

OpenCV的主要开发者和OpenCV社区的主要贡献者携手，深入解析OpenCV技术在计算机视觉项目中的应用，Amazon广泛好评

通过典型计算机视觉项目，系统讲解使用OpenCV技术构建计算机视觉相关应用的各种技术细节、方法和最佳实践，并提供全部实现源码，为读者快速实践OpenCV技术提供翔实指导

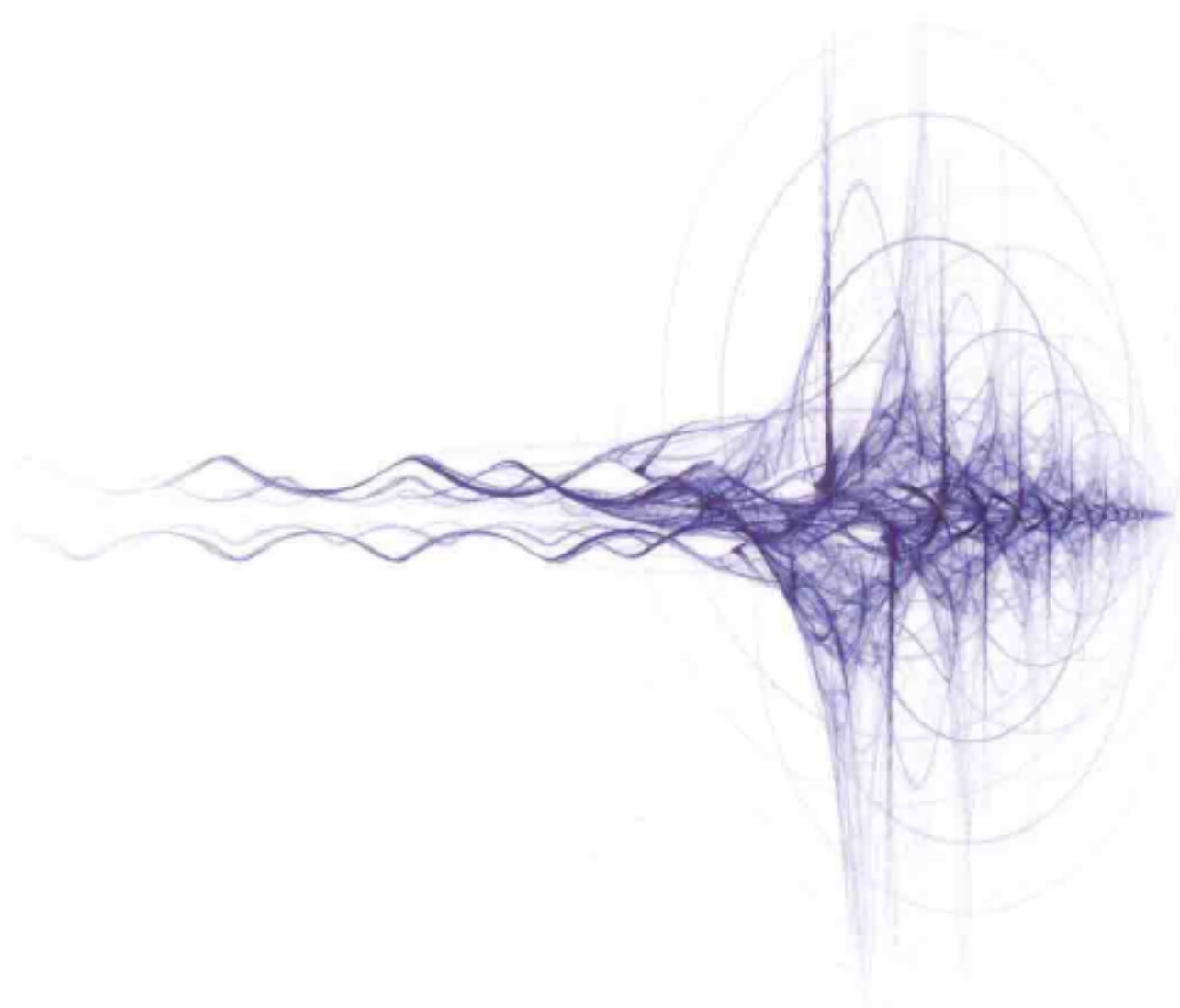
Mastering OpenCV with Practical Computer Vision Projects

# 深入理解OpenCV

## 实用计算机视觉项目解析

Daniel Lélis Baggio Shervin Emami David Millán Escrivá 著  
Khvedchenia Ievgen Naureen Mahmood Jason Saragih Roy Shilkrot

刘波 译



机械工业出版社  
China Machine Press

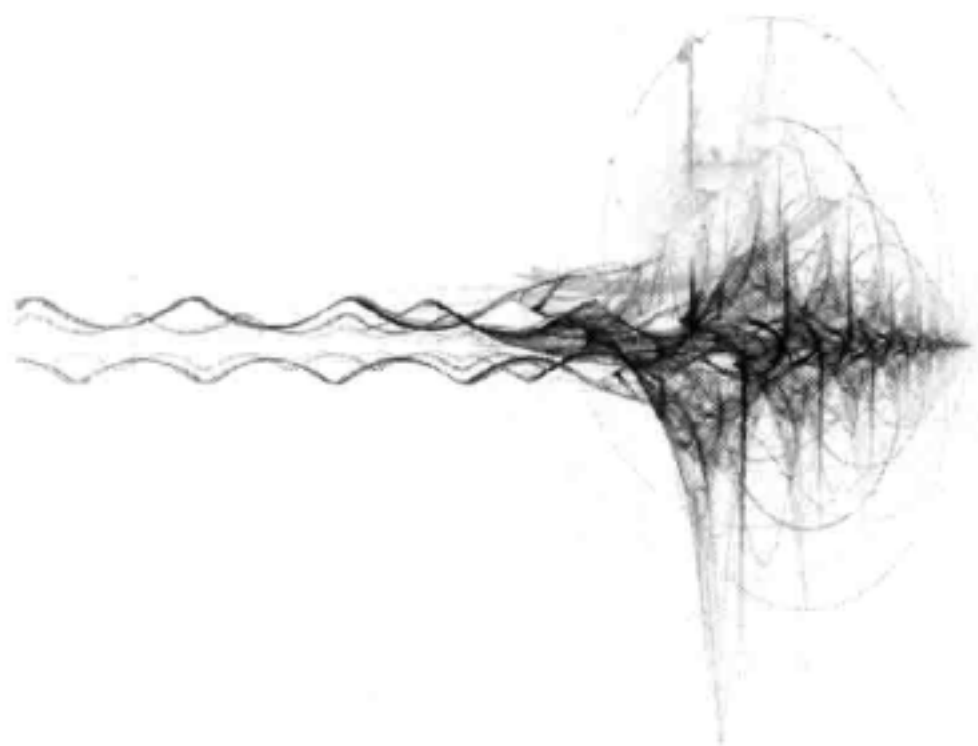
Mastering OpenCV with Practical Computer Vision Projects

# 深入理解OpenCV

## 实用计算机视觉项目解析

Daniel Lélis Baggio Shervin Emami David Millán Escrivá 著  
Khvedchenia Ievgen Naureen Mahmood Jason Saragih Roy Shilkrot

刘波 译



机械工业出版社  
China Machine Press



## 图书在版编目 (CIP) 数据

深入理解 OpenCV：实用计算机视觉项目解析 / (巴西) 博格等著；刘波译．—北京：机械工业出版社，2014.9

(华章程序员书库)

书名原文：Mastering OpenCV with Practical Computer Vision Projects

ISBN 978-7-111-47818-8

I. 深… II. ① 博… ② 刘… III. 图像处理软件—程序设计 IV. TP391.41

中国版本图书馆 CIP 数据核字 (2014) 第 204140 号

---

本书版权登记号：图字：01-2013-7571

Daniel Lélis Baggio, et al: *Mastering OpenCV with Practical Computer Vision Projects* (ISBN: 978-1-84951-782-9).

Copyright © 2012 Packt Publishing. First published in the English language under the title “Mastering OpenCV with Practical Computer Vision Projects”.

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2014 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

## 深入理解 OpenCV：实用计算机视觉项目解析

---

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：陈佳媛

责任校对：董纪丽

印 刷：三河市宏图印务有限公司

版 次：2014 年 9 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：15（含彩插 0.25 印张）

书 号：ISBN 978-7-111-47818-8

定 价：59.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

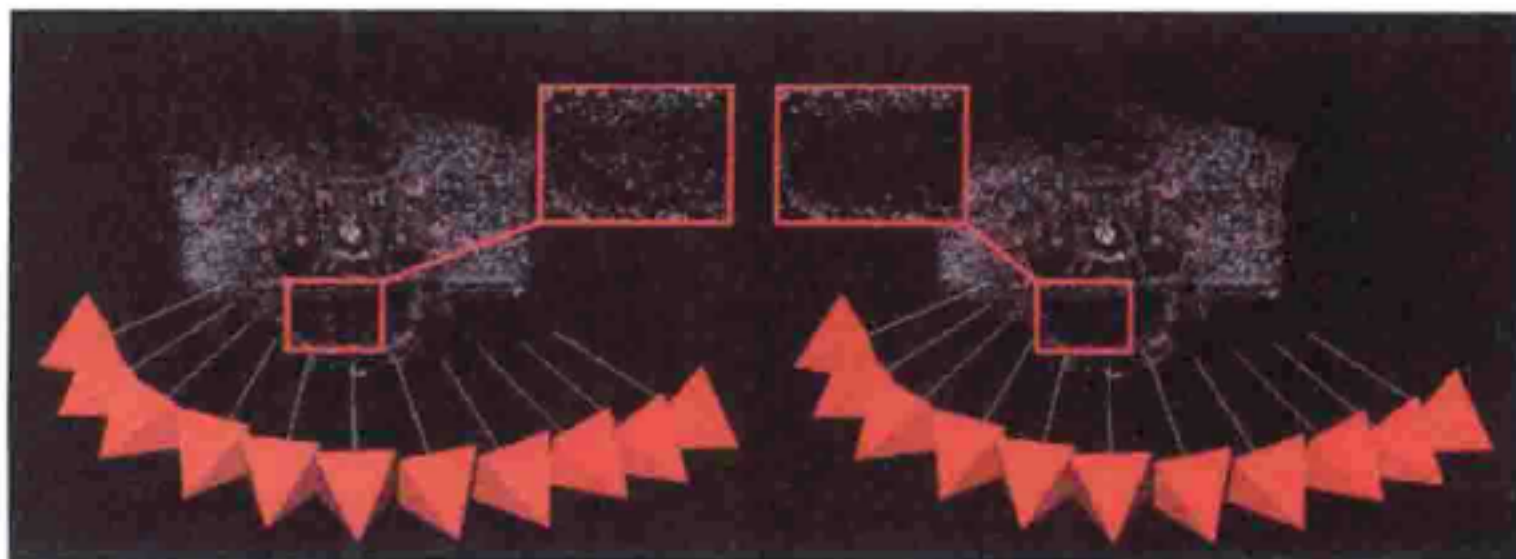
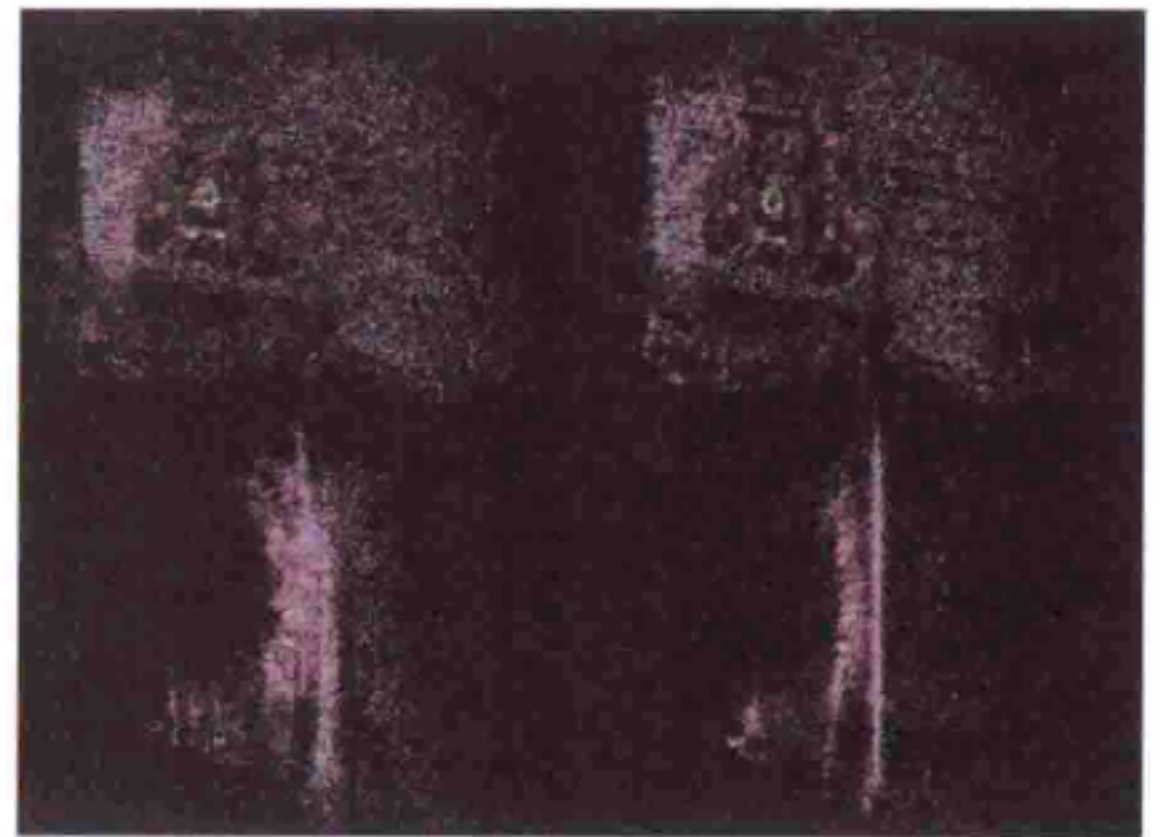
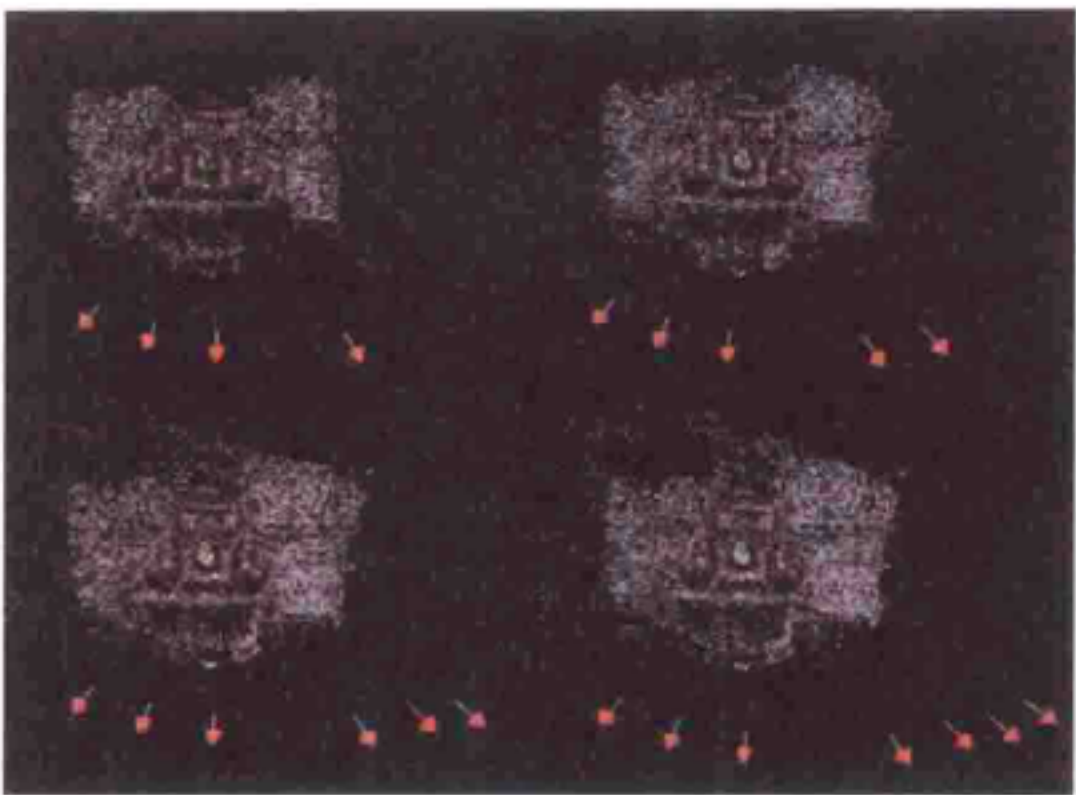
读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

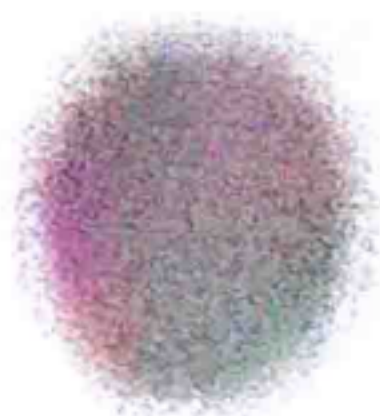
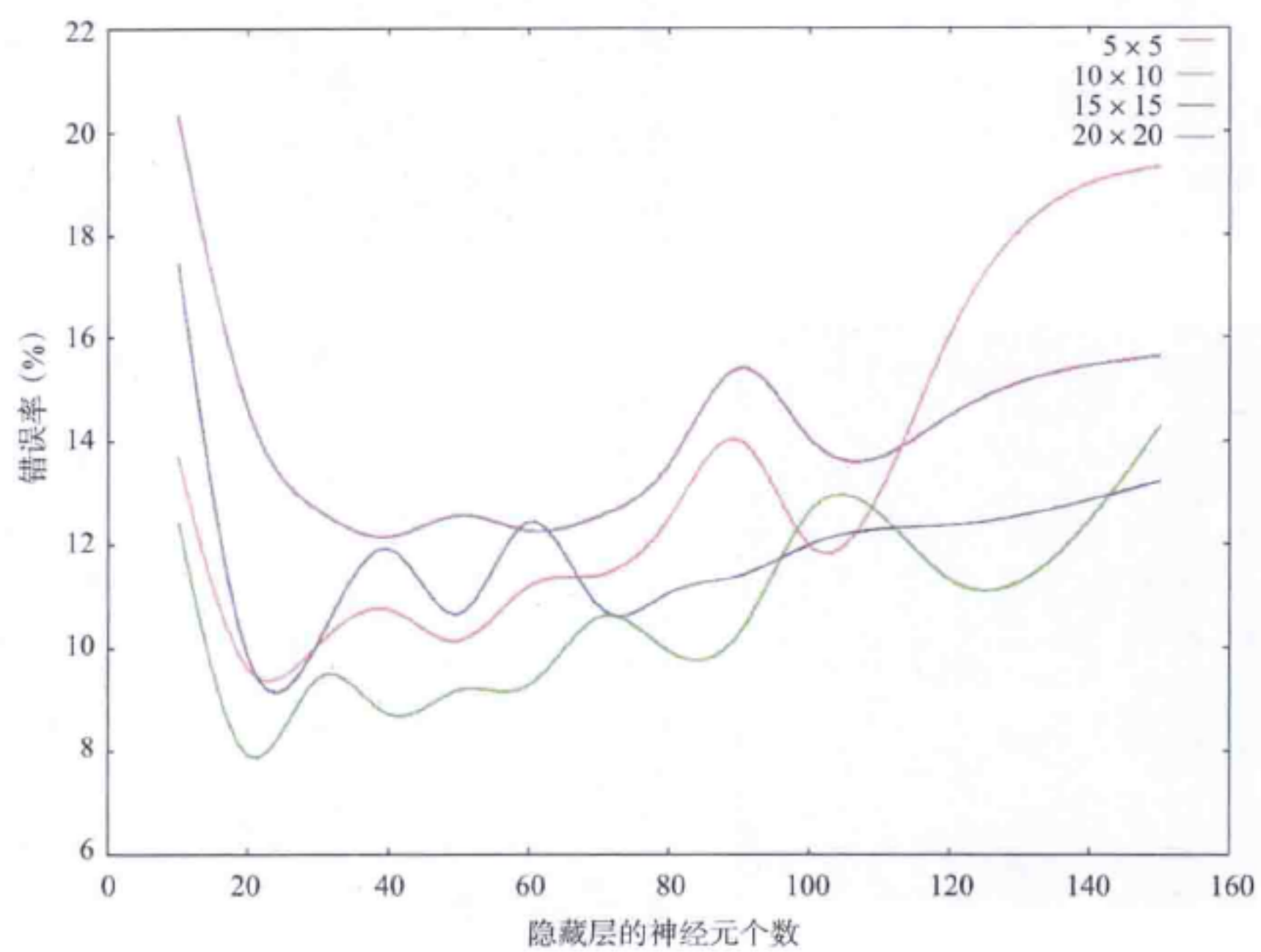
封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东









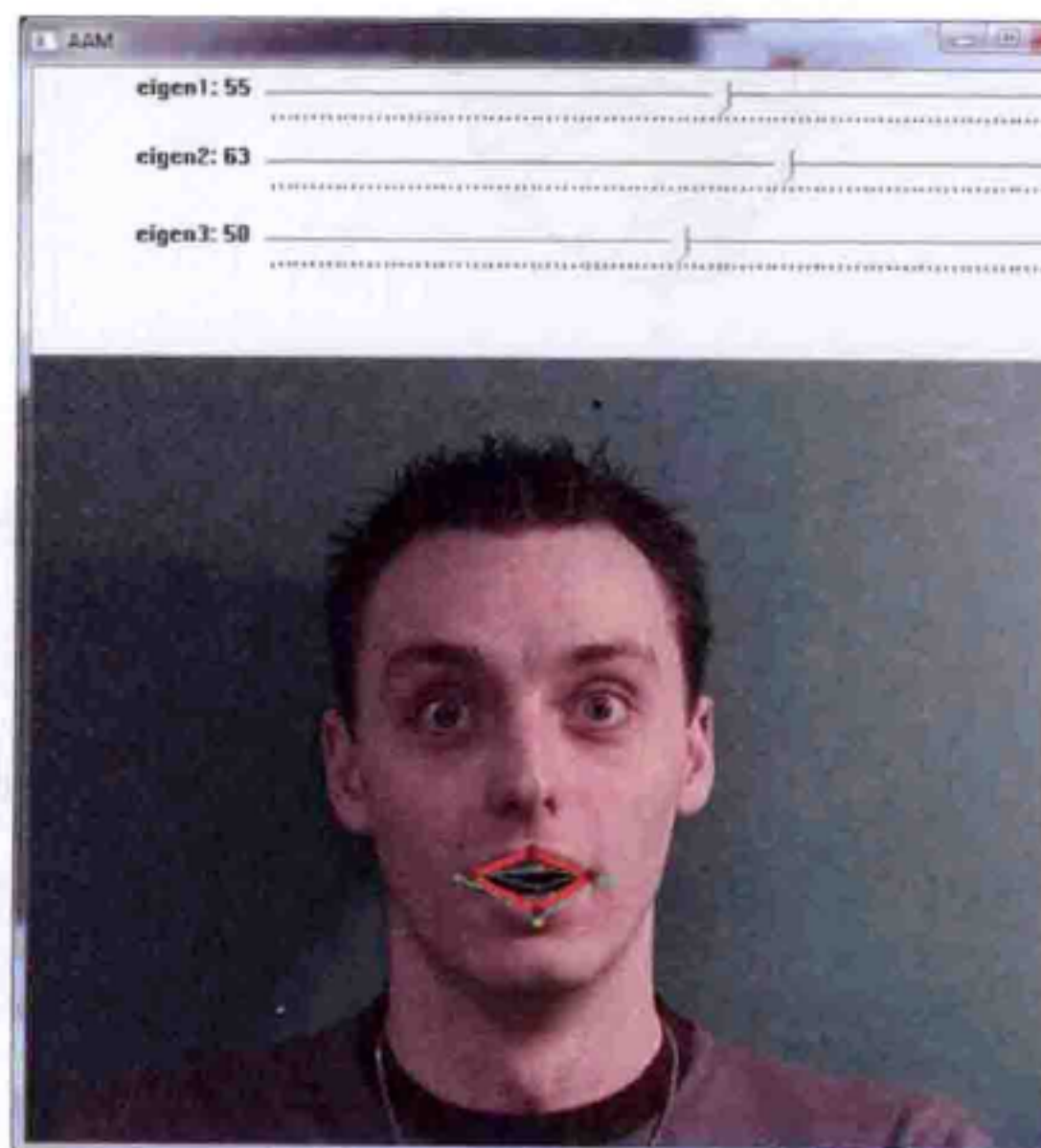
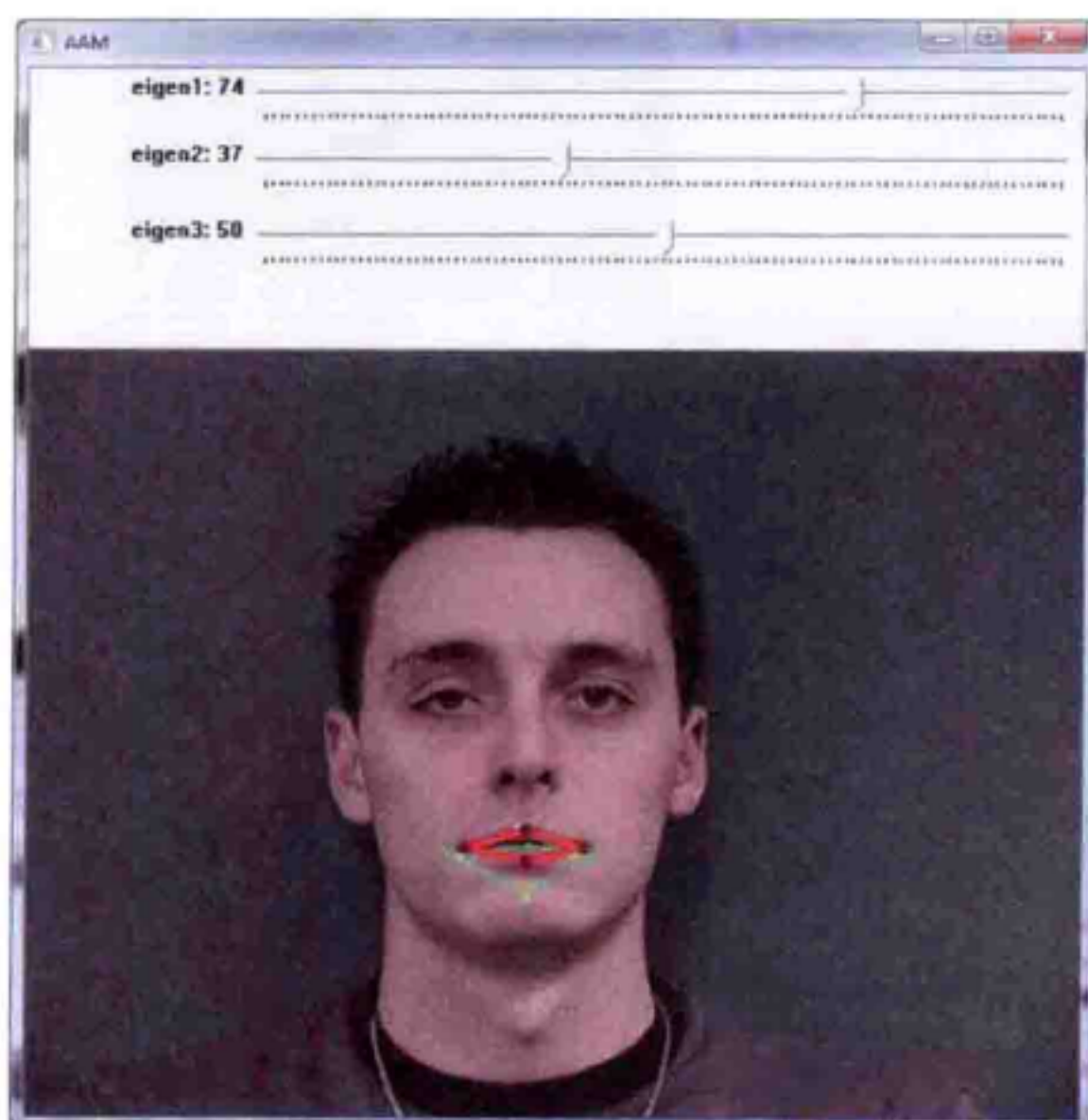
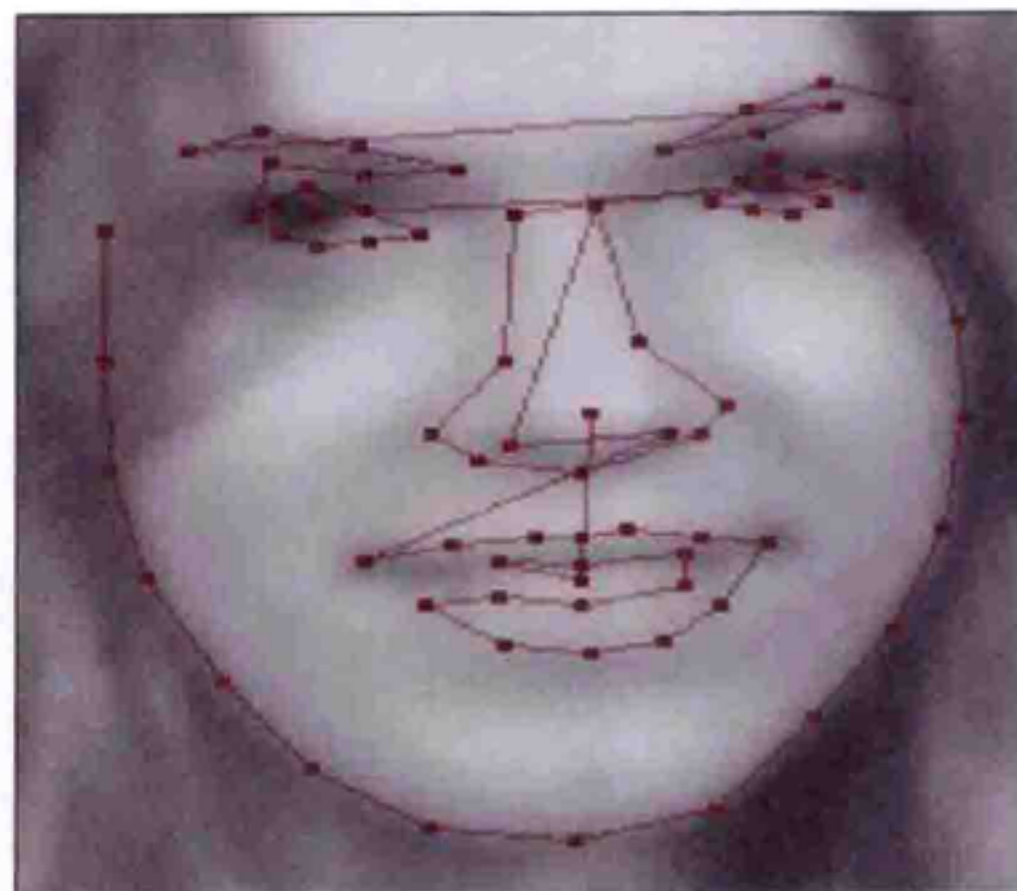
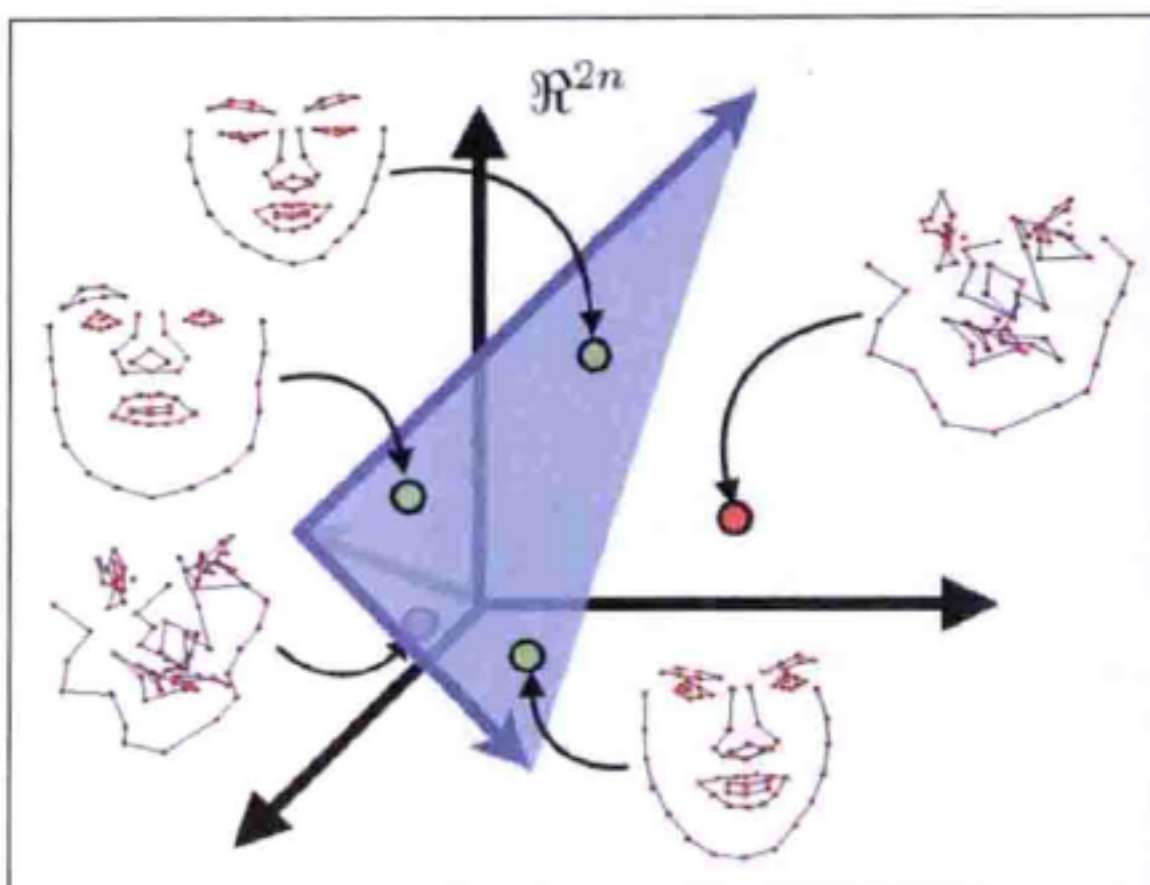
没有对齐的形状



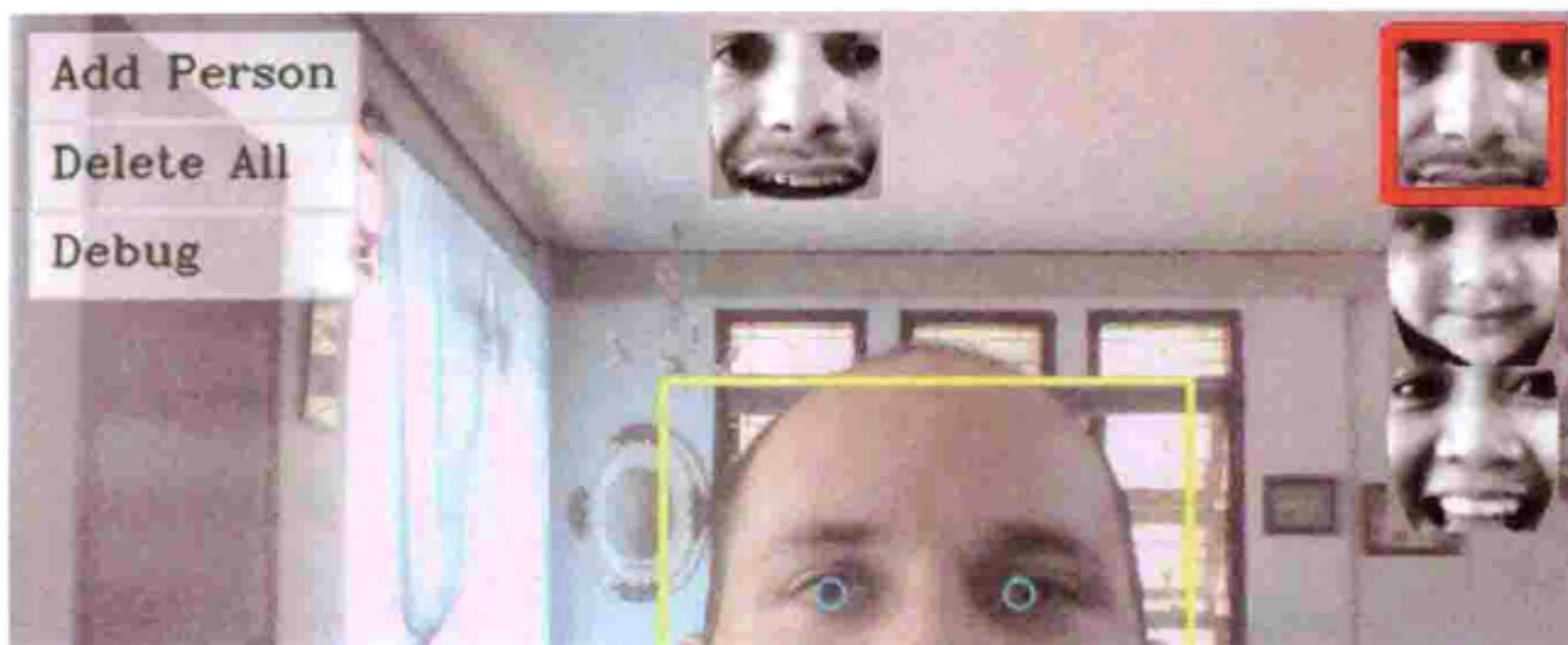
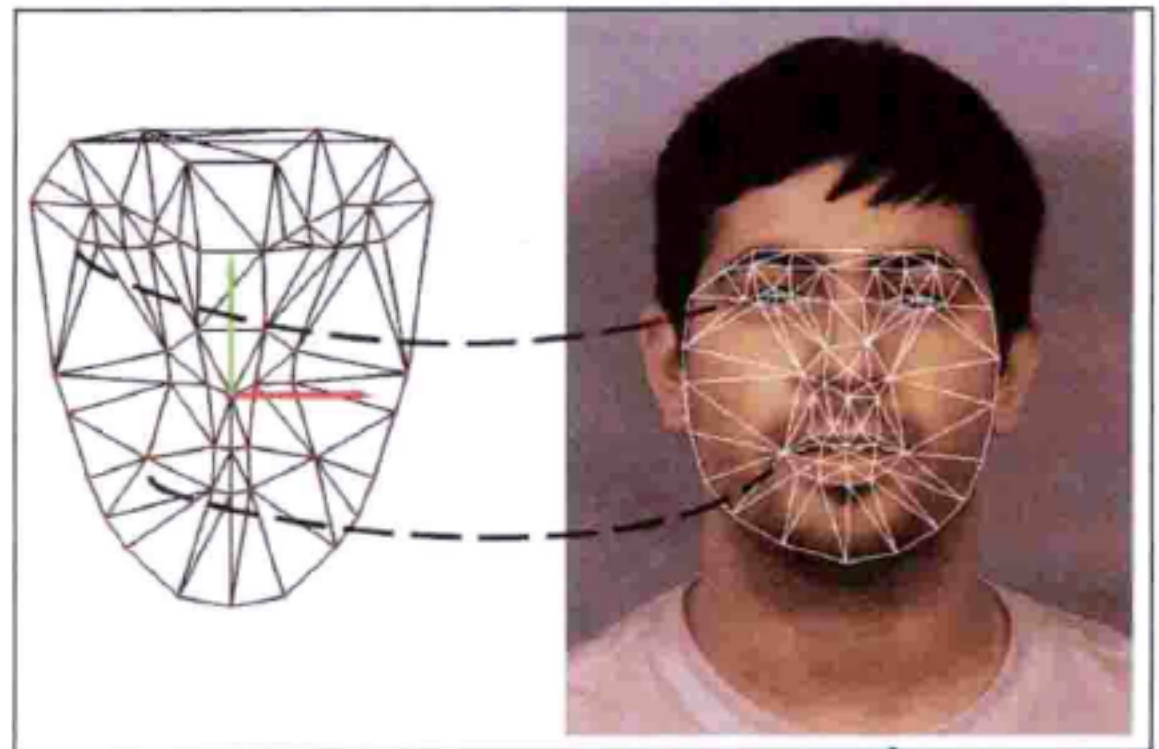
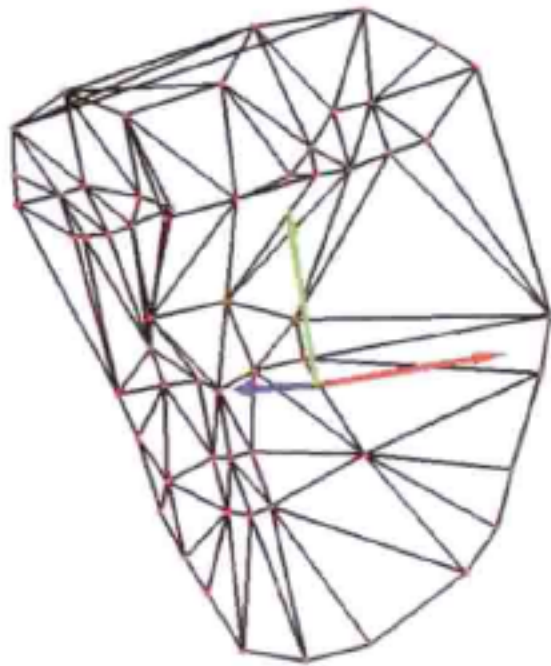
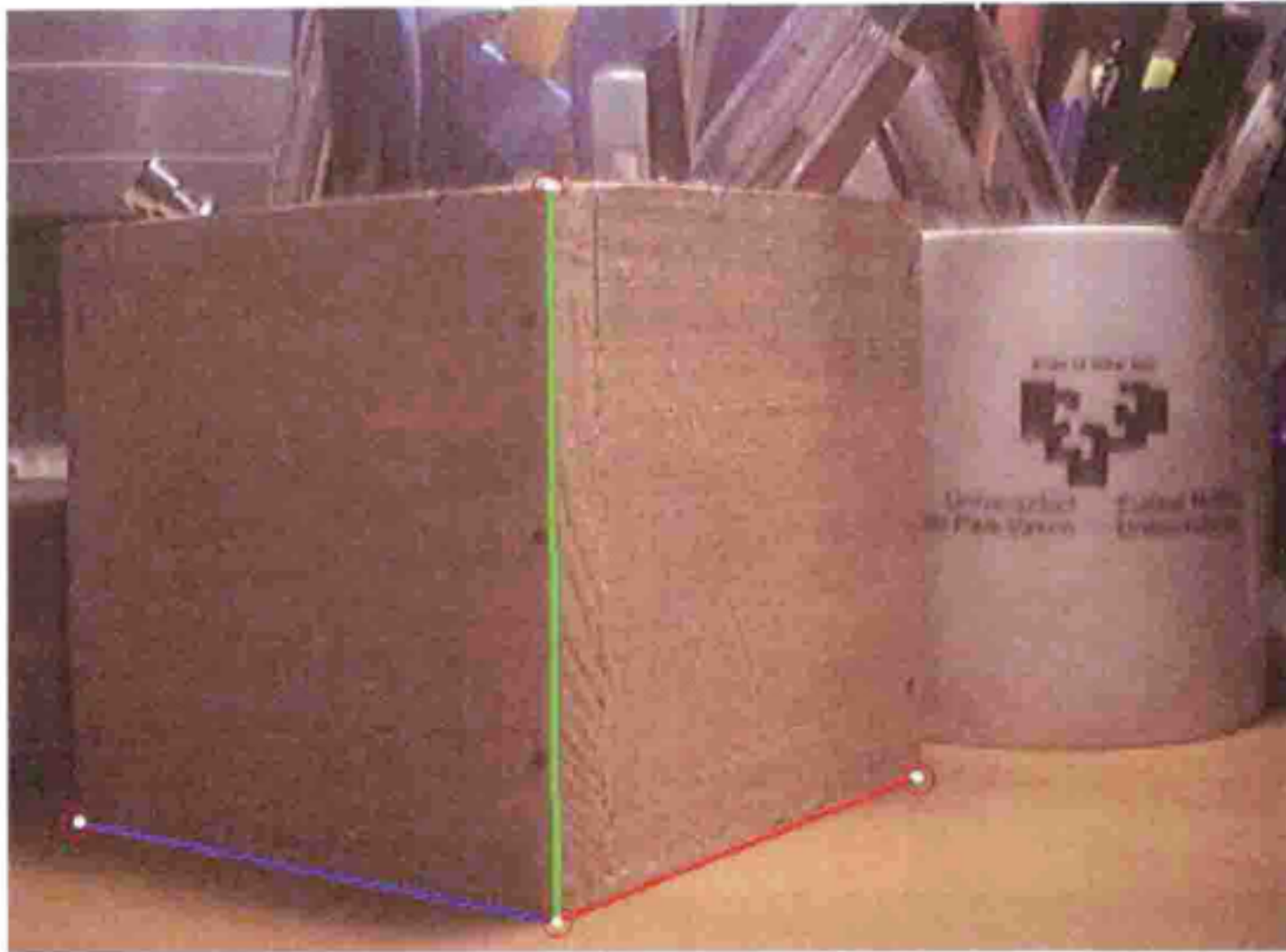
平移对齐



Procrustes 对齐









## *The Translator's Words* 译者序

视觉是人类获取信息的主要来源。图像、视频等视觉信息载体也是当今大数据时代最大的数据源之一，在计算机工程、通信、生物学、医学、军事等领域有着广泛应用。由于计算机视觉涉及多个领域的专业知识，以及视觉对象的复杂性和视觉任务的多样，这使计算机视觉研究很困难。

OpenCV 是开源、跨平台的计算机视觉库，其全称是 Open Source Computer Vision Library。它是由英特尔公司发起并参与开发的，可在商业和研究领域中免费使用。OpenCV 能开发实时的图像处理、运动跟踪、目标检测等程序。

但目前通过实际应用项目来介绍 OpenCV 的书很少。本书通过 8 个典型的计算机视觉项目来介绍 OpenCV 强大、高效的功能。这 8 个项目涵盖了计算视觉的如下领域：基于 iPhone 或 iPad 的增强现实；从运动中得到 3D 结构；车牌识别；人脸识别与跟踪；三维头部姿态估计等。这些项目均用 C/C++ 实现，对于关键代码，作者给出非常详细的介绍。在每章中，作者不但介绍项目的应用背景、整体框架、软件设计方法，同时也深入浅出地介绍了与项目相关的机器学习理论。毫不夸张地讲，这是一本用 OpenCV 来实践计算机视觉应用难得的好书。

翻译本书的过程也是我学习的过程，虽然辛苦但也不觉得累。为了做到专业词汇准确权威，书本内容正确，意译部分既不失原著意境又无偏差，在翻译过程中查阅了大量相关资料。但由于时间和精力有限，书中内容难免存在纰漏。若有问题，读者可通过电子邮件 [liubo7971@173.com](mailto:liubo7971@173.com) 与我联系，欢迎一起探讨，共同进步。

本书翻译过程得到如下项目资助：（1）重庆市教委研究项目“多核正则化机器学习理

论研究”，项目号：KJ130709；（2）重庆工商大学研究项目“基于多核学习的高维数据分析研究”，项目号：2013-56-09；（3）大数据稀疏表示判别字典学习及其应用技术研究项目号：KJ1400612。

感谢河南工业大学信息科学与工程学院的靳小波博士对本书翻译的支持与鼓励，也感谢我的家人，特别感谢我妻子杨雪莉和女儿刘典。虽然翻译本书占用了本应陪她们的大量时间，但她们一直包容并支持我。

本书包含 9 章，每章都用一个完整项目作为教程，并提供全部源代码，这些源代码包含了用 C++ 实现的 OpenCV 接口。每章都出自作者在 OpenCV 社区对某一主题所做出的令人瞩目的贡献，OpenCV 的主要开发者也审阅了本书。本书没有解释 OpenCV 函数的基本功能，而是第一本介绍如何使用 OpenCV 来解决整个问题的书，其中包括几个 3D 摄像机项目（增强现实、从运动中恢复 3D 结构、Kinect 交互）和几个面部表情分析项目（例如：皮肤检测、简单的面部和眼部检测、复杂的面部特征跟踪、三维头部姿势估计和人脸识别）。因此，本书能很好地与现有 OpenCV 书籍配合使用。

## 本书的主要内容

第 1 章包含一个针对桌面应用和 Android 应用的完整教程及相关源代码，这些应用可从真实摄像机图像中自动生成一幅卡通画或图画。在生成过程中，包括皮肤颜色变换在内的几种卡通类型可供选择。

第 2 章包含一个完整教程，该教程讲解如何针对 iPhone 或 iPad 设备来构建基于标记的增强现实 (AR) 应用，并给出每个步骤和源代码的解释。

第 3 章包含一个怎样开发无标记增强现实桌面应用的完整教程，并解释了无标记增强现实 (AR) 和其源代码。

第 4 章通过 OpenCV 实现运动中结构恢复的概念来介绍运动中的结构 (SfM)。读者将学习如何从 2D 图像重构 3D 几何结构以及如何估计摄像机位置。

第 5 章包含一个完整教程及相关源代码，该教程是通过模式识别算法（支持向量机和人工



神经网络)而建立的自动车牌识别应用。读者将学习如何训练和预测模式识别算法来判断一幅图像是否为车牌。这对通过一组特征来识别字符也有帮助。

第 6 章包含构建一个动态人脸跟踪系统的完整教程及相关源代码,该系统能模拟和跟踪人脸的一些复杂部位。

第 7 章包含理解主动外观模型(AAM)和通过 OpenCV 来根据有不同脸部表情的数据帧创建 AAM 的所有背景知识。除此以外,该章解释如何根据 AAM 提供的拟合能力来匹配给定帧。然后采用 POSIT 算法来找到 3D 头部姿态。

第 8 章包含实时人脸识别应用的完整教程和源代码,该应用包括基本的脸部和眼部检测算法,能处理图像中的人脸旋转和不同光照条件。

第 9 章包含一个交互式流体模拟器(称为流体墙)的完整开发流程,它采用 Kinect 传感器。该章将解释怎样通过 OpenCV 的光学流方法来使用 Kinect 数据并将其集成到一个流体求解算法中。请读者通过链接 [http://www.packtpub.com/sites/default/files/downloads/7829OS\\_Chapter9\\_Developing\\_Fluid\\_Wall\\_Using\\_the\\_Microsoft\\_Kinect.pdf](http://www.packtpub.com/sites/default/files/downloads/7829OS_Chapter9_Developing_Fluid_Wall_Using_the_Microsoft_Kinect.pdf) 下载第 9 章。

## 阅读前的准备工作

阅读本书不需要具有计算机视觉的专业知识,但在阅读本书之前应该有良好的 C/C++ 编程技能和 OpenCV 的基本经验。没有 OpenCV 经验的读者不妨阅读《Learning OpenCV》来了解 OpenCV 的特性或阅读《OpenCV 2 Cookbook》来了解如何以受推崇的 C++ 方式来使用 OpenCV,因为本书将展示如何解决现实问题,并假定读者熟悉 OpenCV 和 C/C++ 开发的基础知识。

除具有 C/C++ 和 OpenCV 的经验外,读者还需要一台计算机和相应的 IDE 环境(例如:Visual Studio、XCode、Eclipse、QtCreator,它们可以运行在 Windows、Mac 或者 Linux 上)。有些章节有进一步的要求,特别是:

- ❑ 为了开发 Android 应用,读者需要一台 Android 设备、多种 Android 开发工具和基本的 Android 开发经验。
- ❑ 为了开发 iOS 应用,读者需要 iPhone、iPad 或 iPod Touch 设备、iOS 开发工具(包含一台苹果电脑、XCode IDE、苹果开发者证书)以及基本的 iOS 和 Objective-C 的开发

经验。

- 几个桌面项目需要一台与计算机相连的摄像机。任何普通的 USB 摄像机就足够了，但它至少是 100 万像素。
- 某些项目（包括 OpenCV 本身）会使用 CMake 来构建以支持跨平台和跨编译器。需要对构建系统有基本的理解，最好具有跨平台构建的知识。
- 需要了解线性代数方面的知识，例如：向量和矩阵的基本操作以及特征分解。

## 本书的读者对象

本书对想用基本的 OpenCV 知识来创建实际的计算机视觉项目的开发者来说是一本绝佳指南，此外，对于经验丰富并想获得更多计算机视觉主题的 OpenCV 专家而言也是非常好的一本书。本书向计算机科学相关专业的高年级本科生和研究生，以及研究人员和想用 OpenCV C/C++ 接口来解决实际问题的计算机视觉专家提供循序渐进的实用教程。

## 经典体感应用开发著作



### Kinect应用开发实战：用最自然的方式与机器对话

本书由微软资深企业架构师兼Kinect应用开发专家亲自执笔，既系统全面地讲解了Kinect技术的工作原理，又细致深入地讲解了Kinect交互设计、程序开发和企业应用展望。全书不仅包含大量实践指导意义极强的实战案例，而且还包括大量建议和最佳实践，是学习Kinect for Windows应用开发不可多得的参考书。

本书分为八大部分：准备篇（引言和第1章），从科幻电影的自然人机交互技术谈起，同时针对虚拟现实、增强现实、多点触摸、语音识别、眼球跟踪、人脸识别、体感操作、脑机界面等人机交互技术，结合一些生动例子来说明这些技术的最新发展动态；原理篇（第2~3章），深入剖析Kinect的硬件组成，从原理上分析Kinect的工作机制，并从计算机视觉技术角度去重点分析“体感操作”背后发生的一切；基础篇（第4~5章），对Kinect for Windows SDK进行框架性的导读，并对Kinect自然人机交互的设计提出有益的归纳和建议；开发篇（第6~9章），从Kinect的开发环境准备，到视频数据、深度数据、骨骼跟踪等开发示例，其中包括一个用Kinect测量身高的有趣示例；实例篇（第10~16章），通过一些生动有趣的应用实例（超级玛丽、水果忍者等）开发，帮助读者快速开发入门；进阶篇（第17~19章），包括姿态识别和手势识别的算法实现，Kinect技术结合3D技术的应用，同时结合Kinect在手术室的原型应用这一综合示例，将交互设计、骨骼跟踪、手势识别、语音识别等关键点“串烧”起来；展望篇（第20~22章），汇集Kinect应用的相关创意和奇思妙想，以及Kinect在医疗、教育、动作捕捉、虚拟现实、增强现实、动漫设计乃至冰川研究等诸多领域的发展前景；附录A是关于Kinect SDK命名空间Microsoft.Kinect的详细介绍，附录B是关于自然人机交互技术、计算机视觉技术的相关开源社区动态。

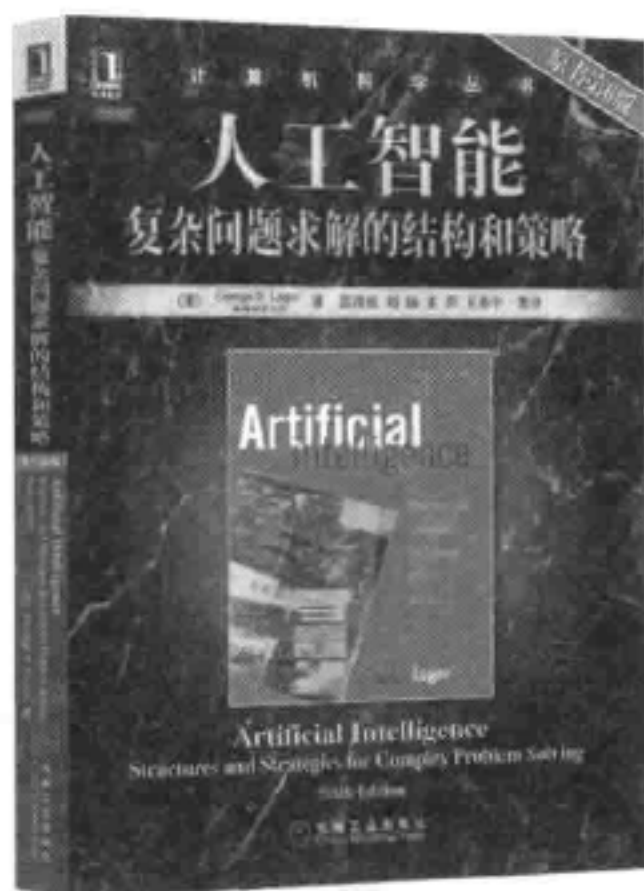
### OpenNI体感应用开发实战

本书是国内首本关于OpenNI的实战性著作，也是首本基于Xtion设备的体感应用开发类著作。具有权威性，由国内体感应用开发领域的专家撰写，华硕官方和CNkinect社区提供支持；具有针对性，深入调研OpenNI社区开发者的需求，据此对内容进行编排；全面且系统，讲解了Xtion和OpenNI的功能使用、技术细节和工作原理，以及体感应用开发的各种知识和技巧；实战性强，包含多个有趣的综合性案例，详细分析和讲解案例的实现过程，确保读者通过本书掌握体感应用开发的技术和方法是本书的宗旨。

全书共19章，分为五个部分：基础篇（1~3章）介绍了自然人机交互技术、Xtion硬件设备的功能和原理、OpenNI的功能和应用；准备篇（4~6章）讲解了如何搭建OpenNI+Xtion的体感应用开发环境，以及OpenNI的一些基本功能；进阶篇（7~13章）详细讲解了人体骨骼追踪、手势识别、手部追踪、录制与重播、生产节点的建立、声音数据的获取和使用、彩色图像数据的获取和贴图等OpenNI的重要功能及其应用方法；实战篇（14~17章）详细讲解了4个有趣且具有代表性的案例，通过这部分内容读者将能掌握体感应用开发的流程与方法；高级篇（18~19章）讲解了体感应用开发中会用到的多种高级功能，如运动捕捉和OpenNI Unity工具包等。



## 推荐阅读



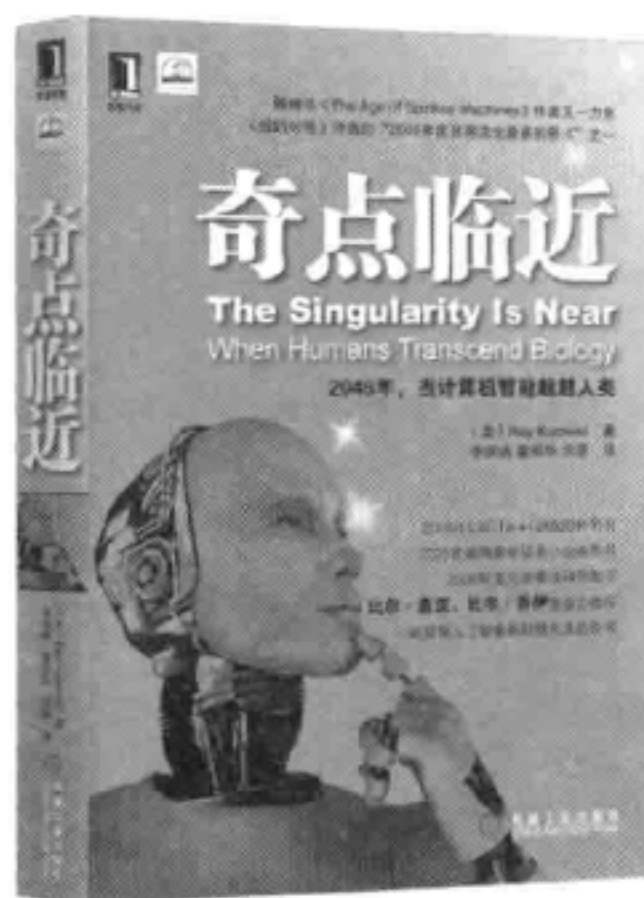
**人工智能：复杂问题求解的结构和策略（原书第6版）**

作者：George F. Luger ISBN: 978-7-111-28345-4 定价：79.00元



**人工智能：智能系统指南（原书第3版）**

作者：Michael Negnevitsky ISBN: 978-7-111-38455-7 定价：79.00元



**奇点临近**

作者：Ray Kurzweil ISBN: 978-7-111-35889-3 定价：69.00元



**机器学习**

作者：Tom Mitchell ISBN: 978-7-111-10993-7 定价：35.00元

# 目 录 *Contents*

## 译者序 前 言

## 第1章 Android系统上的卡通化

### 和皮肤变换 ..... 1

#### 1.1 访问摄像机 ..... 2

#### 1.2 桌面应用处理摄像机 视频的主循环 ..... 3

#### 1.3 生成黑白素描 ..... 4

#### 1.4 生成彩色图像和卡通 ..... 5

#### 1.5 用边缘滤波器来生成 “怪物”模式 ..... 7

#### 1.6 用皮肤检测来生成 “外星人”造型 ..... 8

##### 1.6.1 皮肤检测算法 ..... 8

##### 1.6.2 确定用户放置脸的位置 ..... 9

##### 1.6.3 皮肤变色器的实现 ..... 10

#### 1.7 把桌面应用移植到 Android 系统 ..... 14

##### 1.7.1 安装使用 OpenCV 的 Android 项目 ..... 14

##### 1.7.2 在 Android NDK 应用中 添加卡通化代码 ..... 17

##### 1.7.3 在 Android 系统中显示 保存图像的消息 ..... 24

##### 1.7.4 降低素描图像的随机 椒盐噪声 ..... 27

#### 1.8 总结 ..... 31

## 第2章 iPhone或iPad上基于 标记的增强现实 ..... 32

#### 2.1 使用 OpenCV 创建 iOS 项目 ..... 33

##### 2.1.1 添加 OpenCV 框架 ..... 34

##### 2.1.2 包含 OpenCV 头文件 ..... 35

#### 2.2 应用程序的结构 ..... 36

#### 2.3 标记检测 ..... 43

##### 2.3.1 标记识别 ..... 44

##### 2.3.2 标记编码识别 ..... 50

#### 2.4 在三维空间放置标记 ..... 53

##### 2.4.1 摄像机标定 ..... 53

##### 2.4.2 标记姿态估计 ..... 54

#### 2.5 渲染 3D 虚拟物体 ..... 56

##### 2.5.1 创建 OpenGL 渲染层 ..... 56

2.5.2 渲染 AR 场景 .....	59	第4章 使用OpenCV研究从运动中恢复结构 .....	92
2.6 总结 .....	64	4.1 从运动中恢复结构的概念 .....	93
2.7 参考文献 .....	64	4.2 从两幅图像估计摄像机运动 .....	94
<b>第3章 无标记的增加现实 .....</b>	<b>65</b>	4.2.1 通过丰富的特征描述符进行点匹配 .....	94
3.1 基于标记的 AR 与无标记的 AR .....	65	4.2.2 通过光流进行点匹配 .....	96
3.2 使用特征描述符检测视频中的任意图像 .....	66	4.2.3 搜索摄像机矩阵 .....	99
3.2.1 特征提取 .....	67	4.3 重构场景 .....	102
3.2.2 模式对象定义 .....	69	4.4 从多视图中重构 .....	105
3.2.3 特征点匹配 .....	69	4.5 重构的细化 .....	108
3.2.4 删除离群值 .....	70	4.6 用 PCL 来可视化 3D 点云 .....	111
3.2.5 将示例项目各部分放在一起 .....	76	4.7 使用示例代码 .....	113
3.3 模式姿态估计 .....	77	4.8 总结 .....	114
3.3.1 PatternDetector.cpp .....	77	4.9 参考文献 .....	115
3.3.2 获取摄像机内矩阵 .....	78	<b>第5章 基于SVM和神经网络的车牌识别 .....</b>	<b>116</b>
3.4 应用的基础架构 .....	81	5.1 ANPR 简介 .....	116
3.4.1 ARPipeline.hpp .....	82	5.2 ANPR 算法 .....	118
3.4.2 ARPipeline.cpp .....	82	5.3 车牌检测 .....	119
3.4.3 在 OpenCV 中启用三维可视化支持 .....	83	5.3.1 图像分割 .....	120
3.4.4 使用 OpenCV 来创建 OpenGL 窗口 .....	84	5.3.2 分类 .....	125
3.4.5 使用 OpenCV 捕获视频 .....	85	5.4 车牌号识别 .....	127
3.4.6 渲染增强现实 .....	85	5.4.1 OCR 分割 .....	127
3.4.7 演示应用程序 .....	88	5.4.2 特征提取 .....	129
3.5 总结 .....	91	5.4.3 OCR 分类 .....	130
3.6 参考文献 .....	91	5.4.4 评价 .....	133
		5.5 总结 .....	136

## 第6章 非刚性人脸跟踪 ..... 137

- 6.1 概述 ..... 138
- 6.2 实用工具 ..... 139
  - 6.2.1 面向对象设计 ..... 139
  - 6.2.2 数据收集：图像和视频标注 ..... 140
- 6.3 几何约束 ..... 145
  - 6.3.1 Procrustes 分析 ..... 146
  - 6.3.2 线性形状模型 ..... 148
  - 6.3.3 局部 - 全局相结合  
的表示 ..... 150
  - 6.3.4 训练与可视化 ..... 152
- 6.4 面部特征检测器 ..... 154
  - 6.4.1 相关性块模型 ..... 155
  - 6.4.2 解释全局几何变换 ..... 159
  - 6.4.3 训练与可视化 ..... 161
- 6.5 人脸检测与初始化 ..... 163
- 6.6 人脸跟踪 ..... 166
  - 6.6.1 人脸跟踪实现 ..... 166
  - 6.6.2 训练与可视化 ..... 168
  - 6.6.3 通用与专用人脸模型 ..... 168
- 6.7 总结 ..... 169
- 6.8 参考文献 ..... 169

## 第7章 基于AAM和POSIT的 三维头部姿态估计 ..... 170

- 7.1 主动外观模型概述 ..... 171
- 7.2 主动形状模型概述 ..... 172
  - 7.2.1 感受 PCA ..... 174

7.2.2 三角剖分 ..... 177

7.2.3 扭曲三角化结构 ..... 179

7.3 模型实例化——试试主动  
外观模型 ..... 180

7.4 主动外观模型搜索和拟合 ..... 181

7.5 POSIT 算法 ..... 182

7.5.1 深入理解 POSIT  
算法 ..... 183

7.5.2 POSIT 与头部模型 ..... 185

7.5.3 对摄像机或视频文件  
进行跟踪 ..... 185

7.6 总结 ..... 187

7.7 参考文献 ..... 187

## 第8章 基于特征脸或Fisher 脸的人脸识别 ..... 189

8.1 人脸识别与人脸检测介绍 ..... 189

8.1.1 第一步：人脸检测 ..... 191

8.1.2 检测人脸 ..... 194

8.1.3 第2步：人脸预处理 ..... 196

8.1.4 第3步：收集并  
训练人脸 ..... 204

8.1.5 第4步：人脸识别 ..... 212

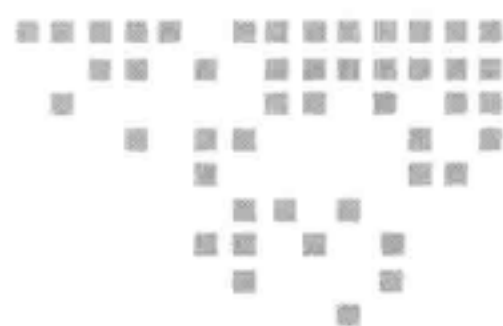
8.1.6 收尾工作：保存和  
加载文件 ..... 215

8.1.7 收尾工作：制作一个  
漂亮的交互式 GUI ..... 215

8.2 总结 ..... 225

8.3 参考文献 ..... 225





## Android 系统上的卡通化和皮肤变换

本章将介绍如何针对 Android 智能手机和平板电脑编写图像处理滤波器，首先在台式机上（用 C/C++）实现，然后移植到 Android 系统上（用 C/C++ 实现的代码与台式机一样，但 GUI 用 Java 来编写），这是移动设备开发所推崇的方式。本章的主要内容如下：

- 如何将现实生活中的图像转换为素描；
- 如何生成彩色图画并将素描叠加上去来生成卡通画；
- 用恐怖的“怪物”模式来创建坏人形象；
- 通过基本的皮肤检测器和皮肤变色器让人脸变成绿色“外星人”皮肤；
- 如何将桌面应用项目转换为移动设备上的应用程序。

下面的屏幕截图来自于运行在 Android 平板电脑上的卡通化应用程序。



本章希望摄像机拍摄的现实世界看起来像卡通画一样。其基本思路是用某种颜色来填充平整部分，然后用粗线来绘制图像较明显的边缘。也就是说，平整区域变得更加平，而

边缘应变得更加明显。可先检测边缘并对平整区域进行平滑处理，然后增加边缘并从顶部开始来产生一个卡通或漫画效果。

当开发移动设备上的计算机视觉应用时，一种好的方法是先创建一个完整的桌面应用版本，再移植到移动设备上，因为开发和调试桌面应用程序比移动应用程序要容易！因此，本章将以一个完整的卡通化桌面应用开始，读者可用自己喜欢的 IDE（如：Visual Studio、XCode、Eclipse、QtCreator 等）来编写该应用。当其在 PC 上正确运行后，我们将在最后一节介绍如何用 Eclipse 将其移植到 Android（或 iOS）系统中。因此会创建两个不同的项目，它们会共享绝大多数源代码，但有不同的图形用户界面。可创建两个项目都可使用的库，但为了简化起见，可将桌面项目和 Android 项目放在一起，让 Android 项目通过 desktop 文件夹来访问一些文件（cartoon.cpp 和 cartoon.h，它们包含了所有图像处理代码）。例如：

- ❑ C:\Cartoonifier\_Desktop\cartoon.cpp
- ❑ C:\Cartoonifier\_Desktop\cartoon.h
- ❑ C:\Cartoonifier\_Desktop\main\_desktop.cpp
- ❑ C:\Cartoonifier\_Android\...

桌面应用有一个 OpenCV GUI 窗口，初始化摄像机，并在处理摄像机的每帧时都调用 `cartoonifyImage()` 函数，该函数包含了本章大多数代码。然后在 GUI 窗口显示被处理的图像。与之类似，Android 应用程序也有一个 Android GUI 窗口，会用 Java 程序来初始化摄像机，并调用前面提到的那个用 C++ 实现的 `cartoonifyImage()` 函数来处理摄像机的每一帧，除此之外还有 Android 菜单并支持触摸输入。本章将从头开始介绍如何创建桌面应用程序，该 Android 应用基于其中的一个 OpenCV Android 示例项目。因此，首先在读者所熟悉的 IDE 中创建桌面应用程序，其中，用于保存 GUI 代码的 `main_desktop.cpp` 文件将在下一节给出，该文件包含主循环、摄像机功能以及键盘输入，同时，还应创建两个项目共享的 `cartoon.cpp` 文件。本章大多数代码都会放在函数 `cartoonifyImage()` 中，该函数保存在 `cartoon.cpp` 文件中。

## 1.1 访问摄像机

可简单调用 `cv::VideoCapture` 对象的 `open()` 方法（它是访问摄像设备的 OpenCV 方法）来访问计算机的摄像头或摄像机。将默认的摄像机编号 0 传递给此函数。一些计算机有多个摄像机或将 0 作为默认摄像机编号使程序不能运行，解决这类问题的通常做法是将用户指定摄像机编号作为命令行参数，比如：若想指定摄像机编号为 1、2 或 -1，这种方法就比较恰当。

为了让程序在高分辨率摄像机上运行得更快，可用 `cv::VideoCapture::set()` 将摄像机的分辨率设为  $640 \times 480$ 。



**注意：**由于摄像机的模式、驱动，或操作系统不同，OpenCV 可能无法改变某些摄像机属性，这对本项目不重要，因此摄像机的属性无法修改也不要担心。

将下面的代码放到 `main_desktop.cpp` 的 `main()` 函数中：

```
int cameraNumber = 0;
if (argc > 1)
    cameraNumber = atoi(argv[1]);

// Get access to the camera.
cv::VideoCapture camera;
camera.open(cameraNumber);
if (!camera.isOpened()) {
    std::cerr << "ERROR: Could not access the camera or video!" <<
    std::endl;
    exit(1);
}

// Try to set the camera resolution.
camera.set(cv::CV_CAP_PROP_FRAME_WIDTH, 640);
camera.set(cv::CV_CAP_PROP_FRAME_HEIGHT, 480);
```

在摄像机被初始化后，可通过 `cv::Mat` 对象（它是 OpenCV 的图像容器）来获取摄像机的图像。可通过使用 C++ 流运算符将 `cv::VideoCapture` 对象转换成 `cv::Mat` 对象，由此可获取摄像机的每帧，这就像从控制台获取输入一样。



**注意：**使用 OpenCV 加载视频文件（例如：AVI 或 MPG 文件）非常容易，这样可代替摄像机。加载视频文件与直接从摄像机获得视频的不同之处在于创建 `cv::VideoCapture` 对象时，应将视频文件名（例如：`camera.open("my_video.avi")`）而不是摄像机编号（例如：`camera.open(0)`）作为参数。这两种方法创建的 `cv::VideoCapture` 对象，使用方式都一样。

## 1.2 桌面应用处理摄像机视频的主循环

如果用 OpenCV 在屏幕上显示一个 GUI 窗口，可调用 `cv::imshow()` 方法，但每显示一帧图像后必须调用 `cv::waitKey(0)` 方法，否则 GUI 窗口的图像根本不会更新！`cv::waitKey(0)` 会使程序暂停除非用户按下任意一个键，若将该函数的参数设为一个正数，例如：`waitKey(20)` 或更大的值，则程序将会暂停一段时间，这段时间长度为该正数值个毫秒单位。

将这个主循环放到 `main_desktop.cpp` 中作为实时摄像机应用程序的基础：



```

while (true) {
    // Grab the next camera frame.
    cv::Mat cameraFrame;
    camera >> cameraFrame;
    if (cameraFrame.empty()) {
        std::cerr << "ERROR: Couldn't grab a camera frame." <<
        std::endl;
        exit(1);
    }
    // Create a blank output image, that we will draw onto.
    cv::Mat displayedFrame(cameraFrame.size(), cv::CV_8UC3);

    // Run the cartoonifier filter on the camera frame.
    cartoonifyImage(cameraFrame, displayedFrame);

    // Display the processed image onto the screen.
    imshow("Cartoonifier", displayedFrame);

    // IMPORTANT: Wait for at least 20 milliseconds,
    // so that the image can be displayed on the screen!
    // Also checks if a key was pressed in the GUI window.
    // Note that it should be a "char" to support Linux.
    char keypress = cv::waitKey(20); // Need this to see anything!
    if (keypress == 27) { // Escape Key

        // Quit the program!
        break;
    }
} //end while

```

### 1.3 生成黑白素描

为了获得视频帧的一幅素描（黑白图画）效果，可用边缘检测滤波器；若要获得一幅彩色图画，可使用边缘保持滤波器（双边滤波器）来进一步平滑平坦区域，同时保持边缘完好。将素描叠加到彩色图画上，便可得到一种卡通效果，该效果与前面最终应用程序的屏幕截图相同。

有许多边缘检测滤波器，例如：Sobel、Scharr、Laplacian 滤波器或 Canny 边缘检测。本章将使用 Laplacian 边缘滤波器，因为同 Sobel 或 Scharr 滤波器相比，它所提取的边缘最接近手工素描，并且它与 Canny 边缘检测一样，可得到清晰的素描效果，但 Canny 边缘检测更容易受视频帧中随机噪声影响，从而使得素描边缘在不同帧之间经常有剧烈变化。

尽管如此，在使用 Laplacian 边缘滤波器之前仍需对图像去噪。可使用中值滤波器来去噪，因为它有很好的去噪效果，同时还可让边缘锐化；而且比双边滤波器的效率高。由于 Laplacian 边缘滤波器只能处理灰度图像，因此必须将 OpenCV 默认的 BGR 格式转换成灰度格式。将下面的代码放在空文件 cartoon.h 的上方，这样使得在访问 OpenCV 和标准 C++

模板时不需要处处都加前缀 cv: 和 std:。

```
// Include OpenCV's C++ Interface
#include "opencv2/opencv.hpp"
```

```
using namespace cv;
using namespace std;
```

将下面的代码以及所有剩下的代码都放到 `cartoonifyImage()` 函数中, 该 `cartoonifyImage()` 函数保存在 `cartoon.cpp` 文件中。

```
Mat gray;
cvtColor(srcColor, gray, CV_BGR2GRAY);
const int MEDIAN_BLUR_FILTER_SIZE = 7;
medianBlur(gray, gray, MEDIAN_BLUR_FILTER_SIZE);
Mat edges;
const int LAPLACIAN_FILTER_SIZE = 5;
Laplacian(gray, edges, CV_8U, LAPLACIAN_FILTER_SIZE);
```

Laplacian 边缘滤波器能生成不同亮度的边缘, 为了使边缘看上去更像素描, 可采用二值化阈值来使边缘只有白色或黑色。

```
Mat mask;
const int EDGES_THRESHOLD = 80;
threshold(edges, mask, EDGES_THRESHOLD, 255, THRESH_BINARY_INV);
```

下面这幅图的左边是原图, 而右边那幅图是生成的边缘掩码 (edge mask), 它与素描很像。在生成彩色图画后 (稍后介绍), 将边缘掩码放到彩色图像上。

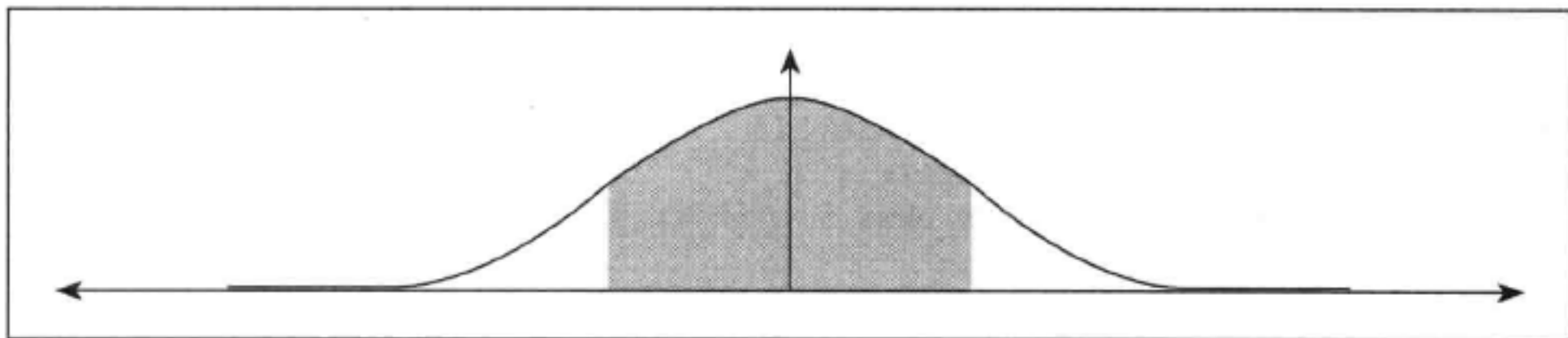


## 1.4 生成彩色图像和卡通

强大的双边滤波器可平滑平坦区域, 同时保持边缘锐化, 因此, 它作为一个自动的卡通化或图画滤波器很不错, 其缺点是效率低 (即该滤波器运行的时间要按秒, 甚至要按分钟而不是毫秒来计算)。因此, 本书采用一些技巧使用户在获得一幅漂亮的卡通化图像时, 也可接受其运行时间。最重要的技巧就是在低分辨率下使用双边滤波器, 这会得到与高分辨率下相似的效果, 但运行速度更快。可将分辨率减少为原图像的四分之一 (例如: 将图像的宽和高各减少二分之一)。

```
Size size = srcColor.size();
Size smallSize;
smallSize.width = size.width/2;
smallSize.height = size.height/2;
Mat smallImg = Mat(smallSize, CV_8UC3);
resize(srcColor, smallImg, smallSize, 0,0, INTER_LINEAR);
```

可通过多个小型双边滤波器来代替一个大型双边滤波器，从而在较短时间内得到很好的卡通效果。本书通过截断滤波器（见下图）来代替整个滤波器（例如：若钟形曲线有 21 个像素宽，则整个滤波器大小为  $21 \times 21$ ），截断滤波器是指能达到满意效果的最小滤波器（例如：即便钟形曲线的大小为  $21 \times 21$ ，但仅使用  $9 \times 9$  滤波器就可以达到满意效果）。截断滤波器会使用滤波器的主要部分（下图曲线的灰色区域），而不会浪费时间在小部分（下图曲线的白色区域）上，这样会使滤波器的效率提高几倍。



控制双边滤波器可使用的 4 个参数分别是：色彩强度、位置强度<sup>⊖</sup>、大小、重复计数。bilateralFilter() 函数不能覆盖它的输入值（这称为“就地处理”），因此需要一个临时的 Mat 变量，该变量在一个滤波器中作为输出变量而在另一个滤波器中作为输入变量。

```
Mat tmp = Mat(smallSize, CV_8UC3);
int repetitions = 7; // Repetitions for strong cartoon effect.
for (int i=0; i<repetitions; i++) {
    int ksize = 9; // Filter size. Has a large effect on speed.
    double sigmaColor = 9; // Filter color strength.
    double sigmaSpace = 7; // Spatial strength. Affects speed.
    bilateralFilter(smallImg, tmp, ksize, sigmaColor, sigmaSpace);
    bilateralFilter(tmp, smallImg, ksize, sigmaColor, sigmaSpace);
}
```

注意，该处理过程使用的是缩小后的图像，因此，在处理后需将图像恢复到原来的大小。然后叠加前面得到的边缘掩码。为了将边缘掩码（即素描）叠加到由双边滤波器所产生的图画上（下页第一幅图左边那幅图像），需将目标变量 dst 的所有元素全置为 0（即目标变量对应的图像全置为黑色），然后将源图像（变量 bigImg）的像素复制到变量 dst 中，与“素描”掩码对应的源图像边缘的像素不会被复制。

```
Mat bigImg;
resize(smallImg, bigImg, size, 0,0, INTER_LINEAR);
dst.setTo(0);
bigImg.copyTo(dst, mask);
```

这会得到原图的卡通版，如下右图所示，这就像将素描掩码叠加到彩色图画上。

<sup>⊖</sup> 在空间上影响像素的多少。——译者注



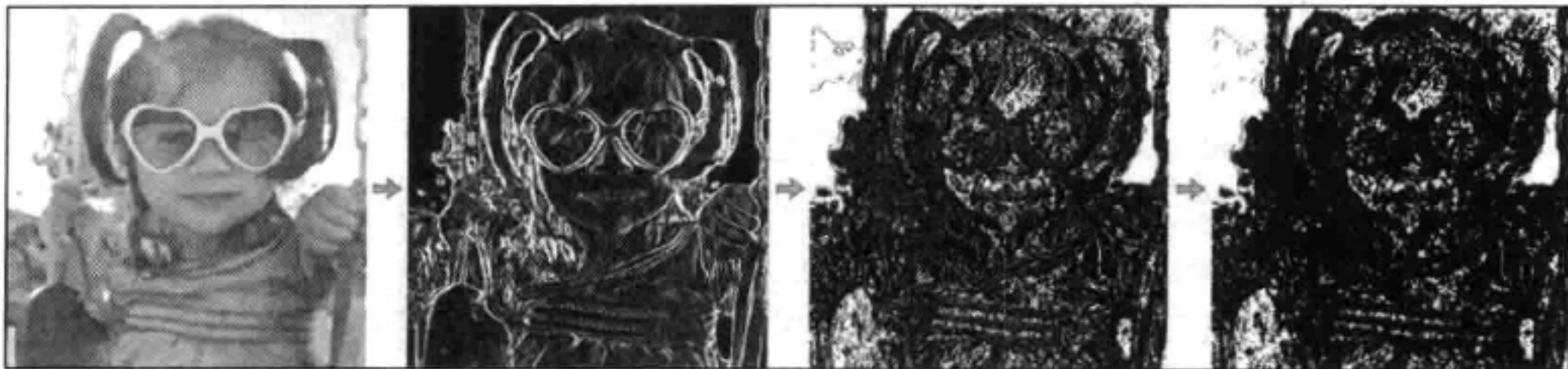


## 1.5 用边缘滤波器来生成“怪物”模式

卡通和漫画总有好人物和坏人物，通过对边缘滤波器的恰当组合，可将和蔼的人物画像变成凶恶的人物画像，其技巧是通过使用小的边缘滤波器，这些滤波器能找到图像的各处边缘，然后通过中值滤波器来合并这些边缘。

在去噪后的灰度图像上执行以上操作，之前将原图像转换成灰度图像的代码和  $7 \times 7$  的中值滤波器都可以用上（下面的第一张图是对灰度图像进行中值平滑滤波后得到的）。接下来不用 Laplacian 滤波器与二值化阈值这种组合方式，而是沿着  $x$  和  $y$  方向采用  $3 \times 3$  的 Scharr 梯度滤波器（见下面的第二张图），然后再采用截断值很低的二值化阈值方法（见下面的第三张图），最后用  $3 \times 3$  的中值平滑滤波就可得到“怪物”掩码（见下面的第四张图）。

```
Mat gray;
cvtColor(srcColor, gray, CV_BGR2GRAY);
const int MEDIAN_BLUR_FILTER_SIZE = 7;
medianBlur(gray, gray, MEDIAN_BLUR_FILTER_SIZE);
Mat edges, edges2;
Scharr(srcGray, edges, CV_8U, 1, 0);
Scharr(srcGray, edges2, CV_8U, 1, 0, -1);
edges += edges2; // Combine the x & y edges together.
const int EVIL_EDGE_THRESHOLD = 12;
threshold(edges, mask, EVIL_EDGE_THRESHOLD, 255, THRESH_BINARY_INV);
medianBlur(mask, mask, 3);
```



将现在得到的“怪物”掩码叠加到彩色卡通图像上，该图像就像是用素描掩码叠加到彩色卡通图像上一样。最终效果如下右图所示。



## 1.6 用皮肤检测来生成“外星人”造型

前面已经生成了素描模式、卡通模式（彩色图画 + 素描掩码）、“怪物”模式（彩色图画 + 怪物掩码），为了更有趣，再来学习一个更复杂的模式：外星人模式。可通过获取人脸的皮肤区域，并将此区域的颜色变为绿色得到该模式。

### 1.6.1 皮肤检测算法

有很多检测皮肤区域的方法，从简单的基于 RGB（Red-Green-Blue）颜色阈值法、HSV（Hue-Saturation-Brightness）值或颜色直方图计算和重投影，到基于混合模型的复杂机器学习算法等，这些复杂的算法需要在 CIE Lab 颜色空间对摄像机标定并且需要用人脸样本数据进行离线训练。即便是复杂的算法，对不同的摄像机、光照条件和皮肤类型也不一定有好的鲁棒性。由于本章所需的皮肤检测要运行在移动设备上且不需要进行标定或训练，而且这里用皮肤检测只是为了做一个“好玩”的图像滤波器，因此在这里会使用简单的皮肤检测算法。但需注意，移动设备上的微型摄像机传感器对颜色的反应往往差异很大，而且要在没有标定的情况下对不同肤色的人进行皮肤检测，这需要算法至少比简单颜色阈值法更具有鲁棒性。

例如：一个简单的 HSV 皮肤检测器就会在图像色调相当红时，饱和度较高（但不是很高），其亮度不太黑或太亮时，会将这些区域的所有像素当成皮肤。除此以外，通常移动设备的摄像机白平衡较差，这使得人的皮肤看上去要偏蓝而不偏红，这也是使用简单 HSV 阈值法的主要问题。

对于这些情况，可采用基于 Haar 或 LBP 级联分类器的人脸识别算法（第 8 章会介绍这些算法），它们具有更好的鲁棒性。在人脸识别的过程中，可检查像素颜色的变化范围，因为人的皮肤像素事先知道，然后对有相似颜色的像素进行全图或邻域扫描，以确定人脸中心。这样做的好处是：不管人的肤色如何，甚至他们的皮肤在图像中偏蓝或偏红，都很有可能找到一些真实皮肤区域。

但是，基于 Haar 或 LBP 级联分类器的人脸识别算法在移动设备上运行缓慢，因此这类

算法可能在实时移动应用中效果不太理想。但对于移动应用来讲，可假定用户能拿着摄像机从近距离直接对准一个人的脸，因为移动设备的摄像机都可拿在用户手上，很方便移动，让用户将人脸放在指定位置并与相机保持一定距离是完全合理的，这样就不需要去检测位置和人脸的大小。这也是一些手机应用会要求用户将脸放在某个位置或用手移动屏幕上的点来确定人脸在图像中的位置的根本原因。因此，可在屏幕中间先画一个人脸轮廓，然后要求用户移动他们的脸到所示位置并通过调节远近来得到恰当的人脸大小。

## 1.6.2 确定用户放置脸的位置

开始外星人模式的第一步是在相机屏幕上画人脸轮廓，以便用户知道将人脸放置在这个位置。以固定的宽高比 0.72 来画一个大椭圆，它占整个图像高度的 70%，这个比率不会使人脸看上去太瘦或太胖。

```
// Draw the color face onto a black background.
Mat faceOutline = Mat::zeros(size, CV_8UC3);
Scalar color = CV_RGB(255,255,0);    // Yellow.
int thickness = 4;
// Use 70% of the screen height as the face height.
int sw = size.width;
int sh = size.height;
int faceH = sh/2 * 70/100; // "faceH" is the radius of the ellipse.
// Scale the width to be the same shape for any screen width.
int faceW = faceH * 72/100;
// Draw the face outline.
ellipse(faceOutline, Point(sw/2, sh/2), Size(faceW, faceH),
        0, 0, 360, color, thickness, CV_AA);
```

