A Camera Based Virtual Keyboard with Touch Detection by Shadow Analysis

Joseph Thomas

December 10, 2013

1 Introduction

Recent years have marked a sharp increase in the number of ways in which people interact with computers. Where the keyboard and mouse were once the primary interfaces for controlling a computer, users now utilize touchscreens, infrared cameras (like Microsoft's Kinect), and accelerometers (for example, within the iPhone) to interact with their technology. In light of these changes and the proliferation of small cameras in many phones and tablets, human-computer interface researchers have investigated the possibility of implementing a keyboard-style interface using a camera as a substitute for actual keyboard hardware.

Broadly speaking, these researchers envision the following scenario: A camera observes the user's hands, which rest on a flat surface. The camera may observe the hands from above the surface, or at an angle.[1] The virtual keyboard's software analyzes those images in real-time to determine the sequence of keystrokes chosen by the user. These researchers envision several applications for this technology:

- In some countries (for example, India), users speak many different languages, which makes producing physical keyboards for many different orthographies expensive. A camera-based keyboard can easily support many languages, as is discussed in [1].
- Smart-phone and tablet users may occasionally want to use a full-sized keyboard with their device, but are unwilling to carry a physical keyboard. Since most mobile devices are equipped with a camera, a camera-based keyboard could provide a software-based solution for this problem.

The objective of this semester project was to implement a virtual keyboard using the image analysis techniques described in [1], [2], and [3]. In the system we implemented, a single low-quality camera captures RGB images of a user's

[1] One might envision that the camera sits at one end of a phone/tablet, which has been propped up in front of the user.

hands, which touch a patterned surface, or *keyboard-mat*, in order to select keystrokes. Based upon this information and an initial calibration image of the keyboard-mat (Figure 3), the system produces a discrete sequence of keystrokes that can be used to control other software. We examine the performance of each of the three main phases of image analysis and compare the efficacy of two techniques for locating the user's fingertips in an image. We also analyze the degree to which our system is sensitive to changes in lighting and frame rate.[2]

2 Phases of Touch Detection, and their Implementation

In this section, we describe the image analysis techniques used to convert the user's surface-touches into keystrokes. Most state-of-the-art camera-based virtual keyboard schemes, including [1], [2], and [3], are implemented using the following sequence of image analysis techniques:

(a) The system's camera captures an image I for processing. By performing skin segmentation on I, we obtain a binary image with a region H representing the hand(s) in the scene.

(b) H is analyzed to determine the locations of the user's fingertips. The references listed above determine contours that parameterize the boundary of H. They associate fingertips with the points on the contours that optimize certain geometric quantities (for example, curvature). We consider two of these geometric approaches to finding the fingertips and compare their performance.

(c) Given the estimated fingertip positions, one must calculate which fingertips are touching the keyboard's keys. [1] and [2] propose a technique called *shadow analysis* to solve this problem.

(d) To map touch points to key presses, we assume the keys of the virtual keyboard are described as rectangles in $\mathbb{R}^2$. We assume the layout of the keyboard is known at compile time and the keyboard-mat has control points from which we can derive a perspective correction transformation. We then apply a simple formula to convert the keyboard-space coordinates of the touch points to keypresses.

Figure 1 provides an example of phases (a)-(c).

2.1 Determining the Hand Region and its Boundary Contours

Our first step in processing an image I is to locate contours that correspond to the boundary of the hand(s) in the scene. We locate the user's hands using a

[2]In lecture, it was mentioned that this report should discuss what the author learned in the course of his project. This appears in Section 4.

simple *skin segmentation* procedure described in [4].

To apply the segmentation procedure, the image must be represented in YCrCb color coordinates. A pixel with color coordinates (Y, Cr, Cb) is determined to be a skin pixel if:

$$133 \leq Cr \leq 173 \text{ and } 77 \leq Cb \leq 127$$

In [4], this thresholding procedure is presented as part of a larger face-recognition scheme. They report that thresholding can be used successfully on its own, provided the background color is sufficiently different from the user's skin color and the lighting conditions are reasonable. This is the situation in our project, where the background is a blue keyboard-mat. The authors of [4] report that their thresholding procedure is effective on a variety of skin colors, in part due to their use of the YCbCr colorspace.

We observed that these thresholds can sometimes fail to record fingernail pixels as part of the hand, but this can be overcome by relaxing the inequalities to:

$$128 \leq Cr \leq 178 \text{ and } 72 \leq Cb \leq 132$$

Thresholding often produces a binary image containing small artifacts. We remove these using morphological opening, which produces an image I' with a small number of 4-connected components. We calculate the connected components of I', and delete those with fewer than 500 pixels (the regions corresponding to the user's hands contains far more pixels).

For each remaining connected component C of I', we calculate a contour (a list of pixel coordinates) that parameterize the *boundary* of C. A pixel in C is a *boundary pixel* if any of its 4-connected neighbors is not in C, and the boundary of C is simply the union of the boundary pixels of C. Since we wish only to detect the boundaries of hands, there is no loss in assuming each connected component is simply connected (i.e. there are no "holes" within C). The algorithm for computing the boundary contour of C is relatively simple, so we will only outline it: Initialize a pointer to point to a pixel in C and then move the pointer in one of the coordinate directions until a boundary pixel p_0 is found. Record this boundary pixel's location, and follow the boundary counterclockwise (recording the locations of the pixels visited) until the pointer returns to p_0.[3]

There is another approach for determining the boundary of a region, based on morphological operations. Given a connected component of C, we could find the boundary by eroding C with a small square structuring element and then subtracting this eroded image from C. However, later processing steps require information concerning which pixels make up the boundary *and in what order* they are encountered as the boundary is traversed counterclockwise. Thus, regardless of whether we apply the described morphological operations, we are obliged to iterate over the pixels in the boundary of C and record them in a list.

[3]Remark: This is the same algorithm used to solve simply connected mazes.

Also, the morphological operations we described are not guaranteed to produce a contiguous one-pixel wide boundary, so they would not substantially simplify our boundary-traversal procedure. For these reasons we decided to omit the morphological operations entirely.

