Screen Estimation for a Novel Pointing Device Based on Corner Detection and Classification

Master Thesis

Authors: Qing Xia, Wei Huang [1]

Aachen, July 2004

[1] Authors: Q. Xia and W. Huang are students from the Department of Media and Knowledge Engineering, Delft University of Technology, Netherlands. This thesis report is a part of their master thesis project in the Man-Machine Interface group of PHILIPS Research Lab, Aachen, Germany.

Thesis committee

Dr. drs. L. J. M. Rothkrantz,	Delft University of Technology, The Netherlands
Dr. K. van der Meer,	Delft University of Technology, The Netherlands
Prof. dr. ir. E. Kerckhoffs,	Delft University of Technology, The Netherlands
Prof. dr. H. Koppelaar,	Delft University of Technology, The Netherlands
Dr. J. Kneissler,	PHILIPS Research Laboratories, Aachen, Germany

Acknowledgements

Firstly, we want to thank PHILIPS Research laboratories in Aachen, Germany, who supplied such a fantastic internship opportunity for us. During this period, we got chances to work in a great research environment at this top-notch research laboratory. That has given us a very rich experience during the course of our two years master study.

During this time, we have received a lot of help and encouragement from our supervisor, Mr. Jan Kneissler, who is very patient and gave us many suggestions and guides from practical implementation to theoretical framework. Without his help, our project cannot be completed so successfully and achieve all the targets that we defined in the beginning.

Another important person, we would like to thank is our professor Leon Rothkrantz. Dr. Rothkrantz introduced this opportunity to us. He is a very nice man. Almost in every party of our classmates, we were talking about his enthusiastic and friendly help to us. Aachen is a city in Germany, far from Delft, but he still scheduled several meetings in Aachen with us. In these meetings he gave us lots of suggestions on the writing of this thesis and many ideas to stretch our imagination. Without him, this paper would not be possible. So we must say "Thanks" to him.

Finally we also want to thank Mrs. Fien Rothkrantz for sending us the scanned version of our thesis with helpful remarks from Dr. Rothkrantz.

Abstract

"UI-Wand" is the name of a new project in the Machine Interface Group of PHILIPS research laboratories in Aachen, Germany. It's a futuristic concept that uses a pen-shaped pointing device with a camera in its tip to recognize the objects in your home and control them by doing some gestures with it. The first logical step towards its goal is to control applications running on devices with a screen, like Computer, Laptop, or TV. So the basic requirement for current UI-Wand control is the pointing positioning and gesture recognition.

This master thesis proposes solutions to several software modules in such a UI-Wand system. Typically, such system includes screen corner detection, screen corners tracking and gesture recognition. Finally we implemented all of the models and algorithms that we chose or designed in the system and implemented a prototype of a UI-Wand driver that demonstrates what can be achieved using such a pointing device.

Index Terms – Pointing device, Corner Detection, Relevance Vector Machine, Tracking, Gesture Recognition.

Man-Machine Interface Group at PHILIPS Research Laboratories in Aachen

The Man-Machine Interfaces Group at the PHILIPS Research Laboratories in Aachen (Germany) investigates innovative paradigms for human-computer interaction and represents the speech recognition competence center within PHILIPS Research. The vision of the group is to "make Man-Machine Interaction intuitive, efficient and enjoyable". The ambition is "to take innovative interaction technologies to the 'main street'".

Since the start in 1973 with studies on speaker verification and isolated speech recognition for small vocabularies, the main research topics have meanwhile moved towards:

- Voice control technology and applications (e.g. hands-free and eyes-free interaction in the car environment)
- Telephone-based dialogue systems (for a demo of an automatic inquiry system in German, call: +49 241 604020)
- Professional dictation applications (e.g. medical report generation)
- Conversational user interfaces
- Management of spoken audio content
- Computer vision applications
- Innovative interaction paradigms

Since the first speech recognition products have been introduced to the market (continuous speech dictation system for professionals in the medical sector in 1993, telephone-based inquiry system in 1994, voice dialing for mobile phones in 1996), the group has expanded its scope and is now working on user interface technology in more general terms, which includes additional aspects of man machine communication. These are covered by research areas like e.g. speech understanding, dialogue management, user and interaction modeling, information retrieval and computer vision.

The aim of Computer Vision is to process, analyze and interpret visual input from cameras using signal processing, pattern recognition and machine learning techniques. Application fields range from production automation, quality control, biometrics, automotive sector (e.g. pedestrian recognition, visibility enhancement, collision free driving) to home entertainment and ambient intelligence (e.g. for personalization, user interaction).

Major topics in Computer Vision are compensation of illumination and motion effects, temporal and spatial segmentation, image interpretation/annotation and tracking/detection/recognition of persons/objects.

Contents

Abstract **iii**

Man-Machine Interface Group at PHILIPS Research Laboratories in Aachen **v**

1 Introduction **3**
- 1.1 UI-Wand project 4
- 1.2 Related work 4
  - 1.2.1 A novel form of a pointing device 5
  - 1.2.2 Remarks to the related work 5
- 1.3 Thesis overview 6

2 Problem Definition **9**
- 2.1 System requirements 9
- 2.2 Hardware components 9
- 2.3 Software components 10
- 2.4 Project goals 11

3 Models and Algorithms Overview **13**
- 3.1 Screen corner detection and pointing positioning 14
  - 3.1.1 Candidates-Winners approach to detect screen corners 15
  - 3.1.2 Pointing projection model 16
- 3.2 Tracking and gesture recognition 18
  - 3.2.1 ROI Tracking algorithm 19
  - 3.2.2 RVM model to gesture recognition 19

4 Corner Detection **23**
- 4.1 Corner detection algorithms comparison 23
- 4.2 Sojka corner detection algorithm 26
  - 4.2.1 Main ideas of Sojka corner detector 26
  - 4.2.2 Sojka corner detection theoretical foundations 28
- 4.3 Algorithm realization 38
  - 4.3.1 Computation procedures 38
  - 4.3.2 Explanations of value estimations 39
- 4.4 Parameter optimization 39
- 4.5 Algorithm tests 41
  - 4.5.1 Accuracy test 41
  - 4.5.2 Stability test 45
  - 4.5.3 Sequence test 46

4.6 Conclusions 47

5 Corner Classification 49

5.1 k-NN and SVM models for classification 49

5.2 Relevance vector machine and sparse Bayesian learning 51

5.2.1 RVM regression model 51

5.2.2 RVM classification model 55

5.2.3 Multi-class classification 57

5.2.4 Optimization of parameters in a kernel function 58

5.3 Features extraction of corners for RVM 59

5.3.1 DCT coefficients as samples features 60

5.3.2 Rotation corners feature extraction 63

5.4 Training sample datasets selection and tests for RVM 66

5.4.1 Manually selected sample dataset 66

5.4.2 Synthetic sample dataset 71

5.4.3 RVM improvement by adaptive sample dataset 77

5.5 Conclusions 82

6 Rectangle Filter 85

6.1 Rectangle filter given corner type 85

6.2 Rectangle filter without type information 87

6.3 Conclusions 88

7 ROI Tracking 91

7.1 ROI tracking filter 91

7.1.1 ROI selection and predication 91

7.1.2 Screen corner detection in ROIs 94

7.2 Missing screen corners predication 95

7.3 Conclusions 96

8 Gesture Recognition 99

8.1 HMM for gesture recognition 100

8.1.1 Introduction of Hidden Markov Model 100

8.1.2 Theory of HMM model for gesture recognition 101

8.1.3 Realization HMM model for gesture recognition 103

8.1.4 Conclusions of HMM for gesture recognition 104

8.2 RVM for gesture recognition 104

8.2.1 Observation getting 105

8.2.2 List analysis for getting candidate gesture traces 106

8.2.3 Candidate trace interpolation 108

8.2.4 Feature extraction 110

8.2.5 Training samples generation 110

8.2.6 RVM training 112

8.2.7 RVM classification 112
8.2.8 Gestures recognition test 112
8.2.9 Conclusions 114

9 UI-Wand System Design 117
9.1 The existing framework introduction 117
9.2 UML models introduction 118
9.3 UML models of Visipirin 120
9.3.1 Individual class description 123
9.3.2 Applications execution 124
9.4 UI-Wand system introduction 125
9.5 Synthetic dataset generation application 127
9.6 Screen corners training application 129
9.6.1 Individual class description 130
9.6.2 Sequence diagram 134
9.7 UI-Wand application 136
9.7.1 Individual class description 137
9.7.2 Sequence diagram 144
9.8 Gestures dataset collection application 146
9.9 Gestures training application 147
9.10 Adaptive screen corner dataset collection application 148
9.10.1 Individual class description 148
9.10.2 Sequence diagram 149

10 UI-Wand Utilities Design 151
10.1 Visualization Utility 151
10.1.1 Individual class description 152
10.1.2 Sequence diagram 152
10.2 UI-Wand application error analysis utility 153
10.2.1 Utility GUI introduction 154
10.2.2 Error analysis utility introduction 157
10.2.3 Error analysis module 158

11 System Tests 167
11.1 Off-line tests 167
11.1.1 Off-line test environment introduction 167
11.1.2 Off-line test goal 167
11.1.3 Off-line test sequences selection 168
11.1.4 Off-line test results 169
11.1.5 Off-line test result analysis 170
11.2 On-line tests 171
11.2.1 On-line test environment introduction 171
11.2.2 On-line test goals 171

11.2.3 On-line test procedure 171
11.2.4 On-line test result and result analysis 172
11.2.5 On-line test result analysis 172
11.3 Conclusions 173

12 Conclusions and Future Works 175
12.1 Conclusions 175
12.2 Future Works 177

Appendix A, White Screen Sequence 181
Appendix B, Black LCD Screen Sequence 183
Appendix C, Ten Off-line Test Sequences Detection Results 185
Appendix D.1, Algorithm Parameters for Off-line Tests 195
Appendix D.2, Algorithm Parameters for On-line Tests 199

Bibliography 201

Part I

Problem Definition

1

Introduction

In the past few decades, the computer mouse acts as a revolutionary device by which everybody can control the cursor easily on a computer graphical interface. But undeniably, there are still lots of limitation of using traditional 2D mouse, for example, if there is no desk available for using a mouse or touchpad on the keyboard and laptop, people will have problems interacting with their computers. This presents a strong case for people to think about developing new devices like computer mice but without such limitations. A lot of pointing devices have appeared recently that enable us to control the cursor on the display and interact with a graphical user interface. These devices are designed to replace conventional 2D pointer devices such as mice, touchpads, trackballs. They have already become a revolution in the compute world. Fig.1.1 shows some examples of pointing devices.

(a)

(b)

(c)

(d)

Figure 1.1: Some existing pointing devices. a) Microsoft X-Wand. b) IBM Hand held Infrared pointing device. c) HeadMouse. d) CyberlinkMindMouse.

Existing prototypes or even commercial products of such pointing devices have already been invented. We divide them into two categories: some are sensor-based approaches and the others are camera-based approaches. Sensor-based approaches often depend on some special electronic sensors, which can be worn on people's forehead, eyes, face or other body part. Firstly the system analyzes the data captured by the sensors. Then it projects these data into the 2D screen model and finally drives the cursor to the right position that the user wants it to go. CyberlinkMindMouse, HeadMouse and X-Wand [Wil03b][Wil03c] (see Fig.1.1) all belong to the sensor-based category. Unlike the sensor-based approaches, the camera-based approaches do not need user to worn any extra device on the body, which is also the most favorable aspect of them. These approaches directly analyze images caught by a camera. Firstly they get frame sequences from the camera, then they use some algorithms to extract features or content from the frames, by tracking feature or analyzing the context in the frame sequences, they can decide where the user is pointing. In the end, the pointing position is transferred from the image coordinates onto the screen coordinates and the system drives the cursor to the right position on the screen. There are some existing prototypes in this category, such as [Cha00] and [Can03], the only difference between them is that [Cha00] uses a web camera directly facing the user and by detecting user's face, the system can know the position on the screen, where the user is facing to. But in [Can03], they use a camera directly pointed to the screen, by analyzing the content of the screen in the captured frames, the system can detect the position, where the user is pointing. With the same coordinates transformation as in [Cha00] the cursor can be driven to the pointing positions. Both of the two categories can be realized to control the computer cursor, but they use totally different technology to implement and meet requirements.

1.1 UI-Wand project

As a pre-development project in the Machine Interface Group of PHILIPS, UI-Wand (User Interface Wand) is a prototype for a pointing device, which can be used to control the cursor either on a computer display or some other types of displays. for example, a digital TV. It is an accessibility device and is particularly useful for man-machine interactive environments. PHILIPS investigates the camera-based approach to realize the UI-Wand system. In parallel to our work, which uses a screen corner detection method to find the screen, a geometric matching method is also investigated by PHILIPS, but results are not yet available. In section 1.2.1, we will introduce some related project, which also used a camera-based approach to realize a similar application.

1.2 Related work

Camera-based approach for pointing devices does not need complex hardware components. Usually it just needs a simple camera with some signal-transition device can realize the whole work. But this kind of systems often needs more effort to develop software with efficient algorithms to analyze the camera images.

1.2.1A novel form of a pointing device

In [Can03], a novel pointing device was investigated enabling to control via a cursor that interaction with a graphical user interface. This device requires only low-cost simple hardware, just a single handheld camera that points towards the display to control the cursor on the screen. It does so by finding display content in the camera view. It calculates where the centre of the camera view is on the screen and moves the cursor accordingly. Thus their control strategy only needs the visibility of a screen display device.

They use colored regions in the display content and the camera image, which are often strongly structured in the screenshots of office programs or presentation slides to extract region of the display. They select the biggest region from a segmentation of the screen content. From the camera image, some biggest regions can also be extracted. Then calculating some score value that can identify the regions in the camera image that correspond with the selected regions. This score includes how different the color of a region in the image compared to the corresponding region in the screen, how much the region is in the centre of the view and its size. After getting the display region in the camera image, they smooth the region borders with a morphological filter in both display image and the camera image. The outline of the border is then tracked and strong corner points with high curvature are extracted. Among these strong corner points, four points that are furthest away from each other are selected as the set of corresponding points. They model the problem of estimating the cursor as an estimation of the homography of the screen in the world. Now they have a set of four corresponding points pairs in the display image and the camera image, they can use them to compute the homography and project the centre of the camera image back on to the plane of the display screen, giving the correct position for the cursor.

This technique is computational efficient, and could therefore allow the user to interact with a normal application whilst achieving a suitable update frequency for the cursor position. The user can control the cursor with more freedom comparing with other approaches, just by pointing the camera towards the display screen. But there are still some restrictions for this prototype, now it can only identify corresponding screen corners within a very structured content (large regions) in the screen, if the content of the screen is very complex and with no obvious regions, it will have problems to find the right screen corners and the application will be aborted.

1.2.2Remarks to the related work

We can find some problems of the existing pointing device described in section 1.2.1:

- When the screen background is more complicated, the existing approach cannot perform very well, for example, there is some moving content or some varying content such as video program shown on the screen.

- Their approach underlies the basic assumption that screen content must contain simple geometric areas with high contrast borders (like e.g. rectangular windows).

- The expandability of the existing pointing device system is not so good, if the display changes, the result may become worse.

- The existing pointing device has architectural difficulty, that is because they need to send the data flow of the screen content to the device and receive the result data flow after that.

- In the previous work, researchers mentioned some methods to speed up their system, but they do not actually realize them.

- The existing pointing device just acts as a simple mouse and gesture recognition has not been realized, which is also a very important usage of pointing devices.

1.3 Thesis overview

In this chapter, we presented an overview of the existing pointing devices and gave a new concept of a UI-Wand at PHILIPS. In the next chapter, in order to make a prototype of UI-Wand, the problem definition is given, which includes the system requirements, the existing hardware and software introduction, and our project goals.

In Chapter 3, we describe briefly the models and algorithms that we used in the different modules in the final implementation of UI-Wand system. We also raised our Candidates-Winners approaches for screen corner detection.

In Chapter 4, Chapter 5 and Chapter 6, we start presenting our models and algorithms in detail. Firstly, in Chapter 4 we describe the Sojka corner detector that is used as a candidate screen corner selection algorithm for our final system. Then, in Chapter 5, RVM, a new classification model is introduced which deals with objects classification problems and in our case it can successfully classify whether a corner is a screen corner or not. But it proves that we still cannot figure out the final screen corners only by corner detection algorithm and classification algorithm. Finally, in Chapter 6, we introduce a rectangle filter algorithm by which we can accurately detect the final screen corners.

In Chapter 7 and Chapter 8, we discuss the issues about gesture recognition. In Chapter 7, we use a ROI tracking filter algorithm to solve the tracking problem and in Chapter 8, we present HMM and RVM models for gesture recognition problem and finally we realize the RVM model and do some test.

In Chapter 9 and Chapter 10, we describe our system design and implementation in details by using UML, where all the applications that we developed for UI-Wand system and some utilities for system evaluation are presented. So readers can easily find out what we have done and it is also helpful for the persons who will continue our works in the future.

In Chapter 11 and Chapter 12, we test our system off-line, on-line and record results, after analysis, finally in Chapter 12 we summarize our projects and give some remarks on the future work.

2
Problem Definition

2.1 System requirements

According to the original design, the UI-Wand system must satisfy the following requirements:

Simple hardware components. The whole system should have a simple structure and the hardware components should be involved as less as possible.

Easy manipulating capability. For more popular applications, we need to make the operation of this UI-Wand very easy. Because in the future it may be used in everybody's normal life such as home, meeting room, office, hospital and other daily situation, the easy operating capability should always be kept in mind.

Real-time processing speed. This is a critical criterion for judging the performance of the UI-Wand, we have to realize real-time processing speed otherwise prototype would be almost useless for actual applications.

High system stability and robustness. The system of UI-Wand should be stable when it is running, which means if we change the running environments, the result should not be changed too much. The more accurate the UI-Wand prototype is, the more promising it may be used in the future.

Good adaptability, adjustability and flexibility for future extension. UI-wand is just the first stage prototype. It may be changed into other applications. We must make it more flexible so that it can be adapted in all possible conditions. Although it is just a prototype and we may not consider everything in the primary stage, the extension ability should be the basic consideration for any prototype design.

2.2 Hardware components

Machine for implementation: Pentium IV-2.4G, 256M memory/Linux
Machine for demonstration: Pentium IV-2.4G, 256M memory/Windows XP

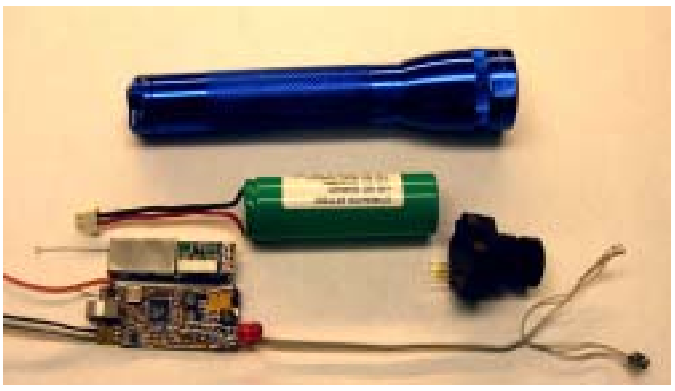
UI-Wand device components: The research prototype is built of standard off-the-shelf components (see Fig. 2.1) (no optimization in size/power consumption), which include:

1. CMOS sensor (PAL, 628x582, 22x22x26mm, 5V/DC, 10mA) with fixed focus optics (f=6mm, FOV=51°x43°)

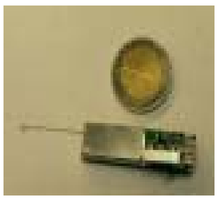
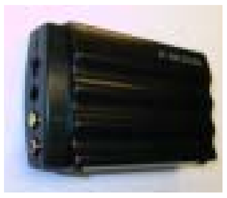
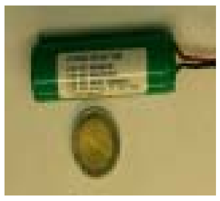
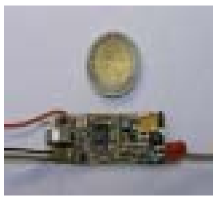
2. Analog audio-video transmitter (2.4GHz, 12V, 70mA) & receiver

3. Accumulator (Lithium-Ion), and charging station (500mA)

4. Microcontroller for charging, button control and activation by motion sensor



(a)

(b) (c) (d) (e) (f)

Figure 2.1: Hardware components of UI-Wand. (a) All components. (b) CMOS sensor with fixed focus optics. (c) Analog audio-video transmitter. (d) Analog audio-video receiver. (e) Accumulator and charging station. (f) Microcontroller for charging, button control and activation by motion sensor.

2.3 Software components

Operating system for implementation and test: Red Hat Linux
Demonstration system: Windows XP
Implementation language: C++ (GNU Compiler)
Implementation tools: KDevelop 2.1.3/Linux, QT Designer 2.0/Linux

2.4 Project goals

UI-Wand is a pointing device, with which we can freely control the computer cursor on the screen. Our project target is to design and realize such a device. The appearance and the using mode of UI-Wand are shown in Fig. 2.2.

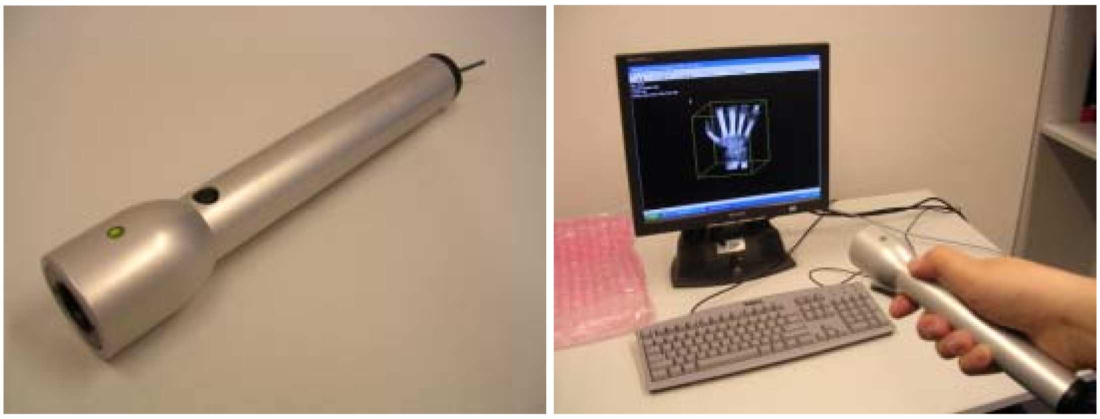


Figure 2.2: (a) Appearance of UI-Wand. (b) The using mode of UI-Wand (right image).

We designed a UI-Wand system based on a camera-based approach. The developing work is based on an existing framework designed by PHILIPS using C++. From analyzing the frames captured by the camera, we want to find the exact position of the screen. Then the cursor can be driven to the right position on the real screen. There are some possible methods to find screen position in a frame, such as edge detection, corner detection, content matching and so on. Content matching approach has already been proved to have worse performance for a more complicated graphical interface. So in the design stage we chose a four-screen corner detection approach, the four screen corners are always the most crucial feature marks for deciding an accurate screen position no matter how complex contents are on the screen. If the four corner positions are obtained, the shape, the size and even other properties of the screen can be obtained. Other feature detection methods such as edge detection can also be used, but the computation complexity will raise (see [Par98] and [Son03]) and since finally these features are all served for marking the four screen corners to accurately find the screen, we decide to realize four-screen corner detection directly from the frames.

In the UI-Wand system, gesture recognition is a functionality requirement. Because it is a new way to send human commands to operate the computer. If the UI-Wand system can recognize some simple gestures, it will be a big functionality improvement for using it.

The whole application should run in real-time, which means when we move the UI-Wand pointing position to a new position within the computer screen, the cursor can move to that new position at the same time.

The UI-Wand is just a prototype of a pointing device and we do not specify the working conditions of it. Thus, the UI-Wand need to be robust for the environment changes such as lighting changes and can run the different applications. In addition, for possible usage in the future, within an enough big working range, the UI-Wand still has to work properly (see Fig.2.3). Finally, in order to test the performance of UI-Wand, we also need some utilities to evaluate the system.

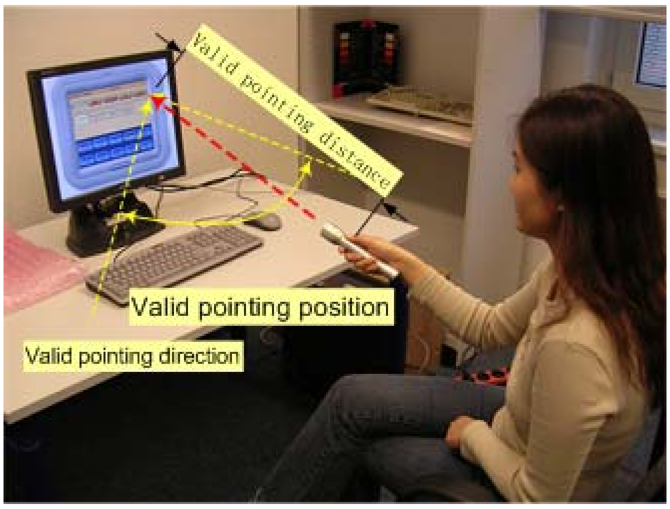


Figure 2.3: Valid pointing position and valid pointing direction.

To conclude this chapter, we define the following project goals:

- Design a UI-Wand system using camera-based approach.
- Realize screen positioning using screen corner detection method.
- Realize a gesture recognition model for our UI-Wand.
- Adapt the system to work in different environments.
- Make sure that the system can run different applications.
- Make sure that the system runs stably within a big valid working space range.
- Develop utilities to evaluate the system.
- Keep the speed of the system approximately to10 frames/sec.
- Develop the whole system based on the existing software framework.

3 Models and Algorithms Overview

Models and Algorithms design is the most important part of a research project. If you designed wrong models or algorithms for your system then no matter how hard you work on it you will never get the satisfying results. So we spent lots of time on choosing and designing suitable models and algorithms for our special problems,

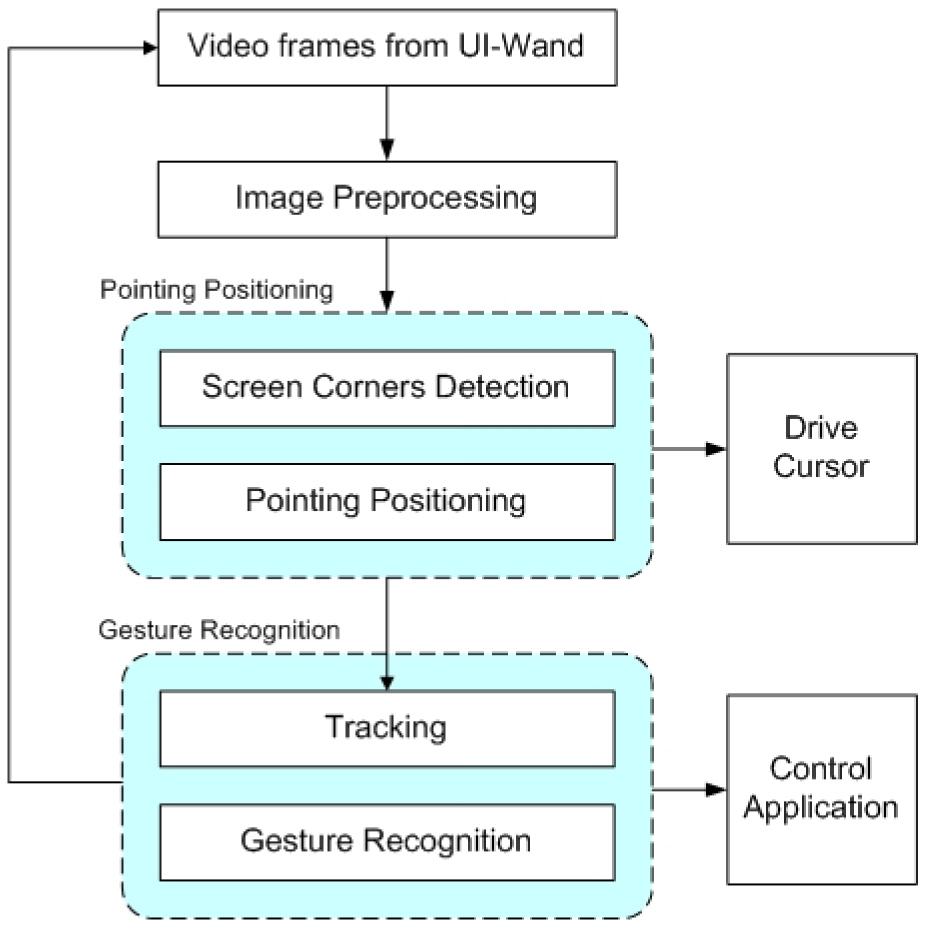


Figure 3.1: Modules in UI-Wand systems. The modules in cyan block are our problems.

which will ensure us that we can construct the final system satisfying the all requirements and goals mentioned in Chapter 2. According to the UI-Wand system functional requirements and project goals, the main problems for us are pointing positioning and gesture recognition (see Fig.3.1). Between them, it is clear for us that the main research task is about positioning. How to localize the position where the camera is pointing at is the main problem to solve. If we can figure out this issue, the work left in gesture recognition is to collect the history of the pointing point as the trace and then doing corresponding analysis for recognition.

In order to give an overview to the models and algorithms that we finally utilized in different modules in the UI-Wand system, we will give some short description about them in this chapter.

3.1 Screen corner detection and pointing positioning

With respect to the pointing positioning problem, although there are not so much previous works on camera pointing devices, we still can work out the common and necessary procedure to realize point positioning. The first step to realize pointing position is to figure out the coordinates of some defined markers on the real screen in an image coordinates system. Then the second step is to find the right place of the screen in a three dimensional space by using these markers coordinates so that they know where is the center of the image captured by the camera in the screen coordinates system by doing some space transformation, and where is the accurate pointing position is (see Fig. 3.2).

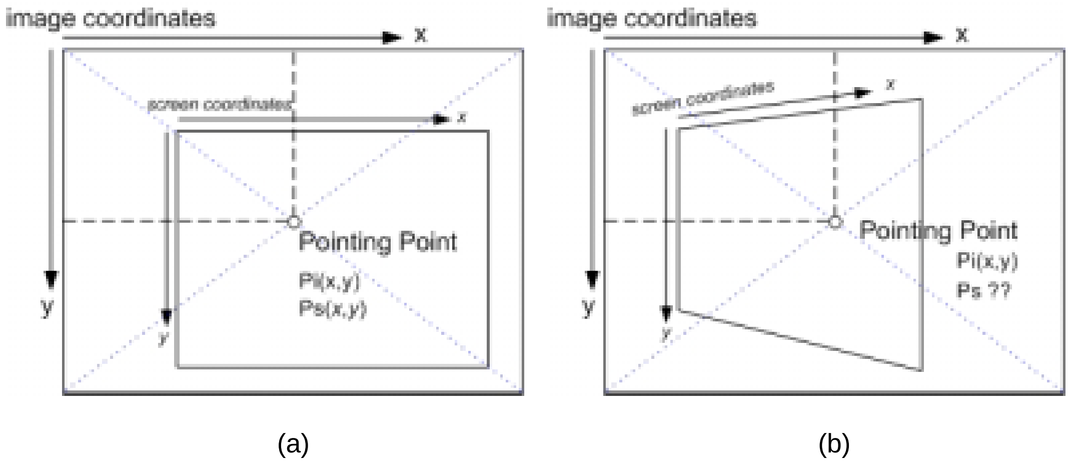


Figure 3.2: The relationship between image coordinates system and screen coordinates system. (a) UI-Wand is shooting in front of a screen, and the screen coordinates of the pointing point can be easily calculated. (b) The UI-Wand is shooting in right front of a screen, which makes calculation of screen coordinates of pointing point much harder.

The tests [Can03] prove that this procedure is effective for pointing positioning, so we keep the same procedure in our case. For calculating the pointing point in three-dimensional space, we directly use PHILIPS's existing algorithm. This algorithm makes several screen position hypothesis in advance and then doing pruning searching to get the result. We will give a brief description in section 3.1.2. So the main research work for us is marker detection.

It is an easy method to detect the screen by mounting some sensor on it, which avoids the problem of display variety and can be detected accurately. But that needs to add extra hardware to the system. Cantzler [Can03] uses screen corners as his markers of the screen. They developed a method to extract the screen corners, which makes use of the content of screen display to match the camera image. Then after analyzing the features of the regions in the camera image, they extract finally an approximate border of the screen in camera image and use the four corners as screen corners.

This approach can directly be used without extra devices but from the model description, it depends on the content of a display and has problems when the display is not structured, furthermore, in a long run, it cannot be used to control other objects.

After summarizing previous works and investigating plenty of other approaches, we decided finally to use corner detection and classification models to realize detection of screen corners as markers for pointing positioning and to name this approach "Candidates-Winners".

3.1.1 Candidates-Winners approach to detect screen corners

Screen corners are a kind of corners that can be detected directly by their geometric properties. In our literature survey, we investigated many such algorithms to do this kind of things, which are called basic features detection such as Harris-Stephens[Har88], Deriche-Giraudon [Der93], SUSAN [Smi97], Compass [Ruz01], Sojka [Soj02a] etc. All of these algorithms can detect corners in an image and the speed is on a real-time level. However, these detection algorithm still cannot figure out which corners are screen corners, even if they have some parameters to control the corner size other features of the corners. So it is obvious we cannot depend only on these algorithms.

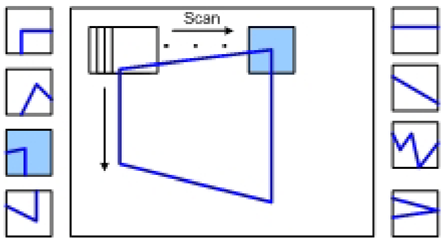


Figure 3.3: Classification model to detect corners in an image. Left column is corners samples, and right column is non-corners samples. The blue sub-window is classified as a corner listed in the left column.

Another way to detect corners, or speaking generally, to detect objects, is to use classification models such as k-Nearest Neighbors [Web02][Zha04], Support Vector Machine [Vap95][Cor95] and Relevance Vector Machine [Tip01]. These models are trained by lots of samples of the objects and after that they can tell which class the object belongs to by doing a simple calculation. To screen corner detection cases, we need to collect plenty of screen corners image samples and put them into one class, and collect other object images as non-corner class. After training, the classifier can scan an image (see Fig. 3.3) and tell where the screen corners are. It seems to be a good way to go, but the problem is that the speed to scan an image is very slow, and also the classification success rate is not so high, which will result in wrong detections.

Given the limitation of these different algorithms and models, we are able to invent a new approach to combine them and avoid their defects. We called it Candidates-Winners approach. In this approach, we use a corner detection algorithm to detect as much as possible corners as screen corners candidates and then use a classification model to classify them and tell which possible four screen corners are. And if there still left some candidates then finally we use a filter algorithm to select out by their geometrical properties. By using this approach, the speed to detect screen corners is raised extremely. That is because after the corner detection algorithms, there are only few positions left as candidate screen corners and the classification model only needs to classify these candidate corners instead of the whole image, which accelerates the detection very much.

In Fig. 3.4 we show the Candidates-Winners approach and the models and algorithm finally used in all modules, Sojka algorithm for corner detection, RVM as classification model, and Rectangle filter algorithm. The reasons to choose them are explained in our literature survey. We will give a more practical explanation in Part II.

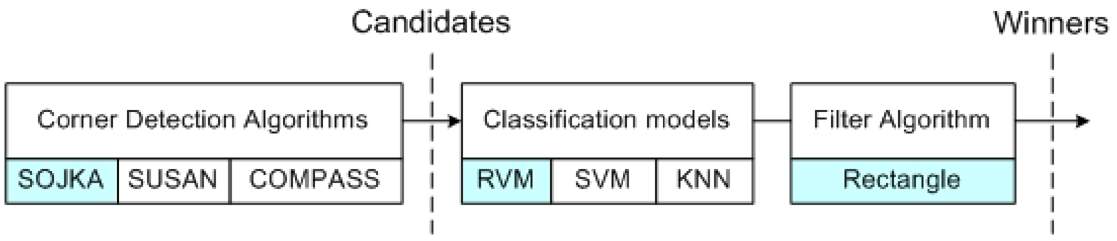


Figure 3.4: Candidates-Winners approach. The algorithms or models with green background color are used in the UI-Wand system.

3.1.2 Pointing projection model

When we detect screen corners and get the four screen corners image coordinates, then we need to use Pointing Projection model to find out the screen coordinates of the center of the camera image, P_s of pointing point shown in Fig. 3.2. This pointing positioning algorithm is an existing algorithm in PHILIPS lab. Its principle is described as follows:

Firstly, the algorithm defines the spatial configuration by using two coordinates systems (given by orthonormal vectors):

1. World coordinates system: $(\mathbf{w_x}, \mathbf{w_y}, \mathbf{w_z})$.
2. Camera coordinates system: $(\mathbf{c_x}, \mathbf{c_y}, \mathbf{c_z})$.

For simplicity the world coordinates system's origin is assumed to be centered at the mid-point of the visible screen area (see Fig. 3.5a), and that coordinates direction vectors $\mathbf{w}_x$ and $\mathbf{w}_y$ correspond to the screens x and the screens negative y direction (i.e. if the screen is mounted in the "regular" way, $\mathbf{w}_x$ is going to the right and $\mathbf{w}_y$ is pointing upwards). As a consequence, the world z-coordinate is pointing towards the user when standing directly in front of the screen. The camera's coordinates system is arranged in a way such that $\mathbf{c}_x$ and $\mathbf{c}_y$ correspond to the usual counting of pixel coordinate on the camera sensor. The third coordinate vector $\mathbf{c}_z$ then corresponds to the pointing direction (see Fig. 3.5b).

Now a spatial configuration of the pointing device can be described (in the world coordinates system) by the following data:

1. The world coordinates of the pinhole point (i.e. the centre of the camera's lens) (see Fig. 3.5c).

2. An orthonormal matrix O = ($\mathbf{c}_x$ $\mathbf{c}_y$ $\mathbf{c}_z$) representing the camera coordinate vectors.

Projective geometry now allows computing the image of a world point on the image plane. The world-to-camera projection $\pi_{P,O}$ depends on the pinhole position P and the camera orientation O is given by

$$\pi_{P,O} : (x, y, z) \rightarrow (I_x, I_y),$$

$$I_x = \frac{\boldsymbol{\delta} \cdot \mathbf{c}_x}{\boldsymbol{\delta} \cdot \mathbf{c}_z},\ I_y = \frac{\boldsymbol{\delta} \cdot \mathbf{c}_y}{\boldsymbol{\delta} \cdot \mathbf{c}_z} \text{ with } \boldsymbol{\delta} = \begin{pmatrix} x - P_x \\ y - P_y \\ z - P_z \end{pmatrix} \tag{3.1}$$

The procedure to estimate a new pointing device position (P', O') can now be described as a minimization problem:

$$(P',O') = \operatorname{argmin}_{(P,O)} \sum_{i=1}^{4} \left(\pi_{P,O}(C_i) - D_i\right)^2 \tag{3.2}$$

Here we assume that the screen is represented by four corners points labeled $C_1 \dots C_4$ (given in real world coordinates) that correspond to four corner detections in the image $D_1 \dots D_4$ (given in image coordinates). This optimization is generally hard and time consuming, so in the case that the screen position had been found already in the previous frame, and the optimization has already been done for that frame, then we use the previous position (P, O) as a starting point for gradient descent search, which speeds up the optimization significantly.

After figuring out the pointing device position (P', O') in world coordinate, the camera pointing position in the visible screen area (see Fig. 3.5a) can be easily calculated and then the computer cursor can be drive to the right place on the screen.

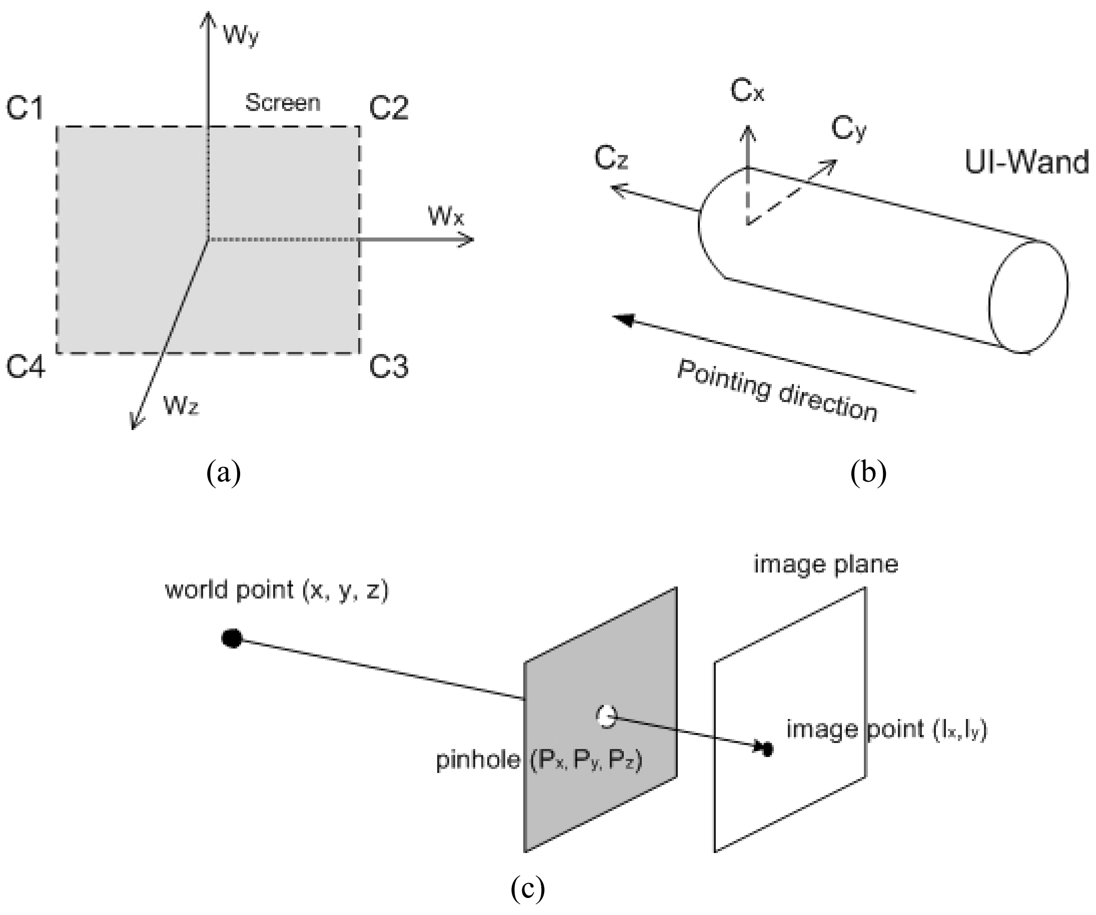


Figure 3.5: Positioning spatial transformation figures. (a) World coordinates. (b) Camera coordinates. (c) The projection of a world point on the image plane.

3.2 Tracking and gesture recognition

If we can accurately detect screen corners and figure out where the pointing point is in a screen coordinate system, the next step is to recognize what gesture the user is making with UI-Wand. There are a lot of works on gesture recognition, like hands gesture recognition [Bla98][Lee99][Hon02], behavior recognition [Psa02], and so on. From these papers, we concluded that the general existing procedure of gesture recognition is critical point tracking and states transformation model to recognize.

In our UI-Wand case, we use a ROI (Region of Interest) track filter to track the screen corners detected in last frames and then, instead of HMM used by others, we use RVM to realize gesture recognition.

3.2.1 ROI Tracking algorithm

The basic method to detect screen corners in a sequence of frames is to use Candidates-Winners approach in every frame, so that we can detect screen corners and find the accurate pointing point consecutively. But since we already know the positions in the last frames, and given a sequence of frames, it is not effective to let Candidates-Winners approach detect screen corners on a whole image. A more efficient way to do that is just to detect a certain area around a screen corner detected in the last frame, which we call ROI (see Fig.3.6). Considering the motion of the screen is not too fast, the size of ROI is small, so that the speed of detection on several ROI get much faster than using Candidates-Winners approach on a whole image. In addition, if we can consider more frames before, then we can use some motion estimation algorithms to predict screen corners in the next frame, such as Kalman filter [Kal60][Web01] or simple motion estimation filter. Then use the predicted point as ROI center and detect, which will reduce ROI size and make the detection faster and more accurate.

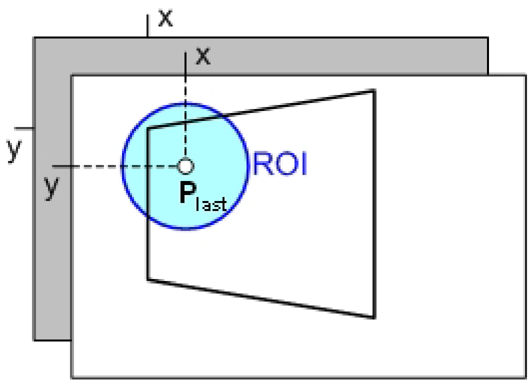


Figure 3.6: One ROI in a frame with a screen inside. P_{last} is the screen corner detected in the last frame drawn in gray color.

3.2.2 RVM model to gesture recognition

The popular model to recognize gestures is HMM or improved HMM mentioned in many papers [Psa02][Lee99]. The reason to use this model is obvious, because the gesture is a point movement in a space (2D or 3D) and is an ordered sequence of states in the spatial-temporal space, which can be described perfectly by HMM. So in our paper, we present how to use HMM to solve our gesture recognition in UI-Wand.

But in the UI-Wand system developed by us, we do not use HMM. Instead we invent another method to recognize gestures, which uses RVM. RVM has proved to be a good model to classification problems. So we think it also can work well in a gesture recognition task. But how successful it can be as an object classifier for states transformation cases, will depend on how gestures features are extracted. Finally we find out that the consecutive motion vectors are the best features to describe a gesture. In order to get this feature, we need to do some trace analysis before. For example, we only need to deal with those traces that consist of fast movement points instead of a trace with points at the same position in some consecutive frames. And also before extracting the features of a trace and giving it to RVM model, we need to know where is the start point of the trace and where is its end point. All of these analyses and final feature extraction work are done in an interpolation trace analysis model that will be described in detail in Chapter 7.

By using motion vectors as features, RVM gives a promising recognition rate for a certain set of gestures we defined. In Chapter 7, we will give more descriptions and test results to show its theory and performance. Fig.3.7 shows a gesture recognition procedure.

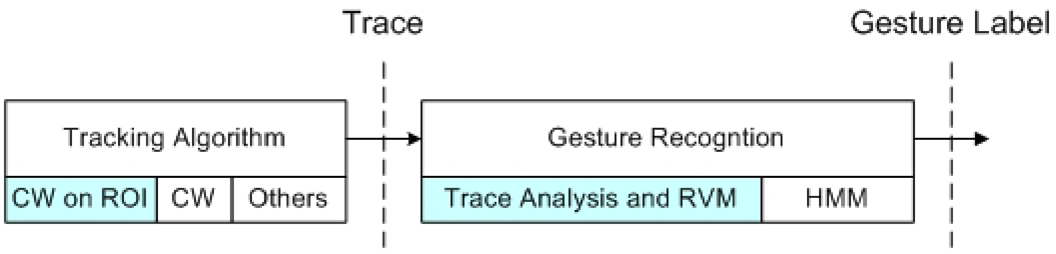


Figure 3.7: Gesture recognition procedure. The algorithms or models with green background color are used in the UI-Wand system. CW means Candidates-Winners approach.

Part II

Models and Algorithms in UI-Wand

4
Corner Detection

In Chapter 3, we have already described our Candidates-Winners approach, which is used to find four screen corner positions in one image. In this chapter, we will describe the corner detection algorithm that we used in our Candidates-Winner approach. The comparison of different corner detection algorithms will be introduced in section 4.1, the main idea of Sojka corner detector and the theoretical foundations of the algorithm will be explained in details in section 4.2, the realization procedures will be depicted in section 4.3, the parameters optimization will be discussed in section 4.4 and finally the test results and some analysis of the algorithm are shown in section 4.5.

4.1 Corner detection algorithms comparison

With respect to its practical applications, corner detection in digital images is studied intensively for approximately three decades. Many algorithms for detecting corners have been developed up to now. They may be divided into two groups. We named these two groups as direct corner detectors and color distribution based corner detectors [Ale98].

The first group contains the algorithms that work directly with the values of brightness of images. They often model an image as a surface; considering the directional gradients or derivatives at each pixel point, if the value of its corner response function exceeds a defined threshold, this pixel will be considered as a corner point. The detectors described in [Bea78], [Kit82], [Har88], [Nob88], [Der93], [Tra98], [Wan95], [Zhe99], [Soj03], all belong to the first group (Fig.4.1 shows the part of image surface containing a corner that is described in [Har88]). Normally each algorithm has its own particular corner response function. So their performances are usually decided by the corner response function that they choose.

The second group does not model an image as a surface. Instead, they consider the statistical color distribution in a circular neighborhood centering at each pixel rather than compute the directional gradients or derivatives. SUSAN corner detector [Smi97], Compass corner detector [Ruz01], [Ruz99a], [Ruz99b] and a proposed detector [Son03] belong to the second group. SUSAN detector classifies each pixel into edge, corner and flat area by checking the USAN principle, this principle is shown in Fig.4.2. Compass detector utilizes a group of colors, instead of a single color, to represent the statistic color distribution in a circular neighborhood. It can handle both uniform-colored region and textured regions. The detector proposed in [Son03] emphases both spatial and statistical color distributions, it is much faster than Compass detector and more accurate than SUSAN detector.

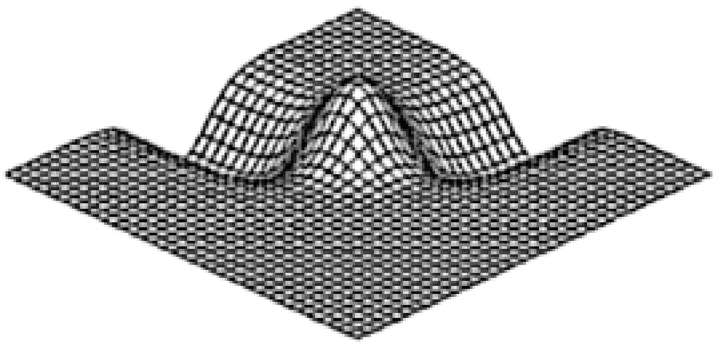


Figure 4.1: A part of image surface containing a corner in the centrum of it (this surface is defined in [Har88]).

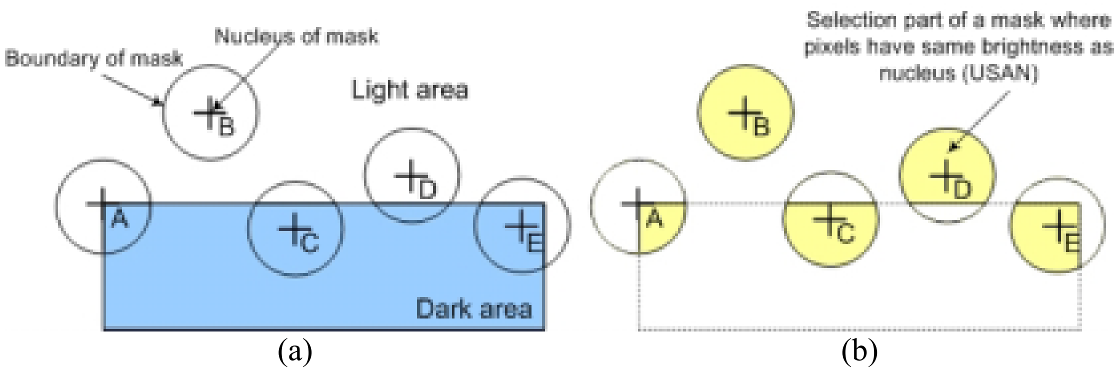

Boundary of mask
Nucleus of mask
Light area
A
B
C
D
E
Dark area
(a)
Selection part of a mask where pixels have same brightness as nucleus (USAN)
A
B
C
D
E
(b)

Figure 4.2: USAN principle in SUSAN detector. (a) Five circular masks locate at different places. (b) Five USAN areas (yellow areas).

Table 4.1 shows the evaluation data from different corner detection algorithms. Comparing these corner detection algorithms, we found that the color distribution based algorithms have very good accurate corner detection rate (especially in [Ruz01], [Ruz99a], [Ruz99b] and [Son03]), but the detection speed is not so fast because they often detect the edge, corner, and junction in one round. These algorithms can detect junctions (for example T-junctions and X-junctions) much better than other detectors, so they are more appropriable for off-line textured image analysis. But in our project, we must reach the real-time processing speed, so we need a very high detection accuracy rate for normal screen corners (not considering junctions or edges). The algorithms described in [Ruz01] and [Son03] are not appropriable for us. SUSAN detector is the fastest one among color distribution based algorithms, but the detection accuracy rate is not so ideal.

Table 4.1: Comparison of different corner detectors.

Index	**Feature Detected**	**Accuracy**	**Speed**	**Real-time Processing**	**Grade**
[Bea78]	Corner	Low	Very fast	Yes	3
[Kit82]	Corner	Low	Fast	Yes	3
[Har88]	Corner	High	Middle	Yes	4
[Der93]	Corner	Very low			2
[Smi97]	Edge & Corner	Low	Middle	Yes	3
[Ruz01]	Corner, Edge, & Junction	Very high	Very slow	No	3
[Son03]	Corner, Edge & Junction	Very high	Middle	Yes	4.5
[Soj03]	Corner	Very high	Middle	Yes	5

After concluding the drawback of color distribution based algorithms, we decided to investigate the direct corner detectors. The processing speed of early direct corner detectors is very fast, but they often have not very high detection accuracy rate, which is the biggest problem of them. But in [Soj03], we can find that this problem has already been resolved very well, it makes great improvement in the detection accuracy rate and its processing speed is just a little slower than the others. We can see this improvement from Table 4.2 (the testing image with reference corners is shown in Fig. 4.3). We now name the corner detection algorithm described in [Soj03] as Sojka corner detector. For the reason that Sojka corner detector satisfies almost all the requirements of corner detector in our project (high accuracy rate, fast speed for real-time processing), we decided to choose it as our corner detector in UI-Wand project. In the next section, we will analyze the main problems caused by the other direct corner detectors and outline the main ideas of the Sojka corner detector.

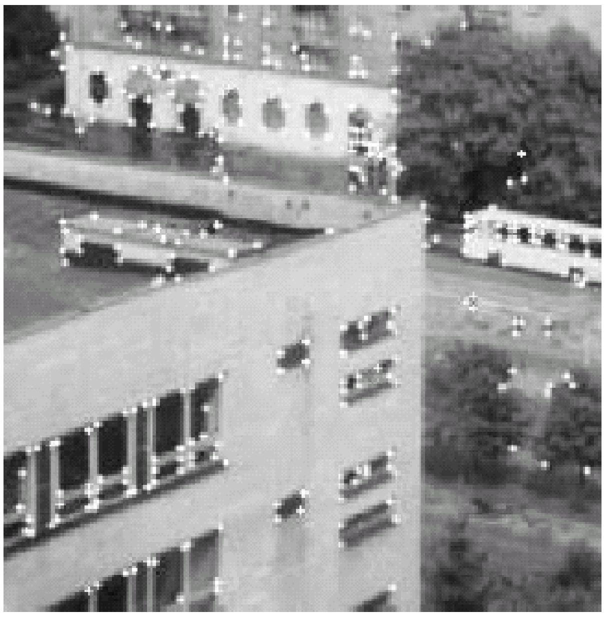


Figure 4.3: Reference corners in a testing image (291 reference corners in total).

Table 4.2: Comparing of direct corner detectors (we also add SUSAN here).

Detector Name	Total Corners	Correct Detections	False Detections	Multiple Detections	Total Error	Localization Error	Grade (out of 5)
[Bea78]	291*	155	21	10	167	1.85	2.0
[Der93]		142	25	10	184	2.05	1.5
[Har88]		187	10	6	120	0.98	3.5
[Kit82]		163	26	15	169	1.87	2.0
[Smi97]		152	29	1	169	1.63	2.5
[Soj03]		229	9	8	79	0.81	4.5

* The total corners number is the reference corner number in the real image.

4.2 Sojka corner detection algorithm

4.2.1 Main ideas of Sojka corner detector

The majority of the existing direct corner detectors determine the values of a corner response function. In a given point of the image, its value is computed by examining the function of brightness and/or its derivatives in a certain neighborhood of this point. The value of the corner response function usually reflects the angle and the contrast of the corner. The corners are detected at those points at which the value of the corner response function is greater than a chosen threshold and at which, at the same time, the function exhibits its extremum. We see two major problems in these approaches:

1. The known detectors work well if the situations in the neighborhoods are not complicated. In more complicated situations, problems can arise. Considering the situation that is depicted in Fig. 4.4. In the areas a, b and c, the magnitude of the gradient of brightness is non-zero. In Fig. 4.4, for determining the angle of the corner at *Q*, obviously only area a is relevant, the area b is obviously less substantial, and the area c is even almost irrelevant. The existing corner detectors do not take into account this fact and unselectively examine the whole neighborhood of *Q*, which may lead to an error decision on whether or not *Q* is a corner. This drawback cannot be avoided by using small neighborhoods. In a small neighborhood, all the measurements would not be precise and reliable enough mainly due to noise.

2. A substantial drawback of the known corner detectors is caused by thresholding the values of the corner response function. Let α stands for the angle of the corner (see Fig. 4.4). Supposing that we can measure this angle. In digital images, however, we can do this only with a certain limited precision. It is obvious that if the difference $|\pi - \alpha|$ is less than the precision that can be achieved under the conditions of the measurement, the point should not be detected as a corner. The known corner detectors do not check the angle value and use usually only the corner response function for thresholding. Since this function combines the angle and the contrast of the corner, it may happen that a small value of the difference $|\pi - \alpha|$ is compensated by a high value of contrast, which leads to erroneous detections of corners on contrast edges. The problem cannot be avoided by increasing the threshold, which, in turn, would lead to miss the corners with a lower contrast.

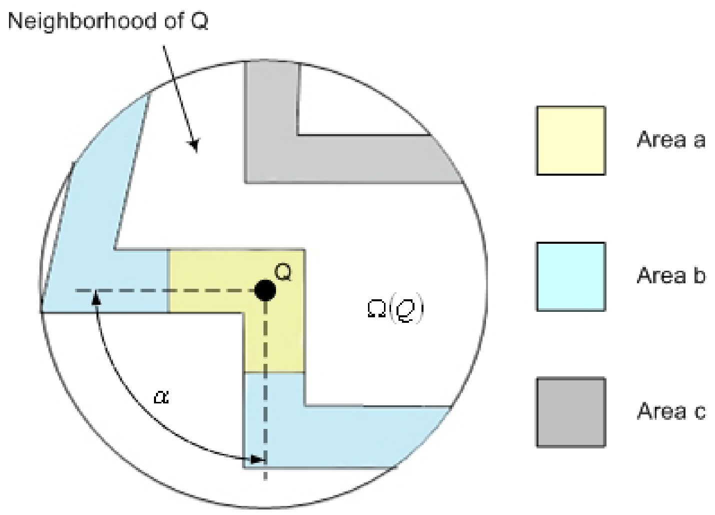


Figure 4.4: A neighborhood of pixel Q. In Area a, b and c, the magnitude of gradient of brightness is non-zero values. Area a is relevant for determining the angle of corner (denoted by α) at pixel Q. Area b and c are all irrelevant for determining the angle of corner at pixel Q.

Coming from the knowledge stated above, the Sojka corner detection algorithm [Soj02a][Soj02b][Soj03] also determines the corner response function that combines the angle and the contrast of the corner. The function is designed in such a way that it exhibits its local maxima at corner points. The main new features of the algorithm are the following:

1. The new algorithm exploits the information contained in the neighborhood $\Omega(Q)$ selectively. It determines which areas of neighborhood are relevant for determining whether or not Q is a corner. It is done by introducing the probability $P_{SG}(X)$ of the event that a point X (an arbitrary point in the neighborhood) belongs to the approximation of the straight segment containing Q of the isoline of brightness. In Fig. 4.4, for example, the values of $P_{SG}(X)$ are high at the points that form the area a, but at all the other points, the values of $P_{SG}(X)$ are low. The value of $P_{SG}(X)$ can be computed from the values of the function of brightness and its gradient by making use of the Bayesian estimations.

2. The algorithm includes the explicit computation of the corner angle. The expected precision of the angle measurement is estimated. A point Q can only be accepted as a corner if the difference is significantly greater than the estimated precision of the angle measurement.

3. A quantity expressing the obviousness of the corner is computed. This value, in essence, characterizes the size of the area that is relevant for deciding whether or not Q is a corner (i.e., the area a in Fig. 4.4) and the magnitude of the gradient of brightness in this area. A point can only be accepted as a corner if its obviousness is greater than a predefined threshold.

4.2.2 Sojka corner detection theoretical foundations

4.2.2.1 Corner model

Sojka corner detector defines a function $\psi(\xi)$ to represent the value of brightness in the direction across the edges and its derivative with a single extremum at $\xi = 0$, which is regarded as an edge point. The edge is often oriented by the rule that the higher brightness lies to the left and the lower brightness lies to the right. Because the corner is an intersection of two non-collinear straight edges, so the shape of a corner can be defined as a model. In this model, the gradient direction of brightness along one edge that comes into the corner and along the other edge that comes out from the corner are defined as two angles φ_1 and φ_2, where $\varphi_1, \varphi_2 \in \langle 0, 2\pi)$. Let $\mathbf{n}_i = (\cos\varphi_i, \sin\varphi_i)$ with $i = 1,2$, which are the direction vectors of the edges' gradient. The axis of the corner is a line passing through the corner along the direction of increasing brightness and halving both the corner angle and the angle between φ_1 and φ_2. It follows that the angle between the corner axis and φ_i is always no bigger than $2/\pi$.

The corner is always convex or concave; if $\mathbf{n}_1 \times \mathbf{n}_2 > 0$, it is convex and if $\mathbf{n}_1 \times \mathbf{n}_2 < 0$, it is concave. Now "Sojka Corner Detector" defines the function of brightness at a point X as follows:

$$b(X) = \begin{cases} \min\{\psi(\mathbf{n}_1 \cdot (X - C)), \psi(\mathbf{n}_2 \cdot (X - C))\} & \textit{if } \mathbf{n}_1 \times \mathbf{n}_2 \geq 0 \\ \max\{\psi(\mathbf{n}_1 \cdot (X - C)), \psi(\mathbf{n}_2 \cdot (X - C))\} & \textit{otherwise} \end{cases} \tag{4.1}$$

Let $b(X)$, $g(X)$ and $\varphi(X)$ denote the value of brightness, the magnitude of the brightness gradient and the direction of the brightness gradient at the image pixel X. The algorithm works with the samples (pixels) in an image. Here it is assumed that the value of sample at point X is equal to the value of the corresponding continuous theoretical function at X. This holds for all the mentioned functions ($b(X)$, $g(X)$ and $\varphi(X)$).

4.2.2.2 Principles of the Sojka corner detector

From the definition of the corner model we can introduce how Sojka corner detector works now. Let Q be an image pixel point and Ω be its disc shape neighborhood (see Fig. 4.5), which contains a finite number of isolated points, for example X. As the definition of the theoretical isoline curves, there must exist a certain curve that passes the area of the pixel point X, but not really passing through X. So X is the approximation and lies in a certain distance from of the theoretical curve. This distance can be defined as the deviation $d(X)$ and its sign indicates on which side of the curve that X lies. The deviations of the pixel points approximating a curve may be regarded as a random variable and its probability density can be presented by $p_d(z)$. Considering the isoline curve whose brightness is equal to the brightness of Q (see Fig. 4.5, the black continuous line passing through Q).

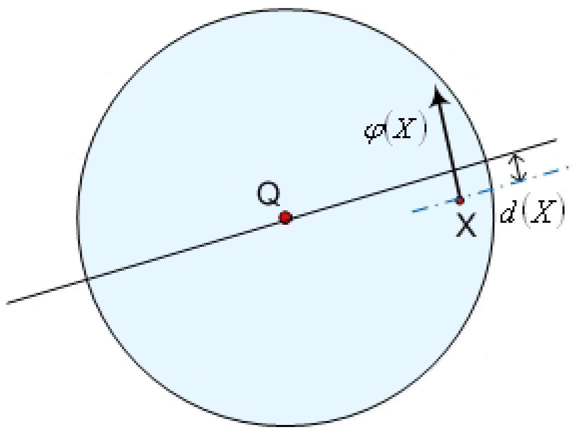


Figure 4.5: The neighborhood of a pixel point *Q* (blue circle). The isoline of brightness is passing through *Q* (black continuous line). A pixel point *X* is the approximation of the isoline (the deviation *d(X)* of *X* from the segment).

The difference of the brightness at X (respect to Q) is defined by $\Delta b(X) = b(X) - b(Q)$. From the former definition of the pixel point brightness (Eq. (4.1)), the distance between X and Q in the direction across the isoline is the deviation of X from the isoline passing through Q, so it can be calculated by the inverse function ψ^{-1} as:

$$d(X) = \xi(X) - \xi(Q) = \psi^{-1}(b(X)) - \psi^{-1}(b(Q)) \tag{4.2}$$

The difference of brightness at X is:

$$\Delta b(X) = b(X) - b(Q) = \psi(\xi(Q) + d(X)) - b(Q) \equiv \beta(d(X)) \tag{4.3}$$

The function $\beta(.)$ is for brevity introduced as the brightness difference function. Conversely the deviation of X from the isoline $b(Q)$ can be determined by the difference of the brightness $\Delta b(X)$ as:

$$d(X) = \beta^{-1}(\Delta b(X)) = \psi^{-1}(b(Q) + \Delta b(X)) - \xi(Q) \tag{4.4}$$

The difference of brightness Δb at the points of approximation may also be regarded as a random variable. Let BrQ denote the event that a point belongs to the approximation of the isoline whose brightness is $b(Q)$. Then the conditional probability density $p_{\Delta b}(z|BrQ)$ can be introduced. Taking into account the fact that for a derivation l from the isoline, the difference of brightness is $\beta(l)$, it is very easy to obtain the equation:

$$p_d(l)dl = p_{\Delta b}(\beta(l)|BrQ)d(\beta(l)) \tag{4.5}$$

It follows that:

$$p_{\Delta b}(z|BrQ)=\frac{p_d(\beta^{-1}(z))}{\beta'(\beta^{-1}(z))} \tag{4.6}$$

where $\beta' = d\beta / dl$ and impractical computation $z = \Delta b(X)$ and $l = d(X)$. Applying Eq. (4.3) and taking into account the model of the corner, it is easy to get:

$$\frac{d\beta(d(X))}{d(d(X))}=\frac{d}{d(d(X))}[\psi(\xi(Q)+d(X))-b(Q)]=\frac{d}{d(\xi(X))}\psi(\xi(X))=g(X) \tag{4.7}$$

where $g(X)$ is the magnitude of the brightness gradient. By substituting Eq. (4.7) into Eq.(4.6) and combining the result with Eq.(4.4), the following can be obtained:

$$p_{\Delta b}(\Delta b(X)|BrQ)=\frac{p_d(\beta^{-1}(\Delta b(X)))}{g(X)}=\frac{p_d(d(X))}{g(X)} \tag{4.8}$$

The result obtained until now is conform the way that how the isoline of brightness is viewed in a continuous image. It is sufficient to set $p_d(z)=\delta(z)$, where $\delta(z)$ stands for the Dirac delta function. Finally $p_{\Delta b}(z|BrQ)=\delta(z)$ can be obtained, which is the expected result.

