About Portable Keyboards with Design and Implementation of a Prototype Using Image Processing

THOMAS WÖLLERT (DIPL.-INF. (FH))

Matriculationno. 05478901-0199
Semestergroup IG2

Semester Thesis SS2006
Human-Machine-Interaction

Master of Science
Computer Graphics and Image Processing

Department of Computer Science and Mathematics
Munich University of Applied Sciences

fachhochschule münchen
munich university of applied sciences

„Your program does not do
the things you thought it would
but the things you programmed ...“

anonymous

Abstract

Type:	Semester Thesis
Author:	Wöllert, Thomas (Dipl.-Inf. (FH))
Titel:	About Portable Keyboards with Design and Implementation of a Prototype Using Image Processing
Date:	1^{st} June 2006
Number of Pages:	40
Field of Study:	Master of Science - Computer Graphics and Image Processing
University:	Munich University of Applied Sciences, Germany
Advisor:	Prof. Dr. G. Socher

With the hardware getting smaller and smaller it sometimes seems, that todays keyboards are a bit left behind by the development. PDAs and cellphones are close to being a complete personal computer, but controlling such devices often relies on over-sized keyboards. Over the last years some advances have been made in portable keyboard design more, or less giving up the original keyboard-look to earn other advantages.

The first part of this thesis provides an overview of the current state of development regarding portable keyboards. Various examples like rollable keyboards, touchscreens, interactive gloves and miniature keyboards are presented, all with their specific advantages and drawbacks. A specific example, the Celluon projection keyboard by Canesta Inc., is explained in more detail.

After the introduction, the second part focuses on the prototype design and implementation of a virtual keyboard using image processing. The technical basics are presented leading to requirements both for software, hardware and the implementation. Testing this prototype as well as discussing the results completes this part of the document.

Finally the last part gives a summary of the gathered results as well as an outlook on future developments.

Keywords: projection keyboard, virtual keyboard, Java, JMF, JDMS, DirectShow, webcam, gui, camera, image processing, blob coloring algorithm

Contents

List of Figures

List of Tables

List of Sources

Chapter 1

Introduction

In the context of the course Human-Machine-Interaction, one focus was set upon portable input devices, especially keyboards. With the hardware getting smaller and smaller it sometimes seems that todays keyboards are a bit left behind by the development. PDAs and cellphones are close to being a complete personal computer, but controlling such devices often relies on over-sized keyboards. Over the last years some advances have been made in portable keyboard design, more or less giving up the original keyboard-look to earn other advantages.

1.1 Task

The first objective was to get an overview over the latest developments in portable keyboards. One of these developments, the Celluon [3] projection keyboard by Canesta Inc. [2], should be examined in more detail.

Due to the focus lying on the projection keyboard, the decision was made to implement a prototype virtual keyboard as practical part of this thesis. Webcams were chosen to record the users movement gathering the necessary information via digital image processing.

1.2 Motivation

Looking at the common computer desktop, people are usually astonished by the fast advances, which are made not only in comfort and usability but also in getting the devices even smaller than the day before. Looking closer, it is not surprising, that someone must also wonder, why the most up-to-date keyboard is almost looking like the first one, developed decades ago (see Fig. 1.1).

Complete keyboards are still needed for comfortable control of a computer. On some notebooks it is already a drawback, that the keyboard is smaller than normal, resulting in user

Figure 1.1: Keyboard, Push-Buttons, and Mouse, 1960s at Stanford Research Institute [4]

errors and discomfort. So the question is, how it is possible to make the keyboard small enough to keep holding up with todays small hardware.

1.3 Overview

The first part of this thesis provides an overview of the current state of development regarding portable keyboards. Various examples like rollable keyboards, touchscreens, interactive gloves and miniature keyboards are presented, all with their specific advantages and drawbacks. A specific example, the Celluon projection keyboard by Canesta Inc., is explained in more detail.

After the introduction, the second part focuses on the prototype design and implementation of a virtual keyboard using image processing. The technical basics are presented leading to requirements both for software, hardware and the implementation. Testing this prototype as well as discussing the results completes this part of the document.

At the end, the last part gives a summary of the gathered results as well as an outlook to future developments.

Chapter 2

Latest Developments in Portable Keyboards

The normal user at home is often not aware of new technical developments, so giving an overview of the current technology in portable keyboards had to be made. Due to time limitations this summary is done by giving examples of industrial products available today as well as the mentioned Celluon projection keyboard.

2.1 Industrial Examples

Todays industrial products can be divided into two categories: products ,that keep the basic look of keyboards and those, giving up this nature for better portability and usability in the desired environment. The following list is far from being complete, but gives a good impression of what is possible and available on the market today.

2.1.1 Foldable Keyboards

A direct approach to address the portability problem was to keep the layout of a normal keyboard and simply make it foldable. This includes automatically all the keyboard's usability advantages and offers the customers a piece of hardware they know from their keyboard at home.

As an example Eleksen Ltd. [5] is currently offering a textile keyboard, which connects to a desktop computer, PDA or cellphone via bluetooth. Including a small battery pack, this enables the user to work normally for up to ten hours [6] (see Fig. 2.1).

Like already mentioned the layout is very similar to normal desktop keyboards. This enables the user to work as he or she is accustomed to, without the need to learn using a new

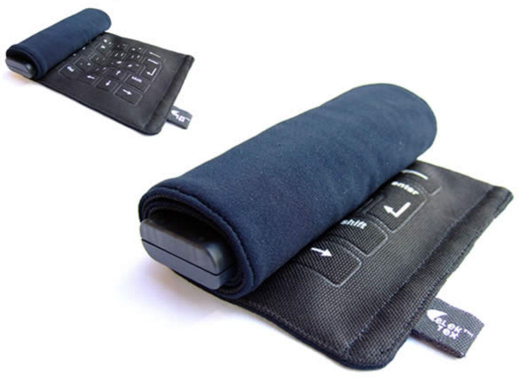


Figure 2.1: Bluetooth textile keyboard (Courtesy of Eleksen)

input device from scratch. This advantage is also the devices' drawback, because due to the dimensions of such keyboards, they still take some space in the pocket, even when folded. Additionally people still need some amount of space when using the keyboard, which might be a problem during portable situations (i.e. in the train, bus or plane).

2.1.2 Miniature Keyboards

The next kind of keyboard gives up some of the advantages of normal dimensions and layout for better portability. The following example is currently produced by Hama GmbH & Co. KG [7]. The *Bluetooth Freedom Mini Keyboard* [8] (see Fig. 2.2) is considerably smaller than normal keyboards, but still offers thirtynine keys and connects to its host device via bluetooth.

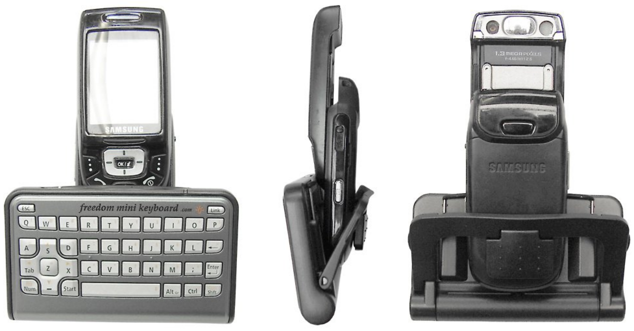


Figure 2.2: *Bluetooth Freedom Mini Keyboard* (with cellphone) (Courtesy of Hama)

As visible in Fig. 2.2 doing extensive work with such a small keyboard is not really possible. It is surely valuable for short periods of time and small amounts of text but the usability is suffering a lot from the small keys. Additonally the user has to get used to the new layout, which might result in some difficulties.

2.1.3 Touchscreens

By totally giving up the look of keyboards, touchscreens gain a lot of advantages. Their usage is mouse-like more intuitive by simply pressing onto the screen. Also due to their digital nature touchscreens can adapt their layout to the actual program situation i.e. by hiding some buttons which are not necessary. Due to that they are often used in embedded environments (i.e. car navigation systems, see Fig. 2.3).

