, g_grayImage, Size(3,3) );

    // 创建原图窗口并显示
    namedWindow( WINDOW_NAME1, WINDOW_AUTOSIZE );
    imshow( WINDOW_NAME1, g_srcImage );

    //创建滚动条
    createTrackbar( " 阈值:", WINDOW_NAME1, &g_nThresh, g_maxThresh,
on_ThreshChange );
    on_ThreshChange( 0, 0 );//调用一次进行初始化

    waitKey(0);
    return(0);
}

//----------------------【thresh_callback()函数】----------------------
//      描述：回调函数
//-----------------------------------------------------------------------
void on_ThreshChange(int, void* )
{
    // 对图像进行二值化，控制阈值
    threshold( g_grayImage, g_thresholdImage_output, g_nThresh, 255,
THRESH_BINARY );

    // 寻找轮廓
```

```
    findContours( g_thresholdImage_output, g_vContours, g_vHierarchy,
RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0) );

    // 遍历每个轮廓，寻找其凸包
    vector<vector<Point> >hull( g_vContours.size() );
    for( unsigned int i = 0; i < g_vContours.size(); i++ )
    {
        convexHull( Mat(g_vContours[i]), hull[i], false );
    }

    // 绘出轮廓及其凸包
    Mat drawing = Mat::zeros( g_thresholdImage_output.size(), CV_8UC3 );
    for(unsigned  int i = 0; i< g_vContours.size(); i++ )
    {
        Scalar color = Scalar( g_rng.uniform(0, 255),
g_rng.uniform(0,255), g_rng.uniform(0,255) );
        drawContours( drawing, g_vContours, i, color, 1, 8,
vector<Vec4i>(), 0, Point() );
        drawContours( drawing, hull, i, color, 1, 8, vector<Vec4i>(), 0,
Point() );
    }

    // 显示效果图
    imshow( WINDOW_NAME2, drawing );
}
```

程序运行后，可以在原始图窗口中调节阈值，改变全局变量 g_nThresh 的值，以改变 g_thresholdImage_output 参数，最后 findContours 以此参数为输入，查找不同的轮廓图。经过一系列的处理，最后得到的凸包图形便有精细度上的差异。运行效果如图 8.13～8.16 所示。

图 8.13 原始图窗口

图 8.14 阈值为 50 时的效果图

图 8.15 阈值为 113 时的效果图

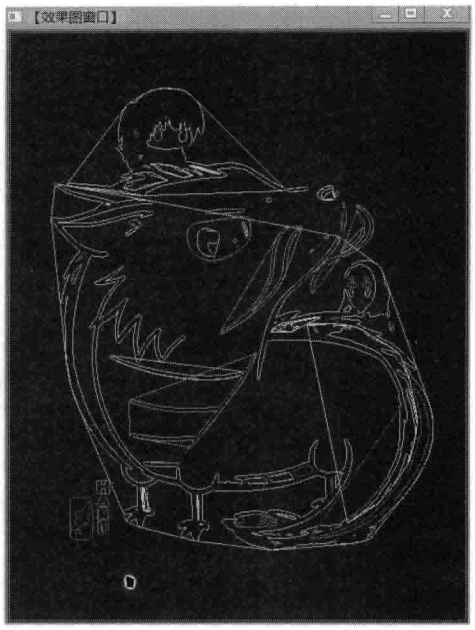


图 8.16 阈值为 164 时的效果图

8.3 使用多边形将轮廓包围

在实际应用中，常常会有将检测到的轮廓用多边形表示出来的需求。本节就为大家讲解如何用多边形来表示出轮廓，或者说如何根据轮廓提取出多边形。先让我们一起学习用 OpenCV 创建包围轮廓的多边形边界时会接触到的一些函数。

8.3.1 返回外部矩形边界：boundingRect()函数

此函数计算并返回指定点集最外面（up-right）的矩形边界 。

```
C++: Rect boundingRect(InputArray points)
```

其唯一的一个参数为输入的二维点集，可以是 std::vector 或 Mat 类型。

8.3.2 寻找最小包围矩形：minAreaRect()函数

此函数用于对给定的 2D 点集，寻找可旋转的最小面积的包围矩形。

```
C++: RotatedRect minAreaRect(InputArray points)
```

其唯一的一个参数为输入的二维点集，可以为 std::vector<>或 Mat 类型。

8.3.3 寻找最小包围圆形：minEnclosingCircle()函数

minEnclosingCircle 函数的功能是利用一种迭代算法，对给定的 2D 点集，去寻找面积最小的可包围它们的圆形。

```
C++: void minEnclosingCircle(InputArray points, Point2f& center, float&
radius)
```

- 第一个参数，InputArray 类型的 points，输入的二维点集，可以为 std::vector<> 或 Mat 类型。
- 第二个参数，Point2f&类型的 center，圆的输出圆心。
- 第三个参数，float&类型的 radius，圆的输出半径。

8.3.4 用椭圆拟合二维点集：fitEllipse()函数

此函数的作用是用椭圆拟合二维点集。

```
C++: RotatedRect fitEllipse(InputArray points)
```

其唯一的一个参数为输入的二维点集，可以为 std::vector<>或 Mat 类型。

8.3.5 逼近多边形曲线：approxPolyDP()函数

approxPolyDP 函数的作用是用指定精度逼近多边形曲线。

```
C++: void approxPolyDP(InputArray curve, OutputArray approxCurve,
double epsilon, bool closed)
```

- 第一个参数，InputArray 类型的 curve，输入的二维点集，可以为 std::vecto 或 Mat 类型。
- 第二个参数，OutputArray 类型的 approxCurve，多边形逼近的结果，其类型应该和输入的二维点集的类型一致。
- 第三个参数，double 类型的 epsilon，逼近的精度，为原始曲线和即近似曲线间的最大值。
- 第四个参数，bool 类型的 closed，如果其为真，则近似的曲线为封闭曲线（第一个顶点和最后一个顶点相连），否则，近似的曲线曲线不封闭。

讲解完上述的几个函数，下面我们来通过完整的示例程序理解所学。笔者为大家准备了两个示例程序，分别为用矩形和圆形边界包围生成的轮廓。

8.3.6 基础示例程序：创建包围轮廓的矩形边界

此示例程序以 minAreaRect 函数为核心，先随机生成 3～103 个彩色点，然后绘制一个可以旋转的最小的矩形，将这些点全部包含进去。操作依然是通过键盘按下任意键来重新生成随机点，并寻找最小面积的包围矩形。

详细注释的程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;
using namespace std;
```

```
int main()
{
    //初始化变量和随机值
    Mat image(600, 600, CV_8UC3);
    RNG& rng = theRNG();

    //循环，按下ESC,Q,q键程序退出，否则有键按下便一直更新
    while(1)
    {
        //参数初始化
        int count = rng.uniform(3, 103);//随机生成点的数量
        vector<Point> points;//点值

        //随机生成点坐标
        for(int  i = 0; i < count; i++ )
        {

         Point point;
         point.x = rng.uniform(image.cols/4, image.cols*3/4);
         point.y = rng.uniform(image.rows/4, image.rows*3/4);

         points.push_back(point);
        }

        //对给定的 2D 点集，寻找最小面积的包围矩形
        RotatedRect box = minAreaRect(Mat(points));
        Point2f vertex[4];
        box.points(vertex);

        //绘制出随机颜色的点
        image = Scalar::all(0);
        for( int i = 0; i < count; i++ )
         circle( image, points[i], 3, Scalar(rng.uniform(0, 255),
rng.uniform(0, 255), rng.uniform(0, 255)), FILLED, LINE AA );


        //绘制出最小面积的包围矩形
        for( int i = 0; i < 4; i++ )
         line(image, vertex[i], vertex[(i+1)%4], Scalar(100, 200, 211),
2, LINE AA);

        //显示窗口
        imshow( "矩形包围示例", image );

        //按下ESC,Q,或者q，程序退出
        char key = (char)waitKey();
        if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC'
         break;
    }

    return 0;
}
```

程序注释已经十分详细。我们运行程序，便可以看到随机生成的彩色的点，以及包围着它们的矩形。按下除【ESC】、【Q】、【q】之外的任意键，便可以得到新的彩色的点和矩形。如图 8.17、8.18 所示。

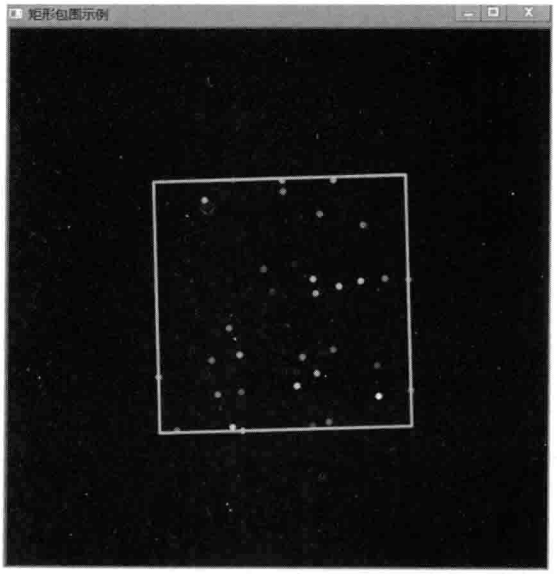


图 8.17 程序截图（1）

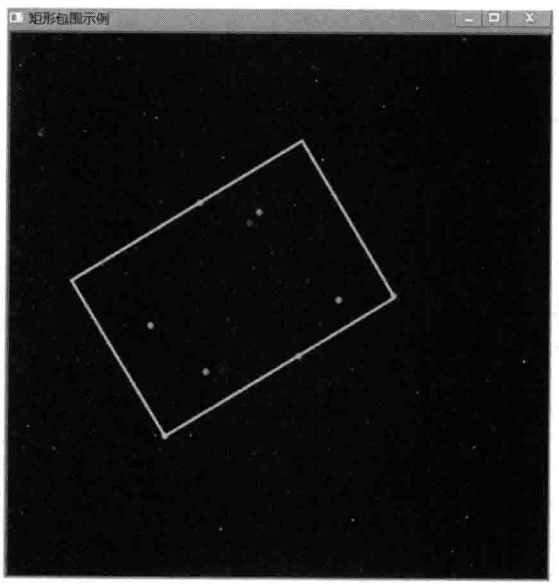


图 8.18 程序截图（2）

8.3.7 基础示例程序：创建包围轮廓的圆形边界

本小节给出的程序和上一小节的程序类似，只不过核心函数替换为 minEnclosingCircle，依然是先随机生成 3～103 个彩色点，不同的是绘制的是一个圆而不是矩形，将这些点组成的轮廓包含进去。为了大家学习方便，这里依然是贴出此程序详细注释的源代码。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------

#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;
using namespace std;

int main()
{
    //初始化变量和随机值
    Mat image(600, 600, CV_8UC3);
    RNG& rng = theRNG();

    //循环，按下ESC,Q,q键程序退出，否则有键按下便一直更新
    while(1)
    {
        //参数初始化
        int count = rng.uniform(3, 103);//随机生成点的数量
        vector<Point> points;//点值

        //随机生成点坐标
        for(int  i = 0; i < count; i++ )
        {

         Point point;
         point.x = rng.uniform(image.cols/4, image.cols*3/4);
         point.y = rng.uniform(image.rows/4, image.rows*3/4);

         points.push_back(point);
        }

        //对给定的 2D 点集，寻找最小面积的包围圆
        Point2f center;
        float radius = 0;
        minEnclosingCircle(Mat(points), center, radius);

        //绘制出随机颜色的点
        image = Scalar::all(0);
        for( int i = 0; i < count; i++ )
         circle( image, points[i], 3, Scalar(rng.uniform(0, 255),
rng.uniform(0, 255), rng.uniform(0, 255)), FILLED, LINE_AA );


        //绘制出最小面积的包围圆
        circle(image, center, cvRound(radius), Scalar(rng.uniform(0,
255), rng.uniform(0, 255), rng.uniform(0, 255)), 2, LINE_AA);

        //显示窗口
        imshow( "圆形包围示例", image );
```

```
        //按下 ESC,Q,或者 q, 程序退出
        char key = (char)waitKey();
        if( key == 27 || key == 'q' || key == 'Q' ) // 'ESC'
         break;
    }

    return 0;
}
```

程序注释依然是十分详细。我们运行程序，便可以看到随机生成的彩色的点，以及包围着它们的圆。按下除【ESC】、【Q】、【q】之外的任意键，便可以得到新的彩色的点和圆。如图 8.19、8.20 所示。

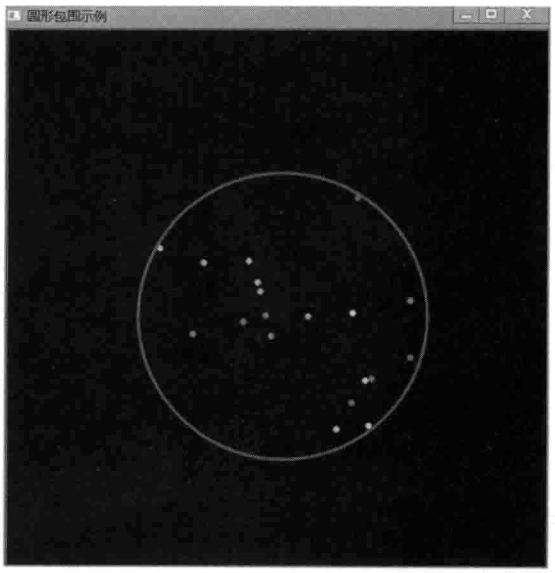


图 8.19 程序截图（1）

图 8.20 程序截图（2）

8.3.8 综合示例程序：使用多边形包围轮廓

经过上述两个基础示例程序的学习，相信大家应该对 minAreaRect 和 minEnclosingCircle 函数的用法有了一定的认识。这两个示例程序中，处理的轮廓都是程序自己随机生成的点。在接下来这个综合一点的示例程序中，让我们载入一幅图像，用上文中学到的函数来创建包围轮廓的矩形和圆形边界框。

程序详细注释的代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;
using namespace std;

//----------------------------【宏定义部分】----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图窗口】"        //为窗口标题定义的宏
#define WINDOW_NAME2 "【效果图窗口】"        //为窗口标题定义的宏

//-------------------------【全局变量声明部分】-------------------------
//  描述：全局变量的声明
//-------------------------------------------------------------------------
Mat g_srcImage;
Mat g_grayImage;
int g_nThresh = 50;//阈值
int g_nMaxThresh = 255;//阈值最大值
RNG g_rng(12345);//随机数生成器

//-------------------------【全局函数声明部分】-------------------------
//   描述：全局函数的声明
//-------------------------------------------------------------------------
void on_ContoursChange(int, void* );
static void ShowHelpText();

//----------------------------【main()函数】----------------------------
//   描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【0】改变 console 字体颜色
    system("color 1A");

    //【1】载入 3 通道的原图像
    g_srcImage = imread( "1.jpg", 1 );
     if(!g_srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定的图片存在~！ \n"); return false; }
```

```
    //【2】得到原图的灰度图像并进行平滑
    cvtColor( g_srcImage, g_grayImage, COLOR_BGR2GRAY );
    blur( g_grayImage, g_grayImage, Size(3,3) );

    //【3】创建原始图窗口并显示
    namedWindow( WINDOW_NAME1, WINDOW_AUTOSIZE );
    imshow( WINDOW_NAME1, g_srcImage );

    //【4】设置滚动条并调用一次回调函数
    createTrackbar( " 阈值:", WINDOW_NAME1, &g_nThresh, g_nMaxThresh,
on_ContoursChange );
    on_ContoursChange( 0, 0 );

    waitKey(0);

    return(0);
}

//----------------------【on_ContoursChange()函数】----------------------
//      描述：回调函数
//-----------------------------------------------------------------------
void on_ContoursChange(int, void* )
{
    //定义一些参数
    Mat threshold_output;
    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;

    // 使用 Threshold 检测边缘
    threshold( g_grayImage, threshold_output, g_nThresh, 255,
THRESH_BINARY );

    // 找出轮廓
    findContours( threshold_output, contours, hierarchy, RETR_TREE,
CHAIN_APPROX_SIMPLE, Point(0, 0) );

    // 多边形逼近轮廓 + 获取矩形和圆形边界框
    vector<vector<Point> > contours_poly( contours.size() );
    vector<Rect> boundRect( contours.size() );
    vector<Point2f>center( contours.size() );
    vector<float>radius( contours.size() );

    //一个循环，遍历所有部分，进行本程序最核心的操作
    for( unsigned int i = 0; i < contours.size(); i++ )
    {
        approxPolyDP( Mat(contours[i]), contours_poly[i], 3, true );//
用指定精度逼近多边形曲线
        boundRect[i] = boundingRect( Mat(contours_poly[i]) );//计算点集
的最外面（up-right）矩形边界
        minEnclosingCircle( contours_poly[i], center[i], radius[i] );//
对给定的 2D 点集，寻找最小面积的包围圆形
    }
```

```
    // 绘制多边形轮廓 + 包围的矩形框 + 圆形框
    Mat drawing = Mat::zeros( threshold_output.size(), CV_8UC3 );
    for( int unsigned i = 0; i<contours.size(); i++ )
    {
        Scalar color = Scalar( g_rng.uniform(0, 255),
g_rng.uniform(0,255), g_rng.uniform(0,255) );//随机设置颜色
        drawContours( drawing, contours_poly, i, color, 1, 8,
vector<Vec4i>(), 0, Point() );//绘制轮廓
       rectangle( drawing, boundRect[i].tl(), boundRect[i].br(), color,
2, 8, 0 );//绘制矩形
        circle( drawing, center[i], (int)radius[i], color, 2, 8, 0 );//
绘制圆
    }

    // 显示效果图窗口
    namedWindow( WINDOW_NAME2, WINDOW_AUTOSIZE );
    imshow( WINDOW_NAME2, drawing );
}
```

运行此程序。会得到两个窗口，一个带滑动条的原始图窗口，和用于显示创建了包围轮廓的矩形和圆形边界框的效果图窗口。我们可以通过调节滑动条来改变阈值，以得到不同的效果图。如图 8.21～8.24 所示。

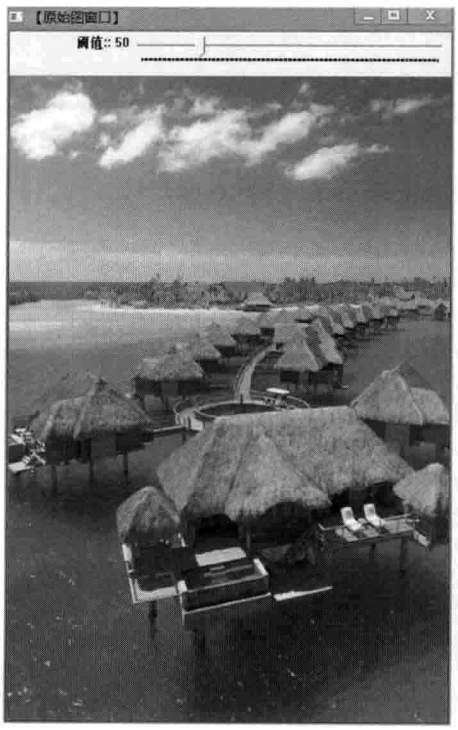


图 8.21 原始图

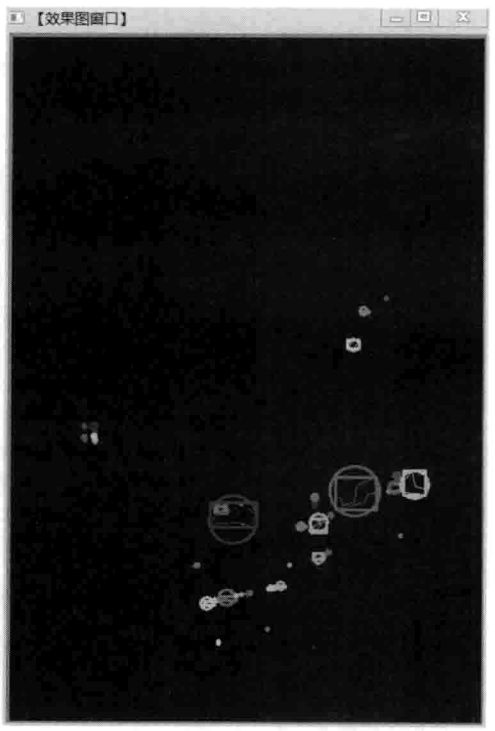


图 8.22 阈值为 21 时的效果图

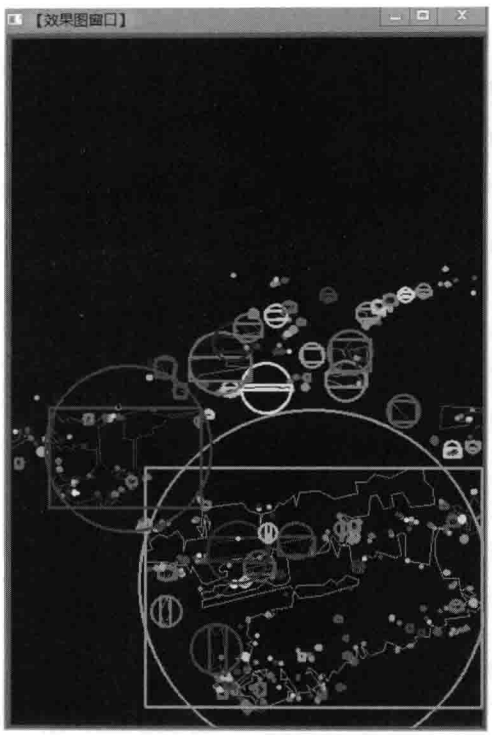


图 8.23 阈值为 55 时的效果图

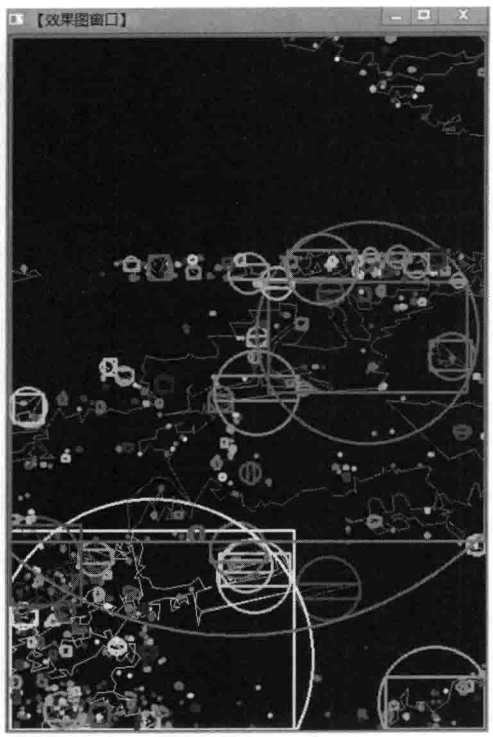


图 8.24 阈值为 103 时的效果图

8.4 图像的矩

矩函数在图像分析中有着广泛的应用，如模式识别、目标分类、目标识别与方位估计、图像编码与重构等。一个从一幅数字图形中计算出来的矩集，通常描述了该图像形状的全局特征，并提供了大量的关于该图像不同类型的几何特性信息，比如大小、位置、方向及形状等。图像矩的这种特性描述能力被广泛地应用在各种图像处理、计算机视觉和机器人技术领域的目标识别与方位估计中。一阶矩与形状有关，二阶矩显示曲线围绕直线平均值的扩展程度，三阶矩则是关于平均值的对称性的测量。由二阶矩和三阶矩可以导出一组共 7 个不变矩。而不变矩是图像的统计特性，满足平移、伸缩、旋转均不变的不变性，在图像识别领域得到了广泛的应用。

那么，在 OpenCV 中，如何计算一个图像的矩呢？一般由 moments、contourArea、arcLength 这三个函数配合求取。

- 使用 moments 计算图像所有的矩(最高到 3 阶)
- 使用 contourArea 来计算轮廓面积
- 使用 arcLength 来计算轮廓或曲线长度

下面对其进行一一剖析。

8.4.1 矩的计算：moments()函数

moments()函数用于计算多边形和光栅形状的最高达三阶的所有矩。矩用来计算形状的重心、面积，主轴和其他形状特征，如 7Hu 不变量等。

```
C++: Moments moments(InputArray array, bool binaryImage=false )
```

- 第一个参数，InputArray 类型的 array，输入参数，可以是光栅图像（单通道，8 位或浮点的二维数组）或二维数组（1N 或 N1）。
- 第二个参数，bool 类型的 binaryImage，有默认值 false。若此参数取 true，则所有非零像素为 1。此参数仅对于图像使用。

需要注意的是，此参数的返回值返回运行后的结果。

8.4.2 计算轮廓面积：contourArea()函数

contourArea()函数用于计算整个轮廓或部分轮廓的面积

```
C++: double contourArea(InputArray contour, bool oriented=false)
```

- 第一个参数，InputArray 类型的 contour，输入的向量，二维点（轮廓顶点），可以为 std::vector 或 Mat 类型。
- 第二个参数，bool 类型的 oriented，面向区域标识符。若其为 true，该函数返回一个带符号的面积值，其正负取决于轮廓的方向（顺时针还是逆时针）。根据这个特性我们可以根据面积的符号来确定轮廓的位置。需要注意的是，这个参数有默认值 false，表示以绝对值返回，不带符号。

其调用示例如下。

```
vector<Point> contour;
contour.push_back(Point2f(0, 0));
contour.push_back(Point2f(10, 0));
contour.push_back(Point2f(10, 10));
contour.push_back(Point2f(5, 4));

double area0 = contourArea(contour);
vector<Point> approx;
approxPolyDP(contour, approx, 5, true);
double area1 = contourArea(approx);

cout << "area0 =" << area0 << endl <<
       "area1 =" << area1 << endl <<
       "approx poly vertices" << approx.size() << endl;
```

8.4.3 计算轮廓长度：arcLength()函数

arcLength()函数用于计算封闭轮廓的周长或曲线的长度。

```
C++: double arcLength(InputArray curve, bool closed)
```

- 第一个参数，InputArray 类型的 curve，输入的二维点集，可以为 std::vector

或 Mat 类型。

- 第二个参数，bool 类型的 closed，一个用于指示曲线是否封闭的标识符，有默认值 closed，表示曲线封闭。

8.4.4 综合示例程序：查找和绘制图像轮廓矩

学习完函数的讲解，让我们一起通过一个综合的示例程序，真正了解本节内容的实战用法。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//----------------------------【宏定义部分】-----------------------------
//      描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图】"                    //为窗口标题定义的宏
#define WINDOW_NAME2 "【图像轮廓】"          //为窗口标题定义的宏


//-------------------------【全局变量声明部分】--------------------------
//      描述：全局变量的声明
//-------------------------------------------------------------------------
Mat g_srcImage; Mat g_grayImage;
int g_nThresh = 100;
int g_nMaxThresh = 255;
RNG g_rng(12345);
Mat g_cannyMat_output;
vector<vector<Point> > g_vContours;
vector<Vec4i> g_vHierarchy;

//-------------------------【全局变量声明部分】--------------------------
//      描述：全局变量的声明
//-------------------------------------------------------------------------
void on_ThreshChange(int, void* );
static void ShowHelpText();

//----------------------------【main()函数】-----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main( int argc, char** argv )
{
    //【0】改变 console 字体颜色
    system("color 1E");

    // 读入原图像，返回 3 通道图像数据
```

```
    g_srcImage = imread( "1.jpg", 1 );

    // 把原图像转化成灰度图像并进行平滑
    cvtColor( g_srcImage, g_grayImage, COLOR_BGR2GRAY );
    blur( g_grayImage, g_grayImage, Size(3,3) );

    // 创建新窗口
    namedWindow( WINDOW_NAME1, WINDOW_AUTOSIZE );
    imshow( WINDOW_NAME1, g_srcImage );

    //创建滚动条并进行初始化
    createTrackbar( " 阈值", WINDOW_NAME1, &g_nThresh, g_nMaxThresh,
on_ThreshChange );
    on_ThreshChange( 0, 0 );

    waitKey(0);
    return(0);
}

//-----------------------【on_ThreshChange()函数】----------------------
//      描述：回调函数
//-----------------------------------------------------------------------
void on_ThreshChange(int, void* )
{
    // 使用 Canndy 检测边缘
    Canny( g_grayImage, g_cannyMat_output, g_nThresh, g_nThresh*2, 3 );

    // 找到轮廓
    findContours( g_cannyMat_output, g_vContours, g_vHierarchy,
RETR_TREE, CHAIN_APPROX_SIMPLE, Point(0, 0) );

    // 计算矩
    vector<Moments> mu(g_vContours.size() );
    for(unsigned int i = 0; i < g_vContours.size(); i++ )
    { mu[i] = moments( g_vContours[i], false ); }

    //  计算中心矩
    vector<Point2f> mc( g_vContours.size() );
    for( unsigned int i = 0; i < g_vContours.size(); i++ )
    { mc[i] = Point2f( static_cast<float>(mu[i].m10/mu[i].m00),
static_cast<float>(mu[i].m01/mu[i].m00 )); }

    // 绘制轮廓
    Mat drawing = Mat::zeros( g_cannyMat_output.size(), CV_8UC3 );
    for( unsigned int i = 0; i< g_vContours.size(); i++ )
    {
        Scalar color = Scalar( g_rng.uniform(0, 255),
g_rng.uniform(0,255), g_rng.uniform(0,255) );//随机生成颜色值
        drawContours( drawing, g_vContours, i, color, 2, 8, g_vHierarchy,
0, Point() );//绘制外层和内层轮廓
        circle( drawing, mc[i], 4, color, -1, 8, 0 );;//绘制圆
    }
```

```
    // 显示到窗口中
    namedWindow( WINDOW_NAME2, WINDOW_AUTOSIZE );
    imshow( WINDOW_NAME2, drawing );

