

Seeing through the Fog: An Algorithm for Fast and Accurate Touch Detection in Optical Tabletop Surfaces

Christopher Wolfe, T.C. Nicholas Graham, and Joseph A. Pape
 School of Computing, Queen's University, Kingston, Ontario, Canada
 {wolfe, graham, pape}@cs.queensu.ca

ABSTRACT

Fast and accurate touch detection is critical to the usability of multi-touch tabletops. In optical tabletops, such as those using the popular FTIR and DI technologies, this requires efficient and effective noise reduction to enhance touches in the camera's input. Common approaches to noise reduction do not scale to larger tables, leaving designers with a choice between accuracy problems and expensive hardware. In this paper, we present a novel noise reduction algorithm that provides better touch recognition than current alternatives, particularly in noisy environments, without imposing higher computational cost. We empirically compare our algorithm to other noise reduction approaches using data collected from tabletops at research labs in Canada and Europe.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Input devices and strategies.

General terms: Human Factors, Algorithms, Measurement

Keywords: Optical tabletop input, frustrated total internal reflection, FTIR, diffuse illumination, DI

INTRODUCTION

Optical multi-touch tabletops, such as those based on frustrated total internal reflection (FTIR) or diffuse illumination (DI), use an infrared-filtered camera to track touches. Significant image processing is required to detect touches and objects in the raw camera input. This detection needs to be extremely accurate: spurious touches will cause the user interface to misbehave, while missed touches will annoyingly interrupt gestures. The processing must also minimize latency, or users will perceive lag when manipulating objects.

The current generation of input processing libraries provides good support for fairly small surfaces with powerful computers. A modern desktop computer with either CCV [2] or reacTIVision [7] processes input from a typical 640x480 camera at 30-50 frames per second, while still executing interesting applications. This is adequate for small non-portable surfaces, similar in size to the existing SMART Table and Microsoft Surface.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ITS'10, November 7-10, 2010, Saarbrücken, Germany.

Copyright 2010 ACM 978-1-4503-0399-6/10/11...\$10.00.

Emerging applications stretch the limits of these approaches. Bringing optical tabletops to large-scale simulation, design, and command and control requires larger and vastly higher-resolution tables [14]. Our experiments show that such scenarios exceed the capabilities of commodity processors. The resulting slow input processing leads to sluggish response times. Moreover, requiring extensive CPU resources for touch detection leaves less processing power for the user's applications. Clearly, faster input processing is required.

The largest computational task in input processing is noise reduction, where the camera image is transformed to make touches more visible. Ideal camera input would consist of a black background with a white blob corresponding to each touch. In practice, the infrared light detected by tabletop cameras can also come from other sources, such as overhead lights and sunlight. This extraneous light is termed noise, and must be distinguished from useful signal in the input images.

In this paper, we present and evaluate the *Dual Quad* (DQ) noise reduction filter. This filter is applied using a novel fast algorithm for piecewise-quadratic image filters. Our approach offers a combination of speed and accuracy unmatched by current techniques.

We have evaluated the DQ filter using input videos from tables at five laboratories in Canada and Europe. Specifically, we have extended *EquisFtir* to compare our approach to common filters from the image processing literature, and also tested the popular CCV [2], *Touchlib* [11] and *reacTIVision* [7] libraries. As such, this is the first paper to comprehensively evaluate the accuracy and performance of input processing libraries for digital tabletops. Our experiments show that the DQ filter provides improved accuracy without sacrificing speed. The DQ filter is detailed in this paper, implemented as part of our open-source *EquisFtir* [16] library, and could easily be added to other libraries.

To summarize the contributions of this paper:

- We introduce a novel algorithm for noise reduction, which offers a unique combination of speed and increased touch detection accuracy.
- We describe a method, based on signal detection theory [9], for comparing the accuracy of image processing algorithms underlying tabletop input libraries.
- We have performed the first comprehensive empirical comparison of input processing libraries for optical tabletop surfaces.

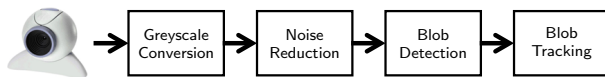


Figure 1: Software pipeline for detecting touches on camera-based tabletops.

The paper is organized as follows. First, as background, we examine the image processing performed by current input libraries for optical tabletop surfaces, including their approaches to the noise reduction. We then introduce the DQ filter, our novel noise reduction algorithm. Finally, we present the results of our experimental assessment of various noise reduction algorithms.

BACKGROUND: TABLETOP INPUT LIBRARIES

Many multi-touch tabletops, include commercial systems, are based on optical technologies. The dominant approaches are frustrated total internal reflection (FTIR) and diffuse illumination (DI). Some commercial examples are the SMART Table, which uses FTIR, and the Microsoft Surface, using DI. Both approaches flood the translucent table surface with near-infrared (IR) light. This light illuminates fingers or objects touching the surface, producing a bright spot when viewed in the IR spectrum. An IR-filtered camera is used to observe the surface, generating a sequence of images over time. The task of an input library is to find bright regions in these images, and use them to reconstruct the user's actions.

Numerous input libraries have been developed. The most popular are CCV [2], Touchlib (CCV's predecessor) [11], and reacTIVision [7].

These input libraries typically provide an event stream describing the user's interactions with the table. With simple touches, these events consist of: *presses*, when the user places a finger on the table; *moves*, when the user drags their finger over the surface; and *releases*, when the user lifts their finger. Some systems, like reacTIVision, can identify objects placed on the table, and extract their orientations. Applications or higher-level libraries then recognize gestures (e.g. for zooming and rotation) based on these low-level events.

Accurately detecting touches is critical to the usability of these tables. Even small error rates can be annoying to users: reporting touches when no touch was present (called "false alarms") can lead to unexpected application behavior, while missed touches can lead to interrupted gestures.

Reliable tabletop libraries are difficult to construct, because they must ensure low latencies while dealing with noisy input. Sources of noise include:

- **Ambient IR light:** IR light is produced by many external sources, most noticeably the sun and incandescent lights. If not removed, this extraneous light can obscure actual touches and generate false touches.
- **Camera noise:** Commodity cameras modified for IR sensing introduce significant random variation in pixel values. Many cameras, particularly those communicating over USB, perform lossy compression of the video, adding artifacts that further obscure actual touches.