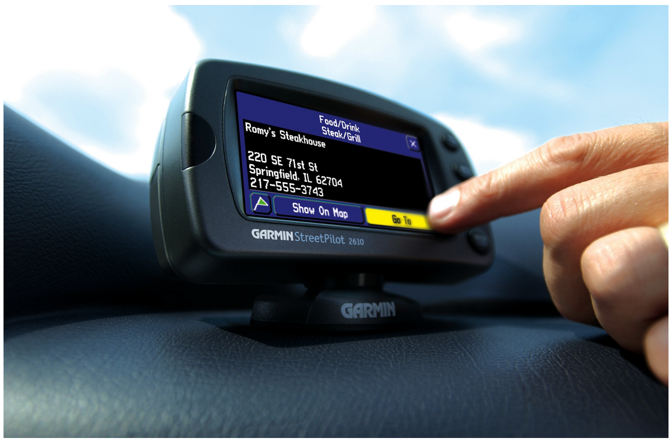
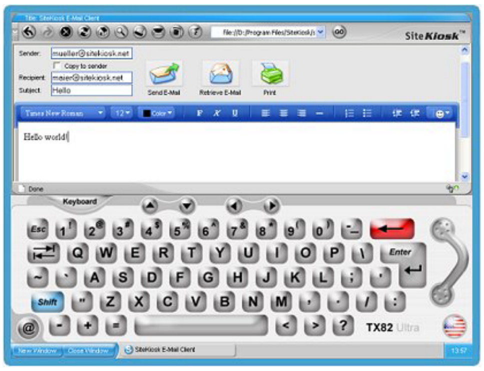


Figure 2.3: Touchscreen Garmin *StreetPilot 2660* [9]

In some cities using the internet is possible via public terminals i.e. at subway or train stations. Due to their public nature these touchscreens are normally protected with special glass making it impossible to place a normal keyboard there. In addition hygienic reasons also render a normal keyboard impractical. As a replacement the computer is simulating the whole keyboard in the lower part of the screen (see Fig. 2.4), where the user can access it via touch screen.



Sender: mueller@sitekiosk.net
Copy to sender
Recipient: maier@sitekiosk.net
Subject: Hello

Hello world!

Figure 2.4: Touchscreen *Kiosk Mode - Internet Browser Software* [10]

As seen in Fig. 2.4 offering a normal-sized keyboard layout on a touchscreen takes up a significant amount of space. The dimension of the touchscreen itself (i.e. PDAs) also restricts the size of the keyboard, which renders such setups impossible for small portable devices. Some companies have tried to remedy this drawback by exchanging the keys with software algorithms transforming written into digital letters. Still the usability in writing longer documents is as limited as if using miniature keyboards (see Sec. 2.1.2).

2.1.4 Glove Keyboard

Some advances have been made in totally replacing the keyboard with a new input device. Focused in the portability and the problem, that the user might not have a plain area to lie out a keyboard, the developers based their design on gloves. These are currently developed by KITTY Technologies see [11]. Technically the gloves are based on the same idea as keyboards: By connecting two wires a signal is generated, representing a specific key. On the keyboard this is done by actually pressing the key. The glove is accomplishing the same goal by placing the wires on the outside of the fingers. The user now has to connect two fingers at pre-defined positions to produce a certain key stroke (see Fig. 2.5). In the shown example the user produces a 'g' by pressing the forefinger onto the mid sensor of his left-hand thumb [1].

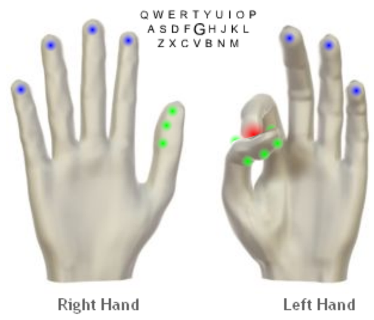


Figure 2.5: KITTY usage Example (Courtesy of KITTY Technologies)

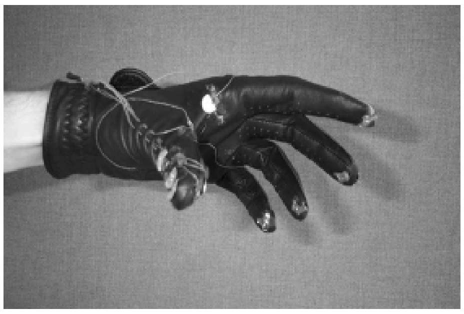


Figure 2.6: KITTY Glove Prototype (Courtesy of KITTY Technologies)

This approach offers significant advantages in portability and usability. The major drawback is, that users have to learn a new input device, which is totally different from traditional approaches like keyboard and mouse.

2.2 Celluon Projection Keyboard

As already mentioned a more detailed focus was set upon the Celluon projection keyboard produced by Canesta Inc. The idea was to remove the bulky hardware replacing the keyboard itself by a projected image. As seen in Fig. 2.7 the device consists solely of a small tower placed in front of the user. When he/she starts typing on this projected image the tower is registrating the key strokes transferring the information to the connected host device (i.e. PDA, cellphone).

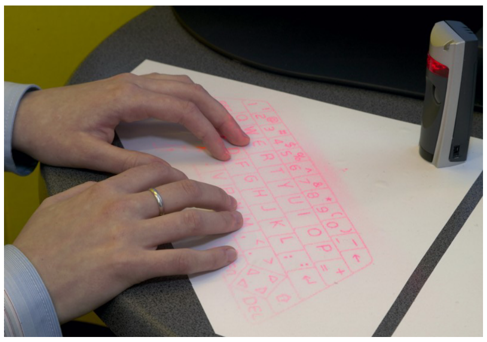


Figure 2.7: Celluon usage Example (Courtesy of Canesta Inc.)

2.2.1 Technical Data

The following description of the technical data is somehow basic, but it was not possible to get any more detailed information due to the patents associated with the Celluon keyboard.

The Celluon device consists of three different parts stored in the tower (see Fig. 2.8). A red laser diode (pattern projector) in the top of the tower projects the keyboard layout onto the table in front of it. Detecting the key strokes is taken care of by the sensor module. In order to actually „see" the user typing the sensor is supported by an IR light source at the bottom of the tower. This emitted IR light can be seen by the sensor, when it hits the fingers of the user (see Fig. 2.9). The whole detection process is simplified and speed up by the fact, that, due to the IR detection, no image processing or gesture recognition is needed.

Celluon buys its advantages with some grave disadvantages, i.e. in the production process. Adjusting the keyboard to other country layouts is not simply done by repainting the keys, but must be incorporated into the pattern projector. Reading the manual revealed some weaknesses of the design: The battery package can only support the device for up to 220 minutes of continuous use. Due to the nature of projection the visibility of the keyboard might not be enough for a working environment under normal light conditions. It also shares the same problem as the foldable keyboard (see Sec. 2.1.1) when it comes to the needed plain surface. The Celluon even has more surface requirements, as it needs one as flat as possible and non-reflective for the detection to work.

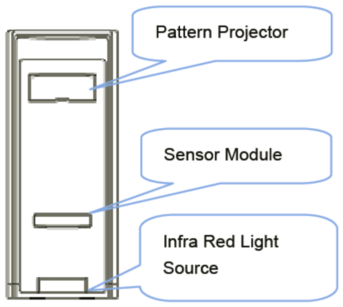


Figure 2.8: Celluon Hardware (Courtesy of Canesta Inc.)

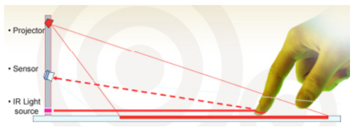


Figure 2.9: Celluon usage Example (Courtesy of Canesta Inc.)