    // 通过 m00 计算轮廓面积并且和 OpenCV 函数比较
    printf("\t 输出内容: 面积和轮廓长度\n");
    for(unsigned int i = 0; i< g_vContours.size(); i++ )
    {
        printf(" >通过 m00 计算出轮廓[%d]的面积: (M_00) = %.2f \n OpenCV 函
数计算出的面积=%.2f , 长度: %.2f \n\n", i, mu[i].m00,
contourArea(g_vContours[i]), arcLength( g_vContours[i], true ) );
        Scalar color = Scalar( g_rng.uniform(0, 255),
g_rng.uniform(0,255), g_rng.uniform(0,255) );
        drawContours( drawing, g_vContours, i, color, 2, 8, g_vHierarchy,
0, Point() );
        circle( drawing, mc[i], 4, color, -1, 8, 0 );
    }
}
```

运行程序，便可检测和计算出各个区域的轮廓面积和长度，在窗口中输出。如图 8.25～8.29 所示。

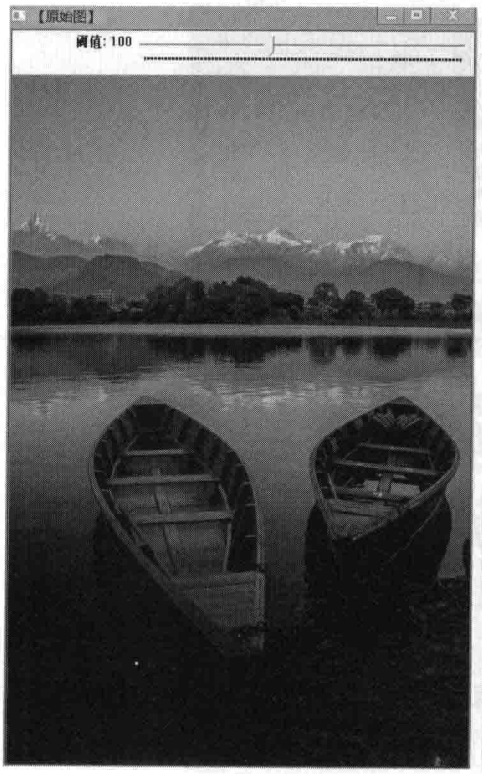


图 8.25 原始图

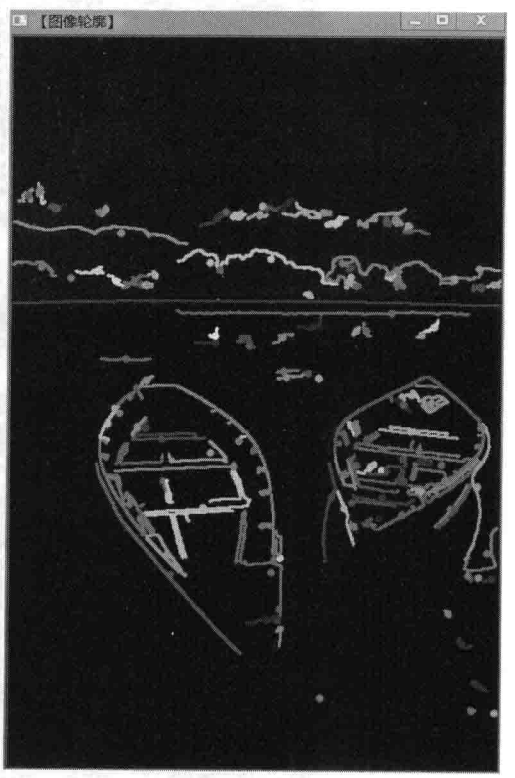


图 8.26 阈值为 100 时的图像轮廓

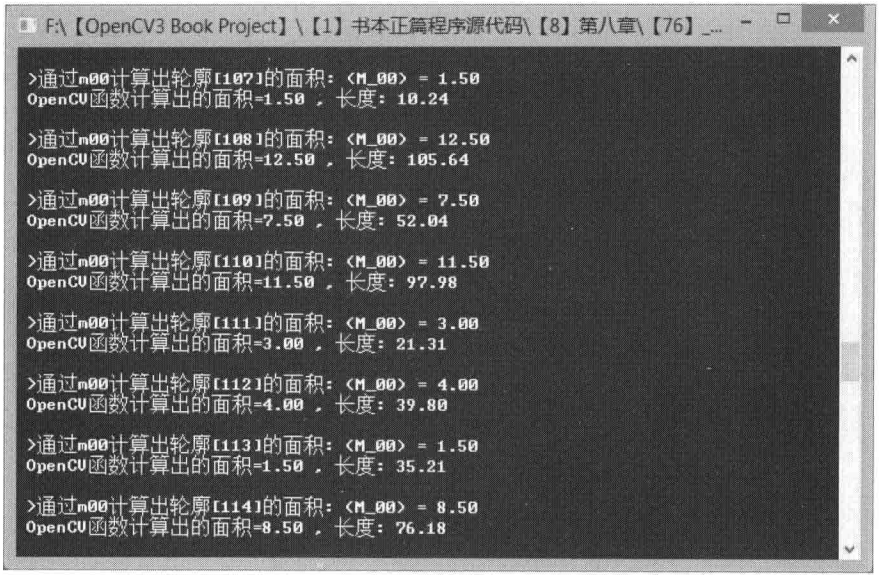


图 8.27 阈值为 100 时的输出窗口

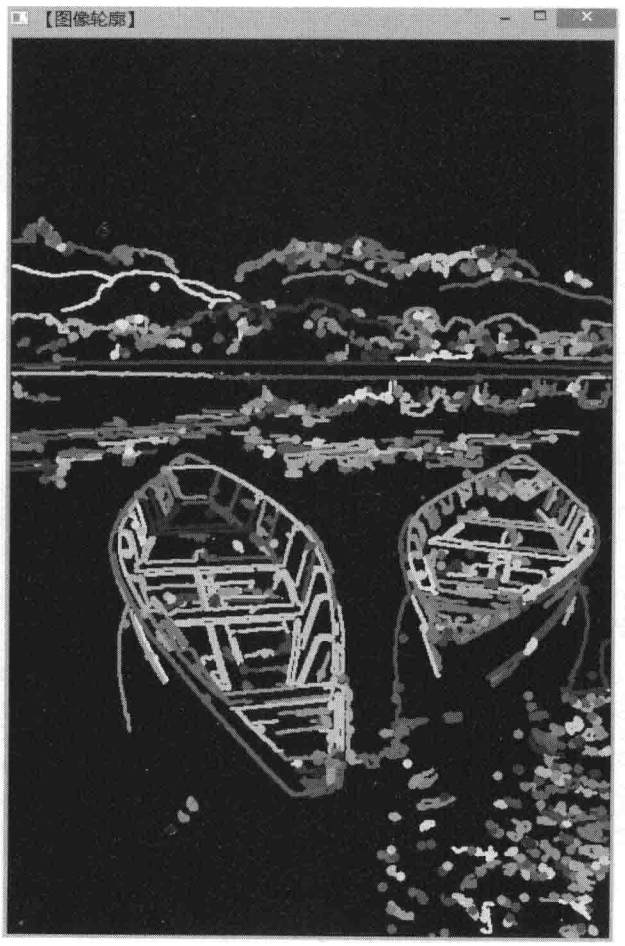


图 8.28 阈值为 40 时的图像轮廓

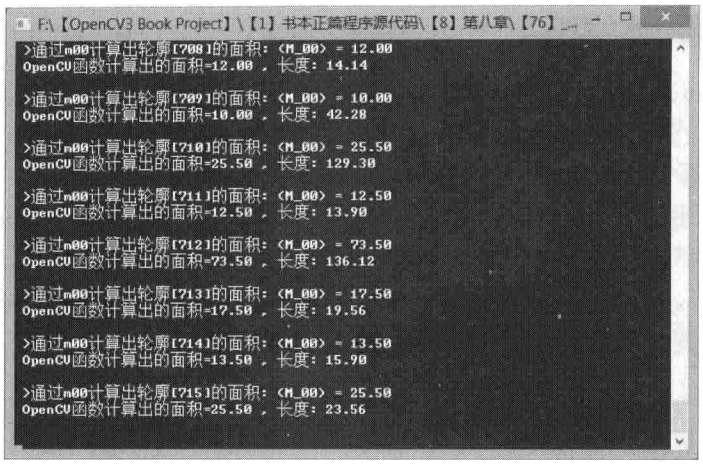


图 8.29 阈值为 40 时的输出窗口

8.5 分水岭算法

在许多实际运用中，我们需要分割图像，但无法从背景图像中获得有用信息。分水岭算法（watershed algorithm）在这方面往往是非常有效的。此算法可以将图像中的边缘转化成“山脉”，将均匀区域转化为“山谷”，这样有助于分割目标。

分水岭算法，是一种基于拓扑理论的数学形态学的分割方法，其基本思想是把图像看作是测地学上的拓扑地貌，图像中每一点像素的灰度值表示该点的海拔高度，每一个局部极小值及其影响区域称为集水盆，而集水盆的边界则形成分水岭。分水岭的概念和形成可以通过模拟浸入过程来说明：在每一个局部极小值表面，刺穿一个小孔，然后把整个模型慢慢浸入水中，随着浸入的加深，每一个局部极小值的影响域慢慢向外扩展，在两个集水盆汇合处构筑大坝，即形成分水岭。

分水岭的计算过程是一个迭代标注过程。分水岭比较经典的计算方法是由 L. Vincent 提出的。在该算法中，分水岭计算分两个步骤：一个是排序过程，一个是淹没过程。首先对每个像素的灰度级进行从低到高的排序，然后在从低到高实现淹没的过程中，对每一个局部极小值在 h 阶高度的影响域采用先进先出（FIFO）结构进行判断及标注。分水岭变换得到的是输入图像的集水盆图像，集水盆之间的边界点，即为分水岭。显然，分水岭表示的是输入图像的极大值点。

也就是说，分水岭算法首先计算灰度图像的梯度；这对图像中的“山谷”或没有纹理的“盆地”（亮度值低的点）的形成是很有效的，也对“山头”或图像中有主导线段的“山脉”（山脊对应的边缘）的形成有效。然后开始从用户指定点（或者算法得到点）开始持续“灌注”盆地直到这些区域连成一片。基于这样产生的标记就可以把区域合并到 0 一起，合并后的区域又通聚集的方式进行分割，好像图像被“填充”起来一样。

8.5.1 实现分水岭算法：watershed()函数

函数 watershed 实现的分水岭算法是基于标记的分割算法中的一种。在把图像传给函数之前，我们需要大致勾画标记出图像中的期望进行分割的区域，它们被标记为正指数。所以，每一个区域都会被标记为像素值 1、2、3 等，表示成为一个或者多个连接组件。这些标记的值可以使用 findContours()函数和 drawContours()函数由二进制的掩码检索出来。不难理解，这些标记就是即将绘制出来的分割区域的“种子”，而没有标记清楚的区域，被置为 0。在函数输出中，每一个标记中的像素被设置为“种子”的值，而区域间的值被设置为-1。

```
C++: void watershed(InputArray image, InputOutputArray markers)
```

第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，且需为 8 位三通道的彩色图像。

第二个参数，InputOutputArray 类型的 markers，函数调用后的运算结果存在这里，输入/输出 32 位单通道图像的标记结果。即这个参数用于存放函数调用后的输出结果，需和源图片有一样的尺寸和类型。

8.5.2 综合示例程序：分水岭算法

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//----------------------------【宏定义部分】----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME "【程序窗口 1】"        //为窗口标题定义的宏


//-----------------------【全局函变量声明部分】-----------------------
//      描述：全局变量的声明
//-------------------------------------------------------------------------
Mat g_maskImage, g_srcImage;
Point prevPt(-1, -1);

//-----------------------【全局函数声明部分】-----------------------
//      描述：全局函数的声明
//-------------------------------------------------------------------------
static void ShowHelpText();
static void on_Mouse( int event, int x, int y, int flags, void* );


//-----------------------------【main()函数】-----------------------------
```

```
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main( int argc, char** argv )
{
    //【1】载入原图并显示，初始化掩膜和灰度图
    g_srcImage = imread("1.jpg", 1);
    imshow( WINDOW_NAME, g_srcImage );
    Mat srcImage,grayImage;
    g_srcImage.copyTo(srcImage);
    cvtColor(g_srcImage, g_maskImage, COLOR_BGR2GRAY);
    cvtColor(g_maskImage, grayImage, COLOR_GRAY2BGR);
    g_maskImage = Scalar::all(0);

    //【2】设置鼠标回调函数
    setMouseCallback( WINDOW_NAME, on_Mouse, 0 );

    //【3】轮询按键，进行处理
    while(1)
    {
        //获取键值
        int c = waitKey(0);

        //若按键键值为ESC时，退出
        if( (char)c == 27 )
         break;

        //按键键值为2时，恢复源图
        if( (char)c == '2' )
        {
         g_maskImage = Scalar::all(0);
         srcImage.copyTo(g_srcImage);
         imshow( "image", g_srcImage );
        }

        //若检测到按键值为1或者空格，则进行处理
        if( (char)c == '1' || (char)c == ' ' )
        {
         //定义一些参数
         int i, j, compCount = 0;
         vector<vector<Point> > contours;
         vector<Vec4i> hierarchy;

         //寻找轮廓
         findContours(g_maskImage, contours, hierarchy, CV_RETR_CCOMP,
CV_CHAIN_APPROX_SIMPLE);

         //轮廓为空时的处理
         if( contours.empty() )
             continue;

         //复制掩膜
         Mat maskImage(g_maskImage.size(), CV_32S);
```

```
        maskImage = Scalar::all(0);

        //循环绘制出轮廓
        for( int index = 0; index >= 0; index = hierarchy[index][0],
compCount++ )
            drawContours(maskImage, contours, index,
Scalar::all(compCount+1), -1, 8, hierarchy, INT_MAX);

        //compCount 为零时的处理
        if( compCount == 0 )
            continue;

        //生成随机颜色
        vector<Vec3b> colorTab;
        for( i = 0; i < compCount; i++ )
        {
            int b = theRNG().uniform(0, 255);
            int g = theRNG().uniform(0, 255);
            int r = theRNG().uniform(0, 255);

            colorTab.push_back(Vec3b((uchar)b, (uchar)g, (uchar)r));
        }

        //计算处理时间并输出到窗口中
        double dTime = (double)getTickCount();
        watershed( srcImage, maskImage );
        dTime = (double)getTickCount() - dTime;
        printf( "\t 处理时间 = %gms\n",
dTime*1000./getTickFrequency() );

        //双层循环，将分水岭图像遍历存入 watershedImage 中
        Mat watershedImage(maskImage.size(), CV_8UC3);
        for( i = 0; i < maskImage.rows; i++ )
            for( j = 0; j < maskImage.cols; j++ )
            {
                int index = maskImage.at<int>(i,j);
                if( index == -1 )
                    watershedImage.at<Vec3b>(i,j) =
Vec3b(255,255,255);
                else if( index <= 0 || index > compCount )
                    watershedImage.at<Vec3b>(i,j) = Vec3b(0,0,0);
                else
                    watershedImage.at<Vec3b>(i,j) = colorTab[index -
1];
            }

            //混合灰度图和分水岭效果图并显示最终的窗口
            watershedImage = watershedImage*0.5 + grayImage*0.5;
            imshow( "watershed transform", watershedImage );
        }
    }
```

```
    return 0;
}


//-----------------------------[ onMouse()函数 ]-------------------------
//      描述：鼠标消息回调函数
//-----------------------------------------------------------------------
static void on_Mouse( int event, int x, int y, int flags, void* )
{
    //处理鼠标不在窗口中的情况
    if( x < 0 || x >= g_srcImage.cols || y < 0 || y >= g_srcImage.rows )
        return;

    //处理鼠标左键相关消息
    if( event == EVENT_LBUTTONUP || !(flags & EVENT_FLAG_LBUTTON) )
        prevPt = Point(-1,-1);
    else if( event == EVENT_LBUTTONDOWN )
        prevPt = Point(x,y);

    //鼠标左键按下并移动，绘制出白色线条
    else if( event == EVENT_MOUSEMOVE && (flags & EVENT_FLAG_LBUTTON) )
    {
        Point pt(x, y);
        if( prevPt.x < 0 )
         prevPt = pt;
        line( g_maskImage, prevPt, pt, Scalar::all(255), 5, 8, 0 );
        line( g_srcImage, prevPt, pt, Scalar::all(255), 5, 8, 0 );
        prevPt = pt;
        imshow(WINDOW_NAME, g_srcImage);
    }
}
```

程序运行截图如图 8.30～8.33 所示。

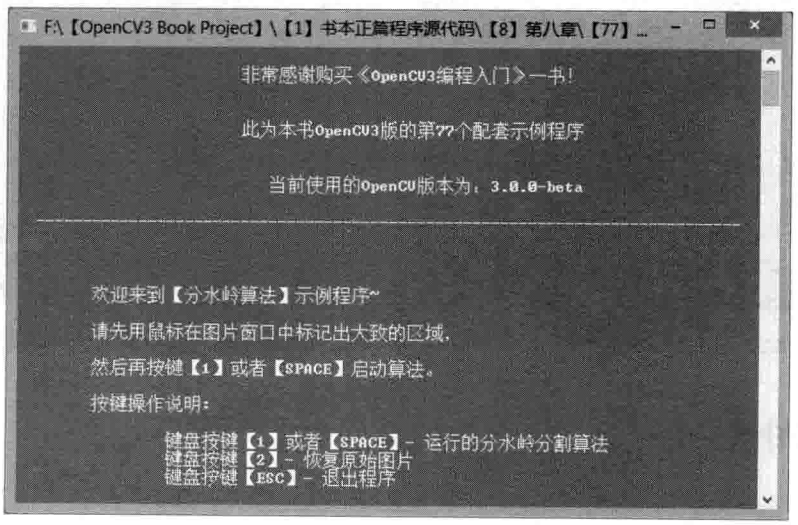


图 8.30 说明窗口

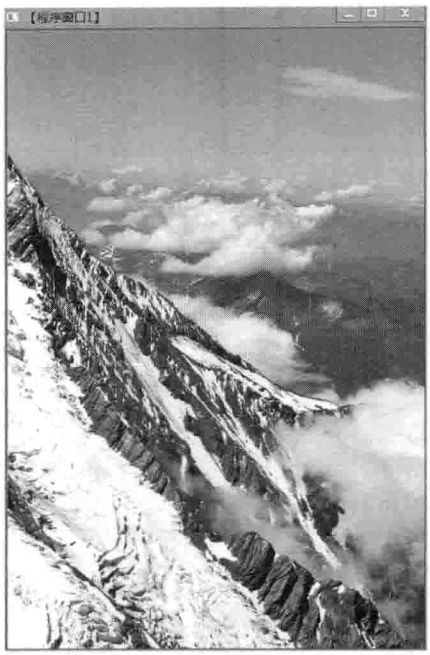


图 8.31 原始图窗口

图 8.32 用鼠标勾勒出区域后的图

图 8.33 分水岭算法效果图

8.6 图像修补

在实际应用中，我们的图像常常会被噪声腐蚀，这些噪声或者是镜头上的灰尘或水滴，或者是旧照片的划痕，或者由于图像的部分本身已经损坏。而“图像修复”（Inpainting），就是妙手回春，解决这些问题的良方。图像修复技术简单来

说，就是利用那些已经被破坏区域的边缘，即边缘的颜色和结构，繁殖和混合到损坏的图像中，以达到图像修补的目的。图 8.34～8.36 就是示例程序截图，演示将图像中的字迹移除的效果。

由于黑白印刷的缘故，本节书本上的图实际显示效果可能会受影响。请大家找到本节的示例程序（程序序号为 78）自己编译运行，进行图像修补技术的体验。

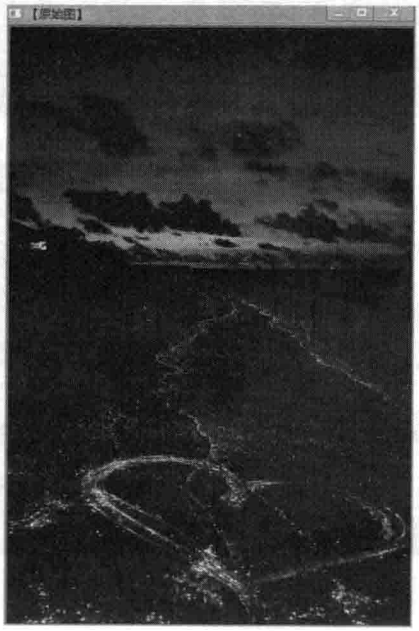


图 8.34 原始图

图 8.35 含字迹的图

图 8.36 经过修补后的图

如果被破坏的区域不是太大，并且在被破坏区域边缘包含足够多的纹理和颜色，那么图像修补技术可以很好地恢复图像。当然，当图像损坏区域过大时，我们“妙手回春”的能力也是有限的。如图 8.37、8.38 所示。

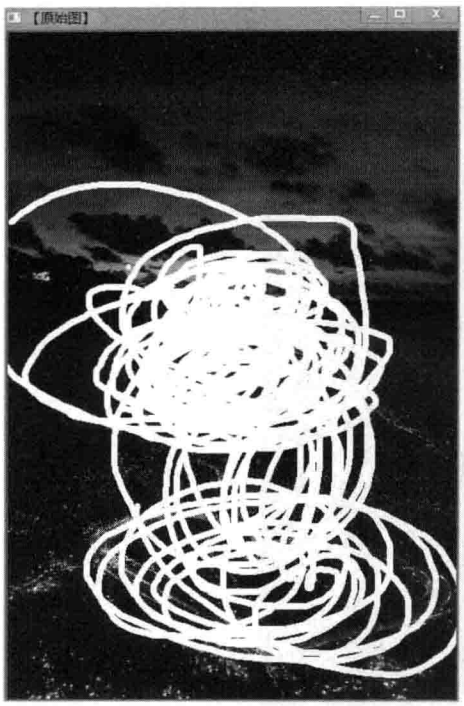


图 8.37 受过过大破坏后的图

图 8.38 经过修补后的图

8.6.1 实现图像修补：inpaint()函数

在新版 OpenCV 中，图像修补技术由 inpaint 函数实现，它可以用来从扫描的照片中清除灰尘和划痕，或者从静态图像或视频中去除不需要的物体。其原型声明如下。

```
C++: void inpaint(InputArray src, InputArray inpaintMask, OutputArray
dst, double inpaintRadius, int flags)
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，且需为 8 位单通道或者三通道图像。
- 第二个参数，InputArray 类型的 inpaintMask，修复掩膜，为 8 位的单通道图像。其中的非零像素表示需要修补的区域。
- 第三个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，和源图片有一样的尺寸和类型。
- 第四个参数，double 类型的 inpaintRadius，需要修补的每个点的圆形邻域，为修复算法的参考半径。
- 第五个参数，int 类型的 flags，修补方法的标识符，可以是表 8.4 所示两者之一。

表 8.4 图像修补方法的标识符

标识符	说明
INPAINT_NS	基于 Navier-Stokes 方程的方法
INPAINT_TELEA	Alexandru Telea 方法

OpenCV2 中 INPAINT_NS 和 INPAINT_TELEA 标识符可以分别写作 CV_INPAINT_NS 和 CV_ INPAINT_TELEA

8.6.2 综合示例程序：图像修补

函数和概念讲解完毕，下面我们依然是学习一个以本节所讲内容为核心的示例程序，将本节所学内容付诸实践，融会贯通。此示例程序会先让我们在图像中用鼠标绘制出白色的线条破坏图像，然后按下键盘按键【1】或【SPACE】进行图像修补操作。且如果对自己的绘制不够满意，可以按下键盘按键【2】恢复原始图像。操作说明如图 8.39 所示。

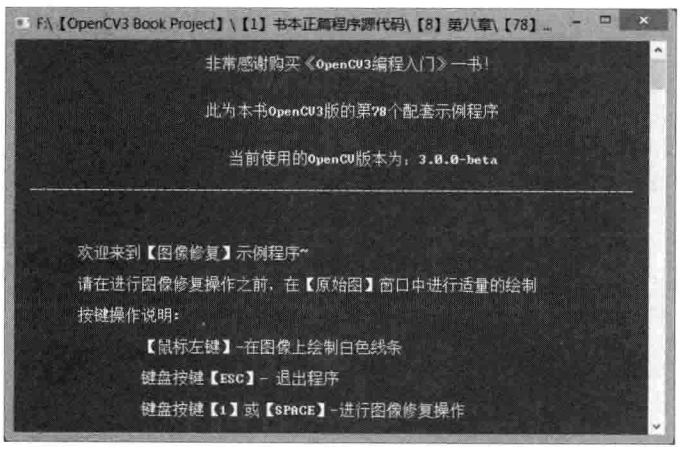


图 8.39 程序操作说明

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/photo/photo.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//----------------------------【宏定义部分】----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图】"        //为窗口标题定义的宏
#define WINDOW_NAME2 "【修补后的效果图】"        //为窗口标题定义的宏
```

```
//-----------------------------【全局变量声明部分】-----------------------------
//        描述：全局变量声明
//-------------------------------------------------------------------------
Mat srcImage1, inpaintMask;
Point previousPoint(-1,-1);//原来的点坐标

//-----------------------------【On_Mouse()函数】-----------------------------
//        描述：响应鼠标消息的回调函数
//-------------------------------------------------------------------------
static void On_Mouse( int event, int x, int y, int flags, void* )
{
    //鼠标左键弹起消息
    if( event == EVENT_LBUTTONUP || !(flags & EVENT_FLAG_LBUTTON) )
        previousPoint = Point(-1,-1);
    //鼠标左键按下消息
    else if( event == EVENT_LBUTTONDOWN )
        previousPoint = Point(x,y);
    //鼠标按下并移动，进行绘制
    else if( event == EVENT_MOUSEMOVE && (flags & EVENT_FLAG_LBUTTON) )
    {
        Point pt(x,y);
        if( previousPoint.x < 0 )
         previousPoint = pt;
        //绘制白色线条
        line( inpaintMask, previousPoint, pt, Scalar::all(255), 5, 8,
0 );
        line( srcImage1, previousPoint, pt, Scalar::all(255), 5, 8, 0 );
        previousPoint = pt;
        imshow(WINDOW_NAME1, srcImage1);
    }
}


//-----------------------------【main()函数】-----------------------------
//        描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main( int argc, char** argv )
{
    //载入原始图并进行掩膜的初始化
    Mat srcImage = imread("1.jpg", -1);
     if(!srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定图片存在~！ \n"); return false; }
    srcImage1 = srcImage.clone();
    inpaintMask = Mat::zeros(srcImage1.size(), CV_8U);

    //显示原始图
    imshow(WINDOW_NAME1, srcImage1);

    //设置鼠标回调消息
    setMouseCallback( WINDOW_NAME1, On_Mouse, 0 );
```

```
    //轮询按键，根据不同的按键进行处理
    while (1)
    {
        //获取按键键值
        char c = (char)waitKey();

        //键值为 ESC，程序退出
        if( c == 27 )
         break;

        //键值为 2，恢复成原始图像
        if( c == '2' )
        {
         inpaintMask = Scalar::all(0);
         srcImage.copyTo(srcImage1);
         imshow(WINDOW_NAME1, srcImage1);
        }

        //键值为 1 或者空格，进行图像修补操作
        if( c == '1' || c == ' ' )
        {
         Mat inpaintedImage;
         inpaint(srcImage1, inpaintMask, inpaintedImage, 3,
INPAINT_TELEA);
         imshow(WINDOW_NAME2, inpaintedImage);
        }
    }

    return 0;
}
```

程序的运行截图在概念讲解部分已经贴出过，在这里就不再贴出。请参考图8.34～图 8.39。

8.7 本章小结

本章中，我们先学习了查找轮并绘制轮廓，然后学习了如何寻找到物体的凸包，接着是使用多边形来包围轮廓，以及计算一个图像的矩。在本章后面几节，还学习了分水岭算法和图像修补操作的实现方法。

本章核心函数清单

函数名称	说明	对应讲解章节
findContours	在二值图像中寻找轮廓	8.1.1
drawContours	在图像中绘制外部或内部轮廓	8.1.2
convexHull	寻找图像点集中的凸包	8.2.2

续表

函数名称	说明	对应讲解章节
BoundingRect	计算并返回指定点集最外面（up-right）的矩形边界	8.3.1
minAreaRect	寻找可旋转的最小面积的包围矩形	8.3.2
minEnclosingCircle	利用一种迭代算法，对给定的 2D 点集，寻找面积最小的可包围他们的圆形	8.3.3
fitEllipse	用椭圆拟合二维点集	8.3.4
approxPolyDP	用指定精度逼近多边形曲线	8.3.5
moments	计算多边形和光栅形状的最高达三阶的所有矩	8.4.1
contourArea	计算整个轮廓或部分轮廓的面积	8.4.2
arcLength	计算封闭轮廓的周长或曲线的长度	8.4.3
watershed	实现分水岭算法	8.5.1
inpaint	进行图像修补，从扫描的照片中清除灰尘和划痕，或者从静态图像或视频中去除不需要的物体	8.6.1

本章示例程序清单

示例程序序号	程序说明	对应章节
69	轮廓查找	8.1.3
70	查找并绘制轮廓	8.1.4
71	凸包检测基础	8.2.3
72	寻找和绘制物体的凸包	8.2.4
73	创建包围轮廓的矩形边界	8.3.6
74	创建包围轮廓的圆形边界	8.3.7
75	使用多边形包围轮廓	8.3.8
76	图像轮廓矩	8.4.4
77	分水岭算法的使用	8.5.2
78	实现图像修补	8.6.2

第 9 章
直方图与匹配

导读

在进行物体图像和视频信息分析的过程中，我们常常会习惯于将眼中看到的物体用直方图（histogram）表示出来，得到比较直观的数据官感展示。直方图可以用来描述各种不同的参数和事物，如物体的色彩分布、物体边缘梯度模板，以及表示目标位置的当前假设的概率分布。

本章你将学到：

- 什么是直方图
- 直方图的计算与绘制
- 如何进行直方图的对比
- 反向投影技术
- 模板匹配技术

9.1 图像直方图概述

直方图广泛运用于很多计算机视觉运用当中，通过标记帧与帧之间显著的边缘和颜色的统计变化，来检测视频中场景的变化。在每个兴趣点设置一个有相近特征的直方图所构成“标签”，用以确定图像中的兴趣点。边缘、色彩、角度等直方图构成了可以被传递给目标识别分类器的一个通用特征类型。色彩和边缘的直方图序列还可以用来识别网络视频是否被复制。如图 9.1 所示。

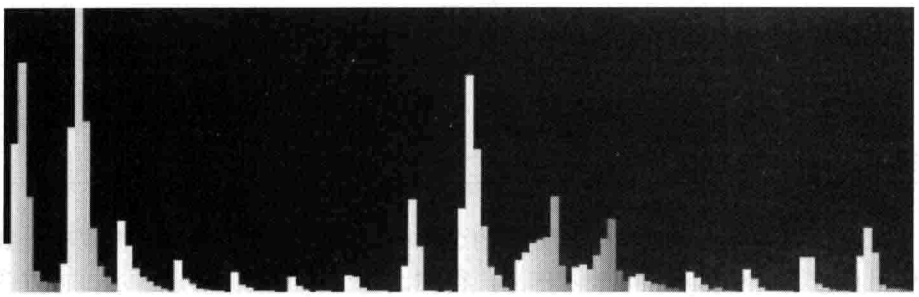


图 9.1 某图像的 H-S 颜色直方图

其实，简单点说，直方图就是对数据进行统计的一种方法，并且将统计值组织到一系列事先定义好的 bin 当中。其中，bin 为直方图中经常用到的一个概念，可翻译为“直条”或“组距”，其数值是从数据中计算出的特征统计量，这些数据可以是诸如梯度、方向、色彩或任何其他特征。且无论如何，直方图获得的是数据分布的统计图。通常直方图的维数要低于原始数据。总而言之，直方图是计算机视觉中最经典的工具之一。

在统计学中，直方图（Histogram）是一种对数据分布情况的图形表示，是一种二维统计图表，它的两个坐标分别是统计样本和该样本对应的某个属性的度量。

我们在图像变换的那一章中讲过直方图的均衡化，它是通过拉伸像素强度分布范围来增强图像对比度的一种方法。大家在自己的心目中应该已经对直方图有一定的理解和认知。下面就来看一看对图像直方图比较书面化的解释。

图像直方图（Image Histogram）是用以表示数字图像中亮度分布的直方图，标绘了图像中每个亮度值的像素数。可以借助观察该直方图了解需要如何调整亮度分布。这种直方图中，横坐标的左侧为纯黑、较暗的区域，而右侧为较亮、纯白的区域。因此，一张较暗图片的图像直方图中的数据多集中于左侧和中间部分，而整体明亮、只有少量阴影的图像则相反。计算机视觉领域常借助图像直方图来实现图像的二值化。

直方图的意义如下。

- 直方图是图像中像素强度分布的图形表达方式。
- 它统计了每一个强度值所具有的像素个数。

上面已经讲到，直方图是对数据的统计集合，并将统计结果分布于一系列预

定义的 bins 中。这里的数据不仅仅指的是灰度值，且统计数据可能是任何能有效描述图像的特征。下面看一个例子，假设有一个矩阵包含一张图像的信息（灰度值 0-255），让我们按照某种方式来统计这些数字。既然已知数字的范围包含 256 个值，于是可以将这个范围分割成子区域（也就是上面讲到 bins），如：

$$[0,255]=[0,15]\cup[16,31]\cup\cdots\cup[240,255]$$
$$range=bin_1\cup bin_2\cup\cdots\cup bin_{n=15}$$

然后再统计每一个 bin_i 的像素数目。采用这一方法来统计上面的数字矩阵，可以得到图 9.2（其中 x 轴表示 bin，y 轴表示各个 bin 中的像素个数）。

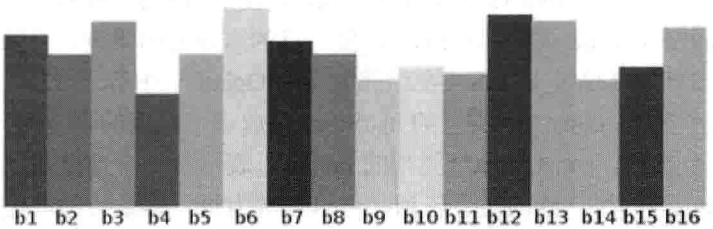


图 9.2 图像数字矩阵的直方图

以上就是一个说明直方图的用途的简单示例。其实，直方图并不局限于统计颜色灰度， 而是可以统计任何图像特征，如梯度、方向等。

让我们具体讲讲直方图的一些术语和细节。

- dims：需要统计的特征的数目。在上例中，dims = 1 因为我们仅仅统计了灰度值(灰度图像)。
- bins：每个特征空间子区段的数目，可翻译为“直条”或“组距”。在上例中，bins = 16。
- range：每个特征空间的取值范围。在上例中，range = [0,255]。

9.2 直方图的计算与绘制

直方图的计算在 OpenCV 中可以使用 calcHist()函数，而计算完成之后，可以采用 OpenCV 中的绘图函数，如绘制矩形的 rectangle()函数，绘制线段的 line()来完成。

9.2.1 计算直方图：calcHist()函数

在 OpenCV 中，calcHist()函数用于计算一个或者多个阵列的直方图。原型如下。

```
C++: 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 )
```

- 第一个参数，const Mat*类型的 images，输入的数组(或数组集)，它们需为

相同的深度（CV_8U 或 CV_32F）和相同的尺寸。

- 第二个参数，int 类型的 nimages，输入数组的个数，也就是第一个参数中存放了多少张“图像”，有几个原数组。
- 第三个参数，const int*类型的 channels，需要统计的通道（dim）索引。第一个数组通道从 0 到 images[0].channels()-1，而第二个数组通道从 images[0].channels()计算到 images[0].channels()+images[1].channels()-1。
- 第四个参数，InputArray 类型的 mask，可选的操作掩码。如果此掩码不为空，那么它必须为 8 位，并且与 images[i]有同样大小的尺寸。这里的非零掩码元素用于标记出统计直方图的数组元素数据。
- 第五个参数，OutputArray 类型的 hist，输出的目标直方图，一个二维数组。
- 第六个参数，int 类型的 dims，需要计算的直方图的维度，必须是正数，且不大于 CV_MAX_DIMS（在当前版本的 OpenCV 中等于 32）。
- 第七个参数，const int*类型的 histSize，存放每个维度的直方图尺寸的数组。
- 第八个参数，const float**类型的 ranges，表示每一个维度数组（第六个参数 dims）的每一维的边界阵列，可以理解为每一维数值的取值范围。
- 第九个参数，bool 类型的 uniform，指示直方图是否均匀的标识符，有默认值 true。
- 第十个参数，bool 类型的 accumulate，累计标识符，有默认值 false。若其为 true，直方图在配置阶段不会被清零。此功能主要是允许从多个阵列中计算单个直方图，或者用于在特定的时间更新直方图。

9.2.2 找寻最值：minMaxLoc()函数

minMaxLoc()函数的作用是在数组中找到全局最小值和最大值。它有两个版本的原型，在此介绍常用的那一个版本。

```
C++: void minMaxLoc(InputArray src, double* minVal, double* maxVal=0,
Point* minLoc=0, Point* maxLoc=0, InputArray mask=noArray())
```

- 第一个参数，InputArray 类型的 src，输入的单通道阵列。
- 第二个参数，double*类型的 minVal，返回最小值的指针。若无须返回，此值置为 NULL。
- 第三个参数，double*类型的 maxVal，返回的最大值的指针。若无须返回，此值置为 NULL。
- 第四个参数，Point*类型的 minLoc，返回最小位置的指针（二维情况下）。若无须返回，此值置为 NULL。
- 第五个参数，Point*类型的 maxLoc，返回最大位置的指针（二维情况下）。若无须返回，此值置为 NULL。
- 第六个参数，InputArray 类型的 mask，用于选择子阵列的可选掩膜。

9.2.3 示例程序：绘制 H—S 直方图

下面的示例说明如何计算彩色图像的色调，饱和度二维直方图。

注意：色调（Hue），饱和度（Saturation）。所以“H-S 直方图”就是 “色调—饱和度直方图”。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;


//----------------------------【main()函数】----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】载入源图，转化为 HSV 颜色模型
    Mat srcImage, hsvImage;
    srcImage=imread("1.jpg");
    cvtColor(srcImage,hsvImage, COLOR_BGR2HSV);

    //【2】参数准备
    //将色调量化为 30 个等级，将饱和度量化为 32 个等级
    int hueBinNum = 30;//色调的直方图直条数量
     int saturationBinNum = 32;//饱和度的直方图直条数量
    int histSize[ ] = {hueBinNum, saturationBinNum};
    // 定义色调的变化范围为 0 到 179
    float hueRanges[] = { 0, 180 };
    //定义饱和度的变化范围为 0（黑、白、灰）到 255（纯光谱颜色）
    float saturationRanges[] = { 0, 256 };
    const float* ranges[] = { hueRanges, saturationRanges };
    MatND dstHist;
    //参数准备，calcHist 函数中将计算第 0 通道和第 1 通道的直方图
    int channels[] = {0, 1};

    //【3】正式调用 calcHist，进行直方图计算
    calcHist( &hsvImage,//输入的数组
                1, //数组个数为 1
                channels,//通道索引
                Mat(), //不使用掩膜
                dstHist, //输出的目标直方图
                2, //需要计算的直方图的维度为 2
                histSize, //存放每个维度的直方图尺寸的数组
                ranges,//每一维数值的取值范围数组
                true, // 指示直方图是否均匀的标识符，true 表示均匀的直方图
                false );//累计标识符，false 表示直方图在配置阶段会被清零

