Thor: Whiteboard Capture and Indexing

Mihai Parparita[*] **Szymon Rusinkiewicz**[†]

January 5, 2004

Abstract

We present the first phase of the Thor project. Using consumer grade digital cameras, we describe how a system can be built that records whiteboard content, and then stores the resulting strokes in a space-efficient, vector format. This acquisition portion is designed to serve as the foundation for a larger system that enables captured whiteboard data to be indexed and then searched using graphical queries. The system is evaluated, and results are presented.

Keywords: Computer vision, stroke capture, indexing, searching.

INTRODUCTION

Whiteboards (and their close cousins blackboards) have become increasingly prevalent in office and classroom settings. In the former, they are used for impromptu meetings, for pre-drafting and for keeping track of to do lists and other reminders, as enumerated in [6]. Blackboards have long been used as instructional tool, possessing the advantage over pre-made slides that material can be constructed on the spot, better showing the process by which information presented was arrived at. The trade-offs of using a whiteboard are intransience and lack of concrete organizational schemes. Having limited space, whiteboards cannot be added to infinitely. The only option is to erase existing content, but unless it is saved in a somewhat clunky manner (scribbles are either copied by hand or a picture is taken), data is lost. The other disadvantage is that information arrangement is very ad-hoc, and a viewer who is unfamiliar with the layout is reduced to scanning the entire whiteboard in order to find content (though emphasis marks such as borders and different colors may be used as aids, but again, their meaning is very much writer-dependent).

As a way of counteracting these issues, several systems have been proposed, including [6, 7]. Using a stroke capture mechanism, they work around the intransience problem. To help with the limited space issue, [6] transforms the entire whiteboard surface into a projected area, the contents of which can (on a group by group basis) be moved around and reduced in size in order to make room for more strokes.

The primary drawbacks of these projects are the use of specialized equipment and a lack of focus on large-scale organization. The contribution that the Thor system makes to the field is to provide a very easily attainable capture method, and to focus more on the information location portion of the organizational side (this second aspect fill be focused on for the remainder of the project). Previous systems used enhanced capture environments such as Smartboards (in [6]) and the Mimio product (in [7]), both of which use specialized marker tracking methods (ultrasonic in some cases, radio tracking in others). Though these have the advantages of being very precise in terms of pen location and stroke timing, they do have the usual drawbacks of specialized equipment: lack of portability and price/complexity. In contrast, Thor uses consumer-grade digital cameras that are set to periodically capture traditional whiteboards or blackboards. These are easily available, and can be transported to any setting.

With regards to the organizational angle, Thor will implement a scalable database-backed storage mechanism that can easily queried for information. This will be covered in the final version of this write-up.

[*]Princeton University. email: mparpari@princeton.edu
[†]Princeton University. email: smr@cs.princeton.edu

The rest of this paper provides an overview of the system and evaluates its performance on real-world data.

SYSTEM OVERVIEW

The stroke acquisition portion of Thor system is divided into a few distinct phases. First, the camera must be calibrated. Then, the capture loop can be entered. This consists of recording a new image, computing the differences between it and the previous frame, thinning of captured strokes and removal of occluders (e.g. people) and conversion of stroke pixel data into control points.

Calibration

Although the capture system relies on digital cameras, the analog-to-digital conversion that goes on within them is still an imprecise process. Noise, subtle changes in lighting conditions and other phenomena can all affect the captured image, resulting in "differences" that do not reflect actual changes in the written content. As a result, we must get a feel for what constitutes the normal range of values for pixels in a supposedly static environment. The way in which the Thor system achieves this is to capture several (i.e. 8 or more) frames of the environment in its neutral state (nothing is written, there is no definite movement in front of the camera). This can be be done for a short duration, to get a feel for the instantaneous level of changes, or over a longer period of time (e.g. a whole day) to see what the complete range of variations can be. Once this is done, the margin of error for each pixel can be determined, so that true differences can be separated from spurious ones.

This calibration stage also allows us to determine what constitutes the "background" for the strokes that are written and then later are erased. Since whiteboards are not layered beyond the board itself and strokes drawn on top of it, capturing the background in the beginning allows us to not have to specifically keep track of what is obscured and then later revealed.

Capture

Capture is currently done by mounting the contents of a digital camera as yet another directory in the file system, and then traversing it while looking for image documents (importing is done via QuickTime, so that most major graphics formats are supported transparently). This allows whiteboard meetings to be recorded without a computer being present. In the future, cameras that support remote control could be used to record these meetings in real-time.