为了让人脸看上去更加明显，可再画两个眼睛的轮廓。为了让所画的眼睛看上去更加真实，不要直接用椭圆作为眼睛轮廓，而是用上椭圆作为眼睛的上半部分，用下椭圆作为眼睛的下半部分，可通过指定起始位置和角度来实现：

```
// Draw the eye outlines, as 2 arcs per eye.
int eyeW = faceW * 23/100;
int eyeH = faceH * 11/100;
int eyeX = faceW * 48/100;
int eyeY = faceH * 13/100;
Size eyeSize = Size(eyeW, eyeH);
// Set the angle and shift for the eye half ellipses.
int eyeA = 15; // angle in degrees.
int eyeYshift = 11;
// Draw the top of the right eye.
ellipse(faceOutline, Point(sw/2 - eyeX, sh/2 - eyeY),
        eyeSize, 0, 180+eyeA, 360-eyeA, color, thickness, CV_AA);
// Draw the bottom of the right eye.
ellipse(faceOutline, Point(sw/2 - eyeX, sh/2 - eyeY - eyeYshift),
        eyeSize, 0, 0+eyeA, 180-eyeA, color, thickness, CV_AA);
```



```
// Draw the top of the left eye.
ellipse(faceOutline, Point(sw/2 + eyeX, sh/2 - eyeY),
        eyeSize, 0, 180+eyeA, 360-eyeA, color, thickness, CV_AA);
// Draw the bottom of the left eye.
ellipse(faceOutline, Point(sw/2 + eyeX, sh/2 - eyeY - eyeYshift),
        eyeSize, 0, 0+eyeA, 180-eyeA, color, thickness, CV_AA);
```

可用同样方法来画下嘴唇。

```
// Draw the bottom lip of the mouth.
int mouthY = faceH * 48/100;
int mouthW = faceW * 45/100;
int mouthH = faceH * 6/100;
ellipse(faceOutline, Point(sw/2, sh/2 + mouthY), Size(mouthW,
        mouthH), 0, 0, 180, color, thickness, CV_AA);
```

为了提示用户将脸放在指定位置上,可在屏幕上显示一条提示信息。

```
// Draw anti-aliased text.
int fontFace = FONT_HERSHEY_COMPLEX;
float fontScale = 1.0f;
int fontThickness = 2;
char *szMsg = "Put your face here";
putText(faceOutline, szMsg, Point(sw * 23/100, sh * 10/100),
        fontFace, fontScale, color, fontThickness, CV_AA);
```

现在可将已画好的人脸轮廓叠加到要显示的图像上,叠加过程采用 alpha 融合来将卡通化图像与该轮廓结合在一起。

```
addWeighted(dst, 1.0, faceOutline, 0.7, 0, dst, CV_8UC3);
```

最终得到的人脸轮廓如右图所示。这里并没有去检测人脸位置,因为用户可将人脸放置到这个轮廓中。



### 1.6.3 皮肤变色器的实现

无须去检测皮肤颜色然后看此区域是否有这样的颜色来确定皮肤区域,而是直接使用 OpenCV 的 `floodFill()` 函数。该函数类似于一些图像处理软件中的颜料桶工具。屏幕中间区域就是皮肤像素(因为在拍摄时会人脸放在人脸轮廓里,而人脸轮廓在整幅图像中间)。为了将整个人脸变成绿色皮肤,只需对图像中心位置的像素进行绿色漫水填充。这样做至少会让人脸的某些部分变成绿色。但现实中的颜色,其饱和度和亮度可能会由于人脸不同部位而有所不同,这会使漫水填充不能覆盖面部的所有像素,除非将其阈值设为很低,但这样做又会覆盖人脸以外的像素。为了解决这个问题,可在图像中心区域不采用简单的漫水填充,而是对人脸区域中 6 个不同的皮肤像素点进行漫水填充。

OpenCV 的 `floodFill()` 函数有一个很好的性质:它会将漫水填充的效果存储到外部图像中而不修改输入图像。这一特性可得到一幅掩码图像,该图像可用于调整皮肤像素的颜色而不需要改变亮度和饱和度,也能在所有皮肤像素都为绿色(这会导致很多人脸细节丢

失)的情况下得到更真实的图像。

在 RGB 颜色空间改变皮肤颜色效果并不好。因为改变皮肤颜色需要脸部图像的亮度变化,但皮肤颜色不允许变化太大,而 RGB 无法从色彩中获取亮度。解决该问题的方法之一是采用 HSV 颜色空间,因为它能从色彩(色度)和多彩(饱和度)中获取亮度。但 HSV 的色度取值以红色开始并以红色结束,若皮肤接近红色,就需要处理小于 10% 和大于 90% 的色度值,因为在这两个范围内都是红色。可使用 Y'CrCb 颜色空间(在 OpenCV 中,它是 YUV 的变种)来解决此问题。Y'CrCb 颜色空间不仅能从颜色中获取亮度,而且对于通常的皮肤颜色其取值唯一。实际上对于大多数摄像机,图像和视频在转换成 RGB 前都使用某种 YUV 颜色空间,所以在很多情形下不需要转换就可得到 YUV 格式的图像。

可对图像进行卡通化之后再用外星人滤波器获得像卡通画一样的外星人模式;也就是说,可获取缩小彩色图像和全尺寸的边缘掩码,其中彩色图像由双边滤波器产生。皮肤检测通常在低分辨率下工作较好,因为它与分析高分辨率像素的近邻的平均值等价(也可看作是用低频信号代替高频噪声信号)。接下来的工作是在对图像进行缩放后进行的,其缩放大小与用双边滤波器处理图像时一样(即只取图像一半的宽度和高度)。下面先将彩色图像转换为 YUV 格式:

```
Mat yuv = Mat(smallSize, CV_8UC3);
cvtColor(smallImg, yuv, CV_BGR2YCrCb);
```

另外还需收缩边缘掩码,使其与彩色图像有相同的尺寸。当存储一幅单独的掩码图像时,若用 OpenCV 的 floodFill() 函数会有困难,即掩码图像应在整个图像周围用 1 个像素作边界,若输入图像大小为  $W \times H$  个像素,则单独的掩码图像大小为  $(W+2) \times (H+2)$  个像素。但 floodFill 函数也允许初始化边缘掩码来确保漫水填充算法不会越界,这一特性可阻止漫水填充的区域扩展到人脸外面。因此,需要两幅掩码图像:大小为  $W \times H$  的边缘掩码图像和大小为  $(W+2) \times (H+2)$  的边缘掩码图像,该掩码图像包含了图像的边界。可让多个 cv::Mat 对象(或头部)引用同一数据,甚至可让一个 cv::Mat 对象引用另一个 cv::Mat 图像的某一区域。因此不必分配两个单独的图像,然后复制边缘掩码像素给它们,只需分配一个包含边界的掩码图像并创建一个大小为  $W \times H$  的 cv:Mat 头(仅用它来引用没有边界的掩码图像)。也就是说,仅有一个  $(W+2) \times (H+2)$  大小的像素数组,但有两个 cv:Mat 对象,一个用来指向大小为  $(W+2) \times (H+2)$  的图像,另一个用来指向图像中间大小为  $W \times H$  的区域:

```
int sw = smallSize.width;
int sh = smallSize.height;
Mat mask, maskPlusBorder;
maskPlusBorder = Mat::zeros(sh+2, sw+2, CV_8UC1);
mask = maskPlusBorder(Rect(1,1,sw,sh)); // mask is in maskPlusBorder.
resize(edges, mask, smallSize);          // Put edges in both of them.
```

整个边缘掩码(如下左图所示)既有强边缘也有弱边缘;可用二值化阈值法来得到所

需的强边缘（效果见下中图）。为了在边缘间加入一些间隙，可采用形态学算子 `dilate()` 和 `erode()` 来删除一些间隙（也会涉及“close”算子），效果见下右图：

```
const int EDGES_THRESHOLD = 80;
threshold(mask, mask, EDGES_THRESHOLD, 255, THRESH_BINARY);
dilate(mask, mask, Mat());
erode(mask, mask, Mat());
```



如前所述，需要对人脸的许多像素点使用漫水填充算法，从而保证人脸图像的各种颜色和整个阴影都能被处理。可在鼻子、脸颊和前额选择 6 个点，如下左图所示。注意，这些点的位置依赖于早期所确定的面部轮廓：

```
int const NUM_SKIN_POINTS = 6;
Point skinPts[NUM_SKIN_POINTS];
skinPts[0] = Point(sw/2, sh/2 - sh/6);
skinPts[1] = Point(sw/2 - sw/11, sh/2 - sh/6);
skinPts[2] = Point(sw/2 + sw/11, sh/2 - sh/6);
skinPts[3] = Point(sw/2, sh/2 + sh/16);
skinPts[4] = Point(sw/2 - sw/9, sh/2 + sh/16);
skinPts[5] = Point(sw/2 + sw/9, sh/2 + sh/16);
```

现在仅需为漫水填充算法找到一些好的下界和上界。漫水填充算法基于 Y'CrCb 颜色空间，因此基本上可决定亮度、红色分量和蓝色分量有多大变化。这里需要包括阴影和高亮显示以及反射在内的亮度变化较大，而颜色不需要变化。

```
const int LOWER_Y = 60;
const int UPPER_Y = 80;
const int LOWER_Cr = 25;
const int UPPER_Cr = 15;
const int LOWER_Cb = 20;
const int UPPER_Cb = 15;
Scalar lowerDiff = Scalar(LOWER_Y, LOWER_Cr, LOWER_Cb);
Scalar upperDiff = Scalar(UPPER_Y, UPPER_Cr, UPPER_Cb);
```

调用 `floodFill()` 时，为了存储外部掩码而必须为 `flags` 参数指定 `FLOODFILL_MASK_ONLY` 选项，该参数的其他选项均用默认值。

```
const int CONNECTED_COMPONENTS = 4; // To fill diagonally, use 8.
const int flags = CONNECTED_COMPONENTS | FLOODFILL_FIXED_RANGE \
    | FLOODFILL_MASK_ONLY;
Mat edgeMask = mask.clone(); // Keep a copy of the edge mask.
// "maskPlusBorder" is initialized with edges to block floodFill().
```



```
for (int i=0; i< NUM_SKIN_POINTS; i++) {
    floodFill(yuv, maskPlusBorder, skinPts[i], Scalar(), NULL,
        lowerDiff, upperDiff, flags);
}
```

在下图中，左边这幅图有 6 个漫水填充位置，右边这幅图生成的外部掩码，其中皮肤为灰色，边缘为白色。注意，为了让皮肤像素（其值为 1）清晰可见，本书修改了右边这幅图像（本来皮肤像素为黑色，修改后显示为灰色）。



而图像变量 `mask`（它对应的图像显示在上右图）包含如下值：

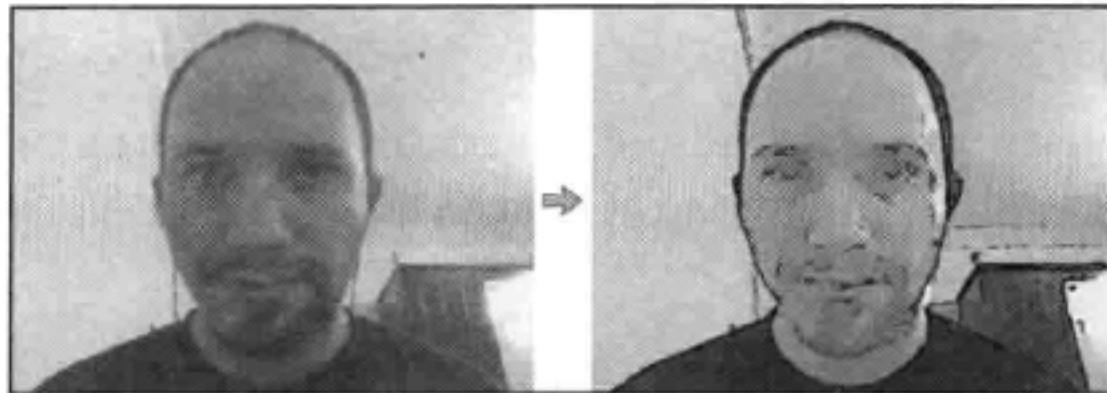
- 值为 255 的边缘像素
- 值为 1 的皮肤像素
- 剩下是值为 0 的像素

而变量 `edgeMask` 仅包含边缘像素（其值为 255）。为了仅得到皮肤像素，可从变量 `mask` 中去掉边缘：

```
mask -= edgeMask;
```

图像变量 `mask` 仅包含值为 1 的皮肤像素和值为 0 的非皮肤像素。为了改变原图像的皮肤颜色和亮度，可用 `cv::add` 来增加原图绿色分量，其区域由皮肤掩码（变量 `mask`）决定。

```
int Red = 0;
int Green = 70;
int Blue = 0;
add(smallImgBGR, CV_RGB(Red, Green, Blue), smallImgBGR, mask);
```



注意，上右图不仅让皮肤变绿，还使皮肤变亮（看起来像在黑暗中发光的外星人）。如果仅改变皮肤颜色而不使其变得更亮，可使用其他颜色变化方法，例如：将绿色分量加 70，而将红色和蓝色分量减 70；或使用 `cvtColor(src, dst, "CV_BGR2HSV_FULL")` 将图像转换成 HSV 颜色空间，然后调整色度和饱和度。

就这样吧！准备将应用移植到移动设备上，使其运行在不同模式下。

## 1.7 把桌面应用移植到 Android 系统

到目前为止，这个程序能在台式机上正常运行，下面要将该程序制作成 Android 或 iOS 应用。本章重点介绍如何生成 Android 应用，在将该程序移植到基于 iOS 的苹果 iPhone 和 iPad 以及其他类似设备上时，此方法仍有效。在开发 Android 应用时，可直接使用 Java 实现的 OpenCV，但程序的效率不如 C/C++ 代码高，而且由于是为移动设备编写的程序，同样的代码不能运行在台式机上。因此建议读者用 C/C++ 来开发大多数 OpenCV+Android。（如果读者想用纯 Java 来开发 OpenCV 应用，可使用 Samuel Audet 提供的 JavaCV 库，该库可在 <http://code.google.com/p/javacv/> 下载，在这个库上用 Java 开发的应用，既可运行在台式机上，也可在 Android 系统中运行。）



**注意：**该 Android 项目会将摄像机视频作为实时输入，因此它不能工作在 Android 模拟器上。本项目需要一台有摄像机，安装有 Android 系统 2.2 (Froyo) 及以后版本的真实设备。

本 Android 应用的用户界面使用 Java 来编写，但图像处理部分，采用 C++ 文件 `cartoon.cpp`，该文件也能用于桌面应用。为了在 Android 应用中可使用 C/C++，必须使用 NDK (Native Development Kit)，它基于 JNI (Java Native Interface)。可对 `cartoonifyImage()` 函数创建一个 JNI 封装，这样该函数能在基于 Java 的 Android 应用使用。

### 1.7.1 安装使用 OpenCV 的 Android 项目

基于 OpenCV 的 Android 接口每年都有很大的变化，比如：访问摄像机的方法，因此，本书并不会详细介绍 Android 应用安装。读者可访问 <http://opencv.org/platforms/android.html> 查找安装和使用 OpenCV 生成本地 (NDK) Android 应用的最新指南。OpenCV 自带了一个称为 `Sample3Native` 的 Android 示例项目，该项目可通过 OpenCV 来访问摄像机，还可在屏幕上显示被处理过的图像。此项目很有用，因为它是开发本章 Android 应用的基础，读者应自己熟悉此项目（项目的安装步骤可访问 [http://docs.opencv.org/doc/tutorials/introduction/android\\_binary\\_package/android\\_binary\\_package\\_using\\_with\\_NDK.html](http://docs.opencv.org/doc/tutorials/introduction/android_binary_package/android_binary_package_using_with_NDK.html)）。接下来将修改这个 Android OpenCV 项目，使该项目能卡通化摄像机的视频帧并将其显示在屏幕上。

如果开发 Android 应用时遇到困难，比如碰到编译错误或摄像机总是显示空白帧，可尝试访问如下网站来解决：

- 1) 前面提到的那个基于 OpenCV 的 Android Binary Package NDK 教程。
- 2) 官方的 Android-OpenCV 谷歌组 (<https://groups.google.com/forum/?fromgroups#!forum/android-opencv>)。
- 3) OpenCV's Q & A 网站 (<http://stackoverflow.com/questions/tagged/opencv+android>)。

- 4) StackOverflow Q & A 网站 (<http://stackoverflow.com/questions/tagged/opencv+android>)。
- 5) 其他的网站 (例如: <http://www.google.com>)。
- 6) 若在访问这些网站后仍无法解决遇到的问题, 可向 Android-OpenCV 谷歌组等提问, 所提问题需要有详细的错误信息。

### 1. Android 系统上图像处理的颜色格式

开发桌面应用时, 仅需处理基于 BGR 的像素格式, 因为输入 (如: 摄像机、图像或视频文件) 是 BGR 格式, 输出 (如: HighGUI 窗口、图像或视频文件) 也是 BGR 格式。但开发移动设备上的应用时, 通常必须自己转换本地颜色格式。

### 2. 从摄像机输入的颜色格式

查看 jni 目录下的示例代码 jni\_part.cpp, 其变量 myuv 指向彩色图像数据, 该图像的颜色格式为 Android 系统默认的摄像机格式 “NV21” YUV420sp。该图像数组的第一部分为灰度像素数组, 接着的像素数组大小只有前面那个数据的一半, 该数组交替排列 U、V 彩色通道 (按 UVUV 交替排列)。若只想访问灰度图像, 直接获取具有 YUV420sp 格式的图像的第一部分即可。但若想得到彩色图像 (例如: BGR 或 BGRA 颜色格式的图像), 必须要用 cvtColor() 函数来转换颜色格式。

### 3. 用于显示的颜色格式

若查看 OpenCV 的 Sample3Native 项目代码, 会发现变量 mbgra 指向一幅彩色图像, 该图像可在 Android 设备上以 BGRA 格式显示。OpenCV 默认格式为 BGR (它与 RGB 的字节顺序相反), 而 BGRA 格式仅在每个像素后面增加一个未被使用的字节, 即每个像素按 Blue-Green-Red-Unused 进行保存。可以按 BGR 完成所有处理, 在屏幕上显示之前, 将其转换为最终输出的 BGRA 格式, 或者确保图像处理程序不仅能处理 BGR 格式, 还可处理 BGRA 格式。OpenCV 的很多函数都支持 BGRA 格式, 从而使这种方式变得简单。必须要确保创建的图像的通道数与通过 Mat::channels() 函数查看到的图像通道数一样。若在代码中直接访问像素, 需要分别处理 3 通道的 BGR 图像和 4 通道的 BGRA 图像。



**注意:** 一些 CV 操作在 BGRA 格式下运行得较快 (因为按 32 位对齐), 而另一些操作在 BGR 格式下运行得较快 (因为读和写需要较少的内存), 因此, 最好的做法是让应用同时支持 BGRA 格式和 BGR 格式, 然后找出在何种颜色格式下运行最快。

下面从一个简单的工作开始: 通过 OpenCV 来获取摄像机的帧, 但只是显示而不处理这些帧。这很容易用 Java 代码做到, 但重要的是如何用 OpenCV 来实现此功能。如前所述, 摄像机图像以 YUV420sp 色彩格式交给 C++ 代码处理后并以 BGRA 色彩格式输出。因此, 需要准备两个 cv::Mat 变量, 一个保存输入的图像数据, 另一个用于保存输出的图像数据, 仅需要用 cvtColor 函数来完成这两种图像数据的颜色格式转换。为了使用 C/C++ 编写基于



Java 的 Android 应用程序的代码，我们需要使用特殊的 JNI 函数名，该函数名要与 Java 类名和包名匹配，具体的格式如下：

```
JNIEXPORT <Return> JNICALL Java_<Package>_<Class>_<Function>(
    JNIEnv* env, jobject, <Args>)
```

下面创建一个 C/C++ 函数 ShowPreview(), 它会用在 Java 包 Cartoonifier 的 CartoonifierView 类中，将此函数添加到 jni\jin\_part.cpp 文件中：

```
// Just show the plain camera image without modifying it.
JNIEXPORT void
JNICALL Java_com_Cartoonifier_CartoonifierView_ShowPreview(
    JNIEnv* env, jobject,
    jint width, jint height, jbyteArray yuv, jintArray bgra)
{
    jbyte* _yuv = env->GetByteArrayElements(yuv, 0);
    jint* _bgra = env->GetIntArrayElements(bgra, 0);

    Mat myuv = Mat(height + height/2, width, CV_8UC1, (uchar*)_yuv);
    Mat mbgra = Mat(height, width, CV_8UC4, (uchar*)_bgra);

    // Convert the color format from the camera's
    // NV21 "YUV420sp" format to an Android BGRA color image.
    cvtColor(myuv, mbgra, CV_YUV420sp2BGRA);

    // OpenCV can now access/modify the BGRA image "mbgra" ...

    env->ReleaseIntArrayElements(bgra, _bgra, 0);
    env->ReleaseByteArrayElements(yuv, _yuv, 0);
}
```

最初看到此代码时会觉得很复杂。函数的前两行访问给定的 Java 数组，接下来的两行定义了两个 cv::Mat 对象，让其指向给定像素缓冲区的数据（即并不会为这两个对象分配内存来保存图像数据，myuv 会指向数组 \_yuv 的数据，mbgra 会指向数组 \_bgra 的数据），最后两行会释放加在 Java 数组上的本地锁。此函数仅将图像由 YUV 格式转换成 BGRA 格式，但它是创建这类新函数的模板。在显示该图像之前，为了分析和修改这个基于 BGRA 格式的 cv::Mat 对象，需扩展此函数。



**注意：**在 OpenCV v2.4.2 中，文件 jni\jni\_part.cpp 有如下代码：

```
cvtColor(myuv, mbgra, CV_YUV420sp2BGR, 4);
```

此代码看上去像是将图像转换为 3 通道的 BGR 格式（OpenCV 的默认格式），但实际上参数 4 表示将图像转换为 4 通道的 BGRA 格式（Android 默认的输出格式）。为了减少这种混乱，可使用下面的等价形式：

```
cvtColor(myuv, mbgra, CV_YUV420sp2BGRA);
```

若不用 OpenCV 默认的 BGR 格式，而用 BGRA 格式的图像作为输入和输出，下面两种方式可实现该要求：

- 在图像处理之前，将 BGRA 转换为 BGR，图像处理过程用 BGR 格式，在 Android 系统显示时，将 BGR 转换为 BGRA。
- 修改所有代码，使其除了能处理 BGR 格式外，还可处理 BGRA 格式，这样做就不会因为在 BGRA 与 BGR 之间转换而降低性能。

为了简单起见，本节仅考虑 BGRA 与 BGR 之间的色彩转换，不会让程序同时支持 BGR 和 BGRA 格式。如果编写一个实时应用程序，应该考虑支持 4 通道的 BGRA 格式以提高其潜在性能。可做一个简单修改来使 BGRA 与 BGR 之间的转换更快一些：原来的转换过程是将 YUV420sp 格式的图像转换为 BGRA 格式，然后再由 BGRA 格式转换为 BGR，修改后只需直接由 YUV420sp 格式转换为 BGR 格式。

用 ShowPreview() 函数（如前所示）来构建应用并在设备上运行是一个不错的主意，因为当后面的 C/C++ 代码有问题时，可由此函数返回修改。为了在 Java 中调用该函数，只需在 CartoonifyImage() 函数的声明附近再增加一个 Java 函数声明，CartoonifyImage() 函数在 CartoonifyView.java 文件的底部：

```
public native void ShowPreview(int width, int height,
    byte[] yuv, int[] rgba);
```

然后就像示例代码（Sample3Native 示例项目的代码）调用 FindFeatures() 函数一样调用此函数。将 ShowPreview() 函数放在 processFrame() 函数的中间，processFrame() 函数在 CartoonifierView.java 文件中。

```
ShowPreview(getFrameWidth(), getFrameHeight(), data, rgba);
```

现在就可以生成并在设备上运行该程序，但只能实时看到摄像机的图像，没有对摄像机的图像进行处理。

### 1.7.2 在 Android NDK 应用中添加卡通化代码

将桌面应用程序的 cartoon.cpp 文件添加到 Android 应用中。文件 jni\Android.mk 保存着 C/C++/ 汇编源代码的文件名、头文件搜索路径、本地库，以及本项目的 GCC 编译器环境设置。

1) 添加 cartoon.cpp（如果想让调试更容易，还需增加 ImageUtils\_0.7.cpp 文件）到变量 LOCAL\_SRC\_FILES 中，这些文件都在 desktop 文件夹而不是默认的 jni 文件夹中。增加之后为：

```
LOCAL_SRC_FILES:= jni_part.cpp:
LOCAL_SRC_FILES += ../../Cartoonifier_Desktop/cartoon.cpp
LOCAL_SRC_FILES += ../../Cartoonifier_Desktop/ImageUtils_0.7.cpp
```

2) 添加头文件搜索路径以便在父目录中找到 cartoon.h。

```
LOCAL_C_INCLUDES += $(LOCAL_PATH)/../../Cartoonifier_Desktop
```

3) 在文件 jni\jni\_part.cpp 的前面插入下面的代码来代替 #include<vector>:

```
<vector>:
#include "cartoon.h"    // Cartoonifier.
#include "ImageUtils.h" // (Optional) OpenCV debugging
                        // functions.
```

4) 添加 JNI CartoonifyImage() 函数到 jni\jni\_part.cpp 文件中, 该函数的作用是卡通化图像。CartoonifyImage() 函数的结构与前面创建的 ShowPreview() 函数一样, 它们的不同之处在于 ShowPreview() 函数只能显示但不处理摄像机图像。注意: 可直接将 YUV420sp 格式转换为 BGR 格式, 因为不需要处理 BGRA 格式的图像。

```
// Modify the camera image using the Cartoonifier filter.
JNIEXPORT void
JNICALL Java_com_Cartoonifier_CartoonifierView_CartoonifyImage(
    JNIEnv* env, jobject,
    jint width, jint height, jbyteArray yuv, jintArray bgra)
{
    // Get native access to the given Java arrays.
    jbyte* _yuv = env->GetByteArrayElements(yuv, 0);
    jint* _bgra = env->GetIntArrayElements(bgra, 0);

    // Create OpenCV wrappers around the input & output data.
    Mat myuv(height + height/2, width, CV_8UC1, (uchar*)_yuv);
    Mat mbgra(height, width, CV_8UC4, (uchar*)_bgra);

    // Convert the color format from the camera's YUV420sp
    // semi-planar
    // format to OpenCV's default BGR color image.
    Mat mbgr(height, width, CV_8UC3); // Allocate a new image
    buffer.
    cvtColor(myuv, mbgr, CV_YUV420sp2BGR);

    // OpenCV can now access/modify the BGR image "mbgr", and should
    // store the output as the BGR image "displayedFrame".
    Mat displayedFrame(mbgr.size(), CV_8UC3);

    // TEMPORARY: Just show the camera image without modifying it.
    displayedFrame = mbgr;

    // Convert the output from OpenCV's BGR to Android's BGRA
    // format.
    cvtColor(displayedFrame, mbgra, CV_BGR2BGRA);

    // Release the native lock we placed on the Java arrays.
```



```

env->ReleaseIntArrayElements(bgra, _bgra, 0);
env->ReleaseByteArrayElements(yuv, _yuv, 0);
}

```

5) 上面的代码不会修改图像, 可使用前面开发的卡通化程序来对图像进行处理。因此, 可调用前面的函数 `cartoonifyImage()` 来处理图像, 该函数在桌面应用的 `cartoon.cpp` 文件中。用下面代码替换掉代码 `displayedFrame = mbgr` 即可:

```

cartoonifyImage(mbgr, displayedFrame);

```

6) 这样就行了! 然后编译这些代码 (Eclipse 应使用 `ndk-build` 来编译这些 C/C++ 代码) 并在设备上运行。读者可看到一个卡通化的 Android 应用 (正确的结果应如本章开始所显示的屏幕截图那样)! 如果编译或运行有问题, 可返回前面的步骤来解决出现的问题 (如有需要可查看本书提供的源代码)。

### 1. Android 应用总结

读者很快就会发现运行在设备上的应用程序有四个问题:

- ❑ 程序运行非常慢, 处理每帧要花好几秒! 解决此问题的方法是: 只在用户触摸到屏幕并觉得这幅照片好时, 才卡通化这一帧, 其他情形只显示摄像机图像。
- ❑ 程序需要处理用户的输入, 例如: 可在素描、彩色图画、怪物模式和外星人模式之间进行切换。
- ❑ 如果能将卡通化结果存储成图像文件与其他人共享会非常好。也就是说, 对于一幅卡通化图像, 当用户触摸屏幕时, 就可将结果保存成图像文件并存放在 SD 卡上, 而且还可在 Android 图库集中显示。
- ❑ 素描的边缘有很多随机噪声。可用特殊的“椒盐”降噪过滤器来解决此问题。

### 2. 通过点击屏幕使图像卡通化

可调用前面的 JNI 函数 `ShowPreview()` 来预览摄像机的视频 (直到用户想卡通化选择的视频帧)。在卡通化视频帧之前, 会一直等待用户触摸屏幕。当用户触摸屏幕时, 只需卡通化一幅图像, 因此, 可设置一个标志来表明接下来的帧要被卡通化, 卡通化完成后再重置这个标志, 于是又会预览摄像机视频。这存在一个问题: 卡通化图像仅能被瞬间显示, 然后又会预览摄像机视频。因此, 需要引入第二个标志来指示当前图像在被摄像机视频覆盖前, 需在屏幕上被冻结一段时间, 这样用户才能看到被卡通化的图像。

1) 将下面的代码添加到 `CartoonifierApp.java` 文件的前面, 该 Java 文件位于 `src\com\Cartoonifier` 文件夹:

```

import android.view.View;
import android.view.View.OnTouchListener;
import android.view.MotionEvent;

```

2) 修改 `CartoonifierApp.java` 文件前面的类定义:

```
public class CartoonifierApp
extends Activity implements OnTouchListener {
```

3) 在 onCreate() 函数的底部插入下面的代码:

```
// Call our "onTouch()" callback function whenever the user
// touches the screen.
mView.setOnTouchListener(this);
```

4) 添加 onTouch() 函数来处理触摸事件:

```
public boolean onTouch(View v, MotionEvent m) {
    // Ignore finger movement event, we just care about when the
    // finger first touches the screen.
    if (m.getAction() != MotionEvent.ACTION_DOWN) {
        return false; // We didn't use this touch movement event.
    }
    Log.i(TAG, "onTouch down event");
    // Signal that we should cartoonify the next camera frame and
    save
    // it, instead of just showing the preview.
    mView.nextFrameShouldBeSaved(getBaseContext());
    return true;
}
```

5) 添加 nextFrameShouldBeSaved() 函数到 CartoonifierView.java 文件:

```
// Cartoonify the next camera frame & save it instead of preview.
protected void nextFrameShouldBeSaved(Context context) {
    bSaveThisFrame = true;
}
```

6) 在 CartoonifierView 的上面增加如下变量:

```
private boolean bSaveThisFrame = false;
private boolean bFreezeOutput = false;
private static final int FREEZE_OUTPUT_MSECS = 3000;
```

7) CartoonifierView 的 processFrame() 函数能在卡通图像和预览图像之间切换, 但该函数还应考虑这种情形: 用户不打算花几秒钟来显示被冻结的卡通图像。因此, 可用下面的代码来重载 processFrame()。

```
@Override
protected Bitmap processFrame(byte[] data) {
    // Store the output image to the RGBA member variable.
    int[] rgba = mRGBA;
    // Only process the camera or update the screen if we aren't
    // supposed to just show the cartoon image.
    if (bFreezeOutput) {
        // Only needs to be triggered here once.
        bFreezeOutput = false;
        // Wait for several seconds, doing nothing!
        try {
```

```

        wait(FREEZE_OUTPUT_MSECS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return null;
}
if (!bSaveThisFrame) {
    ShowPreview(getFrameWidth(), getFrameHeight(), data,
        rgba);
}
else {
    // Just do it once, then go back to preview mode.
    bSaveThisFrame = false;
    // Don't update the screen for a while, so the user can
    // see the cartoonifier output.

    bFreezeOutput = true;

    CartoonifyImage(getFrameWidth(), getFrameHeight(), data,
        rgba, m_sketchMode, m_alienMode, m_evilMode,
        m_debugMode);
}

// Put the processed image into the Bitmap object that will be
// returned for display on the screen.
Bitmap bmp = mBitmap;
bmp.setPixels(rgba, 0, getFrameWidth(), 0, 0, getFrameWidth(),
    getFrameHeight());

return bmp;
}

```

8) 现在重新编译并运行该应用, 会发现它工作得很好。

### 3. 将图像保存到文件并添加到 Android 图形库中

下面将图像另存为一个 PNG 文件并在 Android 图形库中显示。Android 图形库是为 JPEG 文件而设计的, 但 JPEG 会损坏单色和有边缘的卡通图像, 因此会考虑将 PNG 文件添加到图形库中, 但该方法比较烦琐。Java 函数 `savePNGImageToGallery()` 可实现这样的功能。在前面的 `processFrame()` 函数底部, 有一个 `Bitmap` 对象, 它是用来保存输出数据的; 可寻找一种方法将 `Bitmap` 对象保存为 PNG 文件。用 Java 实现的 OpenCV 函数 `imwrite()` 可将图像保存为 PNG 文件, 但在链接时需要 OpenCV 的 Java API 和 C/C++ API (就像 OpenCV4Android 示例项目 “tutorial-4-mixde” 所做的那样)。由于本项根本不需要 OpenCV 的 Java API, 下面将介绍如何仅用 Android API 来将图像保存为 PNG 文件, 具体实现过程如下:

1) 可使用 Android 的 `Bitmap` 类来将图像保存为 PNG 格式的文件。在保存文件之前需先确定文件名。可用当前日期和时间作为文件名, 这样可保存很多文件而不会出现文件名重复, 而且也帮助用户记住拍摄时间。只要在 `processFrame()` 函数的 `return bmp` 语句前插



入如下代码即可：

```
if (bFreezeOutput) {
// Get the current date & time
SimpleDateFormat s = new SimpleDateFormat("yyyy-MM-dd,HH-mm-ss");
String timestamp = s.format(new Date());
String baseFilename = "Cartoon" + timestamp + ".png";

// Save the processed image as a PNG file on the SD card and show
// it in the Android Gallery.
savePNGImageToGallery(bmp, mContext, baseFilename);
}
```

2) 在文件 CartoonifierView.java 前面添加下面的代码：

```
// For saving Bitmaps to file and the Android picture gallery.
import android.graphics.Bitmap.CompressFormat;
import android.net.Uri;
import android.os.Environment;
import android.provider.MediaStore;
import android.provider.MediaStore.Images;
import android.text.format.DateFormat;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.text.SimpleDateFormat;
import java.util.Date;
```

3) 在类 CartoonifierView 的开始处插入下面的代码：

```
private static final String TAG = "CartoonifierView";
private Context mContext; // So we can access the Android
// Gallery.
```

4) 在类 CartoonifierView 的 nextFrameShouldBeSaved() 函数中添加如下代码：

```
mContext = context; // Save the Android context, for GUI
// access.
```

5) 在类 CartoonifierView 中添加一个 savePNGImageToGallery() 函数：

```
// Save the processed image as a PNG file on the SD card
// and shown in the Android Gallery.
protected void savePNGImageToGallery(Bitmap bmp, Context context,
String baseFilename)
{
    try {
// Get the file path to the SD card.
String baseFolder = \
Environment.getExternalStoragePublicDirectory( \
```

```

Environment.DIRECTORY_PICTURES).getAbsolutePath() \
+ "/";
File file = new File(baseFolder + baseFilename);
Log.i(TAG, "Saving the processed image to file [" + \
file.getAbsolutePath() + "]");

// Open the file.
OutputStream out = new BufferedOutputStream(
new FileOutputStream(file));
// Save the image file as PNG.
bmp.compress(CompressFormat.PNG, 100, out);
// Make sure it is saved to file soon, because we are about
// to add it to the Gallery.
out.flush();
out.close();

// Add the PNG file to the Android Gallery.
ContentValues image = new ContentValues();
image.put(Images.Media.TITLE, baseFilename);
image.put(Images.Media.DISPLAY_NAME, baseFilename);
image.put(Images.Media.DESCRPTION,
"Processed by the Cartoonifier App");
image.put(Images.Media.DATE_TAKEN,
System.currentTimeMillis()); // msecs since 1970 UTC.
image.put(Images.Media.MIME_TYPE, "image/png");
image.put(Images.Media.ORIENTATION, 0);
image.put(Images.Media.DATA, file.getAbsolutePath());
Uri result = context.getContentResolver().insert(
MediaStore.Images.Media.EXTERNAL_CONTENT_URI, image);
}
catch (Exception e) {
    e.printStackTrace();
}
}

```

6) 如果需要在移动设备上保存文件, 则在安装 Android 应用时需要权限。将下面这一行插入 `AndroidManifest.xml` 中, 放在使其在获取摄像机访问权限的那行后面 (即 “`<uses-permission android:name= "android.permission.CAMERA" />`” 的后面), 因为这两行很相似。

```

<uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

```

7) 生成并运行该应用! 当通过触摸屏幕来保存照片时, 可看到显示在屏幕上的卡通化图像 (也许在经过 5 秒或 10 秒的处理后才能看到)。一旦卡通化图像显示在屏幕上, 就意味着它已被保存在 SD 卡和照片库中。退出此程序, 打开 Android 照片库应用, 可查看图片专辑。可在屏幕的全分辨率下观看 PNG 格式的卡通图像。

### 1.7.3 在 Android 系统中显示保存图像的消息

当新的图像被保存到 SD 卡和 Android 照片库时，若想显示一个提示信息，可以按如下步骤进行操作；否则可跳过这一节。

1) 将下面的代码加在 CartoonifierView.java 前面：

```
// For showing a Notification message when saving a file.
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.ContentValues;
import android.content.Intent;
```

2) 将下面的代码添加到类 CartoonifierView 的构造函数前面：

```
private int mNotificationID = 0;

// To show just 1 notification.
```

3) 在 processFrame() 函数中，调用 savePNGImageToGallery() 的 if 语句上方插入下面的代码：

```
savePNGImageToGallery() in processFrame():
showNotificationMessage(mContext, baseFilename);
```

4) 将 showNotificationMessage() 函数添加到 CartoonifierView 类中。

```
// Show a notification message, saying we've saved another image.
protected void showNotificationMessage(Context context,
    String filename)
{
    // Popup a notification message in the Android status
    // bar. To make sure a notification is shown for each
    // image but only 1 is kept in the status bar at a time,
    // use a different ID each time
    // but delete previous messages before creating it.
    final NotificationManager mgr = (NotificationManager) \
context.getSystemService(Context.NOTIFICATION_SERVICE);

    // Close the previous popup message, so we only have 1
    //at a time, but it still shows a popup message for each
    //one.
    if (mNotificationID > 0)
        mgr.cancel(mNotificationID);
    mNotificationID++;

    Notification notification = new Notification(R.drawable.icon,
        "Saving to gallery (image " + mNotificationID + ") ...",
        System.currentTimeMillis());
    Intent intent = new Intent(context, CartoonifierView.class);
    // Close it if the user clicks on it.
```



```

notification.flags |= Notification.FLAG_AUTO_CANCEL;
PendingIntent pendingIntent = PendingIntent.getActivity(context,
0, intent, 0);
notification.setLatestEventInfo(context, "Cartoonifier saved " +
mNotificationID + " images to Gallery", "Saved as '" +
filename + "'", pendingIntent);
mgr.notify(mNotificationID, notification);
}

```

5) 再次生成并运行此应用! 无论什么时候触摸屏幕保存图像时, 都会弹出一条提示消息。如果想让该消息在图像处理之前弹出, 可在 `cartoonifyImage()` 之前调用 `showNotificationMessage()`, 然后移动生成日期和时间字符的代码, 使显示的消息和保存的文件名是相同的字符串。

### 通过 Android 的菜单来选择卡通模式

用户可通过菜单来选择卡通模式:

1) 将下面的头文件加到文件 `src\com\Cartoonifier\CartoonifierApp.java` 的前面:

```

import android.view.Menu;
import android.view.MenuItem;

```

2) 将下面的成员变量加到 `CartoonifierApp` 类中。

```

// Items for the Android menu bar.
private MenuItem mMenuAlien;
private MenuItem mMenuEvil;
private MenuItem mMenuSketch;
private MenuItem mMenuDebug;

```

3) 将下面的函数添加到 `CartoonifierApp` 类中:

```

/** Called when the menu bar is being created by Android. */
public boolean onCreateOptionsMenu(Menu menu) {
    Log.i(TAG, "onCreateOptionsMenu");
    mMenuSketch = menu.add("Sketch or Painting");
    mMenuAlien = menu.add("Alien or Human");
    mMenuEvil = menu.add("Evil or Good");
    mMenuDebug = menu.add("[Debug mode]");
    return true;
}

/** Called whenever the user pressed a menu item in the menu bar.
 */
public boolean onOptionsItemSelected(MenuItem item) {
    Log.i(TAG, "Menu Item selected: " + item);
    if (item == mMenuSketch)
        mView.toggleSketchMode();
    else if (item == mMenuAlien)
        mView.toggleAlienMode();
    else if (item == mMenuEvil)
        mView.toggleEvilMode();
}

```

```

else if (item == mMenuDebug)
mView.toggleDebugMode();
return true;
}

```

4) 在 CartoonifierView 类中添加如下成员变量:

```

private boolean m_sketchMode = false;
private boolean m_alienMode = false;
private boolean m_evilMode = false;
private boolean m_debugMode = false;

```

5) 将下面的函数添加到 CartoonifierView 类中:

```

protected void toggleSketchMode() {
m_sketchMode = !m_sketchMode;
}
protected void toggleAlienMode() {
m_alienMode = !m_alienMode;
}
protected void toggleEvilMode() {
m_evilMode = !m_evilMode;
}
protected void toggleDebugMode() {
m_debugMode = !m_debugMode;
}

```

6) 将参数模式值以参数形式传递给 JNI 函数 cartoonifyImage(), 因此, 修改 CartoonifierView 类中的 CartoonifyImage() 函数:

```

public native void CartoonifyImage(int width, int height,
byte[] yuv,
int[] rgba, boolean sketchMode, boolean alienMode,
boolean evilMode, boolean debugMode);

```

7) 现在修改 processFrame() 函数的 Java 代码以传递当前模式值。

```

CartoonifyImage(getFrameWidth(), getFrameHeight(), data,
rgba,
m_sketchMode, m_alienMode, m_evilMode, m_debugMode);

```

8) 文件 jni\jni\_part.cpp 中 CartoonifyImage() 函数的 JNI 定义如下:

```

JNIEXPORT void JNICALL Java_com_Cartoonifier_CartoonifierView_
CartoonifyImage(
    JNIEnv* env, jobject, jint width, jint height,
    jbyteArray yuv, jintArray bgra, jboolean sketchMode,
    jboolean alienMode, jboolean evilMode, jboolean debugMode)

```

9) 然后通过 jni\jni\_part.cpp 文件的 JNI 函数将模式值传递给 cartoonifyImage 函数, 该函数保存在 cartoon.cpp 文件中, 是用 C/C++ 实现的。在开发 Android 应用时, 在同一时间只能显示一个 GUI 窗口, 但对于桌面应用, 在调试时很方便显示其他窗口。因此, 对

debugMode 变量不再取布尔值，而是在非调试状态下取 0，在移动设备的调试状态下取 1（在这种状态下，使用 OpenCV 创建 GUI 窗口会引起系统崩溃），在桌面应用的调试状态下取 2（在这种状态下可创建多个窗口）。

```
int debugType = 0;
if (debugMode)
    debugType = 1;
```

```
cartoonifyImage(mbgr, displayedFrame, sketchMode, alienMode,
evilMode, debugType);
```

10) 用下面的代码更新 cartoon.cpp 文件中相应的 C/C++ 代码：

```
void cartoonifyImage(Mat srcColor, Mat dst, bool sketchMode,
bool alienMode, bool evilMode, int debugType)
{
```

11) 并且更新 cartoon.h 中的 C/C++ 定义：

```
void cartoonifyImage(Mat srcColor, Mat dst, bool sketchMode,
bool alienMode, bool evilMode, int debugType);
```

12) 生成并运行该应用，然后试着按下窗口底部的小菜单按钮。读者会发现素描模式是实时的，但彩色模式受到双边滤波的影响会有很大的延迟。

#### 1.7.4 降低素描图像的随机椒盐噪声

目前大多数智能手机和平板电脑的照相机有明显的图像噪声，这通常可接受，但对于  $5 \times 5$  的 Laplacian 边缘滤波器影响很大。边缘掩码（比如：素描模式）经常会有很多被称为“椒盐”噪声的黑色小斑点，它是在白色背景下由几个相邻黑色像素构成的。使用均值滤波通常可以很好地去掉这种噪声，但在本章的项目中，均值滤波却不能有效去除椒盐噪声，因为边缘掩码大多数情形都是边缘和噪声点都为黑色（值为 0），而背景为纯白色（值为 255）。若使用标准的闭形态学算子，会删除很多边缘。因此，本项目采用一个自定义滤波器，它可删除被白色像素完全包围的小黑色区域。这样就可以去掉很多噪声点，同时对实际边缘影响不大。

通过遍历图像的每个黑色像素，以该像素为中心，检查  $5 \times 5$  正方形区域的边界像素是否为白色，如果它们都为白色，则说明有一个黑色噪声小岛，可用白色像素来填充整个块以删除黑色噪声。为了简化这个  $5 \times 5$  的滤波器，可忽略掉图像周围的两个边界像素。

下图左边的图像来自 Android 平板电脑，中间那幅图为素描模式下的图像（该图像有小的黑色椒盐噪声点），右边那幅图为使用上述方法删除椒盐噪声的结果，这幅图的皮肤看起来更加清晰。

下面是函数 removePepperNoise() 的实现代





码。为简便起见，该函数会对图像适当处理。

```
void removePepperNoise(Mat &mask)
{
    for (int y=2; y<mask.rows-2; y++) {
        // Get access to each of the 5 rows near this pixel.
        uchar *pUp2 = mask.ptr(y-2);
        uchar *pUp1 = mask.ptr(y-1);
        uchar *pThis = mask.ptr(y);
        uchar *pDown1 = mask.ptr(y+1);
        uchar *pDown2 = mask.ptr(y+2);

        // Skip the first (and last) 2 pixels on each row.
        pThis += 2;
        pUp1 += 2;
        pUp2 += 2;
        pDown1 += 2;
        pDown2 += 2;
        for (int x=2; x<mask.cols-2; x++) {
            uchar value = *pThis; // Get this pixel value (0 or 255).
            // Check if this is a black pixel that is surrounded by
            // white pixels (ie: whether it is an "island" of black).
            if (value == 0) {
                bool above, left, below, right, surroundings;
                above = *(pUp2 - 2) && *(pUp2 - 1) && *(pUp2) &&
                    *(pUp2 + 1) && *(pUp2 + 2);
                left = *(pUp1 - 2) && *(pThis - 2) && *(pDown1 - 2);
                below = *(pDown2 - 2) && *(pDown2 - 1) && *(pDown2) &&
                    *(pDown2 + 1) && *(pDown2 + 2);
                right = *(pUp1 + 2) && *(pThis + 2) && *(pDown1 + 2);
                surroundings = above && left && below && right;
                if (surroundings == true) {
                    // Fill the whole 5x5 block as white. Since we know
                    // the 5x5 borders are already white, we just need to
                    // fill the 3x3 inner region.
                    *(pUp1 - 1) = 255;
                    *(pUp1 + 0) = 255;
                    *(pUp1 + 1) = 255;
                    *(pThis - 1) = 255;
                    *(pThis + 0) = 255;
                    *(pThis + 1) = 255;
                    *(pDown1 - 1) = 255;
                    *(pDown1 + 0) = 255;
                    *(pDown1 + 1) = 255;
                    // Since we just covered the whole 5x5 block with
                    // white, we know the next 2 pixels won't be black,
                    // so skip the next 2 pixels on the right.
                    pThis += 2;
                    pUp1 += 2;
                    pUp2 += 2;
                }
            }
        }
    }
}
```

```

        pDown1 += 2;
        pDown2 += 2;
    }
}
// Move to the next pixel on the right.
pThis++;
pUp1++;
pUp2++;
pDown1++;
pDown2++;
}
}
}

```

### 1. 显示应用的每秒帧数 (FPS)

每秒帧数 (fps) 对于运行慢的应用 (比如本应用) 来讲不太重要, 但在屏幕上显示它还是有意义, 如果想显示每秒帧数 (fps) 是多少, 请执行如下步骤:

1) 将文件 `src\org\opencv\samples\imagemanipulations\FpsMeter.java` 从示例项目文件夹 (例如: `C:\OpenCV-2.4.1\samples\android\image-manipulations`) 复制到本项目的 `src\com\Cartoonifier` 文件夹。

2) 将包名 `FpsMeter.java` 替换为 `com.Cartoonifier`。

3) 在文件 `CartoonifierViewBase.java` 中在语句 `private byte[] mBuffer;` 后定义成员变量 `mFps`:

```
private FpsMeter mFps;
```

4) 在构造函数 `CartoonifierViewBase()` 的 `mHolder.addCallback (this);` 语句后面初始化 `mFps` 对象:

```
mFps = new FpsMeter();
mFps.init();
```

5) 在 `run()` 函数的 `try/catch` 块后面测量每帧的 FPS:

```
mFps.measure();
```

6) 用下面的代码在屏幕上显示 FPS, 该代码添加在 `run()` 函数的 `canvas.drawBitmap()` 函数后面。

```
mFps.draw(canvas, (canvas.getWidth() - bmp.getWidth()) / 2, 0);
```

### 2. 使用不同的摄像机分辨率

如果让应用运行得更快, 则效果会变差, 因此可考虑从设备上获取更小的视频图像或在获取图像时就缩小它。基于示例代码的卡通化程序会使用最靠近摄像机预览的分辨率, 该分辨率的高与屏幕高度一样。若设备是一个 500 万像素, 屏幕仅为  $640 \times 480$  的摄像机, 卡通化应用使用的分辨率可能为  $720 \times 480$  等。如果想对摄像机设置不同的分辨率, 需修改 `CartoonifierViewBase.java` 文件的 `surfaceChanged()` 函数中 `setupCamera()` 的参数。例如:

```

public void surfaceChanged(SurfaceHolder _holder, int format,
    int width, int height) {
    Log.i(TAG, "Screen size: " + width + "x" + height);
    // Use a camera resolution of roughly half the screen height.
    setupCamera(width/2, height/2);
}

```

一种获取摄像机最高分辨率的简易方法是传递一个很大的分辨率给函数，例如  $10\,000 \times 10\,000$ ，它将选择最大的有效分辨率（注意：函数选择的是最大预览分辨率，摄像机视频的分辨率通常小于单幅静止图像的分辨率）。或者为了让应用运行较快，可传递  $1 \times 1$  的分辨率给函数，它将采用最小的摄像机分辨率（例如： $160 \times 120$ ）。

### 3. 自定义应用

到目前为止，已经完成了整个 Android 卡通化应用的创建，读者可从中了解该应用工作的基本原理和各部分的功能；也可以自定义该应用！比如：可修改 GUI、应用的状态和工作流、卡通化滤波器常量、皮肤检测算法，或根据读者自己的想法来替代卡通化代码。

可用很多方法来改进皮肤检测算法，比如：使用更复杂的皮肤检测算法（如：通过一些最近的 CVPR 或 ICCV 国际会议论文来训练高斯模型，这些论文可在 <http://www.cvpapers.Com> 下载）或将人脸检测算法（见第 8 章）加到皮肤检测器中，这样做是为了能直接检测用户的脸而不是叫用户将他们的脸放到屏幕中心。要注意：人脸检测算法在某些设备或高分辨率摄像机上运行可能会花较长时间，因此，这类方法会受到相对较慢的处理速度限制，但智能手机和平板电脑的处理速度每年都有很大提高，在将来这也许不是一个问题。

提高移动设备上计算机视觉应用效率的最好方法是尽可能减少摄像机分辨率（例如：用 50 万像素代替 500 万像素），尽可能少地分配和释放图像所占内存，尽可能少做图像转换（例如：整个代码都支持 BGRA 格式）。也可向设备的 CPU 供应商（例如：NVIDIA 的 Tegra，Texas Instruments（德州仪器）的 OMAP，Samsung 的 Exynos，Apple 的 A<sub>x</sub>，或 Qualcomm Snapdragon）或某系列 CPU 供应商（如 ARM 的 Cortex-A9）索取优化的图像处理或数学库。

另外还需注意：有可能你的设备有优化的 OpenCV 版本。

本书自带的 ImageUtils.cpp 和 imageUtils.h 文件使得用户自定义 NDK 和桌面图像处理代码变得更容易。它包括像 printMatInfo() 这样的函数，该函数能打印许多关于 cv:Mat 对象的信息，从而使得调试 OpenCV 变得更加容易。另外，为了在 C/C++ 代码中很容易地增加详细的时间统计，可用时间的宏。例如：

```

DECLARE_TIMING(myFilter);

void myImageFunction(Mat img) {
    printMatInfo(img, "input");

    START_TIMING(myFilter);
}

```



```

    bilateralFilter(img, ...);
    STOP_TIMING(myFilter);
    SHOW_TIMING(myFilter, "My Filter");
}

```

然后可以在控制台看到有如下信息输出：

```

input: 800w600h 3ch 8bpp, range[19,255][17,243][47,251]
My Filter: time: 213ms (ave=215ms min=197ms max=312ms, across 57 runs).

```

当 OpenCV 并不像预期那样运行时，这种方法很有用；实际上，移动设备开发要使用 IDE 调试器通常都很困难，并且 printf() 语句一般都不能在 Android NDK 工作。但 ImageUtils 中的函数既可工作在 Android 系统也可工作在桌面上。

## 1.8 总结

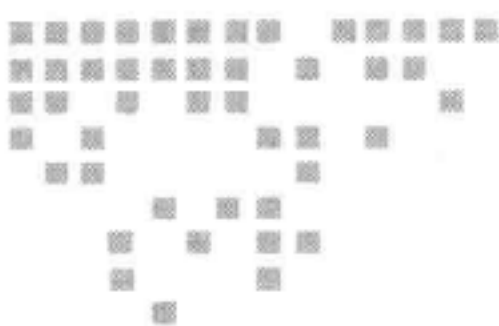
本章展示了几种不同类型的图像处理滤波器，这些滤波器可产生各种卡通效果：看上去像铅笔画的素描模式；看上去像一幅彩色画的图画模式；将素描模式叠加到图画模式上就得到卡通模式，该模式下图像看上去像一幅卡通画。另外还展示了其他有趣的效果，例如：怪物模式，它通过大幅提高噪声边缘而得到；外星模式，它通过将面部皮肤变成绿色并使其变明亮而得到。

有许多商业智能手机应用也可对人脸生成与之类似的效果，比如：卡通滤波器和皮肤颜色变换滤波器。也有一些使用类似概念的专业工具，如：平滑皮肤的后期视频处理工作，该工具可美化女性的脸使其看上去更年轻，采用平滑她们的皮肤，同时使边缘和非皮肤区域锐化来得到。

本章介绍如何将桌面应用移植为 Android 移动设备应用，首先按照开发桌面应用的方法进行开发，然后移植成移动设备上的应用，并创建一个适合于移动应用的用户界面。两个项目共享图像处理代码，读者可在桌面应用中修改卡通化滤波器，然后重新生成 Android 应用，就可自动展示修改后的效果。

这些使用 OpenCV4Android 项目的步骤经常变化，并且 Android 的发展本身也不是一成不变；因此，本章介绍了如何通过在一个 OpenCV 示例项目上增加功能来构建 Android 应用。希望读者可在 OpenCV4Android 将来的版本中添加相同的功能到同样的项目中。

本书同时提供桌面项目和 Android 项目的源代码。



## iPhone 或 iPad 上基于标记的增强现实

增强现实 (Augmented Reality, AR) 是真实环境的实时视图, 其中, 真实环境的元素被计算机生成的图形增强。因此, 该技术可提高当前人们对现实世界的感受。增强通常指有环境元素的实时性和语义上下文。受益于先进 AR 技术 (如: 将计算机视觉和对象识别加到 AR 技术中), 用户周围的环境信息可变得互动并可通过数字化方式进行操控。环境及相关对象上的虚拟信息可叠加到真实环境中。

本章将在 iPhone/iPad 设备上创建一个 AR 应用。该应用将从头开始创建, 它使用标记在图像上画一些虚拟物体, 这些图像是从摄像机上得到的。本章将介绍如何在 XCode IDE 环境中创建项目, 并配置该项目, 使其能使用 OpenCV。另外还会介绍其他内容, 例如: 从内置摄像头中获取视频, 使用 OpenGL ES 进行 3D 场景渲染, 解释一个常规 AR 应用的生成过程。

在开始之前, 给出所需知识和软件的简要清单:

- ❑ 需要一台安装了 XCode IDE 的苹果计算机。因为 iPhone/iPad 应用开发只能用苹果的 XCode IDE。这是建立该平台应用程序的唯一途径。
- ❑ 需要一台 iPhone、iPad 或 iPod Touch 设备。为了在设备上运行开发的应用程序, 还必须每年花 99 美元来购买苹果开发者认证 (Apple Developer Certificate)。没有此认证不可能在设备上运行应用程序。
- ❑ 还需要 XCode IDE 的基础知识。本书假定读者有一些该 IDE 的开发经验。
- ❑ 也需要 Object-C 和 C++ 编程语言的基础知识。但应用程序的所有复杂源代码都会被详细解释。

本章将介绍更多关于标记的知识, 会对完整的检测程序进行解释。学完本章, 读者可以写自己的标记检测算法, 能在三维空间中根据摄像机的位置来估计标记位置, 并利用它

们之间的这种变换（摄像机位置与标记位置之间是一个仿射变换）来可视化任意三维物体。

本章的示例项目可在随书的材料中找到。该项目对读者学习创建第一个在移动设备上的增加现实应用而言是好的开始。

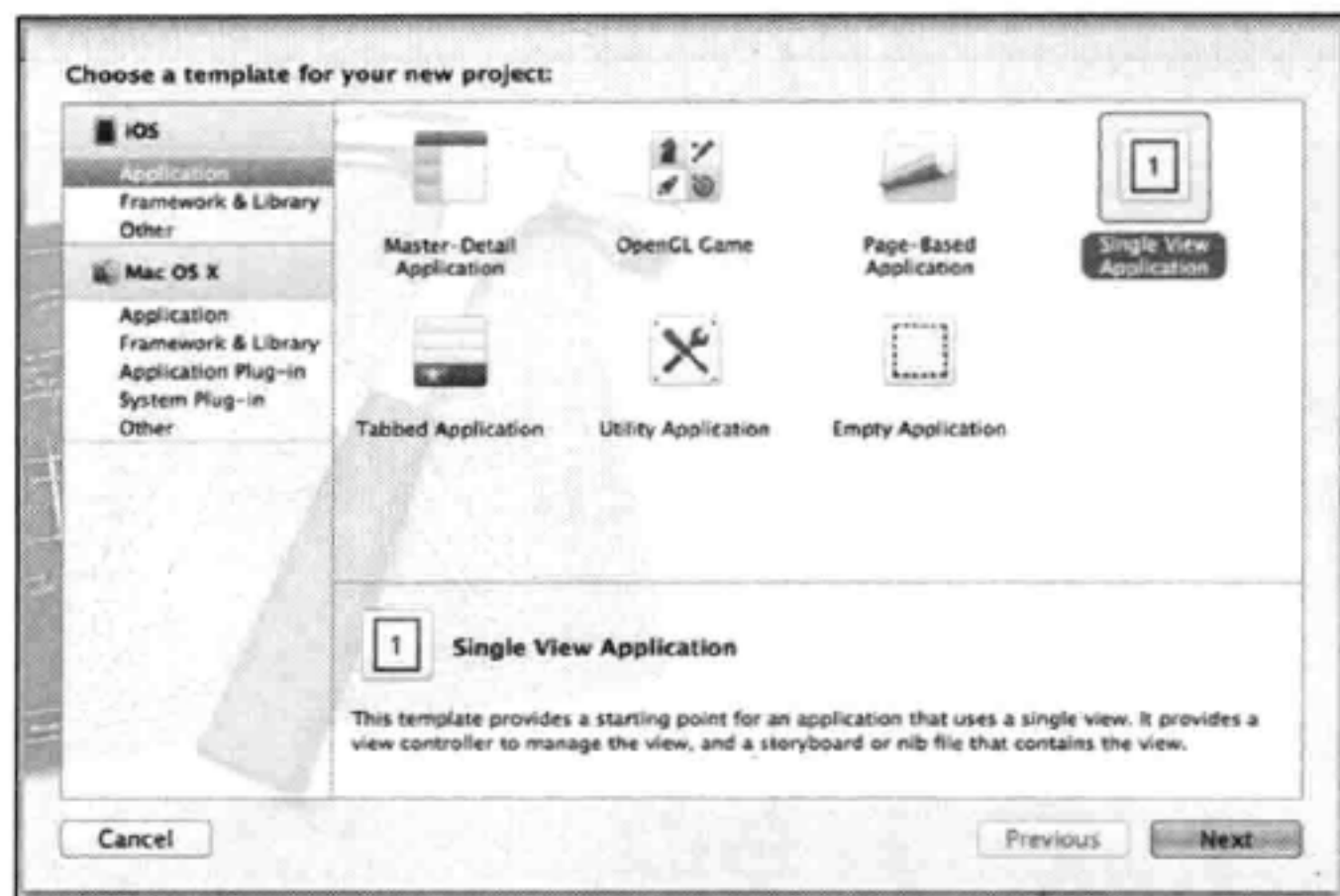
本章将涵盖下面的内容：

- ☐ 使用 OpenCV 创建一个 iOS 项目。
- ☐ 整个应用的结构。
- ☐ 标记标测。
- ☐ 标记验证。
- ☐ 标记的代码识别。
- ☐ 在 3D 环境中放置一个标记。
- ☐ 渲染 3D 虚拟对象。

## 2.1 使用 OpenCV 创建 iOS 项目

本节针对 iPhone/iPad 设备创建一个可演示的应用，该应用使用 OpenCV（Open Source Computer Vision）库来检测视频帧中的标记并渲染其上的 3D 对象，主要包括如何获取原始视频的数据流，通过 OpenCV 库进行图像处理，在图像中查找标记，提供一个 AR 覆盖（overlay）。

首先通过选择 iOS Single View Application 模板来创建一个新的 XCode 项目。其屏幕截图如下：



现在必须将 OpenCV 添加到该项目中。因为该应用会使用这个库的很多函数来检测标记和估计位置。

OpenCV 是一个用于进行实时计算机视觉编程的函数库。它最初由 Intel 开发，现在



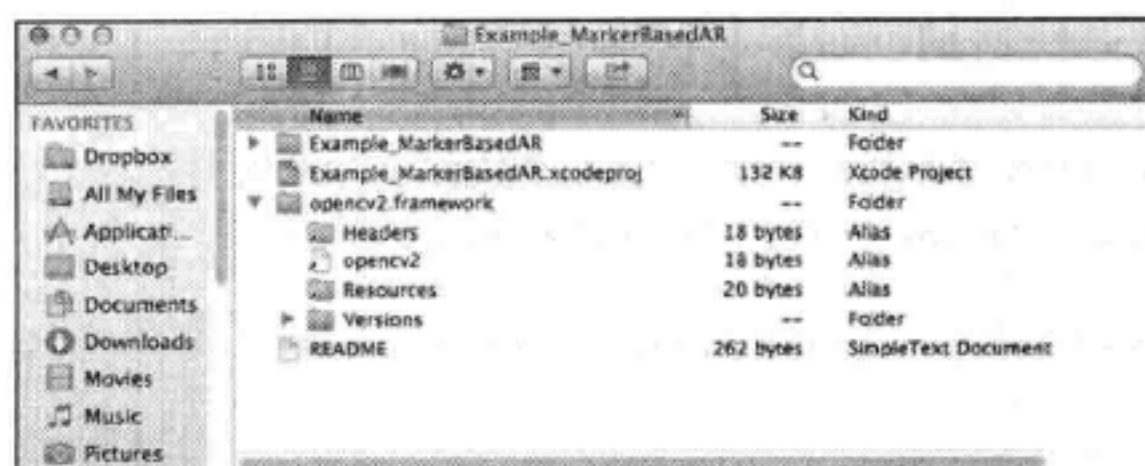
Willow Garage 和 Itseez 也支持它。该库由 C 和 C++ 编写而成。官方的 Python、非官方的 Java 以及 .NET 语言都与之绑定。

### 2.1.1 添加 OpenCV 框架

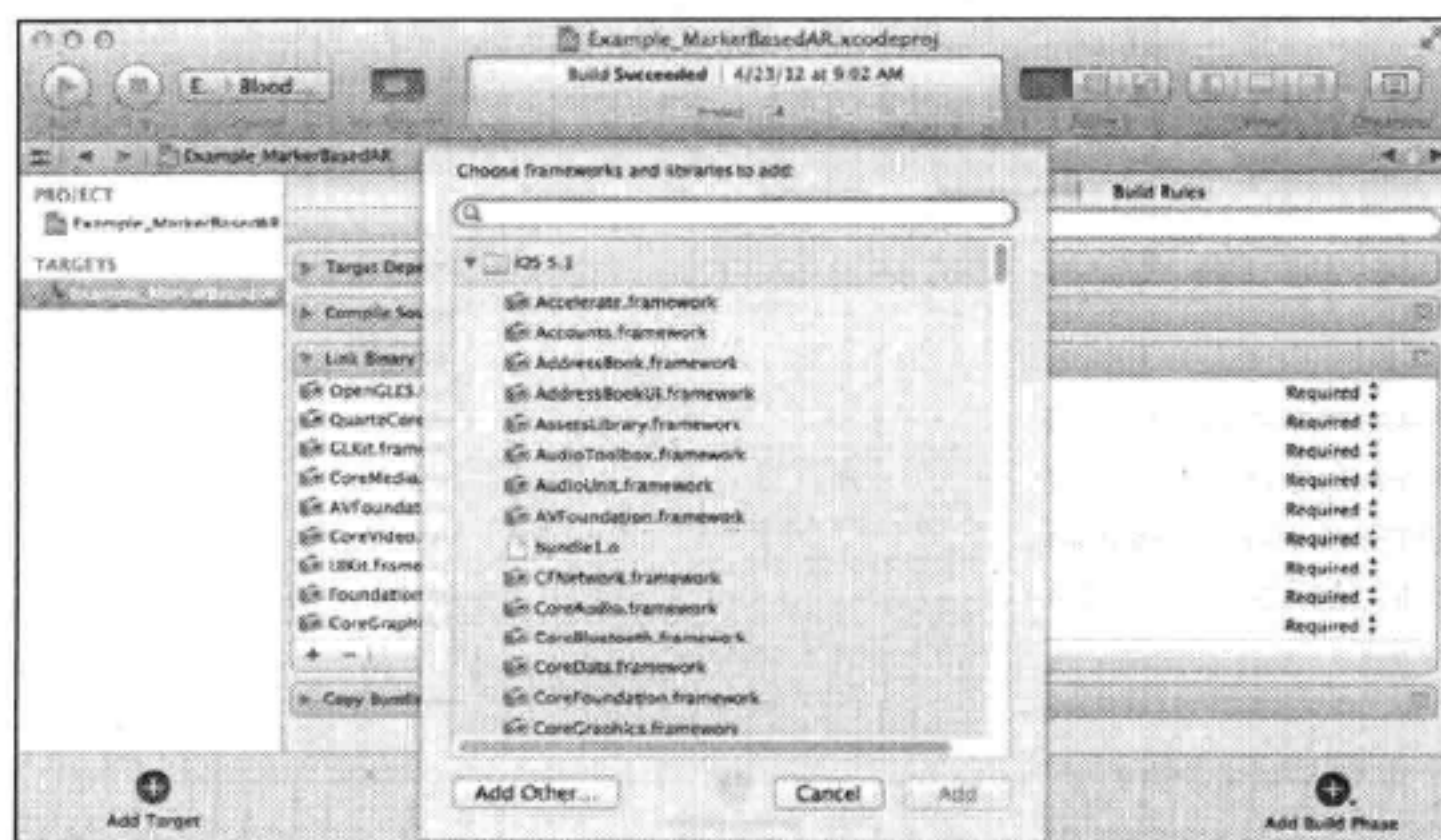
幸运的是该库跨平台，因此，它能在 iOS 设备上使用。官方支持 iOS 平台是从 OpenCV 库 2.4.2 版本开始，读者可从 <http://opencv.org/> 下载该库所发行的安装包。针对 iOS 的 OpenCV 链接指向压缩的 OpenCV 框架。如果读者是一个 iOS 开发的新手也不用担心，一个框架其实就像一个文件包。通常情况下每个框架包会含有一系列头文件和静态链接库列表。应用框架为开发者提供了一种简单发布预编译库的方法。

当然，读者可从头开始构建自己的库。OpenCV 的帮助文档详细解释了此过程。为了简单起见，本章还是按推荐的方式来使用框架。

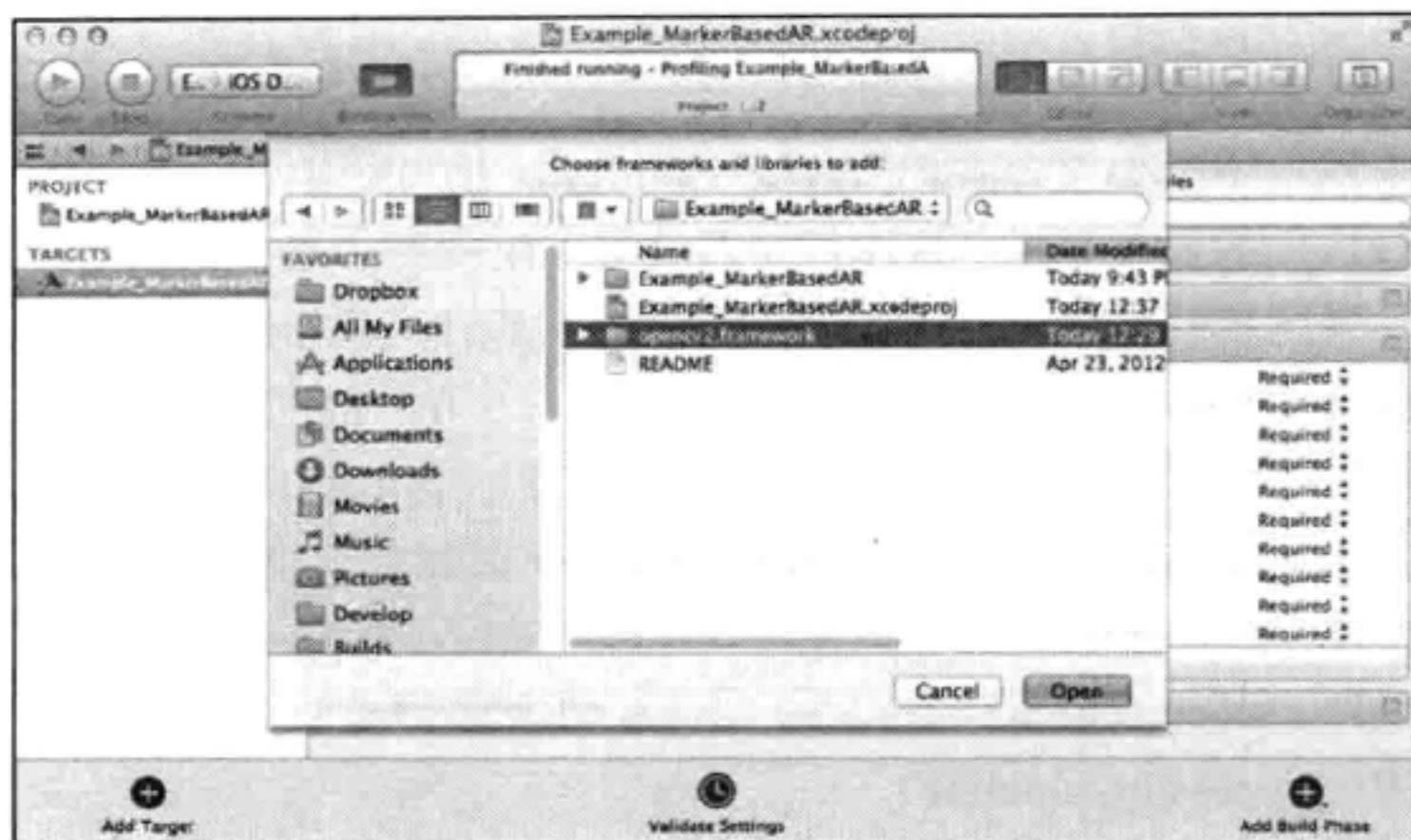
在下载这个框架文件后，提取其内容到项目文件夹中，屏幕截图如下：



为了让 XCode IDE 在生成应用程序的过程中使用各种框架，点击 Project options，然后找到 Build phases 选项卡。在这里可增加或删除生成应用所涉及的一系列框架。点击加号可添加新的框架，如下图所示：



从这里可选择一系列标准框架。但若要添加自定义框架，应点击 Add Other 按钮，将会弹出一个打开文件的对话框，选择项目文件夹中的 opencv2.framework，如下图所示：



### 2.1.2 包含 OpenCV 头文件

现在，基本完成将 OpenCV 框架添加到项目中，还剩下最后一件事：添加 OpenCV 的头文件到项目的预编译头文件中。预编译头文件有一个很好的特性：可减少编译时间。通过添加 OpenCV 头文件到预编译头文件中，也会使所有源代码自动包含 OpenCV 头文件。在项目源代码树中找到 .pch 文件，按下面的方法来修改它。

下列代码显示如何在项目源代码树中修改 .pch 文件。

```
//
// Prefix header for all source files of the 'Example_MarkerBasedAR'
//

#import <Availability.h>

#ifndef __IPHONE_5_0
#warning "This project uses features only available in iOS SDK 5.0 and
later."
#endif

#ifdef __cplusplus
#include <opencv2/opencv.hpp>
#endif

#ifdef __OBJC__
#import <UIKit/UIKit.h>
#import <Foundation/Foundation.h>
#endif
```

现在可在项目的任意地方调用 OpenCV 函数。

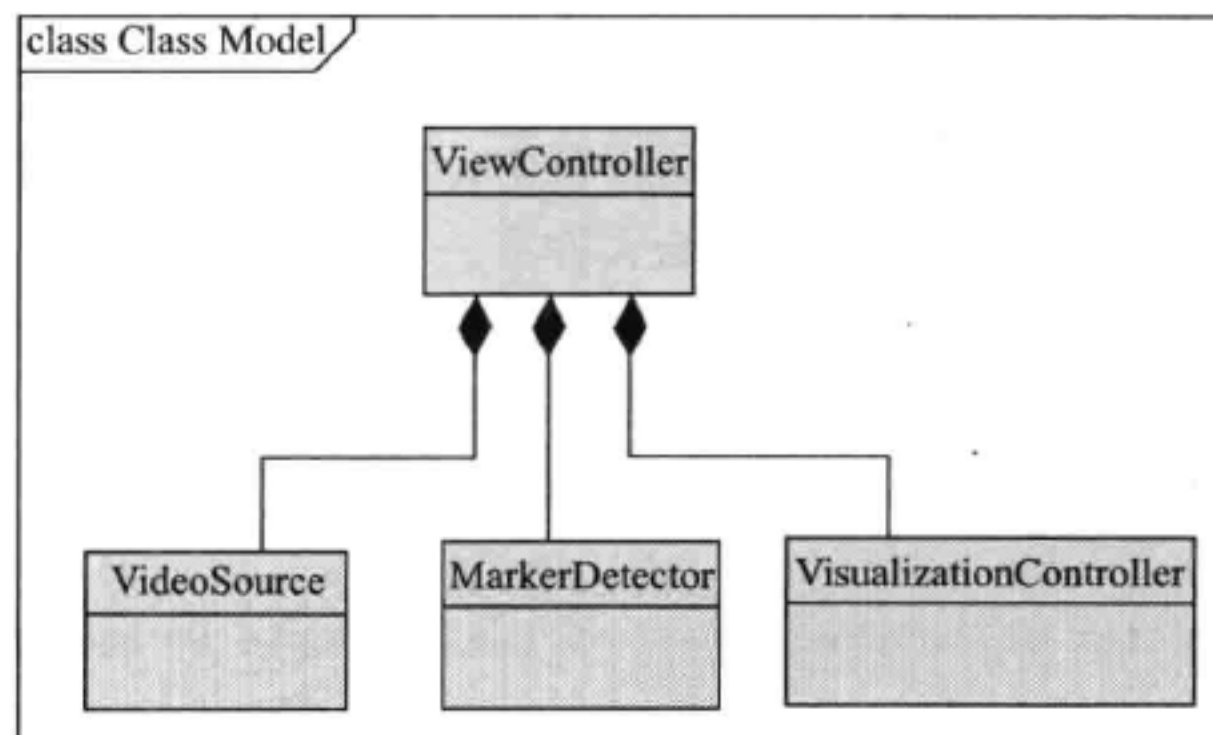
到此为止，项目模板就配置好了，可进行下一步工作了。友情提示：可为此项目制作一个副本，这样在创建下一个项目时可以节省时间。

## 2.2 应用程序的结构

每个 iOS 应用程序至少包含一个 `UIViewController` 接口的实例，该实例用来处理所有视图事件和管理应用的业务逻辑。视图控制器管理一组视图，这些视图是应用程序中用户界面的一部分。视图控制器作为应用程序控制层的一部分，其作用是协调模型对象和其他控制器对象，包括其他视图控制器，这样应用会呈现一个单一连贯的用户界面。

本章所实现的应用仅有一个视图，这就需要选择单一 `Single-View Application` 模板来创建它。该视图用来展示被渲染的图像。`ViewController` 类包含三个主要组件，这是每个 AR 应用程序必须包含（见下图）：

- ❑ 视频源组件 (video source)
- ❑ 处理流程 (processing pipeline)
- ❑ 可视化引擎 (visualization engine)



视频源组件负责通过用户程序从摄像机中获取新的帧。这意味着视频源组件能够选择摄像设备（前置或后置摄像头），调整摄像头参数（例如：所捕获视频的分辨率、白平衡、快门速度），在不冻结主界面的情况下获取帧。

在 `MarkerDetector` 类中封装了图像处理程序。该类为用户程序提供一个非常简单的接口。通常这类接口会像 `processFrame` 和 `getResult` 这样为一组函数。实际上，这些函数实际上都应属于 `ViewController`。若视图层没有很强的需求，不应将底层的数据结构和算法暴露出来。`VisualizationController` 组件包含所有增强现实可视化的逻辑，它也封装渲染引擎的具体实现细节。这样的代码相干性 (coherence) 低，用户随意修改这些组件也无须重写其他代码。

这样可使用户在其他平台和编译器上也能方便地使用这些独立模块，比如：对 `MarkerDetector` 类的代码无须做任何修改，就能很容易地用其开发 Mac、Windows 和 Linux 系统上的桌面应用程序。同样，可将 `VisualizationController` 移植到 Windows 平台，并使用 `Direct3D` 进行渲染。在这种情形下，仅需要重新实现 `VisualizationController` 的接口，但不



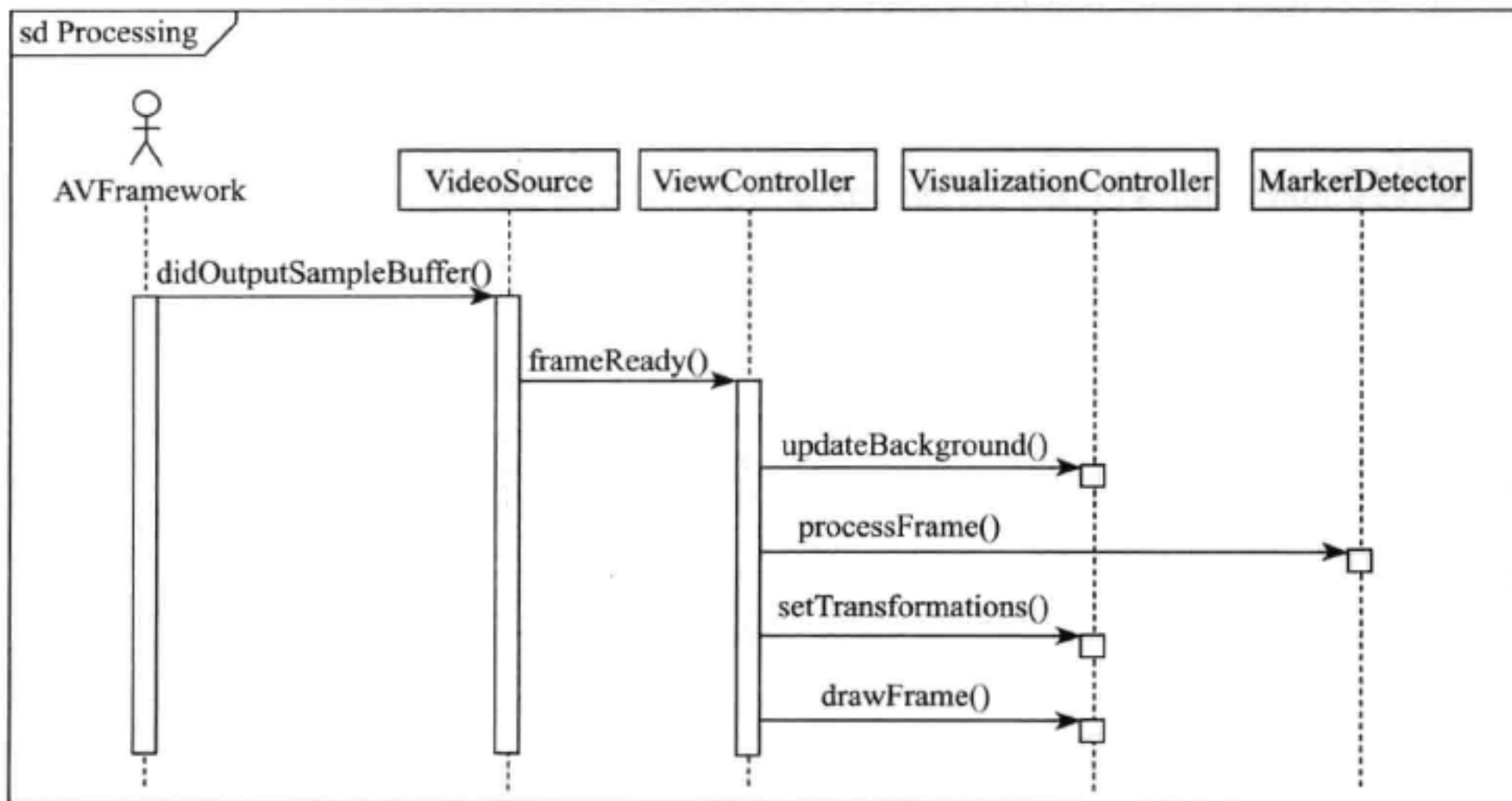
用修改其他代码。

主处理程序从视频源（VideoSource）组件获取一个新帧开始。这会让视频源组件通知用户程序用回调方式来处理该事件。ViewController 会处理这个回调并执行下面的操作：

- 1) 发送一个新的帧给可视化控制器。
- 2) 使用自定义流程来处理新帧。
- 3) 将已检测到的标记发送给可视化程序。
- 4) 渲染场景。

下面详细介绍该程序。AR 场景渲染包括绘制一幅背景图像，该图像含有最后接收到的那一帧的内容。人造的 3D 物体稍后绘制。当发送新帧给可视化程序时，需要复制图像数据到渲染引擎的内存缓冲区中。这还不是渲染，只是用新的位图来更新文本。

第二步是新帧处理和标记检测，即将图像作为输入并由此获得在其上的标记列表。这些标记传给可视化控制器，由其处理。下面这个时序图展示了此处理过程：



下面将实现视频获取组件。该组件负责获取所有帧并通过回调函数来发送已获取到的帧的消息。稍后将实现一个标记检测算法。该检测算法是本应用的关键。这部分会使用很多 OpenCV 函数来处理图像，检测这些图像的轮廓，找到标记框（rectangles），并估计其位置。然后重点使用增强现实来可视化前面处理过的图。将所有这些处理过程整合在一起，就完成了第一个 AR 应用。接下来将详细介绍整个过程。

## 访问摄像机

增强现实应用必须包括视频捕获和 AR 可视化这两个主要过程。视频捕获阶段包括从设备接收视频帧，执行必要的色彩转换，并且将其发送给图像处理流程。对 AR 应用来讲，

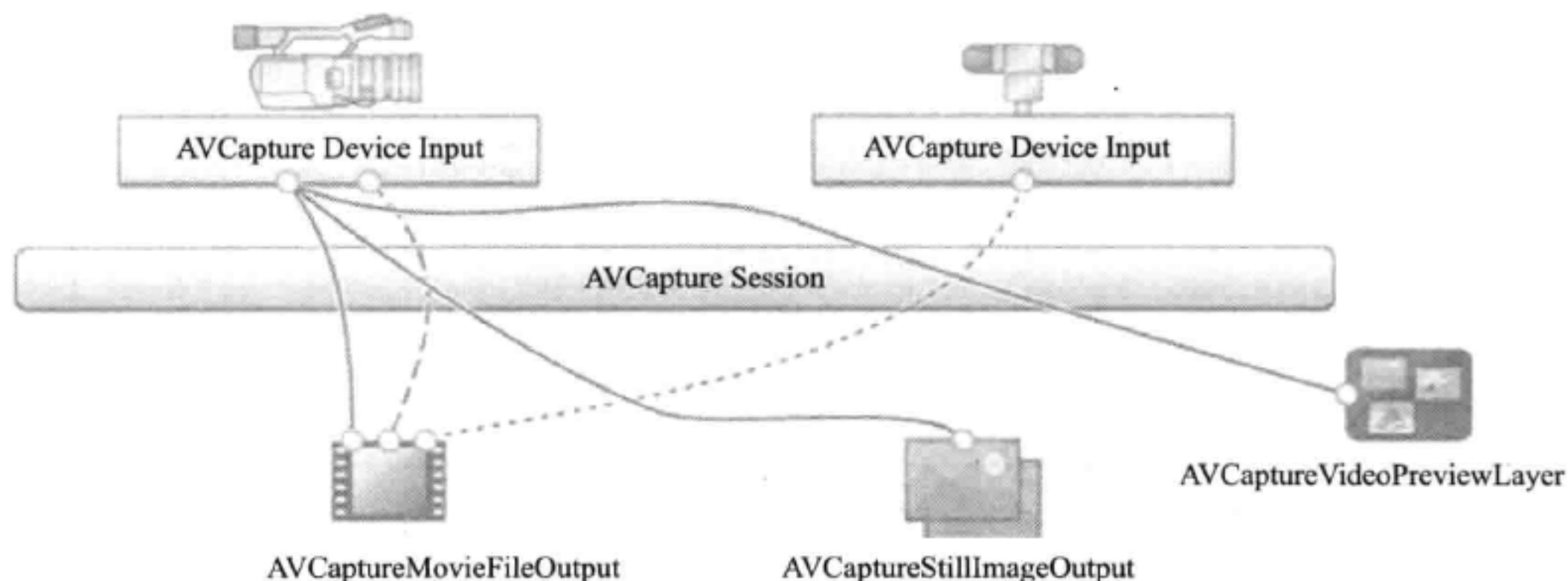
单帧处理的时间很关键，因此，帧的获取应尽可能高效。为了达到高性能，最好的办法是直接从摄像机读取帧。从 iOS 4 开始支持这种方式。AVFoundation 框架有现成的 API 函数来直接读取内存中的图像缓冲区。

使用 AVCaptureVideoPreviewLayer 类和 UIImageGetScreenImage 函数从摄像机获取视频的示例有很多。但这些技术适用于 iOS 3 或更早的版本，现在已经过时了。这类技术主要有两个弊端：

- ❑ 不能直接访问帧数据。为了得到一个位图，必须要创建一个中间实例 UIImage，复制一幅图像到这个实例中，然后再从该实例中取回。对于 AR 应用，这样做的代价太高，因为这是一个毫秒级的过程，若每秒丢失一些帧会显著降低用户体验。
- ❑ 为了绘制一个 AR，必须要添加一个透明的叠加视图来呈现 AR 效果。读者若参考 Apple 公司的帮助文档，就会发现用移动设备的处理器来混合透明层很难，应尽量避免非透明层。

类 AVCaptureDevice 和 AVCaptureVideoDataOutput 允许用户配置、捕获以及指定未处理视频帧，这些帧都是 32 bpp BGRA 格式（bpp 是 bit per pixel 的缩写）。也可设置输出帧的分辨率。但这样做会影响整体性能，因为较大的帧会花费更多的处理时间并需要更大的内存空间。

通过 AVFoundation API 来获取高性能视频有一个好的选择。它提供了一个更快、更简洁的方法来直接从摄像机缓冲区中获取帧。但首先需了解下图关于 iOS 的视频获取流程：



AVCaptureSession 是捕获视频的最基本对象，应该首先创建它。捕获视频的会话需要两个设备：输入设备和输出设备。输入设备既可是物理设备（摄像机）也可是视频文件（没有在上图显示）。在本项目中，输入设备是内置摄像头（前置或后置）。输出设备可以用下列接口之一来表示：

- ❑ AVCaptureMovieFileOutput
- ❑ AVCaptureStillImageOutput
- ❑ AVCaptureVideoPreviewLayer
- ❑ AVCaptureVideoDataOutput

AVCaptureMovieFileOutput 接口用于将视频写到文件中, AVCaptureStillImageOutput 接口用于生成静态图像, AVCaptureVideoPreviewLayer 接口用于在屏幕上进行视频预览。本项目将用到 AVCaptureVideoDataOutput 接口,因为它可直接访问视频数据。



**注意:** iOS 平台建立在 Objective-C 编程语言基础上。因此为了配合 AVFoundation 框架,本项目的类也必须要用 Objective-C 来实现。本节所有的代码清单使用 Objective-C++ 语言编写(将 C++ 和 Objective-C 结合的语言)。

为了封装视频捕获程序,需要创建 VideoSource 接口,其代码如下所示:

```
@protocol VideoSourceDelegate<NSObject>

- (void) frameReady: (BGRAVideoFrame) frame;

@end

@interface VideoSource : NSObject<AVCaptureVideoDataOutputSampleBufferDelegate>
{
}

@property (nonatomic, retain) AVCaptureSession *captureSession;
@property (nonatomic, retain) AVCaptureDeviceInput *deviceInput;
@property (nonatomic, retain) id<VideoSourceDelegate> delegate;

- (bool) startWithDevicePosition: (AVCaptureDevicePosition) devicePosition;
- (CameraCalibration) getCalibration;
- (CGSize) getFrameSize;

@end
```

在这个回调函数中,先锁定图像缓冲区以禁止任何新的帧对其修改,获取一个指向图像数据和帧维度的指针。然后构造临时的 BGRAVideoFrame 对象,然后通过具体的委托(delegate)将其传递给调用者。它是由外面具体委托程序传递进来的。该委托(delegate)有如下原型:

```
@protocol VideoSourceDelegate<NSObject>
- (void) frameReady: (BGRAVideoFrame) frame;

@end
```