2.2.2 Applications

Customers can already buy the Celluon tower but the more interesting aspect of this product is to incorporate it in existing portable devices like cellphones. Possible applications have been shown on the CeBIT 2005 in Germany [12]. Siemens invested some effort to implement the Celluon into a new cellphone prototype (see Fig. 2.10).

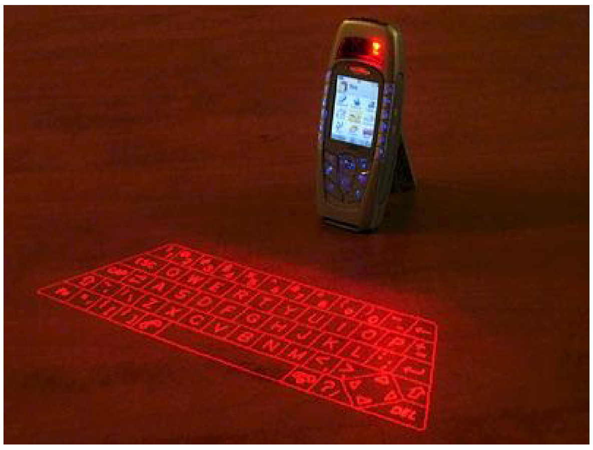


Figure 2.10: Siemens *New Interactive Phone* (Courtesy of Siemens)

Chapter 3

Implementation of a Prototype Virtual Keyboard

The last chapters gave some insight in the field of portable keyboards focusing on the Celluon projection keyboard. The next step was to design and implement a prototype of a virtual keyboard.

3.1 Technical Basics

It was clear from the beginning, that the same approach as the Celluon could not be matched in this thesis. The needed laser diode together with the pattern projector could not be replicated and the algorithms used in the detection processes were unknown too. So only the basic design was kept, but with new hardware components taking the roles of the projector, sensor and IR light source.

The projection problem was solved by a very simple measure: As the result should only be a prototype with the main focus on the detection process, the projection was replaced by a drawn keyboard on a sheet of paper.

Now the detection problem could be broken down into two different areas:

- Threshold detection: When did the user hit a key?
- Overview detection: Which key did the user hit?

3.1.1 Threshold Detection

To stay as close as possible to the usual way a keyboard is used, the moment a key is hit has been defined by the moment the user puts his/her finger down onto the surface. In order to detect the height of the finger above the surface a camera must be placed on one side of the keyboard, having a looking angle of zero degrees (see Fig. 3.1). Moving the camera up higher would result in detection errors. Now it was possible to define a simple area causing an alarm in case the user moves his/her finger into that area, which would then count as „key pressed“.

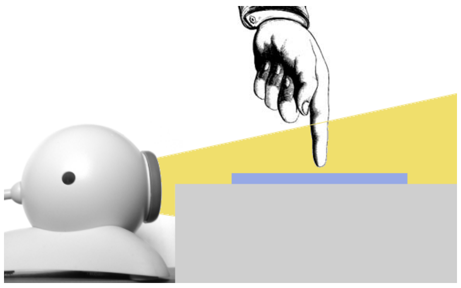


Figure 3.1: Threshold Detection - Camera Placement

3.1.2 Overview Detection

Now with the camera placed on the side of the keyboard area also using it to detect, which key has been hit, was not possible. Optimal requirements for this detection would be an image of the keyboard area made from the top, the moment the user's finger hits the surface. In order to solve this problem a second camera was needed placed in an upper position looking down onto the virtual keyboard (see Fig. 3.2). Now simple rectangle areas could be defined acting as virtual keys.

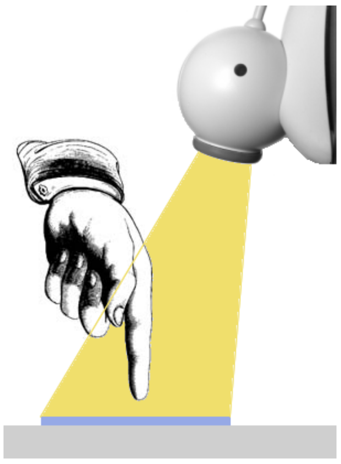


Figure 3.2: Overview Detection - Camera Placement

3.1.3 Camera Hardware

A single Logitech QuickCam Express [13] (see Fig. 3.3) webcam was already available. Being cheap, small and offering a good image quality buying the same model as second camera was a logical choice:

Resolution	max. 640x480 pixel
Colorsystem	RGB or YUV
Framerate	up to 30 frames per second

Figure 3.3: Logitech *QuickCam Express* (Courtesy of Logitech)

3.2 Programming Language and Software

In order to avoid platfrom-specific implementations Java in its newest version 6.0 [15] was chosen as programming language. This decision was also backed up by the extensible knowledge in Java already present especially in creating graphical user interfaces.

To ease up the programming and compilation of the sources, the free Eclipse Project version 3.1 [18] was used as development environment.

3.3 Camera Setup

Initializing and running both cameras turned out to be more problematic than expected, due to shortcomings within Microsoft Windows XP and the Sun Java Virtual Machine.

3.3.1 Problems - Microsoft Windows XP

Microsoft Windows XP was able to detect both USB cameras, but proved to be unable to run them at the same time. It seems that using the same camera is something either Windows or the camera driver did not understand. A test with a different older webcam turned out to be successfull when running both at the same time.

Doing another review of the detection process revealed, that simultaneously running both cameras is not necessary. Only one camera needs to run at startup determining, when the user has crossed the „key pressed“-threshold. As soon as the threshold alarm is generated, the first camera can be disabled. Activating the second camera one single picture can be taken and examined. Now the threshold camera can be reactivated waiting for more key strokes.

3.3.2 Problems - Sun Java Virtual Machine

The standard Java Development Kit [14] is by default not able to access cameras. Supporting these functions Sun developed the Java Media Framework (JMF) [16].

First attempts to access one of the webcams displaying their video stream proved to be successful. Setbacks started when trying to access the second webcam (this did not involve both cameras running at the same time). JMF is accessing webcams via an older Video for Windows (VfW) implementation, which is acting as control interface for the camera. A short research discovered, that Video for Windows is only supporting one device at a time. This would actually not be a problem because just one webcam needs to run at the same time, but switching from one webcam to the other within VfW was not possible. VfW is able to access the first webcam plugged into the system, all others are ignored. JMF did not offer any other way of accessing external cameras and Sun stopped the development, so the idea of using it was discarded.

Research revealed, that Windows is offering another more up-to-date interface to access external cameras called DirectShow which is a part of Microsoft DirectX. Now the problem shifted to the fact, that DirectX is only accessible by changing the programming language to C++ using Microsoft Visual Studio. Some developers already had the same problem and saw the necessity to write a DirectShow interface for Java called Java Media DirectShow (JMDS) [17]. Using this project as a basis, it was possible to control and switch between both cameras. Further tests also included running the two webcams at the same time again, but this always resulted in fatal crashes of the Java Virtual Machine.

3.4 Design

Having selected a programming language, development environment and initialized the cameras, it was now possible to start the application's design process. The taken steps are described in the following sections giving an overview of the requirements for the hardware configuration as well as the user interface and the needed image processing algorithms.

3.4.1 Requirements Analysis

Like for any software project laying out the basic requirements of the application to create is a necessity. These requirements included guidelines for creating the user interface as well as important subjects for the detection algorithm and hardware configuration.

Requirements - Hardware configuration

As described in sections 3.1.1 and 3.1.2 each camera had to be configured individually for its specific task. Before describing the needs for the detection in particular, the following list contains features needed by both detection tasks:

Common for both Detections

- Choosing the correct camera
- Setting the camera format (colors, resolution, framerate, etc.)

