程序设计实践

设

计

报

告

课题名称：基于 OpenCV 的视频人脸抓取、统计、比对程序

学生姓名：王思恢

班　　级：2016211110

班内序号：03

学　　号：2016210271

日　　期：2018.6.30

1. 课题概述

1.1 课题目标和主要内容

本课题旨在以 OpenCV 等图像处理、机器视觉、机器学习工具为基础，实现对于视频流中人脸的识别、提取、表情与特征点分析、对比、统计等功能。

对于处理实时性要求不高的图像、视频文件，可以用 OpenCV 自带的基于 adaBoost 算法的 Haar 特征分类器进行人脸的识别。

对于处理实时性要求较高的视频流，需要运用 libfacedetect 库提供的接口进行人脸识别。基于 libfacedetect 库的人脸识别的优点是能够实时提供人脸的位置与特征点信息，而且 libfacedetect 库对于人脸姿态的多角度求解问题提供了一定的支持，能够识别多种角度的人脸并估计出角度信息。另一方面，libfacedetect 库的缺点是为了提升速度而牺牲了一定的准确性，这使得它更适合人数少、人脸面积大的场景（如电脑摄像头），而不大适合人数多、人脸面积小的场景（如毕业照）。在后一种情况下，需要在预处理时对图像进行放大才能使 libfacedetect 得到准确的识别结果，而这样一来 libfacedetect 与 OpenCV 相比并无明显的速度优势。

综上所述，在人数较多，不要求实时性的条件下，适用 OpenCV 进行人脸识别；在要求实时性且人脸较少的条件下，适用 libfacedetect 进行人脸识别。

人脸图像的提取功能主要依赖 OpenCV 相应的视频、图像读写与处理函数实现。

人脸 68 个特征点的提取可以通过 dlib 或 libfacedetect 进行实现。后者能同时探测人脸并提取特征点，且速度很快，但由于后者只提供了部分接口，在需要利用特征点进行人脸的对齐等操作时，仍然应当使用 dlib。

表情识别可以通过多种方式实现：

一是基于 OpenCV 的 Haar 特征分类器实现微笑检测，其原理与 OpenCV 的人脸检测相同。

二是运用机器学习的方法进行分类。目前运用的人脸表情数据来自 JAFFE，其样本较小，考虑适用支持向量机 SVM。目前测试过的方法包括：从图片中提取 HOG 特征，采用线性核函数进行训练；从图片中提取人脸 68 个关键点特征，采用线性核函数进行训练；从图片中提取人脸 68 个关键点特征，采用 sigmoid 核函数进行训练。从目前的测试情况看，采集人脸 68 个关键点，采用 sigmoid 核函数训练的效果相对较好，可以进一步调整训练参数，以实现对于愤怒、厌恶、害怕、高兴、平静、难过、惊讶 7 种表情的识别与分类。

对人脸的分类或相似度分析可以通过 OpenCV2.x 自带的类与函数进行实现。可以通过主

成分分析（PCA）或分析局部二进制编码直方图（LBPH）的方法计算人脸的相似度，将相似度低于阈值的人脸识别为陌生脸，分配一个 ID 标签，对该数据进行训练，这样循环往复，可以得到一个视频中出现的不同人脸的分类数目。在实际测试中，这种方法不具备实时性，有漏识别的情况，经常有把同一个人的不同角度的面部图像识别为不同人脸的现象，有时还会把无关信息误识别为人脸特征。

另一种实现人脸分类与相似度分析的办法是运用机器学习的方法。目前，利用 Cohn-Kanade 数据库的 123 个人，10708 张照片（大约 70%用于训练，其余用于测试）对深度学习网络 VGGFace 进行微调（finetuning），初步迭代实现了一个修改版的基于 Caffe 的人脸模型。如果能够解决库依赖与开发环境兼容性的问题，可以考虑在程序中调用 Caffe 训练的人脸识别模型。

基于上述功能，进一步封装、整合、二次开发，可以实现人脸的统计、将分好类的人脸保存到相册，标注视频中某些人出现的关键点并进行回放等功能。

关于立项时提出的探索内容：

1. 关于人脸 3D 建模：想法是将正面人脸经过归一化处理后，将其 68 个特征点与标准人脸 3D 模型的正面投影的 68 个特征点进行对比，相应调整 3D 模型的参数，得到对于人脸图像的拟合。将所得到的 3D 模型用图像中的对应点进行渲染，得到仿真的人脸 3D 模型。将该模型左右、上下旋转一定的角度，作为同一个人的人脸图像的训练样例，以改善人脸识别对于不同角度人脸的识别率。

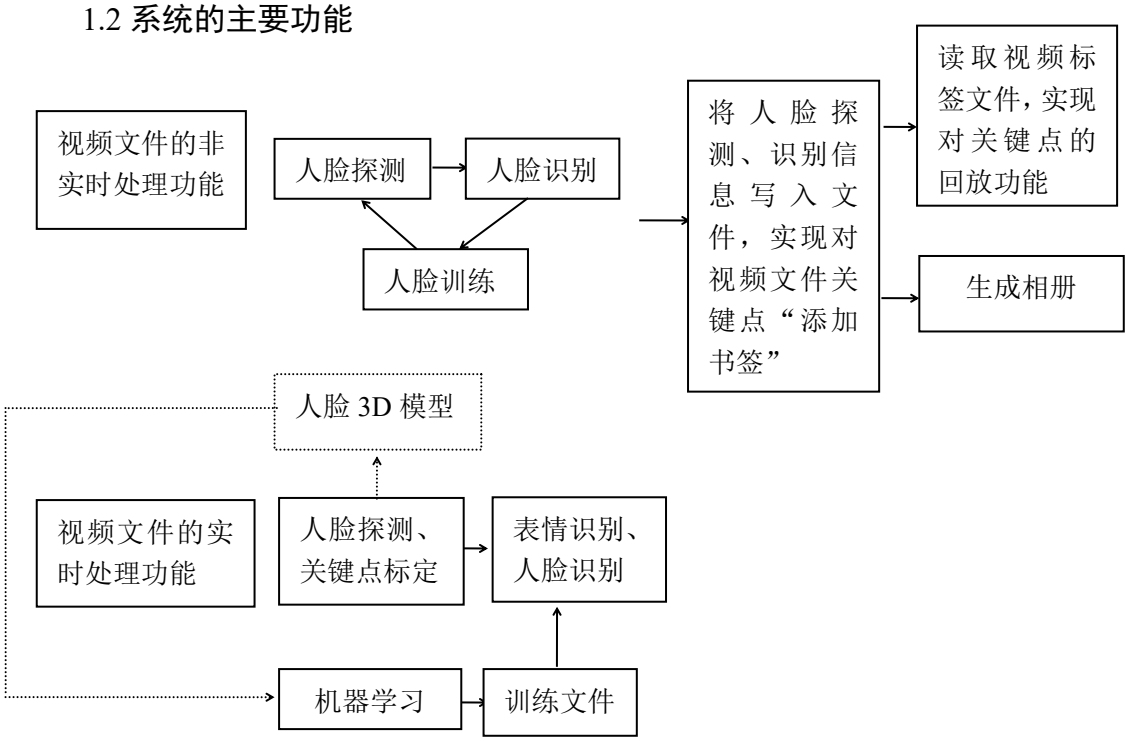
由于时间紧张，可能无法提出比较原创的算法，如果能够解决库依赖与兼容性的问题，将考虑用 3DMM、libigl、nanogui 来实现相关功能。

2. 关于视频关键帧提取、视频运动模糊消除的问题：尝试运用直方图比较的方法对相邻帧进行分类，但测试效果不佳。关于运动模糊消除的问题，文献中指出，运动模糊可以理解为运动物体与环境颜色间的卷积，如果能够估计出去卷积的方法，将能够一定程度上解决运动模糊的问题。此外，我个人的思路是通过边缘检测和 Sift/Surf 关键点的检测，估计出运动速度的大小和方向，找出不同帧之间的对应点和对应轮廓。此后，参考将人脸按照关键点进行对齐并求平均脸的做法，将对应点对齐并求出某种意义上的均值，看看实际效果如何。由于时间有限，编程能力与工具有限，暂时没有实现相应的算法和功能。

综上所述，本程序基于 OpenCV、dlib、libfacedetect、caffe 等工具实现。如果实现人脸 3D 建模功能，还需要 3DMM、libigl、nanogui 等工具。上述工具还依赖 boost、hdf5 等众多库文件。

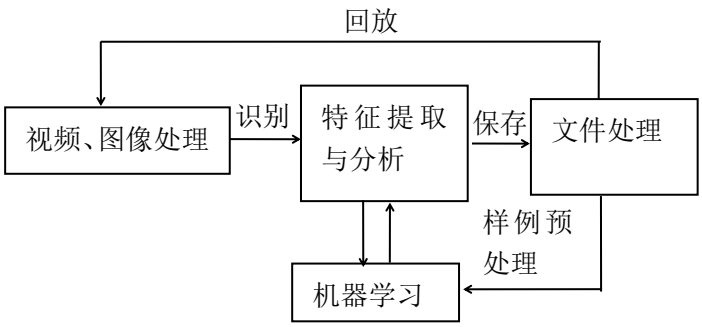
在实际测试中，caffe 只与 VS2013 兼容，libigl 等工具则要求 VS2015 update2 以上版本。OpenCV2.x 的 contrib 库中支持人脸的分类功能，在 OpenCV3.x 中这一功能又被取消了。OpenCV2.x 的某些版本中，CvSVM 对象会造成内存释放的错误；OpenCV3.x 的某些版本中，SetImageROI 后执行 cvSave 会抛出运行时错误，有些版本的人脸识别的 xml 文件实测报错，无法使用，由于以上种种兼容性、稳定性问题，本次程序设计可能需要建立一个以上的工程文件，使用 VS2013 以及 VS2017，OpenCV2.4.10、OpenCV2.4.13、OpenCV3.1.0 等工具综合完成。

