Josiah Matlack
EECS 332 – Digital Image Analysis
Final Paper
14 March 2013

I. INTRODUCTION

Digital image analysis is a captivating field of computer science, with broad, interesting applications. The field allows for conveniences in every day life – such as face detection on Facebook and digital cameras, image manipulation in programs like GIMP or Adobe Photoshop, recognition of common image paradigms like Chase check upload and QR code scanning, and easy searching of huge databases of images using services like Google Image Search. Analysis also provides for critical functions, such as military intelligence and observation, face detection for security purposes, and much, much more. In fact, the aforementioned applications barely scrape the surface of what is possible, and often already practiced, in this field. Personally, I was enthralled by the power and ubiquity of digital image analysis techniques surveyed in this course. I had failed to realize the importance and prevalence of some of the key concepts, like edge detection, Gaussian smoothing, and color segmentation, but this course allowed a glimpse into this exciting world.

Despite the fact that this is merely an introductory course on the subject, I felt like I learned a huge breadth of topics. In fact, I would feel comfortable tackling all but the most advanced techniques in the field of digital image analysis, armed with the knowledge instilled by this course. Among the things I picked up are: fast iteration techniques for processing images, design and application of structuring elements, Gaussian averaging and normalization, eight-neighbor operations, matrix and vector operations for fast image analysis, and basic image processing techniques divulged in MPs one through six.

Personally, I enjoyed this course and its associated projects very much. Using the backbone of knowledge learned in the course and textbook, as well as generally basic coding skills, I was able to implement advanced image processing functions normally witnessed only in expensive, proprietary software like Adobe Photoshop.

I thought most of the topics were covered clearly, and at a good level of detail. However, there were a few key areas that I would have liked to explore a little more. Primarily, these are a more in-depth discussion (especially regarding possible techniques) of face detection, an exploration of the applications of edge detection, and a close look at techniques for comparing images against one another. Obviously, as the last topic is covered by a project, it is nonessential.

II. PROJECT DESCRIPTION

My project group consisted of John Chandler, Maciek Swiech, and myself. For our final project we decided to implement the FingerCursor program. The goal of FingerCursor is to allow a user to control a cursor using their fingertip, rather than a conventional mouse. The problem underlying this goal is to detect, locate, and track a fingertip accurately and robustly, through a sequence of video frames.

Put in broader terms, given a video with a recognizable human hand, the program should be able to identify where the tip of the finger is in any given video frame, and place the cursor at this position. This identification should function despite poor video quality, fast hand movements, unpredictable lighting, and a myriad of other potential suboptimal conditions.

An optional goal for this project was to allow for complex cursor gestures by the user, much like a conventional mouse pad. While we had hoped to implement a feature such as this, we were unable to due to time constraints. A further discussion of this topic can be found in Section V.

III. DESIGN AND IMPLEMENTATION

Our FingerCursor project was designed, developed, implemented, and tested over the course of roughly half a dozen meetings. In the first meeting, the general format for the project was drawn up on a whiteboard. This format was followed for the project until completion, with only minor modification to the original steps, which were as follows:

1. Video input, decoding and frame sampling
2. Gathering color training data
3. Color segmentation
4. Edge detection
5. Fingertip approximation and identification
6. Result drawing
7. Video output

In the first meeting, we implemented the first three steps. For video input, we used the built-in MATLAB function avireader() for reading in the video files, all provided in the .avi container format. This provided us with a movie object, consisting of a set of frames that can be read as images. However, we discovered during testing in the third meeting that the second two video files, joy.avi and pointer.avi, could not be read by this function due to an underlying error in the files. AVI is a container type, so to solve this problem, we switched to the non-deprecated VideoReader() function, and used VLC to convert the joy.avi and pointer.avi files to mp4 files. This function uses the object-oriented paradigm to store a video as an object, with relevant accessors such as NumberOfFrames() and FrameRate(). In addition, VideoReader() allowed us to use MATLAB's built-in directory structure to read from and write to video files easily, without having to use hard-coded file parameters. Using VideoReader(), each frame of a test video is read and processed sequentially.

