Thor: Efficient Whiteboard Capture and Indexing

Mihai Parparita Szymon Rusinkiewicz

Princeton University
{mparpari, smr}@cs.princeton.edu

May 3, 2004

Abstract

We present the Thor project, a system that uses consumer-grade digital cameras to record whiteboard content, and then stores the resulting strokes in a space-efficient, vector format. This acquisition portion then serves as the foundation that enables captured data to be indexed and then searched using graphical queries. The query process uses proxy shapes to provide fast, efficient matching. The system is tested on a generated dataset, and results and an evaluation are presented.

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

1 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 [11]. Blackboards have long been used as instructional tools, possessing the advantage over pre-made slides that material can be constructed on the spot, better showing the process by which the information presented was arrived at. The trade-offs of using a whiteboard are intransience and lack of concrete organizational schemas. 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 [11, 15]. Using a stroke capture mechanism, they work around the intransience problem. To help with the limited space issue, [11] 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 searching portion of the organizational side. Previous systems used enhanced capture environments such as Smartboards (in [11]) and the Mimio product (in [15]), 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 implements a scalable database-backed storage mechanism that can be easily queried for information. Strokes are stored, along with "signatures" (specifically, distributions of certain stroke attributes) that allow us to compare them with a graphical query. Since direct comparisons between the query strokes and all of the items in the database are prohibitive, we introduce the concept of a "proxy shape." All of the stored strokes are compared to a small set of these proxies (very basic strokes such as squares and circles) in a precomputation step (done at capture time or offline). The query strokes are compared to the same shapes, and then the items in the database that have same distances to the proxies are selected. Searching is thus reduced to simple numerical comparisons (specifically, range queries) that can be very efficiently implemented.

The rest of this paper provides an overview of the system and evaluates its performance on real-world data. Section 2 describes how data is captured and converted into strokes, while sections 3 and 4 show how the resulting strokes are respectively indexed and searched through. Section 5 describes how the storage of strokes is implemented. We perform a real-world evaluation of the Thor system, looking at it from both a performance point of view (section 6) and a search quality one (section 7). Finally, we discuss some of our findings in section 8 and provide further directions to explore in section 9.

2 Importing Strokes

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, removal of occluders (e.g. people), thinning of captured strokes, further image cleanup, and finally conversion of stroke pixel data into control points.

2.1 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 the 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.

2.2 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. This allows whiteboard meetings to be recorded without a computer being present, thus increasing portability of the entire system. In the future, cameras that support remote control could be used to record these meetings in real-time.

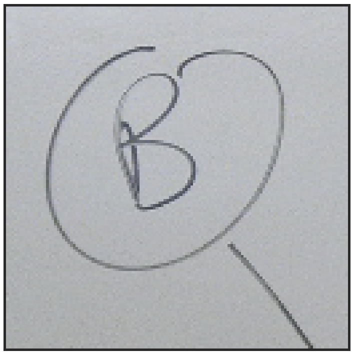


Figure 1: Sample captured image

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 from a higher one), of being faster to process, and of letting the camera store more images without need for periodic downloading to a computer. Most cameras use USB 1.1 connections (with a transfer rate limit of 12 Mbps) and even if that bottleneck were to be removed, the limiter would then be the storage media, which usually has speeds of around 3 MB/s. This prevents capture from being done much more often than once every 10 seconds. Figure 1 shows an subsection of a captured image.

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.

2.3 Differences

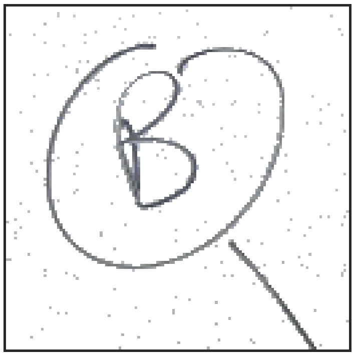


Figure 2: Speckling in difference image

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 (eight-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 two, and it has no neighbors with values greater than or equal to two, 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 seven or eight 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.

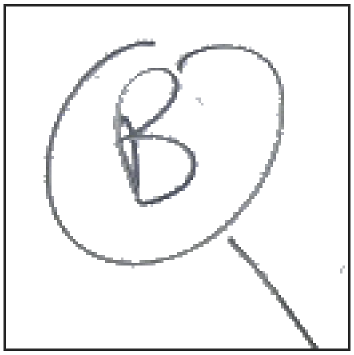


Figure 3: Despeckling figure 2

Once the differences are computed, the 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 [5]). 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.

2.4 Thinning

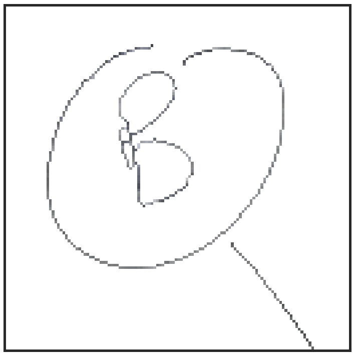


Figure 4: Results of thinning

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 [17]. 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:

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. [17] does not specify an exact iteration termination function, so one had to be devised for Thor. This is not as simple as seeing if the same pixel was visited twice, since it could be encountered from one direction and then another, or one portion of the stroke may be much thicker, necessitating many more passes. We therefore keep track of the pixel immediately following the last deleted one, as well as its successor. If we encounter this pair (in the same order) twice, it means that we have done a full loop without deleting any more edge pixels, therefore we are done with this contour loop.

2.5 Stroke Extraction

Once we have the set of thinned stroke pixels, 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 [15] 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.

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 matter of picking any pixel, and then following its non-background neighbors one 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 [7], 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 5. 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 5. 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:

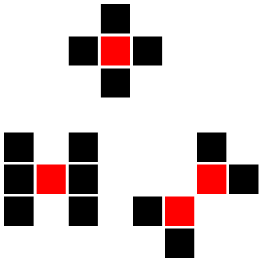


Figure 5: Crossing pixels that must be visited more than once

Transitions	Non-bg. Neighbors	Visits
0	*any*	0
1	*any*	1
2	< 5	1
2	≥ 5	2
3	< 3	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 $d \times k + d' \times (1.0 - k)$

If we do not find a new stroke pixel in the expected direction, then we first test the pixels immediately preceding and succeeding that direction, followed by the ones two pixels away, and so on.