2.2 Fingertip Detection

In this step, we estimate the locations of the user's fingertips (in image-space) based on geometrical features of the contours and regions obtained from the last step.

We represent a contour $\tilde{\gamma}$ as a sequence of positions $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots\}$ in image-space. Given a contour, one can derive several other sequences of geometrical significance and use these to locate the fingertips.

The previous processing step gives us a contour $\tilde{\gamma}$ consisting of pixel locations p_i such that the Euclidean distance between p_i and p_{i+1} is 1 for all i, and the angle θ_i between the displacement vectors $p_{i+1} - p_i$ and $p_{i-1} - p_i$ is in $\{-\pi/2, 0, \pi/2, \pi\}$. This angular information does not give a very good idea of how the contour bends around the hand, so we consider a subsequence γ in which the points of the contour are spaced further apart. In experiments, we took γ to be every tenth point of $\tilde{\gamma}$. We then define a sequence of angles θ_i (in $(-\pi, \pi]$) from γ as we did with $\tilde{\gamma}$, by considering the angle between displacement vectors.

A second important property we derive from γ is *curvature*. Curvature can most easily be understood in the situation where γ is parameterized by arclength. In this situation, the acceleration vector $\gamma''(t)$ is always perpendicular to the contour. The curvature $K(t)$ is simply the magnitude of $\gamma''(t)$, with larger values indicating that the curve is bending more severely. The data we use to represent γ does not parameterize the contour by arclength, but the authors of [3] report there is still a formula for calculating $K(t)$

$$K(t) = \frac{\det(\gamma'(t), \gamma''(t))^2}{\|\gamma'(t)\|^3}$$

which they discretize, using the data $\gamma = \{p_i\}$, to obtain:

$$\begin{aligned} v_i &= p_{i+1} - p_{i-1} \\ a_i &= v_{i+1} - v_{i-1} \\ K_i &= \frac{\det(v_i, a_i)^2}{\|v_i\|^3} \end{aligned}$$

Given this geometrical data, we have two approaches for locating the fingertips in an image. The first approach is to calculate the sequence of angles θ_i, and associate fingertips with points where θ_i is maximized. The second approach is to calculate the curvature values K_i and associate fingertips with positions where K_i is maximized. Naturally, both approaches require non-maximum suppression or other post-processing to be effective. Both of these approaches are

described in [3] and we compare their performance. Figure 2 provides an example of both detection schemes.

2.3 Touch Detection

In this phase of processing, we are given as input the estimated positions of the user's fingertips and must output which of those tips are estimated to be in contact with the keyboard-mat. We used a technique called *shadow analysis*, described in [1], to solve this problem.

The first step in shadow analysis is to extract the shadow of the user's hands from the scene. We accomplish this by thresholding the image in 8-bit HLS colorspace coordinates. From examining many test images, we found that a threshold of $L < 0.30 \cdot 255$ produced a binary image S that clearly depicts the shadows in the image.[4]

For each fingertip position p, we examine square 20×20 neighborhood of p in the binary image S. If the percentage of shadow pixels in the neighborhood is smaller than a fixed percentage s, we conclude that the fingertip is in contact with the table. We refer to s as the *shadow threshold*. In the experimental section, we investigate the effect of varying this threshold.

2.4 Mapping Touch Points to Keystrokes

The previous phase of processing yields a list of image-space coordinates that estimate where the user has touched the keyboard. In this section, we describe how we translate those image-space coordinates to rectilinear keyboard-space coordinates, and then to actual keystrokes, correcting for perspective. The perspective correction technique we used did not appear in any reference we read, instead we derived it from ideas in the lecture on stereo vision.

An image captured by the camera in our system is a *perspective projection* of a three dimensional scene. Our system assumes the layout of the keyboard is known at compile time and that the keyboard-mat given to the user has several control points printed on it, arranged in a rectangle of known height and width. For test purposes, our keyboard consisted of a square-shaped grid with 25 keys, as depicted in Figure 3.

We designate the four control points c_0, c_1, c_2, c_3. Each point $c_i = (x_i, y_i, z_i)$ is a point in the plane of the keyboard mat. We assume our camera is a pinhole camera of focal length d, and carry out all of our calculations in "camera coordinates", so that the camera's aperture is at the origin with its optical axis pointing in the $-z$ direction, with no rotation about the z-axis. For simplicity, we assume all of our length-valued variables have the same units.

To obtain the positions of the control-points in image-space, we begin by thresholding an 8-bit RGB test image captured at the beginning of processing with the inequality $B < 100$. The resulting binary image has four connected components, one for each control point. For each connected component, we

[4] It would be easy to allow users to adjust this threshold at runtime, so we decided not to worry about the fact that our chosen threshold depends on our experimental setup.