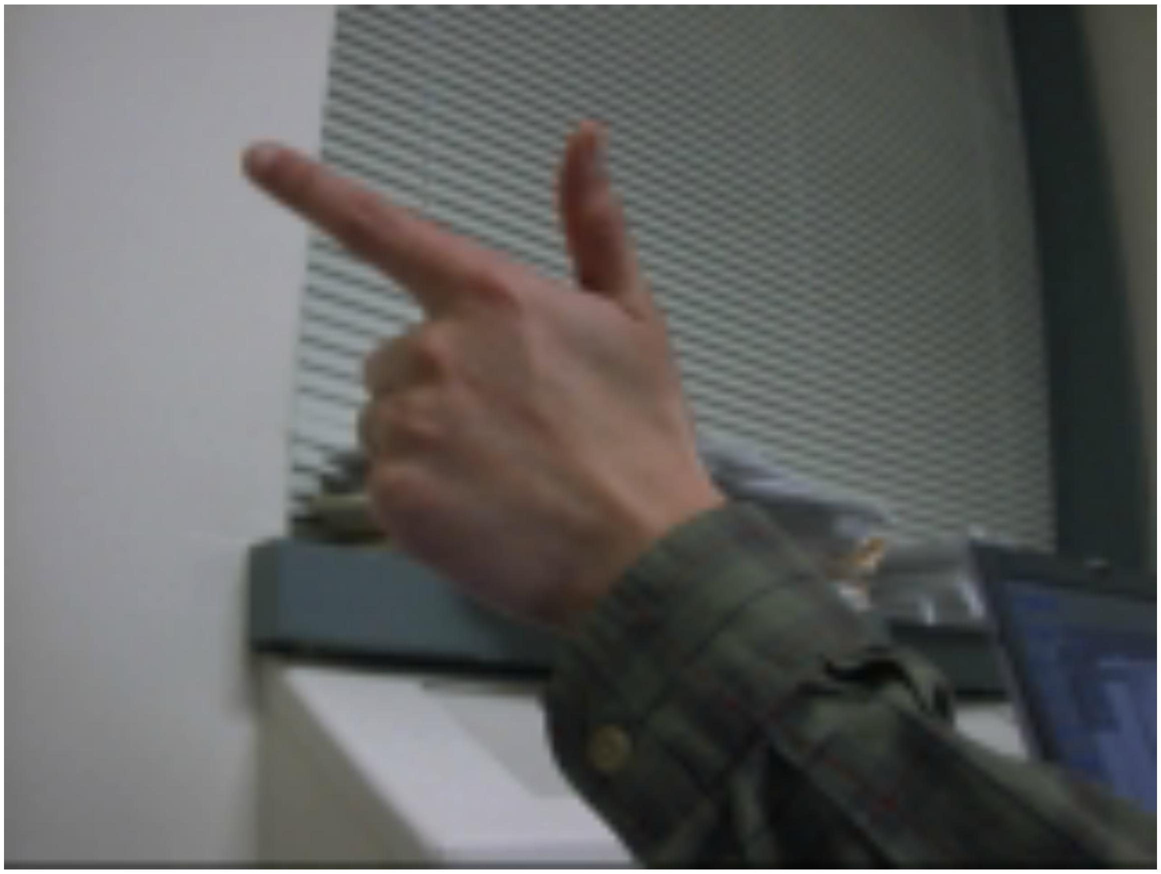
Figure I: the first frame from gun.avi, used extensively in testing

After reliably reading and outputting video frames, we focused on gather training data for our color segmentation model. Unimpressed with the lack of precision in rectangular sampling tools like imcrop(), we opted to use the roipoly() function, which allows the user to select an n-point polygon for the sampling space. This tool allowed us to avoid grabbing unintentional sampling data, such as the sleeve or wedding ring in some of the test images, while simultaneously maximizing the amount of skin sampled per test image. We wrote a wrapper around roipoly() called TrainDetector() that outputs an RGB color space matrix corresponding to the shape bounded by the user. After carefully taking detailed samples of skin from the first frames of the gun.avi, joy.mp4, and pointer.mp4 videos, we had a fairly large corpus of skin data to draw upon (around 15,000 pixels). TrainDetector() also uses the same directory- and file-reading mechanisms previously applied for video reading in FingerCursor() to iterate over files in a directory, presenting all .bmp files to the user for sampling.

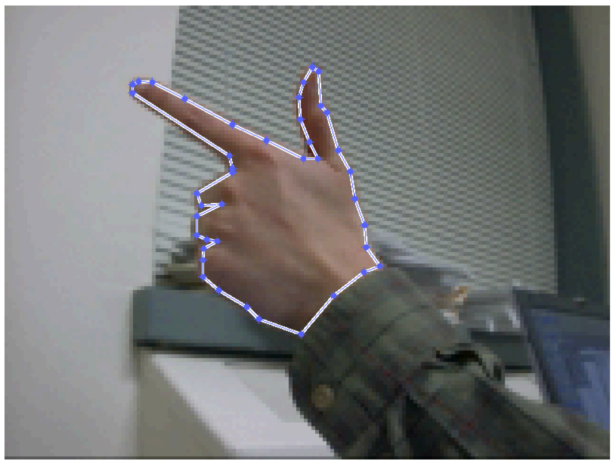

Figure II: Polygonal skin sampling

The last step covered in the first meeting was color segmentation. To tackle this problem, we started off with MATLAB code from MP #4, written by group member Maciek

Swiech. This code uses the standard technique of histogram normalization to produce a color-segmented model. This step consists primarily of two functions: HistoMaker() and Make<color space>(). Make<color space>() consists of three separate functions: MakeRGB, MakeNRGB, and MakeHSI. As the names suggest, the purpose of these functions is to transform a given video frame into the corresponding color space. HistoMaker() accepts an argument corresponding to the color space desired (RGB, NRGB, or HSI). Then, each color value in the image is put into a respective bin. These bins are normalized using variables based on the color space, producing a basis for color comparison.