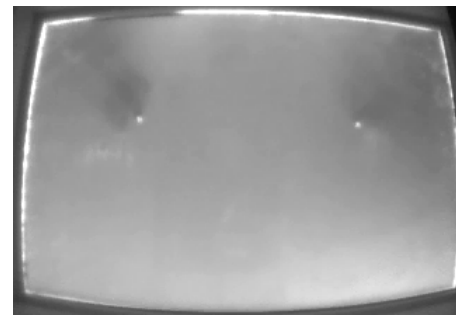


Figure 2: Example of raw camera input. Two touches show as bright white spots, with nearby shadows.

- **Shadows:** As users move their hands and objects over the table surface, they can produce shadows in the ambient IR light. These rapid changes in the background are more difficult to exclude than stable environmental light.

These sources of noise can be reduced with more expensive equipment (particularly the camera), and professional manufacturing processes. For example, the FTIR surface of the SMART Table contains a layer which blocks most ambient IR light. This absolute block is not possible with tables that use DI, so they may use pulsed light sources or additional cameras (like the Microsoft Surface). All of these modifications increase the price and complexity of the table, often far above that of commodity components. As a result, both commercial and lower-cost non-commercial tables benefit from a improved noise reduction algorithms.

By examining the source code of CCV, EquisFtir, Touchlib and reacTIVision, we have extracted a general description of their image processing pipelines. As shown in Figure 1, this consists of four image processing steps: *greyscale conversion*, *noise reduction*, *blob detection* and *blob tracking*. The following represents the first comprehensive description of how this pipeline is implemented in current input libraries.

Phase 1: Greyscale Conversion

Many commodity cameras produce a color image, with IR light appearing as white. Because these cameras sense each color separately [10], combining color channels when IR-filtered reduces the effective camera noise. All of the examined libraries use a standard luminance conversion based on human perception of color. When the camera is greyscale, or provides a greyscale format, this phase can be omitted.

Phase 2: Noise Reduction

The greyscale image typically contains far more than light produced by user touches. Figure 2 shows an example image captured on an FTIR table, with two touches. In addition, the image contains lighter-colored speckles and smudges, as well as bright and dark regions. Everything in the image other than the touches is termed *noise*. The presence of noise increases the difficulty of accurately detecting user touches.

The goal of noise removal is to simplify the image so that areas where the user is touching the table are close to white, and the rest of the image is close to black. Two steps are commonly used for noise reduction: *background subtraction* and *image filtering*.

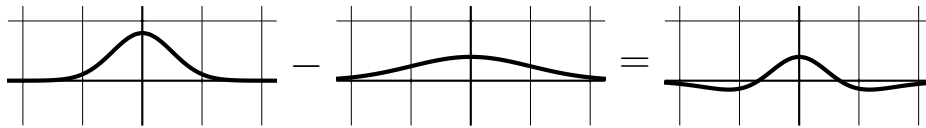


Figure 3: A Gaussian mid-pass filter (Mid Gauss) is constructed by subtracting a filter for low-frequency noise from a filter for high-frequency noise.

Background Subtraction This step provides a simple way to remove unchanging noise. Simple background subtraction, as used in *reactIVision*, acquires a single image from the camera and subtracts it from all ensuing images. *EquisFtir* builds this image from the minimum of multiple frames to reduce the effects of camera noise. *CCV* gradually adjusts the subtracted image to account for slow changes in the background noise.

Background subtraction eliminates some forms of noise, but is not a complete solution. In particular, it deals poorly with rapid changes in background noise, such as the room lights turning on or a cloud blocking the sun.

Image Filtering After background subtraction, most remaining noise consists of random camera noise and changed ambient lighting. The camera noise is either single pixels from its sensor, or distorted blocks introduced by its lossy compression. Both affect small areas and have sharp edges, so can be reduced by blurring the image. Changes to ambient lighting, such as the sun coming out from behind a cloud, are only visible through the table's semi-opaque top surface, so affect large areas and have much softer edges.

Image processing theory regards this random camera noise as high frequency, and the ambient lighting as low frequency [1]. Touches lie in the intermediate frequencies: they are larger and smoother than camera noise, but smaller and sharper than ambient lighting. Discarding the high and low frequencies, therefore, will remove much of the remaining noise. The standard approach to this problem uses a *mid-pass filter* (also called “band-pass”), which discards data outside a selected range of frequencies. A mid-pass filter can be constructed by subtracting the results of two low-pass filters (often called “blurs” in image processing). Figure 3 shows a common 1-D mid-pass filter based on the difference of two Gaussian filters. The narrower (first) filter discards high frequencies, while the wider (second) removes remaining low frequencies. We later refer to its 2-D analogue as *Mid Gauss*.

The low-pass filters we focus on replace each pixel with a weighted average of its surroundings. The size of this surrounding region is the *diameter* of the filter. The Gaussian function used above provides a common weighting, because it removes high frequencies with minimal artifacts. Naively computing the weighted average for each pixel would require $O(p \times d^2)$ time, where p is the number of pixels in the image, and d is the diameter of the filter. Fortunately, the Gaussian function is *separable*, meaning it can be applied as a 1-D function to rows and columns in turn. This reduces the complexity to $O(p \times d)$, but remains (as we shall demonstrate) expensive for runtime use.

To improve runtime performance, *CCV* uses an approximation of the Gaussian mid-pass filter. *CCV* replaces the Gaussian with a box filter¹, which takes the mean of pixels within a square region. The box filter can be applied in $O(p)$ time, using only a few additions and multiplications per pixel. Unfortunately the box filter is prone to introducing artifacts around sharp corners [13].

Mid-pass filtering by its nature favors fixed-sized touches. This means that it will discard large signals, such as a palm press and many forms of fiducial markers. *reactIVision* avoids this problem by not filtering the image, supporting fiducial markers at the expense of some noise tolerance. A development version of *CCV* adds an entirely separate image processing pipeline for recognizing fiducial markers on objects, at significant runtime cost. The evaluated versions of *CCV* and *EquisFtir* do not support fiducial markers.