1.2 系统的主要功能



2. 系统设计

2.1 系统总体框架



2.2 系统详细设计

1.VideoMarker 部分的设计：

功能模块：

在本次程序设计实践中，对于图片、视频文件的非实时性的功能被整合在一个界面下。其功能模块如下图所示：

Video Marker 程序功能模块

<table>
<tr><td>界面部分</td><td colspan="6">内核部分</td></tr>
<tr><td rowspan="3">功能选择
窗体重绘
控件使能
程序控制的总体逻辑</td><td colspan="2">图像与视频静态帧</td><td colspan="2">动态视频</td><td colspan="2">视频标记与回放</td></tr>
<tr><td>处理</td><td>转化为灰度图
多种边缘检测
操作撤销与恢复
保存图片</td><td>播放控制</td><td>播放、暂停、播放倍速调整、逐帧前进或后退</td><td>写入文件</td><td>创建.vdm文件,将与人脸识别相关的信息（帧的信息、人脸矩形位置信息、人脸分类信息）写入文件</td></tr>
<tr><td>检测</td><td>正面人脸检测
微笑检测</td><td>检测</td><td>正面人脸检测
微笑检测
运动检测</td><td>读取文件</td><td>根据.vdm文件保存的信息回放识别到人脸的帧，将相关帧生成html相册</td></tr>
</table>

VideoMarker 依赖的类与方法：

对于图像与静态帧的处理功能，主要依靠 OpenCV 提供的 Mat 类与相关库函数进行实现。

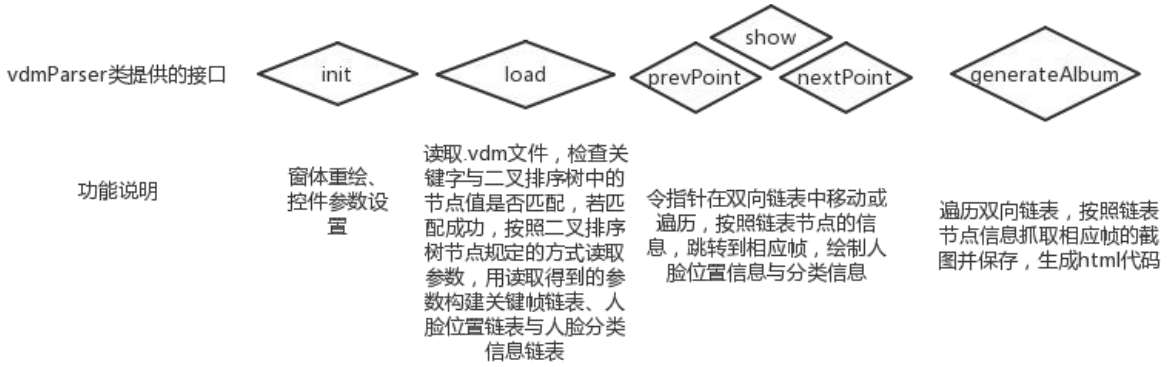
对于视频的播放控制功能，主要依靠 OpenCV 提供的 VideoCapture 类与相关库函数进行实现。

对于人脸检测与微笑检测功能，主要依靠 OpenCV 提供的 CascadeClassifier 类与相关的库函数 detectMultiScale 进行实现。

对于运动检测功能，主要依靠 OpenCV 提供的 features2D 模块与库函数 calcOpticalFlowPyrLK 进行实现。

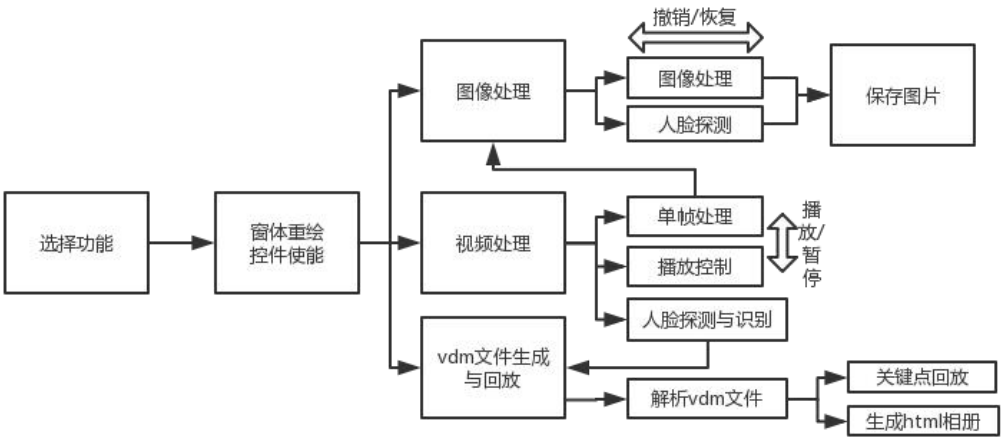
对于视频的标记与回放功能，主要依靠自行定义的 vdmParser 类进行实现。

以下是 vdmParser 类的功能简图：



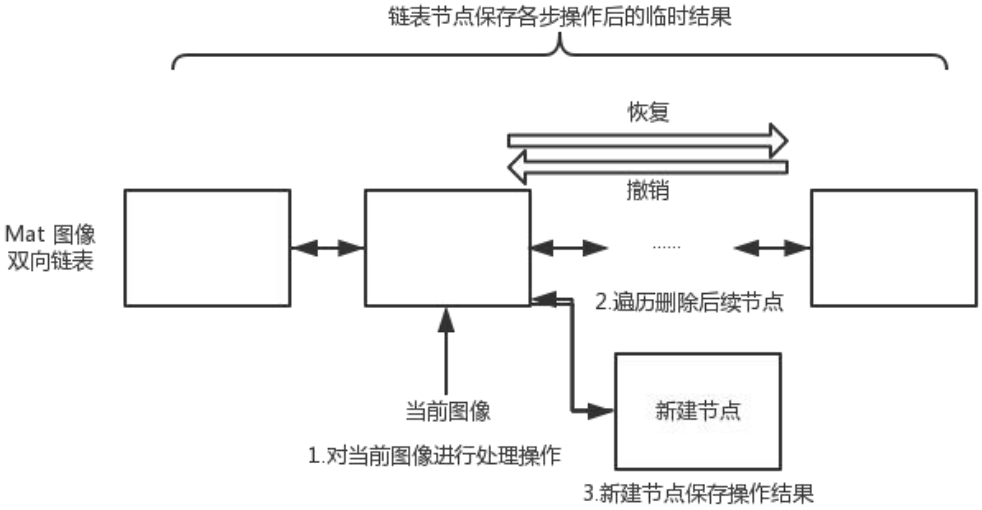
Video Marker 的程序流程图：

Video Marker 部分的程序流程图如下图所示：

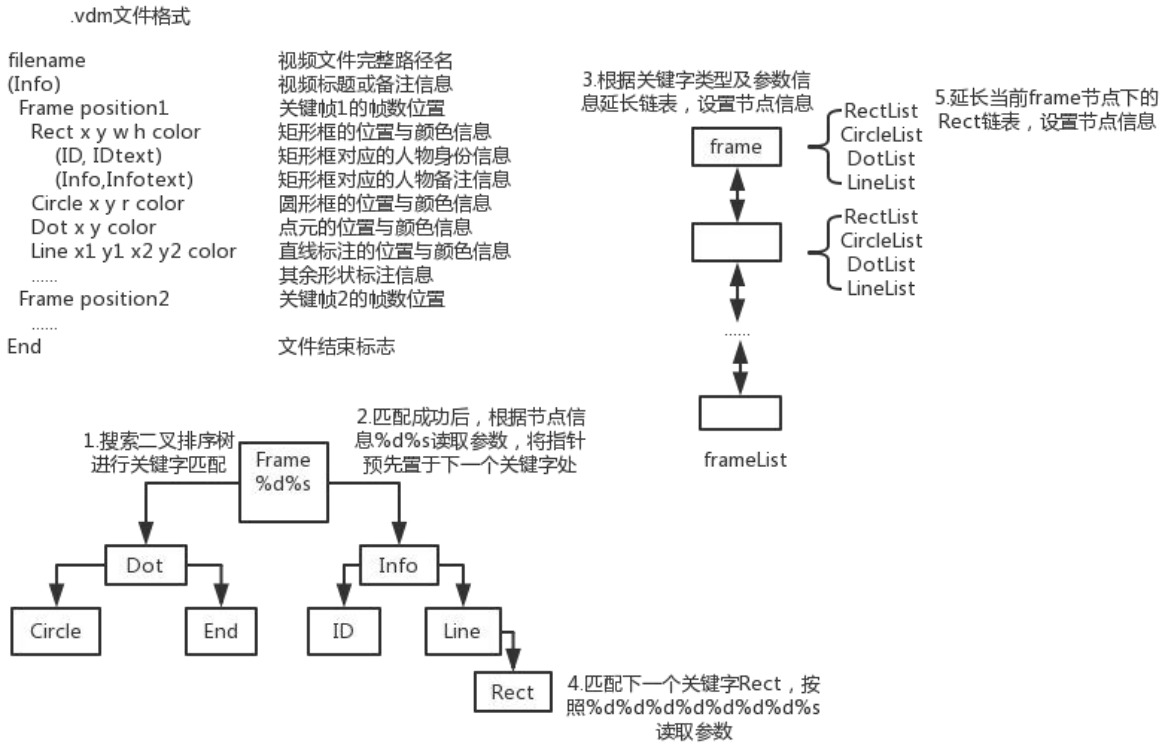


Video Marker 使用的存储结构：

在图像处理部分，操作的撤销与恢复功能的实现用到了链表这一存储结构。



vdm 文件的读取与解析功能的实现用到了链表和二叉排序树的存储结构。



2. faceDetect 与 faceRec 部分的设计：

功能模块：

对于来自摄像头的视频流的实时处理，本程序实现的主要功能有：人脸 68 个关键点的标注、人脸角度信息的输出、情绪的检测与分类、人脸的运动追踪、人脸识别初步（区分出自己的脸与其他人的脸）。

在上述功能中，初步的人脸识别功能基于 OpenCV2.4.13 提供的 LBPHFaceRecognizer 类进行实现。另一方面，OpenCV2.4.13 提供的支持向量机 CvSVM 类在实际调测中会出现运行时错误，因而在实现情绪的检测与分类功能时，我转而依靠 OpenCV3.1 提供的 Ptr<SVM>。基于这样的原因，在本次程序设计中，对摄像头视频流的处理由两个独立的工程共同完成，其中一个基于 OpenCV2.4.13，专门完成人脸识别的初步功能，另一个基于 OpenCV3.1，实现 68 个关键点标注、人脸角度信息输出、情绪检测与分类、人脸运动追踪等功能。