More specific needs are described in the following paragraphs:

Threshold Detection

- Lower detection height threshold
- Upper detection height threshold
- Show/Hide visualization of the detection area
- Adjust the detection parameters (determined by the used image processing algorithm)
- Enable/Disable the detection

Overview Detection

- Management of the detection areas acting as keys (create, delete, etc.)
- Show/Hide visualization of the detection areas
- Adjust the detection parameters (determined by the used image processing algorithm)
- Enable/Disable the detection

Requirements - User Interface

Some requirements for the user interface have already been laid out in the previous section, because the user should have the ability to configure all hardware settings in the graphical interface but some more features were still needed:

- Opening a main frame supporting a Multiple Document Interface (MDI)
- Menubar and toolbar to open the different configuration and detection dialogs
- Live-preview of recorded images
- Possibility to save/load configurations

The decision for a Multiple Document Interface was made because of the different dialogs needed. Aside of that the configuration does not had to be visible all the time, when the detection is running, so creating separate windows was the best choice.

3.4.2 Image Processing Algorithms

Having summed up the requirements for camera configuration and user interface some more thinking had to be done about how the detection itself should really work. One possibility was to implement a small gesture recognition by using the optical flow to detect the user's finger when pressing the keyboard. Due to the limited time available for the thesis it was decided to simplify the detection by focusing on the brightness of the object used to press the virtual keys. Searching a do-it-yourself store resulted in using a pen with a small lamp in the front often used in darker working areas. The light itself was bright enough to stand out from the background of the recorded image (see Fig. 3.4).

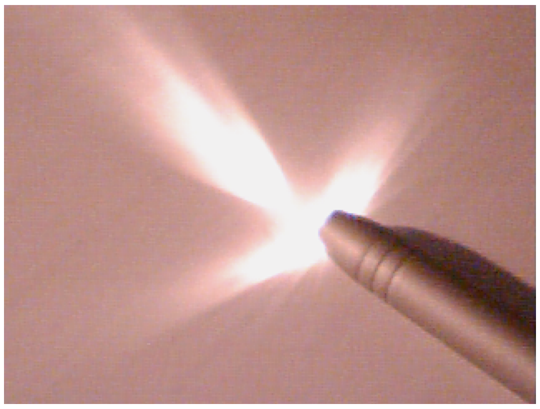


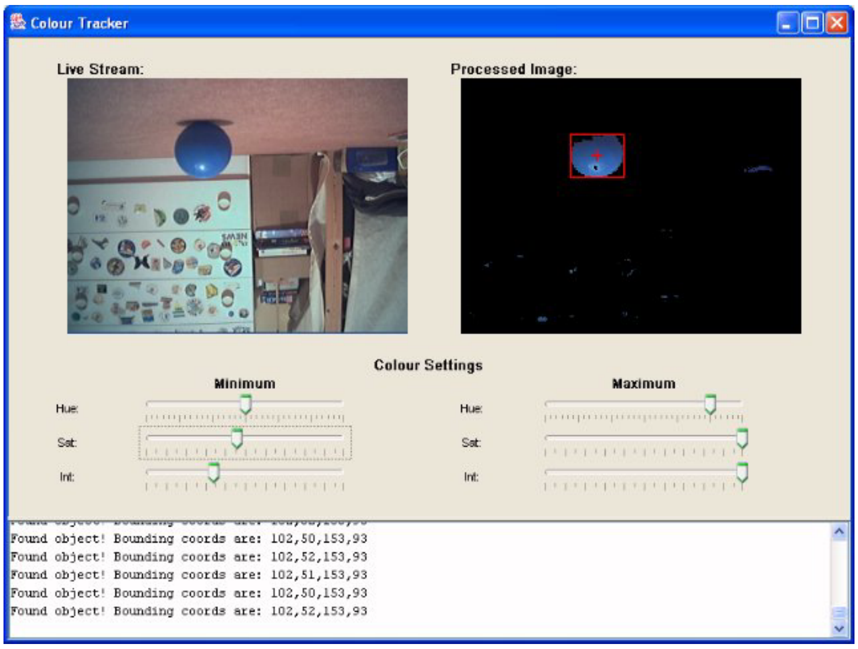
Figure 3.4: Webcam Picture of the Light-pen used for Detection

Now the requirements for the detection algorithm could be narrowed down to the brightness or color of an object. Live configuration of the parameters was a must-have, because light conditions could differ from one room to another. Aside of that, adjustments should be presented as easy as possible to the user resulting in the following additional features:

- Setting the lower detection brightness limit
- Setting the upper detection brightness limit
- Show/Hide visualization of the detection results
- Possibility to make independent settings for both cameras

Focusing on free detection algorithms an internet search turned up a project of David Bull (MSc.) studying at the University of Essex called *Java Colour Tracker* [19].

After taking a computer vision course he implemented a simple modular application focussing on the identification of objects by their color. Practically he used the program to enable a *Lego Mindstorms* robot with a mounted webcam to chase a blue ball (see Fig. 3.5).



```
Found object! Bounding coords are: 102,50,153,93
Found object! Bounding coords are: 102,52,153,93
Found object! Bounding coords are: 102,51,153,93
Found object! Bounding coords are: 102,50,153,93
Found object! Bounding coords are: 102,52,153,93
```

Figure 3.5: Colour Tracker: Detecting a blue Ball (Courtesy of David Bull)

The recorded image is segmented into regions of different color. As long the color of the object to track is known, the regions, whose average color falls within a user specified range, can be selected. Dealing with multiple objects of the same or similar color is tricky, so it was decided, that the largest region will be tracked.

Once the region to be tracked is identified a bounding box for this area is created. The calculated coordinates are then passed to the application to decide, if the threshold has been crossed or which key region has been selected by the user.

Blob Coloring Algorithm

The algorithm[1] itself is known as *Blob coloring* which takes care of finding regions of a specified color in an image. It works by passing a „backwards L“ shaped template (see Fig. 3.6) over the image from left to right and top to bottom.

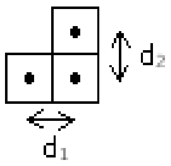


Figure 3.6: Blob Coloring Algorithm: „backwards L“ shaped Template

This is done because it is needed to calculate the color/brightness-distance between the current pixel and the one to the left, and between the current pixel and the one above. Such a „distance between two pixels“ is defined based on the type of the image:

- Grayscale image: the distance is the difference of the two gray levels of the pixels
- RGB image: the distance is equal to the Euclidean distance in the RGB color space
- HSI image: the distance is the difference in hue or itensity

The Euclidean distance in the RGB color space E_{RGB} between two pixels is defined as follows:

$$E_{RGB} = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

A pixel is considered to belong to a different region if the distance d_i between the adjacent pixel is greater than a certain threshold T. There are four possible results, when comparing the current pixel to its left neighbour and the one above:

1. $(d_1 > T)$ and $(d_2 > T)$ - The current pixel is different from both neighbours, so assign it a new region.
2. $(d_1 < T)$ and $(d_2 > T)$ - The current pixel is different from the pixel above, but similar to the left-hand one, so assign it to the same region as the left pixel.
3. $(d_1 > T)$ and $(d_2 < T)$ - The current pixel is different from the left-hand one, but similar to the one above, so assign it the same region as the pixel above.
4. $(d_1 < T)$ and $(d_2 < T)$ - The current pixel is similar to both neighbours, so assign it to the same region as the neighbours.

[1] The description of the algorithm is based on informations by David Bull (see [19])

A problem occurs in case 4 when the current pixel is similar to both neighbours, but the regions for the neighbouring pixels differ. In this situation it has become apparent that although both the neighbouring pixels have different, regions the regions are in fact equivalent (see Fig. 3.7). In the shown example the currently examined template is marked in red. Green pixels have previously been assigned to region „1“ while blue pixels belong to region „2“. Now if the current pixel is similar to both neighbours region „1“ and region „2“ are connected and the same. A solution for this problem is presented in section 3.5.5 on page 23.