Touchlib uses a median filter to remove noise. This filter replaces each pixel by the median value of its neighborhood; in the case of *Touchlib*, over a 5×5 square centered on the pixel. The median filter is useful to remove speckles, such as camera noise, without blurring larger features in the image. However, it can not remove lower-frequency noise.

These different approaches strike different compromises between accuracy, runtime performance, and generality. As we shall see, the core contribution of this paper is the introduction of a novel approximation to the difference-of-Gaussian filter that provides a superior balance between accuracy and runtime performance.

Phase 3: Blob Detection

Once the noise has been removed from the image, blob detection algorithms are used to identify bright regions. These blobs, hopefully, correspond to where users are touching the surface. The blob detection algorithms used in the evaluated libraries first *threshold* the image, then extract *connected regions*, and *cull* some blobs.

Thresholding This step determines whether each pixel of the image is part of the foreground (in a blob), or part of the background. *CCV* and *EquisFtir* both use a simple threshold: every pixel with a value greater than that configured by the user is considered part of a blob. *reactIVision* divides the image into tiles and uses an adaptive threshold based on each pixel's neighborhood. Adaptive thresholding reduces the need for manual configuration, and can eliminate some low-frequency noise.

¹*CCV* uses a slight variant of the mid-pass filtering described above. They subtract the wider blur from the original image and then apply the narrower blur to the result. This slightly changes the diameters of the filters, but has no other significant effect.

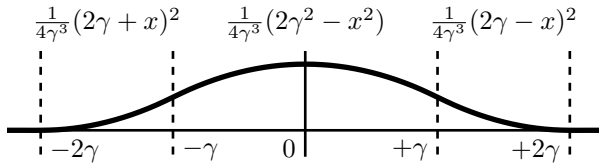


Figure 4: We use the piecewise quadratic function q_γ as a fast approximation of the Gaussian.

Connected Regions Once the image is separated into foreground and background pixels, any connected group of foreground pixels is considered a single blob. The simplest algorithm searches the image from top to bottom, left to right. Upon encountering a foreground pixel it uses a flood-fill to label the blob, and stores its statistics. Because this flood-fill must backtrack to deal with irregular shapes, it is relatively slow. Merge trees, as applied by EquisFtir and reactIVision, can avoid backtracking by combining partial blobs during the top-to-bottom scan [4, 5]. While extracting connected regions, reactIVision also builds the tree of nested bright and dark blobs that encode its fiducials. CCV uses the contour finder provided by OpenCV [12]. Whenever it encounters a blob during the top-down scan, it traces the blob's entire border and saves a representation of the contour. Many variations of this technique exist, some of which are quite efficient [3].

Culling After blobs are detected, the library has a chance to discard those which are not touches. reactIVision keeps blobs with dimensions within approximately 60% of the configured finger size, unless they fall within a recognized fiducial. CCV keeps blobs with total area within user-specified bounds. EquisFtir keeps blobs if the sum of the contributing pixels exceeds the configured threshold.

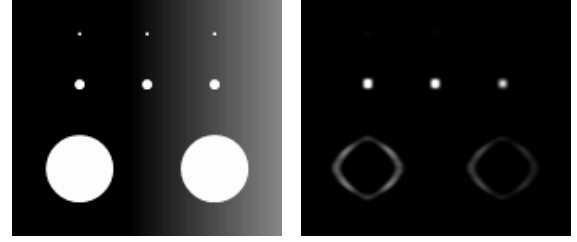
Each approach discards small bright regions which happen to slip through the noise filters. Screening on the sum of the pixels allows EquisFtir to discard faint finger-sized blobs that barely exceeded the per-pixel threshold. An upper bound on blob size, as used in reactIVision and CCV, prevents the library from reporting huge spurious touches in some extremely noisy inputs.

Phase 4: Blob Tracking

To report presses, releases and moves, the library must find blobs in the current frame that correspond to previous touches. The libraries make the reasonable assumption that touches can move only short distances between frames, so associate the closest pairs within a threshold distance. The number of blobs in typical images is small enough that the efficiency of this algorithm is not crucial, so the libraries use a simple $O(b^2)$ search, where b is the maximum number of detected blobs.

THE DUAL-QUAD NOISE REDUCTION FILTER

Having reviewed the image processing pipelines of existing toolkits, we now introduce this paper's core contribution, a novel algorithm for noise reduction. Our *Dual Quad* (DQ) filter is similar in effect to the mid-pass filters discussed earlier. However, rather than being defined by the difference



(a) Original input

(b) After Dual Quad filter

Figure 5: Results of filtering a synthetic image.

of two 2-D filters, it is a 1-D filter applied to both rows and columns. As will be shown in the evaluation section, when implemented within EquisFtir it provides improved touch detection accuracy and excellent performance.

Approximating the Gaussian with a Piecewise Quadratic

The foundation of our DQ filter is a piecewise quadratic function that provides a smooth 1-D blur. Figure 4 shows the shape of this function given a positive integer size γ , which is one quarter of the filter diameter. The three quadratic functions define different regions of the overall function, and it remains zero outside the diameter. More precisely, q_γ is a family of functions parameterized by size γ :

$$q_\gamma(x) = \frac{1}{4\gamma^3} \begin{cases} (2\gamma + x)^2 & \text{if } -2\gamma \leq x \leq -\gamma \\ 2\gamma^2 - x^2 & \text{if } -\gamma < x < +\gamma \\ (2\gamma - x)^2 & \text{if } +\gamma \leq x \leq +2\gamma \\ 0 & \text{otherwise} \end{cases}$$

The division by $4\gamma^3$ keeps the total area of the function equal to one as the size changes. Note that the non-zero domain of the function is actually $[-2\gamma, +2\gamma]$, so the filter diameter is 4γ . In practice we keep γ to a power of two, which allows scaling to be performed via a bit shift rather than division.

Initially this function appears to be a poor replacement for the Gaussian, because traditional techniques would still require $O(p \times d)$ time. As we shall see, however, we can overcome this problem with a novel algorithm that runs in $O(p)$ time, with sufficiently small constant factors to outperform both the 1-D Gaussian and other common approaches.