Face Detect 程序功能模块

界面部分	内核部分		
窗体重绘、控件调整、信息输出、全局变量设置与程序总体逻辑	基于libfacedetect的人脸关键点标注和人脸角度信息输出	基于OpenCV3.1的机器学习模块中的SVM和JAFFE数据集训练得到的情绪分类器	基于OpenCV的features2D模块实现的人脸运动追踪功能

Face Rec 程序功能模块

界面部分	内核部分
窗体重绘、控件调整、信息输出、全局变量设置与程序总体逻辑	基于OpenCV2.4.13的contrib库中的faceRecognizer类，实现认出自己的脸的功能

依赖的类与方法：

人脸的 68 个关键点标注和人脸角度计算的功能基于 libfacedetect 提供的接口实现。

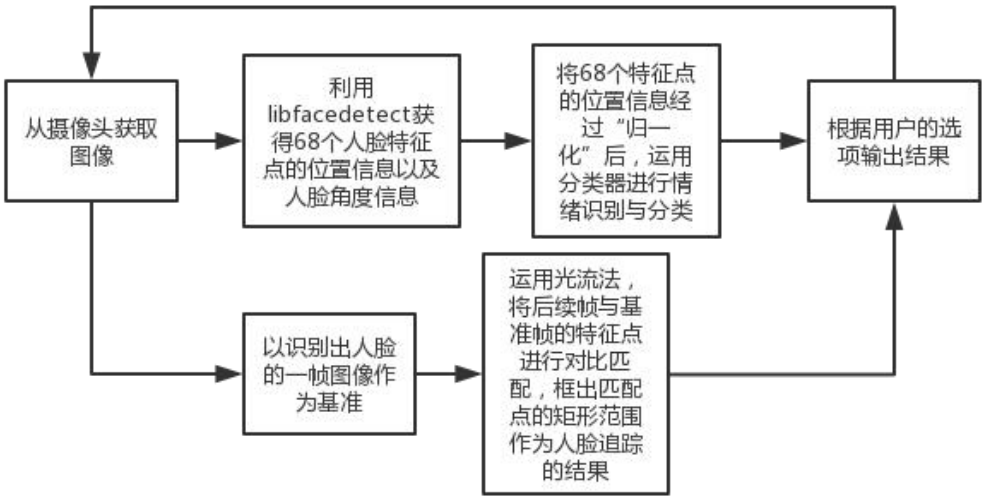
情绪识别、分类的功能实现分为两步：第一步，生成训练样例，提取样例的特征，训练支持向量机 SVM，最终将训练得到的分类器的数据保存为 xml 文件；第二步，提取人脸图像的特征，载入情绪分类数据，进行情绪识别、分类。

在训练阶段和测试阶段，提取人脸图像特征时，都用到了 libfacedetect 提供的标注人脸特征点的接口。在训练、生成数据文件、载入数据文件与分类的过程中，都用到了 OpenCV3.1 的 Ptr<SVM>类。

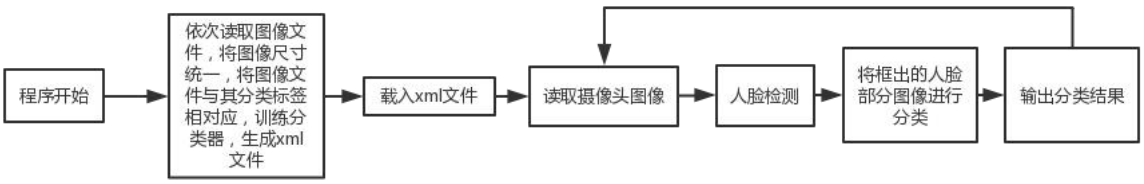
在人脸的运动追踪功能的实现过程中，运用了 OpenCV 的 features2D 模块提供的角点检测函数 goodFeaturesToTrack 和光流法实现运动检测的函数 calcOpticalFlowPyrLK。

在识别自己的脸的功能实现中，运用了 OpenCV2.4.13 的 contrib 库中提供的 LBPHFaceRecognizer 类。

Face Detect 的程序流程图：



FaceRec 的程序流程图：



使用的存储结构：

在人脸的运动追踪功能中，保存基准帧与后续匹配帧的特征点信息时，用到了顺序表 vector 的数据结构。

在保存人脸 68 个特征点的位置信息时，用到了 C 语言风格的数组作为存储结构。

在读取图像与标签，训练分类器的过程中，用到了 vector<Mat>与 vector<int>作为存储结构，保存每幅训练图像的内容与分类信息。

2.3 关键算法分析

算法 1：由视频文件生成 vdm 文件：

该算法没有独立的函数实现，而是作为窗体响应 OnTimer 事件的一个分支过程出现。

算法功能：

对于视频中探测出人脸的相应帧，将帧的位置信息、人脸的位置信息写入 vdm 文件，将所有探测出的人脸图像进行特征分类，将这一分类信息也一并写入 vdm 文件。

算法基本思想：

遍历整个视频，运用 OpenCV 提供的函数 detectMultiScale 进行人脸检测，按照事先设定好的 vdm 文件的格式写入帧的位置信息、人脸的矩形框的位置信息。

维护三个顺序表 vector<Mat>person、vector<int>Id、vector<list<Mat> >Category。每检测到一个人脸，将其图像添加到 person 中。

如果此时人脸分类计数 personCnt 为 0，说明人脸分类器没有经过训练，则分配该人脸的 ID 为 1，将该 ID 值添加到 Id 中，人脸分类计数 personCnt 加一，用此时得到的 person 与 Id 训练人脸分类器。

如果此时人脸分类计数 personCnt 不为 0，说明人脸分类器已经被训练过，则用分类器预测当前人脸的分类。如果分类（不）可靠度 confidence 超过某个阈值，说明分类不可靠，应将当前人脸视为一张新脸，此时为该人脸分配新的 ID，将 ID 值添加到 Id 中，令人脸分类计数 personCnt 加一，在 Category 中新增一个图像链表元素用于保存所有与当前人脸分类相同的人脸图像。

如果分类可靠度 confidence 在阈值范围内，说明分类可靠，当前人脸是一张熟悉的脸。通过人脸分类器计算的分类值给出当前人脸对应的 ID 值，将 ID 值添加到 Id 中，在 Category 对应此 ID 的位置处的链表中添加当前人脸图像。

用更新后的 person 与 Id 训练人脸分类器，如此循环往复，直至视频遍历结束，将 ID 信息也按照格式写入 vdm 文件。

算法空间、时间复杂度分析：

时间复杂度：对于人脸分类器 Ptr<FaceRecognizer>的训练函数 train 与预测函数 predict 是 OpenCV 提供的库函数，其时间复杂度不易估计。如果认为人脸分类器的训练与预测的时间与 person、Id 的规模无关，也与已有人脸分类数 personCnt 的大小无关，则本算法的时间复杂度应当为 O(n)，其中 n 为视频文件中出现的人脸总数。

然而在实际测试中会发现，一旦视频文件的帧数稍稍增加，用本算法进行处理所耗费的时间就大大延长。这说明本算法的时间复杂度至少在 $O(n^2)$或 $O(n^2)$以上。

空间复杂度：数组 person、Id、Category 的最终规模都等于视频中出现的人脸总数 n，如果不考虑库函数所用的算法是否申请临时的存储空间，则本算法的空间复杂度为 O(n)。

代码逻辑：

```
...//窗体OnTimer事件响应函数
else if(程序处于生成vdm文件的模式&&视频遍历未结束){
   从VideoCapture对象读取一帧图像img;
   对img进行图像预处理（转为灰度图，直方图均衡）;
   用detectMultiScale进行人脸探测，将探测得到的人脸位置矩形信息保存在vector<Rect>对象faces中;
   if(faces非空)fprintf_s(...);//在vdm文件中写入帧的位置信息
   for(遍历数组faces){
      fprintf_s(...);//在vdm文件中写入人脸的位置信息
      截取人脸部分图像faceClip,将其尺寸调整为指定大小;
      if(person数组为空){
         person.push_back(faceClip);
         Id.push_back(++personCnt);
         model->train(person,Id);//更新person,Id数组,训练人脸分类器model
         album=new list<Mat>;
         album->push_back(faceClip);
         category.push_back(*album);//创建新链表，保存与当前人脸分类相同的人脸图像
         fprintf_s(...);//将人脸ID信息写入vdm文件
      }
      else{
         model->predict(faceClip,predictedID,confidence);//用训练过的分类器对当前人脸图像进行预测
         if(confidence>60){//预测不可靠
            执行与person数组为空时相同的代码
         }
         else{
            Id.push_back(predictedID);
            category[predictedID-1].push_back(faceClip);
            ...//除以上两处改动外,其余代码与person数组为空时相同
         }
         model->train(person,Id);
      }
   }
}
else if(程序处于生成vdm文件的模式&&视频遍历结束){
   fprintf_s(...);//在vdm文件中写入结束标志
   fclose(...);
   改变程序状态为播放视频模式,关闭Timer;
}
```

算法 2：读取 vdm 文件，通过解析 vdm 文件，实现关键帧回放和相册生成功能：

函数名：通过 vdmParser::init()、vdmParser::load()、vdmParser::prevPoint()、vdmParser::nextPoint()、vdmParser::generateAlbum()联合实现，在 vdmParser.h 中定义与实现了上述函数。

算法功能：打开 vdm 文件，读取关键帧信息、人脸位置信息、人脸分类信息（由 vdmParser::load()实现），实现关键帧回放（由 vdmParser::prevPoint()和 vdmParser::nextPoint()实现）、相册生成（由 vdmParser::generateAlbum()实现）功能。

算法基本思想：

vdm 文件解析算法的基本思想：

对 vdm 文件的解析本质上就是对于有一定语法规则的字符串的解析。

我的想法是，能否建立一个用于匹配的数据结构，该结构的每个节点实际上保存了某个匹配关键字或语法单元的语法特性，可以用来指导下一步的匹配或解析行为。也就是说，整个文件解析的过程应当表示为：关键字匹配、读取关键字节点的属性信息、在属性信息的指导下进行解析、下一次匹配。

落实到本次程序设计实践中，我构造了一个平衡二叉排序树，该排序树的各节点即为 vdm 文件格式的 8 个“保留字”：Circle、Dot、End、Frame、ID、Info、Line、Rect。在构造时，应保证每个节点的保留字按字典序大于其左子节点的保留字，按字典序小于其右子节点的保留字。

