Using Drawn Input Devices via Computational Vision

Final project by

Boaz Jan

b.oazjan AT gmail you know the rest

Ehud Barnea

barneaeh AT cs.bgu.ac.il

Introduction

We would like to create a program using a simple webcam that anyone could use.
The user draws any homemade controller and the program should identify it and its buttons and the user's interaction with the drawn controller (specifically key presses).

This way anyone can draw a controller as he wishes and use it for every application, game or emulator possible.

We pursue this idea by first identifying the input device presented to the webcam and its buttons.
Second we identify the user's hands and understand when he clicks by matching the fingertip's location with buttons' locations.

Approach and Method

We wanted to provide a method of interacting with any camera containing computer seamlessly, by using the camera as an input device and a drawn controller to discern between the various possible actions. Thus we've drawn ourselves a neat little controller, in freehand, and considered our options.

1. Drawn input device (DID) Detection

1.1 Detection of line edge points

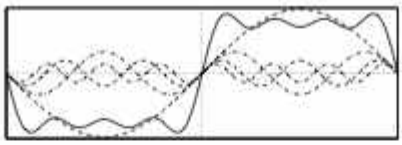
As specified previously, the DID is supposed to be a simple hand drawing over a piece of paper. All the prominent edge detectors (e.g. Canny, Sobel or Prewitt) are built with the only goal of detecting step edges.
These gradient-based edge detectors fail to detect line edges properly as they detect 2 lines to the sides of an original line [2].
Our goal in this preliminary phase is to find line-edge points which are as accurate as possible, in order to have a good base for understanding the underlying shapes.

1.1.1 General edge detection by phase congruency

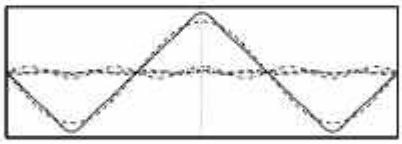
The aforementioned step edge detectors rely on intensity gradient. A different way of searching for general edge points is in the frequency domain.
An image can be represented as a function of sinusoidal waves with frequency,

phase and amplitude.
It has been found [3][4] that in edge points all the sinusoidal waves are in phase as so we will look for these exact points.

