

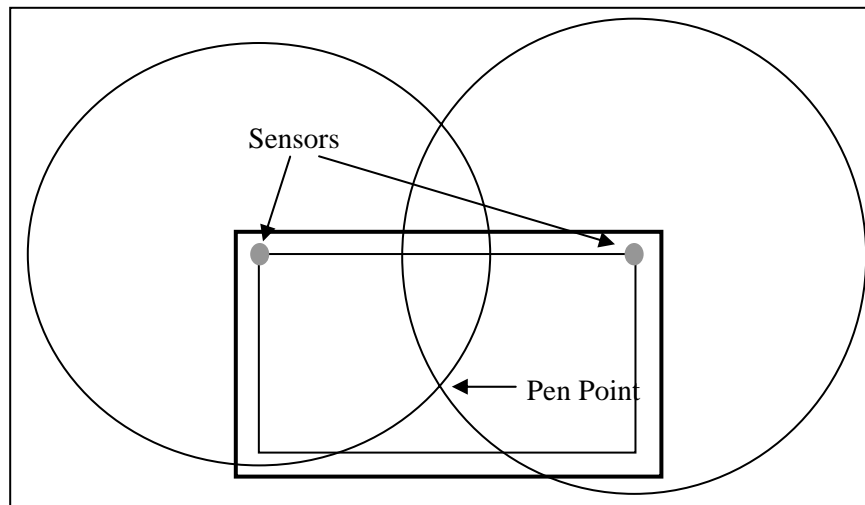
# Digital Ink

Since digital interaction became possible people have wanted to “draw on the screen”. In this chapter we will look at the technologies of digital ink and the various ways it can be used and processed. For most graphical user interfaces the mouse is the input device of choice because it is cheap and easy to grasp when moving to and from the keyboard. However, the mouse is a very poor device when drawing on the screen. The speed and accuracy of human movement generally depends upon the mass of the muscle groups involved and the mass of the body part being manipulated. The mouse is manipulated with the wrist and forearm and thus is relatively slow and inaccurate. For drawing on the screen, the pen or stylus is the preferred instrument because it uses only the fingers which are much more dexterous. A stylus uses the small-mass finger bones manipulated by relatively large-mass muscles in the forearm. This approach also exploits the training derived from pen and ink. Some systems remove the pen and simply use the finger itself as the input device.

This chapter will first discuss various pen-input technologies followed by some basic algorithms for processing strokes of digital ink. Gesture recognition will then be introduced as a basis for many interactive techniques. Digital ink also has a property of allowing very free-from modes of expression. As such, several systems have been developed that use digital ink as their basic data type. The drawing metaphor of digital ink has lead to several systems that exploit it for sketching tools. Lastly we will discuss the use of digital ink for annotating existing material in the user interface.

## Pen input devices

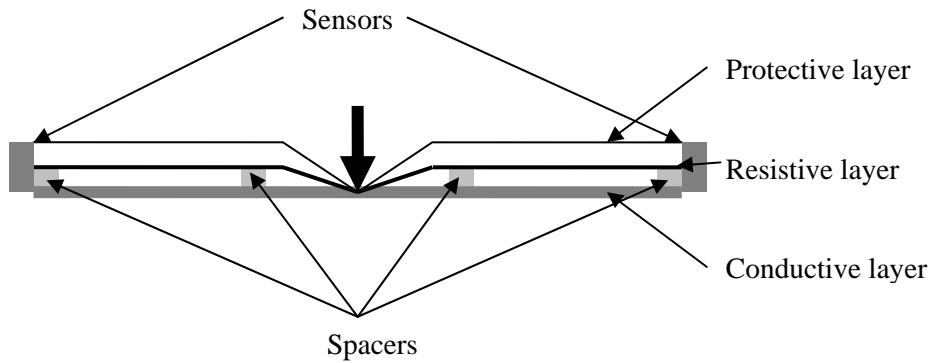
Most pen input technologies are either distance, projection-based or position encoded. The distance techniques use some physical phenomenon to measure the distance between the pen and two or more known points. Knowing the distance between the pen and two different points one can compute the intersection of two circles to determine a pen position. Note that the intersection between two circles has two solutions. However, in most instances one of the solutions is easily discarded as shown in figure 1.



**Figure 1 – Distance sensed pen input**

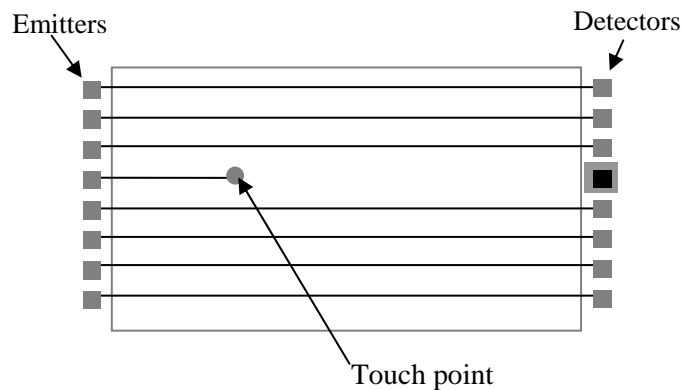
One common distance measure is the speed of sound. The pen simultaneously emits an infrared pulse and an ultrasonic pulse. The infrared pulse, traveling at the speed of light, is essentially instantaneous. The sensors each start a timer until the ultrasonic pulse arrives. The time multiplied by the speed of sound yields the distance between the pen and the microphone sensor. The Mimio<sup>1</sup> input device uses two microphones suction-cupped to a white board at a fixed distance apart. The pens are triggered by pressure against the board. This provides an inexpensive input device for a large area. The inputs are accurate to 1/100<sup>th</sup> of an inch up to 8 feet from the sensor pair.

Many PDA touch pads use a resistive distance technique. A conductive and a resistive layer are placed very close together but separated by flexible spacers. Pressure on the outer layer makes contact between the resistive and conductive layers as shown in figure 2. Before pressure, no current flows. After pressure, the resistance measured at each edge is proportional to the distance between that edge and the contact point.



**Figure 2 – Resistive pen sensing**

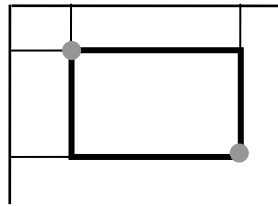
Projection-based techniques sense each X and Y coordinate independently to get a projection of the pen or finger point. A simple technique uses light emission and sensing. As shown in figure 3, a set of infrared light emitters are arranged along one side of the touch area. On the opposite edge are a corresponding set of detectors. The control software activates each emitter/detector pair in turn. If a finger or stylus is in the way, no light is detected at the corresponding sensor and we assume that the touch point is somewhere along the line between the emitter and the sensor. The other coordinate is sensed in a similar fashion using the other edges. Together these provide the X, Y coordinates of a single touch point.



**Figure 3 – Light interference sensing**

Another more common projection-based technique<sup>2</sup> uses two layers of fine wires going in the X and Y directions. Each wire in turn has an AC signal placed on the line. The pen contains an inductor. When the pen is placed close to a wire the inductor modifies the signal, which can be sensed. The wire with the strongest signal modification is the one that is closest to the pen. The pen tip can also have a pressure sensor designed so that pressure changes the inductance. This change in inductance can also be sensed to provide the tablet with information about the pressure on the tip.

The DiamondTouch<sup>3</sup> technique uses capacitive coupling. This exploits the fact that the human body is largely water and makes an excellent dielectric. A grid of small antennas is embedded in a table top. The user sits on a conductive pad that is grounded to the sensor circuit. When the user's finger is on or very close to an antenna, the user's body forms a capacitive circuit with the sensor. The capacitance difference between touching and not touching can be sensed. If different users sit on different pads, they each form a different circuit that can be independently sensed. The antennas are arranged in a grid so that X and Y projections of a touch position can be independently obtained. Projection-based techniques are challenged when multiple touches occur simultaneously. Two touches will produce two signals in X and two in Y. This essentially defines a rectangle. If a rectangle is all that is required, this works great. If, however, the point positions are needed we have ambiguous data. Either pair of opposite corners would explain the multiple touches, as shown in figure 4.



**Figure 4 – Ambiguous touches in projection sensing**

The Anoto<sup>4</sup> pen uses a very different position encoding technology. The tip of the pen is a small camera that views specially printed paper. On the paper there is a pattern of microscopic dots. Each region of the paper has a unique spacing of these dots that can be sensed by the camera. The dots can be preprinted or printed using a laser printer. They give the paper a slightly gray shade. By “reading” the pattern of dots under the camera the pen can identify where it is located in *dot space*. The Anoto dot pattern can generate enough unique

patterns to cover 60,000,000 square kilometers of paper. The Anoto pen is usually coupled with a normal ink pen. This allows users to write on the encoded paper as one would with a normal pen, while the camera digitally logs the same stroke information. By reading the stroke information out of the pen the digital ink can be recovered as well as a unique identification of the page on which the ink was written.

The above techniques are among the most common. Others have been developed. In chapter 24 we will discuss additional camera-based techniques for getting two-dimensional input that can be used for digital ink.

### Stroke Processing

The key data type in digital ink is the stroke, which is an array of points in the same sequence as the user entered them. Frequently the coordinates of each point are accompanied by the timestamp of when the point was received from the input device. The timing of the stroke points is sometimes used in recognition tasks where a particular user's style is important, such as in verifying a signature. For this chapter we will generally ignore timing information.

Digital ink strokes can be produced by capturing all of the mouse-move events between mouse-down and mouse-up. Pen events are generally passed through the same event mechanism as mouse events. As the pen moves, the stroke is echoed on the screen either as dots at each event point or as lines connecting the event points for a cleaner looking ink stroke. In modern user interfaces, there is generally a lot of processing associated with input events. In particular the event/damage/redraw cycle must occur on each input. For digital ink we want the stroke to be as clean and fluid as possible. For this reason, many tablet software systems will handle inking at a very low level and then generate a single stroke event upon pen-up. This allows the ink stroke to be drawn without incurring the whole event/damage/redraw loop. As processors get faster, this will be less of an issue. Regardless of the event handling technique, the result is a single stroke (array of points) that is passed on to the remainder of the digital ink system.