在每个保留字节点中，保存着指导解析行为的数据，例如 Frame 节点为”%d\n%s”，Rect 节点为”%d%d%d%d%d%d%d\n%s”，当 vdm 文件中的关键字与保留字节点匹配后，将按照节点数据指导程序读取参数，然后递归地读取、匹配关键字，这样，可以在一定程度上减少手工编写的代码量和过程分支的量，可以在程序需要更改时减少出错的可能性，可以实现数据对于程序的某种“二次编程”。

利用 vdm 文件实现关键帧回放：

一边解析 vdm 文件，一边按照相应的关键字和参数延长帧链表 frameList 和矩形框链表 rectList。令帧链表 frameList 的每一个节点都对应一个矩形框链表 rectList。在回放时，读取帧链表 frameList 的节点信息，将视频跳转到对应帧，遍历该节点对应的 rectList 的全部节点，绘制人脸矩形框，打印人脸分类信息，就实现了对于关键帧信息的标注。然后通过遍历 frameList 中的全部节点，就实现了对于全部关键帧的回放功能。

算法空间、时间复杂度分析：

时间复杂度：vdmParser::load()读取 vdm 文件中关键字和参数的过程的时间复杂度与 vdm 文件中的关键字与参数的总数 n 有关，其时间复杂度应当为 O(n)。由于所有关键字的字长都很小，每次关键字匹配过程花费的时间大致相当，因而关键字匹配过程的时间复杂度与 vdm 文件中的关键字总数 m 有关，而我们也可以用 O(n)来估计。全部构造链表的过程的时间复杂度与所有链表的全部节点总数有关，我们也可以用 O(n)来估计。综上所述，vdmParser::load()的时间复杂度为 O(n)。

vdmParser::prevPoint()和 vdmParser::nextPoint()需要与 OnTimer 事件结合起来，才能完成对全部关键帧的回放功能。假如 vdm 文件中的关键字的总数为 n，则整个回放过程的时间复杂度为 O(n)。

vdmParser::generateAlbum()完成了对于所有链表的遍历，其时间复杂度与全部链表的节点数总和有关，可认为其时间复杂度为 O(n)。

空间复杂度：

vdmParser::load()创建链表的过程的空间复杂度为 O(n)；vdmParser::prevPoint()和 vdmParser::nextPoint()的空间复杂度为 O(1)，vdmParser::generateAlbum()写入 html 文件的过程的空间复杂度最多为 O(n)，其中 n 为视频中标记的人脸总个数。

代码逻辑：

vdmParser::load()的代码逻辑：

vdmParser::load()用到了 vdmParser 类的私有辅助函数 match()、readTitle()、readFrame()、readShape()。

辅助函数 match()的代码逻辑：

```
errorInfo match(const char* ch) {
	int i = 0;
	treeNode *ptr = &tagNode[root];//从二叉排序树的根节点开始匹配搜索
	while (ch[i] != '\0') {
		if (ptr == NULL) {//匹配失败
			return errorInfo("Failed to parse the file!");
		}
		if (ch[i]<ptr->tagName[i]) {//如果当前字符不匹配，搜索左子树或右子树，从i=0处重新匹配
			ptr = ptr->left;
			i = 0;
		}
		else if (ch[i]>ptr->tagName[i]) {
			ptr = ptr->right;
			i = 0;
		}
		else {//当前字符相互匹配，继续匹配下一个字符
			++i;
		}
	}
	if (ptr->tagName[i] != '\0') {//所有字符都匹配成功，但匹配字符的长度小于二叉排序树中关键字的长度，仍为匹配失败
		return errorInfo("Failed to parse the file!");
	}
	else {//若匹配成功
		switch (ptr->id) {//根据关键字的不同
		case 0:
			if (fscanf_s(fp, ptr->parameter, &x1, &y1, &radius, &r, &g, &b, nextTag, 11) != 7) {//按照节点数据读入形状参数
				.....//处理文件格式错误的情况
			}
			else {
				return errorInfo(0);
			}
			break;
		}
	}
}
```

load()的总体逻辑：

```
errorInfo load(const char* filename, bool isRelativePath = true) {
	if (isRelativePath) {//如果提供的路径是相对路径
		char path[255];
		char fileName[360];
		_getcwd(path, 255);//获得当前程序所在路径
		sprintf_s(fileName, 360, "%s%s", path, filename);//补齐绝对路径
		fopen_s(&fp, fileName, "rb");//打开vdm文件
	}
	else {
		fopen_s(&fp, filename, "rb");
	}
	if (fp == NULL) {
		return errorInfo("Can not open the file!");
	}
	readTitle();//读入vdm文件第一行或前两行，视频的路径信息与备注信息
	readFrame();//逐帧读取vdm文件信息
	return errorInfo(0);
}
```

readFrame()的代码逻辑：

```
void readFrame() {
        if (_case == 3) {//只有match()得到的关键字分类编号为3，说明关键字为"Frame"
            frameList.add(*(new frame(framePos)));//延长关键帧链表
            _case = match(nextTag).errCode;//读取下一个关键字的分类编号
            if (_case == 5) {//如果下一个关键字是"Info"，还需要读取视频备注信息
                frameList.tail->data.setInfo(info);
                _case = match(nextTag).errCode;
            }
            readShape();//然后读取形状标注信息
        }
    }
```

readShape()的代码逻辑：

```
void readShape() {
    if (_case == 0 || _case == 1 || _case == 6 || _case == 7) {//如果关键字为Circle、Dot、Line、Rect之一
        while (_case == 0 || _case == 1 || _case == 6 || _case == 7) {//持续读取形状信息
            switch (_case) {//分别处理不同的形状信息
            case 0:
                frameList.tail->data.circleList.add(*(new _circle(x1, y1, radius, r, g, b)));//在相应的链表中添加信息
                _case = match(nextTag).errCode;//读入下一个关键字
                if (_case == 4) {//如果下一个关键字是ID
                    frameList.tail->data.circleList.tail->data.setID(ID);//添加该形状标记的ID信息
                    _case = match(nextTag).errCode;//再读入下一个关键字
                    if (_case == 5) {//如果下一个关键字是Info
                        frameList.tail->data.circleList.tail->data.setInfo(info);//添加该形状标记的备注信息
                        _case = match(nextTag).errCode;再读入下一个关键字
                    }
                }
                else if (_case == 5) {//如果下一个关键字是Info，处理方法与上面相同
                    frameList.tail->data.circleList.tail->data.setInfo(info);
                    _case = match(nextTag).errCode;
                    if (_case == 4) {
                        frameList.tail->data.circleList.tail->data.setID(ID);
                        _case = match(nextTag).errCode;
                    }
                }
                break;
                ......//处理其他形状信息的代码，原理相同
            }
        }
    }
    if (_case == 3) {//直到读取到下一个"Frame"关键字，跳转到readFrame()
        readFrame();
    }
    ......
}
```

vdmParser::nextPoint()的代码逻辑：

```
void nextPoint(bool& showFlag){
        建立若干链表节点指针变量;
        if (currentFrame->next == NULL){
            处理没有下一个关键帧的情况
        }
        else{
            currentFrame = currentFrame->next;//读取下一个关键帧
            cap.set(CV_CAP_PROP_POS_FRAMES, currentFrame->data.position);//视频跳转到相应帧
            cap.read(img);//将该帧图像读取到Mat对象img中
            pc = currentFrame->data.circleList.head->next;//初始化各链表节点指针，准备遍历
            pd = currentFrame->data.dotList.head->next;
            pl = currentFrame->data.lineList.head->next;
            pr = currentFrame->data.rectList.head->next;
            while (pc != NULL) {//遍历circleList
                circle(img, Point(pc->data.px, pc->data.py), pc->data.radius, Scalar(pc->data.clrB, pc->data.clrG, pc->data.clrR));
                //根据节点信息绘制相应的形状
                if (pc->data.shapeID[0] != 0) {
                    putText(...);//打印ID信息
                }
                if (pc->data.shapeInfo[0] != 0) {
                    putText(...);//打印备注信息
                }
                pc = pc->next;
            }
            while (pd != NULL) {
                打印所有形状dot的信息
            }
            while (pl != NULL) {
                打印所有形状line的信息
            }
            while (pr != NULL) {
                打印所有矩形框的信息
            }
            ((CSliderCtrl*)_m_slid)->SetPos(currentFrame->data.position);//令视频播放的进程条的位置与帧的位置相一致
            im = img;//以下为利用CvvImage.h提供的接口在MFC中显示Mat对象的图像内容的标准代码
            _m_cWnd->GetClientRect(&_m_rect);
            cvIm.CopyOf(&im, 1);
            cvIm.DrawToHDC(_m_cWnd->GetDC()->GetSafeHdc(), &_m_rect);
            ReleaseDC(_m_hWnd, _m_cWnd->GetDC()->GetSafeHdc());
        }
    }
```

vdmParser::prevPoint()的代码逻辑与 vdmParser::nextPoint()的代码逻辑大致相同，不再重复。

vdmParser::generateAlbum()的代码逻辑：

```
 void generateAlbum(const string& filename){
            获取当前文件夹路径;
            根据系统时间，在当前路径下创建子文件夹;
            fopen_s(&htmWriter, filename.c_str(), "wb");//创建html文件
            fprintf_s(htmWriter, "%s\r\n", "<html>");
            fprintf_s(htmWriter, "%s%s%s\r\n", "<title>",videoName,"</title>");//写html文件头
            for (_frameList::_frameNode *p = frameList.head->next; p != NULL; p = p->next){//遍历关键帧链表
                _m_cap.set(CV_CAP_PROP_POS_FRAMES, p->data.position);//读取每个关键帧的图像
                _m_cap >> _m_frame;
                sprintf_s(imgName,"%s%s%d%s",imgDirectory,"\\img",p->data.position,".jpg");//保存关键帧图像
                imwrite(imgName, _m_frame);
                fprintf_s(htmWriter, "%s%s%s%d%s\r\n", "<img src=\"",imgDirectory,"\\img",p->data.position,".jpg\">");
                //在html文件中添加显示该图像的代码
            }
            fprintf_s(htmWriter, "%s\r\n", "</html>");//写html文件尾
            fclose(htmWriter);
        }
    };
```

算法 3：人脸运动追踪功能：

函数名：void track(Mat& img1, Mat& img2, Rect& r1, Rect& r2)