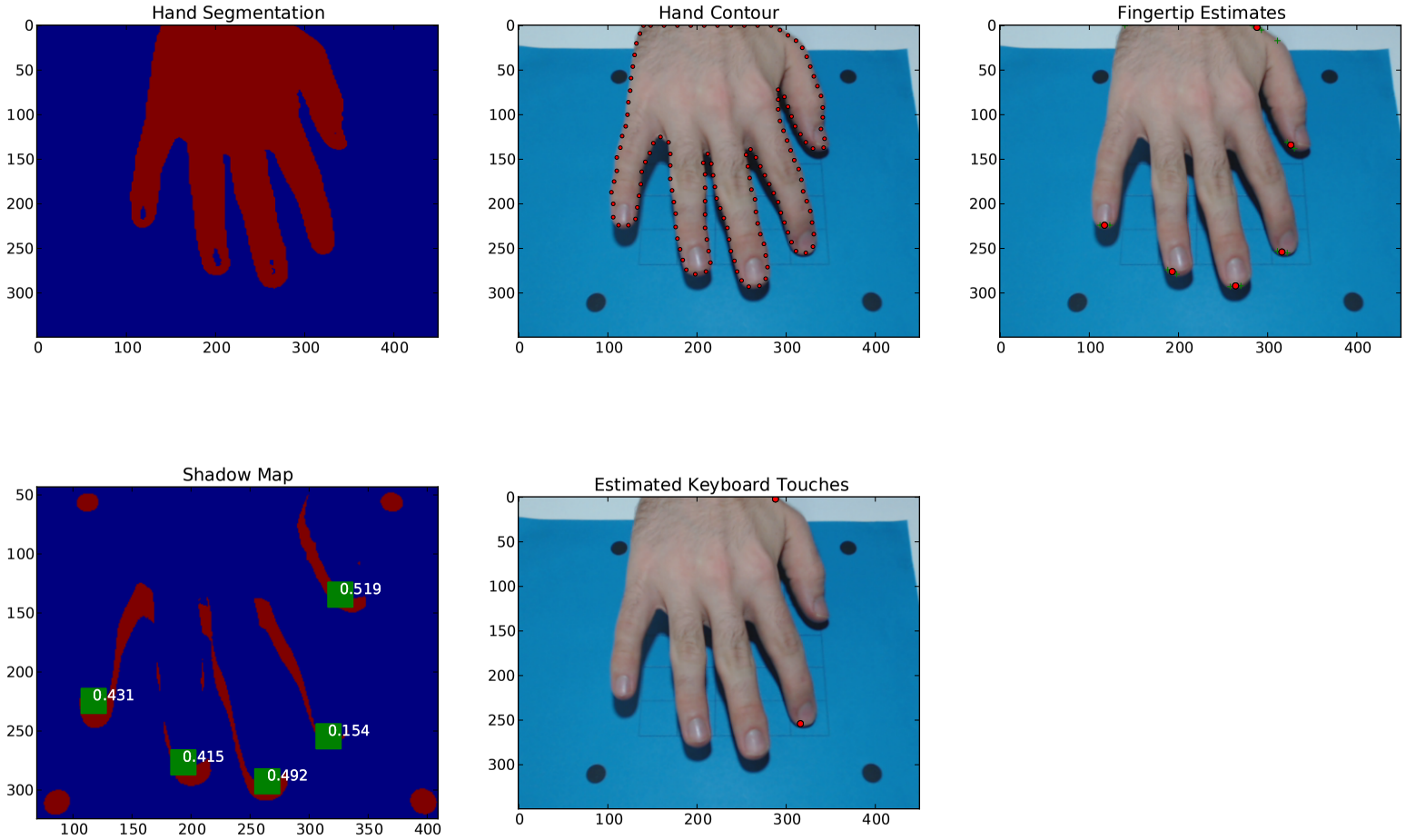

Figure 1: An example of the phases of image analysis. Note that the system captures false positives away from the keyboard-mat. These are removed by a trivial post-processing step that discards any point too close to the boundary of the image. In the phase titled "Shadow Map," the transparent green squares indicate the neighborhood of the fingertip considered for shadow analysis. The nearby numbers indicate what proportion a neighborhood consists of shadow pixels.

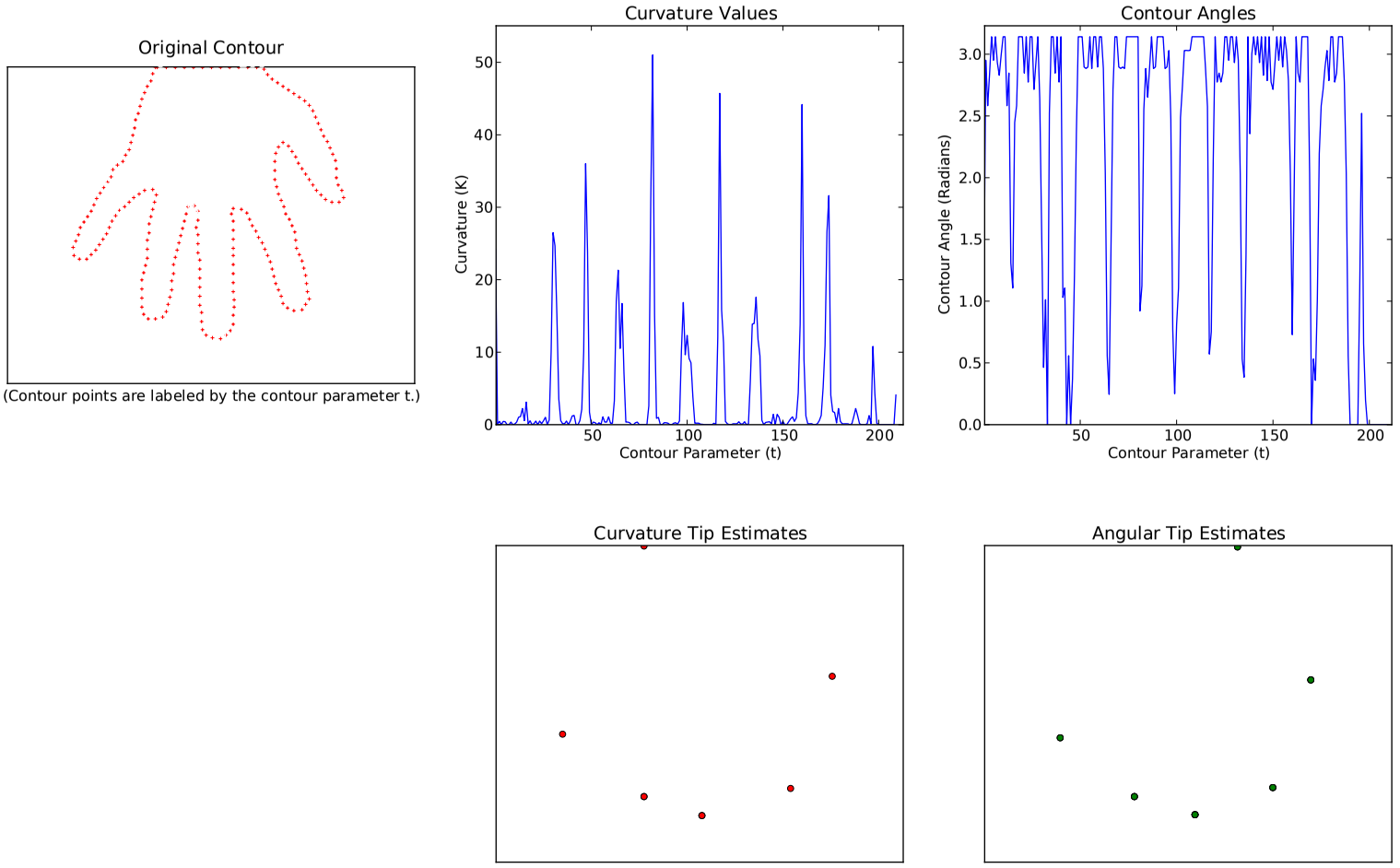

Figure 2: An example of curvature-based and angle-based fingertip detection schemes.

