Hand Mouse: Real Time Hand Motion Detection System Based on Analysis of Finger Blobs

Ibrahim Furkan Ince [*1], Manuel Socarras-Garzon [*2], Tae-Cheon Yang [*3]
[*1] *Graduate School of Digital Design, Kyungsung University, Busan, South Korea*
[*2], ***Corresponding author*** *Busan, South Korea*
[*3] *Department of Computer and Information Science, Kyungsung University, Busan, South Korea*
furkan@ks.ac.kr, sly.nomad@googlemail.com, tcyang@ks.ac.kr
doi: 10.4156/jdcta.vol4.issue2.5

Abstract

Hand detection is a fundamental step in many practical applications as gesture recognition, video surveillance, and multimodal machine interface and so on. The aim of this paper is to present the methodology for hand detection and propose the hand motion detection method. Skin color is used to segment the hand region from background and hand blob is extracted from the segmented finger blobs. Analysis of finger blobs gives us the location of hand even when hand and head blobs are visible in the same image. In this paper, we propose a fast, computationally inexpensive solution which uses any type of computer video camera to control a cursor through hand movements and gesticulations. The design and evaluation phases are presented in detail. We have performed extensive experiments and achieve very encouraging results. Finally, we discuss the effectiveness of the proposed method through several experimental results.

Keywords

Hand tracking, Hand motion detection, Hand mouse, Haar-like features, AdaBoost classifier, Lucas-Kanade optical flow tracker, Blob extraction.

1. Introduction

The aim of this paper stems from the fact that as computer systems continues to become ever more ubiquitous, the more the need arises for a complimentary input interface. With hardware manufacturers on the cusp of releasing the next generation of graphical display systems using the latest in 3D imagery [7], software developers will naturally follow suit with 3D operating systems, games and so forth. From this it is fair to assume that anyone with a 3D solution for user input would benefit greatly if a product is ready to go into the market in time. Just as the mouse was developed as a more intuitive means of controlling a cursor in a 2D graphical interface, the next logical progression would be to have an interface which uses the full range of movement of the hand. Contending technologies include the various types of gloves and wands used in virtual reality. Though a simple solution their cost is far too high to soften the consumer market. This is based on the belief that people are less likely to buy a new input peripheral when a mouse can do the job. Computer science however offers an elegant solution requiring little in manufacturing costs which lies in the field of computer vision. Current strands of research into object and motion detection is more than advanced enough to tackle the problem of finding a hand in a scene and deriving position and purpose accurately. HandMouse contains elements of the Open Computer Vision Library (OpenCV) to process incoming frames from a standard USB camera. Hand Detection is achieved through the use of a detection method which utilizes an extended set of Haar-like features with a cascaded AdaBoost classifier as proposed in [9]. Motion tracking is achieved through a modified Lucas-Kanade optical flow tracker as outlined in [2]. The aim of this paper is to demonstrate the feasibility that with existing computer vision technologies a fast and computationally inexpensive solution for human computer interaction is possible. We also outline the problems with the current implementation and suggest possible answers to resolving them.

2. Background

The camera has been used in the project was Logitech 2 megapixel web camera without infra-red filter. We did not apply zoom feature to focus on the image and get more precise image data features. In order to understand what has been done in the project, first of all, basic terms have to be explained in detail.

2.1 Object Detection

Recognition of an object in a scene is a key field of study in computer vision. It is also central to our proposal of controlling a computer through the detection of hand gestures to match pre-determined commands. Though there are varieties of methods by which this can be achieved, we will only outline that which is necessary for understanding our implementation. Our chosen detection method is the machine learning technique of boosting to efficiently classify haar-like features within a video frame from a pre-trained classifier cascade as proposed in [13] and expanded upon in [9]. The advantages of this method are that when implemented correctly it is fast, efficient and accurate. It is also effective in detecting objects which are either partially occluded or if the video frame is noisy.

2.1.1 Haar-like Features

These are rectangular areas within a detection window which are used to determine either a line, edge, or point based on the weighted sum of the pixels within each feature area. Figure 2.1 shows both the original feature set as outlined in [13] along with the extended set of 45˚ rotated rectangles as proposed in [Lienhar02]. The black rectangles are negatively weighted while the white rectangles are given positive weightings. Rectangles are used due to their extreme computational efficiency. As Figure 2.1 illustrates it is very easy to calculate the dimensions of a rectangle within a window by specifying it in the tuple $r=(x, y, w, h, \alpha)$ with $0 \le x, x+w \le W,\ 0 \le y, y+h \le H,\ x,y,w,h \ge 0$ and $\alpha \in \{0^\circ, 45^\circ\}$.

Using a detection window of 24x24 pixels, with all possible combinations of horizontal and vertical locations and scales (restricted to within a scaling factor of $X=W/w$ and $Y=H/h$) gives a set of 117,941 features. From this useful information can be derived about the window which would otherwise be difficult to learn from the raw set of pixel values.

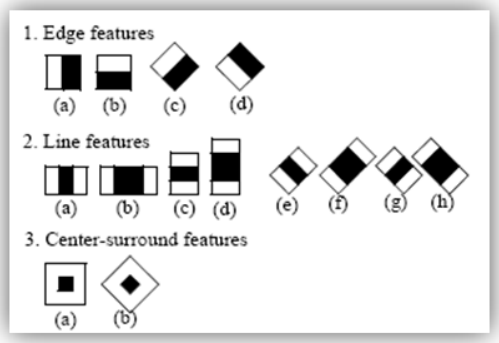


Figure 2.1 Haar-like feature prototypes

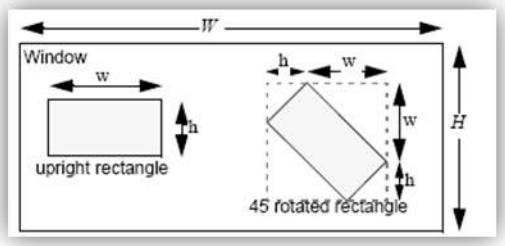


Figure 2.2 Example of an upright and 45˚rotated angle

$$feature_I = \sum_{i \in I=\{1,\ldots,N\}} \omega_i \cdot PixelSum(r_i)$$

Equation 2.1 Equation for an arbitrary haar-like feature where

$\omega \in \{-1,1\}$, r is a given rectangular area, and N is the number of such areas in the given feature.

2.1.2 Feature Calculation

The computation of all the features within a window can be performed very quickly and in constant time for any size by using two value tables known as the summed area table (SAT) for upright rectangles and the rotated area summed table (RSAT) for rotated rectangles. Both of these are the sums of pixel values within an rectangular area where

$$SAT(X,Y) = \sum_{x \le X, y \le Y} I(x, y)$$

Equation 2.2 Summed Table Area

$$RSAT(X,Y) = \sum_{x \le X, x \le X - |Y-y|} I(x,y)$$

Equation 2.3 Rotated Summed Table Area

By using these, the sum of pixel intensity values can be achieved in just four table lookups with

$$PixelSum(r) =$$
$$SAT(x-1,y-1) + SAT(x+w-1,y+h-1)$$
$$- SAT(x-1,y+h-1) - SAT(x+w-1,y-1)$$

Equation 2.4 For upright rectangle areas and

$$PixelSum(r) =$$
$$RSAT(x+w,y+w) + RSAT(x-h,y+h)$$
$$- RSAT(x,y) - RSAT(x+w-h,y+w+h)$$

Equation 2.5 For rotated rectangle areas. A more detailed description can be found in [9].