算法功能：从基准帧 img1 的矩形区域 r1 中寻找特征点，在帧 img2 中寻找相匹配的特征点，将匹配特征点所在的矩形框位置信息保存到 r2 中。

算法基本思想：

OpenCV 提供的 detectMultiScale 函数的实时性不够好，其解决思路之一是尽量减少调用

detectMultiScale 的次数。一旦运用 detectMultiScale 函数探测到人脸后，可以用特征点提取与运动追踪的办法后续跟踪人脸，而不应一直调用 detectMultiScale 函数来追踪人脸。这一“只调用 detectMultiScale 函数一次做人脸探测，其余时间用 track 做人脸后续追踪”的想法，就是本算法的初衷。

运动追踪的一大优势是受人脸的角度与遮挡情况的约束较小，这正好弥补了人脸探测功能的不足。

当然，目前运用本算法实现的运动追踪还有一些缺点，比如将 r1 中的背景特征点与 img2 中的背景特征点相互匹配，导致 r2 矩形框过大，不够精确；在复杂场景下，运用 detectMultiScale 与运用 track 的时机与条件不够明确，等等。

算法空间、时间复杂度分析：

track()调用的库函数 goodFeaturesToTrack 与库函数 calcOpticalFlowPyrLK 的时间复杂度、空间复杂度不易计算，若不予考虑，则 track()的时间复杂度与空间复杂度均为 O(n)。

代码逻辑：

```
void track(Mat& img1, Mat& img2, Rect& r1, Rect& r2){
    Mat region = img1(r1);//将基准帧img1的矩形区域r1保存为Mat对象region
    cvtColor(region, region, CV_BGR2GRAY);//将region预处理，转为灰度图
    goodFeaturesToTrack(region, features, 500, 0.01, 10.0);//探测region中的特征点，
    //将特征点在region中的位置信息保存到features数组中
    for (int i = 0; i < features.size(); i++){//还原特征点在img1中的位置坐标
        features[i].x += r1.x;
        features[i].y += r1.y;
    }
    calcOpticalFlowPyrLK(img1, img2, features, match, status, err);
    //在帧img2中寻找与基准帧img1的特征点features相匹配的特征点，
    //保存其位置信息到数组match
    找出数组match中点的最左，最右、最上、最下的位置信息
    r2.x = xMin;//构造r2矩形的信息
    r2.y = yMin;
    r2.width = xMax - xMin;
    r2.height = yMax - yMin;
}
```

2.4 其他

本程序运用了 STL 提供的 vector 与 list，也用到了模板类，这提高了代码的简洁性，减少了代码运行出错的可能性。

实际上，在我最初编写 vdmParser 部分的代码时，并没有使用 STL。我发现，我自行实现的链表类 circleList、rectList、dotList、lineList 自身运行并不会出错，但如果我想要实现一个模板类 linkedList，把 circleList、rectList 等等写成 linkedList<circle>、linkedList<rect>的形式，或者再引入继承机制，给 circle、rect 类赋予基类 shape，然后用 linkedList<shape>定义各个链表的属性与方法，在实际调试中会发现屡屡出错。通过调试，我发现一个奇异的现象是，有时 linkedList<circle>的对象在添加新节点时表现出我预期的行为，linkedList<rect>的对象在添加新节点时却表现出完全不同的错误的行为，而且正确和错误的角色还会发生变化，比如有时反过来 linkedList<rect>正确而 linkedList<circle>错误。我认为这不是关键字冲突造成的（比如我定义的 shape、rect 与 OpenCV 中的名字发生重复）。

之前编程的经验也让我意识到，模板类的实现机制与普通类有一定的差异。自行编写的代码在实现模板类时，可能引入一些奇怪的问题。或者非模板类所支持的、不会报错的编程风格，在模板类中可能并不能完全支持。

基于这样的考虑，我转而使用 STL 完成了后面的代码编写，这大大减少了我调试 bug 花费的时间。

关于情绪分类器的训练：

在本次程序设计中，我运用 OpenCV3.1 的机器学习模块，基于 JAFFE 数据集，训练了自己的基于 SVM 的情绪分类器。训练与生成 xml 文件的过程并没有呈现在最终的程序界面中，现介绍如下。

首先，下载 JAFFE 数据集，对数据进行预处理。具体而言，应建立两个文本文件 imgList.txt 和 imgLabels.txt，前者给出每张训练图片的完整路径，后者给出每张训练图片对应的情绪分类标号。在这里，我将 angry、disgust、fear、happy、neutral、sad、surprise 分别按照 1~7 进行标号。

然后，用支持向量机 SVM 进行训练，其核心代码如下：

```
void train(){
	ifstream inLabels("imgLabels.txt"), inimgs("imgList.txt");
	string imgName;
	int imgLabel;
	int cnt = 0;
	int lblVec[213][1];//保存每张图片的情绪分类标号
	float LMVec[213][136];//保存每张人脸的68个关键点的归一化位置信息
	Ptr<SVM> svm = SVM::create();//建立SVM
	svm->setType(SVM::C_SVC);
	svm->setKernel(SVM::RBF);//采用sigmoid核函数
	svm->setGamma(0.01);
	svm->setC(10.0);
	svm->setTermCriteria(TermCriteria(CV_TERMCRIT_EPS, 1000, FLT_EPSILON));
	while ((inimgs >> imgName) && (inLabels >> imgLabel)){//依次读取图像及情绪分类信息
		++cnt;
		Mat src = imread(imgName, 0);
		resize(src, src, imgSize);//将图像调整至标准大小
		Mat gray;
		gray = src.clone();
		if(gray.channels()==3)cvtColor(gray, gray, CV_BGR2GRAY);//只有训练图片是彩色图片时，才进行灰度转化
		int * pResults = NULL;//调用libfacedetect库的接口的标准代码
		unsigned char * pBuffer = (unsigned char *)malloc(DETECT_BUFFER_SIZE);
		if (!pBuffer){
			fprintf(stderr, "Can not alloc buffer.\n");
			return;
		}
		pResults = facedetect_multiview_reinforce(pBuffer, (unsigned char*)(gray.ptr(0)), gray.cols, gray.rows, (int)gray.step, 1.2f, 3, 48, 0, true);
		//libfacedetect库的人脸探测函数，以上是对于人脸多角度强化探测的版本
		if (*pResults > 0){//如果探测到的人脸数多于0
			for (int i = 0; i < (pResults ? *pResults : 0); i++)
			{
				short * p = ((short*)(pResults + 1)) + 142 * i;
				int x = p[0];//获取人脸位置信息
				int y = p[1];
				int w = p[2];
				int h = p[3];
				int neighbors = p[4];
				int angle = p[5];//获取人脸角度信息
				float coeff = h / 300.0;
				for (int j = 0; j < 68; j++){//计算出在高为300的标准矩形框中，68个人脸关键点的位置坐标
					LMVec[cnt - 1][i * 2] = (p[6 + 2 * j] - x) / coeff;
					LMVec[cnt - 1][i * 2 + 1] = (p[6 + 2 * j + 1] - y) / coeff;
				}
			}
		}
		else.....//异常处理
		lblVec[cnt - 1][0] = imgLabel;
	}
	inLabels.close();
	inimgs.close();
	Mat data = Mat(213,136,CV_32FC1,LMVec);//将人脸关键点信息和情绪分类标号信息保存为Mat对象
	Mat lb = Mat(213, 1, CV_32SC1, lblVec);
	svm->train(data, ROW_SAMPLE, lb);//对SVM进行训练
	svm->save("mTrnSVMEMv102.xml");//保存xml文件
}
```

3. 程序运行结果分析

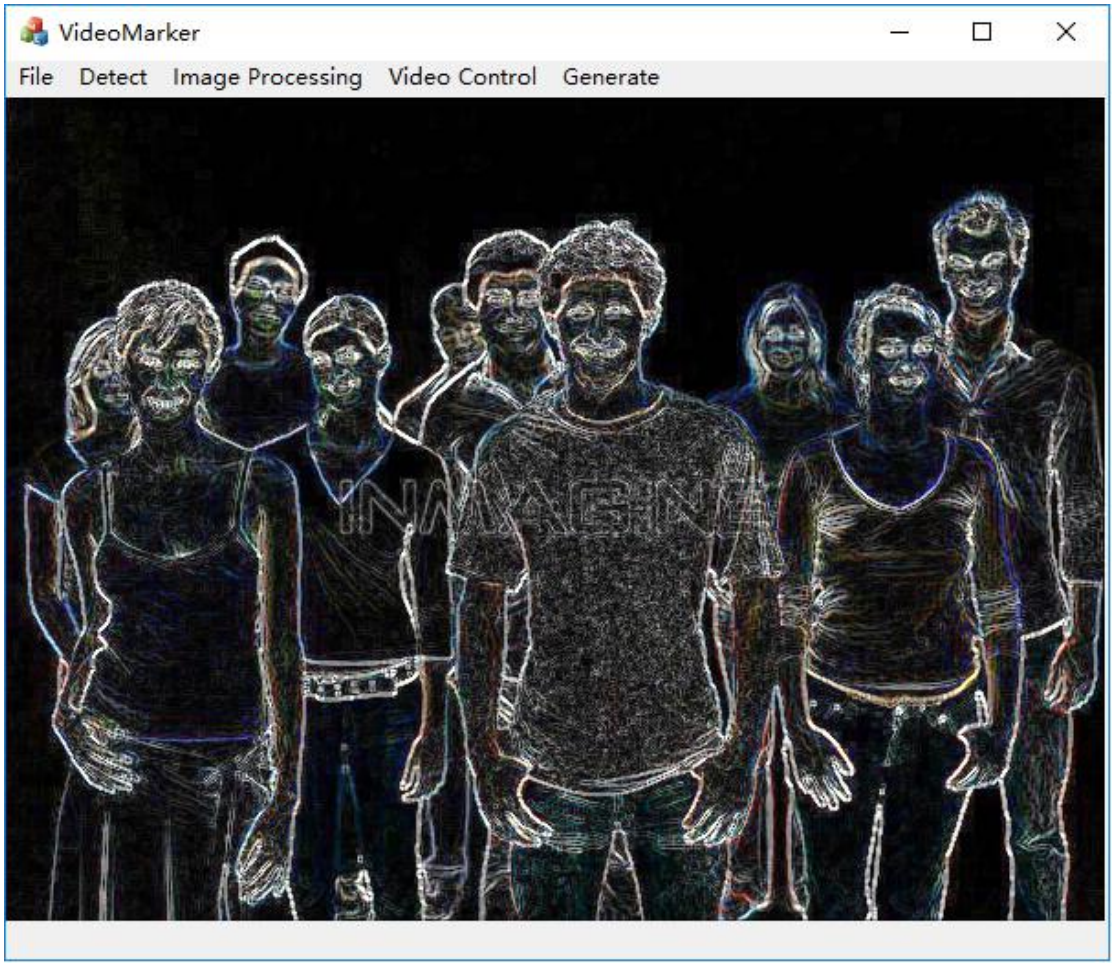
图像人脸识别的功能验证：

编译运行 VideoMarker，点击菜单 File-Import-Image，载入图片后点击菜单 Detect-Face-Frontal，得到运行结果如下：

可以看出，照片中的大多数人脸都能够被 OpenCV 的 detectMultiScale 函数所识别。实际上，如果修改 detectMultiScale 函数的参数，照片中其余两个人脸也能被识别出来，只是需要花费更长的时间。