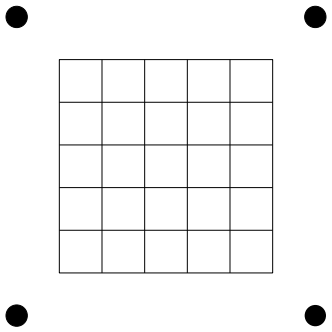

Figure 3: The keyboard-mat used for our tests. The black disks indicate control points.

calculate its center of mass, and take that to be the position of the corresponding control point. We use a simple system of inequalities to assign each control point to the appropriate variable c_i. We will denote the image-space coordinates of c_i by (u_i, v_i). Hence:

$$u_i = -d\frac{x_i}{z_i} \qquad v_i = -d\frac{y_i}{z_i}$$

We assume that the keyboard-mat is aligned so that the vectors $c_1 - c_0$ and $c_2 - c_3$ are parallel to the x-axis with length w and the vectors $c_3 - c_0$ and $c_2 - c_1$ are perpendicular to the x-axis, with length h. Thus, under the perspective projection, the keyboard-mat looks like the image in Figure 4.

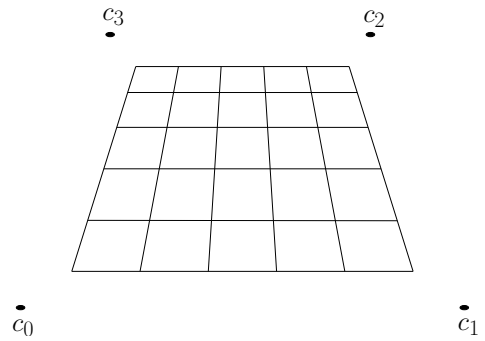

Figure 4: An example of the keyboard mat, under perspective projection

Suppose (u, v) are the image coordinates of a point $c = (x, y, z)$ on the keyboard mat. When we speak of the *keyboard-space coordinates* of that point, we mean $s, t \in \mathbb{R}$ so that:

$$c = c_0 + t(c_1 - c_0) + s(c_3 - c_0)$$

In order to convert image-space coordinates to keyboard-space coordinates, we must solve for s and t, in terms of (u_i, v_i), (u, v), h, and w. Because of the assumptions on the c_i we may rewrite the equation above as:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} + t \begin{bmatrix} w \\ 0 \\ 0 \end{bmatrix} + s \begin{bmatrix} 0 \\ h \\ z_3 - z_0 \end{bmatrix}$$

Hence:

$$u = -d\frac{x}{z} = -d\frac{x_0 + tw}{z_0 + s(z_3 - z_0)} \tag{1}$$

$$v = -d\frac{y}{z} = -d\frac{y_0 + sh}{z_0 + s(z_3 - z_0)} \tag{2}$$

One difficulty is that we do not know z or z_i for any i. However because $c_1 - c_0$ are parallel to the x-axis we have:

$$u_1 - u_0 = -d\frac{x_1}{z_0} + d\frac{x_0}{z_0} = -d\frac{w}{z_0}$$
$$\frac{z_0}{-d} = \frac{w}{u_1 - u_0}$$

The quantity $z_0/(-d)$ is so useful that we will name it explicitly

$$q_0 := \frac{z_0}{-d} = \frac{w}{u_1 - u_0}$$
$$q_3 := \frac{z_3}{-d} = \frac{w}{u_2 - u_3}$$

Using this formula, we can rewrite Equations 1 and 2 as:

$$u(q_0 + s(q_3 - q_0)) = q_0 u_0 + tw$$
$$v(q_0 + s(q_3 - q_0)) = q_0 u_0 + sh$$

and solve for s, t to get the formulae:

$$s = \frac{vq_0 - v_0 q_0}{v(q_0 - q_3) + h}$$
$$t = \frac{1}{w}(u(q_0 + s(q_3 - q_0)) - q_0 u_0)$$

It might appear that in order for the units to come out correctly, we would need to know the width of a one pixel sensor element in our camera. However, observe that everywhere a u or v variable appears in the formula above, it is multiplied by a q variable. This has the effect that any conversion factor will necessarily cancel out, so we do not need to know the width of a sensor element. Similarly, one can see that the focal length d of the camera does not appear anywhere; it is not needed to calculate s and t. Further analysis shows that it is the *ratio* of h and w that is really important for using the above formula,

not their exact values.[5] Once we have the keyboard-coordinates of a point, we assign it to a button identifier (a pair in $\{1,\ldots,5\}\times\{1,\ldots,5\}$) by applying the mapping $(t,s)\mapsto(\lfloor 7t\rfloor,\lfloor 7s\rfloor)$ and discarding output that does not fall within our set of button identifiers.

3 Experimental Results

Detecting the user's keystrokes means performing three tasks:

(i) Locating the user's fingertips.

(ii) Detecting the shadows in a neighborhood of each fingertip.

(iii) Transforming the image coordinates of any touch-point to keyboard coordinates so that it may be assigned to a key-press.

Error in any one step contributes to error in later steps. However, by choosing our data carefully we can analyze how well our system performs steps (i) and (iii) in isolation. In particular, we find that the transformations in step (iii) are not a significant source of error compared to steps (i) and (ii).

3.1 Equipment Details and External Libraries

All of the experiments described in this section were carried out on a Dell Inspiron 530S running Ubuntu 12.10. This computer has an Intel Core 2 Duo CPU (2.40GHz) and 2.9 GiB of RAM. We captured video using a Logitech c920 webcam mounted to a frame that helped ensure the camera would be squarely aligned with the keyboard-mat. The camera captured 8-bit BGR images at a 640×480 resolution, and was positioned approximately 30 centimeters from the keyboard mat such that the angle between the optical axis of the camera and the tabletop was approximately 60 degrees.