    //【4】为绘制直方图准备参数
    double maxValue=0;//最大值
    minMaxLoc(dstHist, 0, &maxValue, 0, 0);//查找数组和子数组的全局最小值和最大值存入 maxValue 中
    int scale = 10;
```

```
    Mat histImg = Mat::zeros(saturationBinNum*scale, hueBinNum*10,
CV_8UC3);

    //【5】双层循环，进行直方图绘制
    for( int hue = 0; hue < hueBinNum; hue++ )
        for( int saturation = 0; saturation < saturationBinNum;
saturation++ )
        {
         float binValue = dstHist.at<float>(hue, saturation);//直方图直
条的值
         int intensity = cvRound(binValue*255/maxValue);//强度

         //正式进行绘制
         rectangle( histImg, Point(hue*scale, saturation*scale),
             Point( (hue+1)*scale - 1, (saturation+1)*scale - 1),
             Scalar::all(intensity), FILLED );
        }

        //【6】显示效果图
        imshow( "素材图", srcImage );
        imshow( "H-S 直方图", histImg );

        waitKey();
}
```

程序中新出现的MatND类是用于存储直方图的一种数据结构，其用法简单，常常在直方图相关OpenCV程序中出现。而+-**程序运行截图如图9.4、9.5所示。

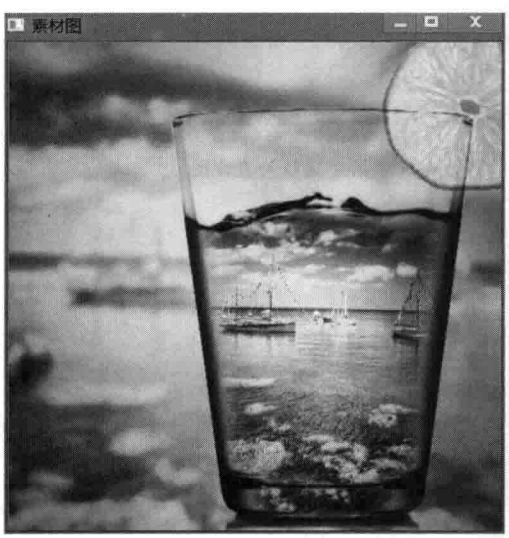


图9.3 素材图

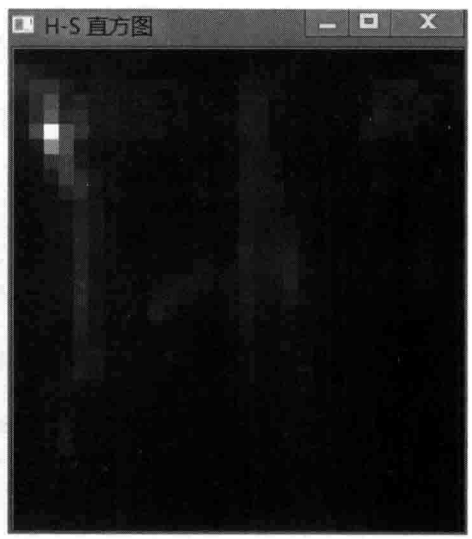


图9.4 图像的H-S二维直方图

9.2.4 示例程序：计算并绘制图像一维直方图

上文中已经讲解了calcHist()函数的用法，并绘制出了图像的H-S二维直方图。而本节我们会通过一个示例，来学习图像一维直方图的计算和绘制过程。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//        描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//-----------------------------【main()函数】--------------------------
//        描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】载入原图并显示
    Mat srcImage = imread("1.jpg", 0);
    imshow("原图",srcImage);
    if(!srcImage.data) {cout << "fail to load image" << endl; return
0;}

    //【2】定义变量
    MatND dstHist;       // 在cv中用CvHistogram *hist = cvCreateHist
    int dims = 1;
    float hranges[] = {0, 255};
    const float *ranges[] = {hranges};   // 这里需要为const类型
    int size = 256;
    int channels = 0;

    //【3】计算图像的直方图
    calcHist(&srcImage, 1, &channels, Mat(), dstHist, dims, &size,
ranges);    // cv中是cvCalcHist
    int scale = 1;

    Mat dstImage(size * scale, size, CV_8U, Scalar(0));
    //【4】获取最大值和最小值
    double minValue = 0;
    double maxValue = 0;
    minMaxLoc(dstHist,&minValue, &maxValue, 0, 0);  //  在cv中用的是
cvGetMinMaxHistValue

    //【5】绘制出直方图
    int hpt = saturate_cast<int>(0.9 * size);
    for(int i = 0; i < 256; i++)
    {
        float binValue = dstHist.at<float>(i);          //   注意hist中
是float类型    而在OpenCV1.0版中用cvQueryHistValue_1D
        int realValue = saturate_cast<int>(binValue * hpt/maxValue);
        rectangle(dstImage,Point(i*scale, size - 1), Point((i+1)*scale
- 1, size - realValue), Scalar(255));
    }
    imshow("一维直方图", dstImage);
    waitKey(0);
```

```
    return 0;
}
```

运行此详细注释的代码，可以得到如图 9.5、9.6 所示的运行结果。

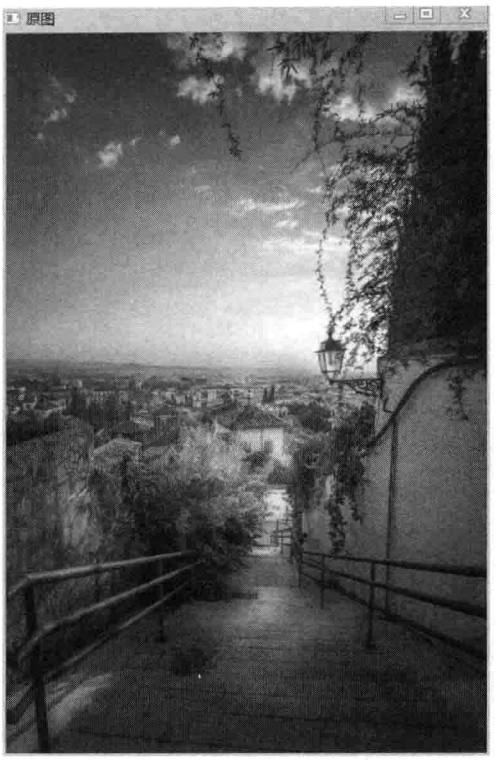


图 9.5 素材图

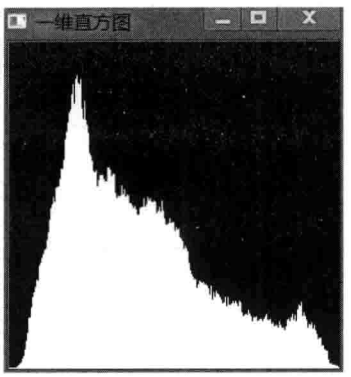


图 9.6 素材图的一维直方图

9.2.5 示例程序：绘制 RGB 三色直方图

上文我们讲解的是单个分量的一维直方图的绘制，接下来看看如何分别绘制图像的 RGB 三色直方图。详细注释的代码如下。

```
//-----------------------【头文件、命名空间包含部分】-----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;


//-----------------------------【main()函数】-----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】载入素材图并显示
    Mat srcImage;
    srcImage=imread("1.jpg");
```

```
    imshow( "素材图", srcImage );

    //【2】参数准备
    int bins = 256;
    int hist_size[] = {bins};
    float range[] = { 0, 256 };
    const float* ranges[] = { range};
    MatND redHist,grayHist,blueHist;
    int channels_r[] = {0};

    //【3】进行直方图的计算(红色分量部分)
    calcHist( &srcImage, 1, channels_r, Mat(), //不使用掩膜
        redHist, 1, hist_size, ranges,
        true, false );

    //【4】进行直方图的计算(绿色分量部分)
    int channels_g[] = {1};
    calcHist( &srcImage, 1, channels_g, Mat(), // do not use mask
        grayHist, 1, hist_size, ranges,
        true, // the histogram is uniform
        false );

    //【5】进行直方图的计算(蓝色分量部分)
    int channels_b[] = {2};
    calcHist( &srcImage, 1, channels_b, Mat(), // do not use mask
        blueHist, 1, hist_size, ranges,
        true, // the histogram is uniform
        false );

    //-----------------------绘制出三色直方图-------------------------
    //参数准备
    double maxValue_red,maxValue_green,maxValue_blue;
    minMaxLoc(redHist, 0, &maxValue_red, 0, 0);
    minMaxLoc(grayHist, 0, &maxValue_green, 0, 0);
    minMaxLoc(blueHist, 0, &maxValue_blue, 0, 0);
    int scale = 1;
    int histHeight=256;
    Mat histImage = Mat::zeros(histHeight,bins*3, CV_8UC3);

    //正式开始绘制
    for(int i=0;i<bins;i++)
    {
        //参数准备
        float binValue_red = redHist.at<float>(i);
        float binValue_green = grayHist.at<float>(i);
        float binValue_blue = blueHist.at<float>(i);
        int intensity_red =
cvRound(binValue_red*histHeight/maxValue_red);  //要绘制的高度
        int intensity_green =
cvRound(binValue_green*histHeight/maxValue_green);  //要绘制的高度
        int intensity_blue =
cvRound(binValue_blue*histHeight/maxValue_blue);  //要绘制的高度
```

```
        //绘制红色分量的直方图
        rectangle(histImage,Point(i*scale,histHeight-1),
         Point((i+1)*scale - 1, histHeight - intensity_red),
         Scalar (255,0,0));

        //绘制绿色分量的直方图
        rectangle(histImage,Point((i+bins)*scale,histHeight-1),
         Point((i+bins+1)*scale - 1, histHeight - intensity_green),
         Scalar (0,255,0));

        //绘制蓝色分量的直方图
        rectangle(histImage,Point((i+bins*2)*scale,histHeight-1),
         Point((i+bins*2+1)*scale - 1, histHeight - intensity_blue),
         Scalar (0,0,255));

    }

    //在窗口中显示出绘制好的直方图
    imshow( "图像的RGB直方图", histImage );
    waitKey(0);
    return 0;
}
```

运行此程序，可以得到如图 9.7、9.8 所示的结果。

图 9.7 素材图

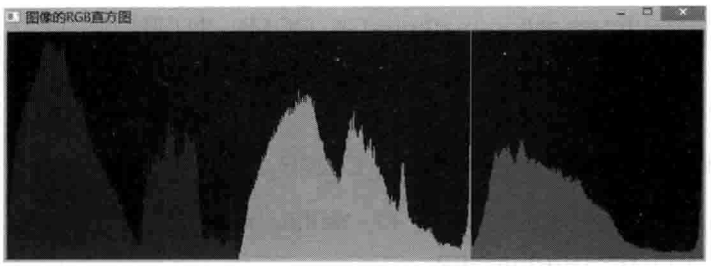


图 9.8 素材图的 RGB 直方图

至此，图像的一维和二维直方图，以及图像的三色直方图我们都通过实例进行了演示和讲解。

9.3 直方图对比

对于直方图来说，一个不可或缺的工具便是用某些具体的标准来比较两个直方图的相似度。要对两个直方图（比如说 H_1 和 H_2）进行比较，首先必须选择一个衡量直方图相似度的对比标准（$d(H_1,H_2)$）。在 OpenCV 2.X 中，我们用 compareHist() 函数来对比两个直方图的相似度，而此函数的返回值就是 $d(H_1,H_2)$。

9.3.1 对比直方图：compareHist()函数

compareHist()函数用于对两幅直方图进行比较。有两个版本的 C++原型，如下。

```
C++: double compareHist(InputArray H1, InputArray H2, int method)
C++: double compareHist(const SparseMat& H1, const SparseMat& H2,
int method)
```

它们的前两个参数是要比较的大小相同的直方图，第三个变量是所选择的距离标准。可采用如下 4 种方法，比较两个直方图（H1 表示第一个，H2 表示第二个）：

1. 相关，Correlation（method=CV_COMP_CORREL）

$$d(H_1,H_2)=\frac{\sum_I (H_1(I)-\bar{H}_1)(H_2(I)-\bar{H}_2)}{\sqrt{\sum_I (H_1(I)-\bar{H}_1)^2 \sum_I (H_2(I)-\bar{H}_2)^2}}$$

其中：

$$\bar{H}_k=\frac{1}{N}\sum_J H_k(J)$$

且 N 等于直方图中 bin（译为“直条”或“组距”）的个数。

2. 卡方，Chi-Square (method=CV_COMP_CHISQR)

$$d(H_1,H_2)=\sum_I \frac{(H_1(I)-H_2(I))^2}{H_1(I)}$$

3．直方图相交，Intersection(method=CV_COMP_INTERSECT)

$$d(H_1,H_2)=\sum_I \min\left(H_1(I),H_2(I)\right)$$

4．Bhattacharyya 距离 (method=CV_COMP_BHATTACHARYYA)

这里的 Bhattacharyya 距离和 Hellinger 距离相关，也可以写作 method=CV_COMP_ HELLINGER

$$d(H_1,H_2)=\sqrt{1-\frac{1}{\sqrt{\bar{H}_1\bar{H}_2N^2}}\sum_I\sqrt{H_1(I)\bullet H_2(I)}}$$

此处的宏定义在当前版本的 OpenCV3 中依然沿用 “CV_” 前缀，在未来版本中应该会有更改。如需使用，可以分别用 int 类型的 1、2、3、4 替代 CV_COMP_CORREL、CV_COMP_CHISQR、CV_COMP_INTERSECT、CV_COMP_BHATTACHARYYA 这四个宏。或者 “#include<cv.h>” 加入 cv.h 头文件。

9.3.2 示例程序：直方图对比

此次的示例程序为大家演示了如何用 compareHist()函数进行直方图对比。代码中的 MatND 类是用于存储直方图的一种数据结构，用法简单，在这里就不多做讲解，大家看到详细注释的示例程序就会明白。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;


//-----------------------------【main()函数】--------------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】声明储存基准图像和另外两张对比图像的矩阵( RGB 和 HSV )
    Mat srcImage_base, hsvImage_base;
    Mat srcImage_test1, hsvImage_test1;
    Mat srcImage_test2, hsvImage_test2;
    Mat hsvImage_halfDown;

    //【2】载入基准图像(srcImage_base) 和两张测试图像 srcImage_test1、
srcImage_test2，并显示
    srcImage_base = imread( "1.jpg",1 );
    srcImage_test1 = imread( "2.jpg", 1 );
    srcImage_test2 = imread( "3.jpg", 1 );
    //显示载入的 3 张图像
```

```
    imshow("基准图像",srcImage_base);
    imshow("测试图像 1",srcImage_test1);
    imshow("测试图像 2",srcImage_test2);

    // 【3】将图像由 BGR 色彩空间转换到 HSV 色彩空间
    cvtColor( srcImage_base, hsvImage_base, COLOR_BGR2HSV );
    cvtColor( srcImage_test1, hsvImage_test1, COLOR_BGR2HSV );
    cvtColor( srcImage_test2, hsvImage_test2, COLOR_BGR2HSV );

    //【4】创建包含基准图像下半部的半身图像(HSV 格式)
    hsvImage_halfDown = hsvImage_base( Range( hsvImage_base.rows/2,
hsvImage_base.rows - 1 ), Range( 0, hsvImage_base.cols - 1 ) );

    //【5】初始化计算直方图需要的实参
    // 对 hue 通道使用 30 个 bin,对 saturatoin 通道使用 32 个 bin
    int h_bins = 50; int s_bins = 60;
    int histSize[] = { h_bins, s_bins };
    // hue 的取值范围从 0 到 256, saturation 取值范围从 0 到 180
    float h_ranges[] = { 0, 256 };
    float s_ranges[] = { 0, 180 };
    const float* ranges[] = { h_ranges, s_ranges };
    // 使用第 0 和第 1 通道
    int channels[] = { 0, 1 };

    // 【6】创建储存直方图的 MatND 类的实例
    MatND baseHist;
    MatND halfDownHist;
    MatND testHist1;
    MatND testHist2;

    // 【7】计算基准图像,两张测试图像,半身基准图像的 HSV 直方图
    calcHist( &hsvImage_base, 1, channels, Mat(), baseHist, 2, histSize,
ranges, true, false );
    normalize( baseHist, baseHist, 0, 1, NORM_MINMAX, -1, Mat() );

    calcHist( &hsvImage_halfDown, 1, channels, Mat(), halfDownHist, 2,
histSize, ranges, true, false );
    normalize( halfDownHist, halfDownHist, 0, 1, NORM_MINMAX, -1,
Mat() );

    calcHist( &hsvImage_test1, 1, channels, Mat(), testHist1, 2,
histSize, ranges, true, false );
    normalize( testHist1, testHist1, 0, 1, NORM_MINMAX, -1, Mat() );

    calcHist( &hsvImage_test2, 1, channels, Mat(), testHist2, 2,
histSize, ranges, true, false );
    normalize( testHist2, testHist2, 0, 1, NORM_MINMAX, -1, Mat() );


    //【8】按顺序使用 4 种对比标准将基准图像的直方图与其余各直方图进行对比
    for( int i = 0; i < 4; i++ )
    {
```

```
        //进行图像直方图的对比
        int compare_method = i;
        double base_base = compareHist( baseHist, baseHist,
compare_method );
        double base_half = compareHist( baseHist, halfDownHist,
compare_method );
        double base_test1 = compareHist( baseHist, testHist1,
compare_method );
        double base_test2 = compareHist( baseHist, testHist2,
compare_method );

        //输出结果
        printf( " 方法 [%d] 的匹配结果如下：\n\n 【基准图 - 基准图】: %f, 【基
准图 - 半身图】: %f,【基准图 - 测试图 1】: %f, 【基准图 - 测试图 2】: %f
\n-----------------------------------------------------------------\
n", i, base_base, base_half , base_test1, base_test2 );
    }

    printf( "检测结束。" );
    waitKey(0);
    return 0;
}
```

学习完详细注释的代码，让我们一起看看程序的运行截图。首先是三张素材图，如图 9.9～9.11 所示。

图 9.9 基准图像

图 9.10 测试图像（1）

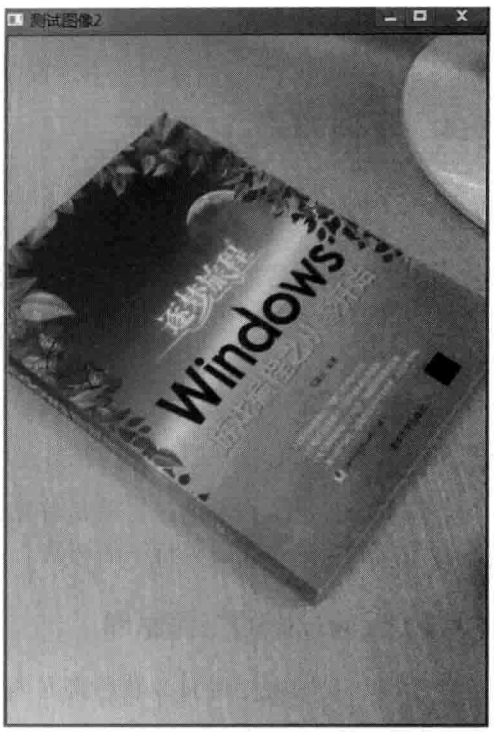


图 9.11 测试图像（2）

需要注意的是，在上述代码中还会将基准图像与它自身及其半身图像进行对比。而我们知道，当将基准图像直方图及其自身进行对比时，会产生完美的匹配；当与来源于同一样的背景环境的半身图对比时，应该会有比较高的相似度；当与来自不同亮度光照条件的其余两张测试图像对比时，匹配度应该不是很好。输出的匹配结果如图 9.12 所示。

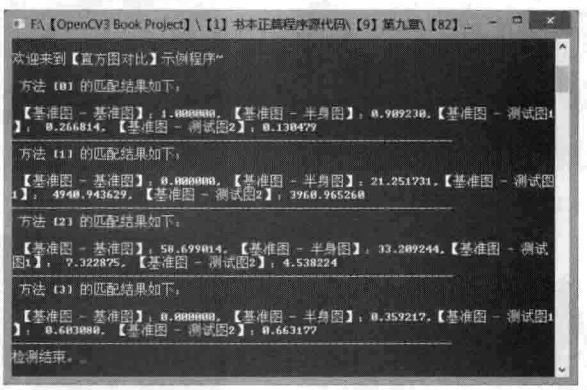


图 9.12 直方图对比结果截图

其中的方法 0 至 3，分别表示之前讲过的 Correlation、Chi-square、Intersection、Bhattacharyya 对比标准。其中，对于 Correlation（方法 0）和 Intersection（方法 2）标准，值越大表示相似度越高。可以发现，【基准图—基准图】的匹配数值结果相对于其他几种匹配方式是最大的，符合实际情况。【基准图—半身图】的匹配结果次大，正如我们预料。而【基准—测试图 1】和【基准图—测试图 2】的匹配结果

却不尽人意，同样和之前的预料吻合。

9.4 反向投影

9.4.1 引言

如果一幅图像的区域中显示的是一种结构纹理或者一个独特的物体，那么这个区域的直方图可以看作一个概率函数，其表现形式是某个像素属于该纹理或物体的概率。

而反向投影（back projection）就是一种记录给定图像中的像素点如何适应直方图模型像素分布方式的一种方法。

简单的讲，所谓反向投影就是首先计算某一特征的直方图模型，然后使用模型去寻找图像中存在的该特征的方法。

9.4.2 反向投影的工作原理

下面，我们将使用 H-S 肤色直方图为例来解释反向投影的工作原理。

首先通过之前讲过的求 H-S 直方图的示例程序，得到如图 9.13～9.16 所示的 H-S 肤色直方图。

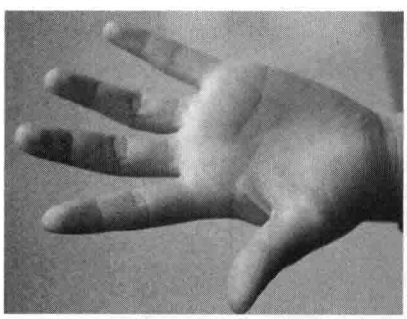


图 9.13 第一张素材图

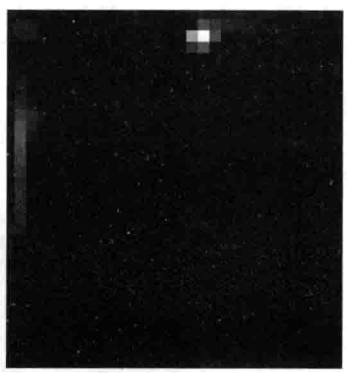


图 9.14 第一张素材图的 H-S 直方图

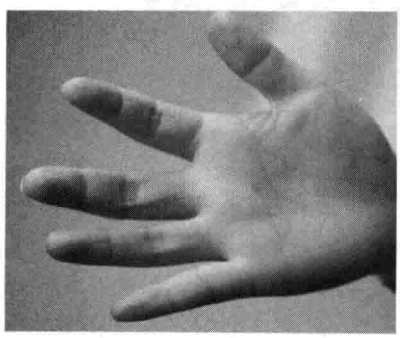


图 9.15 第二张素材图

图 9.16 第二张素材图的 H-S 直方图

而我们要做的，就是使用模型直方图（代表手掌的皮肤色调）来检测测试图像中的皮肤区域。以下是检测步骤。

（1）对测试图像中的每个像素（p(i,j)），获取色调数据并找到该色调（$h_{i,j}$,$s_{i,j}$）在直方图中的 bin 的位置。

（2）查询模型直方图中对应 bin 的数值。

（3）将此数值储存在新的反射投影图像中。也可以先归一化直方图数值到 0-255 范围，这样可以直接显示反射投影图像（单通道图像）。

（4）通过对测试图像中的每个像素采用以上步骤，可以得到最终的反射投影图像。如图 9.17 所示。

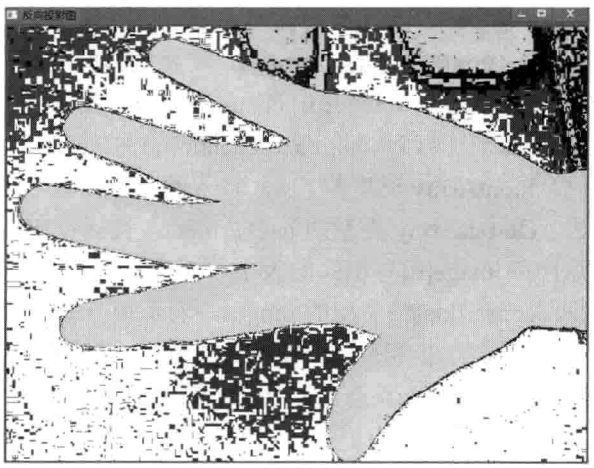


图 9.17 最终的反射投影图像

（5）使用统计学的语言进行分析。反向投影中储存的数值代表了测试图像中该像素属于皮肤区域的概率 。比如以图 9.17 为例，亮起的区域是皮肤区域的概率更大，而更暗的区域则表示是皮肤的概率更低。另外，可以注意到，手掌内部和边缘的阴影影响了检测的精度。

9.4.3 反向投影的作用

反向投影用于在输入图像（通常较大）中查找与特定图像（通常较小或者仅 1 个像素，以下将其称为模板图像）最匹配的点或者区域，也就是定位模板图像出现在输入图像的位置。

9.4.4 反向投影的结果

反向投影的结果包含了以每个输入图像像素点为起点的直方图对比结果。可以把它看成是一个二维的浮点型数组、二维矩阵，或者单通道的浮点型图像。

9.4.5 计算反向投影：calcBackProject()函数

calcBackProject()函数用于计算直方图的反向投影。

```
C++: void calcBackProject(
const Mat* images,
 int nimages,
 const int* channels,
 InputArray hist,
 OutputArray backProject,
 const float** ranges,
 double scale=1,
 bool uniform=true )
```

- 第一个参数，const Mat*类型的 images，输入的数组(或数组集)，它们须为相同的深度（CV_8U 或 CV_32F）和相同的尺寸，而通道数则可以任意。
- 第二个参数，int 类型的 nimages，输入数组的个数，也就是第一个参数中存放了多少张“图像”，有几个原数组。
- 第三个参数，const int*类型的 channels，需要统计的通道 (dim)索引。第一个数组通道从 0 到 images[0].channels()-1，而第二个数组通道从 images[0].channels()计算到 images[0].channels() + images[1].channels()-1。
- 第四个参数，InputArray 类型的 hist，输入的直方图。
- 第五个参数，OutputArray 类型的 backProject，目标反向投影阵列，其须为单通道，并且和 image[0]有相同的大小和深度。
- 第六个参数，const float**类型的 ranges，表示每一个维度数组（第六个参数 dims）的每一维的边界阵列，可以理解为每一维数值的取值范围。
- 第七个参数，double scale，有默认值 1，输出的方向投影可选的缩放因子，默认值为 1。
- 第八个参数，bool 类型的 uniform，指示直方图是否均匀的标识符，有默认值 true。

9.4.6 通道复制：mixChannels()函数

此函数由输入参数复制某通道到输出参数特定的通道中。有两个版本的 C++ 原型，采用函数注释方式分别介绍如下。

```
C++: void mixChannels(
const Mat* src, //输入的数组，所有的矩阵必须有相同的尺寸和深度
size_t nsrcs,//第一个参数 src 输入的矩阵数
Mat* dst,//输出的数组，所有矩阵必须被初始化，且大小和深度必须与 src[0]相同
size_t ndsts, //第三个参数 dst 输入的矩阵数
const int* fromTo,//对指定的通道进行复制的数组索引
 size_t npairs)//第五个参数 fromTo 的索引数


C++: void mixChannels(
const vector<Mat>& src,//输入的矩阵向量，所有的矩阵必须有相同的尺寸和深度
 vector<Mat>& dst, //输出的矩阵向量，所有矩阵须被初始化，且大小和深度须与 src[0]
相同
 const int* fromTo, //对指定的通道进行复制的数组索引
 size_t npairs)//第三个参数 fromTo 的索引数
```

此函数为重排图像通道提供了比较先进的机制。其实，之前我们接触到的split()和 merge()，以及 cvtColor()的某些形式，都只是 mixChannels()的一部分。

下面给出一个示例，将一个 4 通道的 RGBA 图像转化为 3 通道 BGR（R 通道和 B 通道交换）和一个单独的 Alpha 通道的图像。

```
Mat rgba( 100, 100, CV_8UC4, Scalar(1,2,3,4) );
Mat bgr( rgba.rows, rgba.cols, CV_8UC3 );
Mat alpha( rgba.rows, rgba.cols, CV_8UC1 );

//组成矩阵数组来进行操作
Mat out[] = { bgr, alpha };
// 说明：将 rgba[0] -> bgr[2], rgba[1] -> bgr[1],
// 说明：将 rgba[2] -> bgr[0], rgba[3] -> alpha[0]
int from_to[] = { 0,2, 1,1, 2,0, 3,3 };
mixChannels( &rgba, 1, out, 2, from_to, 4 );
```

9.4.7 综合程序：反向投影

下面将给大家展示一个浓缩了本节内容的讲解内容经过详细注释的反向投影示例程序。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
using namespace cv;

//-----------------------------【宏定义部分】-----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图】"        //为窗口标题定义的宏


//-------------------------【全局变量声明部分】-------------------------
//          描述：全局变量声明
//-------------------------------------------------------------------------
Mat g_srcImage; Mat g_hsvImage; Mat g_hueImage;
int g_bins = 30;//直方图组距

//-------------------------【全局函数声明部分】-------------------------
//          描述：全局函数声明
//-------------------------------------------------------------------------
void on_BinChange(int, void* );

//-----------------------------【main()函数】-----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
```

```
    //【1】读取源图像，并转换到 HSV 空间
    g_srcImage = imread( "1.jpg", 1 );
        if(!g_srcImage.data ) { printf("读取图片错误，请确定目录下是否有
imread函数指定图片存在~！ \n"); return false; }
    cvtColor( g_srcImage, g_hsvImage, COLOR_BGR2HSV );

    //【2】分离 Hue 色调通道
    g_hueImage.create( g_hsvImage.size(), g_hsvImage.depth() );
    int ch[ ] = { 0, 0 };
    mixChannels( &g_hsvImage, 1, &g_hueImage, 1, ch, 1 );

    //【3】创建 Trackbar 来输入 bin 的数目
    namedWindow( WINDOW_NAME1 , WINDOW_AUTOSIZE );
    createTrackbar("色调组距 ", WINDOW_NAME1 , &g_bins, 180,
on_BinChange );
    on_BinChange(0, 0);//进行一次初始化

    //【4】显示效果图
    imshow( WINDOW_NAME1 , g_srcImage );

    // 等待用户按键
    waitKey(0);
    return 0;
}


//-----------------------【on_HoughLines()函数】-----------------------
//          描述：响应滑动条移动消息的回调函数
//-----------------------------------------------------------------------
void on_BinChange(int, void* )
{
    //【1】参数准备
    MatND hist;
    int histSize = MAX( g_bins, 2 );
    float hue_range[] = { 0, 180 };
    const float* ranges = { hue_range };

    //【2】计算直方图并归一化
    calcHist( &g_hueImage, 1, 0, Mat(), hist, 1, &histSize, &ranges, true,
false );
    normalize( hist, hist, 0, 255, NORM_MINMAX, -1, Mat() );

    //【3】计算反向投影
    MatND backproj;
    calcBackProject( &g_hueImage, 1, 0, hist, backproj, &ranges, 1,
true );

    //【4】显示反向投影
    imshow( "反向投影图", backproj );

    //【5】绘制直方图的参数准备
    int w = 400; int h = 400;
```

```
    int bin_w = cvRound( (double) w / histSize );
    Mat histImg = Mat::zeros( w, h, CV_8UC3 );

    //【6】绘制直方图
    for( int i = 0; i < g_bins; i ++ )
        { rectangle( histImg, Point( i*bin_w, h ), Point( (i+1)*bin_w,
h - cvRound( hist.at<float>(i)*h/255.0 ) ), Scalar( 100, 123, 255 ), -1 ); }

    //【7】显示直方图窗口
    imshow( "直方图", histImg );
}
```