The Difference of Quadratics

Our DQ filter approximates the 1-D difference of Gaussians using the difference of piecewise quadratic functions. As in the difference of Gaussians (Figure 3), we begin with a narrow filter and subtract a wider one with the same total area. More precisely, we define the dual-quad filter dq_γ as:

$$dq_\gamma(x) = q_\gamma(x) - q_{2\gamma}(x)$$

Applying this 1-D filter to the rows and columns of an image yields output similar to the 2-D mid-pass filters. Figure 5(b) shows the result of dq_4 applied to a synthetic image. The input image (Figure 5(a)) contains a variety of blob sizes, ranging from tiny noise to large circles, and a background ranging from pure black to a lighter grey. In the output, many of these features have been removed: the background gradient

and small dots are completely gone, and the large circles have been reduced to a dim outline. Meanwhile, the finger-sized blobs remain bright and are available for easy recognition.

Fast Implementation of the Dual-Quad Filter

As discussed above, a simple implementation of the DQ filter would require similar processing time to a 2-D Gaussian. We now present our novel approach to implementing piecewise quadratic filters, which reduces this time to $O(p)$ with small constant factors.

Our algorithm is motivated by the fast box filter, a classical example of a filter that runs in fast constant time per pixel [8]. This is made possible by an accumulator, which is updated in $O(1)$ as the filter “slides” through a row or column. Extending this concept to multiple accumulators allows us to apply our piecewise quadratic filter in similar $O(p)$ total time. To our knowledge, this approach is unique within the image processing literature.

Fast Box Filter To give the intuition behind our algorithm, we first review the fast box filter, and then build toward the DQ filter. Consider a filter f summing pixels over odd diameter d , applied to a row or column s around position x . For convenience, let c be $\lfloor \frac{d}{2} \rfloor$. Then:

$$f(x) = \frac{1}{d} \sum_{i=x-c}^{x+c} s(i)$$

The fast sum filter relies on producing $f(x+1)$ from $f(x)$ in constant time:

$$\begin{aligned} f(x+1) &= \sum_{i=x+1-c}^{x+1+c} s(i) \\ &= \left(\sum_{i=x-c}^{x+c} s(i) \right) - s(x-c) + s(x+1+c) \\ &= f(x) - s(x-c) + s(x+1+c) \end{aligned}$$

The fast box filter is simply this sum divided by d . We can generalize this recurrence using the derivative of the filter with respect to x :

$$f(x+1) = f(x) + \int_x^{x+1} f'(x) dx$$

If this integral can be evaluated in $O(1)$, as in the case in the box filter, the entire image can be filtered in $O(p)$, where p is the number of pixels in the image.

Fast Piecewise Quadratic Filter Unlike the fast box filter, the first derivative of our quadratic filter can not be trivially computed. Indeed, by virtue of being a piecewise quadratic, the first derivative is piecewise linear (Figure 6(a)) and the second derivative is piecewise constant (Figure 6(b)). However, the third derivative (Figure 6(c)) is instantaneous, so can be evaluated in $O(1)$ by reading four pixel values:

$$q_\gamma'''(x) = \frac{1}{4\gamma^3} (s(x-2\gamma) - 2s(x-\gamma) + 2s(x+\gamma) - s(x+2\gamma))$$

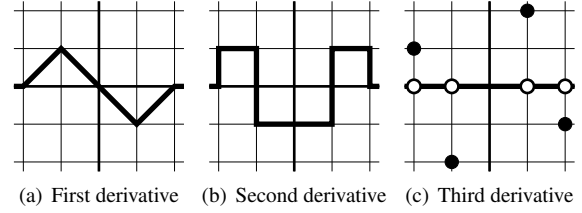


Figure 6: The derivatives of q_γ . These figures use the same axes as Figure 4.

Because γ is a positive integer, q_γ and its derivatives are continuous except at some pixel boundaries. That is, $q_\gamma'(x)$ is linear, $q_\gamma''(x)$ is constant, and $q_\gamma'''(x)$ is zero. We can therefore use the fast filtering recurrence to express the functions, and simplify the corresponding integrals:

$$\begin{aligned} q_\gamma(x+1) &= q_\gamma(x) + \int_x^{x+1} q_\gamma'(x) dx \\ &= q_\gamma(x) + (q_\gamma'(x+1) + q_\gamma'(x))/2 \\ q_\gamma'(x+1) &= q_\gamma'(x) + \int_x^{x+1} q_\gamma''(x) dx \\ &= q_\gamma'(x) + q_\gamma''(x+1) \\ q_\gamma''(x+1) &= q_\gamma''(x) + \int_x^{x+1} q_\gamma'''(x) dx \\ &= q_\gamma''(x) + q_\gamma'''(x+1) \end{aligned}$$

This allows us to store $q_\gamma(x)$, $q_\gamma'(x)$ and $q_\gamma''(x)$ in accumulators, and then calculate $q_\gamma(x+1)$, $q_\gamma'(x+1)$ and $q_\gamma''(x+1)$ in $O(1)$. Using these recurrences, each row or column can be filtered in time linear on its length, and the entire image can be filtered in $O(p)$.

Fast Dual-Quadratic Filter The dq_γ filter is defined as a subtraction of two q functions, so is itself a piecewise quadratic function with the same properties as q . Using three accumulators with the same recurrences as above, we merely need to define the third derivative $dq_\gamma'''(x) = q_\gamma'''(x) - q_{2\gamma}'''(x)$.

This algorithm is sketched in Figure 7. The accumulators $a0$, $a1$ and $a2$ represent $dq_\gamma(x)$, $dq_\gamma'(x)$ and $dq_\gamma''(x)$, respectively. During each iteration the accumulators are updated based on their recurrences. For efficiency, scaling is postponed until the final output is written.

The 1-D dq_γ algorithm is applied to each column of the input image, and then to each row of that result. This entire operation runs in $O(p)$ time. Further, each step consists only of additions and shift operations (multiplications by powers of two), so is inexpensive to compute. Our actual implementation partially unrolls the loop to handle cases where the input image would read out of bounds, and has specialized variants for processing rows and columns.