For experiments involving shadow analysis, lighting was an important consideration. We used 14 watt, uncovered halogen bulbs to illuminate the scene, suspended approximately 50 cm from the keyboard-mat.

We wrote all of the code used to carry out our experiments in Python. We used Numpy and Scipy for three standard image analysis procedures: thresholding, morphological operations, and connected component labeling. We used the OpenCV (Open Computer Vision) library to retrieve images from our hardware and encode/decode video.

3.2 Fingertip Detection Performance

In this section, we compare the performance of "angle based" and "curvature based" fingertip detection. In our tests, both detectors were given identical contours, generated by the procedures described in 2.1. We tested the detectors

[5] This is useful if one plans to produce copies of the same keyboard-mat at different scales.

on a collection of 40 images displaying a total of 116 fingertips. For each image, we manually determined the positions of the fingertips in the scene, and recorded the results to a log file. For our purposes, a fingertip-position meant a pixel location that appears at the very end of a finger (i.e. at the boundary of the finger and the background) and is centered across the fingertip. Within an image, each finger received exactly one fingertip label.

We scored a detector's output $P = \{p_i\}_{i=1}^{N}$ against the true fingertip points Q using the following procedure: For each $q \in Q$, we examined a neighborhood N of radius 30 pixels about q. If $P \cap N = \varnothing$, we scored q as a false negative. If the detector produced one or more points within the neighborhood, we selected the closest point p_i as a match for q, recorded the Euclidean (L^2) distance between p_i and q, and removed p_i from P so that it could not be paired with another fingertip. In this case, q was counted as a true positive. Any point in P that was not paired with a point in Q was recorded as a false positive. Then, we plotted the resulting pairing between P and Q for human inspection.[6] Thus, each test yields two kinds of statistics: counts of the true/false positives and false negatives as well as an L^2 score that records how accurately our system estimates the positions of the true positives.

The main difficulty a fingertip detector must overcome is that the orientation of the hands in the scene may prevent the camera from recovering a useful silhouette. For example:

- Knuckles can be inadvertently detected if fingers are curled.
- Fingers may shadow or occlude one another.
- If two fingers are close together (without overlapping) they may be segmented as a single region with a boundary that makes it difficult to isolate the tips.

For this reason, we divided our corpus into two parts. We placed in Dataset I (22 images, 60 true fingertip points) those images that did not exhibit any of the problems described above and placed the rest in Dataset II (18 images, 56 true fingertip points). Our results appear in Table 1.[7]

Any user interface imposes constraints upon the user. For example, touch screen users know they cannot rest their hands on the screen as they type. The images in Dataset I represent ideal conditions, where the user understands the limitations of the keyboard interface. The images in Dataset II are meant to give an idea of how well the system performs on "badly conditioned" inputs, so

[6] Using this plot, it is easy to discern which points are really true positives, false positives, and false negatives, so that we may check the scoring system's work. This is why we consider the scoring system to be an acceptable compromise between keeping the testing software relatively simple and automating our tests (so that many tests can be run).

[7] This table presents the True Positive (TP), False Positive (FP), and False Negative (FN) counts for our detectors, but not the True Negative (TN) count. For the purposes of detecting fingertips, virtually every pixel in the image will be correctly classified as a True Negative, yielding counts in the millions, which is misleading when compared with the other counts. This is also why we decided not to calculate the *false positive rate* (FN/(TN + FN)); since TN is so large relative to FN, the false positive rate is always extremely close to 0.

Table 1: Comparison of Fingertip Detectors

Dataset I (60 Points)

Detector	True Positives	Mean L^2 Distance (Pixels)	False Positives	False Negatives
Angle Based	60	5.78	9	0
Curvature Based	58	5.05	4	2

Dataset II (56 Points)

Detector	True Positives	Mean L^2 Distance (Pixels)	False Positives	False Negatives
Angle Based	52	4.43	20	4
Curvature Based	49	5.38	21	7

that one can see how much a user must adjust his habits to use our fingertip detection system successfully.

Table 1 shows that the two detection methods display comparable performance, which is not surprising considering that they measure the same geometrical notion (bending of a curve) in different ways. On well-conditioned data, the curvature-based fingertip detector made slightly fewer errors and estimated the positions of the fingertips slightly more accurately (i.e., with smaller mean L^2 error) than the angle-based detector. However, the curvature-based detector also scored slightly worse on the ill-conditioned data. Given the similarities in performance between the two schemes, we elected to use the angle-based curvature detector in our other tests.

3.3 Keyboard Size and Imaging Geometry

In 2.4 we made several assumptions about the geometry of our imaging system in order to calculate a transformation T from image coordinates to keyboard coordinates. The programs we designed to perform our experiments superimpose horizontal and vertical lines on all video output from the camera, so that the camera can be aligned with the keyboard-mat before each test. In this section, we discuss whether the error that remains in our imaging geometry or the size of the keyboard-mat affects the accuracy of our transformation from image coordinates to keyboard coordinates.

We printed the keyboard-mat displayed in Figure 3 at four different scales, such that the individual (square) buttons had side length 22 mm, 16 mm, 12 mm, and 7 mm. For each keyboard, we prepared and manually labeled a collection of 50 images, in which a single finger selects one of the 25 buttons (each button is selected twice in the images).[8] Under these conditions, we found that our system

[8]In this configuration, we can be very sure that our system correctly labels the fingertip

was able to correctly label all 50 of the images, for each of the four keyboard-mats. From this we conclude that the true geometry of our imaging system is sufficiently compatible with our assumptions, and neither the transformation T nor the size of the keyboard-mat has a significant effect on our system's accuracy. In other words, the errors we will discuss in 3.4 arise because of errors in fingertip detection and shadow analysis, and not error in the transformation T.

3.4 Touch Detection Experiments

From an implementation perspective, shadow analysis is a very appealing technique for detecting whether the user is touching the keyboard mat. The technique is computationally inexpensive, does not require special hardware,[9] and can be applied on a frame-by-frame basis. However, shadow analysis requires strong assumptions on the lighting conditions in the scene. Specifically, we assume:

(a) There is a strong point-source light illuminating the scene, that produces well defined shadows.