在 VideoSourceDelegate 中, VideoSource 接口负责通知用户程序:有一个新的帧有效了。视频获取的初始化过程如下:

- 1) 创建一个 AVCaptureSession 实例,并预先设定视频采集会话要达到的质量。
- 2) 选择创建 AVCaptureDevice。可选择前置或后置摄像头或使用默认的摄像头。



3) 使用创建的视频采集设备来初始化 `AVCaptureDeviceInput`，并将其添加到视频采集会话中。

4) 创建一个 `AVCaptureVideoDataOutput` 的实例并用视频帧的格式、回调委托程序以及分配的队列来初始化它。

5) 添加输出设备到视频采集会话中。

6) 开始视频采集会话。

下面将更详细地解释这些步骤。在创建视频采集会话后，需要预先指定预期的视频质量，以便优化系统性能。本项目不需要处理高清视频。640 × 480 或更小的分辨率是一个很好的选择。

```
- (id)init
{
    if ((self = [super init]))
    {
        AVCaptureSession * capSession = [[AVCaptureSession alloc] init];

        if ([capSession canSetSessionPreset:AVCaptureSessionPreset640x480])
        {
            [capSession setSessionPreset:AVCaptureSessionPreset640x480];
            NSLog(@"Set capture session preset AVCaptureSessionPreset640x480");
        }
        else if ([capSession canSetSessionPreset:AVCaptureSessionPresetLow])
        {
            [capSession setSessionPreset:AVCaptureSessionPresetLow];
            NSLog(@"Set capture session preset AVCaptureSessionPresetLow");
        }

        self.captureSession = capSession;
    }
    return self;
}
```



**注意：**通常需要使用恰当的 API 来检查硬件的兼容性；并不保证每个摄像机与实际会话的预设兼容。

在创建一个视频采集会话后，应添加输入设备，`AVCaptureDeviceInput` 实例表示一个物理摄像机。函数 `cameraWithPosition` 是一个辅助函数，它用来返回指定位置的摄像机（前置、后置或默认）：

```
- (bool) startWithDevicePosition:(AVCaptureDevicePosition)
devicePosition
{
    AVCaptureDevice *videoDevice = [self cameraWithPosition:devicePosit
```

```

ion];

if (!videoDevice)
    return FALSE;

{
    NSError *error;

    AVCaptureDeviceInput *videoIn = [AVCaptureDeviceInput
    deviceInputWithDevice:videoDevice error:&error];
    self.deviceInput = videoIn;

    if (!error)
    {
        if ([[self captureSession] canAddInput:videoIn])
        {
            [[self captureSession] addInput:videoIn];
        }
        else
        {
            NSLog(@"Couldn't add video input");
            return FALSE;
        }
    }
    else
    {
        NSLog(@"Couldn't create video input");
        return FALSE;
    }
}

[self addRawViewOutput];
[captureSession startRunning];
return TRUE;
}

```

请注意上面的错误处理代码。设置返回值是一件很重要的事，因为指示硬件设备是否工作是一个好习惯。如果不这样做，代码在意想不到的情况下崩溃后，用户也不知道发生了什么事。

上面已经创建了一个视频采集会话并可获取视频帧。现在可增加一个输出设备，它是用来接收帧数据的对象。AVCaptureVideoDataOutput 类用来处理视频流中未压缩的帧。摄像机提供的帧是基于 BGRA、CMYK 或简单的灰度颜色格式。对于本项目，BGRA 颜色格式最适合，因为会使用这种格式的帧来做可视化和图像处理。下面的代码是 addRawViewOutput 函数的实现：

```

- (void) addRawViewOutput
{
    /*We setupt the output*/

```

```

    AVCaptureVideoDataOutput *captureOutput = [[AVCaptureVideoDataOutput
alloc] init];

    /*While a frame is processes in -captureOutput:didOutputSampleBuff
er:fromConnection: delegate methods no other frames are added in the
queue.
    If you don't want this behaviour set the property to NO */
    captureOutput.alwaysDiscardsLateVideoFrames = YES;

    /*We create a serial queue to handle the processing of our frames*/
    dispatch_queue_t queue;
    queue = dispatch_queue_create("com.Example_MarkerBasedAR.
cameraQueue",
    NULL);
    [captureOutput setSampleBufferDelegate:self queue:queue];
    dispatch_release(queue);

    // Set the video output to store frame in BGRA (It is supposed to be
faster)
    NSString* key = (NSString*)kCVPixelBufferPixelFormatTypeKey;
    NSNumber* value = [NSNumber
    numberWithInt:kCVPixelFormatType_32BGRA];

    NSDictionary* videoSettings = [NSDictionary
dictionaryWithObject:value
    forKey:key];
    [captureOutput setVideoSettings:videoSettings];

    // Register an output
    [self.captureSession addOutput:captureOutput];
}

```

现在视频采集会话已配置好了。它将从摄像机获取帧并传给用户程序开始。当新的帧有效时，AVCaptureSession 对象会执行 captureOutput: didOutputSampleBuffer:fromConnection 回调函数，在此函数中，为了使图像数据格式变得更适用，需执行一些数据转换，然后将其传递给用户程序：

```

- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
    fromConnection:(AVCaptureConnection *)connection
{
    // Get a image buffer holding video frame
    CVImageBufferRef imageBuffer = CMSampleBufferGetImageBuffer(sampleB
uffer);

    // Lock the image buffer
    CVPixelBufferLockBaseAddress(imageBuffer, 0);

    // Get information about the image
    uint8_t *baseAddress = (uint8_t *)CVPixelBufferGetBaseAddress(image
Buffer);

```



```

size_t width = CVPixelBufferGetWidth(imageBuffer);
size_t height = CVPixelBufferGetHeight(imageBuffer);
size_t stride = CVPixelBufferGetBytesPerRow(imageBuffer);

BGRAVideoFrame frame = {width, height, stride, baseAddress};
[delegate frameReady:frame];

/*We unlock the image buffer*/
CVPixelBufferUnlockBaseAddress(imageBuffer,0);
}

```

用一个引用变量（imageBuffer）指向存储帧数据的图像缓冲区。为了阻止新的帧修改当前图像缓冲区，需锁定此变量。锁定后，就可以独占方式访问数据帧。通过 CoreVideo API 可以得到图像的维度、宽度（每行的像素）以及指向图像数据起始位置的指针。



需要注意回调函数代码中 CVPixelBufferLockBaseAddress/ CVPixelBufferUnlockBaseAddress 函数的使用。为了保证缓冲区中数据的一致性和正确性，需持有缓冲区锁。在获得锁之后，数据读取才有效。在完成这些操作后，要记得对缓冲区解锁以便操作系统能向里面写入新的数据。

## 2.3 标记检测

标记通常设计成一幅黑色的矩形图像，且里面包含白色区域。由于已知情形有限，标记检测程序设计得很简单。首先需要在输入图像中找到一个封闭的轮廓，然后在这个轮廓中以矩形方式挖掉图像，并检查这是否接近标记模式。

在本示例中，标记模式为  $5 \times 5$ ，它看上去如右图所示：

在本章的示例项目中，其标记检测程序封装在 Marker Detector 类中。



```

/**
 * A top-level class that encapsulate marker detector algorithm
 */
class MarkerDetector
{
public:

    /**
     * Initialize a new instance of marker detector object
     * @calibration[in] - Camera calibration necessary for pose
     estimation.
     */
    MarkerDetector(CameraCalibration calibration);

    void processFrame(const BGRAVideoFrame& frame);

```

```

const std::vector<Transformation>& getTransformations() const;

protected:
bool findMarkers(const BGRAVideoFrame& frame, std::vector<Marker>&
detectedMarkers);

void prepareImage(const cv::Mat& bgraMat,
                  cv::Mat& grayscale);

void performThreshold(const cv::Mat& grayscale,
                     cv::Mat& thresholdImg);

void findContours(const cv::Mat& thresholdImg,
                  std::vector<std::vector<cv::Point> >& contours,
                  int minContourPointsAllowed);

void findMarkerCandidates(const std::vector<std::vector<cv::Point>
>&
contours, std::vector<Marker>& detectedMarkers);

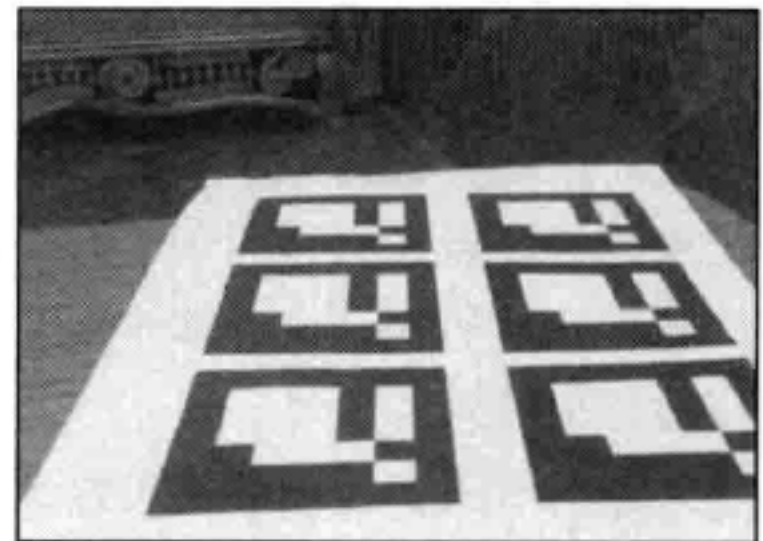
void detectMarkers(const cv::Mat& grayscale,
                  std::vector<Marker>& detectedMarkers);

void estimatePosition(std::vector<Marker>& detectedMarkers);

private:
};

```

为了帮助读者更好地理解标记检测程序，下面将详细介绍如何处理一帧视频图像。以下示例图像来自 iPad 的摄像头。



### 2.3.1 标记识别

下面是标记检测程序的流程：

- 1) 将输入图像转换为灰度图像。
- 2) 执行二值化阈值操作。
- 3) 检测轮廓。
- 4) 搜索可能的标记。
- 5) 检测和解码标记。
- 6) 估计标记的三维姿态。

#### 1. 灰度转换

必须将输入图像转换成灰度图像，因为标识通常仅包含黑白块，这使得更容易在灰度图像上操作这些块。幸运的事，OpenCV 色彩转换很简单。

请看下面这段 C++ 代码：

```
void MarkerDetector::prepareImage(const cv::Mat& bgraMat, cv::Mat&
    grayscale)
{
    // Convert to grayscale
    cv::cvtColor(bgraMat, grayscale, CV_BGR2GRAY);
}
```

该函数将 BGRA 格式的图像转换为灰度图像（如果需要，应分配图像缓冲区），其结果保存在第二个参数中。接下来的所有操作都基于这幅灰度图像。

## 2. 图像二值化

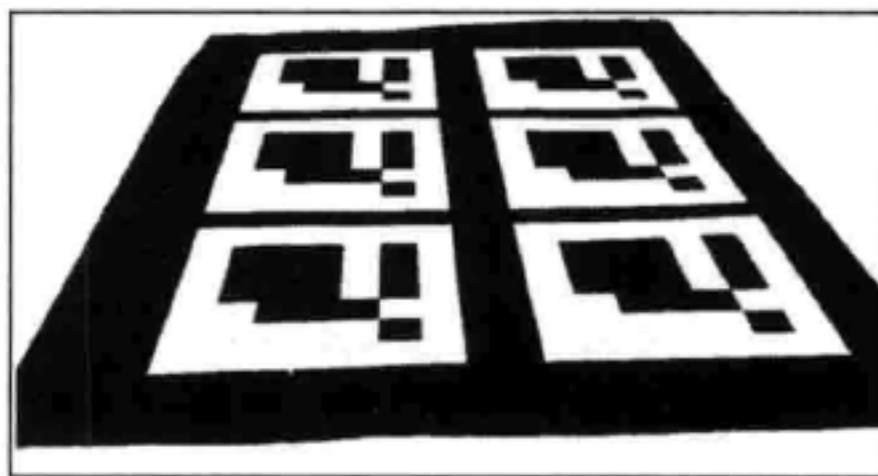
二值化操作将图像的每个像素变成黑色（像素值为 0）或白色（像素值为 255）。这一步是为检测轮廓做准备。有几种阈值化方法，每种方法都有优点和缺点。最简单且最快的方法是绝对阈值化方法。该方法的结果值依赖于当前像素值和某个阈值。如果像素值大于阈值，则将该像素设为白色（其像素值为 255）；否则设为黑色（其像素值为 0）。

该方法有一个很大的缺点：其结果取决于光照条件和软强度（soft intensity）变化。更合理的方法是采用自适应阈值法。二者主要的差别在于：以需要二值化的像素为中心，将给定半径内的所有像素的平均强度作为该像素的强度，从而使轮廓检测更具有鲁棒性。下面这段代码就是 MarkerDetector 类中所采用的自适应阈值法：

```
void MarkerDetector::performThreshold(const cv::Mat& grayscale,
    cv::Mat& thresholdImg)
{
    cv::adaptiveThreshold(grayscale, // Input image
        thresholdImg, // Result binary image
        255, //
        cv::ADAPTIVE_THRESH_GAUSSIAN_C, //
        cv::THRESH_BINARY_INV, //
        7, //
        7 //
    );
}
```

对输入的图像进行自适应阈值处理后，得到的图像与右图相似。

标记通常看上去像一个方块，其内部是黑白相间的区域。所以寻找标记的最好方式是：找到闭轮廓后用 4 个顶点的多边形逼近它们。



## 3. 轮廓检测

函数 `cv::findContours` 可检测所输入的二值图像的轮廓：

```
void MarkerDetector::findContours(const cv::Mat& thresholdImg,
    std::vector<std::vector<cv::Point>>
    & contours,
    int minContourPointsAllowed)
{
    std::vector< std::vector<cv::Point> > allContours;
```



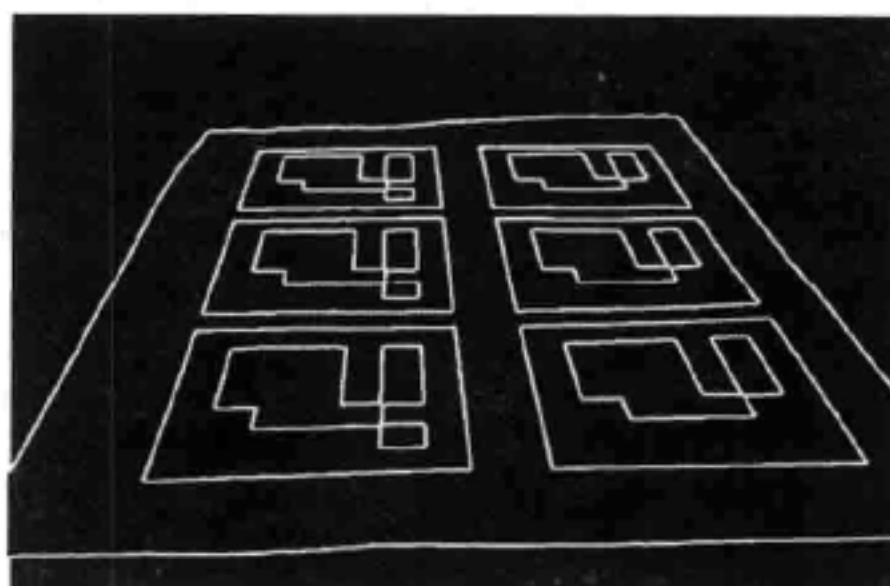
```

cv::findContours(thresholdImg, allContours, CV_RETR_LIST, CV_
CHAIN_APPROX_NONE);

contours.clear();
for (size_t i=0; i<allContours.size(); i++)
{
    int contourSize = allContours[i].size();
    if (contourSize > minContourPointsAllowed)
    {
        contours.push_back(allContours[i]);
    }
}
}

```

该函数的返回值为一个多边形列表，其每个多边形都表示一个轮廓。若轮廓包含的像素个数比 `minContourPointsAllowed` 的值要小，则此函数不会选择这样的轮廓，因为本项目对小轮廓不感兴趣。这些小轮廓可能并没包含标记，或轮廓不能由这样的小尺寸标记检测到，右图为检测到的轮廓图像。



#### 4. 搜索候选标记

在找到轮廓后，将开始多边形逼近。这样做是为了减少轮廓的像素。这是一种好的做法，它可筛选出非标记区域，因为标记总能被 4 个顶点的多边形表示。如果多边形的顶点多于或少于 4 个，就绝对不是本项目想要的标记。这种方法可由下面的代码来实现：

```

void MarkerDetector::findCandidates
(
    const ContoursVector& contours,
    std::vector<Marker>& detectedMarkers
)
{
    std::vector<cv::Point> approxCurve;
    std::vector<Marker> possibleMarkers;

    // For each contour, analyze if it is a parallelepiped likely to
    be the
    marker
    for (size_t i=0; i<contours.size(); i++)
    {
        // Approximate to a polygon
        double eps = contours[i].size() * 0.05;

        cv::approxPolyDP(contours[i], approxCurve, eps, true);

        // We interested only in polygons that contains only four
        points
    }
}

```

```

    if (approxCurve.size() != 4)
        continue;

    // And they have to be convex
    if (!cv::isContourConvex(approxCurve))
        continue;

    // Ensure that the distance between consecutive points is
    large enough
    float minDist = std::numeric_limits<float>::max();

    for (int i = 0; i < 4; i++)
    {
        cv::Point side = approxCurve[i] - approxCurve[(i+1)%4];
        float squaredSideLength = side.dot(side);
        minDist = std::min(minDist, squaredSideLength);
    }

    // Check that distance is not very small
    if (minDist < m_minContourLengthAllowed)
        continue;

    // All tests are passed. Save marker candidate:
    Marker m;

    for (int i = 0; i < 4; i++)
        m.points.push_back( cv::Point2f(approxCurve[i].x, approxCurve[i].y) );

    // Sort the points in anti-clockwise order
    // Trace a line between the first and second point.
    // If the third point is at the right side, then the points
    are anti-
    clockwise
    cv::Point v1 = m.points[1] - m.points[0];
    cv::Point v2 = m.points[2] - m.points[0];

    double o = (v1.x * v2.y) - (v1.y * v2.x);

    if (o < 0.0) //if the third point is in the left side,
    then
        sort in anti-clockwise order
        std::swap(m.points[1], m.points[3]);

    possibleMarkers.push_back(m);
}

// Remove these elements which corners are too close to each
other.

```

```

// First detect candidates for removal:
std::vector< std::pair<int,int> > tooNearCandidates;
for (size_t i=0;i<possibleMarkers.size();i++)
{
    const Marker& m1 = possibleMarkers[i];

    //calculate the average distance of each corner to the nearest
corner
    of the other marker candidate
    for (size_t j=i+1;j<possibleMarkers.size();j++)
    {
        const Marker& m2 = possibleMarkers[j];

        float distSquared = 0;

        for (int c = 0; c < 4; c++)
        {
            cv::Point v = m1.points[c] - m2.points[c];
            distSquared += v.dot(v);
        }

        distSquared /= 4;

        if (distSquared < 100)
        {
            tooNearCandidates.push_back(std::pair<int,int>(i,j));
        }
    }
}

// Mark for removal the element of the pair with smaller perimeter
std::vector<bool> removalMask (possibleMarkers.size(), false);

for (size_t i=0; i<tooNearCandidates.size(); i++)
{
    float p1 = perimeter(possibleMarkers[tooNearCandidates[i].
first
].points);
    float p2 =
    perimeter(possibleMarkers[tooNearCandidates[i].second].
points);

    size_t removalIndex;
    if (p1 > p2)
        removalIndex = tooNearCandidates[i].second;
    else
        removalIndex = tooNearCandidates[i].first;

    removalMask[removalIndex] = true;
}

```



```

// Return candidates
detectedMarkers.clear();
for (size_t i=0;i<possibleMarkers.size();i++)
{
    if (!removalMask[i])
        detectedMarkers.push_back(possibleMarkers[i]);
}
}

```

现在得到一系列的平行六面体，它们可能是标记。为了验证它们是否为标记，需要执行下面三个步骤：

1) 首先，为了获得矩形区域的正面视图，应该删除透视投影。

2) 然后用 Otsu 算法得到图像进行二值化处理时的阈值。Otsu 算法假定图像直方图呈双峰分布，然后搜索一个阈值，该阈值使得类间 (extra-class) 的方差尽量大，而使类内 (intra-class) 的方差尽量小。

3) 最后进行标记编码识别。每个标记都有一个内部编码。标记被分成  $7 \times 7$  的网格，其中内部  $5 \times 5$  的网格包含 ID 信息。其余部分是外部黑色边界。因此，首先需要检查外部黑色边界是否存在，然后读取  $5 \times 5$  网格中是否存在有效的标记编码 (有时需要旋转标记编码来得到有效的标记编码)。

为了得到矩形标记图像，必须通过透视变换来变换输入图像，其变换矩阵可通过函数 `cv::getPerspectiveTransform` 计算得到。该函数通过四边形顶点来得到透视变换矩阵。函数的第一个参数为标记在图像空间中的坐标；第二个参数为方形标记图像四个顶点的坐标。

下面介绍如何将标记图像变换为方形图像：

```

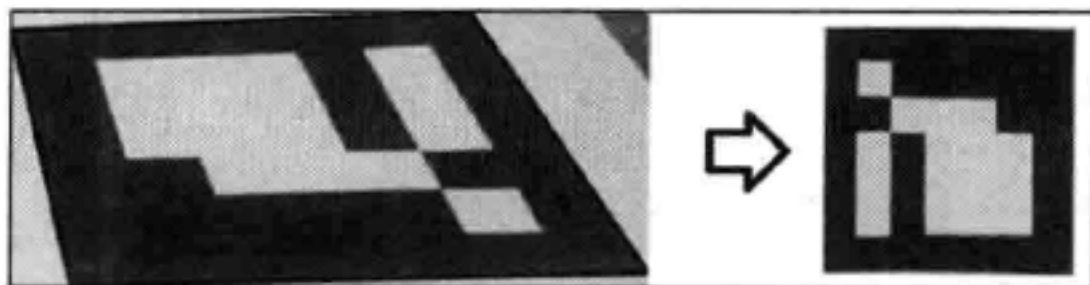
cv::Mat canonicalMarker;
Marker& marker = detectedMarkers[i];

// Find the perspective transformation that brings current marker to
// rectangular form
cv::Mat M = cv::getPerspectiveTransform(marker.points, m_
markerCorners2d);

// Transform image to get a canonical marker image
cv::warpPerspective(grayScale, canonicalMarker, M, markerSize);

```

下图就是通过透视变换将扭曲的图像转换为方形图像。



下面检查所得的方形图像是否为一个有效的标记图像。可通过标记编码来得到位图掩码。为了让标记只包含黑白两种颜色，可用 Otsu 阈值法来去掉灰度像素，使图像只留下黑

白像素。

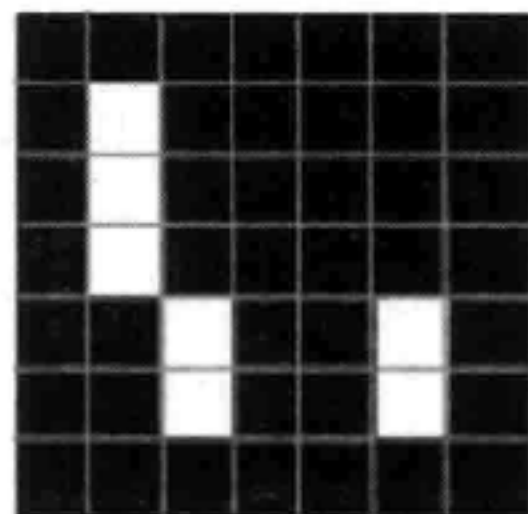
```
//threshold image
cv::threshold(markerImage, markerImage, 125, 255, cv::THRESH_BINARY |
cv::THRESH_OTSU);
```



### 2.3.2 标记编码识别

每个标记都有一个内部编码，即内部  $5 \times 5$  的网格区域的每一行由 5 位 (bit) 表示。这种编码方案是对海明码的少量修改而得到的。总的来讲，除每行的 5 位编码以外，还有两位是信息位，其他 3 位是错误检测码，因此，会有 1024 种不同的标记编号 (ID)。

标记的这种编码与海明码最大的区别为：会对第一位（编码的第 3 位和 5 位的奇偶校验位）取反。如果编码为 0（其海明码为 00000），则标记编码为 10000。这样做是为了防止一个完全黑色的矩形变成一个有效标记编码。



通过计算有标记编码的每个网格区域的黑白像素个数来得到一个  $5 \times 5$  位的掩码。为了统计一幅图像的非零像素的个数，可使用 `cv::countNonZero` 函数。该函数可计算给定的一维或二维数组中非零元素个数。`cv::Mat` 类型可以得到一个子图像 (subimage) 视图，它是 `cv::Mat` 的新实例，包含了原图像的一部分。例如：若有一个  $400 \times 400$  的 `cv::Mat` 对象，下面这段代码会对  $50 \times 50$  的图像块创建一个子矩阵 (submatrix)，该图像块的起始位置为 (10,10)。

```
cv::Mat src(400,400,CV_8UC1);
cv::Rect r(10,10,50,50);
cv::Mat subView = src(r);
```

#### 1. 读取标记代码

使用这种方法，可很容易在标记图像上找到黑白网格区域。

```
cv::Mat bitMatrix = cv::Mat::zeros(5,5,CV_8UC1);
```

```
//get information(for each inner square, determine if it is black
or white)
for (int y=0;y<5;y++)
{
    for (int x=0;x<5;x++)
```

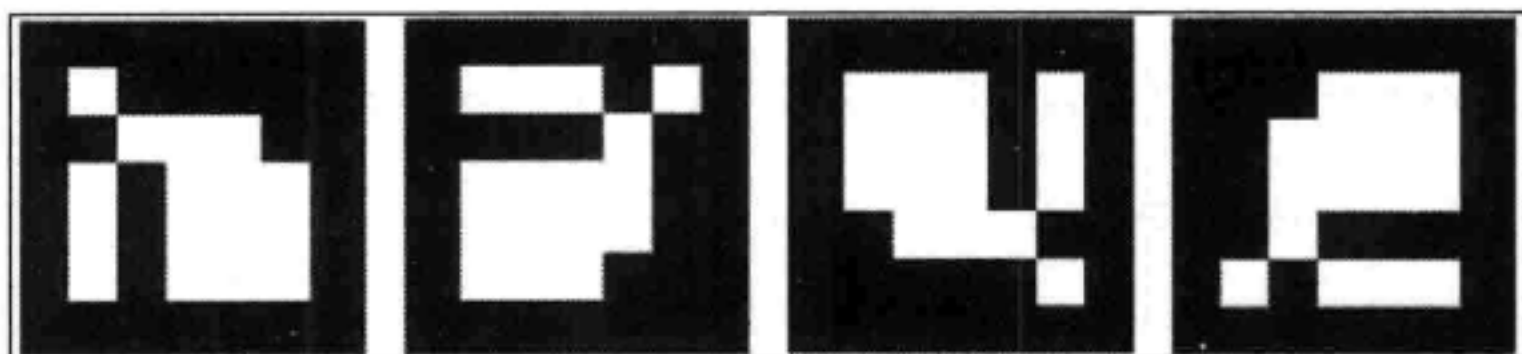
```

{
    int cellX = (x+1)*cellSize;
    int cellY = (y+1)*cellSize;
    cv::Mat cell = grey(cv::Rect(cellX,cellY,cellSize,cellSize));

    int nZ = cv::countNonZero(cell);
    if (nZ > (cellSize*cellSize) / 2)
        bitMatrix.at<uchar>(y,x) = 1;
}
}

```

从下面这幅图可看出, 同样的标记由于摄像机的视点不一样, 可能有四种不同的表示。



必须从这 4 种可能的标记图像中找到正确的标记位置。在前面, 对每两个信息位引入三个奇偶校验位。有了这些校验位, 就能对每个可能的标记定位 (marker orientation) 找到海明距离。正确的标记位置其海明码为 0, 而其他旋转形式的标记, 海明码不为零。

下面的代码段将位矩阵 (bit matrix) 旋转 4 次, 然后找到正确的标记定位。

```

//check all possible rotations
cv::Mat rotations[4];
int distances[4];

rotations[0] = bitMatrix;
distances[0] = hamDistMarker(rotations[0]);

std::pair<int,int> minDist(distances[0],0);

for (int i=1; i<4; i++)
{
    //get the hamming distance to the nearest possible word
    rotations[i] = rotate(rotations[i-1]);
    distances[i] = hamDistMarker(rotations[i]);

    if (distances[i] < minDist.first)
    {
        minDist.first = distances[i];
        minDist.second = i;
    }
}

```

这段代码针对海明距离来获取最小误差, 从而找到位矩阵的方向。若是一个正确的标记 ID, 则该误差为零。也就是说, 会碰到一个错误的标记模式 (可能是损坏的图像或错误位置的标记检测)。



## 2. 标记位置细化

在找到正确的标记定位后，要旋转它的角点，使其符合标准顺序。

```
//sort the points so that they are always in the same order
// no matter the camera orientation
std::rotate(marker.points.begin(), marker.points.begin() + 4 -
nRotations,
marker.points.end());
```

在检测到标记并对标记 ID 解码后，需要细化它的角点。此操作对下一步在三维空间估计标记位置很有用。可通过 `cv::cornerSubPix` 函数按亚像素级精度来查找角点位置。该函数的用法为：

```
std::vector<cv::Point2f> preciseCorners(4 * goodMarkers.size());

for (size_t i=0; i<goodMarkers.size(); i++)
{
    Marker& marker = goodMarkers[i];

    for (int c=0; c<4; c++)
    {
        preciseCorners[i*4+c] = marker.points[c];
    }
}

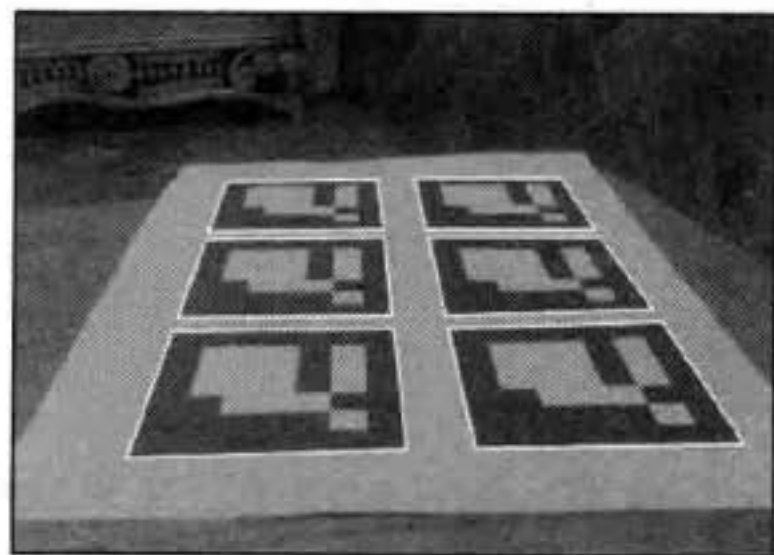
cv::cornerSubPix(grayScale, preciseCorners, cvSize(5,5),
cvSize(-1,-1), cvTermCriteria(CV_TERMCRIT_ITER,30,0.1));

//copy back
for (size_t i=0; i<goodMarkers.size(); i++)
{
    Marker&marker = goodMarkers[i];

    for (int c=0; c<4; c++)
    {
        marker.points[c] = preciseCorners[i*4+c];
    }
}
```

整个代码的第一步是为函数准备输入的数据，即复制输入数组的顶点列表。第二步是调用 `cv::cornerSubPix`，并将实际图像、顶点以及影响位置细化的质量和性能的参数传递给此函数。完成之后，将细化位置复制给标记角点，如右图所示。

由于 `cornerSubPix` 函数太复杂，在标记检测的前期不会使用它。在顶点较多的情况下，调用该函数的代价很高，因此只针对有效的标记才调用它。



## 2.4 在三维空间放置标记

增强现实试图将虚拟内容与真实物体融合。为了将三维模型放置在场景中，需要知道它关于摄像机的姿态。可在直角坐标系中使用欧氏空间 + 变换来表示这个姿态。

在三维情形下，标记的位置与其在二维空间的投影有如下关系：

$$P=A*[R|T]*M;$$

其中：

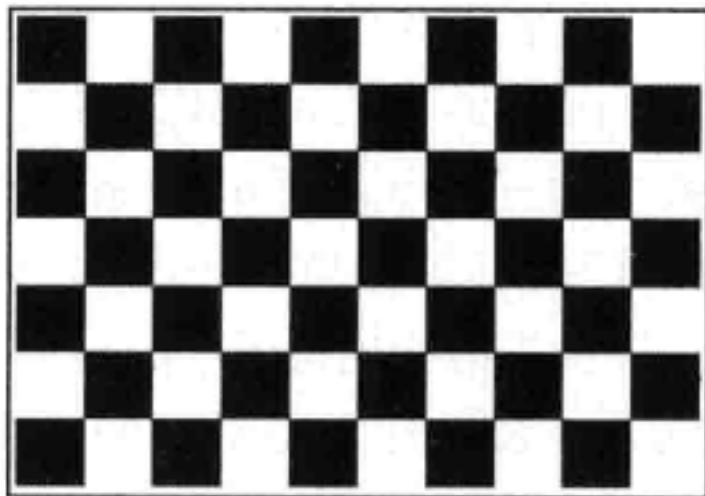
- $M$  表示三维空间的一个点；
- $[R|T]$  表示一个  $[3|4]$  矩阵，该矩阵为一个欧氏空间变换；
- $A$  表示摄像机矩阵或内部参数 (intrinsic parameter) 矩阵；
- $P$  表示  $M$  在屏幕上的投影。

在执行标记检测后，需要知道二维情形下标记的 4 个角点位置（在屏幕空间中的投影）。下一节将介绍如何获取矩阵  $A$ 、参数向量  $M$  以及计算变换矩阵  $[R|T]$ 。

### 2.4.1 摄像机标定

每个摄像机都有唯一的参数。例如：焦距、主点 (principal point) 以及透镜的畸变模型。查找摄像机内参数的过程为摄像机标定。对基于增强现实的应用来讲，对摄像机标定很重要，因为它将透视变换和透镜的畸变都反映在输出图像上。为了让用户在增强现实应用中获得最佳体验，应该用相同的透视投影来增强物体的可视化效果。

标定摄像机需要特殊模式图像（例如：棋盘板或具有白色背景的黑圆圈），被标定的摄像机需从不同角度对特殊模式图像拍摄 10 ~ 15 张照片，然后通过标定算法来找到最优的摄像机内部参数和畸变向量。



在本项目在 CameraCalibration 类中实现了对摄像机的标定功能。

```
/**
 * A camera calibration class that stores intrinsic matrix and
 * distortion coefficients.
 */
class CameraCalibration
{
public:
    CameraCalibration();
    CameraCalibration(float fx, float fy, float cx, float cy);
    CameraCalibration(float fx, float fy, float cx, float cy, float
        distortionCoeff[4]);

    void getMatrix34(float cparam[3][4]) const;
```

```

const Matrix33& getIntrinsic() const;
const Vector4& getDistorsion() const;

private:
Matrix33 m_intrinsic;
Vector4 m_distorsion;
};

```

对标定过程的详细介绍超过了本章范围。感兴趣的读者可参考 OpenCV 的 `camera_calibration` 示例项目或阅读《OpenCV: Estimating Projective Relations in Images》，该书的附属信息和源代码可在链接 <http://www.packtpub.com/article/opencv-estimating-projective-relations-images> 下载。

本项目会针对当前所有 iOS 设备 (iPad 2、iPad 3 和 iPhone 4) 提供相应的内部参数。

## 2.4.2 标记姿态估计

在三维空间中，可通过标记角点的精确位置来估计摄像机与标记之间的变换。此操作称为二维到三维的姿态估计。该估计过程会在物体与摄像机之间找到一个欧氏空间的变换（该变换仅由旋转和坐标平移构成）。

先来观察右图：

图中的  $C$  表示摄像机中心，点  $P_1$ - $P_4$  是现实坐标系中的三维点，而  $p_1$ - $p_4$  是将点  $P_1$ - $P_4$  投影到摄像机的图像平面而得到的。标记位置估计的目的就是在已知三维世界的标记位置 ( $p_1$ - $p_4$ )、有内部参数矩阵的摄像机  $C$  以及已知图像平面的投影点 ( $P_1$ - $P_4$ ) 的情况下，找到标记与摄像机之间相对变换关系。但在哪儿能得到三维空间中标记位置的坐标呢？可以想象一下，标记总是方形且所有顶点都在一个平面上，因此可按如右方式定义标记的角点：

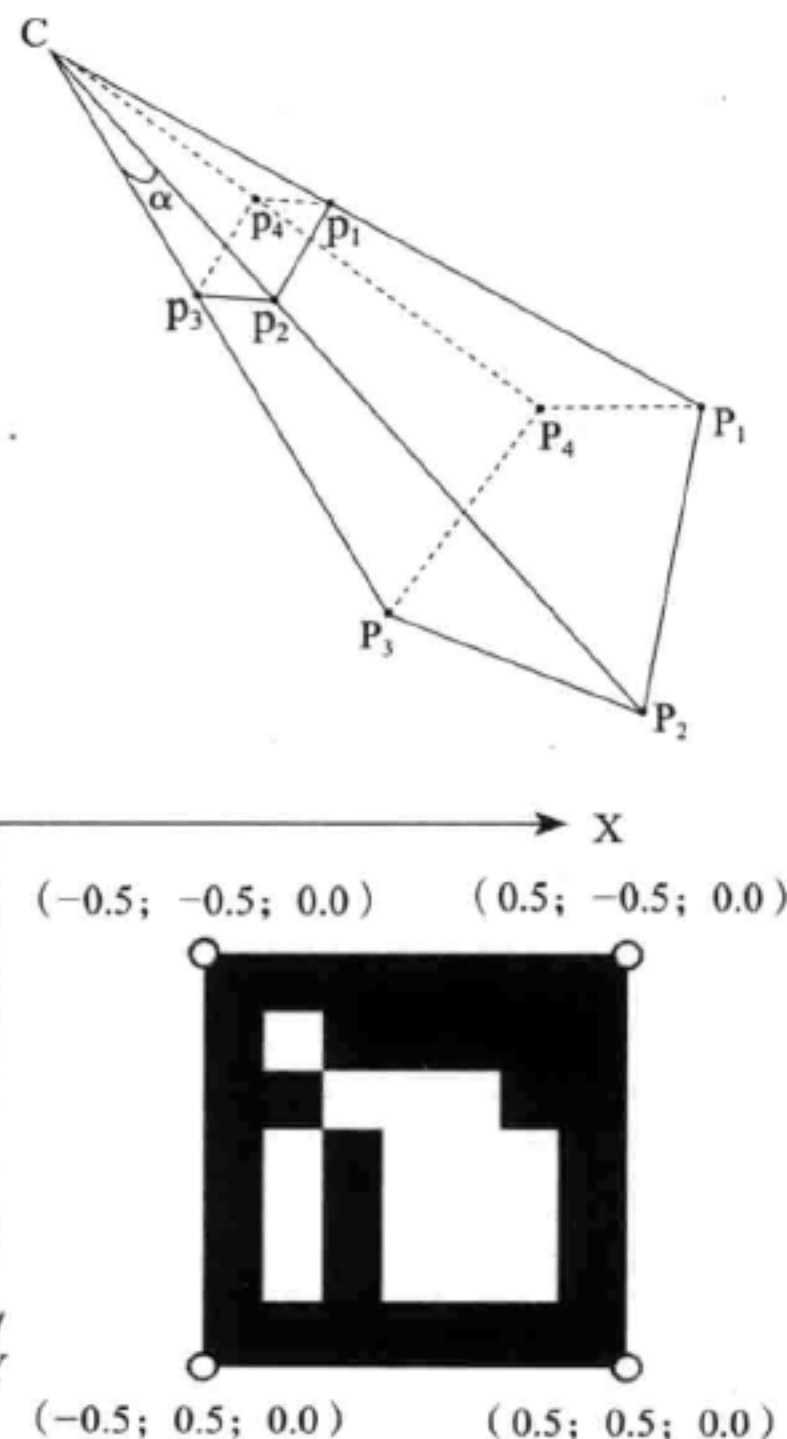
可将标记放在  $XY$  平面上 ( $Z$  分量为零)，其标记的中心为点  $(0.0, 0.0, 0.0)$ 。这是一个很好的提示，因为在这种情形下，坐标系统的起始处就是标记的中心 ( $Z$  轴是与标记平面垂直)。

为了用已知 2D-3D 的对应关系来找到摄像机位置，可用 `cv::solvePnP` 函数：

```

void solvePnP(const Mat& objectPoints, const Mat& imagePoints, const
Mat&
cameraMatrix, const Mat& distCoeffs, Mat& rvec, Mat& tvec, bool
useExtrinsicGuess=false);

```





其中，参数 `objectPoints` 是输入数组，它包含着对象所在空间的点，这里可将 `std::vector<cv::Point3f>` 类型所定义的变量传递给该参数。只要是  $3 \times N$  或  $N \times 3$  的 OpenCV 矩阵也都作为该参数的输入，其中  $N$  是点的数量。本项目将三维空间中标记坐标的列表传递给该参数。

数组 `imagePoints` 是相应图像点（或投影）数组。该参数的类型也是 `std::vector<cv::Point2f>` 或 `cv::Mat`，其大小为  $2 \times N$  或  $N \times 2$  的矩阵， $N$  表示点的数量。本项目将找到的标记顶点列表传递给该参数。

- ❑ `cameraMatrix` 是  $3 \times 3$  的摄像机内部参数矩阵；
- ❑ `distCoeffs` 是  $4 \times 1$ 、 $1 \times 4$ 、 $5 \times 1$  或  $1 \times 5$  的畸变向量；
- ❑ `rvec` 是输出的旋转向量，该向量将点由模型坐标系统转换为摄像机坐标系统；
- ❑ `tvec` 是输出的平移向量；
- ❑ 变量 `useExtrinsicGuess` 的值为 `true`，函数将使用所提供的 `rvec` 和 `tvec` 向量作为旋转和平移的初始值，然后会不断优化它们。

函数按最小化重投影误差来计算摄像机的变换，即让所得的投影向量 `imagePoints` 与被投影向量之间的距离平方和最小。

被估计的变换由旋转向量（`rvec`）和平移向量（`tvec`）来定义，这也称为欧氏变换或刚性变换。

刚性变换的正式定义为：对任意向量  $v$ ，会得到一个变换向量  $T(v)$ ，其具体形式为

$$T(v) = Rv + t$$

其中， $R^T = R^{-1}$ （即  $R$  为正交变换）， $t$  是一个平移向量，它给出了整个变换的起始点。另外，刚性变换还有一个性质：

$$\det(R) = 1$$

这意味着  $R$  只是旋转（一个保持方向的正交变换）并不会产生反射。

可使用函数 `cv::Rodrigues` 将旋转向量转换为  $3 \times 3$  的旋转矩阵。该函数将旋转向量转变为与之等价的旋转矩阵。



**注意：**`cv::solvePnP` 将在三维空间找到与标记姿态的摄像机位置，因此需要反转所得到的变换。所得变换将用摄像机的坐标系统来描绘标记变换，该坐标系统对渲染过程更方便。

下面给出了函数 `estimatePosition` 的实现过程，它会找到给定标记的位置。

```
void MarkerDetector::estimatePosition(std::vector<Marker>&
detectedMarkers)
{
    for (size_t i=0; i<detectedMarkers.size(); i++)
    {
        Marker& m = detectedMarkers[i];
```

```

cv::Mat Rvec;
cv::Mat_<float> Tvec;
cv::Mat raux,taux;
cv::solvePnP(m_markerCorners3d, m.points, camMatrix,
distCoeff,raux,taux);
    raux.convertTo(Rvec,CV_32F);
    taux.convertTo(Tvec ,CV_32F);

cv::Mat_<float> rotMat(3,3);
cv::Rodrigues(Rvec, rotMat);

// Copy to transformation matrix
m.transformation = Transformation();

for (int col=0; col<3; col++)
{
    for (int row=0; row<3; row++)
    {
        m.transformation.r().mat[row][col] = rotMat(row,col); // Copy
rotation
        component
    }
    m.transformation.t().data[col] = Tvec(col); // Copy translation
    component
}

// Since solvePnP finds camera location, w.r.t to marker pose, to
get
marker pose w.r.t to the camera we invert it.
m.transformation = m.transformation.getInverted();
}

```

## 2.5 渲染 3D 虚拟物体

到目前为止，已经介绍了如何找到图像的标记，然后计算它们相对于摄像机的精确空间位置。接下来将在标记上画一些物体。如前所述，会采用 OpenGL 函数来渲染场景。3D 可视化是增强现实的核心部分。OpenGL 为创建高质量渲染提供了所有的基本功能。

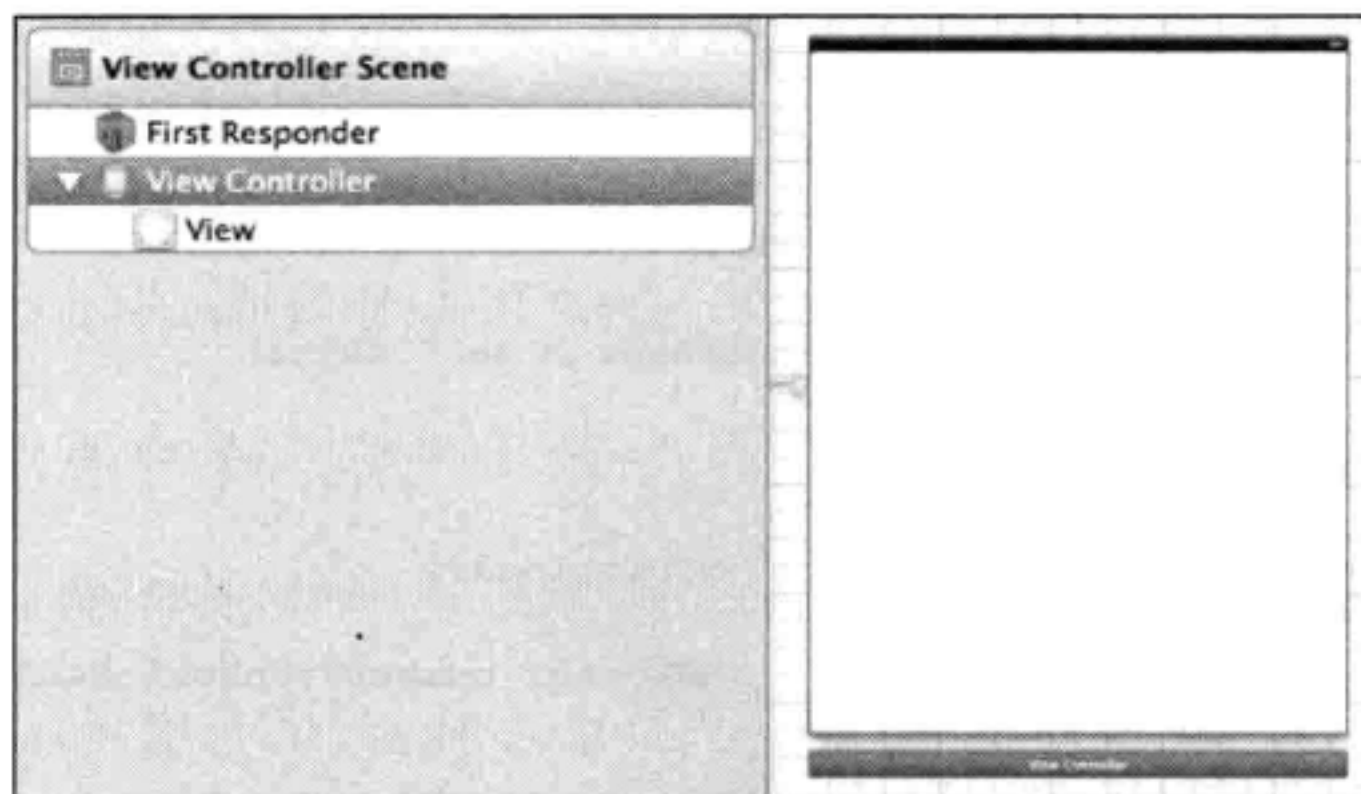


**注意：**有许多商业和开源的 3D 引擎（如：Unity、Unreal Engine、Ogre，等等），但所有这些引擎要么使用 OpenGL，要么使用 DirectX 来控制显卡。DirectX 有专有 API 且仅支持 Windows 平台。因此，OpenGL 是创建跨平台渲染系统的唯一选择。

### 2.5.1 创建 OpenGL 渲染层

为了在应用中使用 OpenGL 函数，需要获得 iOS 图形上下文表面（graphics context

surface), 该表面将给用户呈现渲染场景。用户所看到的这种上下文通常与 View 相关。下图是在 XCode 的 Interface Builder 中的应用程序界面的层次结构。



下面引入 EAGLView 类来封装 OpenGL 上下文的初始化逻辑:

```
@class EAGLContext;

// This class wraps the CAEAGLLayer from CoreAnimation into a
convenient UIView subclass.
// The view content is basically an EAGL surface you render your
OpenGL scene into.
// Note that setting the view non-opaque will only work if the EAGL
surface has an alpha channel.
@interface EAGLView : UIView
{
@private
    // The OpenGL ES names for the framebuffer and renderbuffer used
    to render
    to this view.
    GLuint defaultFramebuffer, colorRenderbuffer;
}

@property (nonatomic, retain) EAGLContext *context;
// The pixel dimensions of the CAEAGLLayer.
@property (readonly) GLint framebufferWidth;
@property (readonly) GLint framebufferHeight;

- (void)setFramebuffer;
- (BOOL)presentFramebuffer;
- (void)initContext;
@end
```

该类与接口定义文件中的 View 相关, 因此, 当加载 NIB 文件时, 会实例化一个新的



EAGLView 实例，这将会接收 iOS 的事件并初始化 OpenGL 渲染上下文。

下面是函数 initWithCoder 的实现代码：

```
//The EAGL view is stored in the nib file. When it's unarchived it's
sent -initWithCoder:.
- (id)initWithCoder:(NSCoder*)coder
{
    self = [super initWithCoder:coder];
    if (self) {
        CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;

        eaglLayer.opaque = TRUE;

        eaglLayer.drawableProperties = [NSDictionary
dictionaryWithObjectsAndKeys:
                                [NSNumber numberWithBool:FALSE],
                                kEAGLDrawablePropertyRetainedBacking,
                                kEAGLColorFormatRGBA8,
                                kEAGLDrawablePropertyColorFormat,
                                nil];

        [self initContext];
    }

    return self;
}

- (void)createFramebuffer
{
    if (context && !defaultFramebuffer) {
        [EAGLContext setCurrentContext:context];

        // Create default framebuffer object.
        glGenFramebuffers(1, &defaultFramebuffer);
        glBindFramebuffer(GL_FRAMEBUFFER, defaultFramebuffer);

        // Create color render buffer and allocate backing store.
        glGenRenderbuffers(1, &colorRenderbuffer);
        glBindRenderbuffer(GL_RENDERBUFFER, colorRenderbuffer);
        [context renderbufferStorage:GL_RENDERBUFFER
fromDrawable:(CAEAGLLayer *)self.layer];
        glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_
WIDTH, &framebufferWidth);
        glGetRenderbufferParameteriv(GL_RENDERBUFFER, GL_RENDERBUFFER_
HEIGHT, &framebufferHeight);

        glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_RENDERBUFFER, colorRenderbuffer);

        if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_
```

```

COMPLETE)
    NSLog(@"Failed to make complete framebuffer object %x",
          glCheckFramebufferStatus(GL_FRAMEBUFFER));

    //glClearColor(0, 0, 0, 0);
    NSLog(@"Framebuffer created");
}
}

```

## 2.5.2 渲染 AR 场景

从前面的代码可以看出，EAGLView 类没包含 3D 对象和视频的可视化方法，这是因为 EAGLView 只提供渲染上下文。这样将功能分开有利于后面改变可视化逻辑。

为了对增强现实进行可视化，可单独创建一个名为 VisualizationController 的接口：

```

@interface SimpleVisualizationController : NSObject<VisualizationCont
roller>
{
    EAGLView * m_glview;
    GLuint m_backgroundTextureId;
    std::vector<Transformation> m_transformations;
    CameraCalibration m_calibration;
    CGSize m_frameSize;
}

-(id) initWithGLView:(EAGLView*)view calibration:(CameraCalibration)
calibration frameSize:(CGSize) size;

-(void) drawFrame;
-(void) updateBackground:(BGRAVideoFrame) frame;
-(void) setTransformationList:(const std::vector<Transformation>&)
transformations;

```

函数 drawFrame 负责在给定 EAGLView 目标视图上进行 AR 渲染，其执行步骤如下：

- 1) 清除场景；
- 2) 在绘制的背景上执行正交投影；
- 3) 绘制从一个摄像机视角接收到的最新图像；
- 4) 通过摄像机内部参数执行透视投影；
- 5) 对每个检测到的标记，将其坐标系移动到三维空间的标记位置（即将  $4 \times 4$  的变换矩阵作为 OpenGL 模式 - 视图矩阵）；
- 6) 渲染任意的三维物体；
- 7) 显示帧缓冲区的内容。

当要绘制的帧准备好后，就可调用 drawFrame 函数。当新的摄像机帧已经加载到视频级缓冲区中，并且标记检测阶段已完成，就可认为要绘制的帧准备好了。

下面的代码是 drawFrame 函数的实现：

```
- (void)drawFrame
{
    // Set the active framebuffer
    [m_glview setFramebuffer];

    // Draw a video on the background
    [self drawBackground];

    // Draw 3D objects on the position of the detected markers
    [self drawAR];

    // Present framebuffer
    bool ok = [m_glview presentFramebuffer];

    int glErrorCode = glGetError();
    if (!ok || glErrorCode != GL_NO_ERROR)
    {
        std::cerr << "GL error detected. Error code:" << glErrorCode <<
std::endl;
    }
}
```

可通过执行正交投影并将当前帧图像作为屏幕纹理来轻松绘制背景。下面是使用 GLES 1 API 的代码，它可实现这样的功能：

```
- (void) drawBackground
{
    GLfloat w = m_glview.bounds.size.width;
    GLfloat h = m_glview.bounds.size.height;
    const GLfloat squareVertices[] =
    {
        0, 0,
        w, 0,
        0, h,
        w, h
    };

    static const GLfloat textureVertices[] =
    {
        1, 0,
        1, 1,
        0, 0,
        0, 1
    };

    static const GLfloat proj[] =
    {
        0, -2.f/w, 0, 0,
```



```

        -2.f/h, 0, 0, 0,
        0, 0, 1, 0,
        1, 1, 0, 1
    };

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(proj);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glDisable(GL_COLOR_MATERIAL);

    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, m_backgroundTextureId);

    // Update attribute values.
    glVertexPointer(2, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glTexCoordPointer(2, GL_FLOAT, 0, textureVertices);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glColor4f(1,1,1,1);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisable(GL_TEXTURE_2D);
}

```

在场景中渲染人造物体需要一些技巧。首先需要用摄像机的内（标定）矩阵来调整 OpenGL 投影矩阵。没有这一步，会得到错误的透视投影，这会使人造物体看上去不真实，这些物体看上去就像“悬在空中”，而不是现实世界的一部分。这是任何一个增强现实应用都要注意的问题。

下面这段代码是由摄像机标定矩阵所创建的 OpenGL 投影矩阵。

```

- (void)buildProjectionMatrix:(Matrix33)cameraMatrix: (int)screen_
width: (int)screen_height: (Matrix44&) projectionMatrix
{
    float near = 0.01; // Near clipping distance
    float far = 100; // Far clipping distance

    // Camera parameters
    float f_x = cameraMatrix.data[0]; // Focal length in x axis
    float f_y = cameraMatrix.data[4]; // Focal length in y axis (usually
the
    same?)
    float c_x = cameraMatrix.data[2]; // Camera primary point x
    float c_y = cameraMatrix.data[5]; // Camera primary point y

```

```

projectionMatrix.data[0] = - 2.0 * f_x / screen_width;
projectionMatrix.data[1] = 0.0;
projectionMatrix.data[2] = 0.0;
projectionMatrix.data[3] = 0.0;

projectionMatrix.data[4] = 0.0;
projectionMatrix.data[5] = 2.0 * f_y / screen_height;
projectionMatrix.data[6] = 0.0;
projectionMatrix.data[7] = 0.0;

projectionMatrix.data[8] = 2.0 * c_x / screen_width - 1.0;
projectionMatrix.data[9] = 2.0 * c_y / screen_height - 1.0;
projectionMatrix.data[10] = -( far+near ) / ( far - near );
projectionMatrix.data[11] = -1.0;

projectionMatrix.data[12] = 0.0;
projectionMatrix.data[13] = 0.0;
projectionMatrix.data[14] = -2.0 * far * near / ( far - near );
projectionMatrix.data[15] = 0.0;
}

```

将这个矩阵传递给 OpenGL 后，就可以画一些人造物体了。每个变换可表示成  $4 \times 4$  的矩阵，并加载到 OpenGL 模型视图矩阵，这会将坐标系统变换成三维空间中的标记位置。

例如：可画一个垂直于每个标记的坐标轴，用它来表示空间中的方向，并用有渐变颜色的矩形覆盖在整个标记上。这种可视化效果会带给人们视觉上的冲击，这也是本项目要达到的效果。

下面这段代码是 drawAR 函数的实现：

```

- (void) drawAR
{
    Matrix44 projectionMatrix;
    [self buildProjectionMatrix:m_calibration.getIntrinsic():m_
frameSize.width
:m_frameSize.height :projectionMatrix];

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(projectionMatrix.data);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);

    glPushMatrix();
    glLineWidth(3.0f);

    float lineX[] = {0,0,0,1,0,0};
    float lineY[] = {0,0,0,0,1,0};
    float lineZ[] = {0,0,0,0,0,1};
}

```

```

const GLfloat squareVertices[] = {
    -0.5f, -0.5f,
    0.5f,  -0.5f,
    -0.5f,  0.5f,
    0.5f,   0.5f,
};
const GLubyte squareColors[] = {
    255, 255,  0, 255,
    0,   255, 255, 255,
    0,    0,  0,  0,
    255,  0, 255, 255,
};

for (size_t transformationIndex=0;
transformationIndex<m_transformations.size(); transformationIndex++)
{
    const Transformation& transformation =
        m_transformations[transformationIndex];

    Matrix44 glMatrix = transformation.getInverted().getMat44();

    glLoadMatrixf(reinterpret_cast<const GLfloat*>(&glMatrix.
data[0]));

    // draw data
    glVertexPointer(2, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColorPointer(4, GL_UNSIGNED_BYTE, 0, squareColors);
    glEnableClientState(GL_COLOR_ARRAY);

    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glDisableClientState(GL_COLOR_ARRAY);

    float scale = 0.5;
    glScalef(scale, scale, scale);

    glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
    glVertexPointer(3, GL_FLOAT, 0, lineX);
    glDrawArrays(GL_LINES, 0, 2);

    glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
    glVertexPointer(3, GL_FLOAT, 0, lineY);
    glDrawArrays(GL_LINES, 0, 2);

    glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
    glVertexPointer(3, GL_FLOAT, 0, lineZ);
    glDrawArrays(GL_LINES, 0, 2);
}

glPopMatrix();
glDisableClientState(GL_VERTEX_ARRAY);
}

```

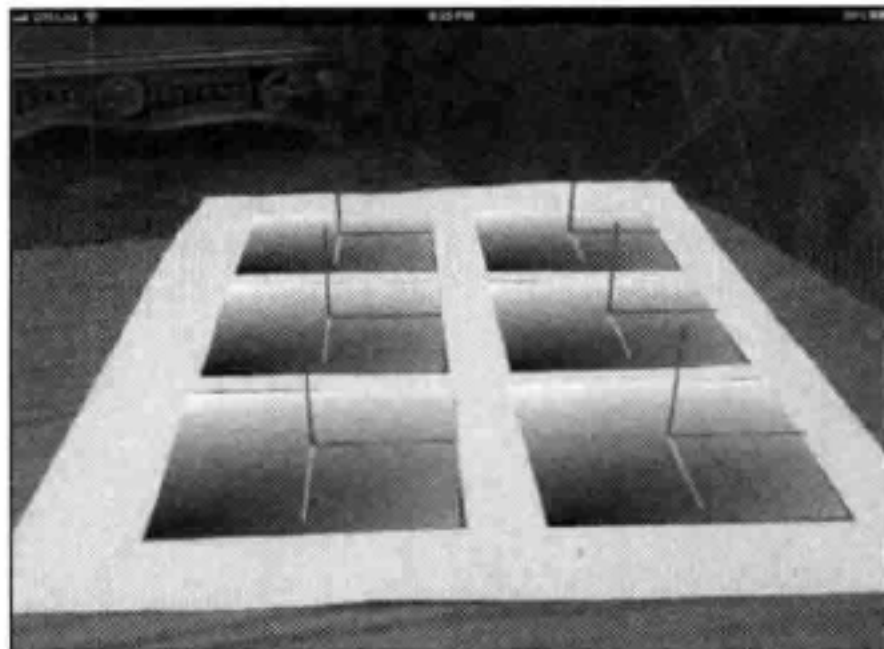


如果运行本项目，便可得到右图所示的效果。

虽然对场景可视化没有使用特殊的 3D 渲染引擎，但本项目拥有自定义 3D 渲染所需的所有数据。本项目所能提供的数据有：

- ❑ 来自摄像机，且基于 BGRA 格式的帧；
- ❑ 正确的投影矩阵，它能针对 AR 场景渲染得到合适的透视投影；
- ❑ 发现一系列标记姿态。

读者可在任何 3D 引擎上使用这些数据来创建基于标记的 AR 应用。



从上图可看出，渐变的矩形和三个坐标轴都能很精确地放置在标记上。这是增强现实的关键，即现实了图像与人造物体的无缝融合。

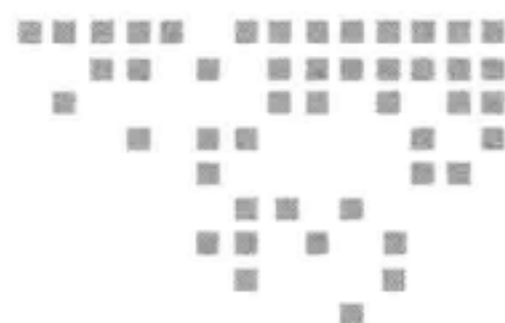
## 2.6 总结

本章介绍了如何针对 iPhone/iPad 设备来创建一个增强现实应用。读者可从中学到如何在 XCode 开发环境中使用 OpenCV 库来创建具有震撼力的应用。在移动设备上，可使用 OpenCV 来完成复杂、实时的图像处理。

本章也介绍如何处理初始化图像（对灰度阴影图像进行平移并进行二值化处理）；如何找到闭轮廓并用多边形来逼近它们；如何在图像中查找标记并对其进行编码；如何在空间中计算标记位置和在增强现实中实现 3D 物体可视化。

## 2.7 参考文献

- *ArUco: a minimal library for Augmented Reality applications based on OpenCV* (<http://www.uco.es/investiga/grupos/ava/node/26>)
- *OpenCV Camera Calibration and 3D Reconstruction* ([http://opencv.itseez.com/modules/calib3d/doc/camera\\_calibration\\_and\\_3d\\_reconstruction.html](http://opencv.itseez.com/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html))
- *OpenCV: Estimating Projective Relations in Images* (<http://www.packtpub.com/article/opencv-estimating-projective-relations-images>)
- *Multiple View Geometry in Computer Vision (second edition)*, R.I. Hartley and A. Zisserman, Cambridge University Press, ISBN 0-521-54051-8



## 无标记的增加现实

本章读者将学习如何使用 OpenCV 创建标准的真实项目（运行在台式机上），以及如何实现一种新的无标记增强现实方法，该方法不是将印有方块的标记而是将实际环境作为输入。本章将首先介绍一些无标记增强现实理论，然后介绍如何将其应用到实际的项目中。

以下是本章将涉及的主题：

- ❑ 基于标记的 AR 与无标记的 AR；
- ❑ 使用特征描述符检测视频中的任意图像；
- ❑ 模式姿态估计；
- ❑ 应用程序框架；
- ❑ 在 OpenCV 中启用 OpenGL 的可视化支持；
- ❑ 渲染增强现实；
- ❑ 演示。

在开始之前，需要简要列出本章所需知识以及软件：

- ❑ CMake 的基础知识。CMake 是一个跨平台的开源软件，它可用来生成编译配置、测试和对软件进行打包。像 OpenCV 的库一样，本章示例项目也是用 CMake 构建的。CMake 可从 <http://www.cmake.org/> 下载。
- ❑ C++ 编程语言的基础知识。但所有复杂的应用程序源代码都将被详细说明。

### 3.1 基于标记的 AR 与无标记的 AR

上一章已经介绍了如何使用基于标记的特殊图像来增强一个真实场景。使用标记的优

点是：

- ❑ 检测算法简单；
  - ❑ 对光线变化鲁棒性好。
- 标记也有不足。具体表现如下：
- ❑ 如果出现部分重叠它将不会工作；
  - ❑ 标记的图像必须是黑色和白色；
  - ❑ 在大多数情况下，标记的图像要有方形（因为它很容易被检测到）；
  - ❑ 标记的视觉外观不美观（non-esthetic）；
  - ❑ 与真实世界的对象没有共同点。

因此，标记是增强现实工作的一个很好的起始点，但如果想了解更多，需要从基于标记 AR 转移到无标记的 AR。无标记 AR 是基于现实世界的对象识别技术。可采用无标记 AR 的几个例子：杂志封面、公司标志、玩具等。通常，若任何对象对场景其他部分有足够多的描述和判别信息就能成为无标记 AR 的对象。

无标记 AR 的优点是：

- ❑ 可用于检测现实世界的对象；
- ❑ 即使目标对象部分重叠也可工作；
- ❑ 可以有任意的形状和纹理（实心或平滑渐变纹理除外）。

无标记 AR 系统可在三维空间使用真实图像和对象来定位摄像机，然后在真实的图像顶部呈现绚丽的效果。无标记 AR 的核心是图像识别和目标检测算法。对于基于标记的 AR 而言，其标记的形状和内部结构都是固定的，但现实世界的对象不能由这种方式来定义。此外，对象可以有一个复杂的形状，需要修改姿态估计算法来找到它们正确的三维变换。



**注意：**为了介绍无标记 AR 的思想，这里将使用一个平面图像作为目标。这里将不考虑有复杂形状的对象。本章后面将讨论在 AR 中使用复杂形状。

无标记 AR 计算量很大，所以移动设备往往不能够确保流畅的 FPS。本章将针对桌面平台，如：PC 或 Mac。出于这个目的，需要一个跨平台的编译系统。这里将使用 CMake 来构建系统。

### 3.2 使用特征描述符检测视频中的任意图像

图像识别是一种计算机视觉技术，这种技术按一个具体的位图模式来对输入图像进行搜索。图像识别算法应能检测所需模式，即便输入图像与原始图像相比，被缩放、旋转，或具有不同的亮度。

如何将模式图像与其他图像进行对比？由于该模式图像会受到透视变换影响，显然不能直接将模式图像与测试图像按像素比较。在这种情况下，使用特征点和特征描述符就能



很好地解决该问题。目前关于特征是什么，没有普遍、确切的定义，若有，往往取决于问题或应用的类型。通常的特征被定义为一幅图像中“有意义”的部分，特征是许多计算机视觉算法的基础。本章将使用术语特征点（feature point）。它是图像的一部分，由一个圆心、半径和方向来定义。每个特征检测算法都尝试检测相同的特征点而不管是否采用了透视变换。

### 3.2.1 特征提取

特征检测是从输入图像中找到感兴趣的区域。有很多的特征检测算法，这些算法用来搜索边缘、角点或斑点（blob）。本章感兴趣的是角点（corner detection）检测。角点检测分析图像中的边缘。角点边缘检测算法在图像梯度中搜索变化快的梯度，通常采用的方法是在 X 和 Y 方向寻找图像梯度的一阶导数的极值。

特征点的方向通常通过计算一个特定区域内主要图像（dominant image）梯度的方向来得到。当图像被旋转或缩放，主要梯度方向由特征检测算法重新计算。这意味着不管图像是否旋转，特征点的方向也不会改变。这种功能被称为旋转不变性（rotation invariant）。

另外，还需提一下特征点的大小问题。一些特征检测算法使用固定大小的特征，而另一些算法则分别对每个关键点计算最佳大小。知道了特征大小，就可找到被缩放图像的相同特征点。这就叫特征尺度不变性。

OpenCV 中有几个特征检测算法。这些算法都来源于基类 `cv::FeatureDetector`。创建特征检测算法可用下面这两种方式来完成：

❑ 通过具体特征检测器所在类的构造函数显式调用：

```
cv::Ptr<cv::FeatureDetector> detector =
cv::Ptr<cv::FeatureDetector>(new cv::SurfFeatureDetector());
```

❑ 或通过算法名来创建特征检测器：

```
cv::Ptr<cv::FeatureDetector> detector =
cv::FeatureDetector::create("SURF");
```

这两种方法各有优点，所以读者可选择最喜欢的一种来创建。显式方式创建允许传递额外参数给特征检测器的构造函数，而通过算法名称创建可在运行时更容易地切换算法。

为了检测特征点，应该调用 `detect` 方法：

```
std::vector<cv::KeyPoint> keypoints;
detector->detect(image, keypoints);
```

检测到的特征点被放在关键点（keypoint）容器中。每个关键点包含其圆心、半径、角度和得分，并与特征点的“质量”或“强度”有一些相关性。每个特征检测算法都有自己的评分计算方法，其中一种可行的方法是比较检测算法检测到的关键点的分数。



**注意：**基于角点的特征检测会使用灰度图像来查找特征点，描述符 - 提取 (descriptor-extraction) 算法也需要灰度图像。当然，它们都可做隐式色彩转换。但这种情况下的色彩转换将被执行两次。可通过执行显式颜色转换来将输入图像转换为灰度图像，特征检测和描述符的提取会利用灰度图像来提高性能。

模式检测的最好结果是由检测器计算关键点方向和大小来获得的。这样会使得关键点具有旋转不变性和伸缩不变性。最有名且鲁棒性好的关键点检测算法是基于 SIFT 和 SURF 的特征检测 / 描述提取的。不幸的是，它们都受专利保护，因此不能免费用于商业用途。但它们在 OpenCV 中已被实现，读者可自由使用它们。也有优秀且免费的替代品，它们是 ORB 或 FREAK 算法。ORB 检测是一种 FAST 特征检测器的改进。原始的 FAST 检测器非常快，但它不能计算方向或关键点的大小。幸运的是，ORB 算法可估计关键点的方向，但仍需固定特征大小。下面将介绍一种简单且效果好的方法来解决 ORB 算法的这个问题。但首先需要解释为什么特征点在图像识别中如此重要。

如果处理图像，它的每个像素通常有 24 位颜色值，若分辨率为  $640 \times 480$ ，则图像大小为 912KB。如何找到现实世界中的模式图像？按像素匹配会花很长的时间，并且仍需处理旋转和缩放。这绝对不是一种好的选择。使用特征点可以解决这个问题。通过检测关键点，其得到的特征对图像各部分都有描述，这包含大量信息（这是因为角点探测器会得到边缘，角点和其他有非明显变化的图像）。因此，要找到两帧之间的对应关系，只需要匹配关键点。

其中一种特征点表示形式为描述符向量，即由关键点来定义图像块，由此组成描述符向量。有许多通过特征点来得到描述符的方法。它们各有长处和短处。例如，基于 SIFT 和 SURF 的描述符 - 提取算法计算量很大，但它们优秀的判别能力使其具有很好的鲁棒性。由于在本章的示例项目中将 ORB 作为一个特征检测器，因此会用到 ORB 描述符 - 提取算法。



**注意：**用同样的算法来进行特征检测和描述符 - 提取一直都是不错的方法。因为它们彼此能很好地配合。

特征描述符都是用固定大小（16 个或更多的元素）的向量来表示的。比如，一幅分辨率为  $640 \times 480$  像素的图像，会有 1500 个特征点。那么特征描述符的大小为  $1500 \times 16 \times \text{sizeof}(\text{float}) = 96 \text{ KB}$ （对 SURF 算法而言）。这是原始图像的 1/10。此外，用描述符而不是光栅位图 (raster bitmap) 会更容易进行操作。可对两个特征描述的相似性进行评分。这种评分通常会使用它们的 L2 范数或海明距离（这根据所使用的特征描述类型而定）。

特征描述符 - 提取算法由基类 `CV_::DescriptorExtractor` 派生出来。同样，特征检测算法可通过指定名称或显式的构造函数调用来创建。

### 3.2.2 模式对象定义

为了描述一个模式对象，需引入一个名为 `pattern` 的类，它拥有训练图像、特征和提取描述符列表，以及针对初始模式位置在二维和三维情形下的一致性。

```
/**
 * Store the image data and computed descriptors of target pattern
 */
struct Pattern
{
    cv::Size          size;
    cv::Mat           data;
    std::vector<cv::KeyPoint> keypoints;
    cv::Mat           descriptors;

    std::vector<cv::Point2f>  points2d;
    std::vector<cv::Point3f>  points3d;
};
```

### 3.2.3 特征点匹配

寻找帧与帧之间对应关系的过程，可看成一组描述符与另一组中每一个元素的最近邻搜索。这就是所谓的“匹配”过程。在 OpenCV 中描述符匹配的算法主要有两种：

❑ 暴力匹配 (brute-force matcher)(`cv::BFMatcher`)

❑ 基于 flann 的匹配 (`cv::FlannBasedMatcher`)

蛮力匹配是指依次查找（穷举搜索）第一组中每个描述符与第二组中哪个描述符最接近。`CV::FlannBasedMatcher` 采用快速近似最近邻搜索算法找到对应的匹配（为了完成这样的匹配，采用高效的第三方库来完成近似最近邻的查找）。

描述符匹配的结果会得到两个描述符集合之间对应关系的列表。第一组描述符通常被称为训练集，因为它有相应的模式图像。第二组被称为查询集，因为它拥有用于需要查找的模式。正确的匹配越多（说明图像上的模式就越多），给定模式在图像上就越有机会被匹配到。

为了提高匹配速度，可在调用匹配函数之前，训练一个匹配器。训练阶段是用来优化 `CV::FlannBasedMatcher` 的性能。为了完成训练，`train` 类将建立描述符的索引树。这会提高大型数据集的匹配速度（例如，如果想从数百幅图像中找到匹配）。对于 `CV::BFMatcher`，`train` 类不需要做什么，因为没有预处理，它只需要在内部字段（`internal field`）中存储训练描述符即可。

#### **PatternDetector.cpp**

下面这段代码使用模式图像来训练描述符匹配器：



```

void PatternDetector::train(const Pattern& pattern)
{
    // Store the pattern object
    m_pattern = pattern;

    // API of cv::DescriptorMatcher is somewhat tricky
    // First we clear old train data:
    m_matcher->clear();

    // That we add vector of descriptors
    // (each descriptors matrix describe one image).
    // This allows us to perform search across multiple images:
    std::vector<cv::Mat> descriptors(1);
    descriptors[0] = pattern.descriptors.clone();
    m_matcher->add(descriptors);

    // After adding train data perform actual train:
    m_matcher->train();
}

```

为了匹配查询描述符，可使用 `cv::DescriptorMatcher` 类中的一种方法，这些方法如下：

❑ 找到最佳匹配的简单列表：

```

void match(const Mat& queryDescriptors,
           vector<DMatch>& matches,
           const vector<Mat>& masks=vector<Mat>() );

```

❑ 找到每个描述符的  $K$  近邻匹配：

```

void knnMatch(const Mat& queryDescriptors,
              vector<vector<DMatch> >& matches, int k,
              const vector<Mat>& masks=vector<Mat>(),
              bool compactResult=false );

```

❑ 找到相应距离不比指定距离远：

```

void radiusMatch(const Mat& queryDescriptors,
                 vector<vector<DMatch> >& matches, maxDistance,
                 const vector<Mat>& masks=vector<Mat>(),
                 bool compactResult=false );

```

### 3.2.4 删除离群值

在匹配阶段很可能发生误匹配。在匹配过程中有两种错误：

- ❑ 假阳性匹配 (false-positive match)：对应的特征点是错误的；
- ❑ 假阴性匹配 (false-negative match)：两个图像上的特征点无法匹配。

假阴性匹配明显不好，但不能处理它们，因为匹配算法已经认为它们之间没有匹配。因此，要尽量减少假阳性匹配的数量。为了尽量减少错误的对应关系，可以交叉匹配技术，这种技术的思想是用查询集来匹配训练描述符，反之亦然。只返回在这两个匹配中同

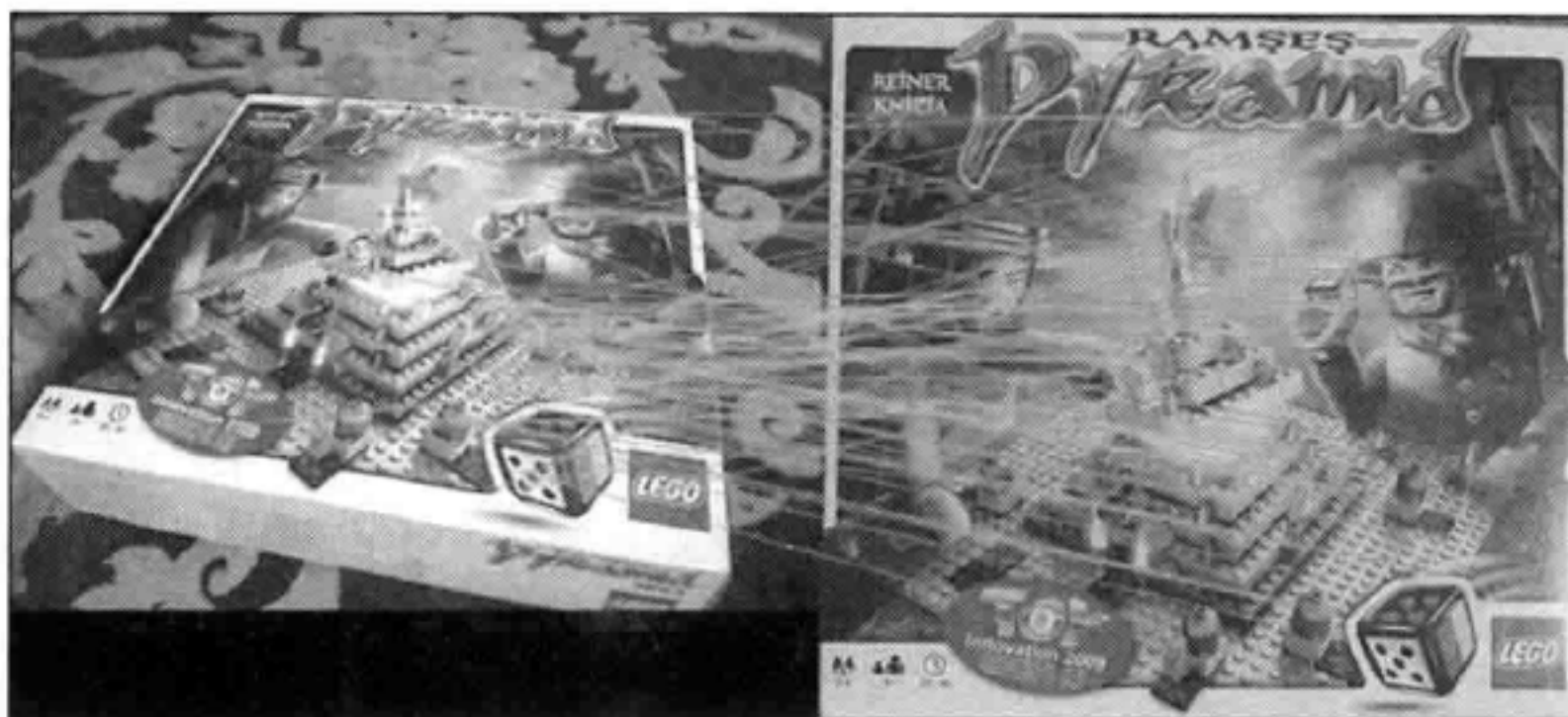
时出现的匹配。当有足够多的匹配时，这种技术在离群值数目极少的情况下通常会产生最佳效果。

### 1. 交叉匹配过滤

在 `cv::BFMatcher` 类中可进行交叉匹配。为了能进行交叉检测实验，要创建 `cv::BFMatcher` 类的实例，需将构造函数的第二个参数设置为 `true`：

```
cv::Ptr<cv::DescriptorMatcher>
matcher(new cv::BFMatcher(cv::NORM_HAMMING, true));
```

下面的截图展示了交叉检测得到的匹配结果：



### 2. 比率测试

第二个著名的离群值删除技术是比率测试。可用 KNN 匹配，最初的  $K$  为 2，即对每个匹配返回两个最近邻描述符。仅当第一个匹配与第二个匹配之间的距离比率足够大时（比率的阈值通常为 2 左右），才认为这是一个匹配。

#### PatternDetector.cpp

下面代码在进行描述符匹配时使用了比率测试，其鲁棒性很好：

```
void PatternDetector::getMatches(const cv::Mat& queryDescriptors,
std::vector<cv::DMatch>& matches)
{
    matches.clear();

    if (enableRatioTest)
    {
        // To avoid NaNs when best match has
        // zero distance we will use inverse ratio.
        const float minRatio = 1.f / 1.5f;

        // KNN match will return 2 nearest
        // matches for each query descriptor
        m_matcher->knnMatch(queryDescriptors, m_knnMatches, 2);

        for (size_t i=0; i<m_knnMatches.size(); i++)
```

```

    {
        const cv::DMatch& bestMatch    = m_knnMatches[i][0];
        const cv::DMatch& betterMatch = m_knnMatches[i][1];

        float distanceRatio = bestMatch.distance /
                               betterMatch.distance;

        // Pass only matches where distance ratio between
        // nearest matches is greater than 1.5
        // (distinct criteria)
        if (distanceRatio < minRatio)
        {
            matches.push_back(bestMatch);
        }
    }
}
else
{
    // Perform regular match
    m_matcher->match(queryDescriptors, matches);
}
}

```

比率测试可删除几乎所有的异常值。但在某些情况下，假阳性匹配可通过这个测试。下一节将介绍如何去掉异常值的其余部分，只留下正确的匹配。

### 3. 估计单应性

为了得到更多的匹配，可采用随机采样一致性（RANSAC）方法来进行异常过滤。若希望所使用的图像（可看成是平面对象）是刚性的，只需在模式图像的特征点与查询图像的特征点之间找到单应性变换即可。单应性变换会将模式中的点变换到用作查询的图像坐标系中。为了找到这样的变换，可使用 `CV::findHomography` 函数。它使用 RANSAC 来探测输入点的子集，由此找到最好的单应性矩阵。这种方法有一个缺点：该函数通过计算单应性矩阵的重投影误差来标记每个点是否为离群值或非离群值。

#### PatternDetector.cpp

下面的代码通过 RANSAC 算法进行单应性矩阵估计，其目的是过滤在几何上不正确的匹配：

```

bool PatternDetector::refineMatchesWithHomography
(
    const std::vector<cv::KeyPoint>& queryKeypoints,
    const std::vector<cv::KeyPoint>& trainKeypoints,
    float reprojectionThreshold,
    std::vector<cv::DMatch>& matches,
    cv::Mat& homography
)
{
    const int minNumberMatchesAllowed = 8;

    if (matches.size() < minNumberMatchesAllowed)

```



```

    return false;

    // Prepare data for cv::findHomography
    std::vector<cv::Point2f> srcPoints(matches.size());
    std::vector<cv::Point2f> dstPoints(matches.size());

    for (size_t i = 0; i < matches.size(); i++)
    {
        srcPoints[i] = trainKeypoints[matches[i].trainIdx].pt;
        dstPoints[i] = queryKeypoints[matches[i].queryIdx].pt;
    }

    // Find homography matrix and get inliers mask
    std::vector<unsigned char> inliersMask(srcPoints.size());
    homography = cv::findHomography(srcPoints,
                                    dstPoints,
                                    CV_FM_RANSAC,
                                    reprojectionThreshold,
                                    inliersMask);

    std::vector<cv::DMatch> inliers;
    for (size_t i=0; i<inliersMask.size(); i++)
    {
        if (inliersMask[i])
            inliers.push_back(matches[i]);
    }

    matches.swap(inliers);
    return matches.size() > minNumberMatchesAllowed;
}

```

下面是使用这种技术细化后的可视化匹配。



单应性搜索很重要，因为所得到的变换是在查询图像中发现模式位置的关键。

#### 4. 单应细化

当搜索单应性变换时，已经有了查找它们在三维空间位置的所有必需的数据。然而，可通过找到更精确的模式角点来提高其位置的精确性。这可通过将已找到的模式用到已估

计的单应性上，由此来扭曲输入图像以得到单应细化。结果应该是非常接近原来训练的图像。单应细化有助于找到更精确的单应变换。



然后，将得到另一个单应和另一组非离群值特征。第一个单应性（H1）与第二（H2）的单应性作矩阵乘法会得到精确的单应性。

#### PatternDetector.cpp

下面这段代码为模式检测程序的最终版本：

```
bool PatternDetector::findPattern(const cv::Mat& image,
PatternTrackingInfo& info)
{
    // Convert input image to gray
    getGray(image, m_grayImg);

    // Extract feature points from input gray image
    extractFeatures(m_grayImg, m_queryKeypoints,
        m_queryDescriptors);

    // Get matches with current pattern
    getMatches(m_queryDescriptors, m_matches);

    // Find homography transformation and detect good matches
    bool homographyFound = refineMatchesWithHomography(
        m_queryKeypoints,
        m_pattern.keypoints,
        homographyReprojectionThreshold,
        m_matches,
        m_roughHomography);

    if (homographyFound)
    {
        // If homography refinement enabled
        // improve found transformation
        if (enableHomographyRefinement)
        {
```

```

// Warp image using found homography
cv::warpPerspective(m_grayImg, m_warpedImg,
    m_roughHomography, m_pattern.size(),
    cv::WARP_INVERSE_MAP | cv::INTER_CUBIC);

// Get refined matches:
std::vector<cv::KeyPoint> warpedKeypoints;
std::vector<cv::DMatch> refinedMatches;

// Detect features on warped image
extractFeatures(m_warpedImg, warpedKeypoints,
    m_queryDescriptors);

// Match with pattern
getMatches(m_queryDescriptors, refinedMatches);

// Estimate new refinement homography
homographyFound = refineMatchesWithHomography(
    warpedKeypoints,
    m_pattern.keypoints,
    homographyReprojectionThreshold,
    refinedMatches,
    m_refinedHomography);

// Get a result homography as result of matrix product
// of refined and rough homographies:
info.homography = m_roughHomography *
    m_refinedHomography;

// Transform contour with precise homography
cv::perspectiveTransform(m_pattern.points2d,
    info.points2d, info.homography);
}
else
{
    info.homography = m_roughHomography;

    // Transform contour with rough homography
    cv::perspectiveTransform(m_pattern.points2d,
        info.points2d, m_roughHomography);
}
}

return homographyFound;
}

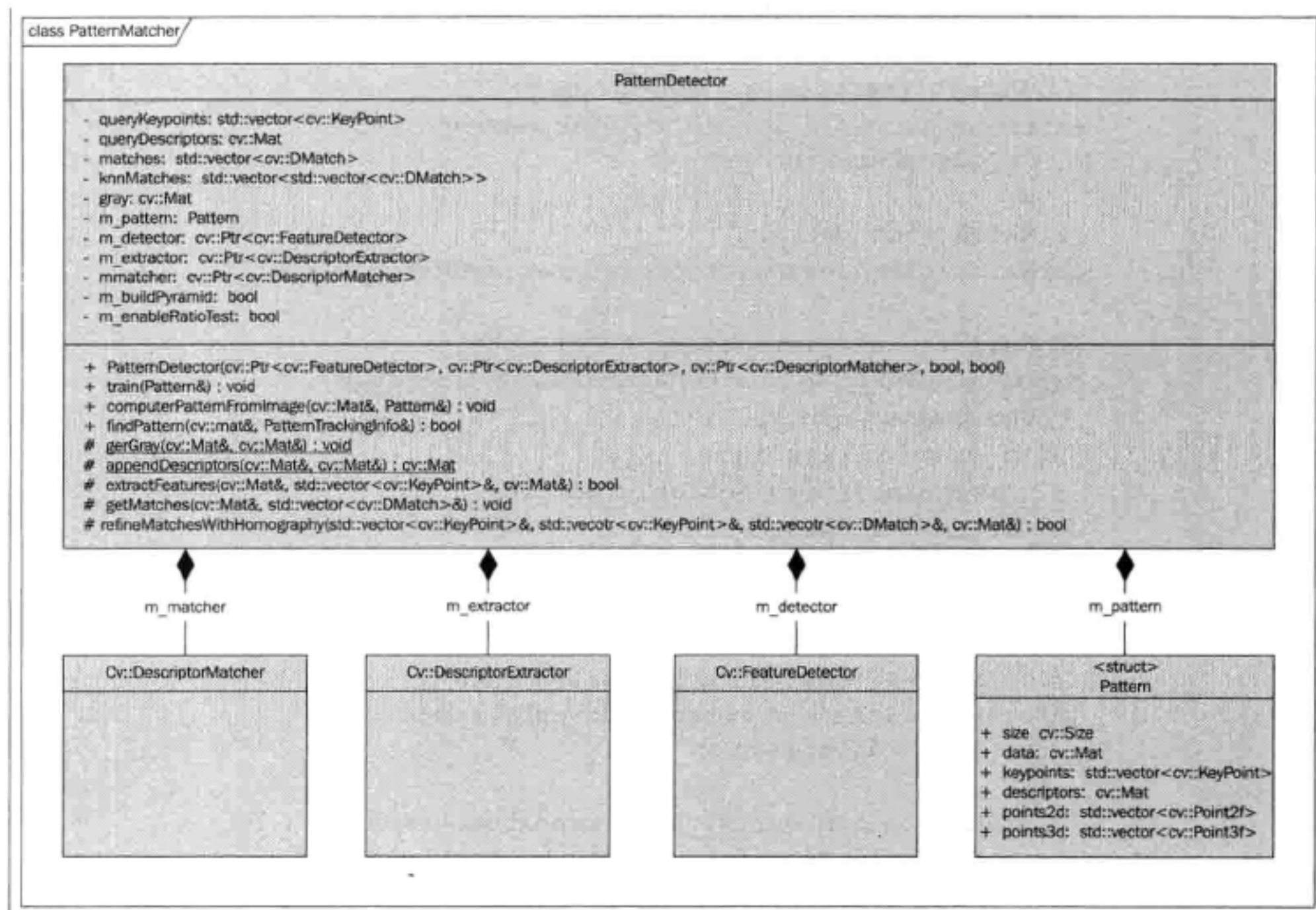
```