There are several standard steps that must be performed in almost every digital ink application. They are:

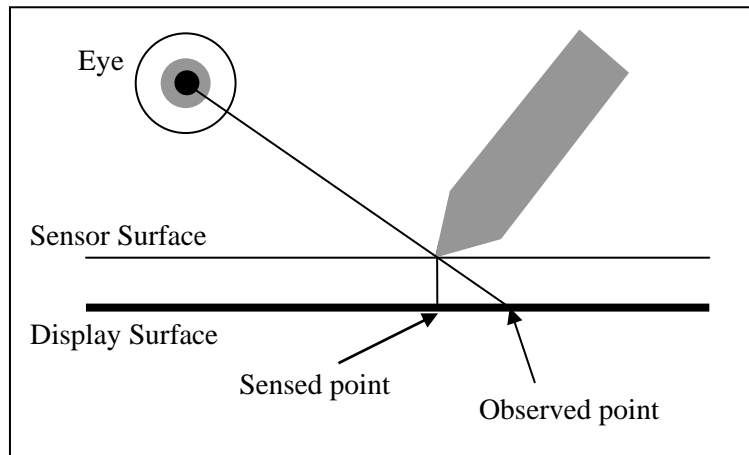
- Registration and parallax to deal with the relationship between the input device and a display.

- Cleaning to remove noise from the stroke that was produced by the input device.
- Normalization to make stroke recognition independent of size and location.
- Comparing strokes is frequently used in various recognition tasks.

**Registration and parallax**

Only an input device is required to acquire digital ink. However, it is common to integrate the digitizer with a display device. The display is either front projected or rear projected. In either case there is a difference between the coordinate system of the display and the coordinate system of the pen sensor. Frequently the resolutions of the digitizer and the display are very different. Depending on the sensing technique, the input resolution may be much higher or much lower than the display resolution.

These coordinates must be aligned so that the ink appears displayed exactly under the pen. Rear projected displays, where the image comes from under the sensor, can also suffer from parallax. This occurs when there is some visible thickness to the sense/display assembly. Because users rarely look straight down on a surface, this thickness can cause the pen to appear at a different position on the display than where it is actually located, as shown in figure 5. The parallax in figure 5 occurs with right-handed users. The parallax is in the opposite direction for left-handed users and may differ vertically based on whether the device is hand-held, resting on a table or displayed on a wall.



**Figure 5 – Pen parallax**

Both the coordinate alignment problem and the pen parallax problem can be handled with a registration process. In this process the display will show the users 5-8 points in sequence. For each point the user is asked to click on a spot that appears to the user to be over the displayed point. By having a particular user do this process while in a normal usage configuration, we get information that resolves both problems.

At the end of the user's registration process we have  $P_i$  points in pen coordinates and a corresponding set of  $D_i$  points in display coordinates. What we want is a function  $F$  such that  $F(P_i)=D_i$  for all samples  $i$ . The simplest form of  $F$  is a pair of linear functions of the form:

$$aP_x + bP_y + c = D_x$$

$$dP_x + eP_y + f = D_y$$

These equations will handle any combination of scaling, skew, rotation or position misalignment. What we need is a mechanism for reliably getting coefficients  $a$  through  $f$  from our set of sample points. In the linear system there are 3 unknowns for each equation. By collecting 5 points we have more than enough information to resolve the coefficients using linear least squares. The simple linear function, works great on digitizers that are placed directly over flat screens. It does not always account for situations where a projector is producing the displayed image. The projector optics can produce some non-linear distortions that must be accommodated. For this, functions of the following form handle most situations.

$$aP_x + bP_y + \frac{c}{P_x} + \frac{d}{P_y} + eP_x^2 + fP_y^2 + gP_xP_y + h = D_x$$

$$jP_x + kP_y + \frac{l}{P_x} + \frac{m}{P_y} + nP_x^2 + qP_y^2 + rP_xP_y + s = D_y$$

Because each equation has 8 unknowns we will need 10 or more sample points to resolve the coefficients. Both sets of equations are linear in their coefficients even though the second set of equations has non-linear terms. This allows us to use a linear least squares approximation to get coefficients that yield a function with the least error. The algorithm for this is found in Appendix A1.2f.

### Cleaning

At times a digitizer will produce very erratic points. These are due to some momentary noise in the sensing circuit. Where this occurs we need to remove those extraneous points because they make very ugly ink strokes and can disturb recognition algorithms. In many modern tablet systems the cleaning step is done before any strokes or input events are sent to the application. However, cleaning is a simple process. What we are looking for are points that are unreasonably far from their neighboring points. For any point  $P_i$  we use the distance between  $P_i$  and  $P_{i-1}$ . A simple technique is to eliminate any point where this distance is above some threshold. The threshold can be determined from the normal user hand speed and the sampling rate. The threshold can also be determined by sampling a number of strokes and computing the mean and standard deviation of the point distance. The threshold can be set at the mean plus two times the standard deviation.

### Normalize

We frequently want to recognize what an ink stroke is without considering the stroke's size or position. For normalization (after cleaning) we compute the maximum and minimum in X and Y of all points in the stroke. We then compute a transform

$$S = \frac{1}{\max(X_{\max} - X_{\min}, Y_{\max} - Y_{\min})}$$

$$T = \text{Scale}(S, S) \bullet \text{Translate}(-X_{\min}, -Y_{\min})$$

This will translate the upper left corner of the stroke to the origin, making it position independent and will scale the longest axis of the stroke to 1.0, making it scale independent. Using the same scale factor in X and Y retains the shape of



the stroke, which is usually very important. This transformation  $T$  is applied to all points in the stroke to compute a new normalized stroke.

### Thinning

Many digitizers will produce far more points than are useful for recognition. These numerous points also produce low level noise that can interfere with some recognition algorithms. Low level noise can also come from user hand tremors. A thinning process can reduce the number of points and remove much of the low level noise. We always keep the first and last points of the stroke because their locations are frequently important. Starting with point  $P_i$  we collect all succeeding points until we find one that is more than some threshold distance  $d$  from  $P_i$ . We collect these points and replace them with a single point that is their average. We then move to our new point and continue the process. We end up with a stroke that has points spaced approximately  $d$  apart. In normalized strokes a threshold of 0.01 to 0.1 is appropriate. The larger the threshold, the less noise in the stroke but there are more corner details removed.

### Comparing strokes

It is frequently helpful to be able to measure the difference between two strokes  $A$  and  $B$ . This comparison usually involves a comparison of pairs of points between two strokes. A simple measure is the average distance between each point and the closest point to it on the other stroke. There is a simple algorithm (figure 6) for this distance that is linear in the number of points in both strokes. This algorithm assumes that each point's closest point on the other stroke is very near to the previous point's match. Given two points in the sequence that already match, the next matching pair will come from one of the current points and one of the next points in the sequence or from the pair of next points. This assumption allows the search for matching points to be very limited and thus the linear time algorithm.

```

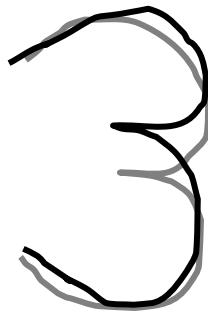
float strokeDistance(A, B)
{
    int i=0; int j=0;
    float dist = distance(A[i],B[j]);
    int count=1;
    while (i<A.size-1 && j<B.size-1)
    {
        float d1=distance(A[i+1],B[j]);
        float d2=distance(A[i],B[j+1]);
        float d3=distance(A[i+1],B[j+1]);
        if (d1<d2)
        {
            if (d1<d3)
            {
                dist+=d1;
                i++;
            }
            else
            {
                dist+=d3;
                i++;
                j++;
            }
        }
        else // d1>=d2
        {
            if (d2<d3)
            {
                dist+=d2;
                j++;
            }
            else
            {
                dist+=d3;
                i++;
                j++;
            }
        }
        count++;
    }
    while (i<A.size)
    {
        dist+=distance(A[i],B[B.size-1]);
        count++;
        i++;
    }
    while (j<B.size)
    {
        dist+=distance(A[A.size-1], B[j]);
        count++;
        j++;
    }
    return dist/count;
}

```

**Figure 6 – Stroke distance algorithm**

The distance used to compare a pair of points is generally the Euclidean distance. However, some researchers have used different distances based on

additional features of the points. Consider the two strokes in figure 7. It may be desirable for the inflection points in the middle of the strokes to be matched with each other to more accurately characterize the strokes. Because of the way in which these two strokes were drawn other points may actually be closer. If we modify our *distance()* function to include other features about the points besides their Euclidean distance those matches are more likely to occur. Possible features would be the angle formed between  $A[i-1]$ ,  $A[i]$  and  $A[i+1]$ , or  $A[i-1]-A[i+1]$ . Comparing features such as these as well as geometric distance produces a match of similar points to similar points.



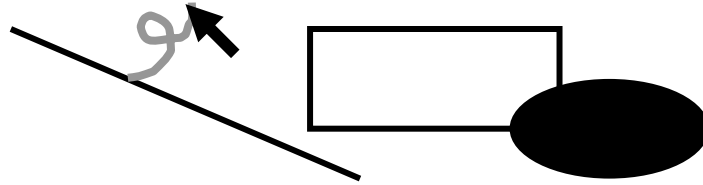
**Figure 7 – Two strokes to compare**

The algorithm in figure 6 is a greedy algorithm that makes a locally optimal decision on which points to pair with each other. If we start with two points ( $A[0]$ ,  $B[0]$ ) we can view the problem as a graph search where nodes of the graph are point pairs ( $A[i]$ ,  $B[j]$ ). From any such pair the same three choices of next possible match are possible (as in figure 6). However, using the least-cost path algorithm (Appendix A2.2) we can compute the globally minimal matching among points. This distance measure can be more accurate but is more expensive to compute because it is inherently non-deterministic. This *elastic matching* approach was used by Tappert<sup>5</sup> for handwriting recognition and in SHARK2<sup>6</sup> for comparison of sokgraphs.

### **Gesture/Character recognition**

One of the goals of pen-based interaction has been to drive user interfaces by user gestures rather than by menus or special keystrokes. Gestures have a number of advantages. The most important benefit is that the desired action and the target object for that action are expressed all in one stroke. For example the gray ink

gesture in figure 8 indicates a deletion, and its starting location also indicates the object to be deleted. This combination of action and object, can be very efficient. Forsberg et al.<sup>7</sup> used this ability to create a music composition tool. Each type of note is recognized from a unique gesture, while the position of the note on the music staff indicated the desired pitch and rhythmic position. This is much more efficient than menu-oriented systems. A very common gesture system is character recognition where each gesture corresponds to a character to be entered.



**Figure 8 – Delete gesture**

Use of gestures to interact involves three steps: 1) assembly of strokes into gestures, 2) classification of the strokes into a specific gesture class, and 3) extraction of geometric features from the stroke to use for the rest of the interaction.

#### **Assembly of strokes**

Many gestures are combinations of one or more strokes. Figure 9 shows several such gestures. Figure 8 also shows a common stroke notation where the beginning of the stroke is indicated by a dot. This is the notation that would be used throughout this chapter.