(b) The light source has been aligned with the camera, so that in the camera coordinates defined in 2.4, it lies in the yz-plane, between (and above) the camera and the user's hands.

In this section, we discuss the consequences of violating these assumptions.

We adjusted our lighting conditions in several ways. First, we considered the effect of shifting the point-source light so that it appears some distance to the side of the keyboard mat, instead of directly above it, which shifts the shadows in the scene. We also considered the effect of adding diffuse light, which has the effect of weakening the shadows in the scene, and the effect of using only diffuse light for illumination, which yields very weak shadows. Figure 5 provides examples of these illumination conditions, for comparison.

For each of the five lighting conditions, we recorded a total of 50 images (taking care to keep the hand poses the same between different lighting conditions) with 50 fingertips to be classified as either touching or not touching the keyboard-mat (25 touching, 25 not touching). In the cases where the fingertip touched the keyboard-mat, the system needed to report the correct key in order for the result to be scored as a true positive. For these experiments, we used a 20×20 window around each fingertip point, with shadow threshold 0.15.

Table 2 summarizes our results, which clarify our lighting assumptions in two ways. Our system performed acceptably on Dataset 2, so we can refine assumption (b) by saying that the point source light does not need to be perfectly aligned with the camera, so long as the shadows of the fingertips appear near the fingertips. If the shadows are too far from the fingertips, as was the case

and whether it is in contact with the tabletop.

[9] For example the authors of [5] use a 3D range camera to determine distances to fingertips and the keyboard-mat, so that touch detection is a matter of comparing distances.

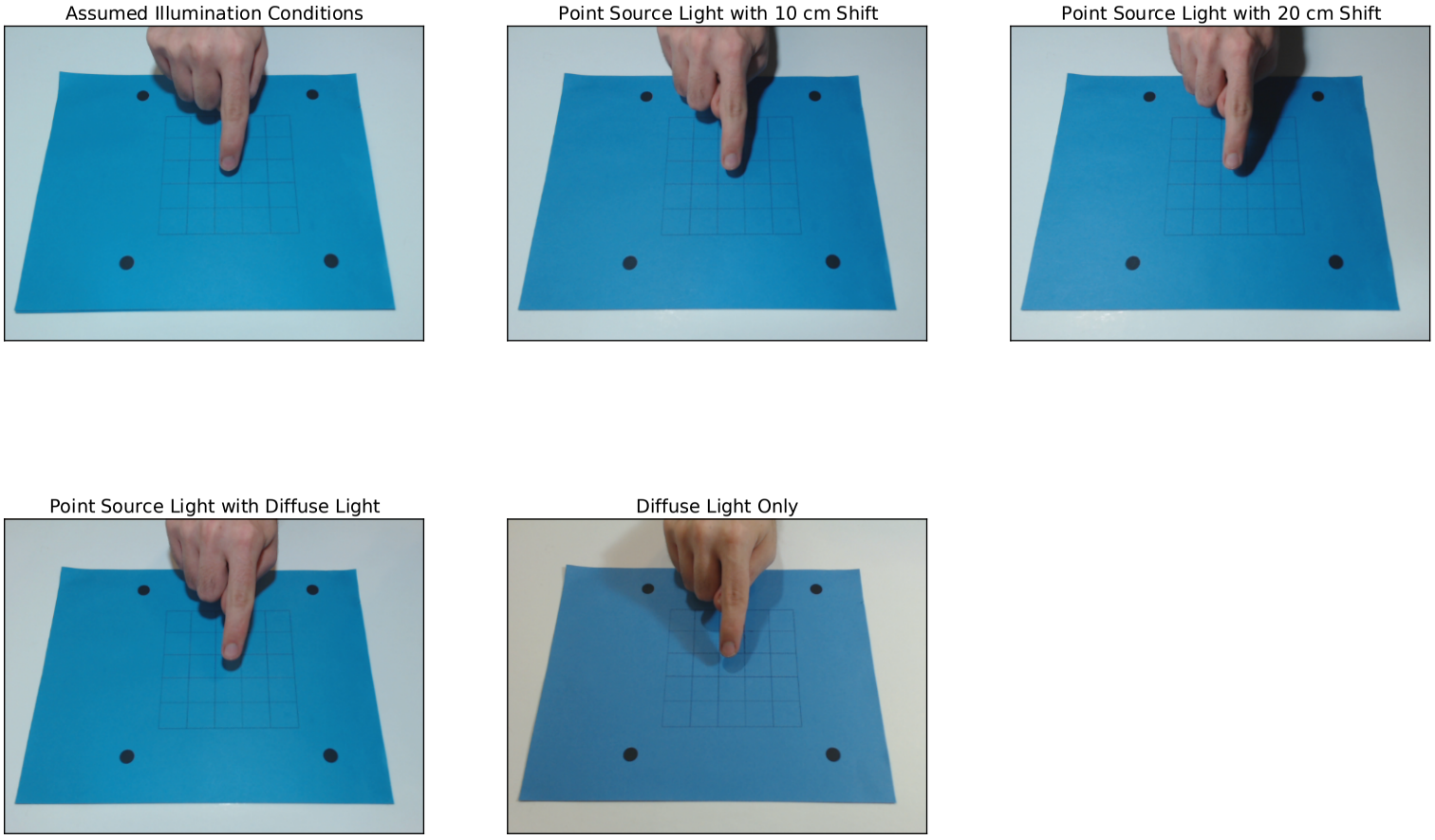

Figure 5: Examples of the five illumination conditions we considered. Note the position and strength of the index finger's shadow in each image.