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 [3]. We define the simplification 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 (how many original pixels does the interpolated stroke fall on) 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 may 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 (i.e. difference in overlap) divided by the stroke length (to normalize) grows beyond a certain limit.

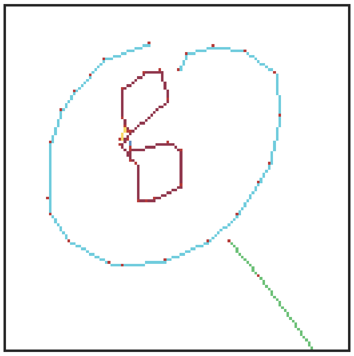


Figure 6: Stroke extraction outcome

2.6 Hair Removal

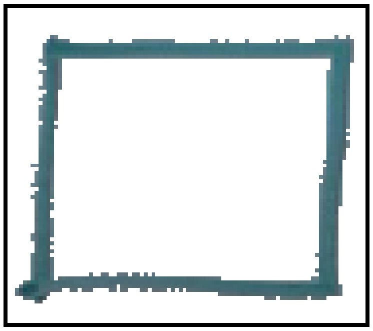


Figure 7: Difference image with ragged edges

One issue with the described stroke extraction algorithm is that it is vulnerable to being mis-led by spurious thinned portions we call "hairs." As it can be seen in figure 7, the difference image has a ragged enough boundary to throw off the thinning algorithm, the results of which are shown in figure 8. This is caused by the aforementioned jagged edges protruding by two or more pixels. When the stroke extraction extraction algorithm is applied to this image, it can tend to follow these hairs when the fall in the same direction as most of the previous stroke.

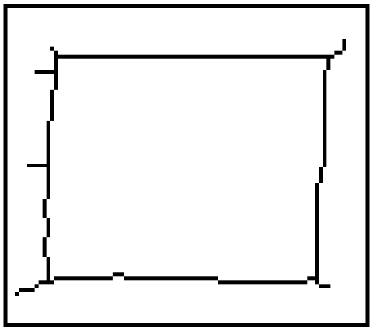


Figure 8: Thinned image exhibiting "hairs"

For example, if we are proceeding in a clockwise fashion and we get to the lower-left corner of the rectangle, we would be much more likely to follow the hair (and reach a dead end) as opposed to making the turn and continuing along the other side of the rectangle. This results in shapes that would intuitively be described with only a single stroke being separated into several. Furthermore, because these hairs are caused by what are initially small protuberances in stroke outlines, it can mean that seemingly similar input data can be decomposed into very different strokes. This lack of consistency can impact the accuracy of the indexing and searching that we do in later phases.

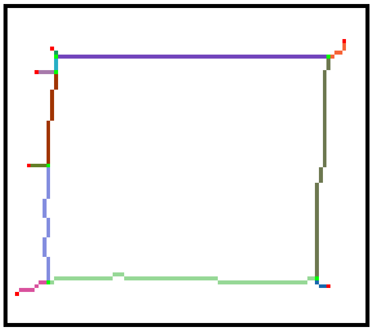


Figure 9: Results of graph building

In order to remove these "hairs," we chose to implement a graph-based method, somewhat similar to the one described in [7]. We create a graph where (future) stroke crossings are vertices and edges represent vertices that are reachable by following previously unvisited pixels. To determine what constitutes a crossing, we use the same definition as we do in section 2.5, when determining how many times a pixel can be visited for stroke extraction. We also differentiate between vertices at the periphery of the graph (with only one edge connected to them) and inner ones (with more than one edge connected to them). Figure 9 shows the results of this graph construction, with peripheral vertices being drawn in red, inner ones in green, and the pixels that make up the edges connecting them in random colors.

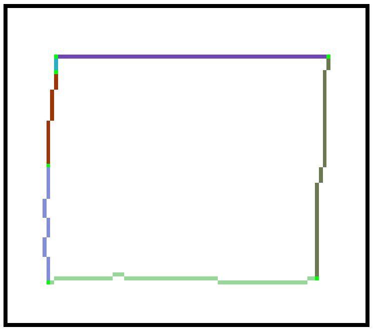


Figure 10: Removal of "hair" edges

Using this graph structure we can very easily remove the pixels that make up the unwanted hairs. One key characteristic of a hair edge is that must contain at least one peripheral vertex. However, we cannot remove all such edges indiscriminately. Instead, we first compute the distribution of edge lengths (where the length is defined by the number of pixels that make up each edge, not the Euclidean distance between its vertices). Only those edges with a peripheral vertex and with a length below a certain threshold value are actually removed (they removed both from the graph and from the image - their corresponding pixels are replaced with the background color). This is an iterative process, since removal of an edge can convert what was previously an inner vertex into a peripheral one, thus uncovering more hairs. Figure 10 shows the results of this operation. This image can then be run through the stroke extraction step, and results in a single stroke being used to describe the entire shape, as one would intuitively expect.

2.7 Stroke Normalization

Some of the ways in which we determine characteristics of a stroke depend very heavily on the arrangement of the points that are used to describe it. Unfortunately these points, immediately following the stroke simplification step, are not evenly distributed. One can image two large rectangles, one unadorned and one with a complex flourish in one of its corners. The former may be described by as few as as four points, while the other may have many more, with those that make up the flourish dominating, count-wise. However, the two strokes are perceptually more similar than different, and so the unbalanced effect of the flourish must be compensated for. We can achieve this by normalizing the number of points that are used to describe a stroke. By doing this normalization so that the points are evenly spaced, we can ensure that things like the aforementioned flourish do not have an undue influence on any "signature" that we compute for the stroke.