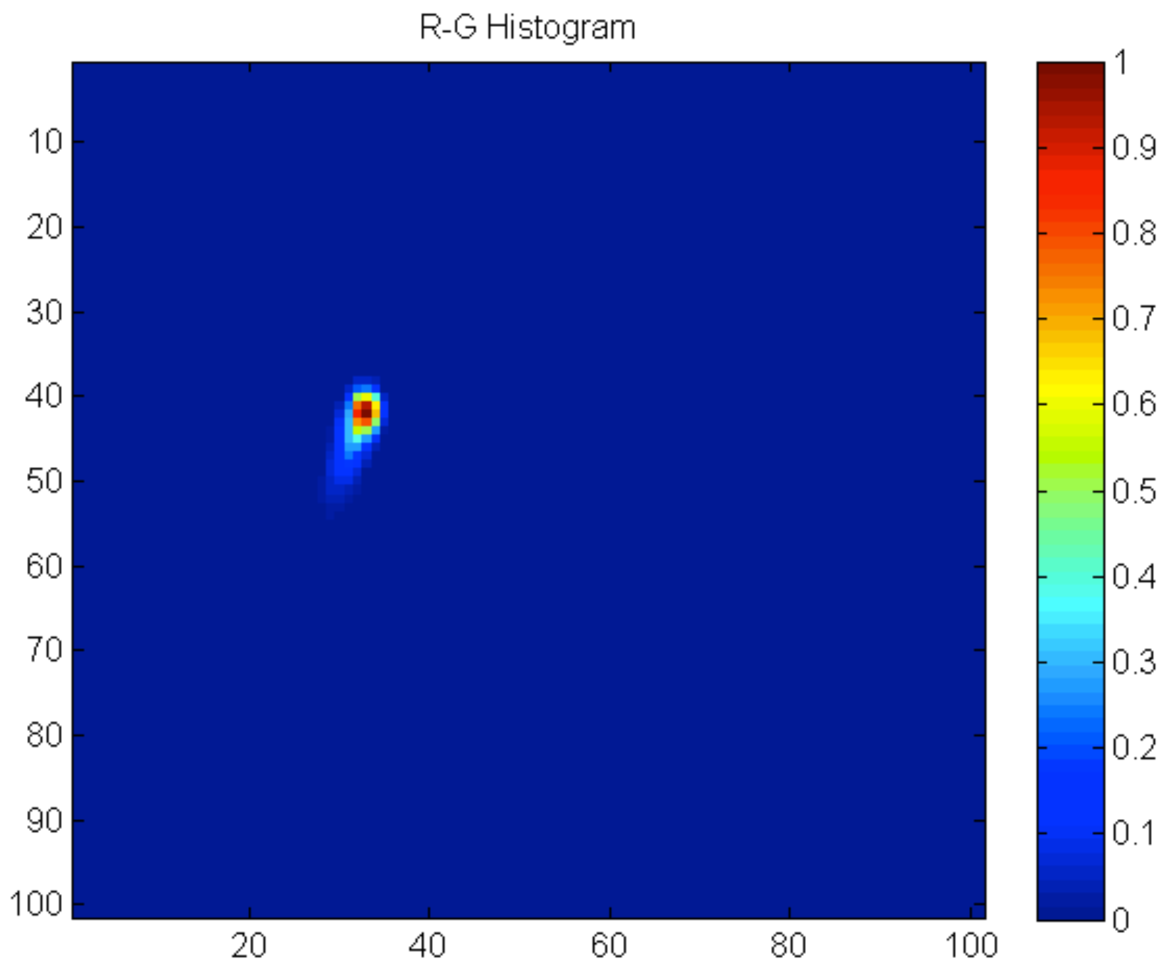

Figure III: normalized histogram data for skin pixels after Gaussian smoothing and thresholding

In the second meeting, we decided that the color segmentation was not meeting the standards we required, and decided to add some improvements. Among these were adding the Gaussian blur to the HistoMaker() function, and normalizing based on a Gaussian curve. Using this curve eliminated much of the noise in the image, especially at color segment boundaries, such as the sleeve. By smoothing the color space of the entire image, our color segmentation is robust in the face of poor lighting and blemishes or abnormalities that may be present on test hands.

After this normalization, any color pixels that did not match the sampled skin data are zeroed out. Initially, in the first meeting, we used the original threshold Maciek had written for comparison of skin pixels. This threshold was 0.05, meaning that any data in the histogram normalization less than 0.05 would be removed. However, after making successive changes in the following meetings, we determined that we could attain better results by using a threshold of 0.01. After these optimizations, our color segmentation code worked significantly better on the testing images, filling in many of the gaps that were previously missing, and eliminating jagged borders and missing data near image boundaries.