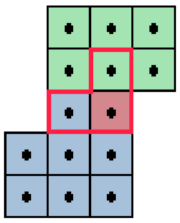


Figure 3.7: Blob Coloring Algorithm: Equivalent Region Problem

3.5 Implementation

By completing the design phase, all requirements to begin with the actual implementation have been fulfilled. Further details would have to be worked out, while they arise during the development.

3.5.1 Additional Software - *log4j*

During the beginning of the implementation the need for additional Java software packages came up. Especially for logging-purposes which is important during the programming and debugging phase of a project but also later on to easily inform the user about certain program actions. Log4j [20] has been chosen due to the facts that it is freely available under the public license and previous experiences have shown, that it is easy to set up and use.

Enabling logging at runtime is easily done without modifying the source code, because the whole logging behaviour is controlled through a properties text file. The output-style is also open to be changed to the programmer's needs.

3.5.2 Camera Usage

Management of both cameras is taken care of by the `CameraManager`. This singleton serves as mediator between the Java Media DirectShow (JMDS) (see Sect. 3.3.2) interface and the main application. `Camera` classes take care of representing the external devices. Due to the singleton nature of the `CameraManager` all classes can access the `Camera` objects, recording single images or collecting video streams.

During the image processing (which is described in section 3.5.5) the recorded images must be altered. In order to ease up this altering process, worker objects can be assigned to every webcam via the `CameraManager`. As long as these workers implement a specific interface, they can be added to the worker lists. Before an image is actually delivered to the application (i.e. the live-preview window), the picture is handed over to all registered workers, so they can perform the image processing. That way new workers can be added easily with the results being displayed in the preview windows immediatly, when the worker is added to the list of the active camera.

3.5.3 Configuration Management

The configuration management is centered around another singleton called `ConfigManager`. This class is responsible for storing all parameters set for both cameras and the image processing. All parameters are following the *key=value* scheme. I.e. `overview.contrast` is the key with its associated contrast value. Configurations can be loaded from or saved to files, which follow the same *key=value* style and can be edited with any text editor. An example file containing the parameters for the threshold camera can be found in Listing 3.1.

Listeners can add themselves to the `ConfigManager`, if they want to get notified about changes to the configuration. All config dialogs are storing their settings in the Manager, which causes an update event to be sent to all registered listeners (i.e. other dialogs) so they can update their data.

```
#
#Sun Apr 09 00:39:15 CEST 2006
threshold.camera=1
threshold.format=2
threshold.brightness=0
threshold.contrast=19
threshold.detect.border.high=58
threshold.detect.border.low=29
threshold.detect.brightness.high=255
threshold.detect.brightness.low=232
```

Listing 3.1: Implementation - Treshold Camera Config File

3.5.4 User Interface

The user interface centers around the main window `VKMainFrame`, which takes care of creating and displaying all buttons, menus and the multiple document desktop area (see Fig. 3.8).

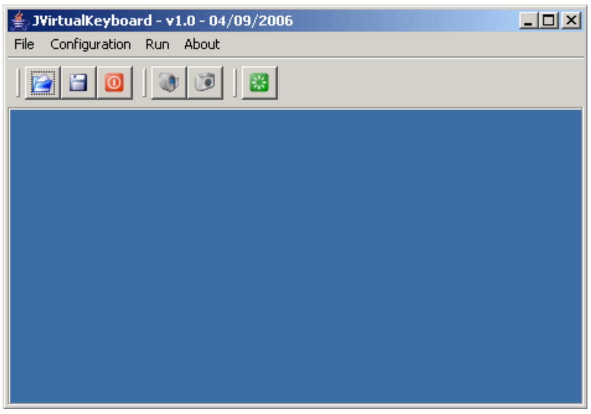
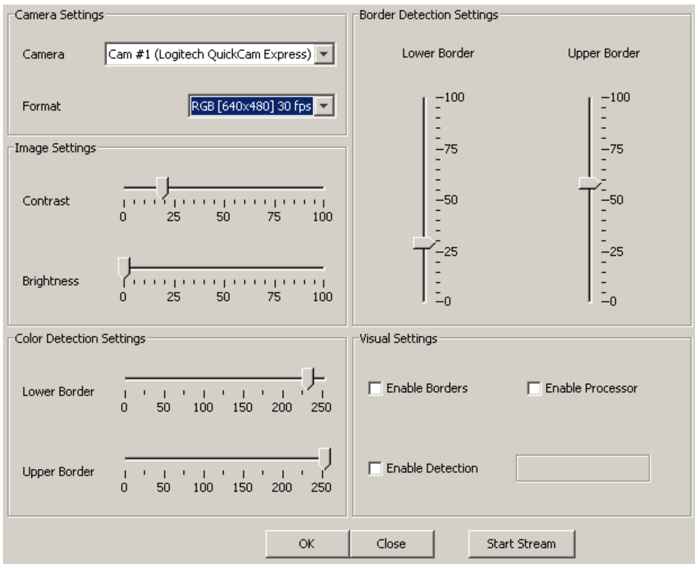


Figure 3.8: User Interface - Main Frame

Configuration - Threshold

This dialog is designed to configure the threshold camera and detection (see Fig. 3.9). It also shows a live preview to control the settings (not visible in the dialog screenshot).



Camera Settings
Camera: Cam #1 (Logitech QuickCam Express)
Format: RGB [640x480] 30 fps

Image Settings
Contrast
Brightness

Color Detection Settings
Lower Border
Upper Border

Border Detection Settings
Lower Border
Upper Border

Visual Settings
Enable Borders
Enable Processor
Enable Detection

OK
Close
Start Stream

Figure 3.9: User interface - Configuration Threshold Dialog (excl. live-preview)

Camera Settings

Enables the user to select the **Camera** to use and their image **Format** (colors, resolution, framerate).

Image Settings

In order to optimize the detection algorithms the **Contrast** and **Brightness** for the image can be changed manually.

Color Detection Settings

These settings apply for the blob coloring algorithm of the threshold detection. Due to the fact, that the problem has been reduced to look for a certain brightness, there are only two sliders (not three as by using RGB colors: red, green, blue). One slider affects all three parts of the RGB color to search for, i.e. RGB(1, 1, 1) or RGB(243, 243, 243) settings are possible. A range of brightnesses to look for can be given by setting the **Lower Border** and **Upper Border**.

Border Detection Settings

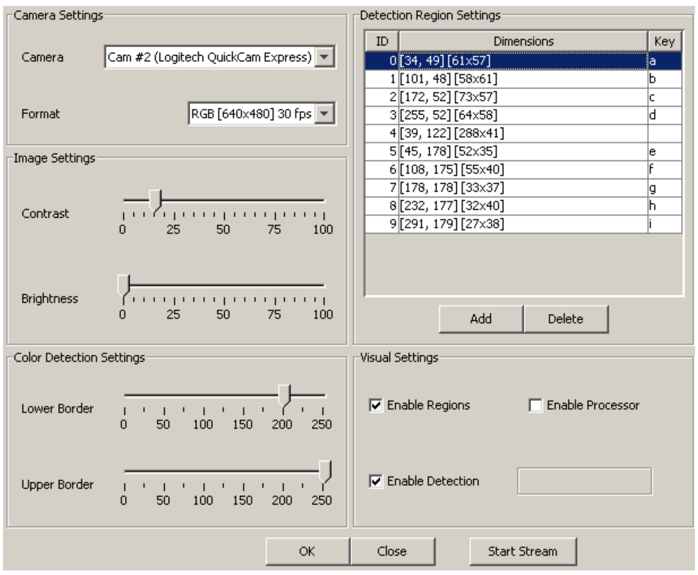
These **Lower Border** and **Upper Border** settings affect the size of the area, where the detection takes place. Only when the user's pen light crosses into this area and is detected, an alarm is generated.