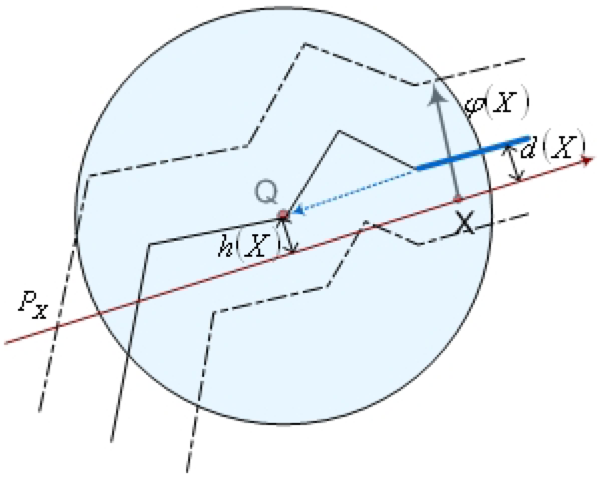


Figure 4.6: The isoline of brightness passing through Q (black continuous line). An isoline segment is aiming at Q (blue bold line); the deviation *d(X)* of pixel point *X* from the segment. The deviation *h(X)*of *Q* from *Px* (deep red continuous line).

Now assume that Ω contains one or more corners conforming to the former corner model. The isolines of brightness are then formed by sequences of line segments. Considering an isoline segment lying on a straight line passing through Q (see Fig. 4.6) but the segments on the same isoline may not necessary pass through Q. X is a pixel point of the approximation of the segment and let p_x denote the straight line perpendicular to the direction $\varphi(X)$ passing through X and orienting the same way as edges. Let $h(X)$ be the deviation of Q from p_x (its sign is negative if Q lying to the left of p_x. The same as d, the deviation h for the approximation points of the segment of an isoline may be regarded as a random variable. Let $DirQ$ to denote the event that a segment of an isoline aims at the pixel point Q, and the conditional probability density $p_h(z|DirQ)$ of the deviation h can be introduced. Then from the discrete sampling of the image, it is very clearly that $h(X)=d(X)$ and $p_h(z|DirQ)=p_d(z)$ (see Fig. 4.6).

From the above relations and definitions, the two conditional probabilities can be introduced. One conditional probability is $P(BrQ|\Delta b(X))$, which is of the event that a point belongs to the approximation of the isoline whose brightness is $b(Q)$ providing that the difference of brightness is $\Delta b(X)$. The other one is $P(DirQ|h(X))$, which is of the event that a point belongs to the approximation of an isoline segment aiming at Q providing that the deviation from Q to X in the direction of brightness gradient is $h(X)$. Using Bayes' formula yields the result:

$$P(BrQ|\Delta b(X))=p_{\Delta b}(\Delta b(X)|BrQ)\frac{P(BrQ)}{p_{\Delta b}(\Delta b(X))} \tag{4.9}$$

$$P(DirQ|h(X))=p_h(h(X)|BrQ)\frac{P(DirQ)}{p_h(h(X))} \tag{4.10}$$

Because it is already be confirmed that $p_h(z|DirQ)=p_d(z)$, Eq. (4.10) can be transferred into:

$$P(DirQ|h(X))=p_d(h(X))\frac{P(DirQ)}{p_h(h(X))} \tag{4.11}$$

4.2.2.3 Probability estimations

In order to apply these equations from above into the practical calculation, it is necessary to carry out the estimations of the probabilities $P(BrQ)$, $P(DirQ)$ and the probability densities $p_{\Delta b}(z)$, $p_d(z)$. Suppose the size of the neighborhood area Ω is A and the size of a pixel point area is A_0. The linear size (mentioned as *diameter*) of Ω and one pixel may be considered by the values $\sqrt{A}$ and $\sqrt{A_0}$. For estimating the probability $P(BrQ)$, suppose that l_Q is the length of the isoline in Ω whose brightness is $b(Q)$, and then the isoline must pass through $l_Q/\sqrt{A_0}$ pixels in Ω approximately. For estimating the probability $P(DirQ)$, still suppose that l_Q is the total length of the straight isoline segments in Ω aiming at Q. Because of the discrete pixel sampling of the image, $P(BrQ)=P(DirQ)$ is held (see Fig.4.7). Thus having:

$$P(BrQ)=P(DirQ)\approx\frac{l_Q}{\sqrt{A_0}}\frac{A_0}{A}=\frac{l_Q\sqrt{A_0}}{A} \tag{4.12}$$

The probability density $p_{\Delta b}(z)$ is defined by the expression:

$$p_{\Delta b}(z) = \lim_{\varepsilon \to 0} \frac{A\{X | z < b(X) \le z + \varepsilon\}}{\varepsilon A} \tag{4.13}$$

where $A\{X | z < b(X) \le z + \varepsilon\}$ denotes the size of that part of Ω that the inequality $z < b(X) \le z + \varepsilon$ holds for the brightness. In the practical computation, the value of $p_{\Delta b}(z)$ will only need for $z = \Delta b(X)$.

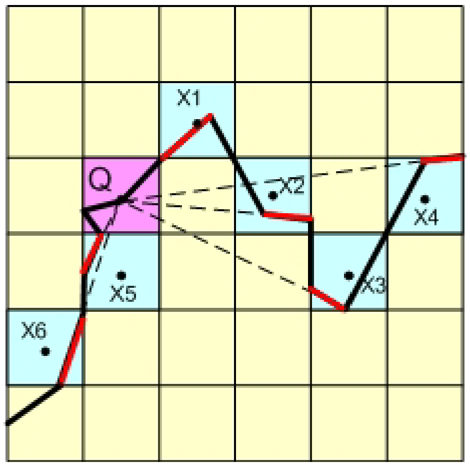


Figure 4.7: A part of the neighborhood of pixel Q (yellow area). The isoline pixel point areas are: X1, X2, X3, X4, X5, X6 (blue area). One isoline curve with the same brightness magnitude is passing through pixel Q (black bold continuous line). Some isoline segments lie on the isoline and is aiming at Q (red bold line lie on the black continuous isoline).

Assume that the length of the isoline passing through X is l_X and that no other points with the same brightness, $b(X)$ in Ω. With respect to the model of corner that is defined before, the magnitude of the brightness gradient remains constant value $g(X)$, along the whole isoline. So the following can be obtained:

$$p_{\Delta b}(\Delta b(X)) \approx \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} \frac{l_X \dfrac{\varepsilon}{g(X)}}{A} = \frac{l_X}{g(X) A} \tag{4.14}$$

Then for determining the probability density $p_h(z)$, supposed that the deviations $h(z)$ vary between the value $-1/2\sqrt{A} \le h \le 1/2\sqrt{A}$ and that the probability of the particular values is evenly distributed. It follows that:

$$p_h(z) \approx \frac{1}{\sqrt{A}} \tag{4.15}$$

From Eq. (4.12), Eq. (4.14) and Eq. (4.15), the estimations can be got, and then substituting these estimations into the expressions from Eq. (4.8) into Eq. (4.9) and Eq. (4.11). Letting $l_Q \approx l_X$ and $l_Q \approx \sqrt{A}$, we can get:

$$P(BrQ|\Delta b(X)) = \sqrt{A_0}\,\frac{l_Q}{l_X}\,p_d\left(\beta^{-1}(\Delta b(X))\right) \approx \sqrt{A_0}\,p_d\left(\beta^{-1}(\Delta b(X))\right) \tag{4.16}$$

$$P(DirQ|h(X)) = l_Q\sqrt{\frac{A_0}{A}}\,p_d(h(X)) \approx \sqrt{A_0}\,p_d(h(X)) \tag{4.17}$$

Let's say that the direction of the corner axis at Q is known, considering the angle difference at X as $\Delta\varphi(X)$ for the angle between the corner axis and the direction $\varphi(X)$. This difference is always non-negative, it is less than or equal to π. According to the explanation in the earlier part of this section, the angle difference for all X that form the area of a corner should satisfy $\Delta\varphi(X) \le \pi/2$, if this condition does not be satisfied, it means that the point X does not belong to the area of the corner candidate Q. Let $AngQ$ denote the event that a point belongs to the possible corner area of Q, and then the conditional probability $P(AngQ|\Delta\varphi(X))$ of the event that a point belongs to the corner area providing that the angle difference is $\Delta\varphi(X)$ can be introduced. Under the assumption that in the theoretical model of sampling that the exact value of $\varphi(X)$ is available for the direction of the corner axis, it may easily write:

$$P(AngQ|\Delta\varphi(X)) = \begin{cases} 1 & \text{if } 0 \le \Delta\varphi(X) \le \pi/2 \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

4.2.2.4 Corner decision

Finally, in the case of a continuous and error free representation of an image, a pixel point X belongs to a straight isoline segment containing Q if the following conditions are satisfied:

1. The brightness at X is equal to the brightness at Q, $\Delta b(X) = 0$.
2. The line p_X passes through Q, $h(X) = 0$.
3. For the angle difference $\Delta\varphi(X)$, the inequality $0 \le \Delta\varphi(X) \le \pi/2$ holds.
4. The above three conditions are all satisfied not only at X, but also at all other points of the line segment $\overline{QX}$.

So a probability $P_{SG}(X)$ may be introduced for the discrete representation of the real image that used up to now. This probability indicates the event that a pixel point X belongs to the approximation of a straight isoline segment containing point Q, and it can be set as:

$$P_{SG}(X) = \min_{Y \in QX}\left\{P(BrQ|\Delta b(Y))\,P(DirQ|h(Y))\,P(AngQ|\Delta\varphi(Y))\right\} \tag{4.19}$$

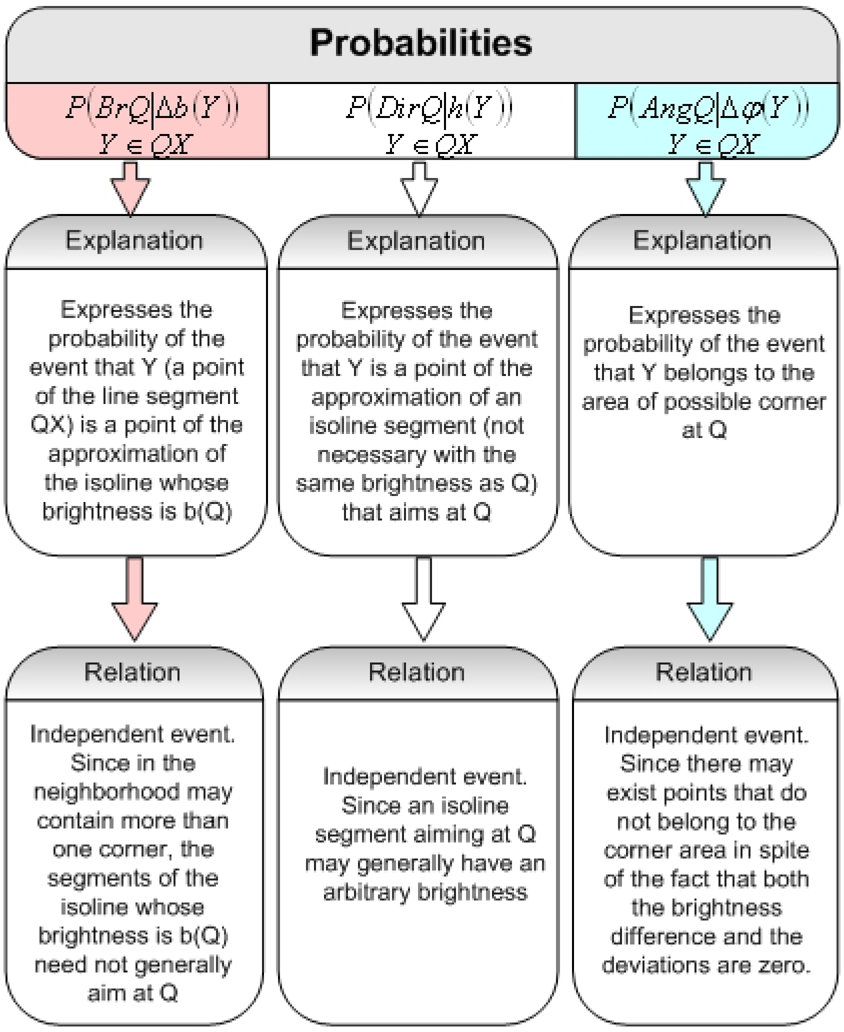

Probabilities
$P(BrQ|\Delta b(Y))$ $Y \in QX$
$P(DirQ|h(Y))$ $Y \in QX$
$P(AngQ|\Delta\varphi(Y))$ $Y \in QX$
Explanation
Expresses the probability of the event that Y (a point of the line segment QX) is a point of the approximation of the isoline whose brightness is b(Q)
Explanation
Expresses the probability of the event that Y is a point of the approximation of an isoline segment (not necessary with the same brightness as Q) that aims at Q
Explanation
Expresses the probability of the event that Y belongs to the area of possible corner at Q
Relation
Independent event. Since in the neighborhood may contain more than one corner, the segments of the isoline whose brightness is b(Q) need not generally aim at Q
Relation
Independent event. Since an isoline segment aiming at Q may generally have an arbitrary brightness
Relation
Independent event. Since there may exist points that do not belong to the corner area in spite of the fact that both the brightness difference and the deviations are zero.

Figure 4.8: Explanations and relations among the expression in Eq. (4.19).

Now let us use Fig. 4.8 to analysis these probabilities presented in Eq. (4.19). From the Fig 4.8, it can be shown that the three probabilities are all independent from the other two. Especially, in some area there may have more than one corner, there may exist points that do not belong to the corner area in spite of the fact that the conditions $\Delta b(X) = 0$ and $h(X) = 0$ are both satisfied. This fact can be detected by the condition $0 \le \Delta\varphi(X) \le \pi/2$ (see Fig.4.9), Y is a point, which satisfies the conditions $\Delta b(X) = 0$ and $h(X) = 0$, but not satisfies the condition $0 \le \Delta\varphi(X) \le \pi/2$).

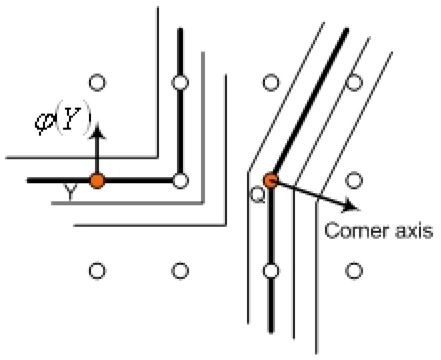


Figure 4.9: The two isoline (black bold lines) have the same brightness and h(Y)=0 is also satisfied at Y, but Y still does not belong to the area of corner at Q. Small circles are the positions of the image pixels.

Now let us confirm that Eq. (4.19) is available for all the points that belongs to the line segment $\overline{QX}$. Supposed that X is a pixel point lying on a straight isoline segment containing Q. It is clear that all the points of the line segment $\overline{QX}$ must lie on the isoline segment too. Fig. 4.10 shows the fact that the conditions $\Delta b(X)=0$, $h(X)=0$ and $0 \le \Delta\varphi(X) \le \pi/2$ do not suffice to decide whether or not X belongs to an isoline segment containing Q. The operator "min" in Eq.(4.19) corresponds to the idea that for any arbitrary points of $\overline{QX}$, a point belongs to the approximation of the straight isoline segment containing Q are dependent among all the points, which means if the event occurs at the point that its probability is the lowest one, it will also occur at the remaining points of the line segment $\overline{QX}$.

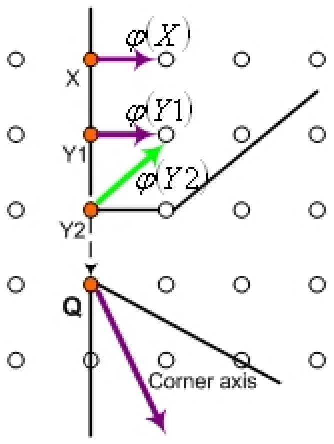


Figure 4.10: An example of neighborhood of Q (although through the isoline segment XY2 on QX, point X and Y1 satisfy all the three condition, but it is still shown that Y2 does not satisfy the condition of angle difference that must be less than $\pi/2$).

After analysis of the Eq. (4.19), then substituting the estimations from Eq. (4.16) and Eq. (4.17) into Eq. (4.19), we will get:

$$P_{SG}(X) = A_0 \min_{Y \in QX} \{ p_d(\beta^{-1}(\Delta b(Y))) p_d(h(Y)) P(AngQ|\Delta\varphi(Y)) \}. \qquad (4.20)$$

Let us determine at Q the "angle of break" of the isoline that passes through Q. We examine the values of $\varphi(X)$ in Ω, because the relevance of the value between Q and X depends not only on the probability $P_{SG}(X)$, but also on the distance between the points Q and X, let us use $r(X)$ to stand for the distance and introduce a weight $w_r(r(X))$, which is sensitive to the distance. So the relevance can be expressed by the following equation:

$$\begin{aligned} w(X) &= P_{SG}(X)w_r(r(X)) \\ &= A_0 \min_{y\in QX}\left\{p_d\left(\beta^{-1}(\Delta b(Y))\right)p_d(h(Y))P(AngQ|\Delta\varphi(Y))\right\}w_r(r(X)) \end{aligned} \tag{4.21}$$

In order to determine the angle of break at Q, we compute the quantities μ_φ and σ_φ^2, which stand for the average weighted edge direction and the average weighted square value of the difference between the edge direction and the average edge direction.

$$\mu_\varphi = \frac{\sum_{X_i\in\Omega} w(X_i)\varphi(X_i)}{\sum_{X_i\in\Omega} w(X_i)\varphi} \tag{4.22}$$

$$\sigma_\varphi^2 = \frac{\sum_{X_i\in\Omega} w(X_i)\left[\varphi(X_i)-\mu_\varphi\right]^2}{\sum_{X_i\in\Omega} w(X_i)} \tag{4.23}$$

Now let us compute for a particular corner candidate Q using μ_φ and σ_φ^2. There are two functions defined for Q:

$$Corr(Q) = g(Q)\sigma_\varphi^2(Q) \tag{4.24}$$

$$Appar(Q) = \sum_{X_i\in\Omega} P_{SG}(X_i)g(X_i)\left|\varphi(X_i)-\mu_\varphi\right| \tag{4.25}$$

These two functions satisfied the following two theorems in the case: there is a corner existing in an image, which is conforming the corner model introduced before. Assume that both the image and the neighborhood Ω are infinite, that $p_d(z)$ is an arbitrary non-negative and symmetric function with a single maximum at $z = 0$, and that $w_r(.)$ is an arbitrary positive and decreasing function defined for non-negative arguments. The following two theorems hold:

Theorem 1: *The function $\sigma_\varphi^2(Q)$ has its maximum just at the points lying on the axis of the corner. At these points, the value $\mu_\varphi(Q)$ determines the direction of the corner axis, and the value $\pi - 2\sigma_\varphi(Q)$ is equal to the angle of the corner.*

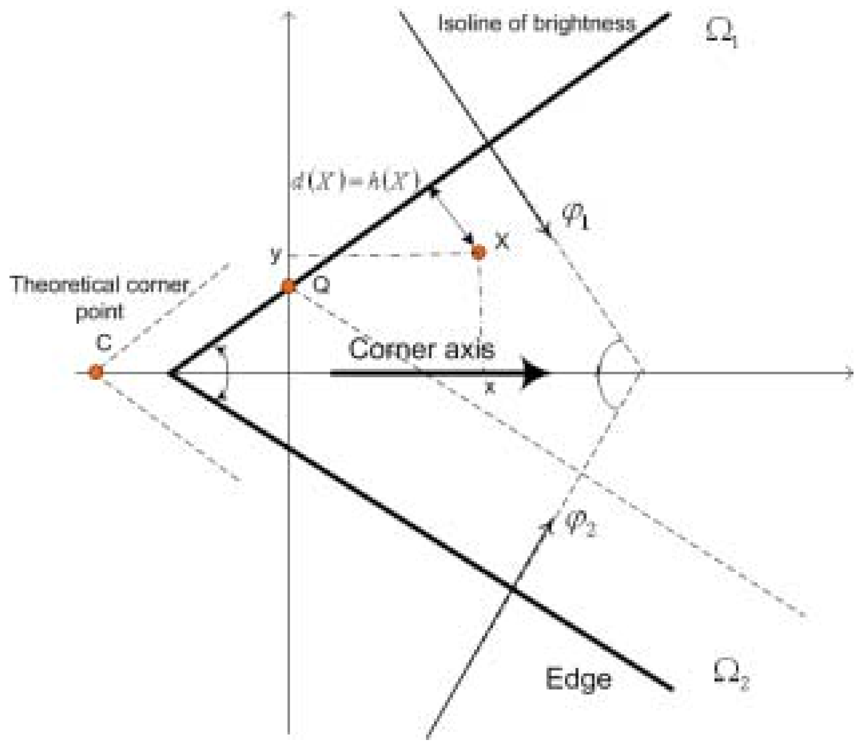


Figure 4.11: The notation used in the proof of Theorem 1.

Proof: In the case being considered, the function $\varphi(X)$ takes only two values, denoted by φ_1 and φ_2 (see Fig. 4.11), we use Ω_1 and Ω_2 to denote each part of the image (a half-plane) where $\varphi(X)=\varphi_1$ and $\varphi(X)=\varphi_2$. Taking into account, in this case, the value of $P(AngQ|\Delta\varphi(X))$ is equal to 1 everywhere, and substituting the first equation form Eq. 4.2 to Eq. 4.21, and then we have:

$$w(X)=A_0\min_{Y\in QX}\{p_d(d(Y))p_d(h(Y))\}w_r(r(X)) \tag{4.26}$$

For any fixed Q, we can compute the values:

$$W_1\sum_{X_i\in\Omega_1}w(X_i) \qquad \textit{and} \qquad W_2=\sum_{X_i\in\Omega_2}w(X_i) \tag{4.27}$$

From Eq. (4.22) and Eq. (4.23), we obtain:

$$\mu_\varphi=\frac{1}{W_1+W_2}(W_1\varphi_1+W_2\varphi_2) \quad \textit{and} \quad \sigma_\varphi^2=\frac{W_1W_2}{(W_1+W_2)^2}(\varphi_1-\varphi_2)^2 \tag{4.28}$$

Under the condition that $W_1>0, W_2>0$ it is very easy to see that the term $W_1W_2/(W_1+W_2)^2$ takes its maximum value 1/4 when $W_1=W_2$. Substituting $W_1=W_2$ into Eqs. (4.28), we can get the result $\mu_\varphi=(\varphi_1+\varphi_2)/2, \sigma_\varphi=|\varphi_1-\varphi_2|/2$, which confirm the **Theorem 1**.

Theorem 2. *Under the assumptions of Theorem 1, the function Corr(Q) has its maximum just at the corner point.*

Proof: From the proof of Theorem 1, it follows that if Q moves along the line parallel to the corner axis, the values of W_1, W_2, therefore, also the value of $\sigma_\varphi^2(Q)$ remain constant. It also follows that the value of $\sigma_\varphi^2(Q)$ decreases with the increasing distance of Q from the corner axis. Because the magnitude of the gradient of brightness $g(Q)$ is maximal at the edge points, as Q departs from the edge, $g(Q)$ decreases. If Q moves along the line that is parallel to the edge, then $g(Q)$ remains constant. Introduce a coordinate system whose origin lies at the theoretical corner point. Let its x-axis and y-axis coincide with the corner axis and with the edge (see Fig. 4.11). View temporarily $Corr(Q)$ as a continuous function and compute its derivatives. It follows that:

$$\frac{\partial Corr(Q)}{\partial x} = \frac{\partial g(Q)}{\partial x}\sigma_\varphi^2(Q), \quad and \quad \frac{\partial Corr(Q)}{\partial y} = g(Q)\frac{\partial \sigma_\varphi^2(Q)}{\partial y} \tag{4.29}$$

We have $\partial\sigma_\varphi^2(Q)/\partial y = 0$ at the points of the corner axis, and $\partial g(Q)/\partial x = 0$ at the edge points. Therefore, with respect to Eq. (4.29), we have $\partial Corr(Q)/\partial x = \partial Corr(Q)/\partial y = 0$ at the corner point. It can be easily seen that the value of $Corr(Q)$ at the corner is a maximum. At a non-corner point, we have $\partial g(Q)/\partial x \neq 0$ and/or $\partial\sigma_\varphi^2(Q)/\partial y \neq 0$. Consequently, taking into account that $g(Q) > 0, \sigma_\varphi^2(Q) > 0$. We can see that $\partial Corr(Q)/\partial x \neq 0$ and/or $\partial Corr(Q)/\partial y \neq 0$, which implies that there exist no other points except the corners giving the maximum of $Corr(Q)$.

4.3 Algorithm realization

4.3.1 Computation procedures

The algorithm of Sojka corner detection is realized in the following way. First the value of the magnitude and the direction of the gradient of brightness ($g(X)$ and $\varphi(X)$ are calculated for all the image pixel points. The derivatives $\partial b(X)/\partial x, \partial b(X)/\partial y$ are replaced by the differences, which are computed using two masks:

$$\begin{bmatrix} -0.1 & 0 & 0.1 \\ -0.3 & 0 & 0.3 \\ -0.1 & 0 & 0.1 \end{bmatrix} \quad and \quad \begin{bmatrix} 0.1 & 0.3 & 0.1 \\ 0 & 0 & 0 \\ -0.1 & -0.3 & -0.1 \end{bmatrix}$$

A pixel point is selected as a candidate for corner when the magnitude of the gradient of brightness is greater than a predefined threshold. Then the selected candidates are checked by determining the values of $\mu_\varphi(Q)$, $\sigma_\varphi(Q)$, $Corr(Q)$ and

$Appar(Q)$. Finally, the candidates at which the value of $Corr(Q)$ exhibits its local maximum and at which the value of $\sigma_\varphi(Q)$ and $Appar(Q)$ are greater than chosen thresholds is decided as a corner.

4.3.2 Explanations of value estimations

There are some explanations about computing the value of $\mu_\varphi(Q)$, $\sigma_\varphi(Q)$, $Corr(Q)$ and $Appar(Q)$. The neighborhood $\Omega(Q)$ is square-shape and centered at pixel Q. The size of $\Omega(Q)$ is always defined within a limited value, but it is not so crucial since the effective size of $\Omega(Q)$ is always determined adaptively by the values of $P_{SG}(X)$. During calculating, the probability density p_d and the weight function $w_r(X)$, the two functions are all fixed estimation functions. We chose p_d to be the normal distribution and $w_r(X)$ to be a Gaussian distribution. Both distributions are with zero means and the values of σ are left as the parameters of the algorithm. In practical, $\psi(\xi)$ is an unknown function and we can not use the Eq. (4.2) and Eq. (4.4) to determine the value of $\beta^{-1}(\Delta b(Y)) = d(Y)$. With the respect of our corner model, we use the estimation value to get it:

$$\beta^{-1}(\Delta b(Y)) \approx \frac{2\Delta b(Y)}{g(Y)+g(Q)} \tag{4.30}$$

Using Eq. (4.30), we can easily get $P(BrQ|\Delta b(Y))$ through Eq. (4.16). But for getting the result from Eq. (4.19), we still need to obtain the probability $P(AngQ|\Delta\varphi(X))$. From Eq. (4.18), we can see that it is necessary for us to get $\Delta\varphi(X)$ beforehand. Until now we do not know the exact value of corner axis direction of Q, we can use the approximation $\varphi(Q)$ for it. Then we get $\Delta\varphi(X) = |\varphi(Q) - \varphi(X)|$ and put this value together with the estimation value of Eq. (4.29) into Eq. (4.20), we can finally obtain the value of $P_{SG}(X)$ from Eq. (4.31):

$$\begin{cases} P_{SG}(X) \approx \min\limits_{Y\in QX}\left\{ p_d\left(\dfrac{2\Delta b(Y)}{g(Y)+g(Q)}\right) p_d(h(Y))\right\} & 0 \le \Delta\varphi(X) \le \pi/2 \\ P_{SG}(X) = 0 & other\ po\text{int}\,s \end{cases} \tag{4.31}$$

Once the value of $P_{SG}(X)$ are available for all $X \in \Omega(Q)$, the value of $\mu_\varphi(Q)$, $\sigma_\varphi(Q)$, $Corr(Q)$ and $Appar(Q)$ can be easily computed using Eq. (4.22), Eq. (4.23), Eq. (4.24) and Eq. (4.25).

4.4 Parameter optimization

In the Sojka corner detection algorithm, we must define some parameters beforehand. Now we will give the definitions of these parameters in details and show how we specify these parameters.

halfPsgMaskSize The half of the mask size. The size of the neighborhood Ω is (halfPsgMaskSize*2+1). Generally, the bigger mask size gives better detection results but the computation time may be longer. The usual values of halfExtMaskSize vary from 4 to 7. So the size of neighborhood Ω should between 9×9 or 15×15 pixels. From the test experience, we concluded that this parameter has not so much effects on the detection results. So we will not change this parameter and fix it on value 4 for faster computing speed.

corrAngleThresh The threshold for the angle of break of the boundary at the corner point. When a pixel point at which the boundary is broken more than this threshold may be accepted as a corner. Usually the value is approximately 0.5 (an angle size in radians) considering the noises in the image. This value is more or less stable for all images. We should choose a higher value of this threshold if we use small masks, because the precision of measuring the angles is generally lower than in bigger masks. Because we just want to detect screen corners, which are big corners in the images, even considering the distortion cases, the corner angel will not be too small. For this reason, we fix this parameter on the value of 0.5 (about 30 degree).

noiseGradSizeThresh The threshold for the size of the gradient of brightness. All the pixel points with gradient magnitude smaller than this threshold value will be considered as noise. So if we increase the threshold, it means some less obvious corners will be missed and if we decrease this threshold, more unobvious corners will be detected but the computation time will be longer than higher threshold. Normally this threshold should be set between 0.04 and 0.08 of image range (the difference between the maximum and the minimum value of brightness in the image). So consider the image noise level and the type of corner we want to detect, we choose higher value for the white screen corner detection and lower value for black screen corner detection, for the reason that the quality of the white screen images is better than the black screen images. This parameter makes a very big effect on the final detection results and also the detection speed, so this parameter need to be specified very carefully according to the quality of the images. We will show the different detection results of the Sojka corner detection algorithm with changing this parameter in the test part of this chapter.

corrApparenceThresh The threshold for the apparent level of corners. The apparent level combines the contrast, size and the shape of the possible corner area. Usually this threshold is set between 0.0 and 5.0 according to the image quality, higher value can be chosen to detect more corners than the lower value. This parameter does not affect so much on the detection speed, but obviously affect the detection results. We will change this parameter in our test part and show this effect.

sigmaD The σ value for the normal distribution of the probability density p_d. Obviously, this value should be less than 1.0. Values between 0.7 and 1.0 are optimal. This parameter is not so crucial for the algorithm and usually we set this value to 0.75 according to the test experiences that we have done during implementation of this algorithm.

sigmaR The σ value for the Gaussian expression of the weight function w_r. The weights depend on the distances from the candidates in the neighborhood Ω. So considering the value of halfPsgMaskSize that we have chosen, we choose 2.5 as its value.

halfExtMaskSize This is the size of the area in which it is checked whether the response function $Corr()$ has its maximum at a pixel point. If there are not so many multiple detection problems appear, it can be set to 1, else we can set to a higher value, 2 or 3, but finally it should not extend the value of halfPsgMaskSize. Here, we set this parameter to 2, because the possible multiple detections could make our Candidates-Winners more difficulties to make the decision on choosing between these multiple detection results.

4.5 Algorithm tests

From the comparing result in section 4.1, we have already seen the outstanding accuracy rate performed by Sojka corner detection algorithm comparing with other algorithms. In order to know the real capability of this algorithm for our UI-Wand system, we use some real images captured by the UI-Wand to do the test. Because in the algorithm, there are a lot of parameters that affect the detection results and they must be specified beforehand. From the last section, we already see the main usage of these parameters and the normal values that can be set to them. But actually only two of them (noiseGradSizeThresh and corrApparenceThresh) show very big influence on the results. So in the test, we will just change these two parameters and show their effects. The parameters and their value ranges are described in Table 4.3, Through the tests we want to know three main capabilities of the algorithm: the accuracy rate, the stability and the speed. All the tests are running under Linux system and on an Intel Pentium IV 2.2GHz computer. We got all the test results from our UI-Wand application Error Analysis utility, which will be introduced in Chapter 11.

Table 4.3: Test parameters.

No.	Parameter name	Changing range
1	halfPsgMaskSize	4 (fixed)
2	corrAngleThresh	0.5 (fixed)
3	noiseGradSizeThresh	3~16 (variable)
4	corrApparenceThresh	0-5 (variable)
5	sigmaD	0.75 (fixed)
6	sigmaR	2.5 (fixed)
7	halfExtMaskSize	2 (fixed)

4.5.1 Accuracy test

In order to know the real detection accuracy by the Sojka corner detection algorithm, we designed this accuracy rate test for the algorithm. In the test, we chose two different type of screens: a PHILIPS brilliance 109MP white computer screen and a PHILIPS brilliance 180P2 black LCD computer screen (see Fig. 4.12). Each sequence has 30 continuous captured images. For each sequence, we changed the value of the two most crucial parameters, and see the differences of the detection number, the four screen corners detection rates, the average variance of the detection screen corners to the real corners and the speed of running the algorithm.

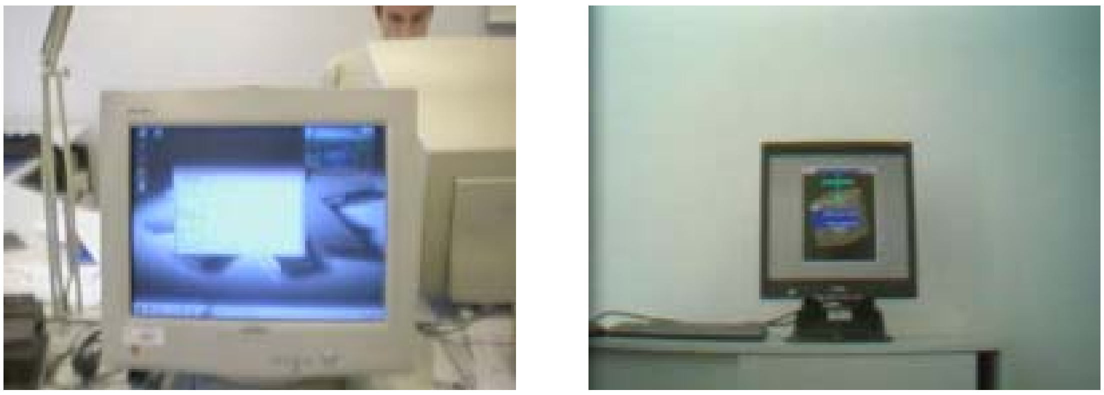

(a) (b)

Figure 4.12 Two different screens we use to do the test. A PHIILIPS brilliance 109MP white computer screen. A PHILIPS brilliance 180P2 black LCD computer screen.

4.5.1.1 White screen sequence test

Test goal

In this test, we chose a white frame sequence, which contains 30 images (see Appendix A) and we changed two of the parameters in Sojka corner detection algorithm to see the effects of these parameters on the algorithm. We must mention that our program just annotates less than 250 corners in the test images, if the algorithm detects more than 250 corners in one image, no corner will be annotated and output (the sign "-" in Table 4.4 stand for this case). The value of parameter values used in this sequence and test result are shown in Table 4.4, where "noiseGradSizeThresh" is set from 8 to 16 for the reason that in this range at least 3 screen corners can be detected and the number of corners detected is acceptable. The result image example is shown in Fig.4.13.

Test result

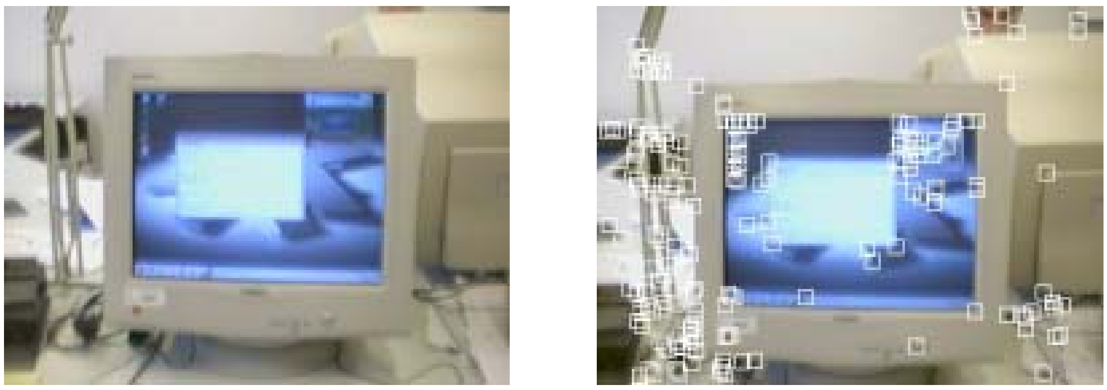

(a) (b)

Figure 4.13: A test result example of Sojka corner detection algorithm with noiseGradSizeThresh =14 and corrApparenceThresh =5. (a) Original image. (b) Detection results.

Table 4.4: Accuracy rate test results (Test sequence see Appendix A).

	noiseGradSizeThresh	**corrApparenceThresh**	**CDN**	**SCDR**	**ADV**	**Speed**
1	8	1	-	-	-	196.33
2	8	3	243	94.67%	1.82	176.00
3	8	5	197	86.67%	1.78	190.33
4	10	1	233	93.50%	1.80	144.33
5	10	3	209	90.00%	1.84	152.67
6	10	5	140	77.50%	1.82	162.33
7	12	1	223	87.50%	1.77	123.33
8	12	3	156	83.33%	1.85	130.33
9	12	5	102	71.67%	1.78	121.67
10	14	1	179	90.83%	1.73	108.67
11	14	3	119	80.00%	1.86	104.67
12	14	5	73	63.33%	1.76	101.00
13	16	1	144	84.17%	1.82	87.33
14	16	3	92	73.33%	1.97	86.67
15	16	5	52	53.33%	1.74	87.33

*CDN: Corner Detection Number
SCDR: four Screen Corners Detection Rate
ADV: Average Detection Variance
Speed: algorithm processing speed in millisecond

Test analysis

From the white screen test results, we can see the parameter "noiseGradSizeThresh" affect the detection accuracy and the detection speed in such a way: when the value is increasing, the corner detection number, the screen corner detection rate and detection speed will decrease; but the average detection variance will be around a constant value.

The parameter "corrApparenceThresh" does not affect the detection speed so much compared with "noiseGradSizeThresh", but it has obvious influence on the corner detection number and screen corner detection rate, the higher values make worse result than the lower values. The average detection variance is still stable.

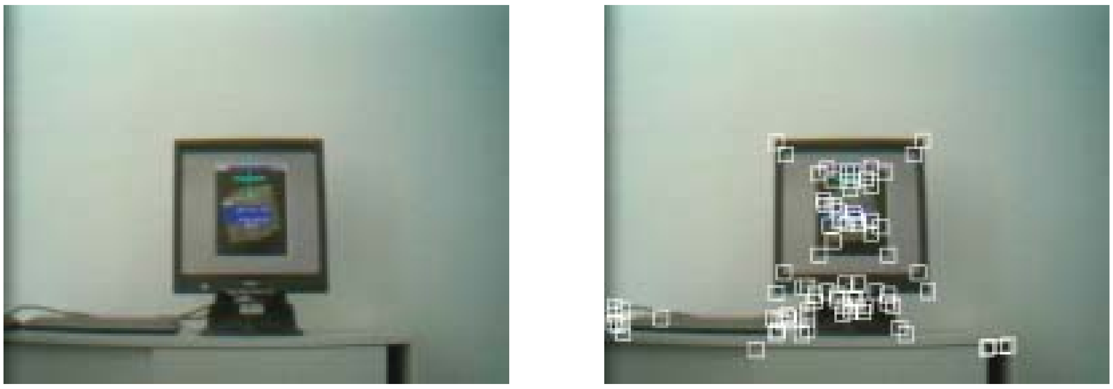
Considering the above situations, we choose a best parameters resolution for this sequence (light gray line shown in Table 4.4), which has best combination of the detection accuracy rate and the detection speed.

4.5.1.2 Black screen sequence test

Test goal

The same as the white screen test, in this test, firstly we chose a black LCD screen sequence for test (see Appendix B). Then we changed two of the parameters in Sojka corner detection algorithm to see the effects of the parameter to the detection accuracy of the algorithm. The value of parameters used in this sequence and test result are shown in Table 4.5, where "noiseGradSizeThresh" is set from 3 to 11 for the reason that in this range at least 3 screen corners can be detected and the number of corners detected is acceptable. The result image example is shown in Fig.4.14.

Test result



(a) (b)

Figure 4.14: Test result example of Sojka corner detection algorithm with noiseGradSizeThresh =5 and corrApparenceThresh =5. (a) Original image. (b) Detection results.

Table 4.5: Accuracy rate test result (test sequence see Appendix B).

No.	noiseGradSizeThresh	corrApparenceThresh	CDN	SCDR	ADV	Speed
1	3	1	151	96.67%	2.05	144.67
2	3	3	120	96.67%	2.05	141.33
3	3	5	100	96.67%	2.08	158.00
4	5	1	103	96.67%	2.06	116.67
5	5	3	109	96.67%	2.06	115.33
6	5	5	75	100%	2.04	113.67
7	7	1	85	96.67%	2.05	97.67
8	7	3	74	100%	2.04	94.33
9	7	5	53	98.33%	2.05	95.67
10	9	1	74	95.83%	2.02	83.09
11	9	3	60	99.17%	2.02	83.00
12	9	5	38	95.00%	2.01	81.56
13	11	1	72	97.50%	1.97	91.67
14	11	3	47	95.83%	1.96	72.33
15	11	5	29	90.00%	1.93	73.67

*CDN: Corner Detection Number
SCDR: four Screen Corners Detection Rate
ADV: Average Detection Variance
Speed: algorithm processing speed in millisecond

Test analysis

The same with the former test, from Table 4.5, we can see the parameter "noiseGradSizeThresh" and the parameter "corrApparenceThresh" affect the detection results in the same manner.

We also choose a best parameters resolution for this sequence (light gray line shown in Table 4.5), which has best combination of the detection accuracy rate and the detection speed.

4.5.2 Stability test

Test goal

Through this test, we want to know the stability of the Sojka corner detection algorithm. Because from some tests among different corner detectors (in [Soj03]), we notice that a lot of algorithms are not stable. When the same images are rotated, the detection results can be very different. In our UI-Wand system, rotation often happens, so the stability of the corner detector is a very important criterion for evaluating the performance of the algorithm. Here, we test this by rotating each image in the black screen sequence with different angles. We use 7 different rotation angles: 0, $\pi/30$, $\pi/20$, $\pi/12$, $\pi/8$, $\pi/4$ and $\pi/2$ for testing (see Fig.4.15). The test parameter is selected according to the best parameter combination getting from black screen accuracy rate test. The test result is shown in Table 4.6.

Test result

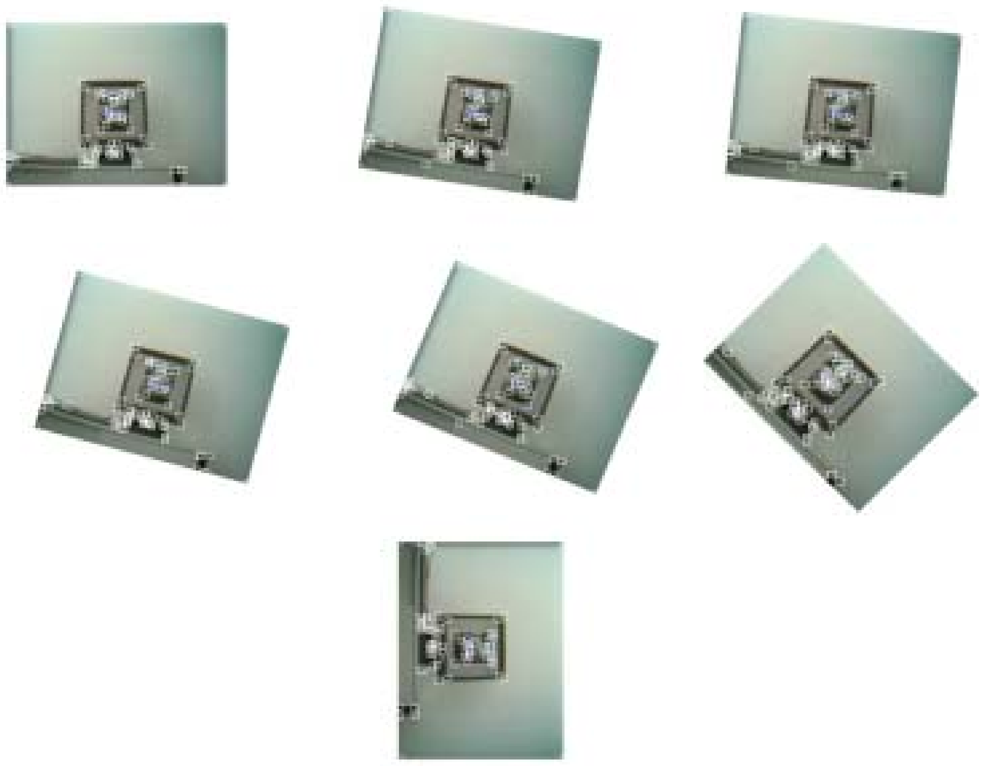


Figure 4.15: Result example for stability test (the rotated angles are along the order: 0, $\pi/30$, $\pi/20$, $\pi/12$, $\pi/8$, $\pi/4$, $\pi/2$).

Table 4.6: Stability test (test sequence see Appendix B).

noiseGradSizeThresh	corrApparenceThresh	Rotated angle	CDN
5	5	0	83
		$\pi/30$	78
		$\pi/20$	78
		$\pi/12$	78
		$\pi/8$	81
		$\pi/4$	83
		$\pi/2$	83

*CDN: Corner Detection Number

Test analysis

From the corner detection number, we can see the good stability of Sojka corner detection algorithm. No matter what angle the image will be rotated the detection result will not be affected. This capability also guarantees the stability of our UI-Wand application.

4.5.3 Sequence test

Test goal

This test is done on the same two screen sequences. From the test results, we want to see the different parameters set for different sequences, the screen corner detection accuracy rate, detection variance and processing speed of the Sojka corner detection algorithm. We chose the best combination of the parameters from the former tests. The test result is shown in Table 4.7.

Test result

Table 4.7: Sequence tests (test sequences see Appendix A and Appendix B).

Parameter name	Black-screen sequence	White-screen sequence
noiseGradSizeThresh	5	14
corrApparenceThresh	5	1
Detection result		
Left_Top corner detected rate	100%	100%
Right_Top corner detected rate	100%	100%
Left_Bottom corner detected rate	100%	100%
Right_Bottom corner detected rate	100%	63.30%
Screen corner detected rate	100%	90.83%
Left_Top corner detected variance	1.96	1.76
Right_Top corner detected variance	2.54	2.3
Left_Bottom corner detected variance	1.59	1.43
Right_Bottom corner detected variance	2.07	1.41
Screen corner detected average variance	2.04	1.73
Speed (in millisecond)	127.33	108.67

Test analysis

From Table 4.7, we can see that in different frame sequences the parameter could be set very differently according to the quality of the processed images. The black screen sequence are more blur than the white screen sequence, so the "noiseGradSizeThresh" must be set smaller. So the value of "corrApparenceThresh" are set higher in the black screen sequence test because we only want to detect large corners and make the corner detection number as few as possible for real-time processing. But for the white sequence test, we do not consider the real-time problems, so we only choose the parameters with a high accuracy rate. Both parameter selections for these two sequences are based on the test results from section 4.6.1.

4.6 Conclusions

The test result shows that for the whole black screen sequence, the four screen corners can be accurately detected by the algorithm with a fast detection speed. As to the white screen sequence, the algorithm can detect three corners correctly, but always makes mistake in detecting the fourth corner (Right Bottom corner). This problem is caused by gray level images. In order to accelerate the speed of the algorithm, in reality we run Sojka corner detection algorithm on gray level image. By discarding color information, we can get a three times faster processing speed of the algorithm, which let our final system reach the real-time requirement. But the problem it brings is that it will make some inaccurate detection when the pixels with different color have the same values in gray. The missing Right-Bottom corner is just this case. From Fig. 4.17 we can see that there is a light blue tools bar on the bottom of the display on the screen. Though we can easily observe that it is different with the gray color of the screen border, if transfer this light blue tools bar to gray level then it is the same with the screen border. We tried to avoid this problem by using Sojka corner detection algorithm on color images, but the test results proved that the speed got much decrease as we expected, which heavily affect its real-time processing feature, thus in the final system, we still use it for gray level images.

5 Corner Classification

With a lot of screen corner candidates detected, what we need to do in next step is to find out the real screen corners among them. As we already introduced in Chapter 3, we will use a classification model to do this job. Considering screen corners as an object and use lots of samples to train a model then let it decide whether a corner is a screen corner or not. Among classification models, k-Nearest Neighbors and Support Vector Machine are two popular methods suitable for object classification problems, but both of them have some disadvantages, therefore, we choose another new model, Relevance Vector Machine, to finish the classification task in this step. In section 5.1, we compare some existing classification models, and then in section 5.2 we start introducing the theory of RVM models. In section 5.3 we describe the important step before classification and feature extraction. In section 5.4 we discuss the sample dataset selection, which affects the performance of RVM very much. Finally our conclusion is drawn in section 5.5

5.1 k-NN and SVM models for classification

Before understanding RVM, let us recall k-NN and SVM, especially SVM, upon which RVM is based.

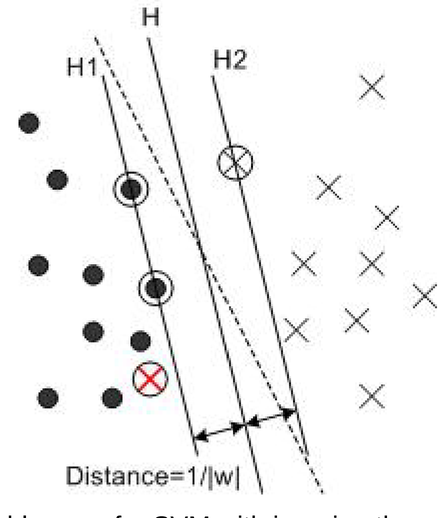


Figure 5.1: Linearly separable case for SVM with ignoring the red-cross point. Linearly non-separable case by considering the red-cross point. The points with circle around are support vectors. The misclassified sample, red cross point is a support vector in non-separable case.

k-Nearest Neighbors (k-NN) algorithm, is a simple nonparametric technique of pattern recognition. For density estimation, its principle is very easy. The key issue is just to calculate the radius of the hyper sphere, which include a certain number of nearest neighbors inside it [Web02]. Then it can define the density of the point (the center of the hyper sphere) and realize classification by classifying the current point to the class that has most points in this sphere. In spite that k-NN can finish a general classification task [Rit75], but the exhaustive k-NN search requires intensive dissimilarity computations, particularly for a large training set. So people often seek help for another classification model, SVM.

To describe SVM [Var95][Col95] up one sentence, given a set of feature vectors which belong to either of two classes, a SVM finds the hyperplane leaving the largest possible fraction of points of the same class on the same side, while maximizing the distance of either class from the hyperplane. This hyperplane is called Optimal Separating Hyperplane (OSH) (see Fig. 5.1), which minimizes the risk of misclassifying not only the examples in the training set but also the examples of the test set. There are several cases in SVM, namely the linearly separable case, the linearly non-separable case, and the non-linearly case.

- In the linearly separable case (see Fig. 5.1), we assume that there is a set of training samples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_l$, $\mathbf{x}_i \in \Re^d$, and the target values are $y_1, y_2, \ldots, y_l$, $y_i \in \{1, -1\}$, which indicate the class membership of $\mathbf{x}_i$. Then if hyperplane $\mathbf{wx} + b$ can separate these sample points, then we can express it as:

$$y_i(\mathbf{wx}_i + b) \geq 1,\ \forall i \in \{1, \ldots, l\} \tag{5.1}$$

According to statistical theory, OSH should not only be a separable hyperplane, but also maximize the margin. So the problem is to minimize $|\mathbf{w}|$ in $d(\mathbf{w}, b) = \frac{2}{|\mathbf{w}|}$ with in subjected to the constrain of Eq. (5.1). Based on Lagrange multipliers method, the problem can be transformed to minimize: $\sum_i \alpha_i - \frac{1}{2}\sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$, with in subjected to the constrain, $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$, $\forall i \in \{1, \ldots, l\}$. Every Lagrange multiplier is associated with one sample in the training set. And those samples whose $\alpha_i > 0$, are called support vectors. The final classification function we obtained is:

$$f(\mathbf{x}_*) = \mathrm{sgn}\Big(\sum_i^N \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x}_* + b\Big) = \mathrm{sgn}(\mathbf{wx}_* + b), \tag{5.2}$$

where N is the number of support vectors.

- For the linearly non-separable case, we cannot use the optimization method and SVM introduced an error term and the problem turns to minimize: $|\mathbf{w}|^2 + C(\sum_i \xi_i)$. In this case, support vectors are the points on the hyperplane, and the samples were classified incorrectly (see Fig. 5.1).

- Another case in SVM is that the problem is non-linearly case. And to deal with this, SVM utilizes the kernel function to solve the transform from non-linear to linear. By introducing a kernel function, the new classification function changes to:

$$f(\mathbf{x}_*) = \mathrm{sgn}(\sum_{i=1}^{N} \alpha_i y_i \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_*) + b) = \mathrm{sgn}(\sum_{i=1}^{N} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_*) + b) \tag{5.3}$$

In the SVM, the kernel function cannot be used arbitrarily. The kernel function must be a function satisfying Mercer's condition, such functions as, polynomial function $K(\mathbf{x},\mathbf{y}) = (\mathbf{xy}+1)^p$, Gaussian function $K(\mathbf{x},\mathbf{y}) = e^{-|\mathbf{x}-\mathbf{y}|^2/2\sigma^2}$, and etc.

[Pon98] shows that SVM is a very good model for 3D object recognition. After test, SVM get an extremely high recognition rate and another important result is that SVM can even recognize the object with some noise or distortion. In addition, a lot of papers prove that SVM is suitable for other classification cases such as face recognition [Hei03] and tracking problem [Avi01] [Wil03]. So if there were no better models, we would choose SVM as our corner classification model.

5.2 Relevance vector machine and sparse Bayesian learning

A big disadvantage of SVM is that it is a hard classification method, which cannot supply a probability output for the classification results. For some cases, maybe it is not important, but for some projects the probability value for a classified target will be very helpful information. So there are some papers that add some calculation steps into SVM to get a score for the classification, but they are not based on Bayesian theory. To solve this problem, Tipping give us an alternative model to SVM, which is called RVM, Relevance Vector Machine [Tip01].

RVM is a new classification model based on Bayesian framework. The model utilities a form that is identical to SVM. The differences are that by exploiting a probabilistic Bayesian learning framework, the author derives accurate prediction models which typically utilize dramatically fewer basis functions than a comparable SVM while offering a number of additional advantages, which include the benefits of probabilistic predictions, automatic estimation of parameters, and the facility to utilize arbitrary basis functions, which are not necessary to be 'Mercer' kernels.

5.2.1 RVM regression model

So let's introduce RVM from the regression model, upon which the classification model is based.

Similar to SVM, we are given a set of examples of input vectors $\{\mathbf{x}_\mathrm{n}\}_{\mathrm{n}=1}^{N}$ along with corresponding targets $\{t_n\}_{n=1}^{N}$, the latter of which might be real values (in regression) or class labels (classification). From this 'training' set we wish to learn a model of the dependency of the targets on the inputs with the objective of making accurate predictions of *t* for previously unseen values of x.

Typically, we base our predictions upon some function y(x) defined over the input space, and 'learning' is the process of inferring this function. The popular y(x) is that of the form:

$$y(\mathbf{x};\mathbf{w}) = \sum_{i=1}^{M} w_i \phi_i(\mathbf{x}) = \mathbf{w}^{\mathrm{T}} \boldsymbol{\varphi}(\mathbf{x}), \tag{5.4}$$

where the output is a linearly weighted sum of M, generally non-linear and fixed, basis functions $\phi(\mathbf{x}) = (\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), ..., \phi_M(\mathbf{x}))^{\mathrm{T}}$. The objective is to estimate good values for parameters $\mathbf{w} = (w_1, w_2, ..., w_M)^{\mathrm{T}}$.

In regression case, the author follows the standard probabilistic formulation and assumes that the targets are samples from the model with additive noise:

$$t_n = y(\mathbf{x}_n; \mathbf{w}) + \varepsilon_n, \tag{5.5}$$

where ε_n are independent samples from some noise process, mean-zero Gaussian with variance σ^2. Thus $p(t_n \mid \mathbf{x}) = N(t_n \mid y(\mathbf{x}_n), \sigma^2)$ and the function y is Eq. (5.4). Tipping also defined his basis function as: $\phi_i(\mathbf{x}) \equiv K(\mathbf{x}, \mathbf{x}_i)$, then he got the likelihood of the complete data set as:

$$p(\mathbf{t} \mid \mathbf{w}, \sigma^2) = (2\pi\sigma^2)^{-N/2} \exp\left\{ -\frac{1}{2\sigma^2} \|\mathbf{t} - \mathbf{\Phi w}\|^2 \right\}, \tag{5.6}$$

where $\mathbf{t} = (t_1 \cdots t_N)^T, \mathbf{w} = (w_0 \cdots w_N)^T$ and $\mathbf{\Phi}$ is the $N \times (N+1)$ matrix with $\mathbf{\Phi}$ = $[\phi(\mathbf{x}_1), \phi(\mathbf{x}_2), \ldots, \phi(\mathbf{x}_N)]^T$, wherein $\phi(\mathbf{x}_n) = [1, K(\mathbf{x}_n, \mathbf{x}_1), K(\mathbf{x}_n, \mathbf{x}_2), \ldots, K(\mathbf{x}_n, \mathbf{x}_N)]^T$.

The next step is to do the maximum-likelihood estimation for **w**, which is to maximize Eq. (5.6). This process is identical to the least-squares solution. We can get this by noting that minimizing squared-error is equivalent to minimizing the negative logarithm of the likelihood:

$$-\log p(\mathbf{t} \mid \mathbf{w}, \sigma^2) = \frac{N}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \sum_{n=1}^{N} \{t_n - y(x_n; \mathbf{w})\}^2 \tag{5.7}$$

The second term in Eq. (5.7) explain everything. But if we only do this, we would expect over-fitting by maximum-likelihood estimation of **w** and σ^2. In least-square solution, people always use a weight penalty term to avoid this, but in RVM they use another method to finish this thing, which is RVM key feature. Tipping adopts a Bayesian perspective, and constrains the parameters by defining an explicit prior probability distribution over them:

$$p(\mathbf{w} \mid \boldsymbol{\alpha}) = \prod_{i=0}^{N} N(w_i \mid 0, \alpha^{-1}), \tag{5.8}$$

with α a vector of N+1 hyperparameters associated independently with every weight. To complete the specification of this hierarchical prior, he also defined the prior distribution for σ^2 and $\boldsymbol{\alpha}$, which are two Gamma distributions:

$$p(\boldsymbol{\alpha}) = \prod_{i=0}^{N} \mathrm{Gamma}(\alpha_i \mid a, b), \text{ and } p(\sigma^{-2}) = \mathrm{Gamma}(\beta \mid c, d), \tag{5.9}$$

with $\beta \equiv \sigma^2$ and where

$$\mathrm{Gamma}(\alpha \mid a,b) = \Gamma(a)^{-1} b^a \alpha^{a-1} e^{-b\alpha} \tag{5.10}$$

with $\Gamma(a) = \int_0^\infty t^{a-1} e^{-t} dt$, “gamma function”.

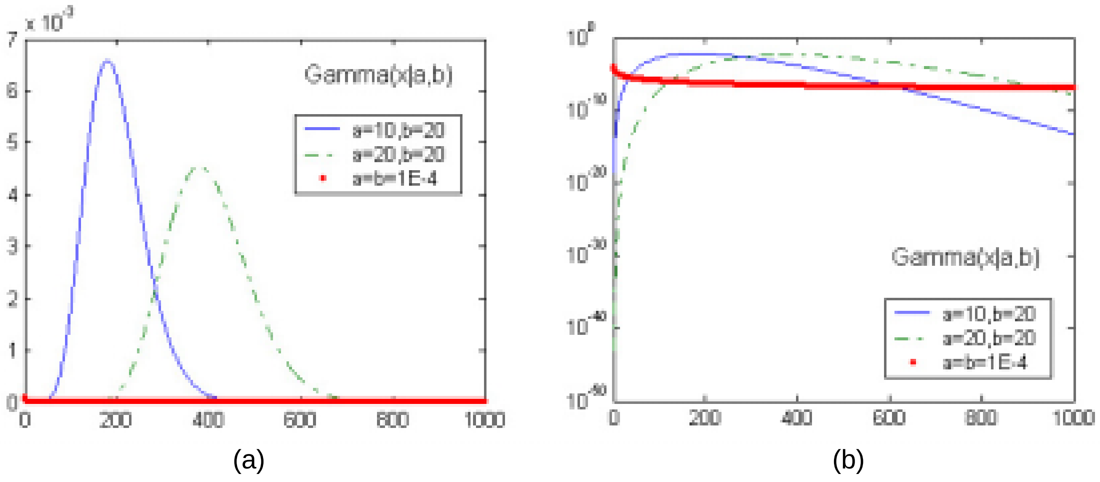


Figure 5.2: Gamma distribution. (a) Gamma distributions with 3 different parameter pairs. (b) The same distribution over logarithmic scale.

Parameters a, b and c, d control the Gamma distribution, in order to let it generate an uniform hyperpriors, a, b, c, d are set to a very small value, so that they got flat non-informative priors (see Fig. 5.2), which means that the “scales” $\boldsymbol{\alpha}$ and σ^2 are equally likely thus predictions are independent of linear scaling of both t and basis function outputs, i.e. results do not depend on the unit of measurement of the targets.

This formulation of prior distributions is a type of automatic relevance determination (ARD) prior [Mac94]. Using such priors in a neural network, individual hyperparameters would typically control groups of weights. The final tests results prove that using a broad prior over the hyperparameters allows the posterior probability mass to concentrate at very large values of some of these α variables, with the consequence that the posterior probability of the associated weights are concentrated at zero. Although introducing more parameters to the model look like make more complex to it, from a Bayesian perspective, the author correctly integrate out all of those parameters or approximate accurately, so that this theory is not a problem from a methodological perspective. So now we should want to know, how does RVM integrate out those noisy parameters and what is the approximation procedure.

Let’s consider, having learned from the training values **t**, how we make a prediction for data t_* given a new input datum $\mathbf{x}_*$. By using classical framework, least-square method, after getting learned quantity $\mathbf{W}_{PLS}$ we can predict it by $y(\mathbf{x}_*;\mathbf{W}_{PLS})$. By MAP (Maximum a postprior) Bayesian framework, we can figure out quantity $p(\mathbf{w} \mid \mathbf{t},\alpha,\sigma^2)$ and then make prediction by $p(t_* \mid \mathbf{W}_{MAP},\sigma^2)$. But Tipping believe that they are not “true Bayesian”. The true one is to predict by $p(t_* \mid \mathbf{t},\alpha,\sigma^2)$ with learned quantity $p(\mathbf{w} \mid \mathbf{t},\alpha,\sigma^2)$.

In general, for any model, if we wish to predict t_* given some training data **t**, what we really want is: $p(t_* \mid \mathbf{t})$, and the distribution is the following expression:

$$p(t_* \mid \mathbf{t}) = \int p(t_* \mid \mathbf{w}, \boldsymbol{\alpha}, \sigma^2) p(\mathbf{w}, \boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) d\mathbf{w} d\boldsymbol{\alpha} d\sigma^2 \tag{5.11}$$

In Eq. (5.11), $p(\mathbf{w}, \boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t})$ cannot be computed analytically, so we must look for an effective approximation. From Bayes' rule, the posterior over all unknowns given the data is:

$$p(\mathbf{w}, \boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) = \frac{p(t \mid \mathbf{w}, \boldsymbol{\alpha}, \sigma^2) p(\mathbf{w}, \boldsymbol{\alpha}, \sigma^2)}{p(\mathbf{t})} \tag{5.12}$$

But it cannot be computed since we cannot perform the normalizing integral:

$$p(\mathbf{t}) = \int p(t \mid \mathbf{w}, \boldsymbol{\alpha}, \sigma^2) p(\mathbf{w}, \boldsymbol{\alpha}, \sigma^2) d\mathbf{w} d\boldsymbol{\alpha} d\sigma^2 \tag{5.13}$$

Therefore the author deposes the posterior as:

$$p(\mathbf{w}, \boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) = p(\mathbf{w} \mid \mathbf{t}, \boldsymbol{\alpha}, \sigma^2) p(\boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) \tag{5.14}$$

Since $p(\mathbf{t} \mid \boldsymbol{\alpha}, \sigma^2) = \int p(\mathbf{t} \mid \mathbf{w}, \sigma^2) p(\mathbf{w} \mid \boldsymbol{\alpha}) d\mathbf{w}$ is a convolution of Gaussians, the first term in Eq. (5.14) is directly computable by Bayes rule:

$$\begin{aligned} p(\mathbf{w} \mid \mathbf{t}, \boldsymbol{\alpha}, \sigma^2) &= \frac{p(\mathbf{t} \mid \mathbf{w}, \sigma^2) p(\mathbf{w} \mid \boldsymbol{\alpha})}{p(\mathbf{t} \mid \boldsymbol{\alpha}, \sigma^2)} \\ &= (2\pi)^{-(N+1)/2} |\boldsymbol{\Sigma}|^{-1/2} \exp\left\{ -\frac{1}{2} (\mathbf{w} - \boldsymbol{\mu})^{\mathrm{T}} \boldsymbol{\Sigma}^{-1} (\mathbf{w} - \boldsymbol{\mu}) \right\}, \end{aligned} \tag{5.15}$$

where the posterior covariance and mean are respectively:

$$\begin{aligned} &\boldsymbol{\Sigma} = (\sigma^{-2} \boldsymbol{\Phi}^T \boldsymbol{\Phi} + \mathbf{A})^{-1}, \\ &\boldsymbol{\mu} = \sigma^{-2} \boldsymbol{\Sigma} \boldsymbol{\Phi}^{\mathrm{t}} \mathbf{t}, \\ &\text{with } \mathbf{A} = \mathrm{diag}(\alpha_0, \alpha_1, \ldots, \alpha_N) \end{aligned} \tag{5.16}$$

For the second term of Eq. (5.14), the hyperparameter posterior $p(\boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t})$, of which the value have to be approximated. By a delta-function at its mode, we can get its most-probable values $\boldsymbol{\alpha}_{MP}$, σ^2_{MP}. But actually, we don't need the entire mass of the posterior be accurately approximated by the delta-function. For predicative purposes, rather than requiring $p(\boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) \approx \delta(\boldsymbol{\alpha}_{MP}, \sigma^2_{MP})$, we only need

$$\int p(t_* \mid \boldsymbol{\alpha}, \sigma^2) \delta(\boldsymbol{\alpha}_{MP}, \sigma^2_{MP}) d\boldsymbol{\alpha} d\sigma^2 \approx \int p(t_* \mid \boldsymbol{\alpha}, \sigma^2) p(\boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) d\boldsymbol{\alpha} d\sigma^2 \tag{5.17}$$

So the relevance vector learning becomes the search for the hyperparameter posterior mode, $p(\boldsymbol{\alpha}, \sigma^2 \mid \mathbf{t}) \propto p(\mathbf{t} \mid \boldsymbol{\alpha}, \sigma^2) p(\boldsymbol{\alpha}) p(\sigma^2)$ with respect to $\boldsymbol{\alpha}$ and β. For the case of uniform hyperpriors, the author just maximized the term

$$p(\mathbf{t} \mid \boldsymbol{\alpha}, \sigma^2) = \int p(\mathbf{t} \mid \mathbf{w}, \sigma^2) p(\mathbf{w} \mid \boldsymbol{\alpha}) d\mathbf{w}$$
$$= (2\pi)^{-N/2} \mid \sigma^2 \mathbf{I} + \mathbf{\Phi} \mathbf{A}^{-1} \mathbf{\Phi}^{\mathrm{T}} \mid^{-1/2} \exp\left\{ -\frac{1}{2} \mathbf{t}^T (\sigma^2 \mathbf{I} + \mathbf{\Phi} \mathbf{A}^{-1} \mathbf{\Phi}^{\mathrm{T}})^{-1} \mathbf{t} \right\}, \quad (5.18)$$

with respect to σ^2 and $\boldsymbol{\alpha}$.

