Optical Verification of Mouse Event Accuracy

Denis Barberena
Department of Electrical Engineering
Stanford University
Email: denisb@stanford.edu

Mohammad Imam
Department of Electrical Engineering
Stanford University
Email: noahi@stanford.edu

Ilyas Patanam
Department of Electrical Engineering
Stanford University
Email: ilyasp@stanford.edu

***Abstract*—Optical verification of mouse movements can independently verify the integrity of mouse event data without the overhead of software instrumentation. We propose a methodology to accurately validate mouse sensor values utilizing object-tracking techniques on physical mouse movements. By capturing mouse movements with a camera, applying morphological image processing operators, and a probabilistic Hough transform, we demonstrate a proof of concept that compares the mouse movement observed by the camera verified by mouse event data captured on a computer. Our software utilizes OpenCV libraries and can run real time on an Android device or on a computer for post-hoc analysis. Potential use cases for tracking mouse data includes verifying sensor problems, for example, poor tracking or skipping and calibration of a mouse. This data can also be used to determine the validity of mouse event data as an input to various software anti-cheat methods.**

I. Introduction

During our research on tracking mouse movements using a camera, we considered various object tracking algorithms including key-point detection, kernel-based tracking, contour tracking, and optical flow. Unlike traditional applications of these algorithms, our goal was to focus on providing an accurate representation of small mouse movements which we have deemed as less than 15 pixels on screen. The aforementioned methods did not provide sufficient accuracy or consistent repeatability. As we refined our approach, we iterated through multiple tracker designs, various edge detecting algorithms, and morphological operations. Based on constraints and processing capability of OpenCV on an NVIDIA Shield Android tablet, we developed a fast real-time mouse tracking algorithm using a sequence of morphological image operations and color mapping techniques.

II. Related Work

Various video object tracking methods exist from optical flow to multi-object tracking; mechanisms for tracking both marker-based tracking and marker-less feature based tracking [1] and many others. Based on our findings, we noticed that camera based mouse tracking was a unique objective and a previously unexplored topic altogether. The underlying premise of object tracking is to find the displacement of a pixel (ideally, a window of pixels) from one frame to the next. One of the main design challenges we faced was distinguishing between very minute movements and input jitter due to camera noise. With a key-point detection based approach, we noticed during random sample consensus (RANSAC), the rotation angle determined in the homography matrix had too large of a variance to meet our requirements. We attempted multiple edge detection algorithms, including Canny edge detector, however, these methods were not optimized computationally to find an edge around a rotated rectangle[1]. Tracking a particular pixel in the image is not robust because pixel values can alter due to noise and interpolation with adjacent pixels [2]. Based on our tests, a morphological gradient provided an acceptable contour without adding substantial computational overhead.

III. Technical Approach

A. Software Structure

1) Capture mouse events using HTML/JavaScript: We wrote a script to capture mouse events data, which is a list of the (x, y) coordinates of the cursor on the screen. For post processing, we developed a script to convert the point data to a normalized center and overlay the points as vector graphics.
2) Dynamic Calibration using OpenCV -Determine the relative size of the tracker with respect to input frame and dynamically generate set of initial values to optimize our Morphological Image Processing library
3) Morphological Image Processing using OpenCV - Library of functions implementing the processing algorithm we use to determine the mouse sensor location for a particular frame.
4) Java Post-processing framework - Apply our algorithm on prerecorded video utilizing our processing library on a quad core computer.
5) Android Application with OpenCV run-time - real-time application utilizing both the Calibration and Morphological Image Processing libraries with lower fidelity output than the Java counterpart.

B. Data Acquisition and Scaling

We utilized a browser based JavaScript mouse event listener to capture mouse movement locations at 60 hz to collect sample data. Simultaneously, we capture the physical mouse mouse movement with a 30 frames per second (FPS), 5 mega-pixel (MP) camera. Using the first 30 frames, we determined the absolute starting coordinates which would be the same for both the mouse events and the camera output irrespective of the scaling factor of the mouse sensor. Since mouse events are captured on an orthogonal X-Y plane, we determine an initial rotation for our video to correct for the camera orientation. For our real-time method, the frame-rate drops to 15 FPS due to overhead of processing each frame via the Java Native

Interface for OpenCV on Android OS. Using the set of points collected from our video, we determine a scaling factor in both the X and Y plane based on the ratio of differences between two subsequent points from our image processing algorithm and difference between subsequent mouse events. We approximated the scaling factor at 400 dots per inch (DPI) to be 7 on the X direction and 7.5 in the Y direction.