如果所有的离群值在这个阶段被删除，匹配的数目仍然保持合理的规模（至少模式图像有 25% 的特征与输入图像对应），并且还能正确找到模式图像，这样就可继续进行下一个阶段即摄像机中模式姿态在三维空间的位置估计。



### 3.2.5 将示例项目各部分放在一起

为了得到特征检测器、描述符提取以及匹配算法的实例，需创建一个 **PatternMatcher** 类，它将封装所有数据。该类包含特征检测、描述符 - 提取算法、特征匹配逻辑以及控制检测过程的设置。



该类提供了一种方法来计算所有必要的数，并以此从给定图像中生成一种模式结构：

```
void PatternDetector::computePatternFromImage(const cv::Mat&
image, Pattern& pattern);
```

该方法在输入图像上查找特征点，并使用具体的检测器和提取器来提取算法描述符，并用这些数据来填充模式结构，以便稍后使用。

当计算 **Pattern** 时，通过调用 **train** 方法来训练一个检测器，其中 **Pattern** 将作为该方法的参数。

```
void PatternDetector::train(const Pattern& pattern)
```

该方法将当前的目标模式作为参数，这个模式就是需要查找的模式。此外，它还会用模式描述符集来训练描述符匹配器。在调用此方法完成训练后，就可查找训练图像了。**findPattern** 是 **PatternDetector** 类中最后一个公共成员函数，模式检测由它来完成。这个函数封装了前面所描述的整个功能，包括特征检测、描述符提取，以及带离群值过滤的匹配。

下面再次简单地列出需要执行的步骤：

- 1) 将输入图像转换为灰度图像。
- 2) 使用具体的特征检测算法在查询图像上检测特征。
- 3) 按所检测到的特征点从输入图像中提取描述符。
- 4) 基于模式描述符进行描述符匹配。
- 5) 通过交叉检测或比率测试来删除离群点。
- 6) 通过非离群点找到单应性变换。
- 7) 通过单应性来扭曲查询图像，从而细化单应性，如前面步骤中介绍的那样。
- 8) 通过多个粗糙和细分的单应性来查找精确的单应性。
- 9) 将模式角点变换到图像坐标系，从而得到输入图像的位置。

### 3.3 模式姿态估计

这里所做的姿势估计与上一章标记姿态估计有类似的地方。通常需要 2D-3D 的对应关系来估计相机的外参数。分配四个三维点以协调单位矩形的角点和与位图图像角点对应的二维点，其中，单位矩形的角点位于 XY 平面内 (Z 轴向上)。

#### 3.3.1 PatternDetector.cpp

下面的代码使用 buildPatternFromImage 函数来创建 Pattern 对象：

```
void PatternDetector::buildPatternFromImage(const cv::Mat& image,
    Pattern& pattern) const
{
    int numImages = 4;
    float step = sqrtf(2.0f);

    // Store original image in pattern structure
    pattern.size = cv::Size(image.cols, image.rows);
    pattern.frame = image.clone();
    getGray(image, pattern.grayImg);

    // Build 2d and 3d contours (3d contour lie in XY plane since
    // it's planar)
    pattern.points2d.resize(4);
    pattern.points3d.resize(4);

    // Image dimensions
    const float w = image.cols;
    const float h = image.rows;

    // Normalized dimensions:
    const float maxSize = std::max(w,h);
```

```

const float unitW = w / maxSize;
const float unitH = h / maxSize;

pattern.points2d[0] = cv::Point2f(0,0);
pattern.points2d[1] = cv::Point2f(w,0);
pattern.points2d[2] = cv::Point2f(w,h);
pattern.points2d[3] = cv::Point2f(0,h);

pattern.points3d[0] = cv::Point3f(-unitW, -unitH, 0);
pattern.points3d[1] = cv::Point3f( unitW, -unitH, 0);
pattern.points3d[2] = cv::Point3f( unitW,  unitH, 0);
pattern.points3d[3] = cv::Point3f(-unitW,  unitH, 0);

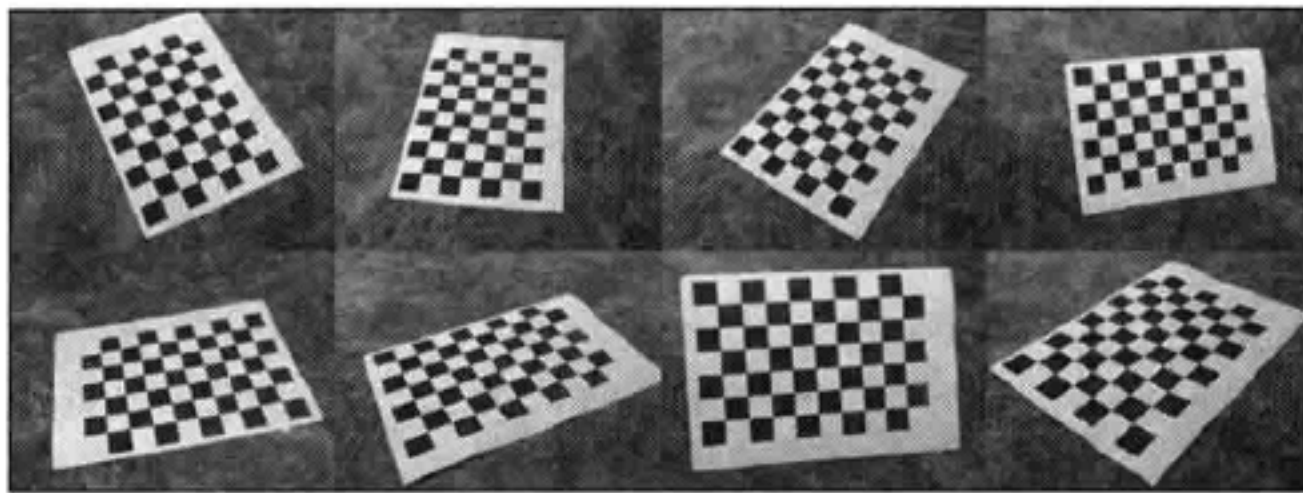
extractFeatures(pattern.grayImg, pattern.keypoints,
                pattern.descriptors);
}

```

角点的结构很有用，因为这个模式坐标系统将直接放置在模式位置中心，该位置在 XY 平面内，而 Z 轴可看成是摄像机方向。

### 3.3.2 获取摄像机内矩阵

摄像机的内参数（camera-intrinsic parameter）可利用 OpenCV 发布包中的示例程序 `camera_calibration.exe` 来计算。该程序会使用一系列模式图像来找到内部透镜参数，例如：焦距、主点、失真系数。例如：有一组来自于不同角度用于标定的模式图像，总共有 8 张图像，见下图：



然后通过下面的命令行方式执行标定：

```

imagelist_creator imagelist.yaml *.png
calibration -w 9 -h 6 -o camera_intrinsic.yaml imagelist.yaml

```

第一条命令将创建 YAML 格式的一个图像列表，校验工具希望将当前目录中所有 PNG 文件作为输入。也可以使用完整的文件名，如：img1.png、img2.png 和 img3.png。然后将生成的文件 `imagelist.yaml` 传递给标定程序。此外，标定工具可以从一个普通的摄像机获取图像。

指定校正模式的维度、输入文件以及将写入标定数据的输出文件。

标定完成后，会得到一个 YAML 文件，其结果如下：



```

%YAML:1.0
calibration_time: "06/12/12 11:17:56"
image_width: 640
image_height: 480
board_width: 9
board_height: 6
square_size: 1.
flags: 0
camera_matrix: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [ 5.2658037684199849e+002, 0., 3.1841744018680112e+002, 0.,
    5.2465577209994706e+002, 2.0296659047014398e+002, 0., 0., 1. ]
distortion_coefficients: !!opencv-matrix
  rows: 5
  cols: 1
  dt: d
  data: [ 7.3253671786835686e-002, -8.6143199924308911e-002,
    -2.0800255026966759e-002, -6.8004894417795971e-004,
    -1.7750733073535208e-001 ]
avg_reprojection_error: 3.6539552933501085e-001

```

这里主要关注 camera\_matrix，它是  $3 \times 3$  的摄像机标定矩阵。其具体内容如右：

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

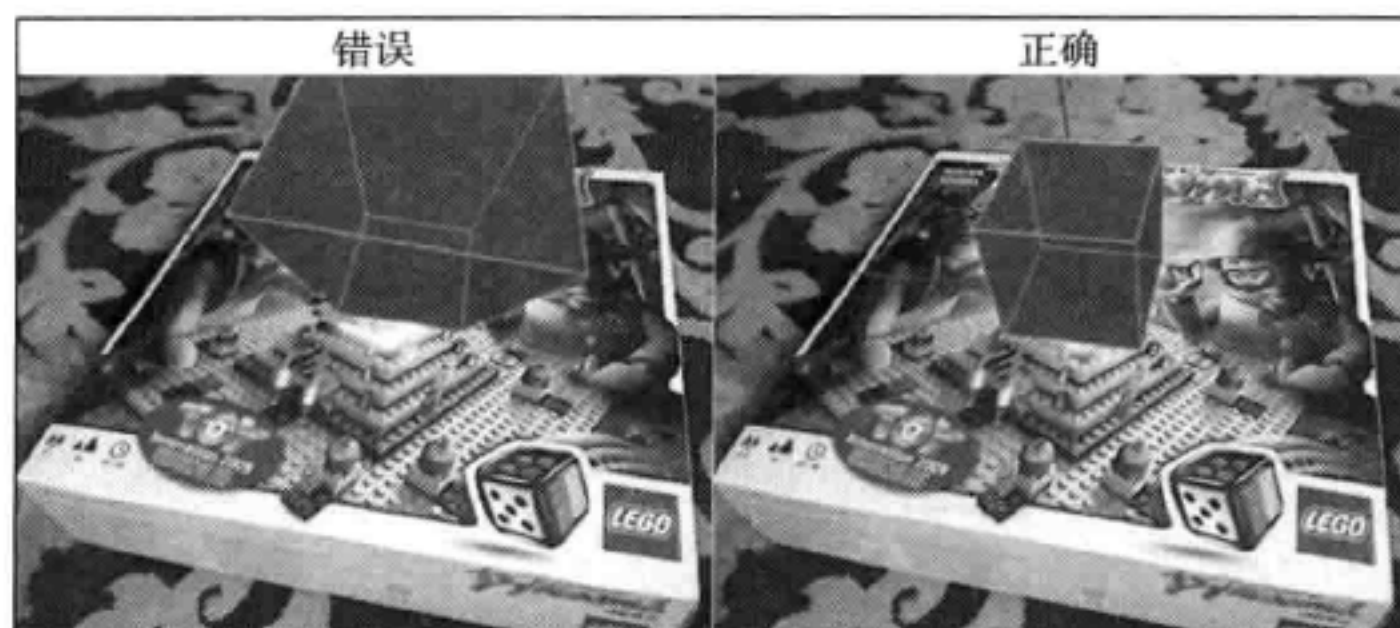
该矩阵有四个分量有用： $f_x$ 、 $f_y$ 、 $c_x$  以及  $c_y$ ，利用这些数据，并使用下面的代码来创建摄像机标定对象实例，并将其用于摄像机标定中：

```

CameraCalibration calibration(526.58037684199849e,
524.65577209994706e, 318.41744018680112, 202.96659047014398)

```

如果没有正确的摄像机标定不可能创造一个看上去很真实的增强现实应用。估计的透视变换将不同于摄像机所具有的变换。这将导致增强的对象看起来会太近或太远。下面是一个故意改变摄像机标定后得到的截图：



正如你所看到的，盒子的视图与整个场景不协调。

为了估计模式位置，使用了 OpenCV 的 `CV::solvePnP` 函数来解决 PnP 问题。也许读者

熟悉此函数，因为在基于标记的 AR 中用到了它。这里需要当前图像中模式角点的坐标和它所参考的三维坐标，该坐标在前面定义过。



**注意：**cv::solvePnP 可在多于四个点的情况下运行，该函数也是创建复杂形状模式的 AR 应用的关键。其思想仍然是相同的，即只需定义相应模式的三维结构以及需查找的对应二维点。当然，这里不适合用单应性估计。

要从训练模式对象获取参考三维点以及相应的二维投影，该投影来自 PatternTrackingInfo 结构。摄像机的标定信息存储在私有变量 PatternDetector 中。

### Pattern.cpp

在三维空间中，模式位置由 computePose 函数进行估计，其具体实现如下：

```
void PatternTrackingInfo::computePose(const Pattern& pattern, const
CameraCalibration& calibration)
{
    cv::Mat camMatrix, distCoeff;
    cv::Mat(3,3, CV_32F,
        const_cast<float*>(&calibration.getIntrinsic().data[0]))
        .copyTo(camMatrix);
    cv::Mat(4,1, CV_32F,
        const_cast<float*>(&calibration.getDistorsion().data[0]))
        .copyTo(distCoeff);

    cv::Mat Rvec;
    cv::Mat_<float> Tvec;
    cv::Mat raux,taux;
    cv::solvePnP(pattern.points3d, points2d, camMatrix,
        distCoeff,raux,taux);
    raux.convertTo(Rvec,CV_32F);
    taux.convertTo(Tvec ,CV_32F);

    cv::Mat_<float> rotMat(3,3);
    cv::Rodrigues(Rvec, rotMat);

    // Copy to transformation matrix
    pose3d = Transformation();

    for (int col=0; col<3; col++)
    {
        for (int row=0; row<3; row++)
        {
            pose3d.r().mat[row][col] = rotMat(row,col);
            // Copy rotation component
        }
        pose3d.t().data[col] = Tvec(col);
        // Copy translation component
    }
}
```

```

// Since solvePnP finds camera location, w.r.t to marker pose,
// to get marker pose w.r.t to the camera we invert it.
pose3d = pose3d.getInverted();
}

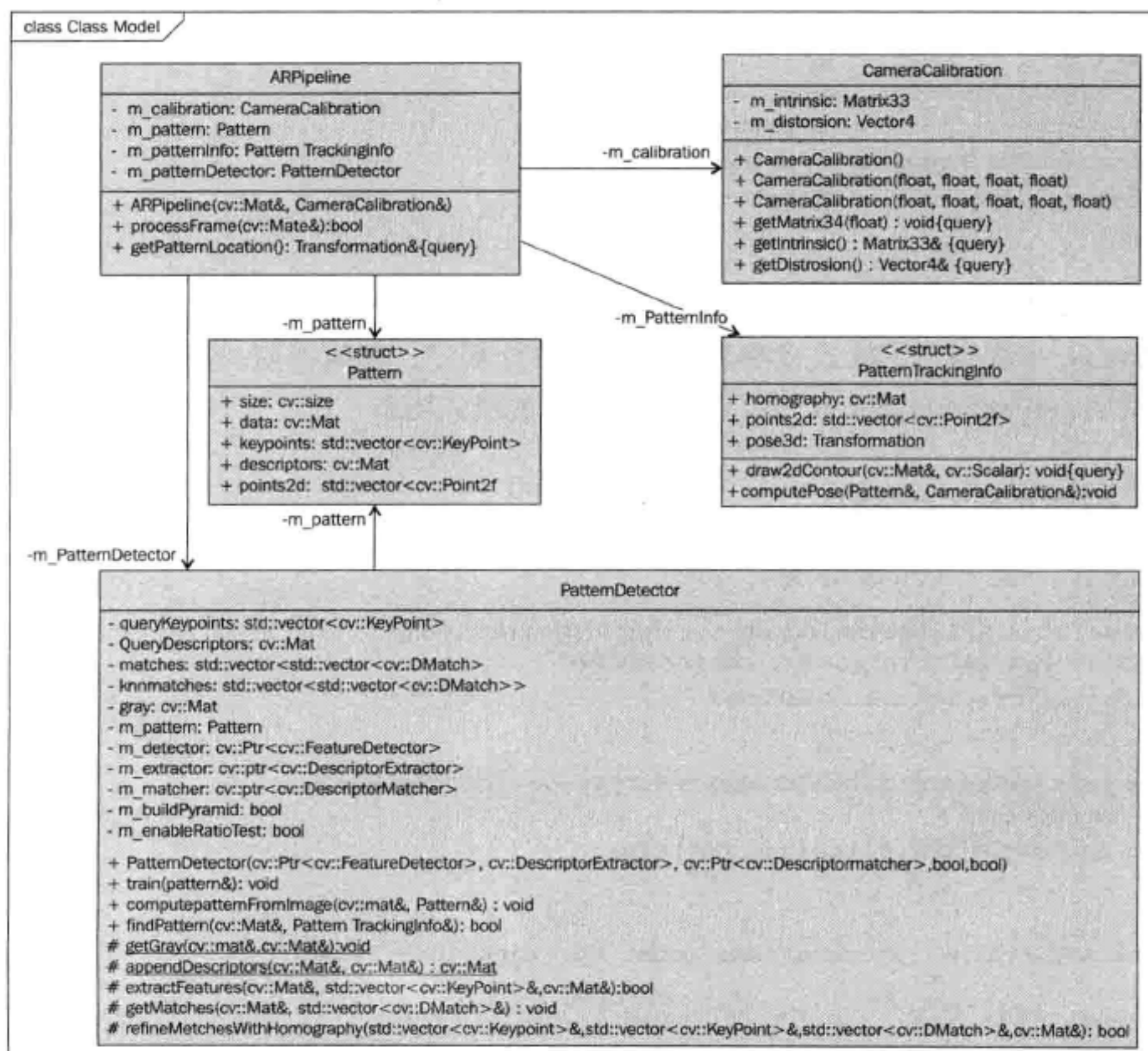
```

### 3.4 应用的基础架构

到目前为止，已经介绍了如何检测模式，以及如何估计它们在摄像机中的三维位置。下面来介绍如何利用这些算法构建一个真正的应用程序。因此，本节的目标是介绍如何使用 OpenCV 从摄像机中获取视频，并针对 3D 渲染创建可视化场景。

由于本章的目标是介绍如何使用无标记的 AR 关键特性，因此只会创建一个简单的命令行应用，该应用能够检测任意模式图像，而不管这些图像是来自视频序列还是图像文件。

这里引入 ARPipeline 类，它拥有全部的图像处理逻辑和中间数据。该类为根对象（root object），包含增强现实和对输入帧进行处理的所有程序必需的子组件。下面是 ARPipeline 及其子组件的 UML 图：





该类包括：

- 相机标定对象
- 模式 - 检测对象的实例
- 一个训练好的模式对象
- 模式跟踪的中间数据

### 3.4.1 ARPipeline.hpp

下面的代码为 ARPipeline 类的定义：

```
class ARPipeline
{
public:
    ARPipeline(const cv::Mat& patternImage,
               const CameraCalibration& calibration);

    bool processFrame(const cv::Mat& inputFrame);

    const Transformation& getPatternLocation() const;

private:
    CameraCalibration    m_calibration;
    Pattern              m_pattern;
    PatternTrackingInfo  m_patternInfo;
    PatternDetector      m_patternDetector;
};
```

在 ARPipeline 的构造函数中，会初始化一个模式对象并用私有字段保存标定数据。processFrame 函数实现了模式检测和人的姿态估计程序，其返回值为模式检测是否成功。可通过调用 getPatternLocation 函数来得到所计算的模式姿态。

### 3.4.2 ARPipeline.cpp

下面的代码是 ARPipeline 类的实现：

```
ARPipeline::ARPipeline(const cv::Mat& patternImage,
                       const CameraCalibration& calibration)
    : m_calibration(calibration)
{
    m_patternDetector.buildPatternFromImage (patternImage,
                                             m_pattern);
    m_patternDetector.train(m_pattern);
}

bool ARPipeline::processFrame(const cv::Mat& inputFrame)
{
```

```

bool patternFound = m_patternDetector.findPattern(inputFrame,
    m_patternInfo);

if (patternFound)
{
    m_patternInfo.computePose(m_pattern, m_calibration);
}

return patternFound;
}

const Transformation& ARPipeline::getPatternLocation() const
{
    return m_patternInfo.pose3d;
}

```

### 3.4.3 在 OpenCV 中启用三维可视化支持

像前一章那样，这里会用 OpenGL 来渲染 3D 效果。这里要自由得多，因为不会像 iOS 环境那样必须遵循 iOS 应用程序体系结构的要求。在 Windows 和 Mac 平台下，可选择许多 3D 引擎。本章将介绍如何用 OpenCV 来创建跨平台的三维可视化。从 2.4.2 版本开始，OpenCV 在可视化窗口中支持 OpenGL。这意味着在 OpenCV 中可轻松渲染任何 3D 内容。

若要在 OpenCV 中开始一个 OpenGL 窗口，需要做的第一件事是生成支持 OpenGL 的 OpenCV。否则，当试图使用 OpenCV 中相关的 OpenGL 函数时会抛出异常。要启用 OpenGL 支持，应该设置标志 `ENABLE_OPENGL=YES`，以生成相应的 OpenCV 库。



**注意：**由于当前版本（OpenCV 2.4.2）在默认情况下不支持 OpenGL。也许 OpenGL 在将来的 OpenCV 版本会将默认情况设为支持。如果是这样，就没有必要手工来开启。

要在 OpenCV 中设置一个 OpenGL 窗口，可执行以下操作：

- ❑ 从 GitHub (<https://github.com/Itseez/> 上克隆 OpenCV 的库)。需要使用命令行的 git 工具或在计算机上安装 GitHub 应用程序来执行这一步。
- ❑ 配置 OpenCV 并为相应的 IDE 生成工作区 (workspace)。需要 CMake 应用程序来完成这一步。可从 <http://www.cmake.org/cmake/resources/software.html> 自由下载 CMake。

为了配置 OpenCV，可使用命令行的 CMake 命令，具体操作如下（从放置在所生成项目的目录中运行）：

```
cmake -D ENABLE_OPENGL=YES <path to the OpenCV source directory>
```

如果读者喜欢 GUI 风格，可用 CMake-GUI，它为用户提供了更友好的项目配置界面：



在所选择的 IDE 中生成 OpenCV 的工作区后，打开该项目，并配置相应选项，以生成库，然后安装它。当这个过程完成后，可使用刚才生成的 OpenCV 库来编译示例项目。

### 3.4.4 使用 OpenCV 来创建 OpenGL 窗口

现在有一个支持 OpenGL 的 OpenCV 库，可用其创建第一个 OpenGL 窗口。OpenGL 窗口的初始化由创建一个命名的窗口开始，这需要设置一个 OpenGL 标志：

```
cv::namedWindow(ARWindowName, cv::WINDOW_OPENGL);
```

ARWindowName 是一个字符串常量，它保存着窗口的名称。这里将使用 Markerless AR 作为该常量的值。这个函数将使用指定的名称来创建一个窗口。CV:: WINDOW\_OPENGL 标志表明要在创建的窗口中使用 OpenGL，然后设置窗口大小：

```
cv::resizeWindow(ARWindowName, 640, 480);
```

接下来需对此窗口设置绘图上下文：

```
cv::setOpenGlContext(ARWindowName);
```

现在窗口就可使用了。为了在窗口上画一些东西，应用以下方法注册一个回调函数：

```
cv::setOpenGlDrawCallback(ARWindowName, drawAR, NULL);
```

该回调函数将被称为重绘窗口。第一个参数为窗口名，第二个参数为回调函数，第三个可选参数将被传递给回调函数。

其中，drawAR 函数应该具有以下形式：

```
void drawAR(void* param)
{
    // Draw something using OpenGL here
}
```



为了通知系统重绘窗口，可使用 CV:: UpdateWindow 函数：

```
cv::updateWindow(ARWindowName);
```

### 3.4.5 使用 OpenCV 捕获视频

OpenCV 几乎可从每台摄像机或每个视频文件中轻松获取帧。无论从摄像机或视频文件中捕获视频，都要使用 cv::VideoCapture 类，如需复习相关内容可查阅 1.1 节。

### 3.4.6 渲染增强现实

这里引入 ARDrawingContext 结构来保存所有必要的的数据，可视化时可能需要这些数据：

- 从相机拍摄的最新图像；
- 摄像机标定矩阵；
- 三维空间中的模式姿态（如果存在的话）；
- 与 OpenGL 相关的内部数据（纹理 ID 等）。

#### 1. ARDrawingContext.hpp

以下代码为 ARDrawingContext 类的定义：

```
class ARDrawingContext
{
public:
    ARDrawingContext(const CameraCalibration& c);

    bool                patternPresent;
    Transformation      patternPose;

    //! Request the redraw of the OpenGL window
    void draw();

    //! Set the new frame for the background
    void updateBackground(const cv::Mat& frame);

private:
    //! Draws the background with video
    void drawCameraFrame ();

    //! Draws the AR
    void drawAugmentedScene();

    //! Builds the right projection matrix
    //! from the camera calibration for AR
    void buildProjectionMatrix(const Matrix33& calibration,
        int w, int h, Matrix44& result);
```

```

    //! Draws the coordinate axis
    void drawCoordinateAxis();

    //! Draw the cube model
    void drawCubeModel();

private:
    bool            m_textureInitialized;
    unsigned int    m_backgroundTextureId;
    CameraCalibration m_calibration;
    cv::Mat         m_backgroundImage;
};

```

## 2. ARDrawingContext.cpp

OpenGL 窗口的初始化在 ARDrawingContext 类的构造函数中完成，如下所示：

```

ARDrawingContext::ARDrawingContext(std::string windowName, cv::Size
frameSize, const CameraCalibration& c)
    : m_isTextureInitialized(false)
    , m_calibration(c)
    , m_windowName(windowName)
{
    // Create window with OpenGL support
    cv::namedWindow(windowName, cv::WINDOW_OPENGL);

    // Resize it exactly to video size
    cv::resizeWindow(windowName, frameSize.width, frameSize.height);

    // Initialize OpenGL draw callback:
    cv::setOpenGlContext(windowName);
    cv::setOpenGlDrawCallback(windowName,
        ARDrawingContextDrawCallback, this);
}

```

由于现在用单独的类来存储可视化状态，可修改 cv:: setOpenGlDrawCallback 回调函数，并将 ARDrawingContext 的一个实例作为其参数。修改后的回调函数如下所示：

```

void ARDrawingContextDrawCallback(void* param)
{
    ARDrawingContext * ctx = static_cast<ARDrawingContext*>(param);
    if (ctx)
    {
        ctx->draw();
    }
}

```

ARDrawingContext 将负责所有渲染增强现实的任务。帧渲染由绘制背景和正投影开始，然后使用恰当的透视投影和模型转换来渲染 3D 模型。以下代码包含了绘制函数的最终版本：

```

void ARDrawingContext::draw()
{
    // Clear entire screen
    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
    // Render background
    drawCameraFrame();
    // Draw AR
    drawAugmentedScene();
}

```

清除屏幕和深度缓存后，需检查视频中出现的纹理是否被初始化。如果是这样，就可开始绘制背景，否则需要通过调用 `glGenTextures` 来创建一个新的二维纹理。

为了绘制背景，需采用一个正投影并绘制实心矩形，它涵盖了所有的屏幕视点。该矩形为一个纹理单元。该纹理用 `m_backgroundImage` 对象的内容填充，其内容被事先加载到 OpenGL 的内存中。此功能与前一章相同，所以这里省略相关代码。

在摄像机绘制图像后，就可绘制一个 AR。这需要设置符合摄像机标定的恰当透视投影。

下面的代码展示了如何从摄像机标定和渲染场景中建立合适的 OpenGL 投影矩阵：

```

void ARDrawingContext::drawAugmentedScene()
{
    // Init augmentation projection
    Matrix44 projectionMatrix;
    int w = m_backgroundImage.cols;
    int h = m_backgroundImage.rows;
    buildProjectionMatrix(m_calibration, w, h, projectionMatrix);

    glMatrixMode(GL_PROJECTION);
    glLoadMatrixf(projectionMatrix.data);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    if (isPatternPresent)
    {
        // Set the pattern transformation
        Matrix44 glMatrix = patternPose.getMat44();
        glLoadMatrixf(reinterpret_cast<const GLfloat*>(&glMatrix.data[0]));

        // Render model
        drawCoordinateAxis();
        drawCubeModel();
    }
}

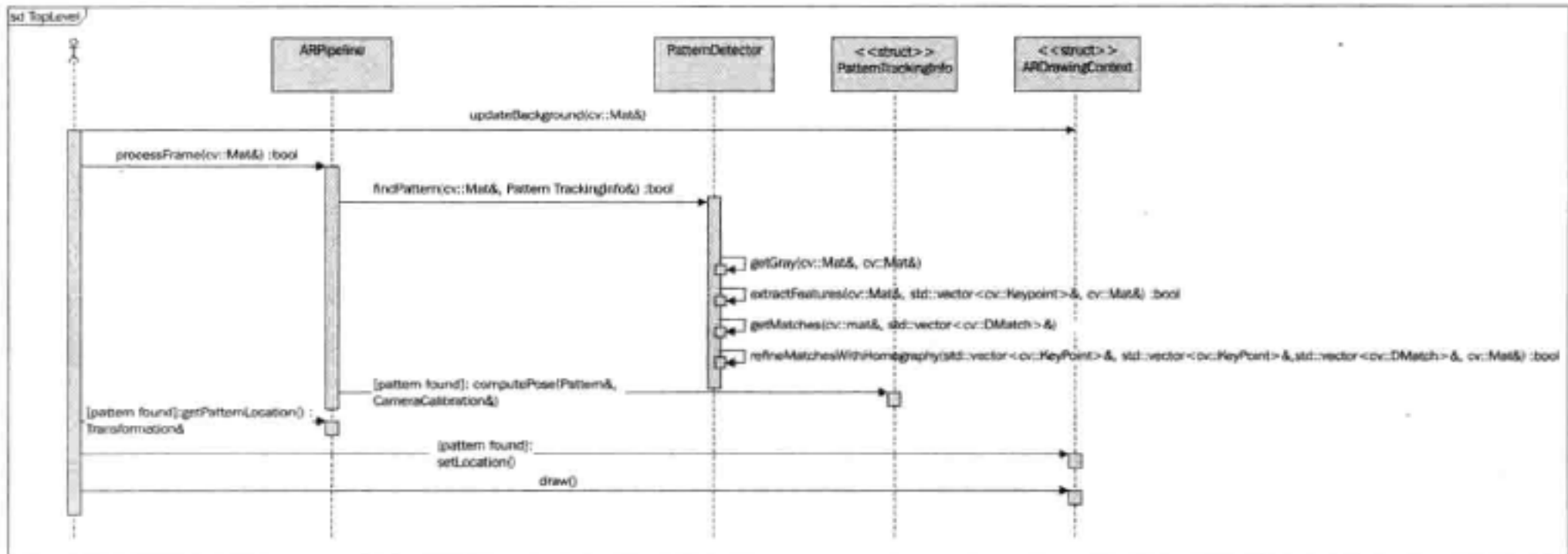
```

这里的 `buildProjectionMatrix` 函数与上一章相同。在使用透视投影后，为了实现模式变换，需设置 `GL_MODELVIEW` 矩阵。为了证明姿态估计正常工作，需在模式位置绘制一个



单位坐标系。

前面几乎完成应用程序的所有事情。现在只需创建一个模式检测算法，然后在三维空间中估计所找到模式的位置，以便建立一个可视化系统来呈现 AR。可查看下面的 UML 序列图，它展示本章应用程序的帧处理（frame-processing）例程：



### 3.4.7 演示应用程序

本章的示范项目支持图像文件、录制的视频，以及网络摄像头所拍摄的实时图像。这里创建了两个函数来实现这些功能。

#### main.cpp

函数 `processVideo` 会处理视频，而 `processSingleImage` 用于处理单个图像，这两个函数的定义如下：

```
void processVideo(const cv::Mat& patternImage,
    CameraCalibration& calibration, cv::VideoCapture& capture);
```

```
void processSingleImage(const cv::Mat& patternImage,
    CameraCalibration& calibration, const cv::Mat& image);
```

从函数名就可很清楚地知道：第一个函数用来处理视频，而第二个函数用来处理单个图像（此函数用于调试很有用）。它们都是常见的图像处理、模式检测、场景渲染以及用户交互程序。

程序中的 `processFrame` 函数的具体实现如下：

```
/**
 * Performs full detection routine on camera frame
 * and draws the scene using drawing context.
 * In addition, this function draw overlay with debug information
 * on top of the AR window. Returns true
 * if processing loop should be stopped; otherwise - false.
 */
bool processFrame(const cv::Mat& cameraFrame, ARPipeline&
```

```

pipeline, ARDrawingContext& drawingCtx)
{
    // Clone image used for background (we will
    // draw overlay on it)
    cv::Mat img = cameraFrame.clone();

    // Draw information:
    if (pipeline.m_patternDetector.enableHomographyRefinement)
        cv::putText(img, "Pose refinement: On ('h' to switch
            off)", cv::Point(10,15), CV_FONT_HERSHEY_PLAIN, 1,
            CV_RGB(0,200,0));
    else
        cv::putText(img, "Pose refinement: Off ('h' to switch
            on)", cv::Point(10,15), CV_FONT_HERSHEY_PLAIN, 1,
            CV_RGB(0,200,0));

    cv::putText(img, "RANSAC threshold: " +
        ToString(pipeline.m_patternDetector.
            homographyReprojectionThreshold) + " ( Use '-'/'+' to
            adjust)", cv::Point(10, 30), CV_FONT_HERSHEY_PLAIN, 1,
            CV_RGB(0,200,0));

    // Set a new camera frame:
    drawingCtx.updateBackground(img);

    // Find a pattern and update its detection status:
    drawingCtx.isPatternPresent =
        pipeline.processFrame(cameraFrame);

    // Update a pattern pose:
    drawingCtx.patternPose = pipeline.getPatternLocation();

    // Request redraw of the window:
    drawingCtx.updateWindow();

    // Read the keyboard input:
    int keyCode = cv::waitKey(5);

    bool shouldQuit = false;
    if (keyCode == '+' || keyCode == '=')
    {
        pipeline.m_patternDetector.homographyReprojectionThreshold
            += 0.2f;
        pipeline.m_patternDetector.homographyReprojectionThreshold
            = std::min(10.0f, pipeline.m_patternDetector.
                homographyReprojectionThreshold);
    }
    else if (keyCode == '-')
    {
        pipeline.m_patternDetector.
            homographyReprojectionThreshold -= 0.2f;
        pipeline.m_patternDetector.homographyReprojectionThreshold

```

```

        = std::max(0.0f, pipeline.m_patternDetector.
        homographyReprojectionThreshold);
    }
    else if (keyCode == 'h')
    {
        pipeline.m_patternDetector.enableHomographyRefinement =
            !pipeline.m_patternDetector.enableHomographyRefinement;
    }
    else if (keyCode == 27 || keyCode == 'q')
    {
        shouldQuit = true;
    }

    return shouldQuit;
}

```

ARPipeline 和 ARDrawingContext 的初始化在 processSingleImage 函数或 processVideo 函数中都可完成，在 processSingleImage 函数中的初始化过程如下：

```

void processSingleImage(const cv::Mat& patternImage,
    CameraCalibration& calibration, const cv::Mat& image)
{
    cv::Size frameSize(image.cols, image.rows);
    ARPipeline pipeline(patternImage, calibration);
    ARDrawingContext drawingCtx("Markerless AR", frameSize,
        calibration);

    bool shouldQuit = false;
    do
    {
        shouldQuit = processFrame(image, pipeline, drawingCtx);
    } while (!shouldQuit);
}

```

上面的代码用模式图像和标定参数来创建 ARPipeline，然后再次用标定参数来初始化 ARDrawingContext。经过这些步骤后，会创建 OpenGL 窗口。然后加载查询图像到一个绘图上下文中，并调用 ARPipeline.processFrame 来查找一个模式。如果找到姿态的模式，就可复制它的位置到这个绘图上下文中以便进一步进行帧渲染。如果没有找到模式，只渲染摄像机的帧，而不需要做任何 AR。

可以通过以下方式运行这个示例程序。

❑ 要运行单个图像，可执行下面的命令：

```
markerless_ar_demo pattern.png test_image.png
```

❑ 要运行录制视频，可执行下面的命令：

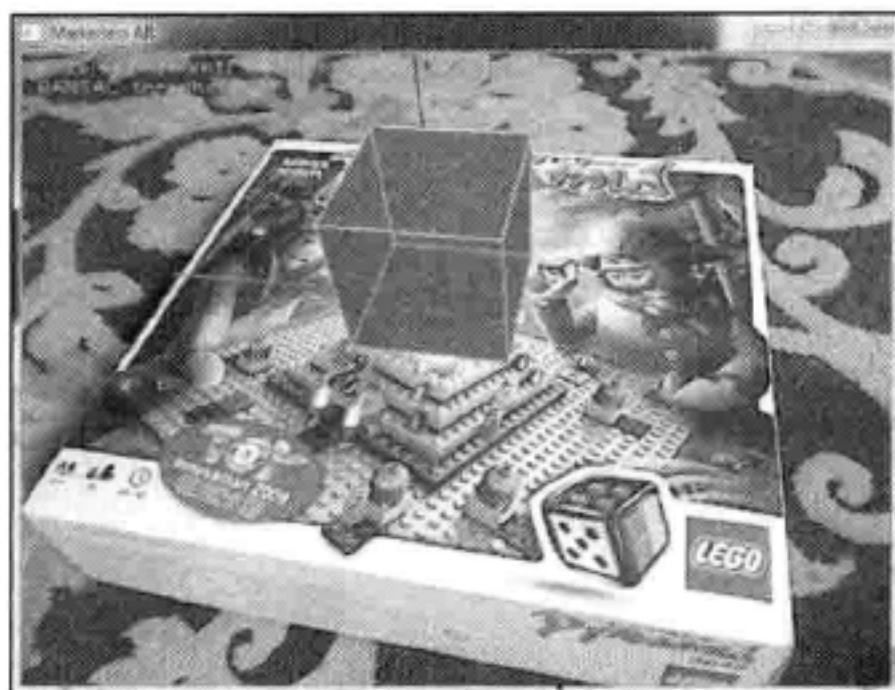
```
markerless_ar_demo pattern.png test_video.avi
```

❑ 要运行来自摄像机的实时图像，可执行下面的命令：

```
markerless_ar_demo pattern.png
```



增强单个图像的效果显示如下图所示。



### 3.5 总结

本章介绍了特征描述符以及如何使用它们来定义一个缩放和旋转都不变的模式描述符。该描述符也可用于查找其他图像中类似的物体。同时，解释了最流行的特征描述符的长处和缺点。本章后半部分介绍如何使用 OpenGL 和 OpenCV 的共同渲染增强现实。

### 3.6 参考文献

- *Distinctive Image Features from Scale-Invariant Keypoints* (<http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf>)
- *SURF: Speeded Up Robust Features* (<http://www.vision.ee.ethz.ch/~surf/eccv06.pdf>)
- *Model-Based Object Pose in 25 Lines of Code*, Dementhon and L.S Davis, *International Journal of Computer Vision*, edition 15, pp. 123-141, 1995
- *Linear N-Point Camera Pose Determination*, L.Quan, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 21, edition. 7, July 1999
- *Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography*, M. Fischer and R. Bolles, *Graphics and Image Processing*, vol. 24, edition. 6, pp. 381-395, June 1981
- *Multiple View Geometry in Computer Vision*, R. Hartley and A.Zisserman, Cambridge University Press (<http://www.umi.acs.umd.edu/~ramani/cmsc828d/lecture9.pdf>)
- *Camera Pose Revisited – New Linear Algorithms*, M. Ameller, B.Triggs, L.Quan (<http://hal.inria.fr/docs/00/54/83/06/PDF/Ameller-eccv00.pdf>)
- *Closed-form solution of absolute orientation using unit quaternions*, Berthold K. P. Horn, *Journal of the Optical Society A*, vol. 4, 629-642

## 使用 OpenCV 研究从运动中恢复结构

本章将讨论从运动中恢复结构 (Structure from Motion, SfM) 的概念, 以便能更好地通过摄像机移动来提取图像几何结构, 这些功能都可通过 OpenCV 的 API 函数来完成。为了使用单台摄像机, 需要对本章采用的方法进行如下限制: 使用单目方法 (monocular approach); 一个离散且稀疏的视频帧集合, 而不是连续的视频流。这样大大简化了接下来将介绍的项目, 这种简化也可帮助读者理解 SfM 的基础知识。为了实现这样的限制, 将使用 Hartley 和 Zisserman (下面简称 H 和 Z) 在他们的成名著作《Multiple View Geometry in Computer Vision》第 9 章至第 12 章中的内容。

本章内容如下:

- 从运动中恢复结构的基本概念;
- 从两幅图像估计摄像机的运行;
- 重构场景;
- 从视图中重构;
- 重构细化;
- 可视化三维点云。

整章都会假定摄像机已被标定过, 即对摄像机预先校准。标定 (Calibration) 是计算机视觉所特有的操作, 可由 OpenCV 的命令行工具来完成, 这在前一章已经讨论过。因此, 本章会假设摄像机内部参数以矩阵  $K$  的形式存在, 其中的一些参数由标定得到。

为了将下面的内容表述清楚, 本章将摄像机作为单个场景视图, 而不是用于成像的光学器件。一个摄像机有空间里的位置和视图方向。在两个摄像机之间, 有平移元素 (通过空间移动) 和视图的方向旋转。

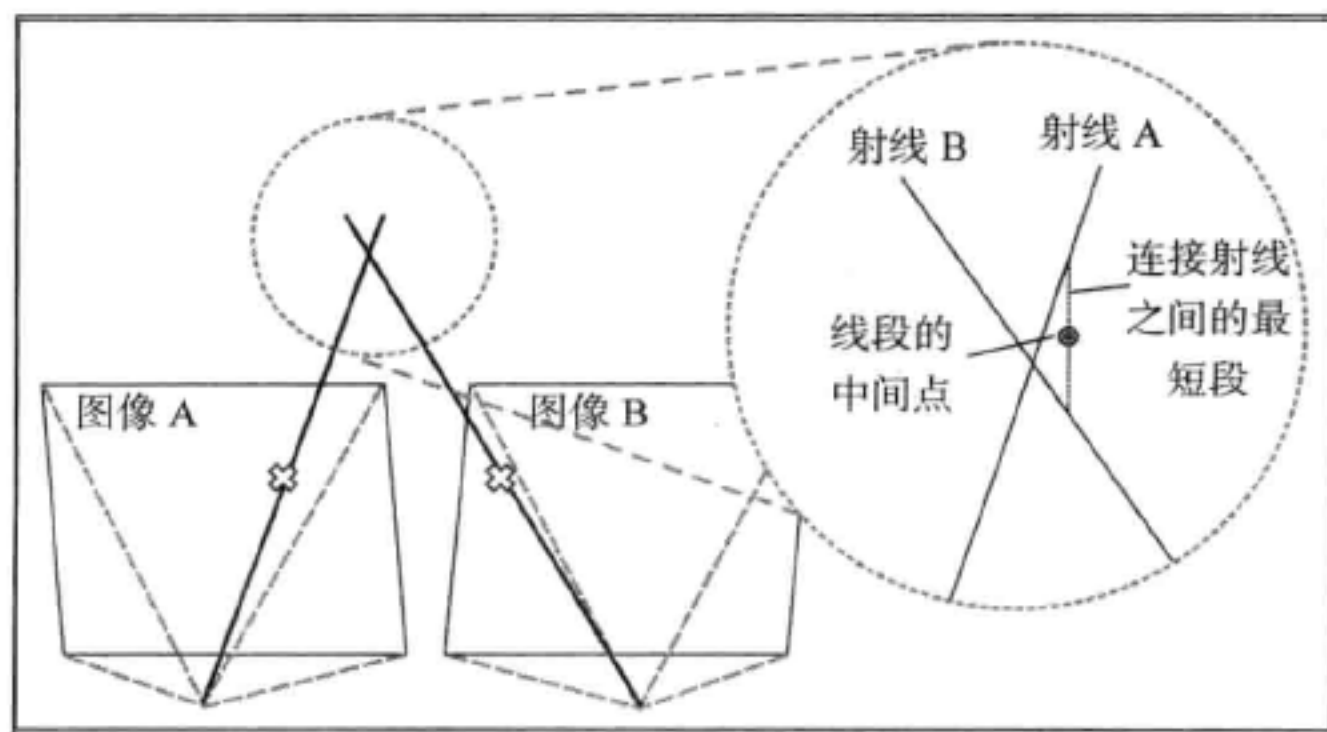
本章也将统一对场景、现实世界、三维空间中关于点的术语，即将这些环境中的点都认为是现实世界中的点。这种方式同样适用于图像或二维平面中的点，它们是图像坐标中的点，以及某些三维点，这些点将位置和时间投影到摄像机的传感器上。

在本章的代码中，有时会请读者参考《Multiple View Geometry in Computer Vision》这本书，例如：// HZ 9.12 表示引用了这本书第 9 章的第 12 个公式。此外，这本书也只给出了部分代码，完整可运行的代码可在随书材料中找到。

## 4.1 从运动中恢复结构的概念

首先应对立体（或任意多视图）、使用标定设备进行三维重构以及 SfM 这三个概念进行区别。对于两台或更多摄像机，通常会假定知道摄像机之间的运动，而对于 SfM，这种运动并不知道，是需要寻找的。对于标定设备，简单来讲就是可得到更准确的三维几何重构，在摄像机之间估计距离和旋转不会有误差，因为这些都已知。实现 SfM 应用的第一步是找到摄像机间的运动。OpenCV 可以有很多方法来找到这种运动，比如：可使用 findFundamentalMat 函数。

下面来讨论一下选择 SfM 算法的目的。大多数情况都是想得到场景的几何形状，例如：物体在摄像机的什么位置以及它们的形状是什么。假设已经知道多台摄像机拍摄同一场景的运动，一个合理的结论是：现在希望重构几何形状。这在计算机视觉中称为三角形（triangulation）法，有很多方法来实现它。可通过射线交集方式来实现，这种方式需要构造两条射线：每条射线会通过摄像机的投影中心和图像平面的一个点。在理想情况下，这些空间中相交的射线，其交点就是真实世界的一个三维点，该三维点会呈现在摄像机中。如下图所示：



在现实中，射线相交非常不可靠。H 和 Z 都反对这种做法。因为射线通常不会相交，可通过使用连接两条射线的最短线段的中点作为其交点。另外，H 和 Z 给出许多三角化三维点的方法，4.3 节将讨论几个这样的方法。当前的 OpenCV 版本并没有包含简单的三角



化 API 函数，因此，在本章中会来实现这些功能。

在介绍完怎样从两个视图恢复三维几何结构后，将会介绍如何用同一场景的更多视图来得到更丰富的重建结构。对于此问题，在 4.5 节会介绍大多数 SfM 方法都会采用的光束调整法，它会优化所估计的摄像机位置的光束和三维点。在 OpenCV 新的图像拼接工具 (Image Stitching Toolbox) 中包含了光束调整法。但很多外部工具可以很好地实现这一功能，这些工具是基于 OpenCV 和 C++，能很好地集成到本项目的流程中。因此，本章将介绍如何集成一个外部的简洁光束调整器，该调整器称为 SSBA 库。

到目前为止，已经介绍了如何用 OpenCV 来实现 SfM 的大致过程，下面将介绍每个部分的具体实现。

## 4.2 从两幅图像估计摄像机运动

在开始找两个摄像机间的运行之前，先检查一下执行这一操作的输入和工具。首先，要有两幅在空间上场景一样，但位置不同的图像（注意：不要差别太大）。这是一个很重要的前提条件，要确保能够满足。接下来看一下数学对象，它们会对图像、摄像机、场景产生约束。

两个有用的数学对象分别为：基础矩阵（用  $F$  表示）和本征矩阵（用  $E$  表示）。这两个矩阵基本一样，不同之处在于本征矩阵使用的是标定摄像机，本项目的摄像机属于这种情形，因此会选择本征矩阵。OpenCV 只允许通过 `findFundamentalMat` 函数来找到基础矩阵。但很容易通过标定矩阵  $K$  来得到本征矩阵，其形式如下：

```
Mat_<double> E = K.t() * F * K; //according to HZ (9.12)
```

该本征矩阵大小为  $3 \times 3$ ，它会分别对两幅图像中的点  $x$  和  $x'$  进行约束，其约束条件为： $x'Ex=0$ 。下面将会看到这个约束条件非常有用。另一个重要的事实是：在恢复摄像机图像时都需要本征矩阵。虽然只是一些缩放操作，但后面仍会用到它。因此，如果获取本征矩阵，就可知道放置摄像机的空间位置，这就是正要找的位置。如果有足够多的约束方程，可很容易计算出本征矩阵。简单地讲，每个方程都可以用于求解该矩阵的一小部分。事实上，OpenCV 仅需 7 个点就能计算出该矩阵，但希望这样的点更多一些，这样可得到鲁棒性更好的解。

### 4.2.1 通过丰富的特征描述符进行点匹配

现在介绍利用约束方程来计算本征矩阵。为了得到约束方程，需要找到图像 A 中每个点在图像 B 中的对应点，但如何才能找到这样匹配的点？一种简单方法是使用 OpenCV 扩展的特征匹配框架，该框架经过几年的发展已变得非常成熟。

特征提取和描述符匹配是计算机视觉的一个基本过程，许多方法所涉及的各种操作都

要用到它。例如：在图像中检测物体的位置和方向或在大的图像数据库中通过给定的查询条件来搜索相似图像。实际上，提取意味着在图像中选择点，这些点能很好地表示特征，并由这些点来计算描述符。描述符是由数字组成的向量，它描述了图像中一个特征点周围的情形。用不同方法得到的描述符其长度和数据类型会不同。匹配是这样一个过程：通过一个集合的描述符找到与另一个特征集合与之相对应的特征。OpenCV 提供了简洁且强大的方法来支持特征提取和匹配。更多特征匹配可以在第 3 章中找到。

下面来看一个非常简单的特征提取和匹配方案：

```
// detecting keypoints
SurfFeatureDetector detector();
vector<KeyPoint> keypoints1, keypoints2;
detector.detect(img1, keypoints1);
detector.detect(img2, keypoints2);

// computing descriptors
SurfDescriptorExtractor extractor;
Mat descriptors1, descriptors2;
extractor.compute(img1, keypoints1, descriptors1);
extractor.compute(img2, keypoints2, descriptors2);

// matching descriptors
BruteForceMatcher<L2<float>> matcher;
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
```

也许读者已经理解了类似的 OpenCV 代码，但下面还是要来简单解释一下这段代码。要得到的三要素为两个图像的特征点、描述符以及两个特征集间的匹配。OpenCV 提供特征检测的范围、描述子提取器、匹配器。在这个简单的例子中，采用了 SurfFeatureDetector 函数来得到加速鲁棒性特征（Speeded-Up Robust Features, SURF）在二维空间中的位置，用 SurfDescriptorExtractor 函数来得到 SURF 描述符。采用暴力匹配器（brute-force matcher）来得到相应的匹配，暴力匹配器是得到两个特征集匹配最直接的方式，它通过将第一个集合的每个特征与第二个集合的特征进行比较来得到最佳匹配（这也是暴力这个词的起源）。

下图可看到来自 Fountain-P11 序列中两个图形匹配的特征点，这幅图可在 <http://cvlabwww.epfl.ch/data/multiview/denseMVS.html> 找到。





实际上，若执行刚才这种简单匹配，只能得到一定的效果，并且可能有一些匹配是错误的。因此，大多数 SfM 算法都会在匹配过程中执行某种过滤形式以减少错误，从而确保匹配正确。其中一种过滤形式叫做交叉检查过滤（cross-check filtering），即如果第一幅图像的一个特征与第二幅图像的一个特征匹配，然后进行一个相反检查，将得到第二幅图像的这个特征点与第一幅图像的相应特征点进行匹配，若能匹配，则可认为成功。OpenCV 的暴力匹配器包含这种过滤。本项目将采用另一种常用的过滤机制，该过滤器基于这样的原理：对于同一场景的两幅图，它们之间存在某种立体视图的联系。实际上，该过滤器使计算基础矩阵的过程更具有鲁棒性。在 4.2.3 节将介绍这种过滤器，用这种计算方法所获得的特征对（feature pair）有很少的错误。

## 4.2.2 通过光流进行点匹配

可用光流（Optical Flow, OF）来代替富特征点匹配，例如 SURF。OpenCV 最近针对两幅图像的流场（flow field）扩展了光流的 API，现在的 API 更快更强大。下面将其作为另一种匹配特征算法。



光流是对两个图像中所选择的点进行匹配的过程。假定两个图像是一个图像序列的一部分且它们彼此接近，多数光流算法会将图像 A 的一个小区域，与图像 B 同一区域进行比较，所谓的小区域是指图像中每个点周围的搜索窗口或一个小块。然后采用计算机视觉很常用的规则：亮度恒定约束（也许有其他名称）进行计算，这两幅图像的一个小块（patches）不会有明显变化，因此，它们相减的差值应接近零。除了匹配这些小块外，基于光流的较新算法采用了许多其他方式来得到更好的结果。其中一种方法使用了图像金字塔，该方法不断调整图像，使其越来越小，从而使图像“由粗到细（from-coarse-to-fine）”，这也是计算机视觉常用的技巧。另一种方法在光流上定义全局约束，即假设朝同一方向移动的点相互靠近。

通过调用 `calcOpticalFlowPyrLK` 函数就可在 OpenCV 中很容易地使用光流。本项目需要保留 OF 的匹配结果，这与丰富特征描述符类似，所以本章后面会互换这两种方法。最后，必须要使用一种特殊匹配方法，它可与前面基于特征的方法互换，下面是该方法的实现代码：

```
Vector<KeyPoint>left_keypoints,right_keypoints;

// Detect keypoints in the left and right images
FastFeatureDetectorffd;
ffd.detect(img1, left_keypoints);
ffd.detect(img2, right_keypoints);

vector<Point2f>left_points;
```



```

KeyPointsToPoints(left_keypoints, left_points);

vector<Point2f>right_points(left_points.size());

// making sure images are grayscale
Mat prevgray, gray;
if (img1.channels() == 3) {
    cvtColor(img1, prevgray, CV_RGB2GRAY);
    cvtColor(img2, gray, CV_RGB2GRAY);
} else {
    prevgray = img1;
    gray = img2;
}

// Calculate the optical flow field:
// how each left_point moved across the 2 images
vector<uchar>vstatus; vector<float>verror;
calcOpticalFlowPyrLK(prevgray, gray, left_points, right_points,
vstatus, verror);

// First, filter out the points with high error
vector<Point2f>right_points_to_find;
vector<int>right_points_to_find_back_index;
for (unsigned inti=0; i<vstatus.size(); i++) {
    if (vstatus[i] &&verror[i] < 12.0) {
        // Keep the original index of the point in the
        // optical flow array, for future use
        right_points_to_find_back_index.push_back(i);
        // Keep the feature point itself
        right_points_to_find.push_back(j_pts[i]);
    } else {
        vstatus[i] = 0; // a bad flow
    }
}

// for each right_point see which detected feature it belongs to
Mat right_points_to_find_flat = Mat(right_points_to_find).
reshape(1, to_find.size()); //flatten array

vector<Point2f>right_features; // detected features
KeyPointsToPoints(right_keypoints, right_features);

Mat right_features_flat = Mat(right_features).reshape(1, right_
features.size());

// Look around each OF point in the right image
// for any features that were detected in its area
// and make a match.
BFMatcher matcher(CV_L2);
vector<vector<DMatch>>nearest_neighbors;

```

```

matcher.radiusMatch(
right_points_to_find_flat,
right_features_flat,
nearest_neighbors,
2.0f);

// Check that the found neighbors are unique (throw away neighbors
// that are too close together, as they may be confusing)
std::set<int>found_in_right_points; // for duplicate prevention
for(int i=0;i<nearest_neighbors.size();i++) {
DMatch _m;
if(nearest_neighbors[i].size()==1) {
    _m = nearest_neighbors[i][0]; // only one neighbor
} else if(nearest_neighbors[i].size()>1) {
    // 2 neighbors - check how close they are
    double ratio = nearest_neighbors[i][0].distance /
nearest_neighbors[i][1].distance;
if(ratio < 0.7) { // not too close
    // take the closest (first) one
    _m = nearest_neighbors[i][0];
} else { // too close - we cannot tell which is better
    continue; // did not pass ratio test - throw away
}
} else {
    continue; // no neighbors... :(
}

// prevent duplicates
if (found_in_right_points.find(_m.trainIdx) == found_in_right_points.
end()) {
    // The found neighbor was not yet used:
    // We should match it with the original indexing
    // of the left point
    _m.queryIdx = right_points_to_find_back_index[_m.queryIdx];
matches->push_back(_m); // add this match
found_in_right_points.insert(_m.trainIdx);
}
}
cout<<"pruned "<< matches->size() <<" / "<<nearest_neighbors.size()
<<" matches"<<endl;

```

函数 `KeyPointsToPoints` 和 `PointsToKeyPoints` 使在结构 `cv::Point2f` 与 `cv::KeyPoint` 之间相互转换变得简单方便。

从上面的这段代码可发现许多有意思的事。首先要注意：使用光流时，其结果显示了一个特征从左边那幅图像的一个位置移动到右边那幅图像的另一个位置。但我们有一个新的特征集，是通过对右边图像检测得到的，并没有同光流中左边图像流出的特征对齐。因此，必须对齐它们。为了找到那些丢失的特征，本项目采用基于 K 近邻的径向搜索，该搜索以兴趣点为中心，两个像素单位为半径来得到落入其中的两个特征。

还需要明白一件事：kNN 比率测试的实现。它经常用于 SfM 以减少误差。从本质上讲，这是一个过滤器。当左边图像的一个特征与右边图像的两个特征匹配时，该过滤器可消除这种混乱（confusing）的匹配。如果右边图像的两个特征太靠近，或它们之间的比率太大（接近 1.0），就可认为它们混乱了，不能使用。另外还需要一个防复制过滤器以便进一步修剪匹配。

下面这幅图展示了从一幅图像到另一幅图像的光流。左边图像的粉红色箭头表示从左边图像到右边图像的移动。在左边第二幅图像是对左边第一幅图像的一个小区域流场（flow field）的放大。左边第二幅图的粉红色箭头展示了该小区域的运动情况，通过观察右边两个原始图像，可知道这些箭头的意义。左手边图像的可视化特征是朝左边移动，粉红色箭头的方向如下图所示。



（附彩图）

在富特征上使用光流的优势在于处理过程通常较快且能容纳更多的匹配点，使重构更密集。有一些光流方法采用的是对所有小块都移动的整体模型，在这种情形下不会考虑匹配富特征。对光流的特性还需说明一点：它最擅长处理取自同一硬件平台的连续图像，而富特征算法却没有这样的性质。两种方法的不同之处为：光流方法通常使用很基本的特征，比如：像关键点周围的图像块，而高阶（high-order）的富特征（比如：SURF）会对每个关键点考虑更高级别（high level）的信息。使用光流或富特征由应用的设计者根据输入的情况而定。

### 4.2.3 搜索摄像机矩阵

前面的工作可获得关键点之间的匹配，在此基础上可计算基础矩阵，然后得到本征矩阵。但必须要对齐两个数据的匹配点，即两个数组的下标要一样，因为 findFundamentalMat 函数需要这样做。还需要将结构 KeyPoint 转换为 Point2f。OpenCV 的 DMatch 类保存着两个关键点之间的匹配，读者要小心该类的成员变量 queryIdx 和 trainIdx，它们必须要对齐，因为它们必须使用 matcher.match() 函数的对齐方式。下面这部分代码展示了如何在两个二维点组成的集合中对齐一个匹配，以及如何用这些匹配来找到基础矩阵：

```
vector<Point2f>imgpts1,imgpts2;
for( unsigned inti = 0; i<matches.size(); i++ )
{
    // queryIdx is the "left" image
    imgpts1.push_back(keypoints1[matches[i].queryIdx].pt);
    // trainIdx is the "right" image
```



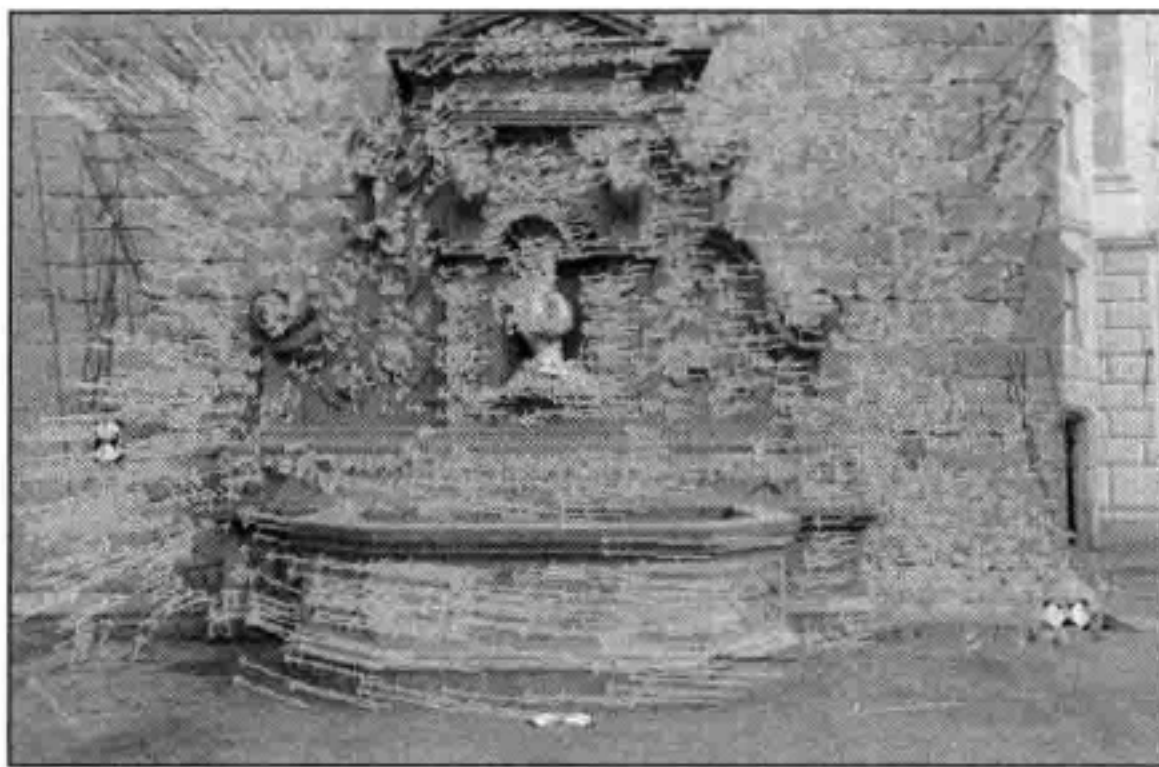
```

imgpts2.push_back(keypoints2[matches[i].trainIdx].pt);
}

Mat F = findFundamentalMat(imgpts1, imgpts2, FM_RANSAC, 0.1, 0.99,
status);
Mat_<double> E = K.t() * F * K; //according to HZ (9.12)

```

稍后会使用二进制向量 `status` 来修剪那些同所恢复的基础矩阵对齐的点。下图是在修剪基础矩阵后一个点匹配的例子。红色箭头表示在查找矩阵 `F` 的过程中删除的特征匹配，而绿色箭头是保留下来的特征匹配。



(附彩图)

下面来查找摄像机矩阵。该过程在 H 和 Z 的书的第 9 章有描述。但本项目将采用一个直接且简单的方式来实现它，因为 OpenCV 使这样的实现变得很容易。首先需简单了解一下所使用的摄像机的矩阵结构。

$$P = [R|t] = \begin{bmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \end{bmatrix}$$

这也是本项目摄像机模型，它由两部分构成：旋转（用 `R` 表示）和平移（用 `t` 表示）。关于这个矩阵有一个很重要的公式： $x=PX$ ，其中  $x$  是图像中的二维点， $X$  是三维空间中的点。也许还有其他的一些重要的信息，但该矩阵给出了图像点与场景点之间的重要关系，因此，现在可利用这些信息来找到摄像机矩阵。接下来介绍如何实现这个过程。下面的代码展示如何将本征矩阵分解为旋转和平移两部分。

```

SVD svd(E);
Matx33d W(0,-1,0, //HZ 9.13
1,0,0,
0,0,1);
Mat_<double> R = svd.u * Mat(W) * svd.vt; //HZ 9.19
Mat_<double> t = svd.u.col(2); //u3

```

```
Matx34d P1( R(0,0),R(0,1), R(0,2), t(0),
R(1,0),R(1,1), R(1,2), t(1),
R(2,0),R(2,1), R(2,2), t(2));
```

这段代码非常简单，最重要的地方是对前面得到的本征矩阵采用了奇异值分解 (Singular Value Decomposition, SVD)，然后与一个具体矩阵  $W$  相乘。本章不会对上述代码做过多的数学解释，读者只需了解 SVD 将本征矩阵  $E$  分解成两部分：旋转部分和平移部分。实际上，最初的本征矩阵由这两个部分相乘得到。感兴趣的读者可看一下本征矩阵的公式： $E=[t]_xR$ ，这个公式可从相关的参考文献找到，从此公式可看出，它由平移与旋转两部分构成。

注意，上面介绍的内容只涉及一台摄像机，另一个摄像机矩阵如何得到呢？现在假定一个摄像机矩阵固定且是标准型（没有旋转和平移），然后执行这个操作。下面这个摄像机矩阵也是标准型：

$$P_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

只需要对这个固定矩阵进行平移和旋转，就可从本征矩阵得到另一个摄像机矩阵。这也意味着从这两个摄像机矩阵恢复的任意三维点都有第一个摄像机在真实世界的原点 (0,0,0)。

但这并不是所有的解。在 H 和 Z 的书中指出这种分解存在四种可能的摄像机矩阵，但只有其中一个是正确的，并解释了原因。这个正确的矩阵会用正的  $Z$  值产生一个重构的点（这个点位于摄像机前面）。在下一节介绍三角形法和三维重构后才能理解这个问题。

还需将错误检测增加到本项目中，因为从点匹配来计算基础矩阵经常会出错，这会影响到摄像机矩阵的生成，用错误的摄像机矩阵执行三角形法没有意义。因此需要检查所得的旋转部分是否为一个有效的旋转矩阵。注意：旋转矩阵的行列式必须为 1（或 -1），这可通过下面的简单操作来验证：

```
bool CheckCoherentRotation(cv::Mat_<double>& R) {
    if(fabsf(determinant(R))-1.0 > 1e-07) {
        cerr<<"det(R) != +/-1.0, this is not a rotation matrix"<<endl;
        return false;
    }
    return true;
}
```

下面将介绍如何将各部分封装成一个函数，用此函数来恢复矩阵  $P$ ，具体代码如下所示：

```
void FindCameraMatrices(const Mat& K,
    const Mat& Kinv,
    const vector<KeyPoint>& imgpts1,
    const vector<KeyPoint>& imgpts2,
```

```

Matx34d& P,
Matx34d& P1,
vector<DMatch>& matches,
vector<CloudPoint>& outCloud
)
{
//Find camera matrices

//Get Fundamental Matrix
Mat F = GetFundamentalMat(imgpts1,imgpts2,matches);

//Essential matrix: compute then extract cameras [R|t]
Mat_<double> E = K.t() * F * K; //according to HZ (9.12)

//decompose E to P' , HZ (9.19)
SVD svd(E,SVD::MODIFY_A);
Mat svd_u = svd.u;
Mat svd_vt = svd.vt;
Mat svd_w = svd.w;

Matx33d W(0,-1,0, //HZ 9.13
1,0,0,
0,0,1);
Mat_<double> R = svd_u * Mat(W) * svd_vt; //HZ 9.19
Mat_<double> t = svd_u.col(2); //u3

if (!CheckCoherentRotation(R)) {
    cout<<"resulting rotation is not coherent\n";
    P1 = 0;
    return;
}

P1 = Matx34d(R(0,0),R(0,1),R(0,2),t(0),
R(1,0),R(1,1),R(1,2),t(1),
R(2,0),R(2,1),R(2,2),t(2));
}

```

从上面的代码可看出：为了重构场景，需要两台摄像机。变量 **P** 是第一部摄像机的标准形矩阵。第二部摄像机的基础矩阵通过计算得到，并存放在变量 **P1** 中。下面将介绍如何使用这些摄像机矩阵来重构场景的三维结构。

### 4.3 重构场景

下面考虑利用目前所得信息来恢复场景的三维结构。正如以前所做的那样，需先了解手里用来实现这一功能的工具和信息。在上一节，从基础矩阵和本征矩阵中获取了两个摄像机矩阵；讨论过如何利用这些工具来获得三维空间中一个点的位置；然后利用所得到的



匹配点的数据来求解方程。这些匹配点也对做逼近计算时产生的计算误差有用。

现在可利用 OpenCV 来实现三角形法。下面将按照 Hartley 和 Sturm 在其文章《*Triangulation*》中所介绍的方法来操作，这篇文章实现并比较了一些三角形方法。本节将实现这篇文章所提出的其中一种线性模型，因为在 OpenCV 中实现它只需编写很简单的代码。

回忆一下，对于二维点匹配和  $P$  矩阵，有两个关键的方程： $x=PX$  和  $x'=P'X$ ，其中  $x$  和  $x'$  是匹配到的二维点， $X$  是两个摄像机拍摄到的真实世界的三维点。如果重写这两个方程，就可得到一个线性方程，并可解出  $X$  的值，这正是本项目要求的。假设  $X=(x,y,z,1)^t$ （一个合理的假设是该点不会太靠近或太远离摄像机中心），由此得到一个形如  $AX=B$  的非齐次线性方程。求解这个方程的代码如下：

```
Mat_<double> LinearLSTriangulation(
    Point3d u, //homogenous image point (u,v,1)
    Matx34d P, //camera 1 matrix
    Point3d u1, //homogenous image point in 2nd camera
    Matx34d P1 //camera 2 matrix
)
{
    //build A matrix
    Matx43d A(u.x*P(2,0)-P(0,0), u.x*P(2,1)-P(0,1), u.x*P(2,2)-P(0,2),
    u.y*P(2,0)-P(1,0), u.y*P(2,1)-P(1,1), u.y*P(2,2)-P(1,2),
    u1.x*P1(2,0)-P1(0,0), u1.x*P1(2,1)-P1(0,1), u1.x*P1(2,2)-P1(0,2),
    u1.y*P1(2,0)-P1(1,0), u1.y*P1(2,1)-P1(1,1), u1.y*P1(2,2)-P1(1,2)
    );
    //build B vector
    Matx41d B(-(u.x*P(2,3)-P(0,3)),
    -(u.y*P(2,3)-P(1,3)),
    -(u1.x*P1(2,3)-P1(0,3)),
    -(u1.y*P1(2,3)-P1(1,3)));

    //solve for X
    Mat_<double> X;
    solve(A,B,X,DECOMP_SVD);

    return X;
}
```

这样就可通过两个二维点来近似一个三维点。还要注意一点：二维点是用齐次坐标（homogeneous coordinates）来表示，这意味着这些点需要增加一个值为 1 的坐标分量。同时确保这些点有归一化坐标，即它们要先与标定矩阵相乘。读者也许会注意到：并不是将每个点与矩阵  $K$  相乘，而是简单地利用  $KP$  矩阵（矩阵  $K$  与矩阵  $P$  相乘）， $H$  和  $Z$  的整个第 9 章都是这样做的。对匹配点，通过一个循环来实现完整的三角形法，代码如下：

```
double TriangulatePoints(
    const vector<KeyPoint>& pt_set1,
    const vector<KeyPoint>& pt_set2,
    const Mat&Kinv,
```

```

const Matx34d& P,
const Matx34d& P1,
vector<Point3d>& pointcloud)
{
vector<double> reproj_error;
for (unsigned int i=0; i<pts_size; i++) {
    //convert to normalized homogeneous coordinates
    Point2f kp = pt_set1[i].pt;
    Point3d u(kp.x,kp.y,1.0);
    Mat_<double> um = Kinv * Mat_<double>(u);
    u = um.at<Point3d>(0);
    Point2f kp1 = pt_set2[i].pt;
    Point3d u1(kp1.x,kp1.y,1.0);
    Mat_<double> um1 = Kinv * Mat_<double>(u1);
    u1 = um1.at<Point3d>(0);

    //triangulate
    Mat_<double> X = LinearLSTriangulation(u,P,u1,P1);

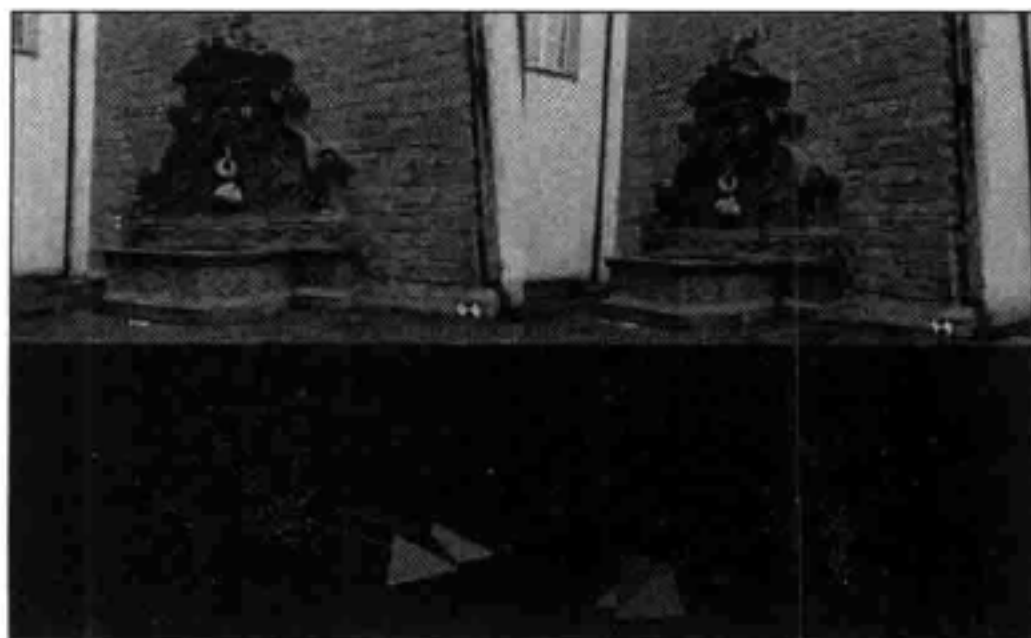
    //calculate reprojection error
    Mat_<double> xPt_img = K * Mat(P1) * X;
    Point2f xPt_img_(xPt_img(0)/xPt_img(2),xPt_img(1)/xPt_img(2));
    reproj_error.push_back(norm(xPt_img_-kp1));

    //store 3D point
    pointcloud.push_back(Point3d(X(0),X(1),X(2)));
}

//return mean reprojection error
Scalar me = mean(reproj_error);
return me[0];
}

```

下图是对两幅图执行三角形法后的结果，这两幅图取自 Fountain P-11 序列，该序列可在 <http://cvlabwww.epfl.ch/data/multiview/denseMVS.html> 下载。顶上的两幅图是场景的两幅原始视图。底部的两幅图是从上面这两幅视图得到的重构点云（point cloud）视图，包括从估计的摄像机中所看到的喷泉。可看到：红色砖墙的右侧部分被重建，并且还突出墙上的喷泉。



(附彩图)

但前面讨论过重构还有一个问题，即度量尺度不统一问题，应该花点儿时间来理解一下度量尺度不统一（up-to-scale）的含义。两个摄像机之间获得运动有任意度量单位，即它不需要指定是厘米或英寸，只需要给出一个单位刻度即可。用于重构摄像机的尺度单位（unit of scale）不统一，这会对后面从多个相机恢复三维结构产生很大的影响，因为每一对摄像机都有自己的刻度单位，而不是统一的刻度单位。

现在讨论如何建立错误度量以便找到更具有鲁棒性的重构。首先应注意重投影意味着只需将三维点三角化并重映射（reimage）到摄像机上以得到一个重投影二维点，然后计算最初的二维点和重投影二维点之间的距离，如果距离较大，则说明在三角化上有误差，不需要将该点加到最终结果里。全局度量采用平均重投影距离，这也给出一些如何将三角化整体表现出来的提示（hint）。高的平均重投影率可能表明 P 矩阵有问题，该问题可能与本征矩阵计算或匹配特征点有关。

可简单回顾一下上一节对摄像机矩阵的讨论。前面提到可由四种不同的方法获取摄像机矩阵 P1，但只有一种结构是正确的。现在已知如何三角化一个点，可增加一个检测确定这四个摄像机矩阵中的哪一个有效。本书将省掉该问题的具体实现细节，因为这些功能可在本书的示例代码中找到。

下面将讨论从多个摄像机恢复同一场景，并组合成一个三维重构结果。

## 4.4 从多视图中重构

现在已经了解如何从两个摄像机恢复运动和场景的几何结构，好像简单地采用同样过程就能得到其他摄像机参数和更多场景点。但事实并不如此简单，因为得到的重构会单位不统一，且每对图像采用了不同的尺度。

有很多方法可以从多视图中正确地重构三维场景。一种方法是摄像机标定（camera resection）或摄像机姿态估计，也就是众所周知的透视 N 点法（Perspective N Point, PNP），可通过使用已知场景点的新摄像机位置来求解该方法。另一种方法是三角化更多点并查看这些点是如何融入存在的几何结构中，该算法通过迭代最近点法（Iterative Closest Point, ICP）来得到新摄像机位置。本节将介绍使用 OpenCV 的 solvePnP 函数来实现第一种方法。

这种重构方法的第一步为：有摄像机标定的增量三维重构，即获得摄像机场景结构的基准线。由于需要寻找任意基于已知场景结构的新摄像机位置，因此需要查找初始结构和摄像机工作的基准线。可使用前面已讨论过的方法，例如：在第一帧和第二帧之间通过寻找摄像机矩阵（使用 FindCameraMatrices 函数）和三角化几何结构（使用 TriangulatePoints 函数）来得到基准线。

在有了初始结构后，就可继续后面的工作。但在开始之前，非常有必要罗列一些注意事项。首先要注意 solvePnP 函数需要两个对齐的三维和二维点向量。对齐的向量是指一个向量的第 i 个位置与另一个向量的第 i 个位置对齐。为了得到这些向量，需要在前面恢复的



三维点中查找这些点，这些点同新帧中的二维点对齐。有一个简单方法可实现该功能，对每个三维点云，用一个向量来表示二维点的位置，然后使用特征匹配来得到一对匹配点。

下面为 3D 点引入新的结构：

```
struct CloudPoint {
    cv::Point3d pt;
    std::vector<int> index_of_2d_origin;
};
```

上面的三维点包含有一个二维点索引，该索引对应每帧的二维点向量中一个二维点，该索引对三维点有帮助。当三角化一个新的三维点，并记录下三角化所涉及的哪个摄像机时，`index_of_2d_origin` 的信息必须被初始化，然后通过 3D 点云来回溯每帧的二维点。整个过程如下：

```
std::vector<CloudPoint> pcloud; //our global 3D point cloud

//check for matches between i'th frame and 0'th frame (and thus the
current cloud)
std::vector<cv::Point3f> ppcloud;
std::vector<cv::Point2f> imgPoints;
vector<int> pcloud_status(pcloud.size(),0);

//scan the views we already used (good_views)
for (set<int>::iterator done_view = good_views.begin(); done_view !=
good_views.end(); ++done_view)
{
    int old_view = *done_view; //a view we already used for
reconstruction

    //check for matches_from_old_to_working between
    <working_view>'th frame and <old_view>'th frame (and thus
    the current cloud)
    std::vector<cv::DMatch> matches_from_old_to_working =
    matches_matrix[std::make_pair(old_view,working_view)];
    //scan the 2D-2D matched-points
    for (unsigned int match_from_old_view=0;
    match_from_old_view<matches_from_old_to_working.size();
    match_from_old_view++) {
        // the index of the matching 2D point in <old_view>
        int idx_in_old_view =
        matches_from_old_to_working[match_from_old_view].queryIdx;

        //scan the existing cloud to see if this point from <old_view>
        exists for (unsigned int pcldp=0; pcldp<pcloud.size(); pcldp++) {
            // see if this 2D point from <old_view> contributed to this 3D
            point in the cloud
            if (idx_in_old_view == pcloud[pcldp].index_of_2d_origin[old_view]
                && pcloud_status[pcldp] == 0) //prevent duplicates
            {
                //3d point in cloud
                ppcloud.push_back(pcloud[pcldp].pt);
```

```

//2d point in image <working_view>
Point2d pt_ = imgpts[working_view][matches_from_old_to_
working[match_from_old_view].trainIdx].pt;
imgPoints.push_back(pt_);

pcloud_status[pcl_dp] = 1;
break;
}
}
}
}
cout<<"found "<<ppcloud.size() <<" 3d-2d point correspondences"<<endl;

```

现在对于一个新的帧而言，场景中的三维点配对（pairing）与二维点一一对齐，可使用这些点来恢复摄像机位置。具体过程如下：

```

cv::Mat_<double> t, rvec, R;
cv::solvePnPRansac(ppcloud, imgPoints, K, distcoeff, rvec, t, false);

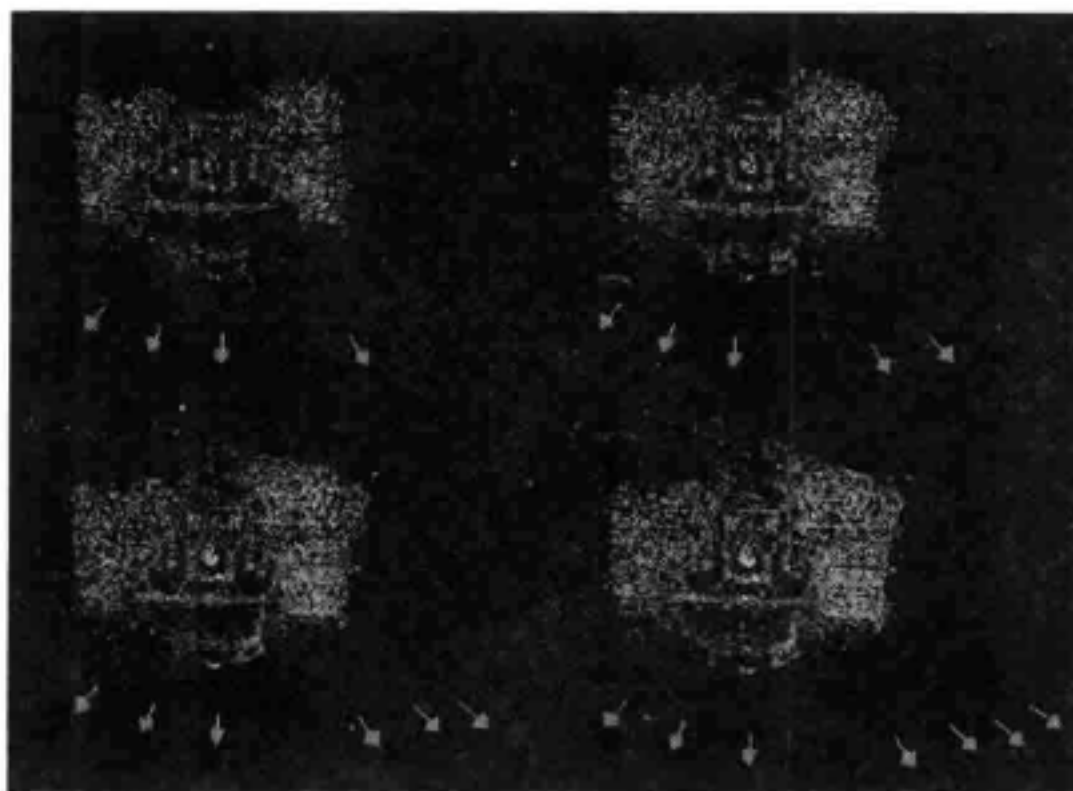
//get rotation in 3x3 matrix form
Rodrigues(rvec, R);

P1 = cv::Matx34d(R(0,0),R(0,1),R(0,2),t(0),
R(1,0),R(1,1),R(1,2),t(1),
R(2,0),R(2,1),R(2,2),t(2));

```

注意，这里使用 solvePnPRansac 函数而不是 solvePnp 函数的原因是 solvePnPRansac 函数对于噪声有更好的鲁棒性，现在得到了一个新矩阵 P1。可简单地再次使用前面定义的 TriangulatePoints 函数，将用更多的三维点填充进点云。

在下图中，从 Fountain-P11 场景的第3幅图开始，可看到对它一个增量重构，Fountain-P11 场景可从 <http://cvlabwww.epfl.ch/data/multiview/denseMVS.html> 下载得到。左上角那幅图是用4幅图重构后的结果。参与拍摄的摄像机用红色金字塔表示，白线表示方向。其他图像展示如何随着摄像机增加，有更多的点增加进点云中。



（附彩图）

## 4.5 重构的细化

细化和优化重构的场景是 SfM 方法的最重要部分之一，也称为光束调整（Bundle Adjustment, BA）。在这个优化步骤中，所有收集的数据都适用于整个模型。三维点的位置和摄像机位置都要被优化，因此，重投影误差应被最小化（即被近似的三维点被投影到图像上，使其尽量靠近原来二维点的位置）。这个过程通常需要解一个很大的线性方程组，该方程组的参数规模会上万。该过程可能会有点慢，但前面采用的步骤会更容易集成光束调整器（bundle adjuster）。例如：对点云中每个三维点要保留原始的二维点。

对光束调整算法的其中一种实现是简单稀疏光束调整（Simple Sparse Bundle Adjustment, SSBA）库。本项目用它作为 BA 的优化器，因为它有简单的 API。它仅需几个输入参数，这些参数很容易从数据结构中得到。SSBA 算法中的关键对象是 `commonInternalsMetricBundleOptimizer` 函数，它用来执行优化。该函数需要摄像机参数，三维点云，与点云中每个点对应的二维图像的点，在摄像机呈现的场景。现在可直接得到这些参数。但应注意，BA 方法假定所有图像都采取相同的硬件，因此图像的本质一样，而用其他实现方式的 BA 没有这样的假设。可按下面的代码来执行光束调整：

```
void BundleAdjuster::adjustBundle(
    vector<CloudPoint>&pointcloud,
    const Mat&cam_intrinsics,
    const std::vector<std::vector<cv::KeyPoint>>&imgpts,
    std::map<int, cv::Matx34d>&Pmats
)
{
    int N = Pmats.size(), M = pointcloud.size(), K = -1;

    cout<<"N (cams) = "<< N <<" M (points) = "<< M <<" K (measurements) = "
    << K <<endl;

    StdDistortionFunction distortion;

    // intrinsic parameters matrix
    Matrix3x3d KMat;
    makeIdentityMatrix(KMat);
    KMat[0][0] = cam_intrinsics.at<double>(0,0);
    KMat[0][1] = cam_intrinsics.at<double>(0,1);
    KMat[0][2] = cam_intrinsics.at<double>(0,2);
    KMat[1][1] = cam_intrinsics.at<double>(1,1);
    KMat[1][2] = cam_intrinsics.at<double>(1,2);

    ...

    // 3D point cloud
    vector<Vector3d> Xs(M);
    for (int j = 0; j < M; ++j)
    {
```



```

Xs[j][0] = pointcloud[j].pt.x;
Xs[j][1] = pointcloud[j].pt.y;
Xs[j][2] = pointcloud[j].pt.z;
}
cout<<"Read the 3D points."<<endl;

// convert cameras to BA datastructs
vector<CameraMatrix> cams(N);
for (inti = 0; i< N; ++i)
{
    intcamId = i;
    Matrix3x3d R;
    Vector3d T;

    Matx34d& P = Pmats[i];

    R[0][0] = P(0,0); R[0][1] = P(0,1); R[0][2] = P(0,2); T[0] = P(0,3);
    R[1][0] = P(1,0); R[1][1] = P(1,1); R[1][2] = P(1,2); T[1] = P(1,3);
    R[2][0] = P(2,0); R[2][1] = P(2,1); R[2][2] = P(2,2); T[2] = P(2,3);

    cams[i].setIntrinsic(Knorm);
    cams[i].setRotation(R);
    cams[i].setTranslation(T);
}
cout<<"Read the cameras."<<endl;

vector<Vector2d > measurements;
vector<int> correspondingView;
vector<int> correspondingPoint;

// 2D corresponding points
for (unsigned int k = 0; k < pointcloud.size(); ++k)
{
    for (unsigned int i=0; i<pointcloud[k].imgpt_for_img.size(); i++) {
        if (pointcloud[k].imgpt_for_img[i] >= 0) {
            int view = i, point = k;
            Vector3d p, np;

            Point cvp = imgpts[i][pointcloud[k].imgpt_for_img[i]].pt;
            p[0] = cvp.x;
            p[1] = cvp.y;
            p[2] = 1.0;

            // Normalize the measurements to match the unit focal length.
            scaleVectorIP(1.0/f0, p);
            measurements.push_back(Vector2d(p[0], p[1]));
            correspondingView.push_back(view);
            correspondingPoint.push_back(point);
        }
    }
}