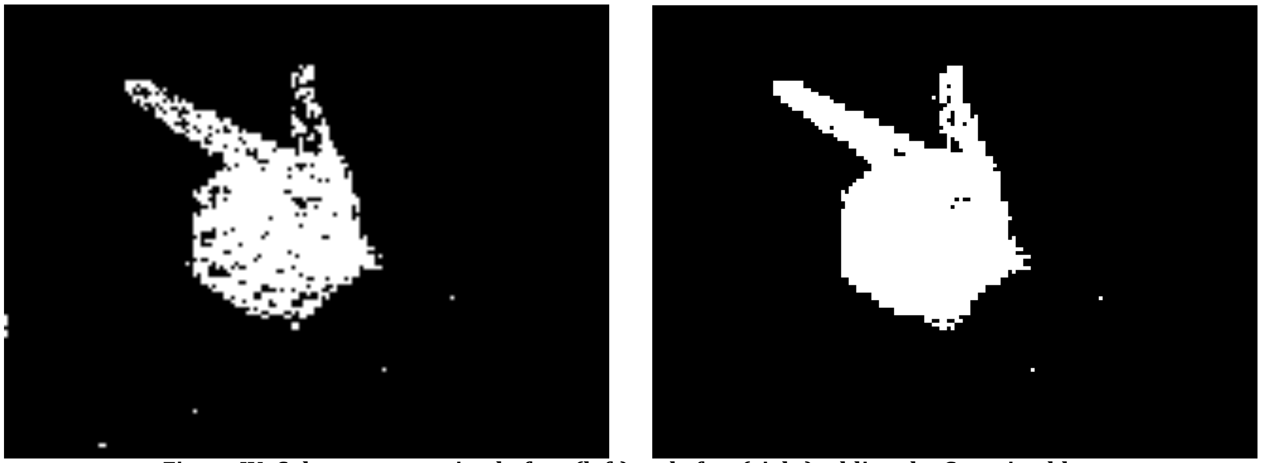
Figure IV: Color segmentation before (left) and after (right) adding the Gaussian blur

In the third and fourth meetings, we tried using all three color spaces, for the sake of complete analysis, but found after extensive testing that RGB was vastly superior in segmenting the skin in our test images. NRGB was a close second, but captured some pixel data from the shirtsleeve in pointer.avi and joy.avi, which produced unpredictable results when later performing the curve-fit analysis.

During the third meeting, we decided that our color segmentation could be further improved with a few additional techniques. Thus, we implemented the ProcessSkin() function. This function performs two operations after converting the image to a binary map. First, it uses imfill() to close all the connected components fully. This erases any of the gaps found in the original color segmentation.

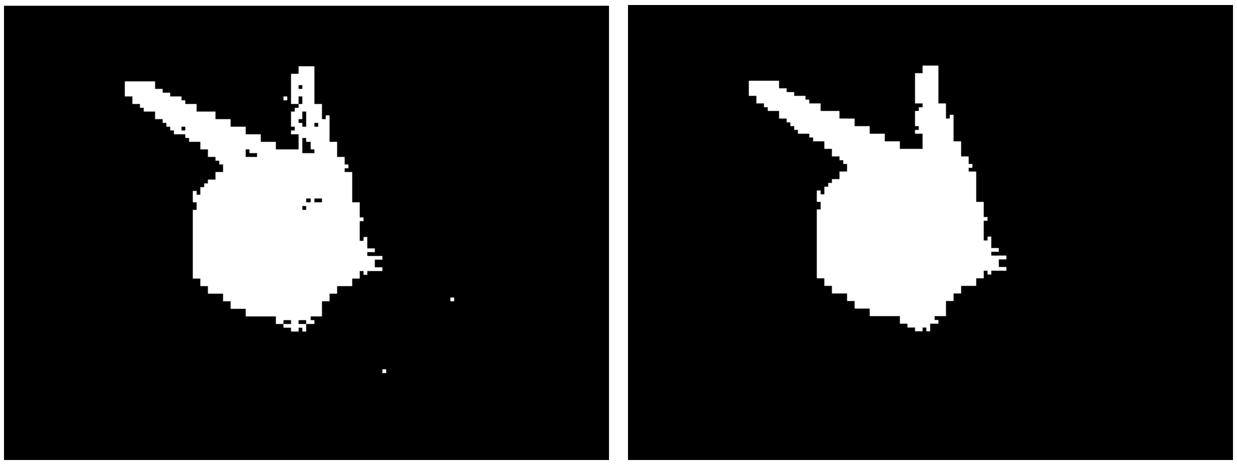
Figure V: Skin detection before (left) and after (right) the ProcessSkin() function is called

For example, in the above figure (Figure V), the gaps of black found within the hand are closed. ProcessSkin() then uses bwareaopen() to remove any connected components below a certain threshold. Through empirical testing, we decided that a test hand would never be less than 150 pixels, and set the threshold to this number. The effect of this function is to remove random scatterings of noise not removed by the Gaussian filter and color segmentation.

Next, in the fourth meeting, we worked on edge detection. Rather than use the code from MP #5, which in most cases had small, but noticeable performance issues, we decided to use the built-in edge detector for MATLAB. We came to an agreement to use a Canny edge detector, since this type of detector functioned the best for all group members in empirical tests during MP #5. In addition, since this built-in function calls external C++ libraries, it operates extremely quickly, allowing for negligible performance degradation. Thus, in the main project (FingerCursor()) method, we called MATLAB's edge(image, 'canny') function. This function provided sharp, clear edges, with limited or no noise, and suited our purposes well.