编译并运行此程序，可以通过滑动条的调节改变直方图组距（bin）的值，得到不同的反向投影效果图。如图 9.18～9.22 所示。

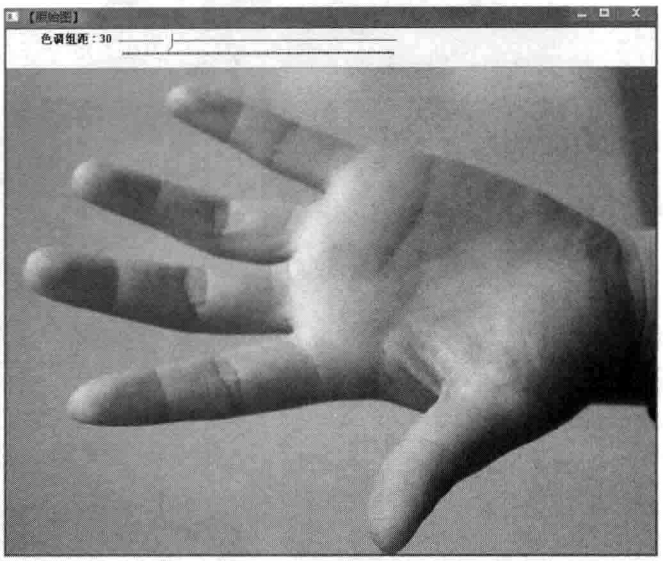


图 9.18 原始图窗口

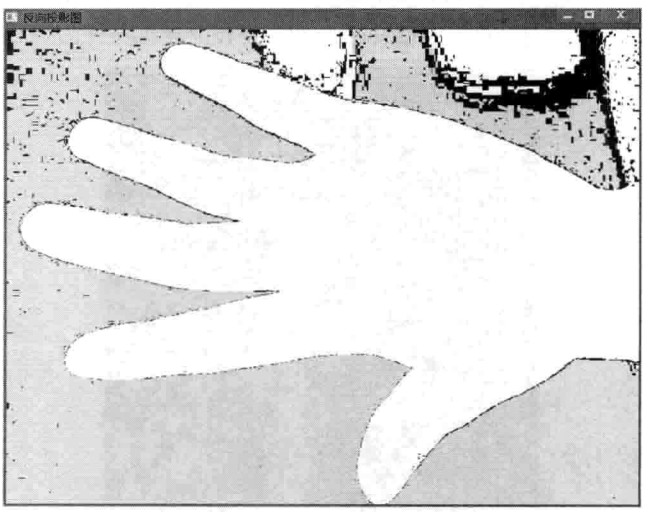


图 9.19 组距 bin 为 8 时的反向投影图

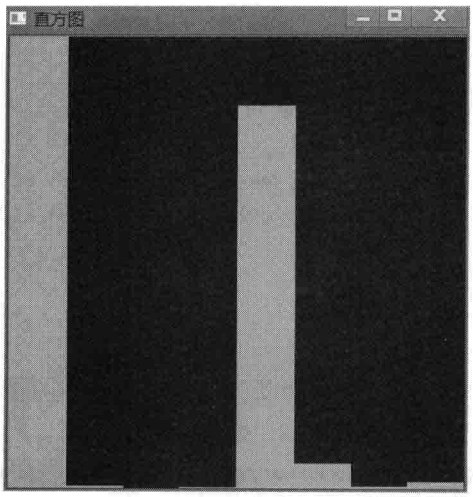


图 9.20 组距 bin 为 8 时的一维 hue 直方图

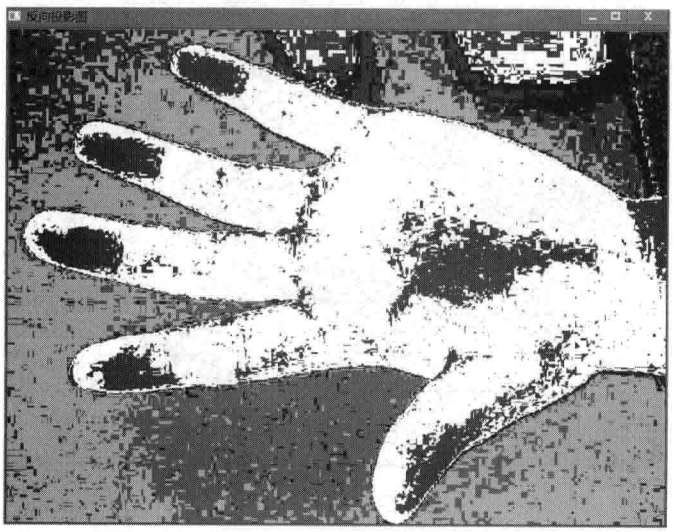


图 9.21 组距 bin 为 30 时的反向投影图

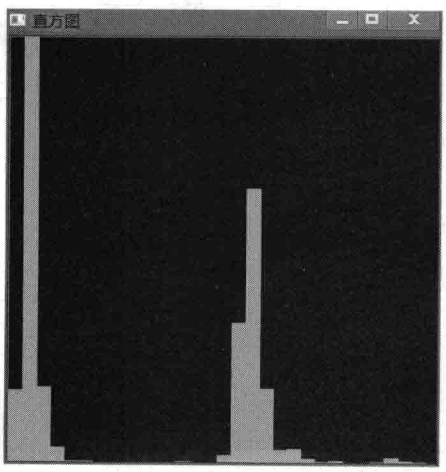


图 9.22 组距 bin 为 30 时的一维直方图

9.5 模板匹配

9.5.1 模板匹配的概念与原理

模板匹配是一项在一幅图像中寻找与另一幅模板图像最匹配（相似）部分的技术。在 OpenCV2 和 OpenCV3 中，模板匹配由 MatchTemplate()函数完成。需要注意，模板匹配不是基于直方图的，而是通过在输入图像上滑动图像块，对实际的图像块和输入图像进行匹配的一种匹配方法。

如图 9.23 所示，通过一个人脸“图像模板”，在整幅输入图像上移动这张“脸”，来寻找和这张脸相似的最优匹配。

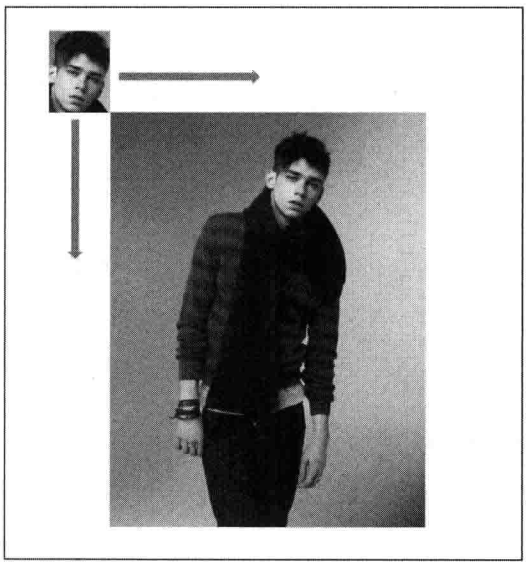


图 9.23 模板匹配图示

9.5.2 实现模板匹配：matchTemplate()函数

matchTemplate()用于匹配出和模板重叠的图像区域。

```
C++: void matchTemplate(InputArray image, InputArray templ, OutputArray
result, int method)
```

- 第一个参数，InputArray 类型的 image，待搜索的图像，且需为 8 位或 32 位浮点型图像。
- 第二个参数，InputArray 类型的 templ，搜索模板，需和源图片有一样的数据类型，且尺寸不能大于源图像。
- 第三个参数，OutputArray 类型的 result，比较结果的映射图像。其必须为单通道、32 位浮点型图像. 如果图像尺寸是 W×H 而 templ 尺寸是 w×h ，则 此参数 result 一定是 (W-w+1)×(H-h+1).
- 第四个参数，int 类型的 method，指定的匹配方法，OpenCV 为我们提供了如下 6 种图像匹配方法可供使用。

1. 平方差匹配法 method=TM_SQDIFF

这类方法利用平方差来进行匹配，最好匹配为 0。而若匹配越差，匹配值则越大。

$$R(x,y)=\sum_{x',y'}\left(T(x',y')-I(x+x',y+y')\right)^2$$

2. 归一化平方差匹配法 method=TM_SQDIFF_NORMED

$$R(x,y)=\frac{\sum_{x',y'}\left(T(x',y')-I(x+x',y+y')\right)^2}{\sqrt{\sum_{x',y'}T(x',y')^2\cdot\sum_{x',y'}I(x+x',y+y')^2}}$$

3. 相关匹配法 method=TM_CCORR

这类方法采用模板和图像间的乘法操作，所以较大的数表示匹配程度较高，0 标识最坏的匹配效果。

$$R(x,y)=\sum_{x',y'}\left(T(x',y')\cdot I(x+x',y+y')\right)$$

4. 归一化相关匹配法 method=TM_CCORR_NORMED

$$R(x,y)=\frac{\sum_{x',y'}\left(T(x',y')\cdot I'(x+x',y+y')\right)}{\sqrt{\sum_{x',y'}T(x',y')^2\cdot\sum_{x',y'}I(x+x',y+y')^2}}$$

5. 系数匹配法 method=TM_CCOEFF

这类方法将模版对其均值的相对值与图像对其均值的相关值进行匹配，1 表示完美匹配，-1 表示糟糕的匹配，而 0 表示没有任何相关性（随机序列）。

$$R(x,y)=\sum_{x',y'}\left(T'(x',y')\cdot I(x+x',y+y')\right)$$

其中：

$$T'(x',y')=T(x',y')-1/(w\cdot h)\cdot\sum_{x'',y''}T(x'',y'')$$
$$I'(x+x',y+y')=I(x+x',y+y')-1/(w\cdot h)\cdot\sum_{x'',y''}I(x+x'',y+y'')$$

6. 化相关系数匹配法 method=TM_CCOEFF_NORMED

$$R(x,y)=\frac{\sum_{x',y'}\left(T'(x',y')\cdot I'(x+x',y+y')\right)}{\sqrt{\sum_{x',y'}T'(x',y')^2\cdot\sum_{x',y'}I'(x+x',y+y')^2}}$$

上述的 6 个宏，在 OpenCV2 依然是可以加上“CV_”前缀,分别写作：CV_TM_SQDIFF、CV_TM_SQDIFF_NORMED、CV_TM_CCORR、CV_TM_CCORR_NORMED 、 CV_TM_CCOEFF 、 CV_TM_CCOEFF_NORMED。

通常，随着从简单的测量（平方差）到更复杂的测量（相关系数），我们可获

得越来越准确的匹配。然而，这同时也会以越来越大的计算量为代价。比较科学的办法是对所有这些方法多次测试实验，以便为自己的应用选择同时兼顾速度和精度的最佳方案。

9.5.3 综合示例：模板匹配

讲解完成基本概念和函数用法，下面依然是放出一个经过详细注释的示例程序源代码，演示了如何种不同的模板匹配方法对人脸进行检测。

```
//-----------------------【头文件与命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;

//-----------------------------【宏定义部分】-----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【原始图片】"        //为窗口标题定义的宏
#define WINDOW_NAME2 "【效果窗口】"        //为窗口标题定义的宏

//-----------------------------【全局变量声明部分】-----------------------
//          描述：全局变量的声明
//-------------------------------------------------------------------------
Mat g_srcImage; Mat g_templateImage; Mat g_resultImage;
int g_nMatchMethod;
int g_nMaxTrackbarNum = 5;

//-----------------------------【全局函数声明部分】-----------------------
//          描述：全局函数的声明
//-------------------------------------------------------------------------
void on_Matching( int, void* );


//-----------------------------【main()函数】-----------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】载入原图像和模板块
    g_srcImage = imread( "1.jpg", 1 );
    g_templateImage = imread( "2.jpg", 1 );

    //【2】创建窗口
    namedWindow( WINDOW_NAME1, CV_WINDOW_AUTOSIZE );
    namedWindow( WINDOW_NAME2, CV_WINDOW_AUTOSIZE );

    //【3】创建滑动条并进行一次初始化
    createTrackbar( "方法", WINDOW_NAME1, &g_nMatchMethod,
g_nMaxTrackbarNum, on_Matching );
```

```
    on_Matching( 0, 0 );

    waitKey(0);
    return 0;

}

//-------------------------【on_Matching()函数】-------------------------
//          描述：回调函数
//-----------------------------------------------------------------------
void on_Matching( int, void* )
{
    //【1】给局部变量初始化
    Mat srcImage;
    g_srcImage.copyTo( srcImage );

    //【2】初始化用于结果输出的矩阵
    int resultImage_cols =  g_srcImage.cols - g_templateImage.cols + 1;
    int resultImage_rows = g_srcImage.rows - g_templateImage.rows + 1;
    g_resultImage.create( resultImage_cols, resultImage_rows,
CV_32FC1 );

    //【3】进行匹配和标准化
    matchTemplate( g_srcImage, g_templateImage, g_resultImage,
g_nMatchMethod );
    normalize( g_resultImage, g_resultImage, 0, 1, NORM_MINMAX, -1,
Mat() );

    //【4】通过函数 minMaxLoc 定位最匹配的位置
    double minValue; double maxValue; Point minLocation; Point
maxLocation;
    Point matchLocation;
    minMaxLoc( g_resultImage, &minValue, &maxValue, &minLocation,
&maxLocation, Mat() );

    //【5】对于方法 SQDIFF 和 SQDIFF_NORMED，越小的数值有着更高的匹配结果. 而
其余的方法，数值越大匹配效果越好
    //此句代码的 OpenCV2 版为：
    //if( g_nMatchMethod  == CV_TM_SQDIFF || g_nMatchMethod ==
CV_TM_SQDIFF_NORMED )
    //此句代码的 OpenCV3 版为：
    if( g_nMatchMethod  == TM_SQDIFF || g_nMatchMethod ==
TM_SQDIFF_NORMED )
        { matchLocation = minLocation; }
    else
        { matchLocation = maxLocation; }

    //【6】绘制出矩形，并显示最终结果
    rectangle( srcImage, matchLocation, Point( matchLocation.x +
g_templateImage.cols , matchLocation.y + g_templateImage.rows ),
Scalar(0,0,255), 2, 8, 0 );
    rectangle( g_resultImage, matchLocation, Point( matchLocation.x +
```

```
    g_templateImage.cols , matchLocation.y + g_templateImage.rows ),
    Scalar(0,0,255), 2, 8, 0 );

        imshow( WINDOW_NAME1, srcImage );
        imshow( WINDOW_NAME2, g_resultImage );

    }
```

运行此程序，可以得到三个窗口。说明窗口、附有滑动条的原始图窗口以及根据滑动条数值变化的匹配效果图窗口。

首先，让我们通过说明窗口了解此程序的使用方法。如图 9.25 所示。

图 9.25 序说明窗口

接着，让我们一起看看匹配的效果图。如图 9.26～9.32 所示。

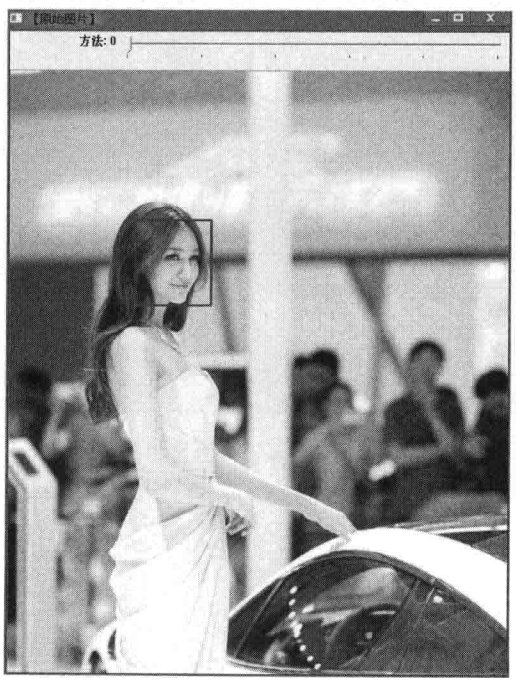


图 9.26 附有滑动条的原始图窗口

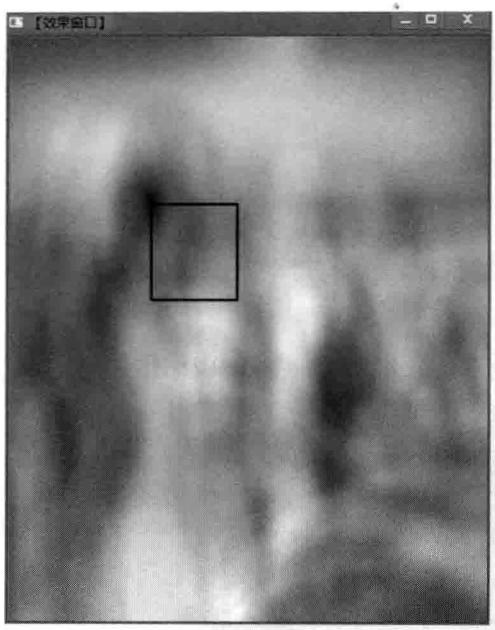


图 9.27 平方差匹配法(SQDIFF)匹配图

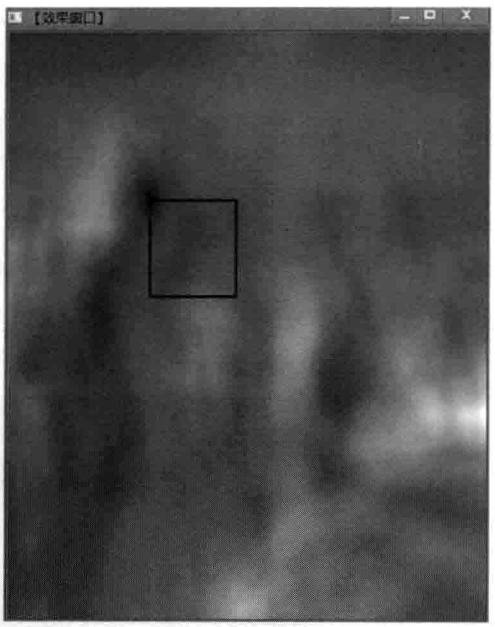


图 9.28 归一化平方差匹配法(SQDIFF NORMED)匹配图

图 9.29 相关匹配法（TM CCORR）效果图

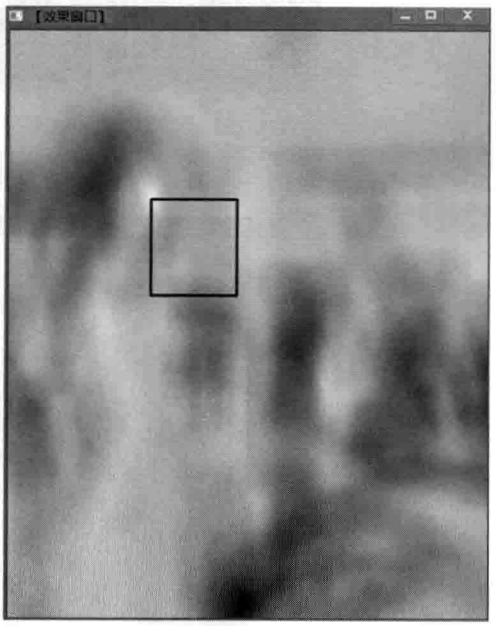


图 9.30 归一化相关匹配法（TM CCORR NORMED）

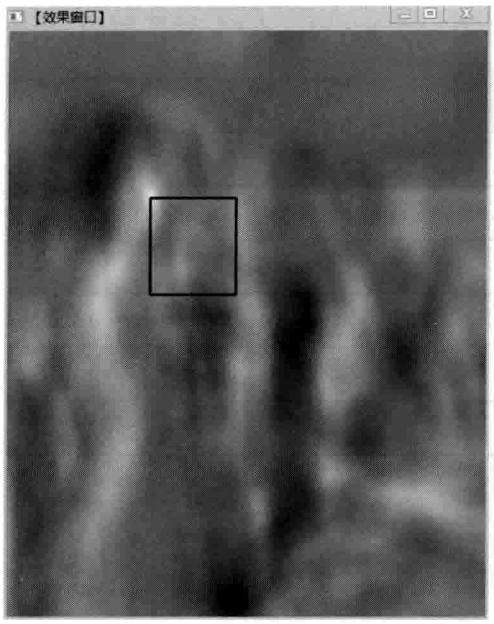


图 9.31 相关系数匹配法（TM COEFF）

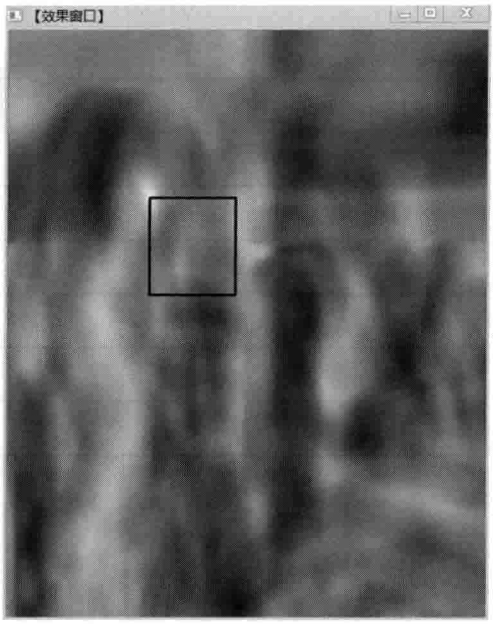


图 9.32 归一化相关系数匹配法（TM COEFF NORMED）

可以发现，除了相关匹配法（TM CCORR）得到了错误的匹配结果之外，其他的 5 种匹配方式结果都匹配较为准确。

9.6 本章小结

本章我们学习了广泛运用于很多计算机视觉运用当中的直方图，而简单点说，直方图就是对数据进行统计的一种方法。然后还讲到了反向投影和模板匹配。所谓反向投影就是首先计算某一特征的直方图模型，最后使用模型去寻找图像中存在的该特征的方法。而模板匹配是一项在一幅图像中寻找与另一幅模板图像最匹配（相似）部分的技术。

本章核心函数清单

函数名称	说明	对应讲解章节
calcHist	计算一个或者多个阵列的直方图	9.2.1
minMaxLoc	在数组中找到全局最小值和最大值	9.2.2
compareHist	对两幅直方图进行比较	9.3.1
calcBackProject	计算直方图的反向投影	9.4.5
mixChannels	由输入参数复制某通道到输出参数特定的通道中	9.4.6
matchTemplate	匹配出和模板重叠的图像区域	9.5.2

本章示例程序清单

示例程序序号	程序说明	对应章节
79	H-S 二维直方图的绘制	9.2.3
80	一维直方图的绘制	9.2.4
81	RGB 三色直方图的绘制	9.2.5
82	直方图对比	9.3.2
83	反向投影	9.4.7
84	模板匹配	9.5.3

第四部分
深入 feature2d 组件

features2d 组件，也就是 Features 2D，是 OpenCV 的 2D 功能框架。

在 OpenCV2 中，将 features2d 组件与其余相关组件配合使用，已经达到了比较好的科研和实用效果。

自 OpenCV2 以来的众多著名的特征检测算子（如 SIFT、SURF、ORB 算子等）所依赖的稳定版的特征检测与匹配相关的核心源代码已经从官方发行的 OpenCV3 中移除，而转移到了一个名为 xfeature2d 的第三方库当中。此库还处于开发阶段，现阶段想投入使用要额外到 Gitbub 下载，并需要进行繁琐的配置。且此库运行效果不稳定，不适合用于开发。所以，本书第 11 章的特征检测与匹配部分，将以稳定的 OpenCV2 进行讲解和代码实现。第 11 章的工程源代码仅在 OpenCV2 版的源码包中存在。第 10 章的内容依然是 OpenCV2、OpenCV3 双版本的实现和工程源码。

本部分包含如下内容：

- 特征检测和描述
- 特征检测器（Feature Detectors）通用接口
- 描述符提取器（Descriptor Extractors）通用接口
- 描述符匹配器（Descriptor Matchers）通用接口
- 通用描述符（Generic Descriptor）匹配器通用接口
- 关键点绘制函数和匹配功能绘制函数

第 10 章
角点检测

导读

角点检测（Corner Detection）是计算机视觉系统中用来获得图像特征的一种方法，广泛应用于运动检测、图像匹配、视频跟踪、三维建模和目标识别等领域中，也称为特征点检测。

角点通常被定义为两条边的交点，更严格地说法是，角点的局部邻域应该具有两个不同区域的不同方向的边界。而实际应用中，大多数所谓的角点检测方法检测的是拥有特定特征的图像点，而不仅仅是“角点”。这些特征点在图像中有具体的坐标，并具有某些数学特征，如局部最大或最小灰度、某些梯度特征等。

将以详细注释的代码为载体，学习如何用新版的 OpenCV 来检测角点。

本章你将学到：

- 什么是角点
- 角点检测的概念
- 用 OpenCV 实现 Harris 角点检测
- 用 OpenCV 实现 Shi-Tomasi 角点检测
- 亚像素级角点检测

10.1 Harris 角点检测

10.1.1 兴趣点与角点

在图像处理和与计算机视觉领域，兴趣点（interest points），也被称作关键点（key points）、特征点（feature points）。它被大量用于解决物体识别、图像识别、图像匹配、视觉跟踪、三维重建等一系列的问题。我们不再观察整幅图，而是选择某些特殊的点，然后对它们进行局部有的放矢地分析。如果能检测到足够多的这种点，同时它们的区分度很高，并且可以精确定位稳定的特征，那么这个方法就具有实用价值。

图像特征类型可以被分为如下三种：

- 边缘
- 角点 (感兴趣关键点)
- 斑点(Blobs)(感兴趣区域)

其中，角点是个很特殊的存在。如果某一点在任意方向的一个微小变动都会引起灰度很大的变化，那么我们就把它称之为角点。角点作为图像上的特征点，包含有重要的信息，在图像融合和目标跟踪及三维重建中有重要的应用价值。它们在图像中可以轻易地定位，同时，在人造物体场景，比如门、窗、桌等处也随处可见。因为角点位于两条边缘的交点处，代表了两个边缘变化的方向上的点，所以它们是可以精确定位的二维特征，甚至可以达到亚像素的精度。又由于其图像梯度有很高的变化，这种变化是可以用来帮助检测角点的。需要注意的是，角点与位于相同强度区域上的点不同，与物体轮廓上的点也不同，因为轮廓点难以在相同的其他物体上精确定位。

另外，关于角点的具体描述可以有如下几种：

- 一阶导数(即灰度的梯度)的局部最大所对应的像素点；
- 两条及两条以上边缘的交点；
- 图像中梯度值和梯度方向的变化速率都很高的点；
- 角点处的一阶导数最大，二阶导数为零，它指示了物体边缘变化不连续的方向。

10.1.2 角点检测

现有的角点检测算法并不是都十分的健壮。很多方法都要求有大量的训练集和冗余数据来防止或减少错误特征的出现。另外，角点检测方法的一个很重要的评价标准是其对多幅图像中相同或相似特征的检测能力，并且能够应对光照变化、图像旋转等图像变化。

在当前的图像处理领域，角点检测算法可归纳为以下三类。

- 基于灰度图像的角点检测

- 基于二值图像的角点检测
- 基于轮廓曲线的角点检测

而基于灰度图像的角点检测又可分为基于梯度、基于模板和基于模板梯度组合三类方法。其中基于模板的方法主要考虑像素领域点的灰度变化，即图像亮度的变化，将与邻点亮度对比足够大的点定义为角点。常见的基于模板的角点检测算法有 Kitchen-Rosenfeld 角点检测算法，Harris 角点检测算法、KLT 角点检测算法及 SUSAN 角点检测算法。

接下来，让我们一起来了解 harris 角点检测。

10.1.3 harris 角点检测

harris 角点检测是一种直接基于灰度图像的角点提取算法，稳定性高，尤其对 L 型角点检测精度高。但由于采用了高斯滤波，运算速度相对较慢，角点信息有丢失和位置偏移的现象，而且角点提取有聚簇现象。

10.1.4 实现 Harris 角点检测：cornerHarris()函数

cornerHarris 函数用于在 OpenCV 中运行 Harris 角点检测算子来进行角点检测。和 cornerMinEigenVal()以及 cornerEigenValsAndVecs()函数类似，cornerHarris 函数对于每一个像素（x,y）在 blockSize×blockSize 邻域内，计算 2x2 梯度的协方差矩阵 $M(x,y)$，接着它计算如下式子：

$$\mathrm{dst}(x,y)=\det M^{(x,y)}-k \cdot (\mathrm{tr}M^{(x,y)})^2$$

就可以找出输出图中的局部最大值，即找出了角点。

其函数原型和参数解析如下。

```
C++: void cornerHarris(InputArray src,OutputArray dst, int blockSize,
int ksize, double k, intborderType=BORDER_DEFAULT )
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，且须为单通道 8 位或者浮点型图像。
- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，即这个参数用于存放 Harris 角点检测的输出结果，和源图片有一样的尺寸和类型。
- 第三个参数，int 类型的 blockSize，表示邻域的大小，更多详细信息在 cornerEigenValsAndVecs()中有讲到。
- 第四个参数，int 类型的 ksize，表示 Sobel()算子的孔径大小。
- 第五个参数，double 类型的 k，Harris 参数。
- 第六个参数，int 类型的 borderType，图像像素的边界模式。注意它有默认值 BORDER_DEFAULT。更详细的解释，参考 borderInterpolate()函数。

讲解完这个函数，我们看一个 Harris 角点检测示例程序，其中还用到了之前讲到的 threshold 函数。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
using namespace cv;

int main()
{
    //以灰度模式载入图像并显示
    Mat srcImage = imread("1.jpg", 0);
    imshow("原始图", srcImage);

    //进行 Harris 角点检测找出角点
    Mat cornerStrength;
    cornerHarris(srcImage, cornerStrength, 2, 3, 0.01);

    //对灰度图进行阈值操作，得到二值图并显示
    Mat harrisCorner;
    threshold(cornerStrength, harrisCorner, 0.00001, 255,
THRESH_BINARY);
    imshow("角点检测后的二值效果图", harrisCorner);

    waitKey(0);
    return 0;
}
```

运行截图如图 10.1 和图 10.2 所示。

图 10.1 原始图

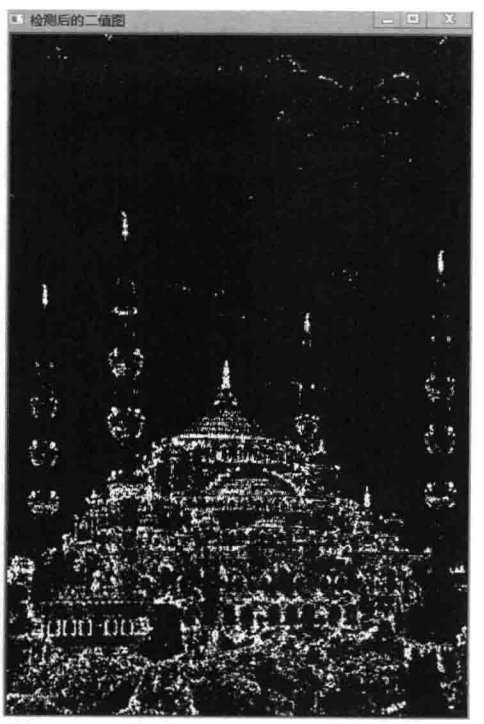


图 10.2 Harris 角点检测二值图

10.1.5 综合示例：harris 角点检测与绘制

本次综合示例为调节滚动条来控制阈值，以控制的 harris 检测角点的数量。一共有三个图片窗口，分别为显示原始图的窗口、包含滚动条的彩色效果图窗口，以及灰度图效果图窗口。

下面让我们一起来欣赏详细注释过后的完整源代码。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
using namespace cv;


//----------------------------【宏定义部分】-----------------------------
//  描述：定义一些辅助宏
//-------------------------------------------------------------------------
#define WINDOW_NAME1 "【程序窗口 1】"        //为窗口标题定义的宏
#define WINDOW_NAME2 "【程序窗口 2】"        //为窗口标题定义的宏

//-------------------------【全局变量声明部分】--------------------------
//      描述：全局变量声明
//-------------------------------------------------------------------------
Mat g_srcImage, g_srcImage1,g_grayImage;
int thresh = 30; //当前阈值
int max_thresh = 175; //最大阈值


//-------------------------【全局函数声明部分】--------------------------
//      描述：全局函数声明
//-------------------------------------------------------------------------
void on_CornerHarris( int, void* );//回调函数


//----------------------------【main()函数】-----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main( int argc, char** argv )
{
    //【1】载入原始图并进行克隆保存
    g_srcImage = imread( "1.jpg", 1 );
     if(!g_srcImage.data ) { printf("读取图片错误，请确定目录下是否有 imread
函数指定的图片存在~！ \n"); return false; }
     imshow("原始图",g_srcImage);
    g_srcImage1=g_srcImage.clone();

    //【2】存留一张灰度图
    cvtColor( g_srcImage1, g_grayImage, COLOR_BGR2GRAY );
```

```
    //【3】创建窗口和滚动条
    namedWindow( WINDOW_NAME1, WINDOW_AUTOSIZE );
    createTrackbar( "阈值: ", WINDOW_NAME1, &thresh, max_thresh,
on_CornerHarris );

    //【4】调用一次回调函数，进行初始化
    on_CornerHarris( 0, 0 );

    waitKey(0);
    return(0);
}

//-----------------------【on_HoughLines()函数】-----------------------
//      描述：回调函数
//-----------------------------------------------------------------------

void on_CornerHarris( int, void* )
{
    //-----------------------【1】定义一些局部变量-----------------------
    Mat dstImage;//目标图
    Mat normImage;//归一化后的图
    Mat scaledImage;//线性变换后的八位无符号整型的图

    //---------------------------【2】初始化---------------------------
    //置零当前需要显示的两幅图，即清除上一次调用此函数时他们的值
    dstImage = Mat::zeros( g_srcImage.size(), CV_32FC1 );
    g_srcImage1=g_srcImage.clone();

    //---------------------------【3】正式检测---------------------------
    //进行角点检测
    cornerHarris( g_grayImage, dstImage, 2, 3, 0.04, BORDER_DEFAULT );

    // 归一化与转换
    normalize( dstImage, normImage, 0, 255, NORM_MINMAX, CV_32FC1,
Mat() );
    convertScaleAbs( normImage, scaledImage );//将归一化后的图线性变换成8
位无符号整型

    //---------------------------【4】进行绘制---------------------------
    // 将检测到的，且符合阈值条件的角点绘制出来
    for( int j = 0; j < normImage.rows ; j++ )
    { for( int i = 0; i < normImage.cols; i++ )
    {
        if( (int) normImage.at<float>(j,i) > thresh+80 )
        {
         circle( g_srcImage1, Point( i, j ), 5,  Scalar(10,10,255), 2,
8, 0 );
         circle( scaledImage, Point( i, j ), 5,  Scalar(0,10,255), 2, 8,
0 );
        }
    }
    }
```

```
    //-------------------------【4】显示最终效果-------------------------
    imshow( WINDOW_NAME1, g_srcImage1 );
    imshow( WINDOW_NAME2, scaledImage );