```

```

    } // end for (k)

    K = measurements.size();

    cout<<"Read "<< K <<" valid 2D measurements."<<endl;

    ...

    // perform the bundle adjustment
    {
        CommonInternalsMetricBundleOptimizeropt(V3D::FULL_BUNDLE_FOCAL_
        LENGTH_PP, inlierThreshold, K0, distortion, cams, Xs, measurements,
        correspondingView, correspondingPoint);

        opt.tau = 1e-3;
        opt.maxIterations = 50;
        opt.minimize();

        cout<<"optimizer status = "<<opt.status<<endl;
    }

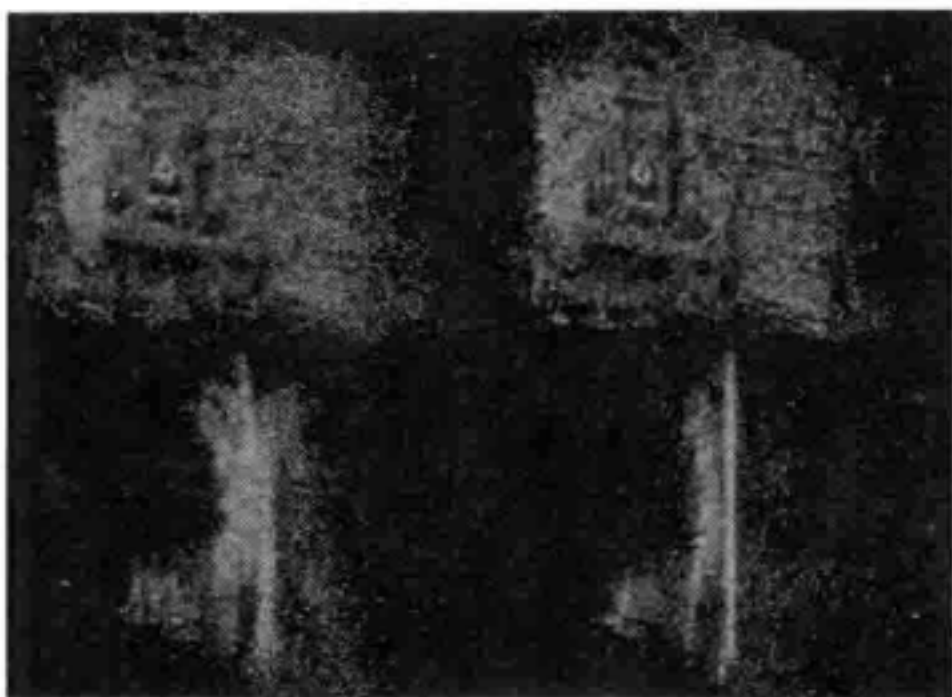
    ...

    //extract 3D points
    for (unsigned int j = 0; j <Xs.size(); ++j)
    {
        pointcloud[j].pt.x = Xs[j][0];
        pointcloud[j].pt.y = Xs[j][1];
        pointcloud[j].pt.z = Xs[j][2];
    }
    //extract adjusted cameras
    for (int i = 0; i< N; ++i)
    {
        Matrix3x3d R = cams[i].getRotation();
        Vector3d T = cams[i].getTranslation();
        Matx34d P;
        P(0,0) = R[0][0]; P(0,1) = R[0][1]; P(0,2) = R[0][2]; P(0,3) = T[0];
        P(1,0) = R[1][0]; P(1,1) = R[1][1]; P(1,2) = R[1][2]; P(1,3) = T[1];
        P(2,0) = R[2][0]; P(2,1) = R[2][1]; P(2,2) = R[2][2]; P(2,3) = T[2];
        Pmats[i] = P;
    }
}

```

这段代码虽然长，但它主要实现内部数据结构与 SSBA 的数据结构之间的转换，并执行优化处理。

下图展示了 BA 的效果。左边两幅图是调整前来自两个角度的点云，而右边的图像是优化后的点云。这两幅图之间的变化很明显，即来自不同视图的点被三角化后没有对齐，但现在大都对齐了。另外也可看到这种调整是怎样创建更好的平面重构的。



(附彩图)

## 4.6 用 PCL 来可视化 3D 点云

若用三维数据来运行程序，很难通过检查重投影误差或原始点信息发现结果是否完全正确。但若检查点云本身，会立即验证其正确性。为了可视化，本项目将使用与 OpenCV 有紧密联系的一个新项目，它叫点云库（Point Cloud Library, PCL）。它为可视化提供了很多工具，也可用来分析点云。如果读者的目标不是一个点云，而是一些高阶（high-order）信息，如：三维模型，这些工具非常有用。

首先应用 PCL 的数据结构来表示点云（实际上是一些三维点的列表）。可按如下方式实现：

```

pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud;

void PopulatePCLPointCloud(const vector<Point3d>& pointcloud,
const std::vector<cv::Vec3b>& pointcloud_RGB
)
//Populate point cloud
{
cout<<"Creating point cloud...";
cloud.reset(new pcl::PointCloud<pcl::PointXYZRGB>);

for (unsigned int i=0; i<pointcloud.size(); i++) {
// get the RGB color value for the point
Vec3b rgbv(255,255,255);
if (pointcloud_RGB.size() >= i) {
rgbv = pointcloud_RGB[i];
}

// check for erroneous coordinates (NaN, Inf, etc.)
if (pointcloud[i].x != pointcloud[i].x || isnan(pointcloud[i].x) ||
pointcloud[i].y != pointcloud[i].y || isnan(pointcloud[i].y) ||
pointcloud[i].z != pointcloud[i].z || isnan(pointcloud[i].z) ||
fabsf(pointcloud[i].x) > 10.0 ||
fabsf(pointcloud[i].y) > 10.0 ||

```



```

fabsf(pointcloud[i].z) > 10.0) {
continue;
}

pcl::PointXYZRGB pclp;

// 3D coordinates
pclp.x = pointcloud[i].x;
pclp.y = pointcloud[i].y;
pclp.z = pointcloud[i].z;

// RGB color, needs to be represented as an integer
uint32_t rgb = ((uint32_t)rgbv[2] << 16 | (uint32_t)rgbv[1] << 8 |
(uint32_t)rgbv[0]);
pclp.rgb = *reinterpret_cast<float*>(&rgb);

cloud->push_back(pclp);
}

cloud->width = (uint32_t) cloud->points.size(); // number of points
cloud->height = 1; // a list of points, one row of data
}

```

为了让可视化目标有一个很好的效果，也可将图像的 RGB 值作为彩色数据。还可对原始点云采用滤波器来删除其中可能的孤立点，所使用的工具为统计孤立点删除（statistical outlier removal, SOR）：

```

Void SORFilter() {

pcl::PointCloud<pcl::PointXYZRGB>::Ptr cloud_filtered (new pcl::PointC
loud<pcl::PointXYZRGB>);

std::cerr<<"Cloud before SOR filtering: "<< cloud->width * cloud-
>height <<" data points"<<std::endl;

// Create the filtering object
pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB>sor;
sor.setInputCloud (cloud);
sor.setMeanK (50);
sor.setStddevMulThresh (1.0);
sor.filter (*cloud_filtered);

std::cerr<<"Cloud after SOR filtering: "<<cloud_filtered->width *
cloud_filtered->height <<" data points "<<std::endl;

copyPointCloud(*cloud_filtered,*cloud);
}

```

然后使用 PCL 的 API 来运行一个简单的点云可视化，具体代码如下：

```

Void RunVisualization(const vector<cv::Point3d>& pointcloud,
const std::vector<cv::Vec3b>& pointcloud_RGB) {
PopulatePCLPointCloud(pointcloud,pointcloud_RGB);
SORFilter();
copyPointCloud(*cloud,*orig_cloud);

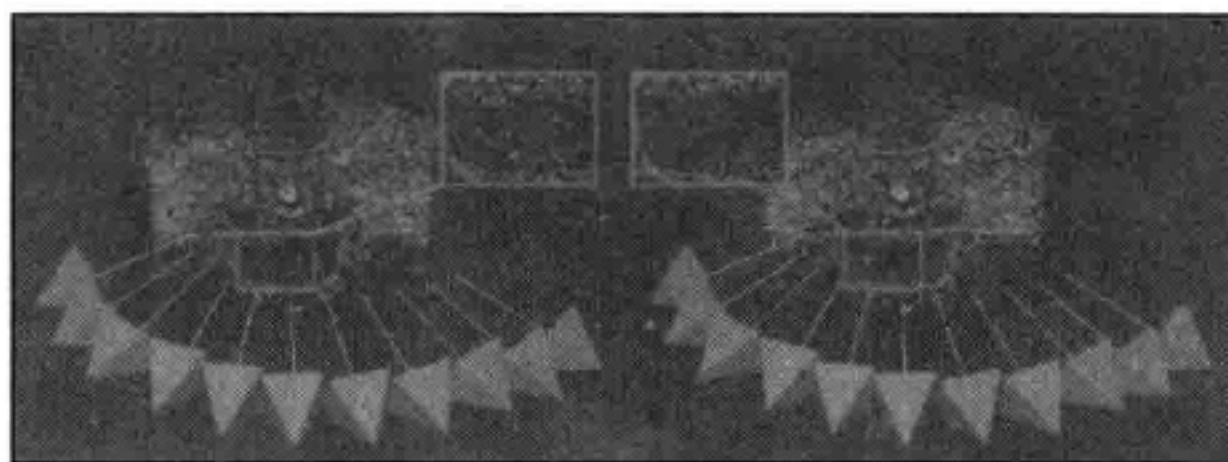
pcl::visualization::CloudViewer viewer("Cloud Viewer");

// run the cloud viewer
viewer.showCloud(orig_cloud,"orig");

while (!viewer.wasStopped ())
{
// NOP
}
}

```

下图是使用统计孤立点删除之后的结果。其中，左图为由 SfM 得到的原始点云，该图有摄像机位置和某部分点云放大的视图。右图为采用 SOR 操作后，采用滤波得到的点云。可看到一些零星散落的点被删除，留下干净的点云。



(附彩图)

## 4.7 使用示例代码

可在随书资料中找到 SfM 示例代码。现在来看如何生成、运行和使用这个项目。代码用 CMake 编译，CMake 是一个跨平台的编译环境，与 Maven 或 SCons 类似。为了生成这个应用程序，读者还应确定满足下列所有先决条件：

- ❑ OpenCV 2.3 或更高版本
- ❑ PCL 1.6 或更高版本
- ❑ SSBA 3.0 或更高版本

首先，读者应安装编译环境。为此，可创建一个名为 build 的文件夹，所有与生成应用相关的文件都放在这里。假定所有命令行操作都在文件夹 build 中进行，若不使用 build 文件夹，但其过程也相似（都需要到文件位置）。

读者应确定 CMake 能找到 SSBA 和 PCL。如果 PCL 被正确安装，则不会有这样的问

题。但为了找到 SSBA 预先生成的可执行程序，必须要通过 `-DSSBA_LIBRARY_DIR=...` 参数来设置正确位置。如果读者使用的是 Windows 操作系统，可用 Microsoft Visual Studio 来生成应用。在这种情况下，生成应用需执行如下命令：

```
cmake -G "Visual Studio 10" -DSSBA_LIBRARY_DIR=../3rdparty/SSBA-3.0/build/ ..
```

如果使用 Linux、Mac OS 或其他类 Unix 操作系统，可执行下面的命令：

```
cmake -G "Unix Makefiles" -DSSBA_LIBRARY_DIR=../3rdparty/SSBA-3.0/build/ ..
```

如果读者喜欢使用 Mac OS 上的 XCode，则可执行下面的命令：

```
cmake -G Xcode -DSSBA_LIBRARY_DIR=../3rdparty/SSBA-3.0/build/ ..
```

CMake 也可针对 Eclipse 生成宏、代码块，以及其他的功能。在 CMake 创建完环境后，就可生成应用。如果读者使用的是类 UNIX 操作系统，简单执行 `make` 命令就可生成应用，不然，需按其他开发环境的生成过程来生成应用。

在生成应用之后，就会得到名为 `ExploringSfMExec` 的可执行程序，它会执行 SfM 过程。不带参数运行这个可执行程序会得到如下结果：`USAGE: ./ExploringSfMExec <path_to_images>`

为了在一组图像上执行这个应用程序，应提供图像文件在磁盘驱动器上的位置。如果提供了一个有效位置，应用程序就会运行起来并在屏幕上看到处理过程和调试信息。当图像的点云显示后，这个应用程序就会结束。按下数字键 1 和数字键 2 会在调整和不调整点云中切换。

## 4.8 总结

本章介绍了如何用 OpenCV 来实现从结构恢复运动，所编写的代码既简单又容易理解。OpenCV 有许多有用的函数和数据结构，这使实现本项目变得容易且清晰。

但最新的 SfM 算法远比本项目复杂。许多问题和更多的常规错误检查因简单化而被忽略掉。也可对本项目进行修改来实现不同的 SfM。比如：H 和 Z 提出一种高精度的三角化方法，该方法在图像域中最小化重投影误差；一些方法在理解多幅图像的特征之间的关系后，甚至会使用 N 视图的三角形法。

如果读者想扩展和深化当前这个 SfM 算法，可看一些其他的 SfM 开源库，这一定是有益的。libMV 是一个很不错的项目，它实现了 SfM 的很多部分，为了得到最好的效果，可将这些部分与本项目互换。华盛顿大学有一个很出名的小组，他们提供了很多 SfM 风格的工具（如：Bundler 和 VisualSfM）。该工作是 Microsoft 公司旗下一个名为 PhotoSynth 产品的原型。SfM 更多的实现很容易在网上找到，只要读者去网上搜索一下，就有很多。

本章并没有深入讨论另一个重要的联系，即 Sfm 与 Visual Localization and Mapping 和



著名的 Simultaneous Localization and Mapping (SLAM) 方法之间的联系。本章的 SfM 所处理的是图像数据集和视频，但一些应用无法事先准备数据集，必须要能实时处理这种问题。这个过程称为映射 (Mapping)，在创建三维地图时就要这样做。该过程需使用特征匹配、二维空间跟踪，然后三角化。

下一章将介绍如何使用 OpenCV 从图像中提取车牌号，这个过程会使用多种机器学习技术。

## 4.9 参考文献

- *Multiple View Geometry in Computer Vision*, Richard Hartley and Andrew Zisserman, Cambridge University Press
- *Triangulation*, Richard I. Hartley and Peter Sturm, *Computer vision and image understanding*, Vol. 68, pp. 146-157
- <http://cvlab.epfl.ch/~strecha/multiview/denseMVS.html>
- *On Benchmarking Camera Calibration and Multi-View Stereo for High Resolution Imagery*, C. Strecha, W. von Hansen, L. Van Gool, P. Fua, and U. Thoennessen, CVPR
- <http://www.inf.ethz.ch/personal/chzach/opensource.html>
- <http://www.ics.forth.gr/~lourakis/sba/>
- <http://code.google.com/p/libmv/>
- <http://www.cs.washington.edu/homes/ccwu/vsfm/>
- <http://phototour.cs.washington.edu/bundler/>
- <http://photosynth.net/>
- [http://en.wikipedia.org/wiki/Simultaneous\\_localization\\_and\\_mapping](http://en.wikipedia.org/wiki/Simultaneous_localization_and_mapping)
- <http://pointclouds.org>
- <http://www.cmake.org>

## 基于 SVM 和神经网络的车牌识别

本章将介绍创建自动车牌识别 (Automatic Number Plate Recognition, ANPR) 所需的步骤。对于不同的情形, 实现自动车牌识别会用不同的方法和技术, 例如, IR 摄像机、固定汽车位置、光照条件等。本章着手构造一个用来检测汽车车牌 ANPR 的应用, 该应用处理的图像是从汽车 2 ~ 3 米处拍摄的, 拍摄环境的光线昏暗模糊, 并且与地面不平行、车牌在图像中有轻微的扭曲。

本章的主要目标是介绍图像分割、特征提取、模式识别基础以及两个重要的模式识别算法: 支持向量机 (Support Vector Machine, SVM) 和人工神经网络 (Artificial Neural Network, ANN), 本章的主要内容。

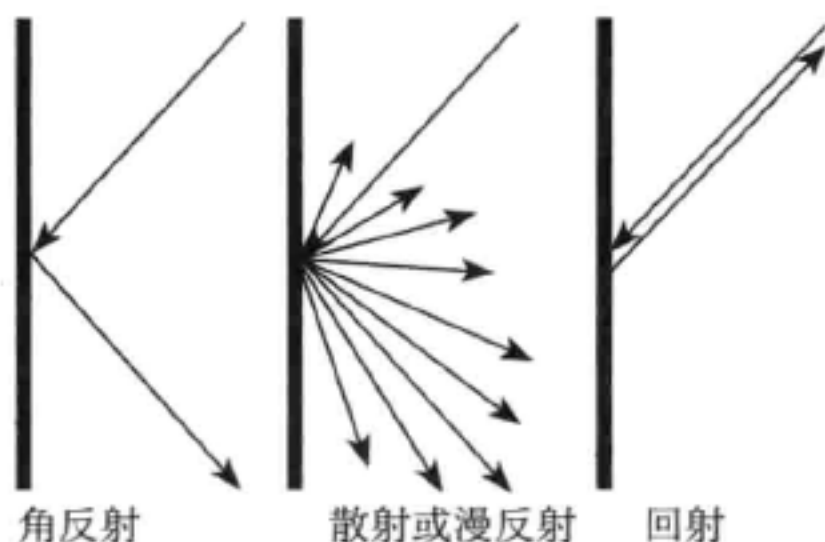
- ☐ ANPR
- ☐ 车牌检测
- ☐ 车牌识别

### 5.1 ANPR 简介

自动车牌识别 (Automatic Number Plate Recognition, ANPR) 也称自动牌照识别 (Automatic License-Plate Recognition, ALPR)、自动车辆识别 (Automatic Vehicle Identification, AVI)、洗车车牌识别 (Car Plate Recognition, CPR), 它是一种使用光学字符识别 (Optical Character Recognition, OCR) 和其他方法 (如, 用图像分割与检测) 来获取车辆牌照的监控方法。

对于一个 ANPR 系统, 其最好结果可用一个红外 (IR) 摄像机来获得数据, 因为在分割这一步中, 对检测和 OCR 分割很简单、干净, 并且误差最小。这是由光学的一些基本

原理决定的,例如入射角等于反射角,当人看到光滑表面(如,平面镜)时就会有这样的反映。粗糙表面(例如,纸)的反射会导致漫射或散射。多数车牌有一个称为回射(retro-reflection)的特性,车牌表面覆盖着一种材料,它由许多微小半球颗粒构成,会导致光线沿路反射回去,可从下图看到这种反射效果。



如果使用结合了结构性红外光学投影器的摄像机,就可只获取红外光,这样就能得到很高品质的图像,对这种图像进行分割,然后检测和识别车牌。这种情形下的车牌独立于任意光照环境,如下图所示。



本章并不会使用IR图像,而是使用普通图像。这样做并不会得到最好的结果,与使用红外摄像机相比,这种做法会得到较高的检测错误和错误识别率,但这两种方法的步骤一样。

每个国家都有不同的车牌尺寸和规格,了解这些规格对得到最好结果并减少错误很有用。本章所使用的算法是为了解释车牌识别的基础知识,其规范来自西班牙车牌,但可适用于任意国家或规范。

本章将使用西班牙车牌。在西班牙,有不同大小和形状的车牌。本章采用最普通(大)的车牌,它的大小为 $520\text{mm} \times 110\text{mm}$ 。两组字符由 $41\text{mm}$ 的空间分离,每个字符间的距离为 $14\text{mm}$ 。第一组字符为四个数字,第二组有三个字母,但不包括元音字母A、E、I、O、U,也不包括字母Ñ或Q,所有字符的大小为 $45\text{mm} \times 77\text{mm}$ 。

这些数据对字符分割很重要,因为可用这些数据来检查字符和空格,以验证得到的是一个字符而不是其他由图像分割得到的对象。下图就是一个这样的车牌。

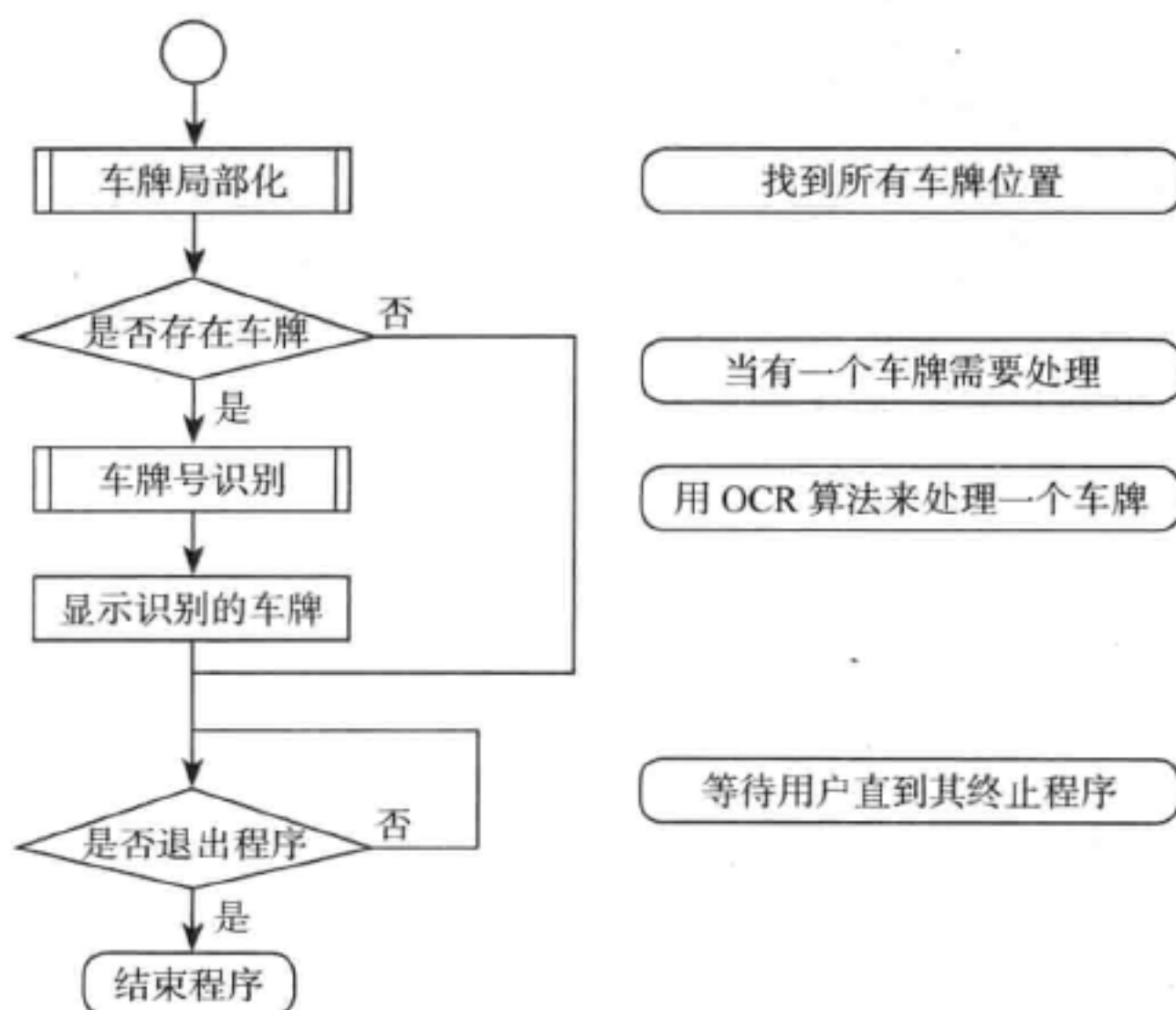




## 5.2 ANPR 算法

在解释 ANPR 代码之前，需要明白主要步骤和使用 ANPR 算法的任务。ANPR 有两个主要步骤：车牌检测和车牌识别。车牌检测的目的是在整个视频帧中检测到车牌位置。当在图像中检测到车牌时，分割的车牌被传到第二个步骤，即车牌识别，它用 OCR 算法来识别车牌上的字母和数字。

下图是两个主要算法的步骤：车牌检测和车牌识别。在完成这些步骤后，程序将在摄像机的帧上绘制已检测到的车牌字符。算法有可能给出错误结果甚至不会返回结果。

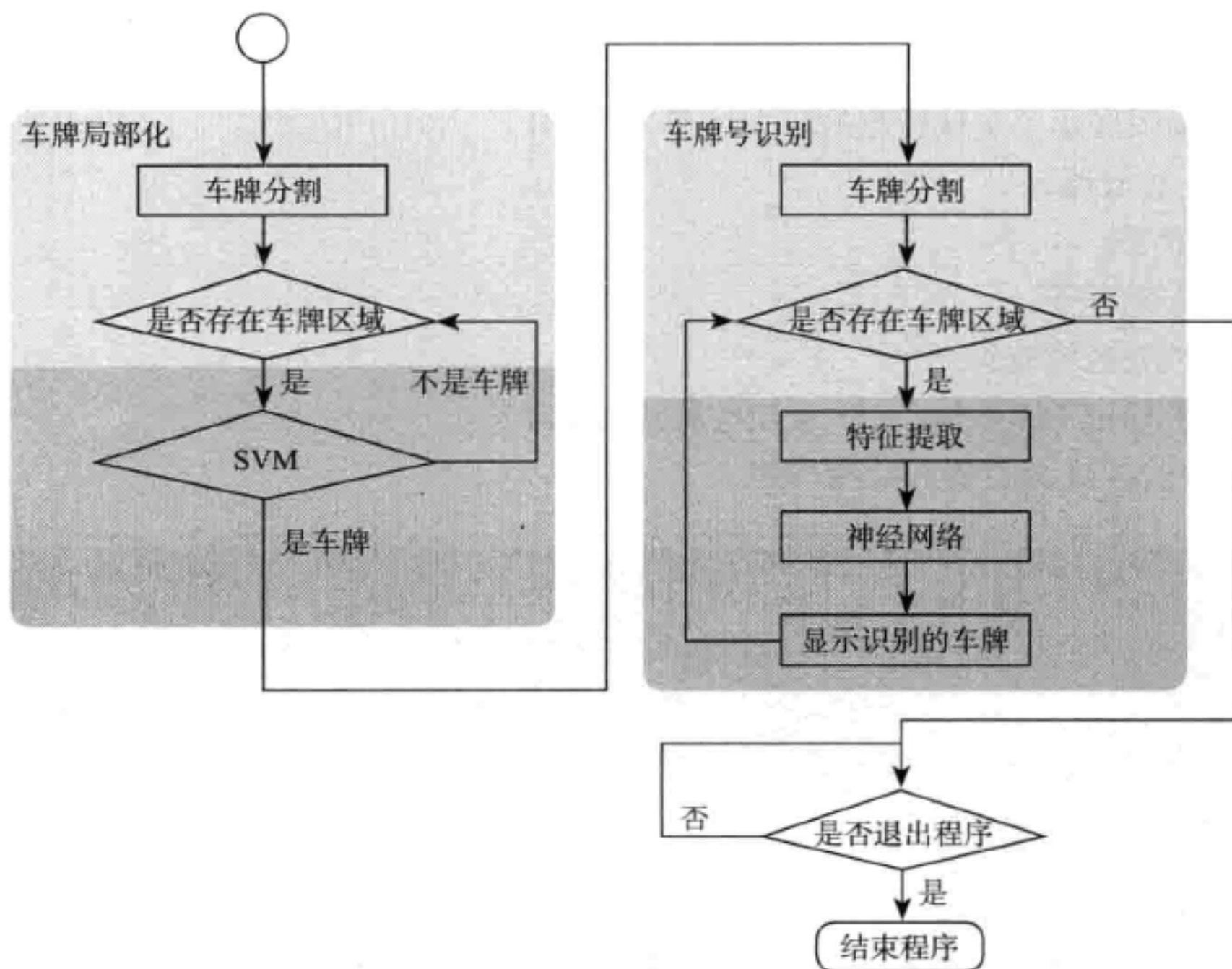


上图展示了本项目的整个步骤，下面还将定义模式识别算法常用的三个额外步骤。

- 1) 分割：这一步会检测并裁剪图像中每个感兴趣的块 / 区域；
- 2) 特征提取：这一步对字符图像集每个部分进行提取；

3) 分类: 这一步会从车牌识别那一步的结果中得到每个字符, 或从车牌检测 (plate-detection) 那一步中将所得图像块分为“是车牌”或“不是车牌”;

下图展示了整个算法应用中模式识别的步骤。



除了这个主要的应用以外, 模式识别算法的主要目的是检测和识别汽车车牌, 下面简单介绍一下两个任务, 这两个任务通常都不会解释。

- 如何训练模式识别系统;
- 如何评估模式识别系统。

但这两个任务比主要应用本身更重要, 因为如果没有正确训练模式识别系统, 整个系统可能会失败或不能正常工作。不同的模式需要不同的训练和评估类型。为了得到最好的结果, 本章需要在不同环境、条件以及特征下评估所建的系统。这两个任务有时一起使用, 因为不同特征可产生不同的结果, 可在 5.4.4 节看到。

## 5.3 车牌检测

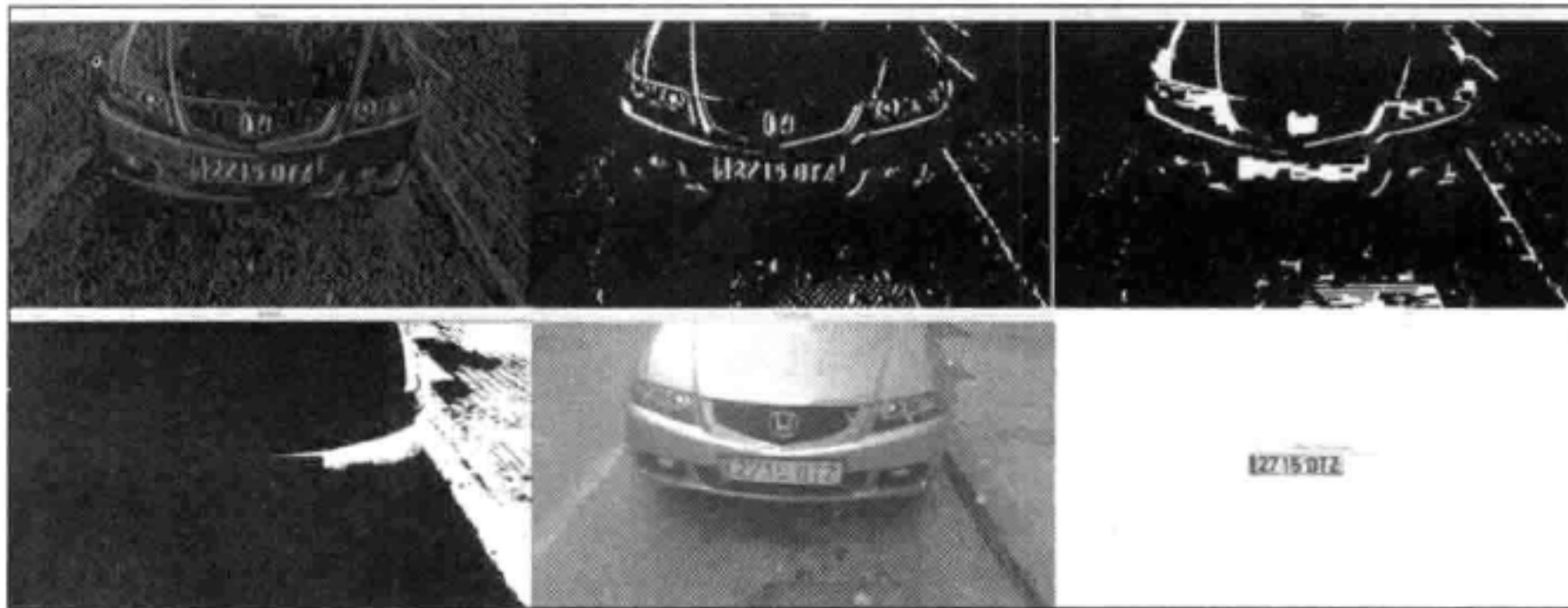
这一步要检测当前帧中所有的车牌。为了实现此功能, 该步骤又分为两个主要步骤: 图像分割和对分割的图像进行分类。这一步的功能不会解释因为将图像块作为一个向量特征。

在第一步 (图像分割) 中, 将使用各种滤波器、形态学算子, 以及轮廓算法来验证所获取图像中有车牌的部分。

在第二步（分类）中，对每个图像块（即，特征）将采用支持向量机（Support Vector Machine, SVM）作为分类器进行分类。在创建主要的应用之前，需训练两个不同的类：车牌和非车牌号。这步所使用的图像是在汽车前面 2 ~ 4 米拍摄平行的正面视角彩色图像，这些图像有 800 像素宽。这些要求对确保正确的图像分割很重要。可创建一个多尺度图像算法来进行检测。

下面这幅图展示了车牌检测的所有过程。

- ❑ Sobel 滤波器；
- ❑ 阈值算子；
- ❑ 闭形态学算子；
- ❑ 一个填充区域掩码；
- ❑ 用红色标记（特征图像中）可能检测到的车牌；
- ❑ 在执行 SVM 分类器后检测车牌。



（附彩图）

### 5.3.1 图像分割

图像分割是将图像分成多个区域的过程。该过程是为了分析而简化图像，同时也使特征提取更容易。

车牌分割有一个重要特征：假定从汽车前面拍摄图像，会在车牌上有大量竖直边（vertical edge），并且车牌不会被旋转，也没有透视扭曲（perspective distortion）。这一性质在分割图像时可用来删除没有任何竖直边的那些区域。

在找到竖直边之前，需将彩色图像转换为灰度图像（因为彩色对本任务没用），删除可能由摄像机产生的噪声或其他环境噪声。利用  $5 \times 5$  的高斯模糊来去噪。如果不用去噪方法，可能会得到很多竖直边，从而造成检测失败。

```
//convert image to gray
Mat img_gray;
cvtColor(input, img_gray, CV_BGR2GRAY);
blur(img_gray, img_gray, Size(5,5));
```



为了找到竖直边，将采用 Sobel 滤波器来找到第一个水平导数（horizontal derivative）。导数是数学函数，它可用来在图像中查找竖直边。Sobel 函数在 OpenCV 中的定义为：

```
void Sobel(InputArray src, OutputArray dst, int ddepth, int xorder,
int yorder, int ksize=3, double scale=1, double delta=0, int
borderType=BORDER_DEFAULT )
```

这里，ddepth 为目标图像的颜色深度，xorder 为对 x 求导数的阶数，yorder 为对 y 求导数的阶数，ksize 是核（kernel）的大小，其取值为 1、3、5、7 当中的一个，scale 用于计算导数值的可选因子，delta 是增加到结果的可选择值，borderType 是像素内插方法。


根据本项目的情况，将使用 xorder=1、yorder=0 和 ksize=3：

```
//Find vertical lines. Car plates have high density of vertical lines
Mat img_sobel;
Sobel(img_gray, img_sobel, CV_8U, 1, 0, 3, 1, 0);
```

在执行完 Sobel 滤波器后，将采用阈值滤波器来得到二值图像，所采用的阈值由 Otsu 算法得到。Otsu 算法的输入是一个 8 位图像，它将自动得到优化的阈值：

```
//threshold image
Mat img_threshold;
threshold(img_sobel, img_threshold, 0, 255, CV_THRESH_OTSU+CV_THRESH_
BINARY);
```

为了在 threshold 函数中使用 Otsu 算法，可将阈值的类型参数与 CV\_THRESH\_OTSU 结合，因为这样做就会忽略阈值参数（即 threshold 函数的第三个参数）。

 **注意：**当定义 CV\_THRESH\_OTSU 的值时，阈值函数会返回由 Otsu 算法得到的优化阈值。

通过采用一个闭形态学算子，可删除在每个竖直边缘线之间的空白区域，并连接有大量边的所有区域的。在这一步中，有可能包含车牌区域。

首先，需要定义在闭形态学算子中所使用的结构元素。可使用 getStructuringElement 函数来定义一个结构矩阵元素，它的维度大小为  $17 \times 3$ ，这可能与其他图像尺寸有所不同：

```
Mat element = getStructuringElement(MORPH_RECT, Size(17, 3));
```

在闭形态学算子中使用 morphologyEx 函数就会用到结构元素：

```
morphologyEx(img_threshold, img_threshold, CV_MOP_CLOSE, element);
```

在使用这些函数后，就会得到包含车牌的区域，但多数区域都不包含车牌号。这些区域可用连通分量分析（connected-component analysis）或用 findContours 函数将其分开。下面这个函数用来获取二进制图像的轮廓，若在该函数中采用的方法不同，则得到的结果轮廓就不一样。本项目只需用任意层次关系和任何多边形来逼近外部轮廓即可：

```
//Find contours of possibles plates
vector< vector< Point> > contours;
```

```

findContours(img_threshold,
             contours,           // a vector of contours
             CV_RETR_EXTERNAL,  // retrieve the external contours
             CV_CHAIN_APPROX_NONE); // all pixels of each contour

```

对每个轮廓检测和提取最小区域的有界矩形区域，可用 OpenCV 自带的 `minAreaRect` 函数来实现。该函数返回一个名为 `RotatedRect` 的旋转矩形类，然后在每个轮廓上使用一个向量迭代器（vector iterator）来得到被旋转的矩形，并在对每个区域做分类之前，做一些初步验证：

```

//Start to iterate to each contour found
vector<vector<Point> >::iterator itc= contours.begin();
vector<RotatedRect> rects;

//Remove patch that has no inside limits of aspect ratio and area.
while (itc!=contours.end()) {
//Create bounding rect of object
RotatedRect mr= minAreaRect(Mat(*itc));
if( !verifySizes(mr)){
    itc= contours.erase(itc);
}else{
    ++itc;
    rects.push_back(mr);
}
}

```

可对检测到的区域做一些基本验证，这些验证基于面积和宽高比。只有这样的区域才被认为是一个车牌：有 40% 的误差范围内的区域，其宽高比为  $520/110=4.727272$ （车牌的宽除以车牌的高），区域面积最小为 15 个像素，并且车牌的高度最多为 125 个像素。这些值的计算依靠图像大小和摄像机位置：

```

bool DetectRegions::verifySizes(RotatedRect candidate ){

    float error=0.4;
    //Spain car plate size: 52x11 aspect 4,7272
    const float aspect=4.7272;
    //Set a min and max area. All other patches are discarded
    int min= 15*aspect*15; // minimum area
    int max= 125*aspect*125; // maximum area
    //Get only patches that match to a respect ratio.
    float rmin= aspect-aspect*error;
    float rmax= aspect+aspect*error;

    int area= candidate.size.height * candidate.size.width;
    float r= (float)candidate.size.width / (float)candidate.size.height;
    if(r<1)
        r= 1/r;

    if(( area < min || area > max ) || ( r < rmin || r > rmax )){

```

```

        return false;
    }else{
        return true;
    }
}

```

可利用车牌背景为白色这一性质做出更多的改进。所有车牌都有一样的背景颜色，为了得到精确的裁剪，可使用漫水填充算法来获取旋转矩形。

裁剪车牌的第一步是要得到几个种子 (seed)，它们位于最后被旋转矩形的中心附近，然后用宽和高得到车牌的最小尺寸，并使用它在块中心 (patch center) 附近产生随机种子。

要选择白色区域，并需要这几个种子至少接触到一个白色像素。然后对于每个像素，使用 floodFill 函数绘制一个新的掩码图像以存储最接近的裁剪区域：

```

for(int i=0; i< rects.size(); i++){
    //For better rect cropping for each possible box
    //Make floodfill algorithm because the plate has white background
    //And then we can retrieve more clearly the contour box
    circle(result, rects[i].center, 3, Scalar(0,255,0), -1);
    //get the min size between width and height
    float minSize=(rects[i].size.width < rects[i].size.height)?rects[i].
    size.width:rects[i].size.height;
    minSize=minSize-minSize*0.5;
    //initialize rand and get 5 points around center for floodfill
    algorithm
    srand ( time(NULL) );
    //Initialize floodfill parameters and variables
    Mat mask;
    mask.create(input.rows + 2, input.cols + 2, CV_8UC1);
    mask= Scalar::all(0);
    int loDiff = 30;
    int upDiff = 30;
    int connectivity = 4;
    int newMaskVal = 255;
    int NumSeeds = 10;
    Rect ccomp;
    int flags = connectivity + (newMaskVal << 8 ) + CV_FLOODFILL_FIXED_
    RANGE + CV_FLOODFILL_MASK_ONLY;
    for(int j=0; j<NumSeeds; j++){
        Point seed;
        seed.x=rects[i].center.x+rand()%(int)minSize-(minSize/2);
        seed.y=rects[i].center.y+rand()%(int)minSize-(minSize/2);
        circle(result, seed, 1, Scalar(0,255,255), -1);
        int area = floodFill(input, mask, seed, Scalar(255,0,0), &ccomp,
        Scalar(loDiff, loDiff, loDiff), Scalar(upDiff, upDiff, upDiff),
        flags);
    }
}

```

floodFill 函数将一个彩色连通分量填充到图像掩码中，这个填充过程从一个种子点开始，并对要填充的像素与邻近像素或种子像素之间设置亮度 / 色彩的最大上界和下界之差。



```
int floodFill(InputOutputArray image, InputOutputArray mask, Point
seed, Scalar newVal, Rect* rect=0, Scalar loDiff=Scalar(), Scalar
upDiff=Scalar(), int flags=4 )
```

参数 `newVal` 是填充时要放到图像的新颜色。参数 `loDiff` 和 `upDiff` 分别是最大下界差和最大上界差，这些值来源于要填充的像素与邻近像素或种子像素之间的亮度 / 色彩之差。

`flag` 的参数组合为：

- ❑ 较低位 (lower bit)：这些位包含用在函数中的连通性值：4 (默认值) 或 8，连通性决定考虑像素的哪种近邻。
- ❑ 较高位 (upper bit)：这些位是 0 或下列值的组合：CV\_FLOODFILL\_FIXED\_RANGE 和 CV\_FLOODFILL\_MASK\_ONLY。

CV\_FLOODFILL\_FIXED\_RANGE 设置当前像素与种子像素之间的差别。CV\_FLOODFILL\_MASK\_ONLY 只会填充图像掩码而不会改变图像本身。

一旦得到裁剪掩码，便可通过图像掩码来得到一个最小面积的矩形，并再次检查它的有效尺寸。对于每个掩码的白色像素，先得到其位置，再用 `minAreaRect` 函数获取最接近的裁剪区域：

```
//Check new floodfill mask match for a correct patch.
//Get all points detected for minimal rotated Rect
vector<Point> pointsInterest;
Mat_<uchar>::iterator itMask= mask.begin<uchar>();
Mat_<uchar>::iterator end= mask.end<uchar>();
for( ; itMask!=end; ++itMask)
    if(*itMask==255)
        pointsInterest.push_back(itMask.pos());
RotatedRect minRect = minAreaRect(pointsInterest);
if(verifySizes(minRect)){
    ...
}
```

现在分割处理已经完成并验证了有效区域，然后可裁剪每个检测到的区域，消除任意可能的旋转，裁剪图像区域，调整图像大小，均衡裁剪图像区域的亮度。

```
//Get rotation matrix
float r= (float)minRect.size.width / (float)minRect.size.height;
float angle=minRect.angle;
if(r<1)
    angle=90+angle;
Mat rotmat= getRotationMatrix2D(minRect.center, angle,1);
```

现在可通过仿射变换（几何学中的仿射变换是指让平行线变换后仍是平行线的变换）旋转输入的图像，具体用 `warpAffine` 函数实现，需要用户设置该函数的输入图像和输出图像、变换矩阵、输出图像大小（在本应用中，它与输入图像一样大），以及所使用的插值方法。如果需要，可定义边界方法和边界值。

```
//Create and rotate image
Mat img_rotated;
warpAffine(input, img_rotated, rotmat, input.size(), CV_INTER_CUBIC);
```

在旋转图像后,可用 `getRectSubPix` 裁剪图像,使用该函数裁剪会以一个点为中心,复制给定宽度和高度的图像。如果图像被旋转,需要用 C++ 的 `swap` 函数来改变宽度和高度大小。

```
//Crop image
Size rect_size=minRect.size;
if(r < 1)
swap(rect_size.width, rect_size.height);
Mat img_crop;
getRectSubPix(img_rotated, rect_size, minRect.center, img_crop);
```

通过裁剪得到的图像并不适用于训练和分类,因为它们的大小不同。另外,每个图像包含不同的亮度,这也增大了它们的相对差异性。为了解决这个问题,可用光照直方图均衡(light histogram equalization)来调整所有图像,使它们具有相同宽度和高度。

```
Mat resultResized;
resultResized.create(33,144, CV_8UC3);
resize(img_crop, resultResized, resultResized.size(), 0, 0, INTER_
CUBIC);
//Equalize cropped image
Mat grayResult;
cvtColor(resultResized, grayResult, CV_BGR2GRAY);
blur(grayResult, grayResult, Size(3,3));
equalizeHist(grayResult, grayResult);
```

针对每个检测的区域,用一个向量来存储裁剪的图像和它的位置。

```
output.push_back(Plate(grayResult,minRect.boundingRect()));
```

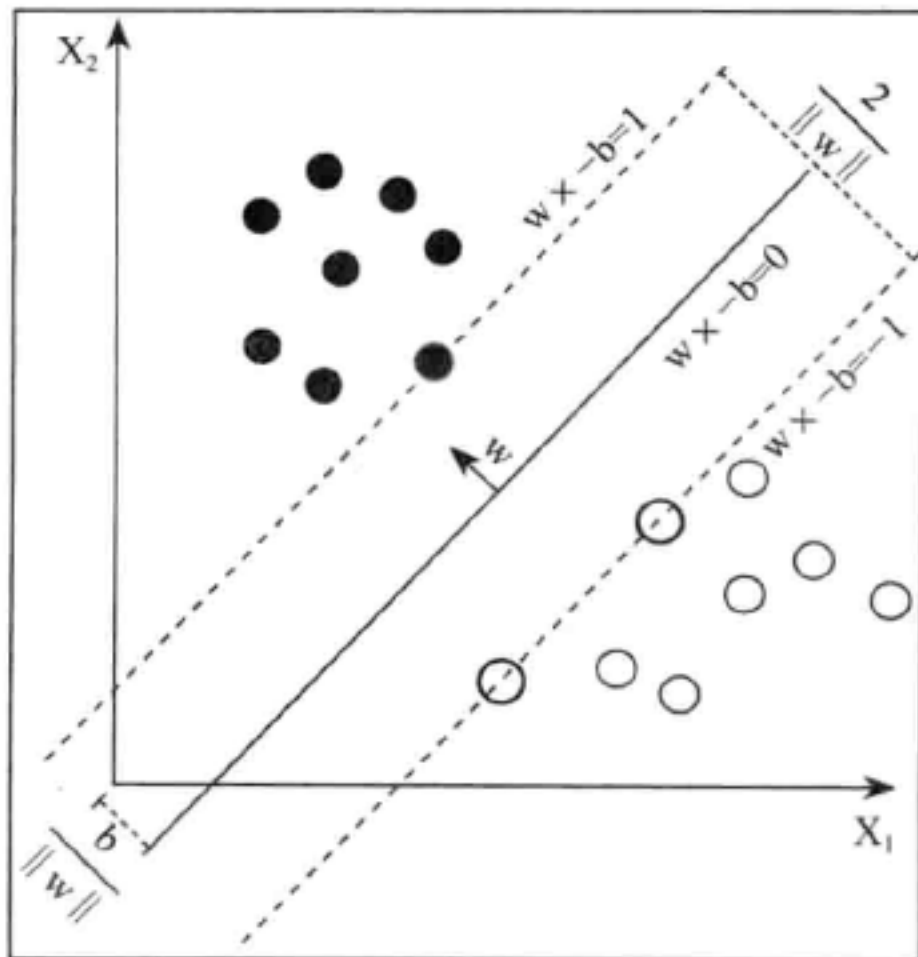
### 5.3.2 分类

在预处理和分割完图像的所有部分后,需要决定每部分是否为车牌号。可用支持向量机(Support Vector Machine, SVM)算法来完成该功能。

支持向量机是一种模式识别算法,它源于二分类的监督学习(supervised-learning)算法。监督学习是通过标签数据进行学习的机器学习算法。用户需要用一些标签数据来训练算法,标签数据是指每个样本都应属于某个具体的类。

SVM 会创建一个或多个超平面,这些超平面可用来判断数据属于哪个类。

一个经典的 SVM 示例是,对一个只有两类的二维平面的点集, SVM 搜索的最优直线以将不同类的点分开。



在开始分类之前，需要训练分类器，该工作要在主应用开始之前完成，这称为离线训练。离线训练并不是一件容易的事，因为它需要充足的数据来训练系统，但不是数据集越大就能得到最好的结果。本项目并没有充足的数据，因为并没有公开的车牌数据库。因此，需要拍摄数百张汽车照片，然后预处理并分割它们。

我们使用 75 张车牌图像和 35 张不是车牌但大小为  $144 \times 33$  像素的图像训练系统。从下图可看到这样的样本数据。这并不是一个大数据集，但对本项目而言，可得到足够好的结果。在真正的应用中，需要更多的数据。



为了简单理解机器学习是如何工作的，可对分类器算法使用图像像素特征（注意：有更好的方法和特征用于训练 SVM，比如，主成分分析（Principal Components Analysis, PCA）、傅里叶变换、纹理分析等。

需要用 DetectRegions 类来创建用于训练的图像，为了保存图像，需将 savingRegions 变量设置为 true。用脚本文件 segmentAllFiles.sh 来重复处理文件夹下所有图像文件。该文件可从本书附带的源代码中得到。

为了使处理过程变得更容易，可将用于训练和测试的所有图像数据保存到 XML 文件中，然后让 SVM 函数直接使用该 XML 文件。文件 trainSVM.cpp 可通过指定的文件夹和图像文件编号来创建这个 XML 文件。



**注意：**用于 OpenCV 的机器学习算法的训练数据存储在一个  $N \times M$  的矩阵中，其中  $N$  为样本数， $M$  为特征数，每个样本是该训练矩阵的一行。

每个样本的类别信息存储在另一个大小为  $N \times 1$  的矩阵中，每个样本的类别用一个浮点数表示。

通过 OpenCV 的 FileStorage 类可以很容易地管理一个 XML 或 JSON 格式的数据文件。该类可存取 OpenCV 的变量、结构或自定义变量。用该类的函数可读取训练数据矩阵和类标签，并可将这些信息存储到 SVM\_TrainingData 和 SVM\_Classes 中。

```
FileStorage fs;
fs.open("SVM.xml", FileStorage::READ);
Mat SVM_TrainingData;
.. Mat SVM_Classes;
```



```
fs["TrainingData"] >> SVM_TrainingData;
fs["classes"] >> SVM_Classes;
```

现在需要设置 SVM 参数，这些参数是使用 SVM 算法的基本参数，可用 CvSVMParams 结构来定义这些参数。通常为了提高训练样本的线性可分性，需要采用映射。该映射可通过核函数（kernel function）来高效地实现，这样做后可增加训练样本的维度。本项目选择 CvSVM::LINEAR 类型，这意味着不做映射：

```
//Set SVM params
CvSVMParams SVM_params;
SVM_params.kernel_type = CvSVM::LINEAR;
```

然后来创建并训练分类器。OpenCV 的 CvSVM 类就是对支持向量机算法的实现，需要用训练数据、类标签和参数数据来初始化它：

```
CvSVM svmClassifier(SVM_TrainingData, SVM_Classes, Mat(), Mat(), SVM_
params);
```

在训练好分类器后，可用 SVM 类的 predict 函数来预测裁剪图像。在本项目中，用 1 表示车牌，用 0 表示非车牌。对每一个可能被检测出是车牌的区域，用 SVM 进行分类并判断这个区域是否为车牌，如果是车牌就保存。下面的代码是主要应用部分，也称为在线处理：

```
vector<Plate> plates;
for(int i=0; i< possible_regions.size(); i++)
{
    Mat img=possible_regions[i].plateImg;
    Mat p= img.reshape(1, 1);//convert img to 1 row m features
    p.convertTo(p, CV_32FC1);
    int response = (int)svmClassifier.predict( p );
    if(response==1)
        plates.push_back(possible_regions[i]);
}
```

## 5.4 车牌号识别

车牌识别的第二步是要用光学字符识别获取车牌上的字符。对每个检测到的车牌，可着手按每个字符来分割车牌，用人工神经网络（Artificial Neural Network, ANN）这种机器学习算法来识别字符。本节将介绍如何评估一个分类算法。

### 5.4.1 OCR 分割

首先，对获取的车牌图像用直方图均衡进行处理，将其作为 OCR 函数的输入，然后采用阈值滤波器对图像进行处理，并将处理后的图像作为查找轮廓（Find Contour）算法的输入。右图展示了这个过程：



这个分割过程的代码如下：

```
Mat img_threshold;
threshold(input, img_threshold, 60, 255, CV_THRESH_BINARY_INV);
if (DEBUG)
    imshow("Threshold plate", img_threshold);
Mat img_contours;
img_threshold.copyTo(img_contours);
//Find contours of possibles characters
vector< vector< Point> > contours;
findContours(img_contours,
             contours,           // a vector of contours
             CV_RETR_EXTERNAL,  // retrieve the external contours
             CV_CHAIN_APPROX_NONE); // all pixels of each contour
```

使用 CV\_THRESH\_BINARY\_INV 可将白色输入值变成黑色，将黑色输入值变为白色，从而反转阈值化的输出结果。为了得到每个字符的轮廓，这是有必要的，因为轮廓算法会查找白色像素。

对每个检测到的轮廓，需要验证其大小并删除所有尺寸较小或宽高比不正确的区域。在本项目中，正确的车牌字符的宽高比为 45/77，但由于字符会有旋转或扭曲，允许车牌字符的宽高比有 35% 的误差。如果一块区域的这个比率超过标准比率的 80%，可认为这个区域为黑色块，而不是一个字符。可用 countNonZero 函数来计算像素值大于 0 的像素个数：

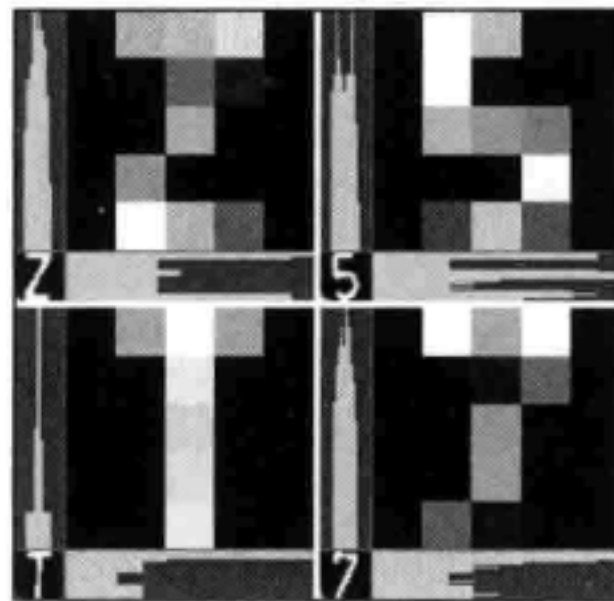
```
bool OCR::verifySizes(Mat r)
{
    //Char sizes 45x77
    float aspect=45.0f/77.0f;
    float charAspect= (float)r.cols/(float)r.rows;
    float error=0.35;
    float minHeight=15;
    float maxHeight=28;
    //We have a different aspect ratio for number 1, and it can be
    //~0.2
    float minAspect=0.2;
    float maxAspect=aspect+aspect*error;
    //area of pixels
    float area=countNonZero(r);
    //bb area
    float bbArea=r.cols*r.rows;
    //% of pixel in area
    float percPixels=area/bbArea;
    if(percPixels < 0.8 && charAspect > minAspect && charAspect <
    maxAspect && r.rows >= minHeight && r.rows < maxHeight)
        return true;
    else
        return false;
}
```

如果一个分割的区域是字符,则必须要对其预处理,使它与所有字符有一样的大小和位置,然后用辅助类 CharSegment 将其保存到一个向量中。该类保存分割后的字符图像和用于调整字符所需的位置,因为查找轮廓算法不会按所需顺序返回轮廓。

### 5.4.2 特征提取

为了用人工神经网络进行训练和分类,下面将对每个分割出来的字符进行特征提取。

与用 SVM 进行车牌检测时的特征提取不同,这里不会使用所有图像像素作为特征,而是采用光学字符识别中更常用的特征,这些特征包含了水平和竖直累积直方图,以及低分辨率的图像样本,从下图中可看出这种图形化的特征,每个图像只有  $5 \times 5$  的低分辨率,且是累积直方图 (accumulation histogram)。



对每个字符,通过使用 countNonZero 函数来按列或按行统计非零像素值个数,并将其保存到新的数据矩阵 mhist 中。对 mhist 进行归一化处理,其过程为:通过 minMaxLoc 函数找到该矩阵的最大值,将它的每个元素都除以这个最大值,并通过 convertTo 函数来最终实现。创建名为 ProjectedHistogram 的函数,用它来实现累积直方图,这个函数将二值图像和直方图类型(水平或竖直)作为输入:

```
Mat OCR::ProjectedHistogram(Mat img, int t)
{
    int sz=(t)?img.rows:img.cols;
    Mat mhist=Mat::zeros(1,sz,CV_32F);

    for(int j=0; j<sz; j++){
        Mat data=(t)?img.row(j):img.col(j);
        mhist.at<float>(j)=countNonZero(data);
    }

    //Normalize histogram
    double min, max;
    minMaxLoc(mhist, &min, &max);

    if(max>0)
        mhist.convertTo(mhist, -1, 1.0f/max, 0);

    return mhist;
}
```

将低分辨率图像作为另一种特征。不使用整幅图像,而是使用低分辨率字符图像,例如,  $5 \times 5$ 。用  $5 \times 5$ 、 $10 \times 10$ 、 $15 \times 15$  和  $20 \times 20$  字符图像来训练,评估哪个字符图像能得到最好结果,然后本项目就用这个字符图像。在得到特征之后,便可创建一个矩阵,该矩



阵每行有  $M$  个特征:

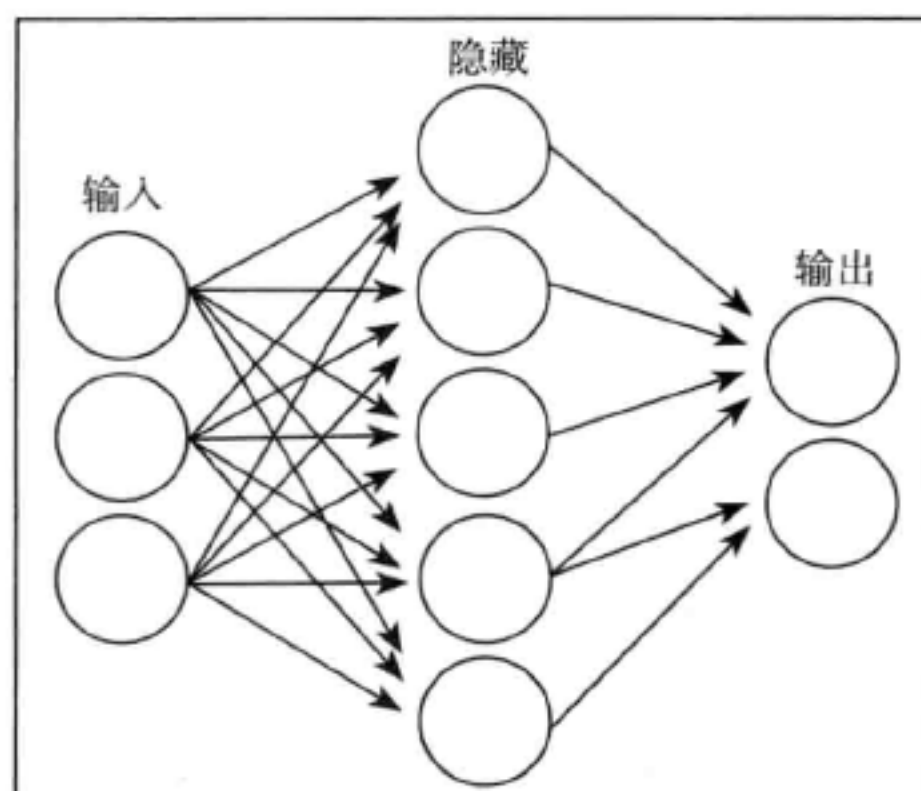
```
Mat OCR::features(Mat in, int sizeData)
{
    //Histogram features
    Mat vhist=ProjectedHistogram(in,VERTICAL);
    Mat hhist=ProjectedHistogram(in,HORIZONTAL);
    //Low data feature
    Mat lowData;
    resize(in, lowData, Size(sizeData, sizeData) );
    int numCols=vhist.cols + hhist.cols + lowData.cols *
    lowData.cols;
    Mat out=Mat::zeros(1,numCols,CV_32F);
    //Assign values to feature
    int j=0;
    for(int i=0; i<vhist.cols; i++)
    {
        out.at<float>(j)=vhist.at<float>(i);
        j++;
    }
    for(int i=0; i<hhist.cols; i++)
    {
        out.at<float>(j)=hhist.at<float>(i);
        j++;
    }
    for(int x=0; x<lowData.cols; x++)
    {
        for(int y=0; y<lowData.rows; y++)
        {
            out.at<float>(j)=(float)lowData.at<unsigned char>(x,y);
            j++;
        }
    }
    return out;
}
```

### 5.4.3 OCR 分类

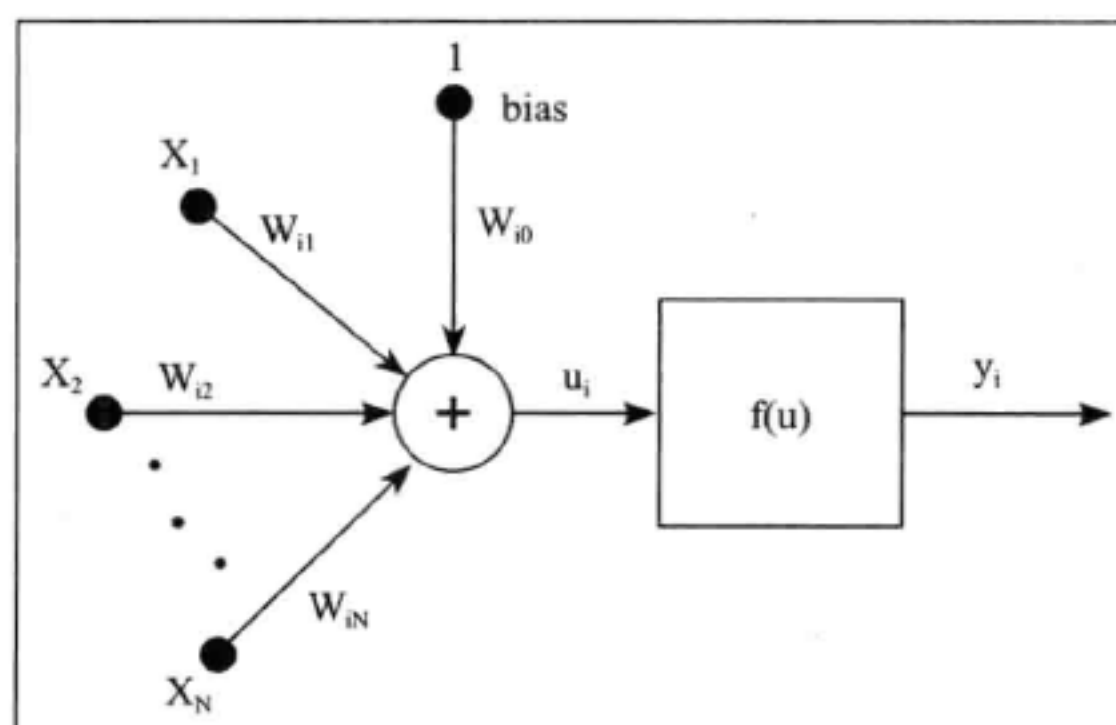
在分类这一步,将使用机器学习算法中的人工神经网络。更具体点,使用多层感知器 (Multi-Layer Perceptron, MLP),它是最常见的 ANN 算法。

MLP 由包含一个输入层、包含一个输出层和一个或多个隐藏层的神经网络组成。每一层由一个或多个神经元同前一层和后一层相连。

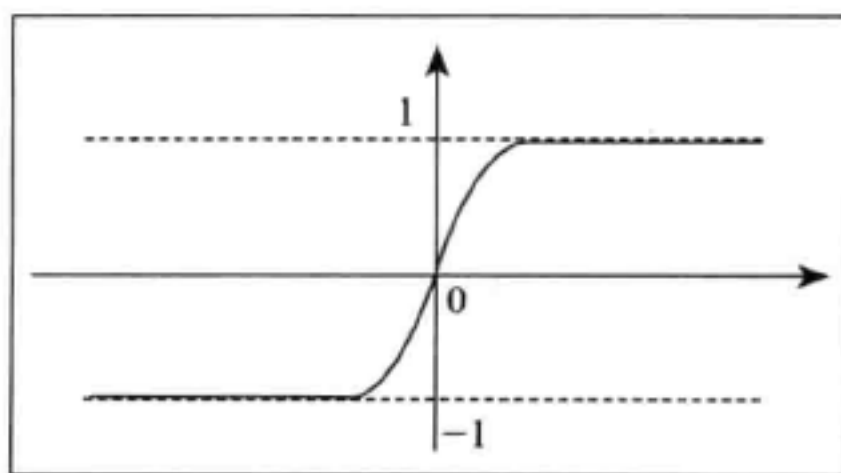
下图表示一个 3 层的神经元感知器 (它是一个二值分类器,即将一个实值向量作为输入,输出则是 0 或 1)。该神经元感知器有 3 个输入和 2 个输出,以及包含 5 个神经元的隐藏层。



在 MLP 中的所有神经元都差不多，每个神经元都有几个输入（连接前一层）神经元和输出（连接后一层）神经元，该神经元会将相同值传递给与之相连的多个输出神经元。每个神经元通过输入权重加上一个偏移项来计算输出值，并由所选择的激励函数（activation function）进行转换。



有三种广泛使用的激励函数：恒等函数、Sigmoid 函数和高斯函数，最常用的默认激励函数为 Sigmoid 函数。下图是 Sigmoid 函数的  $\alpha$  参数和  $\beta$  参数为 1 时的情形。



一个 ANN 训练网将一个特征向量作为输入，将该向量传递到隐藏层，然后通过权重和激励函数来计算结果，并将结果传递给下一层，直到最后传递给输出层才结束，输出层是指其神经元类别编号为神经网络的层数。

通过训练 ANN 算法来计算和学习每一层的权重、突触以及神经元。为了训练分类器，需创建两个数据矩阵，就像 SVM 训练时一样，但训练的标签不再是  $N \times 1$  的矩阵，其中  $N$  表示训练数据的行数，1 为列数，而是用数字标识符作为标签。创建一个  $N \times M$  矩阵，其中， $N$  表示训练样本数， $M$  是类标签（在本项目中，它由 10 个数字 + 20 个字母组成）。如果第  $i$  行的样本属于第  $j$  类，则该矩阵的  $(i,j)$  位置为 1。

1	0	0	...	0	0
1	0	0	...	0	0
0	1	0	...	0	0
0	1	0	...	0	0
0	1	0	...	0	0
...	...	...	...	...	...
0	0	0	...	0	1
0	0	0	...	0	1
0	0	0	...	0	1

函数 `OCR::train` 用来创建所有需要的矩阵并用训练数据、类标签矩阵、在隐藏层的神经元数量来训练一个识别系统。训练数据从 XML 文件中加载，这与 SVM 训练时一样。

为了初始化 ANN 类，必须在每层定义神经元数。例如，在本项目中，仅使用一个隐藏层，则需定义一个 1 行 3 列的矩阵。第一列为特征数，第二列为隐藏层的隐藏神经元数，第三列为样本的类数。

OpenCV 为 ANN 定义了一个 `CvANN_MLP` 类。通过 `create` 函数来初始化该类，初始化时需要指定以下参数的值：神经网络的层数、神经元数、激励函数、`alpha` 和 `beta`。

```
void OCR::train(Mat TrainData, Mat classes, int nlayers)
{
    Mat layerSizes(1,3,CV_32SC1);
    layerSizes.at<int>(0)= TrainData.cols;
    layerSizes.at<int>(1)= nlayers;
    layerSizes.at<int>(2)= numCharacters;
    ann.create(layerSizes, CvANN_MLP::SIGMOID_SYM, 1, 1); //ann is
    global class variable

    //Prepare trainClasses
    //Create a mat with n trained data by m classes
    Mat trainClasses;
    trainClasses.create( TrainData.rows, numCharacters, CV_32FC1 );
    for( int i = 0; i < trainClasses.rows; i++ )
    {
        for( int k = 0; k < trainClasses.cols; k++ )
        {
            //If class of data i is same than a k class
            if( k == classes.at<int>(i) )
                trainClasses.at<float>(i,k) = 1;
            else
                trainClasses.at<float>(i,k) = 0;
        }
    }

    Mat weights( 1, TrainData.rows, CV_32FC1, Scalar::all(1) );

    //Learn classifier
    ann.train( TrainData, trainClasses, weights );
    trained=true;
}
```

在训练完成后，可用 `OCR::classify` 函数来对得到的各种车牌特征进行分类：



```

int OCR::classify(Mat f)
{
    int result=-1;
    Mat output(1, numCharacters, CV_32FC1);
    ann.predict(f, output);
    Point maxLoc;
    double maxVal;
    minMaxLoc(output, 0, &maxVal, 0, &maxLoc);
    //We need to know where in output is the max val, the x (cols) is
    //the class.

    return maxLoc.x;
}

```

CvANN\_MLP 类使用 predict 函数来对特征向量分类。不像 SVM 的 classify 函数, ANN 的 predict 返回一行, 其大小为类的数量, 该向量的每个元素反映了输入样本属于每个类的概率。

为了得到最好的结果, 可使用 minMaxLoc 函数来得到最大和最小响应以及它们在矩阵中的位置。字符的类别为变量 maxLoc 的 x 的最大值来确定。



为了完成每个车牌的最终检测, 需整理车牌上的字符, 然后通过 Plate 类的 str() 函数返回一个字符串, 并在原图像上显示出来:

```

string licensePlate=plate.str();
rectangle(input_image, plate.position, Scalar(0,0,200));
putText(input_image, licensePlate, Point(plate.position.x, plate.
position.y), CV_FONT_HERSHEY_SIMPLEX, 1, Scalar(0,0,200),2);

```

#### 5.4.4 评价

本项目到此已经完成, 但当训练像 OCR 这样的机器学习算法时, 需要知道所使用的最佳特征和参数, 以及如何修正项目中出现的分类、识别和检测错误。

需要在不同情形和参数下评价当前开发的这个系统, 评价错误的产生, 获取让错误最小的参数。

本章用下面这些变量来评价这个 OCR 应用：低分辨率图像特征的大小和隐藏层的隐藏神经元数。

本项目所创建的 evalOCR.cpp 文件会用到由 trainOCR.cpp 程序所生成的 XML 训练数据。OCR.xml 文件对分辨率为  $5 \times 5$ 、 $10 \times 10$ 、 $15 \times 15$  和  $20 \times 20$  下的采样 (downsampled) 图像特征分别保存着相应的训练数据矩阵。

```
Mat classes;
Mat trainingData;
//Read file storage.
FileStorage fs;
fs.open("OCR.xml", FileStorage::READ);
fs[data] >> trainingData;
fs["classes"] >> classes;
```

评价程序会获取每个下采样特征矩阵，然后取 100 行用作训练，而其他行用作测试 ANN 算法，然后给出其误差。

在训练前，要测试每个随机样本并检测其输出是否正确。如果输出不正确，将增加错误计算变量的值，然后通过将错误计算的值除以样本数来进行评价。这意味着用随机数据训练，其错误率会在 0 和 1 之间。

```
float test(Mat samples, Mat classes)
{
    float errors=0;
    for(int i=0; i<samples.rows; i++)
    {
        int result= ocr.classify(samples.row(i));
        if(result!= classes.at<int>(i))
            errors++;
    }
    return errors/samples.rows;
}
```

对每类样本的大小，应用会以命令行方式输出错误率。为了得到一个好的评价，需要用不同的随机样本来训练应用，这会产生不同的测试误差值，然后将所有误差加到一起对其求平均。为了实现这一功能，可通过下面的 UNIX 脚本来自动完成：

```
#!/bin/bash
echo "#ITS \t 5 \t 10 \t 15 \t 20" > data.txt
folder=$(pwd)

for numNeurons in 10 20 30 40 50 60 70 80 90 100 120 150 200 500
do
    s5=0;
    s10=0;
    s15=0;
    s20=0;
```

```

for j in {1..100}
do
    echo $numNeurons $j
    a=$(($folder/build/evalOCR $numNeurons TrainingDataF5)
    s5=$(echo "scale=4; $s5+$a" | bc -q 2>/dev/null)

    a=$(($folder/build/evalOCR $numNeurons TrainingDataF10)
    s10=$(echo "scale=4; $s10+$a" | bc -q 2>/dev/null)

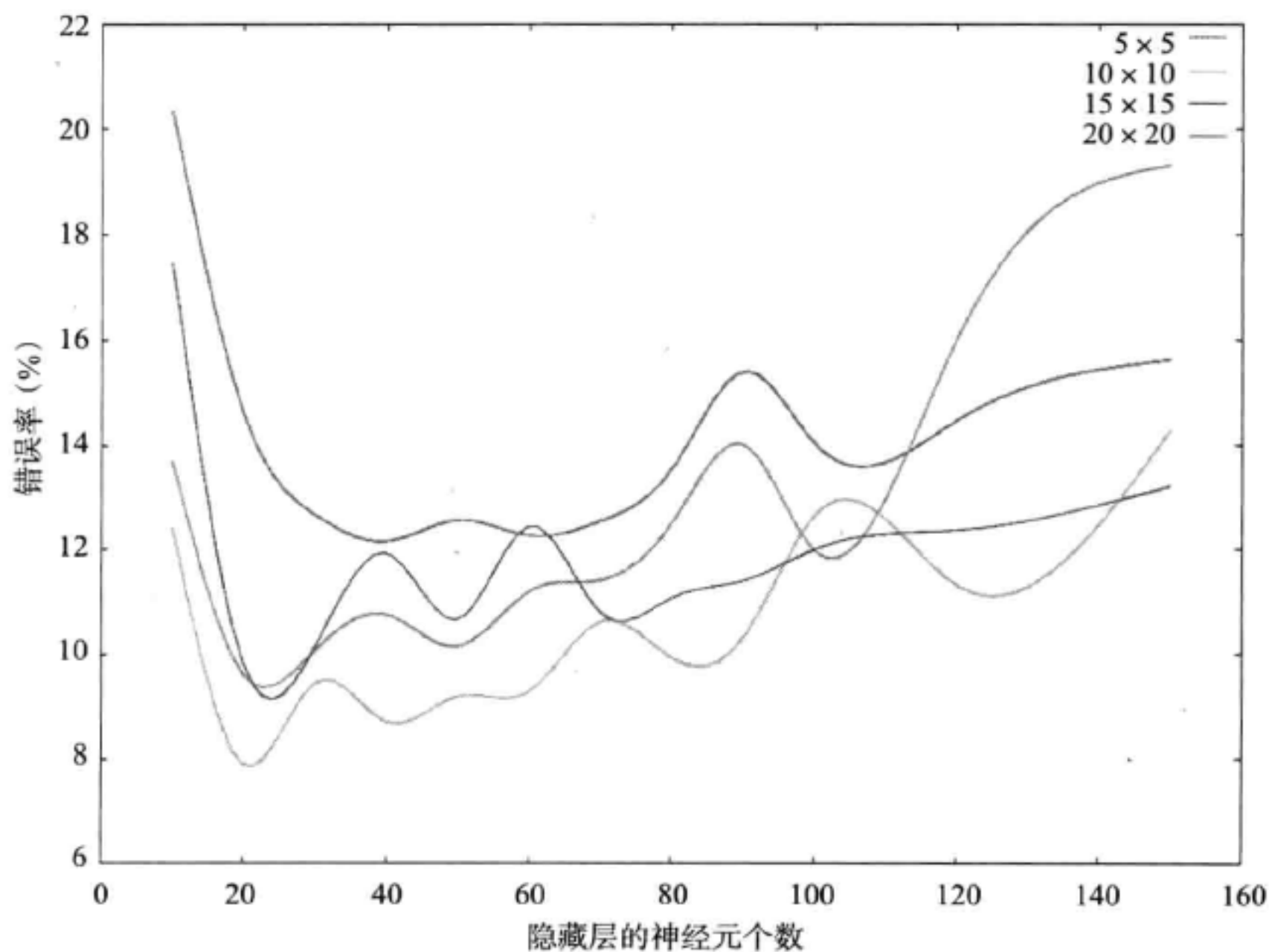
    a=$(($folder/build/evalOCR $numNeurons TrainingDataF15)
    s15=$(echo "scale=4; $s15+$a" | bc -q 2>/dev/null)

    a=$(($folder/build/evalOCR $numNeurons TrainingDataF20)
    s20=$(echo "scale=4; $s20+$a" | bc -q 2>/dev/null)
done

echo "$i \t $s5 \t $s10 \t $s15 \t $s20"
echo "$i \t $s5 \t $s10 \t $s15 \t $s20" >> data.txt
done

```

该脚本会将不同大小的特征图像和隐藏层的不同神经元数所产生的结果保存到 data.txt 文件中。可用 gnuplot 来对此文件中的数据进行绘图。结果如下图所示。



(附彩图)

可看到最低的错误率在 8% 以下，这个结果是在隐藏层的神经元个数为 20，从  $10 \times 10$  的图像块中提取字符特征而得到的。



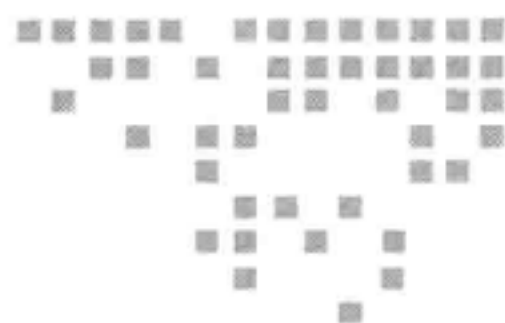
## 5.5 总结

本章介绍了自动车牌识别应用的工作原理以及实现这个应用的两个重要步骤：定位车牌和车牌字符识别。

第一步介绍了如何分割一幅图像，用简单的启发式方法和支持向量机从这些分割好的图像中进行二值分类，以得到车牌和非车牌。

第二步介绍如何用 Find Contours 算法把找到的车牌分割，提取每个字符的特征向量，并使用人工神经网络对其分类。

本章还介绍了如何通过训练随机样本来评价机器学习算法，并得出使用不同参数和特征对训练的影响。



## 非刚性人脸跟踪

非刚性人脸跟踪是对视频流的每帧中人脸特征进行准密 (quasi-dense) 集估计, 尽管现代的非刚性人脸跟踪方法借用了很多相关领域的思想, 包括: 计算机视觉、计算几何、机器学习、图像处理, 但这仍是一个困难的问题。本章所说的非刚性是指面部特征间的相对距离会随着面部表情和人群的不同而变化, 这也是它与人脸检测和跟踪的不同之处, 人脸检测与跟踪只是为了找到每帧视频中人脸位置, 而不是面部特征结构。非刚性人脸跟踪成为研究热点已超过 20 年, 但直到最近出现了多种鲁棒性较强的方法, 并随着处理器速度的提高, 使得构建基于非刚性人脸跟踪的商业应用成为可能。

但商用级人脸跟踪很复杂, 即便是对经验丰富的计算机视觉科学家也是一个挑战。在本章, 读者将看到一个在特定环境下表现很不错的人脸跟踪系统, 该系统会用到一些数学工具和大量 OpenCV 的功能来进行设计, 主要涉及的知识有: 线性代数、图像处理和可视化。本章假定事先知道被跟踪的人, 而且训练数据按图像和有显著标注的形式保存。本章介绍的技术很有用, 但也只是开始, 它为进一步实现更复杂的人脸跟踪系统奠定基础。

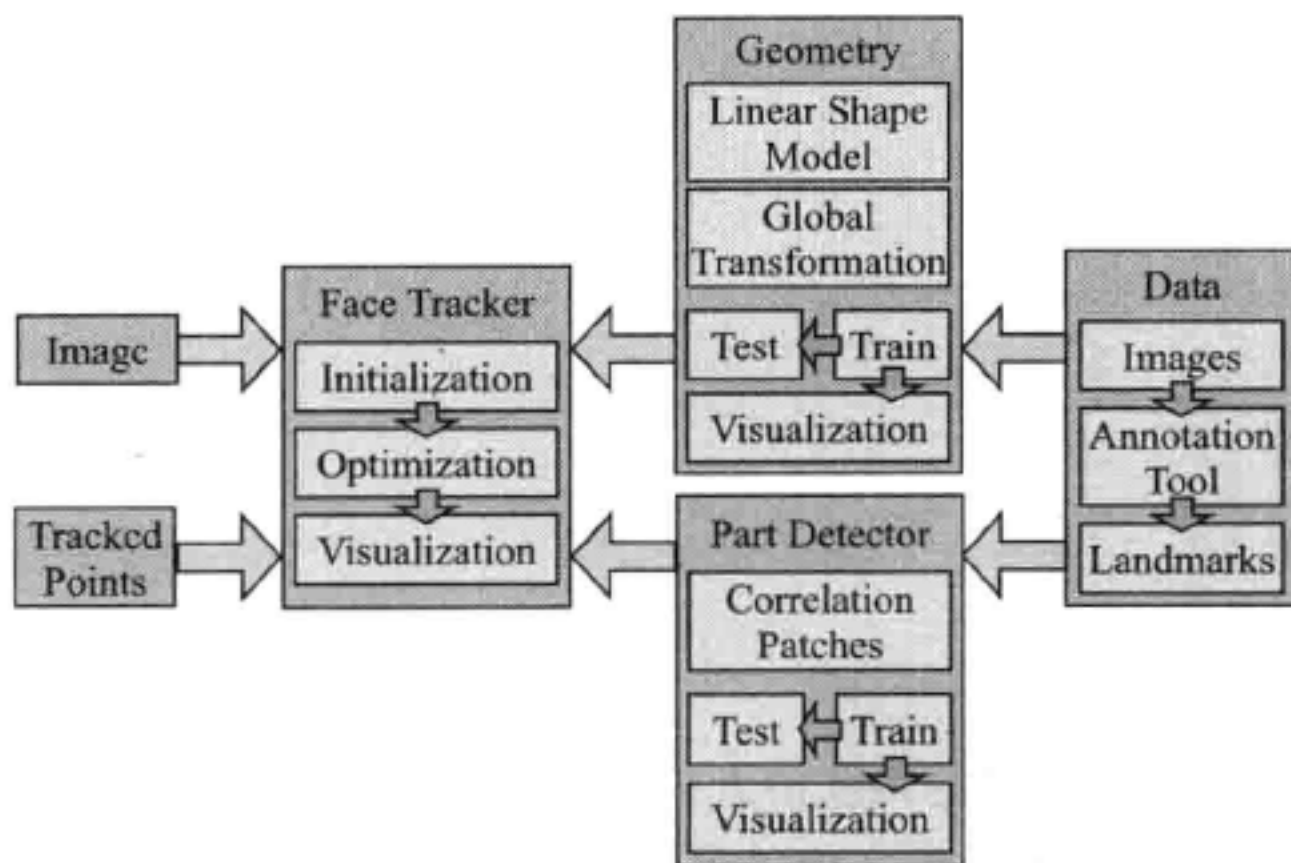
本章涉及的内容如下。

- 概述: 这一节将简要回顾人脸跟踪的历史;
- 实用工具: 这节将简要介绍本章中所使用的常用结构和约定, 包含面向对象设计、数据存储与表示, 以及数据收集和标注的工具;
- 几何约束: 这节将介绍如何从训练数据学习面部几何及其变化情况, 也介绍在跟踪过程中如何利用它们来约束输出, 这包括对面部进行线性的形状模型建模, 以及如何在模型的表示中引入将全局变换;
- 面部特征检测器: 这节将介绍如何学习面部特征的外观 (appearance), 以便在被跟踪

的人脸图像中检测它们；

- 人脸检测与初始化：这节介绍如何使用人脸检测来初始化跟踪程序；
- 人脸跟踪：这节通过图像对齐处理将前面介绍的所有内容联系起来，用于跟踪系统。也会讨论这种情形下如何让该系统运行得更好。

下面的框图说明系统各个部分之间的关系：



**注意：**本章采用的所有方法都遵循数据驱动范式，即所使用的模型都是从数据中学到而不是人为地按某种规则设计的。因此，系统的每个部件由两部分组成：训练和测试。训练由数据生成模型，测试是用新的未知数据来检测这些模型。

## 6.1 概述

非刚性人脸跟踪早在 90 年代中期就随着 Cootes 和 Taylor 提出的主动形状模型（Active Shape Models, ASM）而开始流行。从那以后，许多研究都致力于通过改进原始的 ASM 方法来解决一般人脸跟踪的难题。第一个对 ASM 扩展的里程碑是主动外观模型（Active Appearance Models, AAM），该模型也是由 Cootes 和 Taylor 在 2001 年提出的。后来该方法被 Baker 和他的同事在 00 年代中期通过图像扭曲的基本原理将其形式化（formalized）。另一方面的工作由 Blanz 和 Vetter 沿着 3D 形变模型（3D Morphable Model, 3DMM）开展。该模型与 AAM 一样，这不仅表现在对图像纹理建模时不像 ASM 那样是沿着对象边界来描绘轮廓，而且进一步用从激光人脸扫描得到的高密度三维数据来表示这个模型。从 20 世纪中后期开始，人脸跟踪研究的焦点从如何参数化人脸转到如何建立和优化跟踪目标函数算法。应用机器学习社区的各种技术获得了不同程度的成功。进入本世纪以来，研究焦点再次发生变化，这次是朝着将参数与保证目标函数有全局解相结合的设计策略发展。



尽管对人脸进行了持续艰苦的研究，但也只有很少的商业应用使用它。虽然对许多常用方法都有大量可自由下载的源代码，但人脸跟踪爱好者吸收这些代码仍滞后。尽管如此，过去两年，公共领域对人脸跟踪的潜在用途又感兴趣了，商业级产品开始出现了。

## 6.2 实用工具

在开始讨论人脸跟踪的复杂性之前，首先介绍一些基本任务和对人脸跟踪方法的共同约定。下面这节将讨论这些问题。如果读者没有兴趣，可在第一次阅读时跳过这部分，而直接阅读 6.3 节。

### 6.2.1 面向对象设计

与人脸检测和人脸识别一样，人脸跟踪也由两部分组成：数据和算法。算法通过预先存储（即离线）的数据来训练模型，然后对新来的（即在线）数据执行某类操作。因此，采用面向对象设计是不错的选择，因为面向对象设计可将算法同它所依赖的数据结合在一起。

在 OpenCV v2.x 版本中，可方便引入 XML/ YAML 文件存储类，对算法来讲，会大大简化组织离线数据任务。为了利用此功能，本章引入的所有类都将实现读取和写入序列化功能。下面通过一个假想类来展示该功能：

```
#include <opencv2/opencv.hpp>
using namespace cv;
class foo{
public:
    Mat a;
    type_b b;
    void write(FileStorage &fs) const{
        assert(fs.isOpened());
        fs << "{" << "a" << a << "b" << b << "}";
    }
    void read(const FileNode& node){
        assert(node.type() == FileNode::MAP);
        node["a"] >> a; node["b"] >> b;
    }
};
```

在上面的代码中，Mat 是 OpenCV 矩阵类，type\_b 是一个（假想的）用户定义类，在该类中还定义了一个序列化函数。可对 I/O 函数 read 和 write 实现序列化。FileStorage 类支持两种能被序列化的数据结构类型。为了简单起见，本章所有类将仅采用映射，其中每个用于存储的变量都会创建一个 FileNode::MAP 类型的 FileNode 对象。这需要分配给变量中的每个元素唯一键，虽然对这个键值的选择是任意的，但为了保持一致性，将变量名作为标签。例如，上面的这段代码，read 和 write 函数就用到了这种简单的形式，并借着流运算

符 (<< 和 >>) 从 FileStorage 对象中插入和提取数据。大多数 OpenCV 类都实现了 read 和 write 函数，用这些函数来存储其包含的数据很容易。

除了定义序列化函数以外，为了让 FileStorage 类的序列化能正常工作，还需要另外定义两个函数：

```
void write(FileStorage& fs, const string&, const foo& x){
    x.write(fs);
}
void read(const FileNode& node, foo& x, const foo& default){
    if(node.empty())x = d; else x.read(node);
}
```

由于这两个函数的功能在本节介绍的所有类中都一样，可将其模板化并定义在 ft.hpp 头文件中，这个头文件可在本章源代码中找到。最后，为了让保存和加载采用了序列化的用户定义类变得容易，在该头文件中采用了模板化函数，具体定义如下：

```
template <class T>
T load_ft(const char* fname){
    T x; FileStorage f(fname, FileStorage::READ);
    f["ft object"] >> x; f.release(); return x;
}
template<class T>
void save_ft(const char* fname, const T& x){
    FileStorage f(fname, FileStorage::WRITE);
    f << "ft object" << x; f.release();
}
```

注意，与对象关联的标签都一样（即都是 ft object）。用这些函数定义、保存、加载对象数据是一件愉快的事情。可用下面的例子来说明：

```
#include "opencv_hotshots/ft/ft.hpp"
#include "foo.hpp"
int main(){
    ...
    foo A; save_ft<foo>("foo.xml", A);
    ...
    foo B = load_ft<foo>("foo.xml");
    ...
}
```

注意，.xml 是 XML 格式的数据文件。对任何其他扩展名，都默认为 YAML 格式（可读性更好）。

## 6.2.2 数据收集：图像和视频标注

现代人脸跟踪技术几乎完全是数据驱动，即用来检测图像中面部特征位置的算法依靠面部特征的外观模型和几何依赖性，该依赖性来自样本集中人脸间的相对位置。样本集越大，算法就更具有鲁棒性，因为人脸所表现出的变化范围就更清楚。因此，构建人脸跟踪

算法的第一步是创建用于进行图像/视频的标注工具，用户可用此工具来指定在每个样本图像中想要的面部特征位置。

### 1. 训练数据类型

训练人脸跟踪算法的数据一般由以下四部分构成。

- ❑ 图像：这部分是包含整个人脸图像（图像或视频帧）的集合。为了达到最理想的效果，此集合应指明具体的条件类型（即，身份、光线到摄像机的距离、拍摄装置等），这些条件在后面的人脸跟踪系统中会涉及。此集合展现了目标应用所期望的人脸头部姿态以及面部表情范围，这也很关键。
- ❑ 标注：这部分采用手工方法标注每幅图像中被跟踪的面部特征的相应位置。通常面部特征越多，跟踪系统鲁棒性越高，因为跟踪算法会使用它们的信息来使其相互补充。一般来说，跟踪算法的计算代价会随着面部特征数的增加而呈线性增长。
- ❑ 对称性索引：这部分对定义了双边对称特征的面部特征点都保留了一个编号，以便能用来镜像训练图像，可有效地让训练集大小增加一倍，还可沿着 y 轴对称化数据。
- ❑ 连通性索引：这部分是一组标注的索引对，它们定义了面部特征的语义解释。连通性对可视化跟踪结果很有用。

这四个组件的可视化情形显示在下图中，从左到右依次是原始图像，脸部特征标注，颜色编码的双边对称点，镜像图像与相应的标注，面部特征的连通性。



为了方便管理这种数据，需实现具有读写功能的类，这将会非常有用。在 OpenCV 的 ml 模块中，CvMLData 类可处理常用的机器学习数据，但它缺少处理人脸跟踪数据的功能。因此，本章将使用在 ft\_data.hpp 头文件中定义的 ft\_data 类，它是按人脸跟踪数据的特性专门设计的。所有元素都定义成类的公有成员变量，如下所示：

```
class ft_data{
public:
    vector<int> symmetry;
    vector<Vec2i> connections;
    vector<string> imnames;
    vector<vector<Point2f> > points;
    ...
}
```