2.1.3 Classification

With a method of distinguishing image samples available it is then possible to apply machine learning techniques to classify whether or not an input image contains a hand object. In this case adaptive boosting is used as it is a powerful and fast learning method. It comprises of a combination of weak classifiers with a very high positive hit rate and a false-positive rate which is barely better than chance. This combination forms a strong classifier. The specific boosting variant used is known as Gentle AdaBoost. The algorithm is shown on the next page. It is also discussed in greater detail in [6]. Learning is based on N number of training samples from $(x_1, y_1), \ldots, (x_N, y_N)$ where x being an training image and $y \in \{-1,1\}$ with a negative sample being -1 and a positive 1.

1. Given N image examples from $(x_1, y_1), \ldots, (x_N, y_N)$ with $x \in \Re^k, y \in \{-1,1\}$
2. Start with weights $w_i = \frac{1}{N}, i = 1,\ldots,N$
3. For each feature m, m=1,…,M
   - ✓ Fit the regression function $f_m(x)$ by weight least-squares of y_i to x_i with weights w_i.
   - ✓ Set $w_i \leftarrow w_i \cdot e^{-y \cdot f_m(x)}, i = 1,\ldots,N$ and renormalize weights so that $\sum_i w_i = 1$.
4. Output the classifier $sign\left[\sum_{m=1}^{M} f_m(x)\right]$

Figure 2.3 Gentle AdaBoost training algorithm

At each stage of this cascade of weak classifiers, a stump classifier as shown in point 4 above is used to determine whether or not the input image contains the object based on the calculated weightings in the above algorithm. If a stage determines that it is above a threshold value, in our case greater than 0, it will pass on the input to the next stage. This process filters off negative training examples until the possibility of a false negative is negligible. Figure 2.4 illustrates this cascade process.

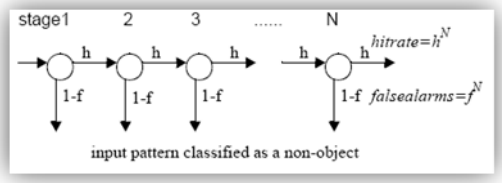


Figure 2.4 Cascade of classifiers with N stages.

At each stage a classifier is trained to achieve a hit rate of h and a false alarm rate of f.

Using say 20 stages with a minimum hit rate of 99.5% and a maximum false alarm rate of 50% per feature layer will give an overall hit rate of about 90% and a false alarm rate of about 9.5×10^{-5} %.

2.1.4 Detection

During the actual detection process a sliding window is passed through an input video frame and passed through the detection cascade which either confirms the detection of a hand or rejects the input. To take into account scaling, the window input is recalled and similarly passed through the cascade until either a hand has been detected or the input window exceeds the dimensions of the video frame.

2.2 Motion Detection

The other key area of research which is pertinent to our work is that of detecting motion in a scene based on video input. Though there are a number of methods to achieve this, we have opted optical flow estimation.

2.2.1 Optical Flow

Optical flow is the approximated motion between two video frames based on the difference in intensity values over time. To identify optical flow between two consecutive video frames *I* and *J*, we start with an initial point in *I* $u = [u_x, u_y]^T$ and try to match it with its corresponding point *v* in *J*, $v = u + d = [u_x + d_x, u_y + d_y]^T$. The vector ***d*** is being the optical flow at that specific point.

Unfortunately the process of finding this vector is not altogether that easy given a number of issues including one known as the aperture problem. This is an ambiguity of the optical flow within areas where there is an insufficient amount of spatial gradient variation.

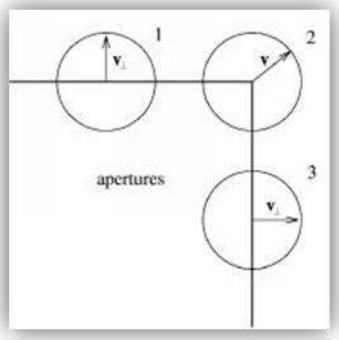


Figure 2.5 The aperture problem

As can be seen in figure 2.5 when a rectangle is moving up left diagonally, the apparent motion from within apertures 1 and 3 would not take into account of horizontal and vertical motion respectively. The perceived motion within aperture 2 is correct given that it has sufficient edge gradient variation. It is therefore important to observe variation over an area large enough to distinguish motion correctly. This can be done by the defining optical flow value which minimizes the function ε within a window *w* of size (w_x, w_y) respectively as

$$\varepsilon(d) = \varepsilon(d_x, d_y) = \sum_{u_y - w_y}^{u_y + w_y} \sum_{u_x - w_x}^{u_x + w_x} (I(x, y) - J(x + d_x, y + d_y))^2$$

Equation 2.6

The matching equation above is central to the Lucas-Kanade feature tracking algorithm outlined in the following section.

2.2.2 Lucas-Kanade Feature Tracking Algorithm

As previously mentioned, the aim of a feature tracker is to find the displacement vector ***d*** that minimizes the matching function in equation 2.6. This means that the first derivative of ε respect to ***d*** is 0. Substituting $J(x + d_x, y + d_y)$ by its first order Taylor expansion about the point $d = [0 \quad 0]^T$ therefore gives the approximation.

$$\frac{\partial \varepsilon(d)}{\partial d} \approx -2 \sum_{u_y - w_y}^{u_y + w_y} \sum_{u_x - w_x}^{u_x + w_x} (I(x, y) - J(x, y) - \left[\frac{\partial J}{\partial x} \quad \frac{\partial J}{\partial y}\right] d) \cdot \left[\frac{\partial J}{\partial x} \quad \frac{\partial J}{\partial y}\right]$$

Equation 2.7

The matrix $\left[\frac{\partial J}{\partial x} \quad \frac{\partial J}{\partial y}\right]$ being the image gradient vector for *J*. The expression $I(x, y) - J(x, y)$ can also be considered to be the derivative over time of a scene *S*. It can then redefine it as in equation 2.8.

$$\partial S(x, y) = I(x, y) - J(x, y) \qquad \textbf{Equation 2.8}$$

$$\nabla S = \begin{bmatrix} S_x \\ S_y \end{bmatrix} = \left[\frac{\partial J}{\partial x} \quad \frac{\partial J}{\partial y}\right]^T \qquad \textbf{Equation 2.9}$$

Image derivatives S_x and S_y can also be computed independently in the first image so long as a central difference operator is used, which in this case it is, as so.

$$S_x = \frac{\partial I(x, y)}{\partial x} = \frac{I(x + 1, y) - I(x - 1, y)}{2} \qquad \textbf{Equation 2.10}$$

$$S_y = \frac{\partial I(x, y)}{\partial y} = \frac{I(x, y + 1) - I(x, y - 1)}{2} \qquad \textbf{Equation 2.11}$$

Updating equation 2.7 with this new notation now gives.

$$\frac{1}{2}\cdot\frac{\partial\varepsilon(d)}{\partial d}\approx \sum_{u_y-w_y}^{u_y+w_y}\sum_{u_x-w_x}^{u_x+w_x}(\nabla S^T d-\partial S)\nabla S^T \qquad \textbf{Equation 2.12}$$

$$\frac{1}{2}\left[\frac{\partial\varepsilon(d)}{\partial d}\right]^T \approx \sum_{u_y-w_y}^{u_y+w_y}\sum_{u_x-w_x}^{u_x+w_x}\begin{bmatrix} S_x^2 \\ S_xS_y \end{bmatrix} - \begin{bmatrix} \partial S & S_x \\ \partial S & S_y \end{bmatrix} \qquad \textbf{Equation 2.13}$$

This equation is now separated into two components, the spatial gradient matrix ***G*** and the image mismatch matrix ***b***. This is needed for efficient computation and will be discussed further on in this section.

$$G=\sum_{u_y-w_y}^{u_y+w_y}\sum_{u_x-w_x}^{u_x+w_x}\begin{bmatrix} S_x^2 & S_xS_y \\ S_xS_y & S_y^2 \end{bmatrix} \qquad \textbf{Equation 2.14}$$

$$b=\sum_{u_y-w_y}^{u_y+w_y}\sum_{u_x-w_x}^{u_x+w_x}\begin{bmatrix} \partial S & S_x \\ \partial S & S_y \end{bmatrix} \qquad \textbf{Equation 2.15}$$

Finally from equation 2.13 the optimum flow vector is

$$d_{opt}=G^{-1}b \qquad \textbf{Equation 2.16}$$

This equation is known as the standard Lucas-Kanade optical flow equation which holds so long as motion between frames is small. This is because the higher order terms in equation 2.7 have been ignored as there are of negligible value if motion is small. This makes computation quicker at the expense of accuracy. This can however be minimized if an iterative approach to solving the above equations is performed. Iteratively refining the image mismatch matrix is achieved by recalculating v_{opt} until the updated optical flow value is either less than a pre-defined threshold or having completed a maximum number of iterations. This method is expanded further in [10].