}
```

编译并运行程序，我们可以得到如下非常漂亮的异域建筑群角点检测效果图。如图 10.3～10.7 所示。

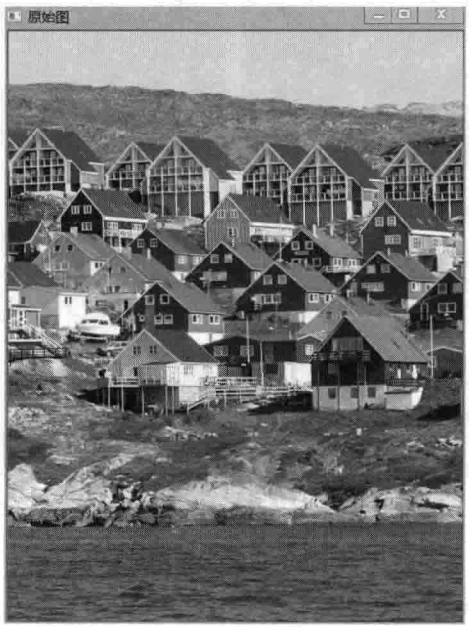


图 10.3 原始图

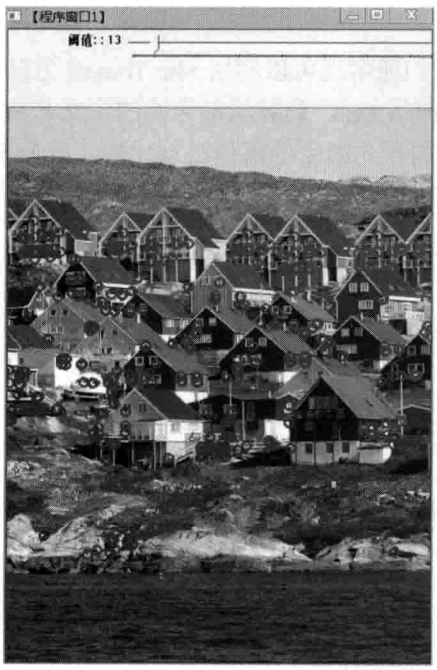


图 10.4 阈值为 13 时的检测图（1）

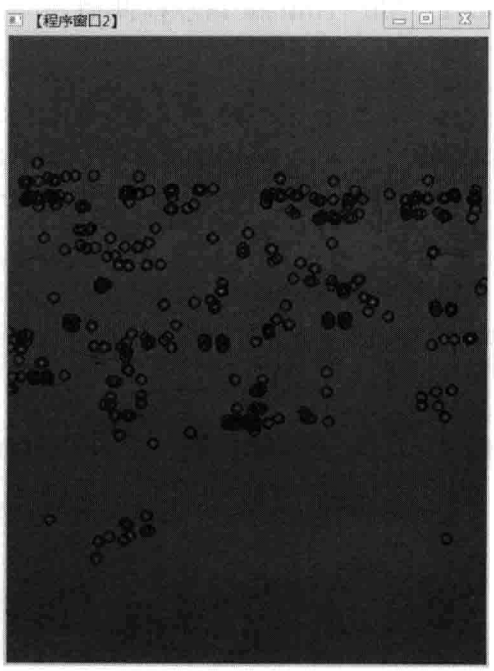


图 10.5 阈值为 13 时的检测图（2）

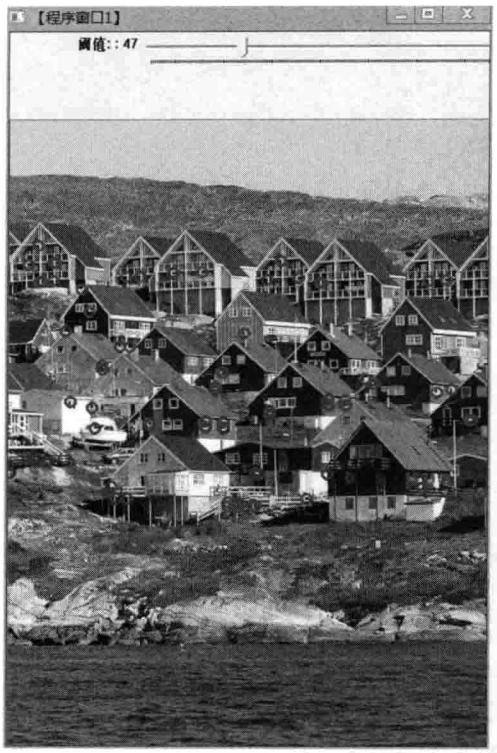


图 10.6 阈值为 47 时的检测图（1）

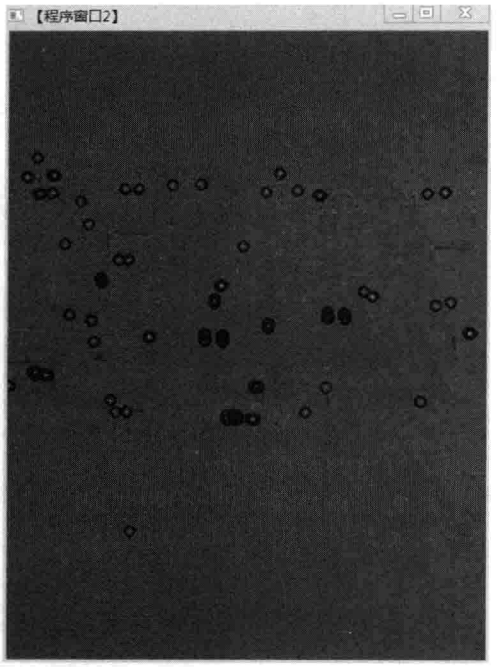


图 10.7 阈值为 47 时的检测图（2）

10.2 Shi-Tomasi 角点检测

10.2.1 Shi-Tomasi 角点检测概述

除了利用 Harris 进行角点检测之外，我们通常还可以利用 Shi-Tomasi 方法进行角点检测。Shi-Tomasi 算法是 Harris 算法的改进，此算法最原始的定义是将矩阵 M 的行列式值与 M 的迹相减，再将差值同预先给定的阈值进行比较。后来 Shi 和 Tomasi 提出改进了方法，若两个特征值中较小的一个大于最小阈值，则会得到强角点。

由于 Shi-Tomasi 算子是 1994 年在文章《Good Features to Track》中被提出的，OpenCV 实现此算法的函数名便定义为 goodFeaturesToTrack。下面我们来看看此函数的用法。

10.2.2 确定图像强角点：goodFeaturesToTrack()函数

goodFeaturesToTrack()函数结合了 Shi-Tomasi 算子，用于确定图像的强角点。

```
C++: void goodFeaturesToTrack(
InputArray image,
 OutputArray corners,
 int maxCorners,
```

```
double qualityLevel,
double minDistance,
InputArray mask=noArray(),
int blockSize=3,
bool useHarrisDetector=false,
double k=0.04 )
```

- 第一个参数，InputArray 类型的 image，输入图像，须为 8 位或浮点型 32 位单通道图像。
- 第二个参数，OutputArray 类型的 corners，检测到的角点的输出向量。
- 第三个参数，int 类型的 maxCorners，角点的最大数量。
- 第四个参数，double 类型的 qualityLevel，角点检测可接受的最小特征值。其实实际用于过滤角点的最小特征值是 qualityLevel 与图像中最大特征值的乘积。所以 qualityLevel 通常不会超过 1（常用的值为 0.10 或者 0.01）。而检测完所有的角点后，还要进一步剔除掉一些距离较近的角点。
- 第五个参数，double 类型的 minDistance，角点之间的最小距离，此参数用于保证返回的角点之间的距离不小于 minDistance 个像素。
- 第六个参数，InputArray 类型的 mask，可选参数，表示感兴趣区域，有默认值 noArray()。若此参数非空（需为 CV_8UC1 类型，且和第一个参数 image 有相同的尺寸），便用于指定角点检测区域。
- 第七个参数，int 类型的 blockSize，有默认值 3，是计算导数自相关矩阵时指定的邻域范围。
- 第八个参数，bool 类型的 useHarrisDetector，默认值 false，指示是否使用 Harris 角点检测。
- 第九个参数，double 类型的 k，有默认值 0.04，为用于设置 Hessian 自相关矩阵行列式的相对权重的权重系数。

另外值得一提的是，goodFeaturesToTrack 函数可用来初始化一个基于点的对象跟踪操作。

10.2.3 综合示例：Shi-Tomasi 角点检测

作为一本编程入门教程，依然是发挥我们的特色——用详细注释，条理清晰的代码说话。下面就将放出一个详细注释的以 goodFeaturesToTrack 函数为核心，进行 Shi-Tomasi 角点检测的示例程序。

```
//-----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <iostream>
using namespace cv;
using namespace std;

//-----------------------------【宏定义部分】-----------------------------
```

```
//  描述：定义一些辅助宏
//------------------------------------------------------------------------
#define WINDOW_NAME "【Shi-Tomasi 角点检测】"        //为窗口标题定义的宏


//-------------------------【全局变量声明部分】-------------------------
//          描述：全局变量声明
//------------------------------------------------------------------------
Mat g_srcImage, g_grayImage;
int g_maxCornerNumber = 33;
int g_maxTrackbarNumber = 500;
RNG g_rng(12345);


//-----------------【goodFeaturesToTrack_Demo()函数】-----------------
//          描述：响应滑动条移动消息的回调函数
//------------------------------------------------------------------------
void on_GoodFeaturesToTrack( int, void* )
{
    //【1】对变量小于等于1时的处理
    if( g_maxCornerNumber <= 1 ) { g_maxCornerNumber = 1; }

    //【2】Shi-Tomasi 算法（goodFeaturesToTrack 函数）的参数准备
    vector<Point2f> corners;
    double qualityLevel = 0.01;//角点检测可接受的最小特征值
    double minDistance = 10;//角点之间的最小距离
    int blockSize = 3;//计算导数自相关矩阵时指定的邻域范围
    double k = 0.04;//权重系数
    Mat copy = g_srcImage.clone();     //复制源图像到一个临时变量中，作为感
兴趣区域

    //【3】进行 Shi-Tomasi 角点检测
    goodFeaturesToTrack( g_grayImage,//输入图像
            corners,//检测到的角点的输出向量
            g_maxCornerNumber,//角点的最大数量
            qualityLevel,//角点检测可接受的最小特征值
            minDistance,//角点之间的最小距离
            Mat(),//感兴趣区域
            blockSize,//计算导数自相关矩阵时指定的邻域范围
            false,//不使用 Harris 角点检测
            k );//权重系数


    //【4】输出文字信息
    cout<<">此次检测到的角点数量为："<<corners.size()<<endl;

    //【5】绘制检测到的角点
    int r = 4;
    for(unsigned int i = 0; i < corners.size(); i++ )
        {
         //以随机的颜色绘制出角点
         circle( copy, corners[i], r, Scalar(g_rng.uniform(0,255),
```

```
g_rng.uniform(0,255),
            g_rng.uniform(0,255)), -1, 8, 0 );
        }

    //【6】显示（更新）窗口
    imshow( WINDOW_NAME, copy );
}


//-----------------------------【main()函数】--------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//---------------------------------------------------------------------
int main()
{
    //【1】载入源图像并将其转换为灰度图
    g_srcImage = imread("1.jpg", 1 );
    cvtColor( g_srcImage, g_grayImage, COLOR_BGR2GRAY );

    //【2】创建窗口和滑动条，并进行显示和回调函数初始化
    namedWindow( WINDOW_NAME, WINDOW_AUTOSIZE );
    createTrackbar( "最大角点数", WINDOW_NAME, &g_maxCornerNumber,
g_maxTrackbarNumber, on_GoodFeaturesToTrack );
    imshow( WINDOW_NAME, g_srcImage );
    on_GoodFeaturesToTrack( 0, 0 );

    waitKey(0);
    return(0);
}
```

运行此程序，得到如图 10.8 所示的角点检测图，并且调节滑动条，可改变能检测到的最大角点数量。如图 10.9 和 10.10 所示。在程序代码中，我们设定可检测到的最大角点数量为 500。

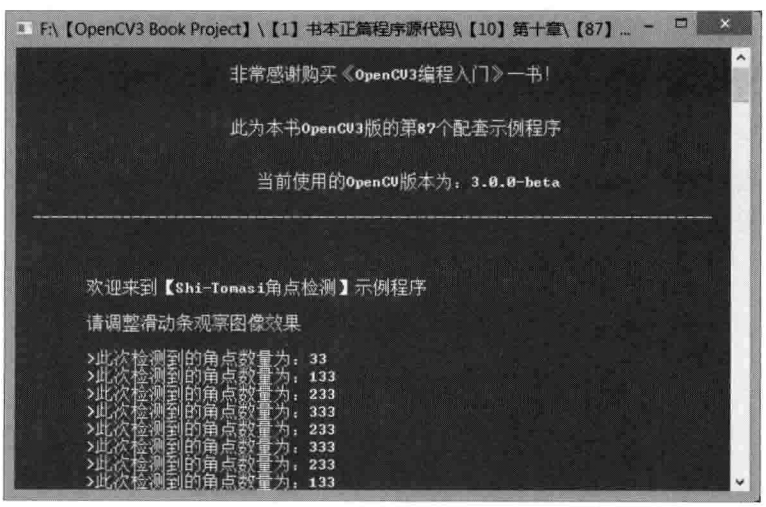


图 10.8 文本信息输出窗口

图 10.9 最大检测角点数为 33 时程序截图 图 10.10 最大检测角点数为 500 时的程序截图

10.3 亚像素级角点检测

10.3.1 背景概述

若我们进行图像处理的目的不是提取用于识别的特征点而是进行几何测量，这通常需要更高的精度，而函数 goodFeaturesToTrack()只能提供简单的像素的坐标值，也就是说，有时候会需要实数坐标值而不是整数坐标值。

亚像素级角点检测的位置在摄像机标定、跟踪并重建摄像机的轨迹，或者重建被跟踪目标的三维结构时，是一个基本的测量值。

下面我们将讨论如何将所求得的角点位置精确到亚像素级精度。一个向量和与其正交的向量的点积为 0，角点则满足如图 10.11 所示情况。

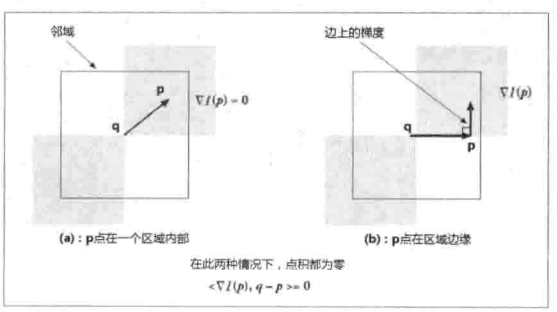


图 10.11 计算亚像素级精度的角点

其中，（a）点 p 附近的图像是均匀的，其梯度为 0；（b）边缘的梯度与沿边缘方向的 q-p 向量正交。在图中的两种情况下，p 点梯度与 q-p 向量的点积均为 0。

图 10.11 中，我们假设起始角点 q 在实际亚像素级角点的附近。检测所有的 q-p 向量。若点 p 位于一个均匀的区域，则点 p 处的梯度为 0。若 q-p 向量的方向与边缘的方向一致，则此边缘上 p 点处的梯度与 q-p 向量正交，在这两种情况下，p 点处的梯度与 q-p 向量的点积为 0。我们可以在 p 点周围找到很多组梯度以及相关的向量 q-p，令其点集为 0，然后可以通过求解方程组，方程组的解即为角点 q 的亚像素级精度的位置，也就是精确的角点位置。

OpenCV 为我们提供了 cornerSubPix()函数，用于发现亚像素精度的角点位置。

10.3.2 寻找亚像素角点：cornerSubPix()函数

cornerSubPix 函数用于寻找亚像素角点位置（不是整数类型的位置，而是更精确的浮点类型位置）。

```
C++: void cornerSubPix(
InputArray image,
 InputOutputArray corners,
 Size winSize,
 Size zeroZone,
 TermCriteria criteria)
```

- 第一个参数，InputArray 类型的 image，输入图像，即源图像。
- 第二个参数，InputOutputArray 类型的 corners，提供输入角点的初始坐标和精确的输出坐标。
- 第三个参数，Size 类型的 winSize，搜索窗口的一半尺寸。若 winSize = Size（5,5），那么就表示使用（5*2+1）×（5*2+1）=11×11 大小的搜索窗口。
- 第四个参数，Size 类型的 zeroZone，表示死区的一半尺寸。而死区为不对搜索区的中央位置做求和运算的区域，用来避免自相关矩阵出现的某些可能的奇异性。值为（-1,-1）表示没有死区。
- 第五个参数，TermCriteria 类型的 criteria，求角点的迭代过程的终止条件。即角点位置的确定，要么迭代数大于某个设定值，或者是精确懂达到某个设定值。criteria 可以是最大迭代数目，或者是设定的精确度，也可以是它们的组合。

10.3.3 综合示例：亚像素级角点检测

这个程序在上一小节 Shi-Tomasi 角点检测示例程序的基础上，在 on_GoodFeaturesToTrack()回调函数中加上以下进行亚像素角点检测的代码，就成为了亚像素级角点检测的示例程序，需添加的代码如下。

```
//------------------【on_GoodFeaturesToTrack()函数】------------------
//          描述：响应滑动条移动消息的回调函数
//-----------------------------------------------------------------------
void on_GoodFeaturesToTrack( int, void* )
{
    //.....

    //【7】亚像素角点检测的参数设置
    Size winSize = Size( 5, 5 );
    Size zeroZone = Size( -1, -1 );
    //此句代码的 OpenCV2 版为：
    //TermCriteria criteria = TermCriteria( CV_TERMCRIT_EPS +
CV_TERMCRIT_ITER, 40, 0.001 );
    //此句代码的 OpenCV3 版为：
    TermCriteria criteria = TermCriteria( TermCriteria::EPS +
TermCriteria::MAX_ITER, 40, 0.001 );

    //【8】计算出亚像素角点位置
    cornerSubPix( g_grayImage, corners, winSize, zeroZone, criteria );

    //【9】输出角点信息
    for( int i = 0; i < corners.size(); i++ )
     {cout<<
     " \t>>精确角点坐["<<i<<"]("<<corners[i].x<<","<<corners[i].y<<")"
         <<endl; }
}
```

运行此程序，得到的窗口和之前的【Shi-Tomasi 角点检测】一致，区别在于在控制台窗口中输出了检测到的浮点型亚像素角点精确坐标值。如图 10.12～10.15 所示。

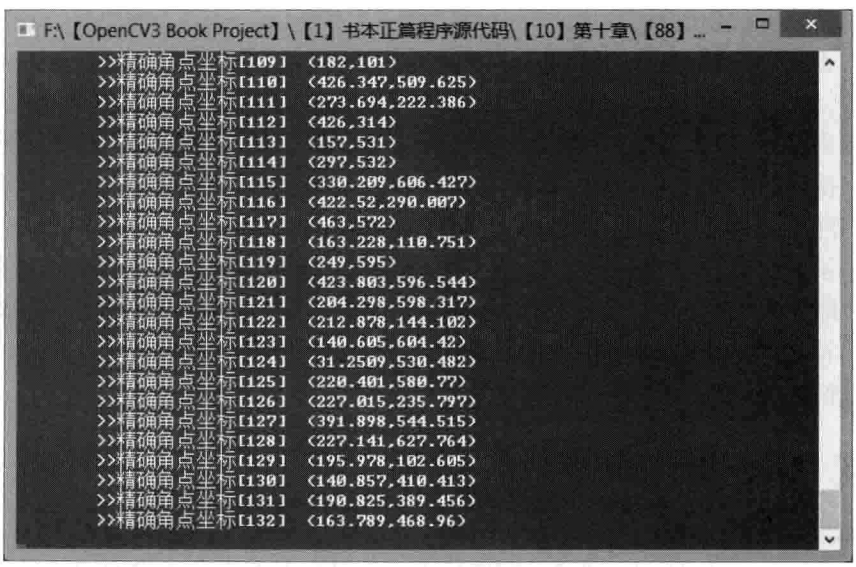


图 10.12 亚像素角点信息输出窗口（最大角点数为 133 时）

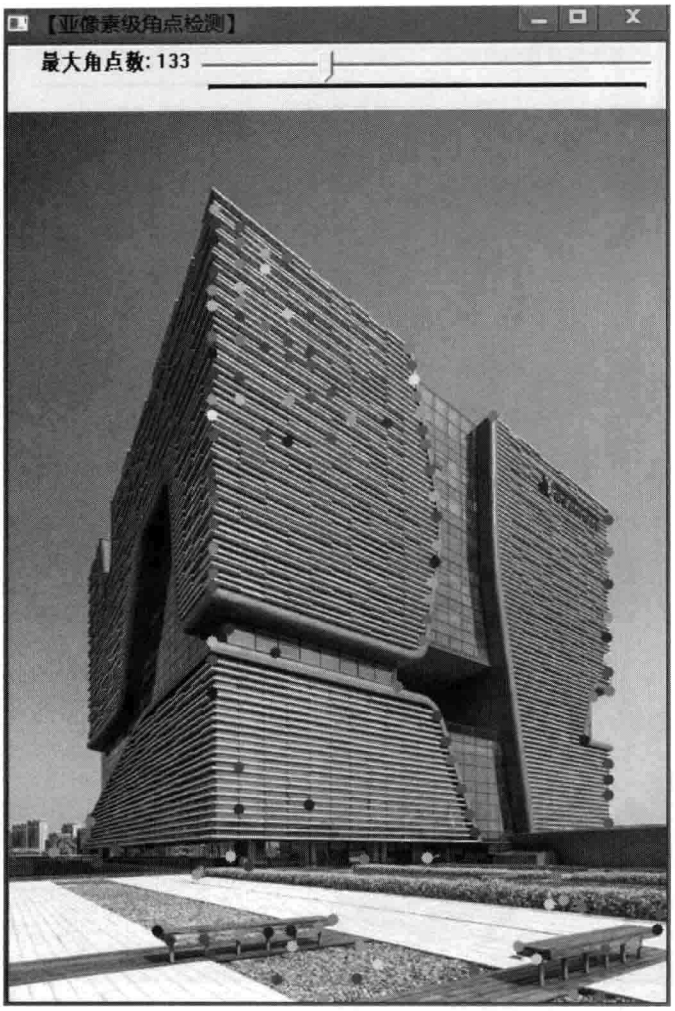


图 10.13 亚像素角点检测窗口（最大角点数为 133 时）

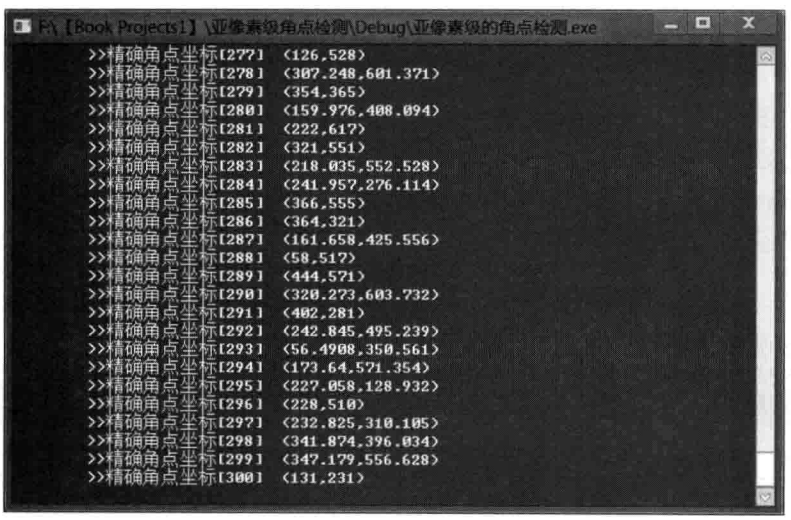


图 10.14 亚像素角点信息输出窗口（最大角点数为 301 时）

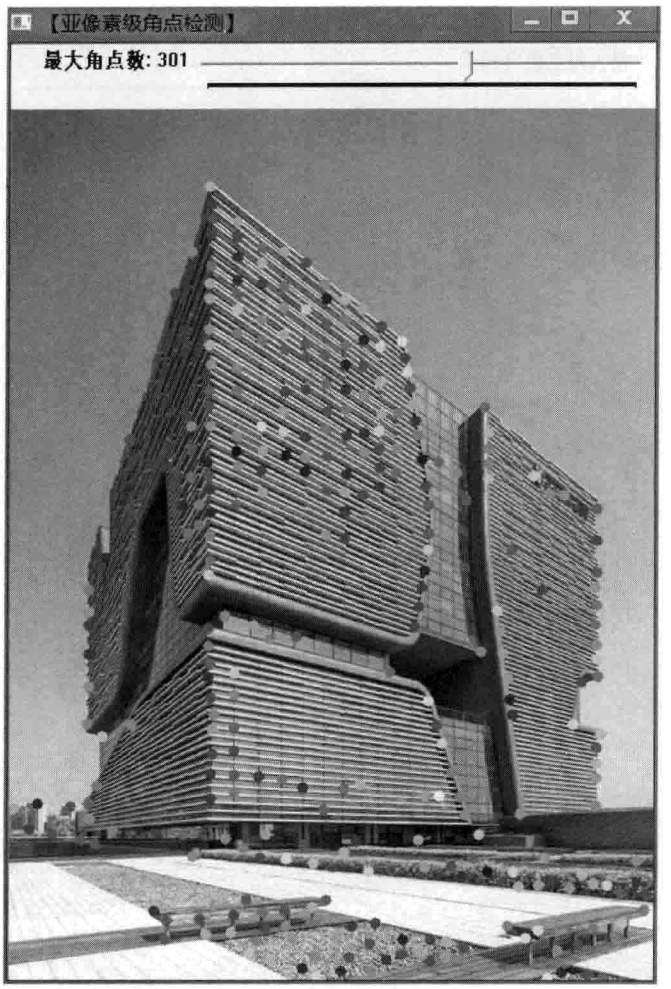


图 10.15 亚像素角点检测窗口（最大角点数为 301 时）

10.4 本章小结

本章我们讲解了 Harris 角点检测和 Shi-Tomasi 角点检测，以及一种亚像素角点检测方法。当然，也可以自己制作角点检测的函数，需要用到 cornerMinEigenVal 函数和 minMaxLoc 函数。最后的特征点选取，判断条件要根据自己的情况编辑。如果对特征点，角点的精度要求更高，可以用 cornerSubPix 函数将角点定位到子像素。

本章核心函数清单

函数名称	说明	对应讲解章节
cornerHarris	运行 Harris 角点检测算子来进行角点检测	10.1.4
goodFeaturesToTrack	结合 Shi-Tomasi 算子确定图像的强角点	10.2.2
cornerSubPix	寻找亚像素角点位置	10.3.2

本章示例程序清单

示例程序序号	程序说明	对应章节
85	实现 Harris 角点检测：cornerHarris()函数的使用	10.1.4
86	harris 角点检测与绘制	10.1.5
87	Shi-Tomasi 角点检测	10.2.3
88	亚像素级角点检测	10.3.3

第 11 章
特征检测与匹配

导读

特征点的检测和匹配是计算机视觉中非常重要的技术之一。在物体检测、视觉跟踪、三维重建等领域都有很广泛的应用。

OpenCV 为我们提供了以下 10 种特征检测方法（破折号后面为对应的类名）。

- “FAST”——FastFeatureDetector
- “STAR”——StarFeatureDetector
- “SIFT”——SIFT (nonfree module)
- “SURF”——SURF (nonfree module)
- “ORB”——ORB
- “MSER”——MSER
- “GFTT”——GoodFeaturesToTrackDetector
- “HARRIS”——GoodFeaturesToTrackDetector（配合 Harris detector 操作）
- “Dense”——DenseFeatureDetector
- “SimpleBlob”——SimpleBlobDetector

其中，Harris 方法在上一章中已经详细讲解过。而在本章中，我们会对最常用 SIFT、SURF 和 ORB 等算法用 OpenCV2 进行说明、讲解以及编程实现。

需要再次说明，由于目前发行的 OpenCV3 中，众多著名的特征检测算子（如 SIFT、SURF、ORB 算子等）所依赖的稳定版的源代码已经从官方发行的 OpenCV3 中移除，而转移到一个名为 xfeature2d 的第三方库当中。所以本章内容将以稳定的 OpenCV2 版本进行讲解和代码实现，本章的工程源代码也仅在 OpenCV2 版的源码包中存在。

11.1 SURF 特征点检测

11.1.1 SURF 算法概览

SURF，英文全称为 SpeededUp Robust Features，直译为“加速版的具有鲁棒性的特征“算法，由 Bay 在 2006 年首次提出。SURF 是尺度不变特征变换算法（SIFT 算法）的加速版。一般来说，标准的 SURF 算子比 SIFT 算子快好几倍，并且在多幅图片下具有更好的稳定性。SURF 最大的特征在于采用了 harr 特征以及积分图像的概念，这大大加快了程序的运行时间。SURF 可以应用于计算机视觉的物体识别以及 3D 重构中。

11.1.2 SURF 算法原理

1. 构建 Hessian 矩阵构造高斯金字塔尺度空间

其实 surf 构造的金字塔图像与 sift 有很大不同，正是因为这些不同才加快了其检测的速度。Sift 采用的是 DOG 图像，而 surf 采用的是 Hessian 矩阵行列式近似值图像。Hessian 矩阵是 Surf 算法的核心。在数学中，海森矩阵（Hessian matrix 或 Hessian）是一个自变量为向量的实值函数的二阶偏导数组成的方块矩阵。为了方便运算，假设函数 f(z，y)，Hessian 矩阵 H 是函数、偏导数组成。首先来看看图像中某个像素点的 Hessian 矩阵，如下：

$$H\left(f\left(x,y\right)\right)=\begin{bmatrix}\dfrac{\partial_2 f}{\partial x^2} & \dfrac{\partial_2 f}{\partial x\partial y}\\ \dfrac{\partial_2 f}{\partial x\partial y} & \dfrac{\partial_2 f}{\partial y^2}\end{bmatrix}$$

即每一个像素点都可以求出一个 Hessian 矩阵。

H 矩阵判别式为：

$$det(H)=\frac{\partial^2 f\partial^2 f}{\partial x^2\partial y^2}-\left(\frac{\partial^2 f}{\partial x\partial y}\right)^2$$

判别式的值是 H 矩阵的特征值，可以利用判定结果的符号将所有点分类，根据判别式取值正负，来判别该点是或不是极值点。

在 SURF 算法中，用图像像素 l(x,y)即为函数值 f(x,y)，选用二阶标准高斯函数作为滤波器，通过特定核间的卷积计算二阶偏导数，这样便能计算出 H 矩阵的三个矩阵元素 L_xx、L_xy、L_yy，从而计算出 H 矩阵：

$$H(x,\sigma)=\begin{bmatrix}L_{xx}(x,\sigma) & L_{xy}(x,\sigma)\\ L_{xy}(x,\sigma) & L_{yy}(x,\sigma)\end{bmatrix}$$

但是由于特征点需要具备尺度无关性，所以在进行 Hessian 矩阵构造前，需

要对其进行高斯滤波。

这样，经过滤波后再进行 Hessian 的计算，其公式如下：

$$L(x,t)=G(t)\cdot I(x,t)$$

L(x,t)是一幅图像在不同解析度下的表示，可以利用高斯核 G(t)与图像函数 I(x)在点 x 的卷积来实现，其中高斯核 G(t)的计算公式：

$$G(t)=\frac{\partial^2 g(t)}{\partial x^2}$$

g(x)为高斯函数，t 为高斯方差。通过这种方法可以为图像中每个像素计算出其 H 行列式的决定值，并用这个值来判别特征点。为方便应用，Herbert Bay 提出用近似值先代替 L(x,t)。为平衡准确值与近似值间的误差引入权值，权值随尺度变化，则 H 矩阵判别式可表示为：

$$det(Happrox)=DxxDyy-(0.9D_{xy})^2$$

其中 0.9 是论文作者给出的一个经验值，其实它是有一套理论计算的，具体请参考 surf 的英文论文。

由于求 Hessian 时要先高斯平滑，然后求二阶导数，这在离散的像素点是用模板卷积形成的，这 2 中操作合在一起用一个模板代替就可以了，比如说 y 方向上的模板如图 11.1 所示。

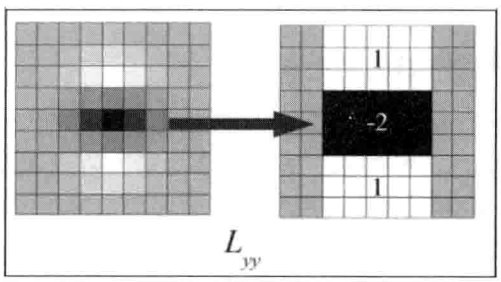


图 11.1 y 方向上的模板

图 11.1 的左边即用高斯平滑然后在 y 方向上求二阶导数的模板，为了加快运算用了近似处理，其处理结果如下图所示，这样就简化了很多。并且右图可以采用积分图来运算，从而大大加快了速度，关于积分图的介绍，可以去查阅相关的资料。

同理，x 和 y 方向的二阶混合偏导模板如图 11.2 所示。

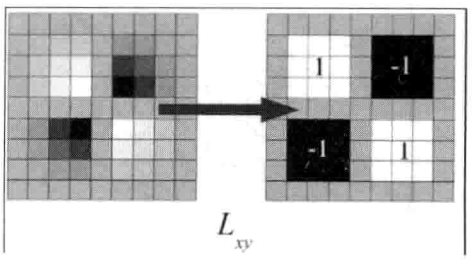


图 11.2 x 和 y 方向的二阶混合偏导模板

上面讲的这么多，只是得到了一张近似 hessian 的行列式图，这类似 sift 中的 DOG 图。但是在金字塔图像中分为很多层，每一层叫做一个 octave，每一个 octave 中又有几张尺度不同的图片。在 sift 算法中，同一个 octave 层中的图片尺寸（即大小）相同，但是尺度（即模糊程度）不同，而不同的 octave 层中的图片尺寸大小也不相同，因为它是由上一层图片降采样得到的。在进行高斯模糊时，sift 的高斯模板大小是始终不变的，只是在不同的 octave 之间改变图片的大小。而在 surf 中，图片的大小是一直不变的，不同 octave 层的待检测图片是改变高斯模糊尺寸大小得到的，当然了，同一个 octave 中不同图片用到的高斯模板尺度也不同。算法允许尺度空间多层图像同时被处理，不需对图像进行二次抽样，从而提高算法性能。图 11.3 是传统方式建立的一个金字塔结构，图像的尺寸是变化的，并且运算会反复使用高斯函数对子层进行平滑处理，说明 Surf 算法使原始图像保持不变而只改变了滤波器大小。Surf 采用这种方法节省了降采样过程，其处理速度自然也就提上去了。

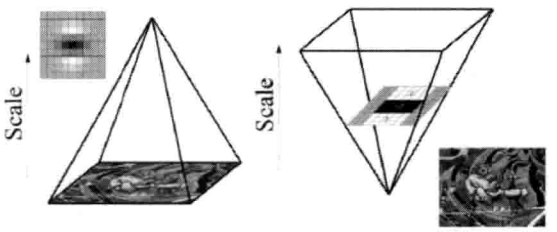


图 11.3 相关金字塔结构

2. 利用非极大值抑制初步确定特征点

此步骤和 sift 类似，将经过 hessian 矩阵处理过的每个像素点与其三维领域的 26 个点进行大小比较，如果它是这 26 个点中的最大值或者最小值，则保留下来，当做初步的特征点。检测过程中使用与该尺度层图像解析度相对应大小的滤波器进行检测。以 3×3 的滤波器为例，该尺度层图像中 9 个像素点之一的检测特征点与自身尺度层中其余 8 个点和在其之上及之下的两个尺度层的各 9 个点进行比较，共 26 个点，图 11.4 中标记的“x”的像素点的特征值若大于周围像素，则可确定该点为该区域的特征点。

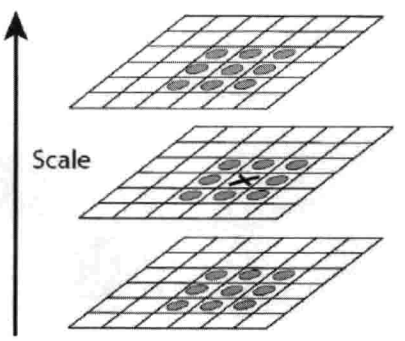


图 11.4 初步确定特征点

3. 精确定位极值点

这里也和 sift 算法中的类似，采用三维线性插值法得到亚像素级的特征点，同时也去掉那些值小于一定阈值的点，增加极值使检测到的特征点数量减少，最终只有几个特征最强点会被检测出来。

4. 选取特征点的主方向

这一步与 sift 也大有不同。Sift 选取特征点主方向是采用在特征点领域内统计其梯度直方图，取直方图 bin 值最大的以及超过最大 bin 值 80%的那些方向做为特征点的主方向。

而在 surf 中，不统计其梯度直方图，而是统计特征点领域内的 harr 小波特征。即在特征点的领域（比如说，半径为 6s 的圆内，s 为该点所在的尺度）内，统计 60 度扇形内所有点的水平 haar 小波特征和垂直 haar 小波特征总和，haar 小波的尺寸变长为 4s，这样一个扇形得到了一个值。然后 60 度扇形以一定间隔进行旋转，最后将最大值那个扇形的方向作为该特征点的主方向。该过程的示意图如图 11.5 所示。

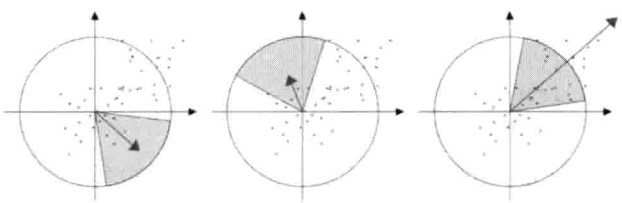


图 11.5 选取特征点的主方向

5. 构造 surf 特征点描述算子

在 sift 中，是在特征点周围取 16×16 的邻域，并把该领域化为 4×4 个的小区域，每个小区域统计 8 个方向梯度，最后得到 4×4×8=128 维的向量，该向量作为该点的 sift 描述子。

在 surf 中，也是在特征点周围取一个正方形框，框的边长为 20s（s 是所检测到该特征点所在的尺度）。该框带方向，方向当然就是第 4 步检测出来的主方向了。然后把该框分为 16 个子区域，每个子区域统计 25 个像素的水平方向和垂直方向的 haar 小波特征，这里的水平和垂直方向都是相对主方向而言的。该 haar 小波特征为水平方向值之和，水平方向绝对值之和，垂直方向之和，垂直方向绝对值之和。该过程的示意图如图 11.6 所示。

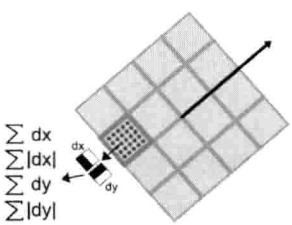


图 11.6 构造 surf 特征点描述算子

这样每个小区域就有 4 个值，所以每个特征点就是 16×4=64 维的向量，相比 sift 而言，少了一半，这在特征匹配过程中会大大加快匹配速度。

6. 总结

Surf 采用 Henssian 矩阵获取图像局部最值十分稳定，但是在求主方向阶段太过于依赖局部区域像素的梯度方向，有可能使找到的主方向不准确。后面的特征向量提取以及匹配都严重依赖于主方向，即使不大偏差角度也可以造成后面特征匹配的放大误差，从而使匹配不成功。另外图像金字塔的层取得不够紧密也会使得尺度有误差，后面的特征向量提取同样依赖相应的尺度，发明者在这个问题上的折中解决方法是取适量的层然后进行插值。

11.1.3 SURF 类相关 OpenCV 源码剖析

OpenCV 中关于 SURF 算法的部分，常常涉及到的是 SURF、SurfFeatureDetector、SurfDescriptorExtractor 这三个类，这一小节我们就来看看它们究竟是什么来头。

在……\opencv\sources\modules\nonfree\include\opencv2\nonfree 路径下的 features2d.hpp 头文件中，可以发现这样两句定义：