**Figure 9 – Multi-stroke gestures**

Deciding which strokes should be combined to form a single gesture can be a messy problem. Many gesture systems get around this problem by defining all gestures to consist of single strokes. This is the technique used by Unistrokes<sup>8</sup> and the original Graffiti<sup>9</sup> character recognizers. Where multi-stroke gestures are

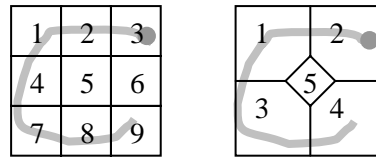
required, strokes are frequently combined when they occur within a very short time. The setting of that time threshold is problematic and the requirement of a time gap between gestures can slow down user input. Another technique is to combine gestures that overlap in the X coordinate. This would work for the X and the T in figure 9, but not for the K. Another technique is to recognize the strokes as separate gestures and also as a single gesture. This creates multiple hypotheses. A language model or the types of objects where the gesture is used can then be used to resolve the ambiguity.

### **Classification of gestures**

The gesture classification problem requires a function  $C(stroke)$  that will take a cleaned, thinned and normalized stroke and produce a class from some finite alphabet of possible classes. The classes might be characters for text input, actions such as “bold”, “delete”, “make thicker” or may be objects such as a line, circle, rectangle or musical note. The classification problem is pretty much the same for all of these cases.

Gesture classification is almost always trained rather than hard coded. One or more strokes are collected from each of the classes of gestures and then one of several classification algorithms is used for training (Appendix A3). Early approaches used sequence classifiers to compare stroke sequences. The elastic matching algorithm or its faster linear variant can be used as a distance metric to drive a nearest neighbor classifier (Appendix A3.1c). However, on many PDAs and on early computers of any kind, the computational demands of point-by-point matching of a stroke against hundreds of example strokes were beyond interactive speeds.

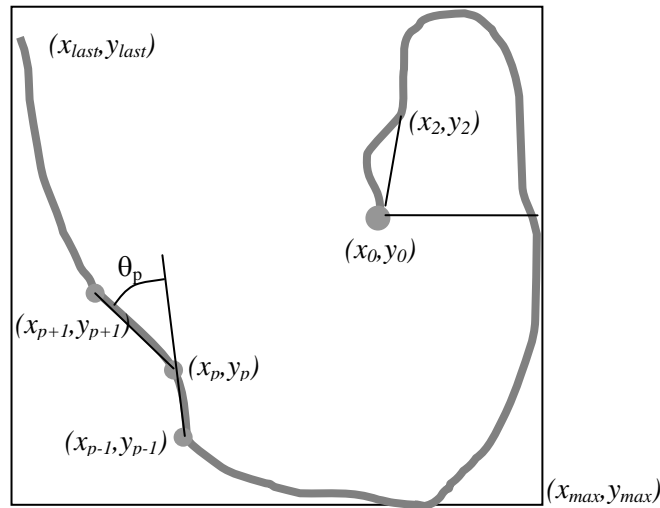
The first practical gesture recognizers used patterns like those shown in figure 10 to reduce the point data of the stroke down to a sequence of cell identifiers. Using the grid on the left, the stroke for “C” is reduced to the sequence 3,2,1,4,7,8,9. Using the pattern on the right the sequence is 2,1,3,4. Recognition takes place using a minimal edit distance algorithm (Appendix A3.2d) to compare these cell sequences with those stored for the example strokes. These recognizers were very fast and imposed little burden on the computers of the time. Training was also very fast and many applications provided the ability for users to train gestures for any menu command. However, these recognizers were not very accurate. Frequent misrecognitions occurred.



**Figure 10 – Character recognition regions**

Most modern gesture recognizers use vector classifiers (Appendix A3.1) rather than sequence classifiers for recognizing gestures. In this approach a function is programmed that will convert a stroke or set of strokes into a fixed vector of features. These features, rather than the original stroke, are passed to the classifier both for training and for recognition. Any of the vector classifier algorithms can be used, but nearest neighbor, linear perceptron or Bayesian classifiers are the most common.

The key to vector classification is the design of the feature function. The set of features very much determines the ability to discriminate among various of stroke shapes. The most common feature set is one developed by Dean Rubine<sup>11</sup>. The following feature set is based on his work with some simplifications. Starting from this feature set it is straightforward to invent new features that discriminate in various ways.



**Figure 11 – Stroke features**

Figure 11 shows a cleaned, thinned and normalized stroke from which we wish to compute a set of features. There are a variety of angles that can be computed as part of these features. In most cases the angles can be replaced by their sines and cosines to provide the same discrimination at lower computational cost. Sample features are:

1.  $x_0$  – the normalized x-coordinate of the start point
2.  $y_0$  – the normalized y-coordinate of the start point
3.  $\frac{x_2 - x_0}{\sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}}$  - cosine of the starting angle. Point 2 is used to get a good sense of the user's direction and to ignore any starting "wiggles" in the stroke.
4.  $\frac{y_2 - y_0}{\sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}}$  - sine of the starting angle.
5.  $x_{max}$  – This along with  $y_{max}$  gives us a measure of the eccentricity of the normalized stroke. For example the normalized stroke for

“I” would have a very small  $x_{max}$  and a  $y_{max}$  of 1.0. The normalized stroke for “-” would be the reverse.

6.  $y_{max}$  – also a measure of stroke eccentricity
7.  $\sqrt{(x_{last} - x_0)^2 + (y_{last} - y_0)^2}$  - distance between the first and last point. This would distinguish between a “C” (large distance) and an “O” (small distance).
8.  $\frac{x_{last} - x_0}{\sqrt{(x_{last} - x_0)^2 + (y_{last} - y_0)^2}}$  - cosine of angle from first point to last point.
9.  $\frac{y_{last} - y_0}{\sqrt{(x_{last} - x_0)^2 + (y_{last} - y_0)^2}}$  - sine of angle from first point to last point.
10.  $\sum_{p=0}^{last-1} \sqrt{(x_p - x_{p+1})^2 + (y_p - y_{p+1})^2}$  - the total summed length of the normalized stroke. This would distinguish between an “O” and a spiral stroke repeated several times. The end points might be the same but the stroke length would be very different.
11.  $\sum_{p=1}^{last-1} \theta_p$  - the sum of the all of the point angles in the stroke.  
Positive and negative angles cancel each other so that for an “I” stroke that is a little wavy this feature would still come out to be 0.0. For an “O” stroke this would come out to  $2\pi$  radians because one whole turn has been made regardless of any waviness in the stroke. Thinning of a stroke is very important to getting good point angles for these features.

$$\theta_p = \tan^{-1} \left( \frac{(x_{p+1} - x_p)(y_p - y_{p-1}) - (x_p - x_{p-1})(y_{p+1} - y_p)}{(x_{p+1} - x_p)(x_p - x_{p-1}) + (y_{p+1} - y_p)(y_p - y_{p-1})} \right) ..$$

12.  $\sum_{p=1}^{last-1} |\theta_p|$  - sum of the absolute values of the point angles. In this case positive and negative angles do not cancel each other. Wavy strokes have higher values for this feature than straight strokes.



13.  $\sum_{p=1}^{last-1} \theta_p^2$  - sum of the squares of the point angles. This feature scores higher for very sharp angles. The square of 180 degrees is very much larger than any sum of 5-10 degree angles.

These features are not exhaustive but they are a good start and are relatively easy to calculate. For a given gesture recognition problem where gestures of different classes are not discriminated well, it is helpful to look at the cases and invent a feature that will discriminate. This new feature(s) can be added to the set and exploited by the training algorithm.

### **Extraction of geometry**

Because we frequently use gestures both for actions and object identification, we need to extract geometry information from the stroke to use as arguments to the action. Geometry extraction is usually performed on the cleaned, thinned but not normalized stroke. Normalization purposely discards much of the geometry for better recognition. The starting point  $(x_0, y_0)$  is normally used to identify the object of an action. When creating objects, such as lines or arcs, the starting and ending points are used to derive the geometry of the object being created. Frequently the bounding box can be used for rectangle or ellipse gestures. Sometimes sharp corners are used for geometry. These can be identified as point angles  $(\theta_p)$  whose absolute value is above some threshold.

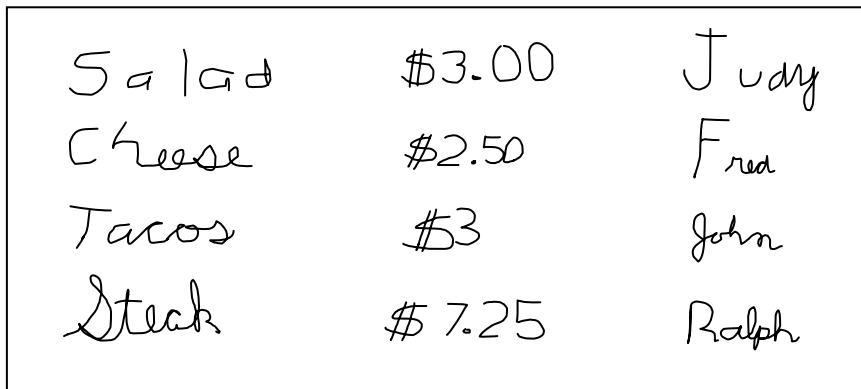
### **Digital ink as a data type**

One of the advantages of pen and paper is that the medium is completely unstructured. As thoughts and ideas come we can organize them on the paper any way that we want. Many systems have been developed to mimic this aspect of paper. The model for such an application is simply a list of stroke objects each containing a sequence of points. Such note-taking applications are trivial to implement. However, the reason for going digital is that the computer is not paper. The computer can erase, move, organize, search and share information in ways that are not possible with paper. What we want are applications that support the freedom of paper while providing the increased flexibility of digital representation.

#### **Tivoli**

One of the most powerful digital ink systems is Tivoli<sup>12</sup> from Xerox PARC. The key insight from Tivoli is the discovery of inherent structure rather than the explicit encoding of structure. Structure in information is critical for a computer to provide leverage for human activity. For example, when writing with a word processor, the user can place the cursor between any two letters and begin typing. The word processor automatically moves characters to make room for each new character, rewraps lines of text, and moves subsequent paragraphs out of the way. The word processor can do all of this because it has an encoding of the structure of the document. It has explicitly encoded the concepts of lines, paragraphs and pages. Using this structure the program knows how to do many mundane tasks for us. On a piece of paper we cannot conveniently insert a word into a sentence that has already been written. Part of the problem is the nature of paper and part of the problem is that there is no encoded structure. Imagine trying to build a word processor where the model was a list of characters each with its own X, Y position. A similar effect can be seen in a spreadsheet. We can select, move, highlight or delete entire rows and columns because their structure is encoded in the model.