Vec2i 和 Point2f 类型都是 OpenCV 类，它们分别表示两个整型向量和二维浮点坐标。symmetry 向量的维数与在人脸图像上的特征点（由用户定义）一样。向量 connections 的每



个分量都定义一对连通的面部特征，这些分量的索引是从零开始的。由于训练集不是直接存储图像，可能非常大，因此用成员变量 `imnames` 来存储每个图像文件名（注意，为了保证有效性，这需要图像文件有相同的相对路径）。最后，对每个训练图像，面部特征位置的集合存放在基于浮点坐标的 `point` 向量中，它是该类的成员变量。

`ft_data` 类实现了许多访问数据的有用方法。为了访问数据集的图像，可用 `get_image` 函数加载图像。使用该函数需指定加载图像的索引 `idx`，以及是否将图像以 `y` 轴做镜像。该函数的实现如下：

```
Mat
ft_data::get_image(
const int idx,    //index of image to load from file
const int flag){ //0=gray,1=gray+flip,2=rgb,3=rgb+flip
    if((idx < 0) || (idx >= (int)imnames.size()))return Mat();
    Mat img,im;
    if(flag < 2)img = imread(imnames[idx],0);
    else          img = imread(imnames[idx],1);
    if(flag % 2 != 0)flip(img,im,1); else im = img;
    return im;
}
```

(0,1) 标记会传递给 OpenCV 的 `imread` 函数，用来指定是否将图像加载作为 3 通道彩色图像或单通道灰度图像。参数 `flag` 的值传给 OpenCV 的 `flip` 函数，用来指定图像是否以 `y` 轴做镜像。

为了通过指定的索引来得到相应图像的一个点集，可使用 `get_points` 函数通过镜像索引来到一个基于浮点的坐标向量，具体代码如下：

```
vector<Point2f>
ft_data::get_points(
const int idx,    //index of image corresponding to points
const bool flipped){ //is the image flipped around the y-axis?
    if((idx < 0) || (idx >= (int)imnames.size()))
        return vector<Point2f>();
    vector<Point2f> p = points[idx];
    if(flipped){
        Mat im = this->get_image(idx,0); int n = p.size();
        vector<Point2f> q(n);
        for(int i = 0; i < n; i++){
            q[i].x = im.cols-1-p[symmetry[i]].x;
            q[i].y = p[symmetry[i]].y;
        }return q;
    }else return p;
}
```

注意，当镜像标志被指定时，这个函数会调用 `get_image` 函数。这需要指定图像宽度，以便正确镜像面部特征坐标。可设计更有效的方法，即将图像宽度作为一个变量进行传递。代码的最后部分展现了成员变量 `symmetry` 的作用。指定索引的镜像特征位置的 `x` 坐标可简

单地由指定索引在变量 `symmetry` 中的位置所对应特征的 `x` 坐标反转加上一个偏移量后得到, 其 `y` 坐标等于指定索引在变量 `symmetry` 中的位置所对应特征的 `y` 坐标。

若指定的索引超过了数据集范围, `get_image` 和 `get_points` 函数都会返回空结构。也可能不对集合中所有图像进行标注。人脸跟踪算法可用于处理丢失的数据。然而, 这类算法的实现相当困难, 已超过本章的范围。`ft_data` 类实现了一个函数, 该函数删除集合中没有进行相应标注的样本, 具体实现如下:

```
void
ft_data::rm_incomplete_samples(){
    int n = points[0].size(), N = points.size();
    for(int i = 1; i < N; i++) n = max(n, int(points[i].size()));
    for(int i = 0; i < int(points.size()); i++){
        if(int(points[i].size()) != n){
            points.erase(points.begin()+i);
            imnames.erase(imnames.begin()+i); i--;
        }else{
            int j = 0;
            for(; j < n; j++){
                if((points[i][j].x <= 0) ||
                    (points[i][j].y <= 0))break;
            }
            if(j < n){
                points.erase(points.begin()+i);
                imnames.erase(imnames.begin()+i); i--;
            }
        }
    }
}
```

标注数最多的样本实例被称为标准样本。如果样本标注点的数量小于样本集中最大样本标注点, 这些样本会通过 `erase` 函数从样本集合中删除。另外, 若点的 (`x,y`) 坐标都小于 1, 则可认为它在相应的图像中已经不存在 (可能是由于遮挡, 能见度差或模糊造成的)。

`ft_data` 类实现了函数 `read` 和 `write` 的序列化, 这样就可很方便地存储和加载该类。例如, 可很简单地通过下面的方法来保存数据集:

```
ft_data D; //instantiate data structure
... //populate data
save_ft<ft_data>("mydata.xml",D); //save data
```

为对数据集进行可视化操作, `ft_data` 实现了许多用于绘图的函数。这些函数的使用说明可参考 `visualize_annotations.cpp` 文件。这个简单的应用程序可通过命令行加载存储在指定文件中的标注数据, 删除不完整的样本, 显示训练图像的相应标注、对称点、面部特征的连通性。OpenCV 的 `highgui` 模块的几个显著特性将在这里使用。虽然 OpenCV 的 `highgui` 模块相当不成熟, 并且不适合于复杂的用户界面, 但它的功能对于装载并进行可视化数据, 以及查看计算机视觉算法的结果都非常有用。这也许是 OpenCV 与其他计算机视

觉库相比的显著特色之一。

## 2. 标注工具

为了使生成的标注能被本章中的代码使用，可在 `annotate.cpp` 文件中找到一个基本的标注工具。该工具将一个视频流作为输入，这个视频流可以来自文件或相机。使用该工具的过程有如下五个步骤。

1) 捕获图像：第一步是将图像流显示在屏幕上，用户按下 S 键就可选择图像进行标注。用来标注的最好特征集是最大限度地涵盖跟踪系统将来需要跟踪的人脸行为。

2) 标注第一幅图：第二步首先会将上一步中第一幅图呈现给用户，然后用户会在这幅图像中选择需要跟踪的面部特征位置。

3) 标注连通性：在这一步中，为了更好地可视化一个形状，需要定义点之间的连通结构。因此，要将上一步的图像呈现给用户，用户只需选择将两组点连接起来，以建立人脸模型的连通性结构。

4) 标注对称性：这一步仍然使用上一步的图像，用户需选出左右对称的点。

5) 标注剩下的图像：这是最后一步，这步所完成的工作是重复第 2 步至第 4 步，只是用户所标注的图像是图像集中另外的图像。

感兴趣的读者可从提高可用性的角度来改进这个工具，甚至可将增量学习的过程集成进来，即每个新加进来的图像被标注，随即用它初始化点，以减少标注的负担，之后便更新跟踪模型。

尽管一些公开的数据集也可在本章使用（下面一节就会介绍这样的例子），但用标注工具可构建特定的人脸跟踪模型，这种模型在性能上常是基础公共数据的模型无法比拟的。

## 3. 准备标注数据 (MUCT 数据集)

开发人脸跟踪系统的一个困难的地方在于，手工标注大图像集且每个图像有很多点需要标注时，烦琐且容易出错。为了让本章的标注工作变得轻松一些，可利用公开的 MUCT 数据集，可在网址 <http://www.milbo.org/muct> 下载。

这个数据集由 3755 张人脸图像构成，每张人脸有 76 个点作为标记 (landmark)。数据集的图像是在不同光照条件和头部姿势下拍摄的人，它们来自不同年龄和种族。

为了使本章代码能使用 MUCT 数据集，可执行下面的步骤。

1) 下载图像集：在这一步，先依次下载文件 `muct-a-jpg-v1.tar.gz` 至 `muct-ejpg-v1.tar.gz`，然后解压得到数据集中的所有图像。这需要建立新的文件夹，所有图像都保存在里面。

2) 下载标注文件：这一步下载包含标注的文件 `muct-landmarks-v1.tar.gz`。将该文件保存在上面那个文件夹中，并解压此文件。

3) 使用标注工具定义连通性和对称性：这一步会在命令行执行命令：`./annotate -m $mdir -d $odir`，其中 `$mdir` 表示 MUCT 数据集被保存的文件夹，`$odir` 表示 `annotations.yaml` 文件



所在的文件夹，该文件含有 `ft_data` 对象数据，这些数据稍后将会被保存到此文件中。



**注意：**使用 MUCT 数据集是为了能早点开始介绍本章所描述的人脸跟踪代码。

## 6.3 几何约束

在人脸跟踪系统中，几何形状是指在人脸图像上预先定义的一组点的空间结构，这组点与真实人脸某些几何形状（如眼角、鼻尖和眉毛边缘）保持一一对应关系。这些点的具体选择与应用有关，一些应用需要超过 100 个点的稠密点集，而另一些仅需要选择稀疏点集。但人脸跟踪算法的鲁棒性一般会随着点数的增加而提高，因为它们各自的度量可通过相对空间依赖性来增强。例如，知道一个眼角的位置就能很好地找到鼻子的位置。但这种情形下鲁棒性也不一定会完全随点数的增加而提高，鲁棒性会在 100 个点后趋于稳定。但增加描述人脸的点集的大小，其计算复杂度也会线性地增加。因此，对计算量有严格要求的应用使用较少的点性能会更好。

还有一种情形，在在线情形（online setting）下，较快的跟踪系统经常会带来更精确的跟踪。因为帧丢失，帧之间的运动变化会增加，查找每帧中人脸结构的优化算法必须搜索更大可能的特征点结构空间。当帧之间的位移变化过大时，这一搜索过程往往会失败。总之，为了得到优化性能，有如何最好地设计面部特征点选择的一般准则，但这样的选择还是要根据具体应用领域而定。

面部几何参数化通常由两个因素组成：全局（刚性）变换和局部的（非刚性）形变。全局变换考虑人脸在图像中的整体位置，它经常允许人脸随意变化（即人脸可出现在图像的任何位置）。这包括人脸在图像中的  $(x,y)$  位置，平面内头部的旋转，脸在图像中的大小。另一方面，局部形变考虑不同人面部形状以及同一个人面部表情的差异。与全局变换相比较，这些局部变形往往更受限于面部特征的高度结构化形状。全局变换是二维坐标的常规处理，可用于任意类型的对象，而局部形变针对的是具体对象，需要从训练集中学习。

本节将介绍面部结构的几何模型，即，形状模型。本章的应用程序要能捕获一个人的表情变换和不同人之间的面部形状差异，或同时捕获这两种情形。这个模型是在 `shape_model` 类中被实现的，该类的定义及实现可分别在 `shape_model.hpp` 和 `shape_model.cpp` 文件中找到。下面的代码是 `shape_model` 类在头文件的一部分，展示了该类的主要功能：

```
class shape_model{ //2d linear shape model
public:
    Mat p; //parameter vector (kx1) CV_32F
    Mat V; //linear subspace (2nxk) CV_32F
    Mat e; //parameter variance (kx1) CV_32F
    Mat C; //connectivity (cx2) CV_32S
```

```

...
void calc_params(
    const vector<Point2f> &pts,    //points to compute parameters
    const Mat &weight = Mat(),    //weight/point (nx1) CV_32F
    const float c_factor = 3.0); //clamping factor
...
vector<Point2f>          //shape described by parameters
calc_shape();
...
void train(
    const vector<vector<Point2f> > &p, //N-example shapes
    const vector<Vec2i> &con = vector<Vec2i>(), //connectivity
    const float frac = 0.95, //fraction of variation to retain
    const int kmax = 10);    //maximum number of modes to retain
...
}

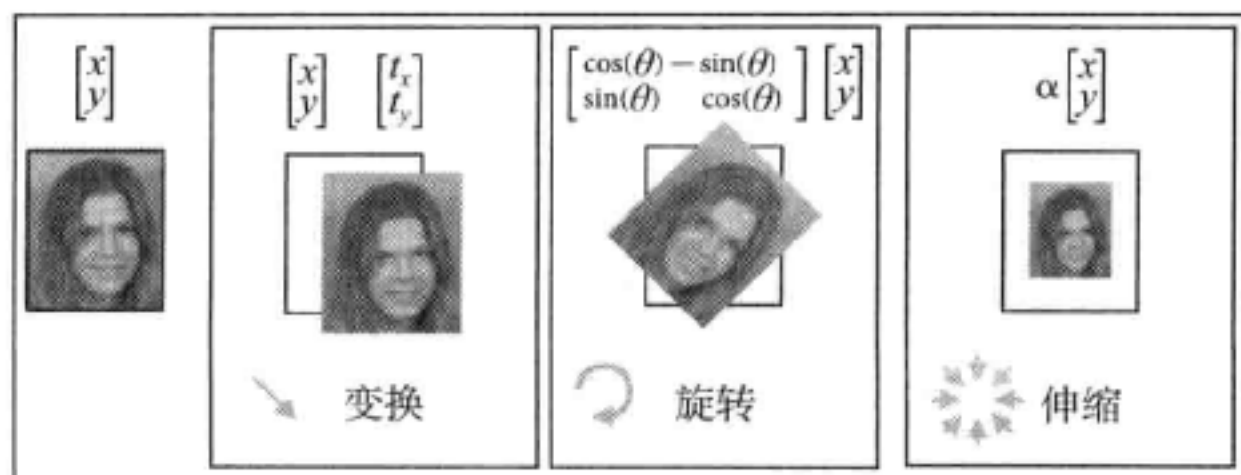
```

表示人脸形状变换的模型被编码在子空间矩阵  $V$  和方差向量  $e$  中。参数向量  $p$  保存着一个关于该模型形状的编码。连通矩阵  $C$  也存储在这个类中，因为它仅用于可视化面部形状的实例。在这个类中，重点需要关注三个函数：`calc_params`、`calc_shape` 和 `train`。`calc_params` 函数将一个点集投影到貌似脸形（face shape）的空间中。它对被投影的每个点有选择地给出单独的置信权重。`calc_shape` 函数通过解码用在人脸模型参数向量  $p$ （通过  $V$  和  $e$  编码）来生成点集。`train` 函数从脸形数据集中学习编码模型，这些脸形都有相同的点数。`frac` 和 `kmax` 是训练过程的参数，它们将会根据数据来设置。

该类的这些函数将在本章后面详细阐述。在下一节首先介绍 Procrustes 分析，它是严格注册（rigidly registering）一个点集的方法，然后介绍用线性模型来表示局部形变。分别在 `train_shape_model.cpp` 和 `visualize_shape_model.cpp` 文件中来训练和可视化形状模型。它们的使用方法将在本节结束前介绍。

### 6.3.1 Procrustes 分析

为了建立脸形的形变模型，首先必须删除原始标注数据中适用于全局刚性运行的部分。当在二维情形建立几何模型时，通常用相似变换来表示刚性运动，相似变换包括伸缩、平面内旋转、变换。下图展示了相似变换所能得到的运动类型。从点集删除全局刚性运动的过程称为 Procrustes 分析。



在数学上, Procrustes 分析的目的是要同时找到一个标准形状和每个数据实例的相似变换, 并让这些数据实例与标准形状对齐。这里用最小平方距离来度量每个经相似变换后的形状与标准形状对齐得如何。为了实现 Procrustes 分析, 采用一个迭代来完成, 其在 `shape_model` 的具体实现如下:

```
#define fl at<float>
Mat shape_model::procrustes(
    const Mat &X,          //interleaved raw shape data as columns
    const int itol,        //maximum number of iterations to try
    const float ftol)      //convergence tolerance
{
    int N = X.cols, n = X.rows/2; Mat Co, P = X.clone(); //copy
    for(int i = 0; i < N; i++){
        Mat p = P.col(i);          //i'th shape
        float mx = 0, my = 0;      //compute centre of mass...
        for(int j = 0; j < n; j++){ //for x and y separately
            mx += p.fl(2*j); my += p.fl(2*j+1);
        }
        mx /= n; my /= n;
        for(int j = 0; j < n; j++){ //remove center of mass
            p.fl(2*j) -= mx; p.fl(2*j+1) -= my;
        }
    }
    for(int iter = 0; iter < itol; iter++){
        Mat C = P*Mat::ones(N,1,CV_32F)/N; //compute normalized...
        normalize(C,C);                      //canonical shape
        if(iter > 0){if(norm(C,Co) < ftol)break;} //converged?
        Co = C.clone();                     //remember current estimate
        for(int i = 0; i < N; i++){
            Mat R = this->rot_scale_align(P.col(i),C);
            for(int j = 0; j < n; j++){ //apply similarity transform
                float x = P.fl(2*j,i), y = P.fl(2*j+1,i);
                P.fl(2*j,i) = R.fl(0,0)*x + R.fl(0,1)*y;
                P.fl(2*j+1,i) = R.fl(1,0)*x + R.fl(1,1)*y;
            }
        }
    }
    return P; //returned procrustes aligned shapes
}
```

算法从去掉每个形状实例的中心区域开始, 随后执行一个迭代, 在这个迭代过程中如下的操作将交替执行, 就像处理所有形状的归一化平均一样来计算标准形状; 然后旋转和缩放每个形状使之能与标准形状有最佳匹配。必须归一化所估计的标准形状, 这样做可解决尺度问题, 并防止将所有形状收缩为零。尺度大小的选择是任意的, 这里标准形状向量 `C` 所采用的长度为 1.0。因为 OpenCV 的 `normalize` 函数默认也是这样。计算平面内的旋转和伸缩, 使得每个形状的实例能与当前估计的标准形状有最好的对齐, 这个过程是通过下面的 `rot_scale_align` 函数来实现的:



```

Mat shape_model::rot_scale_align(
const Mat &src, //[x1;y1;...;xn;yn] vector of source shape
const Mat &dst) //destination shape
{
    //construct linear system
    int n = src.rows/2; float a=0,b=0,d=0;
    for(int i = 0; i < n; i++){
        d+= src.fl(2*i)*src.fl(2*i )+src.fl(2*i+1)*src.fl(2*i+1);
        a+= src.fl(2*i)*dst.fl(2*i )+src.fl(2*i+1)*dst.fl(2*i+1);
        b+= src.fl(2*i)*dst.fl(2*i+1)-src.fl(2*i+1)*dst.fl(2*i );
    }
    a /= d; b /= d; //solve linear system
    return (Mat_<float>(2,2) << a,-b,b,a);
}

```

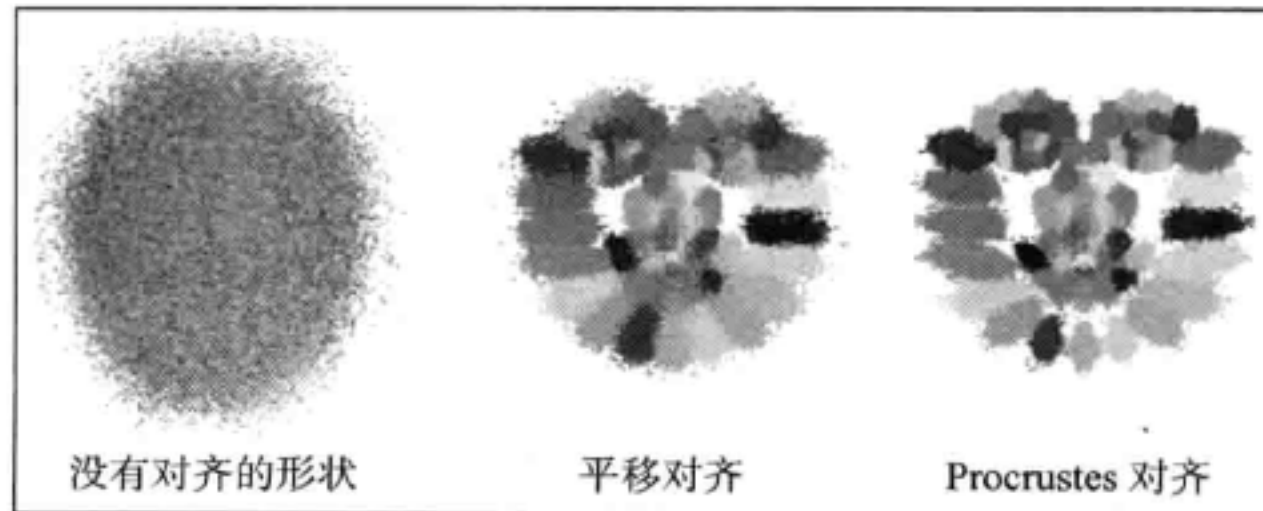
此函数会计算旋转形状与标准形状之间的最小二乘。数学表示如下：

$$\min_{a,b} \sum_{i=1}^n \left\| \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} - \begin{bmatrix} c_x \\ c_y \end{bmatrix} \right\|^2 \rightarrow \begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{\sum_i (x_i^2 + y_i^2)} \sum_{i=1}^n \begin{bmatrix} x_i c_x + y_i c_y \\ x_i c_y - y_i c_x \end{bmatrix}$$

最小二乘问题有闭解（close-form solution），其闭解在上面这个等式的右边。注意，只需要对变量（a,b）求解即可，不要对伸缩和平面内旋转求解，在二维平面中，伸缩与旋转的矩阵非线性相关，此过程比较复杂。这两个变量与伸缩旋转矩阵的关系如下：

$$\begin{bmatrix} a & -b \\ b & a \end{bmatrix} = \begin{bmatrix} k \cos(\theta) & -k \sin(\theta) \\ k \sin(\theta) & k \cos(\theta) \end{bmatrix}$$

对原始标注形状进行 Procrustes 分析后，其可视化效果如下图所示。每个面部特征都用单独的一种颜色显示，在变换归一化后，结构变得很明显，其中面部特征的位置聚集在这些平均特征的周围。在迭代完成伸缩和旋转的归一化之后，同一特征之间聚集得更加紧凑，其分布变更能反映出面部形变引起的变化。最后这一点很重要，因为下一节将由此对这些形变建模。Procrustes 分析的作用可认为是对原始数据的预处理操作，对原始数据执行这样的操作是让有较好人脸局部的形变模型参与训练。



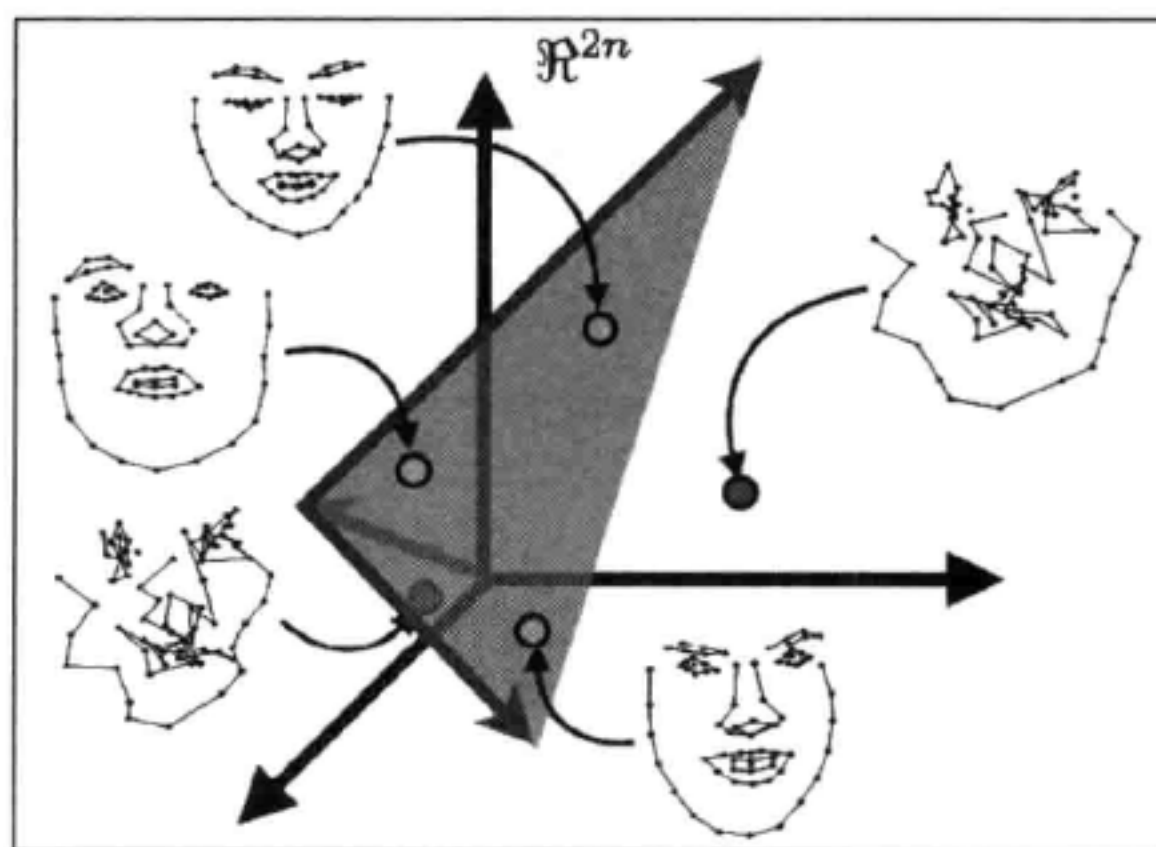
（附彩图）

### 6.3.2 线性形状模型

面部形变模型的目标是找到一组少量参数来表示多个人以及不同表情的脸型是如何变

化的。可使用许多复杂度不同的方法来实现这一目标。这些方法中最简单的一种是使用面部几何的线性表示。虽然简单,但它能精确捕获面部形变。特别是当数据集中的人脸大多为正面姿态时更是如此。它还有一个优点是获得其代表性参数时非常简单且操作的代价很低。与之相对应的是面部几何的非线性表示,将其用在约束跟踪期间的搜索过程时非常有用。

面部几何线性模型的主要思想如下图所示。由  $N$  个面部特征构成的一幅人脸图像可看作是  $2N$  维空间的一个点。线性模型的目标就是找到一个低维超平面嵌入  $2N$  维空间,所有人脸的点(即下图中那些绿色的点)都在这个  $2N$  维空间中。这个超平面仅由整个  $2N$  维空间的一个子集生成,因此通称它为子空间。子空间维数越低,对人脸的表示就简洁,跟踪过程中的约束就越强,这就会得到鲁棒性更好的跟踪。但是,应注意所选择子空间的维数,即要使其有足够的生成能力来生成所有人脸的空间,但不能过多,使非脸型的点(即下图中红色的点)也在生成的空间中。应该注意在模型化一个人的面部数据时,捕获人脸变化的子空间会比模型化多个人的面部数据更小。这也就是为什么针对具体某个人的人脸跟踪器的性能要比针对多人的好的原因之一。

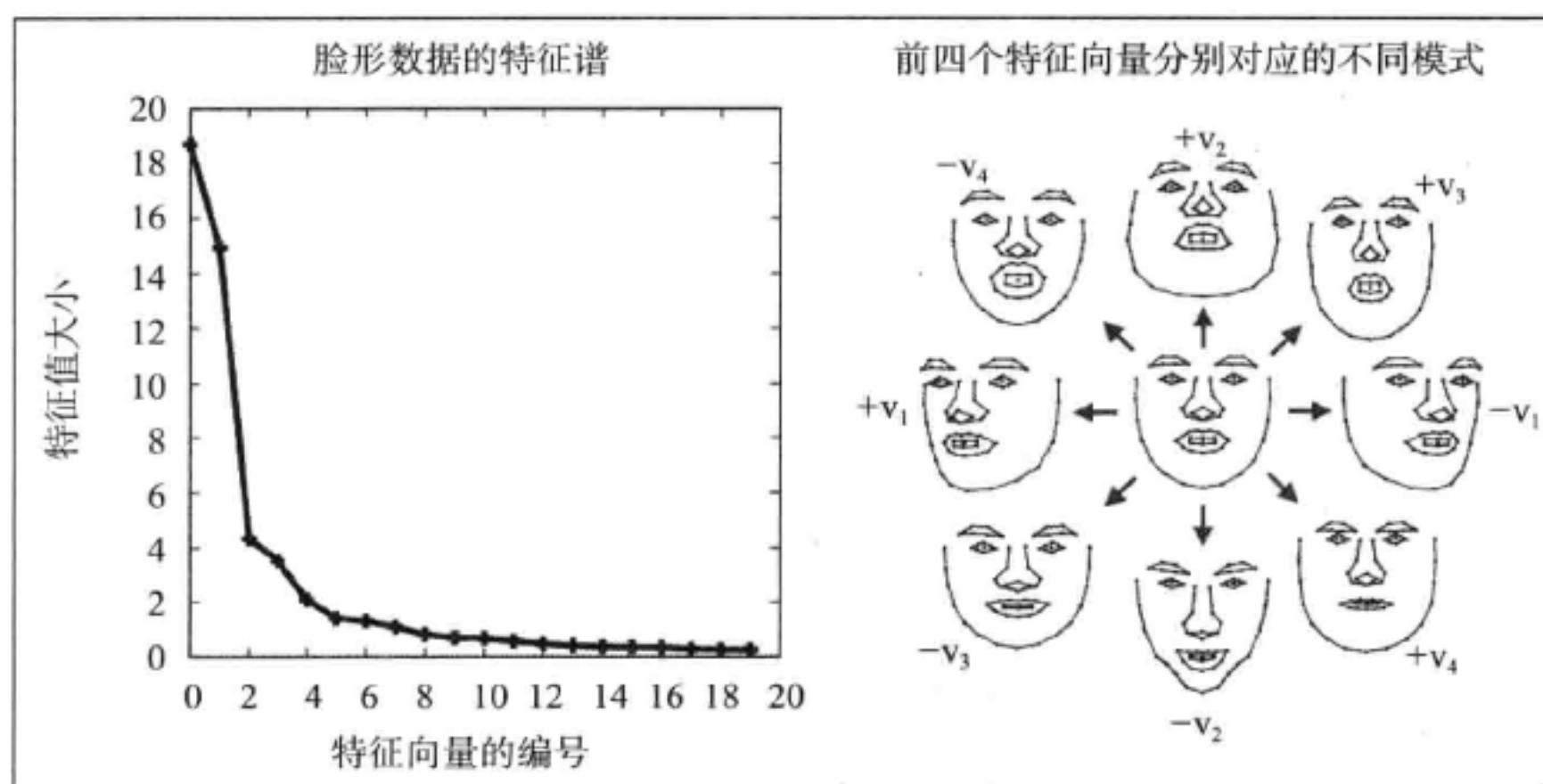



(附彩图)

查找生成数据集的最佳低维子空间的过程叫做主成分分析 (Principal Component Analysis, PCA)。OpenCV 有一个类可用来计算 PCA。但需要预先指定所获得子空间的维数。因为预先得到这个维数很困难,通常用启发式方法来得到,即按所选择的特征向量对应的特征值占整个特征值的比例来确定该维数。在 `shape_model::train` 函数中,PCA 的实现如下:

```
SVD svd(dY*dY.t());
int m = min(min(kmax,N-1),n-1);
float vsum = 0; for(int i = 0; i < m; i++) vsum += svd.w.fl(i);
float v = 0; int k = 0;
for(k = 0; k < m; k++){
    v += svd.w.fl(k); if(v/vsum >= frac){k++; break;}
}
if(k > m) k = m;
Mat D = svd.u(Rect(0,0,k,2*n));
```

变量  $dY$  的每一列表示减掉均值 (mean-subtracted) 后用 Procrustes 对齐的脸形。因此, 奇异值分解 (singular value decomposition, SVD) 能有效地用于这种人脸数据的协方差矩阵 (即,  $dY.t() \cdot dY$ ) 中,  $w$  是 OpenCV 的 SVD 类的成员变量, 它用来存放数据主要变化方向的方差, 其存储顺序是由大到小。选择子空间维数的常用方法是在  $w$  中找一个最小集合, 使该集合元素与整个数据能量的比例大于变量  $frac$ , 其中,  $svd.w$  各元素的值就表示数据在不同方向上的能量大小。由于这些元素是从大到小依次存储的, 只需简单获取前  $k$  个变化方向的能量就可得到要选择的子空间维数。数据变化方向本身存储在 SVD 类的成员变量  $u$  中。  $svd.w$  和  $svd.u$  中的每个元素通常称为特征值和特征向量。这两个变量中元素之间的关系如下图所示:



 **注意:** 上图的特征值会迅速减少, 这种现象暗示数据中的面部变化可用低维子空间来表示。

### 6.3.3 局部 - 全局相结合的表达

一个图像帧的形状由局部形变和全局变换组合在一起得到。从数学角度来讲, 这种参数化可能会产生问题, 因为这些变换组合后的结果会是一个非线性函数, 该函数没有闭解。要绕过这个问题的通常做法是将全局变换作为一个线性子空间, 并将其加到形变子空间中。对于一个具体的人脸, 其相似变换可用子空间来表示, 具体表示如下:

$$\begin{bmatrix} \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \\ \vdots \\ \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \begin{bmatrix} x_n \\ y_n \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \end{bmatrix} = \begin{bmatrix} x_1 & -y_1 & 1 & 0 \\ y_1 & x_1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_n & -y_n & 1 & 0 \\ y_n & x_n & 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ t_x \\ t_y \end{bmatrix}$$



在类 `shape_model` 类中，子空间由 `calc_rigid_basis` 函数得到。该子空间（即，上面那个方程中的  $x$  分量和  $y$  分量）表示的形状是用 Procrustes 对齐（Procrustes-aligned）后得到的平均形状（即，标准形状）。除了按上述形式构成子空间外，还要对矩阵每列进行归一化。在 `shape_model::train` 函数中，上一节介绍的变量  $dY$  是通过投影刚性运动数据的分量来得到的，具体计算过程如下：

```
Mat R = this->calc_rigid_basis(Y); //compute rigid subspace
Mat P = R.t()*Y; Mat dY = Y - R*P; //project-out rigidity
```

该投影可用简单的矩阵乘法来实现，这样做是可行的，因为刚性子空间的列已经被归一化。若这样做没有改变模型所生成的子空间，则只能说明  $R.t()*Y$  是单位矩阵。

在学习形变模型之前，由于刚性变换引起的变化方向已经从数据中删除，所产生的形变子空间会与刚性变换子空间正交。因此，可拼接两个子空间，从而得到脸型的局部 - 全局线性表示。这种拼接可用 OpenCV 的 `Mat` 类的 ROI 提取机制来实现，即分配两个子空间矩阵作为所组合的子空间矩阵的次矩阵。具体实现如下：

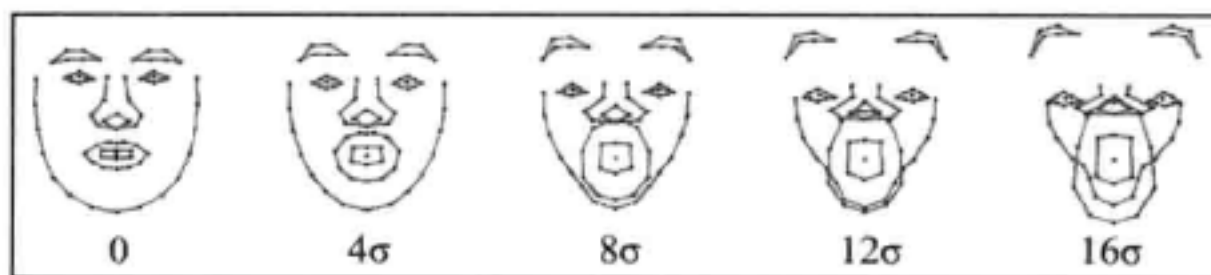
```
V.create(2*n,4+k,CV_32F); //combined subspace
Mat Vr = V(Rect(0,0,4,2*n)); R.copyTo(Vr); //rigid subspace
Mat Vd = V(Rect(4,0,k,2*n)); D.copyTo(Vd); //nonrigid subspace
```

这种模型的正交性意味着描述形状的参数很容易计算，用 `shape_model::calc_params` 函数即可：

```
p = V.t()*s;
```

这里的  $s$  是一个向量化的人脸， $p$  存放着用人脸子空间表示脸型的坐标。

最后要注意的是线性模型化脸型要怎样约束子空间坐标，才使生成的形状有效。在下图中，对子空间表示的人脸做这样的操作：依次对每个人脸在某个变化的方向上增加相应坐标的值，其增量为该方向上 4 倍的标准偏差。注意增加的量越小，得到的形状越像人脸，而增加的量越大，人脸看上去就会越糟糕。



防止这种变形的一个简单方法是限制（clamp）子空间坐标，使其只能在由数据集所决定的一个区间内。通常的做法是用  $\pm 3$  倍的数据标准偏差作为一个箱约束（box constraint），这个部分占数据变化部分的 99.7%。在子空间找到后，可通过 `shape_model::train` 函数来计算该限制值。具体实现如下：

```
Mat Q = V.t()*X; //project raw data onto subspace
for(int i = 0; i < N; i++){ //normalize coordinates w.r.t scale
    float v = Q.fl(0,i); Mat q = Q.col(i); q /= v;
```

```

}
e.create(4+k,1,CV_32F); multiply(Q,Q,Q);
for(int i = 0; i < 4+k; i++){
    if(i < 4)e.fl(i) = -1; //no clamping for rigid coefficients
    else e.fl(i) = Q.row(i).dot(Mat::ones(1,N,CV_32F))/(N-1);
}

```

注意，在对第一维坐标归一化（即，伸缩）后，方差（variance）可用子空间坐标  $Q$  来计算，这样就可防止相对较大的数据样本影响估计。另外还需注意，负值会分配给刚性子空间中坐标的方差（即  $V$  的前 4 个列）。限制函数 `shape_model::clamp` 会检查某个方向的方差是否为负，如果不为负值，才对其使用限制（clamping）。该函数的具体实现如下：

```

void shape_model::clamp(
const float c){ //clamping as fraction of standard deviation
    double scale = p.fl(0); //extract scale
    for(int i = 0; i < e.rows; i++){
        if(e.fl(i) < 0)continue; //ignore rigid components
        float v = c*sqrt(e.fl(i)); //c*standard deviations box
        if(fabs(p.fl(i)/scale) > v){ //preserve sign of coordinate
            if(p.fl(i) > 0)p.fl(i) = v*scale; //positive threshold
            else p.fl(i) = -v*scale; //negative threshold
        }
    }
}

```

这样做是因为在人为拍摄的训练数据中，其图像中的人脸在某一尺度上是直立（upright）且被中心化的。限制脸型的刚性部分会使训练集的结构变得更具约束性。最后，由于每个形变的坐标方差可由尺度归一化（scale-normalized）帧来计算，在限制过程中，坐标必须采用相同尺度。

### 6.3.4 训练与可视化

从标注数据中训练人脸模型的示例程序可从文件 `train_shape_model.cpp` 中找到。命令行参数 `argv[1]` 用来存放标注数据的路径，加载数据到内存中并删除不完整的数据后，就可开始训练。具体实现如下：

```

ft_data data = load_ft<ft_data>(argv[1]);
data.rm_incomplete_samples();

```

每个样本的标注和相应的镜像在被传递给训练函数前会保存在一个向量中，具体保存形式为：

```

vector<vector<Point2f> > points;
for(int i = 0; i < int(data.points.size()); i++){
    points.push_back(data.get_points(i,false));
    if(mirror)points.push_back(data.get_points(i,true));
}

```

通过函数 `shape_model::train` 来训练人脸模型。具体调用如下：

```
shape_model smodel; smodel.train(points,data.connections,frac,kmax);
```

这个函数的参数 `frac`（即保留部分的比例）和参数 `KMAX`（即保留特征向量的最大数目）可通过命令行任意设置。这两个参数的默认值分别为 0.95 和 20，这在绝大多数情形下都能让应用很好地工作。命令行参数 `argv[2]` 含有存储训练得到的人脸模型的路径，可通过下面的函数来实现存储：

```
save_ft(argv[2],smodel);
```

这步很简单，其原因是：在 `shape_model` 类中定义了具有序列化的 `read` 和 `write` 函数

为了可视化训练得到的人脸模型，可用 `visualize_shape_model.cpp` 程序，它能依次展示在每个方向上所学到的非刚性形变。

加载脸形模型到内存后就可进行可视化，加载过程如下：

```
shape_model smodel = load_ft<shape_model>(argv[1]);
```

下面的方法可计算出将模型放置在显示窗口中心的刚性参数值：

```
int n = smodel.V.rows/2;
float scale = calc_scale(smodel.V.col(0),200);
float tranx =
n*150.0/smodel.V.col(2).dot(Mat::ones(2*n,1,CV_32F));
float trany =
n*150.0/smodel.V.col(3).dot(Mat::ones(2*n,1,CV_32F));
```

其中 `calc_scale` 函数用来查找尺度系数，该系数将用 200 像素作为宽度来生成脸形。通过搜索可生成 150 个像素的平移系数来计算平移分量（即均值中心化模型，显示窗口大小为  $300 \times 300$  个像素）。



**注意：**`shape_model::V` 的第一列对应尺度变换，而第三列和第四列分别是 `x` 和 `y` 的平移。

下面将生成参数值的轨迹，该轨迹从零点开始，先向正向端点（`positive extreme`）移动，再称负向端点（`negative extreme`）移动，然后移回零点。具体的代码实现如下：

```
vector<float> val;
for(int i = 0; i < 50; i++)val.push_back(float(i)/50);
for(int i = 0; i < 50; i++)val.push_back(float(50-i)/50);
for(int i = 0; i < 50; i++)val.push_back(-float(i)/50);
for(int i = 0; i < 50; i++)val.push_back(-float(50-i)/50);
```

动画每阶段会变化 50 次，每次变化量都为 0.02。该轨迹用于展现人脸模型并将结果显示在窗口中，其具体实现如下：

```
Mat img(300,300,CV_8UC3); namedWindow("shape model");
while(1){
    for(int k = 4; k < smodel.V.cols; k++){
```



```

for(int j = 0; j < int(val.size()); j++){
    Mat p = Mat::zeros(smodel.V.cols,1,CV_32F);
    p.at<float>(0) = scale;
    p.at<float>(2) = tranx;
    p.at<float>(3) = trany;
    p.at<float>(k) = scale*val[j]*3.0*
                    sqrt(smodel.e.at<float>(k));
    p.copyTo(smodel.p); img = Scalar::all(255);
    vector<Point2f> q = smodel.calc_shape();
    draw_shape(img,q,smodel.C);
    imshow("shape model",img);
    if(waitKey(10) == 'q')return 0;
}
}
}

```



**注意：**刚性系数（即 `shape_model::V` 的前四列）总是设置为先前计算的值，以便将人脸放置在显示窗口中心。

## 6.4 面部特征检测器

检测图像中的面部特征与一般物体检测很相似。OpenCV 有一系列用于构建通用对象检测器的复杂函数。这些检测器中最著名是基于 Haar 特征的级联检测器，它用来实现著名的 Viola-Jones 人脸检测器。但一般物体检测与人脸检测略有不同，这些不同之处是人脸检测所特有的，具体表现为以下内容。

- **精度和鲁棒性：**一般物体检测的目标在于找到图像中粗略的物体位置，面部特征检测器需要对特征位置有一个高精度估计。在物体检测中，几个像素的误差被认为无关紧要。但在面部表情估计中，需通过特征检测来区分微笑和皱眉之间的差异时，这样的误差就会有问题。
- **有限空间支持的模糊性：**在一般物体检测中通常会假设人们感兴趣的对象会有丰富的图像结构，这些结构能可靠地判别出图像中有没有包含要检测的对象。这种假设往往不适合面部特征，面部特征通常只有有限的空间支持。例如，对于一个面部边缘的特征，若通过一个四周封闭的小盒子的中心来观察该特征，然后用同样的方式来观察其他有强边缘的图像块，很容易混淆所观察到的结果。
- **计算复杂度：**一般对象检测的目标是找到图像中所有的对象。然后人脸跟踪需要所有面部特征的位置，这种特征的数量通常从 20 到 100 不等。因此，在实时的人脸跟踪系统中，特征检测器的效率至关重要。

由于这些差异，用于人脸跟踪的面部特征检测器通常需要针对不同目的而专门设计。当然，在人脸跟踪系统中，许多对一般物体的检测技术也可用作面部特征检测。然而，在社区中，仍然没达成哪种技术最适合于该问题的共识。

本节将用一种表示方法来建立人脸特征检测器，该方法也许是人们认为最简单的模型，即：线性图像块模型。尽管它简单，但仍有必要仔细设计它的学习过程。事实上，这种方法用在人脸跟踪算法中，能给出面部特征位置的合理估计。此外，算法的简单性能得到一个非常快速的评估，使实时人脸跟踪成为可能。由于该方法需表示一个图像块，因此这种面部特征检测器称为块模型（patch model）。该模型在 patch\_model 类中被实现，该类的定义与实现可分别在 patch\_model.hpp 文件和 patch\_model.cpp 文件中找到。下面的这段代码给出了 patch\_model 类的主要功能：

```
class patch_model{
public:
    Mat P; //normalized patch
    ...
    Mat //response map
    calc_response(
        const Mat &im, //image patch of search region
        const bool sum2one = false); //normalize to sum-to-one?
    ...
    void
    train(const vector<Mat> &images, //training image patches
        const Size psize, //patch size
        const float var = 1.0, //ideal response variance
        const float lambda = 1e-6, //regularization weight
        const float mu_init = 1e-3, //initial step size
        const int nsamples = 1000, //number of samples
        const bool visi = false); //visualize process?
    ...
};
```

用于检测面部特征的块模型存储在矩阵 P 中。该类有两个重要的函数：calc\_response 和 train。calc\_response 函数会对搜索区域 im 的每个元素（integer displacements）计算块模型的响应值。train 函数用来得到块模型（patch model），其大小由 psize 参数决定。通常这个在训练集上所产生的响应矩阵（response map）尽可能接近理想的响应矩阵。参数 var、lambda、mu\_init 和 nsamples 用于训练过程，在这个过程中，这些参数可随时用来对数据进行性能优化。

本节将详细介绍该类的功能。下面先来讨论相关性块（correlation patch）和它的训练过程，相关性块将被用来学习块模型。接下来介绍 patch\_models 类，该类保存着每个面部特征的一些块模型，并实现了全局变换的功能。在文件 train\_patch\_model.cpp 和文件 visualize\_patch\_model.cpp 中的程序分别用来训练和可视化块模型。在本节最后将简单介绍它们的用法。

### 6.4.1 相关性块模型

学习检测器可用两种主要方法：生成方法和判断方法，这两种方法的思想完全不一样。

生成方法会学习一个图像块底层表示，这种表示在各种情况下都能最恰当地生成对象外观。而判断方法根据已有的对象的信息来对新对象做出最好的判断，这些已有样本来源于运行的系统。生成方法的优势在于能对具体对象的属性进行操作，可直观地查看新的对象实例的情况。著名的特征脸（Eigenface）方法就是一种流行的生成方法。判别方法的优势在于所建模型直接针对当前问题，通过已有对象来对新对象做出判别。所有判别方法中最著名的也许是支持向量机。虽然这两大类方法在许多情况下都工作得很好，但将一个图像块作为面部特征来建模时，判别方法更好一些。



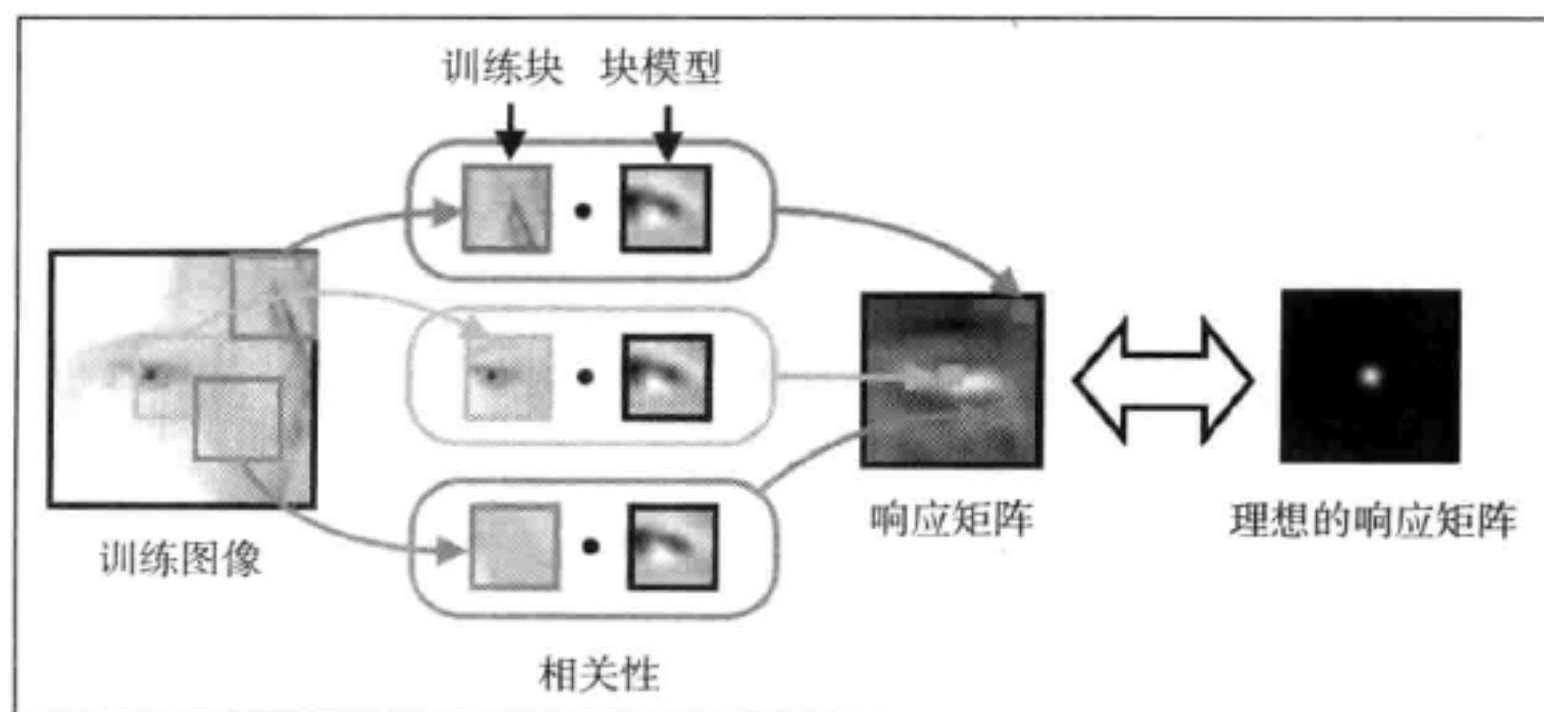
**注意：**特征脸和支持向量机最初是用于分类而不是用于检测或图像对齐，但其基本的数学思想可用于人脸跟踪领域。

### 1. 学习基于判别法的块模型

给定一个标注数据集，特征检测器可从这些数据学习得到。判别块模型的学习目标是为了构造这样的图像块：当图像块与含有面部特征的图像区域交叉相关（cross-correlated）时，对特征区域有一个强的响应，而其他部分响应很弱。该模型用数学化方式可表示为：

$$\min_P \sum_{i=1}^N \sum_{x,y} \left[ R(x, y) - P \cdot I_i \left( x - \frac{w}{2} : x + \frac{w}{2}, y - \frac{h}{2} : y + \frac{h}{2} \right) \right]^2$$

这里， $P$  表示块模型， $I_i$  表示第  $i$  个训练图像， $I_i(a:b, c:d)$  表示是一个矩形区域，它的左上角位置和右下角位置分别是  $(a, c)$  和  $(b, d)$ ，圆点表示内积操作， $R$  表示理想的响应矩阵（response map）。这个目标函数的解就是一个块模型，此模型会得到一个响应矩阵，该矩阵通常在最小二乘的度量标准下最靠近理想的响应矩阵。对理想的响应矩阵  $R$  的一个常见选择（假定面部特征集中在训练图像块的中心）是，除中心外其他地方都设为零。在实践中，由于图像被手工标记，总会有标注误差。为了解决这个问题，通常会用衰减函数来刻画  $R$ ，该函数从中心开始，随着距离增加，函数值会迅速变小。二维高斯分布可很好地充当这种函数，这也相当于假定标注误差服从高斯分布。这个过程可用下图来描述，该图为人脸左眼角外部。





上面给出的目标函数通常称为线性最小二乘。因此，它有一个闭解。但该问题的自由度，也就是该方法的变量数与块中像素一样多。因此，求解最优块模型的计算代价和所需内存会让人望而却步，例如，一个  $40 \times 40$  块模型，就有 1600 个变量。

解决该问题的有效替代方法是将其看成线性方程，用随机梯度下降法来求解。将目标函数图像想象成由块模型变量构成的高低起伏不平的地形 (error terrain)，对于这种情形，随机梯度下降通过迭代来对梯度方向进行粗略估计，并用一个小的步长乘以该方向的反方向作为下一步迭代的方向。在这个应用中，仅需从训练集中随机选择一幅图像，将该图像代入目标函数的梯度公式，由此计算所得结果就是所需的近似梯度。目标函数的梯度公式为

$$D = -\sum_{x,y} (R(x,y) - P \cdot W) W; W = I \left( x - \frac{w}{2} : x + \frac{w}{2}, y - \frac{h}{2} : y + \frac{h}{2} \right)$$

在 patch\_model 类的 train 函数中实现了这个学习过程：

```
void
patch_model::train(
    const vector<Mat> &images, //featured centered training images
    const Size psize,         //desired patch model size
    const float var,          //variance of annotation error
    const float lambda,       //regularization parameter
    const float mu_init,      //initial step size
    const int nsamples,       //number of stochastic samples
    const bool visi){         //visualise training process
    int N = images.size(), n = psize.width*psize.height;
    int dx = wsize.width-psize.width; //center of response map
    int dy = wsize.height-psize.height; //...
    Mat F(dy,dx,CV_32F); //ideal response map
    for(int y = 0; y < dy; y++){ float vy = (dy-1)/2 - y;
        for(int x = 0; x < dx; x++){float vx = (dx-1)/2 - x;
            F.fl(y,x) = exp(-0.5*(vx*vx+vy*vy)/var); //Gaussian
        }
    }
    normalize(F,F,0,1,NORM_MINMAX); //normalize to [0:1] range

    //allocate memory
    Mat I(wsize.height,wsize.width,CV_32F);
    Mat dP(psize.height,psize.width,CV_32F);
    Mat O = Mat::ones(psize.height,psize.width,CV_32F)/n;
    P = Mat::zeros(psize.height,psize.width,CV_32F);

    //optimise using stochastic gradient descent
    RNG rn(getTickCount()); //random number generator
    double mu=mu_init,step=pow(1e-8/mu_init,1.0/nsamples);
    for(int sample = 0; sample < nsamples; sample++){
        int i = rn.uniform(0,N); //randomly sample image index
        I = this->convert_image(images[i]); dP = 0.0;
        for(int y = 0; y < dy; y++){ //compute stochastic gradient
```

```

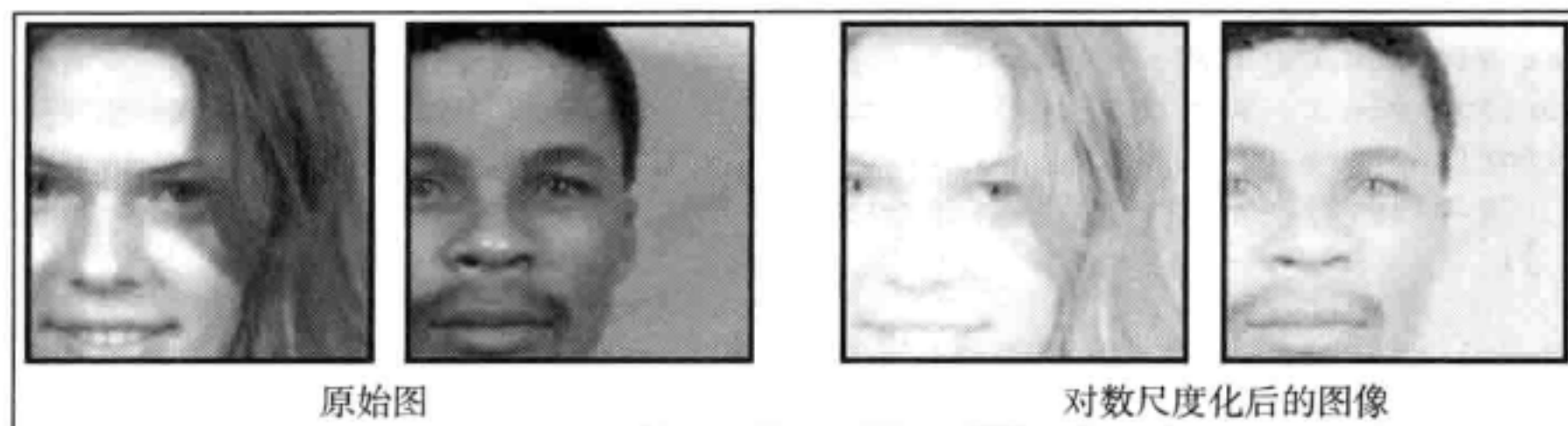
    for(int x = 0; x < dx; x++){
        Mat Wi=I(Rect(x,y,psize.width,psize.height)).clone();
        Wi -= Wi.dot(O); normalize(Wi,Wi); //normalize
        dP += (F.fl(y,x) - P.dot(Wi))*Wi;
    }
}
P += mu*(dP - lambda*P); //take a small step
mu *= step;               //reduce step size
...
}return;
}

```

在上面的这些代码中，加粗的第一部分代码是用来计算理想响应矩阵的。由于图像需要被集中感兴趣的面部特征上，响应矩阵对所有样本都一样。在加粗的第二部分代码中，变量 `step` 表示步长（step size）的衰减率，在经过 `nsamples` 次迭代后，步长的值会接近零。加粗的第三部分代码用来计算随机梯度方向并用其更新块模型。这里有两件事情需要注意，第一，用来训练的图像先传递给 `patch_model::convert_image` 函数，该函数将图像转换为单通道图像（如果是彩色图像），并采用自然对数来得到图像像素强度：

```
I += 1.0; log(I,I);
```

在使用自然对数之前，每个像素值都要加一，因为零在自然对数中没有意义。在训练集上执行这样的预处理的原因是：经对数尺度（log-scale）化后的图像对对比度差异和光照条件变化更具有鲁棒性。下图是两幅人脸图像，其面部区域的对比度不一样，两幅图像在经过对数尺度（log-scale）化处理后，这种差异减少了很多。



第二，需要注意的是，更新等式中会从更新方向减去  $\lambda * P$ ，这样做可有效地防止解变得太大，这个过程经常用于机器学习算法以获得对未知数据的泛化能力。标量  $\lambda$  由用户定义，与求解的问题有关。但通过学习块模型来检测面部特征时， $\lambda$  取一个较小的值通常都会得到更好的效果。

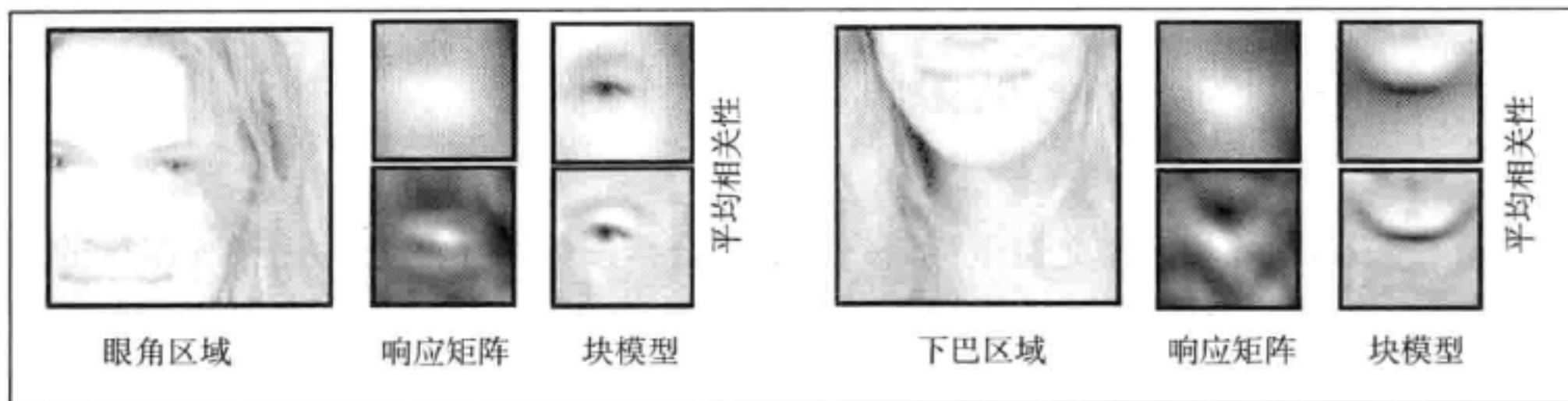
## 2. 生成块模型与判别块模型

虽然上面介绍的判别块模型很简洁，但对于相似的结果，是否生成模型和相应的训练方法会更简单？基于生成法的块模型对应的结果是平均块。这种模型的目标函数是通过获得一个图像块，使其尽量逼近所有样本的面部特征，对这种逼近的度量采用的是最小二乘标准：

$$\min_P \sum_{i=1}^N \|P - I_i\|_F^2$$

其解为所有特征中心（feature-centered）化的训练图像块的平均值。因此，从某种意义上讲，这样得到的解更简单。

下图分别展示用交叉相关（cross-correlating）平均化与用相关块模型来得到一幅图像的响应矩阵的差别，平均块模型和相关块模型也分别显示在这个图中。其中，为了可视化需要，对像素值的范围作了归一化处理。虽然两种块模型有一些相似的地方，但它们所生成的响应矩阵却有本质的不同。相关块模型生成的响应矩阵在特征位置周围变化非常明显，而由平均块模型生成的响应矩阵在特征位置周围相当光滑，变化不明显。观察这些块模型的外观会发现：相关块模型大多为灰色，这对应归一化前像素范围为0的像素。这是按一定策略，将差异很大的值放在显著的面部特征周围后得到的。这样只保留训练块有用的部分，以区分未对齐的结构。相比之下，平均块模型不区分非对齐数据，这使其不能很好地用于面部特征局部化的任务，即不能从改变后的图像块中找到与原图像对齐的图像块。



### 6.4.2 解释全局几何变换

到目前为止，假定训练图像以面部特征进行中心化并以全局尺度和旋转进行归一化。实际上，在跟踪过程中，人脸图像随时会出现尺度或旋转变换。因此，系统必须考虑训练和测试条件之间的差异。一种方法是通过人为地在一定范围内进行尺度变换和旋转来扰动训练图像，这些被扰动的图像可能会出现在系统运行期间。但针对这类数据，简单的相关块模型（correlation patch model）检测器不能生成好的响应矩阵。相关块模型在对尺度变换和旋转进行较小扰动的时候会表现出一定的鲁棒性。由于在一个视频中连续帧的变化相对较小，可利用前一帧对人脸所估计全局变换来对当前图像的尺度变换和旋转进行归一化处理。实现这一过程只需在学习相关块模型时选择一个参考帧（reference frame）即可。

patch\_models 类对每个面部特征存储了相关块模型，以及训练时获得的参考帧。注意这里说的是 patch\_models 类而不是 patch\_model 类，patch\_model 类包含了人脸跟踪器的代码，这些代码直接用于特征检测。下面这段代码是 patch\_models 类的主要功能：



```

class patch_models{
public:
    Mat reference;          //reference shape [x1;y1;...;xn;yn]
    vector<patch_model> patches; //patch model/facial feature
    ...
    void
    train(ft_data &data,          //annotated image and shape data
    const vector<Point2f> &ref, //reference shape
    const Size psize,            //desired patch size
    const Size ssize,            //training search window size
    const bool mirror = false,   //use mirrored training data
    const float var = 1.0,        //variance of annotation error
    const float lambda = 1e-6,   //regularisation weight
    const float mu_init = 1e-3,  //initial step size
    const int nsamples = 1000,   //number of samples
    const bool visi = false);    //visualise training procedure?
    ...
    vector<Point2f> //location of peak responses/feature in image
    calc_peaks(
    const Mat &im,          //image to detect features in
    const vector<Point2f> &points, //current estimate of shape
    const Size ssize = Size(21,21)); //search window size
    ...
};

```

变量 `reference` 用来保存交错坐标  $(x,y)$  的集合，该集合用于归一化训练图像的尺度变换和旋转，以及后期运行过程中的测试图像。在 `patch_models::train` 函数中，首先要使用 `patch_models::calc_simil` 函数来计算给定图像在 `reference` 形状与标注形状（`annotated shape`）之间的相似性，尽管 `shape_model::procrustes` 针对的是一对形状，但此函数求解与它类似。由于尺度变换和旋转通常用于所有的面部特征，因此，图像归一化程序只需调整这个相似变换，使其成为每个特征图像和归一化图像块的中心。在 `patch_models::train` 函数中实现了此过程：

```

Mat S = this->calc_simil(pt), A(2,3,CV_32F);
A.fl(0,0) = S.fl(0,0); A.fl(0,1) = S.fl(0,1);
A.fl(1,0) = S.fl(1,0); A.fl(1,1) = S.fl(1,1);
A.fl(0,2) = pt.fl(2*i) - (A.fl(0,0)*(wsize.width-1)/2 +
                        A.fl(0,1)*(wsize.height-1)/2);
A.fl(1,2) = pt.fl(2*i+1) - (A.fl(1,0)*(wsize.width-1)/2 +
                        A.fl(1,1)*(wsize.height-1)/2);
Mat I; warpAffine(im,I,A,wsize,INTER_LINEAR+WARP_INVERSE_MAP);

```

这里的 `wsize` 是训练图像归一化后总的大小，它是块大小和搜索区域大小之和。在 `reference` 形状与标注形状 `pt` 之间进行相似变换后，取其左上方  $(2 \times 2)$  的块用作仿射变换，将变换后的结果传递给 OpenCV 的 `warpAffine` 函数。仿射变换 `A` 的最后一列是一个调节器（`adjustment`），也就是说，在扭曲（即，归一化平移）后，该调节器要提供第  $i$  个面部特征在归一化图像中进行中心化后的位置。最后，CV:: `warpAffine` 函数有从图像到参考帧

之间产生扭曲后的默认设置。由于变换 reference 形状到图像空间的标注 pt 是通过相似变换计算得到的, 因此, 需要设置 WARP\_INVERSE\_MAP 标志以确保函数在希望的方向上使用扭曲。在 patch\_models::calc\_peaks 函数中也执行完全一样的过程, 只是多一个步骤, 即, 在图像帧中, 会将 reference 形状和当前形状之间计算所得的相似变换重新用于已检测到的面部特征, 以进行非规范化处理 (un-normalize), 并将其放置在图像的适当位置。

```
vector<Point2f>
patch_models::calc_peaks(const Mat &im,
    const vector<Point2f> &points, const Size ssize){
    int n = points.size(); assert(n == int(patches.size()));
    Mat pt = Mat(points).reshape(1, 2*n);
    Mat S = this->calc_simil(pt);
    Mat Si = this->inv_simil(S);
    vector<Point2f> pts = this->apply_simil(Si, points);
    for(int i = 0; i < n; i++){
        Size wsize = ssize + patches[i].patch_size();
        Mat A(2, 3, CV_32F), I;
        A.fl(0, 0) = S.fl(0, 0); A.fl(0, 1) = S.fl(0, 1);
        A.fl(1, 0) = S.fl(1, 0); A.fl(1, 1) = S.fl(1, 1);
        A.fl(0, 2) = pt.fl(2*i) - (A.fl(0, 0)*(wsize.width - 1)/2 +
                                   A.fl(0, 1)*(wsize.height - 1)/2);
        A.fl(1, 2) = pt.fl(2*i+1) - (A.fl(1, 0)*(wsize.width - 1)/2 +
                                      A.fl(1, 1)*(wsize.height - 1)/2);
        warpAffine(im, I, A, wsize, INTER_LINEAR+WARP_INVERSE_MAP);
        Mat R = patches[i].calc_response(I, false);
        Point maxLoc; minMaxLoc(R, 0, 0, 0, &maxLoc);
        pts[i] = Point2f(pts[i].x + maxLoc.x - 0.5*ssize.width,
                         pts[i].y + maxLoc.y - 0.5*ssize.height);
    }return this->apply_simil(S, pts);
}
```

在上面被加粗的第一部分代码中, 计算了正向和逆向的相似变换。需要计算逆向相似变换的原因是每个特征的响应矩阵峰值可按当前所估计形状归一化后的位置进行调整。这步必须在重新使用相似变换之前执行, 这里使用相似变换的目的是通过 patch\_models::apply\_simil 函数将新估计的面部特征位置放回到图像帧中。

### 6.4.3 训练与可视化

在 train\_patch\_model.cpp 文件中有通过标注数据来训练块模型的示例程序。命令行参数 argv[1] 保存着标注数据的路径, 通过它可加载数据到内存, 并删除不完备样本, 然后开始训练:

```
ft_data data = load_ft<ft_data>(argv[1]);
data.rm_incomplete_samples();
```

在 patch\_models 类中, 选择 reference 形状的最简单方法是采用训练集的平均形状, 并对其进行尺度变换来获得所需要的大小。假设前面通过对数据集训练已得到一个形状模型,

且将该模型存储在 `argv[2]` 参数所指向的路径中，可通过 `argv[2]` 参数来加载此模型，然后由此计算选择的 reference 形状。具体实现如下：

```
shape_model smodel = load_ft<shape_model>(argv[2]);
```

下面是对已缩放且被中心化后的平均形状进行计算：

```
smodel.p = Scalar::all(0.0);
smodel.p.fl(0) = calc_scale(smodel.V.col(0),width);
vector<Point2f> r = smodel.calc_shape();
```

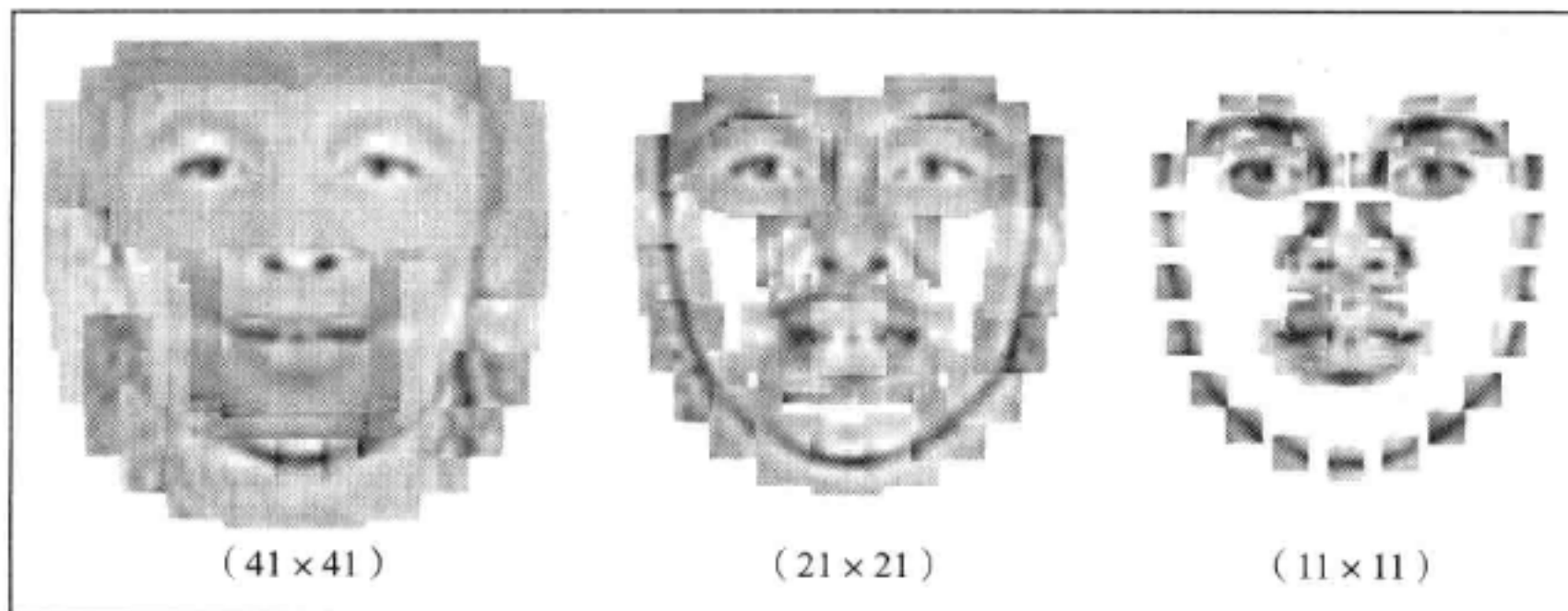
`calc_scale` 函数通过计算尺度缩放因子来将平均形状（即 `shape_model::V` 的第一列）转换成具有 `width` 宽的形状。在得到 reference 形状 `r` 后，可通过下面这个函数来训练块模型集合：

```
patch_models pmodel; pmodel.train(data,r,Size(psize,psize),Size(ssize,ssize));
```

参数 `width`、`psize` 以及 `ssize` 的最佳值跟应用有关，但一般它们分别取 100、11、11 就能得到很好的效果。

虽然这个训练过程很简单，但仍需要一些时间才能完成。这个时间由面部特征数、块大小以及优化算法中随机样本的数量决定。该过程所花的时间从几分钟到一个多小时不等，但每个块的训练可被单独执行，在多核处理器（或处理机）上进行并行化可大大加速此训练过程。

一旦训练完成，就可用文件 `visualize_patch_model.cpp` 中的程序对得到的块模型进行可视化操作。使用该程序的目的是通过可视化方式来查看所得结果，以便验证是否在训练过程中有错误。这个程序可得到：所有块模型的复合图像 `patch_model::P`、参考形状中各特征所在位置的中心、`patch_models::reference`、显示当前活动块周围有界的矩形区域。`cv::waitKey` 函数用来得到用户输入活动块的索引或终止程序。下图为三个复合块图像，它们来自各种块模型空间。虽然使用了同样的训练数据，块模型所在的空间不同，会使其结构差异很大。用这种可视化方式来查看训练结果，其优势在于可从直觉上判断如何修改训练程序的参数，甚至是训练程序本身，这样就可针对具体应用来优化结果。





## 6.5 人脸检测与初始化

迄今为止所介绍的人脸跟踪方法都是假设图像中所找到的面部特征与当前的估计比较接近。虽然整个跟踪过程中帧之间的人脸变化相当小，这样的假设可认为很合理，但必须要面对的问题是，如何用视频序列的第一帧来初始化模型。解决该问题的简单方法是使用 OpenCV 内置的级联检测器来搜索人脸。然而，模型在检测区域的位置将取决于对所跟踪的面部特征的选择。一种基于数据驱动模式的简单方案是，学习人脸检测区域与人脸特征之间的几何关系。

face\_detector 类完全实现了该解决方案。下面这段代码描述了该类的主要功能：

```
class face_detector{ //face detector for initialisation
public:
    string detector_fname; //file containing cascade classifier
    Vec3f detector_offset; //offset from center of detection
    Mat reference;         //reference shape
    CascadeClassifier detector; //face detector

    vector<Point2f> //points describing detected face in image
    detect(const Mat &im, //image containing face
        const float scaleFactor = 1.1, //scale increment
        const int minNeighbours = 2, //minimum neighborhood size
        const Size minSize = Size(30,30)); //minimum window size

    void
    train(ft_data &data, //training data
        const string fname, //cascade detector
        const Mat &ref, //reference shape
        const bool mirror = false, //mirror data?
        const bool visi = false, //visualize training?
        const float frac = 0.8, //fraction of points in detection
        const float scaleFactor = 1.1, //scale increment
        const int minNeighbours = 2, //minimum neighbourhood size
        const Size minSize = Size(30,30)); //minimum window size
    ...
};
```

该类有四个公共成员变量：变量 detector\_fname 的类型为 cv::CascadeClassifier，它保存用于级联分类的文件名；变量 detector\_offset 是一个偏移量集合，它保存着图像的检测区域到人脸位置和尺度的偏移量；参考形状放在变量 reference 中；变量 detector 保存着人脸检测器。face\_detector::detect 方法是人脸跟踪的主要函数，该函数将一幅图像作为输入，还有一些 cv::CascadeClassifier 类所使用的标准参数，该函数返回对图像中面部特征位置的粗略估计。detect 函数的具体实现如下：

```
Mat gray; //convert image to grayscale and histogram equalize
if(im.channels() == 1)gray = im;
else cvtColor(im,gray,CV_RGB2GRAY);
Mat eqIm; equalizeHist(gray,eqIm);
```

```

vector<Rect> faces; //detect largest face in image
detector.detectMultiScale(eqIm, faces, scaleFactor,
                          minNeighbours, 0
                          | CV_HAAR_FIND_BIGGEST_OBJECT
                          | CV_HAAR_SCALE_IMAGE, minSize);
if(faces.size() < 1){return vector<Point2f>();}

Rect R = faces[0]; Vec3f scale = detector_offset*R.width;
int n = reference.rows/2; vector<Point2f> p(n);
for(int i = 0; i < n; i++){ //predict face placement
    p[i].x = scale[2]*reference.fl(2*i ) +
            R.x + 0.5 * R.width  + scale[0];
    p[i].y = scale[2]*reference.fl(2*i+1) +
            R.y + 0.5 * R.height + scale[1];
}return p;

```

上面的代码采用常规的方式来检测人脸，所不同的是在代码中使用了 CV\_HAAR\_FIND\_BIGGEST\_OBJECT 标志，设置该标志表示要跟踪图像中最明显的人脸。加粗代码的功能为根据人脸检测方框来将参考形状放置到图像中。变量 detector\_offset 由三部分构成：偏移量 (x,y)，它表示从人脸中心到人脸检测方框的距离；尺度因子，它表示为了最好地适应图像中的人脸，调整参考形状大小的比例。所有这三部分都是检测方框宽度的线性函数。

检测方框宽度与 detector\_offset 变量之间的这种线性关系可用 face\_detector::train 函数从标注数据集中学到。在加载训练数据到内存并分配参考形状后，学习程序开始了。具体实现如下：

```

detector.load(fname.c_str()); detector_fname = fname; reference = ref.
clone();

```

就像 patch\_models 类的参考形状一样，这里的参考形状选择也很简单，即，归一化数据集中的平均脸形即可。然后对数据集中的每幅图像（或相应的镜像副本调用 cv::CascadeClassifier 方法进行处理。为了防止误检测，需检查得到的检测结果，以确保检测方框中有足够多的标注点（看本节最后那张图）。

```

if(this->enough_bounded_points(pt, faces[0], frac)){
    Point2f center = this->center_of_mass(pt);
    float w = faces[0].width;
    xoffset.push_back((center.x -
                       (faces[0].x+0.5*faces[0].width))/w);
    yoffset.push_back((center.y -
                       (faces[0].y+0.5*faces[0].height))/w);
    zoffset.push_back(this->calc_scale(pt)/w);
}

```

如果检测框中的标注点数目大于变量 frac 的值，为了将这样的图像作为新实体添加到一个 STL vector 类对象中，需让检测框的宽度与偏移量参数之间保持线性关系。这里的 face\_detector::center\_of\_mass 函数是用来计算图像的标注点集合的中心，并用 face\_

`detector::calc_scale` 函数计算由参考形状变换到中心化标注形状的缩放因子。一旦所有图像被处理完, `detector_offset` 变量就被设置为所有图像特定 (image-specific) 的偏移量的中位数 (median):

```
Mat X = Mat(xoffset), Xsort, Y = Mat(yoffset), Ysort, Z =
Mat(zoffset), Zsort;
cv::sort(X, Xsort, CV_SORT_EVERY_COLUMN | CV_SORT_ASCENDING);
int nx = Xsort.rows;
cv::sort(Y, Ysort, CV_SORT_EVERY_COLUMN | CV_SORT_ASCENDING);
int ny = Ysort.rows;
cv::sort(Z, Zsort, CV_SORT_EVERY_COLUMN | CV_SORT_ASCENDING);
int nz = Zsort.rows;
detector_offset =
    Vec3f(Xsort.fl(nx/2), Ysort.fl(ny/2), Zsort.fl(nz/2));
```

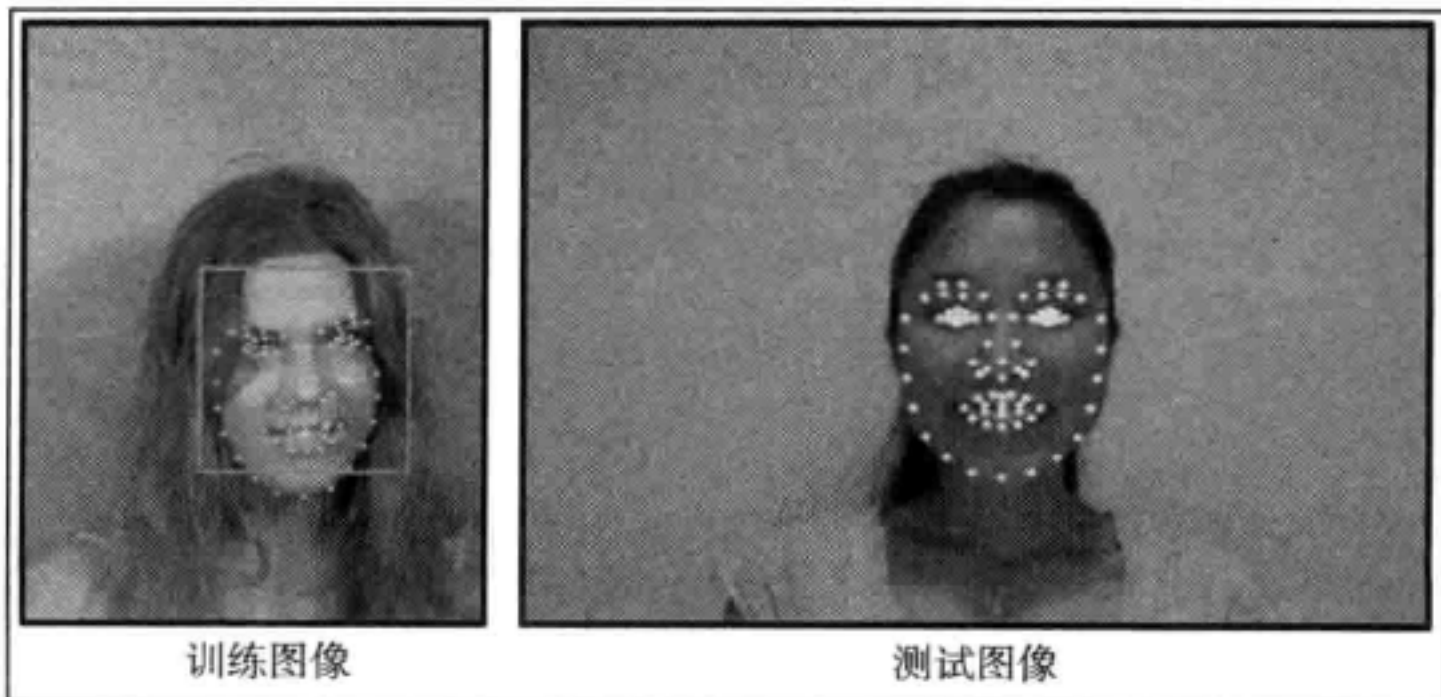
与形状和块模型一样, 在文件 `train_face_detector.cpp` 中有一个简单的示例程序, 该程序展示如何得到 `face_detector` 对象并保存这些对象以便用于后面的跟踪系统。如果第一次加载标注数据和形状模型, 将训练数据集均值中心化 (mean-centered) 后的平均值为参考形状 (即 `shape_model` 类的标识形状 (identity shape)):

```
ft_data data = load_ft<ft_data>(argv[2]);
shape_model smodel = load_ft<shape_model>(argv[3]);
smodel.set_identity_params();
vector<Point2f> r = smodel.calc_shape();
Mat ref = Mat(r).reshape(1, 2*r.size());
```

下面这两个函数用来训练和保存人脸检测器。

```
face_detector detector;
detector.train(data, argv[1], ref, mirror, true, frac);
save_ft<face_detector>(argv[4], detector);
```

为了检验形状放置 (shape-placement) 程序的性能, 文件 `visualize_face_detector.cpp` 中的程序会调用 `face_detector::detect` 函数来处理视频文件或摄像机输入的视频流, 并将结果显示在屏幕上。虽然放置的形状与图像中的人物并不一样, 但它的位置很靠近, 这样的结果可用于下一节将介绍的方法。





## 6.6 人脸跟踪

人脸跟踪问题可认为是寻找一种高效和鲁棒性的方法，它能够将各种面部特征的单独检测与这些特征的几何依赖性结合起来，以得到连续帧中每幅图像面部特征位置的精确估计。基于此，需仔细考虑几何依赖的必要性。下图为用几何约束和不用几何约束所检测出来的面部特征。该结果清楚地说明利用空间上面部特征的相互依赖性非常有好处。两种方法的相对性能通常会因检测结果噪声过多而受影响，其原因在于对于每一个面部特征的响应矩阵达到最大值时并不总是在正确的位置。但面部特征检测器要解决因图像噪声、光照变化、表情变化等带来的问题，仅有的方法是利用特征之间彼此共享的几何关系。



有一个特别简单，但非常有效的方法可将面部几何依赖加到跟踪过程里，该方法为将特征检测结果投影到线性形状模型的子空间中。这相当于最小化原始点与它最接近的且位于子空间的合理形状之间的距离。因此，若特征检测的空间噪声接近高斯分布（Gaussian distributed），投影会得到最优解。在实际情况中，检测误差偶尔并不服从高斯分布，这就需要引入其他方法来解决该问题。

### 6.6.1 人脸跟踪实现

人脸跟踪算法的实现可以在 `face_tracker` 类中找到（见 `face_tracker.cpp` 文件和 `face_tracker.hpp` 文件）。下面这段代码来自其头文件，这展示了该类的主要功能：

```
class face_tracker{
public:
    bool tracking;           //are we in tracking mode?
    fps_timer timer;         //frames/second timer
    vector<Point2f> points;   //current tracked points
    face_detector detector;  //detector for initialisation
    shape_model smodel;      //shape model
    patch_models pmodel;     //feature detectors

    face_tracker(){tracking = false;}

    int track(const Mat &im, //image containing face
              const face_tracker_params &p = //fitting parameters
```

```

face_tracker_params()); //default tracking parameters

void
reset() { //reset tracker
    tracking = false; timer.reset();
}
...
protected:
    ...
    vector<Point2f> //points for fitted face in image
    fit(const Mat &image, //image containing face
        const vector<Point2f> &init, //initial point estimates
        const Size ssize = Size(21,21), //search region size
        const bool robust = false, //use robust fitting?
        const int itol = 10, //maximum number of iterations
        const float ftol = 1e-3); //convergence tolerance
};

```

该类有公共成员实例 (member instances): `shape_model` 类、`patch_models` 类和 `face_detector` 类。可使用这三个类的功能来进行跟踪。`timer` 变量是 `fps_timer` 类的实例, 用来保存帧速率 (frame-rate) 的变化, 在 `face_tracker::track` 函数中会使用它。该变量用于分析块模型和形状模型对算法计算复杂度的影响很有用。`tracking` 成员变量是一个标志, 用来指示跟踪过程的当前状态。当在该类的构造函数和 `face_tracker::reset` 函数中, 会将此标志设置为 `false`, 即表示跟踪器进入检测模式, 为了初始化该模型, 可随后获得的图像使用 `face_detector::detect` 函数。当在跟踪模式下, 对下一幅图像的面部特征位置的初始估计可简单使用前一帧的面部位置。下面是整个跟踪算法的简单实现:

```

int face_tracker::
track(const Mat &im, const face_tracker_params &p) {
    Mat gray; //convert image to grayscale
    if(im.channels()==1) gray=im;
    else cvtColor(im, gray, CV_RGB2GRAY);
    if(!tracking) //initialize
        points = detector.detect(gray, p.scaleFactor,
                                p.minNeighbours, p.minSize);
    if((int)points.size() != smodel.npts()) return 0;
    for(int level = 0; level < int(p.ssize.size()); level++)
        points = this->fit(gray, points, p.ssize[level],
                           p.robust, p.itol, p.ftol);
    tracking = true; timer.increment(); return 1;
}

```

跟踪算法的核心是多层次拟合 (multi-level fitting) 过程, 这与设置恰当的跟踪状态和递增的跟踪时间不一样, 这些操作都很简单, 上面加粗的那段代码就是多层次拟合。拟合算法在 `face_tracker::fit` 函数中实现, 该算法会被调用多次, 每次会使用存储在 `face_tracker_params::ssize` 中不同的搜索窗口尺寸, 并将上次得到的跟踪点作为该函数的输入。

在最简单的情况下，`face_tracker_params::ssize` 函数的作用是根据图像中当前估计的形状来执行面部跟踪：

```
smodel.calc_params(init);
vector<Point2f> pts = smodel.calc_shape();
vector<Point2f> peaks = pmodel.calc_peaks(image,pts,ssize);
```

将结果投影到人脸形状子空间中：

```
smodel.calc_params(peaks);
pts = smodel.calc_shape();
```

通过设置 `robust` 标志为 `true`，跟踪系统就会在检测面部特征位置时统计总的异常点（`outliers`）数，这使得拟合程序不是一个简单的投影过程，而是具有鲁棒性的模型。但实际应用中，当使用的搜索窗口的大小不断减少时，经常不需要统计总的异常点数，因为在被投影的形状上异常点通常都远小于总的点数，而且这些异常点有可能出现在下一次拟合和过程的迭代搜索区域以外。因此，缩小搜索区域是一种促进异常点减少的有效方案。

### 6.6.2 训练与可视化

不像本章描述的其他类，训练一个 `face_tracker` 对象不会涉及任何的学习过程。在 `train_face_tracker.cpp` 文件中简单实现了该功能，其代码如下：

```
face_tracker tracker;
tracker.smodel = load_ft<shape_model>(argv[1]);
tracker.pmodel = load_ft<patch_models>(argv[2]);
tracker.detector = load_ft<face_detector>(argv[3]);
save_ft<face_tracker>(argv[4],tracker);
```

这里的 `argv[1]` 至 `argv[4]` 分别包含对象 `shape_model`、`patch_model`、`face_detector` 和 `face_tracker` 的路径。在 `visualize_face_tracker.cpp` 文件中，对人脸跟踪器的可视化同样简单。可视化程序将 `cv::VideoCapture` 类从摄像机或视频文件的图像流作为输入，该程序有一个简单的循环，该循环在读到图像流最后或用户按 `Q` 键就会终止，否则就会跟踪出现的每一帧。用户随时可按 `D` 键来重置跟踪选项。

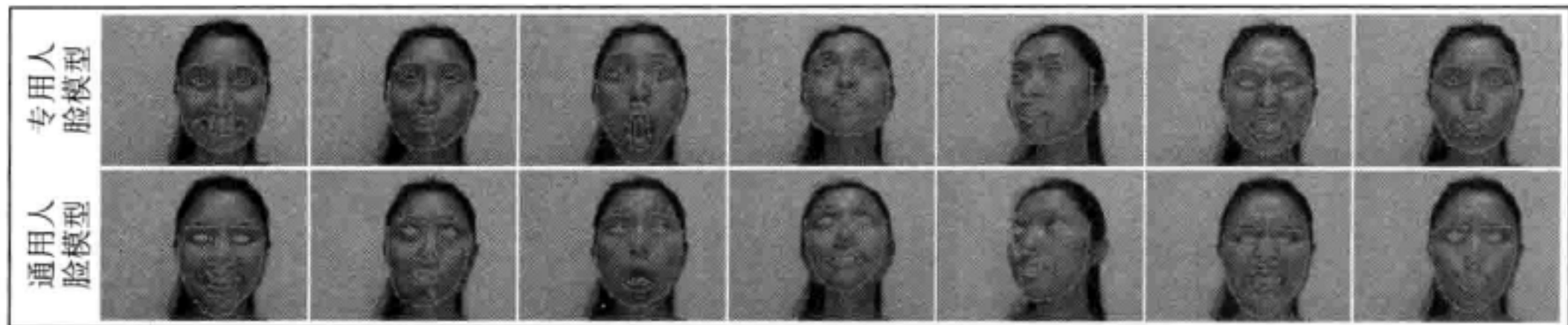
### 6.6.3 通用与专用人脸模型

可以调整训练过程和跟踪过程中的一些变量来优化具体应用的性能。但影响跟踪质量的其中一个主要决定因素是跟踪器对形状和外观变化适应能力。因此，可考虑通用人脸模型和专用人脸模型。通用人脸模型通过使用来自不同身份、表情、光照条件以及其他变化的标注数据来进行训练，而专用人脸模型针对具体某个人进行训练，这种模型所考虑的变化情形要少很多。因此在跟踪时，专用人脸模型部分通常比通用人脸模型部分有更高的精度。

下图显示了这样的例子。通用人脸模型用 MUCT 数据集进行训练，而专用人脸模型用本章前面介绍的标注工具所生成的数据进行训练。下图结果清楚表明专用人脸模型提供了



更加出色的跟踪，它能够获取复杂的表情和头部姿态的变化，而通用人脸模型似乎连获取一些简单表情都很困难。



应当注意，本章所介绍的是最基本的人脸跟踪方法，其目的是为了重点说明各种组件的使用，这些组件在大多数非刚性人脸跟踪算法都会用到。还有许多途径可用来解决这种方法的缺点，但需要专业的数学工具，这已超出本书的范围。目前 OpenCV 尚不支持这样的方法。商业级的人脸追踪软件相对较少，这也证明了该问题在通常情形下很难。本章所介绍的简单方法能很好地工作在特定环境（constrained settings）中。

## 6.7 总结

本章建立了一个简单的人脸追踪系统，它可在特定环境中正常工作，追踪系统只使用少量的数学工具以及大量 OpenCV 的基本图像处理功能和线性代数运算。使用更复杂的技术对形状模型、特征检测器以及拟合算法中的任何一部分改进，都可提高这个简单跟踪器的性能。本节介绍的跟踪器采用了模块化设计，可修改这三个组件中的任何一个而基本不会影响其他组件。

## 6.8 参考文献

- *Procrustes Problems*, Gower, John C. and Dijksterhuis, Garnt B, Oxford University Press, 2004.