Because we cannot maximize Eq. (5.18) in closed form, so we have to do some iterative re-estimation. For $\boldsymbol{\alpha}$, differentiation of (5.18), equating to zero and rearranging, we got

$$\alpha_i^{new} = \frac{\gamma_i}{\mu_i^2}, \text{ with } \gamma_i \equiv 1 - \alpha_i \Sigma_{ii}, \quad (5.19)$$

where Σ_{ii} is the i-th diagonal element of the posterior weight covariance from (5.16). And for the noise variance σ^2, differentiation leads to the re-iteration:

$$(\sigma^2)^{new} = \frac{\|\mathbf{t} - \mathbf{\Phi}\boldsymbol{\mu}\|^2}{N - \Sigma_i \gamma_i}, \quad (5.20)$$

where N is the number of data examples.

Then for the algorithm, it will update them repeatedly together with $\mathbf{\Sigma}$ and $\boldsymbol{\mu}$ from (5.16), until a convergence criterion has been satisfied. During this process, lots of elements in $\boldsymbol{\alpha}$ tend to infinity, which implies that $p(w_i \mid \mathbf{t}, \boldsymbol{\alpha}, \sigma^2)$ becomes highly peaked at zero, so that the corresponding basis functions are pruned and sparsity is realized. Those vectors $\mathbf{x}_i$ for which w_i is not zero are called relevance vectors.

Finally, the predications become simple. Having found the maximizing values $\boldsymbol{\alpha}_{MP}$, σ_{MP}^2, we can now compute predications by:

$$p(t_* \mid \mathbf{t}, \boldsymbol{\alpha}_{MP}, \sigma_{MP}^2) = \int p(t_* \mid \mathbf{w}, \sigma_{MP}^2) p(\mathbf{w} \mid \mathbf{t}, \boldsymbol{\alpha}_{MP}, \sigma_{MP}^2) d\mathbf{w} \quad (5.21)$$

Because both terms in the integrand are Gaussian, thus the result can be readily computed to be:

$$p(t_* \mid \mathbf{t}, \boldsymbol{\alpha}_{MP}, \sigma_{MP}^2) = N(t_* \mid y_*, \sigma_*^2)$$

with

$$y_* = \boldsymbol{\mu}^T \phi(\mathbf{x}_*) \quad (5.22)$$
$$\sigma_*^2 = \sigma_{MP}^2 + \phi(\mathbf{x}_*)^T \mathbf{\Sigma} \phi(\mathbf{x}_*) \quad (5.23)$$

5.2.2 RVM classification model

For the classification case, RVM uses an essentially identical framework as detailed for regression in the previous section. They use a link function to account for the change in the target quantities. But by this they cannot integrate out the weights analytically, so have to introduce another approximation step in the algorithm.

For two class classification case, it is desired to predict the posterior probability of membership of one of the classes given the input **x**. They follow statistical convention and generalize the linear model by applying the logistic sigmoid link function $\sigma(y) = 1/(1+e^{-y})$ (see Fig. 5.3)

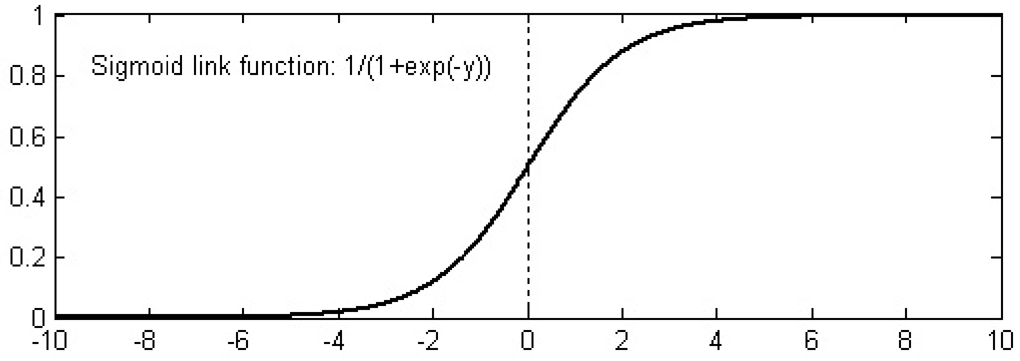


Figure 5.3: Sigmoid link function used in classification model.

to y(x) and adopting the Bernoulli distribution for $p(t \mid \mathbf{x})$, so the likelihood is:

$$P(\mathbf{t} \mid \mathbf{w}) = \prod_{n=1}^{N} \sigma\{y(\mathbf{x}_n;\mathbf{w})\}^{t_n}[1-\sigma\{y(\mathbf{x}_n;\mathbf{w})\}]^{1-t_n} \text{ , } t_n \in \{0,1\} \tag{5.24}$$

and in this model, we don't need noise, or we may assume that the noise is already included in the link function, which can be seen as a probability alternative.

But with such a model, we cannot, like regression case, integrate out the weights analytically, so are denied closed-form expressions for either the weight posterior $p(\mathbf{w} \mid \mathbf{t},\boldsymbol{\alpha})$ or the marginal likelihood $p(\mathbf{t} \mid \boldsymbol{\alpha})$. Therefore, we used an approximation procedure below used by [Mac92], which is based on Laplace's method.

For the current fixed values of α , we need to find out the $\mathbf{w}_{MP}$, giving the location of the mode of the posterior distribution. Since $p(\mathbf{w} \mid \mathbf{t},\boldsymbol{\alpha}) \propto P(\mathbf{t} \mid \mathbf{w})p(\mathbf{w} \mid \boldsymbol{\alpha})$, we can find maximum over **w**, of:

$$\log\{P(\mathbf{t} \mid \mathbf{w})p(\mathbf{w} \mid \boldsymbol{\alpha})\} = \sum_{n=1}^{N}[t_n \log y_n + (1-t_n)\log(1-y_n)] - \frac{1}{2}\mathbf{w}^t\mathbf{A}\mathbf{w} \tag{5.25}$$

with $\sigma\{y(\mathbf{x}_n;\mathbf{w})\}$. This procedure is standard and we may use 2nd-order Newton method to do that. But in [Tip01] they adapted another algorithm, "iteratively-reweighted least-squares" [Nab99]. The quantity Eq. (5.25) is differentiated twice to give the hessian matrix:

$$\nabla_{\mathbf{w}}\nabla_{\mathbf{w}} \log p(\mathbf{w} \mid \mathbf{t},\boldsymbol{\alpha})|_{\mathbf{w}_{MP}} = -(\boldsymbol{\Phi}^{\mathrm{T}}\mathbf{B}\boldsymbol{\Phi} + \mathbf{A}), \tag{5.26}$$

where $\mathbf{B} = diag(\beta_1,\beta_2,\ldots,\beta_N)$ is a diagonal matrix with $\beta_n = \sigma\{y(\mathbf{x}_n)\}[1-\sigma\{y(\mathbf{x}_n)\}]$.

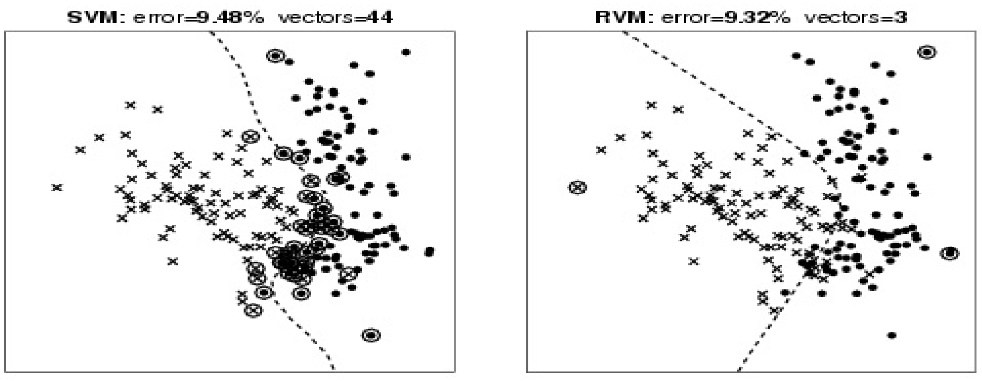
After negating and inversing this Hessian matrix, we get the covariance $\boldsymbol{\Sigma}$ for a Gaussian approximation to the posterior over weights centered at $\mathbf{w}_{MP}$.

$$\mathbf{\Sigma} = (\mathbf{\Phi}^{\mathrm{T}}\mathbf{B}\mathbf{\Phi} + \mathbf{A})^{-1} \tag{5.27}$$

and using the fact that $\nabla_{\mathbf{w}} \log p(\mathbf{w} \mid \mathbf{t}, \boldsymbol{\alpha})\,|_{\mathbf{w}_{MP}} = 0$, we can get:

$$\mathbf{w}_{\mathrm{MP}} = \mathbf{\Sigma}\mathbf{\Phi}^{\mathrm{T}}\mathbf{B}\mathbf{t} \tag{5.28}$$

Then using these two statistics $\mathbf{\Sigma}$ and $\mathbf{w}_{\mathrm{MP}}$, the hyperparameters $\boldsymbol{\alpha}$ are updated using the same expression Eq. (5.19) with regression case. Repeat this procedure until $\boldsymbol{\alpha}$ values converge then $\mathbf{w}_{\mathrm{MP}}$ is what we expected and many elements are zero, so that the sparsity is accomplished (see Fig. 5.4).



(a) (b)

Figure 5.4: Comparison of tests result of SVM (a) and RVM (b). From the figures we can see very clearly that with the same effective decision boundary, RVM has much less vectors to describe it. It is much sparser than SVM.

5.2.3 Multi-class classification

For the current theory, RVM can classify the case with two classes, here we discuss the issue how it can classify more than two classes, which is important for our screen corner detection since we have more classes (different types of screen corners) need to be classified (left-top, right-top, left-bottom and right-bottom screen corner).

Generally, there are two strategies to handle multi-class task with two classes classification model. The first one is "pair-wise". You need to construct a list of RVM models, each of RVM models classify between a pair of classes, so for K classes, we need $K(K-1)/2$ RVM models. The other strategy is "one-versus-others". That means you need K RVM models, and each of them is being able to discriminate a particular class with other classes.

The first one looks more accurate than the second one, but it needs more RVM models. The second one needs less RVM models but seems to be less accurate. But tests in [Ard00] did not show the advantage by using "pair-wise". It still cause a lot of misclassification cases in their experiments. So in our final system, we choose "one-versus-others" for our case. Due to we have four screen corners types need to be classified and one non-corner class, by using "one-versus-others", we construct 5 RVM models in a list, they are "left-top to others", "right-top to others", "left-bottom to others", "right-bottom to others" and "non-corner to others", in every classification, we calculate the probability outputs in these five RVMs and choose the class with the biggest probability value as the final classification result. After testing, this strategy is proved to be a good method and give a real-time speed.

5.2.4 Optimization of parameters in a kernel function

After description in sections 5.2.2 and 5.2.3, we already learned the regression model and classification model in RVM. But now we still left one question unanswered. That is how to choose a kernel function for the current dataset, and how to optimize the parameters within the function. There are many kinds of kernels we can utilize, below are the two common kernels:

Gauss Kernel function:

$$K(\mathbf{x}_m,\mathbf{x}_n)=\exp(-r^{-2}\|\mathbf{x}_m-\mathbf{x}_n\|^2) \tag{5.29}$$

Laplace Kernel function:

$$K(\mathbf{x}_m,\mathbf{x}_n)=\exp(-\sqrt{r^{-2}\|\mathbf{x}_m-\mathbf{x}_n\|^2}) \tag{5.30}$$

In the paper [Tip01], they mentioned that very useful way to choose kernel and parameters is to use k-fold cross-validation method, and they also use it to choose their kernels in testing. But by RVM theory, there is another way to choose the value of parameters and even can extend and optimize more parameters associated with individual input variables. The problem comes from a two-dimensional function: $y(x_1,x_2)=\sin(x_1)/x_1+0.1x_2$, which is based on 100 examples with additive Gaussian noise of standard deviation 0.1. There are two problems using both SVM and RVM to this data:

- The function is linear in x_2, but this will be modeled rather unsatisfactorily by a superposition of nonlinear functions.
- The nonlinear element, $\sin(x_1)/x_1$, is a function of x_1 alone, and so x_2 will simply add irrelevant 'noise' to the input and output of the basis functions and this will be reflected in the overall approximator.

With these two features, both SVM and RVM are difficult to learn accurately. So to overcome this problem, they introduce input variables as two extra 'functions' This is achieved by simply appending two extra columns to the design matrix $\mathbf{\Phi}$ containing the x_1 and x_2 values, and adding corresponding weights and hyperparameters that are updated same to others. Another place need to modify is the kernel function to use new parameters η_1 and η_2. The kernel function now changes to be:

$$K(\mathbf{x}_m,\mathbf{x}_n)=\exp\{-\eta_1(x_{m1}-x_{n1})^2-\eta_2(x_{m2}-x_{n2})^2\} \tag{5.31}$$

Keep the learning procedure identical to before, with the new kernel functions, finally RVM will learned exact values of η_1, η_2 and predict much more accurate.

So by using this method, we can get a set of optimized parameters within a kind of kernel function, which let users avoid using cross-validation methods to do the same thing. But to choose the kernel function, it seems they did not give any suggestions beside k-fold cross-validation method, which may be a potential research topic for others. In reality, considering the limited time, we did not implement RVM method to optimize parameters. For our corner classification case, we use 2-fold cross-validation to choose the kernel function, but it seems the 2-fold cross-validation for a small scale dataset is not accurate enough, so beside it we use an error analysis utility designed by us to evaluate the performance of RVM, which we will explain in the later sections in this chapter.

5.3 Features extraction of corners for RVM

With the theory RVM being exploited, we want to start using it to deal with a real problem, screen corners classification. But before the classification, there is still one thing we need to do. That is we need to confirm our input data **x**, which is the information we want to give the RVM for training and classification.

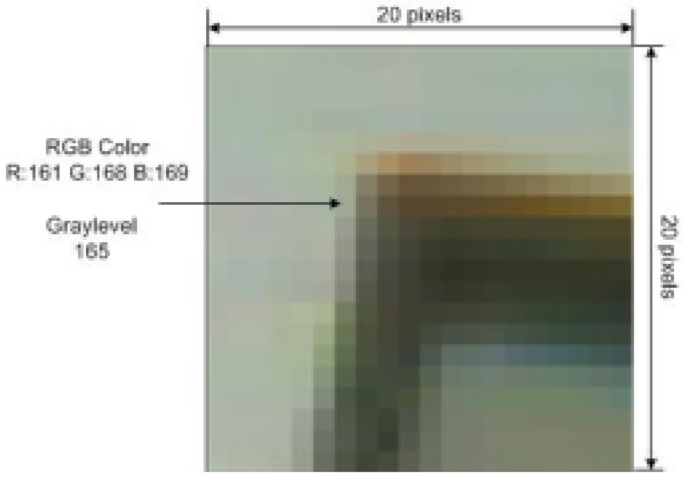


Figure 5.5: A real screen corners sample. The sample window size is 20 by 20 pixels. The RGB color values and gray level value of a pixel is indicated right side of the figure.

The basic information of a sample, a corner in our case, is the pixel values of a sample picture (see Fig. 5.5). But if we directly use pixels values of a sample as input data **x**, then a problem will come with varying dimension. Because in reality we do not know how big a corner is, so we have to let RVM classify the targets with different size. Then that means we have to construct as many models as the number of different size corners we need to know, which increase the complexity of the algorithm. Another problem is that the larger size corner will result in a larger dimension **x** we will apply RVM. For example, if we want to use a 20 by 20 image as a sample window, then the dimension of **x** is 400, which is too big and results in intensive computation.

Therefore, we need a feature extractor that can extract a vector from a sample, the dimension of which need to be small and the values of which should contain enough features to express the sample and let RVM do the classification accurately. In [Ard00], the authors use KLT (Karhunen-Loeve Transform) as feature extraction method and got a very good result. But the problem of KLT is data dependent, since its kernel is not separable. Therefore the derivation of the respective basis for each image sub-block requires unreasonable computational resources so the overall complexity of KLT is significantly high. Thus in our system, we use another transformation model, which has the advantages of KLT with higher computation efficiency: Discrete Cosine Transform.

5.3.1 DCT coefficients as samples features

The discrete cosine transform (DCT) is a technique used in the transformation block of an image compression procedure. It is similar to the discrete Fourier transform, which transforms a signal or image from the spatial domain to the frequency domain.

Actually, what it does is to separate the image into parts of differing importance so that the image data is decorrelated and inter-pixels redundancy is reduced. After decorrelation, each transform coefficient can be encoded independently without losing compression efficiency. That means those coefficients can be looked as the features of an image, which is what we need in screen corner classification.

Now let us see how we can get those DCT coefficients. There are two kind of DCT, one is for 1-dimension and the other is for 2-dimension. In image feature extraction, we need to use 2D DCT, but to understand its principles we start from 1D DCT.

The most common DCT definition of a 1-D sequence of length N is:

$$C(u) = \alpha(u)\sum_{x=0}^{N-1} f(x)\cos\left[\frac{\pi(2x+1)u}{2N}\right] \tag{5.32}$$

for u=0,1,2, …, N-1. Similarly, the inverse transformation is defined as:

$$f(x) = \sum_{x=0}^{N-1}\alpha(u)C(u)\cos\left[\frac{\pi(2x+1)u}{2N}\right] \tag{5.33}$$

for x=0,1,2, …, N-1. In both these two equations $\alpha(u)$ is defined as:

$$\alpha(u) = \begin{cases} \sqrt{\frac{1}{N}} & \textit{for} \quad u = 0 \\ \sqrt{\frac{2}{N}} & \textit{for} \quad u \neq 0 \end{cases} \tag{5.34}$$

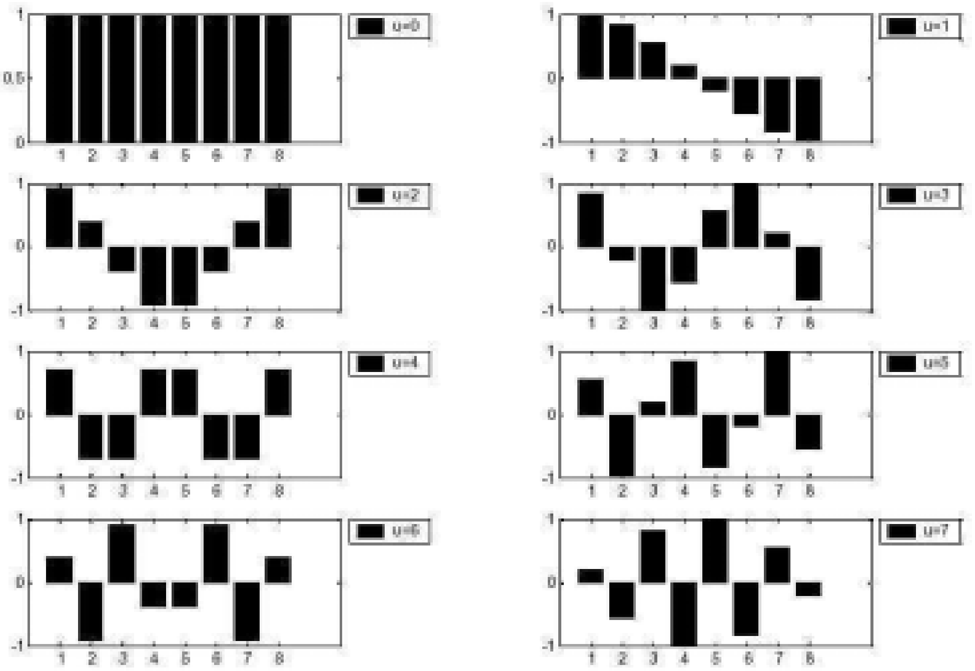


Figure 5.6: One dimensional cosine basis function (N=8).

Considering only coefficient function Eq. (5.32) is useful for our problem, so for the later discussion we will only focus on this function. Ignoring the $f(x)$ and $\alpha(u)$ components in Eq. (5.32), given N=8, the plots of

$$f_{basis} = \cos\left[\frac{\pi(2x+1)u}{2N}\right], \quad x = 0,1,...,N-1 \tag{5.35}$$

with u varying from 0 to 7, namely 8 different frequency models are shown in Fig. 5.6 The first waveform (u=0) renders a constant value, whereas, all other waveforms (u=1, 2, … , 7) give waveforms at progressively increasing frequencies. These waveforms are called the cosine basis function. So the coefficient can be seen as a dot product between a basis function and a vector of input data. If the length of input data is larger than N, then we can divide it into pieces of a fixed size, N.

Another important thing need to mention here is that in each such computation the values of the basis function points will not change. Only the values of input data f(x) will change in each sub-sequence. This property is useful for us, because then we can pre-compute the basis functions offline and then multiply with the following input data, which reduce the computation intensity of the algorithm.

1D DCT is suitable for processing one dimensional signal, like speech waveform. For analysis of 2D signal, like images, we need to seek help for 2D DCT. The definition of 2D DCT is an extension of 1D DCT, which gives the coefficients function as:

$$C(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{M-1} f(x,y)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2M}\right] \tag{5.36}$$

for u = 0, 1, 2, … , N-1 and v = 0, 1, 2, … , M-1. $\alpha(u)$ and $\alpha(v)$ are the same with Eq. (5.34) only need to note is the N now is changed to M in $\alpha(v)$.

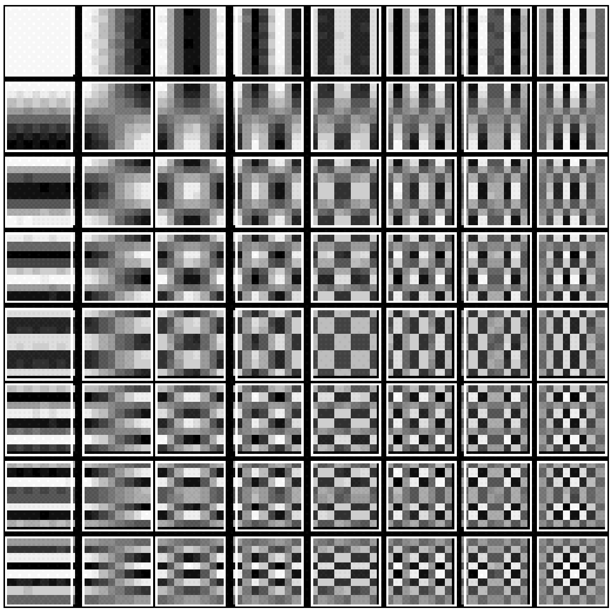


Figure 5.7: 2D DCT basis functions (N=8). White respresents positive amplitudes, black represents negative amplitude and gray represents values in between.

With this definition, similarly we can figure out the basis functions shown in (Fig. 5.7). It can be noted that the basis functions exhibit a progressive increase in frequency both in the vertical and horizontal direction. In UI-Wand system, we use 10 by 10, 20 by 20 or 40 by 40 as sample sizes so in our case the N=M={10, 20, 40}.

Now the coefficients can be obtained by directly use sum of the multiplication of the same entries value of two matrices, one is input image data f(x,y). The other is the basis function value in one frequency model. Actually these coefficients are the weights of a particular DCT basis function.

One thing we need to note here is that the f(x,y) can be a color function or a gray level function. For the former one, the function value will be a triple with R, G, and B values given respectively. For gray level function, the value will be the gray values of the corresponding pixel. In our system, although color information give more hints to the target. It also brings another problem that is unstable under different lighting condition, so we take gray level function, which means we train the RVM with gray sample images and also let it classify the target in gray.

With these coefficients calculated, the image in new space is uncorrelated, and after quantization and entropy encoder, the image can be compressed effectively. But what interests us is another property it supplies. That is the effective bigger coefficients values are concentrated on low frequency basis functions, which means in reality we don’t need remember all of these coefficients, instead we only need some of them and they are enough to express the images without visible information lost.

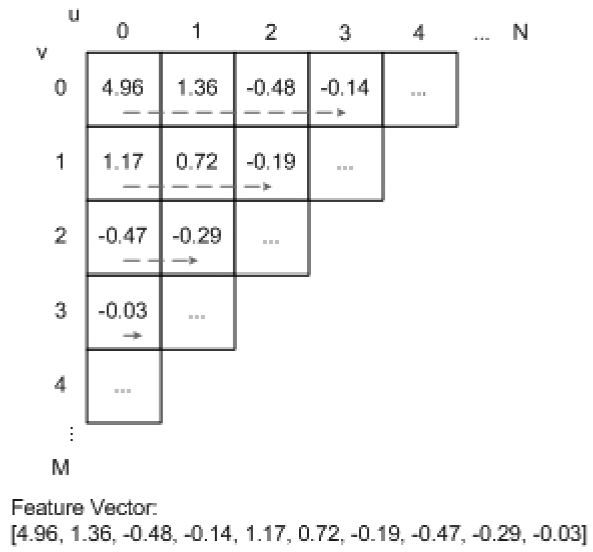


Figure 5.8: A sample of 2D DCT coefficients. With the coefficients calculated, the feature vector is obtained by selecting out the first 10 coefficients.

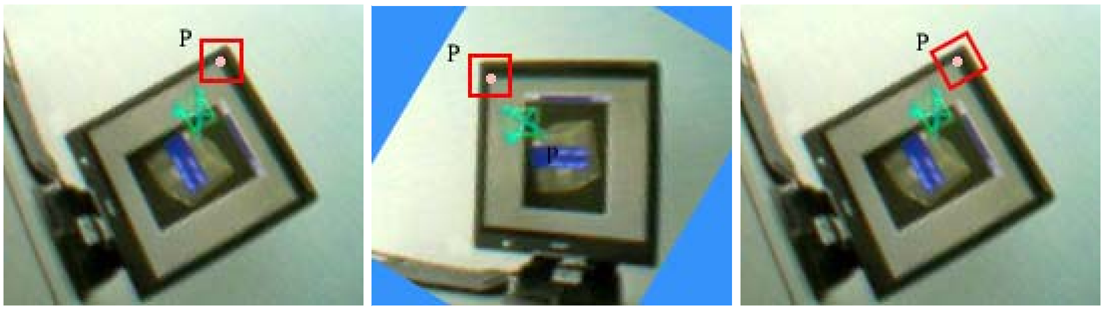
This property is important and useful for our task, feature extraction. Discarding pixel values as feature vectors, we can use DCT onto the images to get a number of coefficients as their feature vectors, and then train RVM by feeding them to it and finally we give a DCT coefficients feature vector of a new image to RVM and let it classify. As the rule of thumb, the first 10 coefficients in the lower frequencies are effective and enough to represent images (see Fig. 5.8), so we extract these 10 values as feature vector in our system.

In the final system, we defined an interface for feature extraction algorithm and implement an algorithm named “ROIWindowedDC”, which can give a sample windows feature vector by utilizing DCT. In that algorithm we can choose how many basis functions and what basis functions we need by setting the parameters, which we will be described in more detailed in Chapter 9.

5.3.2 Rotation corners feature extraction

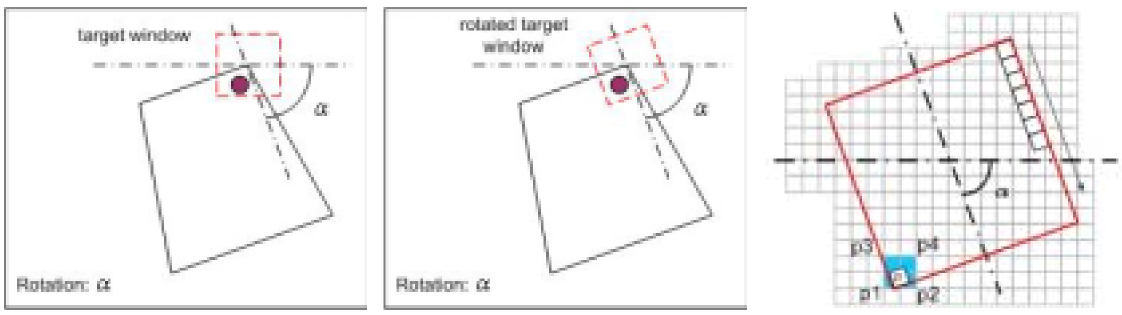
In our UI-Wand system, we want to classify not only if it is screen corner or not, but also we want to find out what’s kind of corner it is. We want to know the position of current screen corner, left-top, right-top, left-bottom, or the right-bottom. Having this information, it is easier for us to finally find out the screen corners by using a geometric method in case when we have too many corners classified as screen corners. We will show that the recognition rate of the classifier for some of sample datasets is not high enough.

However, this kind of classification will cause a problem that is how to justify a screen corner when the image is rotated (see Fig. 5.9a).



(a) (b) (c)

Figure 5.9: Rotated corner target window position. P is the corner point detected by Sojka Detector, the red rectangle is target window need to be classified. (a) Do nothing with rotation. (b) Rotate the image then extract features of target window. (c) Rotate target window and then extract its features.



(a) (b) (c)

Figure 5.10: Rotated corner target window transformation procedure. Red point is just a mark. P is the right-bottom pixel of target window in its coordinate, and p1, p2, p3, p4 are the pixels around P in image coordinate. (a) Rotated screen and with normal target window. (b) Rotated screen and rotated target window. (c) Zoom in rotated target window and pixel values transformation.

It is easy to find a common way to deal with the rotation problem. When a frame of image comes in, and assume we already know if it is a rotated image and the rotation angle in α , then we still can scan line by line in the target window to extract features by just doing a rotation transformation of $-\alpha$ to the whole image. But this method brings two problems, the first one is that after doing that, there will be some regions without information (see the blue area of Fig 5.9b), which will result in useless extra works when we need to use target window scan the whole image to detect screen corners. Another problem is the transformation will cost time thus the processing speed will be getting slower, which is more important for us, because the speed is a significant performance evaluation factor in UI-Wand system.

In Candidate-Winners approach we might avoid the first problem, but the second one is still there. So we have to find another method to solve it. Finally we work out another way to deal with the rotation problem.

We did not rotate the image, and instead we rotated the target window (see Fig. 5.9c) and then scan the target window line by line in its new coordinate (see Fig. 5.10).

The implementation of this algorithm is much simple. Given a rotation angle of α degree, we can construct a rotation transformation matrix:

$$\mathbf{T}_{rotation} = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.37}$$

Then we scan target window in target window coordinates line by line and for each pixel we calculate its new position in image coordinates by multiply its current coordinate with the transformation matrix. After that we use four points around it to interpolate its real value after rotation. This procedure is shown in Fig. 5.10c

The four points interpolation formula we use is:

$$d_P(\alpha,\beta) = d_1(1-\alpha)(1-\beta) + d_2\alpha(1-\beta) + d_3(1-\alpha)\beta + d_4\alpha\beta \tag{5.38}$$

where the parameters are shown in Fig. 5.11.

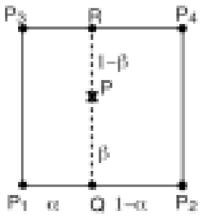


Figure 5.11: Four points interpolation.

So by using this approach to deal with the rotation image, the speed is much faster than the method to rotate the image itself. Therefore not only we can use RVM to classify to whether it is a screen corner or not, now RVM can tell us which type of corner it is as well. This helps a lot for our final system.

But how do we know how many degrees in the image rotated? In our system, there are two ways to get this information frame by frame. One is getting from hardware. There is a hardware component in the real UI-Wand, which can send a rotation signal frame by frame. Keep receiving that signal we can easily update rotation information in our algorithm. Another way is to give a fixed initial status of UI-Wand then tracking it afterwards. For example, we can define a UI-wand initial position from where users must start. After catching the first several frames UI-Wand can detect four screen corners and then by using the pointing position estimation algorithm we can figure out the rotation angle of UI-Wand. Updating this angle frame by frame we can classify the corners always correctly. In the final system, due to problems in rotation sensor, we take the 2nd method.

Hereby, we did not consider other distortion of screen corners in the image, which is caused by the position of UI-Wand in world coordinates. Since these angles will not cause too much problems, in reality, they just cause some distortion in the camera image that can be handled by RVM with more variant samples, which we will discuss in next section.

5.4 Training sample datasets selection and tests for RVM

Having figured out RVM model and feature extraction algorithm, now it is time to study how to let RVM do the real work. The key issue for the time being is to train RVM to a real screen corner classifier.

In the training process, there are two problems need to solve. The first one is how to choose the training data set (samples). The second one is what's kind of kernel should be utilized for this dataset. Both of them are the very important factors that will greatly affect the RVM classification rate.

In UI-Wand system, we used 3 different ways to get different sample datasets for two different type screens that mentioned in Chapter 4, a PHILIPS brilliance 109MP white computer screen and a PHILIPS brilliance 180P2 black LCD computer screen (see Fig. 4.14). Then feeding these dataset to train the RVM with different kernel functions. After comparison, we found out the advantages of these datasets and the suitable kernel functions for them. The following sections will give a description on detailed issues.

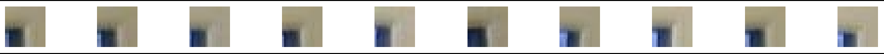
5.4.1 Manually selected sample dataset

The first sample dataset we took is very direct. We use a web camera[2] to capture a sequence of frames from a white screen, and then randomly select out some of them and then manually cut the corners of the screen by image processing tools (see Table 5.1 for dataset specification and Table 5.2 for the samples pictures).

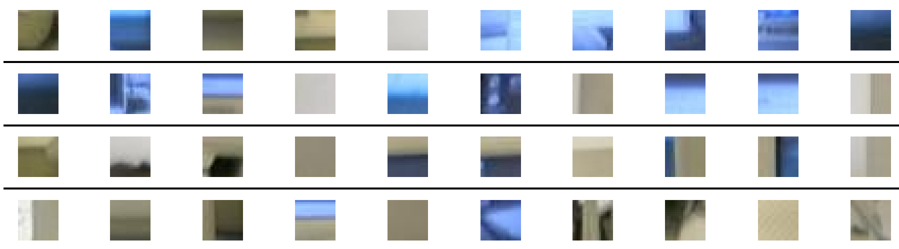
Table 5.1: Manually selected Dataset specification.

Dataset Specification Items	Values and Description
Type	Manually Selected Dataset
Sample Size	20 x 20 pixels
Number of Sample Class	5
Number of Samples	80
Samples Distribution	10/10/10/10/40
Class 0	Left-Top screen corner
Class 1	Right-Top screen corner
Class 2	Left-Bottom screen corner
Class 3	Right-Bottom screen corner
Class 4	Non-Corner

Table 5.2: White screen corners samples selected manually.

Type	Screen Corner Samples
Left-Top	
Right-Top	
Left-Bottom	
Right-Bottom	

[2] PHILIPS Web Camera, 10 frames per second

Non-Corner	

One thing has to be mentioned here is, as we said in above sections, the images that will be operated in RVM are gray level images. Though the input frames contain color, before feature extraction step, we already transfer them to gray.

5.4.1.1 2-fold cross-validation for evaluation

Now we discuss the second problem. By using this dataset, which kernel function should we choose so that the RVM can get a high classification rate after training? In section 5.2.3, we mentioned that k-fold cross-validation is a common and good method for parameters selection. So here, we take this method to choose the function and parameters. In this dataset, there are 80 samples and distribution is 10/10/10/10/40, and considering the dataset is not too big, we only use 2-fold cross-validation, which means we separate the current dataset into two parts with equal size, and the distributions are the same 5/5/5/5/20. Then we use one subset as training dataset to train RVM, and use another one to test it, after which we change the roles of these two subsets then train and test again. After testing, we got the results below (see Table 5.3).

In that table, the "kernels" column presents which kernel we are using, G means "Gaussian", L means "Laplace", the number after the "G" or "L" is the value of the parameter "r" in the kernel functions Eq. (5.35) and Eq. (5.36). LT, RT, LB, RB and NON items in that table, are the abbreviations presenting the five classes in the dataset, left-top, right-top, left-bottom, right-bottom, and non-corner respectively. The last column A.E.R means the average error rates of classification, which is calculated by Eq. (5.37).

$$A.E.R = \left(\sum_{i=0}^{F}\sum_{j=0}^{C} E_{i,j}\right)/N\,, \tag{5.39}$$

where F is the number of sub-dataset (fold number), C is the number of class in this dataset, $E_{i,j}$ is the number of error unit in the *j*th class when using the *i*th sub-dataset as training set. In this case, F=2, C=5 and N=80. The value formatted at "x/y" in each cell of class label columns means $E_{1,j}/E_{2,j}$.

Table 5.3: 2-fold cross-validation table by using manually cut dataset.

KF	LT	RT	LB	RB	NON	A.E.R
G-0.5	1/0	0/0	0/0	0/0	3/7	13.75%
G-1.0	0/0	0/0	0/0	0/0	3/4	8.75%
G-1.5	0/0	0/0	0/0	2/0	3/4	11.25%
G-2.0	0/0	0/0	0/0	5/1	2/4	15.00%
L-0.5	5/0	0/0	0/0	1/0	1/2	11.25%
L-1.0	0/0	0/0	0/0	0/0	1/2	3.75%
L-2.0	0/0	0/0	0/0	0/0	1/3	5.00%
L-3.0	0/0	0/0	0/0	1/0	1/4	7.50%

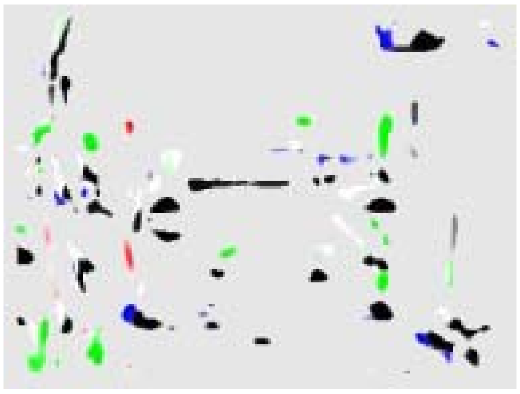
From this table, we note that Laplace kernel function will make the RVM to get a more accurate classification result comparing to Gauss kernel. And the length parameter “r”, 1.0 gives a better performance result. But considering the dataset is not too big and also 2-fold might also cause the inaccurate testing result, so we use a visualization utility, “Scanimg” and an “Error Analysis” utility to check those RVM models performances in the end.

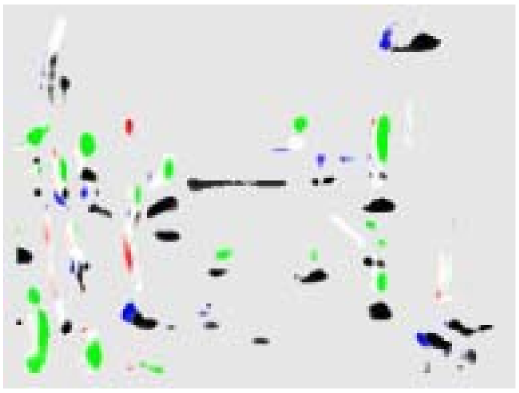
5.4.1.2 Scanimg and Error Analysis utility for evaluation

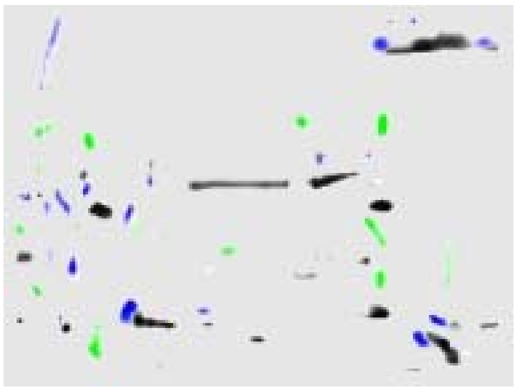
In UI-Wand system, the RVM’s responsibility is to classify screen corner candidates, which means given a list of corners, RVM need to tell us which ones are screen corners. In order to check whether RVM makes a correct decision or not and how accurate it is, firstly, we need to manually mark the screen corners in some images and see them as correct reference points. Then we let the first two steps of Candidates-Winners procedure run on these images, choosing candidates by Sojka corner detection and classifying by RVM, after which we can get RVM classification accuracy by comparing the results with the reference points we marked.

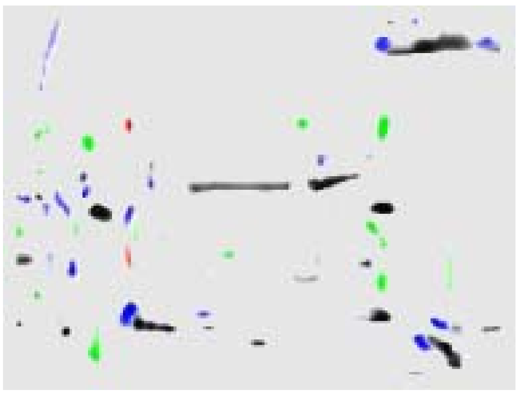
But the method above is dependent on Sojka corner detections, we prefer to do the test only on RVM, thus we follow another way. Instead of classifying the corners selected, we let RVM scan the whole image completely. Fig. 3.3 shows that screen corner detection can be realized by sliding a target window on an image and let RVM classify every sample extracted by the target window. Now we do the same thing here. By specifying the size of the target window and step size in x direction and y direction, we can let target window’s movement cover the whole image, which means we will classify every point in the image. If we give some colors on the classification result, let say, red for left-top corner, green for right-top corner, blue for left-bottom corner, black for right-bottom corner, and gray for non-corner, then the best result after this scanning should be only screen corners in the image has color, the places in the rest of the image should be gray. But actually this was too good to be true. The scanning results by using different kernel functions are shown below (see Fig. 5.12).

(a)


(b)


(c)


(d)


(e)

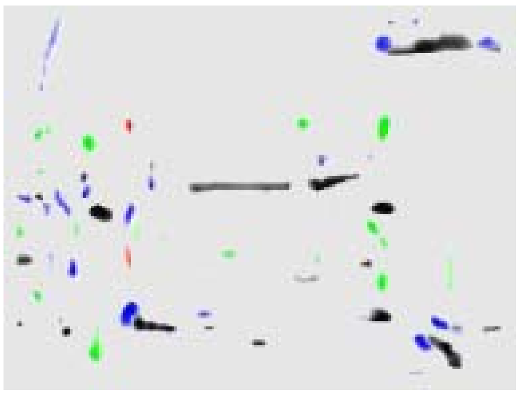

(f)

Figure 5.12: Classification results by using RVM model with different kernel functions. (a) Original image. (b) RVM with Gauss-0.5 (c) RVM with Gauss-1.0 (d) RVM with Laplace-1.0 (e) RVM with Laplace-2.0 (f) RVM with Laplace-3.0. The color Red/Green/Blue/Black/Gray presents respectively, Class0/1/2/3/4.

From these image results generated by Scanimg utility, it is easy for us to note that the result by Gauss kernel is worse than the results by Laplace kernels. And another information we can get is the best model suggested by 2-fold cross-validation, Laplace-1.0 lost one screen corner (left-top). Besides this information, we cannot get anymore interesting stuffs by our eyes. So we have to use our Error Analysis utility to make analysis. Then we got the information below, the items of which are output of Error Analysis utility. The detailed explanations about them can be referred in Chapter 10.

Table 5.4: Error Analysis utility for analysis of RVM models with different kernel functions.

	LT		**RT**		**LB**		**RB**		**NON**		
KF	*MV*	*CGV*	*MV*	*CGV*	*MV*	*CGV*	*MV*	*CGV*	*MUR*	*CD*	*WV*
G-0.5	7.07	1.68	20.22	5.32	7.62	4.23	14.42	7.44	93%	4	21.27
G-1	7.07	2.21	26.31	6.53	7.62	3.94	14.21	6.92	93%	4	21.43
L-1.0	-----	-----	7.28	2.75	10.63	5.38	19.00	4.39	92%	3	-----
L-2.0	4.47	2.12	8.54	3.51	10.00	3.80	13.04	4.85	92%	4	18.39
L-3.0	4.47	2.14	8.54	4.12	10.00	3.85	13.04	4.88	91%	4	18.43

*KF: Kernel function.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2

From this table we can clearly compare the results generated by the RVM models with different kernel functions. It is obvious that we get the following information:

1. Most of kernels are effective. With them RVM can detect four screen corners.

2. Laplace-1.0 is not as good as in the 2-fold cross-validation evaluation.

3. The clusters covering the screen corners are smaller by Laplace kernel RVM.

4. The weighted geometric centers of clusters in Laplace kernel RVM are closer than Gauss kernel RVM.

5. All the kernels got a very high misunderstanding rates, which means there are too many non-corners classified as screen corners

6. Laplace kernels got lower whole variance (WV) finally, which is the same as 2-fold cross-validation.

In order to get more accurate results, we test these kernels on 30 images in a sequence (Appendix A) and got the following Table 5.5.

Table 5.5: Average values of Error Analysis results of scanning 30 images in a sequence (see Appendix A).

RVM		**Error Analysis Utility Outputs**						
KF	*RVN*	*HCV*	*MV*	*GV*	*CGV*	*MUR*	*CD*	*WV*
G-0.5	18.5	70.00	14.13	7.07	4.62	93%	98%	13.08
G-1	27.8	79.88	18.01	9.3	4.72	93%	99%	15.40
L-1.0	21	7.06	14.97	6.73	3.93	94%	74%	6.47
L-2.0	18.4	7.26	11.99	5.62	3.85	93%	98%	5.71
L-3.0	27.6	6.81	12.09	5.68	3.9	93%	98%	5.72

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2

In Table 5.5, we ignore the corner type information and average all attributes values under the four corner types. In addition, we show another item, RVN (the number of relevance vectors) in the final model after training. The value is the average of the relevance vectors in five "one-versus-others" RVM models.

The three extra information from Table 5.4 we can get are:

1. All of these kernels perform very badly on HCV attribute, which means we cannot ensure that the points with highest probability are screen corners.

2. An interesting thing comes from the relationship between GV and CGV. By using any kernels, the geometric center weighted by probability is closer to the reference point than the geometric center. That means probability calculated by RVM indeed is helpful and useful.

3. The amount of relevance vectors in L-2.0 is the smallest, which means L-2.0 is more suitable for this dataset and can make RVM to get an easy convergence.

By using this dataset and Laplace kernel functions, the tests results prove that the RVM can work in the Candidates-Winners approach. But the problem of this dataset is obvious that it need human to select samples, which is very inconvenient for users. Thus we start trying to find out other methods to get training samples automatically.

5.4.2 Synthetic sample dataset

5.4.2.1 Synthetic dataset considering contents

If we look at Table 5.2, we can find that actually the geometric feature is very simple for corners. It is just a combination of a right angle and the content of screen, so we start thinking if it is possible to generate synthetic screen corners automatically? If we can automatically generate such a dataset by which the RVM can make accurate classification, then that will be very convenient for users since they don't need to take corners samples by themselves.

The experimental results show that this is a possible way to go. To the white screen, we generate our synthetic samples according to the following procedure:

1. Choose a similar color to the real screen border for the color of border in synthetic samples.

2. Save the content that the screen will display.

3. Extract four corners of that display then combine them with a border (with different size) generated automatically to form synthetic samples.

Then we generate the synthetic sample dataset shown in Table 5.6 and Table 5.7 below:

Table 5.6: Synthetic dataset for white screen specification.

Dataset Specification Items	**Values and Description**
Type	Synthetic Dataset I
Sample Size	20 x 20 pixels
Number of Sample Class	5
Number of Samples	120
Samples Distribution	20/20/20/20/40
Class 0	Left-Top screen corner
Class 1	Right-Top screen corner
Class 2	Left-Bottom screen corner
Class 3	Right-Bottom screen corner
Class 4	Non-Corner

Table 5.7: Synthetic white screen samples generated automatically.

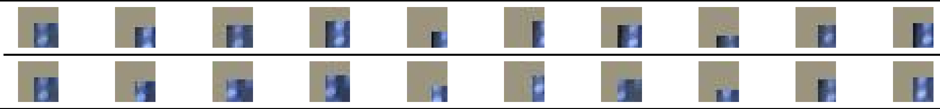
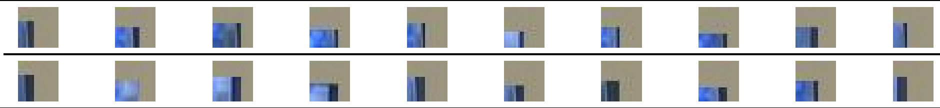
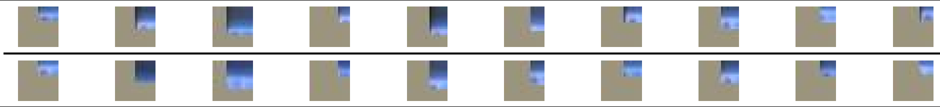
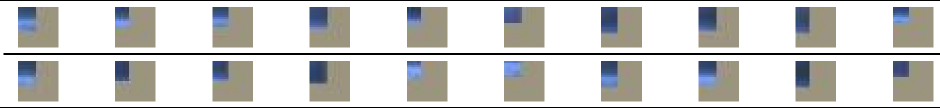
Type	**Screen Corner Samples**
Left-Top	
Right-Top	
Left-Bottom	
Right-Bottom	
Non-Corner	The same as Table. 4.2.4.2

Table 5.8: Error Analysis utility results by using synthetic dataset for white screen in the sequence (see Appendix A).

RVM		**Error Analysis Utility Outputs**						
KF	*RVN*	*HCV*	*MV*	*GV*	*CGV*	*MUR*	**CD**	*WV*
G-0.5	15.6	47.63	28.16	12.57	10.47	94%	99%	17.07
L-2.0	56.2	69.88	24.41	12.00	9.09	93%	99%	18.14

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2

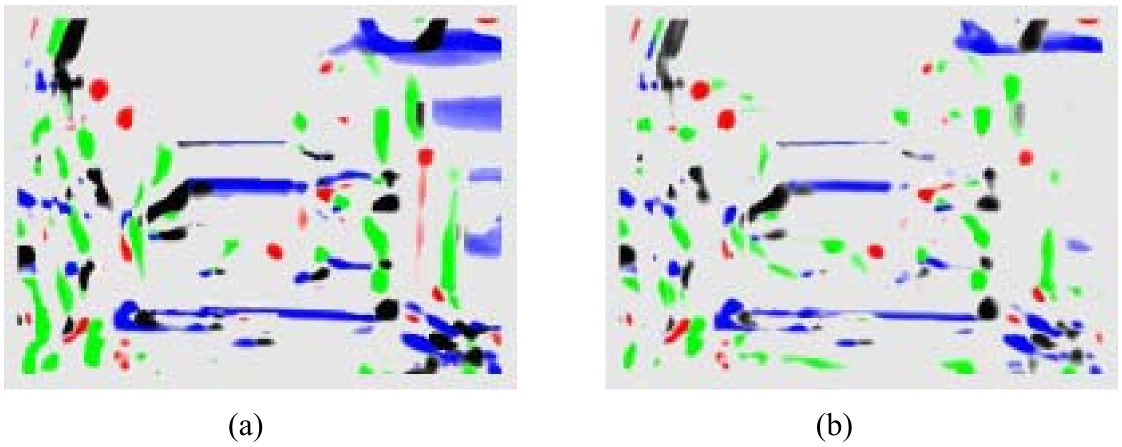


Figure 5.13: Classification results by using white screen synthetic dataset. (a) The result of RVM with G-0.5 kernerl function, (b) The result of RVM with L-2.0 kernel function.

By using this dataset in RVM for training, we got the results from Table 5.8 above. This time we just write down the results of two kernel functions one from Gaussian, and one from Laplace, since other kernel functions will give worse results. From this table we can see that, although the synthetic samples get a little worse but the detection rate still keeps very high with G-0.5 kernel function, which means we can use this dataset in RVM for classification.

5.4.2.2 Synthetic dataset without considering content

There are some problems with the last synthetic dataset. The first one is that we just consider one display content, a windows desktop. So if the content changes, the RVM might give inaccurate classification. The second one is that we did not consider the changes of angles α and β (see Fig. 5.14), caused by rotated images and distortion by UI-wand orientation, which will also result in inaccurate classification.

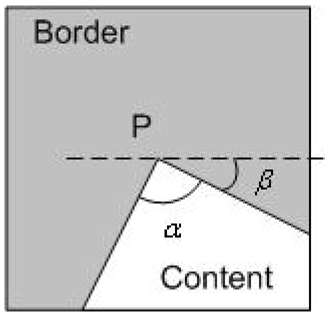


Figure 5.14: Parameters for generating synthetic corners.

In the synthetic dataset for the black LCD screen, we generate more samples that cover all kinds of possible variant factors, so that RVM can give more accurate classification.

The rules we used to generate samples are listed below:

1. Assign a list of values for the color of the border of the screen in a reasonable range

2. Assign a list of positions of P in a reasonable range

3. Assign a list of angles α in a reasonable range.

4. Assign a list of angle β in a reasonable range.

5. Generate the samples with all possible combination of the values in those lists.

The dataset size will affect the speed of training. So in the final system we use the synthetic dataset shown in Table 5.9 and Table 5.10 generated by combination of following values of parameters:

- α values are changed 6 degree every time from 60 to 120. So there are 10 samples with different angles.

- β value keeps 0 degree.

- The color of border and content are randomly pre-selected from 10 real frames and randomly assigned to the 10 samples.

- Non-corners samples are randomly select from some images by program.

Table 5.9: Synthetic dataset for black LCD screen specification.

Dataset Specification Items	**Values and Description**
Type	Synthetic Dataset II
Sample Size	20 x 20 pixels
Number of Sample Class	5
Number of Samples	100
Samples Distribution	10/10/10/10/60
Class 0	Left-Top screen corner
Class 1	Right-Top screen corner
Class 2	Left-Bottom screen corner
Class 3	Right-Bottom screen corner
Class 4	Non-Corner

Table 5.10: Synthetic black LCD screen samples generated automatically.

Corner Types	**Screen Corner Samples**
Left-Top	
Right-Top	
Left-Bottom	
Right-Bottom	

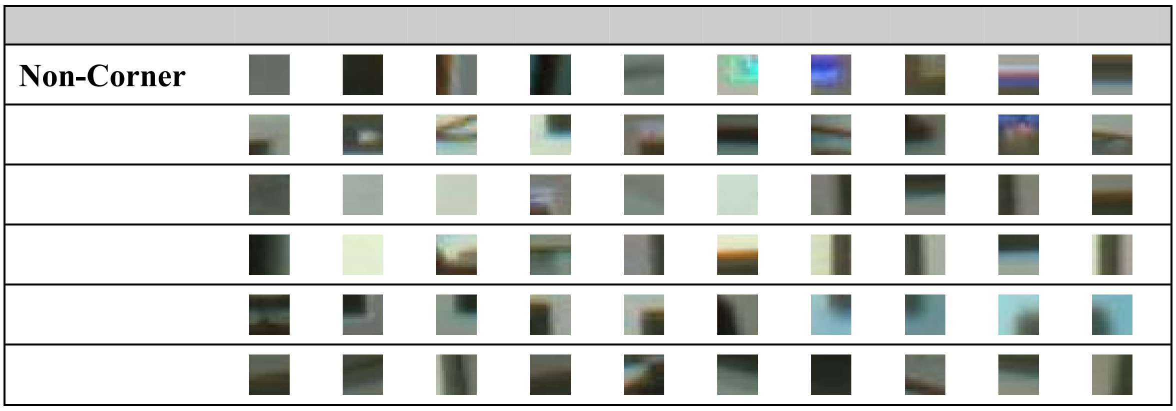


Table 5.11: Error Analysis utility results by using synthetic dataset for black LCD screen on 30 images of a sequence (see Appendix B).

RVM		**Error Analysis Utility Outputs**						
KF	*RVN*	*HCV*	*MV*	*GV*	*CGV*	*MUR*	*CD*	*WV*
G-0.5	4.4	36.94	5.38	3.47	2.57	84%	100%	6.65

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2

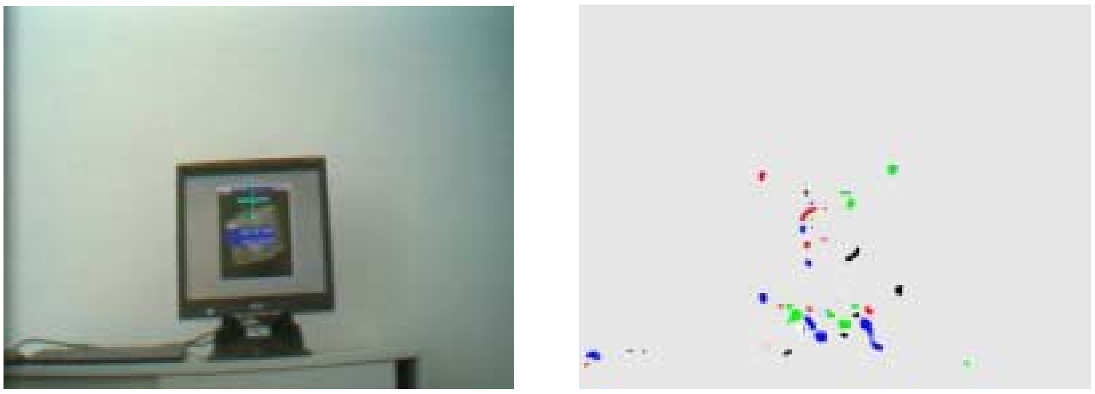


Figure 5.15: The classification results by using black LCD synthetic dataset. (a) Original image (b) Classification results by using RVM with G-0.5 kernel function.

The test results (see Table 5.11 and Fig. 5.15) show that the RVM get a very good classification performance on this synthetic dataset, the detection rate is high to 100 percent in 30 images test and also the whole variance is very low. Another important value we easily noted is RVN in G-0.5. It is very small, 4.4 means that it only needs five samples from 100 to classify, which give a strong support to the advantages of RVM for sparsity. The hyperparameters convergence curves shown in Fig. 5.16:

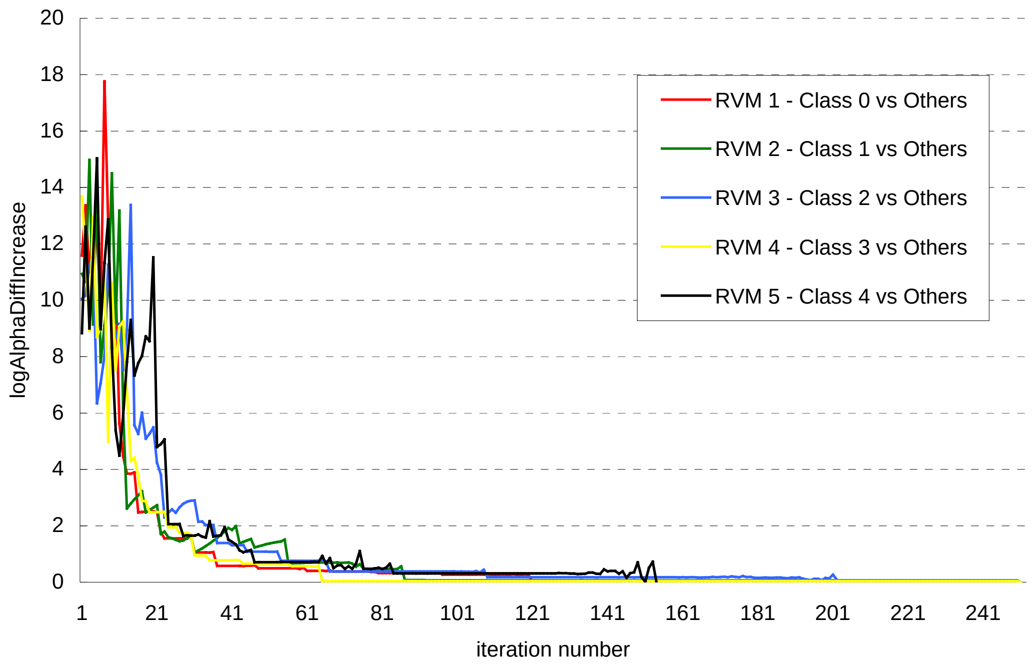


Figure 5.16: The learning process of RVM models for black LCD screen synthetic dataset. Each curve represents one RVM model used to classify four types of screen corner and non-corner. The class0/1/2/3/4 in the figure is left-top/right-top/left-bottom/right-bottom/non-corners in the dataset.

Besides getting ridge of human samples collection work, another advantage of using this synthetic dataset is that we can use a smaller target window to recognize screen corners, which will accelerate the speed of classification very much. Instead of the same size with samples, 20 x 20, we used in reality, 10 x 10 target window to select the target samples to be classified. The reason is obvious. That is because in this dataset we only consider a corner's geometric features, which means we do not consider the situation to recognize if the target is real screen corner. Instead we only go to find out the type of a corner. When a frame with a screen is coming in, we use a very small target window to scan it. It is easy to imagine that within a very small target window the content gray values curve will be flat, so it can be represented by a certain gray value. The same, the border gray curve also can be seen as a flat one. Therefore the samples in the dataset are very suitable for this case. Another problem is that if it can work when the screen is far from the camera. For this problem we have two ways to solve it, one is to shrink the target window continually. The second way is do a Gaussian blur to the image so that the content gray values curve will be smoothed and changed to a flat. Considering the limitation of time, in the final system, we did not realize these ideas, but it did not affect the performance of the system. The final system is running well with this dataset under the fixed target window 10 x 10. It is robust for changes in lighting and in varying shooting positions.

However, as what we said in system design chapter and what you have seen in tests result in this chapter, only with RVM the system cannot directly select out the screen corners. No matter what's kind of kernel functions we use and what's dataset above we choose, the misunderstanding rate keeps very high, which means plenty of non-corner samples are classified to one of screen corners so it cannot tell which four corners are screen corners by giving a list of candidates. Also the highest confidence points are incorrect, which means we cannot choose the four points with highest-class probability as screen corners. The problem is not in kernel functions but in dataset. So we collect another dataset that gives a much better result for us.

5.4.3 RVM improvement by adaptive sample dataset

In our synthetic dataset mentioned in section 5.4.2.2, we know the benefits of using samples without including content of displaying in the screen. But since the samples are faked and the target window suitable for this dataset is small, so the misunderstanding rate is still high and also the non-corner samples cannot be easily found out, because the features are not clear when the sample is small. So in this dataset, we try on the contrary, instead of excluding content of display, we enlarge the samples size and select them from real image so that they can include more features of real screen corners and more important the features of non-corners increase so that the misunderstanding rate get much reduction.

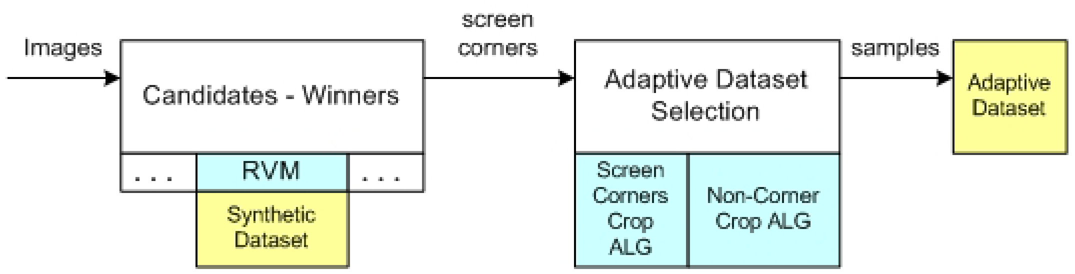


Figure 5.17: The process to generate adaptive dataset automatically. Blue boxes represent model and algorithms utilized. Yellow boxes are dataset.

But how can we get these samples? This is a question that user want to know. Indeed, for this dataset if we have to select manually then we are back to the problems existing in the first dataset. So this time we select it automatically.

It can be done automatically now (see Fig. 5.17) because we already got the one of datasets described before. By using one of above datasets in RVM and running our Candidates-Winners approach, we can be successful in detecting four screen corners in consecutive frames. Then it is easy for us to select screen corners samples with knowledge of their exact positions. To non-corners samples, we just let algorithm to select the samples far from the screen corners, then we got the sample dataset below (see Table 5.12 and Table 5.13).

Table 5.12: Adaptive dataset 1 specification.

Dataset Specification Items	Values and Description
Type	Adaptive Dataset I
Sample Size	40 x 40 pixels
Number of Sample Class	5
Number of Samples	120
Samples Distribution	10/10/10/10/80
Class 0	Left-Top screen corner
Class 1	Right-Top screen corner
Class 2	Left-Bottom screen corner
Class 3	Right-Bottom screen corner
Class 4	Non-Corner

Table 5.13: Samples in adaptive dataset for white screen.

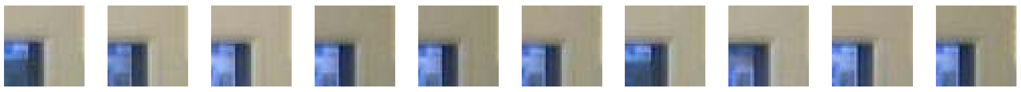
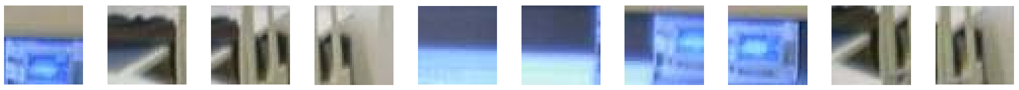
Type	**Screen Corners Samples**
LT	
RT	
LB	
RB	
	
NON	

Table 5.14: Error Analysis utility results by using adaptive dataset for white screen in the sequence (see Appendix A).