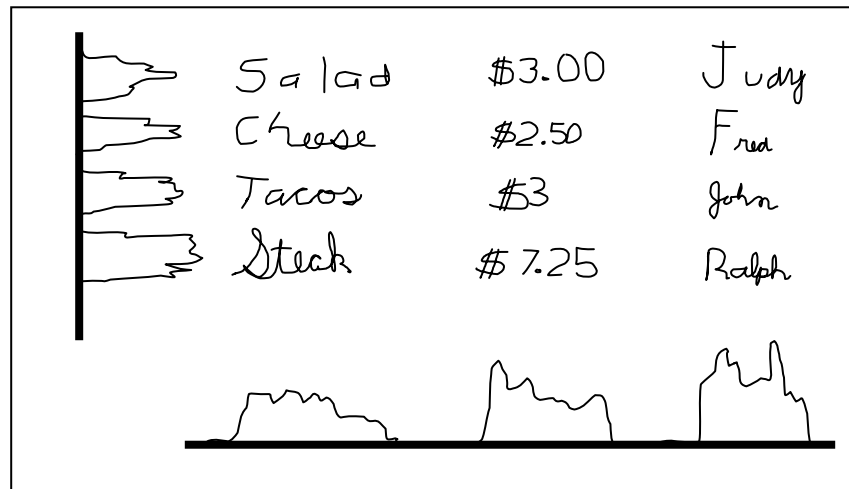
The designers of Tivoli recognized that humans can look at a page of text and know what should happen when a new character is typed or a column is moved. We know this simply from what we see, not because of any encoded data structure. As humans we see only the presented information, not the underlying data structure. If the necessary structure is visually present for humans to see, then it must also be available in the presentation for computers to exploit. In basic Tivoli the only data type is the digital ink stroke. From this information Tivoli can infer many structured behaviors.



Salad	\$3.00	Judy
Cheese	\$2.50	Fred
Tacos	\$3	John
Steak	\$7.25	Ralph

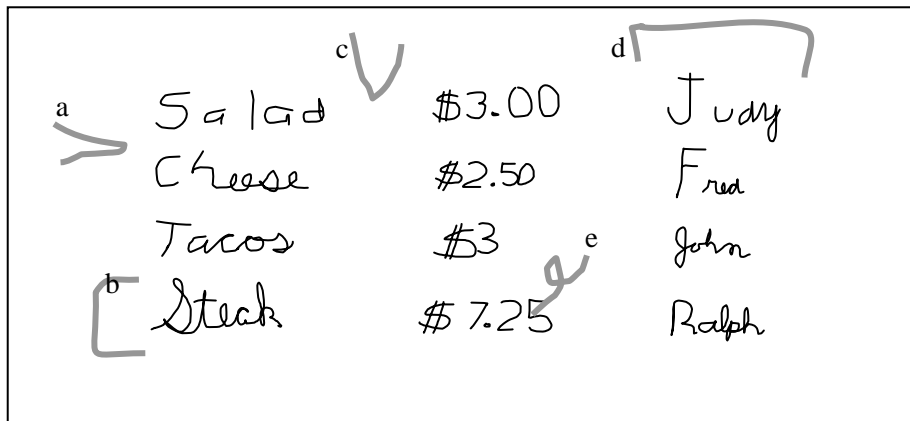
**Figure 12 – Table using digital ink**

The original Tivoli paper focused on three types of structure in digital ink: lists, text and tables. The key tasks that they implemented were selection, deletion and insertion. For a list one frequently wants to add new things into the list at any point, remove things from the list and reorder items in the list. The desired help from the computer is to adjust the remaining elements of the list to open or close up sufficient room for these operations. The ink shown in figure 12 is obviously a table in its structure. However, the only encoded information is the ink strokes themselves. The table items are each made up of several ink strokes. Tivoli's approach is to discover structure as it is needed. One approach for structure discovery uses ink histograms. For each row and column we count the number of ink points, as shown in figure 13. Note that the spaces between rows and columns are indicated by the zero regions in the histograms. This makes it relatively easy to discover these structures from the ink strokes.



**Figure 13 – Ink histograms**

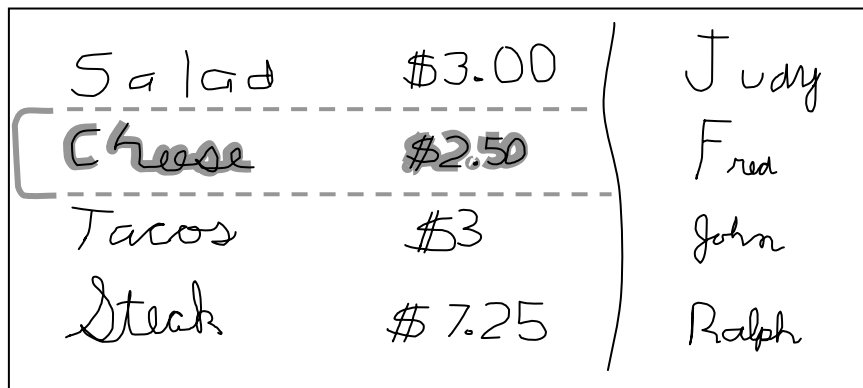
Tivoli defined several gestures that could be used to manipulate ink. These are shown in figure 14. Row and column insertion (a and c) depend on whether something is already selected. If there is no selected ink then these will move ink down or to the right to make room for a new row or column. If there is selected ink, then these will make room, move the selected ink into the open space and close up the space where the ink was moved from. The row and column selection gestures (b and d) project select all ink in the corresponding row or column (using the ink histograms from figure 13). These selections can then be used with the insertion gestures to move ink around. The deletion gesture (e) will either delete a single stroke or, if a row or column was selected, it will delete the entire row or column and close up the empty space. This opening and closing of space and moving ink around to make it happen is far more powerful than simple paper and does not require the user to “create a table object”. The necessary structure is discovered on the fly as the user works with the information. The gestures are easily trained for recognition using Rubine’s feature set.



**Figure 14 – Table manipulation gestures**

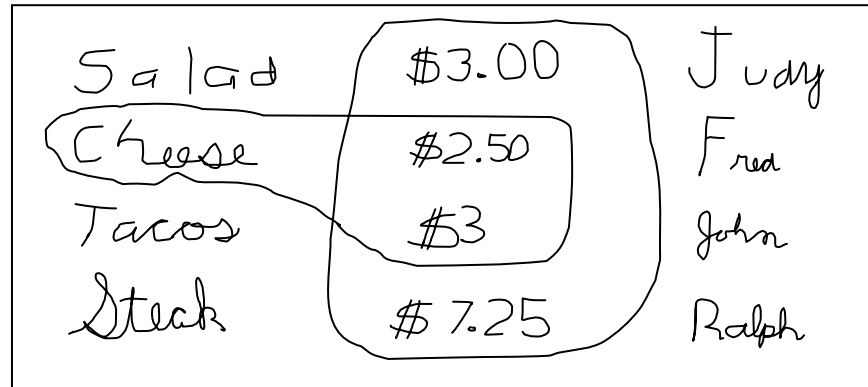
These gestures can also be interpreted by projecting the gesture coordinates down or across and selecting all strokes for which a majority of their points lie inside of the projected regions. This removes the need for the ink histograms.

Sometimes it is desirable to work with more than one independent list or table. These can be partitioned using *border strokes* as shown in figure 15. A border stroke is just another stroke in the ink model. Its presence is detected as a single stroke that is either vertical or horizontal and extends from boundary to boundary. It is treated as just another stroke so that it can be moved and deleted in the same fashion as all of the other strokes. When projecting selection or insertion, the projection stops at any border strokes as shown in figure 15.



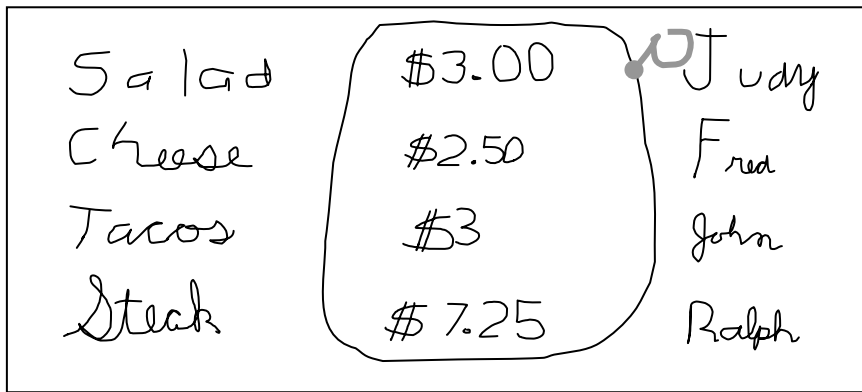
**Figure 15 – Border strokes**

The Tivoli researchers discovered that grouping of ink strokes is a fundamental concept<sup>13</sup>. In the previous discussion items were implicitly grouped by the selection gestures. Border strokes provided additional structure. In many drawing programs, grouping is defined by explicit group operators to bring items together. Tivoli chose two approaches for defining grouping. The first was a system of explicit border objects. Such border objects are created by gestures that look like border strokes and are then replaced by a movable line that separates groups. This is much like tiled windowing systems.



**Figure 16 – Enclosure strokes**

The second grouping technique was to use enclosing strokes to define groups. Drawing a stroke around a set of other strokes creates a group as shown in figure 16. This work developed several gestures for modifying enclosures. A wedge gesture cutting across an enclosure would divide it into two enclosures. A gesture that starts and stops on the enclosure stroke would modify its boundary. Any gesture applied to an enclosure could be applied to all strokes in the enclosure. A stroke that started on one enclosure and ended on another created a link stroke. Moving either of the linked enclosures produced a scaling on the link stroke that kept it connected to the original enclosures. This created various dynamic diagrams. A *balloon* gesture when applied to an enclosure as shown in figure 17 would reduce the selected strokes to a single balloon stroke. Clicking on the balloon stroke would pop up the original enclosed strokes for viewing or pasting. This mechanism creates summary diagrams of varying detail.