Consumer grade cameras currently support resolutions in the three to five megapixel range. The Thor system can use images as low as 1 megapixel (depending of whiteboard size and camera distance). In fact, lower resolution images have the advantage of having less noise (if the camera downsamples), of being faster to process, and of letting the camera store more images without need for periodic downloading to a computer. The fact that most cameras use USB 1.1 connections (limited to 12 Mbps) and even if that bottleneck were to be removed, the limiter would then be the storage media, which has speeds of around 3 MB/s. This prevents capture from being done much more often than once every 10 seconds.

As an alternative capture method, webcam-grade video cameras were considered. These also had the advantage of low cost (sometimes even lower than digital cameras), ubiquity and high refresh rates. However, quality was too low to allow large whiteboards to be captured - despite nominal 640 x 480 resolutions most cameras deal poorly with non-ideal lighting conditions.

Differences

The difference algorithm uses a very simple pixel-by-pixel approach. For each pixel, we look at the change in its RGB components when compared to the previous image. If this change is greater than a fixed constant times the previously measured range, then we mark it as changed. Changed pixels can mean one of two things: a new stroke was drawn, or an existing one was erased. To differentiate between these two cases, we look to see if the new value is similar to the calibrated background color at that location. If this is the case, then background has been revealed, and we conclude that a deletion took place. If not, then a new addition must have been made.

Since the entire stroke extraction process depends on the difference algorithm, care must be taken to ensure that its output is accurate. Therefore, as a post-processing step, we apply a despeckling algorithm to the difference image, in order to remove stray values. The algorithm works in two passes. In the first pass, we compute for each pixel how many non-background (8-way) neighbors it has. In the second pass, we can do one of two things: remove non-background pixels because they are isolated, or fill in background pixels because they are completely surrounded by non-background ones. The first is done by looking to see how many non-background neighbors a pixel has. If this value is less than 2, and it has no neighbors with values greater than or equal to 2, then the pixel is replaced with the background color (we must also look at the counts for neighbors, since if we only looked at the count for the pixel itself we would remove endpoints of single-pixel thick lines). In the "filling in" case, if a background pixel has 8 non-background neighbors, then it is replaced with the average color of its neighbors. This is done in order to prevent these one pixel holes from affecting the thinning algorithm.

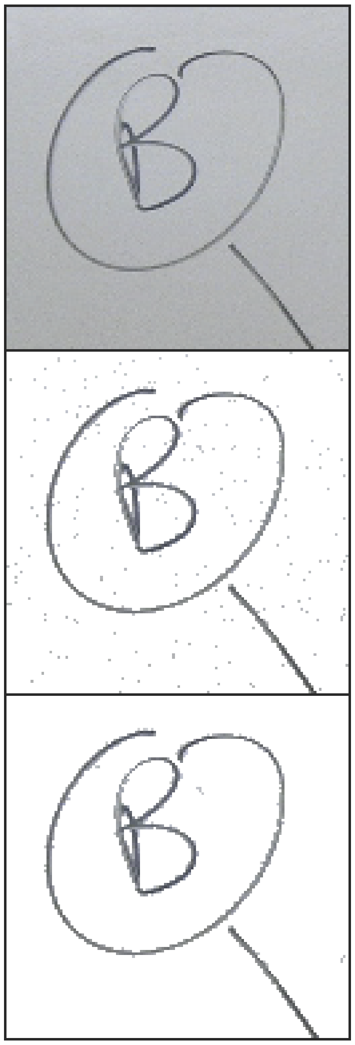

Figure 1: Computation of differences without and with despeckling

Once the differences are computed, the difference image is split up into continuously connected sub-images (so that sets of strokes are separated). For each sub-image, stroke thickness is estimated by dividing the total number of content pixels by the number of perimeter pixels (as described in [2]). If this value is beyond a certain threshold (i.e. the stroke is too thick), we assume that the so-called "stroke" is in fact an occuluder (e.g. a person) and the sub-image should be discarded. The rest of the steps are applied to each remaining sub-image individually.

Thinning

Once the pixels representing the newly-written strokes are determined, we must then reduce them to their most basic structure (this means that, in its current implementation, Thor does not preserve stroke thickness, but since most markers and chalk are of uniform thickness, this is not a serious drawback). The process is called thinning, and it involves traversing all edge pixels of a stroke, iteratively removing all those that are deemed extraneous until a single pixel thick "skeleton" remains. The thinning implementation that we have chosen is described in [8]. This implementation has the advantages of performance and of not "eroding away" diagonal lines. The paper leaves some functions unspecified, so the implementations we have chosen are described below:

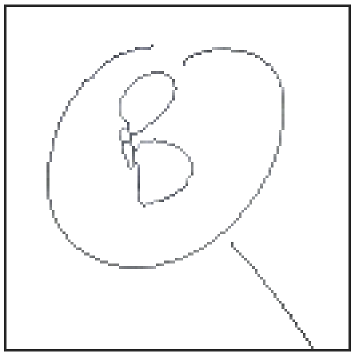

Figure 2: Results of thinning

Seeding function: This function must determine how many contour loops are present in the image, and what their starting points are. Contour loops are defined as continuously connected set of edge pixels (for example, a line would have one contour loop, a hollow circle two, and a figure eight three). For this, we chose to first build a list of all background pixels. Then, the first element from this list is picked, and a 4-way floodfill is done, removing all encountered background pixels from the list. The first encountered edge (non-background) pixel is used as the starting point of the current contour loop. The process is repeated until no more background pixels are left in the list, with each iteration representing a contour loop. To determine the predecessor of a start point, we pick any of its background-colored neighbors, and scan counter-clockwise from it until we find a non-background neighbor (scanning must be done CCW since the successor function works in a clockwise order, and we are working our way backwards).

Termination function: The thinning algorithm works its way around each contour loop, removing extraneous pixels. Since it can take multiple passes for all these to be processed, the process cannot be stopped after a fixed number of steps. [8] does not specify an exact iteration termination function, so one had to be devised for Thor. It is not as simple as seeing if the same pixel was visited twice, since it could be encountered from one direction and then another. We therefore keep track of the pixel immediately following the last one that was deleted, as well as its successor. If we encounter this pair (in the same order) twice, it means that we've done a full loop without deleting any more edge pixels, therefore we are done with this iteration.

Stroke Extraction

Once we have the set of thinned stroke pixel, we can begin to convert them to vector form, i.e. a list of control points that are then connected using various interpolation methods (currently the system uses simple linear interpolation, but more elaborate interpolation methods such as Catmull-Rom splines could also be used). Systems such as the one described in [7] use more complex stroke extraction mechanisms; for example one that separate strokes into straight-line and curved portions. Unfortunately, the system described therein depends on having continuous pen input (i.e. being aware of drawing speed and stroke timing). Since our capture recognition process is off-line, we are reduced to using a simpler mechanism that only deals with the stroke pixels.

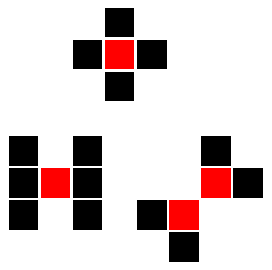
Figure 3: Crossing pixels that must be visited more than once

The stroke extraction algorithm works by first generating vector versions with as many control points as possible (i.e. one for each pixel) and then removes the points in ascending order of error. Therefore, the first step must be to build up these ”dense” strokes that incorporate every single pixel. Superficially, this seems very simple, since we have thinned the images already, and it should be a mater of picking any pixel, and then following one of its non-background neighbors after another. However, things begin to appear more complicated when taking into account the fact that strokes can intersect one another, and thus the same pixel may belong to more than one stroke. Therefore, the first step is to compute for each pixel how many times it can be visited. Based on the methods described in [4], we can determine this based on the number of background to non-background transitions that are made as the pixels neighborhood is walked, as well as on the number of non-background neighbors that it has. The number of transitions is one indicative of how many strokes are intersecting at this point. If there are two transitions, then there is only one stroke and the pixel should only be traversed once. If there are four, then there are two strokes, and the pixel can be visited twice, as in the case of the top example in figure 3. However, we can have cases where the number of intersections is two or three, and yet the pixel is really at an intersection of two strokes, as show in the lower two examples in figure 3. This is where the number of non-background neighbors comes in to play. Therefore, the following table is used to determine the number of visits:

Transitions	Non-bg. Neighbors	Visits
0	*any*	0
1	*any*	1
2	< 4	1
2	≥ 5	2
3	< 2	1
3	≥ 3	2
≥ 4	*any*	2

Once we have determined how many times each pixel can be visited, then we can begin the second phase, the actual traversal of pixels. One thing that must be taken into account is to make sure that when going through an intersection we do not make an unnatural turn (e.g. an X-shaped crossing should not be separated into to L-shaped strokes that graze each other, instead there should be two straight lines that intersect). In order to do this, when looking for the successor pixel of the current one, we begin by first testing in the offset direction that the past few steps have already gone in. The average direction update function is as follows:

1. Number the 8 neighbors of a pixel from 0 to 7 in a clockwise fashion
2. Let D be the current average direction
3. Let D' be the newly picked direction for the current pixel
4. Let K be a decay constant between 0.0 and 1.0
5. Then the new average direction is D * K + D' * (1.0 - K)

Now that we have these "dense" strokes, we must begin the simplication process. This is done in a similar manner to edge collapse-based simplification of 3D meshes, as described in [1]. We define the simplication operation to be the removal of a control point along the stroke (the endpoints are excepted from this).The error function is the difference in overlap of the stroke and its corresponding pixels between when that control point is present and when it is removed. We create a priority queue in which we insert all of these control point removals, and then pick the one with the lowest error delta. Once a point is removed, we update the entries which might have been affected (in the case of linear interpolation, only its immediate successor and predecessor) and repeat the process. We continue this until the total stroke error divided by the stroke length grows beyond a certain limit.