C. Tracker Design

Throughout the course of the project, we developed three trackers, improving the accuracy of our algorithm without restricting movement or usability of the mouse. The first tracker consisted of a white rectangle taped to where the wire attaches to the mouse. Although placing the tracker on the mouse was minimally invasive, the data from this setup introduced a phase shift between the rotation of the mouse and the rotation of the tracker. Additionally, due to the difficulty in isolating a white tracker in varying light conditions we needed numerous morphological operations which not only increased the computational requirements but also denatured the orientation of the tracker when identified. In our second iteration, we were inspired by the idea of chroma keying to build a green tracker since it would have less overlap with skin tones. Similar to our setup from the first iteration, we placed a tracker on the wire, but we also put a smaller tracker on the mouse directly above the scroll wheel, free from obstruction from the user's hand. The larger tracker was used to determine the angle of rotation while the smaller tracker was used to subtract the offset and determine the sensor position.

Drawing Hough lines on the small tracker to determine the angle of rotation was inaccurate especially after morphological processing since multiple angles in the original frame were quantized to a single angle. During this iteration we threshold in the RGB color channel using fixed values to isolate the green channel. In our final iteration, we attached a single rectangular tracker above the scroll wheel. Although the tracker was smaller than the previous, it had more consistently identifiable lines and a predictable height and width ratio. We found the Hough Line detector is susceptible, on some frames, to detect only segments of the outline, so we calculate the angles only using lines equivalent to the longest side. The Hough lines rely on a fixed ratio between the width and the height to categorize their location.

D. Chroma Keying

Initially we used a set of brute forced thresholds for the RGB channels during post processing to identify the tracker. Although effective, this approach is computationally intensive and not viable real-time. Instead, we developed a faster method by converting our images to the HSV color space [3]. HSV color space separates the color information from luminance by using the H and V channel, respectively. Our approach converts the RGB image to HSV values, thresholds on the HSV channels, and returns a binary mask used in later processing steps. In OpenCV, we found an appropriate Hue range [0, 179], Saturation range [0,255] and Value range [0, 255]. Green chroma key in the RGB channel, (0, 255, 0), converts to HSV values of (60, 255, 255). The saturation channel represents the purity of an image, and below a certain threshold regardless of the hue, the color will be perceived to be a shade of gray. After experimentation with various lighting conditions, we set valid saturation values between 25 and 255. For saturation values less than 25, we would detect gray artifacts outside the tracker area, but an upper limit was unnecessary since no other pixels saturated the green channel in the image frames. We set a large range for our chromaticity channel H because we determined that the lighting source alters the chromaticity value of our tracker. This issue likely stems from white balancing inaccuracies and focal length operating in low light conditions of the camera [4]. To adjust for the white balancing issues we set a luminosity V dependent range for H. The maximum and minimum values for the hue range, H_{max} and H_{min} would decrease dependent upon the maximum luminosity value as detailed in the below equations:

$$H_{max} = 57.0 - (255 - V_{max}) * .08 \tag{1}$$

$$H_{min} = 90 - (255 - V_{max}) * .08 \tag{2}$$

When setting the limits for the luminance channel, we realized that when the lower limit of the range, V_min, is too high, we will not detect the green tracker in dark environments, but if the value is too low we detect regions that is perceived as black. After experimenting with images in various lighting conditions, including maximum luminance value of 255, we found a lower limit of 130 captures our tracker with minimal artifacts. Upon processing of images where the maximal luminance value was below 255 we determine the V_{min} to be set according to this equation:

$$V_{min} = V_{abs_max} * .55 - 7. \tag{3}$$

We fix the upper limit of the range, V_{max}, to 255 as we find this to be suitable in all conditions.