In summary, our Dual Quad filter provides a 1-D mid-pass filter which can be executed very efficiently. In the next section, we evaluate the effectiveness of our filter against a variety of other noise reduction filters.

```

// Area of the doubled kernel.
const int area = 16 * gamma * gamma * gamma;

int a0 = 0; // Value of filter
int a1 = 0; // First derivative
int a2 = 0; // Second derivative

// Apply the filter over one pass. We start
// before the edge of the image to handle
// the forward-looking reads.
for (i = -2*gamma; i <= N; ++i) {
    // Update the accumulators
    a1 += a2;
    a0 += a1;

    // Output value is adjusted to center of a
    // pixel, and scaled by the total area.
    output[i] = (a0 - 0.5*a1) / area;

    // Update 2nd derivative accumulator with
    // the high-frequency part of the filter.
    a2 -= 8 * input[i - 2*gamma];
    a2 += 8 * input[i - 1*gamma] * 2;
    a2 -= 8 * input[i + 1*gamma] * 2;
    a2 += 8 * input[i + 2*gamma];

    // Update 2nd derivative accumulator with
    // the low-frequency part of the filter.
    a2 += input[i - 4*gamma];
    a2 -= input[i - 2*gamma] * 2;
    a2 += input[i + 2*gamma] * 2;
    a2 -= input[i + 4*gamma];
}

```

Figure 7: The algorithm applies DQ filter to a single row or column of an image in $O(N)$ time.

EVALUATION

As we saw in the last section, the DQ filter works effectively over synthetic data such as that seen in Figure 5. In this section, we demonstrate that the algorithm works well over data obtained from real tabletop surfaces, providing the best combination of accuracy and performance among the tested filters. Specifically, the DQ filter has the best accuracy of all filtering algorithms tested, and close to the best performance. We tested accuracy and performance of the DQ filter compared to the mid-pass Gaussian filter (Mid Gauss), the mid-pass Box filter (Mid Box; as used in CCV) and the median filter (as used in Touchlib).

We performed two experiments, the first addressing performance, and the second addressing accuracy.

Experiment 1: Performance

To evaluate performance, we considered two scenarios. The “small table” scenario places low-cost processing hardware in a small table; this is similar to a SMART Table or even a tablet PC. We assumed a camera capable of delivering 640×480 input, and used a laptop to simulate the embedded computer. This laptop has an AMD Turion 64 X2 Mobile TL-50 (1.6 GHz) processor, 2 GB RAM, and an ATI X1300 graphics card.

The “large table” scenario assumes a conference-table-sized surface with input captured from two 720×640 cameras, and automatically stitched into a 1280×720 video. For this scenario, we used a modern desktop computer with an Intel Core 2 Quad Q8200 (2.33 GHz) processor, 4 GB RAM, and an NVIDIA GeForce 9600 GT graphics card.

The image processing in both scenarios was executed using a single CPU thread. This simplified timing, and allows a fair comparison with the single-threaded pipelines implemented by CCV and reacTIVision. We measure the runtime performance of the noise reduction algorithms using two metrics:

- *Filter time*: This represents the time (in milliseconds) required to apply a given noise reduction filter to a camera image. This allows direct comparison of the filters within the EquisFtir pipeline.
- *Pipeline processing rate*: This represents the number of frames per second that can be processed by the complete input pipeline (of which the noise reduction filter is one component.) This indicates the fastest camera speed that can be supported by the pipeline.

Method For each of the two scenarios, we used the hardware and input image sizes specified above.

We wrote an application to read frames from a video file, and pass them to the customized EquisFtir pipeline. The execution time of the entire pipeline and of the candidate noise reduction filter was measured programmatically, using the Win32 performance counter. The application wrote out both times in milliseconds, along with the standard deviation. The pipeline processing rate for a given filter was then calculated as $1000/\text{pipeline execution time}$

The performance of the pipeline depended slightly on the number of blobs detected, but was otherwise unaffected by the content of the source video. To minimize this effect, we used a clean video from which the different algorithms produced similar numbers of blobs.

We tested the following filters by embedding them within the EquisFtir image processing pipeline:

- *Dual Quad* (DQ): our novel mid-pass algorithm.
- *Mid Gauss*: the difference of Gaussians using filters provided by OpenCV [12].
- *Mid Box*: our implementation of the difference of boxes filter used in CCV.
- *Median*: the median filter provided by OpenCV, as used in Touchlib.

For calibration, we informally observed processing times reported by the CCV and reacTIVision toolkits.

Performance Results Figure 8 shows the results of this experiment. All differences in the table are significant at the $\alpha = .05$ level. When applied to the same input, CCV and reacTIVision reported comparable processing times.

We see that in both scenarios, the DQ filter dramatically outperformed the Mid Gauss filter and the Median filter. Our implementation of the Mid Box filter was somewhat faster than the DQ filter. This is not surprising, as the box filter is a much simpler function than our piecewise quadratic.

Algorithm	Small Table		Large Table	
	Filter Time (ms)	Pipeline Frame Rate (fps)	Filter Time (ms)	Pipeline Frame Rate (fps)
Dual Quad	9.1	57.7	12.9	40.7
Mid Box	6.9	66.2	11.7	42.9
Mid Gauss	36.1	22.6	37.2	20.6
Median	28.2	27.5	56.2	14.8

Figure 8: Comparison of performance of four noise reduction filtering algorithms inserted into the EquisFtir pipeline. Filter time is the time to execute the noise reduction filter over a single camera image. Pipeline frame rate is the number of frames per second that the entire pipeline can process.

CCV achieved a pipeline frame rate of approximately 45 FPS in the small table case, and 33 FPS in the large table case. CCV was considerably slower than EquisFtir with Mid Box. This is primarily because CCV uses the OpenCV version of the box filter, which is significantly slower than the Mid Box filter we added to EquisFtir.

reactIVision exceeded 100 FPS in both cases. This is possible because reactIVision does not use a noise reduction image filter. The resulting simpler pipeline runs at approximately the same speed as the EquisFtir or CCV pipelines with the noise filter excluded.

We did not record processing times for Touchlib. Our timing of OpenCV's median filter, as used by Touchlib, shows that it is considerably slower to compute than the DQ filter and Mid Box filters.

Over all, we conclude that the DQ and Mid Box filters are both strong candidates for input libraries where performance is an issue. The next experiment shows, however, that the DQ filter is a far better choice when accuracy is taken into consideration.