**Figure 17 – Balloon gesture**

#### **Gesture/Ink/Action segmentation**

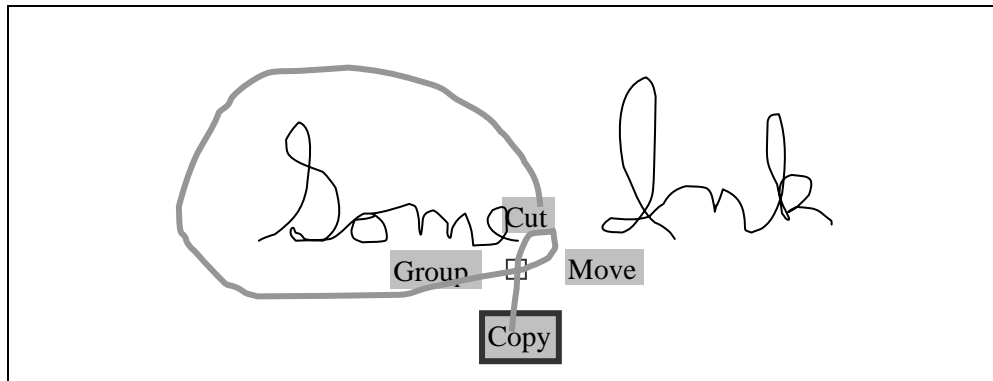
When working with ink as a data type there is an ambiguity between what is a gesture and what is a data object. The Tivoli project, as with many others, used a button on the pen to indicate a gesture rather than an ink stroke. The following is a list of techniques taken from Li et al<sup>14</sup>.

- Button on pen barrel
- Press and hold still until gesture feedback appears. This is used in the Microsoft Table PC.
- A separate button on the tablet that is activated by the non-dominant hand.
- Using a pressure sensing pen, the user presses harder for gestures than for ink strokes.
- Some pens have an *eraser* end that can be separately sensed. The eraser end is used for gestures.

The experiments performed by Li et. al. noted several tradeoffs among these techniques. However, the use of the non-dominant hand was the fastest of the techniques.

The Scriboli<sup>15</sup> system looked at the problem of segmenting a gesture into selection of desired strokes and specification of what action should be taken on the selected strokes. In their approach an enclosing gesture surrounds all of the

strokes to be selected. This gesture then ends in a pigtail by crossing over the stroke. When this pigtail is detected a menu is displayed around the crossover point and the remainder of the gesture selects the correct action from the menu of possibilities. This is shown in figure 18.



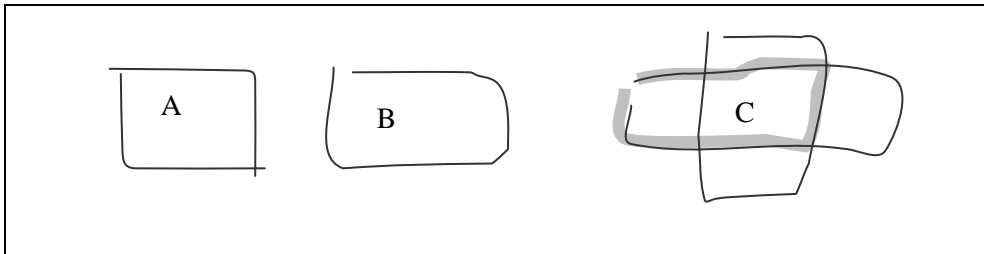
**Figure 18 – Scriboli gesture menus**

### **PerSketch**

One of the advantages of digital ink over scanned images is that the ink stroke itself has more information than a simple map of pixels. As Tivoli demonstrated, dependence upon this information can be a problem. People interact with what they see, not with the order or form in which it was created. In figure 19, object A is perceived as a rectangle even though it is composed of 2 strokes. Object B is also perceived as a rectangle though it consists of only one stroke. For most users these are the same kind of shape regardless of how they were drawn.

Object C is composed of just two strokes, each of which is perceived as a rectangle. However, there are 9 possible other rectangles such as the one highlighted in gray. None of these 9 were explicitly created by the user, but they are perceived by the user. Recognizing the shapes that the user sees rather than the shapes as they were created is important to interacting with them naturally. When only looking at shapes, the user can perceive whatever they wish and the software need not be involved. When trying to select objects on the screen an understanding and support for what users perceive can provide a more intelligent and effective mechanism for selecting the intended shape with minimal effort.

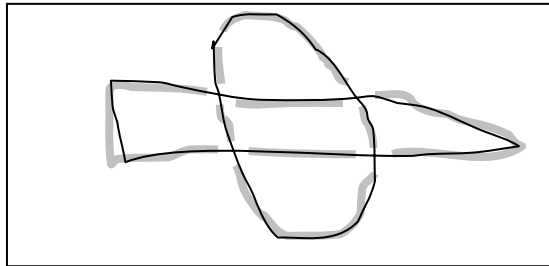




**Figure 19 – Perceived shapes from strokes**

Eric Saund's PerSketch<sup>16</sup> system addresses the perceived shapes that should be discovered from those explicitly created. Rather than trying to recognize shapes from all of their possible meanings, PerSketch uses selection input from the user to identify the shape that the user intends. In figure 19 the fat gray line represents a selection stroke by the user. This stroke disambiguates the desired shape from all other possible shape interpretations in the figure. The goal of PerSketch is to translate a user's selection gesture into a selection of the most likely shape to be selected regardless of the form of the original strokes.

The PerSketch approach is to divide the ink strokes into a series of *prime objects*. A prime object is a stroke with no intersections. Figure 20 shows the 9 individual prime objects derived from two strokes. These prime objects can be composed to form composite objects, or individual shapes.

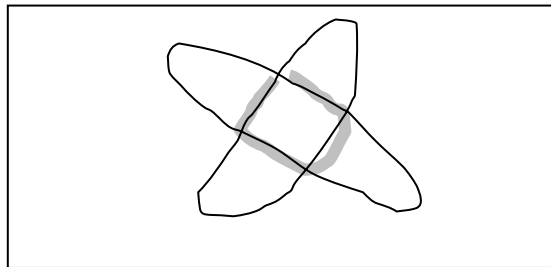


**Figure 20 – Prime objects**

As the number of strokes increases, so does the number of prime objects and the exponential number of ways in which prime objects can be composed into shapes. For use in a sketching tool the algorithms for composing shapes must be very fast. PerSketch uses a *shadow bitmap* as its data structure for rapid discovery of prime objects from ink strokes. A shadow bitmap is an array of pixels that contain pointers to data structures rather than color values.

Background pixels have null pointers. Pixels that correspond to a single prime object contain a pointer to that object. Intersection and touching points contain a pointer to a list of prime objects that touch that point. When a new stroke is entered by the user its pointer is “drawn” into the shadow bitmap. Null pixels are replaced with the stroke’s pointer. Pixels that already contain stroke information cause the stroke being entered to be broken apart at that point. The two parts are added to the list of prime objects at the intersection point and the stroke processing proceeds.

Because there are an exponential number of composite shapes in a drawing we only want to identify a shape when the user actually wants to work with it. Suppose that the user wants to create a propeller shape from the two strokes in figure 21. What is needed is to remove the interior prime objects. These prime objects can be selected by a single gesture as shown in figure 21. The selected objects can now be easily deleted.



**Figure 20 – Selecting a composite shape**

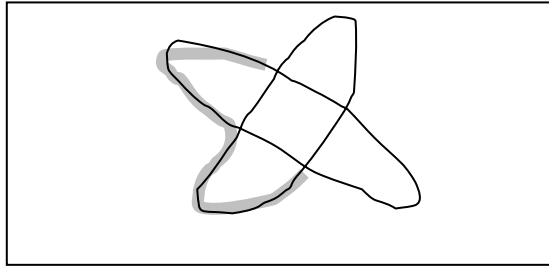
Given the shadow bitmap and its corresponding connected graph of prime objects we need an algorithm for selecting the set of prime objects that most closely corresponds to the selection gesture. PerSketch provides two such algorithms: pose matching and path tracing.

#### pose matching

The pose matching algorithm enumerates all of the composite objects that might correspond to the selection gesture. This incurs the exponential cost of enumerating all possibilities. Before enumerating composites all prime objects, that are more than some threshold distant from the selection gesture, are discarded as irrelevant. Composite objects are computed from the remaining prime objects by traversing the graph of primes from the shadow bitmap. Each unique cycle in the graph that traverses any prime object at most once is a candidate shape.

From each recognized composite shape PerSketch computes a *pose model*. The pose model consists of the *x-center*, *y-center*, *orientation-angle*, *length* and *width*. The center point is computed by averaging all of the points in the strokes. The orientation angle is computed using the algorithm in Appendix A1.3c. This angle is based on the difference in width and height. For square or circular shapes there is no dominant orientation angle. If the aspect ratio of the object is nearly equal, then an orientation angle of zero is used. Once the center and orientation are known we can translate the shape to the origin and rotate the orientation angle to zero. With the stroke points in this normalized coordinate system, the length can be either the average or maximum of the absolute values of *X*. The width is similarly computed using *Y*. Note that we are not looking for the actual length and width as much as a feature that represents them.

Using the pose model we can compare each composite shape to the pose model of the selection gesture. A simple distance metric is not acceptable however because of “unoriented” shapes with no distinct direction. PerSketch uses a different distance metric that deemphasizes orientation when the aspect ratio is close to 1.0. The selected prime objects are those who form a composite shape that has the pose model most similar to the selection gesture.



**Figure 22 – Path traced selection**

#### path tracing

The pose matching approach is somewhat limited in the kinds of selections that are possible. The selection of prime objects associated with the gesture in figure 22 would not perform well using pose matching. The path tracing algorithm is a variation on the stroke distance algorithm in figure 6. Each point in the selection gesture is compared against the nearest points in the shadow bitmap. We are looking for a sequence of prime objects that are nearest to the stroke. When an intersection point is reached, the algorithm must non-deterministically follow all strokes looking for the least match. This is similar to the elastic

matching algorithm described earlier. Path tracing is more expensive than pose matching, but it produces more flexible match possibilities.

### **Flatland**

Flatland<sup>17</sup> takes a somewhat different approach for interaction with digital ink. Unlike Tivoli and PerSketch, which focus on selection and intelligent manipulation groups of strokes, Flatland focuses on the interpretation of what strokes mean. In Flatland, structure is specified by the user in the form of *segments*. A segment behaves somewhat like a window in traditional user interfaces except that the associated interactive techniques are different. A segment has boundaries, like a window, except that they are drawn in a “sketchy” style consistent with the digital ink metaphor. The boundaries of a segment are primarily determined by the size of its content. When a segment is selected the boundary expands to the right and down to make room for the user to draw new content. This is consistent with the way people create lists, notes, drawings and other items using languages that read left-to-right and top-to-bottom. For other languages a different convention would be required. When new strokes are entered, the segment automatically resizes to make more room.

The unique contribution of Flatland is in the way that it handles strokes of digital ink. Each segment consists of a boundary, a behavior and a list of strokes. The only input events are “new stroke” and “move segment”. When a new stroke is received, it is passed to the behavior object associated with the segment. The behavior object consults the strokes and other information already associated with the segment and modifies the strokes in the segment’s list. This very simple input model provides very powerful features as shown in figure 23.