E. Dynamic Calibration

The purpose of our calibration library is twofold. The first is to have the ability to dynamically set parameters to ensure the tracking software is depth and light invariant and the second is to decrease the computational complexity of our tracking library so it can run real-time on an Android device.

We concluded that converting an image from RGB to HSV to threshold the tracker color increases the run-time of our algorithm by a factor 10. To eliminate this processing step, our calibration library performs the slow HSV transform to identify corresponding range of RGB values in the first 30 frames. Subsequent frames will rely on the RGB range generated, avoiding the performance penalty. After using the HSV limits as described above to create a mask to isolate the tracker we apply erosion with a square structuring element of size 4 to erase artifacts in the mask. We then use the mask on the original image to identify the minimum and maximum value for each RGB channel. As we process each frame we keep a running average of each of the RGB limits. We will

use these limits to threshold the image in the RGB channel and extract a mask of our tracker during real time processing.

Using the known ratio of the length of the mouse tracker to distance of mouse sensor from the tracker, we also determine the distance in pixels from the center of the tracker which represents the location of the mouse sensor. Additionally, we record the initial frame rotation with respect to the camera and a region of interest (ROI) to perform our morphological operations. Although the calibration step itself is as computationally intensive as our Java testing framework, the subsequent processing limited to the region of interest reduces the asymptotic computation time from 500 ms per frame to about 10 ms per frame on a quad core processor. This computational optimization allowed us to implement the real-time system on the Android device.

F. Sensor Point Acquisition

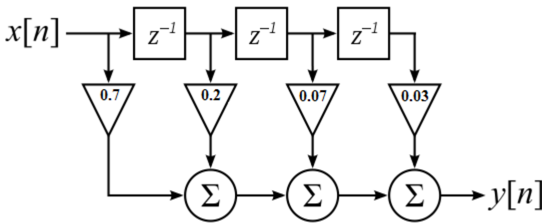

Fig. 1. FIR Angle Filter

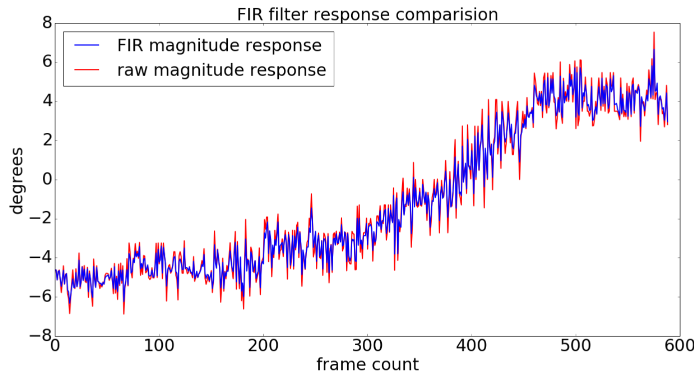

Fig. 2. Note the filter attenuates high frequency noise, sharp changes in the angles due to quantization errors.

After applying a color threshold to the ROI (determined by the parameters gathered during the calibration), resulting in a binary image, the first step in the processing algorithm applies a dilation with a 3x3 square structuring element. This removes any remaining holes in the tracker and quantize the edges such that we can perform a morphological gradient (Dilation - Erosion) to obtain the contour of the tracker [5]. Since the location of the mouse sensor is orthogonal to the short side of the tracker, we apply a Hough transform on the contour to determine the angle of rotation. Using a scaled value of the longer side of the tracker, we approximate the location of the mouse sensor relative to each image frame. Note, our input data resolution quantize a single pixel movement in a frame to 7 pixels in the mouse event data. Thus, our system is sensitive to noise and quantization from the 5 MP camera in low lighting conditions. To mitigate this impact, we apply a continuous 4 input FIR filter to the angle of rotation and a 5 input FIR filter to the dy and dx translations.