As can be seen in this figure, all the phases are in congruence at edge points



$$s(x) = \sum_{n=0}^{\infty} \frac{1}{(2n+1)} sin[(2n+1)x]$$

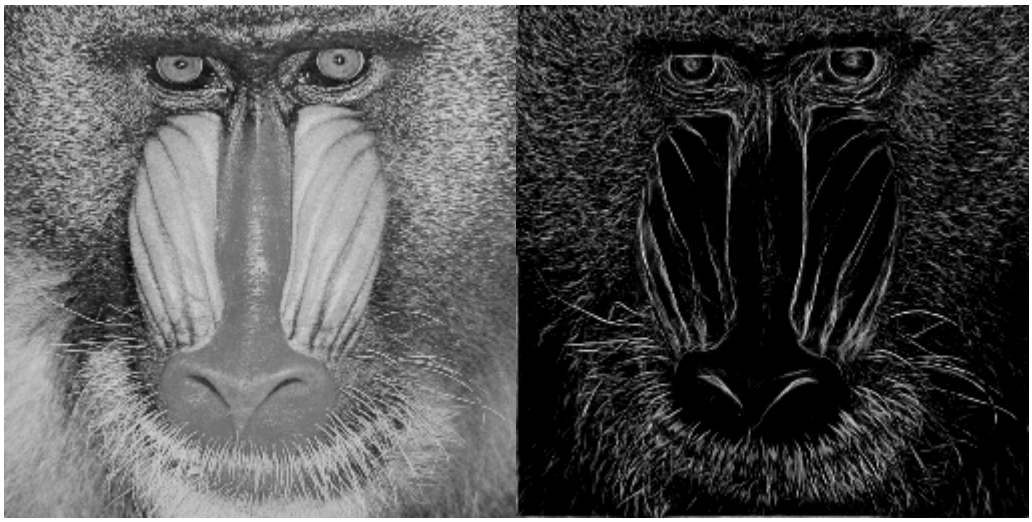


$$s(x) = \sum_{n=0}^{\infty} \frac{1}{(2n+1)^2} sin[(2n+1)x + \pi/2]$$

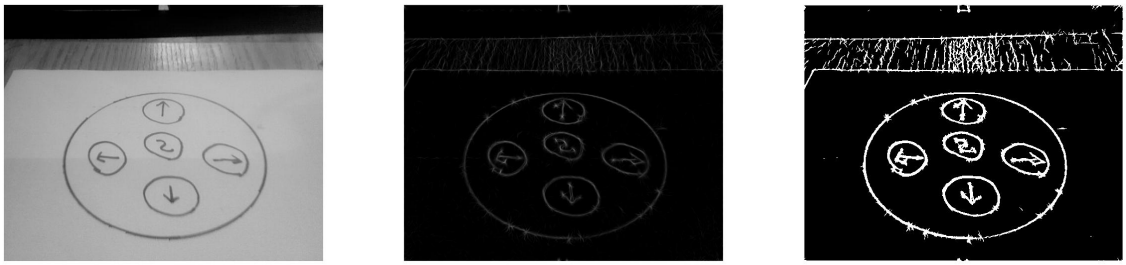
Figure 1. Fourier series of square and triangular waveforms, and sum of the first four terms.

In general terms, these points are found by convolving the image with a set of log-gabor filters of various orientations and using their responses with their amplitude and phase.
A log-gabor is a filter that mimics the V1 simple-cells.

An example of phase congruency. Even the hairs are detected as a single line fragment.



We complete this phase by a process of hysteresis to binarize the generated edge map.



Left – input DID. Middle – phase congruency result. Right – after hysteresis.

1.2 Shape understanding

Once we have found accurate edge points we can begin to group them to relevant shapes.
Before we do so, it is essential to understand the nature of the world we work in. The DID is general shape drawn on paper.
It contains drawn buttons, which are a general shape as well, and inside a button we may or may not have a letter or even another general shape.
General shapes are always harder to deal with, so we will introduce preliminary computations in order to allow us to better work with these shapes.
We will try to fit specific shapes to general shapes when such fitting is possible, but the fitting of specific shapes is not necessary.
It is important to notice that normal controller buttons are mostly convex and made of closed line segments. We will use this information to ease later computations.

1.2.1 Grouping of connected components

Since the buttons, button-markings and even the DID's body itself are all a combination of lines, they are all made of well connected edge points.
From the edge points image we find all the connected components.
Each component is a shape (from previously specified reasons) and so we can already get an initial understanding of the found shapes.
Connected components made of a few number of edge points are considered as noise and thus removed.
Now we move on to a more high-level phase of dealing with those connected components.

1.2.2 Convex hull as a convex-shape sparse representation

Each connected component is made of many points, some of which are a by product of noise gathered in previous phases.
Under these conditions, specific shape fitting is hard and expensive.
Furthermore, a general shape may not be similar to any specific shape, so fitting over general shapes is sometimes impossible.
Never the less, these shapes must be handled with.

To ease the computations we want to find a different way to represent each shape. We have the connected components found earlier, but these contain too much information.
Neuroscientists found that “neurons encode sensory information using a small number of active neurons at any given point in time” [4].
This encoding idea was named “sparse coding”. Instead of keeping and working on the data in its entirety,
we encode the data by using a much smaller amount of memory to represent it.
Using this sparse representation we consume much less memory, and more importantly, out computations will be much less time consuming.

Since the DID shapes are mostly convex we can simply calculate every shape’s convex hull and keep it as our sparse representation.
The convex hull is a set of points much smaller than that of the actual shape, since it has no width (perhaps a width of 1 pixel) and describes only the shape’s contour (this representation is very good because of the convexity).

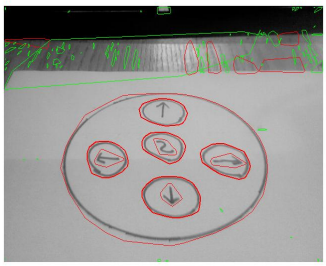
1.2.3 Shape understanding via model fitting

After representing shapes as their convex hulls we can work solely on the hulls, which comprise of much less points.
Since a hull is a set of ordered points shape fitting is now much easier and much quicker.
In this project we haven’t implemented much fitting since the hulls are enough to identify buttons and button clicks, but we did implement ellipse fitting.
Our camera isn’t directly parallel to the DID, so under perspective projection circles become ellipses.
In order to fit an ellipse we use the canonical ellipse equation (and the hull point’s x values) and calculate the difference between the equation and the hull point’s y values.
The found scalar gives us an estimate, or a grade, to how good the fit actually is.
This is the where the sparseness is important. Calculating the fit we use all the hull’s points. A sparser hull means fewer points to calculate.