Experiment 2: Accuracy

Our second experiment compared the accuracy of different noise filters including our novel DQ filter. For consistency, we tested all filters within EquisFtir. To provide a broader comparison, we also evaluated the accuracy of CCV, Touchlib and reactIVision. As we have discussed, CCV uses an OpenCV-based Mid Box filter, Touchlib uses a Median filter, and reactIVision uses no noise reduction filter.

The Discriminability Index The accuracy of an input library is derived from two measurements:

- Hit rate (H): the percentage of the time that a touch was present where the algorithm correctly identified it (how good the system is at correctly identifying touches); and
- False alarm rate (F): the percentage of time where there was no touch, but the algorithm incorrectly reported one (how frequently the system detects a touch when none is there).

It is not sufficient to compare algorithms based on hit rate alone, since a liberal algorithm can always increase its hit rate by allowing more false alarms. (In fact, users of all the evaluated input libraries can choose the tradeoff between hit rate and false alarms by adjusting thresholds.) *Signal detection theory* [9] allows us to combine both hit rate and false alarm rate to provide a single measure of the detection accuracy. Accuracy is measured in terms of a *discriminability index*, or d' for short, where $d' = z(F) - z(H)$.

In this equation, $z(F)$ and $z(H)$ are the z-scores for the false alarm rate and hit rate respectively. (A z-score measures the distance of a value from the mean of a normally distributed population, and is measured in standard deviations.) This approach allows us to compare algorithms based on a single metric.

Some examples of d' values are:

- $H = 50\%$, $F = 50\%$ gives $d' = 0$ (purely random)
- $H = 75\%$, $F = 5\%$ gives $d' = 2.32$
- $H = 90\%$, $F = 1\%$ gives $d' = 3.61$
- $H = 99\%$, $F = 1\%$ gives $d' = 4.65$

Method To obtain a broad comparison, we solicited data from a variety of labs in Canada and Europe. We asked respondents to record video from their tabletop's camera while the table was being used. This video records the raw input that the input library uses to determine touch events. From this data, we were able to compare the effects of our noise reduction algorithms against other openly available approaches. We now describe these steps in detail.

We recruited participants by sending email to our project partners in Canada and Europe, by posting on the NUIGroup forums, and by mailing to a Google Group related to multi-touch table construction. We solicited people who had hand-built FTIR or DI tables, using low-cost cameras. Six groups responded and were sent instructions on how to send us data, and four provided data. As a fifth case, we included data from our own tabletop surface. We refer to these sites as L1 through L5.

We provided participants with a simple Java application that drew moving colored circles on the tabletop surface. The participants followed the circles with their fingers, providing drag inputs. Participants were asked to use the video capture application of their choice to record the images from their camera as they interacted with the test application. They were asked to perform this procedure using two lighting conditions: as they normally used their tabletop, for example turning down lights and closing blinds if they usually did; and in a noisy light environment, with lights on and blinds open. This provided approximately two minutes of video for each condition. All algorithms performed well on the clean data, so we compare accuracy using the noisy datasets.

The resulting videos represented a wide variety of hardware conditions. Four tables used FTIR technology (L1,L2,L3,L5) and one used DI (L4). Cameras included inexpensive Logitech webcams (L1, L5), Sony Playstation Eye (L2, L4) and a professional quality PointGrey FireflyMV (L3).

Algorithm	Lab	Y;N	L1		L2		L3		L4		L5	
			6179;1655	1826;776	3028;736	1675;907	2889;985					
Dual Quad	H	d'	.94	4.8 ^{††}	.83	4.3 [*]	.90	4.3	.91	4.0 ⁺	.98	5.6 [*]
	F		.001		0.0		.001		.004		0.0	
Mid Gauss	H	d'	.99	5.2 [*]	.73	4.0 ^{††}	.16	1.6	.62	3.7 ⁺	.96	5.2 ⁺
	F		.001		0.0		.005		0.0		0.0	
Mid Box	H	d'	.98	5.2 [*]	.71	3.9 ^{††}	.52	0.9	.62	2.0 [†]	.97	5.4 [*]
	F		.001		0.0		.198		.049		0.0	
Median	H	d'	.92	4.6 ^{††}	.66	3.1 ^{††}	.20	0.4 ⁺	.20	0.8	.69	1.1 [†]
	F		.001		.004		.102		.049		.27	
CCV	H	d'	.95	3.6	.76	3.1 ^{††}	.74	3.0	.79	1.9 [†]	.87	4.6 [*]
	F		.024		.008		.008		.132		0.0	
Touchlib	H	d'	.93	4.5 [†]	.64	2.8 [*]	.15	-0.4	.34	1.2	.34	3.1
	F		.001		.008		.254		.049		0.0	
reactIVision	H	d'	N/A	N/A	.45	2.7 [†]	.13	0.4 [*]	.15	-0.5	.03	1.6 [*]
	F		N/A		.003		.057		.307		0.0	

Figure 9: Results for clean and noisy datasets, showing number of “yes” frames (Y), number of “no” frames (N), hit rate (H), false alarm rate (F) and d' for each case. Within each column, d' values whose differences are not statistically significant (at $\alpha = .05$) are grouped with *, † and †† symbols.

For each video, we manually determined a “ground truth” by manually marking touches in each frame. To create the ground truth, we developed an application that played the submitted videos and allowed an analyst to trace the positions of touches. These traces were recorded to file, allowing automated computation of hit rate, false alarm rate and d' values for each of the algorithms we tested.

All algorithms were hand-calibrated to give the best results (in terms of d') that we could obtain.

Accuracy Results

Figure 9 summarizes the results of our experiment. The table shows the d' value for each filter in each of the five datasets. For context, the hit rate and false alarm rate is also shown in each case.

Statistical significance of the difference between d' values was computed in the standard way for signal detection theory, by comparing the 95% confidence interval around the observed differences [9]. Within each column of Figure 9, d' values whose differences are not statistically significant are grouped with *, † and †† symbols.

The results show that the DQ filter provided the best accuracy in all 5 datasets (including statistical ties); Mid Gauss provided best accuracy in 4/5 datasets and Mid Box provided best accuracy in 3/5 datasets.

Analysis

Combining the results of these two experiments, we conclude that the DQ filter represents the best balance between performance and accuracy. The DQ filter provides slightly slower performance than Mid Box in return for substantially higher accuracy. Compared with Mid Gauss, the DQ filter provides dramatically better performance while providing similar or better accuracy.