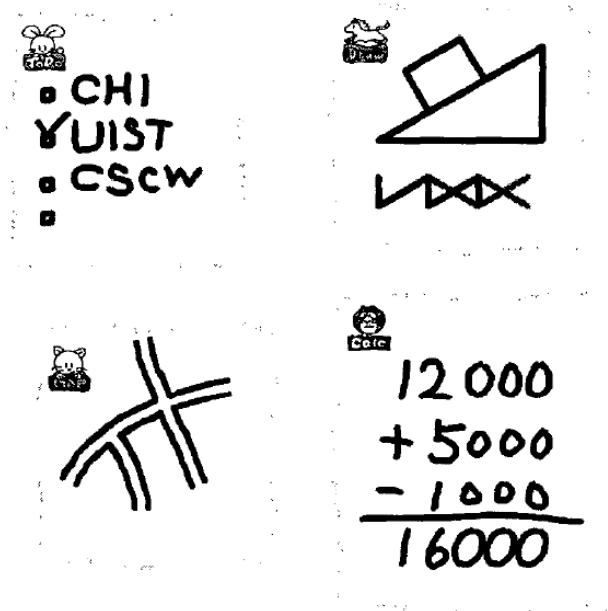


Figure 23 – Flatland<sup>17</sup>

The List behavior, shown in the upper left, will translate checkmark strokes as gestures and mark a list item as complete. Writing strokes next to the bullet mark at the bottom will add those strokes to the list and make a new bullet mark. The Pegasus behavior at the upper right uses the Pegasus stroke beautification system to translate strokes into more regular pictures. The lower left behavior translates single strokes into stroke pairs that resemble streets for easy sketching of maps. The lower right behavior translates strokes as digits, arithmetic operators and the answer line. The results of the calculation are shown by the behavior object adding additional strokes to the segment automatically.

Each of these behaviors is independent of the others. However, the user can change a segment's behavior at any time. When behaviors are changed, the strokes remain part of the segment to be interpreted by the new behavior. Thus a user can slip into Pegasus to draw a tidy drawing, slip into the map behavior to add streets and then change to the simple ink behavior to write notes on the map. The uniform use of ink strokes as data objects and as input events makes this flexibility possible. The Tivoli system could easily be implemented as a behavior.

A user moves a segment by dragging one of its edges. When a segment runs into another segment, that encountered segment is moved out of the way. This may cause a cascade of movements of segments bumping into each other to make room. Overlapping segments are not allowed. If there is not enough room to move a segment where it is wanted, other segments are resized. This moving/resizing is performed recursively to automatically make room. If a shrunken segment is selected, it is restored to normal size and other segments are moved/shrunken to make room. This “window management” scheme requires very little effort on the part of the user and makes all information visible simultaneously.

Unlike Tivoli and PerSketch, a Flatland user must pay more attention to intended structure of what is being drawn. Segments are explicitly created to provide grouping of strokes rather than inferring their structure. Selection of a segment’s behavior is required to control how strokes are interpreted. On the other hand, the additional structure simultaneously provides more user control and more sophisticated behaviors.

### **Sketching Tools**

A second major use of digital ink is in creating drawings or sketches. One of the great advantages of digital ink over pencil and paper is the ability to improve a rough sketch dynamically. An eraser is the only means for correcting pencil drawings and it is not a very satisfying tool. In this section we will look at two kinds of algorithms for improving sketching techniques. The first is the use of cubic splines to improve strokes and to edit strokes once they are created. Secondly is a technique for inferring lines and patching together rough strokes to create a cleaner drawing.

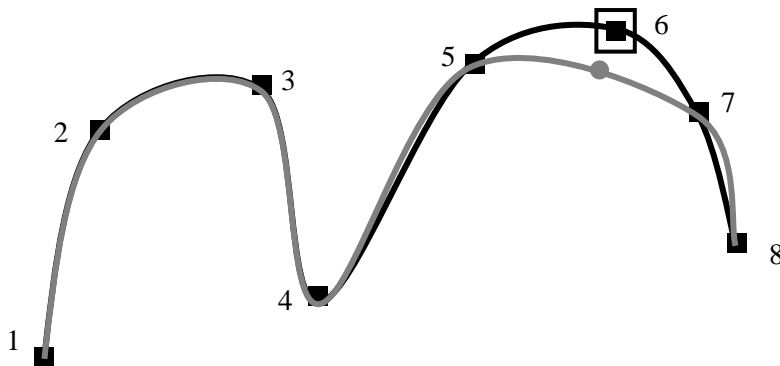
#### **Ink to cubics.**

An early sketch improvement technique is to smooth a stroke by approximating it with a series of cubic splines. Converting an ink stroke to a cubic has three advantages. The first is that the stroke can be smoothed to remove hand jitter from the line. Secondly the cubic curve representation is typically more compact than the ink stroke in that the number of control points is typically much smaller than the number of ink samples. Thirdly the curve’s control points provide handles for editing the curve.

For a given ink stroke represented by a series of points we want to compute a set of control points for a cubic curve. The easiest curve representation is the Catmull-Rom (chapter 12). We can obviously create a curve by using the ink

points as Catmull-Rom control points. This, however, would not provide any advantages because the ink points are too dense. What we want is to remove as many of the points as possible while still retaining the shape of the curve. In most situations the user also provides a *smoothing factor*. The larger the smoothing factor the less small detail is retained from ink stroke. A smoothing factor of zero will exactly match the ink stroke. The smoothing factor defines the maximum distance between an ink point and the nearest point on the smoothed curve. This is very similar to the thinning threshold for cleaning up strokes described earlier. Before converting to a curve the ink should be cleaned and then thinned using a threshold of one half of the smoothing factor.

One approach to smoothing an ink stroke is to successively remove points that are less than the smoothing factor from the resulting curve until no possible deletions are possible. Figure 24 shows an ink stroke consisting of 8 points that has been converted to a Catmull-Rom curve. We want to decide if point 6 can be removed while preserving appropriate smoothness. Because of the nature of piecewise cubic curves, we only need to consider the segment defined by points 4,5,7 and 8. Using this curve we find the closest point using the algorithm from chapter 12. If the distance from point 6 to the closest point is less than the smoothing factor, then point 6 can be discarded.



**Figure 24 – Removing control points**

Note that although the curves beyond points 4 and 8 are not affected, the curves between 4 and 5 and between 7 and 8 are changed. This is because the curve between 4 and 5 had its control point changed from 6 to 7. Given the fact that the smoothing factor is typically small, any point that is a candidate for

deletion will typically cause very little distortion in neighboring curves. We will thus ignore the effect.

With a criterion for deleting points, we have several strategies for considering which points to remove. We always retain the starting and ending point. We could consider points 2 through 7 in turn. This has problems. Each point  $N$  being considered is only half the smoothing distance from point  $N+1$ . As such, it is very unlikely that point  $N$  will be more than the smoothing distance from the curve. This means that point  $N$  will almost always be removed. Proceeding in this way, all points but the end points will be removed, which is not a good thing. This is because the removal of prior points discards information about the curve.

A better algorithm is to consider every other point for deletion. This means that points adjacent to the candidate point are original and are retaining information about the curve in that region. After considering every other point we can repeat the algorithm on the remaining points. The algorithm terminates when there are no points deleted in a given pass. This algorithm is given in figure 25.



```

Point [] smoothCurve(Point [] inputPoints, float smoothing)
{
    boolean continue=true;
    Point [] curve=inputPoints;
    curve.insertPoint(0,inferFirstPoint(curve)); // provide first control point
    curve.addPoint(inferLastPoint(curve)); // provide last control point
    while (continue)
    {
        int len=curve.getLength();
        Point [] result;
        boolean continue=false;
        result.addPoint(curve[0]); // keep the first point
        for (int i=2;i<len-2; i+=2) // never consider the first two or last two points
        {
            result.addPoint(curve[i-1]); // retain skipped point
            if (closeEnough(i, curve,smoothing))
                continue=true; // another pass will be required
            else
                result.addPoint(curve[i]); // retain point
        }
        result.addPoint(curve[len-2]); // keep second to last point
        result.addPoint(curve[len-1]); // keep last point
        curve=result;
        result=null;
    }
    return curve;
}

boolean closeEnough(int i, Pont [] curve, float smoothing)
{
    Use nearest point on a curve (chapter 12) to find the distance
    between curve[i] and the Catmull-Rom curve defined by
    curve[i-2], curve[i-1], curve[i+1] and curve[i+2]. The iteration
    can terminate whenever any point is found that is closer than
    the smoothing value or the distance range being considered is
    closer together than smoothing/4.
}

```

**Figure 25 – Smoothing a curve by point deletion**

Note also that the curve between points 1 and 2 in figure 24 needs a control point 0 and the curve between 7 and 8 needs a control point 9. If the curve were closed, this would not be a problem because we could use the other control points. However, we need point 0 and 9 to assist with the tangent of the curve at points 1 and 8. One simple technique is to use the line between the start point and the second point as an approximation of the appropriate tangent. We can add these points using the functions shown in figure 26. We compute a tangent vector, normalize it to length 1 and then add it to the start or end point to infer the extra control point.

```

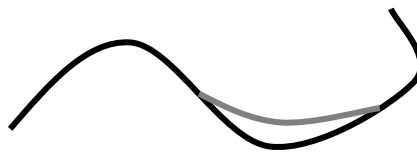
Point inferFirstPoint( Point [] curve )
{
    float dx=curve[0].x-curve[1].x;
    float dy=curve[0].y-curve[1].y;
    float len = sqrt(dx*dx+dy*dy);
    dx=dx/len;
    dy=dy/len;
    Point result;
    result.x=curve[0].x+dx;
    result.y=curve[0].y+dy;
    return result;
}
Point inferLastPoint( Point [] curve )
{
    int len=curve.getLength();
    float dx=curve[len-1].x-curve[len-2].x;
    float dy=curve[len-1].y-curve[len-2].y;
    float len = sqrt(dx*dx+dy*dy);
    dx=dx/len;
    dy=dy/len;
    Point result;
    result.x=curve[len-1].x+dx;
    result.y=curve[len-1].y+dy;
    return result;
}

```

**Figure 26 – Inferring starting and ending control points**

### Sketching changes to curves

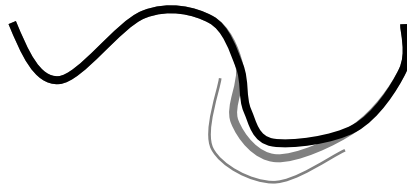
Even with pencil and paper, artists regularly draw lines that are not quite correct. The control points of cubic curves can be moved and new control points can be inserted by clicking on a point on the curve and adding a control point at that location. Control points can be deleted. These curve editing techniques are not as satisfying as simply resketching the area to be modified. Figure 27 shows a curve with an additional gray curve that has been created by the user.