RVM		**Error Analysis Utility Outputs**						
KF	*RVN*	*HCV*	*MV*	*GV*	*CGV*	*MUR*	*CD*	*WV*
G-1.5	45.4	97.01	16.14	7.99	6.50	68%	100%	17.11
L-2.0	17.8	17.99	10.48	5.44	4.33	50%	100%	6.76

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with “wh_V” in section 10.2.3.2

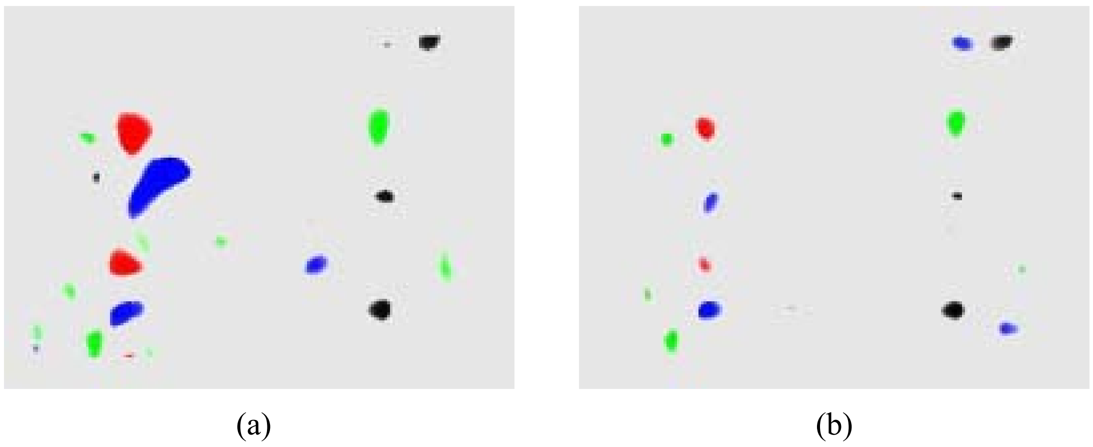


Figure 5.18: The RVM classification result by using Scanimg on the image shown in Fig. 5.11a. (a) RVM with G-1.5 kernel function. (b) RVM with L-2.0 kernel function.

It's good news that the MUR values in Table 5.14 are reduced on both kernel functions. If the percentage is still not direct, Fig. 5.18 will explain the improvements. From (b) figure we can clearly distinguish the four screen corners now. The four clusters form a rectangle with the screen behind.

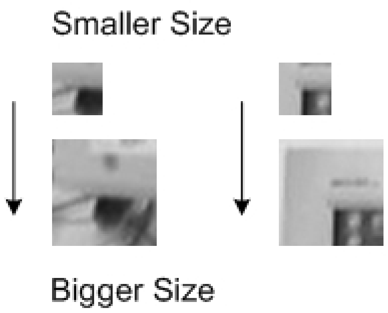


Figure 5.19: Larger size of non-corner sample gives more features than smaller one.

In case of the black LCD we use the same approach to collect the samples, but with some extra operation that helps to raise the rationality of the distribution of those samples. Firstly, we consider the angle changes in corner samples. When we get a screen corner sample, we rotate it with some angles so that we got several samples with different β angles (see Fig. 5.14). Secondly, for the selection of non-corner samples, we separate the non-corner as three classes. The first one is the base class, which includes the border of a real screen that is often classified to screen corners. The second class consists of some samples from the background, which are selected randomly from those places that are far away from screen corners. The third class of non-corner samples are very important, these samples are the samples that were misclassified to one of the screen corners classes by using a small size dataset and target window. We think the small size of these samples is the main reason to cause the bigger MUR value. So in this dataset, we take these non-corner samples with bigger size (see Fig. 5.19), so that they can contain more features. This step is like an adaptive procedure so we named this dataset as adaptive dataset.

By instructing our algorithm that we want to get each screen corner with 2 different β angles, 8 base non-corner samples, 20 background non-corner samples and 40 corrected non-corner samples finally the system selected the following samples as dataset for us (see Table 5.15 and Table 5.16).

Table 5.15: Adaptive dataset 2 for black LCD specification.

Dataset Specification Items	**Values and Description**
Type	Adaptive Dataset II
Sample Size	40 x 40 pixels
Number of Sample Class	5
Number of Samples	108
Samples Distribution	10/10/10/10/68
Class 0	Left-Top screen corner
Class 1	Right-Top screen corner
Class 2	Left-Bottom screen corner
Class 3	Right-Bottom screen corner
Class 4	Non-Corner

Table 5.16: Samples in Adaptive dataset 2 for black LCD.

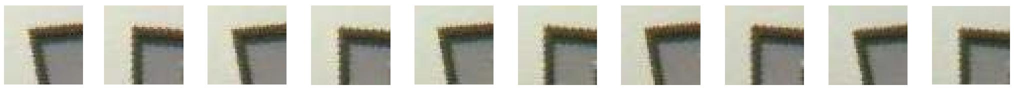
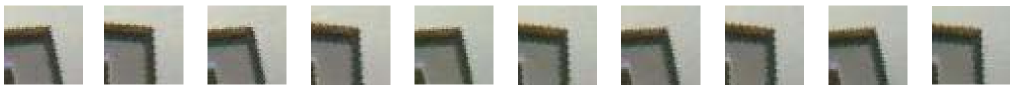
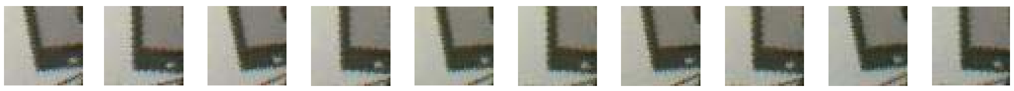
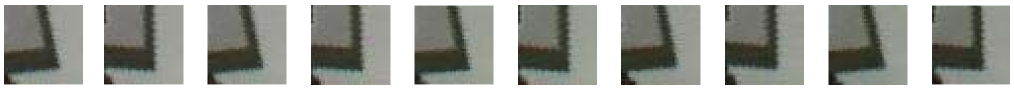
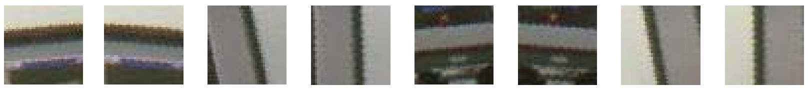
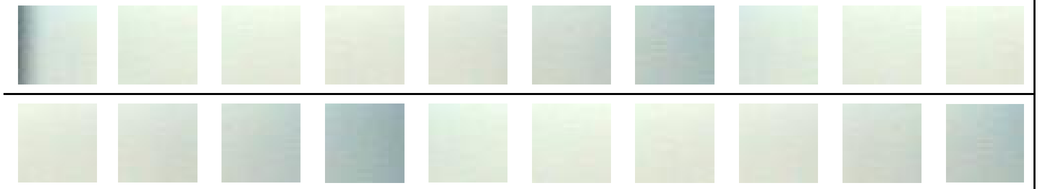
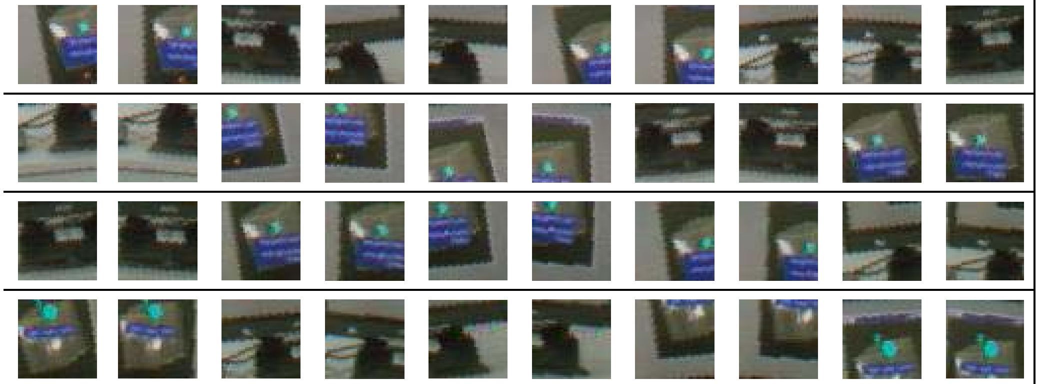
Type	**Screen Corners Samples**
LT	
RT	
LB	
RB	
N-BS	
N-BG	
N-C	

Table 5.17: Error Analysis utility results by using adaptive dataset for black LCD screen on the black LCD screen image sequence

RVM		**Error Analysis Utility Outputs**						
KF	*RVN*	*HCV*	*MV*	*GV*	*CGV*	*MUR*	*CD*	*WV*
L-2.0	15.6	13.01	7.56	4.33	2.84	29%	100%	4.93
L-3.0	12.4	11.67	8.03	4.54	3.16	30%	100%	5.05

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2

Still, after testing the misunderstanding rate is getting very low as shown in Table 5.17 and can be observed directly from Fig. 5.20. It is getting down from 84% (synthetic dataset) to 30% around. And also the HCV values and WV values become smaller.

Though the bigger size of samples brings higher accuracy of classification, at the same time it also brings some troubles. The biggest trouble is feature extraction speed. Since the size becomes 40 x 40 so we need to do more calculations (almost 16 times to use 10 x 10 target window) to get DCT coefficients and also for the rotated images, we need to do much more transformations. Therefore the speed by using this dataset becomes slower. Another drawback is that this dataset is content dependent, so if the contend of displaying is changing, then the recognition rate will decrease definitely. The solution for this problem is to increase the samples with all possible displaying content included, but this will need much more time on training, in the finally system this is not implemented.

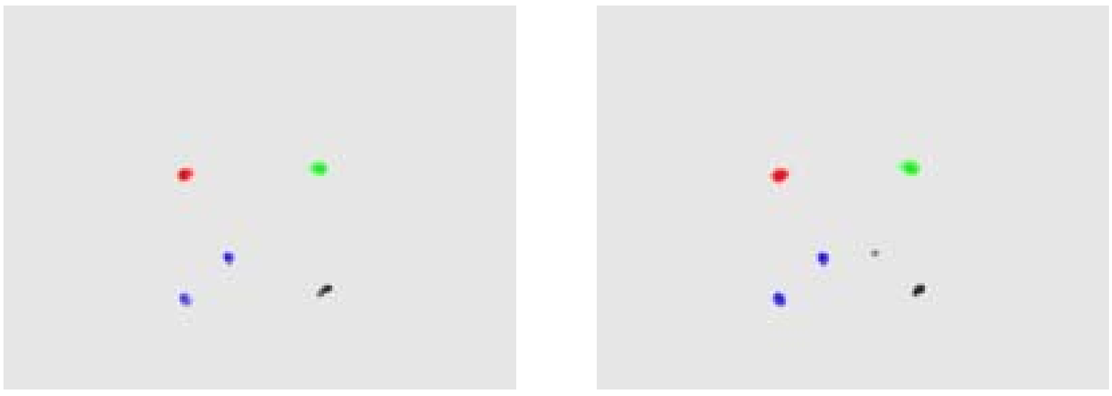

(a) (b)

Figure 5.20: RVM classification result for black LCD screen by using Scanimg utility. (a) The classification results on image shown in Fig.4.3a by using L-2.0 kernel function. (b) The classification results on image shown in Fig.4.3a by using L-3.0 kernel function.

5.5 Conclusions

In this chapter, we have discussed plenty of work on classification by RVM. Now it is time to summarize and come to some conclusions.

In the first section, we introduced the theory of RVM and we know that RVM essentially is sparse Bayesian learning. Though its principle is totally different with SVM, but RVM has to appreciate it, because it still uses the same linear model and try to search out suitable weights and kernels to make predication.

Comparing to SVM, RVM brings some advantages, probability outputs, fewer vectors to the classification model and free kernel functions selection. In the tests on our screen corners recognition, we can easily see that the probability outputs are very useful information, by using them as weight factor, the geometric center is closer to the reference screen corner points. Moreover, in the final system, they are used to select out the screen corner candidates in rectangle filter algorithm. In a synthetic dataset for a black LCD screen, the sparsity of RVM is shown. After training, the final model on average only keeps 4.4 sample vectors for classification model, which much increase the classification speed. But there are still some drawbacks of RVM. The biggest potential problem is the training speed of RVM. Even for our dataset (not very big), the training also needs some time, so if the dataset becomes bigger then it will get much more slower. Therefore, in the paper [TipNew], the author gives an algorithm to raise the speed of training, though it is still slower than SVM, but it get much faster than before. But since we have not the time to implement it, we have to keep it as a future work.

Besides the theory of RVM, the more important part for this chapter is how to use it in screen corner detection. To do the real problem, firstly we use DCT as our feature extraction algorithm, because it can extract samples into small dimensional vectors. That makes the sample size independent and accelerates the training and classification speed. Another thing we consider is the rotation problem. Without rotating the whole image, we designed an algorithm that rotates target window, and reduces the computation and accelerate the speed as well. Finally, we came to the most important step that is sample dataset selection and training. In our research period on RVM, we found 3 ways to get sample dataset:

1. Manually select some screen corners from real images as corner samples and randomly select some images as non-corner samples.

2. Generate automatically screen corners as corner samples and randomly select some images as non-corner samples.

3. By using the Candidates-Winners approach and one of above dataset in RVM, automatically select bigger screen corners as dataset and the previously as wrong classified samples, background samples, and easily misclassified samples as non-corners samples.

Anyone of the three selection procedures has advantages and disadvantages. The first one is more accurate since the samples are selected manually, so we can easily control the samples quality. But the problem is the inconvenience for users. The third one is the most accurate one, because the MUR value of RVM is much lower than others, which can be easily seen from the figures in section 5.4.3. With this dataset we can train RVM, and are able to do the screen corner detection only by RVM. But since the samples' size is bigger than other datasets' the feature extraction speed in this dataset becomes much slower than others that finally effects our real-time speed requirement. So considering the two evident disadvantages of these dataset, finally, we take the synthetic dataset in the UI-Wand. Although its MUR is value still high but the detection rate is on the same level compared to other datasets. And for the black LCD screen, the G-0.5 kernel function is very suitable for the dataset, so that only 4.4 sample vectors keep in the end, which gives a much faster performance to RVM.

The Table 5.18 and Table 5.19 show performance results by using different dataset and kernel functions, which we have already shown separately in the sections before. The values marked in gray are the best value in one output item of Error analysis utility.

Table 5.18: Summary of datasets and kernel functions to RVM performance on white screen.

Dataset	KF	RVN	HCV	MV	GV	CGV	MUR	CD	WV
ME	G-0.5	18.5	70.00	14.13	7.07	4.62	93%	98%	13.08
	G-1	27.8	79.88	18.01	9.3	4.72	93%	99%	15.40
	L-1.0	21.0	7.06	14.97	6.73	3.93	94%	74%	6.47
	L-2.0	18.4	7.26	11.99	5.62	3.85	93%	98%	5.71
	L-3.0	27.6	6.81	12.09	5.68	3.9	93%	98%	5.72
SYN	G-0.5	15.6	47.63	28.16	12.57	10.47	94%	99%	17.07
	L-2.0	56.2	69.88	24.41	12.00	9.09	93%	99%	18.14
ADAP	G-1.5	45.4	97.01	16.14	7.99	6.50	68%	100%	17.11
	L-2.0	17.8	17.99	10.48	5.44	4.33	50%	100%	6.76

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2
ME: Manually selected sample dataset
SYN: synthetic sample dataset
ADAP: Adaptive sample dataset

Table 5.19: Summary of datasets and kernel functions to RVM performance on black LCD screen.

Dataset	KF	RVN	HCV	MV	GV	CGV	MUR	CD	SCO
SYN	G-0.5	4.4	36.94	5.38	3.47	2.57	84%	100%	6.65
ADAP	L-2.0	15.6	13.01	7.56	4.33	2.84	29%	100%	4.93
	L-3.0	12.4	11.67	8.03	4.54	3.16	30%	100%	5.05

*KF: Kernel function.
RVN: The number of relevance vectors.
CD: The number of detected screen corners.
WV: The same item with "wh_V" in section 10.2.3.2
SYN: synthetic sample dataset
ADAP: Adaptive sample dataset

Besides the accuracy of RVM, another important factor to UI-Wand is the speed that we did not mentioned in the previous sections, since the speed is fast enough for real-time application. Now to give a clearer concept to the users, we give some tests results in the following tables. The speed unit is samples per second the model can operate. From Table 5.20, we can see that smaller target size will make the DCT feature extraction faster and Table 5.21 shows the effect of the number of relevance vectors to the speed. There we only list two situations with the most relevance vectors and the least relevance vectors respectively.

Table 5.20: The speed of DCT feature extraction algorithm for different targets size.

Targets Size	**10x10**	**20x20**	**40x40**
Speed (samples/sec)	718	199	49

Table 5.21: The speed of RVM with different number of relevance vectors.

Dataset and Kernel Function	**B/SYN/G-0.5**	**W/SYN/L-2.0**
RVN	4.4	56.2
Speed (samples/sec)	1232	961

Given the with high detection rate and real-time speed, however, RVM still suffer from the high MUR value, which means that the chance to classify a non-corner sample to a screen corner class is very high. Thus we cannot just depend on RVM as winner selection method (see Fig. 5.21). There must be an additional algorithm to select out the final screen corners. So we introduce a rectangle filter as post-processing to finish this final task in Candidates-Winners approach.

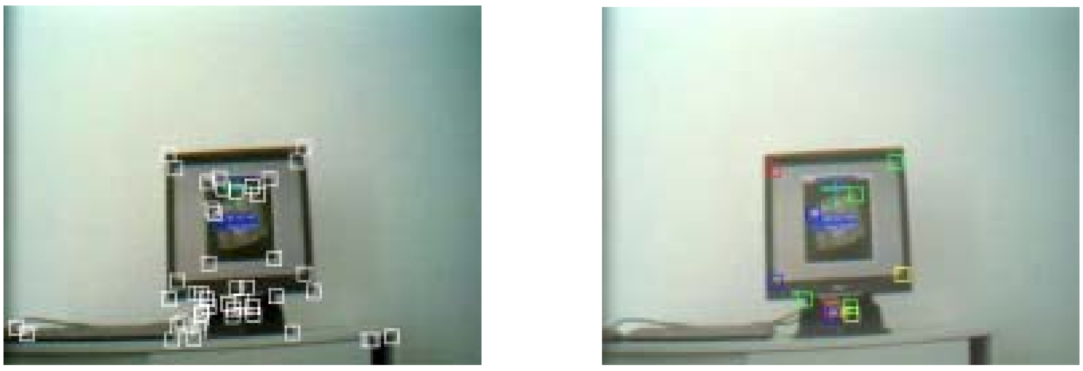

(a) (b)

Figure 5.21: Candidates-Winners approach only with Sojka corner detector and RVM classification with synthetic dataset. (a) Candidates after Sojka corner detection (b) Candidates after RVM classification. Red/Green/Blue/Yellow/ represents left-top/right-top/left-bottom/right-bottom screen corner respectively.

6 Rectangle Filter

We showed in last section that we couldn’t use an adaptive dataset for RVM because it needs too much time on feature extraction step. So in the final UI-wand system we utilize synthetic sample dataset, which can give the system a real-time processing speed, but the problem it brings is that RVM cannot select out the four screen corners directly (see Fig.5.21).

Because still lots of candidates are classified as screen corners by RVM, we need an additional algorithm to select the real screen corners out. Till now there is important information we did not use that is rectangle geometric feature of the screen. Because the physical screens UI-Wand might be used on have a rectangle shape, their four corners should also compose a rectangle shape in the image when UI-Wand just aiming directly in front of screen. Moreover, even when the UI-Wand orientation has some angles with the screen in a reasonable range, the relationship of those four corners is still keeping a similar rectangle shape. Of cause, you can argue that this rectangle will be distorted very much when the angle is very big, but this can be figured out in the tracking algorithm. As long as we know the screen in the first frame then we can track the possible four screen corners in the later frames and we can know the screen shape exactly by continually updating the angles, then the distortion will not be a problem. And for the first frame detection, we can just constrain the user to stand in a suitable position and hold the UI-Wand in a correct way, so that the four corners make a similar rectangle shape, and which ensures a correct detection will be taken.

6.1 Rectangle filter given corner type

The problem displayed in Fig. 6.1a is that RVM classified more than four corners to screen corners. Considering the rectangle feature of the screen, we wrote a rectangle filter to find out which should be the real screen corners. The filter algorithm consists of the following steps:

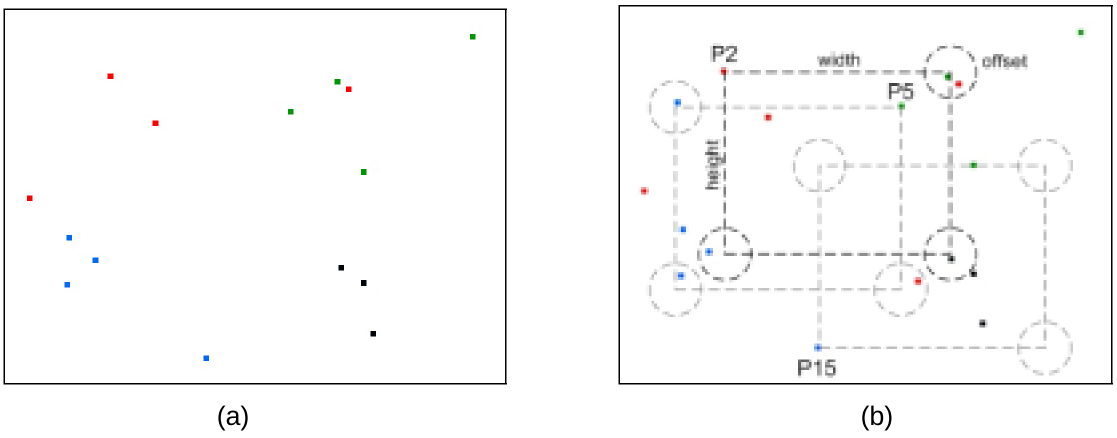
1. Define a rectangle by given a width and height, which the screen might be on.

2. Read a list of corners with type assigned by RVM.

3. Run a circle to get each corner from the list and analyze if it can be a screen corner. Firstly we assume a corner was a screen corner and then we go to find if the three other types screen corners are in the right places defined by the rectangle in the first step. For example, see Fig. 6.1b, assume P2 is a left-top screen corner, then we go “width” pixels left to find if the right-top corner is in an area that is centered at P(P2.x+width,P2.y+0) and radius is the offset value we defined. In this case we can find a possible right-top screen corner. Repeat the same procedure then we go to find if left-bottom and right-bottom exists at

the same time. In this case, both of these two corners are found. So we assume these four corners possibly are the screen corners thus we record them into a list. The gray rectangle in Fig. 6.1b shows the candidates corners that are not screen corners. P5 and P15 are not screen corners because P5 only find one possible left-bottom screen corner and P15 did not find any others.

4. We get the final screen corners after analysis of every corner in the input list.



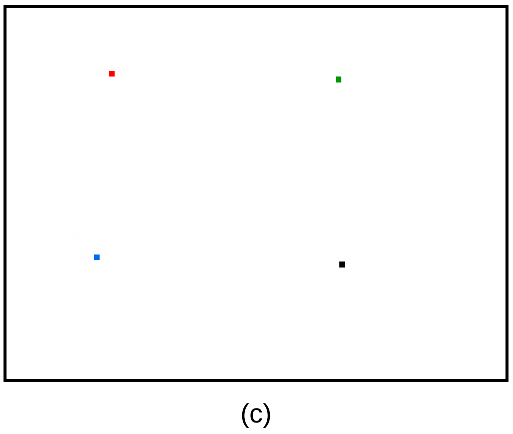


Figure 6.1: Algorithm of rectangle geometric filter given corners type. Red, Green, Blue and Black represent the left-top, right-top, left-bottom and right-bottom screen corners. (a) The corner points after RVM. (b) The rectangle filter algorithm process. The black rectangle is the possible position of the screen and the corners in the black circles are the possible screen corners. (c) The four screen corners after rectangle filter with corner type.

There are two extra rules for two complicated cases. One is for the case that the algorithm detected more than one comer in the offset circle. The other is for the case that the algorithm found more than one screen corners combination. For these problems we can use the probability values outputted to handle. For the first case, we select the corner with the highest probability. For the second case, the same, we select the combination corners with the highest probability.

Another problem we need to consider is the rotated image. When the input image is rotated then we cannot use a right rectangle to filter the screen corers any more. But this problem can be solved easily. We use the same mechanism as in feature extraction. When we know the angle of rotated image, we do a transformation to the rectangle so that it will become tilted as the same angle with rotated image, then it can filter out the screen corners and the problem is solved. Fig. 6.2 shows the filter performance.

By using the rectangle filter described above, we can success fully selecting out the real screen corners. But before we summarize our Candidates-Winners approach, we have to mention here another filter that was used after Sojka corner detection algorithm.

6.2 Rectangle filter without type information

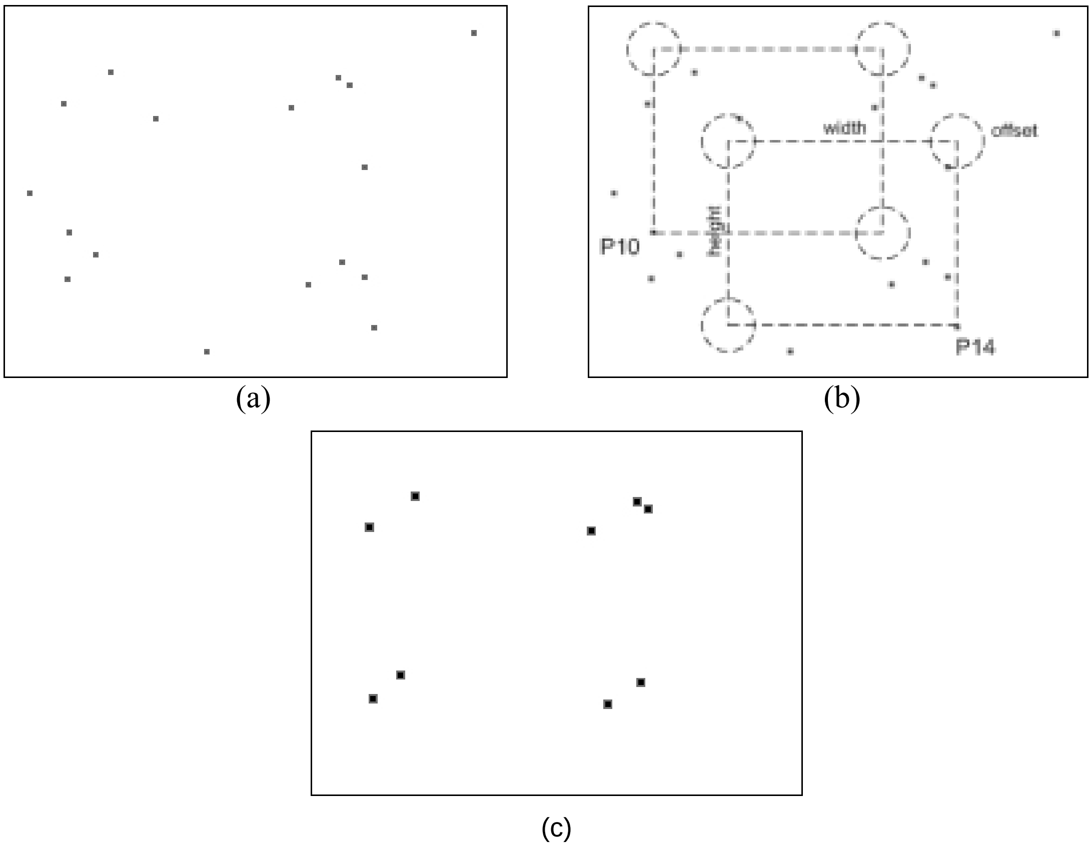


Figure 6.2: Rectangle filter without corner type information. (a) The corners detected by sojka detector. (b) The algorithm of the filter process. (c) The candidates after rectangle filter used after sojka corner detection.

This rectangle filter's principle is the same as the last one, but the difference is that it was used before RVM. So it will filter out the corners without type information and actually the purpose of this filter is to find out the corners that possible form a rectangle shape with each other so as to reduce the number of candidates feeding to RVM.

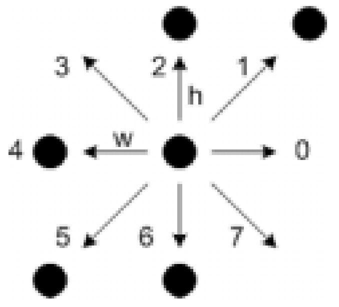


Figure 6.3: Search directions in rectangle filter algorithm.

Let's consider the same situation as shown in Fig. 6.1a. Now this time we have not corner type information so that the corners color change to gray one (see Fig. 6.2a).

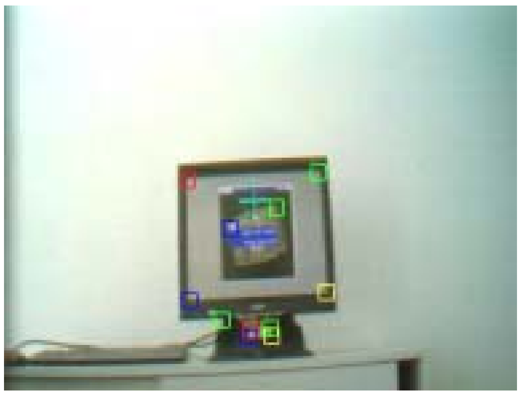
The specific filter algorithm is almost the same with as the last filter. The only difference is in the third step of the algorithm. Now we cannot constrain the search to three directions to find the other three corners, instead we need to explore eights directions to search the possible corners that can form a rectangle with the center corner (see Fig. 6.3). Whether the center point can be filtered out depends on if it can find 3 corners around it, which can form a rectangle together with it. In the case shown in Fig. 6.3 the center point will not be filtered out because it finds out other three corners in direction four, five and six. But in the final system, we give a more flexible rule to both rectangle filters, which keeps those corners that have two other corners around it (in Fig. 6.3, the points in direction one and two let center point satisfy the rule), which make the filters tolerant for missing points situation that can be caused by Sojka detection and RVM.

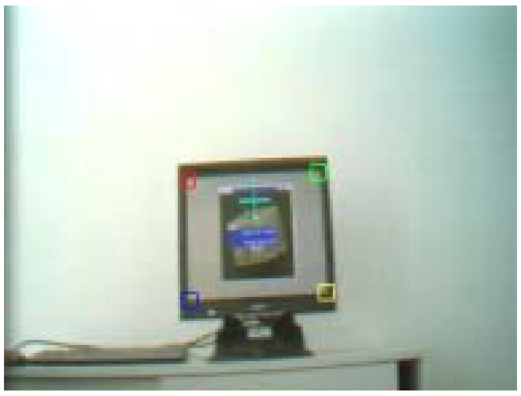
In Fig. 6.2b the corner P10 and p14 are filtered out, because they cannot find other points around it. After this filter we got the corner list as shown in Fig. 6.2c. The number of candidates is reduced so that the speed gets improvement.

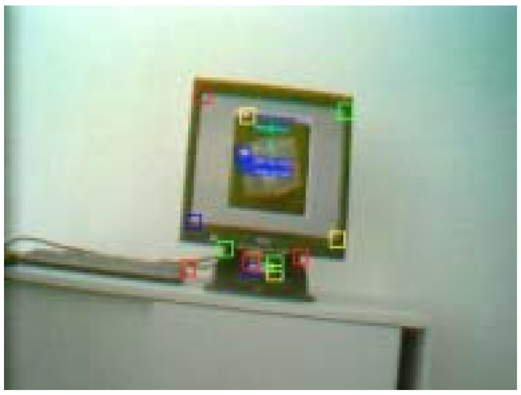
6.3 Conclusions

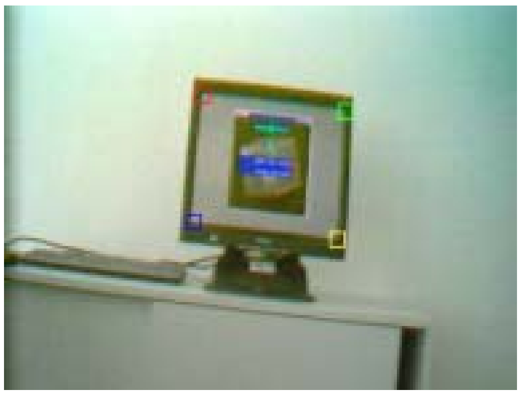
The rectangle filters play the last role in Candidates-Winners approach. After testing, the results show that it gives a very accurate selection in the end (see Fig. 6.4 for some examples). With respect to the speed, it also performs very well. Both of the filter's time complexities are $O(N^2)$, where N is the number of candidates that are received. Because in reality the number is not big (less than 40 candidates), the processing time for one image is generally less than 1 millisecond.

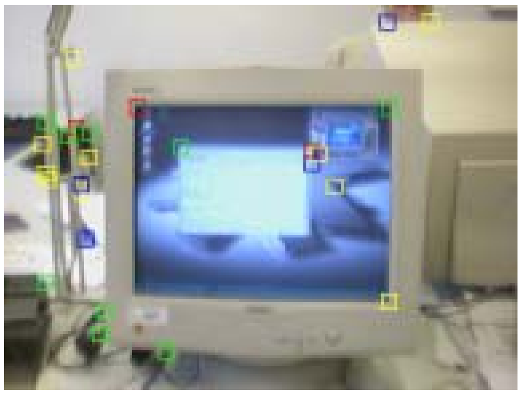
A prerequisite of this filter algorithm is that we have to setup an initial width and height of the screen in the image to fixed values, which means that the user has to start using UI-Wand from a certain location in front of screen. This can be improved by giving a set of possible starting location of user then calculate the possible width and heights pairs or even make some hypothesis of four screen corners. After that use a loss function to figure out which filter can give a best score to a combination of screen corners, which can be seen as the final selection. But we did not realize this idea in our final system.


(a)


(b)


(c)


(d)


(e)

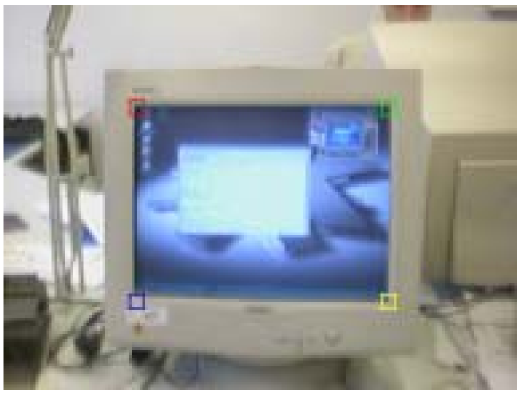

(f)

Figure 6.4: Rectangle filter used in Candidates-Winners approach. (a) (c) (e) The candidates after classification by RVM. (b) (d) (f) The final screen corners selected after the rectangle filter. Red, green, blue and yellow represents left-top, right-top, left-bottom and right-bottom respectively. In (e) the left-bottom corner is missed, but it can be estimated by other 3 corners.

7 ROI Tracking

In the last chapter, we described the models and algorithms used for screen corner detection. After the screen corners have been detected, we can use positioning model mentioned in section 3.2.2 to get pointing position and also can start doing gesture recognition, which is another important functionality of UI-Wand. So in this chapter and next chapter we will discuss issues of gesture recognition. Especially, in this chapter, we will discuss issues about tracking, which gives a faster detection to screen corners in a sequence of frames.

7.1 ROI tracking filter

Before gesture recognition model in UI-Wand, we have to know the trace the UI-Wand has passed. In our solution, the positions of screen corners in a sequence of frames can form a representative and effective trace of UI-Wand. So the current problem is to detect screen corners in every frame of a sequence and save them into a list then pass it to the gesture model to recognize. The screen corner detection model is Candidates-Winners approach, which was already proved to be effective to screen detection. But the problem of using Candidates-Winners approach on a sequence is the speed. Although we consider this problem in the detection step and use compact algorithms in it, the speed of detection still not satisfies the requirement, 10 frames per second, i.e. 100 milliseconds per frame. So we have to use some tracking method in the detection step.

7.1.1 ROI selection and predication

The essential aspect and purpose in tracking algorithm is to reduce the detection complexity in a sequence of frames and accelerate the speed. By using tracking algorithm, we only need to detect the targets we are interested in the first frame and then the algorithm will find the targets in the following frames by just searching the targets around the previous positions. The classical tracking method is to compare the target intensity values with the area around it then choose a place as the target new place that has the lowest difference score of intensity between the new position and the old one.

But in our case, we did not use this typical method. Instead, as what we described in the system design, we still use Candidates-Winners approach to detect screen corners on the frames after the first detection. We constrain the detection on a small area that is a region of interest (ROI) where the screen corners might show up. Since the ROI areas are much smaller comparing to the size of the complete image, so the computation complexity is reduced and the speed acceleration is reached.

Now the question is how to choose the ROIs in a frame. Fig. 3.5 shows the selection method. Given four screen corners detected in the last frame, the ROIs of screen corners in the next frame is the area, centered at the positions in the last frame and a certain value as diameter. The procedure of selection is shown in Fig. 7.1.

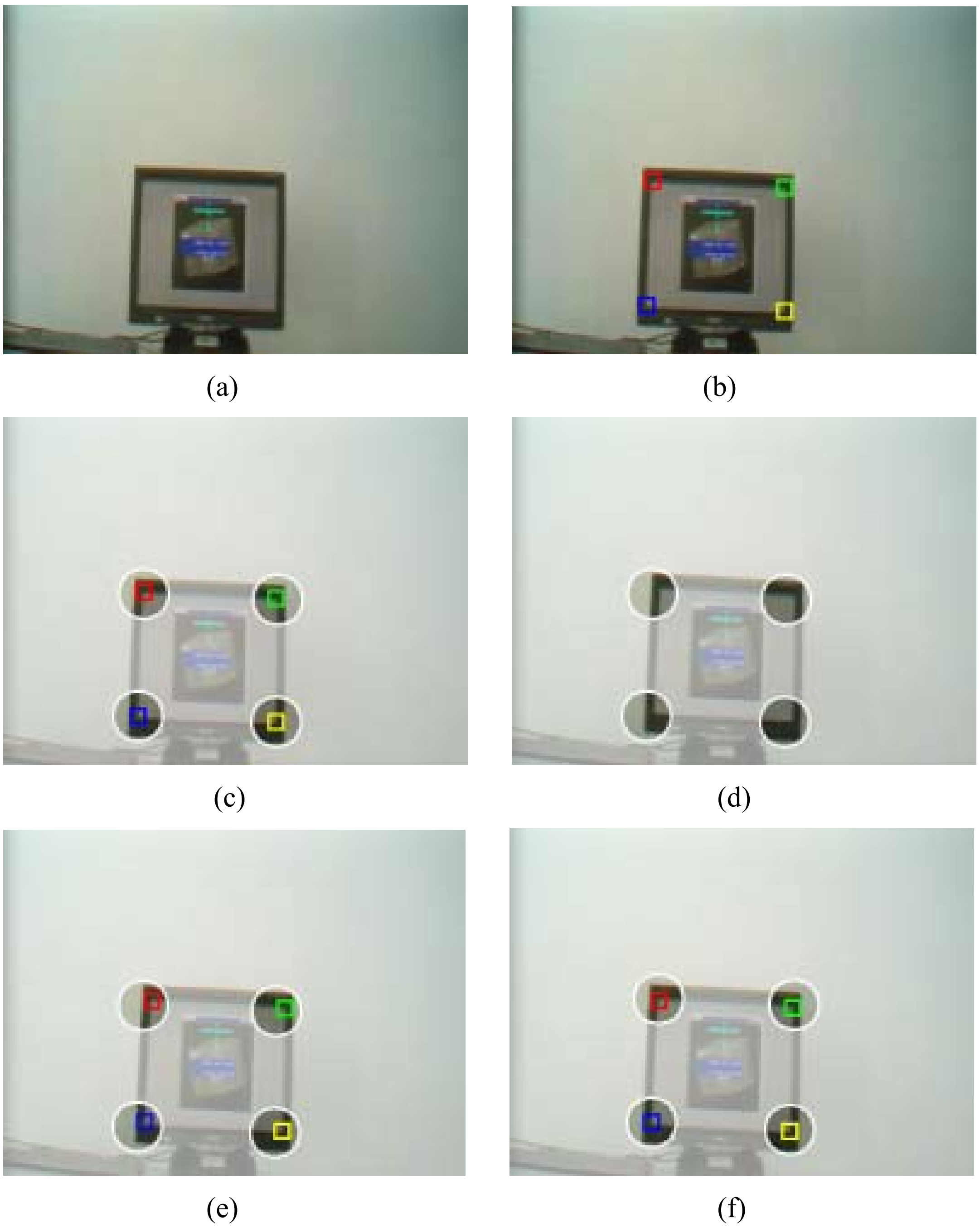


Figure 7.1: ROI selection procedure in a sequence of two frames. The white circle represents ROIs. (a) The first frame. (b) Screen corners detected in the first frame. (c) ROIs selection in first frame according to the screen corners. (d) The ROIs show up in the second frame. (e) The screen corners detected in the ROIs in the second frame. (f) The new ROIs created by the screen corners detected in the second frame.

But this simple selection method has potential problems, because the faster movement of UI-Wand can cause error detection or missing detection by using this kind of selection of ROI. If the user moved UI-Wand very quickly then the screen of image would jump far from the last image, so if the diameter of ROI is too small then definitely you will miss the screen corners. To solve this problem, the first way is to expend the ROI by giving it a bigger diameter value, but that will increase the computation workload. The second way is to make more accurate selection to ROI, so that the system can detect the screen corners keeping the size small.

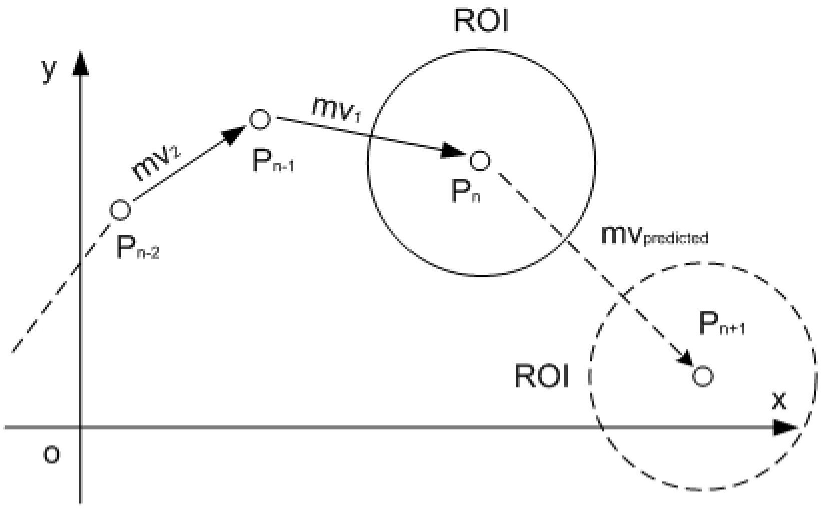


Figure 7.2: Prediction of ROI center procedure. The solid ROI is obtained directly use the point. The dot ROI is obtained by doing a prediction with motion vectors.

The method to make more accurate selection of ROI is to do some predication. We can utilize the positions of screen corners detected in previous frames to predict their possible positions in the new frame. Then use the predicted position as the center and a certain value as diameter to form the new ROIs where we go to detect the screen corners. Kalman filter [Kal60][Wel01] is known as a good model for this kind of predication. But considering the limited time, we did not realize it but use a simple prediction algorithm to replace it. The principle of this algorithm is to predict the center position or ROIs by previous motion vectors (see Fig. 7.2). The algorithm is explained as follows:

1. If there is only one frame in the past, then the center point of ROI in the new frame is at the screen corners positions in the last frame, which is the same with the simple selection mentioned above.

2. If there are two frames in the past then the center point of ROI in the new frame is at $P_3 = P_2 + \mathbf{mv}_1$, where P_2 is the screen corner's position in the second frame and $\mathbf{mv}_1$ is the motion vector calculated by $P_2 - P_1$.

3. If there are more than two frames, we calculate out the center point of ROI by previous two motion vectors, $\mathbf{mv}_1 = P_n - P_{n-1}$ and $\mathbf{mv}_2 = P_{n-1} - P_{n-2}$, where P_n is the position of that point in the nth frame. After that, we predict that the center of ROI in the new coming frame is at $P_{n+1} = P_n + \mathbf{mv}_{predicted}$, where $\mathbf{mv}_{predicted}$ is calculated by $\mathbf{mv}_1$ and $\mathbf{mv}_2$.

$$\|\mathbf{mv}_{predicted}\| = \|\mathbf{mv}_1\| * \|\mathbf{mv}_1\| / \|\mathbf{mv}_2\|$$

The angle of $\mathbf{mv}_{predicted}$, $\alpha_{predicted} = \alpha_1 + w * (\alpha_1 - \alpha_2)$,

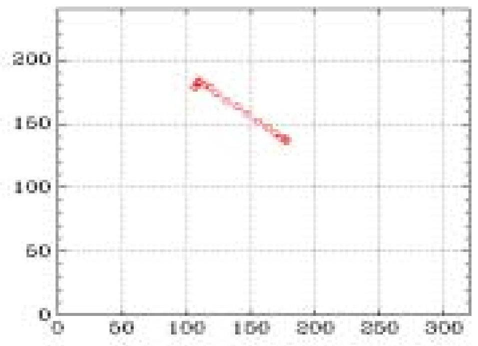
where α_1 is the angle of $\mathbf{mv}_1$, α_2 is the angle of $\mathbf{mv}_2$, and the w is $\|\mathbf{mv}_2\| / \|\mathbf{mv}_1\|$, if $\mathbf{mv}_2 \le \mathbf{mv}_1$, $\|\mathbf{mv}_1\| / \|\mathbf{mv}_2\|$ if $\mathbf{mv}_1 < \mathbf{mv}_2$.

But the problem of this is that it can be disturbed easily by noise. To solve this problem, we add extra rules in this algorithm to let it stop when the movement of points seems strange, e.g. sudden change of the angle or sudden increase or decrease of the magnitude of the motion vectors. With these additional rules, the tests results show that the algorithm becomes more stable.

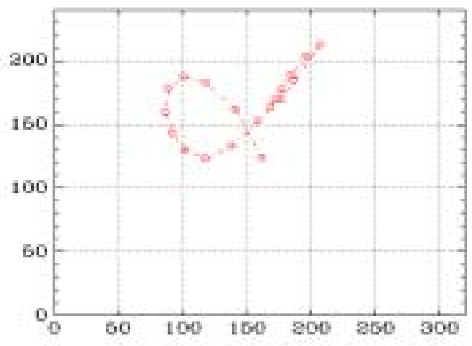
7.1.2 Screen corner detection in ROIs

By the above ROI prediction method, we can directly detect screen corners on the ROIs with smaller size. The special detection approach is almost the same with Candidates-Winners approach, but the only difference is that now we do not need to use rectangle filter to select out the final winners since the four ROIs already satisfying a similar rectangle shape. The final winner selection step is to only choose the biggest screen corner with the highest probability value in one ROI.

The final tracking algorithm accelerates the detection very much and also keeps high detection rate when the UI-Wand moves quickly (see Fig. 7.3). But there are still a lot of cases that the tracking filter would fail to track. So in the final system, if it cannot correctly detect screen corners in several frames consecutively, then the ROI tracking filter would stop working and the UI-Wand would use Candidates-Winners approach on the whole image again to select screen corners, after which the track filter will start running again.



(a)



(b)

Figure 7.3: The trace of screen geometric center in the image, which tracked by ROI tracking filter. The geometric center is calculated according to the four screen corners. (a) The movement from pointing one place to another place. (b) The UI-Wand is doing a “cross” gesture.

Besides the more accurate ROI position can be estimated, another advantage of this prediction is that it can smooth the trace of UI-Wand and give trend information of moving UI-Wand even if Candidates-Winners approach cannot detect the screen corners in the predicted ROIs. Because in the final system, if we cannot detect screen corners in ROIs, then we will save the centers of ROIs as the screen corners positions, though these positions are not the correct ones, at least they can be seen as giving correct movement direction information, which will be useful enough for the gesture recognition.

7.2 Missing screen corners predication

By far, we have described many procedures about filter algorithms including rectangle filter and tracking filter, which can ensure the correct and faster detection of screen corners. But there is still a big problem with these filters, haven't mentioned yet, which is missing detection problem. The variant of lighting conditions and the distortion of the image captured by the camera often make the detection work much harder, so the missing detection situation occasionally happen in Sojka detection algorithm or RVM recognition step. In order to prevent the final system from this problem, we write some algorithms to predict corners when there are some ones missing.

The missing situation can be divided into two categories, one is missing in the first frame before that we did not detect any screen corners, and the other one is missing in the frame before we ever detect a combination of four screen corners. To more convenient, we name the first missing situation as Type-I and the second one as Type-II.

To the Type-I missing, since we have not detected screen corners before, there are few things we can do to predict the missing corner. Actually we can only estimate one missing corner with other three screen corners detected. The estimation method we took is very simple. We just assume the three screen corners as three convexes of a parallelogram then calculate one corner position by the three screen corners coordinates. It is not accurate since in reality, we cannot ensure the screen is a parallelogram, but since this Type-I missing seldom happens, it will not affect the accuracy of the final system.

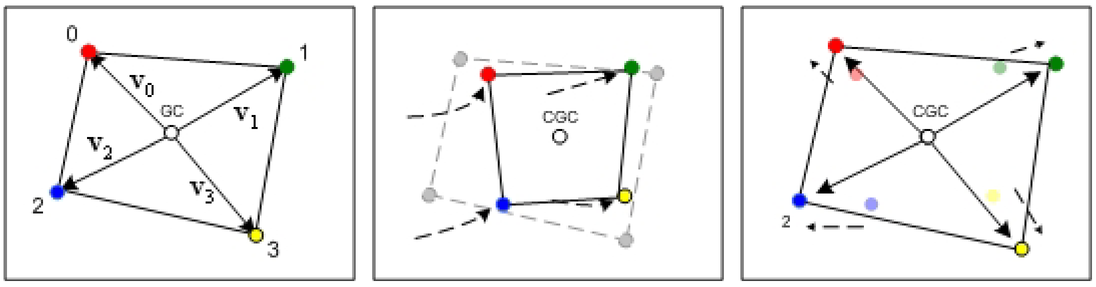
Comparing to Type-I missing, Type-II missing often happened and caused much problems. But since we already have screen corners been detected before, there are much more prediction methods for this type of problem, which also make the prediction more accurate. In the final system, we use the following rules to predict the missing corners.

1. After the detection step in every frame, we analyze the detected screen corners. If the system detected more than three screen corners, we think the screen is detected and then we save the screen corners positions into a list that can be seen as forming the screen shape in that frame. Keeping update of the screen shape will let the system know the recent screen shape.

2. When the tracking filter cannot track anyone of four screen corners, then we predict them like this. Firstly, we predict the screen corners position in the current frame by their previous positions detected in frames before. Then to make the prediction more accurate, we use an alignment method that utilizes the screen shape obtained in the first rule to adjust the four screen corners. The special procedure is shown in Fig. 7.4 below.

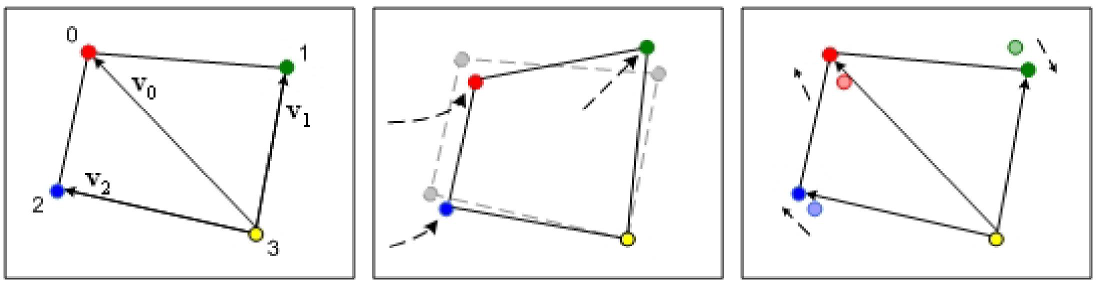
3. When the tracking filter detected more than one screen corner, we do not predict the others by their trace before, instead, we directly use the screen shape obtained in the first rule to align it with the screen corners that we have

detected we can work out the missing corners' positions. The detailed procedure is shown in Fig. 7.5 below.



(a) (b) (c)

Figure 7.4: Prediction and alignment when missing four corners. (a) The screen shape saved in a previous frame. GC is geometric center. (b) The screen corners after prediction by their previous traces. CGC is current geometric center. The gray quadrangle is the position of the real screen. (c) The four screen corners after alignment.



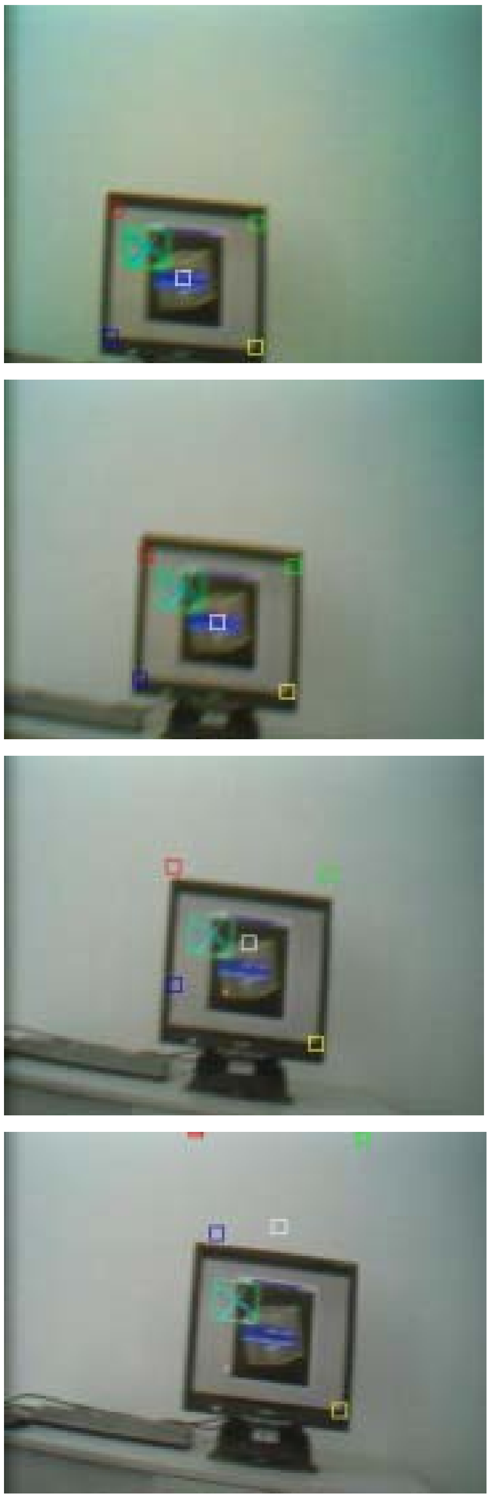
(a) (b) (c)

Figure 7.5: Prediction and alignment when missing three screen corners. (a) The screen saved in a previous frame. (b) The detected and estimated screen corners. The yellow one is detected and the others are estimated. The gray quadrangle is the real screen in the image. (c) Based on the screen shape saved in (a), align the three estimated screen corners according to the right-bottom corner.

7.3 Conclusions

In Fig. 7.5, we show the prediction procedure on the case, one corner detected. The other three corners will be aligned only on this corner. For the cases that there are two or three corner detected, we take the same method but use each of the detected corners as reference corner to align the missing corners, then get the geometric centers as the final positions of those missing corners.

With this missing corners prediction procedure, the tests result shows a significant improvement for the screen corners tracking work. Now it can accurately predict the ROI positions and find out the screen corners even if the movement of UI-Wand is very fast. The Figure 7.6 shows 4 frames in a "cross" gesture, which cannot be tracked correctly only by ROI tracker but was correctly tracked by ROI tracker with missing corners prediction and alignment algorithm. The better tracker filter gives smoother trace information, though sometimes it is still not so accurate but it is enough for gesture recognition that is not sensitive too small deviation if only the system gives correct direction information.


(a)

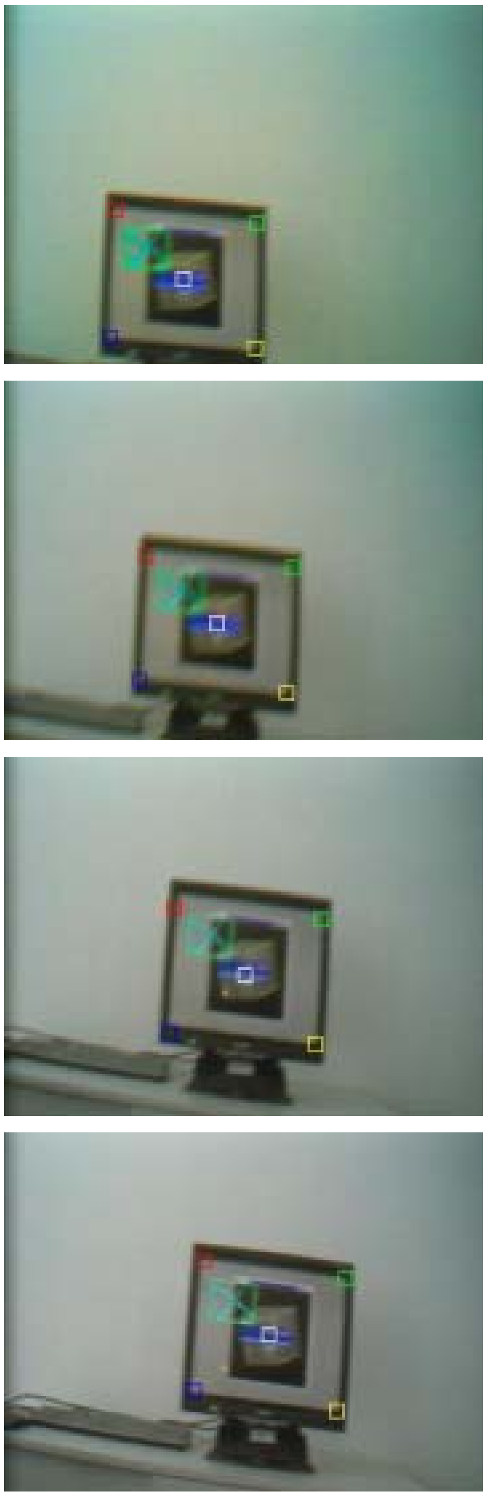

(b)

Figure 7.6: Tracking results with different algorithms. (a) The tracking results by ROI tracker without alignment step. (b) The better tracking results by ROI tracker with alignment step.

8 Gesture Recognition

Pattern recognition forms the mathematical basis of gesture recognition problems. Pattern recognition is a mathematically rigorous field with the purpose of classifying objects into one of a number of classes. The pattern recognition process is generally implemented in a manner that allows automatic recognition without human intervention. In gesture recognition, those gestures in 2D or 3D space are *patterns*. Our gesture recognition process is following the rules of pattern recognition, which is formed in three phases (see Fig.8.1).

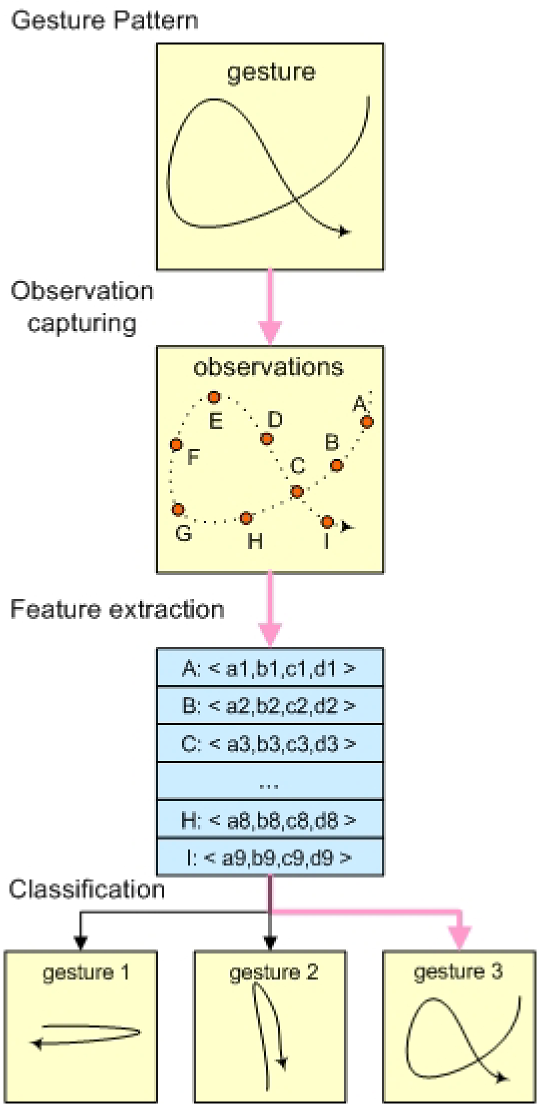


Figure 8.1: Gesture recognition process.

Observation capturing: where the actual gesture patterns are transformed into some discrete geometric points, which show the characteristic of that gesture.

Feature extraction: where the observations vectors of a gesture are transformed into a sequence of feature vectors, because feature vectors contain most of the information necessary for classification of the gestures and they are generally much more tractable for the system.

Classification: where a classifier specifies the class membership of the observations. In our UI-Wand project, we adopt two different approaches for our gesture classification [Jai00]; one approach is Hidden Markov Model (HMM) approach, which is a sequential pattern recognition method to decide the gesture label with highest probability of HMMs. we realized the HMM classification model in this thesis but without implementing in the real world. The other approach is RVM model approach, which partitions the feature space of a candidate gesture trace into disjoint regions and each region corresponds to a gesture class. This RVM model is really implemented in our project.

In the following sections, we will first introduce our HMM approach and then describe our RVM approach. The test result of our RVM gesture recognition approach will be given in the end of this chapter.

8.1 HMM for gesture recognition

Hidden Markov Model (HMM) is already a well-known model in gesture recognition field and there are a lot of projects based on HMM for gesture recognition (i.e. [Yan99], [Liu03] and [Lee99]). It falls into the "supervised pattern recognition" system. In our approach, we use an HMM-based classifier for our gesture recognition.

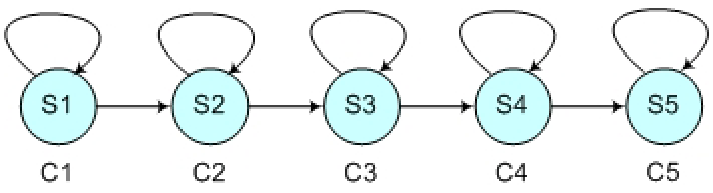


Figure 8.2: A five states left to right HMM model.

8.1.1 Introduction of Hidden Markov Model

A Hidden Markov Model (demoted λ) is a doubly stochastic process [Lus95]. The first stochastic layer is the underlying first-order Markov process (a stochastic process called a j^{th}-order Markov process if the conditional probability density of the current event, given all past and present events, depends only on the j most recent events), each state is a possible observation of the Markov process, and a transition probability from one state to the other state. The second stochastic layer of the HMM is the set of output probabilities for each state, the output probabilities specifies the likelihood of seeing certain observations, given the HMM is actually in a state.

In our HMM approach, we choose a *semi-continuous HMM* model, which is a hybrid of the discrete and continuous HMMs. Like the discrete HMM, the observation vectors are quantized into one of a finite set of classes, however like the continuous

HMM, the observation classes are modeled by a multivariate Gaussian pdf (probability density function), removing the distortion due to quantization and effectively modeling the variance of an observation class.

Now lets describe the process of our HMM approach [Lus95], we divided the whole process into training-model phase (Phase 1) and recognition phase (Phase 2) (see Fig. 8.3).

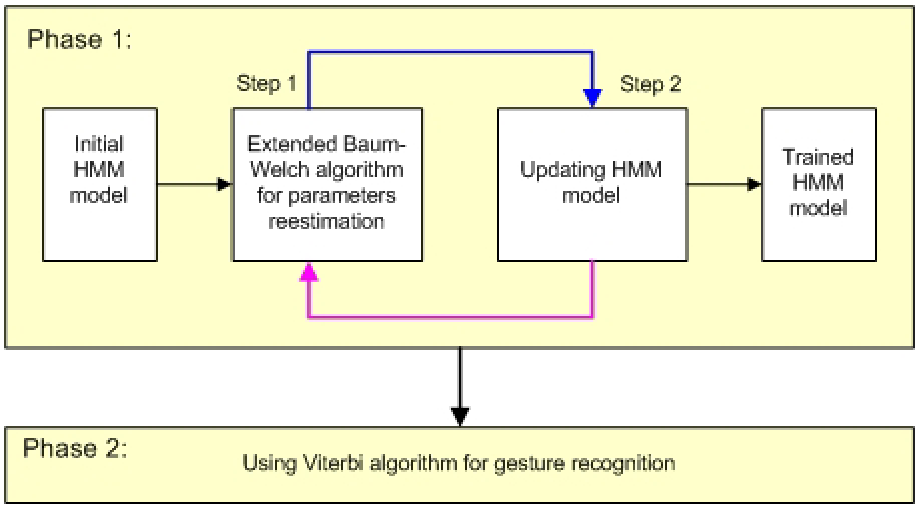

Phase 1:
Step 1
Step 2
Initial HMM model
Extended Baum-Welch algorithm for parameters reestimation
Updating HMM model
Trained HMM model
Phase 2:
Using Viterbi algorithm for gesture recognition

Figure 8.3: Generate HMM model for gesture recognition.

Training phase is to generate the stable HMM models for different gestures. In a HMM model (denoted by λ), there are some parameters involved [Lee99], they are:

- $\{s_1, s_2, s_3, \ldots, s_N\}$: A set of N states in HMM.
- $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots, \mathbf{x}_T\}$: A set of T distinct observation symbols. The observation at time t is denoted as $\mathbf{x}_t$.
- $\mathbf{A} = \{a_{ij}\}$: A $N \times N$ matrix for the state transition probability distributions where a_{ij} is the probability of making a transition from state s_i to s_j.
- $\mathbf{B} = \{b_j(\mathbf{x}_t)\}$: A $N \times T$ matrix for the observation symbol probability distributions where $b_j(\mathbf{x}_t)$ is the probability of emitting $\mathbf{x}_t$ at time t in state s_j.
- Initial conditions for that the HMM model is left to right with non-circle model, which begins at state s_1 and end at state s_N.

8.1.2 Theory of HMM model for gesture recognition

We use 2D mouse gesture trace points on the screen as the input of our HMM model. These data are then transformed into a feature vectors sequences. Each sequence contains a few vectors; each vector has four dimensions and contains screen-scaled position and velocity components. These feature vector sequences generate the training data sets and the testing data sets. After the training data has been generated, we generate a new codebook of specified size and a set of HMMs (one per gesture) with a specified number of states. The codebook is clustered on the observation vectors into different HMM states using a modified K-means algorithm. The HMM are created with a choice of left-right transition matrix.

After initialization, the HMM parameters are iteratively improved with a modified Baum-Welch reestimation algorithm [The03].