Since the hull is an ordered set of points we have a lot of fitting options because we can work with the changes of curvature for example.

1.3 Constructing the DID from found shapes/hulls

We found our shapes (represented by their hulls) but we still don’t know which shapes are buttons.
Understanding which shapes are buttons is somewhat of a perceptual question, so we did something rather simple.
We detect the ellipse of largest area and declare it as the DID’s body. Every shape inside it is labeled as a button if it isn’t inside any other shape but the body.



This image shows the found convex hulls. A red hull means it was recognized as elliptic enough (we don't demand very precise ellipse in this example)
and a wide hull means it was recognized as a button. Notice all the found shape at the top are irrelevant since they are not buttons.

A button has a shape inside it specifying its action. Alas, we haven't implemented any text recognition in this project and so we've only defined
the following 2 DIDs in which we know each button's action by its location (center's x-y coordinates) in relation to the other buttons.
Using text recognition to understand the shape inside every button any DID can be used without preconfiguring it.

With a list of buttons and their hulls all that is left is to understand whether a finger is pressing over one of the found buttons.

2. Click Detection

In its basic usage scenario, a controller needs to be clicked. Which means we have to define what is the "Click" operation in our DID.
Because of our time limit, we chose to use the simplest definition possible - "Click by Concealment". Meaning, a click is when a fingertip and a button-fill share the same point in the image-plane (if we had more time we would have implemented the Click detection with optical flow).
And thus, we needed to detect and track hands in our video feed.

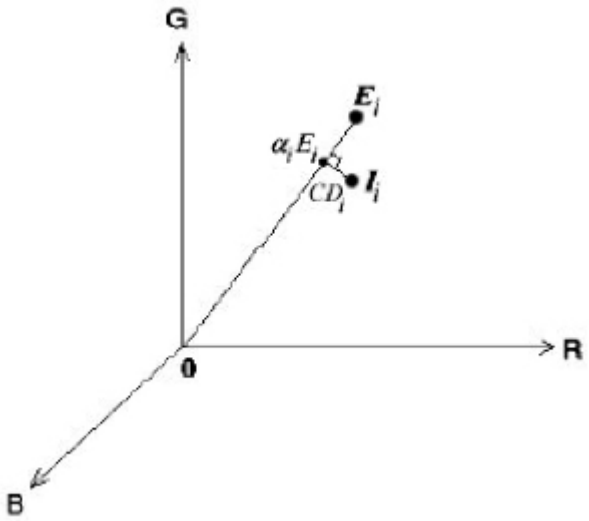
2.1. Hand Detection

Our method of choice for hand detection was a natural one; we used Background Subtraction and Segmentation by Skin Color as our main approach.

2.1.1. Background Subtraction

Our chosen method of Background Subtraction was based on the following article [5]:
"A Statistical Approach for Real-Time Robust Background Subtraction and Shadow Detection"
This paper presents an algorithm for detecting moving objects against a pre learned static background, regardless of shading and illumination.
The algorithm is based on a proposed computational color model, which separates the brightness from the chromaticity component.