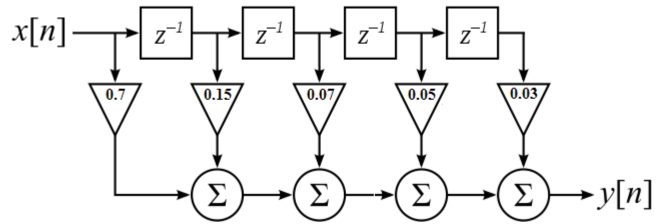

Fig. 3. FIR Displacement Filter

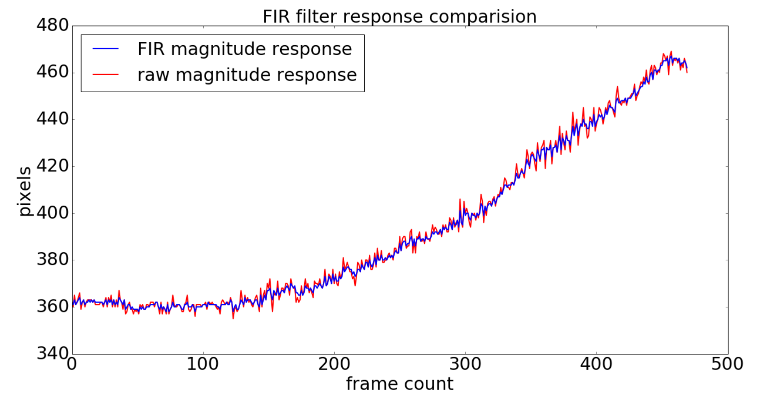

Fig. 4. Note the filter attenuates high frequency noise, multiple pixel displacement between subsequent frames.

G. Processing Steps

The following images show the outputs from the various stages of our algorithm.

Fig. 5. Calibration - The initial calibration step determines the location of the tracker using the HSV channel and a bounding box around the tracker which the subsequent frame will use as the region of interest.

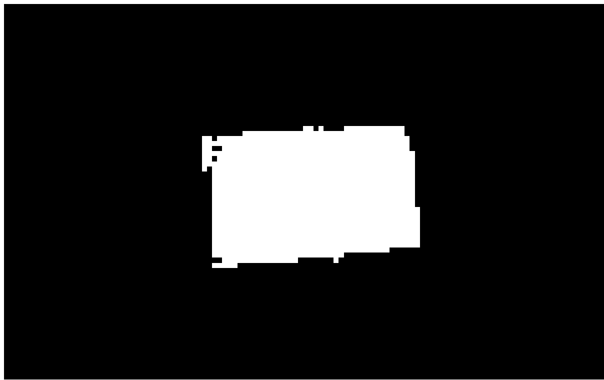

Fig. 6. Threshold in RGB channel - Output after thresholding the image using the limits determined during the initial calibration.

Fig. 7. Dilation - Result after dilating the image with a size 3 square structuring element to fill in any holes in the tracker.

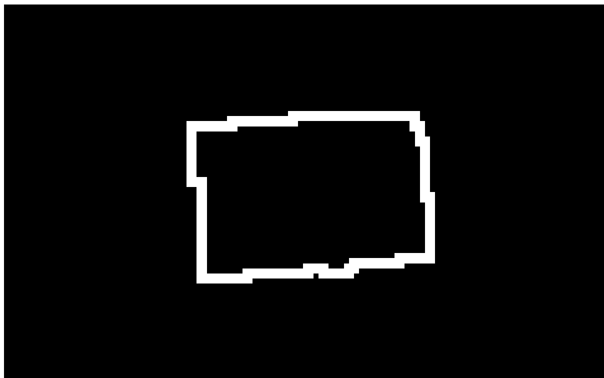

Fig. 8. Edge Detection - Applying a morphological gradient, which subtracts the erosion from the dilation, extracts an outline of the tracker. We compute the centroid of the tracker using the moment of the object.

Fig. 9. Probabilistic Hough Transform - Calculates the tracker rotation, used to normalize the tracker and compute the offset from the centroid to the mouse sensor, 4.5*[long side of the tracker].