A few interesting cases are worth highlighting.

In L3, most filters perform very poorly. As shown in Figure 10, in the L3 video, the background lighting becomes brighter over the course of time. This is consistent, for ex-

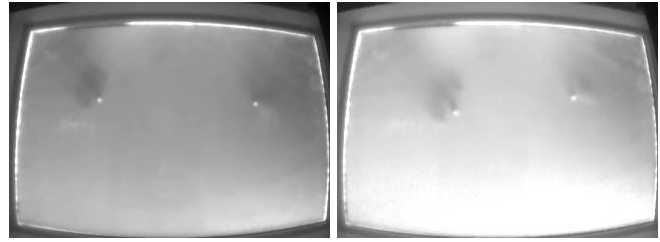


Figure 10: Two excerpts from L3 Noisy showing the change of brightness of the image over a two minute time frame.

ample, with the sun coming out from behind a cloud. This dynamic change in the level of background lighting reduces the effectiveness of EquisFtir’s background subtraction step, which assumes that background noise is completely static.

Our DQ filter is more robust to changes in background light than the other filters, as it is more selective about the shape of a touch. CCV does considerably better than the Mid Box case, because its background subtraction algorithm successfully adjusts to the slow change. This example shows the utility of the dynamic background subtraction, but the fact that EquisFtir still outperformed CCV indicates that there is room for improvement in the noise reduction algorithm used by CCV.

The L1 dataset shows a limitation of the DQ filter. This video was recorded on a smaller table, so touches are proportionally larger with a different distribution of brightness. Here the DQ filter’s selectivity works against it, reducing the hit rate. Even in this case, the DQ filter’s results remain a statistical tie for best accuracy,

It is interesting to compare the CCV and Touchlib libraries to the Mid Box and Median cases, which use the same filters within the EquisFtir pipeline. Touchlib is similar to Median in four of five datasets. CCV is similar to Mid Box in three datasets. Of the other two, one is due to the background lighting changes described above. Contributors to these differences are that the EquisFtir, Touchlib and CCV pipelines differ in several ways (as described earlier). As a result of these differences, we found Touchlib and CCV to be more sensitive to calibration than EquisFtir, sometimes making it difficult to maximize their d' values.

DISCUSSION

Our experiments have shown that the DQ filter is a strong candidate for inclusion in touch-focused input libraries for optical surfaces. We now discuss some of the broader questions around the problem of noise reduction, present issues for further research, and describe limitations of our approach.

Can’t We Just Solve This in Hardware?

Some sources of noise in our experimental datasets can be attributed to low-cost tabletop construction. To some degree, these problems can (and should) be solved with higher-quality hardware and construction techniques. However these problems are not always obvious: our highest d' values were scored by tables using low-cost webcams (L1 and L5), because the camera noise was easily removed by all filters.

Meanwhile, the lowest d' values were from a table using a professional camera, which happened to be saturated by background light (L3).

Table designers need to be mindful of the tradeoffs of different hardware alternatives. The primary balance is between price and quality. We hope that multi-touch tabletops will eventually be a common part of the home, but commercial tabletops still cost far too much for consumers. To expand hobbyist and consumer interest – and preserve research budgets – we cannot automatically adopt hardware solutions without regard to cost. In many areas researchers and hobbyists have explored hardware alternatives, and discovered good compromises. For example, the Sony Playstation Eye offers a low-cost camera with good frame rate and quality.

The problems caused by environmental light have also drawn significant attention. One approach for FTIR tables, as used by the SMART Table, is to use a tabletop surface that blocks infrared light. The necessary materials are expensive, and exclude optical techniques (such as DI) that require the camera to capture light originating above the surface.

Multiple cameras can be used to improve image quality and tolerance to environmental light. The Microsoft Surface, for example, uses five cameras to produce a reported 1280x960 net resolution. This adds the cost of both additional cameras and the hardware required to connect them to a computer. In addition, they add to the image processing requirements, further motivating fast algorithms like the DQ filter.

In general, it makes sense to use the best hardware that the construction budget allows. Nevertheless, no hardware solution can remove all noise. As multi-touch tabletops are increasingly used in difficult environments, the need will remain for better and faster noise reduction algorithms.

Aren't Computers Getting Faster?

Computers continue to become more powerful, with each year seeing improved clock rates, increased numbers of cores, and more powerful graphics processing units (GPUs). It is reasonable to consider whether problems with algorithmic performance will be rendered moot by faster hardware.

Because of these improvements in computer hardware, optical multi-touch tables can become larger, and operate at higher input resolutions and frame rates. Likewise, they can become more portable: pico-projectors are increasingly available, and the iPhone 4 now ships with a built-in 720p video camera. Integrating sensing elements with an LCD display (e.g. ThinSight [6]) could even allow optical multi-touch with a hand-held device, but inherits the processing and power limitations of such a platform.

Furthermore, the tests we have presented assume a significant proportion of the processing resources will be dedicated to input processing (50% for the “small table”, 25% for the “large table”). In fact, the huge majority of the computer's resources should be available for the user's applications, not dedicated to an input device. Therefore, the need for fast image processing algorithms will continue despite improvement in computer hardware.

Alternatives to performing image processing on the CPU also exist. Modern graphics cards include GPUs, which are massively parallel data processors. To obtain a sense of the current state of GPU processing, we applied CCV's GPU implementation of the Mid Gauss filter to our “large table” scenario (using an NVidia GeForce 9600 GT). The pipeline frame rate was approximately 44 FPS, more than double the frame rate of EquisFtir with the CPU-based Mid Gauss. General-purpose GPU libraries, like OpenCL, would support our filter on the GPU, and likely offer a similar speed-up. Relying on GPU processing poses compatibility problems, and has the disadvantage of loading the GPU. Either can interfere with user's actual applications. These problems can be mitigated by limiting the selection of GPUs or adding special-purpose image processing hardware. However the cost of these alternatives suggests that fast CPU-based processing will remain interesting, particularly as the number of CPU cores continues to increase.

In sum, while faster computers help with the input processing problem, demand for more capable tables, lower costs, and preserving processing power for actual applications will continue to drive the need for fast algorithms.

Future Work

Our results highlight two problems for future research.