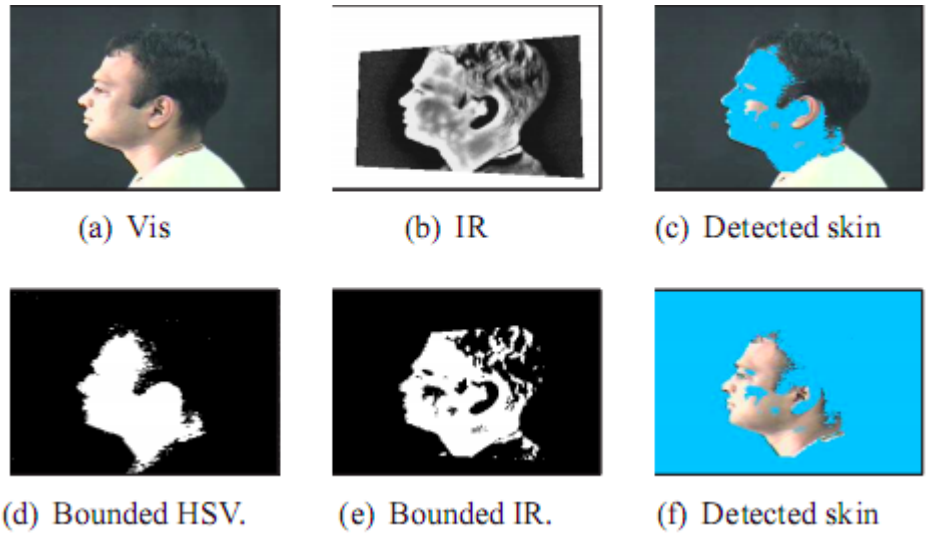
First, a background model is needed. In the background learning process, the first 30 static frames are used to create a median based model for the background due to noise and illumination fluctuations.
When comparing a frame with the background model, the algorithm can will detect any foreign element or shading in the scene.
our main calculated parameters are E, S, A and B.
E is the expected color value for that particular pixel (means of the RGB values).
S is the standard deviation of the color value. A is the variation of the brightness distortion.
B is the variation in the chromaticity distortion.
The brightness distortion is the pixel's strength of brightness with respect to the expected value.
In the figure below, the brightness distortion is a scalar value that brings the observed color close to the expected chromaticity line, which is the line passing from the origin to the point E.



E represents the expected color value of a given pixel, i, and I represents the color value of the pixel i of the current image.
The difference between I and E is decomposed into brightness and chromaticity components. Color distortion is defined as the orthogonal distance between the observed color and the expected chromaticity line.
Once we've finished with the Background learning process, we can compare new images to our background model and classify each pixel using a set of thresholds visible in our source code (the values we're chosen by trial & error process).
At the end of this stage we hold a MASK matrix (Black & White matrix with the same dimension as the original image) that each 1-valued cell corresponds to a pixel with "moving-content", i.e. a pixel that doesn't fit to our detention of background (as mentioned earlier, our definition of a background is based on manually chosen thresholds).

2.1.2. Segmentation by Skin Color

Our chosen method of Segmentation by Skin Color was based on the followin article [6]:
"Detector adaptation by maximizing agreement between independent data sources" This paper presents an algorithm for detecting skin colored pixels using data collected by the writers from experiments with thermal imagery and a dynamic set of thresholds to detect changes in shading and illuminations in the HSV color space.



(a) Vis (b) IR (c) Detected skin

(d) Bounded HSV. (e) Bounded IR. (f) Detected skin

Similarly to the previous stage, this process gave us a MASK matrix that each 1-valued cell corresponds to a pixel with "high probability skin color".

2.1.3. Enhancements & Final Mask

In the 2 masks from the previous stages we can find plenty of noisy pixels an false positive detections due to many reasons (threshold are far from optimal, sudden illuminations shifts, a huge Ehud-sized shadow walking past the webcam).
Thus we decided to perform a logical AND on both masks to get, in theory, only the moving-skin segments of both masks.
The result was conclusive - "We need a new theory".
Thus, we found a decent method of improving both masks (and the result mask) with various morphological and geometric methods. Such as:

- Dilate.
- Erode.
- Finding the various connected components in the image and removing all but the largest one.
- Filling "holes" completely bounded by a single connected components.

Now we have a suitable Mask for the hand.