The simplest way to do this normalization would be to consider the stroke as a parametric function $f(t)$ for t from 0 to 1, and to then resample it at even intervals of t. However, doing so may significantly impact the appearance of the resampled stroke when compared to the original. If we consider a stroke with many sharp angles, it is possible that these corners would be clipped off if a resampled point doesn't fall close enough to a corner point. Therefore, we sacrifice some of the regular sampling in order to better maintain stroke appearance.

Taking this detail-preserving constraint into account, our stroke normalization algorithm works as follows (assuming we are trying to normalize down to N points):

1. Let D be the total length of the stroke, as measured by adding up the Euclidean distances between each of its points.

2. Build a list L of all "critical" points, where "critical" is defined as having an inner angle of less than a threshold value.

3. Iterate over each pair of sequential critical points $p1$ and $p2$ in L.

4. Let d be the distance between $p1$ and $p2$.

5. We must then insert n points between p and $p2$, with $n = d/D \times (N - 1)$.

6. Insertion between $p1$ and $p2$ is done normally, with the points being regularly spaced and linearly interpolated.

In addition to normalizing the number of points, we also normalize for size (we take the larger of the X and Y dimensions and then scale all points such that that dimension is of a known length) and position (the upper left corner of the bounding box that encompasses the stroke points is moved to $(0, 0)$.

It should be noted that stroke simplification is necessary even if we later normalize strokes so that they have the same number of points. Had we left the stroke unsimplified (every pixel in the thinned image had a corresponding stroke point) it would have been very difficult to determine what the critical points were, since all inner angles would have been either 90 or 180 degrees. We could perhaps have measured the the ratio between the Euclidean distance and the distance along the stroke between two random points, but even for a very low ratio we would only have known that somewhere between those two points there were significant curves or corners, but not exactly where (i.e. which point was critical).

3 Extracting Stroke Characteristics

Now that we have placed strokes such that they are of a known point count and size, we can begin to extract certain "signatures" that allow us to compute numerical similarities between them. The assumption is that a user's query also takes the form of strokes that are also passed through the same normalization step. In the current incarnation of the system, the user can draw the query using his or her mouse, an input device that has different characteristics (when it comes to drawing) from the markers that were used to create the strokes in our database. Therefore, we also support the loading of pre-drawn queries, thus the whiteboard may also be used as a query input mechanism.

Our goal in choosing stroke signatures was to have things that were independent of stroke drawing order (unlike [8]), position, scale and rotation. For example, one aspect that we looked at was the distribution of stroke inner angles (the angles formed by the two segments on either side of a stroke point). One can imagine that a (perfect) circle would have all of its values concentrated in a bucket around the 180 degree mark (depending on how many points it was sampled with) while a square would have most of the values in the 180 degree bucket with a few in the 90 degree one. Obviously this characteristic alone does not differentiate between all shapes (e.g. all rectangles, regardless of aspect ratio, would have the same signature), but it is hoped that by combining several such weak signatures we can determine a stronger metric for shape similarity, in much the same way that [6] proceeds.

3.1 Shape Signatures

In addition to the previously mentioned inner angle distribution (from here on referred to as *I3*), we have chosen several other distribution-based signatures. They are based on the work done in 3D shape searching described by [12], specifically:

- *D2*: Distances between two random points
- *D3*: Square root of the area of the triangle defined by three random points
- *A3*: Angle between three random points

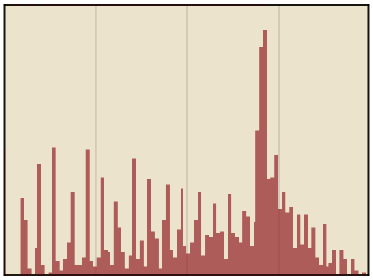


Figure 11: *D2* distribution with period spikes

One of the key words in the preceding list is ”random.” In our initial implementation, we defined random as ”any normalized stroke point.” However, due to the even spacing that we enforced in the normalization step, we began to see periodic spikes in the distribution. Since these were reflective of our resampling technique and not of any stroke characteristics, they would negatively impact attempts to compute stroke similarities (since they would be present in all distributions, they would falsely boost the similarity amounts). This can be seen in figure 11 which shows the *D2* distribution for a square. Although we have the expected spike around the 3/4 mark (due to points on opposite sides being picked), its emphasis is decreased due to the other period peaks.

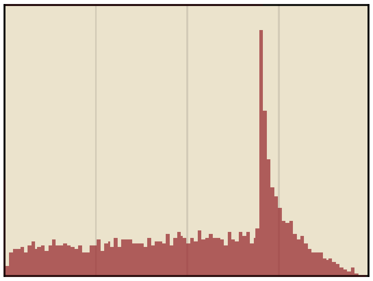


Figure 12: Smoother *D2* distribution

In contrast, figure 12 (same distribution, same shape) exhibits much better characteristics. Here, instead of picking only stroke points, we can pick any spot along the stroke. Generally speaking, this truly random sampling is used for the *D2*, *D3* and *A3* distributions, while *I3* only samples at the normalized stroke points.

We currently use 100 buckets for all distributions, and since we normalize for stroke scale, even the *D2* and *D3* distributions have a known upper bound (for the normalized scale S they are $\sqrt{2} \times S$ and $\sqrt{S \times S}/2$ respectively). When doing random sampling we take 10000 such samples, a number that is a good trade-off between getting enough so that there is little variance from run to run and not having too many so that computing the distribution is not too time-consuming.

3.2 Distribution Distances

Now that we have these distributions for each stroke, we need to able to compute a degree of similarity between them, so that we can in turn determine how similar their parent shapes are. The simplest, most intuitive way of computing distances between histograms is to use a simple sum of squared differences measure. Since both histograms that we would compare have the same number of buckets, we can just loop along both of them, seeing what the differences from bucket to bucket are. Unfortunately this only provides an accurate metric when differences are very slight. One can consider three histograms, one with a large spike at bucket b, one with a similarly proportioned spike at bucket $b + 1$ (and all other buckets being of the same value), and finally one with the same spike at $b + 50$. The sum of square difference metric would report the distance between the first two as being the same as that between the first and third.