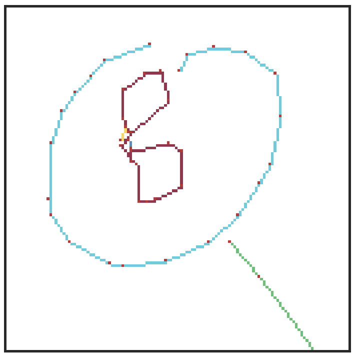
Figure 4: Stroke extraction outcome

PERFORMANCE

The Thor system is conceived to support real-time operation (e.g. images are captured, strokes are extracted and indexed as a whiteboard meeting goes on). As a result, performance is a criterion to consider. Using the current codebase, on a 1 GHz G4 processor, with a dataset of 100 800 x 600 pixel images (average of 174 strokes per frame), end-to-end processing time measured was 5.89 seconds per frame. This falls below our limit of 10 seconds per frame, and leaves enough time for additional operations (such as indexing) to be done. The other time-intensive task, transfer of images from the camera, is not processor-intensive, and can be done in parallel. That is, while the previously captured frame is being processed, the next one is being fetched.

As for the processing per frame, the breakdown into phases is as follows: loading of images takes up 6.0% of the time, computation of differences 56%, thinning 5.8% and stroke extraction 32% (as measured with a statistical sampling tool). Of the time spent computing differences, most of it goes towards separating the difference image into sub images. As it can be seen, certain hot spots have been observed, but they appear to be implementation specific and in the details, therefore easily surmountable, thus performance should not prove to be a barrier in the future.

DISCUSSION

In its current form, the system has achieved its initial goal of providing a foundation for the subsequent tasks of stroke indexing and searching. The system shows that it is possible to extract strokes from data captured by consumer grade cameras. However, ease of use is not quite as ideal as it could be. For example, in order to minimize extraneous frame-to-frame differences, the camera must be attached to a tripod, the white balance must be set to a fixed value, and the focus distance must be specified by hand. The latter two issues could be compensated for by extra computation, it remains to be seen if such a trade-off is worthwhile.

FUTURE WORK

As previously discussed, the work presented herein is only a portion of the final Thor system. Now that the capture and stroke extraction foundation is complete, emphasis can be placed on the stroke indexing and searching aspect of the project. As a core of set of requirements, the system must be scale, rotation and stroke order invariant (a requirement methods such as the one described in [5] do not fulfill). Additionally, it must be able to cope with queries large datasets. A hierarchical approach such as the one suggested by [3] may be appropriate in order to be able to quickly cull out irrelevant entries.

References

[1] Michael Garland and Paul S. Heckbert. Surface Simplification Using Quadric Error Metrics. *In Proceedings of SIGGRAPH '97* (1997)

[2] Patrick Hew and Michael Alder. Strokes from Pen-Opposed Extended Edges. (1999)

[3] Wing Ho Leung and Tsuhan Chen. Hierarchical Matching for Retrieval of Hand-drawn Sketches. *In ICME 2003 Proceedings* (2003)

[4] Ke Liu, Yea S. Huang, and Ching Y. Suen. Robust Stroke Segmentation Method for Handwritten Chinese Character Recognition. *In International Conference on Document Analysis and Recognition 1* (1997)

[5] D. Lopresti, A. Tomkins and J. Zhou. Algorithms for Matching Hand-drawn Sketches. *Progress in Handwriting Recognition* (1997)

[6] Elizabeth D. Mynatt, Takeo Igarashi, W. Keith Edwards, and Antony LaMarca. Flatland: New Dimensions in Office Whiteboards. *Proceedings of CHI'99* (1999)

[7] Metin Sezgin, Thomas Stahovich, and Randall Davis. Sketch Based Interfaces: Early Processing for Sketch Understanding. *Proceedings of PUI2001* (2001)

[8] P.S.P. Wang, and Y.Y. Zhang. A Fast and Flexible Thinning Algorithm. *Proceedings of CHI'99* (1999) *IEEE Transactions on Computers 38*, 5 (1989)