**Figure 27 – resketching a curve**

A simple mechanism for editing the curve as shown is to insert new control points into the original curve at the start and end points of the *editing stroke* (gray). Any control points from the original curve (black) that lie between the newly created control points are replaced with the ink points from the editing stroke. The inserted region plus two points before and after is then smoothed using the algorithm described in figure 25.

When modifying a curve it is frequently desirable to “nudge” the curve rather than edit the curve. Figure 28 shows a curve (thick black) and an editing stroke (thin gray). The editing stroke is a hint as to how the curve should be modified rather than an explicit edit of the curve. The resulting curve (thick gray) is a blend of the original curve and the editing stroke. This technique was proposed by Baudel<sup>18</sup>. The basic approach is to resample the relevant section of the original curve into points at approximately the same resolution as the thinned edit stroke. The new curve points are computed as an interpolation between the original points and the nearest points on the edit stroke. Nearest points cannot be determined by simple Euclidean distance. Each point can be identified by their proportional distance along the curve. Points are matched by this curve position rather than geometry. Sequence position matching rather than geometry matching allows better ability to change the shape of the curve.



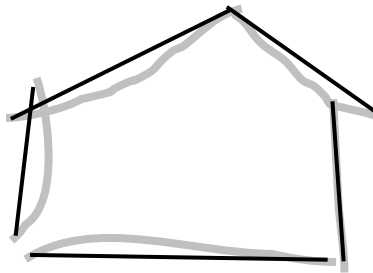
**Figure 28 – “Nudging” curves**

The nudging technique gives more control over curve changes by using several edit strokes to move the curve into the desired shape. The result is more an “averaging” of several strokes than an explicit use of any stroke in particular. This corresponds to common sketching behavior where lines are resketched multiple times to shape them.

### **line sketching**

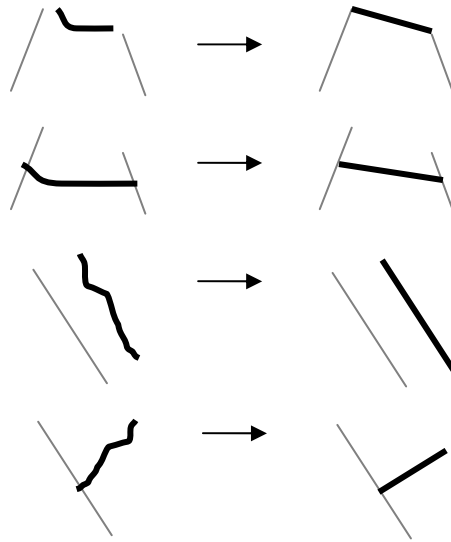
Frequently what is desired is not a sketch but a line drawing. Sketching is the input technique rather than the end result. Recognizing a straight line gesture is relatively easy using Rubine’s features. If the arc length (sum of the distance

between thinned ink points) is close to the distance between the start and end points, then a straight line is inferred between the two end points. Figure 29 shows the input strokes (gray) and the resulting lines (thin black). The result is not very satisfying because the diagram is rather messy.



**Figure 29 – Strokes to lines**

The problem is that we expect lines to be vertical, centered, parallel, connecting at end points are correctly touching other lines. The Pegasus<sup>19</sup> system takes the lines that were entered and generates multiple constraints that such lines might feasibly satisfy. Figure 30 shows several such constraints, including alignment of vertices, snapping to other lines, parallelism and perpendicular lines. Sometimes there are several conflicting constraints that could be applied to a new stroke. In such cases they are all presented to the user, who can then select one.



**Figure 30 – Possible constraints imposed upon lines**

### Annotation

Previously we have discussed techniques for intelligently manipulating ink as well as tools for sketching with digital ink. We will now look at digital ink as an annotation tool. A common use of a pencil is to mark up paper documents. We underline quotes in books, write in the margins, cross out unwanted material or mark where text should be inserted. Digital pen-based annotation comes in two forms: annotating fully digital documents and annotating paper.

Annotating a fixed document with digital ink is relatively straightforward. A document representation, such as PDF, can be treated as a static image and the ink strokes can be displayed over the top. Ink strokes are stored in the coordinates of the page on which they are to be displayed. This duplicates the behavior of pen and ink, but adds little digital advantage.

### XLibris

The XLibris<sup>20</sup> project explored the role of digitally annotated documents by means of a special tablet computing device roughly the shape of an 8 ½ by 11 piece of paper. They were interested in the concept of *active reading* where

people annotate, clip, link and search documents rather than just read them. One of their first contributions was the Reader's Notebook. As the user reads a document, they make annotations on it with a digital pen. The annotations and the underlying document materials are excerpted into the notebook to provide a "highlighted" version of the document, as shown in figure 31. This automatically extracts the text that the reader found interesting. In the physical world this would require user copying the text in some cumbersome fashion. Looking through the notebook the user can see the condensed annotated summary and at any time link back to the original context (which is impossible with physical paper).

There is a great attraction to carrying a personal computer in your pocket. It enables a nomadic style of computing that is not feasible with desktop machines or even with laptops. There are now many hand held devices that are less than one cubic inch in volume and yet have much more storage and processor power than the original Macintosh. However, the user interface to such devices is unacceptable for many applications because of display size.

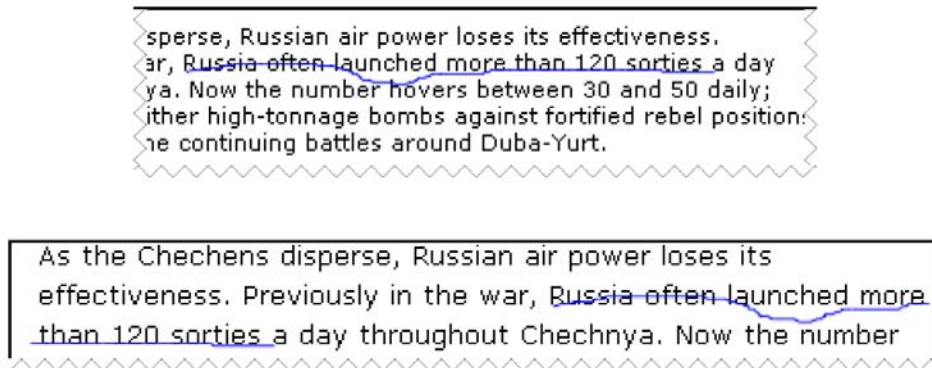
- nomadic style of computing
- more storage and processor power than the original Macintosh.

**Figure 31 – Annotating text**

The fact that digital ink is tied to digital text when annotating a document opens other opportunities. The XLibris team extracted the text underlying a user's annotations and used that text to provide focused document searches. In information retrieval a document is generally represented by a vector of the frequencies of each word in the document. The vector has the length of all words known to the system. These vectors are very sparse and rarely represented as arrays. The similarity of two documents can be measured by the cosine distance (appendix A1.1e) between two such vectors. This measure can be improved by weighting the importance of various words. A common weighting is dividing the frequency of a word in a document by the frequency in all documents. Very frequent word such as "like" thus get very low scores and infrequent words such as "gastrointestinal" get much higher scores. XLibris augmented these weights by increasing the importance of text annotated by the user. This specifically focuses the comparison on those concepts that the reader felt were important. The retrieval results compared favorably with some of the best relevance-feedback techniques<sup>21</sup>.

### **Annotating HTML documents**

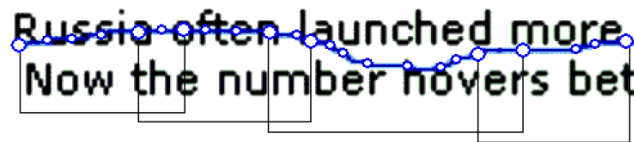
A drawback of the XLibris project was that documents must exist in the XLibris framework and most digital documents do not. One approach is to annotate web pages. Web pages are not static and are dynamically reformatted when the text size, font or window width changes. If the underlying text can move and shift, the ink marks on the text quickly get out of synch. What are needed are ink marks that can track and follow the text as shown in figure 32.



**Figure 32 – Moving marks when reformatting<sup>20</sup>**

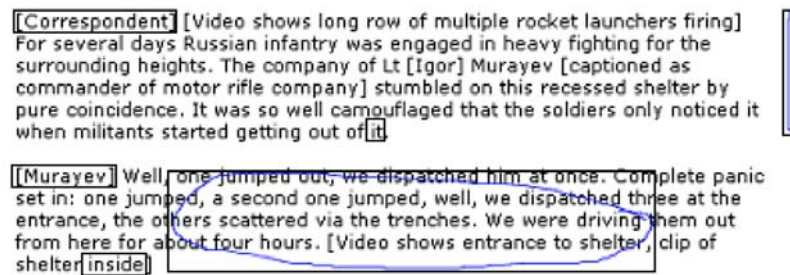
In order to move marks in the presence of reformatting, the ink must be represented in different coordinates. To accomplish this marks are classified into three categories: 1) direct text marks (underline or highlight), 2) margin bars and circled passages and 3) other notes. These classes of strokes can be distinguished using a subset of the Rubine features.

For direct text marks, the ink is broken into overlapping fragments tied to individual words as shown in figure 33. Each ink fragment is stored with a reference to the word to which it is attached and coordinates that are relative to the bounding box for that word. As the word moves and is resized, the ink stroke is moved and resized accordingly. When words remain together, the overlapping of the individual strokes ties them cleanly in what appears to be a continuous stroke.



**Figure 33 – Segmented word marks<sup>20</sup>**

Margin bars and circled passages are associated with regions of text rather than individual words. The associated words are discovered by projecting the bounding box of the stroke horizontally, as shown in figure 34, and then remembering the first and last word of the selected text. The stroke coordinates are relative to the bounding box of the text. As the text is reformatted, the first and last words identify the associated range of text from which a new bounding box is computed. The stroke is drawn in the coordinates of the new bounding box. If the new region flows across a page boundary, the stroke must be split in a corresponding way.