Instead, we use the Earth Mover's Distance (EMD) metric, as described in [13]. EMD works by mapping bucket distances to the solution of a transportation problem. Effectively, we consider one histogram to be the supply while the other represents the demand. We can then create paths between the supply and demand points, with the cost of the path being a function of the distance between the two buckets (our chosen distance metric is $1.0 - e^{-d}$ for d being the difference between the bucket indices). The values of the buckets in each histogram give us the supply and demand at each point. The transportation problem is a linear optimization problem and has well known iterative solutions. Specifically, we use the simplex method as implemented in the IBM OSL solver (part of the COIN library as described in [14]) and provided through the FLOPC++ modelling library. The EMD metric, though much more complex to compute, would accurately differentiate between the two previously mentioned cases for which sum of squared distances would report the same distance.

3.3 Proxy Shapes

The previously described EMD metric provides us with a reasonably accurate way of determining shape similarity (especially when used with all four distributions that we have). However, it does have the trade-off of performance. Some distributions such as *I3* are normally very sparse and thus need few paths between supply and demand buckets. However, others such as *D2* (as shown in figure 12) can encompass almost the full breadth of the distribution, necessitating the full $N \times N$ set of paths. Thus, computing the EM distance between the query shape's distributions and the distributions of a reasonably large dataset would take a prohibitive amount of time. Part of this slowdown is due to the toolkit that we use to compute EMD distance - FLOPC++ is a general purpose library, written with ease of use in mind as opposed to high performance. [12] reports sub-millisecond times for computing dissimilarity, whereas the values we see are closed to a tenth to a quarter of a second. However, even if we were to use a highly-optimized function, we would still only see a linear increase in performance, and the upper limit on the dataset would still be low.

To alleviate this problem, we propose the use of "proxy" shapes that can be used as intermediaries in the distance computation process. When strokes are loaded into the dataset, distances from each stroke's distributions to those of the proxy shapes are computed and stored. The process is repeated for the query strokes, and in order to provide a result set we simply look for strokes with similar distances.

We currently use two proxy shapes, a circle and a square, since they have reasonably distinctive distributions, as can be seen in figure 13. In the future more shapes could be added, since the performance cost of doing more comparisons is minimal.

4 Handling Queries

As previously mentioned, proxy shapes are used to determine which items in the database match the user's query. The process when importing strokes is as follows:

1. Proxy shapes P_1, P_2, ..., P_n are loaded (currently just a circle and a square, thus $n = 2$)

2. For each basic shape i we compute histograms H_{i1}, H_{i2}, ..., H_{im} (the current distributions are *I3*, *D2*, *A3*, and *D3*, thus $m = 4$)

3. When loading stroke data, we compute for each stroke j m histograms H_{j1}, H_{j2}, ..., H_{jm} along with the distances (using the EMD metric) between H_{i1} and H_{j1}, H_{i2} and H_{j2}, ..., H_{im} and H_{jm} (each loaded stroke's histogram has n distances, one to each of its counterparts in the basic shapes)