Table 2: Performance Under Different Lighting Conditions (50 Classifications per Dataset)

Dataset	True Pos.	False Neg.	False Pos.	True Neg.
1. Assumed Illumination Conditions	25	0	0	25
2. Point Source Light with 10 cm Shift	25	0	0	25
3. Point Source Light with 20 cm Shift	30	0	20	25
4. Point Source Light with Diffuse Light	25	0	0	25
5. Diffuse Light Only	32	8	18	17

Table 3: Performance with Different Shadow Thresholds (100 Classifications per Trial)

Shadow Threshold	True Pos.	False Neg.	False Pos.	True Neg.
10 %	22	49	0	29
15 %	57	14	0	29
20 %	68	3	0	29
25 %	69	2	0	29
30 %	69	2	0	29
35 %	69	2	2	27

in Dataset 3, shadow analysis has no hope of correctly discerning whether key-presses occurred. Dataset 4 illustrates that our touch detection system performs acceptably when the point-source light is not the only source of illumination. For example, one can use this system in a room with standard overhead lighting, as long as a strong point source light is the main source of light near the user's hands. On the other hand, Dataset 5 shows a strong point source light is essential for obtaining a useful classification.

To perform shadow analysis, we select two parameters: the size of the square neighborhoods we will inspect and the shadow threshold. We propose that the first parameter, neighborhood size, should be dictated by the width of the user's fingers in the scene (in our case, 20 pixels). If the neighborhood is much larger, pixels near the edge of the neighborhood could easily represent shadows that have nothing to do with whether the fingertip in question contacts the tabletop.

We investigated the effect of the shadow threshold on our system's performance, so that we could better choose this parameter. In [1], the authors explain that they chose their shadow threshold by considering "an extensive set of test cases" but do not list their final threshold value. We manually labeled a dataset of 50 images depicting 71 fingertips in contact with the keyboard-mat and 29 fingertips not in contact with the keyboard-mat. Table 3 lists the results of applying shadow analysis to this data with different shadow thresholds.

As both intuition and the data in Table 3 indicate, a smaller shadow threshold makes it more difficult for a fingertip to be classified as touching the keyboard-mat. Hence, a very low shadow threshold yields few false positives and many false negatives. However, the shadow threshold may be increased substantially, to around 35%, before the number of false positives increases. Based on the table, a shadow threshold between 20% and 25% appears to deliver the best performance.

3.5 Frame Rate

The frame rate of our system depends somewhat on our hardware choices. Logitech reports that their camera is capable of recording video at approximately 30 frames per second (fps) and our own tests agree with that statistic. Our main virtual keyboard program consists of a loop in which we sample an image from the camera, analyze it, and display any resulting button presses. The additional time required to analyze the images means that our virtual keyboard system captures an average of 10 frames per second. (For comparison, the authors of [1] report their system runs at 3 fps, albeit on different hardware.)

We wished to determine the extent to which our program's frame rate affects its accuracy, since this determines how fast a user can type. To simulate different frame rates, we recorded image sequences (at 30 fps) of hands typing, and then ran our system on different subsequences of those frames. For example, to determine the performance of our system at 5 fps, we would provide our program with every sixth frame of the original 30 fps image sequence. In this way, we examined how well the system performed at 5, 10, 15, and 30 fps.[10]

We recorded four different image sequences for our experiments. In sequence V_1 (32 seconds) an index finger presses each of the 25 keys on the keyboard once. In sequence V_2 (15 seconds) a pair of fingers alternate between pressing two distinct keys, for a total of 10 key-presses. In sequence V_3 (5 seconds), an index finger presses a sequence of 5 keys rapidly. In sequence V_4 (7 seconds), a pair of fingers on the same hand rapidly presses a sequence of 8 keys on two different rows.

Since the purpose of our experiment was to determine the effect of frame rate on accuracy, in each video sequence we chose hand poses that would be relatively easy for our system to accurately process. In this way, when the system fails to identify a key-press it does so because it failed to observe a frame in which a fingertip was sufficiently close to a key (and not because the system failed to identify a fingertip entirely). For this reason, we observed no false positives and report only the number of failed identifications in each experiment.

From an initial image sequence $\{I_j\}$ sampled at 30 fps one can obtain several distinct image sequences sampled at 5, 10, and 15 fps. For each dataset V_i, we ran our tests on three subsequences sampled at 5 fps, three subsequences

[10] Most users type somewhere between 30-120 words per minute [6] (here a word means a sequence of 5 characters). From this fact one can roughly estimate that the frame rate should be 2 to 10 fps; hence, the frame rates we chose for our experiments are not completely arbitrary.

sampled at 10 fps, and two subsequences sampled at 15 fps. For each frame rate, we report the average number of errors observed for those trials.

Table 4: System Performance with Different Frame Rates

	Average Number of Errors			
	Frame Rate (fps)			
Dataset (# Keystrokes)	5	10	15	30
V_1 (25)	9.3	5.0	1.5	0.0
V_2 (10)	3.0	0.0	0.0	0.0
V_3 (5)	2.3	0.3	0.5	0.0
V_4 (8)	4.0	3.3	1.0	0.0
All Datasets (47)	18.7 (38.9%)	8.7 (18.1%)	3.0 (6.3%)	0.0 (0.00%)

The results of our experiment, summarized in Table 4, demonstrate that the current frame rate our system provides (about 10 fps) is not satisfactory and should likely be increased to at least 20 fps. The fact that our system made no errors on the 30 fps image sequences and performed best on the V_2 dataset (which featured the slowest rate of typing and longest keypress times) particularly support this conclusion.

We implemented our system in an interpreted programming language (albeit, one with many fast `C/C++` library calls), so converting the system to a faster programming language like `C++` is likely a reasonable way to significantly increase frame rates. We also did not make use of parallel programming in our implementation. For example, one might be able to achieve higher throughput by dedicating one processor to each major step in the virtual keyboard's "pipeline"; that is, one processor extracts the hand region from the input image, the next finds the boundary contour and locates the fingertips, and the last processor determines touch points and maps them to keyboard coordinates.

4 Pedagogical Discussion

In this section we briefly discuss our reasons for selecting this topic as the term project for ECE 532, and what was learned in the process of carrying out the project.

The central goal of this project was to gain a better understanding of how different image analysis techniques are used together to solve a larger, real-world problem. The system we implemented utilizes several techniques discussed in lecture, particularly thresholding, binary morphological operations, connected component analysis, and ideas from stereo vision.