2.2.3 Pyramidal Tracking Algorithm

A major drawback with the Lucas-Kanade algorithm is that it assumes optical flow between frames will be small. Given the frame rates of a typical USB camera is on average about 15fps, optical flow with respect to hand movements is too large to be detected accurately. A simple solution to this problem has been suggested in [2] wherein a pyramidal approach is defined. In this method optical flow is estimated in a series of increasing image resolutions producing a further iteratively refined value as in the previous section. The advantage of this approach is that it is therefore possible to calculate large optical flow estimates in the actual image resolution from small optical flow estimates in lower-resolution images. Figures 2.6 and 2.7 illustrate this.

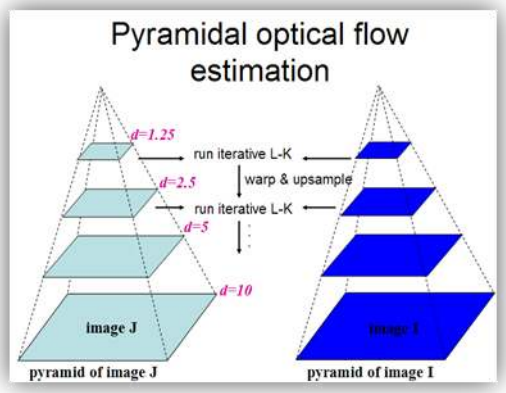


Figure 2.6 – Pyramidal optical flow estimation.

Notice how a small disparity found at the top of the pyramid equates a large disparity at the bottom.

1. Create multi-resolution pyramid for images I and J. $\{I^L, J^L\}, L=0,..,L_m$
2. Initialize pyramidal optical flow estimate $g^{L_m}=\begin{bmatrix} g_x^{L_m} & g_y^{L_m}\end{bmatrix}^T=\begin{bmatrix}0 & 0\end{bmatrix}^T$
3. For $L=L_m$ down to 0
   - ✓ Define tracked point in I^L, $u^L=\frac{u}{2^L}$
   - ✓ Calculate image derivatives ***S*** with respect to *x* and *y*. (equations 2.10 and 2.11)
   - ✓ Calculate spatial gradient matrix ***G***. (equation 2.14)
   - ✓ Initialize iterative LK optical flow estimate $d^0=\begin{bmatrix}0 & 0\end{bmatrix}^T$
   - ✓ For *k=1* to *K* (or until $\|d^k\|<threshold$)
     - ➢ Calculate image difference ∂S_k^L (equation 2.8)
     - ➢ Calculate image mismatch matrix b_k (equation 2.15)

- Calculate iterative LK optical flow estimate d^k (equation 2.16)

$$d^k += d^{k-1}$$

- ✓ Calculate pyramidal optical flow estimate for next level

$$g^{L-1} = \begin{bmatrix} g_x^{L-1} & g_y^{L-1} \end{bmatrix}^T = 2(g^L + d^K)$$

4. Final optical flow vector $d = g^0 + d^k$
5. Point tracked on J to point $u + d$

Figure 2.8 Pyramidal feature tracking algorithm

2.2.4 Blob Detection and Extraction

Blob detection is a fast and sub pixel precise detection of small, compact image primitives called as "blobs". The algorithm is based on differential geometry and incorporates a complete scale-space description. Hence, blobs of arbitrary size can be extracted by just adjusting the scale parameter. In addition to center point and boundary of a blob, also a number of attributes are extracted. Blob extraction is an image segmentation technique that categorizes the pixels in an image as belonging to one of many discrete regions. Blob extraction is generally performed on the resulting binary image from a thresholding step. Blobs may be counted, filtered, and tracked [16].

Fig 1. Numerous blobs were extracted from the source image by using OpenCV Blob Extraction library.

OpenCV blob extraction library finds many blobs [17]; however, purpose of the system and proposed algorithm is to get only hand blob among all the blobs.

3. Technical Basis

In this section, we will be stating the different stages of development which we had to go through to produce the current version of HandMouse. Besides it being our first experience at developing a computer vision application, this project was also our first attempt at developing a whole application using the components of the Microsoft Windows API. Given our chosen task was to develop a computer vision based cursor input alternative that is both responsive and at the same time uses little resources we had a lot of interesting technical challenges ahead of us. Amongst these included how to take in video input from an arbitrary video source, how to process individual frames and detect both a hand gesticulation and its motion over time, and how best to process this information so as to make it useful for cursor input.

3.1 GUI Design

The first step was to have a user interface to work from. As indicated earlier our knowledge of GUI design was limited and the only experience we had was working with Java Swing. Therefore we needed to learn how to develop one in Visual C++ so as to take advantage of certain features of windows programming such as timers and access to the mouse at the operating system level. We therefore began researching windows GUI programming by reading a number of tutorials online, most notably [15], [11]. At this time we began to experiment with code examples from this site to get an idea of what needed to be done. Our attempts were at creating a GUI from scratch with the help of the code examples offered in these sites. What we learnt was that an MFC application window consists of three distinct layers of objects which perform different tasks namely the frame, the view and the document. These are illustrated in figure 3.1.

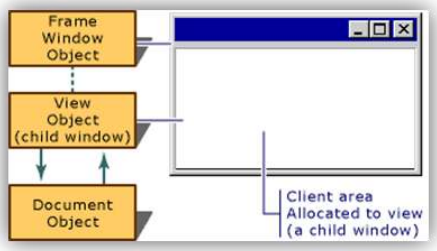


Figure 3.1 Window Design Framework

The frame object is the basis of the window itself. It provides a visible frame surrounding the contents, wherein a status-bar, toolbar, and menu bar could be placed as well as control how the window looks and acts. Within the frame is the view object which manages the contents within the frame. It acts as a holder for any object, which in this case is the video output from the camera and provides functions to

control the look. The document object holds all the code to display the video frames we be manipulating. This is covered in the next section. The main advantage of this approach was that with Visual Studio it is incredibly simple to add controls such as sliders, buttons etc. The process is to create a control visually in resource manager, provide it an integer ID in the resource header file and establish a link to a variable via dynamic data exchange (DDE). This is a protocol for exchanging data between and within applications using Windows messages. It is simpler than COM though more than sufficient a means for letting objects know that the user has clicked on something. This allowed us to avoid a monolithic *WinProc()* function and better modularize program functionality. Later on we decided that we wanted the program to minimize to the system tray in the same manner as volume control dialog. This was so as to make the program as inconspicuous as possible and was rather simple to code. Using an article we had found on the MSDN site [5] we learnt how to we add a function in the frame object class to create a control message which configures the icon appearance and then send it to the taskbar using *Shell_NotifyIcon()*. With a GUI ready to go, we were could now begin work on capturing video input.