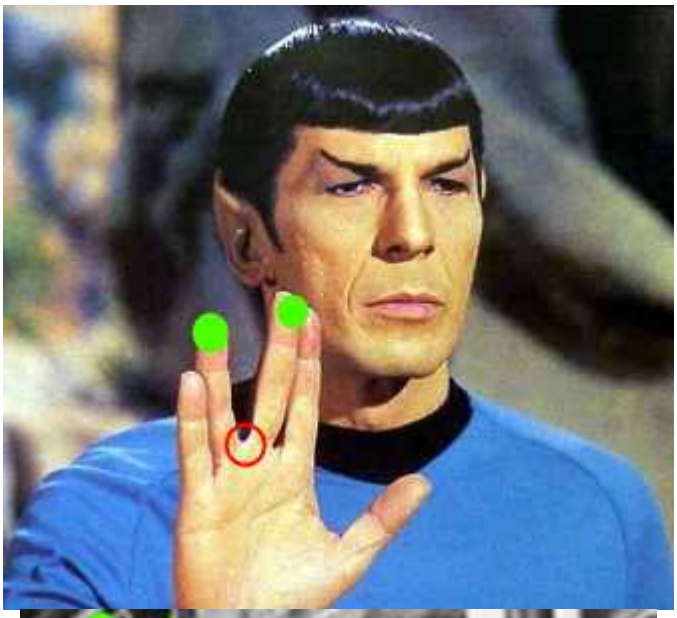
2.2. Fingertip Detection

Now, when we can extract the hand out of the original image, we need to find the

actual tip of the finger which is a single point per finger.
To calculate this lot of points we used an algorithm calld K-Curvature which is defined as follows:

- Let C be the counter of the hand mask and C(i) the i-th point in C
- By calculating the angle theta between the 2 vectors (C(i),C(i+K)) and (C(i),C(i-K)) we can estimate the shape of the contour in the interval [C(i-K),C(i+K)].
- We chose K & theta by trial and error

By providing a cyclic representation of the contour of the hand we get a list of pixels that fit to our K & theta constants,
but mostly we got chains of pixels that fits and not single pixels, so we filtered them and only took the middle value. But now we face a new problem best described but the following picture:



How can we discern between the "bad" angles and the "good" angles? (marked by a red circle & a green fill-circle respectively on the fingertips of Spock and Churchill)
The answer was simple enough, we checked that the y value of C(i) is smaller than the y value of C(i+K) & C(i-K).
This solution sufficed because Click is operated toward the webcam, and it works even if the hand is in a large angle.
Now we have the finger tips.

2.3. Actual Clicking

Since we defined the Click operation as "Click by Concealment", all we need to do is to check if a fingertip and a button's convex-hull collide. Our original plan was to define the Click operation as "Click by Flow", we wanted to understand the movement a fingertip point does while performing a physical click and search for it over time in our application But due to time constraints and implementation difficulties, we couldn't do so.

3. What the code actully do?

- Run Main.m
- A GUI should appear, Click on the "Start Camera" button.
- After the camera's exposure feature balance and the frame contains only background elements, Click on the "Start Tracking" button.
- A Notepad window should appear LEAVE THE NOTEPAD WINDOW FOCUSED AT ALL TIMES!!! (or else matlab wont be able to send to it the output).
- Leave the application to work untill you'll see the message "Start Clicking!" in the Matlab command window.

  The application should detect one of the 2 predefined DIDs (if they're there) and print a button-representing string to the opened Notepad window.
  Our original plan was to send the keys to a Super-Mario/Snake window, but due to time constraints and implementation difficulties, we couldn't do so.

Results

Test #1: Arrows DID
Test #2: Arrows DID
Test #3: H,E,L,O,WORLD! = Hello World!

Conclusions

We succeeded in reaching goals, however, further enhancements can greatly improve out results. Our methods are sensitive to illumination conditions and rendering the algorithms more robust is necessary.
Given more time we could also implement button definition detection via OCR (the drawing inside the button will give the button meaning).
We would also implement click detection via optical flow of the fingertips, specifying that a click occurs only at a certain curvature trace.
Segmentation by skin color may not be a good idea.

Moreover, We would have liked to fit more button shapes (this is not needed since we also work with general shapes).
Thus we conclude that this implementation answers the goal set but some improvements are still due to make it really usable.

Additional Information

- Download This project report in PDF
- Download Oral presentation slides in PDF
- Download Matlab source code.

References

[1] **Introduction to Computational and Biological Vision** - Dr. Ohad Ben-Shahar.
[2] **Edges are not just steps** – Peter Kovesi, ACCV2002.
[3] **Image features from phase congruency** – Peter Kovesi, Videre: Journal of Computer Vision Research 1999.
[4] **Sparse coding of sensory inputs** - Bruno A Olshausen and David J Field, Current Opinion in Neurobiology 2004.
[5] **A Statistical Approach for Real-Time Robust Background Subtraction and Shadow Detection** - Thanarat Horprasert, David Harwood and Larry S. Davis, University of Maryland.
[6] **"Detector adaptation by maximizing agreement between independent data sources"** - Ciaran O Conaire, Noel E. O'Connor and Alan F. Smeaton, 2007