```
typedef SURF SurfFeatureDetector;
typedef SURF SurfDescriptorExtractor;
```

我们都知道，typedef 声明是为现有类型创建一个新的名字，类型别名。这就表示，SURF 类忽然同时有了两个新名字 SurfFeatureDetector 和 SurfDescriptorExtractor。

也就是说，我们平常使用的 SurfFeatureDetector 类和 SurfDescriptorExtractor 类，其实就是 SURF 类，它们三者等价。

然后在这两句定义的上方，可以看到 SURF 类的类声明全貌，在这里由于篇幅原因，只贴出其轮廓。

```
class CV_EXPORTS_W SURF : public Feature2D
{
//……
};
```

可以观察到，SURF 类公共继承自 Feature2D 类，我们再次进行转到定义，可以在路径…\opencv\build\include\opencv2\features2d\features2d.hpp 看到 Feature2D 类的声明。

```
class CV_EXPORTS_W Feature2D : public FeatureDetector, public
DescriptorExtractor
{
//……
};
```

显然，Feature2D 类又是公共继承自 FeatureDetector 以及 DescriptorExtractor 类。继续刨根问底，看看其父类 FeatureDetector 以及 DescriptorExtractor 类的定义。

首先是 FeatureDetector 类。

```
class CV_EXPORTS_W FeatureDetector : public virtual Algorithm
{
//……
};
```

这里，我们看到了以后经常会用到的 detect()方法重载的两个原型，原来是 SURF 类经过两层的继承，从 FeatureDetector 类继承而来的。

```
 CV_WRAP void detect(const Mat& image, CV_OUT vector<KeyPoint>&
keypoints, const Mat& mask=Mat())const;

 void detect(const vector<Mat>& images, vector<vector<KeyPoint> >&
keypoints, const vector<Mat>& masks=vector<Mat>())const;
```

同样，看看 SURF 类的另一个“爷爷”DescriptorExtractor 类的声明。

```
class CV_EXPORTS_W DescriptorExtractor : public virtual Algorithm
{
    //……
};
```

上述代码表明 FeatureDetector 类和 DescriptorExtractor 类都虚继承自 Algorithm 基类。

终于，我们找到 SURF 类“德高望重”的祖先——OpenCV 中的 Algorithm 基类。其原型声明如下。

```
class CV_EXPORTS_W Algorithm
{
    //……
};
```

关于这几个类看似错综复杂的关系，将其绘制出来便一目了然了，如图 11.7 所示。

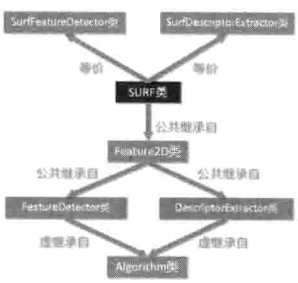


图 11.7 SURF 相关类关系图示

11.1.4 绘制关键点：drawKeypoints()函数

因为接下来的示例程序需要用到 drawKeypoints 函数，在这里顺便讲一讲。顾名思义，此函数用于绘制关键点。

```
C++: void drawKeypoints(const Mat&image, const vector<KeyPoint>&
keypoints, Mat& outImage, constScalar& color=Scalar::all(-1), int
flags=DrawMatchesFlags::DEFAULT )
```

- 第一个参数，const Mat&类型的 src，输入图像。
- 第二个参数，const vector<KeyPoint>&类型的 keypoints，根据源图像得到的特征点。它是一个输出参数。
- 第三个参数，Mat&类型的 outImage，输出图像，其内容取决于第五个参数标识符 falgs。
- 第四个参数，const Scalar&类型的 color，关键点的颜色，有默认值 Scalar::all(-1)。
- 第五个参数，int 类型的 flags，绘制关键点的特征标识符，有默认值 DrawMatchesFlags::DEFAULT。可以在下面这个结构体中选取值。

```
struct DrawMatchesFlags
{
    enum
    {
        DEFAULT = 0, // 创建输出图像矩阵(使用 Mat::create)。使用现存的输出图像绘制匹配对和特征点 。且对每一个关键点，只绘制中间点
        DRAW_OVER_OUTIMG = 1, //不创建输出图像矩阵，而是在输出图像上绘制匹配对
        NOT_DRAW_SINGLE_POINTS = 2, //单点特征点不被绘制
        DRAW_RICH_KEYPOINTS = 4 //对每一个关键点，绘制带大小和方向的关键点圆圈
    };
};
```

11.1.5 KeyPoint 类

KeyPoint 类是一个为特征点检测而生的数据结构，用于表示特征点。

```
class KeyPoint
{
Point2f  pt;  //坐标
    float  size; //特征点邻域直径
    float  angle; //特征点的方向，值为[零,三百六十)，负值表示不使用
    float  response;
    int  octave; //特征点所在的图像金字塔的组
    int  class_id; //用于聚类的 id
}
```

11.1.6 示例程序：SURF 特征点检测

这个示例程涉及到如下三个方面：

- 使用 FeatureDetector 接口来发现感兴趣点。
- 使用 SurfFeatureDetector 以及其函数 detect 来实现检测过程
- 使用函数 drawKeypoints 绘制检测到的关键点。

详细注释的源代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include "opencv2/core/core.hpp"
```

```
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include <iostream>
using namespace cv;


//-----------------------------【main()函数】----------------------------
//   描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main()
{
    //【0】改变 console 字体颜色
    system("color 2F");

    //【0】显示帮助文字
    ShowHelpText();

    //【1】载入源图片并显示
    Mat srcImage1 = imread("1.jpg", 1 );
    Mat srcImage2 = imread("2.jpg", 1 );
    if( !srcImage1.data || !srcImage2.data )//检测是否读取成功
    { printf("读取图片错误，请确定目录下是否有 imread 函数指定名称的图片存在~!
\n"); return false; }
    imshow("原始图 1",srcImage1);
    imshow("原始图 2",srcImage2);

    //【2】定义需要用到的变量和类
    int minHessian = 400;//定义 SURF 中的 hessian 阈值特征点检测算子
    SurfFeatureDetector detector( minHessian );//定义一个
SurfFeatureDetector（SURF） 特征检测类对象
    std::vector<KeyPoint> keypoints_1, keypoints_2;//vector 模板类是能够
存放任意类型的动态数组，能够增加和压缩数据

    //【3】调用 detect 函数检测出 SURF 特征关键点，保存在 vector 容器中
    detector.detect( srcImage1, keypoints_1 );
    detector.detect( srcImage2, keypoints_2 );

    //【4】绘制特征关键点
    Mat img_keypoints_1; Mat img_keypoints_2;
    drawKeypoints( srcImage1, keypoints_1, img_keypoints_1,
Scalar::all(-1), DrawMatchesFlags::DEFAULT );
    drawKeypoints( srcImage2, keypoints_2, img_keypoints_2,
Scalar::all(-1), DrawMatchesFlags::DEFAULT );

    //【5】显示效果图
    imshow("特征点检测效果图 1", img_keypoints_1 );
    imshow("特征点检测效果图 2", img_keypoints_2 );

    waitKey(0);
    return 0;
}
```

示例程序截图如图 11.8～11.13 所示。

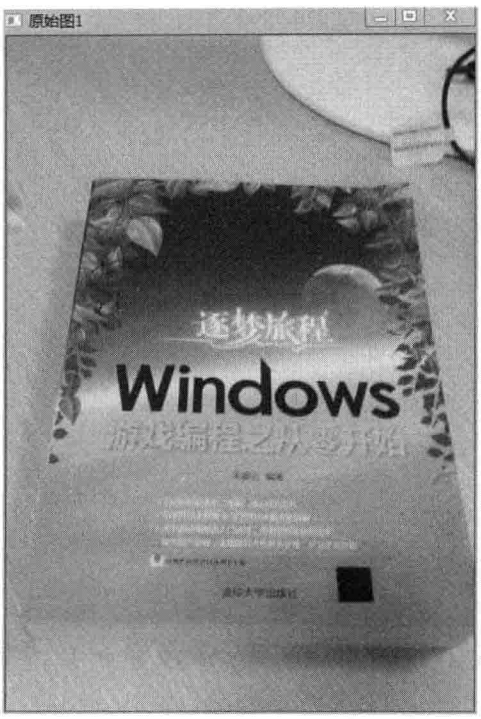


图 11.8 原始图（1）

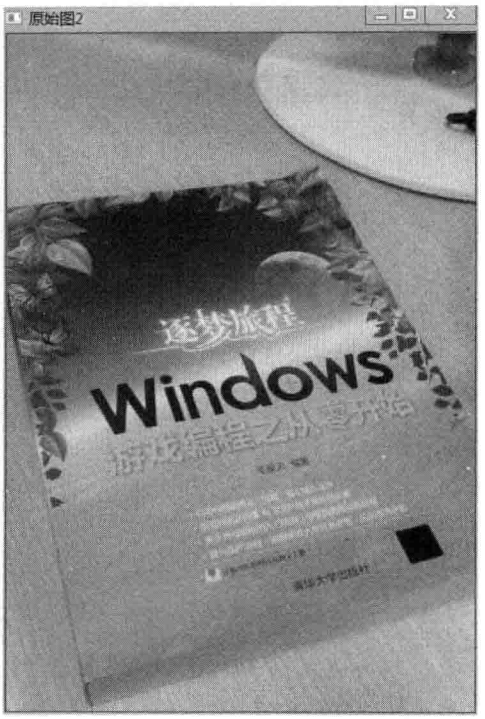


图 11.9 原始图（2）

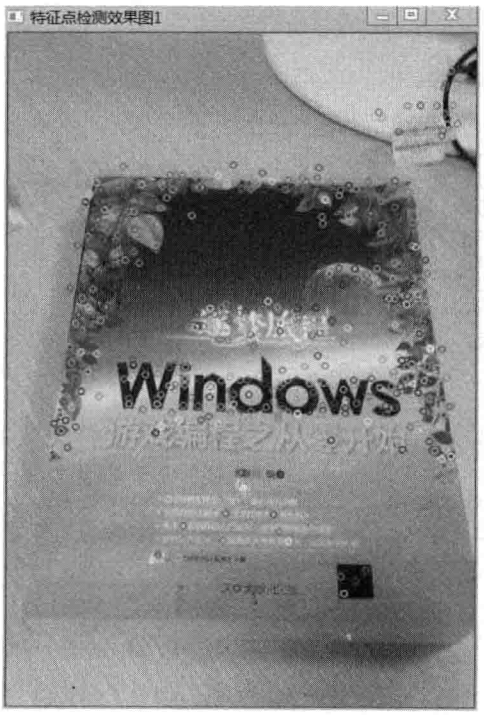


图 11.10 特征检测效果图 1
（DEFAULT 绘制模式）

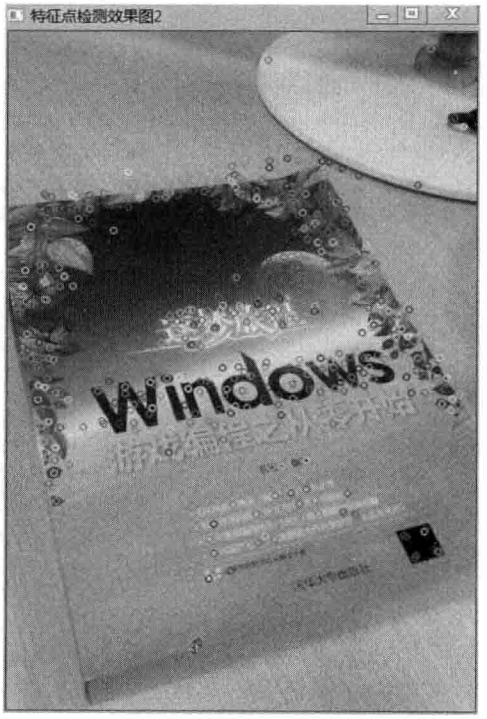


图 11.11 特征检测效果图 2
（DEFAULT 绘制模式）

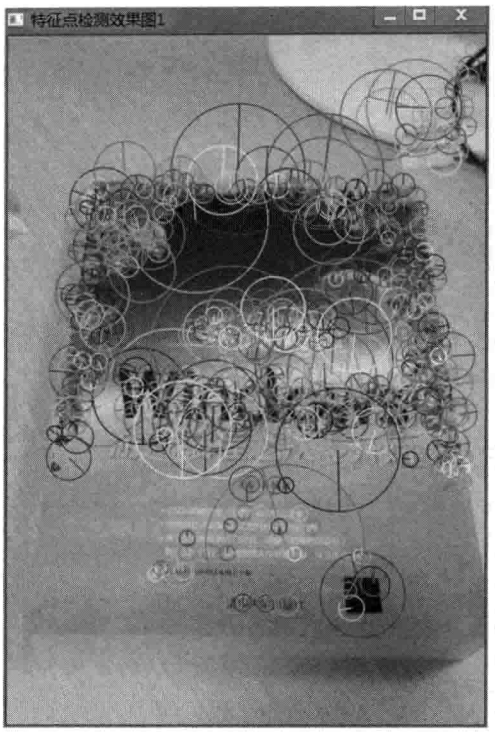


图 11.12 特征检测效果图 1
（DRAW_RICH_KEYPOINTS 绘制模式）

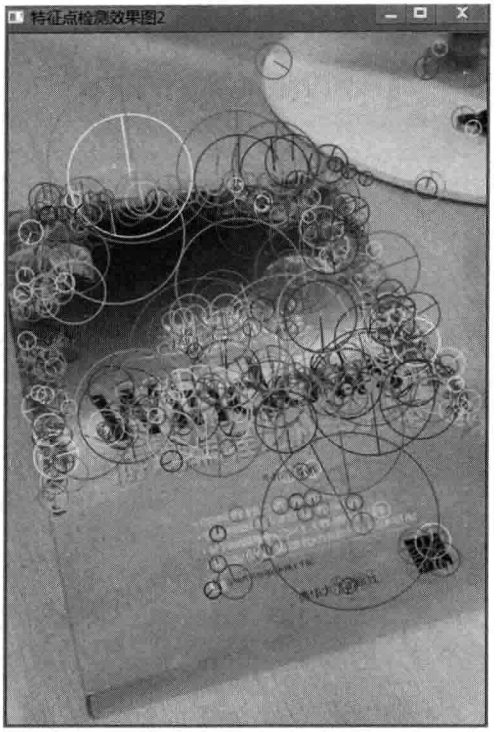


图 11.13 特征检测效果图 2
（DRAW_RICH_KEYPOINTS 绘制模式）

11.2 SURF 特征提取

通过上一节 SURF 相关内容的讲解，我们已经对 SURF 算法有了一定的理解。SURF 算法为每个检测到的特征定义了位置和尺度，尺度值可用于定义围绕特征点的窗口大小。不论物体的尺度在窗口是什么样的，都将包含相同的视觉信息，这些信息用于表示特征点以使得它们与众不同。

在特征匹配中，特征描述子通常用于 N 维向量，在光照不变以及少许透视变形的情况下很理想。另外，优质的描述子可以通过简单的距离测量进行比较，比如欧氏距离。因此，它们在特征匹配算法中，用处是很大的。

在 OpenCV 中，使用 SURF 进行特征点描述主要是 drawMatches 方法和 BruteForceMatcher 类的运用，下面让我们一起来认识它们。

11.2.1 绘制匹配点：drawMatches()函数

drawMatches 用于绘制出相匹配的两个图像的关键点，它有以下两个版本的 C++函数原型。

```
C++: void drawMatches(const Mat& img1,
constvector<KeyPoint>& keypoints1,
 const Mat& img2,
 constvector<KeyPoint>& keypoints2,
```

```
 constvector<DMatch>& matches1to2,
 Mat& outImg,
 const Scalar&matchColor=Scalar::all(-1),
 const Scalar&singlePointColor=Scalar::all(-1),
 const vector<char>&matchesMask=vector<char>(),
 intflags=DrawMatchesFlags::DEFAULT )

C++: void drawMatches(const Mat& img1,
 constvector<KeyPoint>& keypoints1,
 const Mat& img2,
 constvector<KeyPoint>& keypoints2,
 const vector<vector<DMatch>>&matches1to2,
 Mat& outImg,
 const Scalar&matchColor=Scalar::all(-1),
 const Scalar&singlePointColor=Scalar::all(-1),
 constvector<vector<char>>& matchesMask=vector<vector<char>>(),
 intflags=DrawMatchesFlags::DEFAULT )
```

除了第五个参数 matches1to2 和第九个参数 matchesMask 有细微的差别以外，两个版本的基本上相同。下面统一讲解。

- 第一个参数，const Mat&类型的 img1，第一幅源图像。
- 第二个参数，const vector<KeyPoint>&类型的 keypoints1，根据第一幅源图像得到的特征点，它是一个输出参数。
- 第三个参数，const Mat&类型的 img2，第二幅源图像。
- 第四个参数，const vector<KeyPoint>&类型的 keypoints2，根据第二幅源图像得到的特征点，它是一个输出参数。
- 第五个参数，matches1to2，第一幅图像到第二幅图像的匹配点，即表示每一个图 1 中的特征点都在图 2 中有一一对应的点、
- 第六个参数，Mat&类型的 outImg，输出图像，其内容取决于第五个参数标识符 falgs。
- 第七个参数，const Scalar&类型的 matchColor，匹配的输出颜色，即线和关键点的颜色。它有默认值 Scalar::all(-1)，表示颜色是随机生成的。
- 第八个参数，const Scalar&类型的 singlePointColor，单一特征点的颜色，它也表示随机生成颜色的默认值 Scalar::all(-1)。
- 第九个参数，matchesMask，确定哪些匹配是会绘制出来的掩膜，如果掩膜为空，表示所有匹配都进行绘制。
- 第十个参数，int 类型的 flags，特征绘制的标识符，有默认值 DrawMatchesFlags::DEFAULT。可以在如下这个 DrawMatchesFlags 结构体中选取值：

```
struct DrawMatchesFlags
{
    enum
    {
        DEFAULT = 0, // 创建输出图像矩阵(使用 Mat::create)。使用现存的输出图像
```

```
绘制匹配对和特征点。且对每一个关键点，只绘制中间点
        DRAW_OVER_OUTIMG = 1, //不创建输出图像矩阵，而是在输出图像上绘制匹配对
        NOT_DRAW_SINGLE_POINTS = 2, //单点特征点不被绘制
        DRAW_RICH_KEYPOINTS = 4 //对每一个关键点，绘制带大小和方向的关键点圆圈
    };
};
```

11.2.2 BruteForceMatcher 类源码分析

接下来，我们看看本节中会用到的 BruteForceMatcher 类的源码分析，BruteForceMatcher 类用于进行暴力匹配相关的操作。在……\opencv\sources\modules\legacy\include\opencv2\ legacy\legacy.hpp 路径下，可以找到 BruteForceMatcher 类的定义。

```
template<class Distance>
class CV_EXPORTS BruteForceMatcher : publicBFMatcher
{
public:
  BruteForceMatcher( Distance d = Distance() ) :
BFMatcher(Distance::normType,false) {(void)d;}
  virtual ~BruteForceMatcher() {}
};
```

其公共继承自 BFMatcher 类。而 BFMatcher 类位于…\opencv\sources\modules\features2d\include\opencv2\features2d\features2d.hpp

路径之下，我们一起看看其代码。

```
class CV_EXPORTS_W BFMatcher : publicDescriptorMatcher
{
//……
};
```

发现公共其继承自 DescriptorMatcher 类，再次进行溯源，在 D:\Program Files(x86)\opencv\sources\modules\features2d\include\opencv2\features2d\features2d.hpp 路径下找到 DescriptorMatcher 类的定义。

```
class CV_EXPORTS_W DescriptorMatcher :public Algorithm
{
//……
};
```

可以发现，DescriptorMatcher 类和之前我们讲到 FeatureDetector 类和 DescriptorExtractor 类一样，都是继承自它们“德高望重的祖先”Algorithm 基类的。

而用 BruteForceMatcher 类时用到最多的 match 方法，是它从 DescriptorMatcher 类那里继承而来。定义如下。

```
//为各种描述符找到一个最佳的匹配（若掩膜为空）
CV_WRAP void match( const Mat& queryDescriptors, const
Mat&trainDescriptors,
          CV_OUTvector<DMatch>& matches, const Mat& mask=Mat() ) const;
```

11.2.3 示例程序：SURF 特征提取

这个示例程序中，我们利用 SurfDescriptorExtractor 类进行特征向量的相关计算。程序利用了 SURF 特征的特征描述办法，其操作封装在类 SurfFeatureDetector 中，利用类内的 detect 函数可以检测出 SURF 特征的关键点，保存在 vector 容器中。第二步利用 SurfDescriptorExtractor 类进行特征向量的相关计算。将之前的 vector 变量变成向量矩阵形式保存在 Mat 中。最后强行匹配两幅图像的特征向量，利用了类 BruteForceMatcher 中的函数 match。

程序的核心思想如下。

- 使用 DescriptorExtractor 接口来寻找关键点对应的特征向量。
- 使用 SurfDescriptorExtractor 以及它的函数 compute 来完成特定的计算。
- 使用 BruteForceMatcher 来匹配特征向量。
- 使用函数 drawMatches 来绘制检测到的匹配点。

关键点讲解：OpenCV2 引入了一个通用类，用于提取不同的特征点描述子，计算如下。

```
//【4】计算描述子（特征向量）
    SurfDescriptorExtractorextractor;
    Matdescriptors1, descriptors2;
    extractor.compute(srcImage1, keyPoint1, descriptors1 );
    extractor.compute(srcImage2, keyPoints2, descriptors2 );
```

这里的结果为一个 Mat 矩阵，它的行数与特征点向量中元素个数是一致的。每行都是一个 N 维描述子的向量，比如 SURF 算法默认的描述子维度为 64，该向量描绘了特征点周围的强度样式。两个特征点越相似，它们的特征向量也越靠近。这些描述子在图像匹配中尤其有用。例如，我们想匹配同一个场景中的两幅图像。首先，检测每幅图像中的特征，然后提取它们的描述子。第一幅图像中的每一个特征描述子向量都会与第二幅图中的描述子进行比较，得分最高的一对描述子（也就是两个向量的距离最近）将被视为那个特征的最佳匹配。该过程对于第一幅图像中的所有特征进行重复，这便是 BruteForceMatcher 中实行的最基本的策略。相关代码如下。

```
//【5】使用 BruteForce 进行匹配
//实例化一个匹配器
BruteForceMatcher<L2<float> > matcher;
std::vector<DMatch > matches;
//匹配两幅图中的描述子（descriptors）
matcher.match(descriptors1, descriptors2, matches );
```

BruteForceMatcher 是由 DescriptorMatcher 派生出来的一个类，而 DescriptorMatcher 定义了不同的匹配策略的共同接口。调用 match 方法后，在其第三个参数输出一个 cv::DMatch 向量。于是我们定义一个 std::vector<DMatch>类型的 matches。

调用 match 方法之后，便可以使用 drawMatches 方法对匹配到的点进行绘制，

并最终显示出来。相关代码如下。

```
//【6】绘制从两个图像中匹配出的关键点
MatimgMatches;
drawMatches(srcImage1, keyPoint1, srcImage2, keyPoints2, matches,
imgMatches );//进行绘制
//【7】显示效果图
imshow("匹配图", imgMatches );
```

核心部分讲解完毕，下面我们一起来看看详细注释的示例程序的完全体。

```
//-----------------------【头文件、命名空间包含部分】-----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/nonfree/nonfree.hpp>
#include<opencv2/legacy/legacy.hpp>
#include <iostream>
using namespace cv;
using namespace std;

//---------------------------【main()函数】---------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】载入素材图
    Mat srcImage1 = imread("1.jpg",1);
    Mat srcImage2 = imread("2.jpg",1);
    if( !srcImage1.data || !srcImage2.data )
    { printf("读取图片错误,请确定目录下是否有imread函数指定的图片存在~! \n");
return false; }

    //【2】使用SURF算子检测关键点
    int minHessian = 700;//SURF算法中的hessian阈值
    SurfFeatureDetector detector( minHessian );//定义一个
SurfFeatureDetector（SURF） 特征检测类对象
    std::vector<KeyPoint> keyPoint1, keyPoints2;//vector模板类，存放任意
类型的动态数组

    //【3】调用detect函数检测出SURF特征关键点，保存在vector容器中
    detector.detect( srcImage1, keyPoint1 );
    detector.detect( srcImage2, keyPoints2 );

    //【4】计算描述符（特征向量）
    SurfDescriptorExtractor extractor;
    Mat descriptors1, descriptors2;
    extractor.compute( srcImage1, keyPoint1, descriptors1 );
    extractor.compute( srcImage2, keyPoints2, descriptors2 );

    //【5】使用BruteForce进行匹配
```

```
    // 实例化一个匹配器
    BruteForceMatcher< L2<float> > matcher;
    std::vector< DMatch > matches;
    //匹配两幅图中的描述子（descriptors）
    matcher.match( descriptors1, descriptors2, matches );

    //【6】绘制从两个图像中匹配出的关键点
    Mat imgMatches;
    drawMatches(srcImage1, keyPoint1, srcImage2, keyPoints2, matches,
imgMatches);//进行绘制

    //【7】显示效果图
    imshow("匹配图", imgMatches );

    waitKey(0);
    return 0;
}
```

让我们看看运行效果图（图 11.14）。

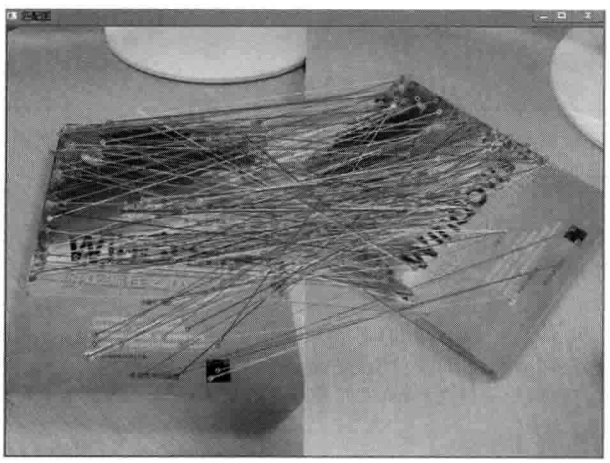


图 11.14 SURF 特征描述效果图

11.3 使用 FLANN 进行特征点匹配

本节，我们将学习如何使用 FlannBasedMatcher 接口以及函数 FLANN()，实现快速高效匹配（快速最近邻逼近搜索函数库，Fast Library for Approximate Nearest Neighbors，FLANN）。

11.3.1 FlannBasedMatcher 类的简单分析

在 OpenCV 源代码中，找到 FlannBasedMatcher 类的脉络如下。

```
class CV_EXPORTS_W FlannBasedMatcher : public DescriptorMatcher
{
//.....
};
```

可以发现 FlannBasedMatcher 类也是继承自 DescriptorMatcher，并且同样主要使用来自 DescriptorMatcher 类的 match 方法进行匹配。下面，让我们讲解一下此类方法的用法。

11.3.2 找到最佳匹配：DescriptorMatcher::match 方法

DescriptorMatcher::match()函数从每个描述符查询集中找到最佳匹配，有两个版本的源码，下面用注释对其进行讲解。

```
C++: void DescriptorMatcher::match(
const Mat& queryDescriptors,//查询描述符集
 const Mat& trainDescriptors,//训练描述符集
 vector<DMatch>& matches,//得到的匹配。若查询描述符有在掩膜中被标记出来，则没有匹配添加到描述符中。则匹配量可能会比查询描述符数量少
 const Mat& mask=Mat() ) //指定输入查询和训练描述符允许匹配的掩膜

C++: void DescriptorMatcher::match(
const Mat& queryDescriptors, //查询描述符集
 vector<DMatch>& matches, //得到的匹配。若查询描述符有在掩膜中被标记出来，则没有匹配添加到描述符中，则匹配量可能会比查询描述符数量少
 const vector<Mat>& masks=vector<Mat>() )//一组掩膜，每个 masks[i]从第 i 个图像 trainDescCollection[i]指定输入查询和训练描述符允许匹配的掩膜
```

11.3.3 示例程序：使用 FLANN 进行特征点匹配

本节我们将演示如何使用 FLANN 进行特征点匹配，详细注释的示例程序代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <opencv2/nonfree/nonfree.hpp>
#include<opencv2/legacy/legacy.hpp>
#include <iostream>
using namespace cv;
using namespace std;


//---------------------------【main()函数】---------------------------
//   描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------
int main( int argc, char** argv )
{
    //【1】载入源图片
    Mat img_1 = imread("1.jpg", 1 );
    Mat img_2 = imread( "2.jpg", 1 );
    if( !img_1.data || !img_2.data ) { printf("读取图片 image0 错误~！ \n");
return false; }
```

```
    //【2】利用 SURF 检测器检测的关键点
    int minHessian = 300;
    SURF detector( minHessian );
    std::vector<KeyPoint> keypoints_1, keypoints_2;
    detector.detect( img_1, keypoints_1 );
    detector.detect( img_2, keypoints_2 );

    //【3】计算描述符(特征向量)
    SURF extractor;
    Mat descriptors_1, descriptors_2;
    extractor.compute( img_1, keypoints_1, descriptors_1 );
    extractor.compute( img_2, keypoints_2, descriptors_2 );

    //【4】采用 FLANN 算法匹配描述符向量
    FlannBasedMatcher matcher;
    std::vector< DMatch > matches;
    matcher.match( descriptors_1, descriptors_2, matches );
    double max_dist = 0; double min_dist = 100;

    //【5】快速计算关键点之间的最大和最小距离
    for( int i = 0; i < descriptors_1.rows; i++ )
    {
        double dist = matches[i].distance;
        if( dist < min_dist ) min_dist = dist;
        if( dist > max_dist ) max_dist = dist;
    }
    //输出距离信息
    printf("> 最大距离(Max dist) : %f \n", max_dist );
    printf("> 最小距离(Min dist) : %f \n", min_dist );

    //【6】存下符合条件的匹配结果(即其距离小于 2* min_dist 的),使用 radiusMatch
同样可行
    std::vector< DMatch > good_matches;
    for( int i = 0; i < descriptors_1.rows; i++ )
    {
        if( matches[i].distance < 2*min_dist )
        { good_matches.push_back( matches[i]); }
    }

    //【7】绘制出符合条件的匹配点
    Mat img_matches;
    drawMatches( img_1, keypoints_1, img_2, keypoints_2,
        good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
        vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

    //【8】输出相关匹配点信息
    for( int i = 0; i < good_matches.size(); i++ )
    { printf( ">符合条件的匹配点 [%d] 特征点 1: %d -- 特征点 2: %d \n", i,
good_matches[i].queryIdx, good_matches[i].trainIdx ); }

    //【9】显示效果图
```

```
    imshow( "匹配效果图", img_matches );