Visual Settings

The checkboxes enable or disable the visualization of the detection process. **Enable Borders** show the set borders directly in the live-preview image, while **Enable Detection** colors all pixels red, which are within the set brightness range. **Enable Processor** starts the detection itself along with the blob coloring algorithm. A bounding box around the object in the preview shows the detection result. The empty label below the last checkbox shows if an alarm has been generated or not.

Configuration - Overview

Parts of this dialog to configure the overview camera match the previously described threshold dialog. Additionally it offers the detection region management to actually map a region to a certain key (see Fig. 3.10).



Camera: Cam #2 (Logitech QuickCam Express)

Format: RGB [640x480] 30 fps

ID	Dimensions	Key
0	[34, 49] [61x57]	a
1	[101, 48] [58x61]	b
2	[172, 52] [73x57]	c
3	[255, 52] [64x58]	d
4	[39, 122] [288x41]	
5	[45, 178] [52x35]	e
6	[108, 175] [55x40]	f
7	[178, 178] [33x37]	g
8	[232, 177] [32x40]	h
9	[291, 179] [27x38]	i

☑ Enable Regions ☐ Enable Processor

☑ Enable Detection

Figure 3.10: User Interface - Configuration Overview Dialog (excl. live-preview)

Detection Region Settings
This list shows all currently set detection regions. These are the actual keys the user can hit on the virtual keyboard. Each region is defined by an id, a rectangle defining the area and a mapped key. When clicking the **Add** button a new region can be appended by simply dragging a rectangle with the mouse pointer in the preview window. The finished rectangle is added to the list. The mapped key can be changed directly in the list by double-clicking on the respective key field. The **Delete** button enables the user to remove a region by simply selecting it in the live preview.

Visual Settings
As before these checkboxes show and hide the results of the detection process. **Enable Regions** draws all set regions into the preview along with their assigned keys. **Enable Detection** and **Enable Processor** match their counterparts in the threshold configuration dialog.

3.5.5 Image Processing

The image processing is the heart of the application and takes care of the entire detection. It is separated into two types of handler classes:

- Image workers, taking care of basic tasks like drawing the selected threshold area into the preview.
- Image processors, these are using the blob coloring algorithm to detect the areas having the selected brightness.

Image workers

As already mentioned image workers take care of the easy tasks. They have to implement the `ImageWorker` interface and can add themself to any cameras' worker list via the `CameraManager`. Currently there are two image worker classes present in the program:

`BorderImageWorker`
This worker is used in the threshold detection and visualizes the set threshold area by simply drawing it into the preview image. Java offers the *XORMode*, which inverts the image drawn into together with a set color. That way all image details are still visible, no matter what color is used for drawing (see Fig. 3.11).

`ColorImageWorker`
The `ColorImageWorker` is used in both the threshold and overview detection dialogs to visualize the brightness, which should be detected. As shown in the config dialogs (see Sect. 3.5.4) a range can be given for the brightness detection. All pixels in the current image falling within this range are colored red by the `ColorImageWorker` (see Fig. 3.12).

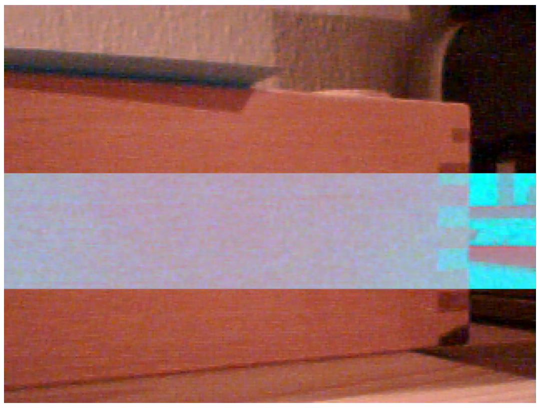


Figure 3.11: User Interface - `BorderImageWorker` Example

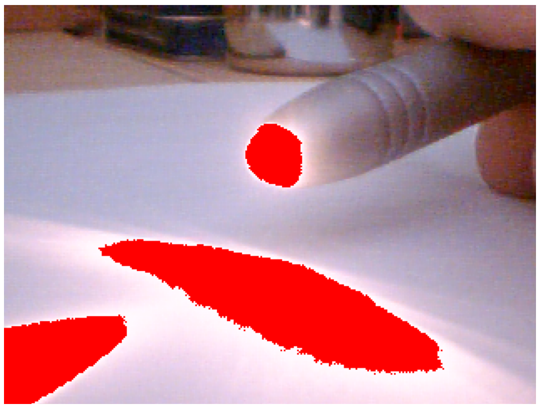


Figure 3.12: User Interface - `ColorImageWorker` Example

Image Processors

The image processors are responsible for the detection work itself and generating alarms if needed. Both the threshold and the overview image processor make use of the blob coloring algorithm (see 3.4.2).

Blob Coloring Algorithm

In implementing the blob coloring algorithm, a two-dimensional integer array is used whose size is equal to that of the image (i.e. 640x480 pixels). This integer array contains the region number for each pixel in the image. So for a given pixel its region number is equal to the value at the same (x,y)-coordinate in the array.

Before the region assignment begins, all pixels in the image are scanned, if they belong to the set color range or not, to speed up the following processing steps. If they are outside of the given range, they are cleared from the image (coloring them black).

Following this pre-processing the „backwards L“ shaped template is used to determine the distance of the current pixel to its left neighbour and the one above it (see Sect. 3.4.2). The fact that all pixels of a different color have been set to black in the previous step is helping in this stage. Calculating the Euclidean distance of the RGB colors is not needed anymore, because only two color regions are present in the image now (black and the colors in the set brightness range). So as soon as one pixel is black and the other one is not, they have a distance of „1“. The four cases (see Sect. 3.4.2) are handled via if-statements with some special cases, when there is no left or above neighbour (i.e. on the left side of the image or at the upper border).

So far the implementation has not been difficult. More problems pose the fourth case of the algorithm, where the current pixel is found to be the same as both neighbours with them having different region numbers. In that case both regions are equivalent aside of their different region numbers. One way to reflect these changes would be to renumber the entire region array taking into account this new fact. However, this would obviously take a lot of processing time as this situation may be encountered numerous times throughout the processing of the image. A better method is to use an array, that maps a region to a different one (see Tab. 3.1).

Region	0	1	2	3	4
Equivalent Region	0	1	1	0	4

Table 3.1: Implementation - Blob Coloring Algorithm - Region equivalence Map

The example shows, that somewhere during the detection it has been discovered, that region 2 is equivalent to region 1 and that region 3 is equivalent to region 0. After scanning the image the region array can be re-calculated, taking this equivalence map into account (i.e. all region 2 pixels will be changed to region 1, etc.).

As stated by David Bull [19] this method is not very time efficient, because it has a drawback: Using the table above what if it is later discovered that region 2 is also equivalent to region 0. Region 2 has already been mapped equivalent to region 1 so if it is changed to make it equivalent to region 0 the information stating, that region 2 is equivalent to region 1 will be lost. So before region 2 can be made equivalent to region 0, region 1 must first be mapped to region 0.

In simple images this is not a problem, but more complex pictures with thousands of regions raise the processing time of one single image up to 15 minutes on an AMD Athlon 2600 CPU with 512MB RAM [19]. In order to process live webcam images this is unacceptable and a solution must be found.

The implemented approach uses a `Vector` of `TreeSets`. Such a vector is a simple one-dimensional resizable array and a treeset is an ordered set. This provides an almost tree-like structure. Figure 3.2 shows the same example as figure 3.1, but this time with the tree structure. Region 1, 2 and 3 are equivalent to region 0 with region 2 also being equivalent to 1.

Region	**Equivalent Regions**
0	
1	0
2	0,1
3	0
4	

Table 3.2: Implementation - Blob Coloring Algorithm - Region equivalence Trees

In order to eliminate unnecessary information for the blob detection, the tree needs to be flattened to always point to the lowest region (i.e. if region 3 points to 2, which points to 1 and 1 points to 0, then starting at region 3 its lowest equivalent region is 0. Propagating this down the path sets 2 to 0 and then 1 to 0). An example is shown in figure 3.3.