图像处理功能的验证：

在上述操作基础上，点击 Image Processing-Undo，可以看到人脸识别框被撤销了。再点击 Image Processing-Edge-Sobel-Whole，得到如下图所示的结果，这是对于原图像进行 Sobel 边缘探测所得到的结果。

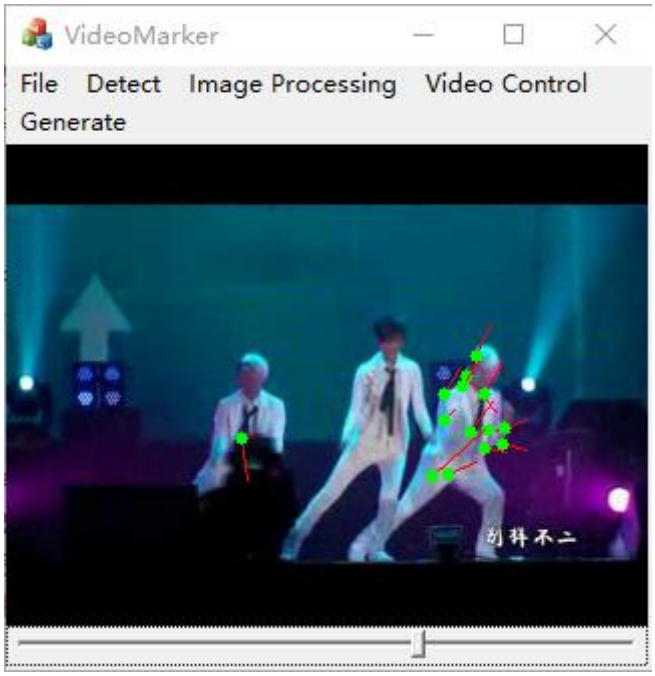


视频播放功能的验证：

点击 File-Import-Video，加载视频文件，分别点击 Detect-Face-Frontal 和 Detect-Face-Smile，可以验证在动态的视频中，OpenCV 识别人脸与微笑的功能。

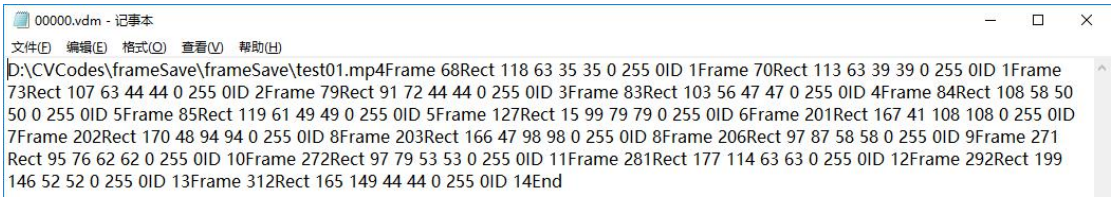
通过 Video Control 菜单中的 Speed...可以调整播放速度；当点击 Video Control-Play/Pause 或 Video Control-Next Frame 或 Video Control-Previous Frame 后，可以看到程序静止在视频的某一帧，此时可以执行 Image Processing 菜单中的功能，对一帧画面进行处理，也可以执行 File-Save Image，将该帧画面抓取保存。

再次点击 Video Control-Play/Pause，恢复视频播放，然后点击 Detect-Motion，可以看到视频运动检测的效果：



生成 vdm 文件的功能验证：

在加载视频的基础上，点击 Generate-Remark Files，填写生成的 vdm 文件的文件名，等待程序处理视频完毕，用记事本打开 vdm 文件，可以看到如下结果：



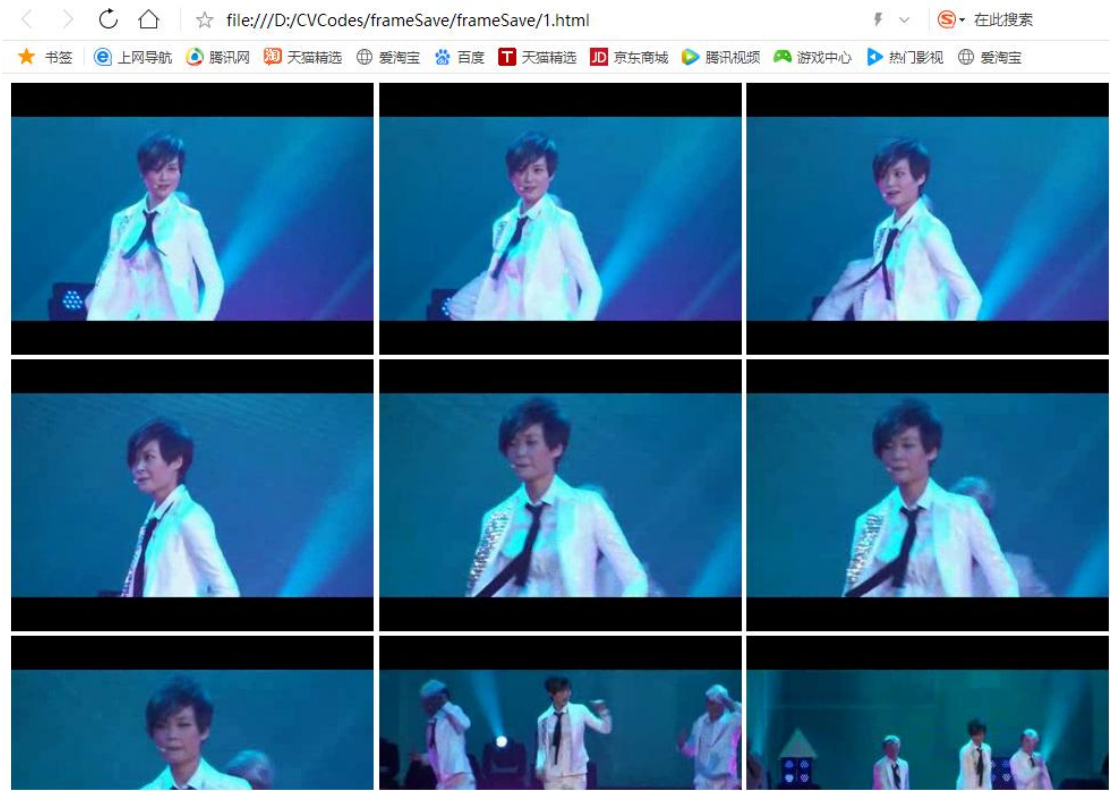
这就是 vdm 文件中保存的各关键帧信息、人脸位置信息与分类信息。

点击 File-Import-Remark，加载刚刚生成的 vdm 文件，然后再点击 Video Control-Play/Pause，可以回放各关键帧并标注相应的人脸信息。也可以点击 Video Control-Next Frame 或 Video Control-Previos Frame，实现关键帧的逐帧观看。