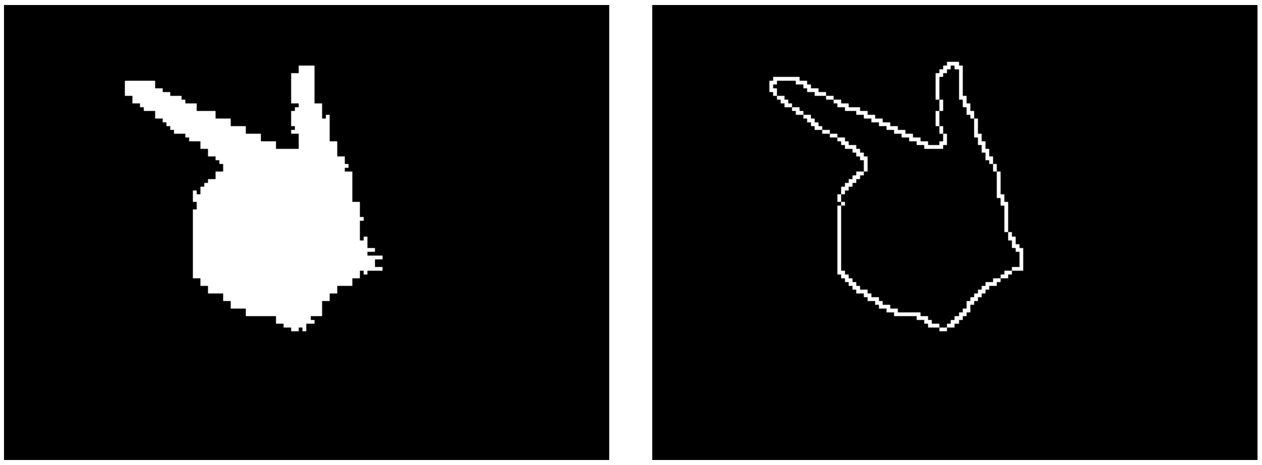

Figure VI: Image detection after skin processing (left) and after edge detection (right)

The next part of our algorithm, approximating the fingertip shape, was the longest and most arduous part of the project. This algorithm was designed and successively refined during the last three group meetings. Drawing on ideas from "A Real-time Multi-cue Hand

Tracking Algorithm Based on Computer Vision"[1], we decided to use an approximation to the curvature of the edge map to estimate the shape of a finger. However, since very large curvature values can be found outside the index finger, such as at the thumb, we elected to modify the original formula presented by the authors. Our modification also incorporates a least squares error calculation, based on a curve approximation. In essence, we model the shape of a fingertip using a curve function. For each sampled point on the edge map, we calculate both the curvature and the squared error of the point from the reference curve. Like a linear regression, we calculate the "closeness" of the fit of the finger to the reference curve, and use this, in combination with a high curvature value, to identify fingertips.

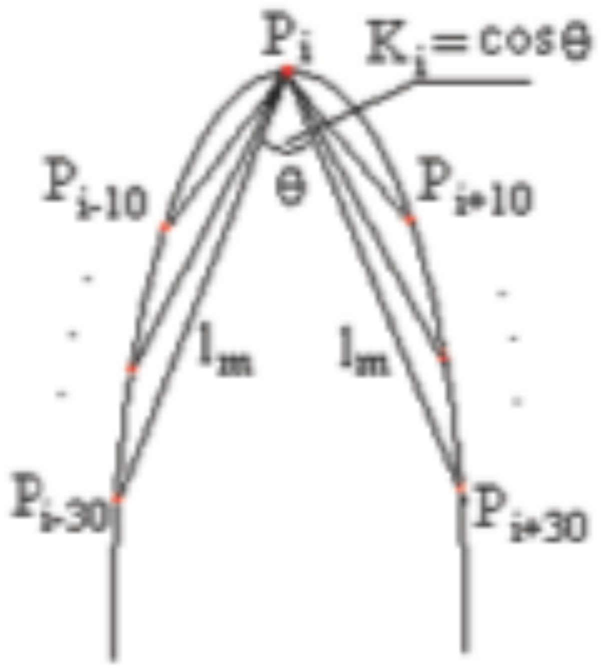

Figure VII: curvature approximation for a fingertip

Originally, the curve approximation we used was $\frac{1}{x^2}$. Starting at the fingertip, the value of θ is effectively 180 degrees. As the tip rounds out, the value of theta rapidly decreases until the vertical part of the finger begins, at which point theta decreases very gradually. Based on empirical analysis of this pattern, the curve $\frac{1}{x^2}$ was chosen, since this curve rapidly decreases from the infinite asymptote, but very slowly decreases at a certain distance from the asymptote. After testing this curve with our images, we discovered that the fit did not meet our needs. In particular, $\frac{1}{x^2}$ is asymptotic as $x \to 0$, but a finger is asymptotic in the other direction, that is, as $x \to \pm\infty$. A fingertip does not strictly form an asymptote, but instead levels out to a 180-degree angle. After a little brainstorming, we realized that the tangent function $y = \tan\theta$ approximates half of a finger, with the appropriate asymptotic relation. In addition, trigonometric functions integrate well with the use of angles, and so fit our approximation very well.