3.2 Video Capturing

The next step in development was to have a means of capturing video input from an arbitrary video capture device. This ruled out the idea of communicating directly with the device as it would have been dependent on hardware such as interface type with little benefit. At this time we had began to look into off the shelf computer vision libraries and had found an ideal candidate in the open computer vision library [12]. In addition to providing all the functions we required to carry out the image processing, it also had functions for handling video I/O. These used the older Video for Windows interface from Windows 3.1 which was sufficient for the moment. This library also had the added advantage of having all the code ready for easily porting the app Linux through the Video for Linux interface. We however needed to consider future development and opted to use as little OpenCV code as possible so that we could later write our own implementation and eventually seek to patent our work. Portability to Linux was also not a priority for this project. In addition to this the VFW interface is obsolete thereby making any future development using it pointless. The best solution was therefore to use DirectShow which works by way of combining components known as 'filters' used to process incoming video data to form a 'filter graph'.

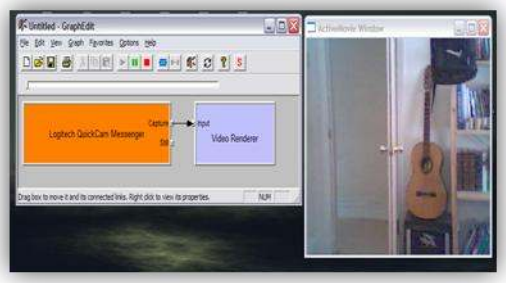


Figure 3.2 Example of a Simple DirectShow Graph

Using the *GraphEdit* application, we were able to quickly make prototype graphs like the one seen in figure 3.2. This shows the simplest possible graph with just the filter relating to the desired capture device streaming video to the video renderer. We began by looking at the sample code provided in the DirectX SDK, most notably the app *AMCap* which showed how to preview and capture video from any recognized video input. *SampleGrabCB* was also useful in showing us how to intercept frames so that they may be processed using the *ISampleGrab* filter. We stumbled upon a video capturing appwizard for Visual C++ 6 developed by [4] which contained the start of a graph similar to that above. Its video capturing code was a separate to the GUI unlike the SDK examples. We therefore used this as the basis of our *DxCapture* class which was then used to create a document object which inherits its capture functions to take in video input. With video input available the next challenge was how to get individual frames so that they can be processed for hand detection. The method to achieve this is to add a filter in between the capture device and the video renderer which then processes the frame through a callback function. Our original attempt was to add the *ISampleGrab* filter which passes a pointer to the frame in memory to the callback function *process()*. To test it we began to work on stub class which just thresholds the incoming frame using OpenCV functions. This is the basis for our *HandMouseDetect* class. Unfortunately this never worked properly and caused the program to crash rather spectacularly. Further reading of the OpenCV message board made us realize that this would not work as the filter is not OpenCV friendly. Luckily an alternative frame grabber filter specifically for OpenCV came as part of the SDK called *IProxyTrans*.

3.3 Frame Processing

The *IProxyTrans* filter works by intercepting frames from an incoming video stream and places them in a

buffer. It then passes a pointer to the current frame through a callback function called *process()* which we've place in the source file *DxVideoCap.cpp*. The next challenge was how to use this pointer to a location in memory into something which could be interpreted as an image file. OpenCV's image format is the *IplImage* structure which is defined in the OpenCV documentation. Standard pointer conversion by doing something like:

```
void *frame;
IplImage *image = (IplImage*)frame;
```

did not work. Debugging the program proved little help as it was not doing anything particularly incorrect. We therefore went back online to figure this out and found an answer at [8]. This page explained that the reason for the problem is in the way C converts pointers. As all pointers are addresses and all addresses generally have the same number of bytes, conversion does not convert any bits. Therefore if the meaning of these bits is different between types, the new pointer would be garbage. C++ fortunately has a call which solves this problem known as *reinterpret_cast*. When this is called, the runtime system cunningly looks at both pointer types and rearranges the bit order according to the conventions of the destination pointer type. We were therefore now ready to start processing frames. A review of the OpenCV message board showed that there are many approaches available for detecting an object within a scene. Its preferred approach was in the use of a sliding window through a frame to find positively classified haar-like components within the window. The thinking behind this is explained in section 2.1. We were convinced this was a good approach having looked at the results of the face detection example that came with the SDK. This code is able to identify faces in all manners of lighting, occlusions and distances from the camera. To get a system like this working we would need to train a cascade classifier useable by the pattern recognition code in OpenCV.

3.3.1 Cascade Training

Training a classifier required us to feed a large set of images through the *haartraining* application which comes with the OpenCV SDK. The training process was also well documented in the OpenCV message board. The first step was to prepare a set of hand images with different reflections, illuminations, backgrounds, scales and rotations. About 10000 such images would be required to build a classifier which is both accurate and robust enough for this purpose. This collection was obtained through a number of means. Primarily we captured webcam footage of our own hand in various positions and in different settings such as in front a solid color (our bed sheets) and more complex backgrounds (by our desk). The next step was to decompile the captured video footage into individual frames. In addition to this we also used image grabbing software to scour the internet for any image matching the keywords relating to hands and gestures. The haartraining application takes in images through a text file listing which in addition to the file path also requires a bounding box for where in hand appears in the image to be defined. Finding the pixel coordinates for this box is a long drawn out affair and so a means of expediting this process was required. After reviewing the documentation on the OpenCV message board we managed to find a small application written by [1] called *objectmarker* which uses OpenCV's image processing code to mark rectangles within an input image and output the rectangles coordinates in a format compatible with the classifier training program. At this time we were also collecting a large set of negative image examples from different sources such as clipart libraries and webcam footage. These images then needed to be resized and convert to the same format as the positive samples. To achieve this quickly we wrote a batch script using *Excel* to run *ImageMagick's* application for command-line image manipulation. With a large number of bitmap images on our hard disk we needed to figure a means to put them in order of negative and positive set and then as to what to of gesture the positive file contained. We therefore altered the source for *objectmarker* for this purpose by pressing adding different key combinations which would then write an entry for this file in the relevant file listing text file. Figure 3.3 shows a screen shot of this process.

Figure 3.3 The image classification process

From these image files we could then use *createsamples* to grab the hand images and then reduce each sample to a uniform size of 24 by 24 pixels. This program would then place all the samples into a single *vec* file which could then be passed to *haartraining*. The overall training process is illustrated in figure 3.4.

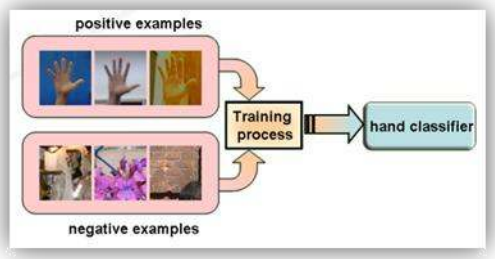


Figure 3.4 The classifier training process

Training could now begin by calling haartraining with the new file listing as input. It takes a very long time to train a classifier and during training it is possible to get an estimate of how good the classifier will probably be as it prints out information on the hit and false alarm rates at each stage. It is also possible to cancel the process and return to it from the last complete stage, something which has been useful given repeated crashes of our system. The only cascade we successfully created was 14 stages long and provided far too many false alarms to be useful. For safe measure we therefore added more positive training images, this time by taking high resolution pictures with a digital camera which where then resized and used to generate more images by feeding random convert settings to *ImageMagick's convert* program. This was done by means of a batch script which was produced again with *Excel*.

3.3.2 Object Detection

While classifier training was going on, we decided to press on with coding and wrote the object detection class. This contains all the code needed to read the classifier file, initialize the variables used in detection, call OpenCV's *cvHaarDetectObjects()* function, and handling of its results. This was fairly painless as there was plenty of documentation on how to do this in the OpenCV API. We however needed to take into consideration how our detection class was going to deal with multiple classifiers as well as how to make the detection process as efficient as possible. For a program with the ability of detecting four gestures it would not be wise to call *cvHaarDetectObject()* four times every frame as it slows down program the program as well as uses more resources. We therefore decided to write a scheduler algorithm which would call this function for each cascade in a round-robin fashion. A further performance improvement was also done by calling this function only once per an interval of frames.