如图所示，关键帧中的人脸部分被圈出，“11”表示该人脸的分类标号。

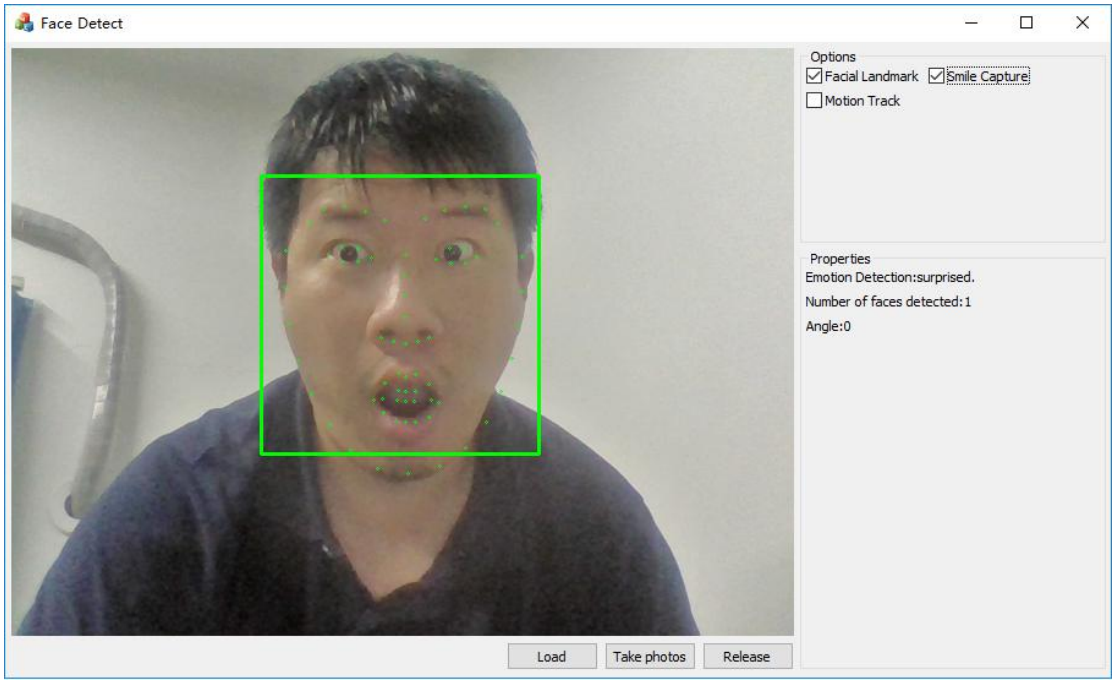
生成 html 相册的功能验证：

点击 Generate-Album，填写生成的 html 文件的文件名，打开对应的 html 文件，可以看到由全部关键帧生成的相册：

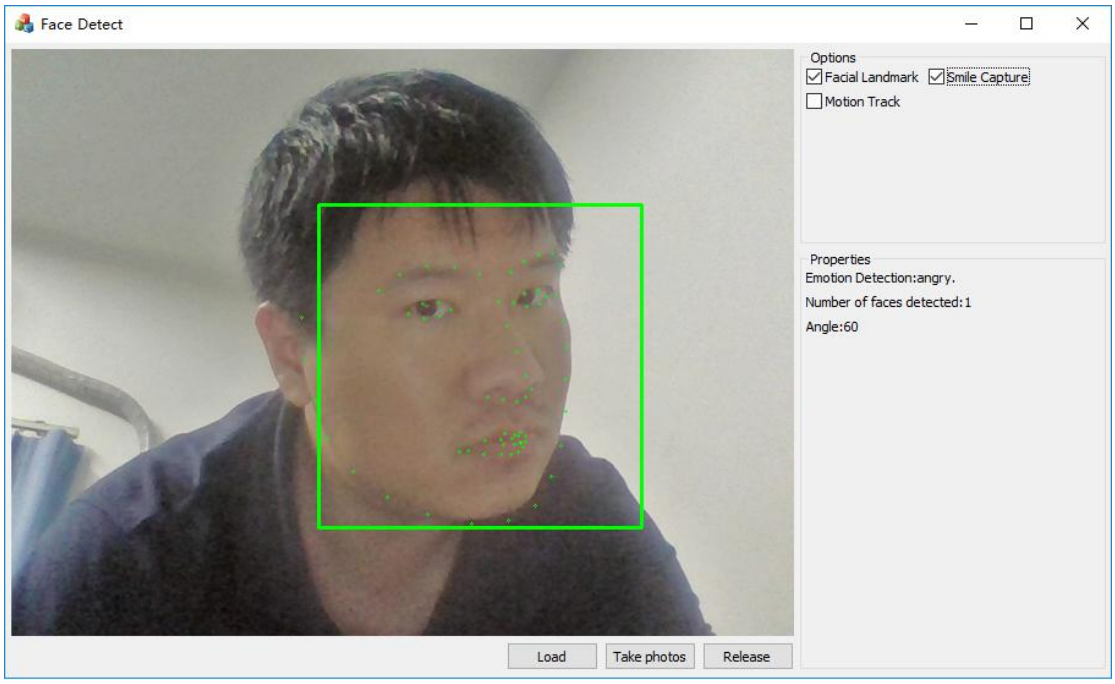


人脸特征点标注、人脸角度信息输出、表情分类识别功能的验证：

打开工程 FaceDetect，在 x64 debug 条件下编译运行，点击按钮 Load，打开摄像头，在 Options 中选中 Facial Landmark 与 Smile Capture 复选框，观察程序运行效果。

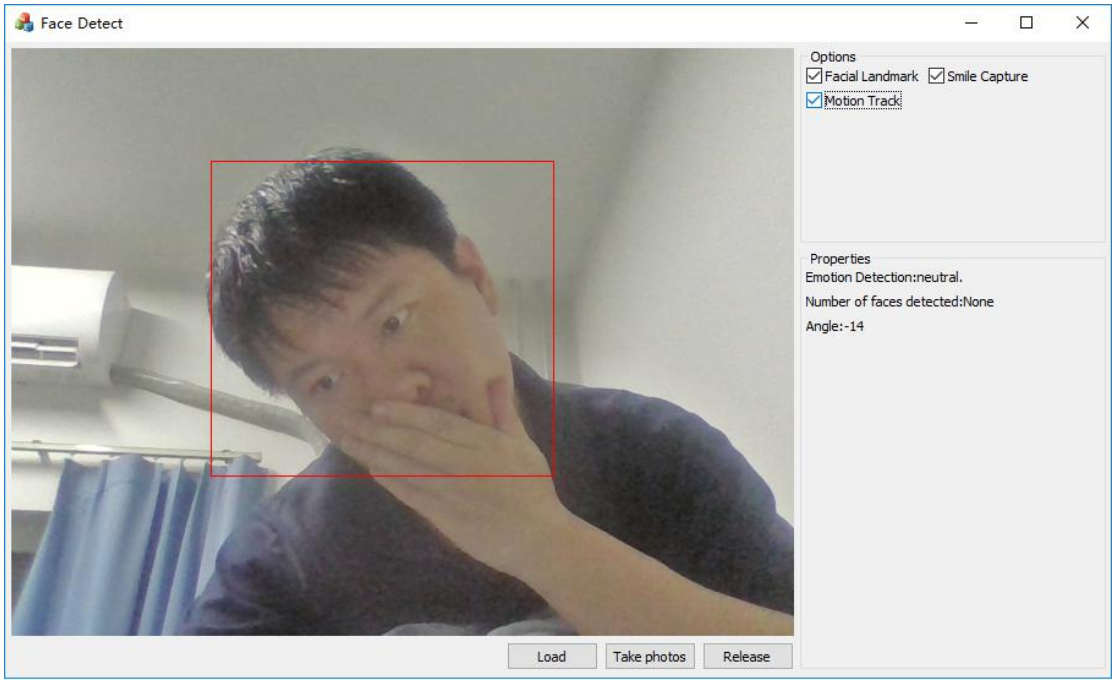


如图所示，可以看到人脸特征点的标注。在 Properties 一栏中，可以看到当前表情被识别为 Surprised，当前面部角度为 0 度。



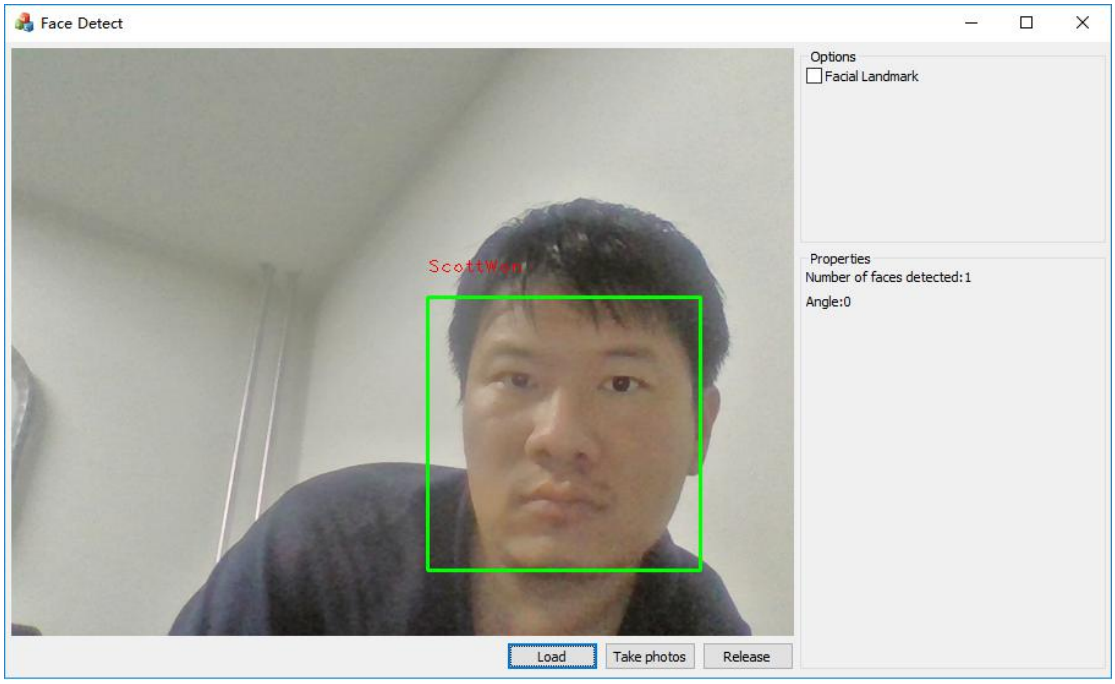
如图所示，此时 libfacedetection 计算出的面部角度为 60 度。

选中 Motion Track 复选框，可以看到，即使人脸发生一定角度的旋转，并且受到一定程度的遮挡，此时 libfacedetection 的人脸检测功能无法检出人脸的存在，但人脸运动追踪功能仍然能找到人脸的大致位置。



识别自己的脸的功能验证：

打开 FaceRec 文件夹下的工程 FaceDetect，编译运行，点击 Load 按钮，打开摄像头，观察人脸识别效果。



由此可见，该程序能够以一定的准确率识别出本人的面部特征。

综上所述，本程序各模块基本实现了相应的功能，具备一定的展示效果。不足之处是 vdm 文件的生成耗费时间过长，几乎无法处理稍长的视频文件，表情识别的准确性还不够好。

4. 总结

4.1 课题的难点和关键点

本课题的难点之一是自行实现 vdmParser 类，使其能够读写文件，解析文件，生成 html 相册。在本程序中，运用二叉排序树的数据结构，使得程序能够把树节点中的数据作为指令指导 vdm 文件的解析过程，这减少了代码编写的工作量与出错的可能性。

本课题的另一个难点是在掌握各种 OpenCV 的功能模块的基础上，能够对于已有知识进行组合，提出一些解决问题的可能方案、思路。运用运动追踪替代人脸检测，以期降低系统负荷，改善多角度、有遮挡情况下的人脸检测，这就是我对于非标准的问题解决方案的一点探索。

此外，在人脸识别的课题中，经常遇到识别率不理想的问题，这就需要我们多方面的研究、调试。最初，我按照教程写出的情绪分类代码只能识别 3 种情绪状态。后来，我通过在网上检索相关资料，找到了适合用来做 7 种情绪分类的 JAFFE 数据集。我通过调用 libfacedetect 库，改善了 OpenCV 识别速度低，不支持多角度的弱点。考虑到原始图像数据量太大，且容易受到背景环境的影响，而情绪的识别主要和人脸的一些关键点的相对位置有关，因而我转而使用 libfacedetect 库中人脸特征点的标注功能，通过提取 68 个关键点的位置信息训练情绪识别，以期在不损失识别准确率的条件下减少训练数据量。最后，我发现人们常用的线性核函数的效果不好，以至于运用线性核函数训练 SVM，常常得到的情绪识别结果是在一到两种情绪之间摇摆。当改用 sigmoid 核函数后，这一现象得到了改善。这就是我对于识别率不理想的问题所作出的改进尝试。

