Lauri Vanhatalo

Online Sketch Recognition: Geometric Shapes

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Helsinki May 29, 2011

Thesis supervisor:

Prof. Eljas Soisalon-Soininen

Thesis instructor:

M.Sc. (Tech.) Ari Talja

Aalto University
School of Science
and Technology

AALTO UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

ABSTRACT OF THE
MASTER'S THESIS

Author: Lauri Vanhatalo

Title: Online Sketch Recognition:
Geometric Shapes

Date: May 29, 2011 Language: English Number of pages:7+89

School of Science

Department of Computer Science and Engineering

Professorship: Software Technology Code: T-106

Supervisor: Prof. Eljas Soisalon-Soininen

Instructor: M.Sc. (Tech.) Ari Talja

Effective sketch recognition is the basis for pen and touch-based human-computer interfaces. In this thesis the concepts, common methods and earlier work in the research area of online symbol recognition are presented. A set of shape recognition approaches proposed in the past by various research teams are briefly introduced. An online shape recognizer using global geometric features is described.
The preprocessing and feature extraction algorithms as well as the shape classification method are described in detail. Recognition heuristics for two simple shapes (arrow and star) are suggested. A graphical user interface was implemented and a group of subjects employed to obtain realistic results of the computational performance and recognition accuracy of the system: the implemented system performs fast but the results on the recognition accuracy were ambiguous. Finally, the problems and restrictions of the approach are discussed.

Keywords: Online sketch recognition, Computational geometry, Stroke curve preprocessing, Pen-based user interface

AALTO-YLIOPISTO
TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Lauri Vanhatalo

Työn nimi: Geometristen muotojen reaaliaikainen tunnistus

Päivämäärä: May 29, 2011 Kieli: Englanti Sivumäärä:7+89

Perustieteiden korkeakoulu

Tietotekniikan laitos

Professuuri: Ohjelmistotekniikka Koodi: T-106

Valvoja: Prof. Eljas Soisalon-Soininen

Ohjaaja: DI Ari Talja

Kynä- ja kosketuskäyttöliittymät vaativat toimiakseen tehokasta ja tarkkaa hahmontunnistusta. Tässä työssä esitellään reaaliaikaisen hahmontunnistuksen käsitteistöä, yleisiä menetelmiä ja aikaisempaa tutkimusta. Lyhyesti käsitellään eri tutkimusryhmien esittämiä hahmontunnistusjärjestelmiä. Lisäksi esitellään geometrisiin piirteisiin perustuva hahmontunnistusjärjestelmä.

Työ antaa yksityiskohtaiset kuvaukset piirtoviivan esiprosessointi- ja piirteenirrotusalgoritmeista sekä hahmoluokittelumenetelmästä. Lisäksi kuvaillaan hahmontunnistusheuristiikka kahdelle yksinkertaiselle muodolle (nuoli ja tähti). Joukko koehenkilöitä käytti työssä toteutettua graafista käyttöliittymää, minkä tuloksena saatiin realistiset tulokset järjestelmän laskennallisesta suorituskyvystä ja tarkkuudesta: toteutettu järjestelmä on laskennallisesti nopea mutta tunnistustarkkuus monitulkintainen. Lopuksi pohditaan valitun lähestymistavan ongelmia ja rajoitteita.

Avainsanat: Reaaliaikainen hahmontunnistus, Laskennallinen geometria, Piirtoviivan esiprosessointi, Kynäkäyttöliittymä

Preface

In a way, this spring has been the culmination of rush and disciplined labor for me. In addition to reading articles, programming and writing the thesis, the work load has included courses, weekly paid work and hobbies. After finishing this thesis I am a huge step closer to completing one degree. While taking that step, an incredible amount of work has also been done for taking the next one.

I want to thank Professor Eljas Soisalon-Soininen for making this thesis possible and for his professional guidance during the spring. I am especially grateful to my instructor Ari Talja for the discussions that removed my blinders at the end of last year and made it possible to finish this thesis in schedule which is likely to please every party involved.

My biggest thankyou is dedicated to my mother, my father, Laura, Jaakko and Joonas. The unfailing support and understanding of my family has made my work straightforward.

In addition, I want to thank my friends for listening, for all the conversations and partly for maintaining my mental health. I also want to thank a group of members of Satakuntalainen Osakunta for participating in the thesis as subjects.