Unfortunately, $x = \tan\theta$ grows larger as it diverges from zero. Our algorithm necessitates a curve that approaches zero from infinity, rather than the other way around. To work around this issue, we used the inverse tangent function, which has an approximate fingertip shape. With this, our equation becomes $y = \tan^{-1} x$, where x is the number of the point being sampled (-30 to 30), and θ is the vertical position of the point on the approximation curve. Solving for the variable we calculate, θ, the equation becomes: $\theta = \tan^{-1}\frac{\tan^{-1} x}{x}$. This is the curve approximation used in our final approach. As a final touch, points further away from the fingertip are given lower weights, as they are not as important in determining a fingertip shape as the asymptotic points near the tip.

In the fifth meeting, we discovered that random noise and other issues caused by poor image quality were causing glitches in the edge map. While normally innocuous, these glitches sometimes produced a sharp point with extremely high curvature, which in turn stole away cursor focus from the fingertip. Some of these issues were resolved with Gaussian blurring and better sampling, as well as adjusting the color space. However, significant errors persisted despite our best efforts at reducing them. Because of these errors, we decided to improve our fingertip approximation by using the centroid of the hand.

The centroid is the center of mass of a closed, connected component in the image space. Using MATLAB's built-in imregionalprops() function, we were able to locate and track the centroid of the hand in each frame with relative ease. This function provides a centroid subroutine, which determines the geographic center of mass of all nonzero pixels in an image. We provided a wrapper around this function, called FindCentroid(), which processes an individual video frame and returns the (x, y) coordinates of the image centroid.

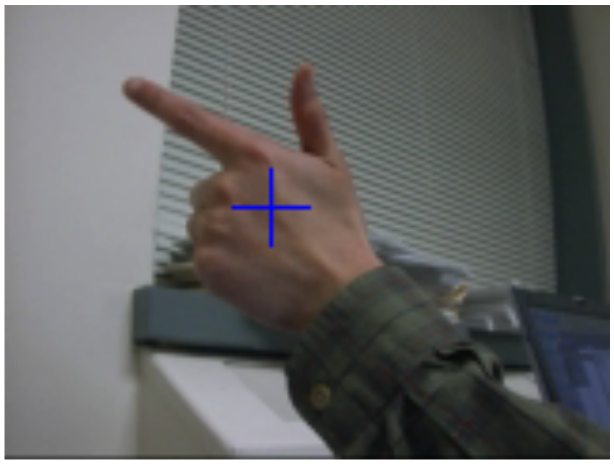

Figure VIII: Frame one of the gun.avi file with the centroid calculation displayed

The use of this function was agreed upon after we recognized that the index finger, when extended, would always be farther from the centroid of the hand than any other point, except for other fingertips. Thus, by incorporating a distance metric into our approximation, we could more robustly track the index fingertip, ignoring noise, image quality, and perhaps most importantly, the thumb, which was causing persistent issues due to its similar shape. By measuring the distance from the centroid to our candidate points of high curvature, we were able to robustly detect the fingertip in every frame of gun.avi and pointer.avi, and a high fraction of frames in joy.avi.

Using this curve approximation, along with the ideas we researched from previous works, we created a fingertip detection approximation algorithm, which works in the following way:

1. For a given sample point “b”, get the coordinates of the immediately preceding point, “a”, and the immediately succeeding point “c”. (We used the range -30 to 30, as used in the paper. Interestingly, the length of the fingertip in the gun.avi test image is roughly 30 pixels.)
2. Calculate the line segments $\overline{ba}$ and $\overline{bc}$.
3. Calculate the angle between $\overline{ba}$ and $\overline{bc}$ with the formula: $\theta = \frac{cos^{-1}(\overline{ba} \cdot \overline{bc})}{|\overline{ba}|\,|\overline{bc}|}$
4. Calculate the deviation from the approximation curve $\tan^{-1} \frac{\tan^{-1} x}{x}$
5. Weight the deviation based on the distance from the tip (farther values get lower weights, as finger tips have rapid curvature and rapid asymptotic behavior)
6. Calculate the mean-squared error from all the sampling points
7. Choose the points with the smallest mean-squared error
8. Weight the points with the smallest mean-squared error based on their distance from the centroid (larger distances get larger weights). The distance, in this case, is

calculated by the formula: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, where ($x_1$, y_1) is the location of the current candidate point, and (x_2, y_2) is the location of the centroid.

9. Return the point with the lowest mean-squared error and largest distance

Initially, we ran into a problem with our sampling curves, but this was due to a logic error. Due to a mistaken calculation, we were returning points with the maximum mean-squared error, rather than the minimum. Once this issue was sorted out, our code functioned as expected, and correctly identified the fingertip in every frame of gun.avi and pointer.mp4, with some error due to noise in joy.mp4.