    //按任意键退出程序
    waitKey(0);
    return 0;
}
```

让我们一起看看程序的运行截图。如图 11.15、11.16。

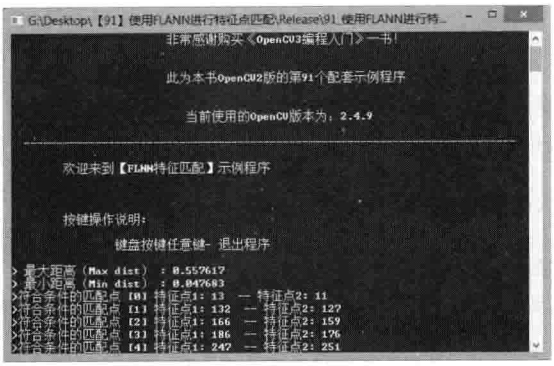


图 11.15 匹配信息窗口

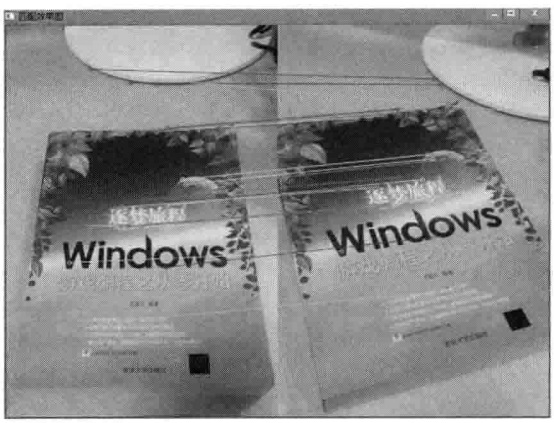


图 11.16 匹配效果图

11.3.4 综合示例程序：FLANN 结合 SURF 进行关键点的描述和匹配

通过上文的学习，我们已经了解了 FLANN 算法和 SURF 算法。

而这个综合示例程序中，将两者结合起来使用——用 SURF 进行关键点和描述符的提取，用 FLANN 进行匹配。

知识点和讲解点都蕴含在程序代码中，不妨整体贴出经过详细注释的程序代码，方便大家把握程序脉络。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所使用的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
```

```
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
using namespace cv;
using namespace std;


//-----------------------------【main()函数】--------------------------
//        描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//---------------------------------------------------------------------
int main()
{
    //【0】改变 console 字体颜色
    system("color 6F");

    //【1】载入图像、显示并转化为灰度图
    Mat trainImage = imread("1.jpg"), trainImage_gray;
    imshow("原始图",trainImage);
    cvtColor(trainImage, trainImage_gray, CV_BGR2GRAY);

    //【2】检测 Surf 关键点、提取训练图像描述符
    vector<KeyPoint> train_keyPoint;
    Mat trainDescriptor;
    SurfFeatureDetector featureDetector(80);
    featureDetector.detect(trainImage_gray, train_keyPoint);
    SurfDescriptorExtractor featureExtractor;
    featureExtractor.compute(trainImage_gray, train_keyPoint,
trainDescriptor);

    //【3】创建基于 FLANN 的描述符匹配对象
    FlannBasedMatcher matcher;
    vector<Mat> train_desc_collection(1, trainDescriptor);
    matcher.add(train_desc_collection);
    matcher.train();

    //【4】创建视频对象、定义帧率
    VideoCapture cap(0);
    unsigned int frameCount = 0;//帧数

    //【5】不断循环，直到 q 键被按下
    while(char(waitKey(1)) != 'q')
    {
        //<1>参数设置
        int64 time0 = getTickCount();
        Mat testImage, testImage_gray;
        cap >> testImage;//采集视频到 testImage 中
        if(testImage.empty())
         continue;

        //<2>转化图像到灰度
        cvtColor(testImage, testImage_gray, CV_BGR2GRAY);

        //<3>检测 S 关键点、提取测试图像描述符
```

```
        vector<KeyPoint> test_keyPoint;
        Mat testDescriptor;
        featureDetector.detect(testImage_gray, test_keyPoint);
        featureExtractor.compute(testImage_gray, test_keyPoint,
testDescriptor);

        //<4>匹配训练和测试描述符
        vector<vector<DMatch> > matches;
        matcher.knnMatch(testDescriptor, matches, 2);

        // <5>根据劳氏算法(Lowe's algorithm),得到优秀的匹配点
        vector<DMatch> goodMatches;
        for(unsigned int i = 0; i < matches.size(); i++)
        {
         if(matches[i][0].distance < 0.6 * matches[i][1].distance)
             goodMatches.push_back(matches[i][0]);
        }

        //<6>绘制匹配点并显示窗口
        Mat dstImage;
        drawMatches(testImage, test_keyPoint, trainImage,
train_keyPoint, goodMatches, dstImage);
        imshow("匹配窗口", dstImage);

        //<7>输出帧率信息
        cout << "当前帧率为: " << getTickFrequency() / (getTickCount() -
time0) << endl;
    }

    return 0;
}
```

程序代码就是如上这些，接着，我们一起看看程序运行截图。如图 11.17、11.18 所示。

图 11.17 匹配原始图

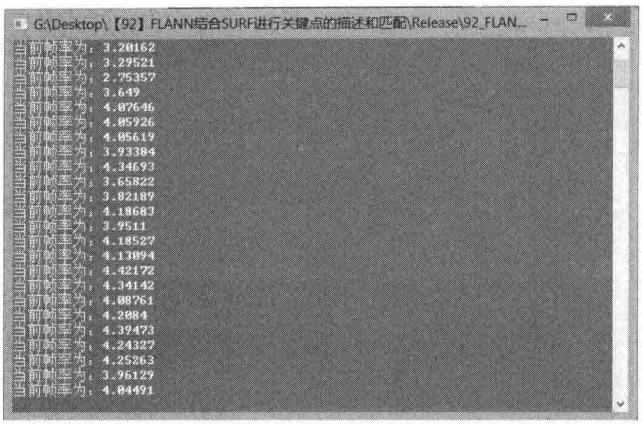


图 11.18 输出帧率的窗口

可以发现，用此方法进行匹配时帧率偏低，算法效率有待加强。

当找一本和原图不相关的书籍对准摄像头，得到的匹配点寥寥无几。如图 11.19 所示。

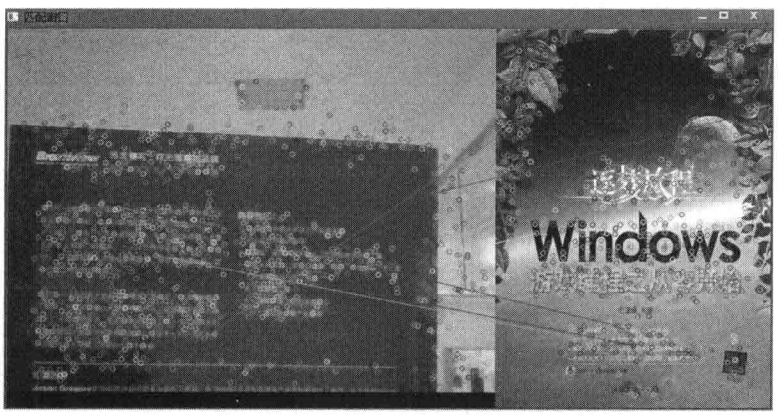


图 11.19 匹配点寥寥无几

而当找到和原始图对应的书本封面对准摄像头时，得到的匹配点多且整齐。如图 11.20 所示。

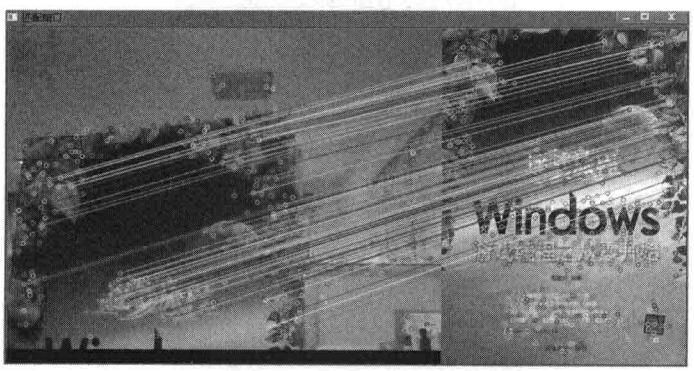


图 11.20 整齐的匹配点

11.3.5 综合示例程序：SIFT 配合暴力匹配进行关键点描述和提取

在上文刚刚讲过 SURF 算法相关的内容，而 SURF 算法作为 SIFT 算法的加速版，理应当有更快的检测速度。表 11.1 是两者一个简单的比较。

表 11.1 SURF 算法-SIFT 算法各项指标对比

比较内容	SIFT	SURF
尺度空间	DOG 与不同尺度的图片卷积	不同尺度的 box filters 与原图片卷积
特征点检测	先进行非极大抑制，再去除低对比度的点。再通过 Hessian 矩阵去除边缘的点	先利用 Hessian 矩阵确定候选点，然后进行非极大抑制
方向	在正方形区域内统计梯度的幅值的直方图，找 max 对应的方向。可以有多个方向。	在圆形区域内，计算各个扇形范围内 x、y 方向的 haar 小波响应，找模最大的扇形方向
特征描述子	16*16 的采样点划分为 4*4 的区域，计算每个区域的采样点的梯度方向和幅值，统计成 8bin 直方图，一共 4*4*8=128 维	20*20s 的区域划分为 4*4 的子区域，每个子区域找 5*5 个采样点，计算采样点的 haar 小波响应，记录∑dx, ∑dy, ∑|dx|,∑|dy|，一共 4*4*4=64 维

且理论上，SURF 是 SIFT 速度的 3 倍。

本节我们来看一个用 SIFT 算法配合暴力匹配的示例程序，看看是否 SURF 有比 SIFT 更快的匹配速度（由于匹配方式不同，这里只是大概的比较）。

可以发现，其代码形式和上一节中的代码基本类似。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
using namespace cv;
using namespace std;



//---------------------------【main()函数】---------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【0】改变 console 字体颜色
    system("color 5F");

    //【1】载入图像、显示并转化为灰度图
    Mat trainImage = imread("1.jpg"), trainImage_gray;
```

```
    imshow("原始图",trainImage);
    cvtColor(trainImage, trainImage_gray, CV_BGR2GRAY);

    //【2】检测 SIFT 关键点、提取训练图像描述符
    vector<KeyPoint> train_keyPoint;
    Mat trainDescription;
    SiftFeatureDetector featureDetector;
    featureDetector.detect(trainImage_gray, train_keyPoint);
    SiftDescriptorExtractor featureExtractor;
    featureExtractor.compute(trainImage_gray, train_keyPoint,
trainDescription);

    //  【3】进行基于描述符的暴力匹配
    BFMatcher matcher;
    vector<Mat> train_desc_collection(1, trainDescription);
    matcher.add(train_desc_collection);
    matcher.train();

    //【4】创建视频对象、定义帧率
    VideoCapture cap(0);
    unsigned int frameCount = 0;//帧数

    //【5】不断循环，直到 q 键被按下
    while(char(waitKey(1)) != 'q')
    {
        //<1>参数设置
        double time0 = getTickCount();
        Mat captureImage, captureImage_gray;
        cap >> captureImage;//采集视频到 testImage 中
        if(captureImage.empty())
         continue;

        //<2>转化图像到灰度
        cvtColor(captureImage, captureImage_gray, CV_BGR2GRAY);

        //<3>检测 SURF 关键点、提取测试图像描述符
        vector<KeyPoint> test_keyPoint;
        Mat testDescriptor;
        featureDetector.detect(captureImage_gray, test_keyPoint);
        featureExtractor.compute(captureImage_gray, test_keyPoint,
testDescriptor);

        //<4>匹配训练和测试描述符
        vector<vector<DMatch> > matches;
        matcher.knnMatch(testDescriptor, matches, 2);

        // <5>根据劳氏算法（Lowe's algorithm），得到优秀的匹配点
        vector<DMatch> goodMatches;
        for(unsigned int i = 0; i < matches.size(); i++)
        {
         if(matches[i][0].distance < 0.6 * matches[i][1].distance)
             goodMatches.push_back(matches[i][0]);
```

```
        }

        //<6>绘制匹配点并显示窗口
        Mat dstImage;
        drawMatches(captureImage, test_keyPoint, trainImage,
train_keyPoint, goodMatches, dstImage);
        imshow("匹配窗口", dstImage);

        //<7>输出帧率信息
        cout << "当前帧率为: " << getTickFrequency() / (getTickCount() -
time0) << endl;
    }

    return 0;
}
```

让我们一起看看运行截图。如图 11.21、11.22 所示。

图 11.21 原始图窗口

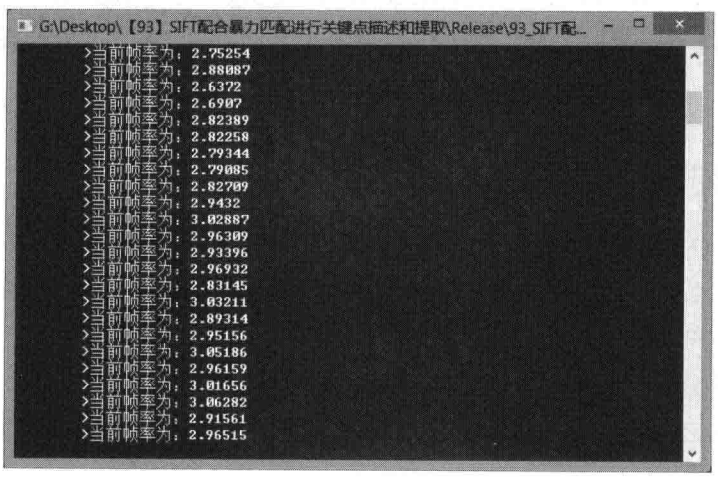


图 11.22 输出帧率的窗口

可以发现，在代码写法基本相同的情况下，用此方法进行匹配时帧率低于 SURF 算法的平均约 4.0 的帧率，算法效率有待加强。

当找一本和原图不相关的书籍对准摄像头，得到的匹配点寥寥无几。如图 11.23 所示。

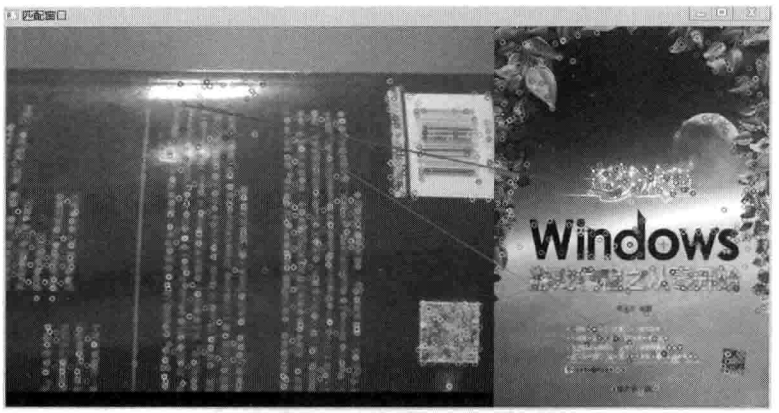


图 11.23 匹配点寥寥无几

而当找到和原始图对应的书对准摄像头时，得到的匹配点多且整齐。如图 11.24 所示。

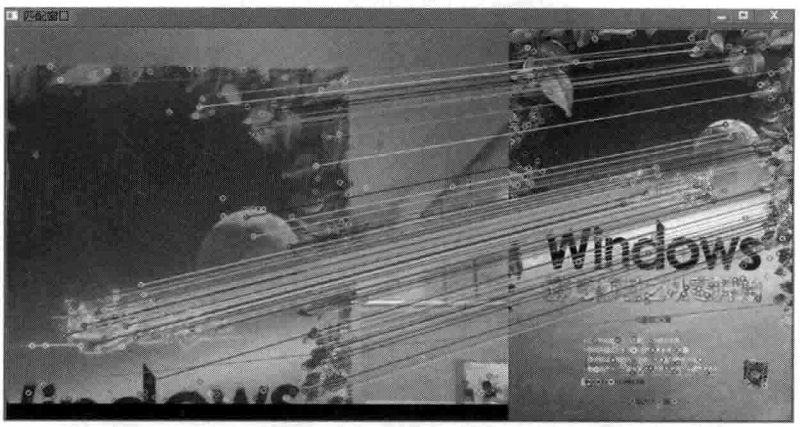


图 11.24 整齐的匹配点（暴力匹配）

11.4 寻找已知物体

在 FLANN 特征匹配的基础上，还可以进一步利用 Homography 映射找出已知物体。具体来说就是利用 findHomography 函数通过匹配的关键点找出相应的变换，再利用 perspectiveTransform 函数映射点群。

也就是分为以下两个大的步骤。

（1）使用函数 findHomography 寻找匹配上的关键点的变换。

（2）使用函数 perspectiveTransform 来映射点.

下面，我们分别对相关该函数进行简单介绍。

11.4.1 寻找透视变换：findHomography()函数

此函数的作用是找到并返回源图像和目标图像之间的透视变换 H：

$$s_i x_i' y_i' 1 \sim H x_i y_i 1$$

```
C++: Mat findHomography(
InputArray srcPoints,
InputArray dstPoints,
 int method=0,
 double ransacReprojThreshold=3,
 OutputArray mask=noArray() )
```

- 第一个参数，InputArray 类型的 srcPoints，源平面上的对应点，可以是 CV_32FC2 的矩阵类型或者 vector<Point2f>。
- 第二个参数，InputArray 类型的 dstPoints，目标平面上的对应点，可以是 CV_32FC2 的矩阵类型或者 vector<Point2f>。
- 第三个参数，int 类型的 method，用于计算单应矩阵的方法。可选标识符为如表 11.2 所示。

表 11.2 计算单应矩阵可选标识符

标识符	含义
0	使用所有点的常规方法（默认值）
CV_RANSAC	基于 RANSAC 的鲁棒性方法（robustmethod
CV_LMEDS	最小中值鲁棒性方法

- 第四个参数，double 类型的 ransacReprojThreshold，有默认值 3，处理点对为内围层时，允许重投影误差的最大值。这就是说，当 $\| \| \text{dstPoints}_i\text{-convertPoints Homogeneous } (H*\text{srcPoints}_i) \| > \text{ransacReprojThreshold}$ 时，这里的点 i 被看做内围层。若 srcPoints 和 dstPoints 是以像素为单位的，那么此参数的取值范围一般在 1 到 10 之间。
- 第五个参数，OutputArray 类型的 mask，有默认值 noArray()，是一个可选的参数，通过上面讲到的鲁棒性方法(CV_RANSAC 或者 CV_LMEDS)设置输出掩码。需要注意的是，输入掩码值会被忽略掉。

11.4.2 进行透视矩阵变换：perspectiveTransform()函数

perspectiveTransform 函数的作用是进行向量透视矩阵变换。

```
C++: void perspectiveTransform(InputArray src, OutputArray dst,
InputArray m)
```

- 第一个参数，InputArray 类型的 src，输入图像，即源图像，填 Mat 类的对象即可，且需为双通道或三通道浮点型图像，其中的每个元素是二维或三

维可被转换的矢量。

- 第二个参数，OutputArray 类型的 dst，函数调用后的运算结果存在这里，即这个参数用于存放函数调用后的输出结果，需和源图片有一样的尺寸和类型。
- 第三个参数，InputArray 类型的 m，变换矩阵，为 3x3 或者 4x4 浮点型矩阵。

11.4.3 示例程序：寻找已知物体

本节我们将放出用 OpenCV 寻找已知物体的详细注释的程序源代码。

```
//----------------------【头文件、命名空间包含部分】----------------------
//      描述：包含程序所依赖的头文件和命名空间
//-------------------------------------------------------------------------
#include "opencv2/core/core.hpp"
#include "opencv2/features2d/features2d.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/calib3d/calib3d.hpp"
#include "opencv2/nonfree/nonfree.hpp"
#include <iostream>
using namespace cv;
using namespace std;


//----------------------------【main()函数】----------------------------
//      描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-------------------------------------------------------------------------
int main()
{
    //【1】载入原始图片
    Mat srcImage1 = imread( "1.jpg", 1 );
    Mat srcImage2 = imread( "2.jpg", 1 );
    if( !srcImage1.data || !srcImage2.data )
        { printf("读取图片错误，请确定目录下是否有 imread 函数指定的图片存在~!
\n"); return false; }

    //【2】使用 SURF 算子检测关键点
    int minHessian = 400;//SURF 算法中的 hessian 阈值
    SurfFeatureDetector detector( minHessian );//定义一个
SurfFeatureDetector（SURF） 特征检测类对象
    vector<KeyPoint> keypoints_object, keypoints_scene;//vector 模板类，
存放任意类型的动态数组

     //【3】调用 detect 函数检测出 SURF 特征关键点，保存在 vector 容器中
    detector.detect( srcImage1, keypoints_object );
    detector.detect( srcImage2, keypoints_scene );

    //【4】计算描述符（特征向量）
    SurfDescriptorExtractor extractor;
    Mat descriptors_object, descriptors_scene;
```

```
    extractor.compute( srcImage1, keypoints_object,
descriptors_object );
    extractor.compute( srcImage2, keypoints_scene, descriptors_scene );

    //【5】使用 FLANN 匹配算子进行匹配
    FlannBasedMatcher matcher;
    vector< DMatch > matches;
    matcher.match( descriptors_object, descriptors_scene, matches );
    double max_dist = 0; double min_dist = 100;//最小距离和最大距离

    //【6】计算出关键点之间距离的最大值和最小值
    for( int i = 0; i < descriptors_object.rows; i++ )
    {
        double dist = matches[i].distance;
        if( dist < min_dist ) min_dist = dist;
        if( dist > max_dist ) max_dist = dist;
    }

    printf(">Max dist 最大距离 : %f \n", max_dist );
    printf(">Min dist 最小距离 : %f \n", min_dist );

    //【7】存下匹配距离小于 3*min_dist 的点对
    std::vector< DMatch > good_matches;
    for( int i = 0; i < descriptors_object.rows; i++ )
    {
        if( matches[i].distance < 3*min_dist )
         {
             good_matches.push_back( matches[i]);
         }
    }

    //绘制出匹配到的关键点
    Mat img_matches;
    drawMatches( srcImage1, keypoints_object, srcImage2,
keypoints_scene,
        good_matches, img_matches, Scalar::all(-1), Scalar::all(-1),
        vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

    //定义两个局部变量
    vector<Point2f> obj;
    vector<Point2f> scene;

    //从匹配成功的匹配对中获取关键点
    for( unsigned int i = 0; i < good_matches.size(); i++ )
    {

    obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );

    scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
    }

    Mat H = findHomography( obj, scene, CV_RANSAC );//计算透视变换
```

```
    //从待测图片中获取角点
    vector<Point2f> obj_corners(4);
    obj_corners[0] = cvPoint(0,0); obj_corners[1] =
cvPoint( srcImage1.cols, 0 );
    obj_corners[2] = cvPoint( srcImage1.cols, srcImage1.rows );
obj_corners[3] = cvPoint( 0, srcImage1.rows );
    vector<Point2f> scene_corners(4);

    //进行透视变换
    perspectiveTransform( obj_corners, scene_corners, H);

    //绘制出角点之间的直线
    line( img_matches, scene_corners[0] +
Point2f( static_cast<float>(srcImage1.cols), 0), scene_corners[1] +
Point2f( static_cast<float>(srcImage1.cols), 0), Scalar(255, 0, 123),
4 );
    line( img_matches, scene_corners[1] +
Point2f( static_cast<float>(srcImage1.cols), 0), scene_corners[2] +
Point2f( static_cast<float>(srcImage1.cols), 0), Scalar( 255, 0, 123),
4 );
    line( img_matches, scene_corners[2] +
Point2f( static_cast<float>(srcImage1.cols), 0), scene_corners[3] +
Point2f( static_cast<float>(srcImage1.cols), 0), Scalar( 255, 0, 123),
4 );
    line( img_matches, scene_corners[3] +
Point2f( static_cast<float>(srcImage1.cols), 0), scene_corners[0] +
Point2f( static_cast<float>(srcImage1.cols), 0), Scalar( 255, 0, 123),
4 );

    //显示最终结果
    imshow( "Good Matches & Object detection", img_matches );

    waitKey(0);
    return 0;
}
```

运行效果图如图 11.25、11.26 所示。

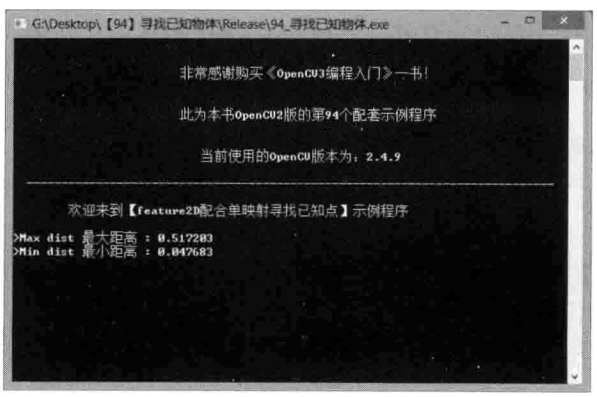


图 11.25 文本输出窗口

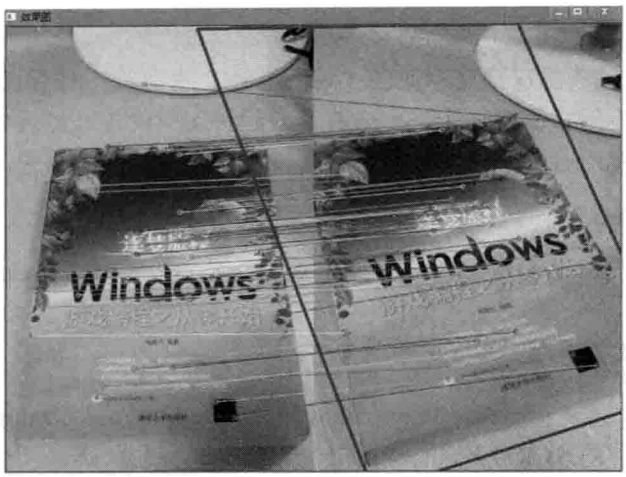


图 11.26 检测效果图

可以发现，图中用细线条匹配对应点，用粗线条指出检测到的物体。

11.5 ORB 特征提取

11.5.1 ORB 算法概述

ORB 为是ORiented Brief的简称，是brief算法的改进版。ORB 于2011 年在《ORB: an efficient alternative to SIFT or SURF》这篇文章中被提出。此文章的摘要中说，ORB 算法比 sift 算法效率高两个数量级，而在计算速度上，ORB 是 sift 的 100 倍，是 surf 的 10 倍。而江湖上流传的说法是，ORB 算法综合性能在各种测评里相较于其他特征提取算法是最好的。

若想要引出 ORB（ORiented Brief）算法，要由 Brief 描述子入手，下面，就来先介绍 Brief 描述子。

11.5.2 相关概念认知

1. 关于 Brief 描述子

Brief 是 Binary Robust Independent Elementary Features 的缩写。这个特征描述子是由 EPFL 的 Calonder 在 ECCV2010 上提出的，主要思路就是在特征点附近随机选取若干点对，将这些点对的灰度值的大小，组合成一个二进制串，并将这个二进制串作为该特征点的特征描述子。

BRIEF 的优点在于速度，而缺点也相当明显。

- 不具备旋转不变性。
- 对噪声敏感
- 不具备尺度不变性。

而 ORB 算法就是试图解决上述缺点中的 1 和 2 提出的一种新概念。

2. 关于尺度不变性

ORB 没有试图解决尺度不变性（因为 FAST 本身就不具有尺度不变性），但是这样只求速度的特征描述子，一般都是应用在实时的视频处理中的，这样的话就可以通过跟踪还有一些启发式的策略来解决尺度不变性的问题。

3. 关于计算速度

经过统计，ORB 算法的执行速度是 SIFT 的 100 倍，是 SURF 的 10 倍。

11.5.3 ORB 类相关源码简单分析

同样是在路径 …opencv\build\include\opencv2\features2d\features2d.hpp 下，我们会发现和之前的 SURF 类源码分析时惊人相似的两句代码：

```
typedef ORB OrbFeatureDetector;
typedef ORB OrbDescriptorExtractor;
```

这就是说，ORB，OrbFeatureDetector，OrbDescriptorExtractor 这三个类，也是完全等价的。然后我们继续溯源，发现 ORB 类同样继承自 Feature2D 类：

```
class CV_EXPORTS_W ORB : public Feature2D
{
//.....
};
```

接下来的分析思路，和我们之前讲解 SURF 类的源码时一样，经过分析，可以得到如图 11.27 所示的类关系脉络图。

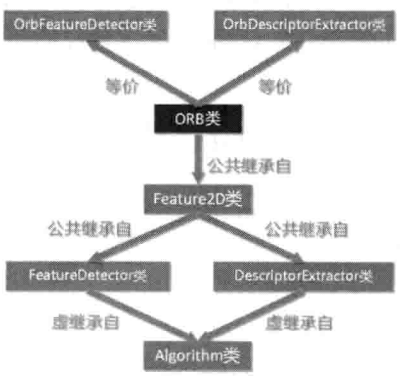


图 11.27 ORB 类相关联系脉络图

11.5.4 示例程序：ORB 算法描述与匹配

在与时俱进的 OpenCV 中，ORB 算法已被实现出来，而我们将其代码修改和注释，便得到了本期的示例程序。

此程序演示了 ORB 的关键点和描述符的提取，采用摄像头获取待检测图像，使用 FLANN-LSH 进行匹配。经过详细注释的代码如下。

```
//----------------------【头文件、命名空间包含部分】----------------------
//          描述：包含程序所依赖的头文件和命名空间
```

```
//-----------------------------------------------------------------------
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
using namespace cv;
using namespace std;


//-----------------------------【main()函数】---------------------------
//          描述：控制台应用程序的入口函数，我们的程序从这里开始执行
//-----------------------------------------------------------------------
int main()
{

    //【0】载入源图，显示并转化为灰度图
    Mat srcImage = imread("1.jpg");
    imshow("原始图",srcImage);
    Mat grayImage;
    cvtColor(srcImage, grayImage, CV_BGR2GRAY);

    //--------------检测 SIFT 特征点并在图像中提取物体的描述符--------------

    //【1】参数定义
    OrbFeatureDetector featureDetector;
    vector<KeyPoint> keyPoints;
    Mat descriptors;

    //【2】调用 detect 函数检测出特征关键点，保存在 vector 容器中
    featureDetector.detect(grayImage, keyPoints);

    //【3】计算描述符（特征向量）
    OrbDescriptorExtractor featureExtractor;
    featureExtractor.compute(grayImage, keyPoints, descriptors);

    //【4】基于 FLANN 的描述符对象匹配
    flann::Index flannIndex(descriptors, flann::LshIndexParams(12, 20,
2), cvflann::FLANN_DIST_HAMMING);

    //【5】初始化视频采集对象
    VideoCapture cap(0);
    cap.set(CV_CAP_PROP_FRAME_WIDTH, 360);//设置采集视频的宽度
    cap.set(CV_CAP_PROP_FRAME_HEIGHT, 900); //设置采集视频的高度

    unsigned int frameCount = 0;//帧数

    //【6】轮询，直到按下 ESC 键退出循环
    while(1)
    {
        double time0 = static_cast<double>(getTickCount());//记录起始时间
        Mat  captureImage, captureImage_gray;//定义两个 Mat 变量，用于视频
采集
```

```
        cap >> captureImage;//采集视频帧
        if( captureImage.empty())//采集为空的处理
         continue;

        cvtColor( captureImage, captureImage_gray, CV_BGR2GRAY);//采集
的视频帧转化为灰度图

        //【7】检测 SIFT 关键点并提取测试图像中的描述符
        vector<KeyPoint> captureKeyPoints;
        Mat captureDescription;
        //【8】调用 detect 函数检测出特征关键点，保存在 vector 容器中
        featureDetector.detect(captureImage_gray, captureKeyPoints);

        //【9】计算描述符
        featureExtractor.compute(captureImage_gray, captureKeyPoints,
captureDescription);

        //【10】匹配和测试描述符，获取两个最邻近的描述符
        Mat matchIndex(captureDescription.rows, 2, CV_32SC1),
matchDistance(captureDescription.rows, 2, CV_32FC1);
        flannIndex.knnSearch(captureDescription, matchIndex,
matchDistance, 2, flann::SearchParams());//调用 K 邻近算法

        //【11】根据劳氏算法（Lowe's algorithm）选出优秀的匹配
        vector<DMatch> goodMatches;
        for(int i = 0; i < matchDistance.rows; i++)
        {
         if(matchDistance.at<float>(i, 0) < 0.6 *
matchDistance.at<float>(i, 1))
         {
             DMatch dmatches(i, matchIndex.at<int>(i, 0),
matchDistance.at<float>(i, 0));
             goodMatches.push_back(dmatches);
         }
        }

        //【12】绘制并显示匹配窗口
        Mat resultImage;
        drawMatches( captureImage, captureKeyPoints, srcImage,
keyPoints, goodMatches, resultImage);
        imshow("匹配窗口", resultImage);

        //【13】显示帧率
        cout << ">帧率= " << getTickFrequency() / (getTickCount() - time0)
<< endl;