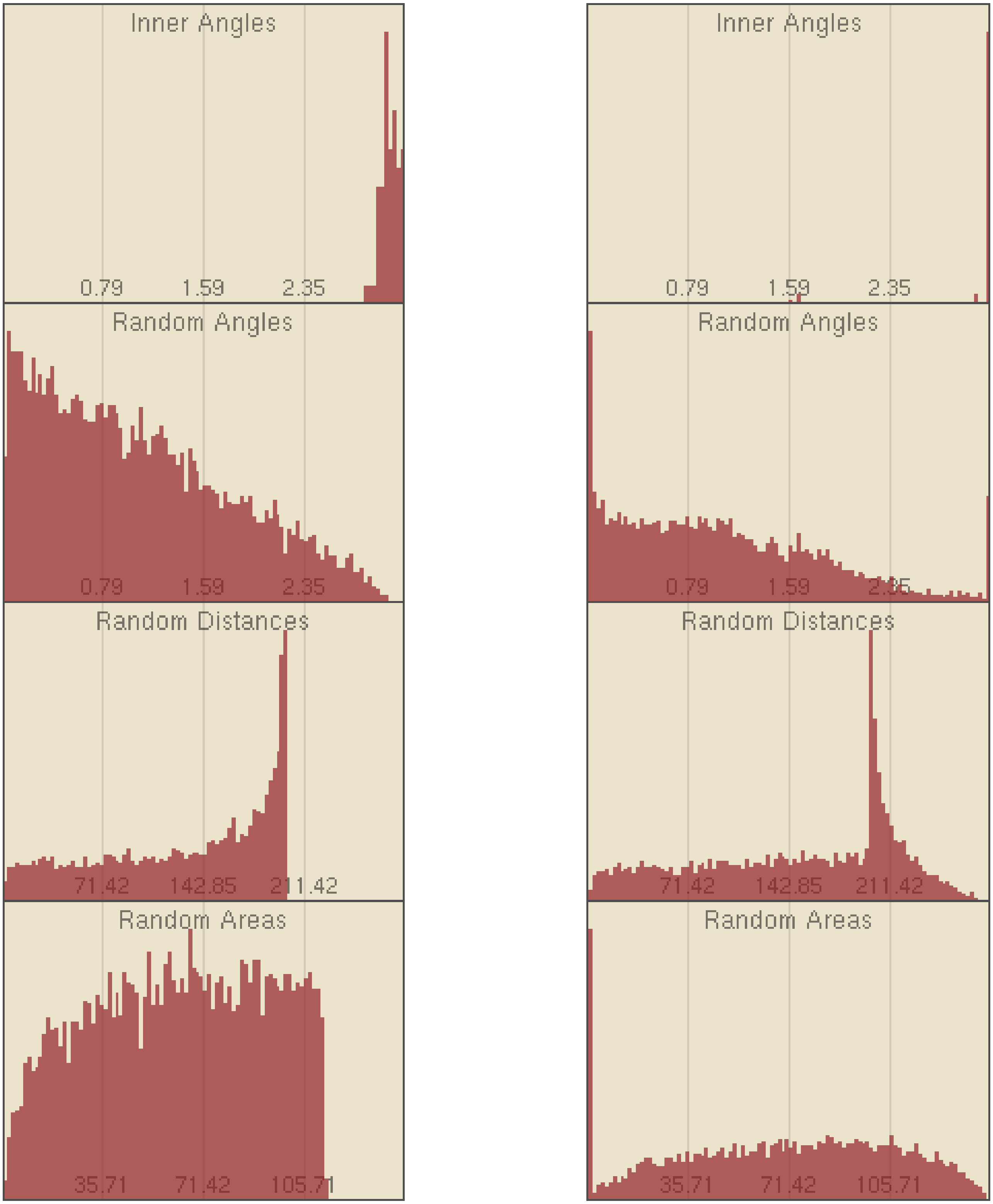


Figure 13: The distributions for the two chosen proxy shapes, a circle (left) and a square (right)

4. All of the above (strokes, histograms, distances) are stored. Distances are represented as ratios, e.g. if stroke histogram H_{j1} had distances of 12 and 36 to basic shape histograms H_{i1} (for i from 1 to n) then we would store 0.25 and 0.75 as the distance values.

When performing a query, we go through the following steps:

1. For the query stroke q, we compute its histograms H_{q1}, H_{q2}, ..., H_{qm}
2. For each of those histograms, we compute the distances to their counterparts in the basic shapes, and determine the ratios as described above.
3. We do a query, looking for strokes in the dataset whose histograms had similar distance ratios to the basic shapes (currently ±0.3).
4. This is done per histogram type, and then if a certain portion of the histograms agree that a stroke was in the same distance range as the query (currently 3/4), then we include it in the result set.
5. We can then see what the parent shapes of the matching strokes are, and pick those shapes that have the most matches.
6. Ranking can be accomplished by ordering shapes so those with strokes that have the closest EMD to the query strokes are picked as being the most relevant. An alternative to consider is, since we are presumably dealing with a much smaller subset of our database, we can afford to do direct EMD comparisons between the query strokes and those in the result set.

5 Database Storage

All of the extracted strokes are stored in a relational database that allows us to easily find items, as described in figure 14. We specifically use the MySQL database since it is easily deployable and has satisfactory performance when performing range queries. Furthermore, the use of a database system gives us distributed storage for free - as long as all instances of the Thor program are connecting to the same server, it is possible to do queries across strokes captured in several different environments. Within the database, data (e.g. stroke points and histogram buckets) is stored in a very simple manner: a count followed by the actual values. For more complex data storage, it should be possible to switch to a more structured arrangement, such as the InkML XML data format, described in [18].