Helsinki, May 29, 2011

Lauri S. Vanhatalo

Contents

Abbreviations

CAD	computer-aided design
CR	character recognition
DCR	direction change ratio
GIS	geographic information system
HMM	hidden Markov model
LADDER	a language to describe drawing, display and editing in sketch recognition
NDDE	normalized distance between direction extremes
OCR	optical character recognition
PC	personal computer
PDA	personal digital assistant
SVG	scalable vector graphics
UI	user interface
UML	unified modeling language
XML	extensible markup language

1 Introduction

The computer mouse was invented over 40 years ago [1, 2]. The roots of the computer keyboard lie even further in the invention of the mechanical typewriter. Despite these facts the mouse and the keyboard have been the "de facto standard" input methods of precise, organized data by human users to a computer since the emergence of electric, digital devices. A certain sequence of keyboard strokes produces the desired sequence of characters in a word processor. To produce a rectangle in an image manipulation program, tool selection and another sequence of mouse movements and clicks is required. Both scenarios illustrate an established, simple and unambiguous yet somewhat unnatural means for data input. That is, simple and unambiguous for the device receiving the input; unnatural for the human user. The reverse approach would be freehand writing and drawing which is more natural for the human user but presents a complex and ambiguous analysis problem for the device.

Using freehand sketches as an input to digital devices concerns three loosely interrelated aspects: input hardware, algorithmic methods and computational power. A sufficient level of sophistication is required of each three aspects. The kind of hardware needed, that is *graphics tablets* or *digitizers* have been available as early as 1950's with the introduction of the "stylus translator" *Stylator*. Also the *RAND Tablet*, introduced in 1964, led the way of computer input devices using freehand drawing [3]. It was not until mid 1970's and early 1980's that the digitizers were popularized along with the emergence of CAD (*computer-aided design*) systems [4]. However, the mouse and the keyboard have not been superseded by the digitizers as input devices although the latter have their applications especially in the field of computer graphics.

In the 21st century the development in *touchscreen* technology has led to a rapid increase in devices utilizing a display that can detect the presence and location of a touch within the display area. Tablet PCs and touchscreen cell phones are currently challenging the traditional human-computer interaction paradigm based on a keyboard used to input discrete data and a mouse used for freehand interaction such as drawing. *Interactive whiteboards* are becoming common in offices and educational facilities. However, the hardware capabilities are only partly utilized since the touchscreen is often displaying a so called *soft keyboard*, which is used as traditional keyboard through the touchscreen interface. Additionally, the touchscreen also functions merely as a replacement for mouse or other pointing device providing a more natural way of interaction. Thus the underlying, conceptual model of data input is no different than using a traditional keyboard and a mouse. With the utilization of proper algorithms the quality of data input can be improved.

The data input of interactive whiteboards and touchscreens is fundamentally a sequence of points in a cartesian coordinate system. The sequence constitutes a path that can represent, for example, handwriting (characters), geometric shapes (rectangles, ellipses) or more complex symbols (components in electronic circuit diagram). A dedicated piece of software, however, is needed to analyze the raw sequence of points and to conclude whether the points represent some more organized

input. The first and most obvious application is the analysis of handwriting. This is partly due to the nature of early computers as mainly text processing devices as opposed to graphics manipulation.

The birth of graphics tablets precipitated the active research on online handwriting recognition in the late 1950's and through 1960's [5]. The activity abated in the 1970's and increased again in the 1980's. The renewed interest in the field of algorithms in 1980's and 1990's was based on the rapid development on all three fronts: more accurate tablets, more compact and powerful computers and more sophisticated algorithms. By the end of 20th century the contemporary state of the art recognition systems had reached the maturity level needed for commercial uses in, for example, hand-held computers and PDAs (personal digital assistant) [6]. In the 21st century, the rapid spread and popularization of new devices has kept the field topical.

Online graphics recognition has gained more research interest recently. The increased interest in the field in 1990's and in the beginning of the new millennium coincides with the advances in computer technology. Graphical user interfaces and the utilization of computers in image manipulation have seen daylight.

Liu states the importance of graphics as a means for expressing and communicating information [7]. Furthermore, he observes the limitations of conventional ways of working with and inputting graphics. Most graphics input/editing systems rely on a mouse and a keyboard and a multitude of toolbars, buttons and menus for inputting shapes and executing editing operations.

> Admittedly, the most natural and convenient way to input graphics is to draw sketches on a tablet using a digital pen, just like drawing freehand graphics on a real sheet of paper using a pencil. However, the sketchy graphic objects and freehand drawings drawn in this way are usually not clear (or not neat/beautiful) in appearance, not compact in representation and storage, and not easy for machines to understand and process. [7, p. 291–292]

The user interface can be improved if the freehand sketches can be recognized and converted into the regular shapes intended by the user. The improvement is even more significant if the recognition can be done online. That is, while the user is drawing and hence gets immediate feedback of his actions. Achieving a usable online recognition system imposes obvious performance requirements on the algorithms and the computational power available.

The development of digitizers and other hardware enabling touch-based interfaces has taken place simultaneously with the rise of computational power provided by the integrated circuits (processors). The *Moore's law* predicting the number of transistors that can be placed on an integrated circuit has held for over 40 years [8]. The law proposed by Intel co-founder Gordon Moore in 1965 and slightly adjusted in 1975 describes the exponential growth in computing capacity that can be achieved at a reasonable cost [9, 10]. This has extensive implications on the set of methods available for online graphics recognition. Algorithms that were computationally too heavy five years ago are probably more than feasible using today's technology.