Region	**Equivalent regions**	**after flattening**
0		
1	0	0
2	1	0
3	2	0
4	3	0
5	0	0
6	3, 5	0
7		
8	1	0
9	7	7
10	7	7
11	9	7
12	10	7

Table 3.3: Implementation - Blob Coloring Algorithm - Equivalence Table before and after flattening

After flattening the equivalence tree and recalculating the region map, it was necessary to find the biggest detected region, because it is likely, that this is the required blob. The search is done by simply summing up all pixels of a certain region. Now a bounding box for the biggest region could be created by looking for the smallest and largest (x,y)-coordinates of the areas' pixels.

Stopping the detection process at this point, assuming that the biggest region is the result of the detection, caused some errors. As shown in figure 3.12 the biggest region is not always the lamp on the pen but the reflection of the light on the surface. Due to that the blob must be characterized by some parameters telling the algorithm if the found region is really the blob which it should be looking for:

- Absolute ration between height and width of the blobs' bounding box
- Minimum region size in pixels

As shown in figure 3.12 on page 22 the pen blob is clearly different from the blob caused by the light reflection on the surface. Taking into account the absolute ratio R_{wh} seemed a logical approach, because the pen light is more likely to form a circle than the reflection.

$$R_{wh} = \left|\frac{BoundingBox_{width}}{BoundingBox_{height}}\right|$$

Tests showed, that the resulting ratio R_{wh} varies a lot depending on the angle of the pen light. So the parameter was weakened to a range of results. Settings for the threshold and overview image processor had to be made independently for this parameter range due to the different looking angles of the cameras.

Also a minimum region size was added, because small reflections of surrounding light sources at other objects could cause an alarm even if the region was only a few pixels in size.

If a region is found, but not fitting to this parameters it is cleared from the region map and the next biggest region is tested.

`ThresholdImageProcessor`

The image processor taking care of the threshold detection is changing the described implementation of the blob coloring algorithm only marginally. In order to speed up the detection only the area, set by the user, is looked into for blobs, not the whole image. As soon as a blob fitting to the parameters has been found an alarm is generated (see Fig. 3.13).

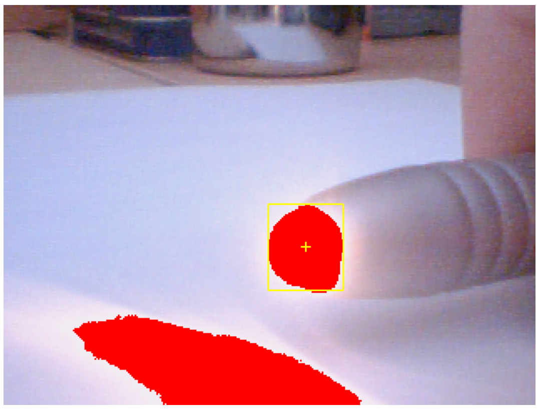


Figure 3.13: User Interface - `ThresholdImageProcessor` Example

`OverviewImageProcessor`

Aside from the blob coloring an additional step had to be taken in this processor after a region and its bounding box have been found. The bounding box is intersected with all key-regions, calculating the area, the detected blob is taking up of the key. This has to be done because the blob never fills out a key in its full size and sometimes two key areas can be affected by the light. Actually activated is the key, which has the largest intersection area with the detected blobs bounding box (see Fig. 3.14).

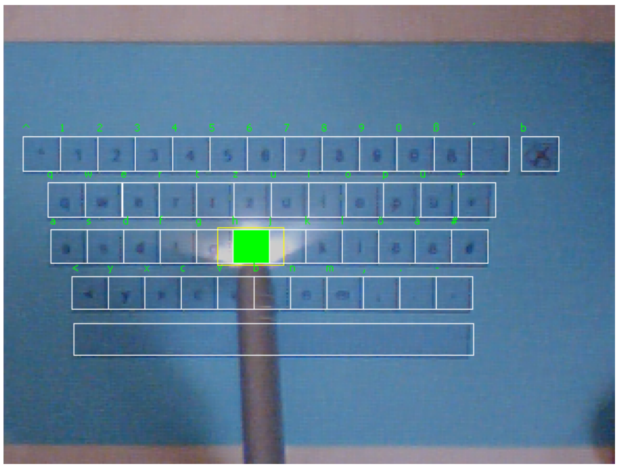


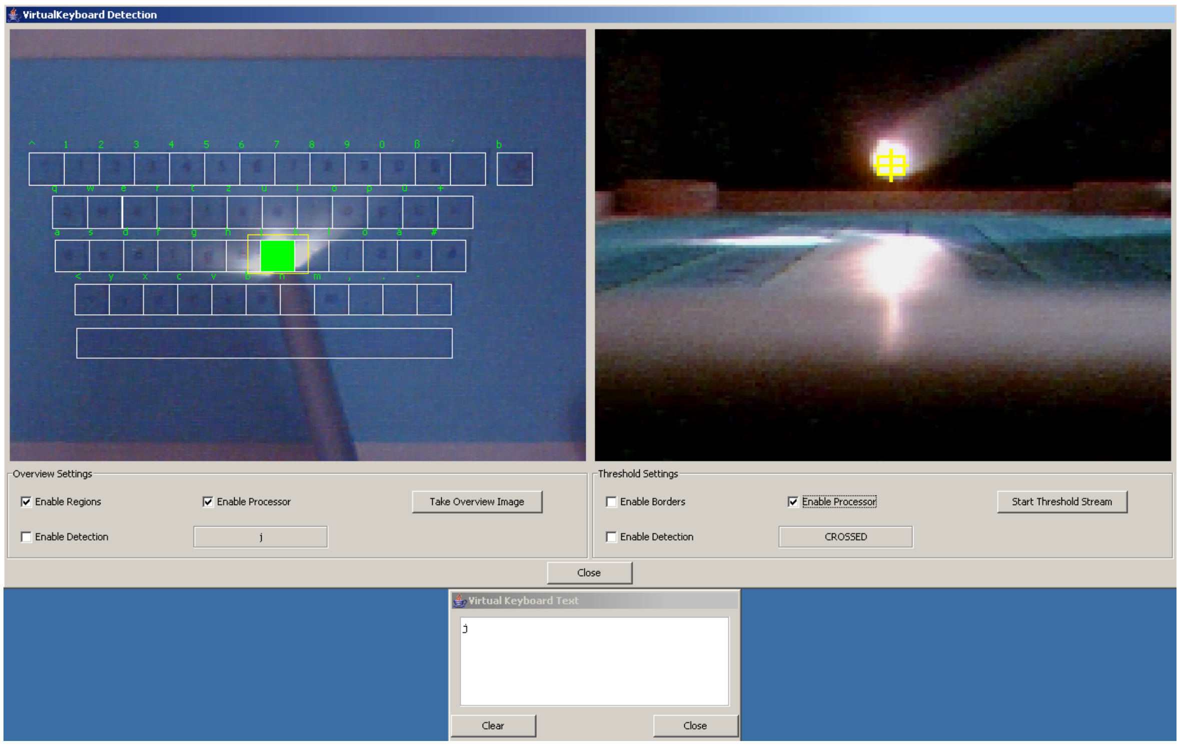
Figure 3.14: User Interface - `OverviewImageProcessor` Example

3.6 Tests and Results

Having completed the programming work testing the application was the next step. To do that, both the threshold and the detection image processor were combined in one window of the user interface (see Fig. 3.15).