This algorithm is outlined in figure 3.5.

```
// Initialisation
unsigned char tick = 0;
unsigned char pollrate = 16;
int POLL_INTERVAL = 256/NUMBER_OF_GESTURES;

1. When process() is called
   if tick%POLL_INTERVAL = 0
     a. run detection code
     b. if tick = 0
     c. draw a rectangle around any detected object
2. tick = tick - pollrate
```

Figure 3.5 Detection Scheduling Algorithm

The advantage of this algorithm was mainly in its scalability. The rate at which the detection code is called as well as the number of gestures can be altered by changing just two variables. The output from *cvHaarDetectObject()* is a vector of rectangles which state where in the frame a detected object is, if any. We therefore now had a means of identifying an object in a frame, as well as its location, size and gesture type. We chose to prioritize gestures in case more than one is detected per frame. Two gestures were chosen, each one signifying a different action to be taken and are prioritized in the following order. An open palm signifies right-click event. A fist gesture signifies left-click event.

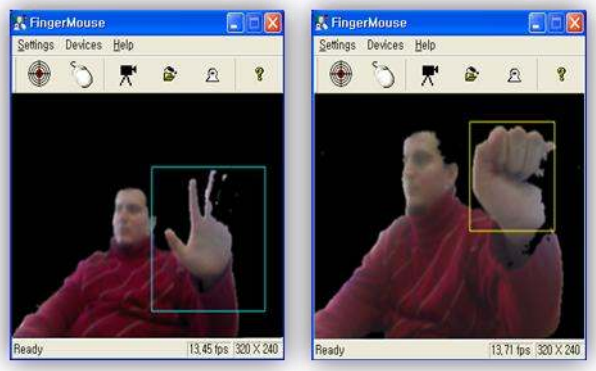


Figure 3.6 Palm Gesture for right-click action and Fist Gesture for left-click action

3.3.3 Motion Detection

While still waiting for a classifier to finish training, we used the cascades which detect faces which came with the OpenCV SDK as a means of testing the detection code. We also used it to see how well it worked at sending movement code to the cursor. This

will be covered in the next section. Unfortunately feeding mouse updated positions from the object detection output caused the cursor to move erratically around the screen. That in addition to the possibility of not having a classifier reading in time for the project presentation meant that a major rethink was in order. We went back online to look for an answer and funnily enough found one in a recently published paper co-written by our former tutor [3]. In it they used a similar approach to detect lions' faces and used a Lucas-Kanade feature tracker to detect movement. Besides an implementation of the iterative Lucas-Kanade tracker covered in section 2.2.2 the API also referred to an implemented which uses multiple image resolutions to enhance accuracy and allow for detection of larger inter-frame motion. Using example code from the API as a basis, we wrote the *LKDetect()* function to detect optical flow within an input frame. This function would take in a frame, convert it to a grayscale image and either generates a series of trackable points for detection or to calculate optical flow of these points in subsequent frames. These feature points are determined in the example code using the function *cvGoodFeaturesToTrack()* by finding pixel areas within an input image which have strong corners. This is determined by the eigenvalue of the area around each pixel and selects a point according to whether this value is above an input threshold. Given that we had a rectangular area where the detect object lies, we decided to pass this information to *LKDetect()* and restrict point generation to within this rectangle. This meant that only points within a detect object would be used to determine optical flow between frames. The code for this was simply calling the *cvCalcOpticalFlowPyrLK()* function which took the array of feature points and returned their updated location in the subsequent frame. From these two arrays we could then determine optical flow over the area of the detected object by calculating the average displacement vector of all the points. Feeding the mouse this average displacement seemed to work well only for short intervals as the points would eventually no longer represent a feature on the detected object. To resolve this we altered the calling of *LKDetect()* so that it would reinitialize the points array at a given interval in line with our scheduling code. This was a significant improvement as feature points were kept within the object bounding rectangle thereby ensuring that only movement form that object would contribute to the point displacement vector value.

3.3.4 Motion Feature Segmentation

With the possibility that we may not have a working hand classifier ready we needed to figure out a means of using only the motion detection code to move the cursor. We also did not want to give up on weeks of work and come up with something that could easily be reconfigured should the classifier ready in time. The answer came in isolating pixel values representing skin within a frame and setting other pixels to zero. Small pixel areas not corresponding to skin would then be cleaned out by applying a median filter over the frame. Only pixels not set to zero could then be used as a feature point. This left me with the problem of what to do with the area representing the head. As it also had skin pixel values it could also be used to move the mouse. This was not desirable. We therefore rewrote the code to generate feature points solely from pixels not within the bounding rectangle of the face. Therefore this object detection code was used to identifying only the intensity values that represent skin and also identifies the face so that it could be disregarded as a source of features. This roundabout way of doing things is not desirable as it would later need to be rewritten to accommodate hand gestures. To make life a lot easier we therefore simplified the code by using the face detection classifier for acquisition of skin pixel areas and restricted feature selection to that bottom half of the frame. This is based on the thinking that if a webcam is placed on top of the monitor with the user sitting in front, hands would generally be in the bottom half of the frame whilst the head would mostly be in the top half. Reducing the rate at which the feature points are reset would then allow for hand movements in the top half of the frame so long as they are only temporary. Though not an ideal solution, in practice it has worked quite well.

3.3.5 Blob Extraction

As we understood the fact that it is impossible to distinguish face and hands separately by previous methods, we decided to think in different manner. When we were playing with the blobs of moving hand, we noticed that moving hand has the largest blob among all the other blobs being extracted by OpenCV blob extraction function. Then, we analyzed all the blobs and developed and algorithm for detecting overall hand blob.

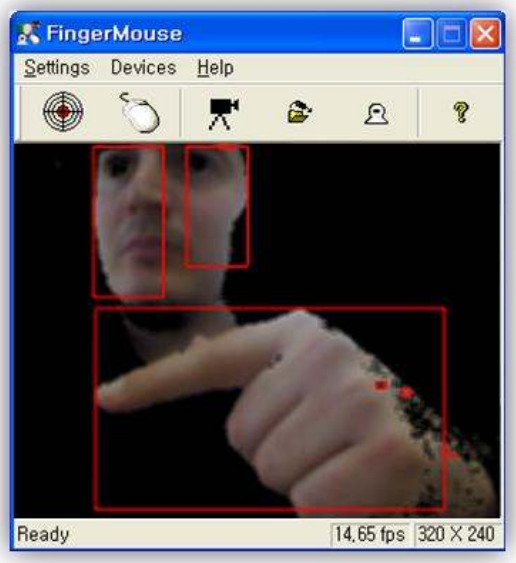
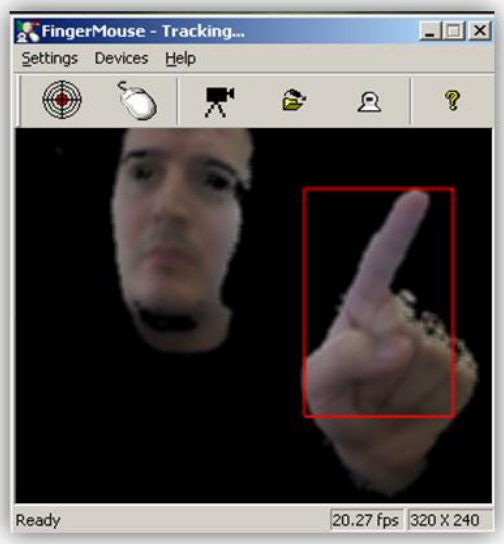


Figure 3.7 HandMouse screen shot with all the blobs: Hand blob is the largest blob.