4.2 本课题的评价

课题本身的评价：

机器视觉与机器学习都是这个时代的研究热点问题。通过本课题，接触到机器视觉与机器学习方面的基本平台与基本工具，对于以后进一步的学习和探索很有益处。通过检索相关资料，对于机器视觉、人脸识别、神经网络方面的基本概念、基本思想或算法、研究动向和前景有了初步了解，这有助于视野的开阔和钻研兴趣的培养。

对自己完成情况的评价：

一方面，本次程序设计融合了我对于人脸识别、图像处理、机器学习工具、MFC、html 语言自学的初步成果，具备一定的完成度和展示度，另一方面，本次课题的完成情况没有达到计划中预期的程度。究其原因，最初对于本课题的难度估计不足，提出了人脸 3D 建模、视频模糊消除等子课题，摊子铺得过大，没能落实为实际的成果；在机器学习平台 caffe 的使用、训练、调测上花费了较多的时间和精力，但没能得到好的分类效果；在各种库的安装、编译，解决不兼容问题的过程中花费了太多的时间，没能在算法思考和代码编写上更加集中精力。如果在立项之初能够考虑到这些情况，在人脸识别的领域选择一到两个问题深入研究，也许会取得更好的效果。

不足之处：

关于 MFC 界面的设计：窗体重绘、控件布局的机制还不够完善；没有实现在窗体上加滚动条，显示超大图片的功能；菜单项使能与禁用的显示效果没有实现。

关于 vdm 文件的生成：该算法效率太低，处理视频文件耗时过多。

关于人脸运动追踪：追踪矩形框不够精确；有时程序会因为基准帧图像为空而出错。

关于表情识别：识别准确性仍然不理想，而且有一个奇异的 bug：在 x64 debug 模式下能得到较灵敏的识别结果，而在 x64 release 模式下识别结果几乎不变。

关于人脸探测与识别：容易受环境影响，戴眼镜时检出率低，人脸受遮挡时检出率低，OpenCV 自带的 FaceRecognizer 的人脸分类功能效果不佳。

改进建议：

关于 MFC 界面设计：实现较一般的窗体布局器；调用弹出对话框输入数据信息。

关于人脸运动追踪：提取人脸轮廓，判断特征点在轮廓内才进行匹配；调试改进代码逻辑，消除 bug；将人脸探测与追踪结合起来，提高效率。

关于人脸分类：尝试其他机器学习工具；实现基于 caffe 与 VGGFace 的人脸分类。

4.3 心得体会

本次程序设计实践锻炼了我的自学能力，开阔了我的视野，提升了我对于机器视觉、机器学习领域的算法、动向的兴趣与了解程度。在编程过程中，我对于怎样把人脸探测与人脸追踪结合在一起，怎样把人脸 3D 建模与人脸识别结合在一起，怎样把建立参数化模型的方法与统计回归的方法结合在一起等问题有了自己的思考。在检索资料的过程中，我了解到本领域很多远超我最初想象程度的成果，这激发着我向更高的起点迈进。

5. 参考文献

[1]人脸识别原理及算法：动态人脸识别系统研究，沈理，刘翼光，熊志勇著，人民邮电出版社

6. 附录

1.关于本次程序设计中各工程所依赖的库文件的安装问题：

VideoMarker 的部署：

工程 VideoMarker 依赖于 OpenCV2.4.13 和 libfacedetection。此外，实现 OpenCV 的图像对象向 MFC 绘图的 CvvImage.h 文件中包含了 windows.h 文件。

OpenCV2.4.13 的安装、部署：

在 opencv.org 网站选择 2.4.13.6 版本的 OpenCV(Win Pack)进行下载、解压安装。

设定 debug x64 条件下的工程属性：

在工程属性的 C/C++ - All Options - Additional Include Directories 一项中，添加目录***\opencv\build\include，其中***为 OpenCV2.4.13 的安装目录。

在 Linker -General - Additional Library Directories 一项中，添加目录***\opencv\build\x64\vc14\lib，其中***为 OpenCV2.4.13 的安装目录。

在 Linker - Input - Additional Dependencies 一项中，添加如下项目：

opencv_calib3d2413d.lib;opencv_contrib2413d.lib;opencv_core2413d.lib;opencv_features2d2413d.lib;opencv_flann2413d.lib;opencv_gpu2413d.lib;opencv_highgui2413d.lib;opencv_imgproc2413d.lib;opencv_legacy2413d.lib;opencv_ml2413d.lib;opencv_nonfree2413d.lib;opencv_objdetect2413d.lib;opencv_ocl2413d.lib;opencv_photo2413d.lib;opencv_stitching2413d.lib;opencv_superres2413d.lib;opencv_video2413d.lib;opencv_videostab2413d.lib.

设定 release x64 条件下的工程属性：

其步骤与 debug x64 条件下的操作大致相同，唯一的不同是，在 Linker - Input - Additional Dependencies 一项中，添加的项目应该为：

opencv_calib3d2413.lib;opencv_contrib2413.lib;opencv_core2413.lib;opencv_features2d2413.lib;opencv_flann2413.lib;opencv_gpu2413.lib;opencv_highgui2413.lib;opencv_imgproc2413.lib;opencv_legacy2413.lib;opencv_ml2413.lib;opencv_nonfree2413.lib;opencv_objdetect2413.lib;opencv_ocl2413.lib;opencv_photo2413.lib;opencv_stitching2413.lib;opencv_superres2413.lib;opencv_video2413.lib;opencv_videostab2413.lib.

最后，将***\opencv\build\x64\vc14\bin 中的 dll 文件拷贝到 system32 文件夹中，或者拷贝到本工程文件夹中。

libfacedetection 的安装、部署：

通过 GitHub 下载 libfacedetection，网址为：https://github.com/ShiqiYu/libfacedetection.

最好通过 Open in Desktop 的方式进行下载，否则容易下载不完全。

下载完成后，解压安装。

设定 debug x64 条件下的工程属性：

在工程属性的 C/C++ - All Options - Additional Include Directories 一项中，添加目录***\include，其中***为 libfacedetection 的安装目录。

在 Linker -General - Additional Library Directories 一项中，添加目录***\lib，其中***为 libfacedetection 的安装目录。

在 Linker - Input - Additional Dependencies 一项中，添加 libfacedetect-x64.lib.

设定 release x64 条件下的工程属性：

与 debug x64 条件下的操作完全相同。

最后，将***\bin 中的 libfacedetect-x64.dll 文件拷贝到 system32 文件夹中，或者拷贝到本工程文件夹中。

FaceDetect 的部署：

工程 FaceDetect 依赖 OpenCV3.1 与 libfacedetection。

OpenCV3.1 的安装、部署：

在 opencv.org 网站选择 3.1.0 版本的 OpenCV(Win Pack)进行下载、解压安装。

设定 debug x64 条件下的工程属性：

在工程属性的 C/C++ - All Options - Additional Include Directories 一项中，添加目录***\opencv\build\include，其中***为 OpenCV3.1.0 的安装目录。

在 Linker -General - Additional Library Directories 一项中，添加目录***\opencv\build\x64\vc12\lib，其中***为 OpenCV3.1.0 的安装目录。

在 Linker - Input - Additional Dependencies 一项中，添加 opencv_world310d.lib.

设定 release x64 条件下的工程属性：

其步骤与 debug x64 条件下的操作大致相同，唯一的不同是，在 Linker - Input - Additional Dependencies 一项中，添加的项目应该为 opencv_world310.lib.

最后，将***\opencv\build\x64\vc12\bin 中的 opencv_world310.dll、opencv_world310d.dll 文件拷贝到 system32 文件夹中，或者拷贝到本工程文件夹中。

FaceRec 的部署：

工程 FaceRec 依赖 OpenCV2.4.13 和 libfacedetection，其部署方法同工程 VideoMarker.

2.对于不同版本的 OpenCV 的功能差异的说明：

不同版本的 OpenCV 提供的接口与相应的编程风格有一定的差异。在 OpenCV 的早期版本中，一般使用 Iplimage*、CvCapture*作为图像处理类与视频处理类，其编程风格接近于 C 语言的编程风格。在较新版本的 OpenCV 中，一般分别使用 Mat、VideoCapture 作为图像处理类与视频处理类，其编程风格是面向对象的 C++的编程风格。

不同版本的 OpenCV 所支持的功能有所不同。

在低版本的 OpenCV 中，存在头文件 CvvImage.h，支持在 MFC 窗体中显示 Iplimage 图像，这一功能在后续版本的 OpenCV 中被取消了。

在 OpenCV2.4.13 中，contrib 库中包含了一系列针对人脸的图像分类器，如 EigenFaceRecognizer、LBPHFaceRecognizer、FisherFaceRecognizer，等等。上述图像分类器分别运用主成分分析（PCA）、局部二进制编码直方图（LBPH）人脸识别等算法，实现了对于人脸的分类功能。在某些更高版本的 OpenCV（比如本人安装的 OpenCV3.1 和 OpenCV3.4.1）中，没有提供相应功能的 contrib 库。在有的版本的 OpenCV 中，上述功能接口被转移到 face.hpp 或 facerec.hpp 中。可以说，各个版本的 OpenCV 对于人脸分类的支持程度与方式各不相同。

在 OpenCV3.4.1 中，提供了针对深度神经网络的 DNN 模块，运用该模块提供的接口可以调用机器学习平台 Caffe 所训练的模型。这一功能在较低版本的 OpenCV 中无法实现。

在实际测试中，各个版本的 OpenCV 代码的功能稳定性各不相同。

在 OpenCV3.4.1 中，正面人脸的 Haar 特征分类器的数据文件 haarcascade_frontalface_alt.xml 存在一定的问题，在本人测试时，需要加载 haarcascades_cuda 文件夹下的 xml 文件方可保证程序正确运行。

在 OpenCV2.4.13 中，使用支持向量机 CvSVM 对象，按照文档中提供的实例编写代码，会遭遇内存释放方面的错误。解决办法是用 OpenCV3.1 中的 Ptr<SVM>来实现相应的功能。