The references we considered outline the image analysis steps required to implement a virtual keyboard, but leave some details to the reader. When we reached an impasse implementing the steps described in our reference papers,

we often had to consider three questions:

- Had we made a mistake in our implementation?
- Should we search the literature for a better technique for performing that step?
- Should we try to carefully characterize the difficulties our system was encountering, in the hope that the step described in our reference papers could be mildly adjusted to overcome those difficulties?

When considering such questions, we learned it was crucial to have a robust library of "utilities" that would allow us to display the data in every step in our system. With the right plotting tools, virtually every problem became easy to understand.

Implementing a virtual keyboard system gave us a better understanding of the trade-offs one must consider when choosing image analysis techniques in the context of a larger system. For example, an important first step in our system is to perform a skin segmentation on an input image. Selecting a technique to perform this segmentation meant balancing competing goals, particularly simplicity of implementation, computation time, and quality of segmentation. Skin segmentation by thresholding, as in 2.1, is easy to implement and runs quickly, but only performs useful segmentations on a relatively limited class of images. A more sophisticated thresholding scheme, like Kittler and Illingworth's information minimization procedure, might yield a more consistent segmentation, at the cost of a more complicated implementation and longer runtime. As we saw in the experimental section, both speed (which affects frame rate) and accuracy (which affects the quality of the analysis we can perform in later steps) are important considerations that must be balanced.

One major difficulty we faced in the course of this project was acquiring the necessary data for testing our system. The papers we consulted, particularly [1] and [2], give a reasonably clear outline for how to construct a virtual keyboard system. However, they do not precisely explain what images are "well-conditioned" input to their virtual keyboard systems. In the beginning phases of the project, we recorded many ill-conditioned test images, so that we often struggled to discern whether the techniques described in the papers were not working or whether the data was problematic.

To put it another way, in the process of developing the software that implements our virtual keyboard system we had to concurrently develop an idea of what "reasonable images" our system could process. For example:

- At one point we realized that our poor choice of lighting conditions and background were causing the skin-segmentation procedures in [1] to fail.
- In another early experiment, we realized that glare and the very poor quality of our first camera were producing test images with inaccurate colors that could not be easily processed.

We overcame these difficulties in part by upgrading to a camera with slightly higher resolution (640×480) and frame rate (30 fps). We also exercised greater control over the lighting conditions of our experiments. We suspect the authors of [1] and [2] constructed similar conditions in which to conduct their experiments. Alternatively, perhaps we are simply unaware of standard "best practices" for recording test data that are well known to computer-vision researchers.[11]

5 Conclusion

In this project, we implemented a virtual keyboard following the steps described in [1] and [2]. Though we achieved a frame rate and accuracy comparable to the references we considered, our experiments show that a virtual keyboard system based on shadow analysis has limitations.

Shadow analysis demands very strong assumptions about the lighting and hand poses that will appear in virtual keyboard images. Unfortunately, as we saw in Section 3, these demands are not necessarily rewarded with dependable performance. We observed similar difficulties with fingertip identification. Both curvature and contour-angle techniques can acceptably identify isolated fingertips, but do not deliver dependable performance when two fingers are merged into a single region or when one finger occludes another. Either technique is prone to assigning false-positives to finger-like objects, particularly knuckles.

We do not believe a truly useful virtual keyboard based on shadow analysis could be incorporated into an existing mobile-computing technology (e.g. a smartphone). To achieve widespread adoption, such a technology must have a far lower rate of error and much weaker lighting assumptions. It is unclear how one could guarantee the assumptions of shadow analysis are satisfied without making burdensome demands on the user.

It would be interesting to know whether shadow analysis could be replaced by a motion analysis technique, such as in [7]. For example, given the location of a fingertip in several consecutive images, one could calculate the fingertip's velocity and identify key-presses as by considering when the fingertip's velocity reverses direction. Naturally, such an approach would require additional data structures and processing steps for tracking multiple fingertips over time.

It may also be the case that current approaches to camera-based virtual keyboards appear unpromising because we are implicitly comparing their performance with the standard physical keyboard interface. One might instead try to characterize what typing tasks users currently perform on their touchscreens and then examine whether those tasks can be performed effectively with a virtual keyboard.

[11] The author is in the graduate mathematics program and has no background in computer vision.

References

[1] Y. Adajania, J. Gosalia, A. Kanade, H. Mehta, and N. Shekokar. Virtual keyboard using shadow analysis. In *Emerging Trends in Engineering and Technology (ICETET), 2010 3rd International Conference on*, page 163 to 165, 2010.

[2] E. Posner, N. Starzicki, and E. Katz. A single camera based floating virtual keyboard with improved touch detection. In *Electrical Electronics Engineers in Israel (IEEEI), 2012 IEEE 27th Convention of*, page 1 to 5, 2012.

[3] M. Hagara and J. Pucik. Fingertip detection for virtual keyboard based on camera. In *Radioelektronika (RADIOELEKTRONIKA), 2013 23rd International Conference*, page 356 to 360, 2013.

[4] D. Chai and K.N. Ngan. Face segmentation using skin-color map in videophone applications. *Circuits and Systems for Video Technology, IEEE Transactions on*, 9(4):551 to 564, 1999.

[5] Huan Du, Thierry Oggier, Felix Lustenberger, and Edoardo Charbon. A Virtual Keyboard Based on True-3D Optical Ranging. In *Proceedings of the British Machine Vision Conference*, volume 1, page 220 to 229, 2005.

[6] R.U. Ayres and K. Martinás. *On the Reappraisal of Microeconomics: Economic Growth and Change in a Material World*. Edward Elgar, 2005.

[7] Sumit Srivastava and Ramesh Chandra Tripathi. Real time mono-vision based customizable virtual keyboard using finger tip speed analysis. In Masaaki Kurosu, editor, *Human-Computer Interaction. Interaction Modalities and Techniques*, volume 8007 of *Lecture Notes in Computer Science*, page 497 to 505. Springer Berlin Heidelberg, 2013.

[8] Jr. Tomita, A. and R. Ishii. Hand shape extraction from a sequence of digitized gray-scale images. In *Industrial Electronics, Control and Instrumentation, 1994. IECON '94., 20th International Conference on*, volume 3, page 1925 to 1930 vol.3, 1994.