PROJECT PALM

December 3, 2014

Nikhil Dixit, Thejas Prasad, Chirag Sakhuja, Julian Sia
The University of Texas at Austin
Dr. Bryan Klingner

OVERVIEW

Project Palm is an attempt to use hand movement to control a computer mouse. The project uses a webcam to segment the user's hand and track traits such as hand position and number of fingers extended. The goal was to be able to reliably translate these attributes into mouse movements and actions like single and double clicking. To achieve this this goal, we divided the project into four discrete objectives: track and extract a hand using a built in webcam, track individual fingers from the extracted hand, analyze traces of fingers to identify gestures, and map gestures to a mouse. We had two final meausurable goals, which we defined as success: produce a bounding box around a 200 frame sequence of a moving hand with at-least 70% accuracy, and detect if the index finger is open or closed in 65% of 240 labeled frames. We were able to meet the first objective exactly by achieving a bounding box accuracy of 69.6%. We had to modify our second goal to track whether two fingers were open or closed (instead of a single finger). Once we did that we were able to get a true-positive detection rate of 49.4%, which was lower than our goal but still good enough to be effective.

BACKGROUND

Hand and finger detection is a relatively mature field. There are many academic papers written on the topic of hand segmentation, most concerned with the idea of HCI (Human Computer Interaction). We found a few papers that had goals very similar to ours. One of them, titled "Adaptive Skin Color Model for Hand Segmentation" by Dawod et. al. explored a novel method for segmenting hands from images (something we also do) by using a color space conversion to YCbCr and then applying edge detection. For each frame, the algorithm performs a conversion to YCbCr, performs some erosion and dilation iterations to remove external noise, and then uses edge detection to find the border of the hand. The interesting part of the algorithm is the post processing that it does after conversion to YCbCr. Dawod et. al. map the (Cb, Cr) coordinates of the image to a fitted ellipse. The axes of the fitted ellipse are changed based on the color of the skin (hence "adaptive"), and everything that doesn't fit is thrown out as noise. Though the algorithm is quite robust for various hand positions and orientations, we did not implement it entirely since it is very involved and it does not do finger extraction.

Another paper that deals more with finger extraction is "Implementation of an Improved HCI Application for Hand Gesture Recognition" by Joshi et. al. They follow a similar scheme

as the previous paper by converting the image to YCbCr and then converting to grayscale. However, their algorithm also does feature extraction by performing a few extra steps: namely cluster formation and centroid calculation. These steps allows the algorithm to then go on and detect fingertips and give information about each fingertip. This paper claims to improve existing segmentation and fingertip calculation by implementing a new Gaussian model when converting from YCbCr to grayscale. The new model results in better data for hand segmentation which in turn results in better data for fingertip calculation and gesture detection. We tried to implement this paper's model but had little success.

In terms of practical implementation details, as a common starting point we referenced various tutorial snippets such as "Iftheqhar's blog: Opencv python hand gesture recognition" and tutorials by Rosebrock on skin detection. As usual, we referenced OpenCV's documentation.

METHODS: LANGUAGES, LIBRARIES, HARDWARE

Our algorithm is written entirely in Python 2.7 and uses OpenCV to do the bulk of our image processing. We chose this route because all of us are now familiar with this combination and we felt that we could jump right into the work. We also used an external library called PyUserInput which has python bindings for GUI manipulation [3]. This was useful because it gave us an easy way to map the hand and finger data into mouse movements and actions.

As for hardware, our algorithm does require a video input source. While developing our algorithm, we used the built-in webcam of an Apple Mac Pro. This webcam gave us the best results mainly because its good lighting performance.

METHODS: ALGORITHM

Our algorithm consists of these main steps for each frame: detecting the hue of the hand, preprocessing the frame, detecting where the hand is and how many fingers are up, and converting the hand/finger data into mouse movement.

The goal of hue detection is to determine optimal bounds for the color of the skin. To accomplisih this, we instruct the user to place his or her palm in a designated area of background in the view of the camera. Then, the camera takes 25 frames, using the 5th and

95th percentiles of the frames to adjust a base set of HSV value limits so that these limits correspond better to the color of the user's palm. These hue limits are then passed to the preprocessing step.

The goal of preprocessing is to segment the hand from the rest of the frame. To do this, we first take the captured frame and convert it to the HSV color space, which worked better for us than the YCbCr space. After we have converted to HSV, we apply a inRange modification, using the limits established in the hue detection phase of the algorithm, to black out parts of the frame which are not skin-colored. This is still very crude, so we apply a series of erosions and dilations to further remove noise. After that is done, we apply a small Gaussian blur to smoothen any rough edges. Finally, we convert the image to grayscale and apply a final threshold to remove any last traces of non-skin colored noise. At the end of preprocessing, we ideally get an image that only has skin colored objects:

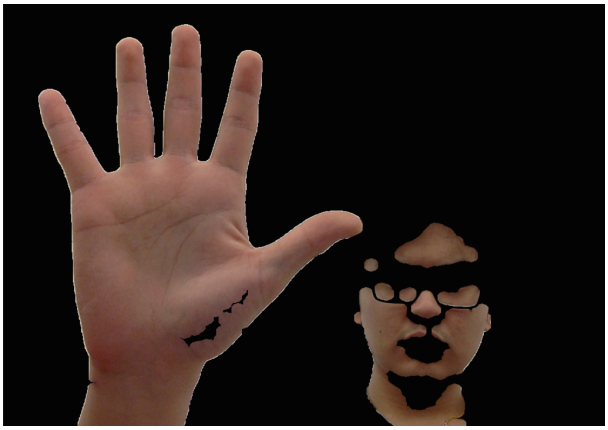

Figure 1: Filtering out non-skin colored patches of the background

After preprocessing, we move into the hand and finger detection algorithm. We first begin by trying to find the hand in the processed image by finding all the contours in the frame. OpenCV has a built-in function call for this which makes the process easy. Note that because the frame also has things in it like the face of the person and other noise, many contours are usually found. To get rid of the false positives, we take the contour with the largest area and throw out the rest. Here is an image of a "hand" detected by our algorithm.

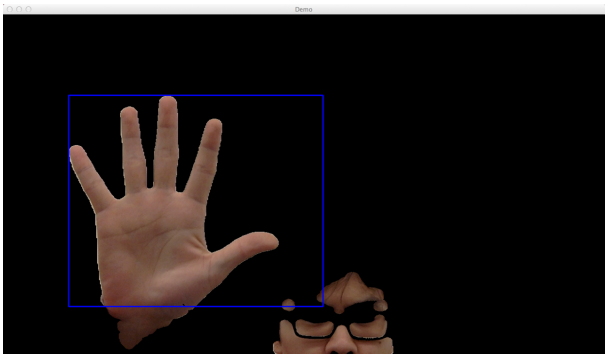

Figure 2: Drawing a bounding box around the largest contour area ("the hand")

Once the hand is found, the next step is to detect how many fingers are up and where they are. To do this we first convert the contour into a convex hull. This puts a convex bounding box around our hand. The hull itself is not displayed, but if we had displayed it on camera, the trace would closely resemble that as demonstrated in in Figure 4. Since the hull is convex, it doesn't ever bulge inwards. This gives rise to the idea of a convexity defect. A defect is simply a place where the contour and the hull are not the same. This works perfectly for finger detection since the wells between fingers are rather large convexity defects:

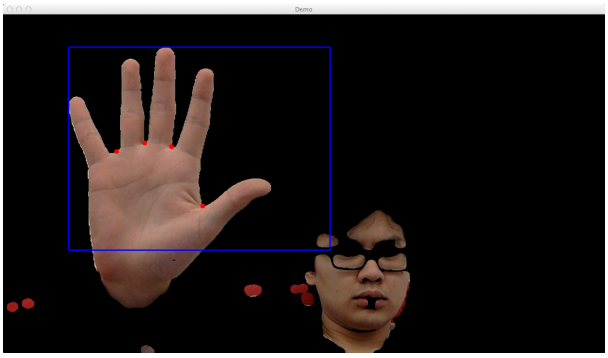

Figure 3: Our final project: finger detection using convexity defects as proxies for fingers

Once we have the convexity defects (another OpenCV call), we need to filter out the ones that do not correspond to finger wells. We used the dismissal algorithm shown at [1]. Each convexity defect is dismissed if it meets either of two criteria:

$$length < 0.4 l_{bb}$$

$$angle > 80°$$

Where l_{bb} is the length of the bounding box around the hand and angle is the angle between the defect and its two adjacent hull points as shown here:

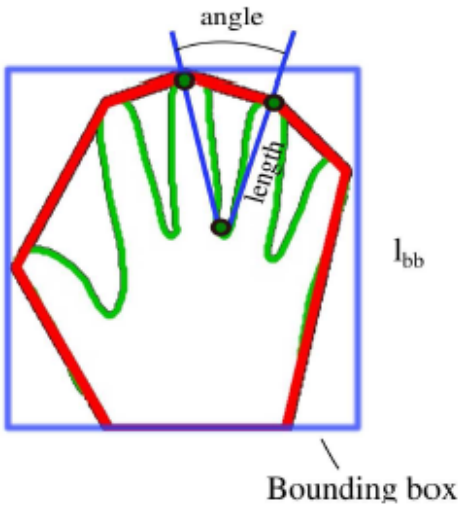

Figure 4: The "length" and "angle" used in Dismissal Heuristic

After the removal of the noise convexity defects, we should only be left with defects which correspond to the finger wells. The final step is to map this information to mouse movement. This step is the easiest of the three. We interpret the number of detected wells as the number of fingers up. This is why our algorithm needs two fingers up to produce any mouse movement (one finger by itself doesn't produce any convexity defects). We can track the center of the hand (which is the center of the bounding box generated by the convex hull function) and map that to mouse movement. When we detect two fingers go up and then down within a certain timeframe, we flag that as a mouse click. Note that by using PyUserInput, a click and mouse movement are just function calls.