Figure 3.7 shows how hand blob is the largest blob among all the other blobs. In our algorithm, we first find the largest blob in all the other blobs, and then as a final whole hand blob we assign the minimum x-position and y-position values (up-left corner of the blob) to hand blob with its width and height values. Finally, we had the following results:

Because skin color segmented image had some noise, we applied high pass filter and extracted hand blobs more clearly.



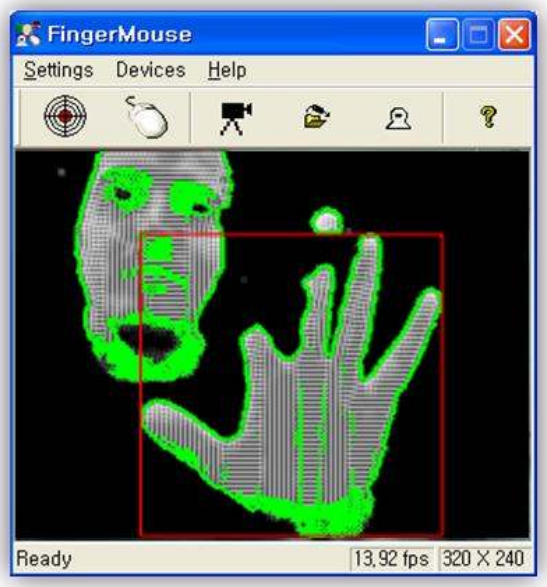


Figure 3.8 HandMouse screen shots with hand blobs only

3.3.6 Blob Motion Detection

After hand blob was detected, then we had to detect the blob motion precisely. However, the deal was cursor control and graffiti input by mouse cursor. For this reason, boundary motion detection had to be developed. We didn't use the simple motion detection algorithm, but rather we developed our new motion detection algorithm which detects the boundary region motion as follows

This algorithm evaluates all the possible neighbour pixels has changed or not with respect to a given threshold value rather than evaluating all the pixels. Therefore, especially for index finger detection, this algorithm really works well.

After detecting motion in hand blob, pixels changed above the threshold has been transferred to the empty PImage so that it will be able to be processed by Lucas Kanade Pyramidal Optical Flow Tracker. Then, we noticed that we had so good results showing that we can use our hands to point the cursor as well as writing texts by means of graffiti gestures and we can use these gestures to manipulate the computer actions like keyboard inputs.

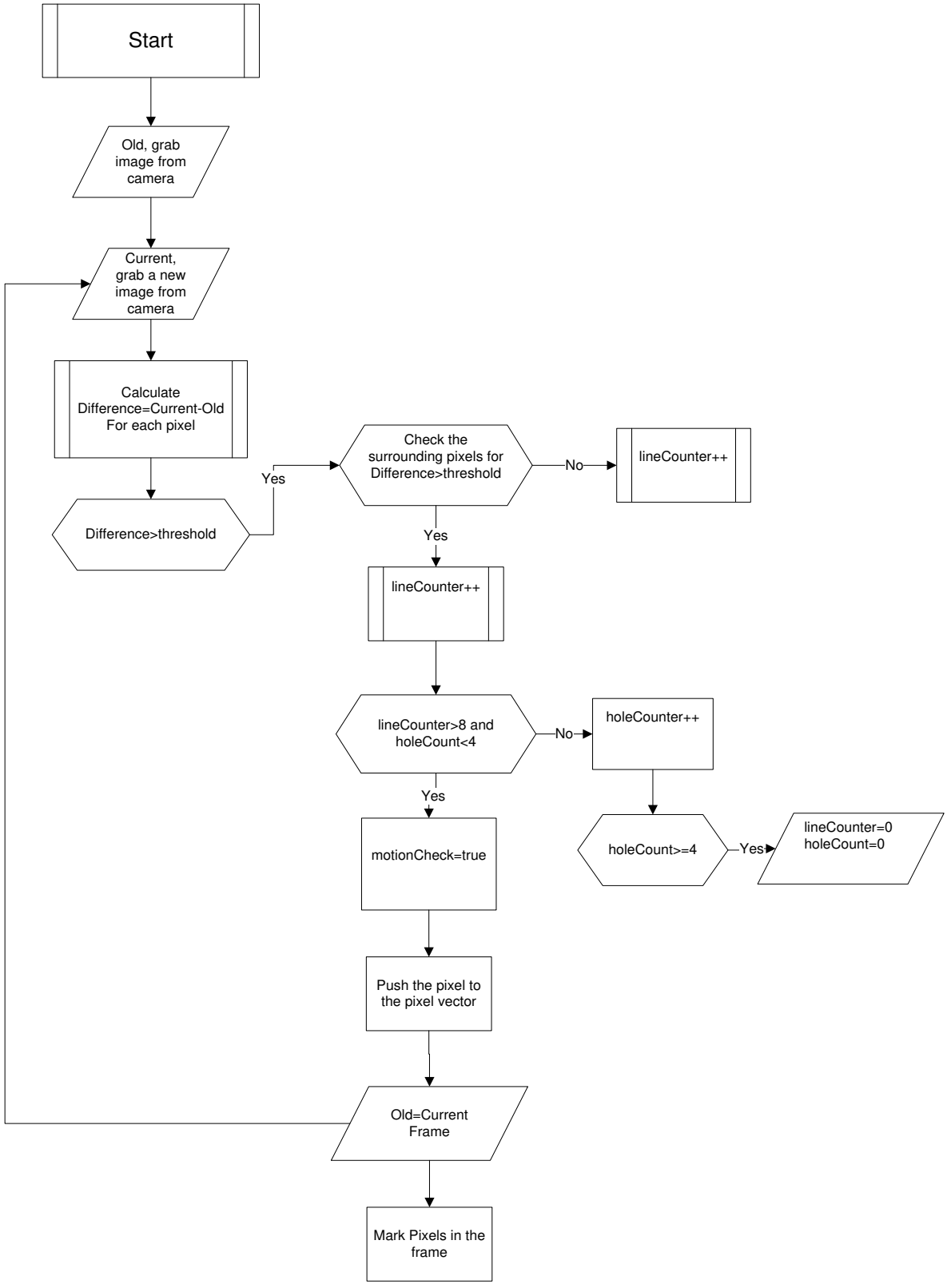


Figure 3.9 This is our implementation of motion detection Algorithm for moving object tracking.

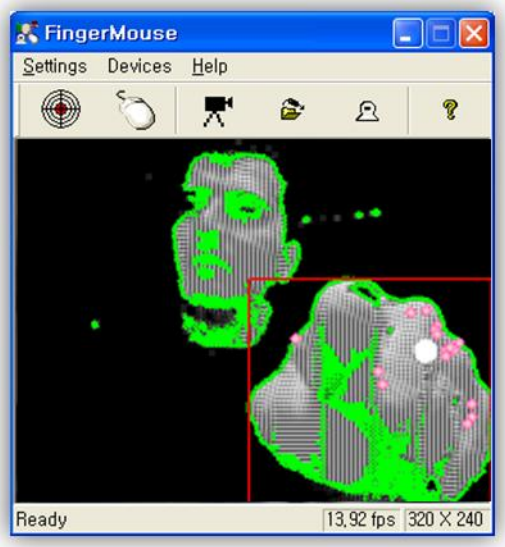


Figure 3.10 Lucas Kanade Optical Flowing Algorithm is tracking the feature points in hand blob. Pink circles are the feature points in motion detection and white one shows the average x-position and y-position of the feature points.

By this method, we get finger mouse that manipulates the position of Windows cursor as shown above.