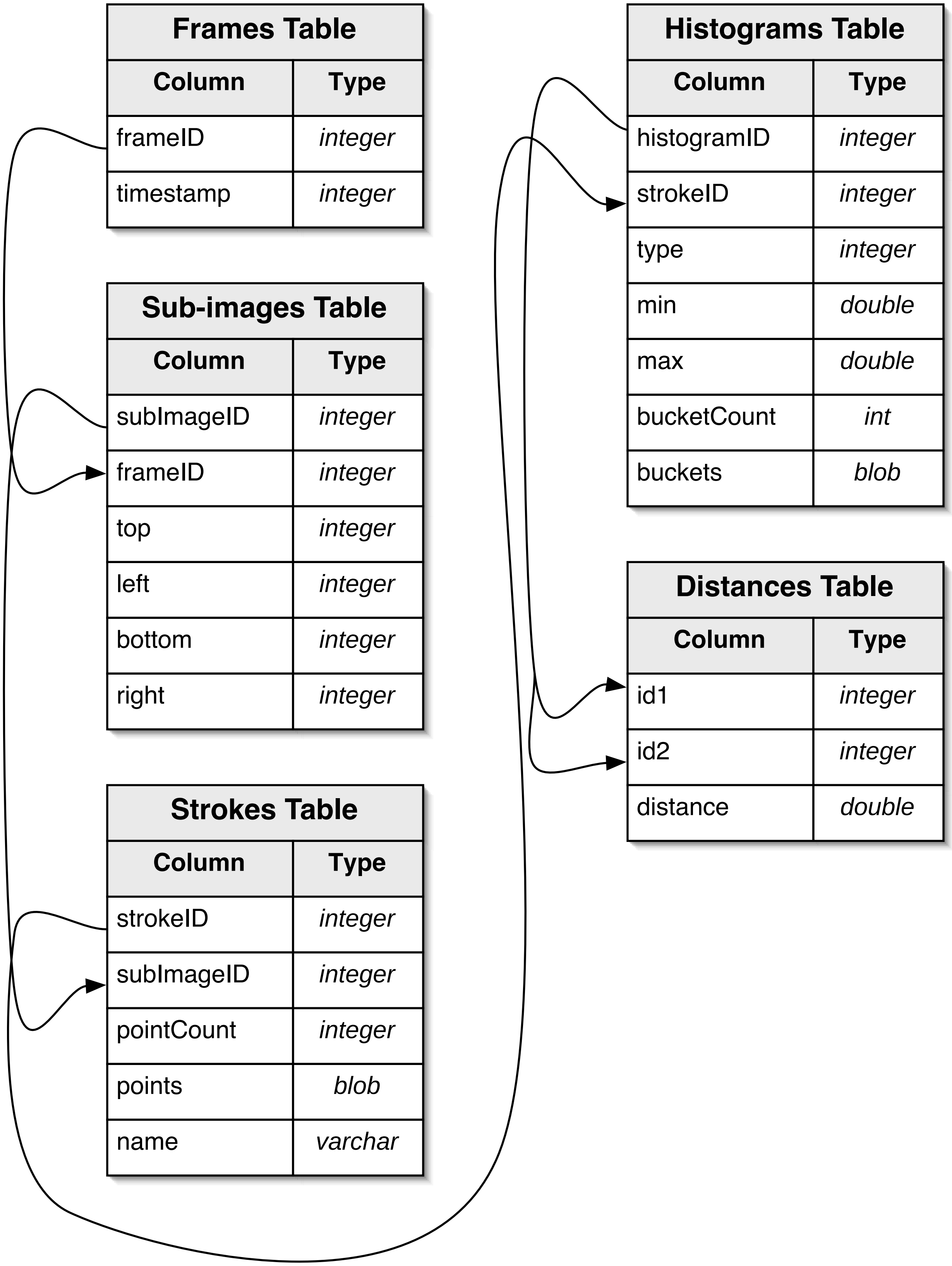
6 Performance

The Thor system is conceived to support real-time operation (e.g. images are captured, and strokes are extracted as a whiteboard meeting goes on). As a result, performance is a criterion to consider.

6.1 Capture Performance

Using the current codebase, on a 1 GHz G4 processor, with an input set of 100 400 x 400 pixel images (average of 10.2 strokes per frame), end-to-end processing time measured was 9.64 seconds per frame. This falls just below our limit of 10 seconds per frame, and with a bit more optimization, will allow image capture to be done. This is because although image capture and transfer from the camera take a significant amount of time (on the order of seconds), both are I/O-bound operations, and thus can be done in parallel with the importing with minimal impact. 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 1.2% of the time, computation of differences 6.8%, thinning 9.9% (with ”hair” removal responsible for 3.9%) and stroke extraction 82% (as measured with a statistical sampling tool). Of the stroke extraction time, the bulk (68% of the total time) is spent performing the stroke simplification operation. Since we immediately normalize strokes following the simplifcation step, it may be worthwhile to investigate a simpler (if less accurate or efficient, point-wise) algorithm that has better performance characteristics.



Frames Table

Column	Type
frameID	*integer*
timestamp	*integer*

Sub-images Table

Column	Type
subImageID	*integer*
frameID	*integer*
top	*integer*
left	*integer*
bottom	*integer*
right	*integer*

Strokes Table

Column	Type
strokeID	*integer*
subImageID	*integer*
pointCount	*integer*
points	*blob*
name	*varchar*

Histograms Table

Column	Type
histogramID	*integer*
strokeID	*integer*
type	*integer*
min	*double*
max	*double*
bucketCount	*int*
buckets	*blob*

Distances Table

Column	Type
id1	*integer*
id2	*integer*
distance	*double*

Figure 14: Database tables showing column names, types and relations

6.2 Searching Performance

Once the strokes are captured, they are ready to be indexed so that they may be later searched through. Importing of all the strokes that we described in the previous section resulted in a running time of 221.2 seconds per frame (21.68 seconds per stroke). Of this time, 29% was spent computing the four distributions for each stroke, and 71% determining the distance from each distribution to their correspondents in the proxy shapes (currently just two). As it can be seen, the distance computation takes a significant amount of time, and the indexing phase would greatly benefit from a more optimized method. However, speed in this phase is not critical to the system, since indexing can be performed off-line, long after capture is completed.

Since searching is composed of similar operations, we can expect similar running times. A query with three strokes run against a database of 1,000 shapes took 45.2 seconds. Of this time, 25% was spent computing the distributions for the query strokes, 46% determining distances to basic shapes, 18% doing the actual SQL queries to determine the matched shapes, and 11% loading the results from the database. It is important to note that the bulk of the query time is spent computing the distributions and distances for the query strokes, the number of which is not likely to go up significantly. By comparison, a small amount of time is spent going through the actual database, and as its size increases time spent doing computations should increase logarithmically. Therefore, we are confident that the Thor system will have much better scaling characteristics than other systems that do more intensive comparisons between the query shape and each item stored in the database.

7 Evaluation

Measuring the efficiency of Thor is a difficult task, especially when trying to see how it performs in relation to other approaches. Papers such as [6] use datasets collected from several volunteers in order to model variances in drawing behavior. However, such an approach is not desirable in our case, since not only does using a non-standard dataset make it difficult to do comparisons, but it is also very difficult to gather a large-enough dataset that we could use for meaningful testing. Such problems are also faced by other projects in fields that are developing, such as the 3D model search engine described in [2], which eventually led to the development of the shape benchmark [16].

In our initial attempts at using a standardized dataset, we considered using handwriting samples that are commonly used to evaluate handwriting recognition systems. Databases such as IAM, described in [9], are freely available. However, they do not quite fulfill our needs. The aims of Thor are slightly different from handwriting projects (since we do not attempt to do any recognition) and thus a direct comparison would be meaningless. Furthermore, handwriting data is often too high frequency to be representative of normal whiteboard content. Finally, measuring accuracy is much more difficult due a lack of a pre-labeled dataset.

The approach we settled on is derived from the evaluation methods described in [10]. We take a reasonably large set of line drawings, in our case 249 images of objects provided by the psychology labeling project described in [1]. For each of these drawings we generate 20 variants that are representative of drawing differences from user to user. To provide these variants, we apply to each image 20 randomly generated low-frequency displacement maps, as shown in figure

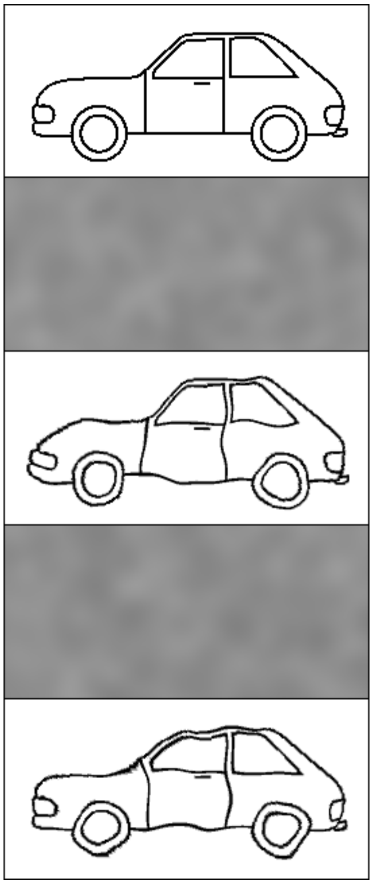


Figure 15: Source, displacement maps and variants

15. Since all of the original drawings were of different objects, we can very easily determine whether a match is correct (a variant of the same shape) or not (another shape).