## 基于 AAM 和 POSIT 的三维头部姿态估计

好的鲁棒性、泛化能力以及坚实的数学基础是一个优秀计算机视觉算法的重要组成部分。所有这些特性在 Tim Cootes 所提出的主动外观模型 (Active Appearance Model, AAM) 均得到体现。本章将介绍如何使用 OpenCV 创建一个主动外观模型, 并介绍如何用 AAM 去搜索用户模型在给定帧中最可能出现的位置。此外, 还将介绍如何使用 POSIT 算法, 以及如何在有“姿态”的图像中找到合适的三维模型。可用这些工具来实时跟踪视频中三维模型, 这真是很不错的技术? 虽然本章只是以头部姿态作为例子, 但几乎任何可变形模型都可以使用这种方法。

本章将介绍如下主题:

- 主动外观模型概述;
- 主动形状模型概述;
- 与主动外观模型有关的模型实例化;
- 主动外观模型搜索与拟合;
- POSIT 算法。

下面对本章可能遇到的术语作相应的解释:

- 主动外观模型 (AAM): 该模型包含有形状和结构 (texture) 的统计信息。它是捕捉形状和结构变化的好方法。
- 主动形状模型 (ASM): 一个形状统计模型, 对学习形状变化很有用。
- 主成分分析 (PCA): 一种正交线变换, 它会将数据变换到新的坐标系中, 该坐标系中的第一个坐标 (称为第一主成分) 表示数据变化最大的方向, 第二个坐标表示数据

变化第二大的方向，这样以此类推。这个过程经常用于降维处理。在对原问题使用降维处理后，可使拟合（fitting）算法变得更快。

- Delaunay 三角剖分（DT）：三角剖分是指对于平面中的一个点集  $P$ ，任何三角形的外接圆都不包含任何其他顶点，这可避免出现瘦长三角形。结构映射需要三角剖分。
- 仿射变换：能表示为一个矩阵乘以向量，然后再加上一个向量，它用于结构映射。
- 迭代由正交投影和尺寸变换提取姿态（Pose from Orthography and Scaling with Iterations, POSIT）：一种用来进行三维姿态估计的计算机视觉算法。

## 7.1 主动外观模型概述

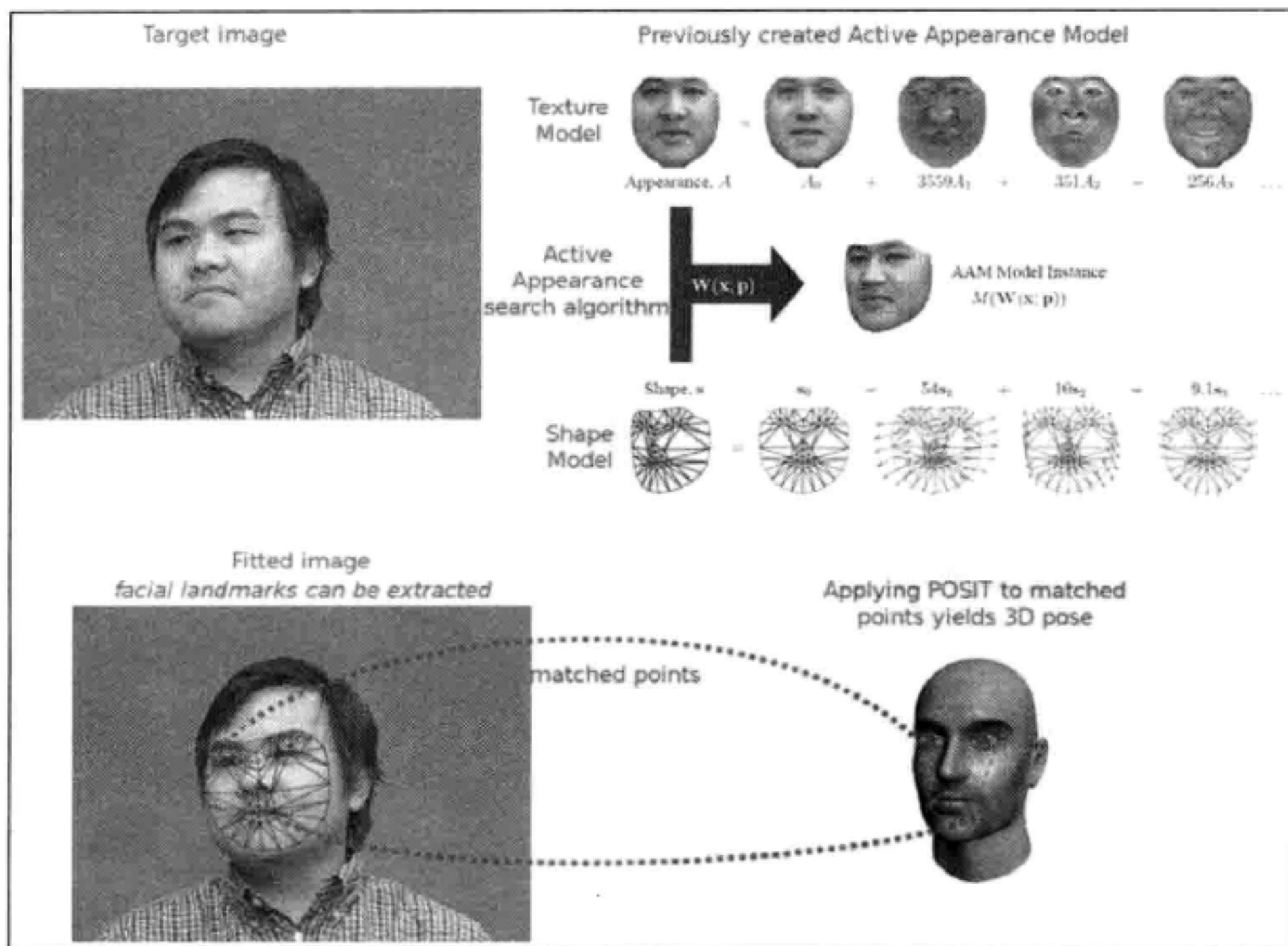
简单来说，主动外观模型是将形状与结构相结合的一种很不错的参数化模型，它很高效，能准确且清楚地知道模型在帧中的确切位置。从 7.2 节开始就会介绍该模型的实现，到时读者会发现它们与地标（landmark）位置紧切相关。下一节将进一步介绍主成分分析和一些实战经验（hands-on experience）。然后介绍 OpenCV 的 Delaunay 函数并由此来学习三角部分。在此基础上，会在三角化结构扭曲部分进一步使用分段仿射来得到对象的结构信息。

在读者拥有足够的技术来建立一个良好模型后，就可在模型实例化部分使用这些技术。在模型实例化部分可通过 AAM 的搜索与拟合来解决逆问题。对二维图像匹配甚至是三维图像匹配来讲这些技术本身是非常有用的算法，但在使用这些算法时，人们也许会问：为什么不将这些算法与 POSIT（Pose from Orthography and Scaling with Iterations）联系起来以得到更好的三维模型拟合算法呢？7.5.1 节会向读者介绍如何在 OpenCV 中使用 POSIT，7.5.2 节还将介绍如何将 POSIT 算法应用到头部检测模型中。通过这种方式，就可用一个三维模型来拟合已经匹配的二维帧。也许细心的读者想知道在什么地方可以使用这种方式？其实，只需在连续的视频帧中将 AAM 和 POSIT 结合起来，通过形变模型的检测就可得到实时的三维跟踪！7.5.3 节会介绍具体的实现细节。

一张图片包含了很多信息，如果有  $N$  张图，包含的信息量更大，用主动外观模型，前面提到的内容都能被很轻松地跟踪，如下图所示。

**本章算法概述：**给定一幅图像（在上图左上方的那幅图），使用基于主动外观模型的搜索算法来找到人脸头部的二维姿态。截图的右上方就是在搜索算法中使用已经训练后的主动外观模型得到的结果。在姿态找到后，POSIT 可用来将所得结果推广到三维姿态的情形。如果将该过程应用到视频序列中，通过检测就可进行三维跟踪。

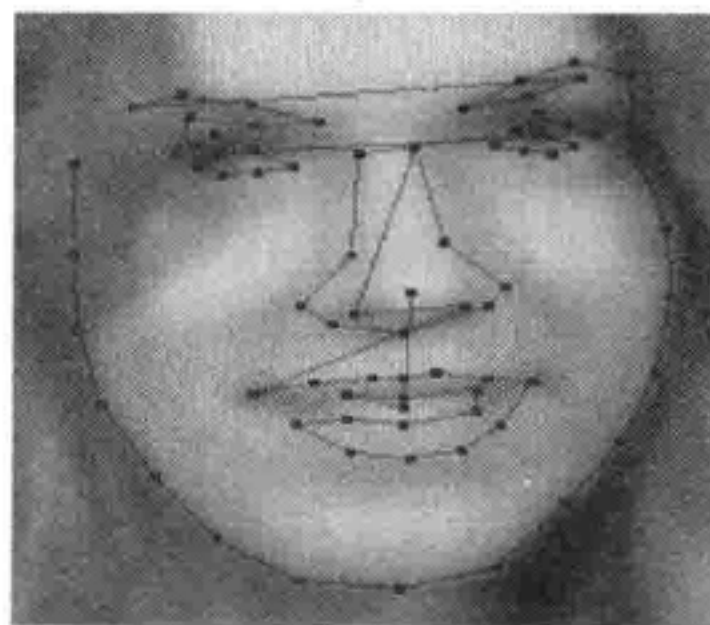




## 7.2 主动形状模型概述

如前所述，AAM 需要形状模型，这可通过主动形状模型（ASM）来得到。下一节将创建一个 ASM，它是基于形状变化的统计模型，由形状变化的组合得到。需要一个带标签的图像训练集，Timothy Cootes 的论文“*Active Shape Models – Their Training and Application*”有这方面的介绍。为了得到一个人脸形状模型，对人脸重要位置需用一些点来进行图像标注，以勾勒出面部的主要特征。下图就是这样的一个例子。

这幅图来自 MUCT 数据集，上面有 76 个标注点。这些点都是人为标注上去的，它们能勾勒出面部比较容易跟踪的特征，例如：嘴的轮廓、鼻子、眼睛、眉毛和脸的形状。



（附彩图）



**注意：**Procrustes 分析是一种基于统计形状的分析，它用来分析形状集合的分布。通过最佳地平移、旋转以及均匀缩放对象来实现 Procrustes 叠加（Procrustes superimposition）。

如果有前面提到的 MUCT 图像集, 就可生成一个基于统计的形状变化模型。由于通过对象上的标注点来描述其形状, 因此首先需在坐标系中使用普鲁克分析来对齐点集, 如果需要, 还可通过向量  $x$  来表示每个形状。然后对数据采用主成分分析 (PCA), 并用下面的公式来逼近任意样本。

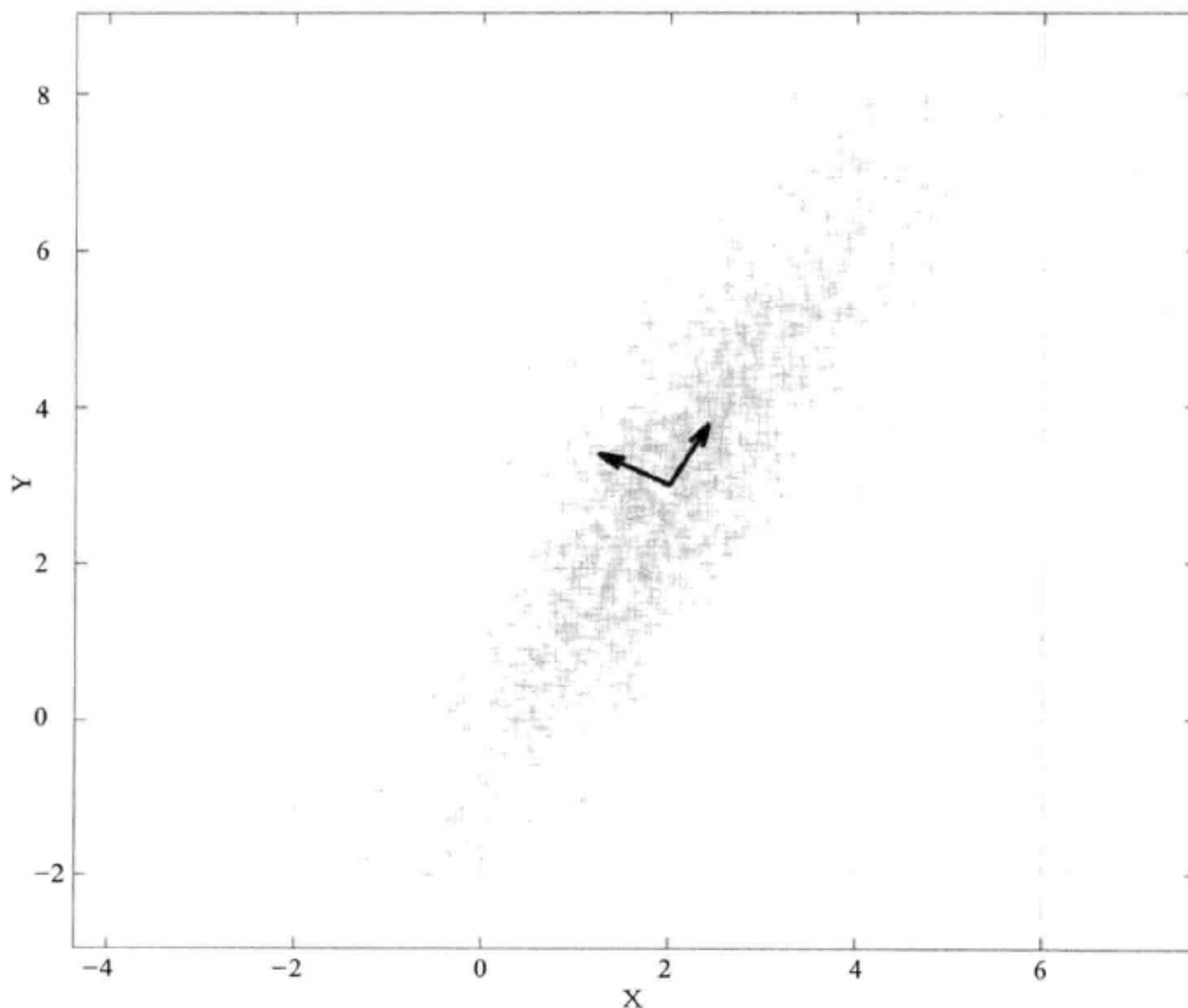
$$x = \bar{x} + P_s b_s$$

该公式中的  $\bar{x}$  为平均形状,  $P_s$  为变化的正交模型集,  $b_s$  为形状参数集。为了能更好地理解这个公式, 在本节剩下部分将创建一个简单应用来展示如何使用 PCA 和形状模型。

为什么非要使用 PCA 的呢? 因为 PCA 会减少模型参数数量, 这对应用非常有帮助。本章后面会对一幅给定图像进行搜索, 那时会看到这样做的好处。在一个网页 ([http://en.wikipedia.org/wiki/Principal\\_component\\_analysis](http://en.wikipedia.org/wiki/Principal_component_analysis)) 中给出了下面这段话来解释 PCA:

*PCA can supply the user with a lower-dimensional picture, a “shadow” of this object when viewed from its (in some sense) most informative viewpoint. This is done by using only the first few principal components so that the dimensionality of the transformed data is reduced.*

下图能清晰解释上面这段话:



这幅图来自：<http://en.wikipedia.org/wiki/File:GaussianScatterPCA.png>。

上图展示了一个以 (2, 3) 为中心的多变量高斯分布的 PCA。图中的向量是协方差矩阵的特征向量，平移这些向量，使它们的起始点在数据的平均值处。

这样，如果想用一个参数来表示模型，一个不错的选择是取指向上图右上角的特征向量的方向来表示数据。此外，若参数有点变化，也可通过推断数据来得到与所需数据相似的值。

### 7.2.1 感受 PCA

可通过主动形状模型和一些测试参数来理解 PCA 给人脸模型带来的益处。由于研究人脸检测和跟踪已经有一段时间了，在网上可找到一些用于研究的人脸数据库。本节将使用 IMM 数据库的几个样本。

首先需了解 OpenCV 的 PCA 类的工作原理。从帮助文档可得出这样的结论：PCA 类用于计算一组特殊的向量基，这些向量是协方差矩阵的特征向量，协方差矩阵通过对输入数据计算而得到。使用此类的 `project` 方法和 `backproject` 方法可在向量与子空间之间相互切换。只取协方差矩阵的前几个特征向量就能很精确地逼近这个新的坐标空间。这说明高维空间的向量可由低维空间的向量来表示，这些低维空间的向量由所投影的子空间坐标构成。

本应用只将少量的标量值作为参数，因此这个类的主要方法为 `backproject`。它只取投影向量的主成分坐标作为参数，并由此来重构原数据。如果取所有成分，就可得到与原数据一样的数据。但用主成分来重构数据与原数据差别很小，这就是为什么要使用 PCA 的一个原因。即便原数据发生了一些变化，但通过参数化标量就可以推断出原数据。

除此以外，PCA 类还能在向量与子空间之间相互切换。从数学角度讲，这意味着可计算从向量到子空间的投影，该子空间由协方差矩阵的主要特征值对应的特征向量构成。这方面的信息可查看相应的帮助文档。

可用显著的点来标注人脸图像，由此生成基于点分布模型 (Point Distribution Model, PDM) 的训练集。如果在二维空间上有  $K$  个标注点，则其形状可描述为

$$X = \{ x_1, y_1, x_2, y_2, \dots, x_k, y_k \}$$

需要注意的是，要让所有图像样本的标签保持一致。例如，在左图中，嘴右边的标注编号为 3，则其他所有图像在这个位置的标注编号都应为 3。

这一系列的标注点将会得到形状的轮廓，可用向量来定义一个给定的形状。通常假定这些标注点在该空间分布为高斯分布，并可用 PCA 来计算由所有训练形状所构成的协方差矩阵的归一化特征向量和特征值。使用排名靠前的特征值所对应的特征向量来创建一个名为  $P$  的矩阵，它的维数为  $2k \times m$ 。这样，这些特征向量表示数据集变化的主要方向。

现在通过下面的等式来定义新的形状：

$$X' = X' + Pb$$



其中,  $X'$  是所有训练形状的平均形状, 这里只需平均每个标注点即可,  $b$  是一个向量, 它的每个元素表示对每个主成分向量的缩放。通过修改向量  $b$ , 就可得到新的形状。 $b$  的取值通过在三倍标准差范围内变化, 这样得到的形状会在训练集中。

下图是对三个不同图像的嘴巴进行标注后的情形。



(附彩图)

从上图可看出, 该形状可由标注点的序列表示。可使用 GIMP 或 ImageJ 这样的软件, 也可用 OpenCV 来创建一个用于标注训练图像的简单程序。假定用户已经完成这个标注过程, 并对所有训练图像都要按顺序保存这些点的  $x$  坐标和  $y$  坐标到文本文件中, 这些信息将被用于 PCA 分析。然后增加两个参数到该文本文件的第一行, 它们分别是训练图像的数量和读取的列数。因此, 对于  $k$  个二维点, 要保存的坐标个数为  $2*k$ 。

下面是一个此类文本文件的内容, 该文件保存了 IMM 数据集中三个图像的标注信息, 其中  $k$  等于 5:

```
3 10
265 311 303 321 337 310 302 298 265 311
255 315 305 337 346 316 305 309 255 315
262 316 303 342 332 315 298 299 262 316
```

现在已经标注完图像, 接下来需将这些数据转变成形状模型。首先, 加载数据到一个矩阵中, 这将通过 loadPCA 函数来实现。下面这段代码是 loadPCA 函数的用法:

```
PCA loadPCA(char* fileName, int& rows, int& cols, Mat& pcaset){
    FILE* in = fopen(fileName, "r");
    int a;
    fscanf(in, "%d%d", &rows, &cols);

    pcaset = Mat::eye(rows, cols, CV_64F);
    int i, j;

    for(i=0; i<rows; i++){
        for(j=0; j<cols; j++){
            fscanf(in, "%d", &a);
            pcaset.at<double>(i, j) = a;
        }
    }

    PCA pca(pcaset, // pass the data
            Mat(), // we do not have a pre-computed mean vector,
```

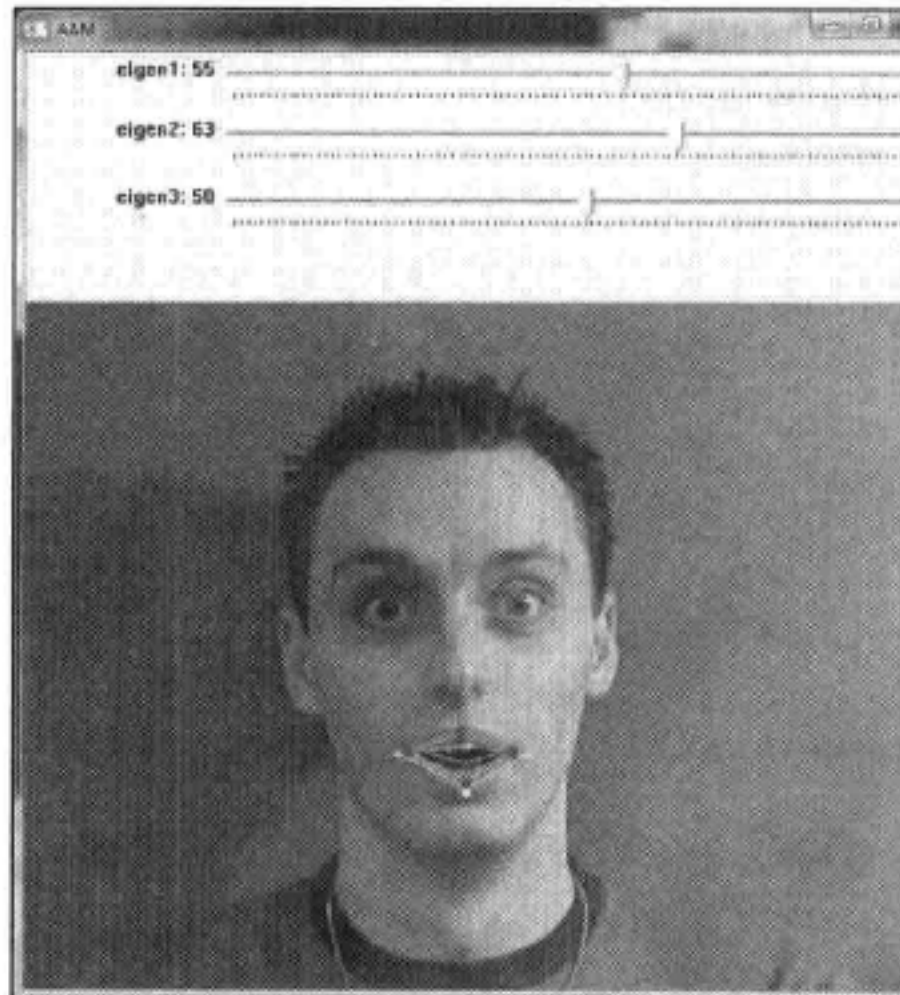
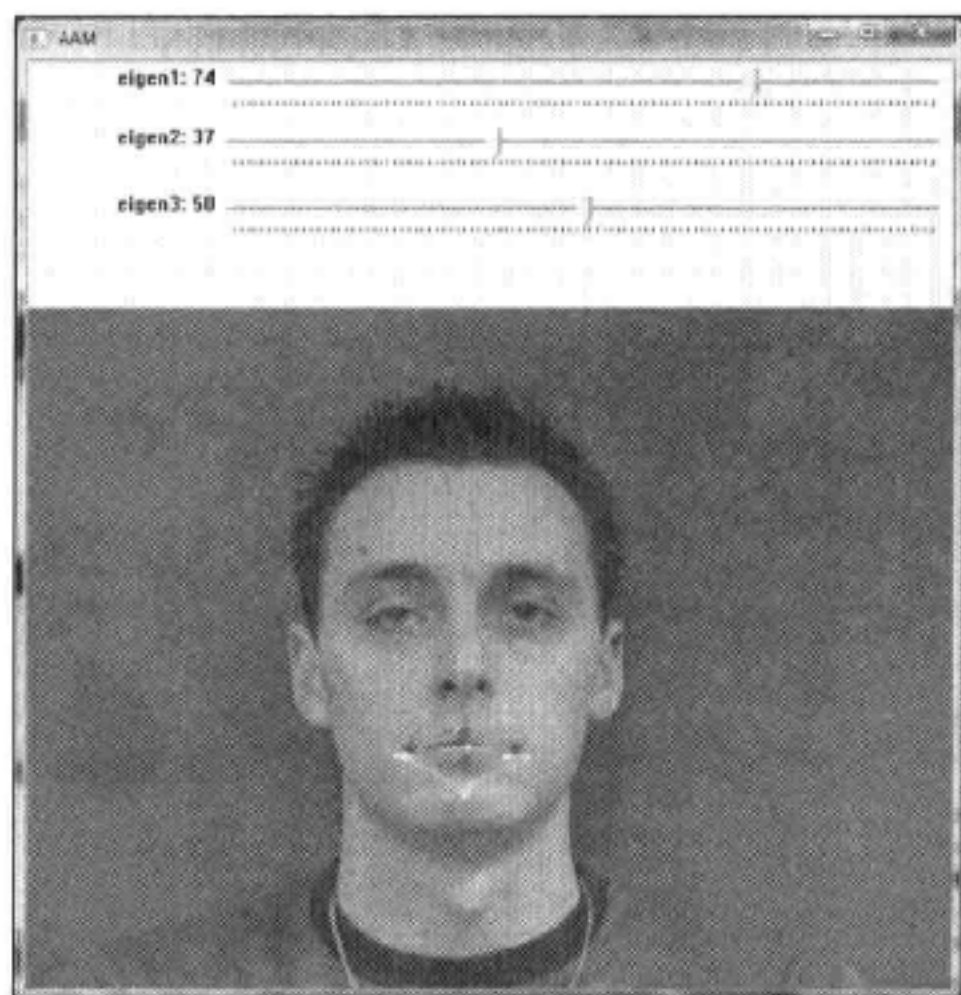
```

// so let the PCA engine compute it
CV_PCA_DATA_AS_ROW, // indicate that the vectors
// are stored as matrix rows
// (use CV_PCA_DATA_AS_COL if the vectors are
// the matrix columns)
pcaset.cols// specify, how many principal components to retain
);
return pca;
}

```

注意该矩阵由这一行代码创建：`pcaset = Mat::eye ( rows,cols,CV_64F)`，这样做会有足够的空间分配给  $2*k$  个值。在两个 for 循环加载数据到矩阵后，会调用 PCA 的构造函数，这需要将加载的数据和一个空矩阵传给该构造函数。该空矩阵用来存放由构造函数所计算的均值向量，这样，该向量只需计算一次。这里还需要说明一点：这些向量将作为矩阵的行被存储，虽然本应用只会使用很少的主成分向量，但还是将给定的行数作为主成分向量的数。

现在，已用训练集完成 PCA 对象的初始化，接下来需要按给定参数来反射投影 (`backProject`) 这些形状，即调用 `PCA.backproject` 函数，第一个参数为输入参数，它是一个行向量，第二个参数用来保存反射投影结果。



(附彩图)

上面这两张屏幕截图展示两种形状结构的差异，产生这种差异是由于滑动条所选择的参数值不同而造成的。黄色和绿色形状表示训练数据，而红色则是根据所选参数生成的形状。

示例程序可让用户为模型选择不同参数，因此可用于主动形状模型实验。需要注意的是，通过滑动条仅能改变前两个标量值（即改变相应的前两个模型），这会得到一个非常接近于训练时的形状。当用 AAM 来搜索一个模型时，这种变化很有用，因为它提供了插值形状 (*interpolated shapes*)。在下一节将讨论三角剖分、结构、AAM、AAM- 搜索。

## 7.2.2 三角剖分

由于所寻找的形状可能会被扭曲,例如,有一张人脸图像的嘴是张开的,因此,需将这些结构映射成一个平均形状,然后对这些归一化结构使用PCA进行处理。为了做到这点,需要用三角剖分。三角剖分的概念很简单,创建一个包含所有标注点的三角形,然后从一个三角形映射到另一个三角形。OpenCV自带了一个名为`cvCreateSubdivDelaunay2D`的函数,它可很方便地创建一个空的Delaunay三角剖分,用户只需考虑如何避免出现瘦长三角形即可。



**注意:** 在数学和计算几何上,对一个平面上的点集 $P$ 和由 $P$ 所构成的三角形集合 $DT(P)$ ,其Delaunay三角剖分的定义为没有 $P$ 中的点在 $DT(P)$ 的任意一个三角形的外接圆里面。Delaunay三角剖分使所有三角形中最小的角最大化,这样会力图避免出现瘦长三角形。该三角剖分以Boris Delaunay在1934年的开创性工作而命名。

在初始化Delaunay划分后,可使用`cvSubdivDelaunay2DInsert`函数将点插入该划分中,下面的代码清楚解释了三角剖分的原理:

```
CvMemStorage* storage;
CvSubdiv2D* subdiv;
CvRect rect = { 0, 0, 640, 480 };

storage = cvCreateMemStorage(0);
subdiv = cvCreateSubdivDelaunay2D(rect, storage);

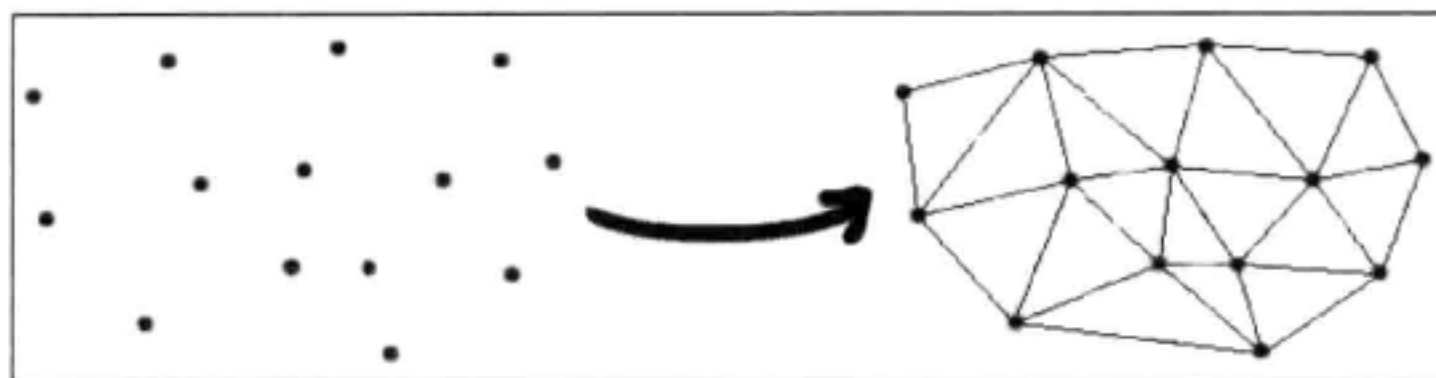
std::vector<CvPoint> points;

//initialize points somehow
...

//iterate through points inserting them in the subdivision
for(int i=0;i<points.size();i++){
    float x = points.at(i).x;
    float y = points.at(i).y;
    CvPoint2D32f floatingPoint = cvPoint2D32f(x, y);
    cvSubdivDelaunay2DInsert( subdiv, floatingPoint );
}
```

注意,这些点都会落在矩形框里面,作为参数传给`cvCreateSubdivDelaunay2D`函数的那个矩形。为了创建一个划分,还需要创建并初始化一片内存结构,这由上面代码的前五行完成,为了创建一个三角剖分,需要用`cvSubdivDelaunay2DInsert`函数来插入点,在上面代码的for循环中就实现了该功能。注意,这些点需要初始化,因为它们通常用作输入参数。下图显示了三角剖分可能的样子。





该图是上面代码产生的结果，即给定一个点集，用 Delaunay 算法来得到了一个三角剖分。

虽然用 OpenCV 函数创建划分很方便，但不可能通过简单遍历就能访问完所有三角形。下面的代码介绍如何通过划分的边来遍历所有三角形：

```
void iterate(CvSubdiv2D* subdiv, CvNextEdgeType triangleDirection){
    CvSeqReader reader;
    CvPoint buf[3];

    int i, j, total = subdiv->edges->total;
    int elem_size = subdiv->edges->elem_size;
    cvStartReadSeq((CvSeq*)(subdiv->edges), &reader, 0);

    for(i = 0; i < total; i++){
        CvQuadEdge2D* edge = (CvQuadEdge2D*)(reader.ptr);

        if(CV_IS_SET_ELEM(edge)){

            CvSubdiv2DEdge t = (CvSubdiv2DEdge)edge;

            for(j=0;j<3;j++){

                CvSubdiv2DPoint* pt = cvSubdiv2DEdgeOrg(t);
                if(!pt) break;
                buf[j] = cvPoint(cvRound(pt->pt.x), cvRound(pt->pt.y));
                t = cvSubdiv2DGetEdge(t, triangleDirection);
            }
        }
        CV_NEXT_SEQ_ELEM(elem_size, reader);
    }
}
```

给定一个划分，调用 `cvStartReadSeq` 函数来初始化其边读取器（edge reader）。在 OpenCV 的帮助文档中，有如下的说明：

*The function initializes the reader state. After that, all the sequence elements from the first one down to the last one can be read by subsequent calls of the macro `CV_READ_SEQ_ELEM(read_elem, reader)` in the case of forward reading and by using `CV_REV_READ_SEQ_ELEM(read_elem, reader)` in the case of reverse reading. Both macros put the sequence element to `read_elem` and move the reading pointer toward the next element.*

得到下一个元素的其中一种方法是调用宏 `CV_NEXT_SEQ_ELEM (elem_size, reader)`，如果序列元素很大，首选此方法。在这种情况下，可使用 `CvQuadEdge2D*` `edge = (CvQuadEdge2D*)(reader.ptr)` 来访问边，这只需将读取器指针转换成类型为 `CvQuadEdge2D` 的指针。宏 `CV_IS_SET_ELEM` 仅检查指定的边是否存在。给定一条边，可调用 `cvSubdiv2DEdgeOrg` 函数来得到其端点。为了让程序始终处理三角形，需要不断调用 `cvSubdiv2DGetEdge` 函数，并将三角形的方向传递给该函数，这些方向包括：`CV_NEXT_AROUND_LEFT` 和 `CV_NEXT_AROUND_RIGHT`。

### 7.2.3 扭曲三角化结构

现在可遍历划分的三角形了，然后可从原始标注图像中生成一个扭曲的三角形。这对映射原始结构到扭曲形状很有用，下面这段代码将展示该过程：

```
void warpTextureFromTriangle(Point2f srcTri[3], Mat originalImage,
                             Point2f dstTri[3], Mat warp_final){

    Mat warp_mat(2, 3, CV_32FC1);
    Mat warp_dst, warp_mask;
    CvPoint trianglePoints[3];
    trianglePoints[0] = dstTri[0];
    trianglePoints[1] = dstTri[1];
    trianglePoints[2] = dstTri[2];
    warp_dst = Mat::zeros(originalImage.rows, originalImage.cols,
originalImage.type());
    warp_mask = Mat::zeros(originalImage.rows, originalImage.cols,
originalImage.type());

    /// Get the Affine Transform
    warp_mat = getAffineTransform(srcTri, dstTri);

    /// Apply the Affine Transform to the src image
    warpAffine(originalImage, warp_dst, warp_mat, warp_dst.size());
    cvFillConvexPoly(new IplImage(warp_mask), trianglePoints, 3, CV_
RGB(255,255,255), CV_AA, 0);
    warp_dst.copyTo(warp_final, warp_mask);
}
```

上面的代码假设数组 `srcTri` 保存了三角形的顶点，而目标三角形的顶点保存在 `dstTri` 数组中。 $2 \times 3$  的 `warp_mat` 矩阵用于从原三角形到目的三角形的仿射变换中。更多信息可参考 OpenCV 关于函数 `cvGetAffineTransform` 的帮助文档。

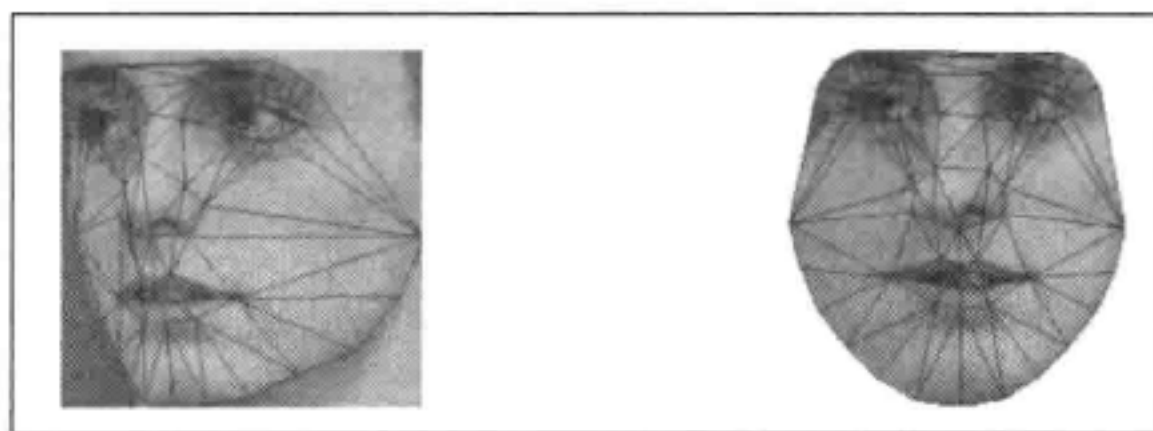
函数 `cvGetAffineTransform` 可通过计算得到一个仿射变换矩阵，计算公式如下：

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \text{mapMatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

在上面这个公式中，第  $i$  个目标值为  $(x'_i, y'_i)$ ，第  $i$  个原始值为  $(x_i, y_i)$ ，其中  $i$  为 0,1,2。

在得到该仿射矩阵后，可将仿射变换用到原图像中，此过程由 `warpAffine` 函数来完成。由于并不需要对整个图像做这样的处理，只需重点处理由掩码 (mask) 得到的那部分三角形即可。掩码可由函数 `cvFillConvexPoly` 得到，最后一行通过创建的掩码从原始图像复制出要处理的三角形。

下图是在每个标注图像中对每个三角形应用该过程而得到的结果。注意，三角形需映射回对应帧 (alignment frame) 中，以便人脸正向朝着读者。该过程用于创建 AAM 的统计结构。



上图是扭曲左边那幅图中所有被映射的三角形为参考帧 (reference frame) 的结果。

### 7.3 模型实例化——试试主动外观模型

AAM 的优势之一是它能很容易地对用于训练图像的模型进行插值，从而通过调整几个形状或模型参数来得到惊人的表现能力。当改变形状参数时，扭曲结果会随训练好的形状数据变化；另一方面，修改外观参数，基本形状的结构也会被修改。扭曲变换将每个三角形从基本形状变换为修改后的形状，因此可在张开嘴的顶部合成一个闭合的嘴，如下图所示：



上图通过在另一幅图像上方执行主动外观模型实例化来得到一个人工合成的紧闭嘴形。该方法也可用来将微笑的嘴合成到喜欢的脸上，以拓展训练图像。

AAM 的目标是只需要对形状和结构改变三个参数就可得到上图的效果。这个示例项目已经开发出来了，为了尝试一下 AAM，读者可从 <http://www.packtpub.com/> 网站上下载该



项目。实例化一个新的模式只需拖动滑动条来改变参数，这与7.2.1节一样。值得注意的是，AAM的搜索和拟合会通过这种灵活的方法来寻找给定模型在图像帧中的最佳匹配，其位置与训练时不一样。下面将讨论此内容。

## 7.4 主动外观模型搜索和拟合

随着对形状和结构模型进行全新组合，会找到一种很好的方式来描述人脸在形状和外观上可能的变化。现在需要分别找出什么样的形状的参数集  $p$  和外观的参数集能使模型尽可能逼近给定的输入  $I(x)$ 。这自然需要计算实例化模型与给定输入图像在  $I(x)$  坐标系中的误差，或将点映射回基本 (base) 外观并计算它们之间的差值。本节将使用后面这种方法。因此，需要最小化下面的函数：

$$\sum_{x \in S_0} \left[ A_0(x) + \sum_{i=1}^m \lambda_i A_i(x) - I(W(x; p)) \right]^2$$

在上面这个式子中， $S_0$  表示  $x$  的像素集，它等于在 AAM 基本网格 (base mesh) 内的  $(x, y) T$ ； $A_0(x)$  是基本网格的结构； $A_i(x)$  是来自 PCA 的外观图像； $W(x; p)$  表示将像素从输入图像映射回基本网格帧 (base mesh frame) 的操作。

可用最近几年研究的方法来求解此目标函数的最小值。第一种方法是迭代累加法。该方法的  $\Delta p_i$  和  $\Delta \lambda_i$  由误差图像的线性函数计算得到，即在第  $i$  次迭代中，形状参数  $p$  和外观参数  $\lambda$  通过  $p_i \leftarrow p_i + \Delta p_i$  和  $\lambda_i \leftarrow \lambda_i + \Delta \lambda_i$  得到。这种迭代有时会收敛，有时不会收敛，不收敛可能的原因为增量 (delta) 不会依赖当前参数。另一种方法是梯度下降法，它非常慢。因此需要去寻找另一种收敛算法。Ian Mathews 和 Simon Baker 在他们的论文 “Active Appearance Models Revisited” 中提出了一种组合方法 (compositional approach)，该方法不更新参数，而是更新整个扭曲操作过程。该方法的详细实现可在论文中找到。这种方法对拟合做出了重要贡献，但在预先计算的步骤中有很大的计算量。整个算法实现过程如下：

Pre-compute:

- (3) Evaluate the gradient  $\nabla A_0$  of the template  $A_0(x)$
- (4) Evaluate the Jacobian  $\frac{\partial w}{\partial p}$  at  $(x; 0)$
- (5) Compute the modified steepest descent images using Equation (41)
- (6) Compute the Hessian matrix using modified steepest descent images

Iterate:

- (1) Warp  $I$  with  $W(x; p)$  to compute  $I(W(x; P))$
- (2) Compute the error image  $I(W(x; P)) - A_0(x)$
- (7) Compute dot product of modified steepest descent images with error image
- (8) Compute  $\Delta p$  by multiplying by inverse Hessian
- (9) Update the warp  $W(x; P) \leftarrow W(x; P) \circ W(x; \Delta P)^{-1}$

Post-computation:

- (10) Compute  $\lambda_i$  using Equation (40). [Optional step]

注意，第（9）步是一个组合更新（见上图），方程（40）和（41）如下所示：

$$\lambda_i = \sum_{x \in S_0} A_i(x) \cdot [I(W(x;p)) - A_0(x)] \quad (40)$$

$$SD_j(x) = \nabla A_0 \frac{\partial W}{\partial p_j} - \sum_{i=1}^m \left[ \sum_{x \in S_0} A_i(x) \cdot \nabla A_0 \frac{\partial W}{\partial p_j} \right] A_i(x) \quad (41)$$

若该算法从最终结果附近的一个点开始，则在绝大多数时候都会收敛，但若是旋转、平移或缩放的变化较大时，就不会收敛。为了得到更多的信息来使该算法收敛，可参数化一个全局的二维相似变换。下面这个等式是那篇论文的第42个方程：

$$N(x;q) = \begin{pmatrix} (1+a) & -b \\ b & (1+a) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

在上述等式中，这四个参数  $q=(a,b,t_x,t_y)^T$  会按如下方法插值：第一对参数  $(a,b)$  与缩放  $k$  和旋转  $\theta$  有关，其中， $a = k\cos\theta - 1$ ， $b = k\sin\theta$ ；第二对参数  $(t_x,t_y)$  是  $x$  和  $y$  的平移，这在论文“*Active Appearance Models Revisited*”对它们进行了介绍。

再多用一点数学变换，就能使用上面的算法来找到有全局二维变换的最佳图像拟合。

这类组合算法有几个性能优势。本项目将使用上面那篇论文所介绍的一种组合算法，即逆向组合推算算法。注意，该方法在拟合过程中预先计算或推算出外观变化的影响，从而提高 AAM 拟合性能。

下图是对不同图像使用逆向组合推算算法收敛后的情形，这些图像来自 MUCT 数据集。



上图展示了在人脸数据集上使用逆向组合推算算法收敛后得到的结果。

## 7.5 POSIT 算法

在找到标注点的二维位置后，就可用 POSIT 算法来得到模型的三维姿态。三维对象的姿态  $P$  可由  $3 \times 3$  的旋转矩阵  $R$  和平移向量  $T$  来得到，即  $P = [R | T]$ 。



**注意：**这部分的大多数内容都是来源于 Javier Barandiaran 所写的 OpenCV POSIT 教程。

POSIT 的含义是反复迭代由正交投影和尺寸变换提取姿态 (Pose from Orthography and Scaling, POS) 算法, 所以它是有迭代的 POS 算法的缩写。运行该算法需要的条件是可检测和匹配图像中 4 个或更多不共面的对象特征点, 且需知道这些点在对象上的相对几何形状。

算法的主要思想是对物体的姿态找到一个好的近似, 可假定所有模型点都在同一平面, 因为比起摄像机到人脸的距离, 它们彼此的深度相差不是很大。在得到初始姿态后, 物体的旋转矩阵和平移向量可通过求解线性方程来得到; 然后通过迭代近似姿态来更好地计算缩放的正交投影, 随后将 POS 算法应用到该投影结果, 而不是原来的那些姿态上。更多的信息, 可参考 DeMenton 的论文 “*Model-Based Object Pose in 25 Lines of Code*”。

### 7.5.1 深入理解 POSIT 算法

为了让 POSIT 算法工作, 需要至少 4 个非共面的三维模型点, 以及二维图像中各自的匹配。需要为 POSIT 的迭代设置结束条件, 一般来说, 结束条件是迭代次数或距离参数。然后调用 cvPOSIT 函数, 该函数可得到旋转矩阵和平移向量。

用 Javier Barandiaran 的教程来作为一个 POSIT 算法的例子, 该例子用 POSIT 算法来得到一个立方体的姿态。这个模型由四个点创建, 其初始化代码如下:

```
float cubeSize = 10.0;
std::vector<CvPoint3D32f> modelPoints;
modelPoints.push_back(cvPoint3D32f(0.0f, 0.0f, 0.0f));
modelPoints.push_back(cvPoint3D32f(0.0f, 0.0f, cubeSize));
modelPoints.push_back(cvPoint3D32f(cubeSize, 0.0f, 0.0f));
modelPoints.push_back(cvPoint3D32f(0.0f, cubeSize, 0.0f));
CvPOSITObject *positObject = cvCreatePOSITObject( &modelPoints[0],
static_cast<int>(modelPoints.size()) );
```

注意, 此模型由 cvCreatePOSITObject 方法创建, 它会返回一个 CvPOSITObject 方法, 该方法会用于 cvPOSIT 函数中。另外, 姿态会参照第一个模型点来计算, 这是将其放到原点的好方法。

需要将二维图像保存到另一个向量中。但注意, 放到数组中这些点必须要与模型点的插入顺序一致, 即第  $i$  个点要与第  $i$  个三维模型点匹配。另外, 二维图像点的原点位于图像中心, 这可能需要平移它们。可插入以下的二维图像点 (当然, 它们会随用户的匹配而发生变化):

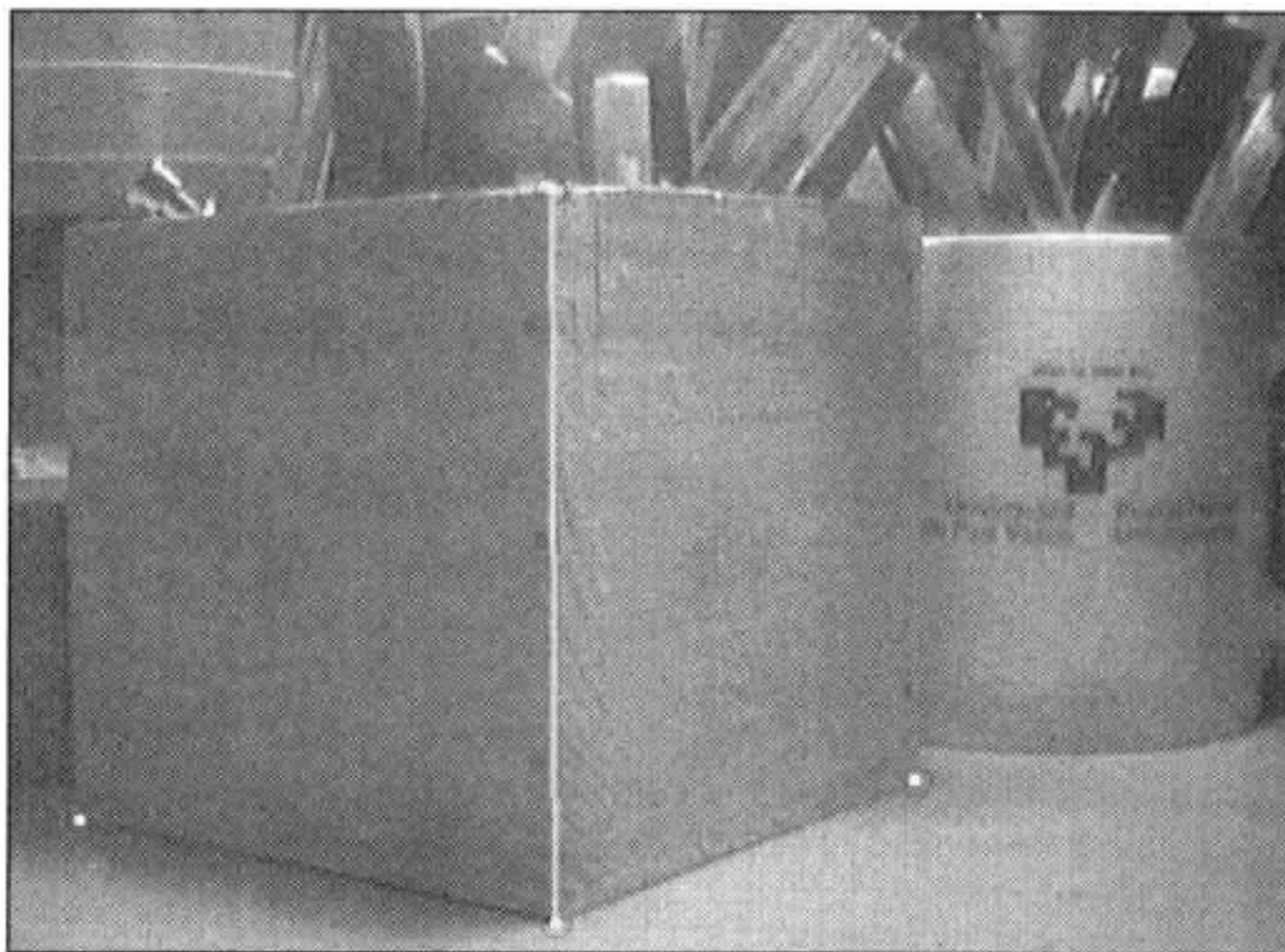
```
std::vector<CvPoint2D32f> srcImagePoints;
srcImagePoints.push_back( cvPoint2D32f( -48, -224 ) );
srcImagePoints.push_back( cvPoint2D32f( -287, -174 ) );
srcImagePoints.push_back( cvPoint2D32f( 132, -153 ) );
srcImagePoints.push_back( cvPoint2D32f( -52, 149 ) );
```

现在只需为矩阵分配内存并设置迭代的结束条件, 然后调用 cvPOSIT, 其体实现如下所示:



```
//Estimate the pose
CvMatr32f rotation_matrix = new float[9];
CvVect32f translation_vector = new float[3];
CvTermCriteria criteria = cvTermCriteria(CV_TERMCRIT_EPS | CV_
TERMCRIT_ITER, 100, 1.0e-4f);
cvPOSIT( positObject, &srcImagePoints[0], FOCAL_LENGTH, criteria,
rotation_matrix, translation_vector );
```

在迭代完成后，cvPOSIT 将存储结果到 rotation\_matrix 和 translation\_vector 中。下图的白色圈表示插入的 srcImagePoints，而坐标轴表示旋转和平移：



(附彩图)

从上图可看到输入的点 and 运行 POSIT 算法后的结果：

- ❑ 白色圆圈表示输入点，而坐标轴为得到的模型姿态。
- ❑ 要注意所使用摄像机的焦距，它可通过标定过程来得到，看看 2.4.1 节复习一下得到有效的标定过程。本节实现的 POSIT 算法只允许方块像素，所以在 x 轴和 y 轴上其焦距不会有空间。

❑ 所需的旋转矩阵有如下格式：

```
[rot[0]  rot[1]  rot[2]]
[rot[3]  rot[4]  rot[5]]
[rot[6]  rot[7]  rot[8]]
```

❑ 平移向量有如下格式：

```
[trans[0]]
[trans[1]]
[trans[2]]
```

## 7.5.2 POSIT 与头部模型

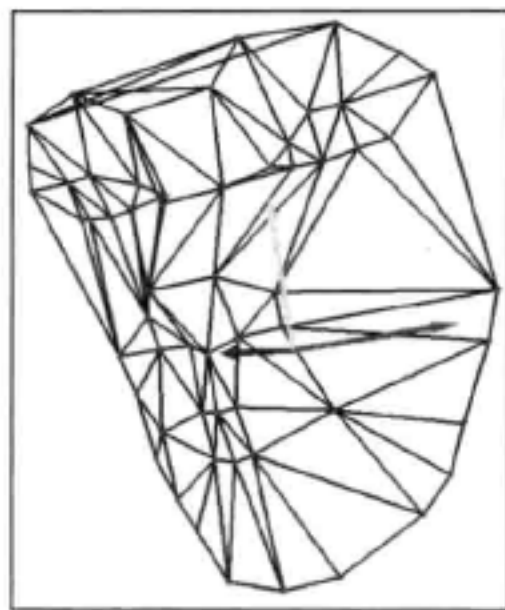
为了用 POSIT 算法来得到头部姿态，需要一个三维头部模型。能在科英布拉大学的系统与机器人研究所找到一个三维头部模型，下载地址为：<http://aifi.isr.uc.pt/Downloads/OpenGL/glAnthropometric3DModel.cpp>。可从文件的以下数组中获取此模型：

```
float Model3D[58][3] = {{-7.308957, 0.913869, 0.000000}, ...
```

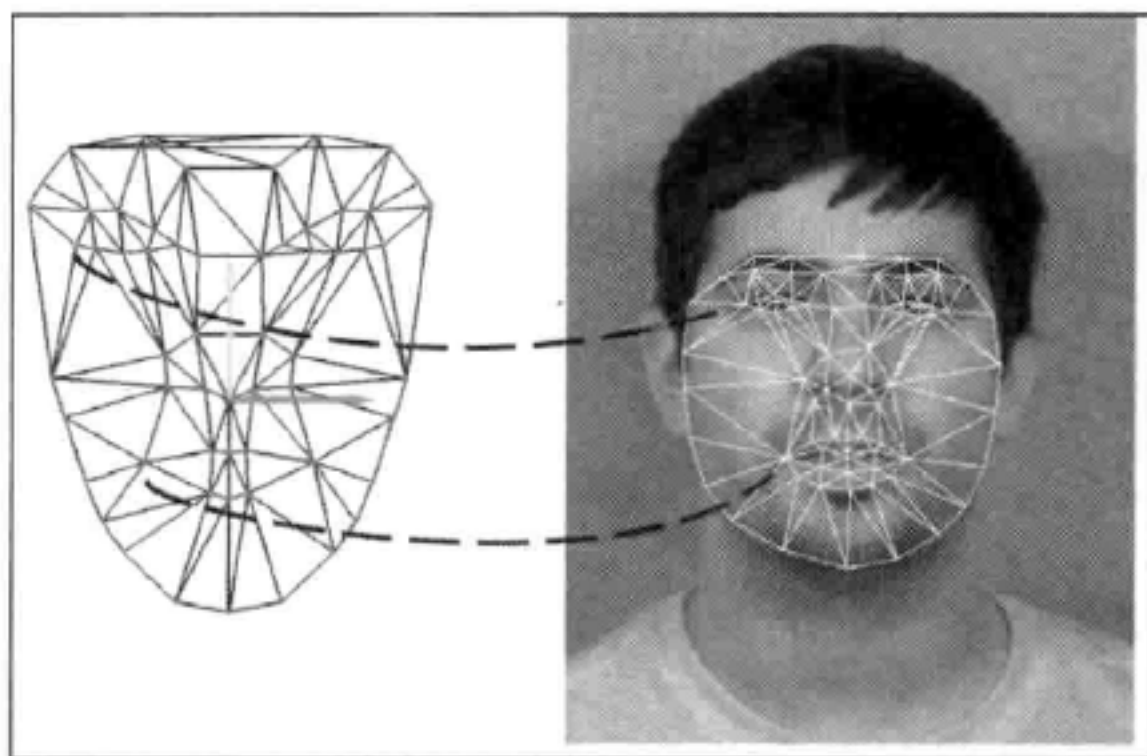
该模型看上去如右图所示。

上图为有 58 个点的三维头部模型，可用于 POSIT 算法。

为了让 POSIT 算法能正常工作，必须对三维头部模型的相关点进行匹配。另外还需要有至少 4 个非共面的三维点及其相应的二维投影才能使 POSIT 算法工作。这些信息都必须作为参数传递，这基本上与 7.5.1 节所介绍的一样。该算法的复杂度与匹配点的数量呈线性关系。右图显示如何进行点匹配：



(附彩图)



(附彩图)

上图是三维头部模型与 AMM 网络之间的正确匹配。

## 7.5.3 对摄像机或视频文件进行跟踪

现在所有工具都已经配置好了，以便进行 6 个自由度 (degree of freedom) 的头部跟踪，可在摄像机的视频流或视频文件中使用该跟踪算法。OpenCV 提供了 VideoCapture 类，它可按如下方式来使用：

```
#include "cv.h"
#include "highgui.h"

using namespace cv;

int main(int, char**)
```

```

{
    VideoCapture cap(0); // opens the default camera, could use a
                        // video file path instead

    if(!cap.isOpened()) // check if we succeeded
        return -1;

    AAM aam = loadPreviouslyTrainedAAM();
    HeadModel headModel = load3DHeadModel();
    Mapping mapping = mapAAMLandmarksToHeadModel();

    Pose2D pose = detectFacePosition();

    while(1)
    {
        Mat frame;
        cap >> frame; // get a new frame from camera

        Pose2D new2DPose = performAAMSearch(pose, aam);
        Pose3D new3DPose = applyPOSIT(new2DPose, headModel, mapping);

        if(waitKey(30) >= 0) break;
    }

    // the camera will be deinitialized automatically in VideoCapture
    // destructor
    return 0;
}

```

整个算法的工作过程为通过 VideoCapture. cap (0) 来初始化摄像机，参数 0 表示使用默认的摄像机。这一步可使摄像机工作起来；接着需要加载训练好的主动外观模型，这由函数 loadPreviouslyTrainedAAM 来完成；还需要为 POSIT 算法加载三维头部模型和三维头部的标注点映射到变量 mapping 中。

在加载完所需数据后，需要用一个已知的姿态来初始化算法，已知的姿态是指已知三维位置、旋转的情况以及 AAM 参数集。这可通过 OpneCV 关于 Haar 特征分类器的人脸检测文档来自动得到这些信息（更详细的信息可参考 6.5 节或 OpenCV 级联分类器的帮助文档），也可用前面标注的帧来手工初始化姿态；还可用一种暴力方式（brute-force approach），即对每个矩形执行一个 AAM 拟合，在第一帧中这样做会变得很慢。这样的初始化意味着要通过相应的参数找到 AAM 的二维标注点（landmark）。

当完成所有的加载，迭代在 while 循环中进行。迭代首先获取一个新的帧，然后通过主动外观模型拟合来找到这帧的标注点。这一步的当前位置是非常重要的，因为会将其作为参数传递给函数 performAAMSearch (pose,aam)。若图像误差变得很小，则说明当前标注点位置已找到，紧接着就会获取下一个标注点位置，最后将这些位置传给 POSIT 算法，该算



法由函数 `applyPOSIT (new2DPose, headModel, mapping)` 来实现，将新的二维姿态、前面加载的 `headModel` 以及 `mapping` 变量分别作为参数传递给此函数。这样，就可在获取的姿态中渲染任意的三维模型，如，一个坐标轴或增强现实模型。由于有了标注点，更多有趣的效果可通过模型参数来获得，例如，张嘴或改变眉毛位置。

由于此过程要依赖前面的姿态来估计下一个姿态，这样会积累误差，并从头部位置偏离。一种解决方法是检查一个给定的图像误差的阈值，若每次超过此阈值，就重新初始化该过程。另外还有一个问题，在跟踪时由于可能发生抖动，需使用了滤波器。对每个平移和旋转坐标，会采用一个简单的均值滤波就能取得好的效果。

## 7.6 总结

本章讨论如何将主动外观模型与 POSIT 算法结合起来以得到三维头部姿态。对如何创建、训练以使用所给的 AAM 作了简单介绍。读者可以在任何其他领域，如医疗、影像或工业中使用这些基础知识。除了讨论 AAM 外，还介绍了 Delaunay 细分，并学习如何使用这个有趣的结构来得到三角网格。本章也讨论如何在三角部分时使用 OpenCV 函数来执行结构映射。另一个有趣主题是关于 AAM 拟合。虽然只介绍了逆向组合推算算法，但该算法却集成了多年的研究成果。

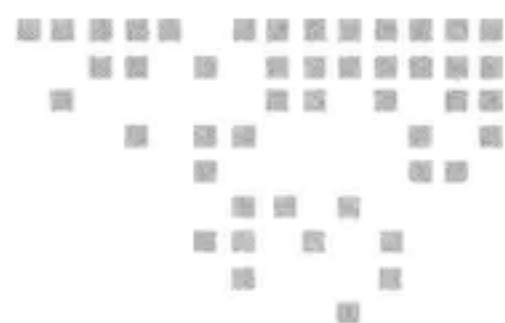
在充分了解 AAM 的理论和用法后，本章还详细讨论了 POSIT 的细节，讨论 POSIT 是为了将二维度量用到三维情形中，以介绍如何使用模型点的匹配来拟合三维模型。本章最后介绍如何在在线人脸跟踪器中通过检测来使用所有工具，并由此得到 6 个自由度的头部姿态，3 个旋转自由度和 3 个平移自由度。本章所有代码都可在网站 <http://www.packtpub.com/> 上下载。

## 7.7 参考文献

- *Active Appearance Models*, T.F. Cootes, G. J. Edwards, and C. J. Taylor, ECCV, 2:484–498, 1998 (<http://www.cs.cmu.edu/~efros/courses/AP06/Papers/cootes-eccv-98.pdf>)
- *Active Shape Models – Their Training and Application*, T.F. Cootes, C.J. Taylor, D.H. Cooper, and J. Graham, *Computer Vision and Image Understanding*, (61): 38–59, 1995 (<http://www.wiau.man.ac.uk/~bim/Papers/cviu95.pdf>)
- *The MUCT Landmarked Face Database*, S. Milborrow, J. Morkel, and F. Nicolls, *Pattern Recognition Association of South Africa*, 2010 (<http://www.milbo.org/muct/>)
- *The IMM Face Database – An Annotated Dataset of 240 Face Images*, Michael M. Nordstrom, Mads Larsen, Janusz Sierakowski, and Mikkel B. Stegmann,

*Informatics and Mathematical Modeling, Technical University of Denmark, 2004,*  
(<http://www2.imm.dtu.dk/~aam/datasets/datasets.html>)

- *Sur la sphère vide, B. Delaunay, Izvestia Akademii Nauk SSSR, Otdelenie Matematicheskikh i Estestvennykh Nauk, 7:793–800, 1934*
- *Active Appearance Models for Facial Expression Recognition and Monocular Head Pose Estimation Master Thesis, P. Martins, 2008*
- *Active Appearance Models Revisited, International Journal of Computer Vision, Vol. 60, No. 2, pp. 135 - 164, I. Mathews and S. Baker, November, 2004*  
([http://www.ri.cmu.edu/pub\\_files/pub4/matthews\\_iain\\_2004\\_2/matthews\\_iain\\_2004\\_2.pdf](http://www.ri.cmu.edu/pub_files/pub4/matthews_iain_2004_2/matthews_iain_2004_2.pdf))
- *POSIT Tutorial, Javier Barandiaran* (<http://opencv.willowgarage.com/wiki/Posit>)
- *Model-Based Object Pose in 25 Lines of Code, International Journal of Computer Vision, 15, pp. 123-141, Dementhon and L.S Davis, 1995* ([http://www.cfar.umd.edu/~daniel/daniel\\_papersfordownload/Pose25Lines.pdf](http://www.cfar.umd.edu/~daniel/daniel_papersfordownload/Pose25Lines.pdf))



## 基于特征脸或 Fisher 脸的人脸识别

本章将介绍人脸检测和人脸识别，并给出一个检测人脸的项目。人脸再次出现时，就能识别它们。人脸识别是一个流行但困难的话题，许多研究者多年来一直致力于人脸识别的研究。因此，本章将介绍人脸识别的简单方法，如果读者想要探索更复杂的方法，这是一个好的开始。

本章将介绍以下内容。

- 人脸检测
- 人脸预处理
- 从收集的人脸训练机器学习算法
- 人脸识别
- 收尾工作

### 8.1 人脸识别与人脸检测介绍

人脸识别是对已知人脸进行分类的过程。就像人看到人的脸时，就能识别他们是自己的家人、朋友或名人。有很多技术能让计算机学习识别人脸。人脸识别通常包括四个主要步骤。

1) 人脸检测：它是在图像中定位人脸区域的过程（在下图中心有一个大矩形）。这一步不关心人是谁，只关心是不是人脸。

2) 人脸预处理：这步是调整人脸图像，使其看起来更加清楚，且相似于其他人脸的过程（下图中心顶部那个灰色的小人脸就是人脸预处理得到的结果）。



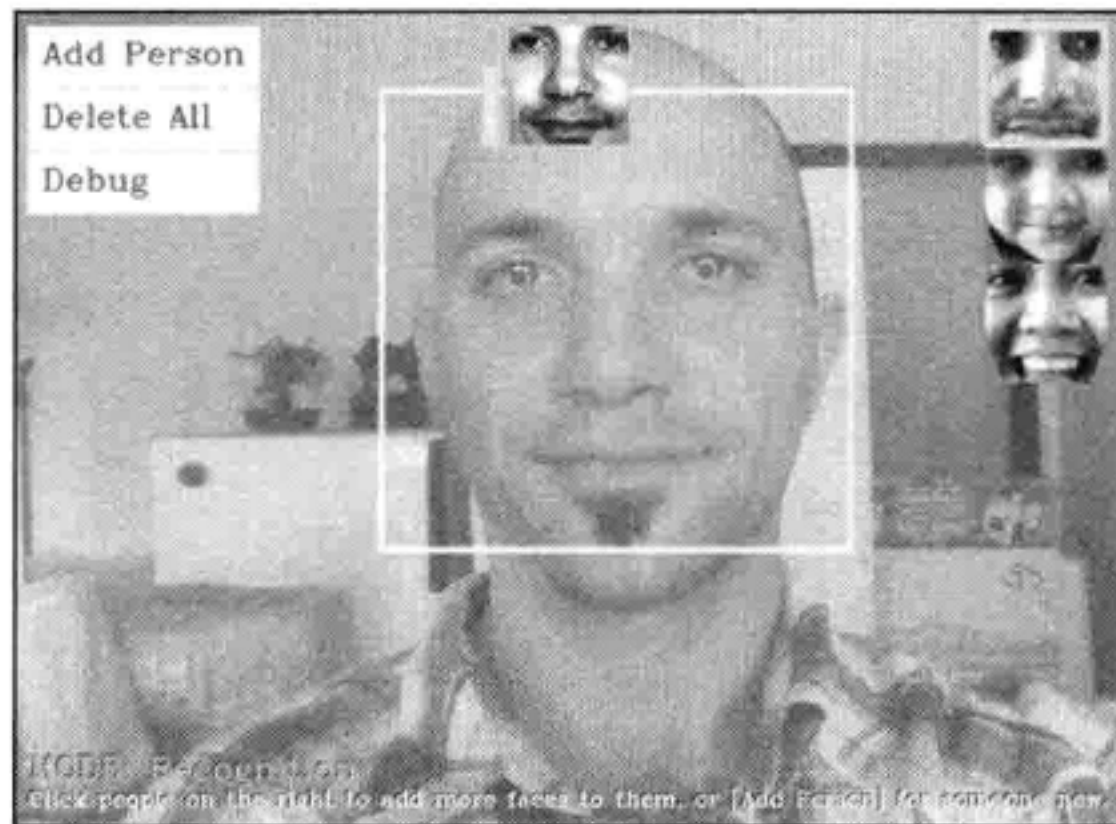
3) 收集和学习人脸：这步会收集许多被预处理过的人脸（要被识别的每个人），然后学习如何识别它们。

4) 人脸识别：在所收集的人脸中查找哪个人脸与摄像机中的人脸最相似（下图右上方的小矩形）。



**注意：**通常人们所说的人脸识别是指寻找人脸的位置（也就是步骤1中所介绍的人脸检测），但本书将使用人脸识别的正式定义，即人脸识别包含步骤4中的人脸识别和步骤1中的人脸检测。

本章的项目叫 WebcamFaceRec，其效果如下图所示，它在右上角有一个小矩形，用来突出显示识别到的人脸。下图中还有一个置信度栏（confidence bar），它紧挨着预处理人脸（就是在标识人脸的矩形框正上方，那个较小的人脸），这里显示的置信度大约为 70%，这也表示所识别人脸的正确性。



在非现实条件下，当前的人脸识别技术相当可靠的，但在现实条件下使用这些人脸识别技术时会不太可靠。例如，很容易看到研究论文中人脸识别的准确率在 95% 以上，但用自己的数据集来测试这些算法，经常会发现其精确度低于 50%。这是因为目前人脸识别技术对图像中的光照类型、光照和阴影方向、准确的人脸方位、面部表情以及人当前的心情都非常敏感。如果训练（采集图像）和测试（来自摄像机图像）的图像都保持不变，则进行人脸识别的算法就可以很好地工作。如果一个人在训练时站在灯的左侧，但测试时站在右边，这就可能会带来很糟糕的结果。因此，用于训练的数据集非常重要。

人脸预处理（步骤2）的目的是减少这类问题，比如，确保所使用的人脸总是有相似的亮度和对比度，还要确保人脸特征始终在同一位置（例如，按某个位置对齐眼睛或鼻子）。好的人脸预处理将有助于提高整个人脸识别系统的可靠性，所以本章将会强调人脸预处理方法。

虽然媒体大肆宣传人脸识别在安全领域的应用，对于任何真正的安全系统，单靠目前

的人脸识别方法不能办到，但人脸识别技术可用于对可靠性要求不高的领域，如不同的人进入房间可播放个性化的音乐或当一个机器人看到人时，会说出其名字。此外，人脸识别还有各种实用的扩展，如性别识别、年龄识别和情感识别。

### 8.1.1 第一步：人脸检测

在 2000 年以前，有许多人脸检测技术，但这些技术要么运行非常慢，要么非常不可靠，或两者兼而有之。但在 2001 年，Viola 和 Jones 采用基于 Haar ( Haar-based) 的级联分类来检测对象，使这些问题有了很大的改观，在 2002 年，Lienhart 和 Maydt 又对其进行改进。这种分类器用作对象检测器时既快（一般在有 VGA 摄像机的桌面上就可实时检测人脸）又可靠（对正面人脸检测的正确率约为 95%）。这种对象检测器是人脸识别（也包括一般的机器人技术和计算机视觉技术）的革命，因为它终于让实时人脸检测和人脸识别成为可能，尤其是 Lienhart 在 OpenCV 中实现的免费对象检测器！它不仅适用于正面人脸，而且对侧面人脸（简称人脸轮廓）、眼睛、嘴巴、鼻子、公司标志等很多其他对象都适用。

为了能使用 LBP 特征进行检测，OpenCV 2.0 版扩展了该对象检测器，LBP 特征进行检测是基于在 2006 年 Ahonen、Hadid 和 Pietikäinen 的工作。由于基于 LBP 的特征检测器通常会比基于 Haar 的特征检测器快好几倍，而且它不需要许可协议，但很多 Haar 检测器要许可协议。

基于 Haar 的脸部检测器的基本思想是，对于面部正面的大部分区域而言，会有眼睛所在区域应该比前额和脸颊更暗，嘴巴应该比脸颊更暗等情形。它通常执行大约 20 个这样的比较来决定所检测的对象是否为人脸，但它必须针对图像中每个可能的位置和每种可能的人脸大小都这样做，因此，对每幅图像，这样的比较实际上经常会做上千次。基于 LBP 的人脸检测器的基本思想与基于 Haar 的人脸检测器类似，但它比较的是像素亮度直方图，例如，边缘、角落和平坦区域的直方图。

不用再去讨论上面两种方法哪种用于人脸识别最好，基于 Haar 和 LBP 的人脸检测器可能通过训练从大的图像集找到人脸，这些图像集存在 XML 文件中以便后续使用。这些级联分类检测器通常至少需使用 1000 个独特的人脸图像和 10000 个非人脸图像（例如，树木的照片、汽车和文本）作为训练，训练过程可能要很长的时间，即使在多核的台式机通常也如此（一般若 LBP 要几个小时，则 Haar 需要一个星期！）。幸好 OpenCV 自带了一些训练好的 Haar 和 LBP 检测器供用户使用。也就是说，只需通过加载不同级联分类器的 XML 文件给对象检测器就可检测正面人脸、轮廓（侧视）的面孔、眼睛或鼻子，而且 Haar 或 LBP 检测器对应的 XML 文件有所不同。

#### 1. 用 OpneCV 实现人脸检测

正如前面提到的，OpenCV 2.4 带有各种训练好的检测器，这些检测器以 XML 格式保存，可用于各种应用。下表列出了一些最常用的 XML 文件。



级联分类器的类型	XML 文件名
人脸检测器 (默认)	haarcascade_frontalface_default.xml
人脸检测器 (快速的 Haar)	haarcascade_frontalface_alt2.xml
人脸检测器 (快速的 LBP)	lbpcascade_frontalface.xml
人脸检测器的属性 (侧视)	haarcascade_profileface.xml
眼部检测器 (分为左眼和右眼)	haarcascade_lefteye_2splits.xml
嘴部检测器	haarcascade_mcs_mouth.xml
鼻子检测器	haarcascade_mcs_nose.xml
整个人身体的检测器	haarcascade_fullbody.xml

Haar 检测器和 LBP 检测器分别存储在 OpenCV 根文件夹下的 data\haarcascades 文件夹和 data\lbpcascades 文件夹中, 例如, C:\opencv\data\lbpcascades\。

对于本章的人脸识别项目, 其检测的是正面人脸。因此可使用 LBP 人脸检测器, 因为它最快而且没有专利协议问题。注意, OpenCV 2.x 自带的 LBP 人脸检测器与 Harr 人脸检测器相比不太稳定, 因此, 若需要更稳定的人脸检测器, 读者可自己训练 LBP 人脸检测器或使用 Harr 人脸检测器。

## 2. 加载 Haar 或 LBP 对象或人脸检测器

为了执行对象或人脸检测, 首先必须用 OpenCV 的 CascadeClassifier 类来加载训练好的 XML 文件, 具体操作如下:

```
CascadeClassifier faceDetector;
faceDetector.load(faceCascadeFilename);
```

通过指定不同的文件名, 就可加载 Haar 或 LBP 检测器。用这种方式加载时, 有一个常见的错误, 即提供了错误的文件夹或文件名。由于编译环境不同, load() 方法会返回 false 或产生一个 C++ 异常 (用一个断言错误来退出程序)。所以处理 load() 方法的这种错误需用 try/catch 块, 如果出错了, 就显示一个友好的错误信息给用户。很多初学者会跳过错误检查, 但当用户未正确加载时, 给用户提示信息很关键, 若不这样做, 用户可能会花很长时间来调试代码的其他部分, 最后才发现是因为数据没有加载。一个简单的错误消息显示如下:

```
CascadeClassifier faceDetector;
try {
    faceDetector.load(faceCascadeFilename);
} catch (cv::Exception e) {}
if ( faceDetector.empty() ) {
    cerr << "ERROR: Couldn't load Face Detector (";
    cerr << faceCascadeFilename << ")!" << endl;
    exit(1);
}
```



### 3. 访问摄像机

为了从一台电脑的摄像头或从视频文件中获取帧，可简单地按摄像机编号或视频文件名来调用 VideoCapture::open() 函数，然后使用 C++ 的流运算符来获取帧，这与 1.1 节介绍的一样。

### 4. 用 Harr 或 LBP 检测器来检测对象

现已加载检测器（在初始化期间只加载一次），可用它来检测摄像机每帧中的人脸。但首先需专门针对人脸检测来预处理摄像机图像，其步骤如下。

1) 灰度色彩转换：人脸检测只适用于灰度图像。因此，应转换彩色摄像机帧为灰度图像。

2) 收缩摄像机图像：人脸检测的速度取决于输入图像的大小（对大的图像很慢，而对小图像很快），即使在低分辨率下，其检测也相当可靠。所以应该缩小摄像机图像，使其有一个比较合理的尺寸（或对检测器的 minFeatureSize 使用一个大的值，这将在稍后解释）。

3) 直方图均衡化：在光线不足的情况下，人脸检测器并不可靠。因此，应该进行直方图均衡化，以改善对比度和亮度。

#### (1) 灰度图像转换

使用 cvtColor() 函数，可很容易地将 RGB 的彩色图像转换为灰度图像。但如果知道一个彩色图像，且必须指定输入图像格式（通常台式机是 3 通道 BGR，而移动设备是 4 通道的 BGRA 格式），就只能这样做（即不需要对灰度图像进行转换）。因此，应该允许三种不同的输入彩色格式，如下面代码所示：

```
Mat gray;
if (img.channels() == 3) {
    cvtColor(img, gray, CV_BGR2GRAY);
}
else if (img.channels() == 4) {
    cvtColor(img, gray, CV_BGRA2GRAY);
}
else {
    // Access the grayscale input image directly.
    gray = img;
}
```

#### (2) 收缩摄像机图像

可使用 resize() 函数来将图像按一定尺寸或缩放因子进行缩放。只要图像尺寸大于  $240 \times 240$  像素，人脸检测器通常都能得到很好的结果（除非要检测的人脸远离摄像机），因为它会寻找比 minFeatureSize（通常为  $20 \times 20$  像素）更大的所有人脸。因此，可将摄像机图像收缩为 320 像素宽，对于来自 VGA 摄像机的图像，或一个有 500 万像素的高清摄像机都可以这样做。另一个重要的操作是，要放大检测结果。因为如果在一个缩小的图像中检测到人脸，则输出的结果也是缩小的。另外，如果不缩小输入图像，其代替的方式是，在

检测器中设置大的 `minFeatureSize` 值。除此以外，还必须确保图像不会变得太宽或太窄。例如， $800 \times 400$  宽屏图像缩小到  $300 \times 200$  时就会看起来很窄。所以，须保持输出与输入有相同的纵横比（宽度与高度之比）。可通过计算来得到要缩小多少图像的宽度，然后将相同的比例因子用到高度上。具体操作如下：

```
const int DETECTION_WIDTH = 320;
// Possibly shrink the image, to run much faster.
Mat smallImg;
float scale = img.cols / (float) DETECTION_WIDTH;
if (img.cols > DETECTION_WIDTH) {
    // Shrink the image while keeping the same aspect ratio.
    int scaledHeight = cvRound(img.rows / scale);
    resize(img, smallImg, Size(DETECTION_WIDTH, scaledHeight));
}
else {
    // Access the input directly since it is already small.
    smallImg = img;
}
```

### （3）直方图均衡化

很容易使用 `equalizeHist()` 函数来执行直方图均衡化，以改善图像的对比度和亮度（就像《*Learning OpenCV: Computer Vision with the OpenCV Library*》解释的那样）。有时，这会使图像看起来很奇怪，但总体上提升亮度和对比度对人脸检测有帮助。下面是 `equalizeHist()` 函数的使用方法：

```
// Standardize the brightness & contrast, such as
// to improve dark images.
Mat equalizedImg;
equalizeHist(inputImg, equalizedImg);
```

## 8.1.2 检测人脸

现在已经完成将彩色图像转换为灰度图像，并对图像进行缩小和直方图均衡化处理。接着将使用 `Cascade Classifier::detectMultiScale()` 函数来进行人脸检测，有一些参数需要传给该函数。

- ❑ `minFeatureSize`：该参数决定了最小的人脸大小，通常为  $20 \times 20$  像素或  $30 \times 30$  像素，但这取决于所使用的用例（`use case`）和图像尺寸。如果正在一个摄像机或智能手机上执行人脸检测，则人脸一般总是很接近摄像机，可将该参数放大为  $80 \times 80$  像素，以便得到更快的检测，如果检测远离人脸，比如，有朋友在海滩，则可将该参数的大小设置  $20 \times 20$  像素。
- ❑ `searchScaleFactor`：该参数决定有多少不同大小的人脸要搜索，通常设置为 1.1 能得到很好的检测结果，若设置为 1.2，会使检测更快，但经常找不到人脸。
- ❑ `minNeighbors`：该参数决定人脸检测器如何确定人脸已被检测到，通常它的值为 3，

如果需要所检测人脸的正确率更高,可将其设置得更大,但这样做可能会使一些人脸无法被检测到。

□ **flags** : 该参数允许用户指定是否要查找所有的人脸(默认)或只查找最大的人脸(CASCADE\_FIND\_BIGGEST\_OBJECT),如果只查找最大人脸,程序会运行得比较快。有其他几个参数值可让人脸检测的速度提高 1% 至 2%,例如, CASCADE\_DO\_ROUGH\_SEARCH 或 CASCADE\_SCALE\_IMAGE。

detectMultiScale() 函数的输出结果是一个 std::vector,其类型为 cv::Rect 类。例如,如果检测到两个人脸,将其作为输出结果存储到两个矩形数组中, detectMultiScale() 函数的用法如下:

```
int flags = CASCADE_SCALE_IMAGE; // Search for many faces.
Size minFeatureSize(20, 20);    // Smallest face size.
float searchScaleFactor = 1.1f;  // How many sizes to search.
int minNeighbors = 4;           // Reliability vs many faces.

// Detect objects in the small grayscale image.
std::vector<Rect> faces;
faceDetector.detectMultiScale(img, faces, searchScaleFactor,
                              minNeighbors, flags, minFeatureSize);
```

可通过 object.size() 函数来查看存储在矩形向量中的元素个数,并由此判断是否有人脸被检测到。

正如前面提到的,如果用一个缩小图像进行人脸检测,得到的结果也是缩小的,因此如果想得到人脸在原始图像中的区域,就需要放大此结果。还需要确定人脸在图像中的整个边界。若人脸并不完全在图像中,则 OpenCV 会立即产生异常。具体代码如下:

```
// Enlarge the results if the image was temporarily shrunk.
if (img.cols > scaledWidth) {
    for (int i = 0; i < (int)objects.size(); i++) {
        objects[i].x = cvRound(objects[i].x * scale);
        objects[i].y = cvRound(objects[i].y * scale);
        objects[i].width = cvRound(objects[i].width * scale);
        objects[i].height = cvRound(objects[i].height * scale);
    }
}
// If the object is on a border, keep it in the image.
for (int i = 0; i < (int)objects.size(); i++) {
    if (objects[i].x < 0)
        objects[i].x = 0;
    if (objects[i].y < 0)
        objects[i].y = 0;
    if (objects[i].x + objects[i].width > img.cols)
        objects[i].x = img.cols - objects[i].width;
    if (objects[i].y + objects[i].height > img.rows)
        objects[i].y = img.rows - objects[i].height;
}
```



注意，上面的代码会查看图像中所有人脸，但若只关心一个人脸，则将 flag 变量修改为：

```
int flags = CASCADE_FIND_BIGGEST_OBJECT |
           CASCADE_DO_ROUGH_SEARCH;
```

WebcamFaceRec 项目封装了 OpenCV 的 Haar 或 LBP 检测器，以便比较容易地在图像中找到人脸或眼睛，具体代码如下：

```
Rect faceRect;    // Stores the result of the detection, or -1.
int scaledWidth = 320; // Shrink the image before detection.
detectLargestObject(cameraImg, faceDetector, faceRect,
                    scaledWidth);
if (faceRect.width > 0)
    cout << "We detected a face!" << endl;
```

现在人脸上有矩形，可在许多方法中来实现该功能，如，从原始图像提取或裁剪人脸图像。下面的代码可实现这样的功能：

```
// Access just the face within the camera image.
Mat faceImg = cameraImg(faceRect);
```

右图的矩形区域经常用于人脸检测器。



### 8.1.3 第2步：人脸预处理

前面介绍过人脸识别极易受光照条件、脸部朝向、面部的表情等因素的影响，因此，尽量消除这些差异非常重要。否则在同样条件下，人脸识别算法经常会认为两个不同人脸比同一个人的两张脸更相似。

人脸预处理的最简单形式就是使用 `equalizeHist()` 函数来做直方图均衡，这与刚才人脸检测那步一样。对光照和位置条件都变化不大的项目来讲，这样做足够了，但在现实环境中，为了让检测算法更可靠，需要很多复杂的技术，包括面部特征检测（例如，检测眼睛、鼻子、嘴巴和眉毛）。为简单起见，本章只使用眼部检测，而忽略其他面部特征，如，嘴和鼻子，若要检测全部面部特征会不太实用。下图为一个经预处理后放大的典型人脸视图，本节会介绍得到这种图像的技术。



#### 1. 眼部检测

人眼检测对人脸预处理非常有用，因为对于正面人脸而言，虽然面部表情、光照条件、摄像机属性、与相机的距离等都会变化，但总可假定人眼是水平的，并对称分部在人脸上，两只眼睛在人脸的位置和大小相当标准。当人脸检测器将别的对象当成人脸时，可通过眼部检测来丢弃这种误判，这是一种很有用的方法。人脸检测器和眼部检测器基本上不会同时出错。所以如果同时采用人脸检测和眼部检测来处理面部图像，出现误判的可能性很小（但在少量人脸识别应用中，眼部检测经常与人脸检测器一样，不能得到很好的效果）。

OpenCV 2.4 自带有一些训练好的眼部探测器，其中一些在睁眼或闭眼的情形下都能检

测到眼睛，而另一些只能检测睁眼的情形。而这之前的版本只能检测睁开的眼睛的情形。

下面的代码可用来检测睁开或闭着的眼睛：

- ❑ haarcascade\_mcs\_lefteye.xml (and haarcascade\_mcs\_righteye.xml)
- ❑ haarcascade\_lefteye\_2splits.xml (and haarcascade\_righteye\_2splits.xml)

下面的代码只能检测睁开的眼睛：

- ❑ haarcascade\_eye.xml
- ❑ haarcascade\_eye\_tree\_eyeglasses.xml



**注意：**由于在训练检测器时，包含有睁开或闭合的左眼和右眼数据是单独训练的，因此，在应用中，对左眼或右眼的检测要用不同的检测器。反之亦然，只针对睁开的眼部检测器会同时用到左眼或右眼的检测器。

如果有人戴着眼镜，就该用检测器 haarcascade\_eye\_tree\_eyeglasses.xml 来检测眼睛，但对于没有戴眼镜的人脸，这样做就不可行。