3.4 Cursor Control

Our initial approach at sending control commands for the mouse was by sending out the windows messages *WM_MOUSEMOVE* and *WM_LBUTTONDBLKCLK* and broadcasting it using the *PostMessage()* function available in the GUI view object. By encoding a message with updating cursor point information and button status the plan was to send out the message and that the operating system would pick it up. Cursor positions were available through the *GetMousePos()* function in the Windows API. However this thinking was flawed as certain applications did not react to these messages. There was also a problem with the mouse not responding should the HandMouse application crash. We therefore abandoned this method and went back to the drawing board. The next approach was to use directinput to control the mouse. This was however a non-starter as directinput does not have any means of sending coordinate information to the mouse except for code which is intended for force-feedback joysticks. Additional navigation around the MSDN site showed that there was functionality for this in code originally intended for the remote desktop feature in Windows XP. The *SendInput()* function allows us to generate both mouse and keyboard movements and button presses at the OS level. With this we were now able to feed the average displacement value from the detection code to *SendInput()* which meant that the cursor could be controlled by hand movements. With no clicking gesture available we then had to figure how best to create a clicking function as the program would be useless without it. The solution is to send a clicking command if the mouse has not moved for a certain period of time. This meant that the mouse controlling code in *process()* needed to remember both mouse positions as well as have a means of timing how long a cursor has been at that point on the screen. Remembering mouse positions was simply a matter of an addition point variable. The click determining code just required a DFA approach with different values of *ClickStatus* being set according to at point the program was in setting up a timer to measure the time no change in cursor position has occurred.

1. Get current cursor position
2. Detect motion and object presence in incoming frame
3. Set cursor position from detected motion displacement x axis sensitivity

   If mouse clicking enabled (*ClickStatus*>0)
   a. *ClickStatus* == 1 (clicking on and in initial state)
      - ✓ Remember current position
   b. *ClickStatus* = 2
   c. *ClickStatus* == 2 (set timer to begin counting interval)
   d. *ClickStatus* == 3 (timer has been set and is counting away)
   e. If cursor position has changed
      - ✓ *ClickStatus* = 1
   f. *ClickStatus* == 4 (timer has reached interval)
      - ✓ Send double click event using *SendInput()*
   g. *ClickStatus* = 1

Figure 3.11 Mouse control algorithm

The timer is initiated at startup with a WM_TIMER message being sent whenever the program has completed a desired time interval from when it first started. This continues to run throughout the program's run-time and can be changed by destroying and

recreating the timer in the GUI's view object. Figure 3.11 outlines the final mouse control algorithm.

4. Design and Implementation

This section outlines our reasons for the choice of tools and design methods that went into this project. This work is very much a 'proof-of concept' to demonstrate the effectiveness of computer vision techniques as a new form of HCI.

4.1 Languages

C++ was chosen for many reasons. It is fast, intuitive and the compiler and debugging tools very good. We also wanted to take advantage of object orientated design as we believe it necessary to have a modularized design framework for later development. Eventually we would like to port to Linux and also get rid of OpenCV altogether. This would mean that the code relating to Windows API calls would be best compartmentalized from the detection code and vice versa. Given OpenCV is written in C, were were therefore able to use the code without any fuss. We also were able to resolve our problems with casting as mentioned in section 3.3 thanks to the *reinterpret_cast* call.

4.2 Development Tools

We've used a range of different tools throughout this project. Below are some of the most relevant to the development process.

4.2.1 Visual Studio .NET 2005

The Microsoft Development environment is always a must when taking into account large scale development as this project required. Firstly there's its ability to manage multiple projects at one go. This meant that all of the code used in this project, including the OpenCV library source and other assorted tools could be viewed in one single window. There multiple build option allowed us to have two different branches of code, one with all the debugging options on for debugging purposes and another release build which has been optimized for top performance. The C++ compiler that comes with the package is optimized for use in Windows which was a definite bonus. It also retained pre-linked object code between compilations which speeded up the development process. The inbuilt debugger which has all the same features as GNU's *Insight* proved invaluable when the program was not working and we had no idea why. Through the resource manager we were able to quickly prototype the GUI look and feel without getting too bogged down on GUI coding. Our only criticism of this package is its tendency to add a lot of unnecessary libraries as default, some of which oddly conflicted with one another through redefined functions. This was a bit of a headache in the earlier stages of development but problems were ironed out through subsequent changes in build configuration as the program got bigger.

4.2.2 CVS

Our chosen version control system was CVSNT, which is windows port GNU's *cvs* program though built with a GUI to control the settings from within Windows control panel. This server was set up on the local computer on a mirrored partition. Though it would have been preferable to have kept the repository on a different machine, computing power was not a luxury we had readily available and so opted for a non-networked option which had worked well. To communicate with the server we used a plug-in for *Visual Studio* called *CVS SCC Proxy* by PushOK. This came about from the fact that *WinCVS* keep crashing on start up and we decided not to spend too much time figuring out why. This paid off really well as all checking in happened seamlessly within the development environment allowing more frequent changes between versions which proved very useful.

4.2.3 Spy++

This tool is used to monitor messages being broadcast by all the processes currently active. As we were having issues with our original mouse control implementation, we needed to know what messages were being sent out and see why mouse control was behaving so erratically. Another really useful feature was its detailed memory usage information. This helped us to keep resource quantify how much in the way of resources the application was using up and eventually led to the fast and low computation expense of the current implementation.

4.2.4 Picture Collection

As we needed to modify a large number of images for classification we used a number of tools to help us obtain and manipulate them in many ways. Firstly there's *Picture Ace* which allows us to scour a web site for images recursively based on given criteria. This was useful in finding negative samples for classification and was used by simply entering random search phrases into Google and saving all found

images that were of sufficient resolution to be useful. For capturing video footage we used *VirtualDub* by Avery Lee. This program allowed us to capture multiple video input and at the same time compress the footage into divx codec format so as to save on disk space usage. We also used Bulletproofsoft's *BPS Video Converter and Decompiler* program to extract frames from within the recently captured footage. This program allowed us to sage frames as individual bitmap files and was the main means of generating positive training samples. More importantly was the use of the *ImageMagick* image manipulation suite. Using its command-line tools we were quickly able to generate batch scripts which created new samples by applying random resizing, rotating and colorizing features. Finally, for any image requiring more complex manipulation, we used GNU's *Gimp*.

4.3 Hardware

A main reason for why we were unable to better develop the classifier was because of our choice of hardware. We carried out all our work on a single Athlon XP 32bit 1.6GHz system, with 1028MB DDR3200 ram, 270Gb hard disk, and Windows XP SP2. Given this specification we had originally assumed it more than enough for the training process. Subsequent experience has unfortunately shown that the classifier training process requires a much higher spec system so as to complete training quicker and hence allow for more experimentation with training configuration to be carried out. At present a stage takes about 17 hours to complete. The ideal system as recommended by [1] is a dual 64-bit Intel P4 system with hyper-threading, 1.5 Gb RAM and running a 64-bit capable OS with robust memory management capabilities. Haar training could be recompiled to take advantage of multiple processors and would make for a much faster classification process. Testing was carried out using three different models of Logitech USB camera of increasing image quality. All three were used to test how well the program worked with standard web cameras. We also played with different resolutions and frame rates and found optimal detection at 160x120 with a frame rate of 30fps.

4.4 Libraries

As we all had limited development time it was necessary to look into the use of off the shelf libraries to expedite the development process. The two main libraries which are used are OpenCV and DirectX. OpenCV has a large collection of image processing and computer vision features as well as excellent matrix manipulation functionality. It is also well documented and widely used which meant it was easy to find an answer to problems arising from developing with it. Use of this library is subject to a BSD licensing agreement. This means that it is perfectly okay to use the library for commercial purposes so long as we make add a specific acknowledgement within the program to Intel. This is owing to them being the founders of the library. The main drawbacks however to using OpenCV is that it is still beta stage and there are still a few bugs with it. A key example was that haartraining was unable to convert a trained cascade from a directory structure from an older version to the current xml file cascade format. This was resolved by downloading the latest haardetection source from the library's CVS server and then to recompile the whole library. As previously mentioned it also still uses video for windows which is an obsolete technology with limited functionality. The use of the DirectX library for video capturing is essential for ensuring future development of the application. In addition to this it is also remarkably versatile as can be seen in the filter structure. It is also faster than VFW and helped to significantly improve detection performance in comparison to OpenCV's face detection example.