**Figure 34 – Margin bars and circled text<sup>20</sup>**

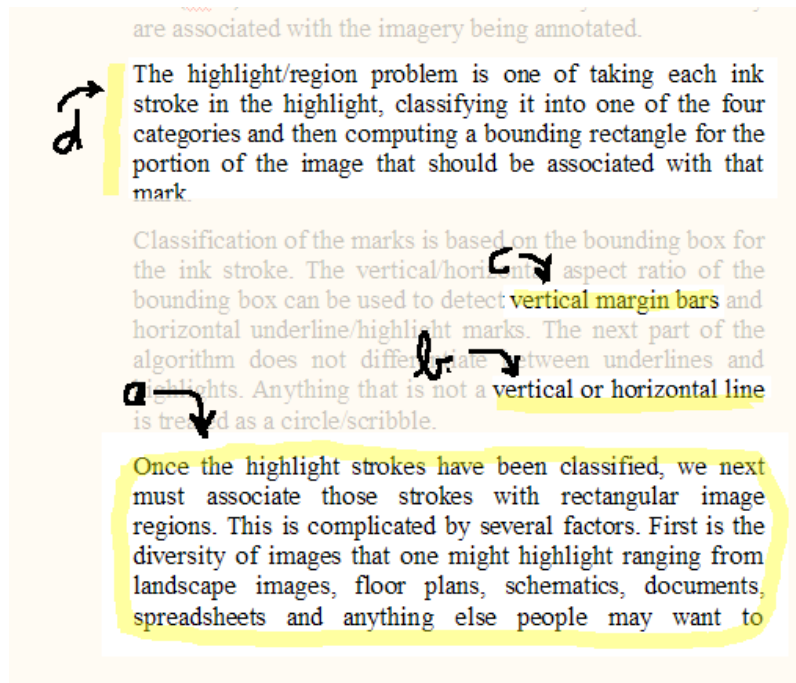
Other marks, such as comments or little drawn stars to highlight a point, should not be resized along with the text. They only need to be repositioned with the text. These marks are associated with text using the same projection technique as margin bars. These marks are always drawn in the same size with only their position defined by the corresponding text bounding box.

### **Annotating screen shots**

The XLibris project confined annotation to documents in a specific format under its own system. This restricts annotation to a small subset of the digital artifacts that people may use. XLibris was extended to web documents, which further expanded its reach. The ScreenCrayons<sup>22</sup> system went further in supporting annotation of anything that appears on the screen. ScreenCrayons operates by taking an image capture of the screen and then overlaying the screen with a borderless window that contains the captured image. This appears like the original screen except that the underlying applications are no longer active. The image is overlaid by a gray transparent layer. The user annotates the image using the XLibris gesture set. The relevant regions for selection are identified by analyzing the white space. When a region is selected, the gray overlay is removed over that region to fully expose the annotated information. In addition



the user can add typed or pen drawn notes or diagrams on the unannotated gray regions (figure 35).



**Figure 35 – Screen Crayons Annotations<sup>22</sup>**

Annotations are associated with “crayons” that are each associated with some topic of interest to the user. All of the annotations associated with that topic are collected together. ScreenCrayons uses the enclosure hierarchy defined by the annotation (regions within regions) to organize a condensed representation of the annotations. The user can then zoom in for more detail on any region including expanding to the full screen capture. The advantage of ScreenCrayons is that notes and other information are easily captured but the context is also retained without any additional effort on the part of the user.

### **Paper annotation**

XLibris and ScreenCrayons support annotation of digital documents on the computer itself. Using the Anoto pen we can extend annotation to actual paper documents. Paper Augmented Digital Documents (PADD)<sup>23</sup> takes digital documents and prints them on paper with the Anoto pen pattern. The printing

system knows which parts of which pages of which documents are printed at every location on the page. When the user writes on the page with the pen, the pen remembers the location of every stroke. These strokes from the pen can then be combined with the original documents to provide many of the advantages offered by XLibris and ScreenCrayons. The PapierCraft<sup>24</sup> system added gestures to the basic system. A user can use pen gestures on an original document to select information and then make gestures on a page of notes to paste in that information and its reference. When the strokes are retrieved from the pen, the notes are reconstructed and the commands interpreted so that the information can be retrieved from the original documents and placed into the digital notes as in figure 36.

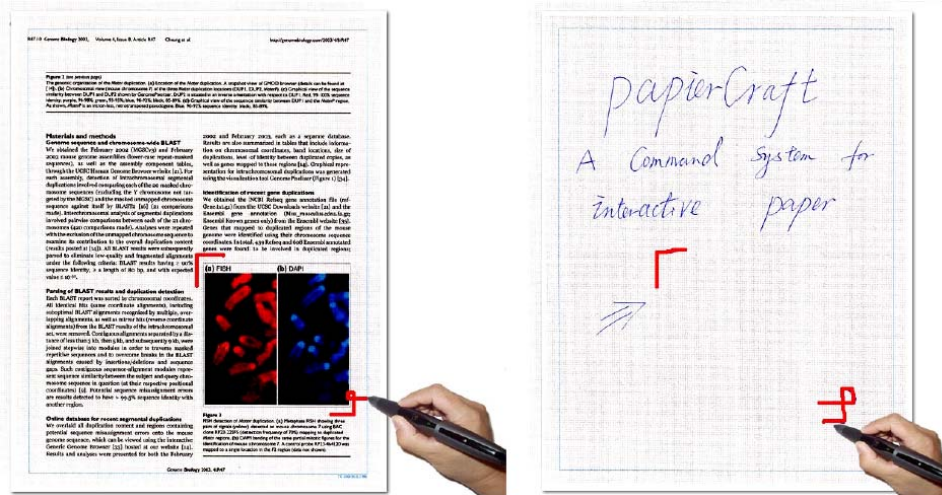


Figure 36 – PapierCraft copy and paste<sup>24</sup>

This ability to integrate other digital information with notes in a physical notebook is found in the ButterflyNet<sup>25</sup> system that supports field biologists. They write their notes in a field notebook as is traditional, but using the Anoto pen they can integrate digital images and samples into the notes for reconstruction digitally when the contents of the pen are retrieved.

## Summary

By rapidly sampling the location of a digital pen one can simulate the appearance of digital ink. This ink can be recognized as gestures that drive the user interface without menus. The ink can also be used as a free form data representation that is more flexible than simple pen and paper. The ink strokes

can exist on their own or as annotations on existing documents either digital or physical.

---

<sup>1</sup> [www.mimio.com](http://www.mimio.com)

<sup>2</sup> [www.wacom.com](http://www.wacom.com)

<sup>3</sup> Dietz, P., and Leigh, D., "DiamondTouch: A Multi-User Touch Technology", *User Interface Software and Technology (UIST '01)*, ACM (2001), pp 219-226.

<sup>4</sup> [www.anoto.com](http://www.anoto.com)

<sup>5</sup> Tappert, C.C. "Cursive Script Recognition by Elastic Matching," *IBM Journal of Research and Development*, 26(6), (1982), pp 756-771.

<sup>6</sup> Kristensson, P., and Zhai, S., "SHARK<sup>2</sup>: A Large Vocabulary Shorthand Writing System for Pen-based Computers," *User Interface Software and Technology (UIST '04)*, ACM (2004), pp 43-52.

<sup>7</sup> Forsberg, A., Dieterich, M., and Zeleznik, R. "The Music Notepad", *User Interface Software and Technology (UIST '98)*, ACM (1998), pp 203-210.

<sup>8</sup> Goldberg, D., and Richardson, C., "Touch-Typing With a Stylus," *InterCHI '93*, ACM, (1993), pp 80-87.

<sup>9</sup> Blinkenstorfer, C. H. "Graffiti," *Pen Computing*, (January 1995), pp. 30-31.

<sup>10</sup> References from Newman and Sproull.

<sup>11</sup> Rubine, D., "Specifying Gestures by Example," *ACM Conference on Computer Graphics (SIGGRAPH '91)*, ACM (1991), pp 329-337.

<sup>12</sup> Moran, T. P., Chiu, P., van Melle, W. and Kurtenbach, G., "Implicit Structure for Pen-based Systems within a Freeform Interaction Paradigm", *Human Factors in Computing Systems (CHI '95)*, ACM (1995), pp 487-494.

<sup>13</sup> Moran, T. P., Chiu, P., and van Melle, W., "Pen-based Interaction Techniques for Organizing Material on an Electronic Whiteboard", *User Interface Software and Technology (UIST '97)*, ACM (1997), pp 45-54.

- <sup>14</sup> Li, Y., Hinckley, K., Guan, Z., and Landay, J. A., "Experimental Analysis of Mode Switching Techniques in Pen-based User Interfaces", *Human Factors in Computing Systems (CHI '05)*, ACM (2005), pp 461-470.
- <sup>15</sup> Hinckley, K., Baudisch, P., Ramos, G., and Guimbretiere, F. "Design and Analysis of Delimiters for Selection-Action Pen Gesture Phrases in Scriboli," *Human Factors in Computing Systems (CHI '05)*, ACM, (2005), pp 451-460.
- <sup>16</sup> Saund, E., and Moran, T. P., "A Perceptually-Supported Sketch Editor", *User Interface Software and Technology (UIST '94)*, ACM (1994), pp 175-184.
- <sup>17</sup> Mynatt, E. D., Igarashi, T., Edwards, W. K., and LaMarca, A., "Flatland: New Dimensions in Office Whiteboards", *Human Factors in Computing Systems (CHI '99)*, ACM, (1999), pp 346-353.
- <sup>18</sup> Baudel, T., "A Mark-based Interaction Paradigm for Free-Hand Drawing", *User Interface Software and Technology (UIST '94)*, ACM (1994), pp 185-192.
- <sup>19</sup> Igarashi, T., Matsuoka, S., Kawachiya, S., and Tanaka, H., "Interactive Beautification: a Technique for Rapid Geometric Design", *User Interface Software and Technology (UIST '97)*, ACM, (1997), pp 105-114.
- <sup>20</sup> Schilit, B. N., Golovchinsky, G., and Price, M. N., "Beyond Paper: Supporting Active Reading with Free Form Digital Ink Annotations," *Human Factors in Computing Systems (CHI '98)*, ACM (1998) pp 249-256.
- <sup>21</sup> Golovchinsky, G., Price, M. N., and Schilit, B. N., "From Reading to Retrieval: Freeform Ink Annotations as Queries," *SIGIR '99*, ACM (1999), pp 19-25.
- <sup>22</sup> Olsen, D. R., Taufer, T., and Fails, J. A., "ScreenCrayons: Annotating Anything," *User Interface Software and Technology (UIST '04)*, ACM, (2004), pp 165-174.
- <sup>23</sup> Guimbretiere, F. "Paper Augmented Digital Documents," *User Interface Software and Technology (UIST '03)*, ACM, (2003), pp 51-60.
- <sup>24</sup> Liao, C., Guimbretiere, F., and Hinckley, K., "PapierCraft: A Command System for Interactive Paper", *User Interface Software and Technology (UIST '05)*, ACM, (2005), pp 241-244.
- <sup>25</sup> Yeh, R., Liao, C., Klemmer, S., Guimbretiere, F., Lee, B., Kakaradov, B., Stamberger, J., and Paepcke, A., "ButterflyNet: A Mobile Capture and Access System for Field Biology Research", *Human Factors in Computing Systems (CHI '06)*, ACM, (2006), pp 571-580.