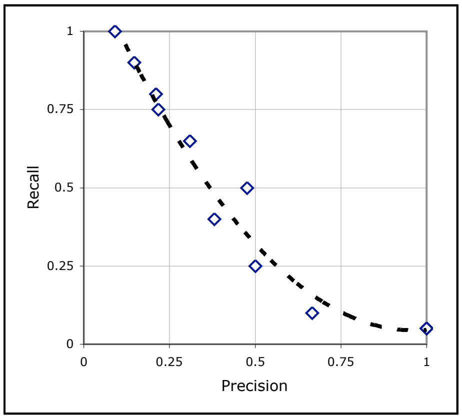


Figure 16: Precision vs. Recall

Figure 16 shows the results of various searches on this generated dataset. We generate various precision/recall combinations by varying the tolerances described in section 4. Considering the limitations of our system (dealing with off-line stroke capture that has no information about writing order or speed) as well as our aims (writer independence and invariance to rotation, scaling, translations and small non-linear transformations), our system does not perform so poorly. Overall the system has an error rate (for all the searches conducted, the number of incorrect matches over the total matches) of 56%. As a point of reference, systems such as the shape context one described in [10] have error rates ranging between 2% to 13% for similar datasets.

8 Discussion

The capture side of 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 (otherwise auto-focus can choose a different focus point depending on activity in the scene, thus affecting the appearance of the whiteboard strokes). The latter two issues could be compensated for by extra computation; it remains to be seen if such a trade-off is worthwhile.

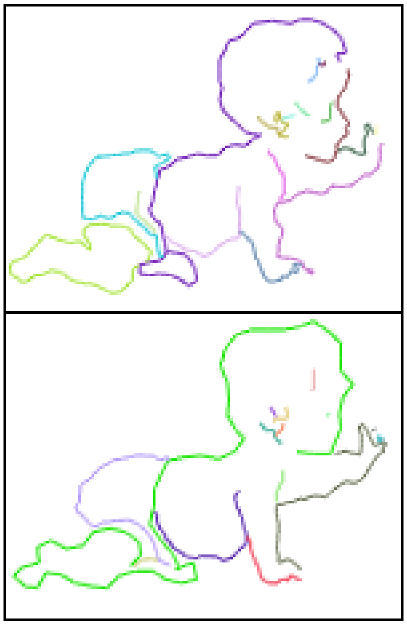


Figure 17: Stroke segmentation differences

The approach we chose does matching based on strokes, and then the shapes with the most matched strokes are presented as the results. This has the advantage of supporting partial matches, i.e. if the query represents only a smaller piece of a shape in the database, it will still be found. This can be contrasted with the approach used in [10] and other papers, where distributions are computed and matches are done directly at the shape level. The trade-off is that we must have a very robust stroke segmentation algorithm, otherwise supposedly similar shapes will not be matched (figure 17 shows an example of this, with different strokes being drawn in different colors). This explains the need for such steps as the ”hair” removal described in section 2.6, as well as the robust thinning algorithm that we chose. Further work can be done in this area, for example by removing or simplifying high-detail areas, which are the locations that are most apt to confuse the stroke extraction process.

Our result weighing system is currently rather coarse. We do not take into account such factors as spatial distribution (whether the arrangement of the query strokes matches the arrangement of the ones in the result) or weigh larger strokes any differently from ones with smaller areas. We also do not weigh the different distributions based on their effectiveness (e.g. [12] found that the *D2* distribution is the most robust when dealing with 3D models). Evaluating the Thor system on an even larger set of data would enable us to determine the appropriate weighing factors.

9 Future Work

Now that we have developed this capture and indexing foundation, we can begin to build on it. For example, currently all strokes and shapes are treated equally, with no semantic meaning attached to any of them. We can begin to differentiate between strokes, using the drawing style itself as the differentiation mechanism. This is similar to the way in which [11] allows the user to maintain a to-do list. However, one limitation is that we are dealing with a one-way medium, thus any changes that the user would make would not be reflected on the whiteboard itself ([11] works around this by using a projector to display changes that the user makes away from the whiteboard). We can also build more specialized recognizers on top of the Thor system, such as [4] which recognizes shapes that can be used to define UML class diagrams.

The most obvious expansion of the system would be to add more distributions and proxy shapes. In the case of the former, one can look for more complex descriptors, such as the shape contexts described in [10]. Shape contexts refer to the distance angle distributions to/from n points on a shape. Such distributions are more complex than the simple 1D histograms that we have worked with this far. We can either simply support this 2D distribution type, or we can reduce it to 1D by first doing a k-means cluster analysis of all distributions encountered in our dataset, and then simply storing the distribution of these clustered values (one per sampled point) for each shape.

In order to achieve the more robust stroke segmentation that we need, we can incorporate more of the graph-based stroke clean-up and extraction methods described in [7]. We can also support distributions at the shape level, giving the user the option of using one or the other when conducting a query.