PROBLEMS WITH OUR ALGORITHM

There are two problems with our algorithm, reliance on skin-color and robustness. We do a mostly static (alternatively viewed as quasi-dynamic) segmentation for skin color. That is, when we do the HSV inRange calculation, our range is static with exception to the rectification that occurs in the hue detection phase. This is non-ideal for different colored hands. In the future, we would like to make this range fully dynamic so that the machine can learn the optimal values depending on the hand. Furthermore, because we rely solely on color, lighting and background changes affect us quite a lot. For example, wearing a brown or orange colored shirt throws off the detector. In the future we would like to also incorporate stereo vision. That way, we can use the depth as another cue and make the detection less prone to outside factors.

RESULTS

We achieved 69.6% accuracy for hand tracking and 49.4% accuracy for finger detection. To measure the hand tracking accuracy, we first shot a 120 frame video sequence. We went through every 10th frame of the video and marked the bounding box around the hand. We only tagged every 10th frame due to time constraints. We then interpolated the bounding box for the frames in between the tagged frames. This worked relatively well since the hand doesn't move extremely fast. We saved all the bounding box data per frame in a ground truth file. To test our algorithm, we loaded the same video as the Video Capture and ran our hand tracking, saving the resulting bounding box generated each frame. To generate a percentage accuracy, we devised the following test:

For each frame, we define $actual\_bb$ to be the bounding box generated by our system, and $ground\_bb$ to be the ground truth bounding box. We generate a vector from each vertex in $actual\_bb$ to its corresponding vertex in $ground\_bb$. These are the error vectors, since the magnitude of each of these vectors then corresponds to the error of the vertex. We then sum all of the magnitudes, which represents the total error of the bounding box. If the error is within 600 (which represents a 150 pixel error per vertex), we say that the our algorithm worked and count it as a true positive. If the error is greater than 1200, we declared this a failure. If the error is between 600 and 1200, we linearly interpolate the amount of credit given for this frame. We do this to account for partial detections, such as when the algorithm only detects the fingers.

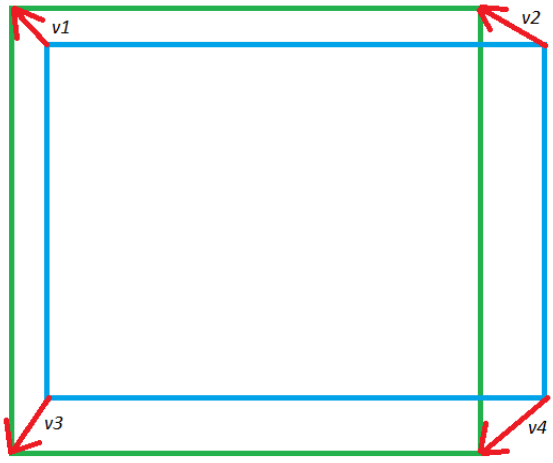

Figure 5: A visualization of our test method for hand tracking. The green box is the ground truth, the blue box is the box generated by our algorithm. The red arrows correspond to the error vectors.

For our second test, we simply tag each frame with the number of fingers held up in that frame. We then simply record the number of hands detected by our algorithm at each frame and compare that to the ground truth. The percentage correctness is given as the number of correctly detected frames divided by the total number of frames. The number reported for this unit test is somewhat low because the unit test implies slightly inaccurate ground truth. Since we only tagged every 10th frame, our interpolation between samples in which the finger count changes will be incorrect for a couple of frames. In doing so, we may falsely mark our algorithm as incorrect for those frames.

REFERENCES

Andresen, S. (n.d.). Simena86/handDetectionCV, [online] Available at: <https://github.com/simena86/handDetectionCV>[Accessed December 2, 2014].

Andresen, S. (n.d.). Simena86/handDetectionCV, [online] Available at: <http://simena86.github.io/images/handRecognition/handangle.png>[Accessed December 2, 2014].

Barton, P. (n.d.). SavinaRoja/PyUserInput, [online] Available at: <https://github.com/SavinaRoja/PyUserInput>[Accessed December 2, 2014].

Dawod, A., Abdullah, J., Alam, J, 2010. Adaptive Skin Color Model for Hand , *2010 International Conference on Computer Applications and Industrial Electronics,* [e-journal] Available at: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5735129> [Accessed 1 December 2014]

Iftheqhar's blog: Opencv python hand gesture recognition. (n.d.). [online] Available at: <http://creat-tabu.blogspot.ro/2013/08/opencv-python-hand-gesture-recognition.html>[Accessed December 2, 2014].

Joshi, M., Patil, S. (2014). A). Implementation of an Improved HCI application for Hand Gesture Recognization. *International Journal of Emerging Trends and Technology in Computer Science,* [e-journal] Available at: <http://www.ijettcs.org/Volume3Issue5/IJETTCS-2014-10-04-52.pdf> [Accessed 1 December 2014]

Rosebrock, A. (2014, August 18). Tutorial: Skin Detection Example using Python and OpenCV, [online] Available at: <http://www.pyimagesearch.com/2014/08/18/skin-detection-step-step-example-using-python-opencv/>[Accessed December 2, 2014].