The "output" quantity in any path procedure is $p(\mathbf{X}|\lambda)$, thus estimating the parameters of the HMM model λ so that $p(\mathbf{X}|\lambda)$ is a maximum is nothing but a maximum likelihood parameter estimation procedure. We define two probabilities:

- $\xi_t(i, j, \mathbf{X}|\lambda) \equiv$ The probability of the joint event:
  a) A path passes through state i at time t.
  b) Through state j at the next time $t+1$.
  c) The model generates the available sequence of observations $\mathbf{X}$ given the parameters of the HMM model λ.

- $\gamma_t(j, \mathbf{X}|\lambda) \equiv$ The probability of the joint event:
  a) A path passes through state j at time t.
  b) The model generates the available observation sequence $\mathbf{X}$, given the parameters of the HMM model λ.

From the above definition, we can see there are two procedures that form the *forward-backward algorithm*, where $\alpha_t(j)$ accounts for the path history terminating at time t and state j; $\beta_t(j)$ accounts for the future of the path, which at time $t+1$ is state j and then evolves unconstrained until the end. The *forward probability* $\alpha_t(j)$ is defined as the joint probability of observing the first t vectors and being in state j at time t:

$$\alpha_t(j) = P(\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_t, s_t = j|\lambda) \qquad (8.1)$$

In a HMM model, states 1 and N are non-emitting, the forward probability is recursively calculated by the equation:

$$\alpha_t(j) = \left[\sum_{i=2}^{N-1} \alpha_{t-1}(i) a_{ij}\right] b_j(\mathbf{x}_t) \qquad (8.2)$$

which with initial conditions $\alpha_1(1) = 1$ and $\alpha_t(j) = a_{1j} b_j(\mathbf{x}_t)$ for $1 < j < N$ and final condition $\alpha_T(N) = \sum_{i=2}^{N-1} \alpha_T(i) a_{iN}$. This recursion asserts that the probability of being in state j at time t and seeing observation $\mathbf{x}_t$ can be calculated by adding the forward probabilities for all possible predecessor states i weighted by the transition probability a_{ij}. The backward probability $\beta_t(i) = P(\mathbf{x}_{t+1}, \mathbf{x}_{t+2,} \ldots \mathbf{x}_T | s_t = i, \lambda)$, in an opposite manner, can be recursively computed using the equation:

$$\beta_t(i) = \sum_{j=2}^{N-1} a_{ij} b_j(\mathbf{x}_{t+1}) \beta_{t+1}(j) \tag{8.3}$$

which with the initial conditions $\beta_T(i) = a_{iN}$ for $1 < i < N$ and the initial condition $\beta_1(1) = \sum_{j=2}^{N-1} a_{1j} b_j(\mathbf{x}_1) \beta_1(j)$.

The forward and backward probabilities lead to a convenient method for finding the likelihood of state occupation $\gamma_t(j) = P(s_t = j | \mathbf{X}, \lambda)$ as follows:

$$\gamma_t(j) = \frac{1}{P} \alpha_t(j) \beta_t(j) \tag{8.4}$$

where $P = P(\mathbf{X}|\lambda) = \alpha_T(N)$, which is the probability of observing the sequence $\vec{X}$ given the HMM model λ. Then we can get $\xi_t(i, j)$ very easily:

$$\xi_t(i, j) = \frac{\alpha_t(i) P(j|i) P(\mathbf{x}_{t+1}|j) \beta_t(j)}{P} \tag{8.5}$$

From the foregoing it is not difficult to see that:

- $\sum_{t=1}^{N} \gamma_t(j)$ can be regarded as the expected number of times that state j occurs, given the observation sequence $\mathbf{X}$ .When the upper index in the summation is $N-1$, this quantity is the excepted number of transitions from state j.

- $\sum_{t=1}^{N-1} \xi_t(i, j)$ can be regarded as the expected number of transitions from state i to j, given the HMM model and the observation sequence $\mathbf{X}$.

8.1.3 Realization HMM model for gesture recognition

The preceding definitions lead us to adopt the following re-estimation formulas as reasonable estimations of the unknown model parameters.

$$\overline{P}(j|i) = \frac{\sum_{t=1}^{N-1} \xi_t(i, j)}{\sum_{t=1}^{N-1} \gamma_t(i)} \tag{8.6}$$

$$\overline{P}(i) = \gamma_1(i) \tag{8.7}$$

In order to realize the algorithm in a software program, the whole iterative algorithm can now be expressed in terms of the following steps (see Phase 1 in Fig 8.3):

1. Initial conditions: Assume initial conditions for the unknown quantities. Compute $P(\mathbf{X}|\lambda)$.

2. Step 1: From the current estimates of the model parameters re-estimate the new model $\bar{\lambda}$ via Eq. (8.6) to Eq. (8.7).

3. Step 2: Compute $P(\mathbf{X}|\bar{\lambda})$. If $P(\mathbf{X}|\bar{\lambda}) - P(\mathbf{X}|\lambda) > \varepsilon$ set $\lambda = \bar{\lambda}$ and go to step 1. Otherwise stop.

We can use this way to train several HMM models, and each model for one gesture. While training the HMM model is a complex and time-consuming process, performing recognition process is much simpler. Given an observation sequence, each HMM model is scored on how well they describe the sequence. The HMM model with the highest score is chosen as the likely generator of the observation and then the label belongs to that model can be given. The gesture should be recognized then.
We use Viterbi algorithm to find the single best state sequence $\hat{\mathbf{s}} = (s_1, s_2, \ldots, s_T)$ for the observation sequence $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T)$. In our approach we only need the score of the state sequence rather than the actual state sequence. So the intermediate score can be defined as:

$$\delta_t(j) = \max_i \left[\delta_{t-1}(i) a_{ij}\right] b_t(j) \tag{8.8}$$

which with the initial conditions $\delta_1(1) = 1$ and $\delta_1(j) = a_{1j} b_1(j)$ for $1 < j < N$ and final condition $\delta_T(N) = \max_i [\delta_T(i) a_{iN}]$. The score of the HMM for a particular observation sequence is simply $P(\mathbf{X}|\lambda) = \delta_T(N)$. To decide the label of a gesture observation, all HMM models go through the Viterbi procedure. The gesture's classification is the gesture label corresponding to the highest scoring HMM.

8.1.4 Conclusions of HMM for gesture recognition

HMM is already a very popular approach in gesture recognition field, but we have to predefine the HMM model for each gesture and need much time and efforts to train these models. The quality of the model is determined by the training samples and quantities. For more complex and maybe continuous gesture, we must find a way to detect the end point of the former gesture and the start point of the next gesture, which is more challenged work. The speed of the HMM procedure is also a problem for real-time recognition, although there are some existing method to speed up HMM process (such as [Lee99]), but we still need to do more research on it. All the above problems cause us to find a new way, which better depends on our existing resource and cutting the developing time. Naturally we think about using our RVM classification model to do the gesture recognition work, which is described in details in the next part.

8.2 RVM for gesture recognition

Relevant Vector Machine (RVM) has already been proved to be a very good classification model in [Tip01] and also been successfully used by us for corner classification in Chapter 5. So we think about using RVM to realize our gesture recognition task. Like other pattern recognition project, we will follow the same process as we did with HMM model approach. First of all we will get the observations of the gesture, analyze the gesture to find candidate gesture trace, interpolate the candidate gesture trace for extracting features from these new observations. Then we will make many samples for training our RVM classification model. After training the RVM model, we can use this model to classify the different gestures. The whole procedure is depicted in Fig.8.4. In the following, we will describe how we design this gesture recognition model.

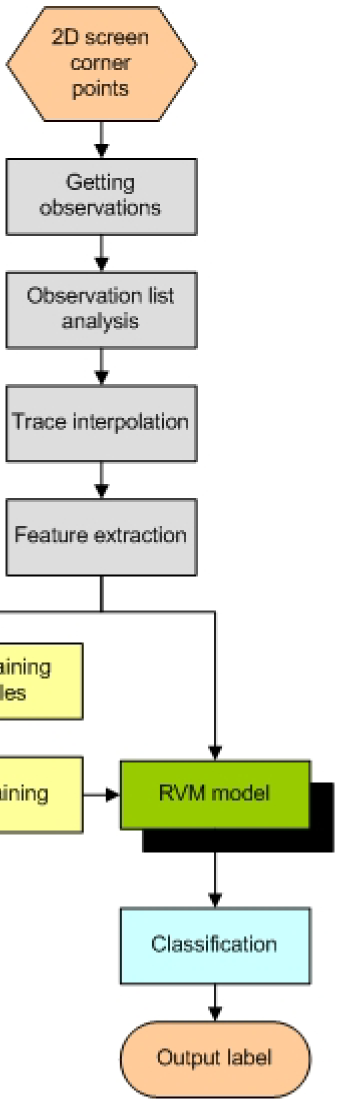


Figure 8.4: RVM for gesture recognition process flow chart.

8.2.1 Observation getting

In Chapter 7, we have already discussed how to use ROI for tracking screen corners. Here we will use these 2D screen corner points to get the gesture observations. We use the geometric centre of the detected four screen corners as the observations (one position per frame and the frame without screen corners will be marked as "MISSING"). From these centre points we can get a gesture trace, but how to choose candidate gesture trace from this trace is a problem for us, because in this trace, there may exist more than one candidate gestures. So we must analyze this trace then we can know which parts of the trace are candidate gesture traces.

8.2.2 List analysis for getting candidate gesture traces

We define a point list (see Fig. 8.5) with limited length (usually it has at least 50 elements in the real case, and can be changed in different conditions) to store the geometric center of the screen corners.

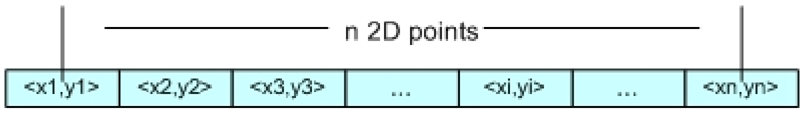


Figure 8.5: Analysis list example (n>=50).

Now let us explain the list analyzing process. In the initial state, the list is empty, and the first observation position enters the list, it is named *base_position*. Then the second position enters the list and compares its position with the *base_position*, if the distance is within the *MOVEMENT_SENSITIVITY*, then its status is marked as "HOLD", else it is marked as "MOVE", where *MOVEMENT_SENSITIVITY* is a limited circle range centered at the *base_position*. The following positions will enter the list one by one in order and be marked in the same way. This process stopped until there is a position, whose status is marked as "MOVE" and its direct previous position status is marked as "HOLD". If the number of its direct previous continuous "HOLD" positions is more than the *HOLDING_NUM_UNITTIME*, which is the minimum number of continuous "HOLD" positions, the process then marks this position's previous "HOLD" status as "MOVE_START", otherwise the process will change the status of all its previous "HOLD" positions into "MOVE".

Now the current position is the new *base_position*. Then the process works the same way as before until one "HOLD" position appears with more than *HOLDING_NUM_UNITTIME* number of continuous previous "HOLD" positions. The current position will be marked as a new initial position and the first "HOLD" position of these directly continuous previous "HOLD" positions will be marked as "MOVE_END". Now we can get a candidate gesture trace from the "MOVE_START" position to the "MOVE_END" position. Then we will send this trace to the gesture classification process or save the trace as a RVM gesture training sample, which depends on what kind of process we would like to do.

After a trace is found, all the processed position will not affect the future analysis of the list and the process is continued from the new initial position. This list analysis process will run in the same way again and again until all the positions of the original trace is processed. The analysis list will always pop-up the position at the beginning of the list and push a new positions in the end. So the length of the list will always be constant. An example of the list is shown in Fig. 8.6.

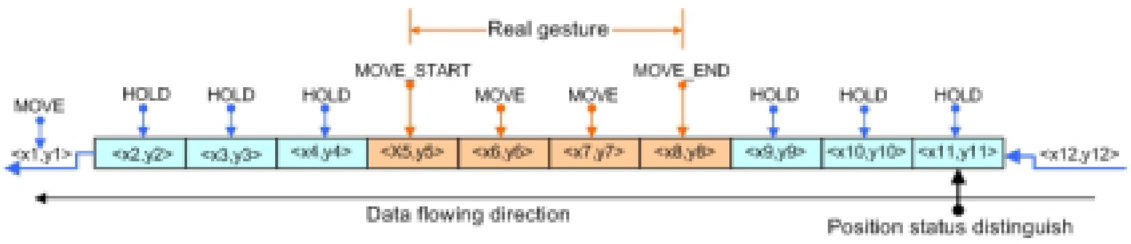


Figure 8.6: An example of analysis list with length of 11 positions (this is just a list example, the real list should have at least 50 positions).

It should be mentioned that, our approach can handle some worse situations, for example, if some frames in the sequence lost the screen corner positions (these positions status in the observation list will be marked as “MISSING”), within some tolerance, our approach still can get candidate gestures correctly. But if the number of “MISSING” positions exceeds the *MISSING_TOLERANCE* the initial position will be updated.

In our practical implementation, we set parameters: *MOVEMENT_SENSITIVITY*, *HOLDING_NUM_UNITTIME* and *MISSING_TOLERANCE* according to the possible situations in different applications.

Pseudo code of this list analyzing process is as following:

```
list_length=0;
list_begin=position(current);
do
{
  list_length++;
  base_position=position(current)
  list_current=list_current.next;
  if (distance(list_current)-distance(base_position)< MOVEMENT_SENSITIVITY)
  {
    list_current_status="HOLD";
    hold_num++;
    list_current=list_current.next;
  }
  else
  {
    list_current_status="MOVE";
    if(hold_num< HOLDING_NUM_UNITTIME)
    {
      do
      {
        list_previous=list_current.previous;
        list_previous_status="MOVE";
      } until (list_previous_status="MOVE")
    }
    else
    {
      hold_num=0;
      if (list_previous=="HOLD")
      {
        list_previous_status="MOVE_START";
      }
      else
      {
        do
        {
          list_previous=list_current.previous;
        } until (list_previous_status="MOVE")

        list_previous_next_status="MOVE_END";
        gesture_trace=trace("MOVE_START","MOVE_END");
        gesture.analysis(gestrue_trace);
      }
      base_position=position(current);
    }
  }
} until (list_length=N)

pop_up(list_begin);
push(list_end.next);
renew(list);
```

From our analysis process, the candidate gesture trace that we take must satisfy the following two conditions:

a) A candidate gesture must be made within a limited time range.

b) A candidate gesture must begin with a short pause and also end with a short pause.

In our final system, we consider three kinds of gestures ("Left-Right" gesture, "Right-Left" gesture and "Cross" gesture), which are designed for our UI-Wand system. Three gesture examples are shown in Fig.8.7.

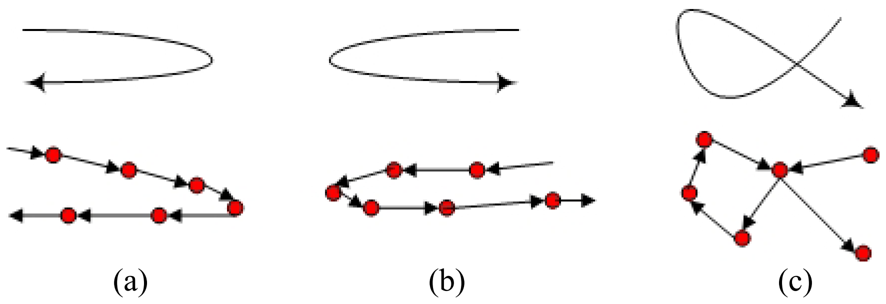


Figure 8.7: Three kinds of gestures. (a) Right-Left gesture. (b) Left-Right gesture. (c) Cross gesture.

8.2.3 Candidate trace interpolation

Now we get candidate gesture traces, but the distribution of the observations along these traces are very unbalanced, this can cause the same gestures to have very different observations lists. If we just extract features from these observations, we can image that the classification result may be bad. So we need better features from these observations. We use trace interpolation algorithm to resolve this problem, since the observations are lying very unbalanced along the trace, if we can re-arrange our observations balanced along the trace, then the same-labeled gesture trace should have much similar position relationship.

From our experience, we decide to locate 10 observations along each a candidate gesture trace. And then the trace interpolation procedure can be done following these steps:

1. Calculate the whole distance of the observations by plus together the distance between each neighbored observation positions.

2. Divide the whole by 10 , the we get a distance l for one step.

3. From the first observation of the trace, walk distance l along the same direction from this observation to the next observation. Here there are two conditions may occur: The first one is if the original distance l_{ori} between these two observations is longer than l , then we interpolate a new observation at the position along this direction and l distance away from first observation. The second one is if the original distance l_{ori} between these two observations is shorter than l, we still walk along the direction

and when we arrive at the second observation, we will continue walk along the direction from the second observation to the third observation and walk for $l - l_{ori}$ distance away from the second observation, then we can interpolate a new observation there.

4. Start from the first new observation position and walk along the direction from this new observation to the next old observation and walk distance and direction changed in the same way as in step 3.

5. Repeat step 3 and step 4 until get the end of observation position.

From the above steps, we can see that the route for finding a new observation is always along the old observations trace, but the observations are located in a more balanced way. The procedure is shown in Fig.8.8.

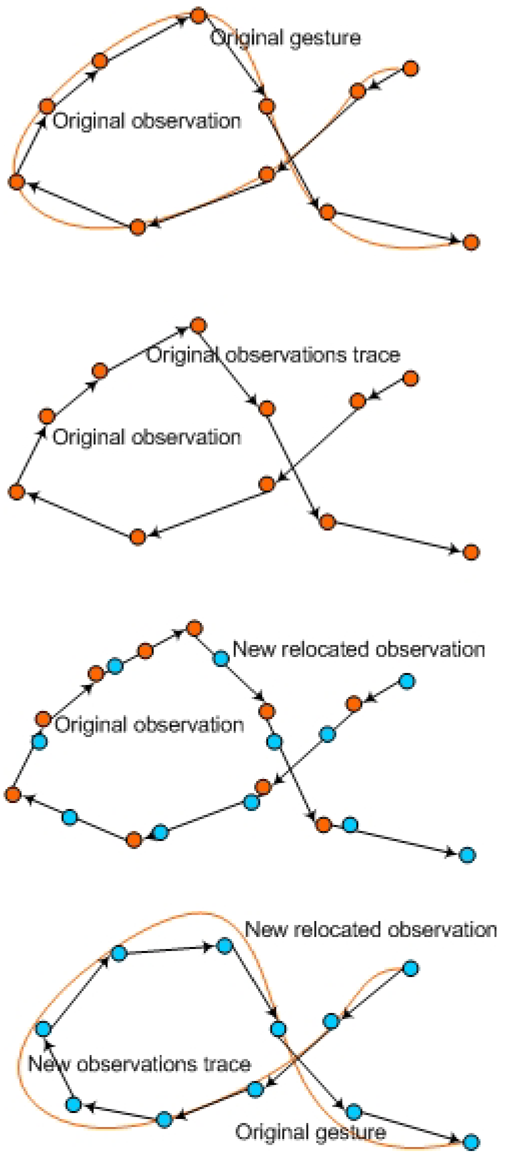


Figure 8.8: Trace interpolation process. Orange points are original observations and orange line is original gesture. The blue points are the interpolated observations.

8.2.4 Feature extraction

This is the crucial step of the gesture recognition task. The quality of the extracted feature directly affects the RVM classification result. So we comprehensively think about the feature selection. For the future real-time application, the feature must be very easy and fast to obtain, it also need to cover enough space and time information for discriminating among different gestures. Considering the above requirement, we choose the motion vector as our gesture features, which contain space information and also velocity information.

First we introduce the motion vector of the gesture trace. Motion vector is a very popular used feature, it is defined very simple, if we just consider a target moving in 2D space when it moves from position $A(x_i, y_i)$ to position $B(x_j, y_j)$, and its moving vector is $\left\langle x_j - x_i, y_j - y_i \right\rangle$ (see Fig. 8.9). By the trace interpolation step, we can get some candidate gesture traces as some position arrays. Then it is very easy to generate a motion vector array for each candidate gesture trace. After collecting all the motion vectors of a candidate gesture trace, the feature of this gesture trace is formed by a single vector, which contains 20 elements and the elements are arranged orderly, for example, assume the case that we get a candidate gesture trace as:

$<2,5><3,4><6,7><8,10><9,13><11,25><15,20><10,17><7,15><5,6>$

Then we can get a motion vector array for this candidate gesture as: $<1,-1><3,4><2,3><1,3><2,2><4,-5><-5,-3><-3,-2><-2,-9>$ and the corresponding feature vector for this gesture trace is $<1,-1,3,4,2,3,1,3,2,2,4,-5,-5,-3,-3,-2,-2,-9>$. Thus we use the same way to extract a feature vector for each selected candidate gesture trace. Finally we can collect these feature vectors as RVM training samples with marked label or we can send them directly to existing RVM for classification.

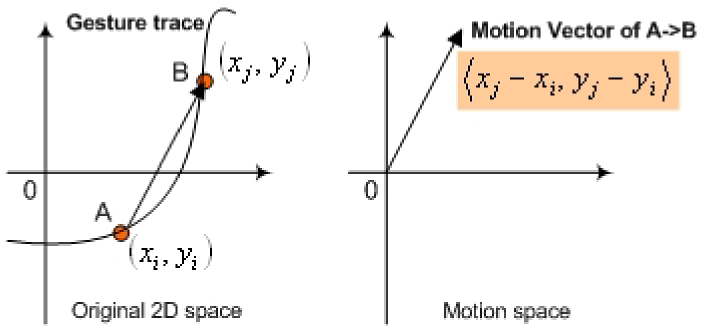


Figure 8.9: Motion vector generation example.

8.2.5 Training samples generation

After feature extraction, we can generate several training samples by extracting feature vectors for each candidate gesture trace. But until now we only can generate three classes training samples from three gesture labels. But to get more accurate gesture recognition rate, we still need to generate lots of non-gesture samples for finally getting a good RVM model that can classify four kinds of labeled gestures: “Left-Right” gesture, “Right-Left” gesture, “Cross” gesture and non-gesture.

Considering the attributes of the three kinds of gestures, we want to generate non-gestures, which cover as many as the movements that are different with those three gestures but can happen when using UI-Wand. One way for generating non-gesture is just to capture some non-gesture frame sequences and analyze them to extract feature vectors. But this way is time-consuming and may not cover some occasions. So we chose another way to get non-gestures. We divide the non-gestures into the following two categories:

Random straight line: this category contains the non-gestures, which are similar to a straight line but with a little direction change in every step (see Fig.8.10). It first chooses a direction as main direction from the eight directions and chooses a step length from three different lengths in Fig.8.10, and then the trace walks a step length along this direction but with a little random deflection. The next step is along the same main direction as before and still deflects a little bit randomly. After ten steps, which are the same with the number of interpolated candidate gesture observations, a non-gesture is generated. Because there are eight kinds of main directions and three different lengths, so after generation, this category contains twenty-four non-gestures.

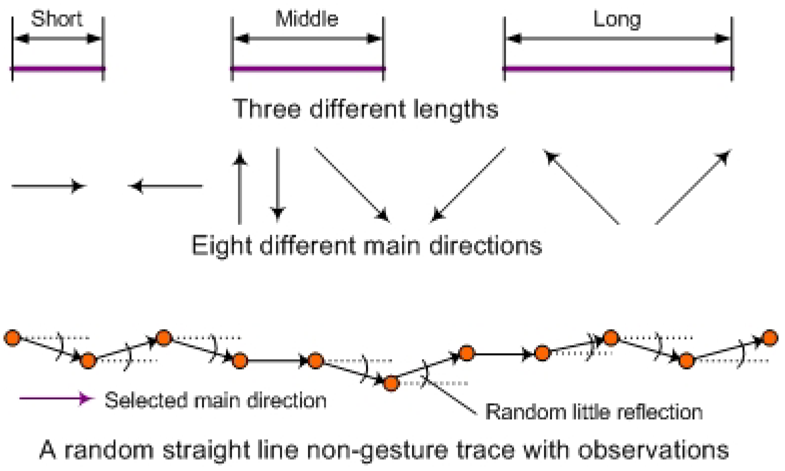

Short
Middle
Long
Three different lengths
Eight different main directions
Selected main direction
Random little reflection
A random straight line non-gesture trace with observations

Figure 8.10: Generation procedure of a random straight line non-gesture.

Random line: this category can also separate into two types. The first type generates such the following traces: it randomly gets a moving direction as main direction and walks along a changeable length along this direction but deflecting a random angle, which is much bigger than that in the random straight line (see Fig.8.11-a). Then it repeats ten times to generate a whole non-gesture trace. The second type generates every non-gestures step in a random direction and also with a little direction change in every step (see Fig.8.11-b). In every step, it chooses a random move direction, and then the trace walks this step length along this direction and with a little random deflection. The following steps are the same as the first one. After ten steps, a non-gesture is generated.

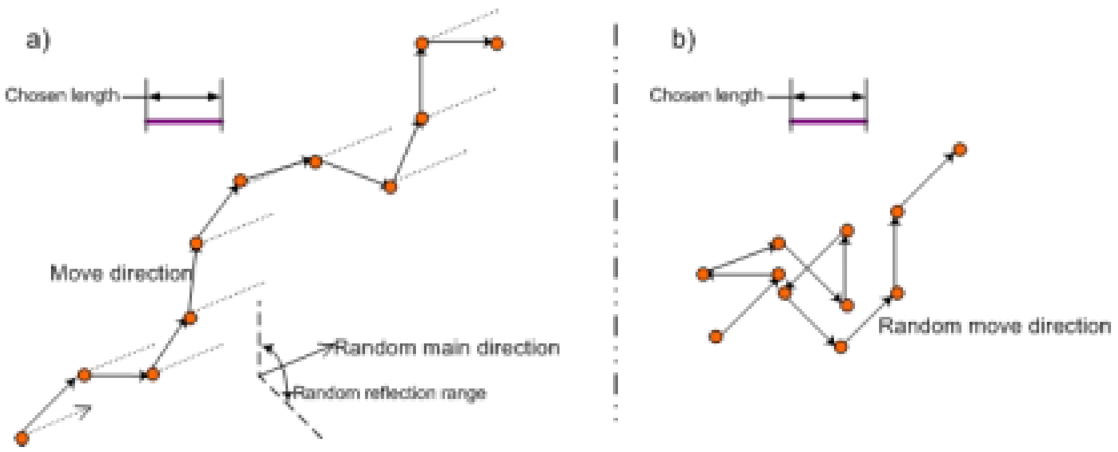


Figure 8.11: Generation procedure of random line non-gesture.

8.2.6 RVM training

Based on the previous work, we already can get the training samples from the previous process. We need to use these training samples to train our RVM gesture recognition model. We still use the same RVM model as described in Chapter 5, but for adaptive reason, we changed the kernel function of it. We choose a new kernel function for RVM, a polynomial function shown as follows:

$$K(\mathbf{x}_{\mathbf{m}},\mathbf{x}_{\mathbf{n}}) = (\mathbf{x}_{\mathbf{m}}\mathbf{x}_{\mathbf{n}} + 1)^{r} \quad (8.9)$$

where $\mathbf{x}_{\mathbf{m}}$ and $\mathbf{x}_{\mathbf{n}}$ are the sample feature vectors, r is the length parameter.

8.2.7 RVM classification

When the new gesture image sequence arrives, in the same way as we described before (observation getting—list analysis for getting available candidate gesture trace—trace interpolation—feature extraction), we can easily abstract candidate gesture samples from it. After the training process, we can use the trained RVM classifier to recognize which kind of gesture it should belong to, specify the class label and give a class confidence for each gesture sample.

8.2.8 Gestures recognition test

After modeling the gesture recognition procedure by RVM, we want to test how well it works. The test step includes two steps. The first one is to collect gestures samples and train the RVM model and then the second step is to use new gestures to test. This gesture test is only on the Black LCD screen.

In gestures samples collection step, as described in the model above, in the training process, in order to give correct gestures label to the gestures the user has to do one gesture in a sequence. In this sequence, we select candidate gestures as gesture samples in the current gesture class. The user should always do one type of gesture until we collect enough samples. So in our samples collection step, we took many sequences of frames doing the same gestures and then run our system to analyze the trace in these sequences and then save the gestures samples and their feature vectors by using trace interpolation algorithm and motion vector extraction algorithm.

Finally, we got 10 gestures samples for each gesture class. To the non-gestures samples collection, we run the automatic generation algorithm and got 60 different samples. The details of these algorithms have already been introduced in former sections. Some of the samples are shown in Fig. 8.12. Please note that the gesture shown in Fig. 8.12 is UI-Wand moving traces that is inverse with user's movement. That is why Fig. 8.12a is a Left-Right gesture and Fig.8.12.b is a Right-Left gesture.

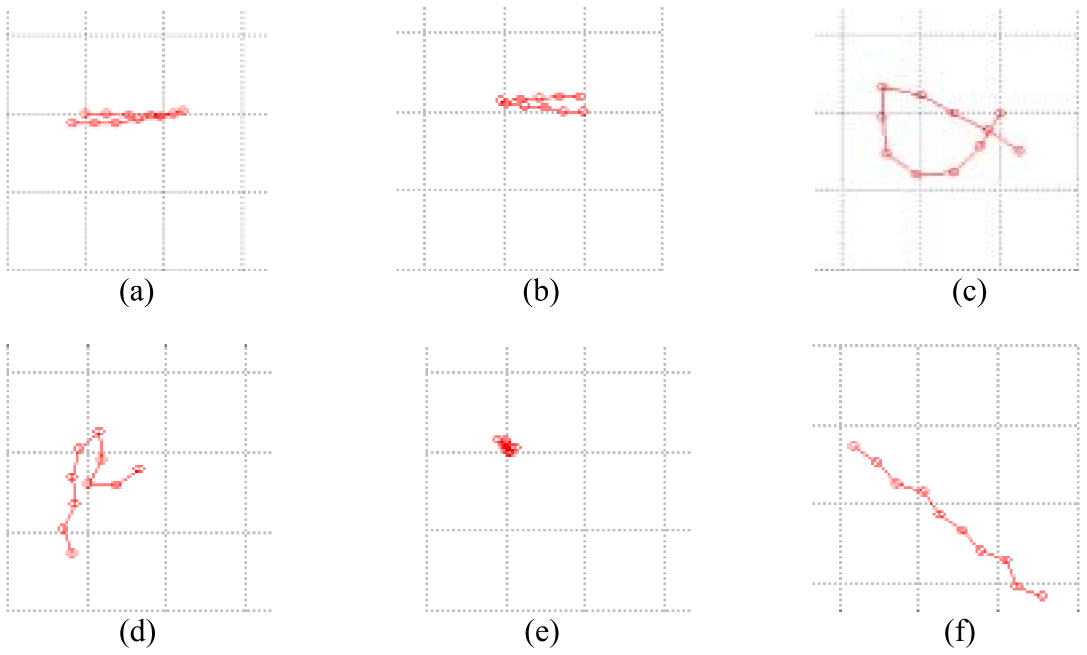


Figure 8.12: Gestures samples extracted from some sequences of frames. (a) One sample of Left-Right gestures. (b) One sample of Right-Left gestures. (c) One sample of cross gestures. (d)(e)(f) Three samples of non-gestures.

After training RVM model we got the final weights file, by which we can start to classify. In order to do the test, we took two sequences of frames. Each of them contains 600 frames and contains many gestures and non-gestures movements. After feeding these two sequences to the UI-Wand system, finally we got the test results in Table 8.1.

Table 8.1: Test results of gesture recognition on two sequences.

	Left-Right	**Right-Left**	**Cross**	**Non**	**Accuracy**
Sequence 1	2/2	3/3	3/1	7/6	80.0%
Sequence 2	2/2	1/0	3/1	9/9	73.3%

The value with format "x/y" in this table means that there are x gestures in the sequence and y of them are recognized correctly. This result does not show very high classification rates. But after checking the results we found an interesting point. In all misclassification cases in effective gestures (Left-Right, Right-Left, Cross gestures), they were misclassified to non-gesture class, which means that effective gestures cannot be misclassified with each other. This point is very important, because only misclassify gestures to non-gestures will not lead users to do a wrong action, which is the worst result for gesture recognition. Moreover, this kind of misclassification can be improved by reselection the dataset samples. In this gesture test, the dataset that we are using contains 60 non-gestures samples, which may be too much comparing to other classes. In addition, since these non-gestures have been generated automatically, it is very possible that some of them are similar to some samples in effective gesture classes, which will result in the inaccuracy of the classification.

8.2.9 Conclusions

Gesture recognition is a challenging task in the human-computer interaction field. There are lots of existing approaches already, the most popular one is the HMM model approach, it can perform very good classification accuracy and can be used in real-time applications, but the drawback of it is also clear. It is still not easy for people to construct a HMM chain for each gesture.

For our task, the previous experiments imply that RVM has a very excellent ability for classification, which supplies more advantages comparing to other approaches. This enlightens us to consider about using our existing RVM model to realize gesture recognition task. The testing result shows that it is a very promising approach for future research.

Part III

The System Implementation Design

9 UI-Wand System Design

9.1 The existing framework introduction

In Part II, we described all models and algorithms suitable for our problems. Now we need to implement and integrate them so that the prototype UI-Wand system can run properly. Before the implementation, the first step is to design the whole system. In PHILIPS, there is an existing C++ software framework for previous projects on computer vision, which is called Visipirin. Visipirin supplies many packages containing a lot of basic and utility classes, which stipulates the structure to new applications. So all our design and implementation works are based on Visipirin.

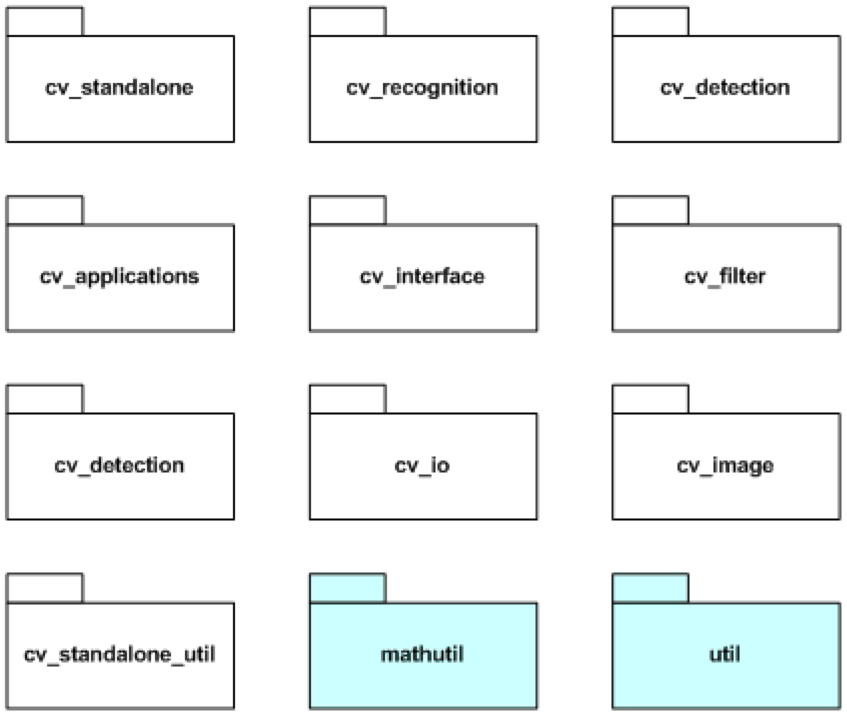


Figure 9.1: Existing C++ packages of pervious projects.

Fig. 9.1 shows the existing packages of Visipirin. The white ones are the packages developed for the projects on computer vision, and the cyan ones are the utility packages, which are developed for all projects.

There are two advantages of this framework that we can describe as follows:

1. Visipirin defined a set of base classes and utility classes suitable for image and video processing. With these classes we can directly go to do the real work for our special applications.

2. Visipirin defined an algorithm class, based on which all of special algorithms are constructed, which gives a very flexible way to combine various algorithms that might be used in the final applications and change their parameters by using parameters file. So in the final system, if we want to try other models or algorithms in the application, instead of modifying the code and compiling the entire system, we just need to revise the application parameters files, which will make a recombination of the algorithms in the application. This advantage is very important for the project during the research period, because this makes much easier for us to test algorithms individually or embed new algorithms into the system.

9.2 UML models introduction

It is difficult somehow for us to design our system based on Visipirin, since there is no documentation about this framework. So before starting the design of our special system, we have to figure out the current framework's principle and then follow its features to construct our system. Because Visipirin is a C++ object-oriented project, the suitable design mechanism is the Unified Modeling Language, also known as UML.

The main advantage of UML is that it can design the object-oriented software on different levels of abstractions and consideration. It can design very high-level software structure and also detailed individual class by using different diagrams. In our system, we reconstruct Visipirin and design our UI-Wand system by using UML models. The main UML diagrams we used to design the system are listed as follows:

1. Class diagrams that show the structure and relationship between classes used in one application.

2. Individual class diagrams that show the main attributes and member functions in one class of the interface. Moreover, above the individual classes, we use some text to explain the functionality of the class, which will be helpful to readers to understand.

3. Sequence diagrams that show the message exchange between the objects in one application, which will let users easily figure out the running procedure in one application.

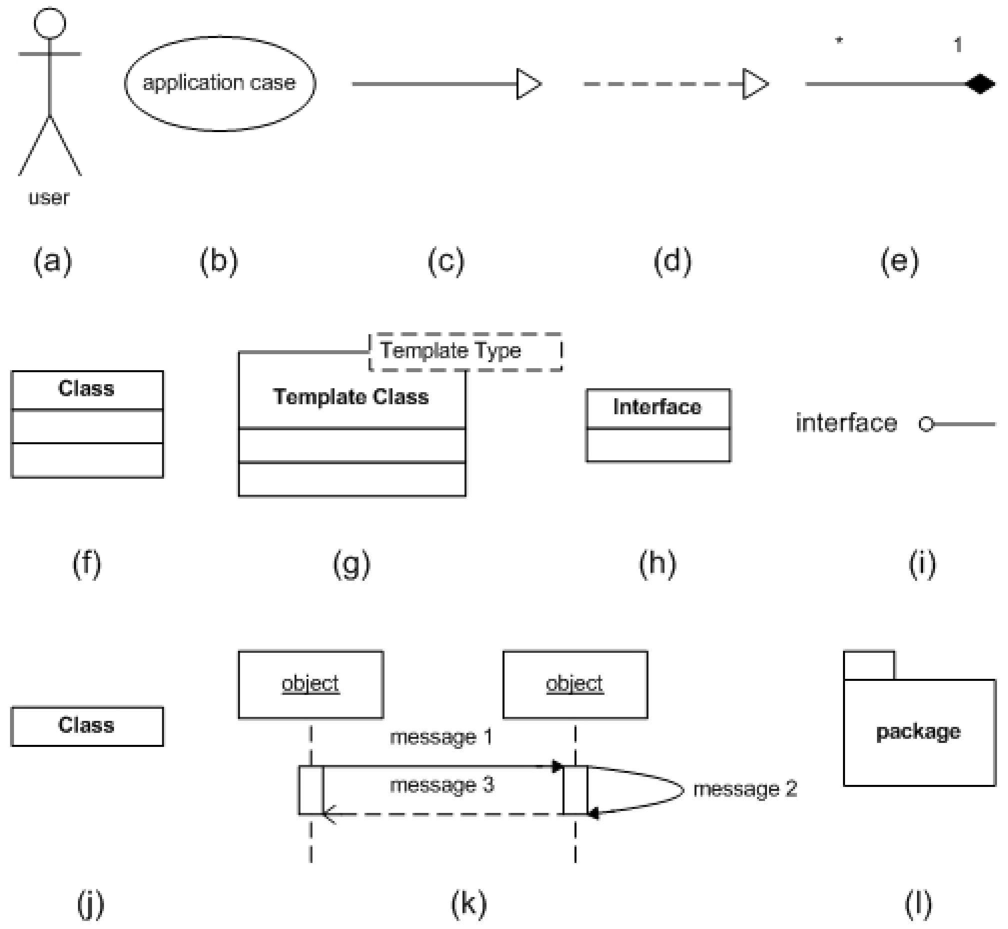


Figure 9.2: UML elements used in our UML diagrams.

Although most of elements and diagrams in UML are standard, sometimes some symbols look different in different tools. In addition, we use some special elements in our UML models to express the concept more simple and clear. Therefore in order to make it clearer, we explain briefly here the elements we used in UML diagrams. Fig. 9.2 shows the elements we used in our UML models design.

a) Represents the user of the system who can use the application in the system.

b) Represents the application that is implemented in the system for users to use.

c) Represents a derivation relationship. It connects two classes. The class in head side is the abstract class of the class in tail side.

d) Represents a derivation relationship. It connects with many classes, the classes pointed by the head of the arrow is the abstract class of every class place in the line segment. This is a symbol we define in order to simplify the class diagram.

e) Represents a combination relationship. It connects with two classes. The class in its tail side is a component of class in the head side.

f) Represents a class indicating its attributes and member functions.

g) Represents a template class, which is often used in our system. The template classes are designed to handle different type of images, i.e. color ones or gray ones.

h) Represents an interface showing its member functions.

i) Represents an interface without showing anything.

j) Represents a class without indicating its attributes and member functions.

k) An example of sequence diagram. The object box is the object used in an application, the message 1 is an invocation from one object to another one, the message 2 is an invocation from one object to itself and the message 3 is the return message.

l) Represents a package, which contains the classes handling the same problem.

9.3 UML models of Visipirin

Now we start to describe Visipirin by UML. If we use Use-Case diagram to express the framework, it will be a very simple diagram (see Fig. 9.3). The use case in the framework is just to let a user run an application, which clearly indicates the framework purpose. After this simple figure, now we start to extend the class diagram of this framework. The main classes of the framework are A*pplication* class and *Algorithm* interface, their relationship is described in Fig. 9.4. From this figure we can clearly see that all algorithms in the system are derived from the abstract interface, *Algorithm*, and *Application* base class can include as many as possible algorithm classes that are implementation to the various specific algorithms interface. For example, in Fig. 9.6, *FileInputPPM* is an implementation class of *InputAlgorithm* interface, which is a special algorithm interface derived from *Algorithm* interface.

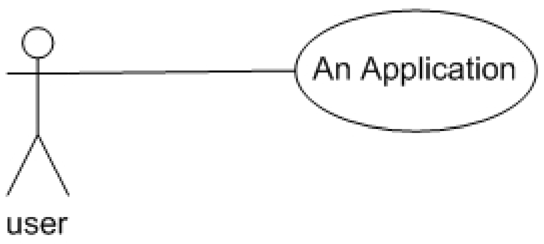


Figure 9.3: The simple use case diagram for existing framework.

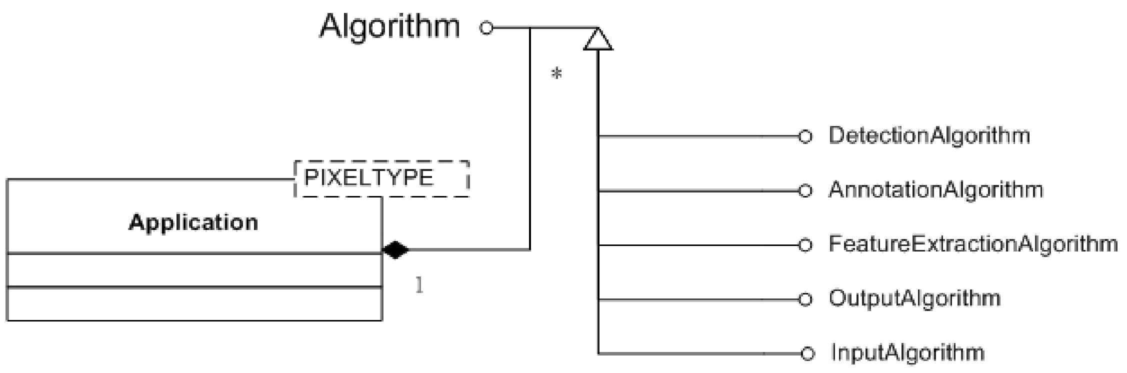


Figure 9.4: Application and Algorithm Class diagram of Visipirin.

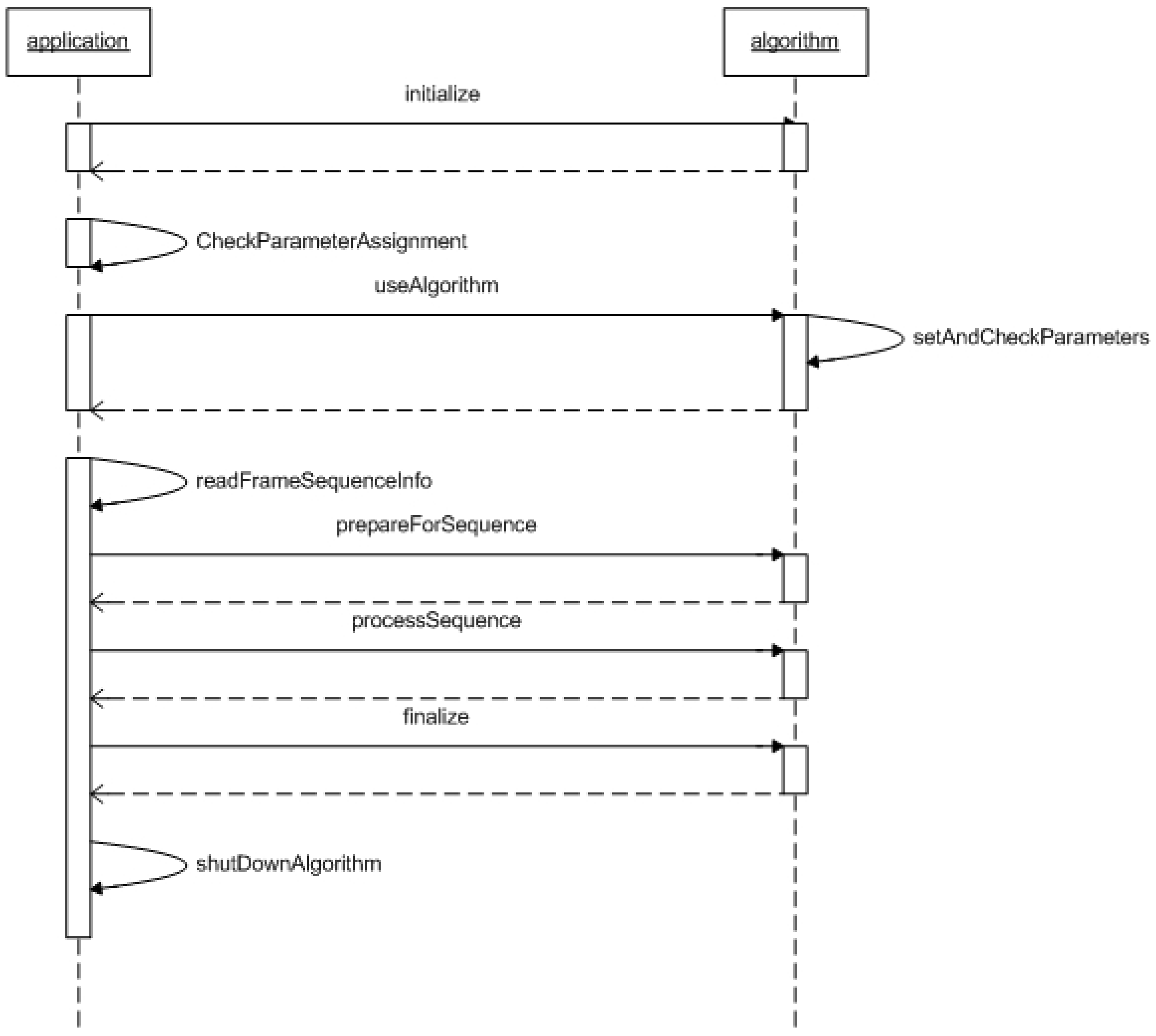


Figure 9.5: Application and Algorithm sequence diagram of Visipirin.

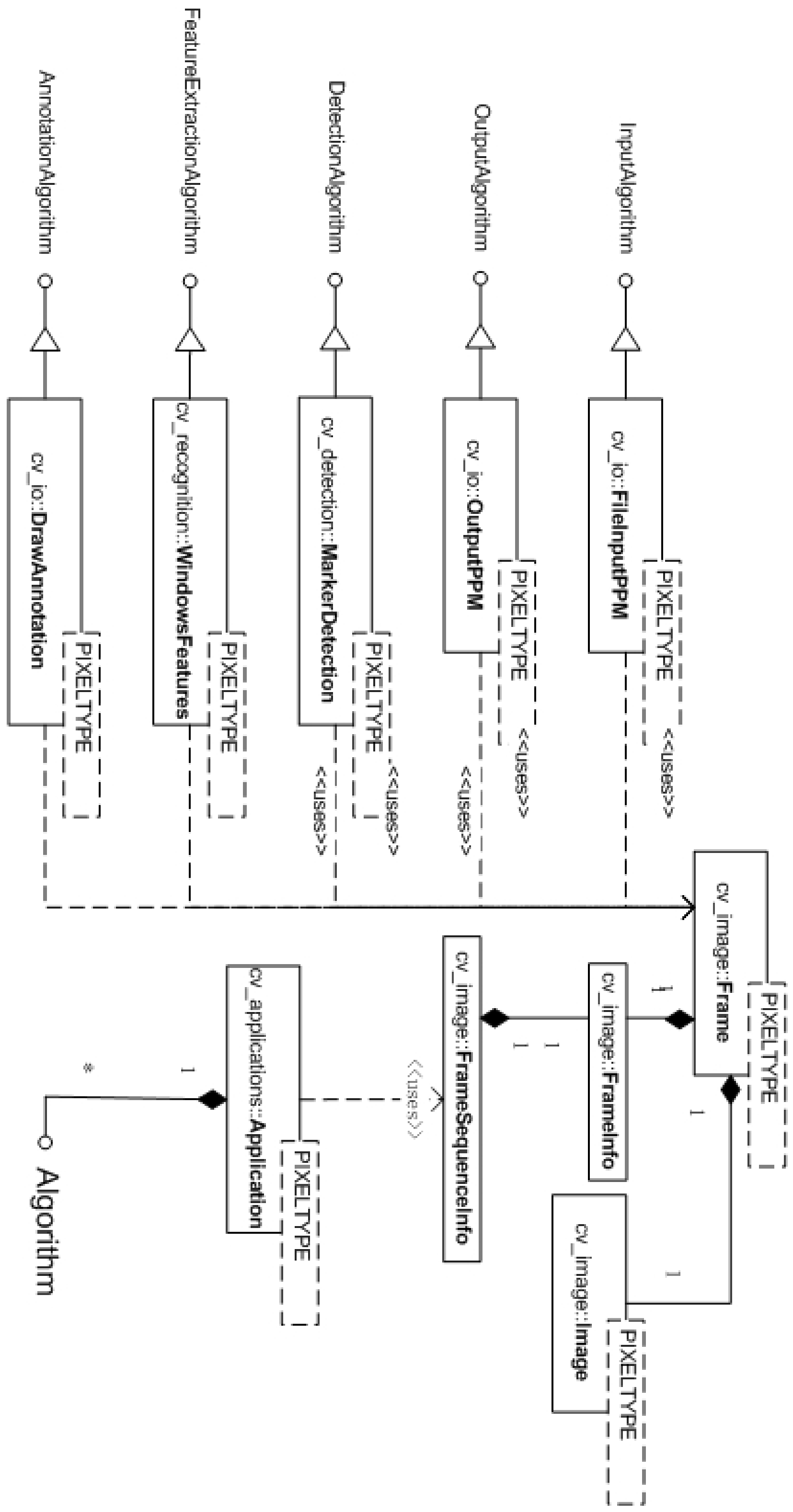


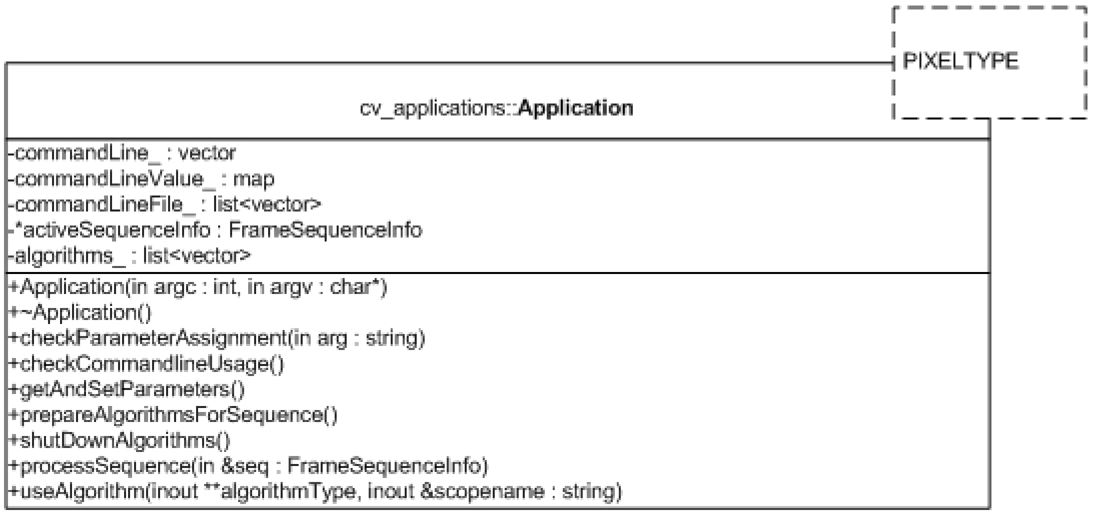
Figure 9.6: Class diagram of Visipirin.

Fig. 9.6 shows some important classes used in Visipirin. Because the special algorithm interface and implementation class are also used in our system, so we will introduce them together with our system later. Here, we just introduce more about *Application* class, *Algorithm* class and the base classes that are common use, so that you can really get the idea of this framework and know its advantage.

9.3.1 Individual class description

Application Class

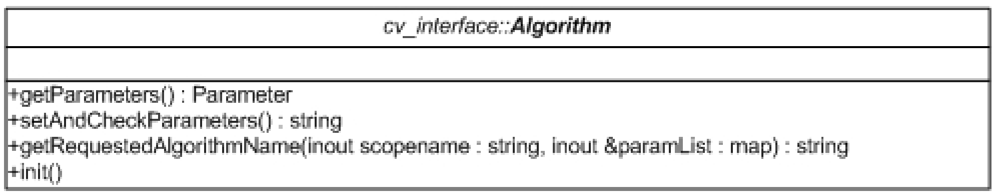
Application class is a base class. Every special application class need to be derived from this class. The basic functionality of this class is to read image sequences from files or from devices and initialize special algorithms indicated from command lines or a parameter file.



```
PIXELTYPE
cv_applications::Application
-commandLine_ : vector
-commandLineValue_ : map
-commandLineFile_ : list<vector>
-*activeSequenceInfo : FrameSequenceInfo
-algorithms_ : list<vector>
+Application(in argc : int, in argv : char*)
+~Application()
+checkParameterAssignment(in arg : string)
+checkCommandlineUsage()
+getAndSetParameters()
+prepareAlgorithmsForSequence()
+shutDownAlgorithms()
+processSequence(in &seq : FrameSequenceInfo)
+useAlgorithm(inout **algorithmType, inout &scopename : string)
```

Algorithm Interface

Algorithm interface defines a common interface for all special algorithms. All of algorithms need to use in an application have to be implemented or derived from this interface.

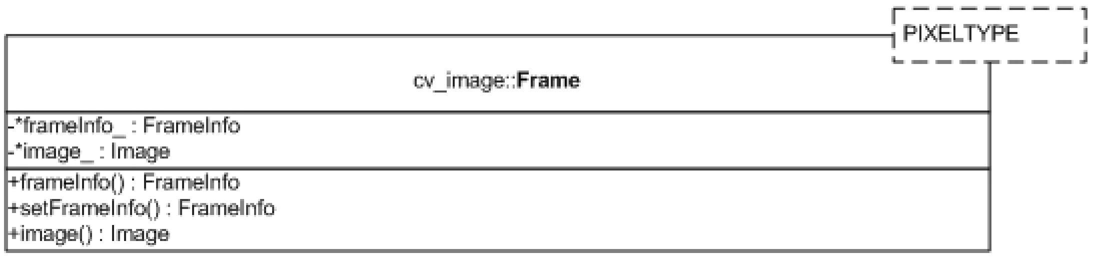


```
cv_interface::Algorithm

+getParameters() : Parameter
+setAndCheckParameters() : string
+getRequestedAlgorithmName(inout scopename : string, inout &paramList : map) : string
+init()
```

Frame Class

This class is very simple. It just combines *Frameinfo* class and *Image* class together.



```
PIXELTYPE
cv_image::Frame
-*frameInfo_ : FrameInfo
-*image_ : Image
+frameInfo() : FrameInfo
+setFrameInfo() : FrameInfo
+image() : Image
```

FrameInfo Class
This class supplies the information about one frame, which includes the timestamp of a frame, the count of a frame in a sequence. These information will be useful for the special algorithm class to operate with a frame.

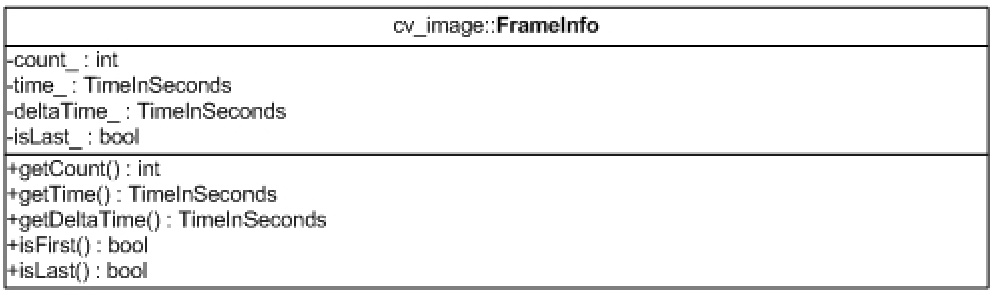
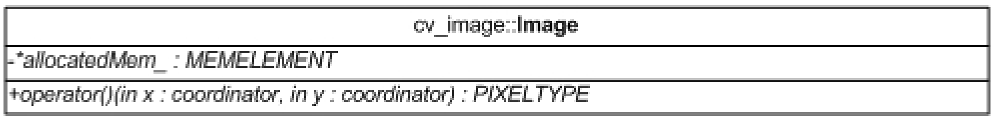
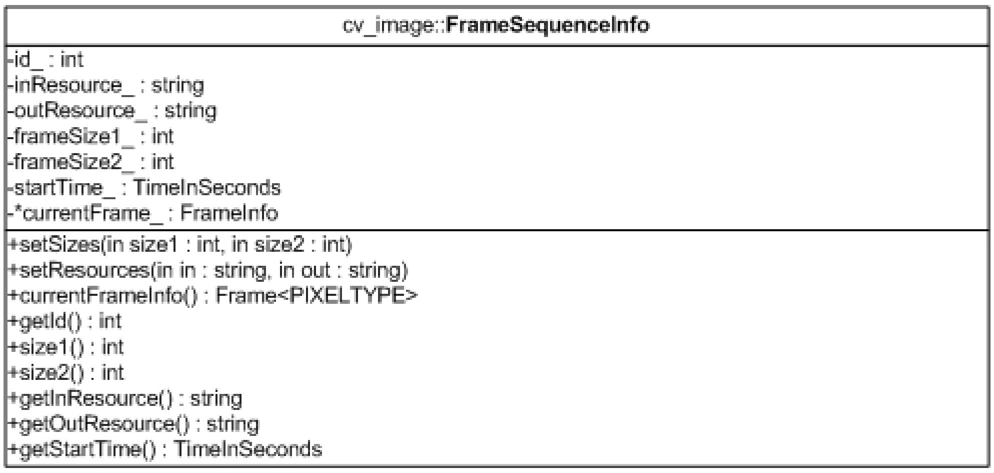

```
cv_image::FrameInfo
-count_ : int
-time_ : TimeInSeconds
-deltaTime_ : TimeInSeconds
-isLast_ : bool
+getCount() : int
+getTime() : TimeInSeconds
+getDeltaTime() : TimeInSeconds
+isFirst() : bool
+isLast() : bool
```

Image Class
Image class an important class. The functionality is to store an image into the memory in order to let other classes operate. Many attributes and member functions of this class comes from its parent class *MatrixAllocator* which is in charge for memory allocation. But since we did not use *MatrixAllocator* class directly we will not give detailed description of it. In *Image* class, we just show one attribute and one member function, which are enough information for your understanding to its function. The *allocatiedMem_* is a pointer to point a block of memory and *operator(x,y)* can access one pixel value on the image.


```
cv_image::Image
-*allocatedMem_ : MEMELEMENT
+operator()(in x : coordinator, in y : coordinator) : PIXELTYPE
```

FrameSequenceInfo Class
In this class information on the frame sequence level are collected. Those are frame sizes, number of frames, name of the sequence, etc. This class is used more often than *Frame* class.


```
cv_image::FrameSequenceInfo
-id_ : int
-inResource_ : string
-outResource_ : string
-frameSize1_ : int
-frameSize2_ : int
-startTime_ : TimeInSeconds
-*currentFrame_ : FrameInfo
+setSizes(in size1 : int, in size2 : int)
+setResources(in in : string, in out : string)
+currentFrameInfo() : Frame<PIXELTYPE>
+getId() : int
+size1() : int
+size2() : int
+getInResource() : string
+getOutResource() : string
+getStartTime() : TimeInSeconds
```

9.3.2 Applications execution

A compiled application class will run like what the Fig. 9.5 shows. The parameters that this application need to have can be inputted from command line or from a parameters file formatted as follows. The first part of which indicate the location of the application to run, the second part is the parameters or parameters file created, and the last part is a sequence of frames to feed into the application to handle.

The command line format:

(a) *applicationname parameters ./theframesyouwanttorun/**

(b) *applicationname parameters.par ./theframesyouwanttorun/**

The parameters have the following format:

(a) *Scopename=AlgorithmName*

(b) *Scopename::ParameterName=ParameterValue*

The *Scopename* indicates the domain within which the parameters will be used. Every different S*copename* can has one line in format (a) which indicates the algorithm that will be used for this domain, and several lines in format (b) which indicates the special parameters needed to run the algorithm. A parameters file example is shown in Table 9.1. This example parameters file tells the application to run an algorithm to do features extraction operation. The algorithm's name is indicated as *WindowedDCT,* so that this application will use the windowed DCT feature extraction algorithm, all needed parameters of which are given after the first effective line.

Table 9.1: Parameters file for feature extraction.

```
# A parameters file for windowed DCT feature extraction application
features=WindowedDCT
features::window_size_x=20
features::window_size_y=20
features::num_steps_x=1
features::num_steps_y=1
features::normalize_mean=false
features::normalize_stddev=false
features::dct_min_modes=0
features::dct_max_modes=3
features::x_feature=false
features::y_feature=false
features::eol_marker=false
```

9.4 UI-Wand system introduction

Based on Visipirin presented above, now we start designing our system. The first step is to define our use cases. The main use case in the UI-Wand system is UI-Wand application, by which the system can realize all the function required such as screen corner detection, pointing positioning, and gesture recognition. But UI-Wand application is not the only case in our system. To run the final application, UI-Wand have to get screen corners samples and train RVM so that it can classify the corners, and the same with gestures recognition, the UI-Wand must get gestures samples and corresponding RVM models to do the classification. All of these tasks are application cases in our system. So in the end, we design six cases (applications) in our final UI-Wand prototype system (see Fig. 9.7). They are Synthetic Dataset Generation Application, Adaptive Dataset Collection Application, Gestures Dataset Collection Application, Screen Corners Training Application, Gestures Training Application, and UI-Wand application. The detailed description about these application cases will be given respectively in the later sections of this chapter. In the same way as we used in before, this section we will still use UML models to design. The model diagrams we used are use case diagram, classes diagram, sequence diagram, detailed individual class diagram and text to explain the functionality and principle of them.

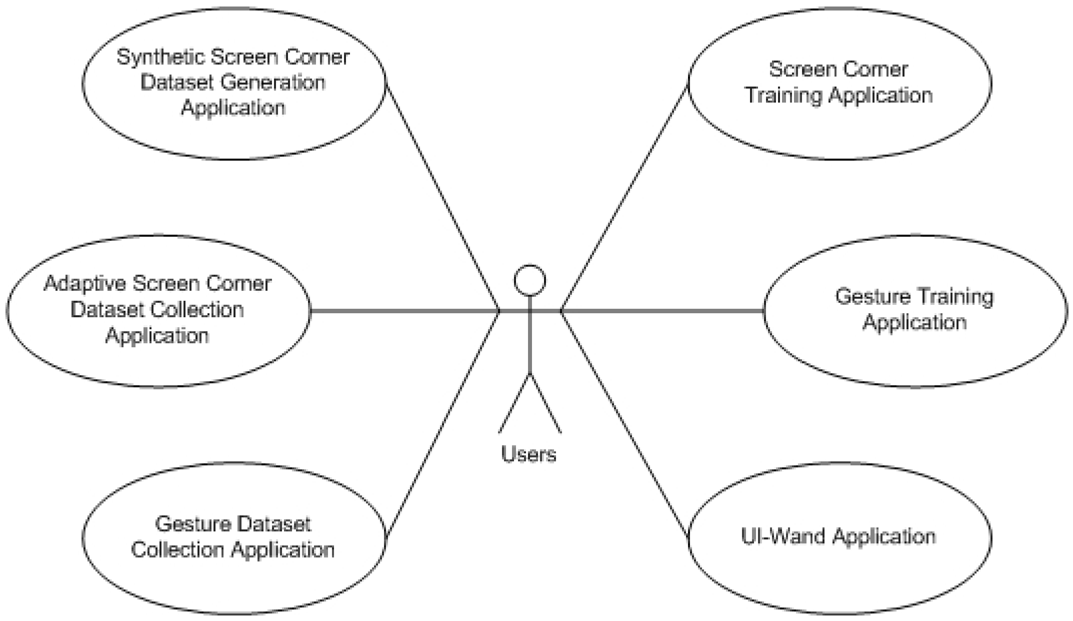

Synthetic Screen Corner Dataset Generation Application
Adaptive Screen Corner Dataset Collection Application
Gesture Dataset Collection Application
Users
Screen Corner Training Application
Gesture Training Application
UI-Wand Application

Figure 9.7: UI-Wand system use cases UML.

9.5 Synthetic dataset generation application

We used structure programming to generate synthetic samples for our RVM classifier. This program is separated from our UI-Wand system. Fig. 9.8 shows the program flow chart. The parameters of this program as follows:

- save_dir: Generation directory. It is the directory given by the user for saving the generated samples. When this directory is given, the program will automatically generate four sub-directories, which are lefttop corner directory, righttop corner directory, leftbottom corner directory and rightbottom corner directory. When the program is running, the generated corners will be saved into this directory corresponding to their class name.

- base_outer_color: Outer border color of the screen. The user specifies this color value. The valid range of this value is between 50 and 205, which is considered as the most popular color distribution range of the normal screen border.