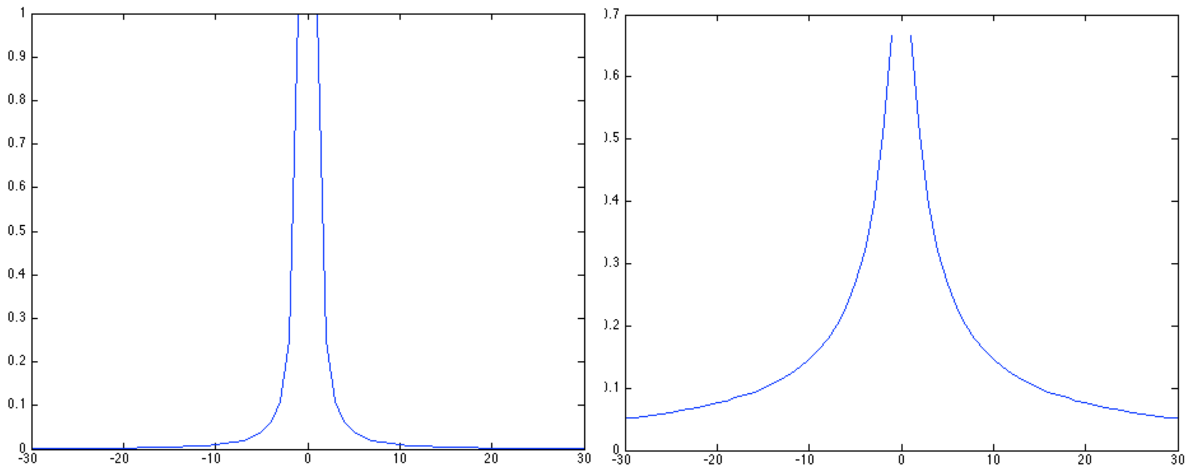

Figure VIII: The initial curve approximation of 1/(x^2) (left) and our final approximation $\tan^{-1}(\tan^{-1}(x) / x)$ (right)

At this point in the process, the calculations for an individual frame are completed. All that remained was drawing the cursor position on the frame, then collecting and processing the remaining frames in the video into an output file. This was our main task during the fifth and sixth meetings, though we had rudimentary code for performing this operation that we had written for testing purposes while developing previous stages of the project.

The output stage begins when the cursor position, returned by FindJustTheTip(), is drawn on the screen by flipping pixels in the frame to the RGB value (255, 0, 0). Then, the frame is either collected into an array (as in FingerCursorFrames()), or appended to a video file (as in FingerCursor()). The array or output file is then played.

Interestingly, when collected into an array of frames, the output cursor (as well as the centroid and edge map, when debugging) is considerably clearer and higher resolution than when appending to a video file. We are not sure what causes this degradation but suspect it is linked to the compression algorithm used in MATLAB's VideoWriter() function.

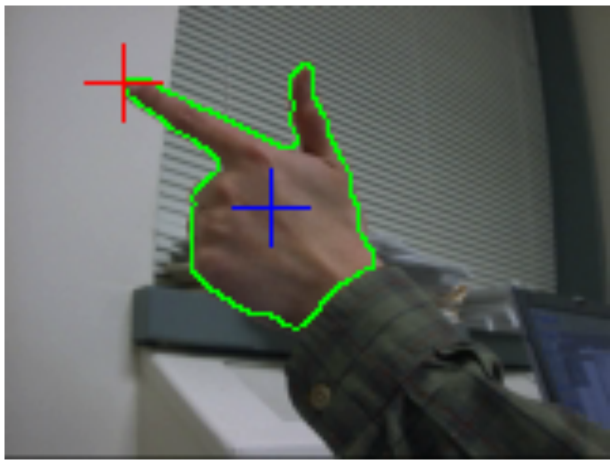

Figure IX: Frame one of the gun.avi file with debugging parameters set, displaying the centroid (in blue), finger tip detection (in red), and edge map detection (in green)

IV: RESULTS

In general, our results were quite impressive. On the gun.avi file, our detection algorithm is 100% accurate, displaying a point on the tip of the index finger in every frame. This result was pleasing, but unsurprising because we used the gun.avi video extensively for testing. In addition, the pointer finger is heavily contrasted with the background of the frame, and the pointer finger is easily recognizable in this video. In pointer.avi, a few mistakes are made from noise on the sleeve and bottom of the hand, but the pointer finger is again correctly identified in greater than 98% of the frames. The glitches here are likely due to the low image quality of the file, as well as confusion from the sampling of skin fragments in poor lighting. Finally, for joy.avi, our results are decent, but not quite as accurate as the previous two videos. Because of the lack of a clear distinction between fingertips, our program usually identifies the tip of the middle finger on the hand, rather than the index finger. This finger is still detected with great accuracy: at or above 95%. However, a smarter algorithm is necessary for distinguishing between unclear fingertip boundaries. In addition we may need to alter the color space or collect better training data in order to eliminate issues with noise from non-skin pixels.