Adaptive Background Subtraction All of the noise reduction algorithms tested rely on background subtraction to remove “touch-like” artifacts. Changes in environmental light, shadows, and shifting or jiggling table components can render static background subtraction ineffectual. To avoid these problems, it is important to investigate improved techniques for *adaptive* background subtraction. CCV uses a form of adaptive background subtraction that supports slow changes, as encountered in L3. Rapid changes, such as turning on the lights in a room or shaking the frame of the table, require improved techniques for achieving this functionality.

Automatic Configuration Configuring optical multi-touch libraries is difficult, because expertise – or significant trial and error – is required to understand the effects and interactions of each tunable parameter. We reviewed the code and documentation of all tested systems, and experimented extensively in pursuit of good results. However, it is hardly reasonable to expect typical users to go to such lengths to choose effective thresholds. It is important to investigate techniques for simplifying and automating the configuration process.

Limitations

The work presented in this paper has a number of limitations that should be addressed in further study.

None of the systems we tested take inter-frame information into account. In particular, blob tracking can be greatly improved by basing detection decisions on a small buffer of frames. If a blob is lost in one frame and recovered in the next, the blob tracker can infer that the finger was temporarily lost and not interrupt a drag operation. This approach comes with the cost of increased latency, since inputs cannot be processed immediately. Because the best tradeoff between accuracy and latency is application-dependent, we be-

lieve that such algorithms should be provided by higher-level gesture and UI libraries, rather than embedded within the input processing pipeline.

Our experiments assumed that users kept their touches widely separated. None of the evaluated libraries handle the merging and splitting of blobs (e.g. two fingers touching and then moving apart). The problem appears in the blob tracking phase: when the fingers move apart, one will be reported as a new press. This problem can be addressed in two ways: using shape information, a library may be able to recognize a “figure-eight” blob and emit two touches; if a single touch splits into two, the library needs to report the correction.

The algorithms described in this paper depend on finger-based input. Interactions with the tabletop using different shapes, like a palm, edge of hand, or a stylus or paintbrush [15], would be filtered out. Since optical tabletop interactions are biased towards finger-based gestures, we believe that this is a reasonable price to pay for the significant increase in accuracy. Other processing approaches could be combined in parallel to recognize different shapes.

Finally, we have compared our work only to published algorithms and those used in open-source libraries. These may differ from the algorithms used in commercial tables, which are far more likely to choose dedicated image processing hardware. While the algorithms used in these products are not available to us to study, our informal experience is that they suffer from similar environmental problems as home-constructed tables. From this, we hypothesize that our results will also be of use in commercial optical tabletop products.

CONCLUSION

In this paper, we have described a novel noise reduction algorithm for finger-based optical multi-touch. This algorithm is based on a piecewise quadratic filter that removes low- and high-frequency noise from camera input, enhancing the mid-frequency signals that correspond to finger-sized input. The approach is practical due to a novel algorithm for applying piecewise quadratic filters in fast constant time per pixel.

We have empirically evaluated our algorithm against common alternatives using raw input from our table as well as four other labs. This has shown that the approach offers significant improvement over other algorithms, and also outperforms the popular CCV, Touchlib and reactTIVision libraries. Our noise reduction algorithm has been implemented and optimized within the EquisFtir library.

ACKNOWLEDGMENTS

We would like to thank the groups who contributed data for our experiment: Robert Biddle, School of Computer Science, Carleton University; Sophie Stellmach with Raimund Dachselt, User Interface and Software Engineering Group, Otto-von-Guericke-Universität Magdeburg; Jimmy Hertz, SasExperience; and Khaled Tangao with Stacey Scott, Collaborative Systems Laboratory, Systems Design Engineering, University of Waterloo. We also gratefully acknowledge the financial support of the NSERC SurfNet Strategic Network.

REFERENCES

1. K.R. Castleman. *Digital Image Processing*. Prentice Hall Press, Upper Saddle River, NJ, 1996.
2. CCV. Community core vision, <http://ccv.nuigroup.com/>.
3. F. Chang, C. Chen, and C. Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004.
4. L. di Stefano and A. Bulgarelli. A simple and efficient connected components labeling algorithm. In *ICIAP*, pages 322–327, 1999.
5. L. He, Y. Chao, K. Suzuki, and K. Wu. Fast connected-component labeling. *Pattern Recognition*, 42(9):1977–1987, 2009.
6. Shahram Izadi, Alex Butler, Steve Hodges, Darren West, Malcolm Hall, Bill Buxton, and Mike Molloy. Experiences with building a thin form-factor touch and tangible tabletop. In *Tabletop*, pages 181–184, 2008.
7. M. Kaltenbrunner and R. Bencina. reactTIVision: a computer-vision framework for table-based tangible interaction. In *Tangible and Embedded Interaction*, pages 69–74, 2007.
8. A. Lukin. Tips & Tricks: Fast Image Filtering Algorithms. In *Proc. GraphiCon*, pages 186–189, 2007.
9. N.A. Macmillan and C.D. Creelman. *Detection Theory: A User's Guide*. Lawrence Erlbaum, second edition, 2005.
10. K. Nice, T.V. Wilson, and G. Gurevich. How stuff works: How digital cameras work. <http://electronics.howstuffworks.com/cameras-photography/digital/digital-camera5.htm>.
11. NUIGroup. Touchlib: A multi-touch development kit, <http://nuigroup.com/touchlib>.
12. OpenCV. OpenCV computer vision library, <http://opencv.willowgarage.com/wiki/>.
13. R. Rau and J.H. McClellan. Efficient approximation of Gaussian filters. *IEEE Transactions on Signal Processing*, 45(2):468–471, 1997.
14. S.D. Scott, A. Allavena, K. Cerar, G. Franck, M. Hazen, T. Shuter, and C. Colliver. Investigating Tabletop Interfaces to Support Collaborative Decision-Making in Maritime Operations. In *ICCRS*, 2010.
15. J.D. Smith, T.C.N. Graham, D. Holman, and J. Borchers. Low-cost malleable surfaces with multi-touch pressure sensitivity. In *Tabletop*, pages 205–208, 2007.
16. C. Wolfe, J.D. Smith, and T.C.N. Graham. A low-cost infrastructure for tabletop games. In *Future Play*, pages 145–151, 2008.