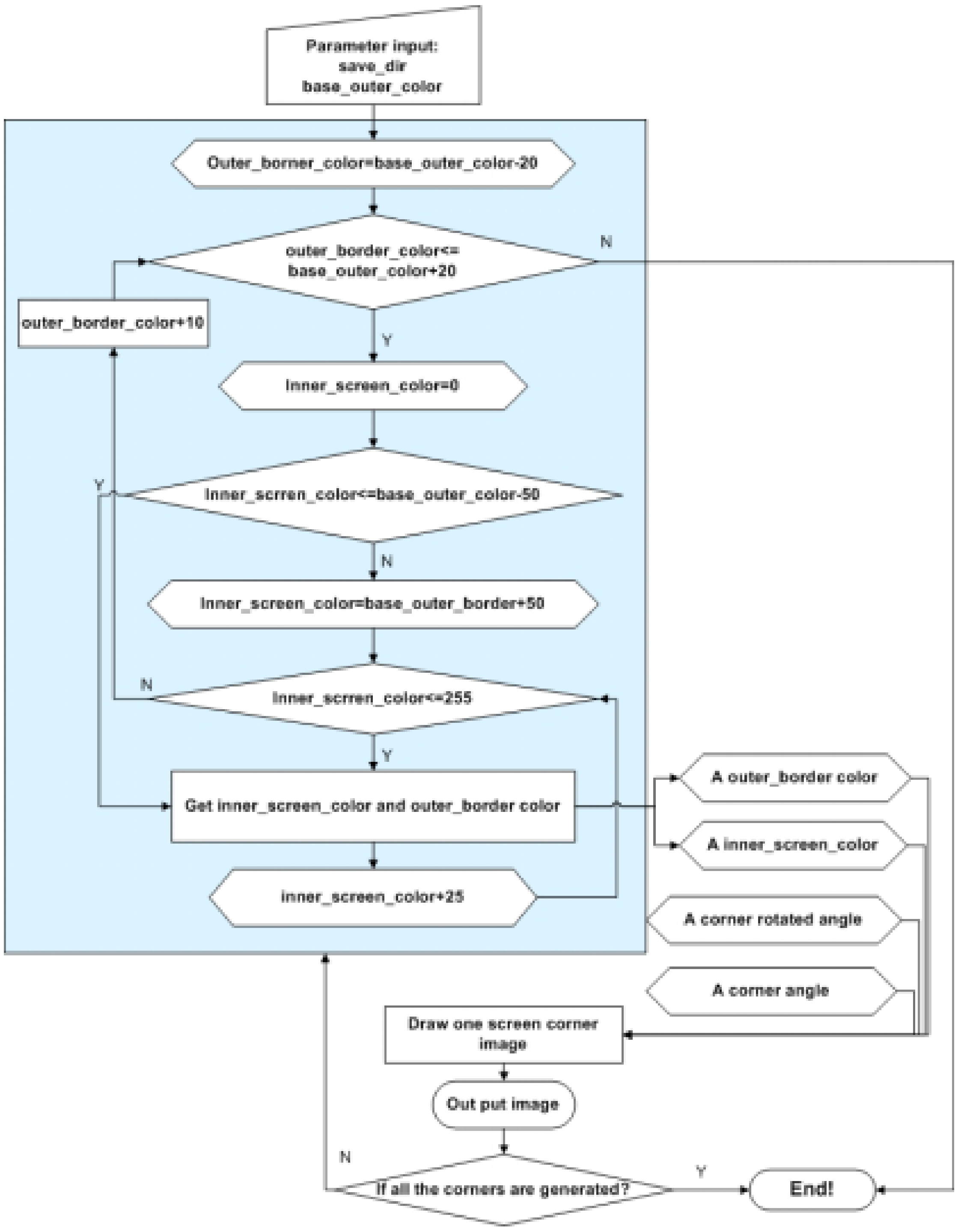


Figure 9.8: Synthetic screen corners dataset generation application flow chart.

9.6 Screen corners training application

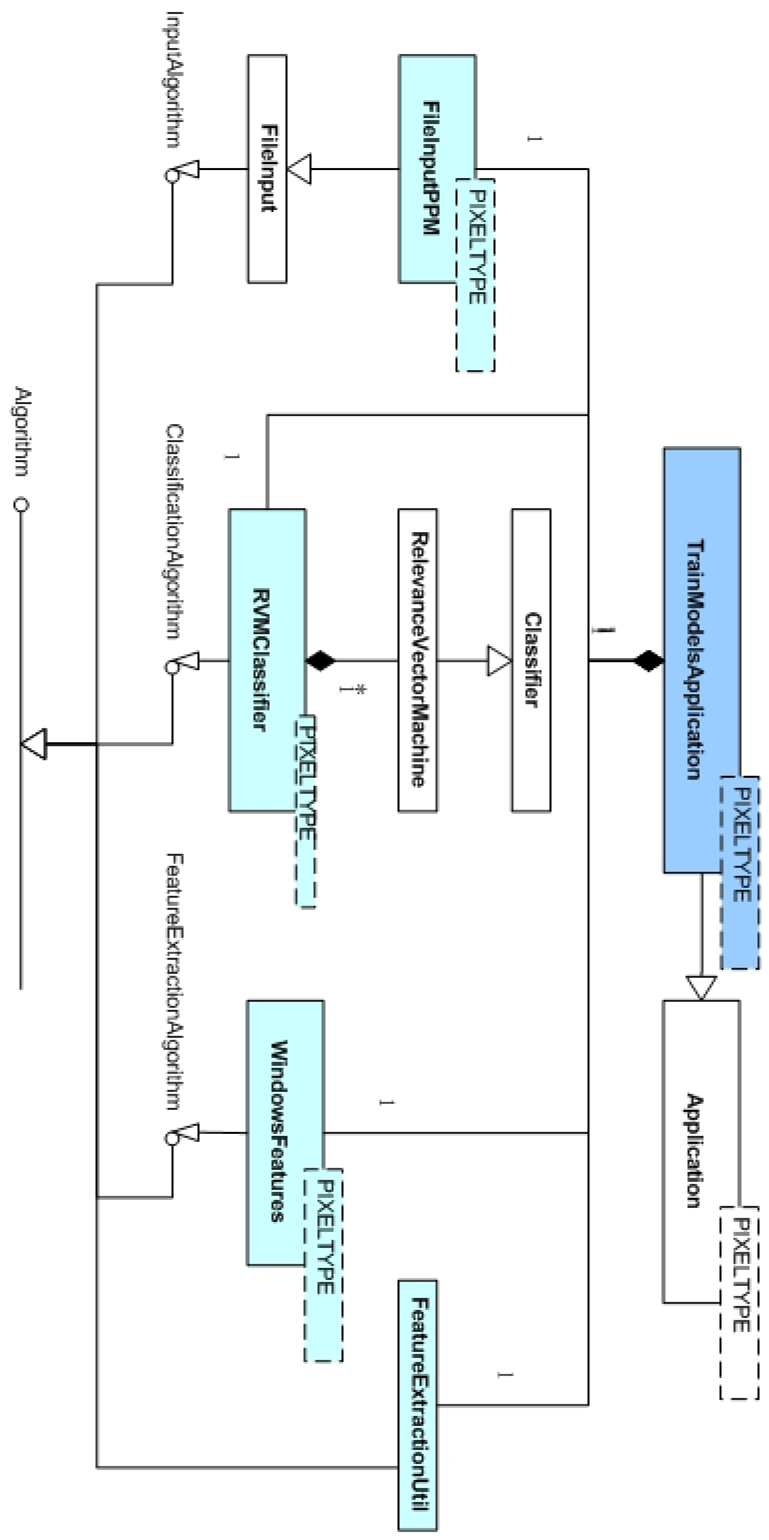


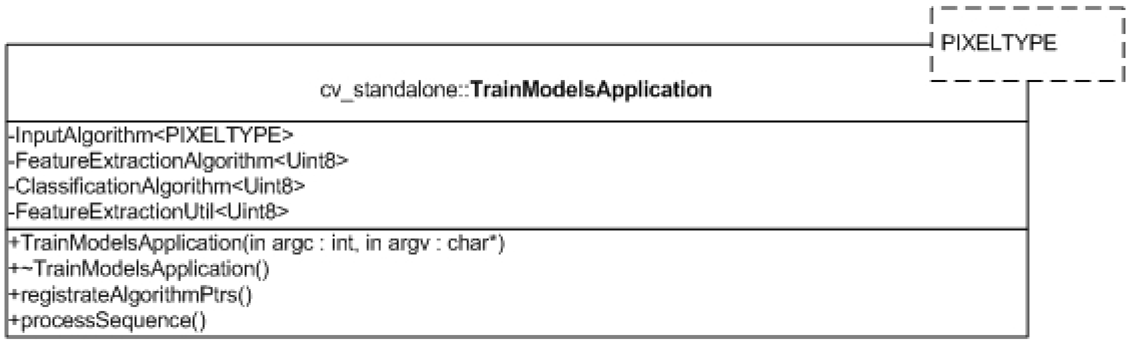
Figure 9.9: Class diagram in screen corners training application case. Classes in cyan are the algorithms used in *TrainModelsApplication* class shown in blue.

Screen corners training application is an application that can read a set of screen corners samples saved as files on the disk and extract their feature vectors and then use these vectors to train RVM models. This is the compulsory application before running final UI-Wand application, because it will supply the weights information to the final classification RVM models used in screen corner detection step. In Fig. 9.9, we can see that the *TrainModelsApplication* uses four different algorithms to finish this task. In order to give clear explanation about the functionality of different classes and algorithms, we use the following method to describe. Firstly, like what we did in section 9.3 we will give a text description about its functionality, and then we will give a detailed individual class diagram to show its important attributes and member functions. For the algorithm class and application class, we will give a table to describe the parameters they accept which control their running status for particular cases.

9.6.1 Individual class description

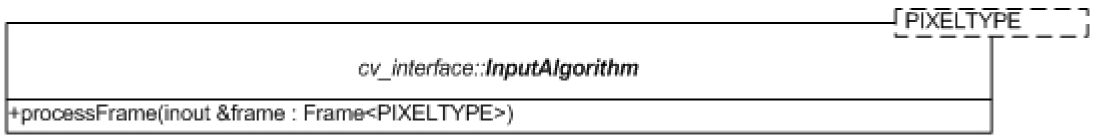
TrainModelsApplication Class

This is the main application class that integrates all algorithms needed for training classification models to let it classify screen corners.


PIXELTYPE
cv_standalone::TrainModelsApplication
-InputAlgorithm<PIXELTYPE>
-FeatureExtractionAlgorithm<Uint8>
-ClassificationAlgorithm<Uint8>
-FeatureExtractionUtil<Uint8>
+TrainModelsApplication(in argc : int, in argv : char*)
+~TrainModelsApplication()
+registrateAlgorithmPtrs()
+processSequence()

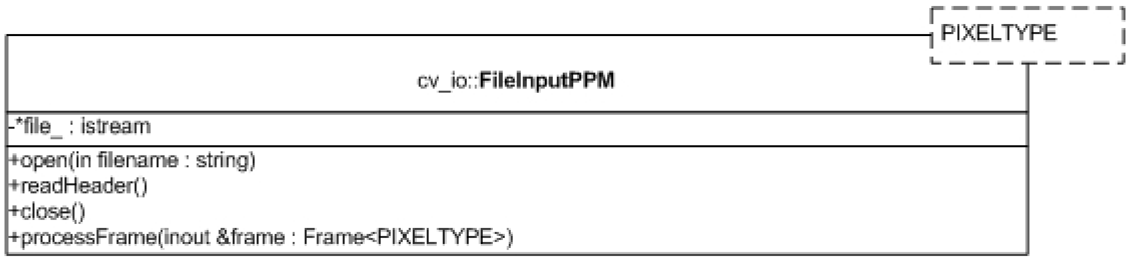
InputAlgorithm Interface

This interface supplies interface function *processFrame* which obtains a non-constant reference to an *Image* to which it will write all pixels of the current frame. The special implementation of this function will be finished in other algorithm implementation classes.


PIXELTYPE
cv_interface::InputAlgorithm
+processFrame(inout &frame : Frame<PIXELTYPE>)

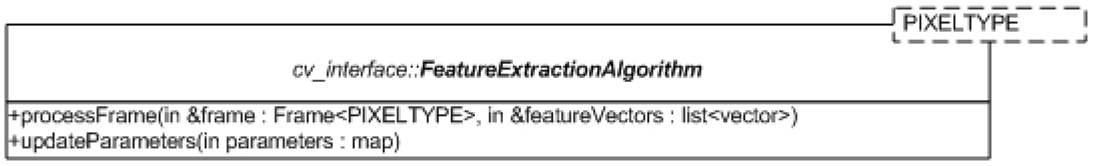
FileInputPPM Class

The functionality of this class is to read a .ppm file, ascII or binary format, from disk and save it into *Frame* class, so that the other algorithm can easily use it.


PIXELTYPE
cv_io::FileInputPPM
-*file_ : istream
+open(in filename : string)
+readHeader()
+close()
+processFrame(inout &frame : Frame<PIXELTYPE>)

FeatureExtractionAlgorithm Interface

This interface supplies two functions for special feature extraction algorithm. *ProcessFrame* is the main function, which will extract the features of a frame then save them into a list. The *updateParameters* is a simple function for updating the parameters used in the special algorithm. In our *AnnotationFeatures* class, we update tilted image angle.

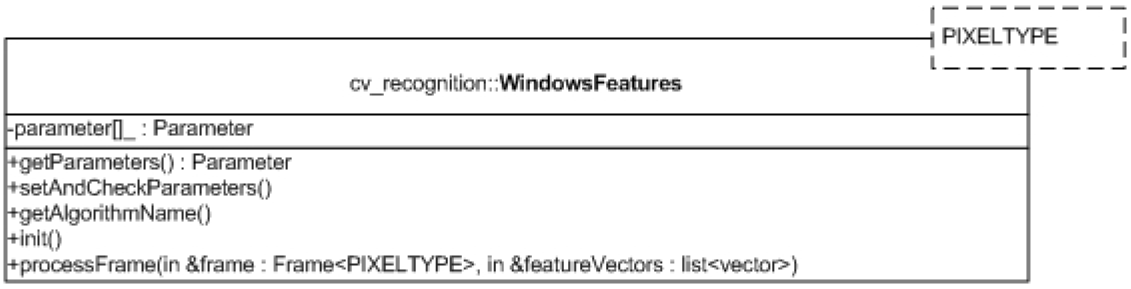

PIXELTYPE
cv_interface::FeatureExtractionAlgorithm
+processFrame(in &frame : Frame<PIXELTYPE>, in &featureVectors : list<vector>)
+updateParameters(in parameters : map)

WindowsFeatures Class

The functionality of this class is to do Discrete Cosine Transform to a sequence of sub-images of an image. The DCT coefficients after transformation will be stored into a vector list, which is used as the feature vectors for this sequence of sub-images. The sub-images selection is controlled by its parameters. Essentially, the parameters will fix a target-window size by defining its width and height, and the algorithm will use this target-window to scan the entire image, by fixing the move step size and the number of move step of the target window. The special main parameters it uses are listed below:

Table 9.2: Parameters used in *WindowsFeatures* class.

Parameters	**Description**
scan_row_wise	Is the outer loop over cols (true) or rows (false)?
eol_marker	If true, an empty vector is added at end of lines.
window_size_x	Window size in x direction.
window_size_y	Window size in y direction.
extract_size_x	Scaled target window size in x direction for extraction.
extract_size_y	Scaled target window size in y direction for extraction.
step_width_x	Distance the window moves in x between two steps (in pixels).
step_width_y	Distance the window moves in y between two steps (in pixels).
num_steps_x	Number of steps in x direction.
num_steps_y	Number of steps in y direction.
dct_min_modes	DCT coefficients with at least that many modes (sum of both directions) are taken as feature components.
dct_max_modes	DCT coefficients with at most that many modes (sum of both directions) are taken as feature components.
x_feature	The x coordinate is taken as feature component.
y_feature	The y coordinate is taken as feature component.

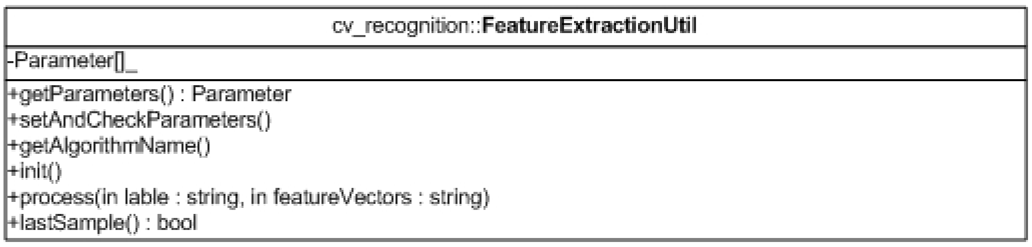

PIXELTYPE
cv_recognition::WindowsFeatures
-parameter[]_ : Parameter
+getParameters() : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+processFrame(in &frame : Frame<PIXELTYPE>, in &featureVectors : list<vector>)

FeatureExtractionUtil Class

The functionality of this class is to help the *TrainModelsApplication* to save feature vectors easily. It indicates which sample belongs to which class by the sample file's name and save the feature vector into the feature vector file. The parameters it uses are listed below:

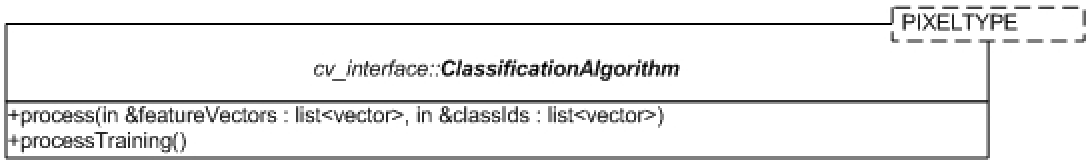
Table 9.3: Parameters used in *FeatureExtractionUtil* class.

Parameters	Description
class_num	The number of classes in a dataset.
samples_num	The number of samples of a dataset.
class_0_label	The string label of class 0.
class_1_label	The string label of class 1.
class_2_label	The string label of class 2.
class_3_label	The string label of class 3.
class_4_label	The string label of class 4.
feature_file	The file to save feature vectors of samples.


cv_recognition::FeatureExtractionUtil
-Parameter[]_
+getParameters() : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+process(in lable : string, in featureVectors : string)
+lastSample() : bool

ClassificationAlgorithm Interface

This interface supplies two interface functions, *process* and *processTraining*. The first function is to execute the classification procedure given a feature vector. The second one is to execute the training process of a corresponding classification models.

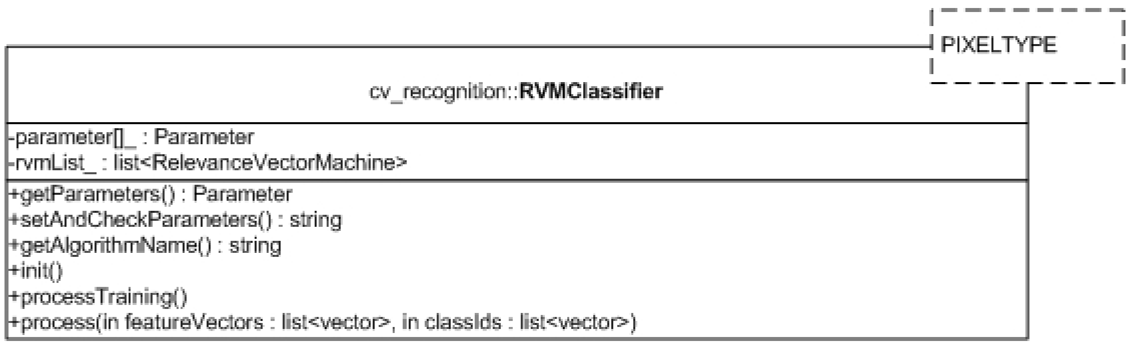

PIXELTYPE
cv_interface::ClassificationAlgorithm
+process(in &featureVectors : list<vector>, in &classIds : list<vector>)
+processTraining()

RVMClassifier Class

The functionality of this class is to create a list of RVM models and to train them by invoking the member function of *RelevanceVectorMachine* class. After the training, final weights for the models will be stored into a weight file indicated by the parameter. Another state of this class is multi-classification. When the train_model parameter is set to false, the class will not run to train RVM models, instead, it will read models weights from the weights file indicated by the parameter and classify a input sample by invoking process function. The parameters file of this class is shown below:

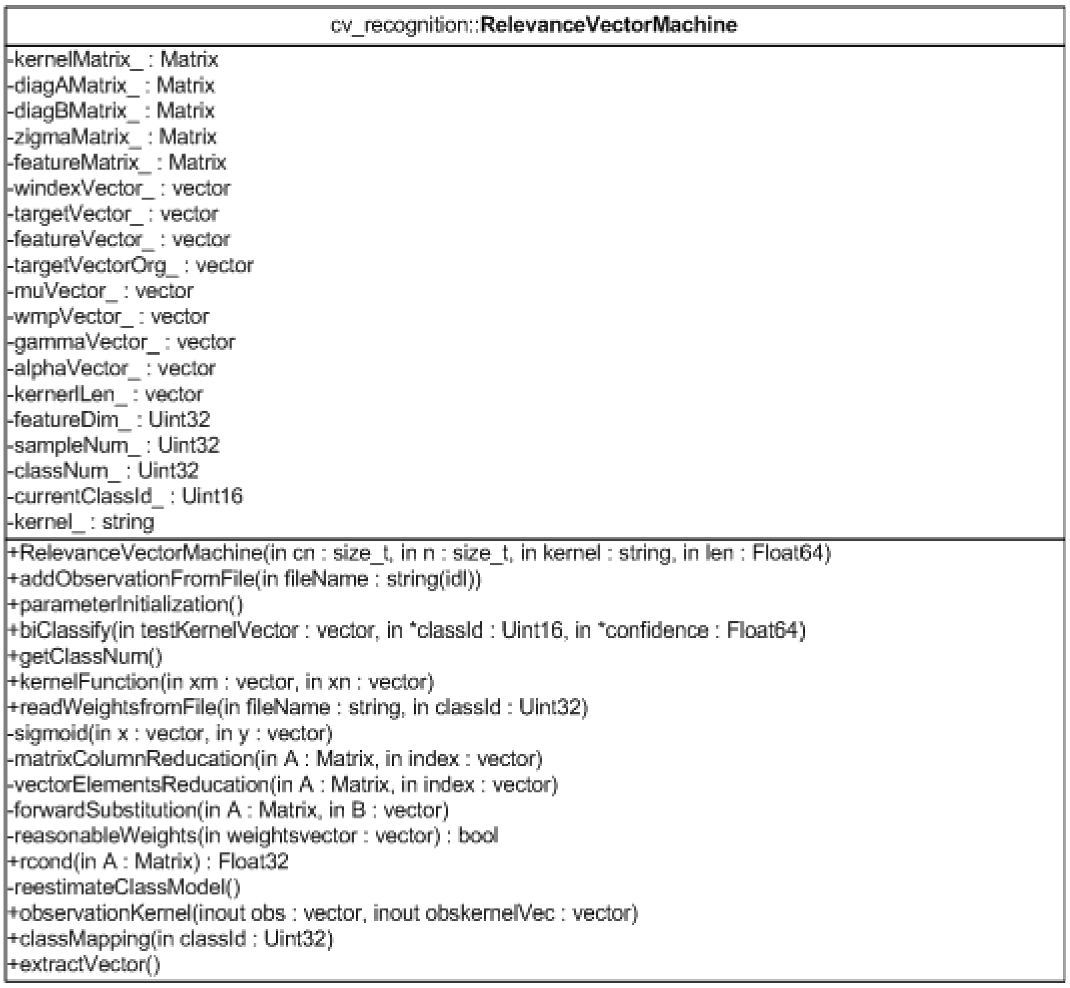
Table 9.4: Parameters of *RVMClassifier* class.

Parameters	**Description**
class_num	The number of classes in a dataset.
feature_dim	The dimension of the feature vectors.
kernel	The kernel function you will use in RVM.
kernel_length	The parameter of kernel function used in RVM.
train_model	If the value is “ture”, the class will be set on training state.
samples_vectors_filename	The location of the file where the feature vectors of samples stored.
model_weights_filename	The weights file storing the final weights of model. When class state is training, it will be written otherwise it would be read.


PIXELTYPE
cv_recognition::RVMClassifier
-parameter[]_ : Parameter
-rvmList_ : list<RelevanceVectorMachine>
+getParameters() : Parameter
+setAndCheckParameters() : string
+getAlgorithmName() : string
+init()
+processTraining()
+process(in featureVectors : list<vector>, in classIds : list<vector>)

RelevanceVectorMachine Class

This class is the real RVM model class but not an algorithm class. It implements the RVM classification model and will really do the training and classification operation. The limitation of this class is that it only can handle two classes classification case.



cv_recognition::**RelevanceVectorMachine**

-kernelMatrix_ : Matrix
-diagAMatrix_ : Matrix
-diagBMatrix_ : Matrix
-zigmaMatrix_ : Matrix
-featureMatrix_ : Matrix
-windexVector_ : vector
-targetVector_ : vector
-featureVector_ : vector
-targetVectorOrg_ : vector
-muVector_ : vector
-wmpVector_ : vector
-gammaVector_ : vector
-alphaVector_ : vector
-kernerlLen_ : vector
-featureDim_ : Uint32
-sampleNum_ : Uint32
-classNum_ : Uint32
-currentClassId_ : Uint16
-kernel_ : string

+RelevanceVectorMachine(in cn : size_t, in n : size_t, in kernel : string, in len : Float64)
+addObservationFromFile(in fileName : string(idl))
+parameterInitialization()
+biClassify(in testKernelVector : vector, in *classId : Uint16, in *confidence : Float64)
+getClassNum()
+kernelFunction(in xm : vector, in xn : vector)
+readWeightsfromFile(in fileName : string, in classId : Uint32)
-sigmoid(in x : vector, in y : vector)
-matrixColumnReducation(in A : Matrix, in index : vector)
-vectorElementsReducation(in A : Matrix, in index : vector)
-forwardSubstitution(in A : Matrix, in B : vector)
-reasonableWeights(in weightsvector : vector) : bool
+rcond(in A : Matrix) : Float32
-reestimateClassModel()
+observationKernel(inout obs : vector, inout obskernelVec : vector)
+classMapping(in classId : Uint32)
+extractVector()

9.6.2 Sequence diagram

Fig. 9.10 shows the procedure of the training of RVM. Firstly, the application will read sample files from the disk. Then the feature extraction algorithm will extract their features and send them to RVM. When RVM receives enough sample feature vectors, it will start to use them to train the classification model. After training, RVM classifier will write the final weights into a file that will be reload into RVM when it need to make classification.

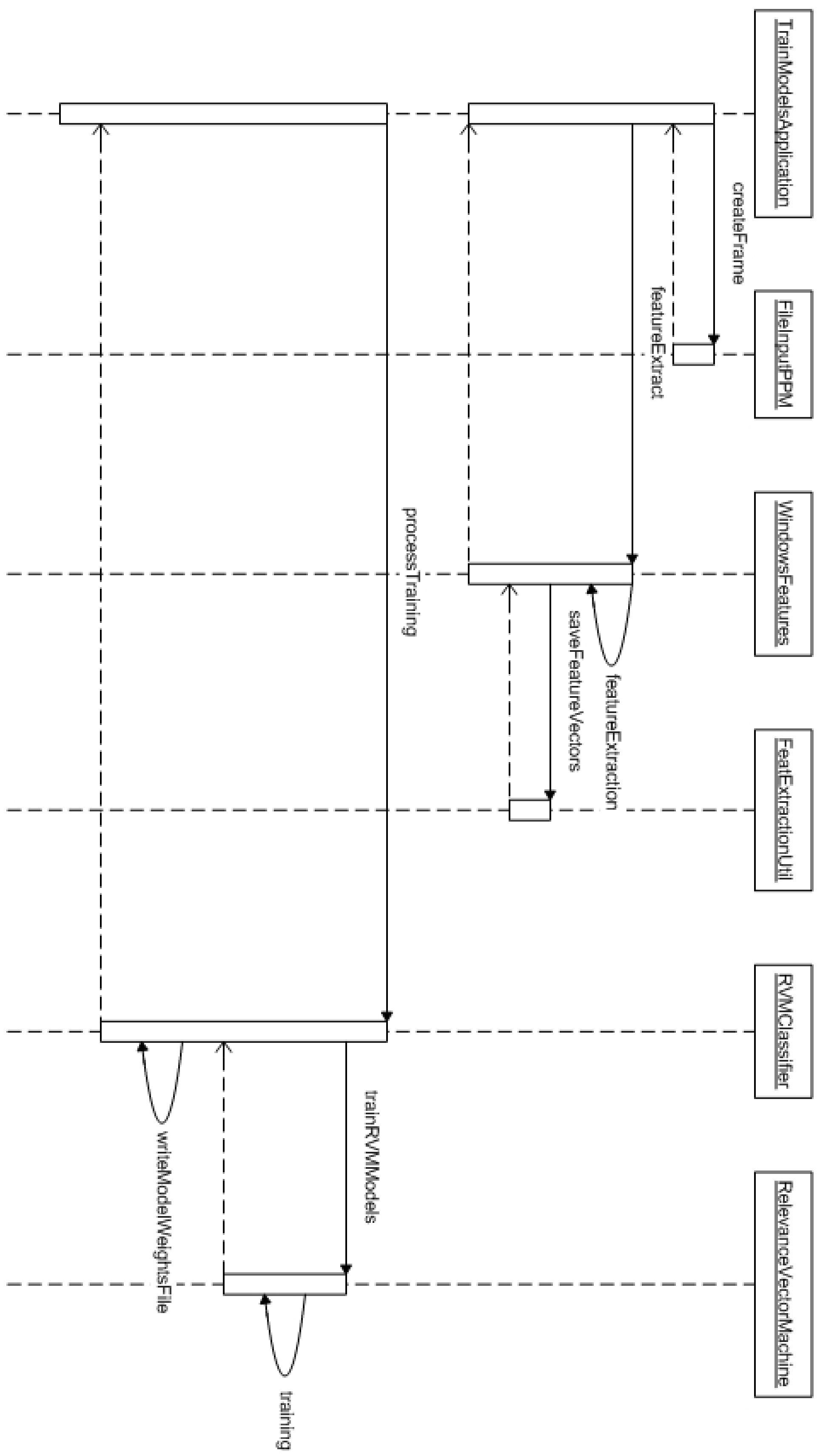


Figure 9.10: Sequence diagram of screen corners training application case.

9.7 UI-Wand application

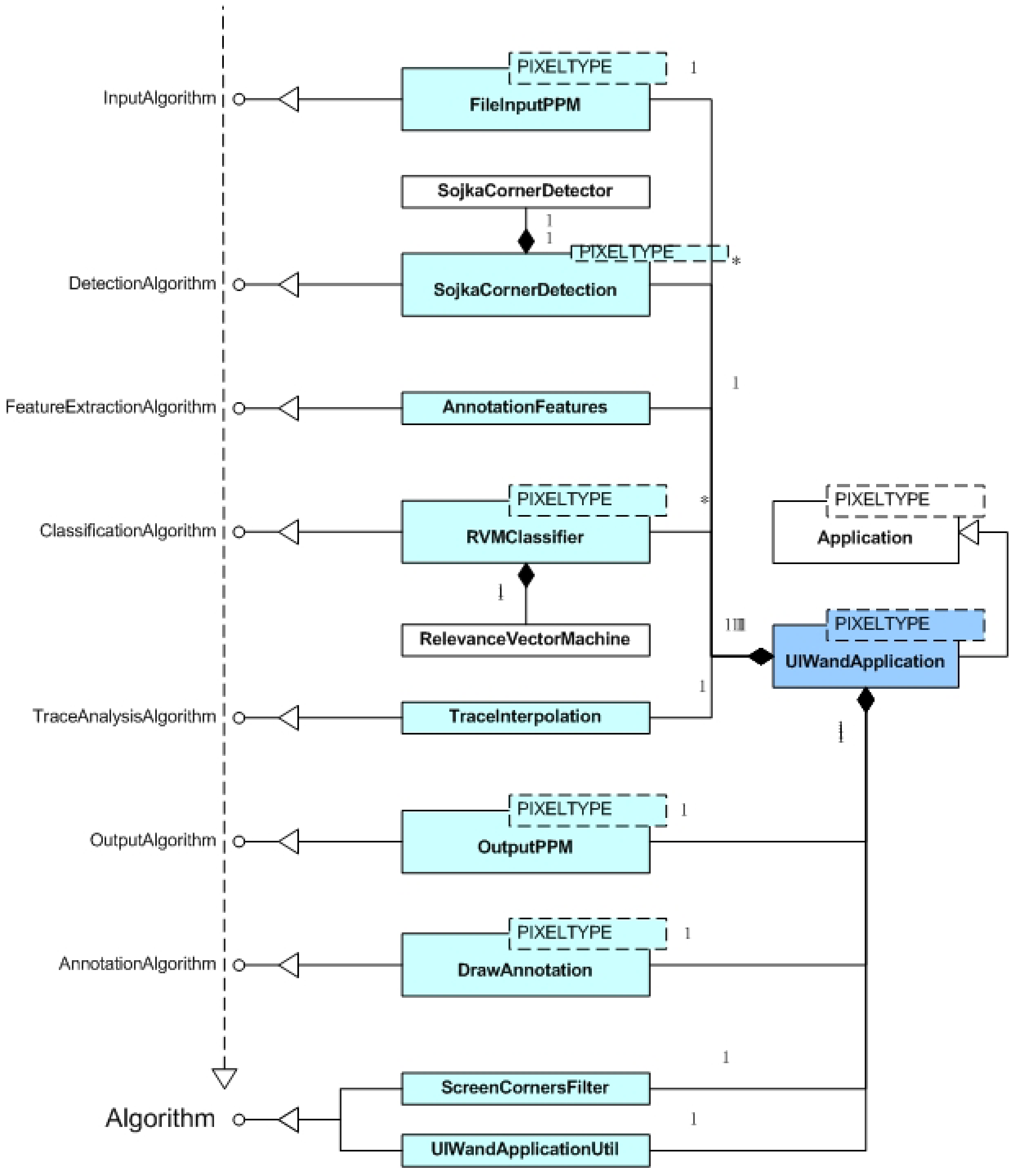


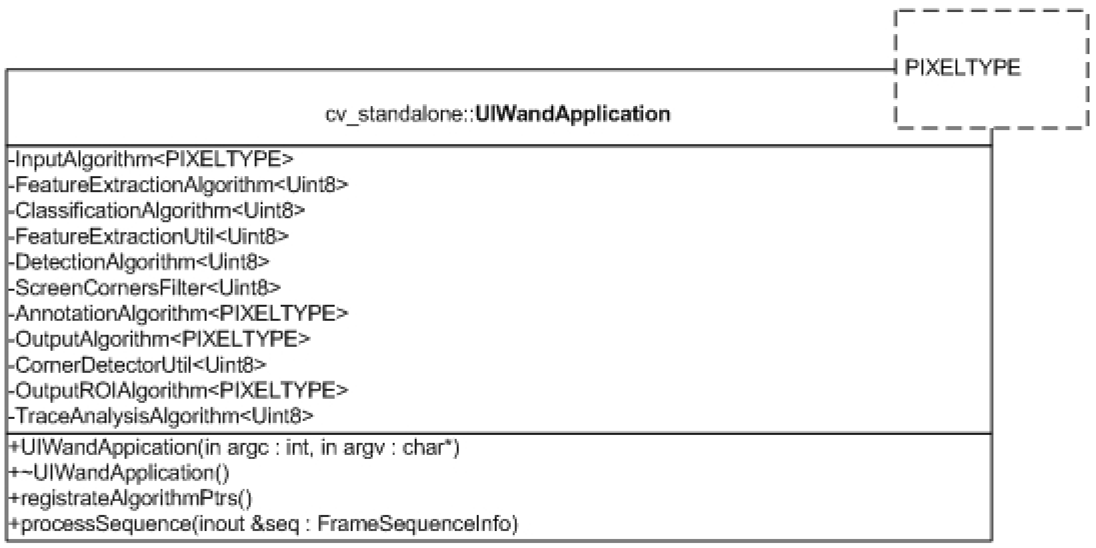
Figure 9.11: Class diagram in UI-Wand application case. Classes in cyan are the algorithms used in *UIWandApplication* class shown in blue. The dot derivation line means that all special algorithms interface are deriving from algorithm interface.

UI-Wand application is the main application in UI-Wand system. It is the final application that detects screen corners, finds pointing position and analyzes the user's gestures. It is also the base application for adaptive dataset generation application, and gestures collection application. So it is the application that has most algorithms embedded in, which can easily be seen from Fig. 9.11. But we noticed that the *InputAlgorithm* interface is implemented by *FileInputPPM* class, which means we still get the sequence of frames from files on disk instead of collecting the sequence of frames from cameras directly. And the pointing positioning analysis class misses as well in this structure. So the structure shown in this figure is not for the final prototype of UI-Wand used for demonstration. Actually it is the UI-Wand application for running off-line. But these are not big problems for understanding the structure of UI-Wand application, because the main algorithms are already combined into the system. For the final prototype UI-Wand system, the only thing we need to do is to replace *FileInputPPM* class with *CamInput* class and add pointing positioning analysis class then the UI-Wand application can work properly on-line.

9.7.1 Individual class description

UIWandApplication Class

This is the main application class that integrates all algorithms needed for UI-Wand.

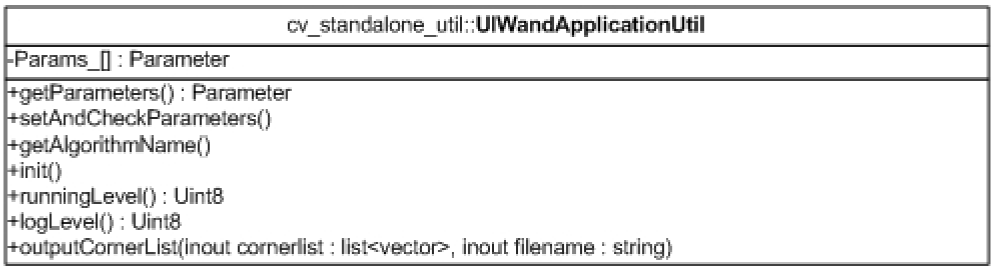

PIXELTYPE
cv_standalone::UIWandApplication
-InputAlgorithm<PIXELTYPE>
-FeatureExtractionAlgorithm<Uint8>
-ClassificationAlgorithm<Uint8>
-FeatureExtractionUtil<Uint8>
-DetectionAlgorithm<Uint8>
-ScreenCornersFilter<Uint8>
-AnnotationAlgorithm<PIXELTYPE>
-OutputAlgorithm<PIXELTYPE>
-CornerDetectorUtil<Uint8>
-OutputROIAlgorithm<PIXELTYPE>
-TraceAnalysisAlgorithm<Uint8>
+UIWandAppication(in argc : int, in argv : char*)
+~UIWandApplication()
+registrateAlgorithmPtrs()
+processSequence(inout &seq : FrameSequenceInfo)

UIWandApplicationUtil Class

This class is designed for control the running states and logging states of UI-Wand application by setting its parameters with different values. Because *UIWandApplication* class contains many algorithms, it is important to test the performance of every algorithm in it. So we design some different running states for it, by running on which the UI-Wand application can select different algorithms for running so that we can easily test their performance. In order to save some results of algorithm, we need to write some information into file, which can also be controlled by the parameter. The special parameters are listed in Table 9.5.

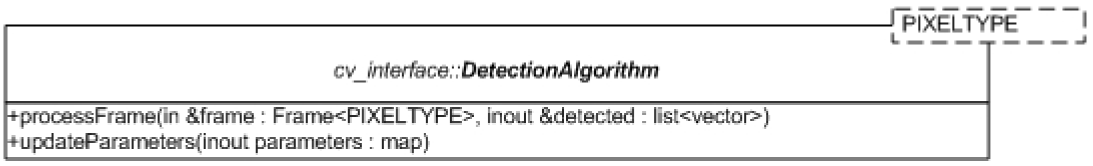
Table 9.5: Parameters of *UIWandApplicationUtil* class

Parameters	Description
Run_level	The running states of UI-Wand application, the possible value for states are 1, 2, 3, and 4. 1. Running only with Sojka Corner Detection. 2. Running with Sojka Corner Detection and a rectangle filter. 3. Running with Sojka Corner Detection and RVM. 4. Running with all algorithms (screen corner detection and gesture recognition).
log_level	If true, the application will save some algorithms results.
log_directory	The location to save results.


cv_standalone_util::**UIWandApplicationUtil**
-Params_[] : Parameter
+getParameters() : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+runningLevel() : Uint8
+logLevel() : Uint8
+outputCornerList(inout cornerlist : list<vector>, inout filename : string)

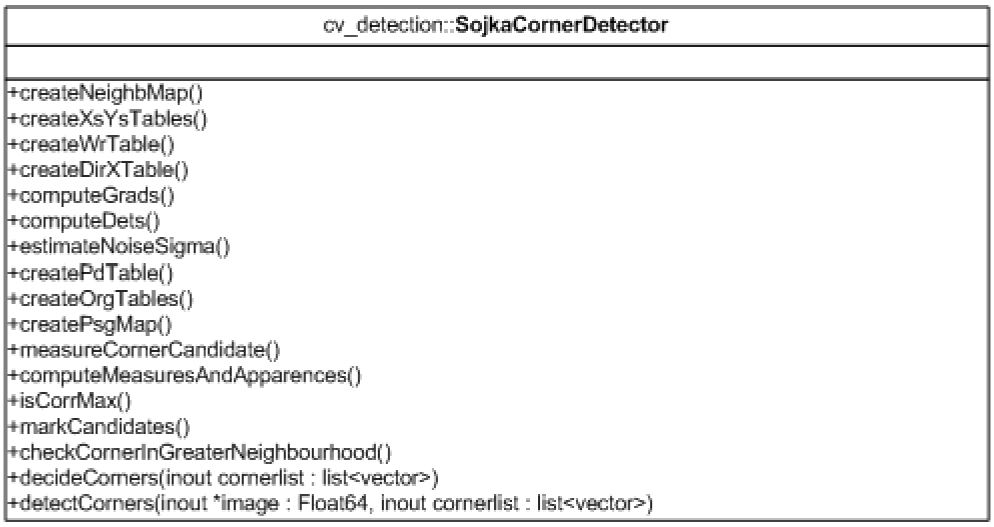
DetectionAlgorithm Interface

This interface supplies two functions for the special implementation class. The first function is to detect a special place in a frame then save the detection results into a list. The second function is just for updating the parameters used in the special algorithm. In our *SojkaCornerDetection* algorithm we update its ROIs.


PIXELTYPE
*cv_interface::**DetectionAlgorithm***
+processFrame(in &frame : Frame<PIXELTYPE>, inout &detected : list<vector>)
+updateParameters(inout parameters : map)

SojkaCornerDetector Class

This class implements the Sojka corner detection model, which can detect the corners in an image.

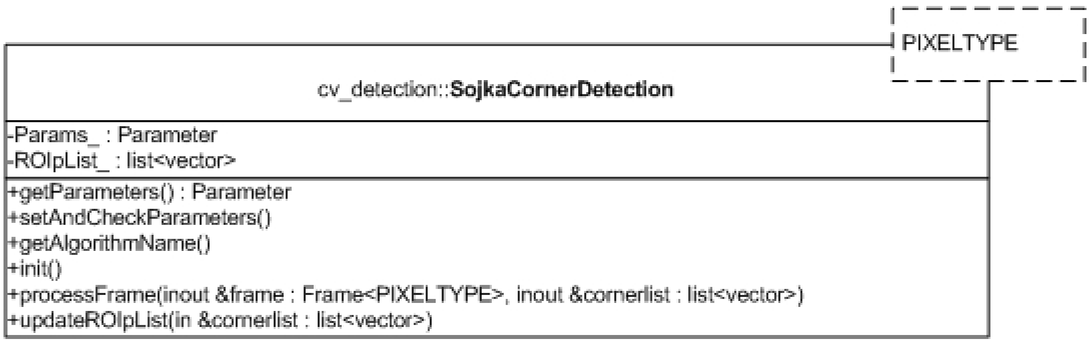

cv_detection::**SojkaCornerDetector**
+createNeighbMap()
+createXsYsTables()
+createWrTable()
+createDirXTable()
+computeGrads()
+computeDets()
+estimateNoiseSigma()
+createPdTable()
+createOrgTables()
+createPsgMap()
+measureCornerCandidate()
+computeMeasuresAndApparences()
+isCorrMax()
+markCandidates()
+checkCornerInGreaterNeighbourhood()
+decideCorners(inout cornerlist : list<vector>)
+detectCorners(inout *image : Float64, inout cornerlist : list<vector>)

SojkaCornerDetection Class

The functionality of this class is to invoke *SojkaCornerDetector* class and let it do the detection work. The main difference of this class in structure with *SojkaCornerDetector* class is that it is an algorithm class that can configure the parameters by parameters file. The functionality difference of this class with the above class is that it can do the corner detection only on some sub-images indicated by ROIs and another improvement is that it can do the corner detection both on gray level image and color image by changing the value of the template. The special parameters are listed in Table 9.6:

Table 9.6: Parameters of *SojkaCornerDetection* class

Parameters	**Description**
halfPsgMaskSize	See section 4.4.
angleThresh	See section 4.4.
noiseGradSizeThresh	See section 4.4.
apparenceThresh	See section 4.4.
sigmaD	See section 4.4.
sigmaR	See section 4.4.
halfExtMaskSize	See section 4.4.
ROI_x_size	The width of region-of-interest area
ROI_y_size	The height of region-of-interest area

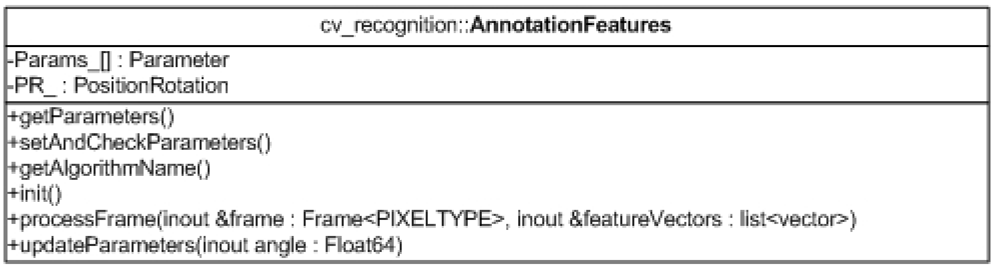

PIXELTYPE
cv_detection::**SojkaCornerDetection**
-Params_ : Parameter
-ROIpList_ : list<vector>
+getParameters() : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+processFrame(inout &frame : Frame<PIXELTYPE>, inout &cornerlist : list<vector>)
+updateROIpList(in &cornerlist : list<vector>)

AnnotationFeatures Class

The functionality of this class is to use DCT to extract sample features, which is similar to *WindowsFeatures* class. But the difference is that, instead of scanning a whole image with a target window to get a list of feature vectors, this class will only extract the feature vectors of target windows defined by a list of points. It is designed for our Candidates-Winners approach requirement. By giving a list of corners, this class will extract the sub-images in target windows with the corners as the centers and size defined by parameters, so that feature vectors represent those corners and can be sent to RVM to classify. The special parameters are listed in Table 9.7:

Table 9.7: Parameters of *AnnotationFeatures* class.

Parameters	Description
window_size_x	Window size in x direction.
window_size_y	Window size in y direction.
extract_size_x	Scaled window size in x direction to be extracted.
extract_size_y	Scaled window size in y direction to be extracted.
extract_mode	Zoom every sample or zoom the whole image.
tilted_angle	The tilted angle of the image.
dct_min_modes	DCT coefficients with at least that many modes (sum of both directions) are taken as feature components.
dct_max_modes	DCT coefficients with at most that many modes (sum of both directions) are taken as feature components.
x_feature	The x coordinate is taken as feature component.
y_feature	The y coordinate is taken as feature component.



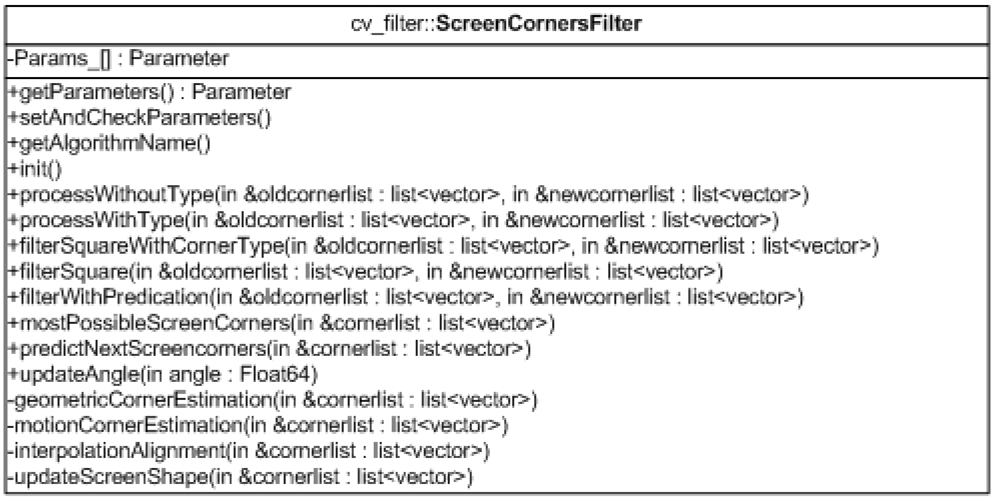
```
cv_recognition::AnnotationFeatures
-Params_[] : Parameter
-PR_ : PositionRotation
+getParameters()
+setAndCheckParameters()
+getAlgorithmName()
+init()
+processFrame(inout &frame : Frame<PIXELTYPE>, inout &featureVectors : list<vector>)
+updateParameters(inout angle : Float64)
```

ScreenCornersFilter Class

It is another important class in our system. The main functionality of this class is to filter out the screen corners by rectangle filters. It also contains missing corner prediction and ROI selection algorithms we introduced in Chapter 6. The parameters are listed below:

Table 9.8: Parameters of *ScreenCornersFilter* class.

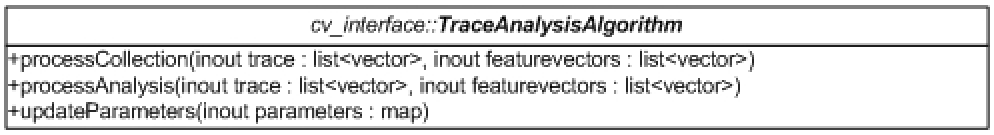
Parameters	Description
distance_x	The width of a possible rectangle used as filter (in pixels).
distance_y	The height of a possible rectangle used as filter (in pixels).
offset	The tolerable distortion of the rectangle.
tilted_angle	The tilted angle of the current image.
predication_num_frame	The number of frame used to predict corners trace (fixed to three in our final system).
history_size	How many history detection you want to save.



```
cv_filter::ScreenCornersFilter
-Params_[] : Parameter
+getParameters() : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+processWithoutType(in &oldcornerlist : list<vector>, in &newcornerlist : list<vector>)
+processWithType(in &oldcornerlist : list<vector>, in &newcornerlist : list<vector>)
+filterSquareWithCornerType(in &oldcornerlist : list<vector>, in &newcornerlist : list<vector>)
+filterSquare(in &oldcornerlist : list<vector>, in &newcornerlist : list<vector>)
+filterWithPredication(in &oldcornerlist : list<vector>, in &newcornerlist : list<vector>)
+mostPossibleScreenCorners(in &cornerlist : list<vector>)
+predictNextScreencorners(in &cornerlist : list<vector>)
+updateAngle(in angle : Float64)
-geometricCornerEstimation(in &cornerlist : list<vector>)
-motionCornerEstimation(in &cornerlist : list<vector>)
-interpolationAlignment(in &cornerlist : list<vector>)
-updateScreenShape(in &cornerlist : list<vector>)
```

TraceAnalysisAlgorithm Interface

This interface supplies three functions for the special implementation class. The *processCollection* function is used to collect sample gestures, extract their features and then save the features into a file. The *processAnalysis* function will just do feature extraction on a trace then return the feature vectors. The third function is *updateParameters* like other interface, which is just for updating the parameters used in the special algorithm. In our *InterpolationTrace* algorithm we update the trace information.



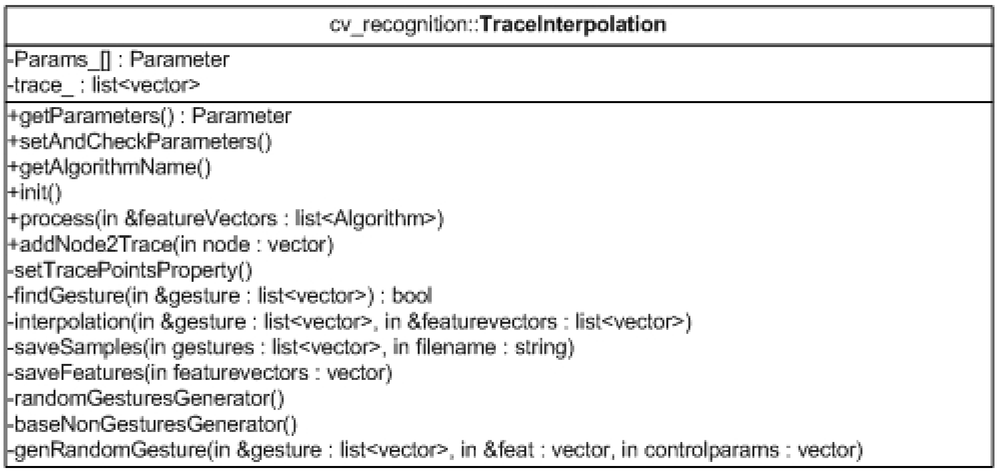
```
cv_interface::TraceAnalysisAlgorithm
+processCollection(inout trace : list<vector>, inout featurevectors : list<vector>)
+processAnalysis(inout trace : list<vector>, inout featurevectors : list<vector>)
+updateParameters(inout parameters : map)
```

TraceInterpolation Class

The functionality of this class is to use interpolation method to do the trace analysis like what we introduced in Chapter 8. There are two running states in this class. One state is for collecting gesture samples from an input sequence and the interpolation analysis for the gestures then saves the feature vectors into a file. The other state is only for the trace analysis for one trace and returns its feature vector for RVM classification. The special parameters used are listed below:

Table 9.9: Parameters of *TraceInterpolation* class.

Parameters	Description
trace_size	How long is the trace to keep?
model	Two values, “collect” or “analysis”.
samples_dir	The directory for saving gesture samples.
feature_vectors_filename	The file for saving feature vectors of gestures.
num_motion_vector	The number of motion vectors for analyzing trace.
current_gesture	The mark to indicate which kind of the gestures are collected now.
gesture0_label	Gesture label for indicating the type of gesture.
gesture1_label	Gesture label for indicating the type of gesture.
gesture2_label	Gesture label for indicating the type of gesture.
gesture3_label	Gesture label for indicating the type of gesture.
gesture4_label	Gesture label for indicating the type of gesture.
num_gesture_samples	How many gestures to get?
num_nongesture_samples	How many invalid gestures at most you want to get?
base_non_gesture	To generate basic invalid gestures?
non_gesture_step_min	The minimum step size for invalid gestures generation.
non_gesture_step_max	The maximum step size for invalid gestures generation.
non_gesture_step_distribution	How many kinds of step to have?
non_gesture_angle_min	The minimum angle degree variant for invalid gestures generation.
non_gesture_angle_max	The maximum angle degree variant for invalid gestures generation.
non_gesture_angle_distribution	How many different angles to have?
movement_sensitivity	The sensitivity to detect moving (in pixels).
holding_num_unittime	The number of frames moving within the range of movement_sensitivity.
missing_tolerance	How many missing frames it can tolerant?



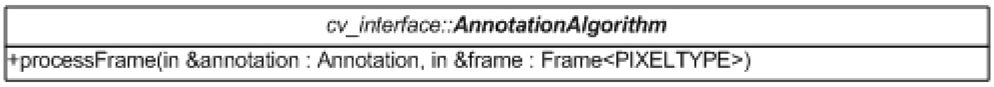
```
cv_recognition::TraceInterpolation

-Params_[] : Parameter
-trace_ : list<vector>

+getParameters() : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+process(in &featureVectors : list<Algorithm>)
+addNode2Trace(in node : vector)
-setTracePointsProperty()
-findGesture(in &gesture : list<vector>) : bool
-interpolation(in &gesture : list<vector>, in &featurevectors : list<vector>)
-saveSamples(in gestures : list<vector>, in filename : string)
-saveFeatures(in featurevectors : vector)
-randomGesturesGenerator()
-baseNonGesturesGenerator()
-genRandomGesture(in &gesture : list<vector>, in &feat : vector, in controlparams : vector)
```

AnnotationAlgorithm Interface

This interface supplies one function for its implementation class, which will draw some annotation into a frame.

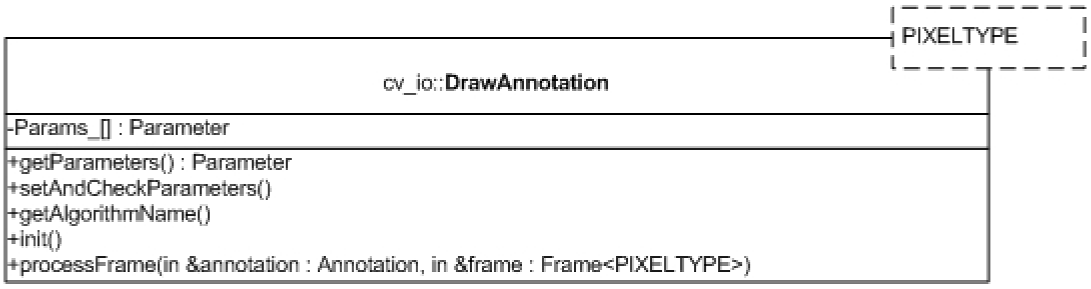

cv_interface::AnnotationAlgorithm
+processFrame(in &annotation : Annotation, in &frame : Frame<PIXELTYPE>)

DrawAnnotation Class

This class is designed for visualization of the detection results. What it does is to use a color rectangle box to draw an annotation in the input image, so that a user can see the detection results directly. Some parameters it accepts are listed below:

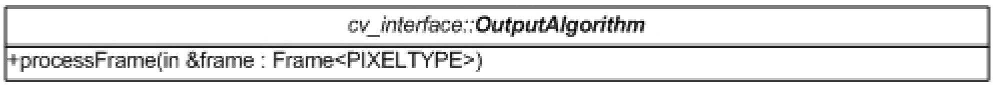
Table 9.10: Parameters of *DrawAnnotation* class.

Parameters	**Description**
box_color_r	Red color component for drawing boxes.
box_color_g	Green color component for drawing boxes.
box_color_b	Blue color component for drawing boxes.



OutputAlgorithm Interface

This interface only supplies one function to implementation class, which will output a frame. The special output methods, such as writing to files, displaying in windows, and etc, will be implemented by the special classes.

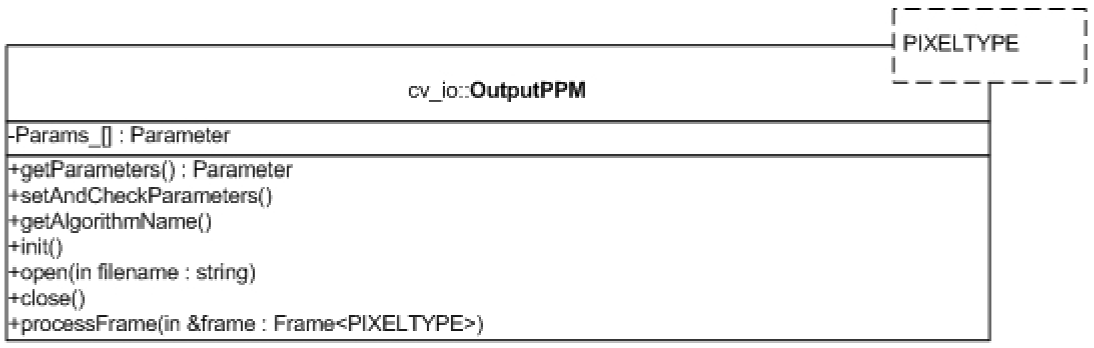

cv_interface::OutputAlgorithm
+processFrame(in &frame : Frame<PIXELTYPE>)

OutputPPM Class

The functionality of this class is to write a *Frame* to a ppm file. The parameters of this class are listed below:

Table 9.11: Parameters of *OutputPPM* class.

Parameters	**Description**
filename_base	Base filename (if not set: use input name).
append_framecount	Append frame count to filename.
append_sequencecount	Append sequence count to filename.
binary_format	Write raw values.



9.7.2 Sequence diagram

Fig. 9.12 presents a period of lifetime of *UIWandApplication*. During this period, the application finishes screen corner detection on one frame and gesture recognition based on the current trace information. This sequence diagram is a brief representation for the real work of *UIWandApplication*, many detailed message and data exchange have not been shown. But it will not affect your understanding to its global procedure.

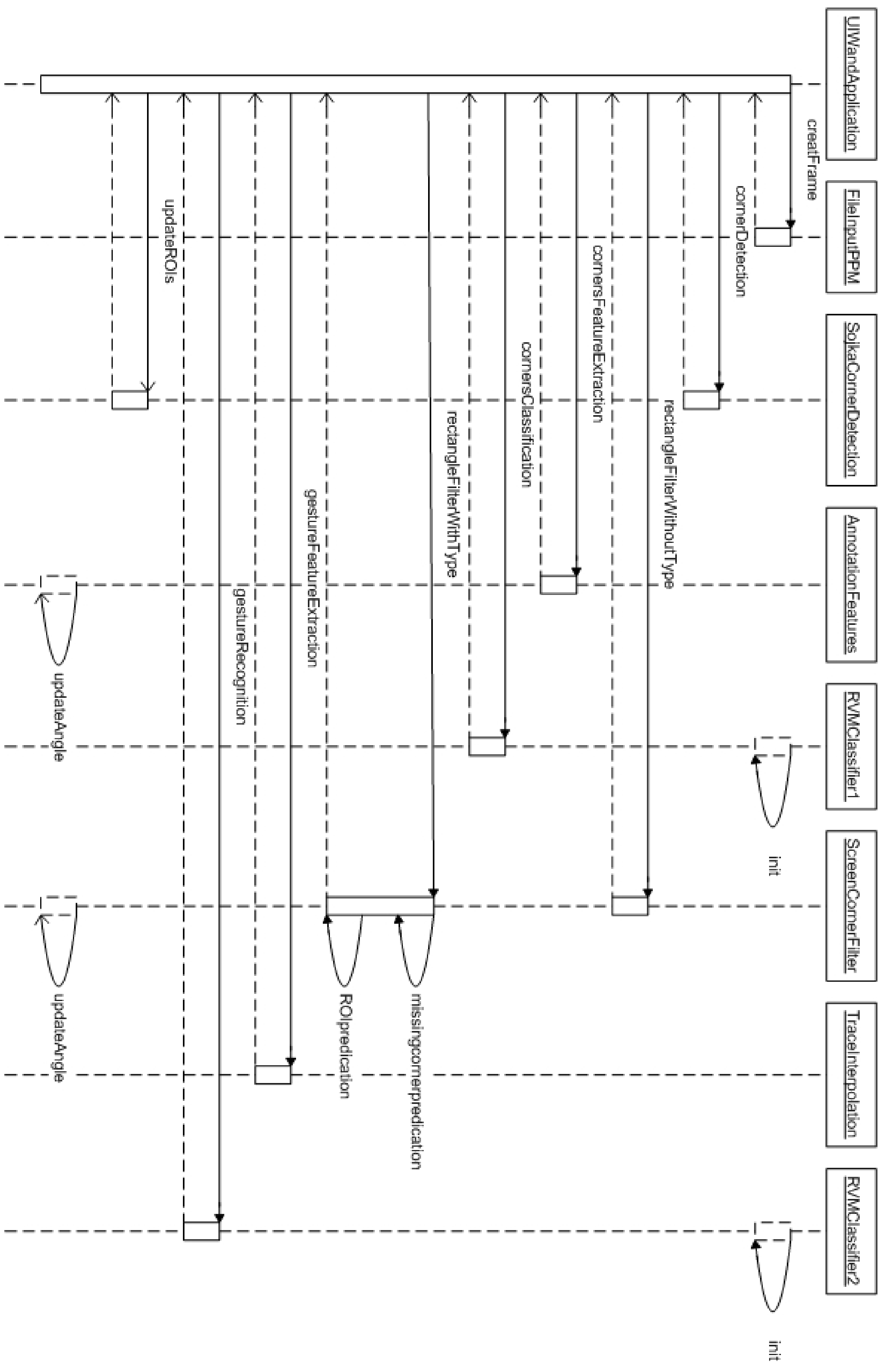


Figure 9.12: The sequence diagram of UI-Wand application. Dot box means the object was or will be activated sometime.

9.8 Gestures dataset collection application

Gesture dataset collection application is based on UI-Wand application. The class diagram is the same with UI-Wand application shown in Fig. 9.11. The main functionality and procedure of this application is to run the UI-Wand application to detect the screen corners and then save them into a history list so that the movement information of UI-Wand is kept as a trace. With this UI-Wand trace information, the *InterpolationTrace* algorithm will do some analysis to the trace, such as justifying if the current trace contains a valid gesture and execute the interpolation operation on the gesture to get its feature vector. The *InterpolationTrace* algorithm keeps analyzing and collects those traces, which are effective gestures until enough gestures samples have been collected and extracted to feature vectors, so that the RVM models can be trained as a gestures recognizer. The detailed procedure is shown in Fig. 9.13, which can be seen as a modification of Fig. 9.12. The *traceInterpolation* object in Fig. 9.12 is just for feature extraction to a gesture, instead the *traceInterpolation* object in Fig. 9.13 is to find samples, extract features and save them into dataset.

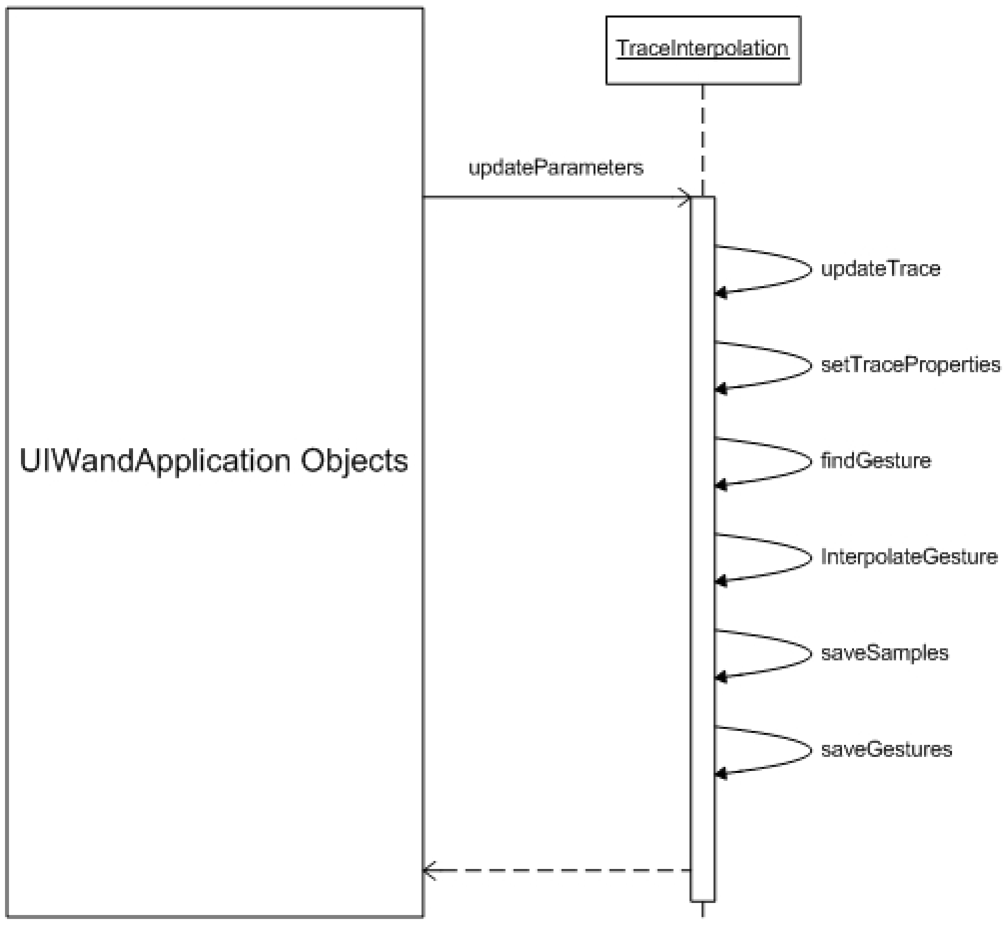


Figure 9.13: Part of sequence diagram for gestures dataset collection application.