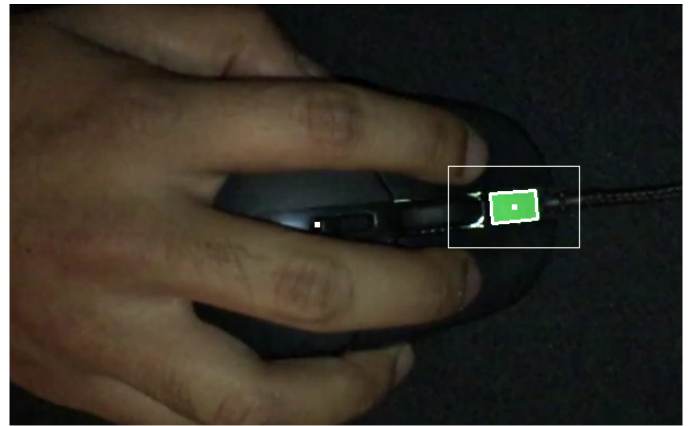

Fig. 10. Final Detected Image - Superimpose output of each processing step in the region of interest. The region of interest shifts by the detected dy, dx change in the mouse sensor for the next image. The white dot in the middle of the tracker marks the centroid of the tracker object. The other white dot, near the center of the image, represents the location of the mouse sensor.

IV. Experimental Results

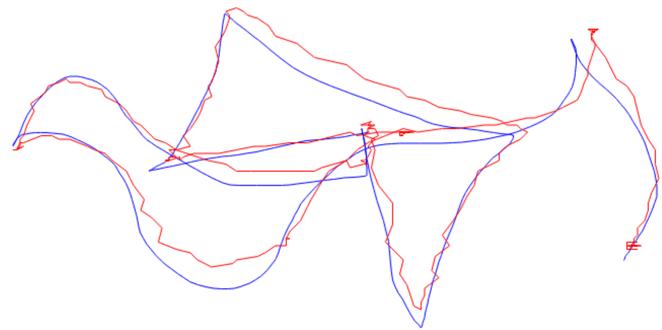

Fig. 11. Mouse event and camera frame tracking vectors - Generated from the HTML/JavaScript code which normalizes the starting points for both plots and adjusts the position based on the scaling factor.

In figure 11, the blue vector represents mouse event captures and the red vector represents the approximation from our image processing algorithm. With respect to trajectory and rate of change differences in this data set, the average error in our algorithm is less than 15 pixels which meets our initial goal. Based on the curve trajectory, our data closely resembles the peaks, valleys, and flat areas reported by the mouse. If we were to apply this method to analyze discrepancies we would look for areas in the blue vector that deviate a certain threshold and angle away from the camera data suggesting a (possibly malicious) change in the mouse event data somewhere in the software stack.

V. Discussion and Future Work

Our initial results demonstrate a proof of concept to accurately detect mouse sensor values from physical mouse movements captured using a camera. Between our tracker iterations and algorithm optimizations, we reduced the margin of error and the complexity of the image processing enabling real-time

processing on an Android device. Some of the challenges we faced include camera resolution and frame capture rate. An optical mouse sensor can be polled at up to 1000 Hz using mouse events compared to our post process data captured at 30 Hz and 15 Hz real-time, which remains a major culprit of our error rate. In order to further improve the accuracy, our algorithm would greatly benefit from a higher resolution camera and higher frame capture rate.

We propose the following extensions for future work: analyze tracker scale changes to determine mouse lift (the physical movement of the mouse without changing mouse event data) and auto-correction by injecting mouse event data periodically as a key-frame to correct for the differences caused by noise, resolution quantization, and data filtering of our algorithm.

References

[1] John Canny. IEEE Transactions on Pattern *A Computational Approach to Edge Detection*

[2] Carlo Tomasi, Takeo Kanade. *Detection and Tracking of Point Features* https://cecas.clemson.edu/ stb/klt/tomasi-kanade-techreport-1991.pdf

[3] Shamik Sural et al. *Segmentation and Histogram Generation Using the HSV Color Space for Image Retrieval* http://www.cse.msu.edu/ pramanik/research/papers/2002Papers/icip.hsv.pdf

[4] Michael S. Tooms. *Colour Reproduction in Electronic Imaging Systems: Photography, Television, Cinematography* 2016

[5] Na Wang, Guo-Yu Wang. *Shape Descriptor with Morphology Method for Color-based Tracking* International Journal of Automation and Computing , 2007,V4(1): 101-108