References

[1] Analia Arevalo. Teasing Apart Actions and Objects: A Picture Naming Study. *Center for Research in Language Newsletter 14*, 2 (2002)

[2] Thomas Funkhouser, Patrick Min, Michael Kazhdan, Joyce Chen, Alex Halderman, David Dobkin, and David Jacobs. A Search Engine for 3D Models. *ACM Transactions on Graphics 22*, 1 (2003)

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

[4] Tracy Hammond, and Randall Davis. Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams. In *Proceedings of AAAI Spring Symposium on Sketch Understanding* (2002)

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

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

[7] 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 Proceedings* (1997)

[8] Daniel Lopresti, Andrew Tomkins, and Jiangying Zhou. Algorithms for Matching Hand-drawn Sketches. *Progress in Handwriting Recognition* (1997)

[9] Urs-Viktor Marti, and Horst Bunke. The IAM-database: an English sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition 5*, 1 (2002)

[10] Greg Mori, Serge Belongie, and Jitendra Malik. Shape contexts enable efficient retrieval of similar shapes. In *Proceedings of Computer Vision and Pattern Recognition* (2001)

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

[12] Robert Osada, Thomas Funkhouser, Bernard Chazelle, and David Dobkin. Shape Distributions. *ACM Transactions on Graphics 21*, 4 (2002)

[13] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. The Earth Mover's Distance as a Metric for Image Retrieval. *International Journal of Computer Vision 40*, 2 (2000)

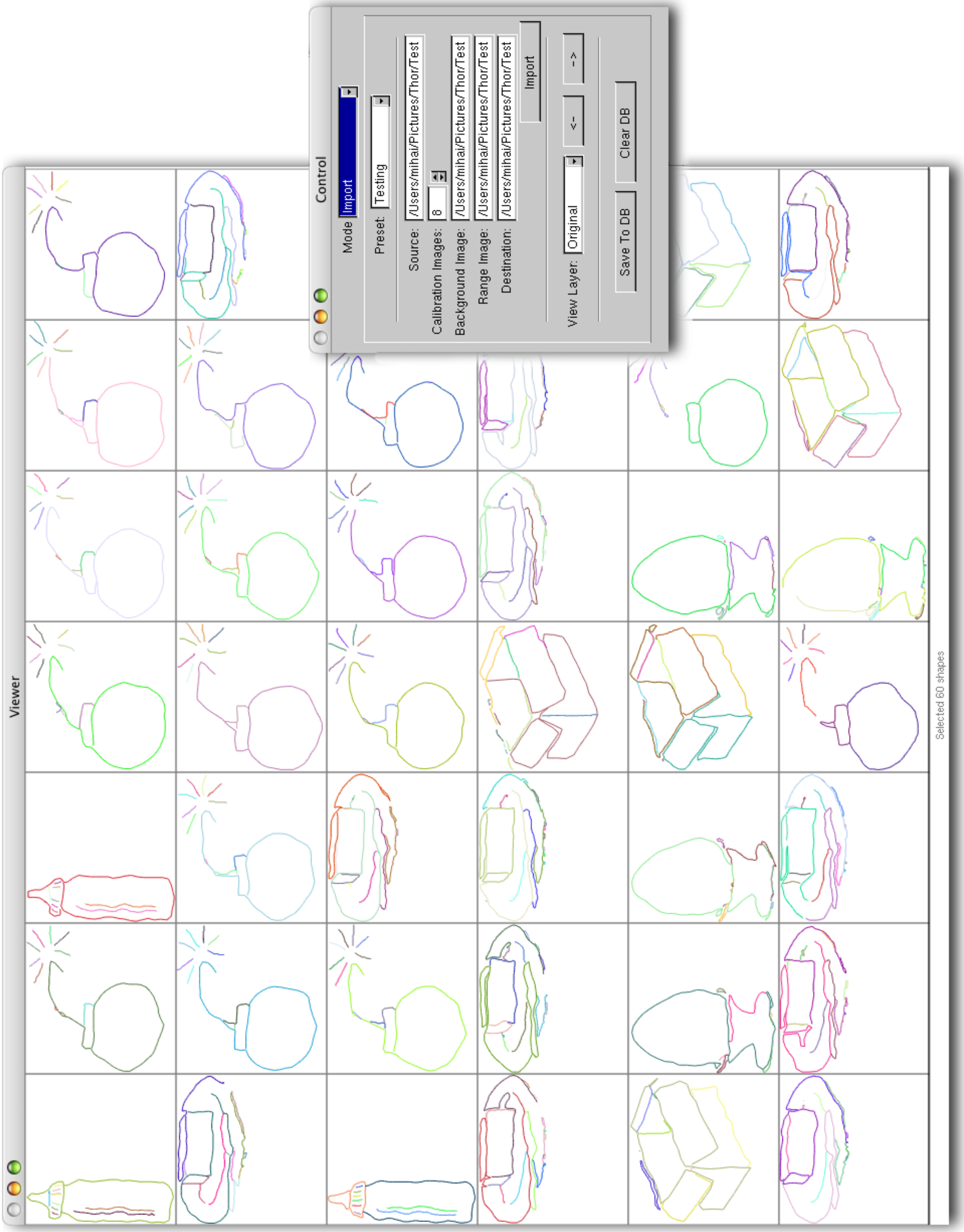
[14] Matthew J. Saltzman. COIN-OR: An Open-Source Library for Optimization. *Programming Languages and Systems in Computational Economics and Finance* (2002)

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

[16] Philip Shilane, Patrick Min, Michael Kazhdan, and Thomas Funkhouser. The Princeton Shape Benchmark. In *Proceedings of Shape Modeling International* (2004)

[17] P.S.P. Wang, and Y.Y. Zhang. A Fast and Flexible Thinning Algorithm. *IEEE Transactions on Computers 38*, 5 (1989)

[18] World Wide Web Consortium. Ink Markup Language. *W3C Working Draft*, (2004) http://www.w3.org/TR/InkML/



Mode Import

Preset: Testing

Source: /Users/mihai/Pictures/Thor/Test

Calibration Images: 8

Background Image: /Users/mihai/Pictures/Thor/Test

Range Image: /Users/mihai/Pictures/Thor/Test

Destination: /Users/mihai/Pictures/Thor/Test

View Layer: Original

Selected 60 shapes

Figure 18: Screenshot of the Thor interface