9.9 Gestures training application

The functionality of gesture training application is to use samples feature vectors collected by gestures dataset collection application to train RVM models so that it can do the classification for gestures. This application framework is very simple. It only use *RVMClassifier* algorithm. The classes diagram is shown in Fig. 9.14. Fig. 9.15 shows the application execution procedure by sequence diagram.

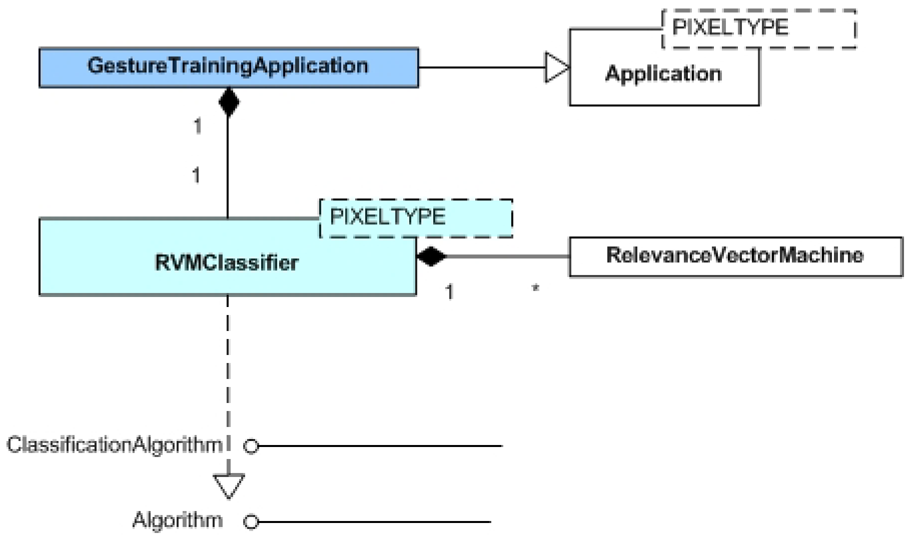


Figure 9.14: Class diagram of gestures training application.

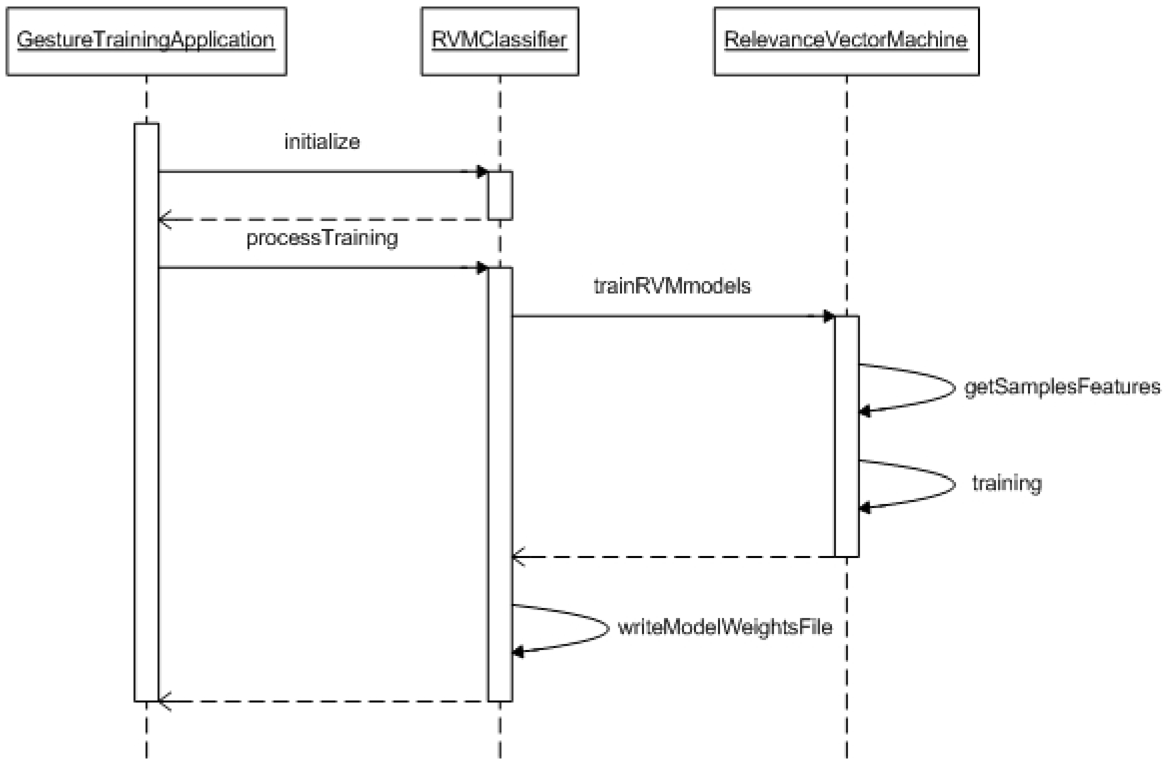
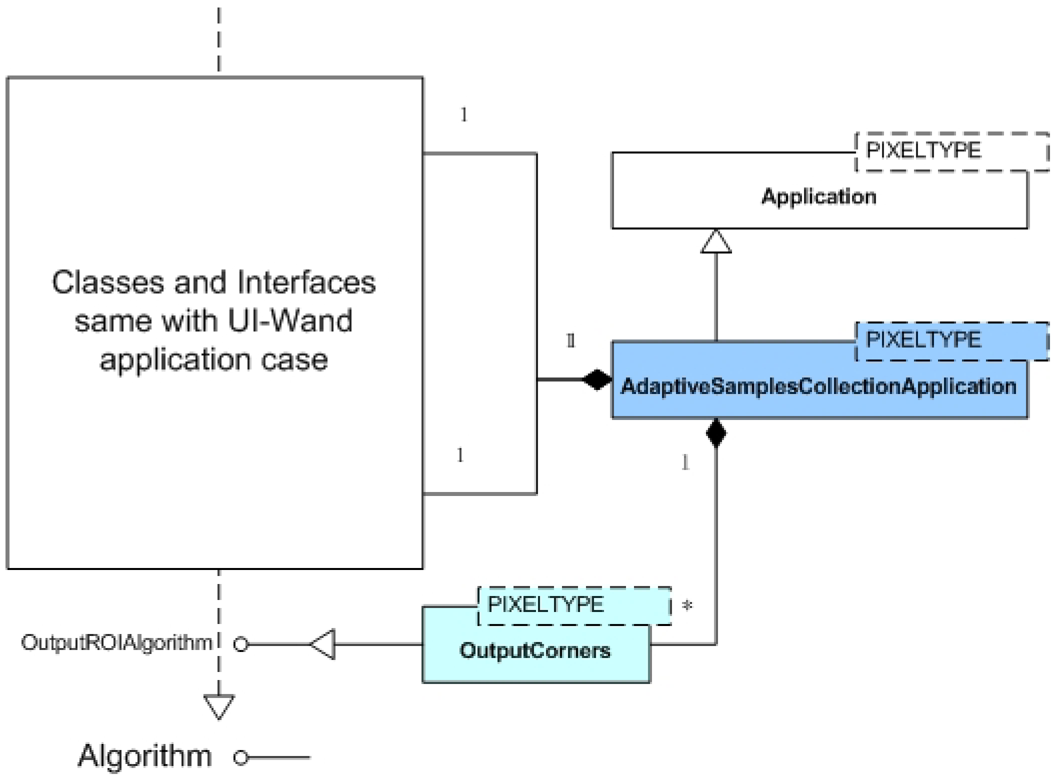


Figure 9.15: Sequence frame of gestures training application.

9.10 Adaptive screen corner dataset collection application

The functionality of this application is to collect bigger screen corners samples and special non-screen corners samples from input frames. By these new bigger sample images, the accuracy of RVM models for classification will be much improved so that UI-Wand can even use only RVM models to realize screen corners. The algorithms used in this application are basically the same as in the with UI-Wand application case. The difference is that we add an *OutROIAlgorithm* interface and *OuputCorners* class into this application, which will help the system to select the samples from frames and save them into the disk as adaptive dataset. The new class and interface is shown in the class diagram (see Fig. 9.16).



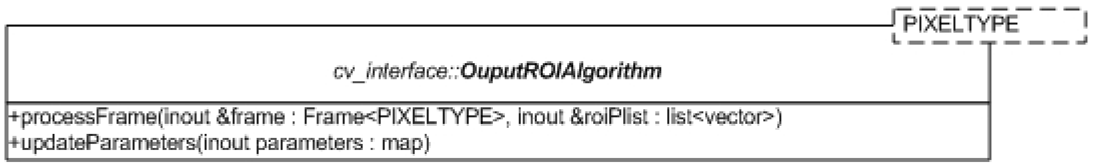
Classes and Interfaces same with UI-Wand application case
1
1
1
PIXELTYPE
Application
PIXELTYPE
AdaptiveSamplesCollectionApplication
1
PIXELTYPE
*
OutputCorners
OutputROIAlgorithm
Algorithm

Figure 9.16: Class diagram of adaptive screen corners dataset collection application case.

9.10.1 Individual class description

OutputROIAlgorithm Interface

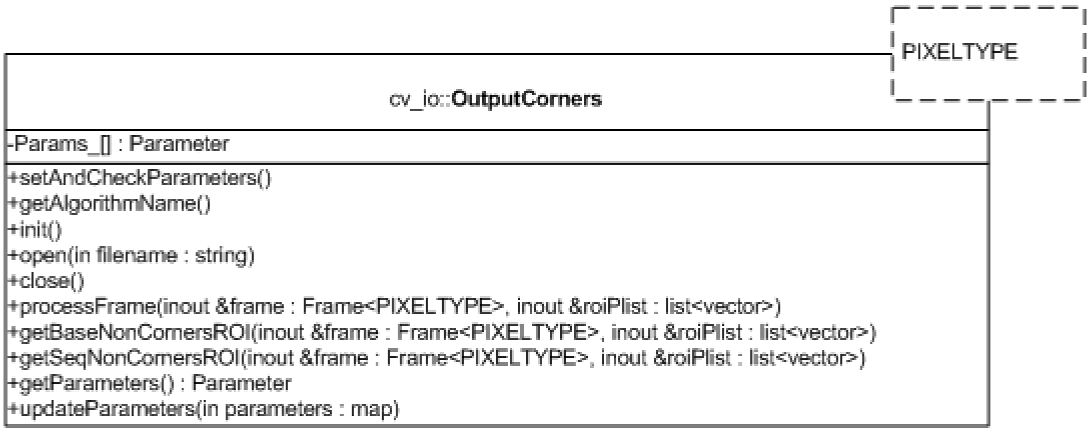
This interface supplies two functions to the implementation class. The first one will select the frame according to the positions in *roiPlist*. The second function is just for updating the parameters used in the special implementation class.



```
+processFrame(inout &frame : Frame<PIXELTYPE>, inout &roiPlist : list<vector>)
+updateParameters(inout parameters : map)
```

OuputCorners Class

The functionality of this class is to find out the ROIs in the frames and select them and then save into dataset. The ROIs in this class are sub-images containing screen corners and sub-images in special places as selective non-screen corners. The special algorithm to select ROIs was introduced in Chapter 7.


PIXELTYPE
cv_io::OutputCorners
-Params_[] : Parameter
+setAndCheckParameters()
+getAlgorithmName()
+init()
+open(in filename : string)
+close()
+processFrame(inout &frame : Frame<PIXELTYPE>, inout &roiPlist : list<vector>)
+getBaseNonCornersROI(inout &frame : Frame<PIXELTYPE>, inout &roiPlist : list<vector>)
+getSeqNonCornersROI(inout &frame : Frame<PIXELTYPE>, inout &roiPlist : list<vector>)
+getParameters() : Parameter
+updateParameters(in parameters : map)

9.10.2 Sequence diagram

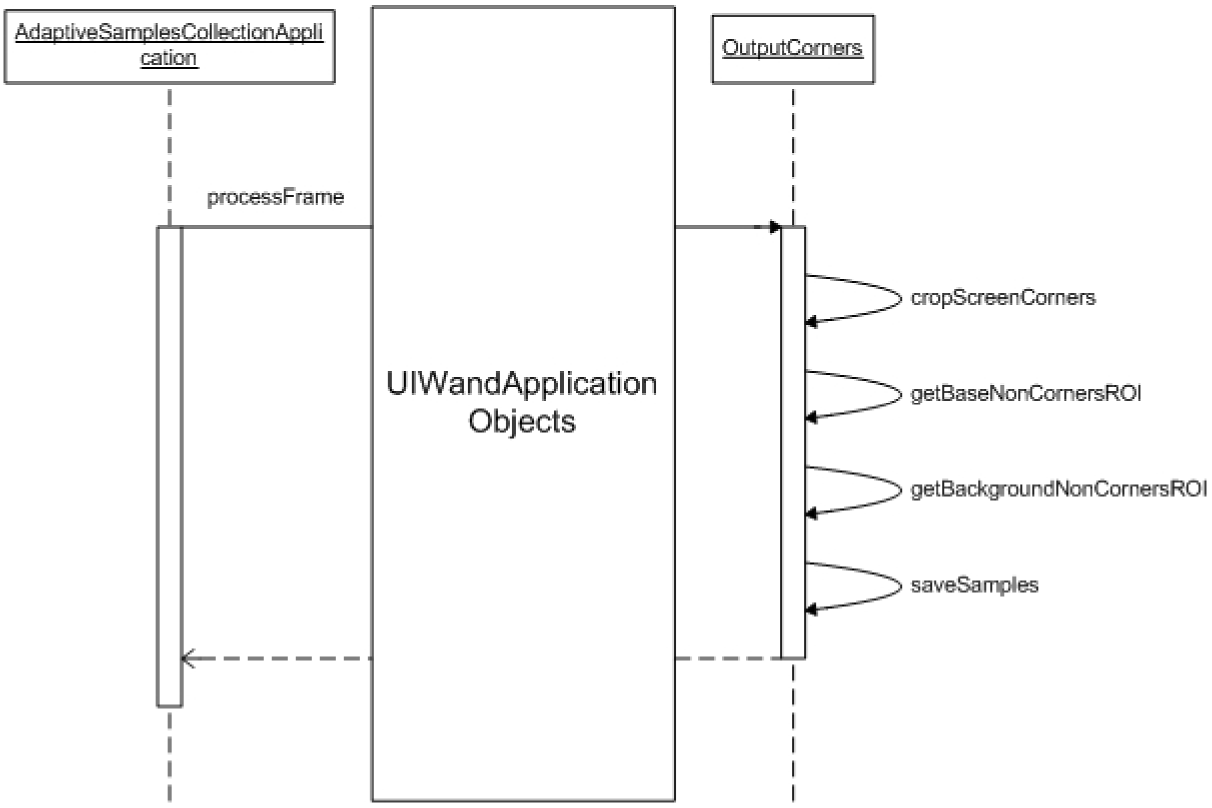

AdaptiveSamplesCollectionApplication
UIWandApplication Objects
OutputCorners
processFrame
cropScreenCorners
getBaseNonCornersROI
getBackgroundNonCornersROI
saveSamples

Figure 9.17: Part of sequence diagram of adaptive screen corners dataset collection application.

10 UI-Wand Utilities Design

In Chapter 9, we introduced the design for the UI-Wand system. Most of the application cases in the system are useful and compulsory for the final running UI-Wand prototype. But those are not all applications we designed. There are some utility application cases we used for helping evaluate the system. In the real system, we design three utilities for the system evaluation. The first one is a visualization utility, by which we can easily see the RVM classification result on one image. The second one is reference corners utility that is a GUI utility. By using this utility we can manually mark four screen corners in one image as reference when we want to know our screen corner detection result. The third utility is error analysis utility, which can specially evaluate the performance of some algorithms by comparing the special algorithms results with the reference corners results. The use cases diagram is shown in Fig. 10.1.

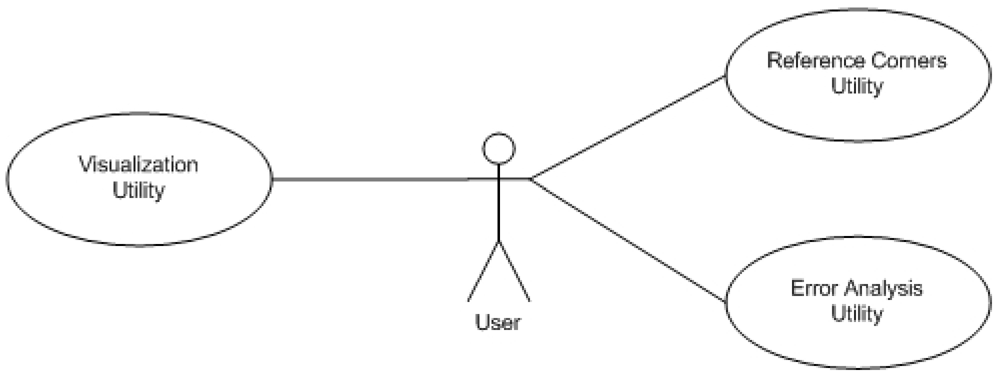


Figure 10.1: Use cases diagram for UI-Wand utilities.

10.1 Visualization Utility

The visualization utility we used for our UI-Wand system is to give a direct impression of RVM models classification to one image. The special results of this utility were already shown in Chapter 4. Now we will see its special design by giving the UML models. The Fig 10.2 below is the visualization application class diagram.

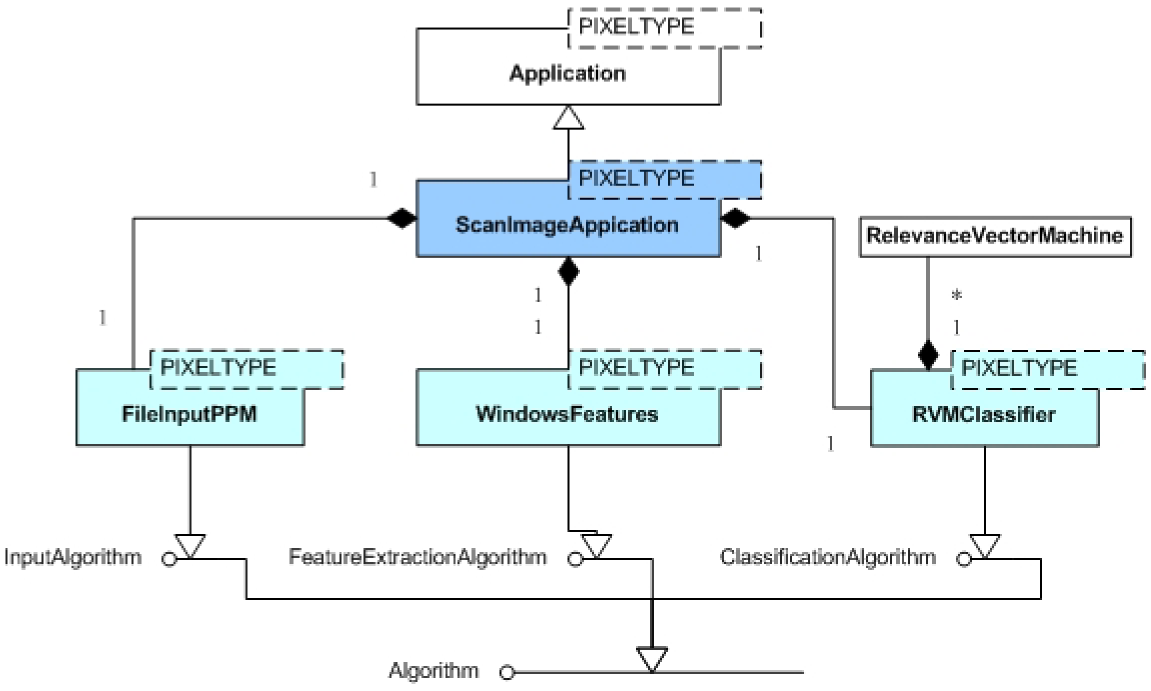
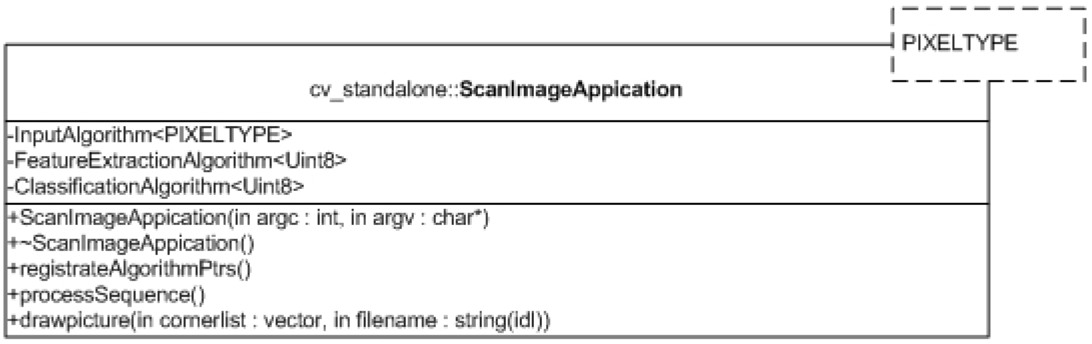


Figure 10.2: Class diagram of visualization utility case.

10.1.1 Individual class description

ScanImageApplication Class

This is the visualization application class, which combines all needed algorithms together to realize RVM classification results visualization. Since the output algorithms are not suitable for our case, so we did not use them to output our results. The visualization result is output by a member function in this class.



```
cv_standalone::ScanImageAppication
-InputAlgorithm<PIXELTYPE>
-FeatureExtractionAlgorithm<Uint8>
-ClassificationAlgorithm<Uint8>
+ScanImageAppication(in argc : int, in argv : char*)
+~ScanImageAppication()
+registrateAlgorithmPtrs()
+processSequence()
+drawpicture(in comerlist : vector, in filename : string(idl))
```

10.1.2 Sequence diagram

Fig. 10.3 shows the sequence diagram of visualization utility. The classification procedure likes UI-Wand application’s one, but the feature extraction algorithms they used are different. The UI-Wand application uses *AnnotionFeatures* to extract the feature vectors of sub-image by giving the special pixel positions (possible screen corner positions) as the center of the sub-images, but *windowsFeatures* class will extract the feature vectors of sub-images by scanning a whole frame. Therefore, if target window step size is set as 1 pixel, then almost every position in the frame will be classified except some margins. So with this classification results we can write a picture file, giving different color to every pixel, which results in the impressive result images shown in Chapter 4.

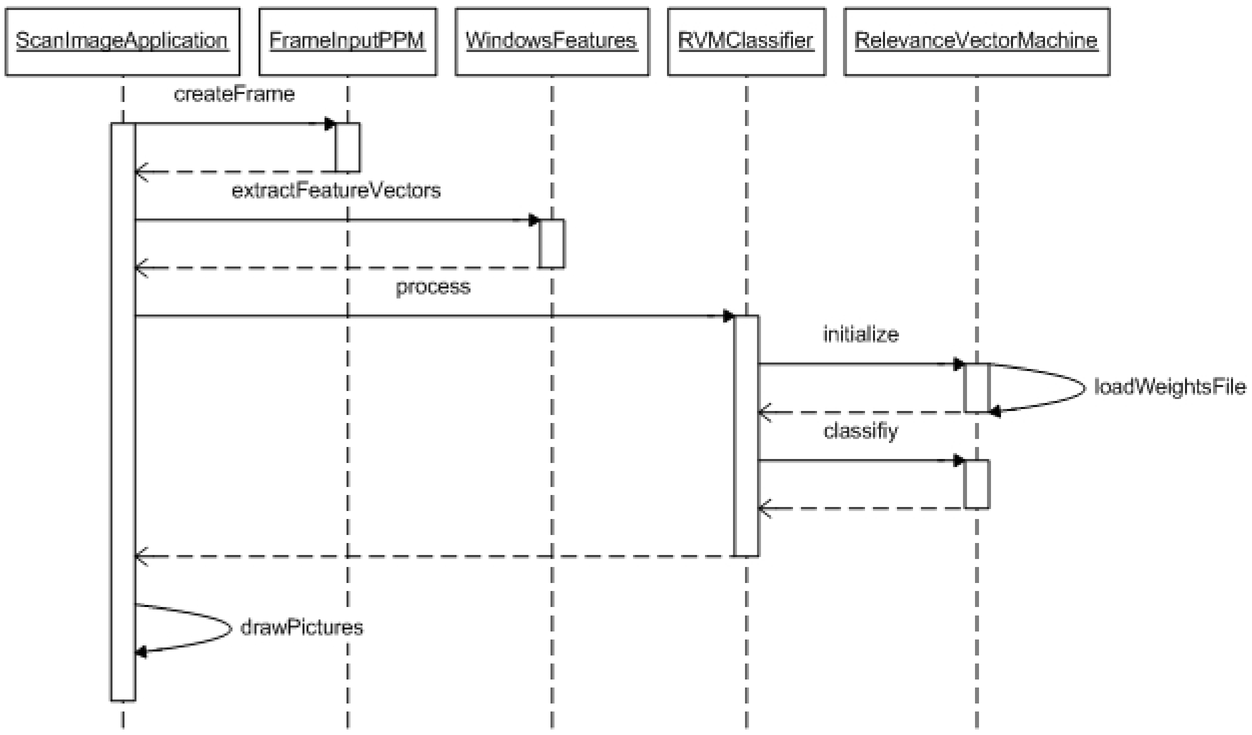


Figure 10.3: Sequence diagram of visualization application.

10.2 UI-Wand application error analysis utility

In order to test the performance of our algorithms and do some analysis about the parameter optimization, we designed an error analysis utility. Users can specify the reference screen corner points by the GUI of this utility and the utility can analyze these reference points together with the result of our UI-Wand system, then some analysis tables can be given by the utility. These tables give users very clear outline for each important quality. In the following part of this section, we will introduce this utility in details.

10.2.1 Utility GUI introduction

The graphical user interface (GUI) of the utility is shown in Fig. 10.4:

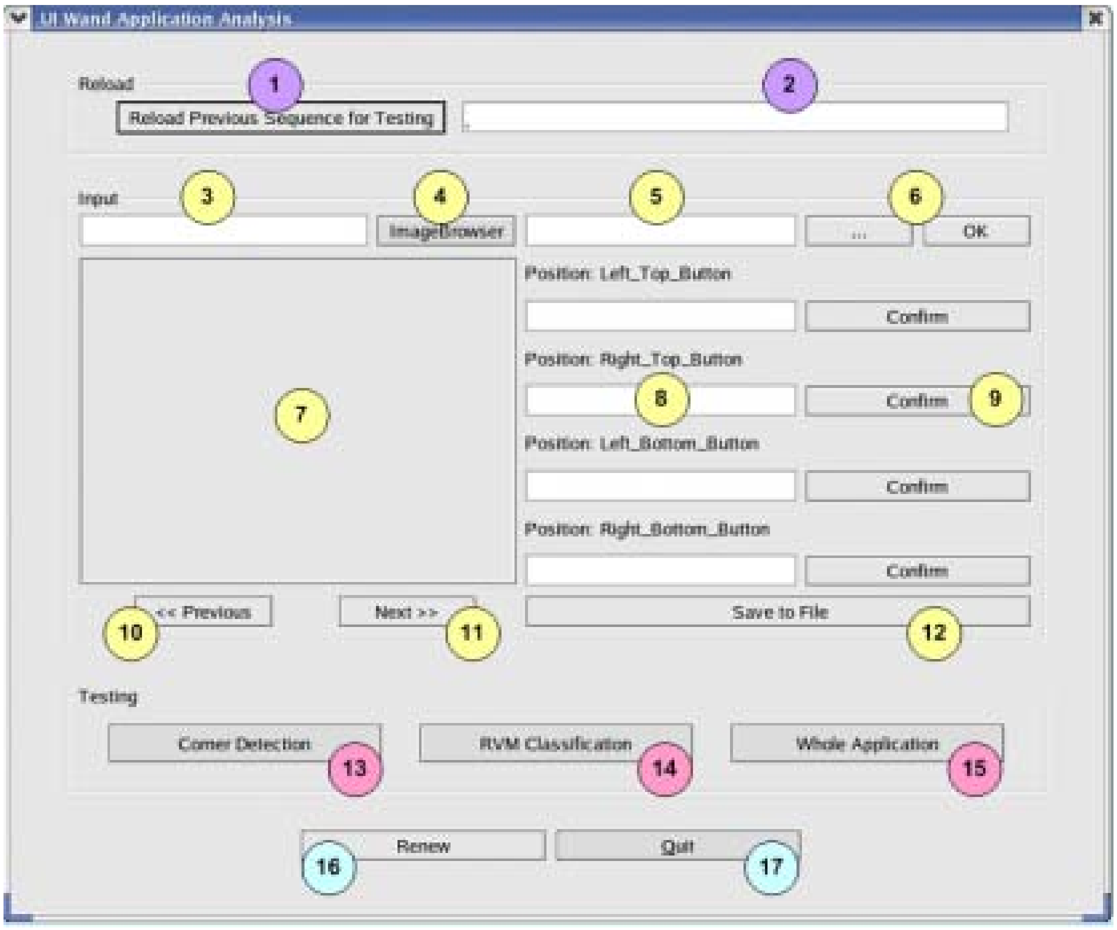


Figure 10.4: Reference points and error analysis GUI.

The graphical interface is mainly divided into three function parts: “Reload” part, “Input” part and “Testing” part. The functions of these parts and the signed labels in each part are explained in the following:

“Reload” part: reload the previous image sequence and the corresponding reference points.

“1” Browse the previous image sequence, which was analyzed already, and reload it to the utility for analyzing again.

“2” Show the directory and file name of reload sequence.

“Input” part: draw the reference points in the selected image sequence and save them into a file. If the reload button was clicked before, this part will show the reloaded sequence with its reference points. The GUI during drawing the reference corner points is shown in Fig. 10.5.

“3” Show the directory and file name of browed image.

“4” Select the new image sequence for analyzing; if you push **“1”** button before, this button is used for reload the previous image sequence and its corresponding reference points into **“7”.**

“5” Show the file directory and name, where the reference corner points will be saved.

“6” Choose the save directory and confirm the file name that reference corner points will be saved.

“7” Show the image sequence that is browsed by **“3”** and draw the reference points in it, the position information will be shown in **“8”**.

“8” Show the reference point position x, y in the 2D image plot.

“9” Confirm the reference point position information.

“10” Browse the previous image of the image that is being shown in **“7”**.

“11” Browse the next image of the image that is being shown in **“7”**.

“12” Save the confirmed reference points information to the file, which the user already chose by **“6”**.

“Testing” part: run the three analyses and show the result table in the other windows.

“13” Run the corner detection algorithm analysis.

“14” Run the RVM classification algorithm analysis.

“15” Run the whole application analysis.

Others: control whole utility.

“16” Renew the utility.

“17” Quit the utility.

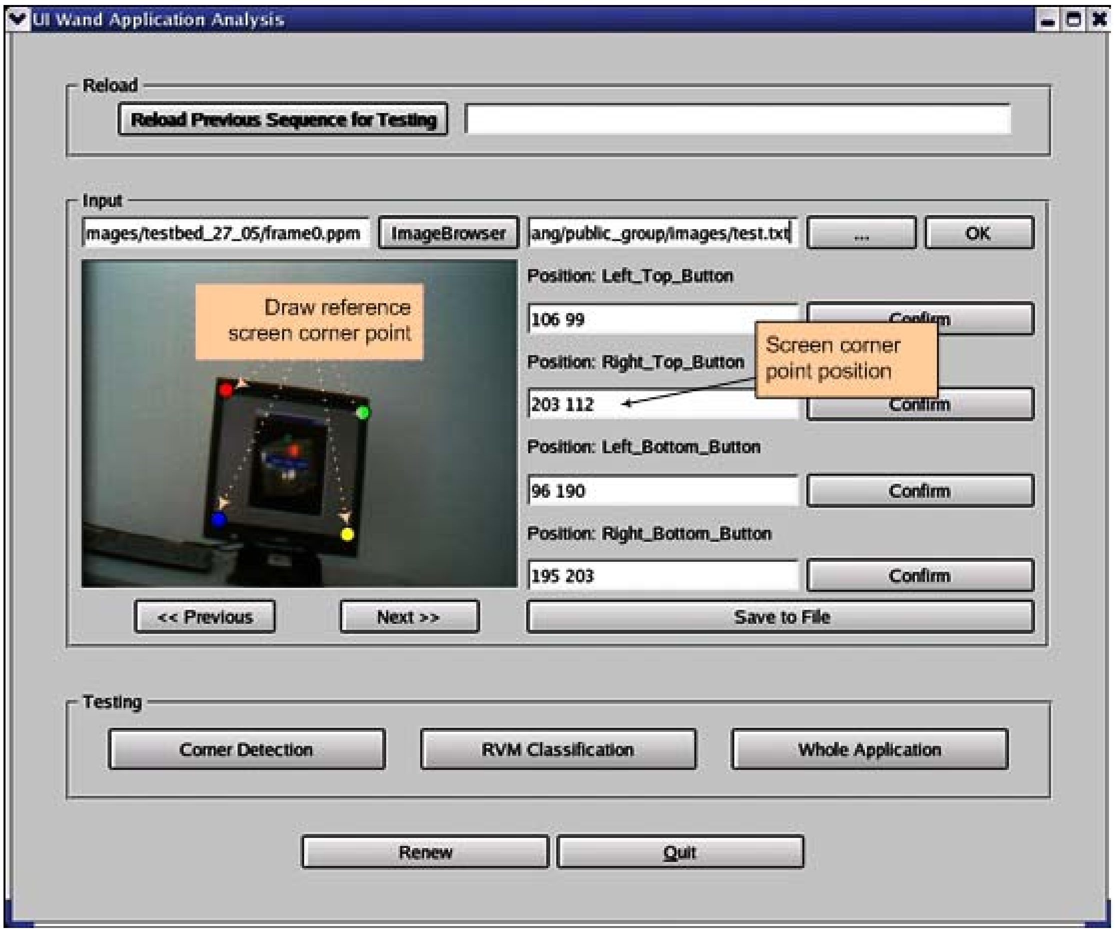

UI Wand Application Analysis
Reload
Reload Previous Sequence for Testing
Input
mages/testbed_27_05/frame0.ppm
ImageBrowser
ang/public_group/images/test.txt
...
OK
Draw reference screen corner point
Position: Left_Top_Button
106 99
Confirm
Screen corner point position
Position: Right_Top_Button
203 112
Confirm
Position: Left_Bottom_Button
96 190
Confirm
Position: Right_Bottom_Button
195 203
Confirm
<< Previous
Next >>
Save to File
Testing
Corner Detection
RVM Classification
Whole Application
Renew
Quit

Figure 10.5: GUI of reference points utility during assigning the reference corner points.

The operation flow of the whole utility is mainly divided into the following two function flows and each flow contains different operations in the GUI for the different intentions. The two flows are:

Reload analysis flow: The user flow chart is shown in Fig. 10.6 (orange flow).
In this flow, the utility will reload the reference corner points in a previous image sequence for analyzing again.

New analysis flow: The user flow chart is shown in Fig. 10.6 (blue flow).
In this flow, the utility will draw the reference corner points in a new image sequence for new analyzing.

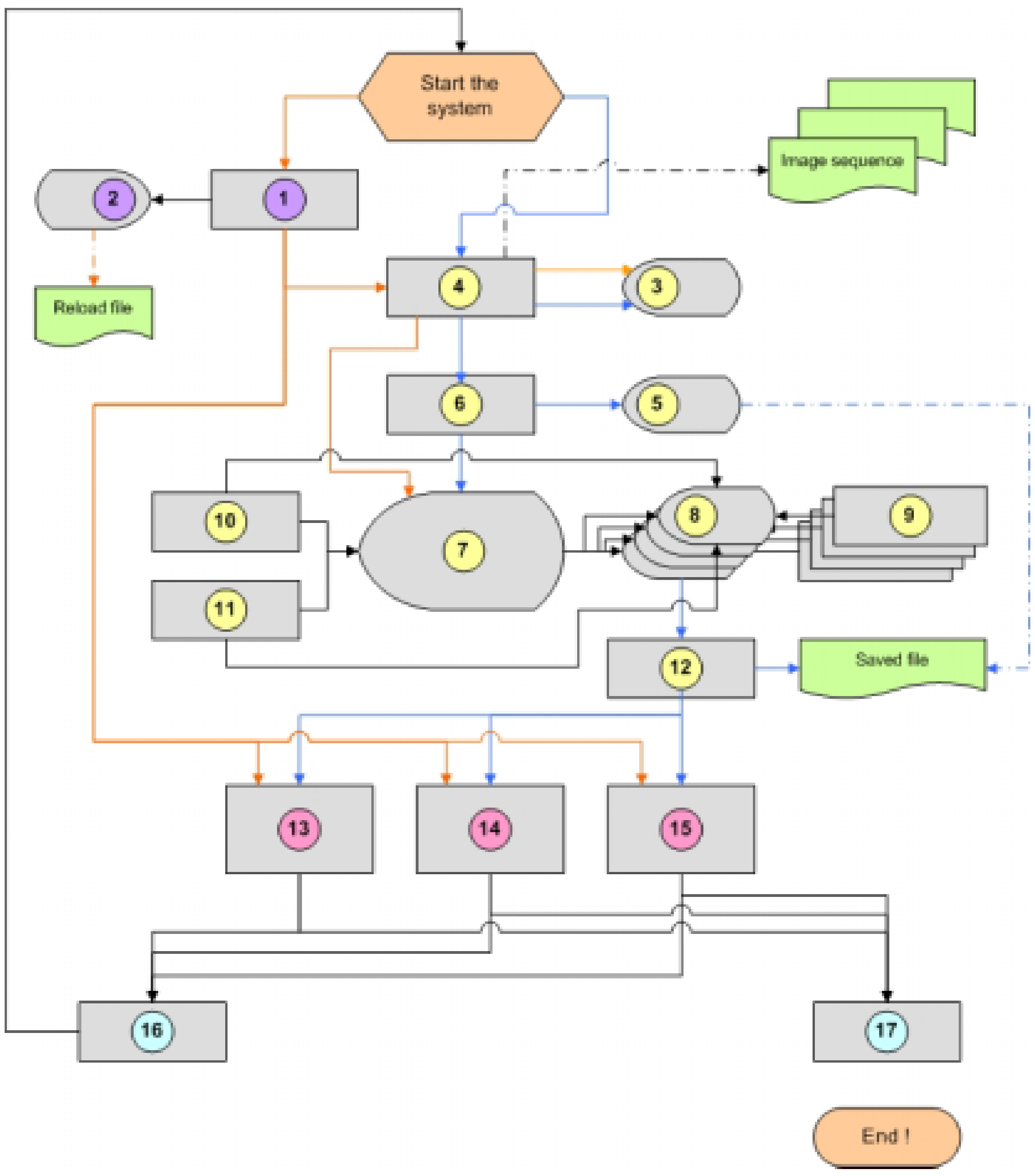


Figure 10.6: Function flow of the analysis utility GUI (reload analysis flow—orange line; new analysis flow—blue line; shared flow line—black line).

10.2.2 Error analysis utility introduction

The main part of this error analysis utility is to give the reference screen corner positions and run the three error analysis programs (Sojka corner detection analysis, RVM corner classification analysis and whole application analysis). So we can divide this utility into two modules: one is reference screen corner positions specification module and the other one is error analysis module.

The reference screen corner positions specification module is used for the users to give the reference screen corner positions. This module will display frames in the GUI and let users draw the reference screen corners directly in the GUI, finally it will save the reference positions to a text file. This module also has reload ability, which can make it easier for users to reload the previous specified frames result whenever they need.

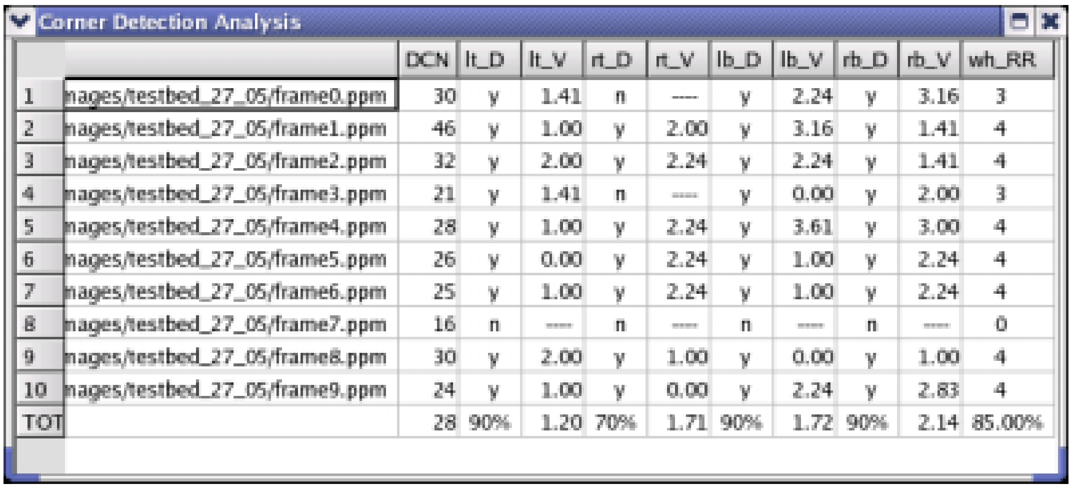
The other module in the utility is the error analysis module. By pressing the three analysis buttons, the error analysis module will do some analysis work and give the analysis table after that. Because this module is more complex, we will discuss it in the next section.

10.2.3 Error analysis module

In order to test on our main algorithms and do some data analysis work, we designed three analysis functions in this module, which can be called by separate function buttons on our GUI.

10.2.3.1 Corner detection error analysis

This function can get the detection result data from Sojka corner detector and compare it with the reference screen corner points. The function first reads the result data from some text files (one detect result file per image) and then does some calculation work with the corner data from the reference corner file. The analysis results will be shown by some parameters in a table (see Fig. 10.7). From this table we can see the performance of the Sojka corner detector.



		DCN	lt_D	lt_V	rt_D	rt_V	lb_D	lb_V	rb_D	rb_V	wh_RR
1	nages/testbed_27_05/frame0.ppm	30	y	1.41	n	----	y	2.24	y	3.16	3
2	nages/testbed_27_05/frame1.ppm	46	y	1.00	y	2.00	y	3.16	y	1.41	4
3	nages/testbed_27_05/frame2.ppm	32	y	2.00	y	2.24	y	2.24	y	1.41	4
4	nages/testbed_27_05/frame3.ppm	21	y	1.41	n	----	y	0.00	y	2.00	3
5	nages/testbed_27_05/frame4.ppm	28	y	1.00	y	2.24	y	3.61	y	3.00	4
6	nages/testbed_27_05/frame5.ppm	26	y	0.00	y	2.24	y	1.00	y	2.24	4
7	nages/testbed_27_05/frame6.ppm	25	y	1.00	y	2.24	y	1.00	y	2.24	4
8	nages/testbed_27_05/frame7.ppm	16	n	----	n	----	n	----	n	----	0
9	nages/testbed_27_05/frame8.ppm	30	y	2.00	y	1.00	y	0.00	y	1.00	4
10	nages/testbed_27_05/frame9.ppm	24	y	1.00	y	0.00	y	2.24	y	2.83	4
TOT		28	90%	1.20	70%	1.71	90%	1.72	90%	2.14	85.00%

Figure 10.7: Corner detection analysis table.

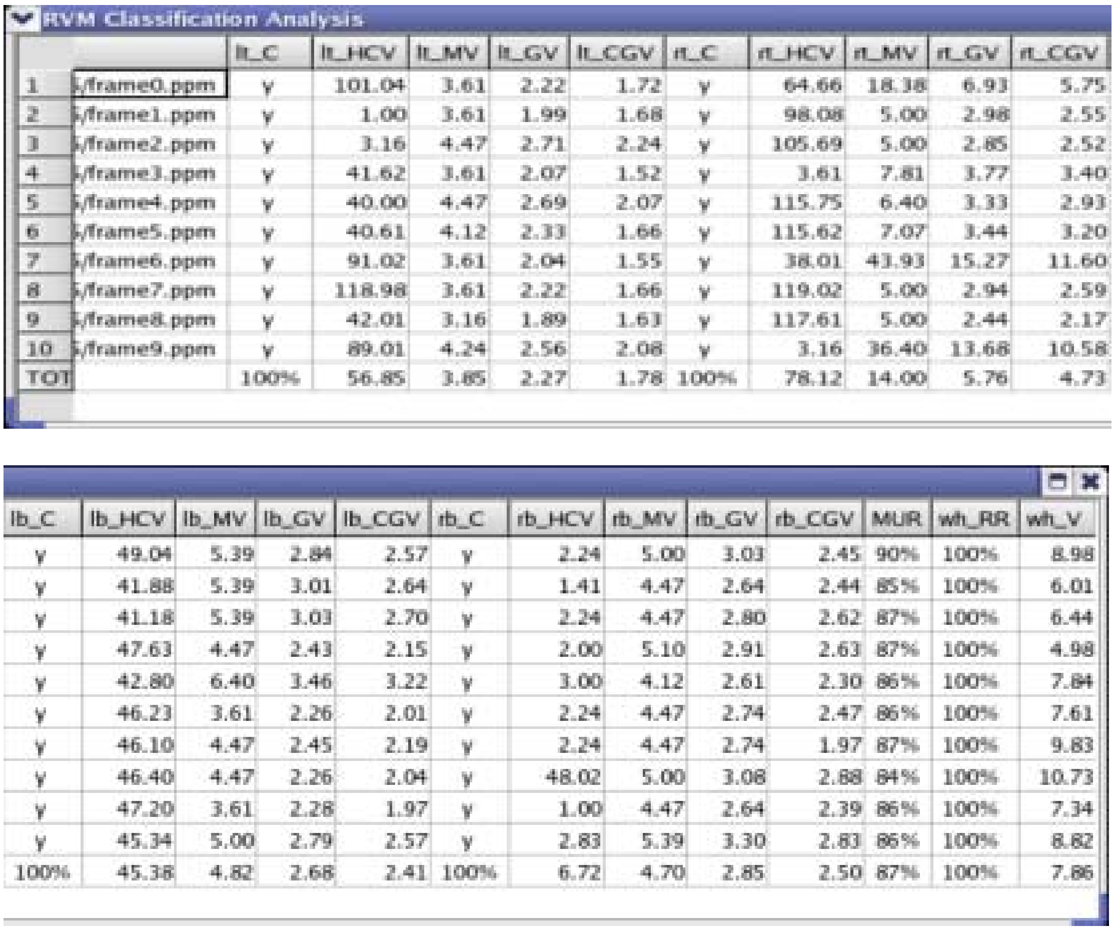
We use the following parameters in the table:

- DCN: detect corner number by Sojka corner detector.
- lt_D: left top corner detection status ("y" if detected any corner within a circle neighborhood of five pixels radius, which is centered at the reference left top corner).
- lt_V: nearest distance variance form detected left top corner to reference left top corner.

- rt_D: right top corner detection status (“y” if detected any corner within a circle neighborhood of five pixels , which is centered at the reference right top corner).
- rt_V: nearest distance variance form detected right top corner to reference right top corner.
- lb_D: left bottom corner detection status (“y” if detected any corner within a circle neighborhood of five pixels radius, which is centered at the reference left bottom corner).
- lb_V: nearest distance variance form detected left bottom corner to reference left bottom corner.
- rb_D: right bottom corner detection status (“y” if detected any corner within a circle neighborhood of five pixels radius, which is centered at the reference right bottom corner).
- lt_V: nearest distance variance form detected right bottom corner to reference right bottom corner.
- wh_RR: whole detection corner number in an image.

10.2.3.2 RVM classification error analysis

This function can get the classification result data from RVM classifier and compare it with the reference corner points. The function first read the result data from some text files (one classification result file per image) and then does some calculation work with the corner data from the reference file. The analysis results will be shown by some parameters in a table (see Fig. 10.8, we cut the table into two images). From this table we can see the performance of the RVM classifier.



		lt_C	lt_HCV	lt_MV	lt_GV	lt_CGV	rt_C	rt_HCV	rt_MV	rt_GV	rt_CGV	lb_C	lb_HCV	lb_MV	lb_GV	lb_CGV	rb_C	rb_HCV	rb_MV	rb_GV	rb_CGV	MUR	wh_RR	wh_V
1	\$/frame0.ppm	y	101.04	3.61	2.22	1.72	y	64.66	18.38	6.93	5.75	y	49.04	5.39	2.84	2.57	y	2.24	5.00	3.03	2.45	90%	100%	8.98
2	\$/frame1.ppm	y	1.00	3.61	1.99	1.68	y	98.08	5.00	2.98	2.55	y	41.88	5.39	3.01	2.64	y	1.41	4.47	2.64	2.44	85%	100%	6.01
3	\$/frame2.ppm	y	3.16	4.47	2.71	2.24	y	105.69	5.00	2.85	2.52	y	41.18	5.39	3.03	2.70	y	2.24	4.47	2.80	2.62	87%	100%	6.44
4	\$/frame3.ppm	y	41.62	3.61	2.07	1.52	y	3.61	7.81	3.77	3.40	y	47.63	4.47	2.43	2.15	y	2.00	5.10	2.91	2.63	87%	100%	4.98
5	\$/frame4.ppm	y	40.00	4.47	2.69	2.07	y	115.75	6.40	3.33	2.93	y	42.80	6.40	3.46	3.22	y	3.00	4.12	2.61	2.30	86%	100%	7.84
6	\$/frame5.ppm	y	40.61	4.12	2.33	1.66	y	115.62	7.07	3.44	3.20	y	46.23	3.61	2.26	2.01	y	2.24	4.47	2.74	2.47	86%	100%	7.61
7	\$/frame6.ppm	y	91.02	3.61	2.04	1.55	y	38.01	43.93	15.27	11.60	y	46.10	4.47	2.45	2.19	y	2.24	4.47	2.74	1.97	87%	100%	9.83
8	\$/frame7.ppm	y	118.98	3.61	2.22	1.66	y	119.02	5.00	2.94	2.59	y	46.40	4.47	2.26	2.04	y	48.02	5.00	3.08	2.88	84%	100%	10.73
9	\$/frame8.ppm	y	42.01	3.16	1.89	1.63	y	117.61	5.00	2.44	2.17	y	47.20	3.61	2.28	1.97	y	1.00	4.47	2.64	2.39	86%	100%	7.34
10	\$/frame9.ppm	y	89.01	4.24	2.56	2.08	y	3.16	36.40	13.68	10.58	y	45.34	5.00	2.79	2.57	y	2.83	5.39	3.30	2.83	86%	100%	8.82
TOT		100%	56.85	3.85	2.27	1.78	100%	78.12	14.00	5.76	4.73	100%	45.38	4.82	2.68	2.41	100%	6.72	4.70	2.85	2.50	87%	100%	7.86

Figure 10.8: RVM corner classification analysis table.

We use the following parameters in the result table:

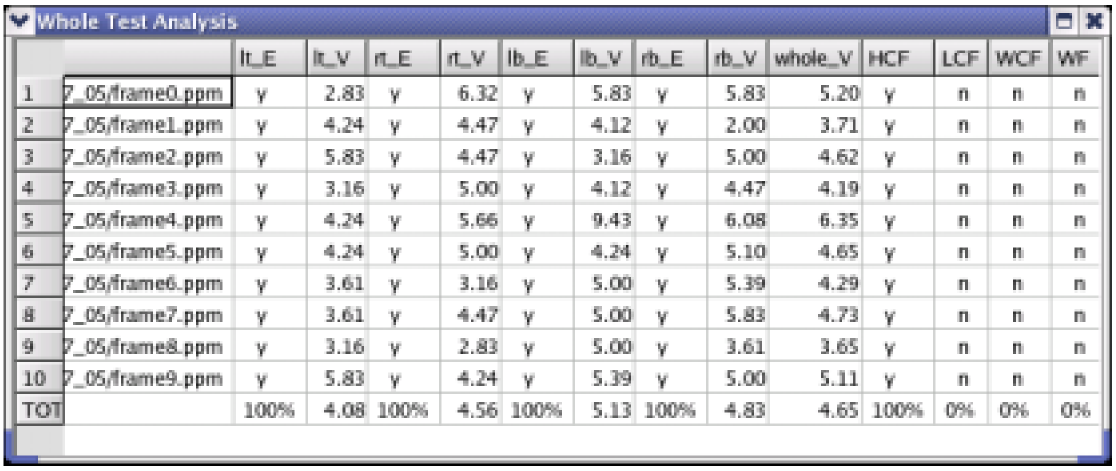
- lt_C: left top corner classification status ("y" if there exists any corner classified into left top corner class within a circle neighborhood of 2 pixels radius, which is centered at the reference left top corner).
- lt_HCV: the distance variance from the point that is classified into left top corner by RVM and with the highest confidence to the reference left top corner.
- lt_MV: the maximum distance variance from the point that is within the cluster that is classified into left top corner by RVM to the reference left top corner (the cluster is defined as a points gathering covering the corresponding reference corner points and in the cluster all the points are neighbored).
- lt_GV: geometric center of the left top corner cluster.
- lt_CGV: geometric center weighted by each point's classification confidence of the left top corner cluster.
- rt_C: right top corner classification status ("y" if there exists any corner classified into right top corner class within a circle neighborhood of 2 pixels radius, which is centered at the reference right top corner).

- rt_HCV: the distance variance from the point that is classified into right top corner by RVM and with the highest confidence to the reference right top corner.
- rt_MV: the maximum distance variance from the point that is within the cluster that is classified into right top corner by RVM to the reference right top corner (the cluster is defined as a points gathering covering the corresponding reference corner points and in the cluster all the points are neighbored).
- rt_GV: geometric center of the right top corner cluster.
- rt_CGV: geometric center weighted by each point's classification confidence of the right top corner cluster.
- lb_C: left bottom corner classification status ("y" if there exists any corner classified into left bottom corner class within a circle neighborhood of 2 pixels radius, which is centered at the reference left bottom corner).
- lb_HCV: the distance variance from the point that is classified into left bottom corner by RVM and with the highest confidence to the reference left bottom corner.
- lb_MV: the maximum distance variance from the point that is within the cluster that is classified into left bottom corner by RVM to the reference left bottom corner (the cluster is defined as a points gathering covering the corresponding reference corner points and in the cluster all the points are neighbored).
- lb_GV: geometric center of the left bottom corner cluster.
- lb_CGV: geometric center weighted by each point's classification confidence of the left bottom corner cluster.
- rb_C: right bottom corner classification status ("y" if there exists any corner classified into right bottom corner class within a circle neighborhood of 2 pixels radius, which is centered at the reference right bottom corner).
- rb_HCV: the distance variance from the point that is classified into right bottom corner by RVM and with the highest confidence to the reference right bottom corner.
- rb_MV: the maximum distance variance from the point that is within the cluster that is classified into right bottom corner by RVM to the reference right bottom corner (the cluster is defined as a points gathering covering the corresponding reference corner points and in the cluster all the points are neighbored).
- rb_GV: geometric center of the right bottom corner cluster.
- rb_CGV: geometric center weighted by each point's classification confidence of the right bottom corner cluster.
- MUR: misunderstanding rate of the classification (number of points that should be non-corner but are classified into screen corners divided by the number of all points that is classified into screen corners).
- wh_RR: whole classification right rate of the screen corner (the number of clusters that cover the right corner divided by the number of screen corners).

- wh_V: weighted average variance of HCV, MV, GV, CGV.

10.2.3.3 Whole application error analysis

This function can get the detection result data from the application and compare it with the reference corner points. The function first read the result data from some text files (one detect result file per image) and then does some calculation work with the corner data from reference file. The analysis results will be shown by some parameters in a table (see Fig. 10.9). From this table we can see the performance of our Candidates-Winners approach.



		lt_E	lt_V	rt_E	rt_V	lb_E	lb_V	rb_E	rb_V	whole_V	HCF	LCF	WCF	WF
1	7_05/frame0.ppm	y	2.83	y	6.32	y	5.83	y	5.83	5.20	y	n	n	n
2	7_05/frame1.ppm	y	4.24	y	4.47	y	4.12	y	2.00	3.71	y	n	n	n
3	7_05/frame2.ppm	y	5.83	y	4.47	y	3.16	y	5.00	4.62	y	n	n	n
4	7_05/frame3.ppm	y	3.16	y	5.00	y	4.12	y	4.47	4.19	y	n	n	n
5	7_05/frame4.ppm	y	4.24	y	5.66	y	9.43	y	6.08	6.35	y	n	n	n
6	7_05/frame5.ppm	y	4.24	y	5.00	y	4.24	y	5.10	4.65	y	n	n	n
7	7_05/frame6.ppm	y	3.61	y	3.16	y	5.00	y	5.39	4.29	y	n	n	n
8	7_05/frame7.ppm	y	3.61	y	4.47	y	5.00	y	5.83	4.73	y	n	n	n
9	7_05/frame8.ppm	y	3.16	y	2.83	y	5.00	y	3.61	3.65	y	n	n	n
10	7_05/frame9.ppm	y	5.83	y	4.24	y	5.39	y	5.00	5.11	y	n	n	n
TOT		100%	4.08	100%	4.56	100%	5.13	100%	4.83	4.65	100%	0%	0%	0%

Figure 10.9: Whole application analysis table.

We use the following parameters in the result table:

- lt_E: left top corner existing status ("y" if the left top corner is detected).
- lt_V: the distance variance from the detected left top corner to the reference left top corner.
- rt_E: right top corner existing status ("y" if the right top corner is detected).
- rt_V: the distance variance from the detected right top corner to the reference right top corner.
- lb_E: left bottom corner existing status ("y" if the left bottom corner is detected).
- lb_V: the distance variance from the detected left bottom corner to the reference left bottom corner.
- rb_E: right bottom corner existing status ("y" if the right bottom corner is detected).
- rb_V: the distance variance from the detected right bottom corner to the reference right bottom corner.
- whole_V: average distance variance of lt_V, rt_V, lb_V, rb_V.
- HCF: have reference corner frame ("y" if there are screen corner in the original image).

- LCF: lost real corner frame ("y" if the original image has reference screen corners but our application can not detect them)
- WCF: wrong detection corner frame ("y" if the original image have no screen corner but our application wrongly detect corners).
- WF: wrong detection frame ("y" if this frame are wrongly detected in any case).

10.2.3.4 Utility implementation

We used QT Designer software under Linux to implement the UI-Wand application error analysis utility. The GUI was designed base on the utility requirement and the two modules in this utility are all implemented based on the basic classes of QT Designer.

Part IV

System Tests and Future Works

11

System Tests

We divided our system test into two parts: off-line test and on-line test. The off-line test focuses on testing our Candidates-Winners approach without capturing image sequences from the camera and mouse positioning process. From off-line test, we want to show the different effects of the algorithms that we used in our Candidates-Winners approach and confirm that this approach can accurately detect screen corner. The on-line test performs test of the whole integrated UI-Wand system, which will capture image sequence from the camera, detect screen corners and find out the pointing position then drive the mouse cursor. From the on-line test, we want to show that the UI-Wand system developed by us can work on real-time in the real environment.

11.1 Off-line tests

In this section we will introduce the test environment of the off-line test including hardware and software conditions. Then the test goal, test materials and the specified system parameters will be clearly described. Finally we show the test results and do some analysis.

11.1.1 Off-line test environment introduction

Hardware conditions:
Machine for test: Pentium IV-2.4G, 256M memory/Linux.

Software conditions:
Test system: Red Hat Linux.
Error analysis tool: UI-Wand application error analysis utility.

11.1.2 Off-line test goal

In off-line test, we mainly test our Candidates-Winners approach, because it is the most crucial part of our system. By this test, we want to know the effect of every algorithm or model used to the screen corner detection in this approach, for example, if the RVM can really classify and reduce the candidate corners, if the system can detect the four screen corners accurately and how fast the algorithms are. In a nutshell, we want to prove that our screen corners detection approach can satisfy the requirements as we defined in Chapter 2.

11.1.3 Off-line test sequences selection

We used two categories of frame sequences for our off-line test, one category contains three long sequences (each has 600 frames) and the other category contains ten short sequences (each has 20-30 frames). These test sequences are all captured in front of the computer screen for demonstration (PHILIPS brilliance 180P2 black LCD computer screen).

Three long test sequences: We used UI-Wand system to get three sequences from UI-Wand in the real demonstration environment. Each sequence contains 600 frames and is captured in real-time (10 frames per second). In each sequence, some gesture actions are made and these gesture actions cover most possible movements what we expect users to do when using our UI-Wand system. These three sequences are used for speed test of our Candidates-Winners approach, because they are continuously captured and processed by our system.

Here we use "BS" to stand for the long sequence. The special movements in the three big sequences are:

LS1 which contains movements of picking up the UI-Wand and pointing to the screen, moving UI-Wand pointing position from one place to another place within screen, pointing UI-Wand to the screen and pushing forward and backward, pointing UI-Wand to the screen and rotating it.

LS2 which contains movements of pointing UI-Wand to the screen and moving UI-Wand to left and right within small range, pointing UI-Wand to the screen and moving it up and down within small range, pointing UI-Wand to the screen and moving it as a cross trace.

LS3 which contains movements of pointing UI-Wand to the screen and moving it to left and right within large range, pointing UI-Wand to the screen and moving it up and down within large range, pointing UI-Wand to the screen and moving it to point on a place outside the screen.

Ten short test sequences: The other ten small sequences we used in the off-line test part are selected from these above three long sequences by hand. Each of these short sequences contains one special movement. Totally, the ten sequences cover most of the possible movements that the user can do. These ten sequences are used for accuracy test of our Candidates-Winners approach, since they show most of the possible movements in using our UI-Wand system.

We briefly use "SS" to stand for the short test sequence. The special movements in the ten small sequences are:

SS1 contains movement of picking up the UI-Wand and pointing to somewhere in the screen.

SS2 contains movement of pointing UI-Wand to the screen and pushing it forward.

SS3 contains movement of pointing UI-Wand to the screen and pulling it backward.

SS4 contains movement of pointing UI-Wand to the screen and moving to left and right within small range.

SS5 contains movement of pointing UI-Wand to the screen and moving up and down within small range.

SS6 contains movement of pointing UI-Wand to the screen and moving it as a cross trace.

SS7 contains movement of pointing UI-Wand to the screen and moving to left and right within large range.

SS8 contains movement of pointing UI-Wand to the screen and moving up and down within large range.

SS9 contains movement of pointing UI-Wand to the screen and left rotating it.

SS10 contains movement of pointing UI-Wand to the screen and right rotating it.

11.1.4 Off-line test results

Before off-line test, we specified the parameters file (see Appendix D.1) of the UI-Wand system. These parameters are set for the three long test sequences.

After testing, we divided the result of our off-line test into Table 11.1 and Table 11.2. Table 11.1 shows the accuracy rate of our Candidates-Winners approach and the detection number in each step of this structure, from which the improvement made by our Candidates-Winners approach can be clearly seen. Table 11.2 shows the process speed of our Candidates-Winners approach and from this table we can see the speed acceleration by using the tracking filter.

Table 11.1: Algorithms combination test in Candidates-Winners approach.

Seq.	**Corner detection**	**Corner detection +RVM classification**	**Corner detection + RVM classification + Filter**	
	Corner number	Corner number	Variance	Rates
SS1	82	16	2.09	87%
SS2	78	22	1.60	100%
SS3	78	23	1.52	100%
SS4	79	17	1.81	100%
SS5	50	10	1.83	75%
SS6	69	13	2.12	93%
SS7	61	11	4.90	70%
SS8	60	10	9.57	90%
SS9	78	20	1.72	100%
SS10	66	18	1.46	100%

Table 11.2: Sequence processing speed test (data is in millisecond).

Seq.	**Corner detection**	**Corner detection + RVM classification**		**Corner detection + RVM classification + Filters**			
	DT	FE	RVM	DT	FE	RVM	WHOLE
LS1	138.80	127.27	60.93	27.93	51.77	9.07	88.77
LS2	127.00	132.82	61.65	29.60	54.98	7.92	92.5
LS3	113.05	123.70	63.03	42.87	52.95	9.83	105.65
AVR	126.28	127.93	61.87	33.47	53.23	8.94	95.64

*DT: corner detection time
FE: feature extraction time
RVM: RVM classification time
WHOLE: whole structure processing time

11.1.5 Off-line test result analysis

Both the test results in Table 11.1 and Table 11.2 clearly show the performance by using our Candidates-Winners approach.

In Table 11.1 we can see when we combine the RVM classification algorithm together with the Sojka corner detection algorithm, the detected corner number will be decreased a lot. When we combine all algorithms in Candidates-Winners approach together, we can accurately detect only the four screen corners from the sequences. But for some sequences such as SS1, SS5, SS6, SS7 and SS8, the final detection rates are not so high compared to the results of other sequences. This is because there are fast movements in the sequences. If a movement is too fast, the captured frames will be very blur and our Candidates-Winners approach cannot accurately detect the screen corners. But we can conclude that our Candidates-Winners approach can handle normal movement sequences very well. For the fast movement, although the detections are not accurate, but as what we mentioned in Chapter 7, the tracking filter will at least predict correct direction information, which is enough for fast moving UI-Wand.

From Table 11.2, we can see the obviously speed up performance by our Candidates-Winners approach. If we only use Sojka corner detection algorithm or its combination with RVM classification algorithm, the processing speed is not satisfied the real-time requirement. But if we use ROI tracking algorithm, the total processing time will be shorter, because the corner number for feature extraction and classification is decreased quite a lot. This improvement can lead our system to satisfy the real-time processing requirement.