The threshold camera (on the right-hand side of the shown image) starts running in a lower resolution (i.e. 176x144 pixels) to achieve a higher framerate and faster reaction to any threshold crossings by the user. As soon as the user crosses into the threshold area with the light pen and the blob is detected an alarm triggers the second camera. To avoid problems by running both cameras at the same time (see Sect. 3.3), the threshold camera is disabled, while the overview webcam takes one single picture at a higher resolution (i.e. 640x480 pixels). This image is then used to detect the blob and determine, which key-area has been hit by the light pen. As soon as the detection is complete the „pressed“ key is displayed in the text area and the threshold camera is reactivated to wait for more key strokes.

While testing the virtual keyboard it was noticed, that switching from the threshold to the overview camera recording a picture and switching back is taking about one to two seconds. This is due to the fact that the threshold camera needs to be disabled and disconnected by the software in order to access the overview camera. Additionally before it is actually possible to take a picture from the camera a mandatory sleep of about 200 milliseconds is necessary, else the delivered picture is just black because the camera did not have time to initialize. The same time is needed on the way back, when reactivating the threshold camera.


VirtualKeyboard Detection

Overview Settings
- [✓] Enable Regions
- [✓] Enable Processor
- Take Overview Image
- [ ] Enable Detection
- j

Threshold Settings
- [ ] Enable Borders
- [✓] Enable Processor
- Start Threshold Stream
- [ ] Enable Detection
- CROSSED

Close

Virtual Keyboard Text

j

Clear Close

Figure 3.15: User Interface - Detection Test

This severly limits the input rate, but was not regarded as a problem due to the prototype stage of this design. To give the user some feedback, when the threshold is crossed and the detection of the key stroke is complete, a sound is played at the beginning and at the end of the process. During that time the light pen has to be kept pointing onto the key to press, else the overview camera might be too slow in taking a picture, before the pen has left the keyboard area again.

Different light conditions around the keyboard have a great impact on the detection parameters. Especially in brighter rooms setting the brightness detection range for the blob coloring algorithm can be a bit tricky. Manual adjustment is always necessary before the keyboard can be used.

Reflections on the keyboard surface are still a problem during the overview detection. Sometimes the light pen is as bright as the reflection from the surface. This confuses the detection and results in either one large quadratic blob made up of the pen and the reflection or in no detection at all, because the blob suddenly has the wrong width-to-height ratio. The blob coloring algorithm itself has proven to be very adaptive, but still works somewhat slow with higher resolution pictures (i.e. 352x288 pixels and above).

Aside of these drawbacks tests have been completed with a high success rate as long as the described limitations are taken into account.

Chapter 4

Conclusions

This chapter concludes the semester thesis documentation, gives a review of the project and sums up the results. Additionally some suggestions for future developments are given.

4.1 Review

In this thesis the main focus has been on portable keyboards and their development. Showing the current state of development, giving examples and discussing their pro and cons provided a good overview of that area.

Following the introduction a virtual keyboard has been designed and implemented starting with a review of the technical basics and the camera hardware. Choosing a programming language, development environment and setting up the cameras led to the design stage of the application. Defining the requirements of both the application, user interface and the image processing algorithms were a necessity, before the actual programming work could be started.

Implementing the camera connection, user interface and especially the blob coloring algorithm made up most of the practical part. Also configuration of the cameras and the detection should be as easy and as visible as possible within the application. Testing the virtual keyboard and discussing the results completed this chapter.

4.2 Results

It has been shown that the old keyboard is far from being extinct. All current developments have certain advantages, but also sometimes big drawbacks or limitations. The goal to discuss these developments has been fulfilled.

The created application showed the complexity of replacing normal keyboard strokes by other detection methods. Connecting two wires together as it is done in every old keyboard is still fast and not prone to errors. Still implementing a virtual keyboard using two cameras has been challenging. Aside of the problems Microsoft Windows XP has by running two webcams of the same type at the same time the image processing and detection proved to be interesting.

4.3 Future Work and Developments

As described in section 3.6 the tests showed some shortcomings and drawbacks of the prototype. Perhaps in the future, further developments in webcam drivers and camera access will make it possible to run both cameras at the same time eliminating the long waiting period, when taking a picture from the overview camera.

Focusing on a light pen also severly limits the user's ability to control the keyboard. Most likely a finger detection, using the optical flow, is better suited for such an application. Still, using image processing as detection always needs a fast hardware to get results within an acceptable time frame. So either the detection itself needs to be more simplified or replaced by another approach (perhaps similar to the one used in the Celluon keyboard described in section 2.2) or the algorithms need to be run on special-purpose hardware to speed them up. The first variant is surely cheaper.

Bibliography

[1] C. Mehring, F. Kuester, S. Kunal Deep, M. Chen: „*KITTY: Keyboard Independent Touch Typing in VR*“, IEEE Virtual Reality 2004.

[2] Canesta Inc., *Homepage*, April 2006, http://www.canesta.com/

[3] Canesta Inc., *Celluon*, April 2006, http://www.canesta.com/html/celluon.htm

[4] Stanford University Libraries & Academic Information Resources, *SiliconBase - The Mouse*, April 2006, http://www-sul.stanford.edu/siliconbase/wip/control.html

[5] Eleksen, *Homepage*, April 2006, http://www.eleksen.com/

[6] Eleksen, *Wireless fabric keyboard*, April 2006, http://www.eleksen.com/index.asp?page=products/wirelesskeyboard/keyboard_1.asp

[7] Hama GmbH & Co KG, *Homepage*, April 2006, http://www.hama.de/

[8] Hama GmbH & Co KG, *Bluetooth Freedom Mini Keyboard* , April 2006, http://www.hama.de/portal/articleId*126221/action*2563

[9] Garmin, *StreetPilot 2660*, April 2006, http://www.garmin.de/Produktbeschreibungen/StreetPilot2660.php

[10] Kiosk Mode, *Homepage*, April 2006, http://www.kiosk-mode.com/

[11] KITTY Tech, *Keyboard Independent Touch Typing Technology*, April 2006, http://www.kittytech.com/

[12] teltarif.de, *Siemens testet die virtuelle Tastatur*, 22nd March 2005, http://www.teltarif.de/arch/2005/kw12/s16621.html

[13] Logitech, *QuickCam Express*, April 2006, http://www.logitech.com/index.cfm/products/details/DE/DE,CRID=2204,CONTENTID=5037

[14] Sun Microsystems, *Java Technology*, April 2006, http://java.sun.com/

[15] Sun Microsystems, *Mustang: Java SE 6*, April 2006, https://mustang.dev.java.net/

[16] Sun Microsystems, *Java Media Framework API (JMF)*, April 2006, http://java.sun.com/products/java-media/jmf/

[17] java.net Projects, *Java Media DirectShow (JMDS)*, April 2006, https://jmds.dev.java.net/

[18] Eclipse.org, *Homepage*, April 2006, http://www.eclipse.org

[19] David Bull, *Projects - Java - Colour Tracker*, http://www.uk-dave.com/projects/java/colourtracker.php

[20] Apache, *Log4j project*, April 2006, http://logging.apache.org/log4j/docs/index.html

Abbreviations

JMDS	Java Media DirectShow [17]
JMF	Java Media Framework
KITTY	Keyboard Independent Touch Typing [11]
MDI	Multiple Document Interface
VfW	Video for Windows

Glossary

JMDS - Java Media DirectShow
JMDS provides a Java wrapper around the Microsoft DirectShow Capture API and exposes them as a Java Media Framework DataSource.

JMF - Java Media Framework
The Java Media Framework is a Java library handling audio and video signals. The API supports capturing from microphones and cameras and allows reading and writing of audio and video files.

MDI - Multiple Document Interface
Multiple Document Interface describes a specific format of graphical user interfaces. MDI offers the ability to open more than one document in a surrounding main frame. These documents are opened within separate subwindows which can be freely arranged and resized within the main frame.

VfW - Video for Windows
Video for Windows is a simple software interface for Microsoft Windows, enabling a programmer to encode and decode video signals as well as to read from framegrabber cards. It is the standard interface for AVI.