        //按下 ESC 键，则程序退出
        if(char(waitKey(1)) == 27) break;
    }

    return 0;
}
```

此程序的运行截图如图 11.28、11.29 所示。

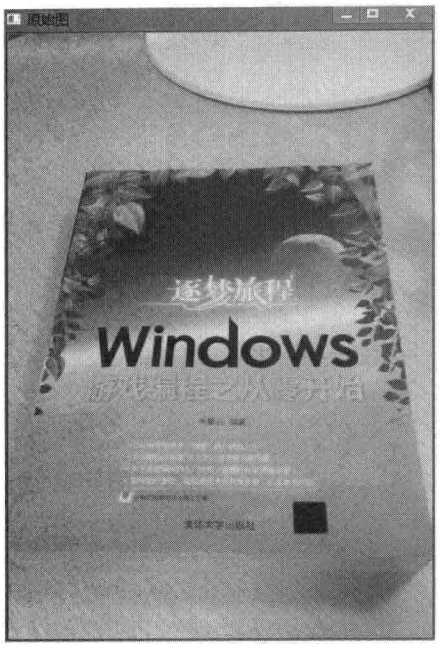


图 11.28 原始图

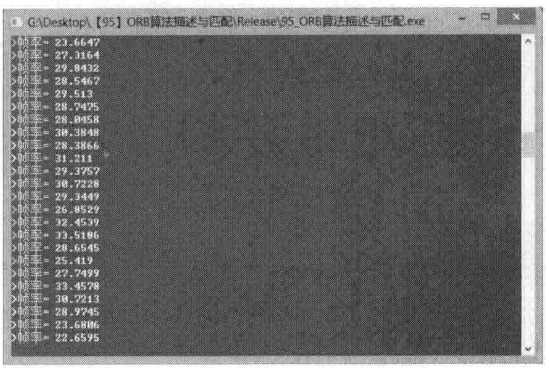


图 11.29 输出帧率的窗口

同样，当找一本和原图不相关的书籍对准摄像头，得到的匹配点寥寥无几。如图 11.30 所示。

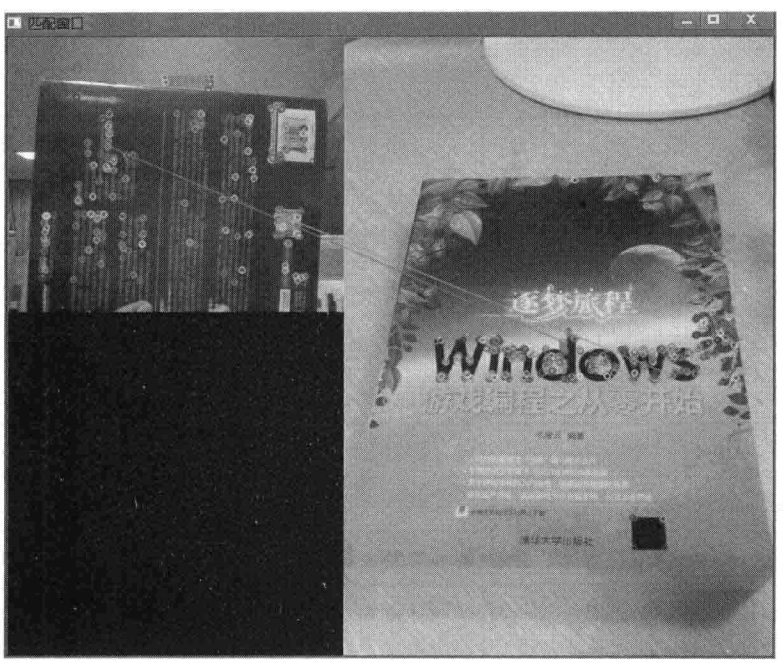


图 11.30 匹配点寥寥无几

而当找到原始图对应的书对准摄像头时，得到的匹配点多且整齐。如图 11.31 所示。

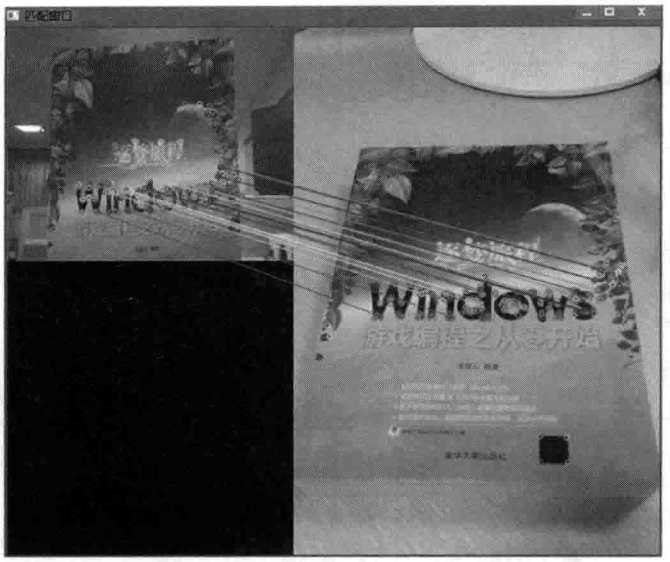


图 11.31 整齐的匹配点

11.6 本章小结

本章中我们主要使用 OpenCV2 讲解和实现了 SURF、SIFT 和 ORB 特征检测方法，其中 SURF 是尺度不变特征变换算法（SIFT 算法）的加速版，而 ORB 为 brief 算法的改进版。此外，还讲解了如何使用 FLANN 实现快速高效匹配。而在 FLANN 特征匹配的基础上，还可以进一步利用 Homography 映射找出已知物体。

本章核心函数清单

函数名称	说明	对应讲解章节
SURF 类、SurfFeatureDetector 类、SurfDescriptorExtractor 类	三者等价，同用于在 OpenCV 中进行 SURF 特征检测	11.1.3
drawKeypoints 函数	绘制关键点	11.1.4
drawMatches 函数	绘制出相匹配的两个图像的关键点	11.2.1
KeyPoint 类	用于表示特征点的信息	11.1.5
BruteForceMatcher 类	进行暴力匹配相关的操作	11.2.2
FlannBasedMatcher 类	实现 FLANN 特征匹配	11.3.1
DescriptorMatcher::match 函数	从每个描述符查询集中找到最佳匹配	11.3.2
findHomography 函数	找到并返回源图像和目标图像之间的透视变换 H	11.4.1
perspectiveTransform 函数	进行向量透视矩阵变换	11.4.2
ORB 类、OrbFeatureDetector 类、OrbDescriptorExtractor 类	三者等价，同用于在 OpenCV 中进行 ORB 特征检测	11.5.3

本章示例程序清单

示例程序序号	程序说明	对应章节
89	SURF 特征点检测	11.1.6
90	SURF 特征提取	11.2.3
91	使用 FLANN 进行特征点匹配	11.3.3
92	FLANN 结合 SURF 进行关键点的描述和匹配	11.3.4
93	SIFT 配合暴力匹配进行关键点描述和提取	11.3.5
94	寻找已知物体	11.4.3
95	利用 ORB 算法进行关键点的描述与匹配	11.5.4

作为一本入门级的 OpenCV 编程教材，我们以程序代码为主线，详细讲解了 OpenCV 最常用的 core、highgui、improc 和 feature2d 这四个组件的用法。至此，本书的主体内容已经结束。

愿大家在本书的帮助下，都能很好地入门和掌握新版 OpenCV。

愿本书能为新版 OpenCV 在国内的普及以及在世界范围内的发展，献上绵薄之力。

附录

A1 配套示例程序清单

本书含有 4 个部分 11 章，共有 95 个主线示例程序，为方便读者查阅和学习，总结成如下表格。

表 A1.1 配套示例程序清单

示例程序序号	程序说明	对应章节
1	OpenCV 环境配置的测试用例	1.3.8
2	快速上手 OpenCV 的第一个程序：图像显示	1.4.1
3	快速上手 OpenCV 的第二个程序：图像腐蚀	1.4.2
4	快速上手 OpenCV 的第三个程序：blur 图像模糊	1.4.3
5	快速上手 OpenCV 的第四个程序：canny 边缘检测	1.4.4
6	读取并播放视频	1.5.1
7	调用摄像头采集图像	1.5.2
8	官方例程引导、赏析之彩色目标跟踪：Camshift	2.1.1
9	官方例程引导、赏析之光流：optical flow	2.1.2
10	官方例程引导、赏析之点追踪：lkdemo	2.1.3
11	官方例程引导、赏析之人脸识别：objectDetection	2.1.4
12	官方例程引导、赏析之支持向量机：支持向量机引导	2.1.5
13	官方例程引导、赏析之支持向量机：处理线性不可分数据	2.1.5
14	printf 函数的用法示例	2.6.2
15	用 imwrite 函数生成 png 透明图	3.1.8
16	综合示例程序：图像的载入、显示与输出	3.1.9
17	为程序界面添加滑动条	3.2.1
18	鼠标操作示例	3.3
19	基础图像容器 Mat 类的使用	4.1.7
20	用 OpenCV 进行基本绘图	4.3
21	操作图像中像素的方法一：用指针访问像素	5.1.5、5.1.6
22	操作图像中像素的方法二：用迭代器操作像素	5.1.5、5.1.6

续表

示例程序序号	程序说明	对应章节
23	操作图像中像素的方法三：动态地址计算	5.1.5、5.1.6
24	遍历图像中像素的 14 种方法	5.1.6
25	初级图像混合	5.2.4
26	多通道图像混合	5.3.3
27	图像对比度、亮度值调整	5.4.3
28	离散傅里叶变换	5.5.8
29	XML 和 YAML 文件的写入	5.6.3
30	XML 和 YAML 文件的读取	5.6.4
31	方框滤波：boxFilter 函数的使用	6.1.11
32	均值滤波：blur 函数的使用	6.1.11
33	高斯滤波：GaussianBlur 函数的使用	6.1.11
34	综合示例：图像线性滤波	6.1.12
35	中值滤波：medianBlur 函数的使用	6.2.4
36	双边滤波：bilateralFilter 函数的使用	6.2.4
37	综合示例：图像滤波	6.2.5
38	膨胀：dilate 函数的使用	6.3.5
39	腐蚀：erode 函数的使用	6.3.5
40	综合示例：腐蚀与膨胀	6.3.6
41	用 morphologyEx()函数实现形态学膨胀	6.4.8
42	用 morphologyEx()函数实现形态学腐蚀	6.4.8
43	用 morphologyEx()函数实现形态学开运算	6.4.8
44	用 morphologyEx()函数实现形态学闭运算	6.4.8
45	用 morphologyEx()函数实现形态学梯度	6.4.8
46	用 morphologyEx()函数实现形态学“顶帽”	6.4.8
47	用 morphologyEx()函数实现形态学“黑帽”	6.4.8
48	综合示例：形态学滤波	6.4.9
49	漫水填充算法：floodFill 函数	6.5.3
50	综合示例：漫水填充	6.5.4
51	尺寸调整：resize()函数的使用	6.6.5

续表

示例程序序号	程序说明	对应章节
52	向上采样图像金字塔：pyrUp()函数的使用	6.6.6
53	向下采样图像金字塔：pyrDown()函数的使用	6.6.6
54	综合示例：图像金字塔与图片尺寸缩放	6.6.7
55	示例程序：基本阈值操作	6.7.3
56	Canny 边缘检测	7.1.2
57	Sobel 算子的使用	7.1.3
58	Laplacian 算子的使用	7.1.4
59	Scharr 滤波器	7.1.5
60	综合示例：边缘检测	7.1.6
61	标准霍夫变换：HoughLines()函数的使用	7.2.4
62	累计概率霍夫变换：HoughLinesP()函数	7.2.5
63	霍夫圆变换：HoughCircles()函数	7.2.8
64	综合示例：霍夫变换	7.2.9
65	实现重映射：remap()函数	7.3.3
66	综合示例程序：实现多种重映射	7.3.4
67	仿射变换	7.4.5
68	直方图均衡化	7.5.3
69	轮廓查找	8.1.3
70	查找并绘制轮廓	8.1.4
71	凸包检测基础	8.2.3
72	寻找和绘制物体的凸包	8.2.4
73	创建包围轮廓的矩形边界	8.3.6
74	创建包围轮廓的圆形边界	8.3.7
75	使用多边形包围轮廓	8.3.8
76	图像轮廓矩	8.4.4
77	分水岭算法的使用	8.5.2
78	实现图像修补	8.6.2
79	H-S 二维直方图的绘制	9.2.3
80	一维直方图的绘制	9.2.4

续表

示例程序序号	程序说明	对应章节
81	RGB 三色直方图的绘制	9.2.5
82	直方图对比	9.3.2
83	反向投影	9.4.7
84	模板匹配	9.5.3
85	实现 Harris 角点检测：cornerHarris()函数的使用	10.1.4
86	harris 角点检测与绘制	10.1.5
87	Shi-Tomasi 角点检测	10.2.3
88	亚像素级角点检测	10.3.3
89	SURF 特征点检测	11.1.6
90	SURF 特征提取	11.2.3
91	使用 FLANN 进行特征点匹配	11.3.3
92	FLANN 结合 SURF 进行关键点的描述和匹配	11.3.4
93	SIFT 配合暴力匹配进行关键点描述和提取	11.3.5
94	寻找已知物体	11.4.3
95	利用 ORB 算法进行关键点的描述与匹配	11.5.4

A2 随书额外附赠的程序一览

附录的这一部分算是购买本书后获赠的福利。很清楚大家会对目前稀缺的基于 OpenCV2 的这余下的二十几个示例程序抱有很大的兴趣，于是便有了这一部分的诞生。

本书在开始策划写作时，准备了一百多个基于 OpenCV2 的示例程序作为教学素材，最终定下来正文部分采用的示例程序是 95 个，也就是现在大家在书本中已经看到的一句一句串起本书主线内容那些“中流砥柱”们。

这样余下来的这些示例程序，也就找到了他们存在的意义。

需要说明的是，这些程序大多取材于 OpenCV 官方示例程序、或者取材、灵感源自于参考文献中列举出的国外的 OpenCV2 教程，也很容易转化到 OpenCV3 中运行。这些程序位于书本 OpenCV2 版的示例程序包中“【2】附赠示例程序源代码”文件夹里，并按序号顺序进行了标注。

为了方便大家查阅和学习，如下便是对这些附赠程序的表格式总结清单：

表 A2.1 附赠程序清单

程序序号	示例名称	说明
1	随机图形和文字生成示例（randomtext）	此程序利用 OpenCV 中的各种绘制函数随机生成图形和文字，有一定的学习和研究价值
2	生成彩色色条（gencolors）	用法 generateColors 函数生成彩色色条并进行显示
3	卡尔曼滤波（kalman）	用 OpenCV 动态绘制卡尔曼滤波，运行程序后可直接得出动画效果。用键盘任意按键重置轨迹并更新速度。使用 ESC 键结束程序
4	渐变过渡各种图形滤波（median_blur）	渐变过渡效果的各种图形滤波的显示，并输出说明性文字到窗口中
5	距离变换（distanceTransform）	此程序用于演示边缘图像之间的距离变换。 按键说明： 【ESC】 -退出程序 【c】-使用 C/Inf 度量 【1】-使用 L1 度量 【2】-使用 L2 度量 【3】- 使用 3 x 3 的掩膜 【5】- 使用 5 x 5 的掩膜 【0】- 采用精确的距离变换 【v】- 切换到 Voronoi 图（Voronoi diagram）模式 【p】 - 切换到基于像素的 Voronoi 图模式 【SPACE】- 在各种模式间切换
6	把图像映射到极指数空间（Log Polar）	此程序用于把图像映射到极指数空间，操作说明如下： 【n】-采用最邻近像素技术（nearest pixel technique） 【b】-采用双线性插值技术（bilinear interpolation technique） 【o】-使用重叠的圆形的接受域（overlapping circular receptive fields） 【a】-使用相邻的接受域（adjacent receptive fields）
7	filter2D 滤波器的用法	用 OpenCV 中的 filter2D 滤波器来模糊一张图片，并将结果存储到“filtered_image.jpg”中
8	grabCut 图像分割示例	此程序演示了 OpenCV 中 GrabCut 图像分割的使用。程序运行后，我们需要用鼠标圈出需要分割的那部分物体。按键说明如下： 【ESC】-退出程序 【r】-恢复原始图片 【n】-开始迭代，和进行下一次迭代 【鼠标左键】-设置选中矩形区域 【Ctrl+鼠标左键】-设置 GC_BGD 像素 【Shift+鼠标左键】-设置 CG_FGD 像素 【Ctrl+鼠标右键】-设置 GC_PR_BGD 像素 【Shift+鼠标右键】-设置 CG_PR_FGD 像素

续表

程序序号	示例名称	说明
9	MeanShift 图像分割示例	此程序演示了 OpenCV 中 MeanShift 图像分割的使用。程序运行后我们可以通过 3 个滑动条调节分割效果。3 个滑动条代表的参数分别为空间窗的半径（spatialRad）、色彩窗的半径（colorRad）、最大图像金字塔级别（maxPyrLevel）
10	用滑动控制图像直方图	此程序结合滚动条的创建，演示了如何用 calcHist 来创建直方图。可以条件滚动条，看到不同形态的图像直方图
11	找到图像最小的封闭轮廓	此程序结合了轮廓查找和多边形曲线精度逼近，来演示如何找到图像最小的封闭轮廓。运行程序即可观察出最终效果
12	Retina 特征点检测	此程序用于演示 Retina 特征点检测，运行后会得到多幅运行效果图
13	摄像头帧数检测	此程序非常简单实用，用于调用摄像头采集图像，并显示当前采集的图像帧数
14	视频截图	此程序也是非常简单实用，用于读取视频并播放，在播放时，按下【Space】空格键可以截图，图片将存放在工程目录下，而【Esc】和【q】键可以退出程序
15	对视频的快速角点检测	此程序用于演示如何对视频进行快速角点检测。按键说明如下： 【t】-抓取一个引用帧的进行匹配 【l】-使引用更新每一帧视频 【q】或【ESC】-退出程序
16	视频简单色彩检测	此程序调用摄像头进行视频采集，输出实时帧率，进行简单色彩检测，并可以用滑动条控制 R、G、B 三个通道的高低阈值
17	跟踪分割视频中运动的物体	此程序演示了一种寻找轮廓，连接组件，清除背景的简单方法，实现跟踪分割视频中运动的物体。程序运行开始后，便开始“学习背景”，我们可以通过【Space】空格键来切换是否打开“背景学习”技术
18	视频的直方图反向投影	此程序用摄像头采集视频，并进行实时的直方图方向投影显示
19	计算视频中两个图像区域的相似度	此程序用摄像头采集视频，然后我们可以在视频上用鼠标选定两个矩形区域，然后 OpenCV 就会为我们算出图像区域的相似度数值，并绘制出 RGB 三色直方图
20	视频前后背景分离	此程序展示了视频前后背景分离的方法，程序首先会“学习背景”，然后进行分割。可以用过【Space】空格进行功能切换

续表

程序序号	示例名称	说明
21	用高斯背景建模分离背景	此程序展示了用高斯背景建模进行视频的背景分离方法，程序首先会“学习背景”，然后进行分割。可以用过【Space】空格进行功能切换

A3 书本核心函数清单

附录的这一部分，为每章（（除 1、2 两章））的末尾小节中“本章核心函数清单”的一个汇总，也可以说是本书讲解的所有 OpenCV 核心函数和类的汇总。同样的，在此处进行列举，以方便读者在需要的时候进行查阅。

表 A3.1 本书核心函数/类清单

函数/类名称	用途	讲解章节
imread	用于读取文件中的图片到 OpenCV 中	3.1.4
imshow	在指定的窗口中显示一幅图像	3.1.5
namedWindow	用于创建一个窗口	3.1.7
imwrite	输出图像到文件	3.1.8
createTrackbar	用于创建一个可以调整数值的轨迹条	3.2.1
getTrackbarPos	用于获取轨迹条的当前位置	3.2.2
SetMouseCallback	为指定的窗口设置鼠标回调函数	3.3
Mat::Mat()	Mat 类的构造函数	4.1.4
Mat::Create()	Mat 类的成员函数，可用于 Mat 类的初始化操作	4.1.4
Point 类	用于表示点的数据结构	4.2.1
Scalar 类	用于表示颜色的数据结构	4.2.2
Size 类	用于表示尺寸的数据结构	4.2.3
Rect 类	用于表示矩形的数据结构	4.2.4
CvtColor()	用于颜色空间转换	4.2.5
line	绘制直线	4.3.4
ellipse	绘制椭圆	4.3.1
rectangle	绘制矩形	4.3.5
circle	绘制圆	4.3.2
fillPoly	绘制填充的多边形	4.3.3
addWeighted	计算两个数组（图像阵列）的加权和	5.2.3

续表

函数/类名称	用途	讲解章节
split	将一个多通道数组分离成几个单通道数组	5.3.1
merge	将多个数组组合合并成一个多通道的数组	5.3.2
dft	对一维或二维浮点数数组进行正向或反向离散傅里叶变换	5.5.2
getOptimalDFTSize	返回给定向量尺寸的傅里叶最优尺寸大小	5.5.3
copyMakeBorder	扩充图像边界	5.5.4
magnitude	计算二维矢量的幅值	5.5.5
log	计算每个数组元素绝对值的自然对数	5.5.6
normalize	进行矩阵归一化	5.5.7
FileStorage 类	进行文件操作的类	5.6.2
boxFilter	使用方框滤波来模糊一张图片	6.1.11
blur	对输入的图像进行均值滤波操作	6.1.11
GaussianBlur	用高斯滤波器来模糊一张图片	6.1.11
medianBlur	使用中值滤波器来模糊一张图片	6.2.4
bilateralFilter	用双边滤波器来模糊处理一张图片	6.2.4
dilate	使用像素邻域内的局部极大运算符来膨胀一张图片	6.3.5
erode	使用像素邻域内的局部极小运算符来腐蚀一张图片	6.3.5
morphologyEx	利用基本的膨胀和腐蚀技术，来执行更加高级形态学变换，如开闭运算，形态学梯度、顶帽、黑帽等，也可以实现最基本的图像膨胀和腐蚀	6.4.7
floodFill	用指定的颜色从种子点开始填充一个连接域，实现漫水填充算法	6.5.3
pyrUp	向上采样并模糊一张图片，说白了就是放大一张图片	6.6.6
pyrDown	向下采样并模糊一张图片，说白了就是缩小一张图片	6.6.6
Threshold	对单通道数组应用固定阈值操作	6.7.1
adaptiveThreshold	对矩阵采用自适应阈值操作	6.7.2
Canny	利用 Canny 算子来进行图像的边缘检测	7.1.2
Sobel	使用扩展的 Sobel 算子，来计算一阶、二阶、三阶或混合图像差分	7.1.3
Laplacian	计算出图像经过拉普拉斯变换后的结果	7.1.4

续表

函数/类名称	用途	讲解章节
Scharr	使用 Scharr 滤波器运算符计算 x 或 y 方向的图像差分	7.1.5
HoughLines	找出采用标准霍夫变换的二值图像线条	7.2.4
HoughLinesP	采用累计概率霍夫变换（PPHT）来找出二值图像中的直线	7.2.5
HoughCircles	利用霍夫变换算法检测出灰度图中的圆	7.2.8
remap	根据指定的映射形式，将源图像进行重映射几何变换	7.3.2
warpAffine	依据公式对图像做仿射变换	7.4.3
getRotationMatrix2D	计算二维旋转变换矩阵	7.4.4
equalizeHist	实现图像的直方图均衡化	7.5.2
findContours	在二值图像中寻找轮廓	8.1.1
drawContours	在图像中绘制外部或内部轮廓	8.1.2
convexHull	寻找图像点集中的凸包	8.2.2
boundingRect	计算并返回指定点集最外面（up-right）的矩形边界	8.3.1
minAreaRect	寻找可旋转的最小面积的包围矩形	8.3.2
minEnclosingCircle	利用一种迭代算法，对给定的 2D 点集，寻找面积最小的可包围他们的圆形	8.3.3
fitEllipse	用椭圆拟合二维点集	8.3.4
approxPolyDP	用指定精度逼近多边形曲线	8.3.5
moments	计算多边形和光栅形状的最高达三阶的所有矩	8.4.1
contourArea	计算整个轮廓或部分轮廓的面积	8.4.2
arcLength	计算封闭轮廓的周长或曲线的长度	8.4.3
watershed	实现分水岭算法	8.5.1
inpaint	进行图像修补，从扫描的照片中清除灰尘和划痕，或者从静态图像或视频中去除不需要的物体	8.6.1
calcHist	计算一个或者多个阵列的直方图	9.2.1
minMaxLoc	在数组中找到全局最小值和最大值	9.2.2
compareHist	对两幅直方图进行比较	9.3.1
calcBackProject	计算直方图的反向投影	9.4.5
mixChannels	由输入参数拷贝某通道到输出参数特定的通道中	9.4.6
matchTemplate	匹配出和模板重叠的图像区域	9.5.2

续表

函数/类名称	用途	讲解章节
CornerHarris	运行 Harris 角点检测算子来进行角点检测	10.1.4
goodFeaturesToTrack	结合 Shi-Tomasi 算子确定图像的强角点	10.2.2
cornerSubPix	寻找亚像素角点位置	10.3.2
SURF 类、SurfFeatureDetector 类、SurfDescriptorExtractor 类	三者等价，同用于在 OpenCV 中进行 SURF 特征检测	11.1.3
drawKeypoints	绘制关键点	11.1.4
drawMatches	绘制出相匹配的两个图像的关键点	11.2.1
KeyPoint 类	用于表示特征点的信息	11.1.5
BruteForceMatcher 类	进行暴力匹配相关的操作	11.2.2
FlannBasedMatcher 类	实现 FLANN 特征匹配	11.3.1
DescriptorMatcher::match	从每个描述符查询集中找到最佳匹配	11.3.2
findHomography	找到并返回源图像和目标图像之间的透视变换 H	11.4.1
perspectiveTransform	进行向量透视矩阵变换	11.4.2
ORB 类、OrbFeatureDetector 类、OrbDescriptorExtractor 类	三者等价，同用于在 OpenCV 中进行 ORB 特征检测	11.5.3

A4 Mat 类函数一览

Mat 类作为新版 OpenCV 中最核心的数据结构，有着非常健全的代码构造。本书正文第四章中已经对 Mat 类常用的一些构造函数和成员函数有了一些介绍，而这里的这一部分是一个补充，会将剩下遗漏的 Mat 类的构造函数、析构函数和成员函数进行完整的介绍。

A4.1 构造函数：Mat::Mat

Mat 类的众多构造函数。

```
C++: Mat::Mat()
C++: Mat::Mat(int rows, int cols, int type)
C++: Mat::Mat(Size size, int type)
C++: Mat::Mat(int rows, int cols, int type,const Scalar& s)
C++: Mat::Mat(Size size, int type, constScalar& s)
C++: Mat::Mat(const Mat& m)
C++: Mat::Mat(int rows, int cols, int type,void* data, size_t
step=AUTO_STEP)
C++: Mat::Mat(Size size, int type, void*data, size_t step=AUTO_STEP)
C++: Mat::Mat(const Mat& m, constRange& rowRange, const Range& colRange)
```

```
C++: Mat::Mat(const Mat& m, constRect& roi)
C++: Mat::Mat(const CvMat* m, boolcopyData=false)
C++: Mat::Mat(const IplImage* img, boolcopyData=false)
C++: template<typename T, int n>explicit Mat::Mat(const Vec<T, n>& vec,
bool copyData=true)
C++: template<typename T, int m, intn> explicit Mat::Mat(const Matx<T,
m, n>& vec, bool copyData=true)
C++: template<typename T> explicitMat::Mat(const vector<T>& vec, bool
copyData=false)
C++: Mat::Mat(const MatExpr& expr)
C++: Mat::Mat(int ndims, const int* sizes,int type)
C++: Mat::Mat(int ndims, const int* sizes,int type, const Scalar& s)
C++: Mat::Mat(int ndims, const int* sizes,int type, void* data, const
size_t* steps=0)
C++: Mat::Mat(const Mat& m, constRange* ranges)
```

这些构造函数的参数含义列举如下：

表 A3.1 Mat 类构造函数参数列举

参数	含义
ndims	数组的维数
rows	2 维数组的行数
cols	2 维数组的列数
size	2 维数组的尺寸 Size(cols, rows) .在 Size()构造函数中行数和列数在顺序上为反转过来的
sizes	指定 n 维数组形状的整数数组
type	数组的类型。使用 CV_8UC1，… …，创建 1-4 通道的矩阵，CV_64FC4 或 CV_8UC(n)，… …，CV_64FC(n)可以创建多通道 （高达 CV_MAX_CN 通道）矩阵
s	一个可选的初始化每个矩阵元素的参数。要在矩阵建成后将所有元素设置为特定值可以用 Mat 的赋值运算符 Mat:operator=(constScala& value)
data	指向用户数据的指针。矩阵构造函数传入 data 和 step 参数不分配矩阵数据。相反，它们只是初始化矩阵头指向指定的数据，这意味着没有数据的复制。此操作是很高效的，可以用来处理使用 OpenCV 函数的外部数据。外部数据不会自动释放，所以你应该小心处理它 step – 每个矩阵行占用的字节数。如果任何值应包括每行末尾的填充字节。如果缺少此参数（设置为 AUTO_STEP），假定没有填充和实际的步长用 cols*elemSize()计算。请参阅 Mat::elemSize()
steps	多维数组（最后一步始终设置为元素大小） 的情况下的 ndims-1 个步长的数组。如果没有指定的话，该矩阵假定为连续

续表

参数	含义
M	分配给构造出来的矩阵的阵列（作为一个整体或部分）。这些构造函数没有复制数据。相反，指向 m 的数据或它的子数组的头被构造并被关联到 m 上。引用计数器中无论如何都将递增。所以，当您修改矩阵的时候，自然而然就使用了这种构造函数，您还修改 m 中的对应元素。如果你想要独立的子数组的副本，请使用 Mat::clone()
img	指向老版本的 IplImage 图像结构的指针。默认情况下，原始图像和新矩阵之间共享数据。但当 copyData 被设置时，完整的图像数据副本就创建起来了
vec	矩阵的元素构成的 STL 向量。矩阵可以取出单独一列并且该列上的行数和矢量元素的数目相同。矩阵的类型匹配的向量元素的类型。构造函数可以处理任意的有正确声明的 DataType 类型。这意味着矢量元素不支持的混合型结构，它们必须是数据（numbers）原始数字或单型数值元组。对应的构造函数是显式的。由于 STL 向量不会自动转换为 Mat 实例，您应显式编写 Mat(vec)。除非您将数据复制到矩阵（copyData = true），没有新的元素被添加到向量中，因为这样可能会造成矢量数据重新分配，并且因此使得矩阵的数据指针无效
copyData	指定 STL 向量或旧型 CvMat 或 IplImage 是应复制到(true)新构造的矩阵中还是(false)与之共享基础数据的标志，复制数据时，使用 Mat 引用计数机制管理所分配的缓冲区。虽然数据共享的引用计数为 NULL，但是分配数据必须在矩阵被析构之后才可以释放
rowRange	矩阵阵列 m 之行数的取值范围。另外，可以使用 Range::all()来取所有的行
colRange	M 列数的取值范围。使用 Range::all()来取所有的列
ranges	表示 M 沿每个维度选定的区域的数组
expr	矩阵表达式。具体请参见上文“矩阵表达式”一节的内容

以上这些都是 Mat 生成一个矩阵的各类构造函数。其实很多时候，默认构造函数就足够了，其可由 OpenCV 函数来分配数据空间。而通过 Mat::create()，构造的矩阵可以进一步分配给其他的矩阵或者矩阵表达式。

A4.2 析构函数 Mat::~Mat

Mat 的析构函数为 Mat::～Mat()。原型声明如下：

```
C++: Mat::~Mat()
```

Mat 类的析构函数其实就是调用成员函数 Mat::release()进行析构操作。

A4.3 Mat 类成员函数

由于 Mat 类的类成员函数众多，不妨让我们通过列表的方式来进行列举和说明：

表 A3.2 Mat 类的类成员函数一览

Mat 类成员函数	作用
Mat::operator =	提供矩阵赋值操作
Mat::row	创建一个指定行数的矩阵头
Mat::col	创建一个指定列数的矩阵头
Mat::rowRange	创建指定行跨度（span）的矩阵头
Mat::colRange	创建指定列跨度的矩阵头
Mat::diag	提取或创建矩阵对角线
Mat::clone	创建一个数组及其基础数据的完整副本
Mat::copyTo	把矩阵复制到另一个矩阵中
Mat::convertTo	在缩放或不缩放的情况下转换为另一种数据类型
Mat::assignTo	提供了一种 convertTo 的功能形式
Mat::setTo	将阵列中所有的或部分的元素设置为指定的值
Mat::reshape	在无需复制数据的前提下改变 2D 矩阵的形状和通道数或其中之一
Mat::t	通过矩阵表达式（matrix expression）实现矩阵的转置
Mat::inv	反转矩阵
Mat::mul	执行两个矩阵按元素相乘或这两个矩阵的除法
Mat::cross	计算 3 元素向量的一个叉乘积
Mat::dot	计算两向量的点乘
Mat::zeros	返回指定的大小和类型的零数组
Mat::ones	返回一个指定的大小和类型的全为 1 的数组
Mat::eye	返回一个恒等指定大小和类型矩阵
Mat::create	分配新的阵列数据
Mat::addref	计数器参考
Mat::release	在必要的情况下，递减引用计数并释放该矩阵
Mat::resize	更改矩阵的行数
Mat::reserve	保留一定数量的行空间
Mat::push_back	将元素添加到矩阵的底部
Mat::pop_back	从底部的列表中删除元素
Mat::locateROI	父矩阵内定位矩阵头
Mat::adjustROI	调整子阵大小及其在父矩阵中的位置

续表

Mat 类成员函数	作用
Mat::operator()	提取矩形子阵
Mat::operator CvMat	创建矩阵 CvMat 头
Mat::operator IplImage	创建 IplImage 矩阵头
Mat::isContinuous	返回矩阵是否连续
Mat::elemSize	返回矩阵元素大小（以字节为单位）
Mat::type	返回一个矩阵元素的类型
Mat::depth	返回一个矩阵元素的深度
Mat::channels	返回矩阵通道的数目
Mat::step1	返回矩阵归一化的一步
Mat::size	返回一个矩阵大小
Mat::empty	如果数组中没有元素，则返回 true
Mat::ptr	返回指定矩阵行的指针
Mat::at	返回对指定数组元素的引用
Mat::begin	返回矩阵迭代器，并将其设置为第一个矩阵元素
Mat::end	返回矩阵迭代器，并将其设置在最后一个元素之后（after-last）

可以发现，Mat 类的成员函数众多，可以实现各种各样的操作。大家需要对 Mat 类进行有关操作的时候，可以在此表中进行查阅。

至此，本书的所有内容都已经结束。

主要参考文献

[1] Rafael C.Gonzalez, Digital Image Processing Third Edition [M], Prentice Hall, 2007

[2] Richard Szeliski,Computer Vision Algorithms and Applications[M], Springer, 2010

[3] Adrian Kaehler, Learning OpenCV, oreilly, 2008

[4] Daniel Lélis Baggio,Mastering OpenCV with Practical Computer Vision Projects, Packt Publishing, 2012

[5] Robert Laganière ,OpenCV 2 Computer Vision Application Programming Cookbook, Packt Publishing,2011

[6] Samarth Brahmbhatt ,Pratical OpenCV,APress,2013

[7] OpenCV 2.4.9.0 documentation, http://docs.opencv.org/modules/refman.html，2014

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

读者热评

很好，文章透彻，程序易懂，一看就会，能够快速掌握。

——枫羽落雪

对于新版OpenCV能有这么透彻的讲解，真是太好了。

——yuhangrenmin

文笔不错，而且对编程的理解很到位，向你学习。再次感谢。

——vacuum8899

很感谢你，看了你写的教程让我少走了很多弯路！

——nicepainkiller

您写的真透彻！解决很多的疑问和麻烦！感动！

——cy_543

写的很好，深入浅出，给的程序也很实用，佩服。

——xuliangfirst

代码写的通俗易懂，对于新手来说真是很有帮助！真心感谢楼主的无私奉献！

——Alex

作者代码写得很规范，注释也很清晰，结合书本讲解，浅显易懂。

——SkyHeight

讲得挺全面的。原理、实现、具体代码，非常适合初学者实用的教程。

——gmf_henu

文章讲解、注释内容写的很详细，初学者都能很容易看懂。

——mcuc51

代码详细具体，注释明确，很好的学习资料，参考书本学习，效果加倍。

——tuling56

相当感谢，写的不错，对于入门的初学者非常不错。

——lqiang0630

怒赞，写代码从细节写起，看这样的代码顺眼！

——liuchyi

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

上架建议：计算机>图像处理

ISBN 978-7-121-25331-7

定价：79.00元

责任编辑：陈晓猛
封面设计：李　玲