11.2 On-line tests

11.2.1 On-line test environment introduction

Hardware conditions:

Test monitor: A PHILIPS brilliance 180P2 black LCD computer screen
Test machine: Pentium IV-2.4G CPU and 256M memory
Devices: UI-Wand hardware components (see Chapter 2)

Software conditions:
Test system: Window XP

Room conditions:

A normal working room with changeable light conditions in PHILIPS Research Laboratories Aachen (see Fig. 11.1)

11.2.2 On-line test goals

Finally we realized the UI-Wand system prototype for computer mouse control as we earlier defined in Chapter 1. The goal of our on-line test is to evaluate the performance of our UI-Wand system in a real environment. In this test, we embedded all the algorithms (camera input algorithm + pointing positioning algorithms+ mouse driver) together, so the test data we got should correctly reflect the final performance of our UI-Wand system.

11.2.3 On-line test procedure

Firstly we set up the whole UI-Wand system and then handle the UI-Wand in different valid positions, which can be close to the screen (>0.9m), far from screen (<4m) and with different left or right pointing angle. We also change the light conditions of the room during our test. So our on-line test can cover the possible conditions of using the UI-Wand and the result we got can present the real performance of the whole system. Examples of on-line test environments are shown in Fig. 11.1.

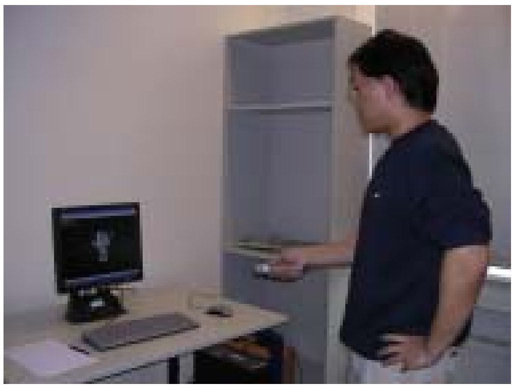
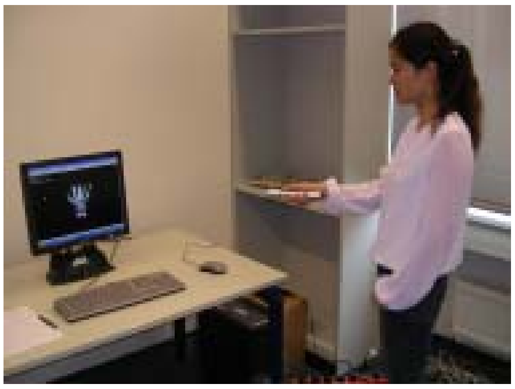



Figure 11.1: On-line test environments examples (the left image is captured in a dark condition and the right image is captured in a bright condition).

11.2.4 On-line test result and result analysis

The parameters file for on-line test is listed in Appendix D.2. Since we cannot use our Error Analysis utility to analyze screen corner detection results in on-line test, so we will describe in section 11.2.5 by text. The only useful result data can present here is the speed of system. We can see the speed data getting during using the UI-Wand from Table 11.3. The speed is shown as frames per second (fps).

Table 11.3: On-line test speed results.

No.	Number of frames	Processing speed (fps)
1	813	10.11
2	875	10.92
3	1381	9.3
4	1490	11.35
5	1524	10.98
6	2186	10.93
7	2328	11.27
8	2420	11.79
9	2758	12.63
10	3309	10.32
AVR	1908	10.96

11.2.5 On-line test result analysis

After on-line test, our UI-Wand system is proved that it can work well within valid pointing distance, directions and a certain lighting changes. The cursor can move along the pointing trace on the screen in real-time base on our hardware conditions now (around 11fps), which can satisfy the real-time processing speed defined in our problem definition part in Chapter 2. From observation, the performance accuracy rate of our system is quite good, in most of cases even fast speed, our system can get an ideal positioning result. But there are some problems of the system: if the background of the screen and the light condition of the room change too much, the positioning results are not so good. These problems are mainly caused by the training dataset currently used for training RVM model, which is not too big and cannot cover enough variant corners. So in the future we can use more training materials under different conditions that may happen in the real world, then the performance of the system should be highly improved.

11.3 Conclusions

After the off-line and on-line tests for our UI-Wand system, the results show a good performance to our UI-Wand system, which satisfies the system requirements that we defined in Chapter 2. Our Candidates-Winners approach used in UI-Wand system can detect the four screen corners with high accuracy rate and fast detection speed, which guarantees the integrated UI-Wand system can work well.

But we still find some problems during test our system. First, the image quality affects the detection accuracy rate, for example, if the UI-Wand movements in the sequence are too fast, the captured frames may be blur, then the Candidates-Winners approach will lost screen corners in some frames. This can lead our system to lose the pointing positions. The resolution of this problem is that we can use a better camera in UI-Wand, which can capture frames in a moving mode with the higher quality. The other problem of our system is that the screen content and the room environment (e.g. strong light conditions changes) affect the performance of the system. If the screen contains a lot of application windows or the monitor is located in front of a complex background, the detection result could be wrong and causes the wrong positioning result. The improvement is to increase the training materials for the RVM model, which can cover more variant conditions that may happen in the real world.

12 Conclusions and Future Works

So far, we have presented all works in this project. All chapters before gave very detailed explanations about a Candidates-Winners approach to detect screen corners, based upon which a novel pointing device of PHILIPS, UI-Wand, manages to estimate the screen and realize pointing positioning on a real-time level. In addition to the pointing positioning functionality, in Chapter 8, we described a new approach based on RVM to recognize gestures, which was proved a promising method by the test results in that chapter. Finally, all the models and algorithms are implemented on Vispirin and got satisfying test results. But, that is not all, in future, there are many algorithms can be improved and many new models or approaches can be utilized in the system. In this chapter we will summarize our current models and algorithms and purpose some works to improve the system in the future.

12.1 Conclusions

The problem of this project for us is to find a way to detect screen corners, by which a camera can estimate the screen position so that the pointing position of the camera can be figured out. After investigation, we found that there are many algorithms or models suitable for the corner detection problem. The most direct way is to use the corner detection algorithm, which can directly research out the corners with various sizes in one image. This kind of method is very fast and can be used for real-time applications, but the problem is that it is very hard for these corner detection algorithms to detect the exact four screen corners of a screen. Without knowing which four corners are screen corners, it is not possible to find out the screen position and figure out the pointing position of a camera. Another way to find the corners is by using classification models. After training a classification model by many corner samples, it can classify if a new sample is a corner or not. Classification models are popular in objects recognition problem and in our case, they can be seen as screen corners recognizer. If the model is ideal, then it can accurately detect the positions of screen corners after analyzing a whole image. But in reality, there are two problems with the classification models. The first one is the speed problem. It need to scan a whole image to detect the screen corners, which will spend a lot of time on feature extraction and classification so that it cannot reach the real-time speed requirement. The second problem is that the classification models are not as accurate as what we expected. Plenty of non-screen corners are recognized as screen corners, which does not allow us to get screen corners directly and have to use some filters to select out the final screen corners.

Considering the advantages and disadvantages of these two methods, finally, we designed a Candidates-Winners approach that can detect screen corners accurately and quickly. The first step of Candidates-Winners approach is to use Sojka corner detection algorithm to select the candidates screen corners, which is a very accurate and real-time detection algorithm. Then in second step, we use a new classification model, RVM that has probability outputs and utilizes much few sample vectors, classify these candidates. Finally, when RVM cannot select out the final four screen corners, we use a rectangle filter utilizing the geometric shape of the screen to search out the exact four screen corners. In this approach, the detections are very accurate in different situations such as fast moving and rotating, and the time on feature extraction and RVM classification got much reduced since only screen corner candidates are needed to be classified. To detect screen corners on consecutive frames capturing from the camera, we use a tracking filter by which we only do the detection work in some region-of-interests that are selected by our ROI tracking filter algorithm. This much increases the speed and ensures that our final system can work on a real-time level.

In order to control the applications more conveniently, the UI-Wand needs to recognize some gestures that can be set as corresponding commands. For this functionality, instead of using general HMM, we directly use motion vectors as feature vectors and RVM as classification model to recognize gestures. The test results indicate that the RVM is a very promising model for gesture recognition, which would get much higher recognition accuracy after future improvement.

The final working system and utilities of the system are implemented based on an existing framework for computer vision, which makes it easy to combine and test algorithms in the system. The UML design for the system shows a readable system framework so that the people working on this project in the future will be much easier to understand its principles and can modify it seamless.

Although the system has some problems when working on-line, but it already satisfies all the project goals listed in Chapter 2.

- It can detect screen corners from frames captured by UI-Wand.
- It is able to utilize RVM models to recognize some gestures.
- The screen corner detection is robust with some lighting changes.
- It can run different applications.
- It works in a range of working space.
- Its speed is faster than 10 frames/sec.

- Its results can be evaluated by system utilities.
- The development is based on Vispirin, an existing framework for computer vision.

12.2 Future Works

Though the final system satisfies the requirements, but the system still has many places that can be improved.

Parameter optimization: Because a lot of algorithms are used in the final system, there are many parameters to be setup. Now, we just setup these parameters manually, but in the future, they should be justified automatically. For example, what is the best combination of parameters in Sojka corners detector, by which it can detect as less as possible corners but without missing screen corners and as faster as it can be. Those parameters controlling the detector will be easily affected by lighting and noise changes. So if we want to let it adapt with more environments, it is better to design an algorithm that can collect some samples of current environment and find out the suitable parameters values.

ROI tracking filter has the similar problem. What is the best "offset" value? It should be decided by the moving speed of UI-Wand. In the future, a parameter optimization algorithm should be written to learn the speed on which the user moves his/her UI-Wand so that the algorithm can find an optimal value.

The parameters optimization in RVM also needs to be improved. Like what we described in Chapter 5, we need to rewrite our training algorithm, so that it can find out the best suitable kernel function and corresponding parameter values.

There are also parameters in other algorithms, like trace analysis algorithm and feature extraction algorithm, need to justify automatically. So parameters optimization is an important work in the future, with which the system will become more stable and flexible.

RVM dataset selection: From the on-line test, we saw that for some applications the RVM classification rates are not very high. One of the reasons is the current sample dataset we used is not too big. The current sample dataset is a synthetic dataset and only contains 10 corner samples for each class, which did not cover enough lighting or gray level changes and should be filled more in the future.

Another factor to affect the classification rates is the reasonability of samples selection. A good distribution of samples in one class will have their feature vectors be distributed averagely in the feature space. But the problem is how to evaluate the distribution. The distribution of samples in the synthetic dataset is reasonable but we still cannot explicitly evaluate the quality before using that for training. So it is better to invent a model that can evaluate the quality of the sample dataset and even reselect some samples in the dataset so that it can construct a more stable classification model.

More complex tracking filter: In the current system, we are using a simple motion estimation algorithm in ROI tracking filter. It is proved effective for some cases, but if the movement is very fast then it has some problems. One reason caused the problem is that the camera capturing rate is 10 frames/sec, which will result in some jerk when movement is fast. Another reason is that maybe our simple motion estimation is not accurate enough. So in the future we should try other prediction filter such as Kalman filter, by which we can get a better tracking result.

Color images: Now both of Sojka corners detector and RVM are based on gray level images, which will result in inaccurate detection when the border gray level is similar to the content's color. We have written a color version Sojka corner detector. Though the detection is getting more accurate but it is 3 times slower than gray level version. So in the future, we can do some research to increase its speed. For RVM model, we have not tried it on color images, since the speed is the problem as well. Maybe the classification rates will be much improved by using color training samples, but the same as Sojka corners detector, the feature extraction speed will become 3 times slower than doing on gray level. Moreover, the color is very unstable information when lighting conditions change, so we need more training samples to train the classification model, which will cause much more training time. But color information indeed help for recognition, so in the future we should do some research on it.

Gesture recognition: The current RVM model constructed for gesture recognition is not so accurate. The possible reason is that the qualities of gestures samples are not very good. So in the future, we should reselect the training samples and do more tests. Another try we need to do is to classify the trace immediately after UI-Wand moves, which is better than the current method, to find a candidates gesture between two pauses. If we can train RVM models and get a threshold for every gesture, then we can classify the trace in every frame, which reduce the delay time and response much rapidly.

The above possible improvements are based on the current system. Now we mention a new method to detect screen corners, which could be a good way to go in the future.

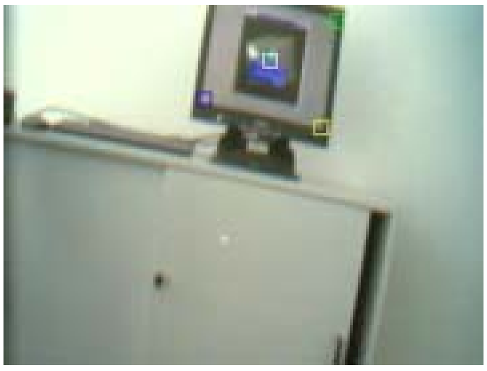
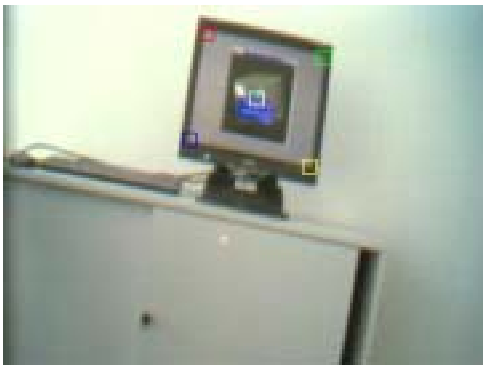
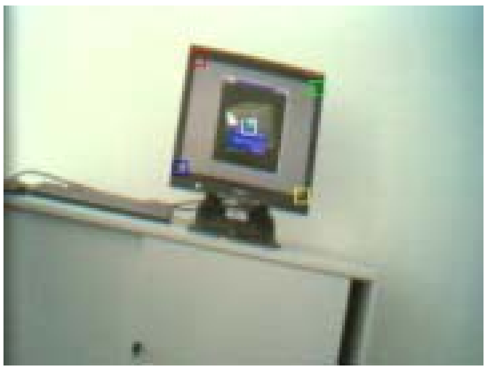
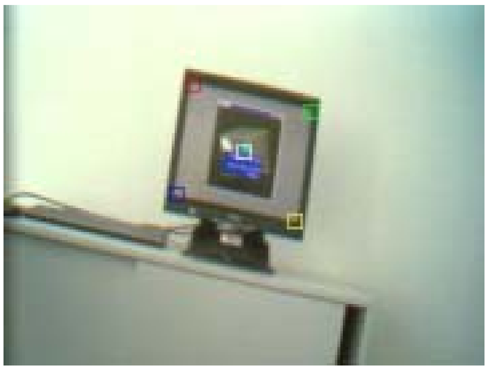
By the current Candidates-Winners approach, we can detect screen corners, but it has some problems when we move the UI-Wand very fast. Because the performance of Sojka corner detection will become worse if the movement is very fast and also the ROI tracking filter cannot track the trace. But there is one algorithm, which do not be affected too much. That is RVM. Because the fast movement will not change too much on the corners features, so if the corner samples are big enough, then they will keep enough features for classification so that the RVM models still can work. If the RVM classification rates are very high, then it is possible for us to only use RVM to detect screen corners. In Chapter 5, we tested many dataset and finally got an adaptive dataset which can get much higher classification rates and very low misunderstanding rates. With this dataset we can directly use RVM to classify a whole image and detect the screen corners. But as what we said in that chapter, the problem of this dataset is the speed. [Avi01][Wil03] mentioned Gaussian pyramids method to make quick match, which accelerates the scanning time very much. We have tried the similar method in our system. Indeed, the speed was significantly accelerated, but it was still not fast enough. So how to accelerate the algorithms is the key problem in the future works.

Appendix A
White Screen Sequence

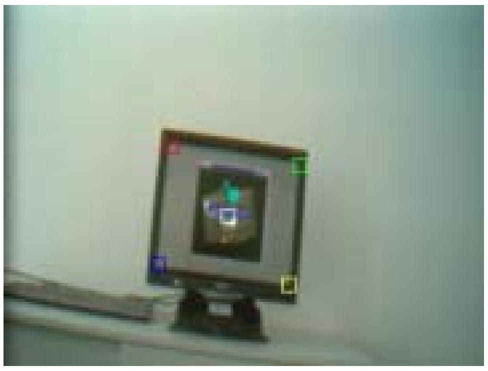
Appendix B
Black LCD Screen Sequence

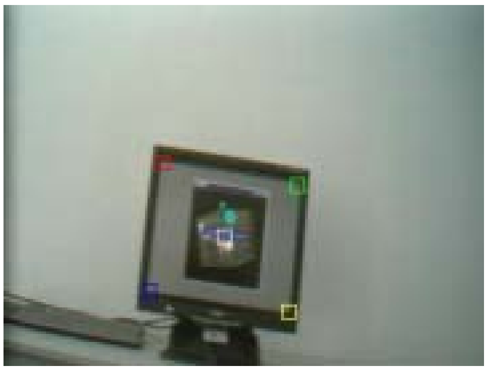
Appendix C
Ten Off-line Test Sequences Detection Results

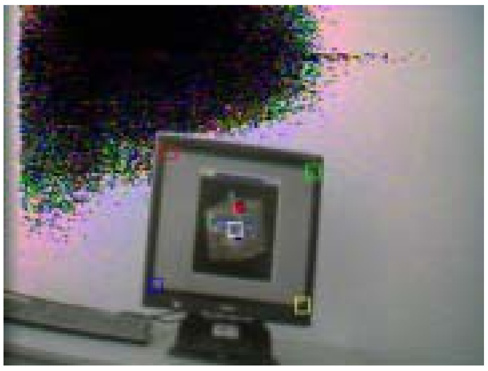
SS1: Pick and Pointing

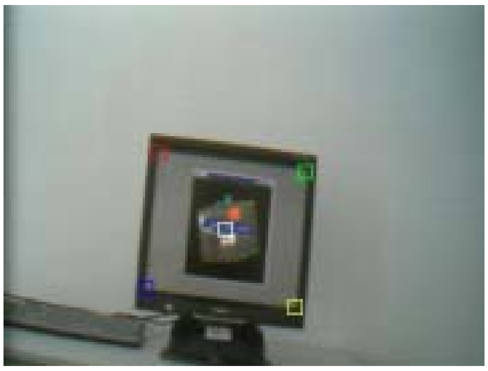
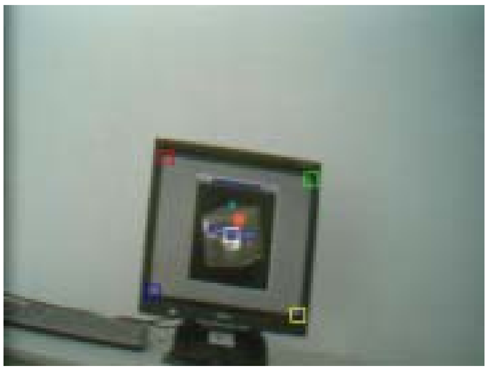
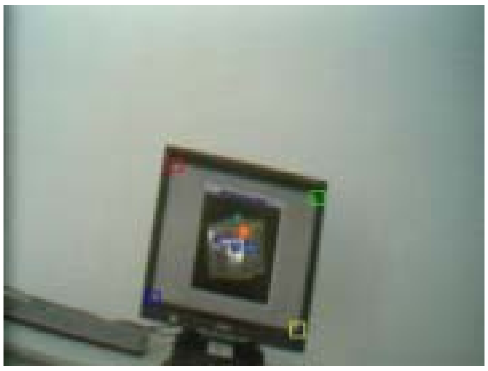
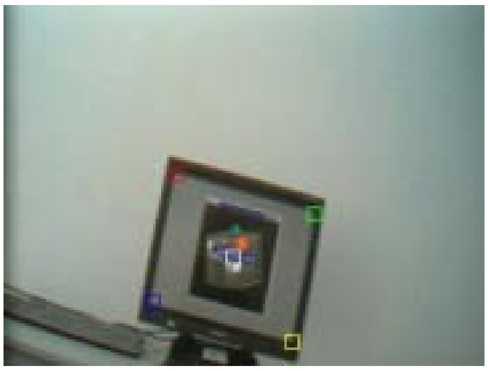
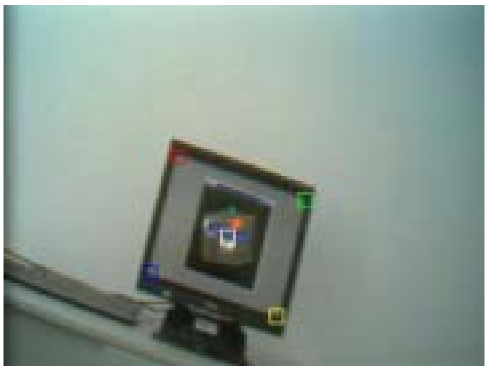
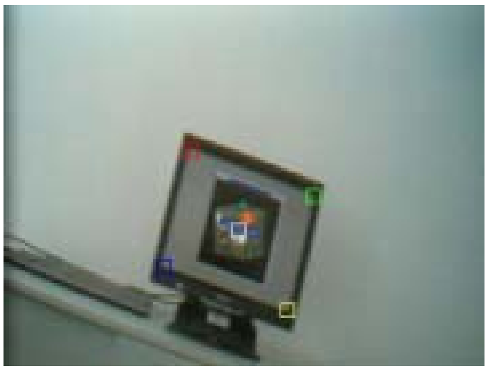
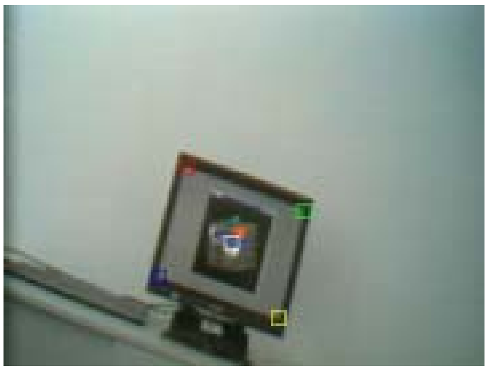
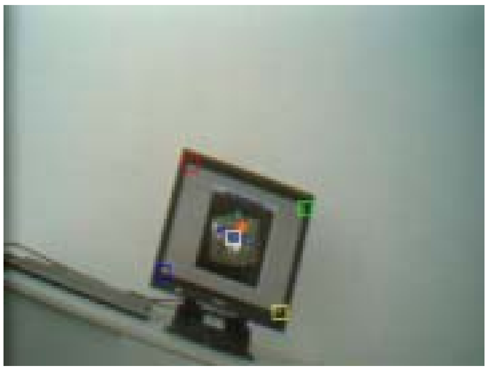
SS2: Forward



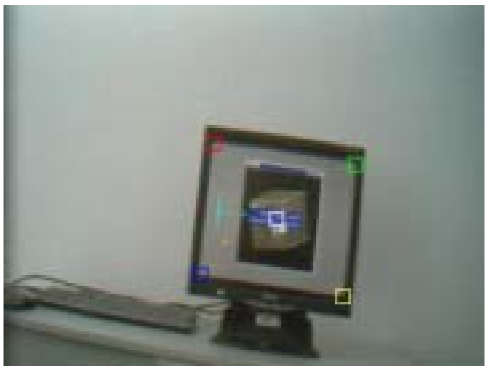




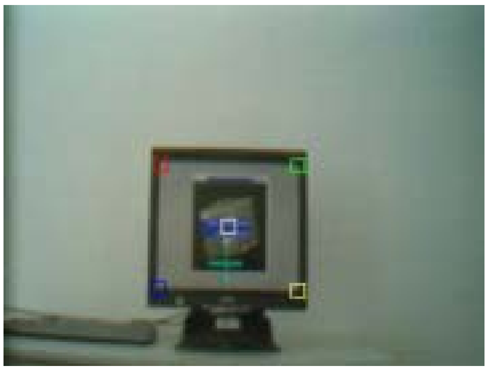
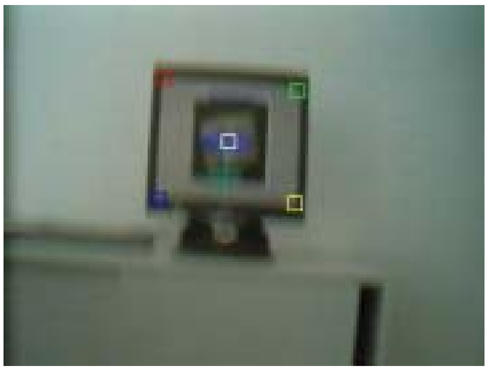
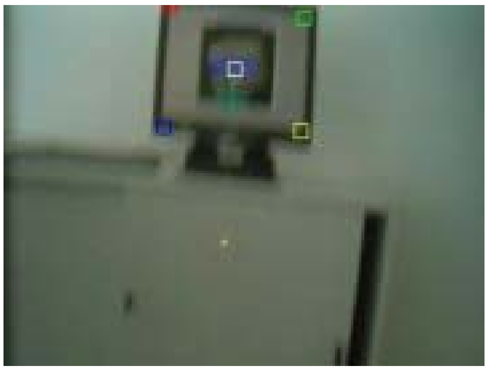
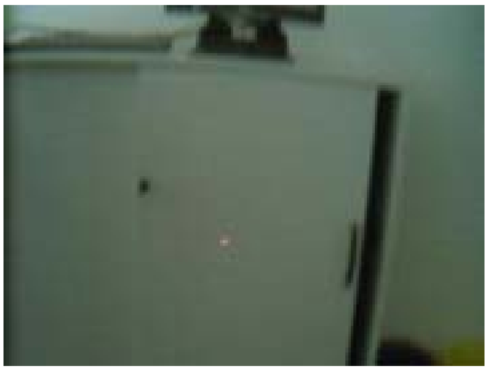
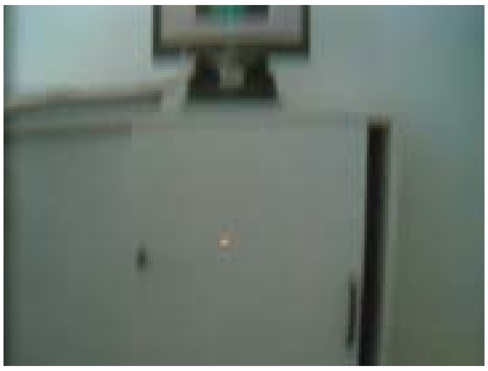
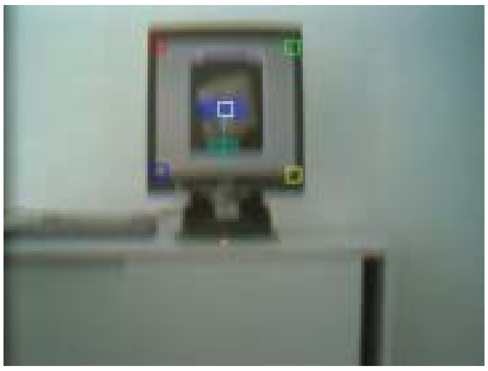
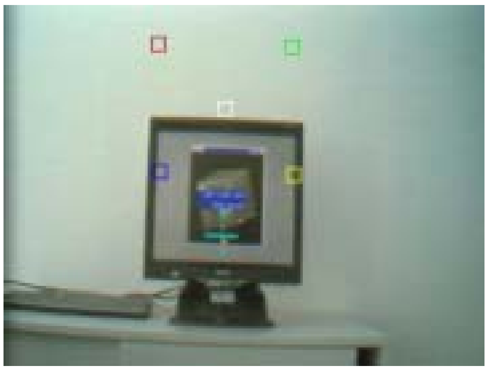
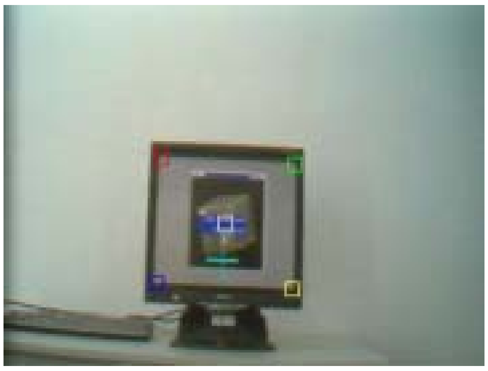
SS3: Backward

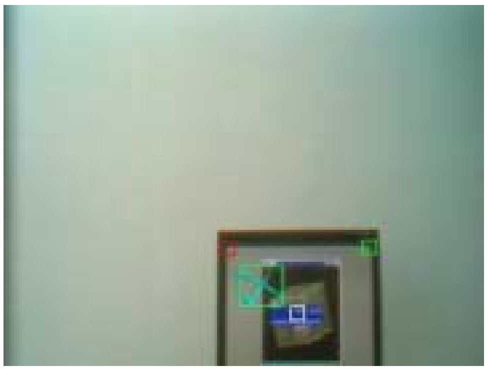
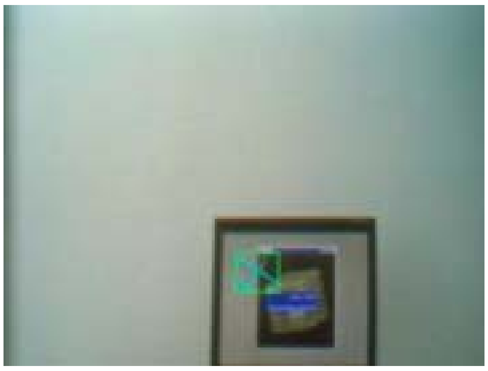
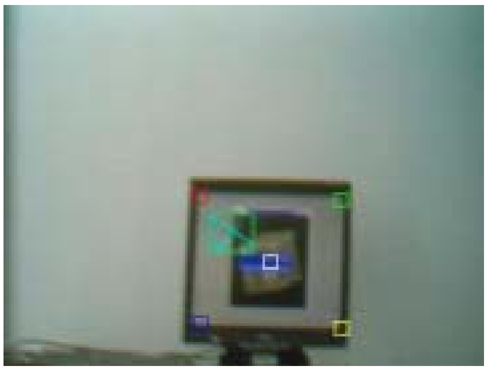
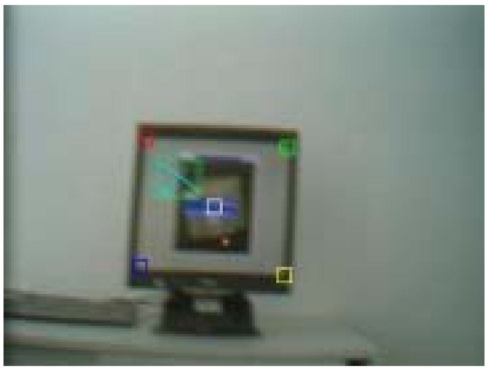
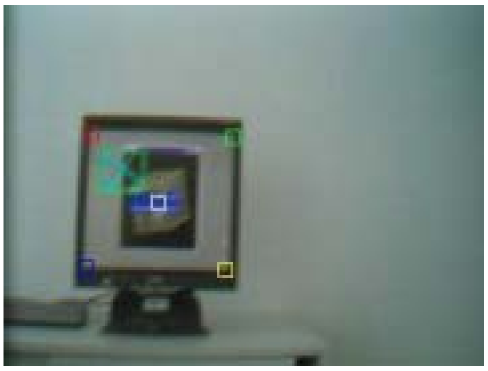
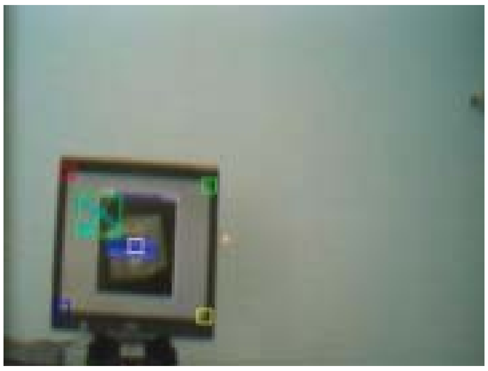
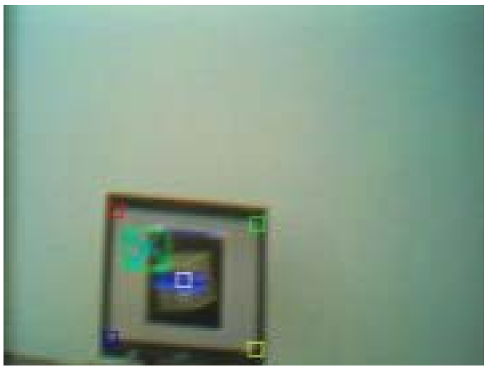
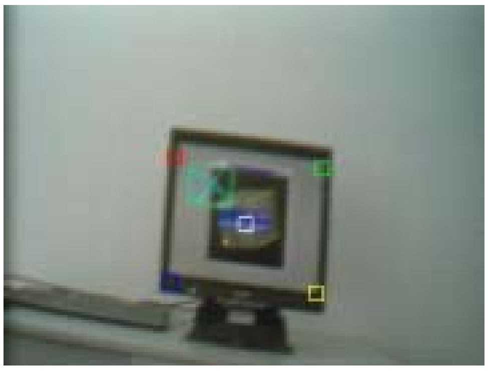
SS4: Left-Right



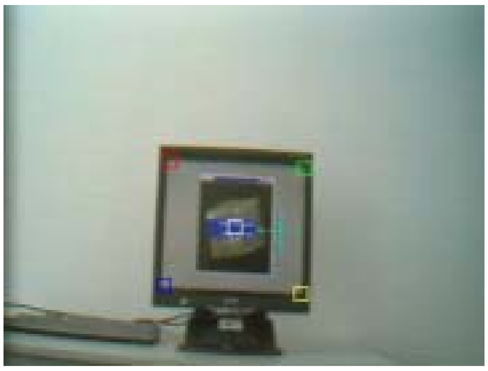
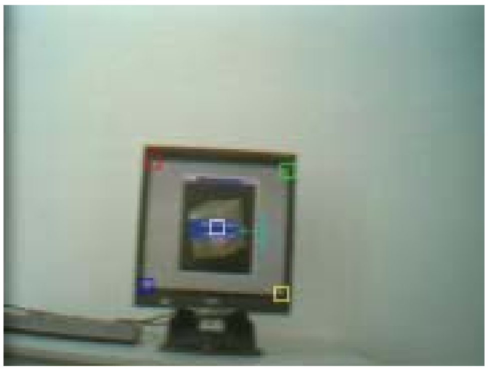
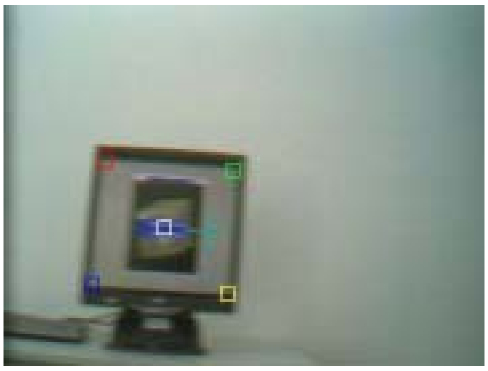
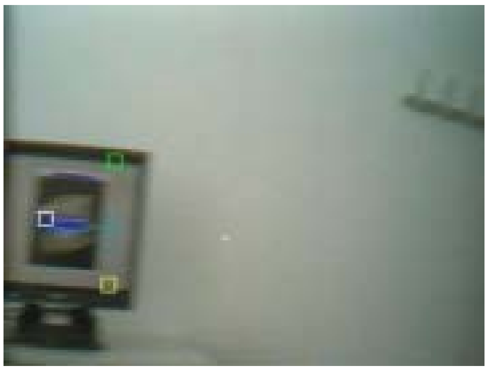
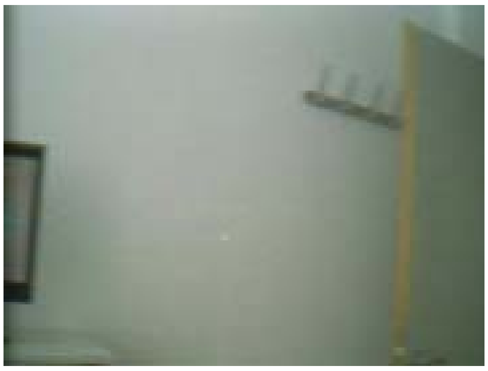
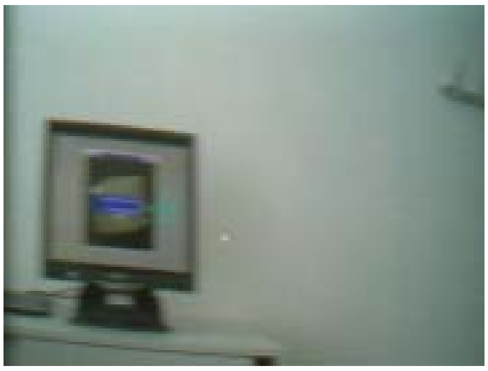
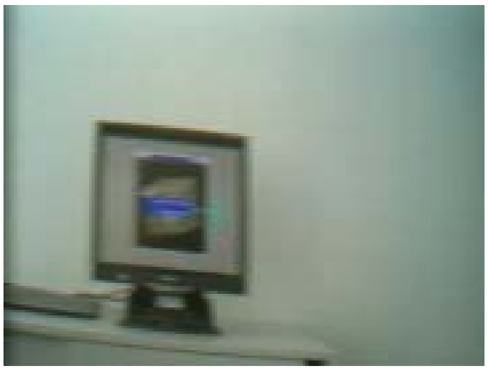
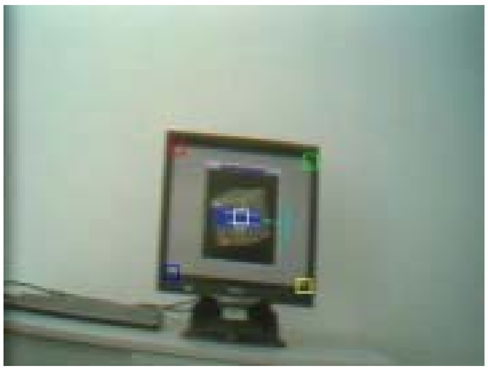
SS5: Down-Up

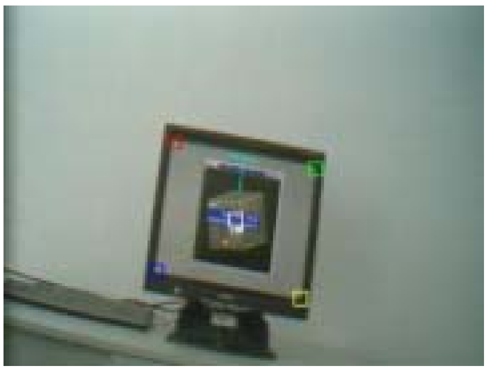
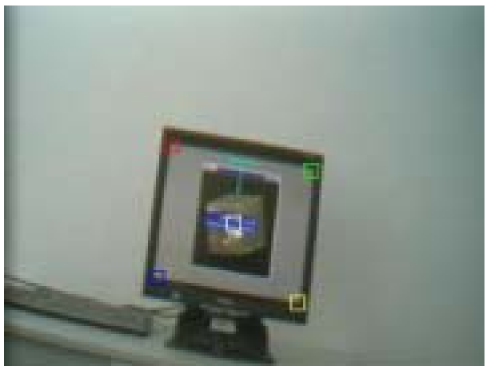
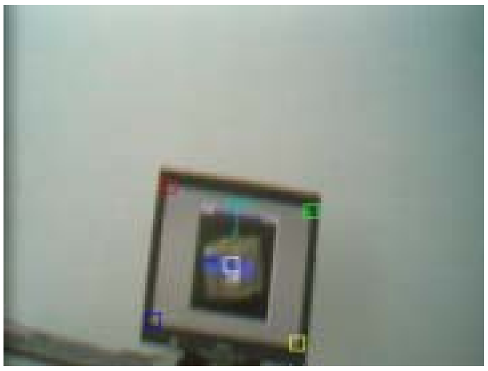
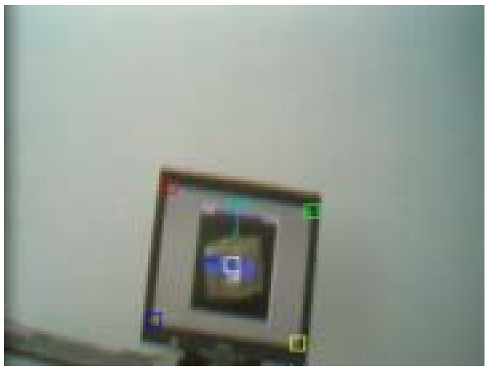
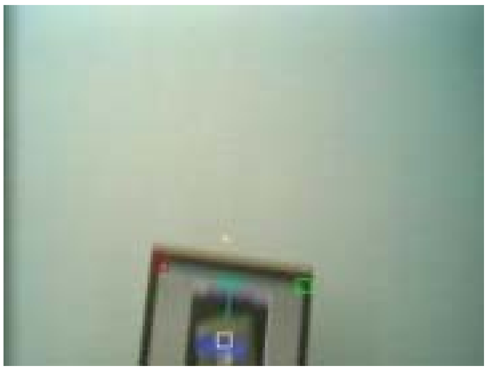
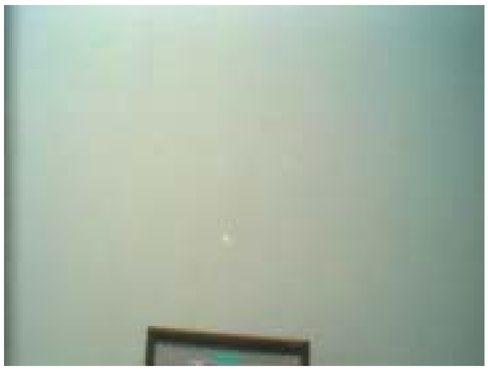
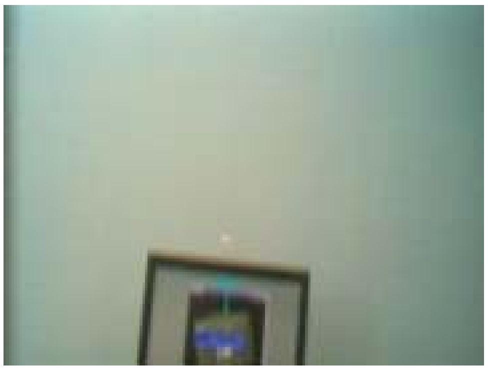
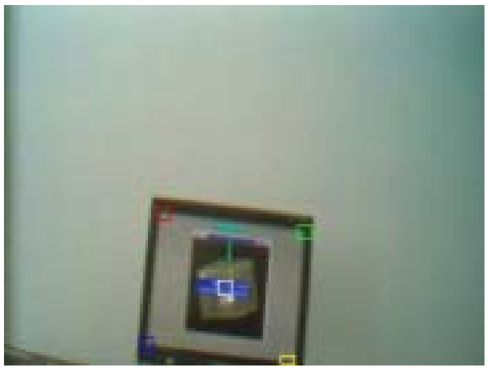
SS6: Cross

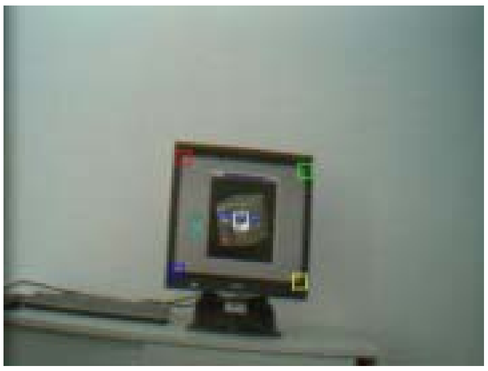
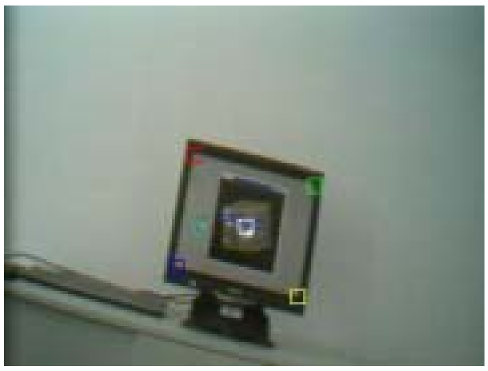
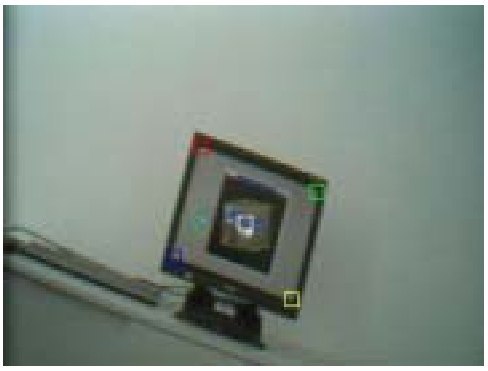
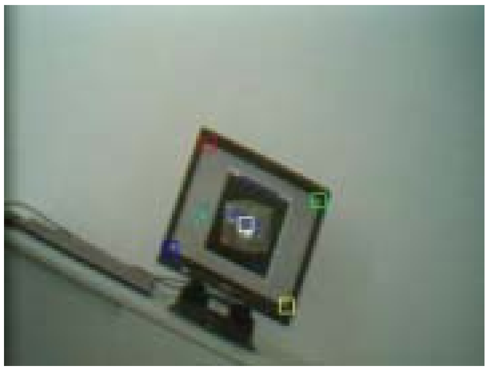
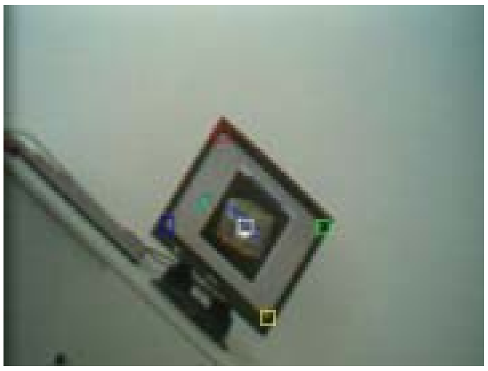
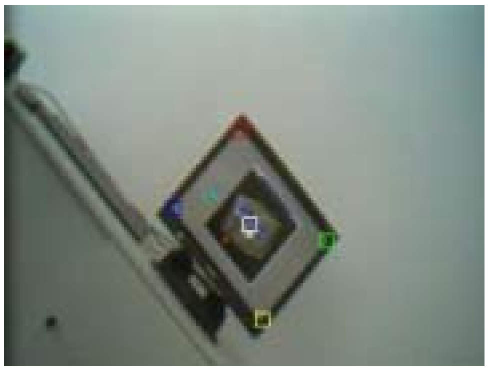
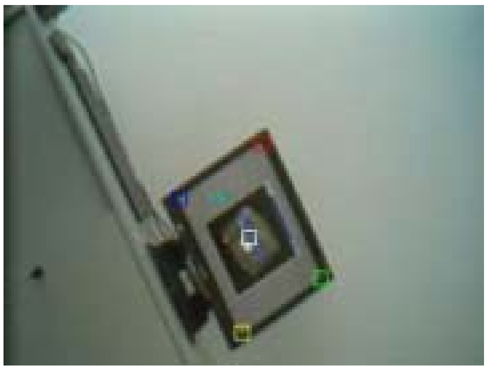
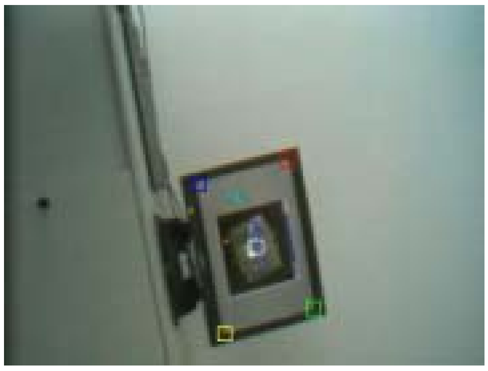
SS7: Right-left

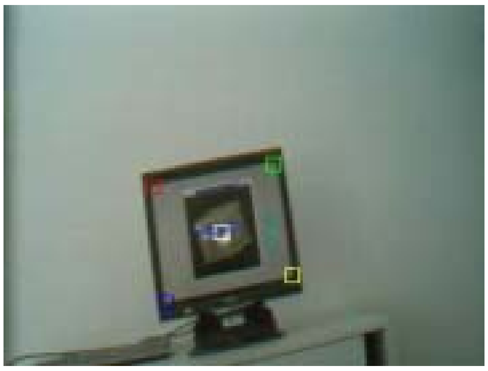
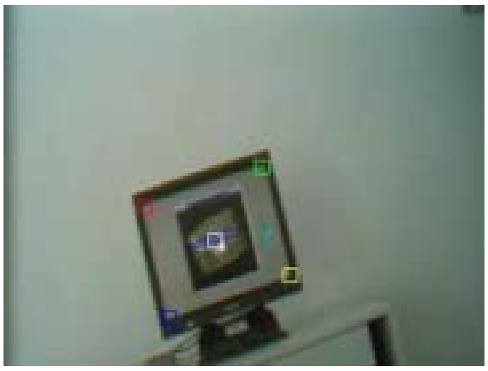
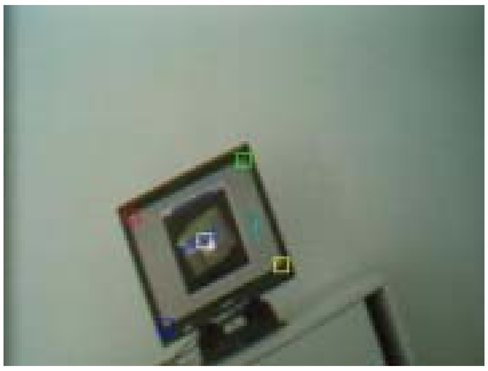
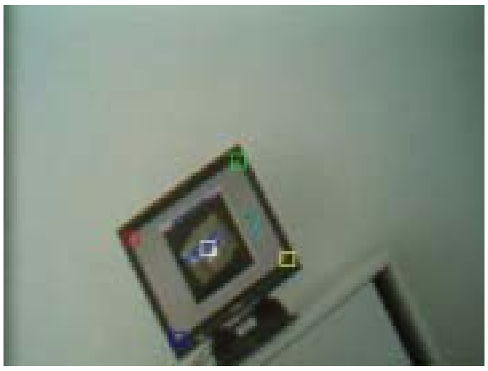
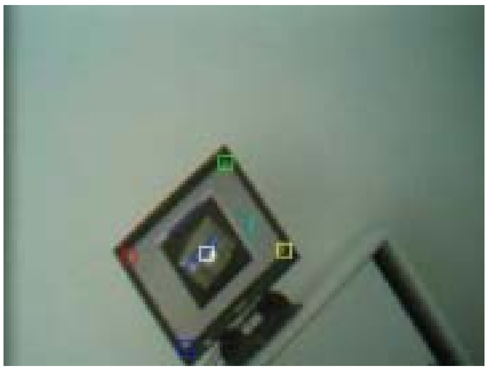
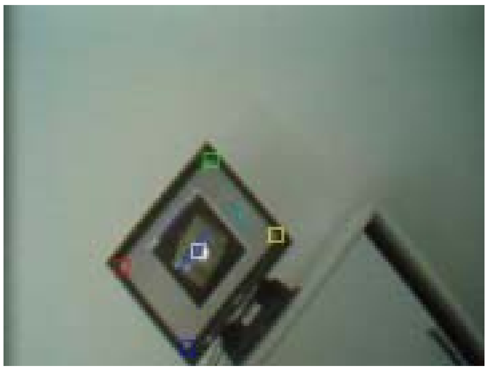
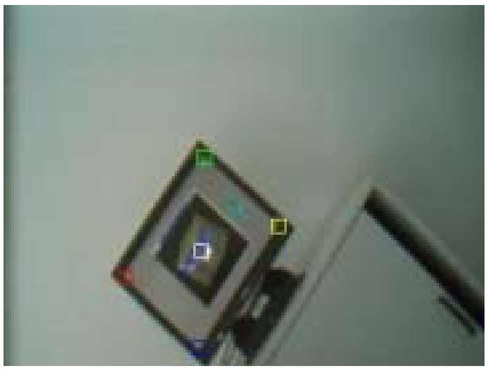
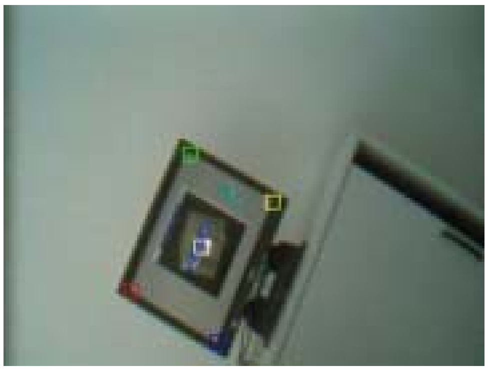
SS8: Up-down

SS9: Left-Rotation

SS10: Right rotation

Appendix D.1
Algorithm Parameters for Off-line Tests

```
# Parameters of UI-Wand application utility algorithm.
# It is called RunningCtrl.
applicationutil=RunningCtrl
applicationutil::run_level=1
applicationutil::log_level=1
applicationutil::log_directory=./cornerlist/

# Parameters of Corner Detection Algorithm.
# In implementation we called sojka detection
# algorithm as BayesCornerDetection.
cornersdetection=BayesCornerDetection
cornersdetection::halfPsgMaskSize=4
cornersdetection::angleThresh=0.5
cornersdetection::noiseGradSizeThresh=5
cornersdetection::apparenceThresh=5
cornersdetection::meanGradSizeThresh=0.0
cornersdetection::inertiaRadiusThresh=0.0
cornersdetection::sigmaD=0.75
cornersdetection::sigmaR=2.5
cornersdetection::halfExtMaskSize=2
cornersdetection::options=1
cornersdetection::annotationWindowedSize=20
cornersdetection::ROI_x_size=50
cornersdetection::ROI_y_size=50

# Parameters of feature extraction algorithm.
# The algorithm extract a sub-window features
# by given its center point.
# Its name is ROIWindowedDCT (AnnotationFeatures Class)
features=ROIWindowedDCT
features::window_size_x=10
features::window_size_y=10
features::extract_size_x=10
features::extract_size_y=10
features::extract_mode=0
features::normalize_mean=false
features::normalize_stddev=false
features::dct_min_modes=0
features::dct_max_modes=3
```

```
features::x_feature=true
features::y_feature=true
features::tilted_angle=-10

# Parameters for the corner classification algorithm.
# In this case we are using RVM.
classification=RVMClassifier
classification::train_model=false
classification::class_num=5
classification::feature_dim=10
classification::discard_class=4
classification::kernel=gauss
classification::kernel_length=0.5
classification::samples_vectors_filename=./featurevectors_autoblack.mod
classification::model_weights_filename=./rvmweights_autoblack.mod

# Parameters of rectangle filter algorithm.
# It can select out the 4 screen corners
# by geometric properties.
screenfilter=ScreenCornersFilter
screenfilter::distance_x=95
screenfilter::distance_y=75
screenfilter::offset=20
screenfilter::tilted_angle=-10
screenfilter::predication_num_frame=3
screenfilter::history_size=30

# Parameters of trace analysis algorithm.
# It will do some analysis on a certain size trace,
# like trace feature extraction and storing.
traceanalysis=InterpolationMV
traceanalysis::model=analysis
 # parameters for collecting gesture samples
 traceanalysis::samples_dir=./training_samples/gestures/
 traceanalysis::feature_vectors_filename=./gfeaturevectors.mod
 traceanalysis::current_gesture=cross
 traceanalysis::gesture0_label=updown
 traceanalysis::gesture1_label=downup
 traceanalysis::gesture2_label=leftright
 traceanalysis::gesture3_label=rightleft
 traceanalysis::gesture4_label=cross
 traceanalysis::nongesture_label=others
 # parameters for generating random gestures
 traceanalysis::non_gesture_step_min=5
 traceanalysis::non_gesture_step_max=30
 traceanalysis::non_gesture_step_distribution=3
 traceanalysis::non_gesture_angle_min=30
 traceanalysis::non_gesture_angle_max=180
 traceanalysis::non_gesture_angle_distribution=4
 traceanalysis::non_gesture_num_perdistribution=3
 # parameters for gestures analysis
 traceanalysis::trace_size=50
 traceanalysis::num_motion_vector=10
 traceanalysis::num_gesture_samples=10
 traceanalysis::num_nongesture_samples=40
 traceanalysis::base_non_gesture=false
 traceanalysis::movement_sensitivity=5
 traceanalysis::holding_num_unittime=6
 traceanalysis::missing_tolerance=6

# Parameters for output algorithm
output=ppm
output::binary_format=true
output::filename_base=./resultdata/result

# Parameters for drawing annotation on images
annotation=Draw
annotation::contour_color_r=255
annotation::contour_color_g=255
annotation::contour_color_b=255
```

Appendix D.2 Algorithm Parameters for On-line Tests

```
# The parameters below are part of parameters used in the UI-Wand prototype
system. We did not list other parameters used for pointing positioning
algorithms, which are independent part from screen corner detection.

# Parameters of Corner Detection Algorithm.
# In implementation we called sojka detection
# algorithm as BayesCornerDetection.
detection=BayesCornerDetection
detection::halfPsgMaskSize=4
detection::angleThresh=0.5
detection::noiseGradSizeThresh=18
detection::apparenceThresh=4
detection::meanGradSizeThresh=0.0
detection::inertiaRadiusThresh=0.0
detection::sigmaD=0.75
detection::sigmaR=2.5
detection::halfExtMaskSize=2
detection::options=1
detection::annotationWindowedSize=20
detection::ROI_x_size=50
detection::ROI_y_size=50

# Parameters of feature extraction algorithm.
# The algorithm extract a sub-window features
# by given its center point.
# Its name is ROIWindowedDCT (AnnotationFeatures Class)
features=ROIWindowedDCT
features::window_size_x=10
features::window_size_y=10
features::extract_size_x=10
features::extract_size_y=10
features::extract_mode=0
features::normalize_mean=false
features::normalize_stddev=false
features::dct_min_modes=0
features::dct_max_modes=3
features::x_feature=true
features::y_feature=true
features::tilted_angle=0
```

```
# Parameters for the corner classification algorithm.
# In this case we are using RVM.
classification=RVMClassifier
classification::train_model=false
classification::class_num=5
classification::feature_dim=10
classification::kernel=gauss
classification::kernel_length=0.5
classification::samples_vectors_filename=./featurevectors_autoblack.mod
classification::model_weights_filename=./rvmweights_autoblack.mod

# Parameters of rectangle filter algorithm.
# It can select out the 4 screen corners
# by geometric properties.
screenfilter=ScreenCornersFilter
screenfilter::distance_x=125
screenfilter::distance_y=125
screenfilter::offset=20
screenfilter::tilted_angle=0.0
screenfilter::predication_num_frame=3
screenfilter::history_size=10

# Parameters file of pointing positioning algorithm.
# The name of this algorithm is EdgeCornerMarkerRegistration
registration.par
```

Bibliography

[Ale98] A.Alexandrov. Corner detection overview and comparison, *Computer Vision*, vol. 588, 2002

[Ard00] E.Ardizzone, A.Chella, R.Pirrone. Feature-based shape recognition by Support Vector Machines, *Proc. of ECAI 2000 Workshop on Machine Learning in Computer Vision,* Berlin, Germany, Aug 2000.

[Avi01] S.Avidan. Support Vector Tracking. Proc. Conf. Computer Vision and Pattern Recognition, Hawaii, 2001.

[Bea78] P.R.Beaudet. Rotationally invariant image operators, *Proc. Fourth Int. Joint Conf. on Pattern Recognition*, Tokyo, pp. 579-583, 1978.

[Bla98] M.J.Black and A.D.Jepson. Recognizing temporal trajectories using the condensation algorithm Automatic Face and Gesture Recognition, *Proc. Third IEEE International Conf.*, pp. 16–21, Apr 1998.

[Bro86] A.J.Broder. Strategies for Efficient Incremental Nearest Neighbor Search, *Pattern Recognition*, vol. 23, pp. 171-178, Nov 1986.

[Can03] H.Cantzler and C.Hoile, A novel form of a pointing device, *Vision, Video, and Graphics*, pp.1-6, 2003.

[Cha00] V.Chauhan, T.Morris. Face and feature tracking for cursor control, *Scandinavian Conference on Image Analysis(O-Th2)*, Feb, 2000.

[Cor95] C.Cortes and V.N.Vapnik. Support Vector Network, *Machine Learning,* vol. 20, pp. 273-297, September 1995.

[Der93] R.Deriche and G.Giraudon. A computational approach for corner and vertex detection, *International Journal of Computer Vision*, vol. 10, issue. 2, pp. 101-124, 1993.

[Har88] C.G.Harris and M.Stephens. A combined corner and edge detector, *Proc 4th Alvey Vision Conf.*, pp.189-192, Manchester, 1988.

[Hei03] B.Heisele. Visual Object Recognition With Supervised Learning. *IEEE Tran. Intelligent Systems,* vol.18, no.3, May-June 2003.

[Hon02] P.Hong, M.Turk and T.S.Huang. Gesture Modeling and Recognition Using Finite State Machine, *IEEE Conf. on Face and Gesture Recognition*, 2002.

[Jai00] A.K.Jain, R.P.W.Duin and J.Mao. Statistical pattern recognition: A review, *Pattern Analysis and Machine Intelligence*, vol.22, no.1, 2000.

[Kal60] R.E.Kalman. A new approach to linear filtering and prediction problems, *Tran.The ASME Journal of Basic Engineering,* vol. 82, series. D, pp.35-45, 1960.

[Kit82] L.Kitchen and A.Rosenfeld. Gray-level corner detection, *Pattern Recognition Letters*, vol. 1, pp. 95-102, 1982.

[Lee99] H.K.Lee and J.H.Kim. An HMM_based threshold model approach for gesture recognition, *IEEE Tran. Pattern Analysis and Machine intelligience*, vol. 21, no.10, 1999.

[Liu03] N.Liu, B.C.Lovell. Gesture classification using Hidden Markov Models and Viterbi Path Counting, *VIIth Digital Image Computing: Techniques and Applications*, 2003.

[Lus95] E.F.Lussier. Markov Models and Hidden Markov Models: A brief tutorial, *Introduction to Artificial Intelligence at the University of California, Berkeley*, 1995 semester of CS188.

[Mac92] D.J.C.MacKay. The evidence framework applied to classification networks*. Neural Computation*, vol. 4, no. 5, pp. 720-736, 1992.

[Mac94] D.J.C.MacKay. Bayesian methods for backpropagation networks. *In E. Domany, J.L. van Hemmen, and K. Schulten, editors, Models of Neural Networks III*, chapter 6, pp. 211-254. Springer, 1994.

[Nab99] I.T.Nabney. Efficient training of RBF networks for classification. In *Proc. Of the Ninth International Conference on Artificial Neural Networks*, pp. 210-215, IEE, 1999.

[Nob88] J.A.Noble, Finding corners, *Image and Vision Computing*, vol. 6, no. 2, pp.121-128, 1988*.*

[Par98] L.Parida, D.Geiger and R.Hummel, Junctions: detection, classification and reconstruction, *IEEE Trans, Pattern Analysis and Machine Intelligence*, vol. 20, no.7, pp.687-698, 1998.

[Pon98] M.Pontil. Support Vector Machines for 3D object Recognition, *IEEE Tran. Pattern analysis and Machine intelligence*, vol. 20, no. 6, pp. 637-646, Jun 1998.

[Psa02] A.Psarrou, S.G.Gong, M.Walter. Recognition of human gestures and Bayesian based on motion trajectories, *Image and Vision Computing*, vol.20, Issues.5-6, pp.349-358, 2002

[Rit75] G.L.Ritter, H.B.Woodruff, S.R.Lowry, and T.L.Isenhour. An Algorithm for a Selective Nearest Neighbor Decision Rule. *IEEE Trans. Information Theory*, vol. 21, pp. 665-669, Nov 1975.

[Ruz01] M.A.Ruzon, and C.Tomasi. Edge, junction, and corner detection using color distributions, *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 23, no. 11, pp. 1281-1295, 2001.

[Ruz99a] M.A.Ruzon and C.Tomasi. Color Edge Detection with the Compass Operator, *IEEE Conf. On Computer Vision and Pattern Recognition*, vol. 2, pp.160-166, 1999.

[Ruz99b] M.A.Ruzon and C.Tomasi. Corner Detection in Textured Color Images, *IEEE Seventh International Conf. On Computer Vision*, vol. 2, pp.1039-1045, 1999.

[Smi97] S.M.Smith, J.M.Brady. SUSAN-A new approach to low level image processing, *International Journal of Computer Vision*, vol. 23, pp. 45-78, 1997.

[Soj02a] E.Sojka. A new algorithm for detecting corners in digital images, *Proc. SCCG 2002*, ACM SIGGRAPH, NY, pp. 55-62, 2002.

[Soj02b] E.Sojka. A new and efficient algorithm for detecting the corners in digital images, *Proc. 24th DAGM Symposium*, *Springer*, *LNCS 2449*, *Berlin*, *NY*, pp. 125-132, 2002.

[Soj03] E.Sojka. A new approach to detecting the corners in digital images, *Accepted for publication in IEEE ICIP*, 2003.

[Son03] J.Song, M.R.Lyu, and M.Cai. A generic color-distribution-based approach to detect edges, corners and junctions simultaneously in color images, submitted to *IEEE Trans. Image Processing*, 2003.

[Tip01] M.E.Tipping. Sparse ayesian learning and the relevance vector machine. *Journal of Machine Learning Research*, vol. 1, pp. 211-244, 2001.

[TipNEW] M.E.Tipping. Fast Marginal Likelihood Maximization for Sparse Bayesian Models.

[The03] S.Theodoridis and K.Koutroumbas. *Pattern Recognition (second edition)*, 2003.

[Tra98] M.Trajkovč and M.Hedley. Fast corner detection, *Image and Vision Computing*, vol. 16, pp. 75-87, 1998.

[Vap95] V.N. Vapnik. *The Nature of Statistical Learning Theory*. New York, NY: Springer-Verlag, 1995.

[Wan95] H.Wang and M.Michaelis. Real-time corner detection algorithm for motion estimation, *Image and Vision Computing*, vol. 13, no. 9, pp. 695-703, 1995.

[Web02] A.R.Webb. Statistical Pattern Recognition, second edition, John WILEY & SONS, Ltd, 2002.

[Wel01] G.Welch and G.Bishop. An introduction to the Kalman filter. *University of North Carolina at Chapel Hill*, *Department of Computer Science*, Course 8 SIGGRAPH, 2001.

[Wil03a] O.Williams, A.Blake and R.Cipolla. A Sparse Probabilistic Learning Algorithm for Real-Time Tracking. *Int. Conf. on Computer Vision*, *ICCV*, Nice, France, 2003.

[Wil03b] A.D. Wilson and S. Shafer. XWand: UI for intelligent spaces, *CHI*, 2003.

[Wil03c] A.D.Wilson and H.Pham. Pointing in intelligent environments with the WorldCursor, *Interact*, 2003.

[Yan99] M.H.Yang and N.Ahuja. Recognizing hand gesture using motion trajectories, *IEEE CS Conf. on Computer Vision and Pattern Recognition*, vol.1, pp. 466-472, Jun 1999.

[Zha04] B.Zhang, S.N.Srihari. Fast *k*-Nearest Neighbor Classification Using Cluster-Based Trees, *IEEE Tran. Pattern Analysis and Machine Intelligence*, vol. 26, no. 4, Apr 2004.

[Zhe99] Z.Zheng, H.Wang and E.K.Teoh. Analysis of gray level corner detection, *Pattern Recognition Letters*, vol. 20, pp.149-162, 1999.