5. Current Status and Future Plans

By the time of writing this paper, the current status of this project is that it is capable of detecting motion from skin pixels and that from this it can efficiently control the mouse. As mentioned we have been unable to present a classifier at this time despite a lot of effort in trying to do so. The detection is nonetheless fully working and requires little modification to work with hand gestures once suitable classifiers become available. In the meantime clicking is possible by means of keeping the cursor at a required point for a specified time. This could be altered in the detection settings dialog window. Easy access to all the camera settings has been provided making it fairly configuring. The code also suffers from problems detecting skin correctly. If we wear a reddish shirt, some of the pixels representing the shirt would be wrongly interpreted as skin which makes the cursor harder to operate. Optimal performance at present involves wearing contrasting clothes to skin color, using a web camera with a high frame rate. The tested has been at 30fps as mentioned in section 4.3 though we would like to see how well it works at higher resolutions using a firewire camera. Functionality for this is possible thanks to the use of DirectX for video capture. Another issue is that the program requires exclusive access to the video camera which makes it incompatible in combination with web-conference software or anything that needs the camera.

We have not had the chance to look into this in any detail but would assume it can be resolved through altering filter graph by adding a multiplexing filter. This would need to be look into.

5.1 Cascade Development

With an image library of tens of thousands of images collected a future plan would be to train more classifiers and add more functionality. To make the program customizable a library of gesture cascades would need to be available and could eventually sold according to function. For example a cascade library for different sign languages could be developed and imported into the program.

Another thing to consider is to separate cascades according to left and right hand orientations. This would allow for a much richer gesture vocabulary though it would also mean doubling the amount of gestures needed to be detected. The OpenCV object detection code would need to be worked on so as to improve performance by calling a single instance of the detector which looks at multiple cascades rather than the other way round. Alternatively the trainer could be rewritten to create a tree like structure, each branch being a cascade for a specific gesture.

5.2 Multiple Cursors

One of our preferred additions to the current implementation would be the ability to arbitrarily add more cursors according to the number of hands detected. This would mean a person could use both hands to control the computer. It would also allow for more than one person to control the computer opening up new possibilities with respect to the games market. Windows however current only supports a single cursor. This is expected to be resolved in the next version of the OS. A solution in the meantime would be to add a component at the device driver level which would act as an alternative input device. A framework for this and even code to add to the system is referred to in [14].

5.3 Voice Recognition

A hand gesture HCI in addition to voice recognition technology would be an ideal combination as it would provided a complete alternative to both the keyboard and the mouse. This would be possible by adding an off the shelve voice recognition library, of which there are many to choose from, and insert detection code to be called in around the same location as the existing hand detection code. More research would be needed into seeing how this would affect program performance.

5.4 3D Hand Detection

With the existing single camera implementation, functions such as scaling of windows are possible and would be using the difference in detected object size to achieve this. For more complex hand functions such as pointing to a wall as a command to project a desktop, some form of 3D hand detection be needed. Solutions such as stereoscopic systems would be the obvious answer. OpenCV provides sufficient functionality to develop this idea further. Looking at this would however mean the beginning of looking into specific hardware for the program. This is something which we would not be tempted to do as our business plan is based on the idea that HandMouse is a solely software solution.

5.5 Business Potential

Our business proposal is to aim to have a future version of the current code that could be downloaded by anyone with a web cam and used with little configuration. We would be taking a shareware approach by distributing a cut down version that can be downloaded for free. The more complete version with suggestion functionality as mentioned in previous sections would be sold through the company home page, which avoids retailing issues. Once a brand has been established, the next phase would be to arrange OEM licensing agreements with camera manufacturers who would then bundle HandMouse with their product. If this goes well, such licensing agreements could be extended to other consumer electronics such as televisions.

6. Conclusion

In this paper, we proposed a motion detection algorithm based on blob analysis of finger blobs through the skin color extracted image. Experimental results show that human hand can be detected by using OpenCV Blob Extraction algorithm after performing some analysis on finger blobs even if hands and human head are available in the same frame. This paper also proposes a hand mouse system based on Lucas-Kanade Pyramidal Optical Flow algorithm applied on the hand motion pixels. We have introduced two human hand gestures by training OpenCV Haar cascade AdaBoost Classifiers as palm and fist gestures for right and left click of the mouse action. Experimental results indicate that mouse usability is almost done with proposed

gestures but not so robust. Further studies can be done and proposed algorithms can be developed in the future.

7. References

[1] Florian Adolf, "OpenCV object detection framework HOWTO", http://robotik.inflomatik.info/

[2] J. Bouguet, "Pyramidal Implementation of the Lucas-Kanade Feature Tracker, Description of the Algorithm", In OpenCV Documentation, Intel Corporation, Microprocessor Labs, (2000).

[3] T. Burghardt, J. Ćalić, and B. Thomas, "Tracking animals in wildlife videos using face detection", In European Workshop on the Integration of Knowledge, Semantics and Digital Media Technology, October 2004.

[4] Yunqiang Chen homepage, http://www.ifp.uiuc.edu/~chenyq/

[5] Paul DiLascia, "System Tray Balloon Tips and Freeing Resources Quickly in .NET", http://msdn.microsoft.com/msdnmag/issues/02/11/CQA/default.aspx

[6] Y. Freund and R. E. Schapire. "Experiments with a new boosting algorithm", In Machine Learning: Proceedings of the Thirteenth International Conference, Morgan Kaufman, San Francisco, pp. 148-156, (1996).

[7] C. Evans-Pughe. "Triple Vision Vision Vision", In The IEE Review Volume 50, Number 4, IEE Publishing, pp. 40-43, (2004).

[8] Robert Laganière, "A step-by-step guide to the use of the Intel OpenCV library and the Microsoft DirectShow technology" http://www.site.uottawa.ca/~laganier/tutorial/opencv+directshow/cvision.htm

[9] R. Lienhart and J. Maydt. "An extended set of Haar-like Features for Rapid Object Detection", In Proc. of the IEEE Conference on Image Processing (ICIP'02), Vol.1, pp. I.900- I.903, (2002).

[10] B. Lucas and T. Kanade. "An Iterative Image Registration Technique with an Application to Stereo Vision", In Proc. of 7th International Joint Conference on Artificial Intelligence (IJCAI), pp. 674-679, (1981).

[11] Microsoft Developer Network MFC Library Reference, http://msdn.microsoft.com/library/default.asp?url=/library/enus/vclib/html/_mfc_Class_Library_Reference_Introduction.asp

[12] Open Computer Vision Library, http://sourceforge.net/projects/opencvlibrary/

[13] P. Viola and M. Jones. "Robust Real-time Object Detection", In Second International Workshop on Statistical and Computational Theories of Vision, Vancouver, USA, (2001).

[14] M. Westergaard, "Supporting Multiple Pointing Devices in Microsoft Windows", In Proceedings of Microsoft Summer Workshop for Faculty and PhDs, Cambridge, England, (2002)

[15] EFNet #Winprog help site, http://www.winprog.org/

[16] Horn, B. K. P Robot Vision. MIT Press. pp. 69–71. ISBN 0-262-08159-8. (1986)

[17] Ince, I.F and Yang, T.C: "A New Low-Cost Eye Tracking and Blink Detection Approach: Extracting Eye Features with Blob Extraction", LNCS, Lecture Notes in Computer Science, Volume: 5754, pp. 526-533 (2009).