We did not get a chance to test our algorithm with other test videos, or skin training data. Time constraints prohibited a deeper analysis of the algorithm with various test videos and skin tones. We did attempt to integrate a webcam demo of our program into the final presentation. Unfortunately, MATLAB only provides this functionality with the Image Processing Toolkit, which is unavailable on the student license. Still, it would be interesting to test the algorithm on different videos, especially ones featuring different skin tones, and having real-time webcam functionality would make this testing readily available.

Overall, I am quite proud of our design. The algorithm is accurate and robust, as requested in the problem description. Fingertips are identified with great accuracy across the testing videos, and I am confident that our program would produce similar results in any test video of acceptable resolution and lighting. One potential issue with our code is the performance – it takes around 20 seconds to compile the result for the joy.avi and pointer.avi videos. Obviously, this is unacceptable for a real-time demonstration or application. It should be noted that on a frame-by-frame basis, this amounts to about 66 milliseconds per frame. With some performance modifications, we could probably reduce this to an acceptable level, say 10 milliseconds per frame, and enable real-time compilation. Performance gains like this could be made through the use of external C++ files, and algorithmic restructuring (including changing our tight and computationally expensive thresholds).

V: REMARKS AND FUTURE WORK

The main areas for improvement and future work that could be applied to our project are as follows:

- Better approximation curve for the shape of a fingertip
    - Our current curve, while close to the shape of a fingertip, is far from perfect. With a better approximation equation, especially one that takes into account the perspective of a fingertip from different distances, our model would be significantly more accurate. In addition, with a more sharply defined curve model, we would need fewer samples to accurately detect a finger, lowering the complexity of our most computationally intensive code block and the total runtime of our function.
- More skin sampling
    - Our current training data comes from sampling the first frame of our test videos. Obviously, this data is not sufficient for generalized analysis of skin data. In order to

generalize the application of our code, we would need significantly more skin samples across a variety of skin tones, lighting conditions, and image qualities.

- More accurate sampling of skin
  - The roipoly() function works well for gathering skin samples, but our histogram normalization and color segmentation code could be further improved. While the RGB color space works fairly well, as aforementioned, the other color spaces, NRGB and HSI, are unusable for pointer detection. Through refined sampling techniques, we could attempt to use these color spaces, which may provide more accurate results.
- Distinguishing of individual digits
  - Our code as it stands is able only to pick the most prominent digit feature. However, the algorithm breaks down when multiple fingers are present, as can be seen in the joy.avi test video. A new algorithmic technique could be devised that would allow for the distinguishing of individual digits, allowing us to correctly identify and mark only the index finger, ignoring the relative "noise" of other fingers or other prominent features.
- Performance optimizations
  - As mentioned before, the performance of our code is too slow for a practical real-time application. Serious performance considerations need to be made before our algorithm could be realistically used for mouse detection; in fact, high-resolution mouse control via this method would likely never be possible due to the tremendous overhead of calculation. This is unsurprising, as hardware mice still

provide vastly better performance than touchscreens, which in turn are orders of magnitude less complex than an image detection scheme.

- Gestures
    - As aforementioned, an optional goal for this project was to implement complex gestures for the finger cursor. This was a goal we had planned to implement but were unable to due to time constraints. An interesting exploration in future work would be to implement some complex hand or finger gestures. For example, we considered the possibility of detecting two rapid, successive swipes from right to left, and interpreting this as a "back" function in a browser. Similarly, gestures could be setup for a left-to-right swipe (meaning forward in the browser), and a top-to-bottom and bottom-to-top swipe (meaning scroll in most programs). There are many other possible gestures, but these are the simpler ones that we considered implementing.

With regard to the course, I thought the course material was covered in an appropriate level of depth. All of the relevant topics were covered through the textbook and relevant examples in class. One area on which I would have enjoyed a more in-depth analysis is face detection. Since we only spent part of one lecture on it, I felt that we didn't fully explore the possibilities and ramifications of this particular sub-field. In addition, more information on face detection could provide for more interesting term projects, as well as projects that have real-world value, as face detection is a valuable, emerging field.

VI: COURSE FEEDBACK AND SUGGESTIONS

Generally speaking, I very much enjoyed this course and its respective content. Learning how to implement important image analysis techniques was very interesting, and a very welcome addition to my skillset as a programmer. I can think of two main suggestions for improvement of the course:

1. Spread out the machine projects in a more coherent schedule. Having two machine projects due in the same week can often be a burden, even if the projects are fairly simple. By starting the projects off early, and eliminating large breaks (like the one between MP #4 and MP #5), the schedule would be much easier to cope with, especially if the student in question is taking a variety of project-oriented courses.
2. The class lectures rely heavily on the content in the textbook. While it is good to cover the concepts of the book in more detail, especially to resolve confusion and reinforce key concepts, I felt like the lectures relied on the book material a little too heavily. Instead, it would be nice to see some of the more basic concepts summarized and cut down in lecture time, and replaced with real-world examples of the techniques being discussed. For example, an example of panorama stitching, as is provided on apps and natively in many smartphones, would be a welcome addition to the lectures.

All in all, EECS 332 – Digital Image Analysis was a thought-provoking and interesting class, which I would recommend to any undergraduate student in computer science.