如果 XML 文件名中包含“left eye”，这意味着是人的左眼，而在摄像机图像中，它自然会位于人脸的右侧，而不是左侧！

前面提到的四个人脸检测器按其可靠性从上到下依次降低。所以如果不需要检测戴眼镜的人脸，则第一个人脸检测器也许是最好的选择。

## 2. 搜索眼部位置

对于眼部检测，有一个很重要的方法，通过裁剪输入图像来得到近似的眼部位置，这与人脸检测时一样，通过裁剪一个小矩形来显示左眼的位置（如果使用的是左眼检测器），同样也可显示右眼位置。若在整个人脸图像或整个照片中只对眼部进行检测会很慢且不可靠。对于不同的人脸区域，使用不同的眼部检测器会得到更好的效果，例如，如果只在实际眼睛周围很窄的区域搜索，则 haarcascade\_eye.xml 检测效果最好，而对于眼部周围较大区域，用 haarcascade\_mcs\_lefteye.xml 和 haarcascade\_lefteye\_2splits.xml 检测器会得到最好效果。

不同眼部检测器（当使用 Haar 人脸检测器时）会在一些区域得到好的效果，这些区域如下表所示，用检测到的人脸矩形框中的相对坐标来表示这些区。

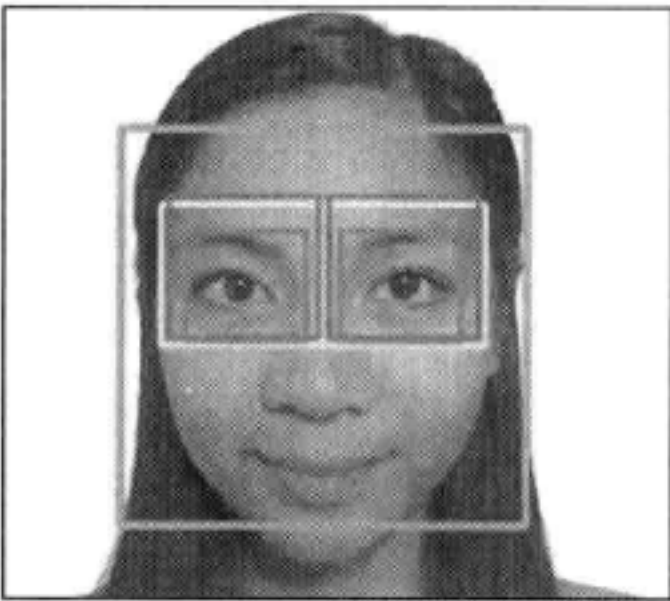
级联分类器	EYE_SX	EYE_SY	EYE_SW	EYE_SH
haarcascade_eye.xml	0.16	0.26	0.30	0.28
haarcascade_mcs_lef	0.10	0.19	0.40	0.36
haarcascade_lefteye_2splits.xml	0.12	0.17	0.37	0.36

下面的源代码可从检测到的人脸中提取左眼和右眼区域：

```
int leftX = cvRound(face.cols * EYE_SX);
int topY = cvRound(face.rows * EYE_SY);
int widthX = cvRound(face.cols * EYE_SW);
int heightY = cvRound(face.rows * EYE_SH);
int rightX = cvRound(face.cols * (1.0-EYE_SX-EYE_SW));

Mat topLeftOfFace = faceImg(Rect(leftX, topY, widthX,
                                heightY));
Mat topRightOfFace = faceImg(Rect(rightX, topY, widthX,
                                heightY));
```

下图显示了不同眼部探测器的理想搜索区域，其中 haarcascade\_eye.xml 和 haarcascade\_eye\_tree\_eyeglasses.xml 对小的搜索区域能获得最好的效果，而 haarcascade\_mcs\_\*eye.xml 和 haarcascade\_\*\_eye\_2splits.xml 在较大的搜索区域能获得最好的效果。注意，检测到的人脸矩形也显示在该图上，其目的是用来与眼部搜索的大区域进行比较，从而让用户感受到这个所谓的大区域究竟有多大。



当使用上表所给出的搜索区域时，对不同的眼部检测有一些相似的检测属性。

级联分类器	Reliability* (正确率)	Speed** (检测速度)	Eyes found (睁眼或闭眼)	Glasses (是否戴眼镜)
haarcascade_mcs_lefteye.xml	80%	18 ms	睁眼或闭眼	否
haarcascade_lefteye_2splits.xml	60%	5 ms	睁眼或闭眼	否
haarcascade_eye.xml	40%	5 ms	只有睁眼这 一种情形	否
haarcascade_eye_tree_eyeglasses.xml	15%	10 ms	只有睁眼这 一种情形	是

列 \*Reliability 的值表示对没有戴眼镜且双眼睁开的情况下使用 Haar 人脸检测器进行正面人脸检测后，得到的正确率。若检测眼睛闭着的情形，则正确率可能会下降，或者如果被检测者戴眼镜，则正确率和速度都会下降。

列 \*Speed 的值以毫秒为单位，针对大小为 320 × 240 像素的图像，CPU 为英特尔酷睿 i7 2.2 GHz (对 1000 张照片做平均)。该列的值在通常情况检测到眼睛比没有检测到眼睛要快很多，因为必须扫描整个图像才能确定没有找到眼睛，注意，haarcascade\_mcs\_lefteye.xml 检测器会比其他眼睛检测器慢得多。

例如，若缩小的照片为 320 × 240 像素，在其上执行直方图均衡，使用 LBP 检测器对正面人脸进行检测，从而得到一个人脸，然后使用 haarcascade\_mcs\_lefteye.xml 检测器来提取人脸的左眼区域和右眼区域，并对每个眼部区域进行直方图均衡。如果在左眼上使用 haarcascade\_mcs\_lefteye.xml 检测器 (眼睛实际上在图像的右上方)，并对右眼使用



haarcascade\_mcs\_righteye.xml 检测器 (眼睛在图像的左上方), 每个眼部检测器会使用 LBP 检测到的正面人脸图像的大约 90% 区域。因此, 若两只眼睛都要被检测到, 眼部检测器需使用检测到的正面人脸图像大约 80% 的区域。

注意, 虽然建议在检测人脸前要缩小摄像机图像, 但检测眼部应采用摄像机的全分辨率, 因为眼睛明显会比脸小很多, 应采用尽可能高的分辨率。



**注意:** 也许读者从上表可得出这样的结论: 在选择一个眼部检测器时, 会决定是否要检测闭着或睁开的眼睛。其实不需要这样做, 读者只需使用一个眼部检测器, 如果无法检测到时, 可尝试用另一个。

无论图像中的眼睛是睁开还是闭着, 检测眼部对很多应用来讲都有用。另外, 如果速度不重要, 最好首先用 mcs\_\*eye 检测器来, 如果失败, 再用 eye\_2splits 检测器。

但对于人脸识别, 如果一个人的眼睛闭着, 会得到完全不同的结果, 所以最好首先只用 haarcascade\_eye 检测器进行检测, 如果失败, 则用 haarcascade\_eye\_tree\_eyeglasses 检测器。

也可用检测人脸的 detectLargestObject() 函数来检测眼部, 且在检测眼部之前不要缩小图像, 只需指定整个眼部区域的宽度, 就可获得较好的检测效果。使用检测器来检测左眼很容易, 如果失败, 再尝试另一种检测器 (对右眼, 也可用同样的方式)。下面的代码是基于 detectLargestObject() 函数的眼部检测:

```
CascadeClassifier eyeDetector1("haarcascade_eye.xml");
CascadeClassifier
    eyeDetector2("haarcascade_eye_tree_eyeglasses.xml");
...
Rect leftEyeRect;    // Stores the detected eye.
// Search the left region using the 1st eye detector.
detectLargestObject(topLeftOfFace, eyeDetector1, leftEyeRect,
    topLeftOfFace.cols);
// If it failed, search the left region using the 2nd eye
// detector.
if (leftEyeRect.width <= 0)
    detectLargestObject(topLeftOfFace, eyeDetector2,
        leftEyeRect, topLeftOfFace.cols);
// Get the left eye center if one of the eye detectors worked.
Point leftEye = Point(-1, -1);
if (leftEyeRect.width <= 0) {
    leftEye.x = leftEyeRect.x + leftEyeRect.width/2 + leftX;
    leftEye.y = leftEyeRect.y + leftEyeRect.height/2 + topY;
}

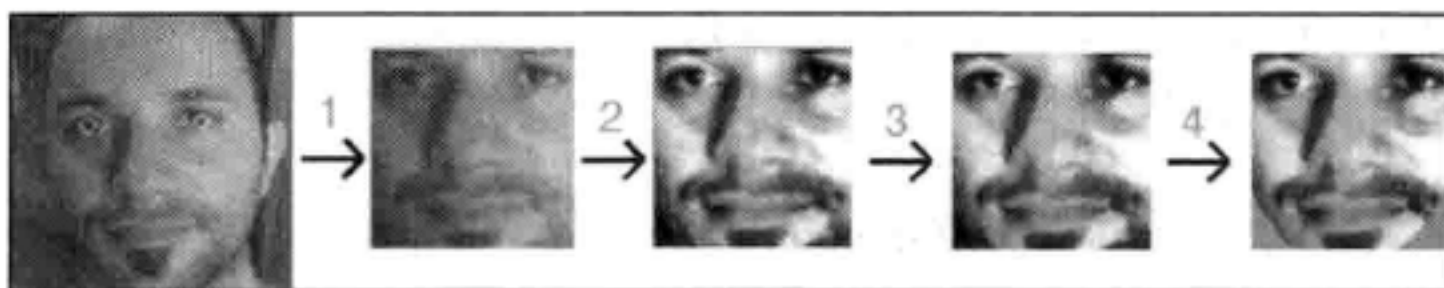
// Do the same for the right-eye
...
```

```
// Check if both eyes were detected.
if (leftEye.x >= 0 && rightEye.x >= 0) {
    ...
}
```

可将检测到的人脸和眼睛结合起来进行人脸预处理，其涉及的操作有以下内容。

- ❑ **几何变换和裁剪**：这个过程包括缩放、旋转和平移图像，以使眼睛能被对齐。然后从人脸图像中删除额头、下巴、耳朵和背景。
- ❑ **对人脸左侧和右侧分别用直方图均衡**：此过程会分别标准化左和右两侧的亮度和对比度。
- ❑ **平滑**：此过程使用双边滤波器来减少图像噪声。
- ❑ **椭圆掩码**：该椭圆掩码会将一些剩余头发和人脸图像背景去掉。

下图分别显示了人脸预处理从第1步到第4步的结果，这些步骤是在已检测到的人脸上进行的。注意，最后那幅人脸图像的左右两侧具有很好的亮度和对比度，而原图像没有。



### (1) 几何变换

所有人脸对齐很重要，否则人脸识别算法可能会出现将鼻子与眼睛进行比较的情形。刚才看到的人脸检测结果在一定程度上是对齐的，但并不是很准确的（即，人脸上的矩形不会总在额头上的相同地方）。

本章采用了更好的对齐方式，即，通过使两个检测到的眼睛在期望的位置上保持水平对齐，由此来对齐人脸。要采用这样的对齐，可用 `warpAffine()` 函数来做几何变换，该操作将完成以下四件事情。

- ❑ 旋转人脸，使两个眼睛保持水平；
- ❑ 缩放人脸，使两个眼睛之间的距离始终相同；
- ❑ 平移人脸，使眼睛总是在所需高度上水平居中；
- ❑ 裁剪人脸的外围，因为总要裁剪图像背景、头发、额头、耳朵和下巴。

仿射扭曲（Affine Warping）需要一个仿射矩阵，该矩阵将两个检测到的眼睛位置变换到期望的位置，然后通过裁剪来得到所需的尺寸和位置。为了得到这种仿射矩阵，需得到两眼之间的中心，然后计算两眼之间的角度，并得到它们之间的距离，具体操作如下：

```
// Get the center between the 2 eyes.
Point2f eyesCenter;
eyesCenter.x = (leftEye.x + rightEye.x) * 0.5f;
eyesCenter.y = (leftEye.y + rightEye.y) * 0.5f;

// Get the angle between the 2 eyes.
```

```

double dy = (rightEye.y - leftEye.y);
double dx = (rightEye.x - leftEye.x);
double len = sqrt(dx*dx + dy*dy);
// Convert Radians to Degrees.
double angle = atan2(dy, dx) * 180.0/CV_PI;

// Hand measurements shown that the left eye center should
// ideally be roughly at (0.16, 0.14) of a scaled face image.
const double DESIRED_LEFT_EYE_X = 0.16;
const double DESIRED_RIGHT_EYE_X = (1.0f - 0.16);
// Get the amount we need to scale the image to be the desired
// fixed size we want.
const int DESIRED_FACE_WIDTH = 70;
const int DESIRED_FACE_HEIGHT = 70;
double desiredLen = (DESIRED_RIGHT_EYE_X - 0.16);
double scale = desiredLen * DESIRED_FACE_WIDTH / len;

```

现在可变换人脸（旋转、缩放和平移）来得到检测到的双眼出现在人脸的所需位置，具体操作如下：

```

// Get the transformation matrix for the desired angle & size.
Mat rot_mat = getRotationMatrix2D(eyesCenter, angle, scale);
// Shift the center of the eyes to be the desired center.
double ex = DESIRED_FACE_WIDTH * 0.5f - eyesCenter.x;
double ey = DESIRED_FACE_HEIGHT * DESIRED_LEFT_EYE_Y -
            eyesCenter.y;
rot_mat.at<double>(0, 2) += ex;
rot_mat.at<double>(1, 2) += ey;
// Transform the face image to the desired angle & size &
// position! Also clear the transformed image background to a
// default grey.
Mat warped = Mat(DESIRED_FACE_HEIGHT, DESIRED_FACE_WIDTH,
                  CV_8U, Scalar(128));
warpAffine(gray, warped, rot_mat, warped.size());

```

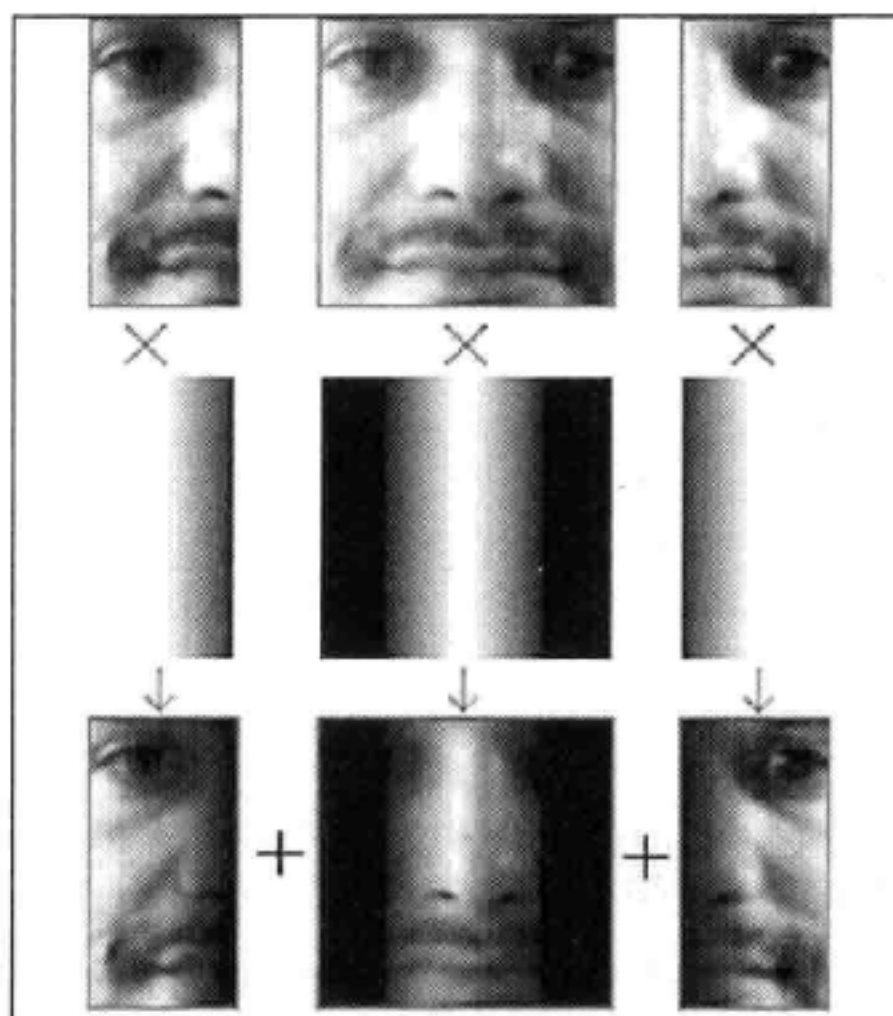
## 2. 分别对人脸左侧和右侧用直方图均衡

在现实情况中，经常有半边脸是强光照，而在另一边是弱光照。这对人脸识别算法会产生很大的影响，因为同一张脸的左侧和右侧会看起来像来自完全不同的两个人。因此，将人脸左右两边分别进行直方图均衡，然后标准化人脸两侧的亮度和对比度。

如果在人脸的左半部和右半部简单使用直方图，将在中间出现一条非常明显的边，因为在左右两侧的平均亮度可能不同。因此，需要删除这条边，可通过从左侧或右侧逐渐向中心使用两种直方图均衡，并将其与整个人脸图像的直方图均衡混合。最左边使用左直方图均衡，最右边使用右直方图均衡，而中心将使用左、右以及整个人脸均衡值的平滑混合。

下图展示了左半部均衡、整体均衡以及右半部均衡后混合在一起的效果。





为了达到上图的效果，需要整个人脸均衡、左半部均衡以及右半部均衡的副本，具体操作如下：

```
int w = faceImg.cols;
int h = faceImg.rows;
Mat wholeFace;
equalizeHist(faceImg, wholeFace);
int midX = w/2;
Mat leftSide = faceImg(Rect(0,0, midX,h));
Mat rightSide = faceImg(Rect(midX,0, w-midX,h));
equalizeHist(leftSide, leftSide);
equalizeHist(rightSide, rightSide);
```

现在将这三幅图像融合在一起。由于图像很小，虽然 `image.at<uchar>(Y,X)` 函数很慢，也很容易用它来直接获取访问像素。因此可直接访问三幅输入图像和输出图像的像素来合并这三幅图像，具体操作如下所示：

```
for (int y=0; y<h; y++) {
    for (int x=0; x<w; x++) {
        int v;
        if (x < w/4) {
            // Left 25%: just use the left face.
            v = leftSide.at<uchar>(y,x);
        }
        else if (x < w*2/4) {
            // Mid-left 25%: blend the left face & whole face.
            int lv = leftSide.at<uchar>(y,x);
            int wv = wholeFace.at<uchar>(y,x);
            // Blend more of the whole face as it moves
            // further right along the face.
            float f = (x - w*1/4) / (float)(w/4);
```

```

        v = cvRound((1.0f - f) * lv + (f) * wv);
    }
    else if (x < w*3/4) {
        // Mid-right 25%: blend right face & whole face.
        int rv = rightSide.at<uchar>(y,x-midX);
        int wv = wholeFace.at<uchar>(y,x);
        // Blend more of the right-side face as it moves
        // further right along the face.
        float f = (x - w*2/4) / (float)(w/4);
        v = cvRound((1.0f - f) * wv + (f) * rv);
    }
    else {
        // Right 25%: just use the right face.
        v = rightSide.at<uchar>(y,x-midX);
    }
    faceImg.at<uchar>(y,x) = v;
} // end x loop
} // end y loop

```

这种分开进行直方图均衡化可显著减少人脸左右两边不同光线的影响，但读者必须明白，人脸是一个有许多阴影的复杂三维形状，不可能完全删除一边的光照影响。

### 3. 光滑

为了减少像素噪声的影响，可在人脸像上使用双边滤波器，双边滤波器不仅对平滑大多数图像效果显著，而且还会保持明显的边缘。直方图均衡会大幅提高像素的噪声，所以将滤波器强度设为 20，以去除过多的像素噪声，但只使用两个像素的邻域，因为这里需要加强平滑小图像区域的像素噪声，具体操作如下：

```

Mat filtered = Mat(warped.size(), CV_8U);
bilateralFilter(warped, filtered, 0, 20.0, 2.0);

```

### 4. 椭圆掩码

虽然通过几何变换已经删除了大部分图像的背景、额头以及头发，但还需要采用椭圆掩码来删除一些拐角区域，如，颈部。这些区域可能来自人脸阴影，特别是如果人脸不是笔直地对着相机更容易出现这些情况。为了创建掩码，需在白色图像上画一个椭圆，其内部为白色，而外部为黑色。该椭圆在水平方向上的半径为 0.5（即它等于人脸的宽度），垂直方向上的半径为 0.8（通常人脸的高度大于其宽度），并且中心坐标在（0.5，0.4）处，如下图所示，该图已经用椭圆掩码删除人脸图像中一些不需要的区域。



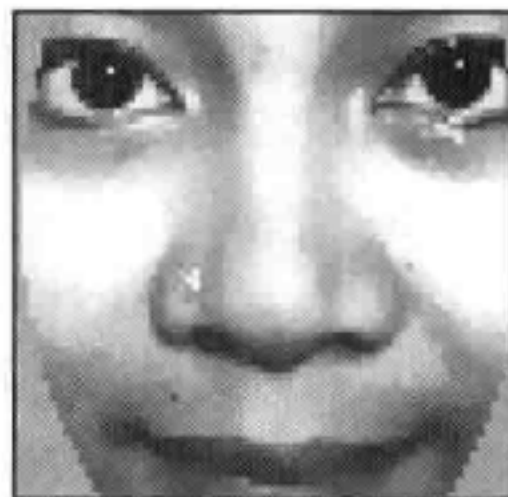
可通过 `cv::setTo()` 函数来得到掩码，该函数通常将整个图像设置为某个像素值，但由

于需要一个掩码，只能将图像的某些部分设为所给定的像素值。这里用灰色填补图像，这使得人脸剩下部分有较小的对比度，具体的代码实现如下：

```
// Draw a black-filled ellipse in the middle of the image.
// First we initialize the mask image to white (255).
Mat mask = Mat(warped.size(), CV_8UC1, Scalar(255));
double dw = DESIRED_FACE_WIDTH;
double dh = DESIRED_FACE_HEIGHT;
Point faceCenter = Point( cvRound(dw * 0.5),
                           cvRound(dh * 0.4) );
Size size = Size( cvRound(dw * 0.5), cvRound(dh * 0.8) );
ellipse(mask, faceCenter, size, 0, 0, 360, Scalar(0),
        CV_FILLED);

// Apply the elliptical mask on the face, to remove corners.
// Sets corners to gray, without touching the inner face.
filtered.setTo(Scalar(128), mask);
```

下面这幅放大的图像是对所有人脸图像进行预处理后得到的其中一个示例图像。值得注意的是，预处理使得在不同亮度、人脸图像旋转、拍摄角度、背景、光照位置等条件下得到的图像用于人脸识别会更加一致。通过预处理的人脸图像将作为人脸识别阶段的输入，这样会在收集用于训练的人脸图像的同时，也可试着识别输入的人脸。



#### 8.1.4 第3步：收集并训练人脸

收集人脸可以看成是这样一个简单的过程：把从摄像机得到的人脸图像通过预处理后放到相应的数组中，也将其类标签放到类标签数组中（类标签是用来指定拍摄的是哪个人的脸）。例如，可分别对第一人和第二人的10幅人脸图像进行预处理后，将结果存放到数组中，并生成由20个整数构成的数组（该数组的前10个元素为0，后10个元素为1），然后将这两个数组作为人脸识别算法的输入。

人脸识别算法会学习如何对不同人的脸进行分类。人脸识别有一个训练阶段，其涉及的人脸图像集合称为训练集。在算法完成训练后，需要将所学结果保存到文件或内存中，然后用其识别从摄像机中获得的人脸，该过程称为测试阶段。如果对从摄像机获取的人脸图像进行预处理，然后直接用于测试，这种图像称为测试图像；如果将多个这样的图像（例如，来自图像文件夹中的所有图像）用作测试，则称这样的图像集合为测试集。

一个好的数据集应包含人脸变化的各种情形，这些变化可能会出现在训练集中。例如，若只测试完全正面的人脸（如，身份证照片），则只需训练集图像有完全正面的人脸即可。但如果用于测试的人脸是朝左或向上的，则应确保训练集中也有这样的人脸图像，否则人脸识别算法很有可能无法识别这类人脸图像，因为出现的人脸与训练的人脸完全不同。这



也适用于其他情形，如，面部表情（例如，训练集中的人脸总是面带微笑，而测试集中的人脸没有微笑）或光照方向（例如，训练集中强光在人脸的左侧，但测试集中却在右侧），若是这样，人脸识别算法很难识别它们。人脸预处理会减少这些因素的影响，但绝对不能完全消除，尤其是人脸方向不一样的问题。由于图像中人脸方向的不同，会使人脸中所有元素位置有很大的不同，从而造成对人脸的识别非常困难。



**注意：**一个好的训练集应包含很多实际情形，得到这种训练集的方法是，获取每个人按头部进行左侧、向上、右侧、向下旋转的图像，并获取人脸正面的图像。然后获取这个人向上和向下倾斜后的人脸图像，还要获取他们的面部表情改变时的图像，例如，微笑、生气以及正常情况下的人脸图像。如果收集人脸图像时都遵循这样的原则，就有可能更好地识别现实中的每个人脸。甚至有时为了得到更好的结果，需多取一个或多个位置或方向上的人脸图像，例如，旋转摄像机，并朝相反方向来回移动，不断重复整个过程，使训练集包含各种不同的光照条件。

所以将一个人的 100 张人脸图像用于训练通常比仅用 10 张要好，但是如果这 100 张人脸图像的外观几乎一样，其识别效果仍然会很差。因此，更重要的是，训练集要包含测试集中各种人脸变化情形，而不仅仅有大量人脸图像。为了确保训练集中人脸不要太相似，应在收集每个人脸图像之间有足够的延迟。例如，若摄像机每秒有 30 帧，则在短短几秒钟内就可收集 100 张人脸图像，这种情况下人可能还没有来得及移动，因此，较好的方法是每秒采集一张人脸图像，然后移动他们的脸。另一种提高人脸图像多样性的简单方法是，只收集与前面差别较大的人脸图像。

### 1. 收集用于训练的预处理图像

为了确保收集的人脸图像之间至少有一秒的间隔，需记下这样的时间间隔。具体操作如下：

```
// Check how long since the previous face was added.
double current_time = (double)getTickCount();
double timeDiff_seconds = (current_time -
                           old_time) / getTickFrequency();
```

若要比对两幅图像像素之间的相似性，可用基于 L2 范数的相对错误评价标准，该标准是将两个图像的相应像素值相减，并对所得的差值求平方和，然后再对结果求平方根。

因此，如果人没有移动，则用上一幅人脸的每个像素值减去当前人脸图像的像素值就会得到一个非常小的数字，但若他们只要朝任何方向稍微移动一下，两幅图像的 L2 误差会很高。L2 误差是针对所有像素的，因此，它的值与图像分辨率有关。为了得到平均误差，可用这个值除以图像像素总数。可将这个简洁实用的功能封装成一个名为 `getSimilarity()` 的函数，该函数的具体实现如下：

```

double getSimilarity(const Mat A, const Mat B) {
    // Calculate the L2 relative error between the 2 images.
    double errorL2 = norm(A, B, CV_L2);
    // Scale the value since L2 is summed across all pixels.
    double similarity = errorL2 / (double)(A.rows * A.cols);
    return similarity;
}

...

// Check if this face looks different from the previous face.
double imageDiff = MAX_DBL;
if (old_preprocessedFace.preprocessedFace.data) {
    imageDiff = getSimilarity(preprocessedFace,
                             old_preprocessedFace);
}

```

通常情况下，如果图像没有移动太多，这种相似度将低于 0.2。如果图像移动了，则相似度会高于 0.4，所以用 0.3 作为收集新人脸的阈值。

可用很多技巧来获取更多的训练数据，例如，使用镜像人脸、加入随机噪声、改变人脸图像的一些像素、按某个比例缩放人脸，或将人脸旋转一些度数（虽然在预处理人脸时会尽力消除这些影响！）。下面将镜像人脸添加到训练集中，在训练集中就会有原人脸和镜像人脸的信息，训练集就会更大，这也有助于减少非对称人脸问题，即，参与训练的人脸图像总是稍微偏左或偏右，但测试时不存在这种情况，通过镜像人脸就可能解决此问题。具体实现过程如下：

```

// Only process the face if it's noticeably different from the
// previous frame and there has been a noticeable time gap.
if ((imageDiff > 0.3) && (timeDiff_seconds > 1.0)) {
    // Also add the mirror image to the training set.
    Mat mirroredFace;
    flip(preprocessedFace, mirroredFace, 1);

    // Add the face & mirrored face to the detected face lists.
    preprocessedFaces.push_back(preprocessedFace);
    preprocessedFaces.push_back(mirroredFace);
    faceLabels.push_back(m_selectedPerson);
    faceLabels.push_back(m_selectedPerson);

    // Keep a copy of the processed face,
    // to compare on next iteration.
    old_preprocessedFace = preprocessedFace;
    old_time = current_time;
}

```

这里用到了数组 `preprocessedFaces` 和 `faceLabels`，它们都是 `std::vector` 类型，分别保存着预处理图像和相应的标签，该标签就是每个人的编号（总人数保存在整型变量

m\_selectedPerson 中)。

为了更加形象地展现添加到训练集中的当前人脸图像, 可提供一个可视通知, 其方法是在整个图像的一个大的白色矩形上显示通知或在很短时间显示获取的人脸, 以便让用户知道这是刚才拍摄的图像。通过 OpenCV 的 C++ 接口来对 CV::mat 的 “+” 做运算符重载, 以增加图像中每个像素的值, 所增加的像素值不能超过 255 (通过 saturate\_cast 来实现此功能, 它会让像素值始终在 0 ~ 255)。在上面人脸图像收集程序的后面插入 displayedFrame 函数, 以显示彩色视频帧的复制:

```
// Get access to the face region-of-interest.
Mat displayedFaceRegion = displayedFrame(faceRect);
// Add some brightness to each pixel of the face region.
displayedFaceRegion += CV_RGB(90,90,90);
```

## 2. 从收集的人脸图像中训练人脸识别系统

为了识别人脸, 对每个人都收集了足够多的人脸图像, 接下来必须选择适合于人脸识别的机器学习算法, 通过它来学习收集的数据, 从而训练出一个人脸识别系统。读者可从文献中找到很多不同的人脸识别算法, 其中最简单的是特征脸和人工神经网络 (Artificial Neural Networks, ANN)。特征脸虽然很简单, 但通常比人工神经网络表现得好, 而且其性能与许多更复杂的人脸识别算法差不多。另外, 对于初学者而言, 它是基本的人脸识别算法, 并可作为测试新算法的基准算法, 基于这些原因, 该算法非常流行。

若读者想进一步了解人脸识别理论, 可阅读下面的内容。

- 特征脸 (也称为主成分分析);
- Fisher 脸 (也称为线性判别分析);
- 其他经典的人脸识别算法 (在网站 <http://www.face-rec.org/algorithms/> 有大量的人脸识别算法);
- 近期计算机视觉研究论文中出现的人脸识别算法 (如 CVPR 和 ICCV, 这两个会议的论文可在 <http://www.cvpapers.com/> 下载), 每年这里会有数百篇人脸识别的论文发表。

但读者使用本书提到的这些算法并不需要了解这些算法的理论。感谢 OpenCV 团队和 Philipp.Wagner 的 libfacerec 贡献, OpenCV 2.4.1 提供了 CV::Algorithm 类, 该类有几种不同的算法, 用其中的一种算法就可以完成简单而通用的人脸识别 (甚至在运行时才选择用该类中的某个算法也可以), 读者不必理解这些算法是如何实现的。比如, 可通过使用 Algorithm::getList() 函数来查找 OpenCV 各版本所提供的有效算法:

```
vector<string> algorithms;
Algorithm::getList(algorithms);
cout << "Algorithms: " << algorithms.size() << endl;
for (int i=0; i<algorithms.size(); i++) {
    cout << algorithms[i] << endl;
}
```



OpenCV v2.4.1 提供了三种人脸识别算法。

- ❑ FaceRecognizer.Eigenfaces : 特征脸, 也称为 PCA, 首先由 Turk 和 Pentland 于 1991 年提出的;
- ❑ FaceRecognizer.Fisherfaces : Fisher 脸, 也称为 LDA, 由 Belhumeur、Hespanha 和 Kriegman 于 1997 年提出的;
- ❑ FaceRecognizer.LBPH : 局部二值模式直方图 (Local Binary Pattern Histograms, LBPH), 由 Ahonen、Hadid 和 Pietikäinen 于 2004 年提出的。



**注意:** 关于这些人脸识别算法更详细的信息, 可以在 Philipp Wagner 的网站上找到, 网站给出了相应的文档、示例以及 Python 实现, 网站地址为: <http://bytefish.de/blog> 和 <http://bytefish.de/dev/libfacerec/>。

在 OpenCV 的 contrib 模块中, 有一个 FaceRecognizer 类, 它实现这些人脸识别算法。由于是动态链接, 有可能程序链接了 contrib 模块, 但实际上在运行时并没有加载 (如果它被认为不需要)。所以建议尝试调用 FaceRecognizer 算法之前先调用 `cv::initModule_contrib()` 函数。此函数仅在 OpenCV 2.4.1 才有效, 它保证人脸识别算法在编译时就有效:

```
// Load the "contrib" module is dynamically at runtime.
bool haveContribModule = initModule_contrib();
if (!haveContribModule) {
    cerr << "ERROR: The 'contrib' module is needed for ";
    cerr << "FaceRecognizer but hasn't been loaded to OpenCV!";
    cerr << endl;
    exit(1);
}
```

为了使用人脸识别算法, 必须要使用 `cv::Algorithm::create<FaceRecognizer>()` 函数来创建 FaceRecognizer 对象。通过将所使用的人脸识别算法的名称作为一个字符串传递给创建函数即可。如果算法在 OpenCV 的版本中有效, 就可使用该算法。因此, 需要做一个运行时错误检查, 以确保 OpenCV 中有所需的算法。例如:

```
string facerecAlgorithm = "FaceRecognizer.Fisherfaces";
Ptr<FaceRecognizer> model;
// Use OpenCV's new FaceRecognizer in the "contrib" module:
model = Algorithm::create<FaceRecognizer>(facerecAlgorithm);
if (model.empty()) {

    cerr << "ERROR: The FaceRecognizer [" << facerecAlgorithm;
    cerr << "] is not available in your version of OpenCV. ";
    cerr << "Please update to OpenCV v2.4.1 or newer." << endl;
    exit(1);
}
```

一旦加载了 FaceRecognizer 算法, 将收集的人脸数据简单地传给 `FaceRecognizer::train()` 函数就可进行训练了, 其调用代码如下:

```
// Do the actual training from the collected faces.
model->train(preprocessedFaces, faceLabels);
```

这一行代码将执行所选人脸识别的整个训练算法（例如，特征脸、Fisher 脸，或其他有效算法）。如果只有很少的人，其总共的人脸训练图像不到 20 幅，则训练很快就会返回结果，但如果涉及很多人的人脸训练图像，可能 `train()` 函数将需要几秒钟甚至几分钟才能给出结果。

### 3. 基本知识介绍

虽然这部分知识不是必须的，但这些知识对查看人脸识别算法在学习训练数据时生成的内部数据结构，特别是如果读者想了解其所选择算法背后的理论，想验证它的工作原理或要找出为什么算法不像所希望的那样工作时都非常有用。内部数据结构会因为算法不同而不同，但幸运的是，特征脸和 Fisher 脸是相同的，所以这里只看这两个算法中的一种即可。它们都是基于一维特征矩阵，当将该矩阵看成二维图像时，就好像是人脸。因此，当使用特征脸算法时，通常称特征向量为特征脸，当使用 Fisher 脸算法时，通常称特征向量为 Fisher 脸。

由这种简单的叫法可知特征脸的基本原理，计算一组指定的图像（特征脸）和混合比例（特征值），可对这些特征脸和特征值按不同的组合方式来得到训练集中的每幅图像，这样做也可将训练集中的图像与其他图像区分开。例如，如果训练集中的某些人脸留有胡子，而其他人没有，则至少有一个长胡子特征脸，训练集中有胡子的人脸图像所对应的特征脸有高的混合比率，这说明此人有胡子，而没有胡子的人脸其混合比率很小。如果训练集有 5 人，每个人有 20 幅人脸图像，那么将有 100 个特征脸和特征值来区分训练集的这 100 个人脸。实际上会对所有特征值排序，前几个特征值和对应的特征脸具有很强的判别能力，而最后几个特征值和对应的特征脸只代表随机像素噪声，不能将它们用于数据分类。因此，通常会丢弃一些排在最后的特征脸，只保留前 50 个特征脸。

与之相比，Fisher 脸的基本原理是，不对训练集中每个图像计算具体的特征向量和特征值，只计算每个人的特征向量和特征值。所以上面那个例子中有 5 人，每一个人有 20 幅人脸图像，特征脸算法将使用 100 特征脸和特征值，而 Fisher 脸算法只有 5 个 Fisher 脸和特征值。

在编译时不能访问特征脸和 Fisher 脸算法的内部数据结构，必须在运行时使用 `cv::Algorithm::get()` 函数来获取它们。这些数据结构在内部用作数学计算的一部分，而不用于图像处理，所以经常作为浮点数来存储它们，其范围一般介于  $0.0 \sim 1.0$ ，而不是 8 位的普通图像像素类型 `UCHAR`，它的范围在  $0 \sim 255$ 。而且，特征脸要么是只包含一个元素的行矩阵或列矩阵，是包含多个元素的行矩阵或列矩阵。因此，在显示这些内部数据结构之前，必须重塑它们以得到正确的矩形形状，并将其转换为 8 位 `UCHAR` 类型的像素值，这些值的取值范围介于  $0 \sim 255$ 。由于矩阵的数据取值范围是  $0.0 \sim 1.0$ 、 $-1.0 \sim 1.0$  或其他范围的值，可使用有 `cv:: NORM_MINMAX` 选项的 `cv::normalize()` 函数，这样无论什么样

的输入范围，都可确保它输出的数据值介于 0 ~ 255。下面创建一个函数来重塑一个矩形，并将其转换为 8 位像素，具体实现如下：

```
// Convert the matrix row or column (float matrix) to a
// rectangular 8-bit image that can be displayed or saved.
// Scales the values to be between 0 to 255.
Mat getImageFrom1DFloatMat(const Mat matrixRow, int height)
{
    // Make a rectangular shaped image instead of a single row.
    Mat rectangularMat = matrixRow.reshape(1, height);
    // Scale the values to be between 0 to 255 and store them
    // as a regular 8-bit uchar image.
    Mat dst;
    normalize(rectangularMat, dst, 0, 255, NORM_MINMAX,
              CV_8UC1);
    return dst;
}
```

使用 ImageUtils.cpp 和 ImageUtils.h 文件可很容易显示 cv::Mat 的结构，这使得调试 OpenCV 代码更容易，对于调试 cv::Algorithm 的数据结构更是如此，显示 cv::Mat 结构的代码如下：

```
Mat img = ...;
printMatInfo(img, "My Image");
```

将会看到控制台输出类似于下面的信息：

**My Image: 640w480h 3ch 8bpp, range[79,253][20,58][18,87]**

这些信息表明：图像的宽为 640，高为 480（即，图像是  $640 \times 480$  或  $640 \times 480$  点阵，这取决于读者如何看待它了），每个像素有三个通道，每个通道为 8 位（即，这是一个标准的 BGR 图像）。它还显示图像中每个颜色通道的最小值和最大值。



**注意：**也可能使用 printMat() 函数而不是 printMatInfo() 函数来打印图像或矩阵的实际信息。这样会很方便查看多通道浮点（multichannel-float）矩阵。这些方法对于初学者来讲很困难。

ImageUtils 的代码主要是基于 OpenCV 的 C 接口，但随着时间的推移，它也渐渐加入了更多的 C++ 接口。总可在 <http://shervinemami.info/openCV.html> 找到最新的 ImageUtils 版本。

#### 4. 平均人脸

特征脸和 Fisher 脸算法都先要计算人脸的平均值，即对所有训练图像进行算术平均，为了得到更好的人脸识别结果，可用每个人脸图像减去平均图像。下面介绍查看训练集的平均人脸。在实现特征脸和 Fisher 脸的代码中，平均人脸被命名 mean，如下所示：



```

Mat averageFace = model->get<Mat>("mean");
printMatInfo(averageFace, "averageFace (row)");
// Convert a 1D float row matrix to a regular 8-bit image.
averageFace = getImageFrom1DFloatMat(averageFace, faceHeight);
printMatInfo(averageFace, "averageFace");
imshow("averageFace", averageFace);

```

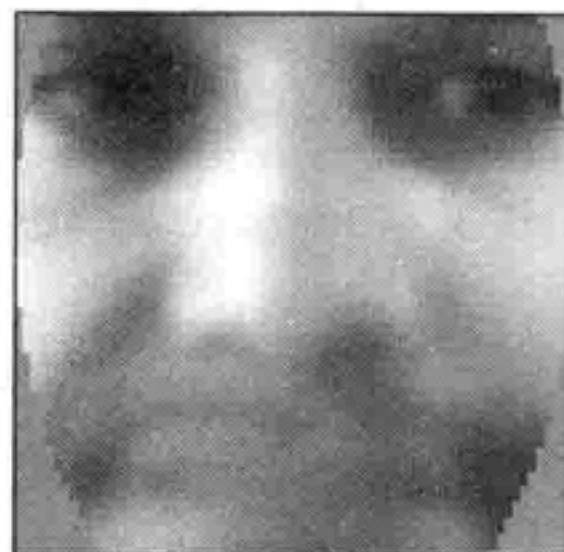
在屏幕上看到的平均人脸图像如下图(扩大)所示,此图将男人、女人以及小孩的人脸图像组合在一起。读者也可看到控制台会输出如下的文字信息:

```
averageFace (row): 4900w1h 1ch 64bpp, range[5.21,251.47]
```

```
averageFace: 70w70h 1ch 8bpp, range[0,255]
```

右图为平均人脸。

注意 `averageFace (row)` 是 64 位浮点单行矩阵,而 `averageFace` 是 8 位像素的矩形图像,其像素值的范围为 0 到 255。



## 5. 特征值、特征脸和 Fisher 脸

下面来看一下各特征值的实际值:

```

Mat eigenvalues = model->get<Mat>("eigenvalues");
printMat(eigenvalues, "eigenvalues");

```

对于特征脸,每个人脸有一个特征值,所以如果有三个人,每个人有四幅人脸图像,就可得到 12 个特征值以及对应的特征向量,这些特征值从大到小依次排序如下:

```

eigenvalues: 1w18h 1ch 64bpp, range[4.52e+04,2.02836e+06]
2.03e+06
1.09e+06
5.23e+05
4.04e+05
2.66e+05
2.31e+05
1.85e+05
1.23e+05
9.18e+04
7.61e+04
6.91e+04
4.52e+04

```

对于 Fisher 脸,每个人只有一个特征值,所以如果有三个人,每个人有四幅人脸图像,就只得到三特征值和对应的特征向量,具体信息如下:

```

eigenvalues: 2w1h 1ch 64bpp, range[152.4,316.6]
317, 152

```

若要查看特征向量(特征脸或 Fisher 脸图像),必须从大的特征向量矩阵提取它们作为列向量。由于在 OpenCV 和 C/C++ 中,数据通常按行占优的顺序存储在矩阵中,这意味着提取一列,应使用 `Mat::clone()` 函数来确保数据是连续的,否则,不能将数据重塑为一个

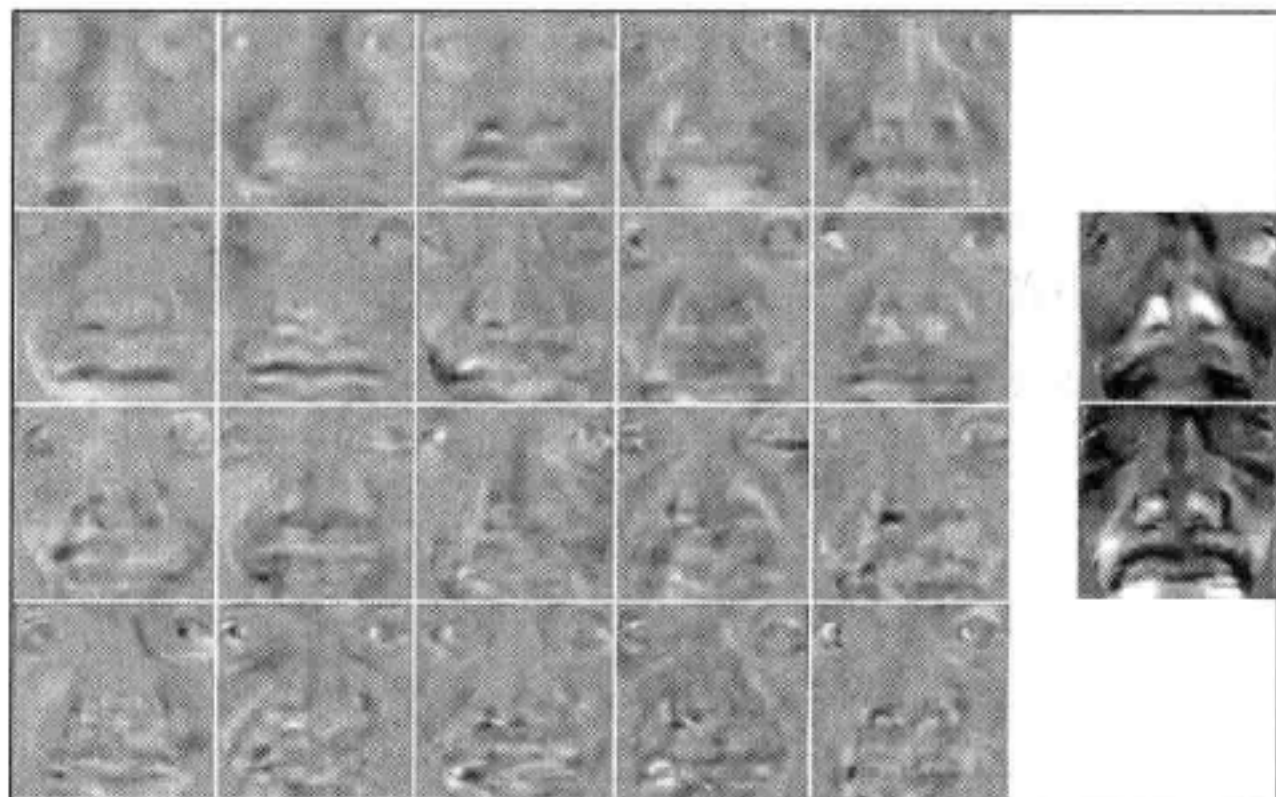
矩形。一旦有一个连续列 Mat，就可使用 `getImageFrom1DFloatMat()` 函数来显示特征向量，就像前面显示平均人脸一样：

```
// Get the eigenvectors
Mat eigenvectors = model->get<Mat>("eigenvectors");
printMatInfo(eigenvectors, "eigenvectors");

// Show the best 20 eigenfaces
for (int i = 0; i < min(20, eigenvectors.cols); i++) {
    // Create a continuous column vector from eigenvector #i.
    Mat eigenvector = eigenvectors.col(i).clone();

    Mat eigenface = getImageFrom1DFloatMat(eigenvector,
                                            faceHeight);
    imshow(format("Eigenface%d", i), eigenface);
}
```

下图将特征向量作为图像。你可以看到三个人，每个人有四幅人脸图像，总共有 12 张特征脸（显示在下图的左侧）或 3 张 Fisher 脸（在下图的右侧）。



注意，特征脸和 Fisher 脸看起来与一些面部特征有相似之处，但真正人脸并不是这个样子。简单讲这是因为平均脸已从它们中减去，所以它们只表明平均脸与每个特征脸的差异。编号表示它们属于哪个特征脸，因为特征脸都是按最显著到最不显著依次排序的，如果有 50 个或更多的特征脸，那么后面的特征脸通常只表示图像中的噪声而应被丢弃。

### 8.1.5 第 4 步：人脸识别

现在已经利用训练图像和人脸标签来完成特征脸或 Fisher 脸的训练，这两种算法属于机器学习算法，本章的最终目标是从给定的人脸图像中找出这个人是谁。最后这一步称为人脸识别或人脸辨识。

### 1. 人脸识别：通过人脸来识别这个人

感谢 OpenCV 提供的 FaceRecognizer 类，可以简单地调用 FaceRecognizer::predict() 函数来识别照片中的人。具体调用如下：

```
int identity = model->predict(preprocessedFace);
```

这个 identity 值是在收集训练的人脸时，给每个人的编号，例如，该值为 0，就表示第一个人，为 1 表示第二个人，以此类推。

这种识别的问题是它总能预测给定的人，即使输入图像不属于训练集中的人，甚至输入的是一辆汽车，算法也会去预测，而且还会强行将图像中的对象归为某一类人，这种结果很有问题！解决此问题的办法是制定置信度标准，即，要判断所得结果有多可靠，如果置信度过低，就可判断这是一个不认识的人。

### 2. 人脸验证：验证图像中是否有想找的人

为了验证预测是否可靠，或者说系统是否能对一个不认识的人进行正确地识别，这需要进行人脸验证（也称为面部认证），以此获得一个置信度，该置信度会说明用户想要找的人与某个人脸图像的相似度，（相对于脸部识别，该过程只将所得结果与多个人脸图像进行比较）。

当执行 predict() 函数时，OpenCV 的 FaceRecognizer 类会返回一个置信度，但该置信度只是基于特征子空间中的距离而得到，其可靠性很差。本章使用这样的方法来计算置信度：使用特征向量和特征值重构人脸图，然后将输入的图像与重构图像进行比较。如果一个人在训练集中有多张人脸图像，用特征向量和特征值重建后应该有非常好的效果，但如果在训练集中没有任何人脸图像（或所给的测试图像与训练图像没有相似的光照条件和面部表情），则重构的人脸看起来与输入脸差别很大，这表明它可能是一个未知的人脸。

前面的特征脸和 Fisher 脸算法也是基于这样的观点：图像可以被一组特征向量（具体的人脸图像）和特征值（混合比）大致表示。因此，如果将所有特征向量组合起来，其组合系数为训练集中其中一个人脸的特征值，则会得到与原始训练图像非常相似的人脸图像。这同样适用于与训练集相似的其他图像。如果将训练得到的特征向量与一个相似的测试图像的特征值相结合，就能够重构一幅图像，该图像有点像测试图像的翻版。

重复这样的操作，OpenCV 的 FaceRecognizer 类就很容易从任何输入图像重构人脸图像，其过程为通过 subspaceProject() 函数将人脸图像投影到特征空间，再用 subspaceReconstruct() 函数从特征空间重构图像。这里还有一个问题：需要将一个基于浮点型的行矩阵转换成一个 8 位的矩形图像（就像显示的平均脸、特征脸时所做的那样），但不需要归一化数据，因为与原始图像相比，其值很合理。如果归一化数据，其亮度和对比度与输入图像相比有所不同，若使用基于 L2 的相对误差来计算相似度会很困难。重构人脸图像的具体实现如下：



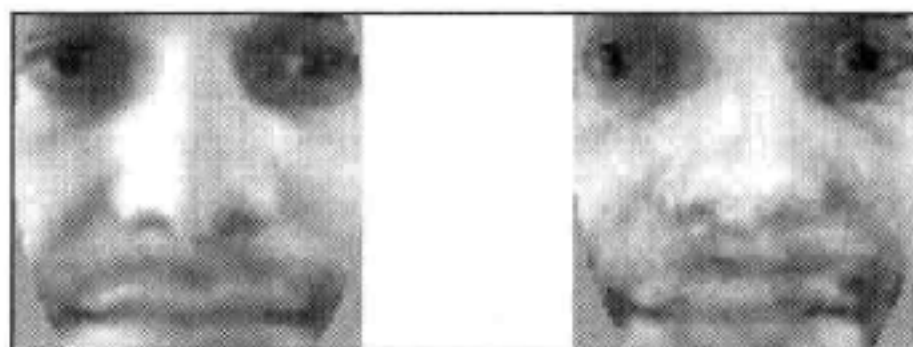
```
// Get some required data from the FaceRecognizer model.
Mat eigenvectors = model->get<Mat>("eigenvectors");
Mat averageFaceRow = model->get<Mat>("mean");

// Project the input image onto the eigenspace.
Mat projection = subspaceProject(eigenvectors, averageFaceRow,
                                preprocessedFace.reshape(1,1));

// Generate the reconstructed face back from the eigenspace.
Mat reconstructionRow = subspaceReconstruct(eigenvectors,
                                             averageFaceRow, projection);

// Make it a rectangular shaped image instead of a single row.
Mat reconstructionMat = reconstructionRow.reshape(1,
                                                  faceHeight);
// Convert the floating-point pixels to regular 8-bit uchar.
Mat reconstructedFace = Mat(reconstructionMat.size(), CV_8U);
reconstructionMat.convertTo(reconstructedFace, CV_8U, 1, 0);
```

下图为两个典型的重构人脸。左边的人脸重构得很好，因为其人脸图像在训练集中，而右边的人脸重构得比较差，因为此人的脸图像并没有出现在训练集或虽然出现了，但其光照条件、面部表情、脸部方向与训练集中相应图像完全不一样。



现在可通过函数 `getSimilarity()` 来计算这样重构的脸与输入人脸之间的相似度。在前面，该函数用于比较两个图像的相似性，若此函数计算出的值小于 0.3，意味着两个图像非常相似。对于特征脸算法，一个特征向量对应一个人脸，重构人脸的效果会很好，因此通常可使用 0.5 作为阈值；对于 Fisher 脸算法，一个特征向量对应一个人，重构人脸的效果会比较差，因此，它需要一个更高的阈值，通常将其设为 0.7。具体操作如下：

```
similarity = getSimilarity(preprocessedFace,
                           reconstructedFace);
if (similarity > UNKNOWN_PERSON_THRESHOLD) {
    identity = -1;    // Unknown person.
}
```

现在可输出所识别的个人编号到控制台，或将其用于所需的地方！但这种人脸识别方法和人脸验证方法仅在某些情形下可靠，即，测试图像要出现在训练集中，则这些算法就会很可靠。因此，要获得好的识别率，需要确保训练集中人脸图像要涵盖全方位光照条件、面部表情，以及希望测试的角度。人脸预处理阶段有助于减少由光照条件和平面内旋

转（如果人将头部向左肩或右肩倾斜）所带来的差别，但对于其他方面的差异，例如，平面外（out-of-plane）旋转（如果人的头朝左转或朝右转），要出现在训练集中才能得到好的识别率。

### 8.1.6 收尾工作：保存和加载文件

可能会增加基于命令行的方法来处理输入文件并将其保存到磁盘中，甚至可执行人脸检测、人脸预处理或将人脸识别作为 Web 服务等。可以很容易地使用 FaceRecognizer 类的 save 函数和 load 函数来实现这些功能。用户也可保存训练数据，然后在程序启动时加载。

很容易将训练模型保存到 XML 或 YML 文件中，具体操作如下：

```
model->save("trainedModel.yml");
```

如果以后要增加更多数据到训练集中，需要同时保存预处理人脸和相应标签数组。

例如，下面的代码就是从文件中加载训练好的模型。需注意，必须指定人脸识别算法（例如，FaceRecognizer.Eigenfaces 或 FaceRecognizer.Fisherfaces），这些算法最初是用于建立训练模型的。

```
string facerecAlgorithm = "FaceRecognizer.Fisherfaces";
model = Algorithm::create<FaceRecognizer>(facerecAlgorithm);
Mat labels;
try {
    model->load("trainedModel.yml");
    labels = model->get<Mat>("labels");
} catch (cv::Exception &e) {}
if (labels.rows <= 0) {
    cerr << "ERROR: Couldn't load trained data from "
          << "[trainedModel.yml]!" << endl;
    exit(1);
}
```

### 8.1.7 收尾工作：制作一个漂亮的交互式 GUI

到目前为止本章所给出的代码已是一个完整的人脸识别系统，但仍需要一种向系统加载数据并使用它的方法。许多用于研究的人脸识别系统都用文本文件列表作为它的输入，其静态图像文件和其他重要的数据都存储在计算机上，例如，人的真实姓名、编号以及人脸区域的真实像素坐标（如，人脸和眼部中心的真实位置）。另一些人脸识别系统会通过手工方式来收集这些信息。

而理想的输出是将结果保存到文本文件中，以便与真实情况进行比较，从而得到该人脸识别系统与其他人脸识别系统比较后的统计信息。

但本章的人脸识别系统是专为学习而开发的，主要目的是便于实践，提高读者兴趣，而不是与最新的研究方法进行比较。因此，需要一个易于使用的 GUI，可用它来收集人脸

图像、训练、测试以及实时获取摄像机图像。本节将介绍具有这些功能的交互式 GUI。读者可使用本章提供的 GUI，或根据自己的需要来修改此 GUI，当然也可以自己设计 GUI 来执行前面介绍的人脸识别程序。

由于 GUI 会执行多个任务，可创建一组用于 GUI 的模式或状态集合，用户可通过点击按钮来改变模式。

- ❑ 启动：该状态会加载数据，并初始化数据和摄像机。
- ❑ 检测：该状态检测人脸，并显示其预处理结果，当用户点击 Add Person 按钮后，显示才结束。
- ❑ 采集：该状态收集当前的人脸图像，直到用户点击窗口中的任意位置为止。该状态会显示每个人的最新人脸。用户可通过点击来选择一个存在的人或点击 Add Person 按钮来收集不同的人脸图像。
- ❑ 训练：在这种状态下，系统用收集的所有人脸图像来进行训练。
- ❑ 识别：该状态包括突出显示识别到的人和相应的置信度。用户可通过点击来选择一个人或点击 Add Person 按钮会回到模式 2（收集）。

用户任何时候都可在窗口中按 Esc 键退出系统。还有一个 Delete All 模式，它用来重启一个新的人脸识别系统。为了能显示额外的调试信息，还需增加一个 Debug 按钮。可用一个名为 mode 的枚举变量来保存当前的所有模式。

### 1. 绘制 GUI 组件

为了在屏幕上显示当前的模式，需创建一个函数以便显示文本。OpenCV 自带的 `cv::putText()` 函数支持几种字体并且具有抗锯齿（anti-aliasing）功能，但用此函数要将文本放置到的恰当位置很困难。`cv::getTextSize()` 函数可计算文本周围的边框，可通过该函数来解决此问题。因此，可以封装一个函数，使其更容易放置文本。封装的函数还需要有如下功能：能沿着窗口的任意边缘放置文本，并确保它完全可见；也允许多行文字或单词彼此相邻，而不会彼此覆盖；能让用户指定左对齐、右对齐、顶部对齐、底端对齐；并能得到方框的边界，以便在任意位置或窗口边缘绘制多行文本。下面是用于显示文本的函数定义：

```
// Draw text into an image. Defaults to top-left-justified
// text, so give negative x coords for right-justified text,
// and/or negative y coords for bottom-justified text.
// Returns the bounding rect around the drawn text.
Rect drawString(Mat img, string text, Point coord, Scalar
                color, float fontScale = 0.6f, int thickness = 1,
                int fontFace = FONT_HERSHEY_COMPLEX);
```

现在可在 GUI 上显示当前的模式。由于窗口背景来源于摄像机的帧，在摄像机的帧上绘制文本是完全可能的，但文本颜色可能与相机背景颜色一样！所以需先绘制一个只要 1 个像素的黑色文本阴影来区别要绘制前景文字。还需在当前模式下显示一行帮助，这样用



户就知道接下可以做什么。下面是如何使用 drawString() 函数来绘制文字的例子：

```
string msg = "Click [Add Person] when ready to collect faces.";
// Draw it as black shadow & again as white text.
float txtSize = 0.4;
int BORDER = 10;
drawString(displayedFrame, msg, Point(BORDER, -BORDER-2),
           CV_RGB(0,0,0), txtSize);
Rect rcHelp = drawString(displayedFrame, msg, Point(BORDER+1,
           -BORDER-1), CV_RGB(255,255,255), txtSize);
```

下面图为 GUI 窗口底部的模式和信息，它们叠加在摄像机图像的底部。



前面提到本应用需要一些 GUI 按钮，下面的函数可很容易地绘制 GUI 按钮。

```
// Draw a GUI button into the image, using drawString().
// Can give a minWidth to have several buttons of same width.
// Returns the bounding rect around the drawn button.
Rect drawButton(Mat img, string text, Point coord,
               int minWidth = 0)
{
    const int B = 10;
    Point textCoord = Point(coord.x + B, coord.y + B);
    // Get the bounding box around the text.
    Rect rcText = drawString(img, text, textCoord,
                           CV_RGB(0,0,0));
    // Draw a filled rectangle around the text.
    Rect rcButton = Rect(rcText.x - B, rcText.y - B,
                       rcText.width + 2*B, rcText.height + 2*B);
    // Set a minimum button width.
    if (rcButton.width < minWidth)
        rcButton.width = minWidth;
    // Make a semi-transparent white rectangle.
    Mat matButton = img(rcButton);
    matButton += CV_RGB(90, 90, 90);
    // Draw a non-transparent white border.
    rectangle(img, rcButton, CV_RGB(200,200,200), 1, CV_AA);

    // Draw the actual text that will be displayed.
    drawString(img, text, textCoord, CV_RGB(10,55,20));

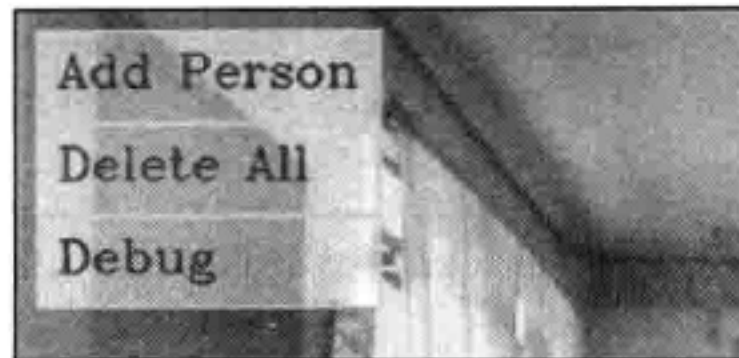
    return rcButton;
}
```

现在使用 drawButton() 函数来创建几个可点击的 GUI 按钮，它们会显示在 GUI 顶部，如下图所示。

前面介绍了 GUI 的一些模式，从启动模式开始，这些模式之间可相互切换（像有限状态机一样）。将当前模式存储在 `m_mode` 变量中。

### （1）启动模式

在启动模式下，只需加载 XML 文件中的检测器来检测人脸和眼睛，并初始化摄像机，这个过程已经介绍过了。还需创建一个 GUI 主窗口，该窗口支持鼠标回调函数，当用户在窗口移动或点击鼠标，OpenCV 将调用该函数。在某些情况下想要设置摄像机的分辨率，例如，如果摄像机支持  $640 \times 480$ ，就可在此模式下设置。具体的代码如下：



```
// Create a GUI window for display on the screen.
namedWindow(windowName);
// Call "onMouse()" when the user clicks in the window.
setMouseCallback(windowName, onMouse, 0);

// Set the camera resolution. Only works for some systems.
videoCapture.set(CV_CAP_PROP_FRAME_WIDTH, 640);
videoCapture.set(CV_CAP_PROP_FRAME_HEIGHT, 480);

// We're already initialized, so let's start in Detection mode.
m_mode = MODE_DETECTION;
```

### （2）检测模式

在检测模式下，经常需要检测人脸和眼睛，并在它们周围绘制矩形或圆形，以显示检测结果，并且还会显示当前预处理后的人脸。事实上，不管是处于哪种模式下，这些信息都需要显示。检测模式唯一特殊的地方是，当用户点击 `Add Person` 按钮，就会切换到下一个模式（收集模式）。

读者可回忆一下本章前面检测的步骤，就知道检测阶段的输出结果有以下内容。

- ❑ `Mat preprocessedFace`: 预处理人脸（如果人脸和眼睛已被检测）；
- ❑ `Rect faceRect`: 检测到的人脸区域坐标；
- ❑ `Point leftEye, rightEye`: 检测到的左眼和右眼的中心坐标。

因此，应检查是否已得到了预处理的人脸，如果已得到，就在人脸和眼睛周围绘制矩形和圆。具体代码如下：

```
bool gotFaceAndEyes = false;
if (preprocessedFace.data)
    gotFaceAndEyes = true;

if (faceRect.width > 0) {
    // Draw an anti-aliased rectangle around the detected face.
    rectangle(displayedFrame, faceRect, CV_RGB(255, 255, 0), 2,
              CV_AA);

    // Draw light-blue anti-aliased circles for the 2 eyes.
```

```

Scalar eyeColor = CV_RGB(0,255,255);
if (leftEye.x >= 0) {    // Check if the eye was detected
    circle(displayedFrame, Point(faceRect.x + leftEye.x,
        faceRect.y + leftEye.y), 6, eyeColor, 1,
        CV_AA);
}
if (rightEye.x >= 0) {  // Check if the eye was detected
    circle(displayedFrame, Point(faceRect.x + rightEye.x,
        faceRect.y + rightEye.y), 6, eyeColor, 1,
        CV_AA);
}
}
}

```

下面的代码将在窗口中心上方叠加当前预处理的人脸：

```

int cx = (displayedFrame.cols - faceWidth) / 2;
if (preprocessedFace.data) {
    // Get a BGR version of the face, since the output is BGR.
    Mat srcBGR = Mat(preprocessedFace.size(), CV_8UC3);
    cvtColor(preprocessedFace, srcBGR, CV_GRAY2BGR);

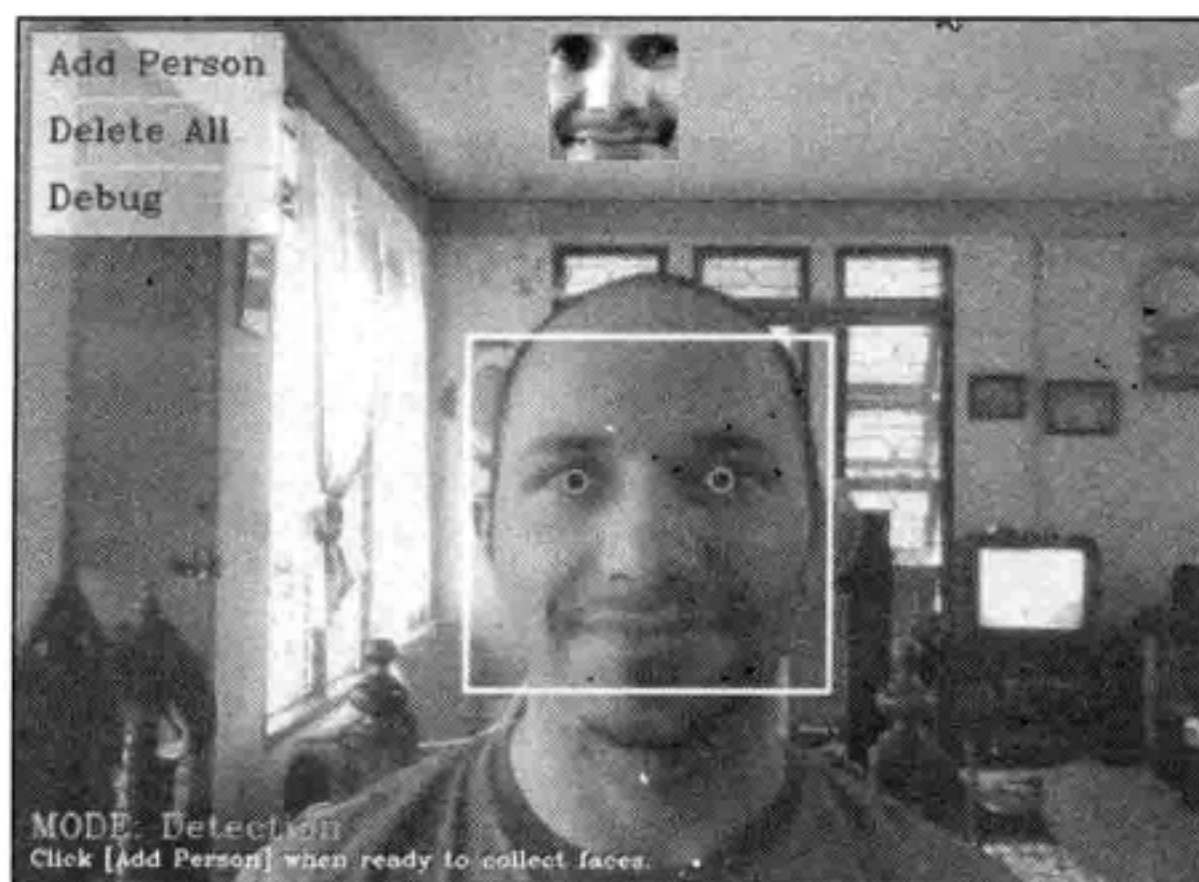
    // Get the destination ROI.
    Rect dstRC = Rect(cx, BORDER, faceWidth, faceHeight);

    Mat dstROI = displayedFrame(dstRC);

    // Copy the pixels from src to dst.
    srcBGR.copyTo(dstROI);
}
// Draw an anti-aliased border around the face.
rectangle(displayedFrame, Rect(cx-1, BORDER-1, faceWidth+2,
    faceHeight+2), CV_RGB(200,200,200), 1, CV_AA);

```

下图为检测模式下 GUI 所显示的内容，预处理人脸被显示在中心区域的上方。并且所检测到的面部和眼睛都已标记出来。





### (3) 收集模式

现在开始介绍收集模式。当用户点击 Add Person 按钮就表示开始收集一个新的人脸。如前所述，收集人脸图像的条件是：每秒收集一次，且当前人脸必须要相对于前一个收集的人脸有明显变化。注意，收集的不仅是预处理的人脸，也包括预处理人脸的镜像。

在收集模式下，要显示每个人的最新人脸，可让用户选择其中一个人脸，将其作为此人用于训练的人脸图像或用户可点击 Add Person 按钮将一个新的人加入训练集中。用户必须单击窗口中间某处才能切换到下一个（训练）模式。

因此，首先需要保持一个引用，这指向每个最新收集的人脸图像。可通过更新整型数组 `m_latestFaces` 来完成。该数组只保存每个人在大数组 `preprocessedFaces` 的索引。在此数组中还要存储镜像人脸，即引用倒数第二个人脸，而不是最后一个。应在将新人脸（和镜像面）添加到 `preprocessedFaces` 数组的代码中增加该功能。具体实现如下：

```
// Keep a reference to the latest face of each person.
m_latestFaces[m_selectedPerson] = preprocessedFaces.size() - 2;
```

注意，无论什么时候增加或删除一个人（例如，若用户点击 Add Person 按钮），`m_latestFaces` 数组总会相应地增长或收缩。下面来实现在窗口右侧（无论是在收集模式还是后面的识别模式）显示每个收集到的最新人脸，具体实现代码如下：

```
m_gui_faces_left = displayedFrame.cols - BORDER - faceWidth;
m_gui_faces_top = BORDER;
for (int i=0; i<m_numPersons; i++) {
    int index = m_latestFaces[i];
    if (index >= 0 && index < (int)preprocessedFaces.size()) {
        Mat srcGray = preprocessedFaces[index];
        if (srcGray.data) {
            // Get a BGR face, since the output is BGR.
            Mat srcBGR = Mat(srcGray.size(), CV_8UC3);
            cvtColor(srcGray, srcBGR, CV_GRAY2BGR);

            // Get the destination ROI
            int y = min(m_gui_faces_top + i * faceHeight,
                        displayedFrame.rows - faceHeight);
            Rect dstRC = Rect(m_gui_faces_left, y, faceWidth,
                             faceHeight);
            Mat dstROI = displayedFrame(dstRC);

            // Copy the pixels from src to dst.
            srcBGR.copyTo(dstROI);
        }
    }
}
```

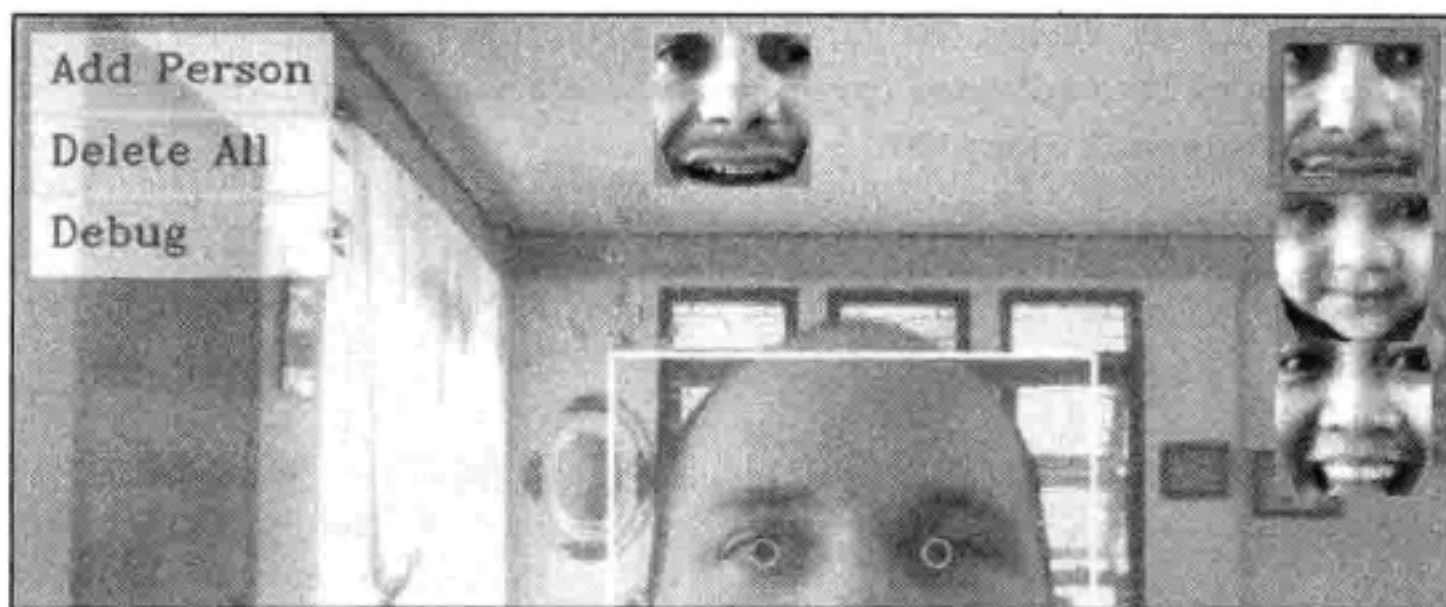
还需要在人脸四周用厚的红色边框来突出当前收集的人脸。具体实现如下：

```

if (m_mode == MODE_COLLECT_FACES) {
    if (m_selectedPerson >= 0 &&
        m_selectedPerson < m_numPersons) {
        int y = min(m_gui_faces_top + m_selectedPerson *
                    faceHeight, displayedFrame.rows -
                    faceHeight);
        Rect rc = Rect(m_gui_faces_left, y, faceWidth,
                      faceHeight);
        rectangle(displayedFrame, rc, CV_RGB(255,0,0), 3,
                  CV_AA);
    }
}

```

下面的部分截图是有几个人脸被收集后的常见效果。用户可在右上方点击任意一个人来收集更多的人脸图像：



(附彩图)

#### (4) 训练模式

当用户最后点击窗口的中间位置，人脸识别算法将在所收集到人脸图像上开始训练。但必须确保收集了足够多的人脸图像，否则程序可能会崩溃。一般情况下，只需确保在训练集中至少有一个人脸（这意味着至少有一人）。但 Fisher 脸算法需要进行多人之间的比较，所以如果训练集中少于两个人，它也将崩溃。因此，必须确定是否选择了 Fisher 脸算法。如果是，则至少需要两个人的人脸图像，否则至少需要一个人的一幅人脸图像。如果没有足够的数数据，则需返回到收集模式，使用户可在训练前增加更多的人脸。

为了确认收集的人脸图像中至少有两个人，可在用户点击 Add Person 按钮时进行验证，如果发现没有两个人，则需增加一个新的人，如果有至少两个人，但有人脸图像的人（也就是有一个人还没有任何人脸图像）少于两个，则不需要增加新人，只需增加相应的人脸图像即可。如果只有两个人，且使用了 Fisher 脸算法，则必须确保在收集模式中有个 m\_latestFaces 的引用指向最后一个人。当某个人没有人脸图像时，m\_latestFaces[i] 的初始值为 -1，一旦他的人脸图像添加进来，该数组元素对应的值就变成 0 或更大的值。具体实现如下：

```

// Check if there is enough data to train from.
bool haveEnoughData = true;
if (!strcmp(facerecAlgorithm, "FaceRecognizer.Fisherfaces")) {
    if ((m_numPersons < 2) ||
        (m_numPersons == 2 && m_latestFaces[1] < 0) ) {
        cout << "Fisherfaces needs >= 2 people!" << endl;
        haveEnoughData = false;
    }
}
if (m_numPersons < 1 || preprocessedFaces.size() <= 0 ||
    preprocessedFaces.size() != faceLabels.size()) {
    cout << "Need data before it can be learnt!" << endl;
    haveEnoughData = false;
}

if (haveEnoughData) {
    // Train collected faces using Eigenfaces or Fisherfaces.
    model = learnCollectedFaces(preprocessedFaces, faceLabels,
                                  facerecAlgorithm);

    // Now that training is over, we can start recognizing!
    m_mode = MODE_RECOGNITION;
}
else {
    // Not enough training data, go back to Collection mode!
    m_mode = MODE_COLLECT_FACES;
}

```

这个训练过程可能要花几分之一秒或几秒钟甚至几分钟，这取决于收集数据的多少。一旦对收集的人脸训练完成，人脸识别系统会自动进入识别模式。

### (5) 识别模式

在识别模式下，置信度会在预处理人脸旁边显示，这样用户就知道识别的可靠程度。如果置信度比设定的阈值高，将会人脸周围绘制一个绿色的矩形以突出显示此结果。用户可添加更多的人脸图像来做进一步的训练。如果他们点击 Add Person 按钮或选择其中一个存在的人，程序就会返回到收集模式。

现在要来获得所识别人的编号，并计算与前面重构人脸之间的相似性。为了显示置信度，需知道，基于 L2 的相似度介于 0 ~ 0.5 称为高置信度，在 0.5 ~ 1.0 称为低置信度，所以可用 1.0 减去介于 0.0 ~ 1.0 的置信度值，然后根据置信度，按比例来绘制填充的矩形，具体代码如下：

```

int cx = (displayedFrame.cols - faceWidth) / 2;
Point ptBottomRight = Point(cx - 5, BORDER + faceHeight);
Point ptTopLeft = Point(cx - 15, BORDER);

// Draw a gray line showing the threshold for "unknown" people.
Point ptThreshold = Point(ptTopLeft.x, ptBottomRight.y -

```



```

        (1.0 - UNKNOWN_PERSON_THRESHOLD) * faceHeight);
rectangle(displayedFrame, ptThreshold, Point(ptBottomRight.x,
        ptThreshold.y), CV_RGB(200,200,200), 1, CV_AA);

// Crop the confidence rating between 0 to 1 to fit in the bar.
double confidenceRatio = 1.0 - min(max(similarity, 0.0), 1.0);
Point ptConfidence = Point(ptTopLeft.x, ptBottomRight.y -
        confidenceRatio * faceHeight);

// Show the light-blue confidence bar.
rectangle(displayedFrame, ptConfidence, ptBottomRight,
        CV_RGB(0,255,255), CV_FILLED, CV_AA);
// Show the gray border of the bar.
rectangle(displayedFrame, ptTopLeft, ptBottomRight,
        CV_RGB(200,200,200), 1, CV_AA);

```

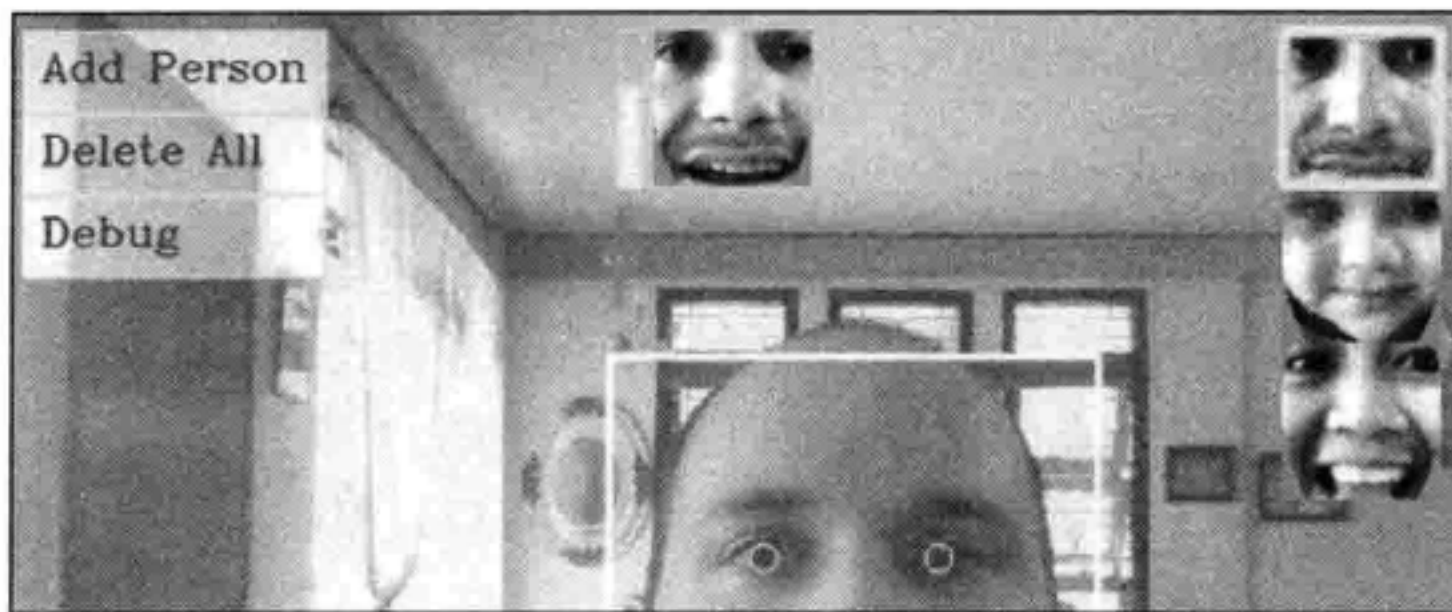
为了突出被识别的人脸，在该人脸四周画一个绿色的矩形框，具体代码如下：

```

if (identity >= 0 && identity < 1000) {
    int y = min(m_gui_faces_top + identity * faceHeight,
        displayedFrame.rows - faceHeight);
    Rect rc = Rect(m_gui_faces_left, y, faceWidth, faceHeight);
    rectangle(displayedFrame, rc, CV_RGB(0,255,0), 3, CV_AA);
}

```

下面的截图是运行在识别模式下的一个典型画面，置信度挨着预处理人脸正上方，并在右上角突出显示所识别出的人脸。



## 2. 检测和处理鼠标点击

前面已创建了所需的所有 GUI 组件，现在只需处理鼠标事件即可。当初初始化显示窗口时，需告诉 OpenCV 什么样的鼠标事件会回调 onMouse 函数。这里并不关心鼠标移动，只关心鼠标点击，所以首先要跳过不属于点击鼠标左键的鼠标事件，具体实现如下：

```

void onMouse(int event, int x, int y, int, void*)
{
    if (event != CV_EVENT_LBUTTONDOWN)
        return;

    Point pt = Point(x,y);

```

```

... (handle mouse clicks) ...

}

```

由于在画按钮时会获得按钮的矩形边框，可通过调用 OpenCV 的 `inside` 函数来检查鼠标点击位置是否在按钮区域。现在可检查所创建的每个按钮。

当用户点击 Add Person 按钮时，只需将 `m_numPersons` 变量加 1，并为 `m_latestFaces` 变量分配更多的空间。选择集合中的一个新的人，并启动收集模式（不管以前在哪个模式下）。

但会出现一个新的问题：为了确保在训练中每个人至少有一个人脸图像，如果所有人都有人脸图像，才为新人分配空间。解决该问题的方法为：查询 `m_latestFaces[m_numPersons-1]` 的值以确定每个人的人脸图像是否已收集。可通过下面的代码来实现此功能：

```

if (pt.inside(m_btnAddPerson)) {
    // Ensure there isn't a person without collected faces.
    if ((m_numPersons==0) ||
        (m_latestFaces[m_numPersons-1] >= 0)) {
        // Add a new person.
        m_numPersons++;
        m_latestFaces.push_back(-1);
    }
    m_selectedPerson = m_numPersons - 1;
    m_mode = MODE_COLLECT_FACES;
}

```

该方法可用于测试其他按钮的点击，如切换和调试标志，具体操作如下所示：

```

else if (pt.inside(m_btnDebug)) {
    m_debug = !m_debug;
}

```

为了处理 Delete All 按钮，需要清空各种数据结构，这些数据结构在主循环内（即，不通过鼠标事件的回调函数访问）。因此，若选择 Delete All 模式，则需删除主循环里的所有东西。另外还必须处理用户点击主窗口（即，不是点击按钮）事件。若点击右边的一个人，选择他就会转到收集模式，或者在收集模式下点击主窗口，则需转到训练模式。具体实现如下：

```

else {
    // Check if the user clicked on a face from the list.
    int clickedPerson = -1;
    for (int i=0; i<m_numPersons; i++) {
        if (m_gui_faces_top >= 0) {
            Rect rcFace = Rect(m_gui_faces_left,
                               m_gui_faces_top + i * faceHeight,
                               faceWidth, faceHeight);
            if (pt.inside(rcFace)) {
                clickedPerson = i;
                break;
            }
        }
    }
}

```

```

        }
    }
}
// Change the selected person, if the user clicked a face.
if (clickedPerson >= 0) {
    // Change the current person & collect more photos.
    m_selectedPerson = clickedPerson;
    m_mode = MODE_COLLECT_FACES;
}
// Otherwise they clicked in the center.
else {
    // Change to training mode if it was collecting faces.
    if (m_mode == MODE_COLLECT_FACES) {
        m_mode = MODE_TRAINING;
    }
}
}
}

```

## 8.2 总结

本章介绍创建实时人脸识别应用所需的所有步骤，并使用了一些基础算法来实现此应用，其中的一个处理步骤为预处理，它允许训练集和测试有所不同。通过人脸检测算法来找到摄像机图像中的人脸位置，接着采用几种形式的人脸预处理来减少不同光照条件、摄像机、人脸方向以及面部表情的影响。然后用预处理收集的人脸图像来训练一个基于特征脸或 Fisher 脸的机器学习系统。最后执行人脸识别，即给出一个未知的人的人脸图，看哪个人脸图像与它最相似，并给出置信度。

本章并没有提供一个命令行工具来处理离线方式的图像文件，而是单独通过一个实时 GUI 来将前面所有步骤组合在一起，以使用户能方便使用该人脸识别系统。读者可根据自己的需要来修改系统功能，例如，让计算机自动登录。若读者有兴趣，可提高识别的可靠性，这需要阅读有关人脸识别的最新会议论文，从而尽量提高应用每步的性能，直到完全满足所需的可靠性。比如，可提高预处理阶段的性能，或使用更多的高级机器学习算法，甚至是更好的人脸验证（face verification）算法，这些方法可在 <http://www.face-rec.org/algorithms/> 和 <http://www.cvpapers.com> 找到。

## 8.3 参考文献

- *Rapid Object Detection using a Boosted Cascade of Simple Features*, P. Viola and M.J. Jones, *Proceedings of the IEEE Transactions on CVPR 2001*, Vol. 1, pp. 511-518



- *An Extended Set of Haar-like Features for Rapid Object Detection*, R. Lienhart and J. Maydt, *Proceedings of the IEEE Transactions on ICIP 2002*, Vol. 1, pp. 900-903
- *Face Description with Local Binary Patterns: Application to Face Recognition*, T. Ahonen, A. Hadid and M. Pietikäinen, *Proceedings of the IEEE Transactions on PAMI 2006*, Vol. 28, Issue 12, pp. 2037-2041
- *Learning OpenCV: Computer Vision with the OpenCV Library*, G. Bradski and A. Kaehler, pp. 186-190, O'Reilly Media.
- *Eigenfaces for recognition*, M. Turk and A. Pentland, *Journal of Cognitive Neuroscience* 3, pp. 71-86
- *Eigenfaces vs. Fisherfaces: Recognition using class specific linear projection*, P.N. Belhumeur, J. Hespanha and D. Kriegman, *Proceedings of the IEEE Transactions on PAMI 1997*, Vol. 19, Issue 7, pp. 711-720
- *Face Recognition with Local Binary Patterns*, T. Ahonen, A. Hadid and M. Pietikäinen, *Computer Vision - ECCV 2004*, pp. 469-48

OpenCV是最常见的计算机视觉库之一，它提供了许多经过优化的复杂算法。本书对已掌握基本OpenCV技术同时想提高计算机视觉的实践经验的开发者来讲是一本非常好的书。每章都有一个单独的项目，其背景也在这些章节中进行了介绍。因此，读者可以依次学习这些项目，也可以直接跳到感兴趣的项目进行学习。

本书详细讲解9个实用的计算机视觉项目，通过本书的学习，读者可以创建各种可运行的项目原型，例如，实时的移动应用、增强现实、从视频中获得三维形状、跟踪人脸和眼睛、车牌识别等。

### 通过阅读本书，你将学到：

- 用简单的人脸、眼睛、皮肤检测、Fisher脸、人脸识别、三维头部方向和复杂的面部特征跟踪来分析人脸。
- 用人工智能（AI）方法（如SVM和神经网络）来实现车牌识别和光学字符识别。
- 针对桌面系统、iPhone或iPad，用简单的人工标识或复杂的原始图像来实现增强现实(AR)。
- 通过平面的二维摄像机和采用从运动相机中得到3D结构（SfM）的重投影方法来生成3D对象模型。
- 重新设计基于桌面的实时计算机视觉应用，使其更适用于Android和iOS移动应用。
- 用Xbox Kinect传感器将整个身体作为输入来执行人机交互。



**[PACKT]**  
PUBLISHING

投稿热线：(010) 88379604  
客服热线：(010) 88378991 88361066  
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com  
网上购书：www.china-pub.com  
数字阅读：www.hzmedia.com.cn

上架指导：计算机/图形图像

ISBN 978-7-111-47818-8



9 787111 478188 >

定价：59.00元