Computers have not only become more efficient but smaller as well. This enables a variety of applications also on hand-held devices.

This thesis presents an approach for analysing the geometric features of a set of points drawn by a human user. Futhermore, the features are utilized to deduce whether the set of points represents a known geometrical shape. That is, a method is described for converting hand-drawn sketches into triangles, rectangles, ellipses, circles, et cetera.

First, the essential concepts of online and offline shape recognition are clarified. Second, the problem of online sketch recognition is described together with applications for the recognition systems. Common methods and a set of overall approaches to the problem in the literature are discussed. Third, the approach on which this thesis builds up on is presented. Along the way the problem set and its details and subtleties are discovered and explained. The following sections explain in detail the various algorithms and heuristics used. Finally, an approach to sketch recognition is presented and its performance evaluated.

This thesis aims to measure the accuracy, applicability and performance of a sketch recognizer based on geometric feature extraction. Is the accuracy of the recognition adequate to improve the easiness and quality of graphical data input? Is the computational performance of the method good enough and the delays involved small enough to achieve real time recognition and enjoyable user experience?

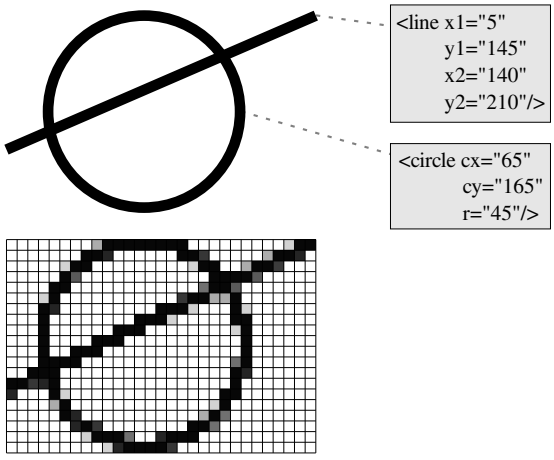
2 Shape recognition: concepts and methods

The field of shape recognition is a wide research area with various subfields and an overlap with such fields of research as pattern recognition, artificial intelligence and computer vision. This section focuses on the clarifying the nature of certain subfields of shape recognition. That is, the essential concepts and background of *optical character recognition* (OCR) and *graphics recognition* are provided as well as an overview of the application domain. Similarities and differences between specific research problems and related methods are illustrated. The problem of *online sketch recognition* is presented in more detail together with the essential research objectives and past advances. A selection of sketch recognition systems in the literature are visited and their approaches and methods briefly described.

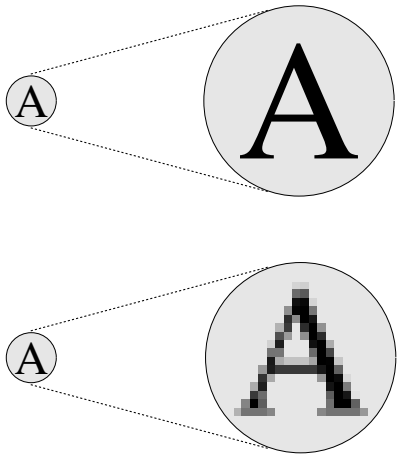
2.1 Vector versus raster graphics

In the branch of computer graphics there are two fundamental ways of representing image data [11]. *Raster graphics* employs the notion of *pixel* (short for *picture element*) which is the smallest building block of an image. Pixels are tiny squares that have a color associated with them. A raster image is composed of a series of pixels grouped together to form a grid or a matrix (see figure 1a). Putting enough pixels in the grid and thus making them blend together creates an illusion of a single, continuous image. Hence, the amount of information in the image depends on the amount of pixels, that is the *resolution* of the image.

Instead of pixels *vector graphics* uses mathematically described shapes to store



(a) Vector image (above) uses mathematical expressions to store the image information (end point coordinates for the line, center point coordinates and radius for the circle). On the other hand, raster image (below) uses a grid of pixels for storage and representation.



(b) Scaling (magnifying) vector and raster graphics. When a vector image (above) is magnified the sharpness and details are conserved. On the contrary, a raster image (below) pixelates and loses its precision.

Figure 1: The differences between vector and raster graphics.

the image information. A vector image consists of objects that represent plots of mathematical expressions together with rendering attributes such as color. The objects are stored as separate elements which allows manipulating them without affecting the others. Vector graphics often requires much less memory compared to raster graphics as there is no need to store thousands or millions of pixel values. Using the mathematical presentation of shapes also has the benefit of resolution independence. Scaling the image does not degrade the sharpness of the image (see figure 1b). Consequently it is often beneficial to use vector graphics. One of the objectives of shape recognition is to vectorize input data for easier handling and more efficient storage.

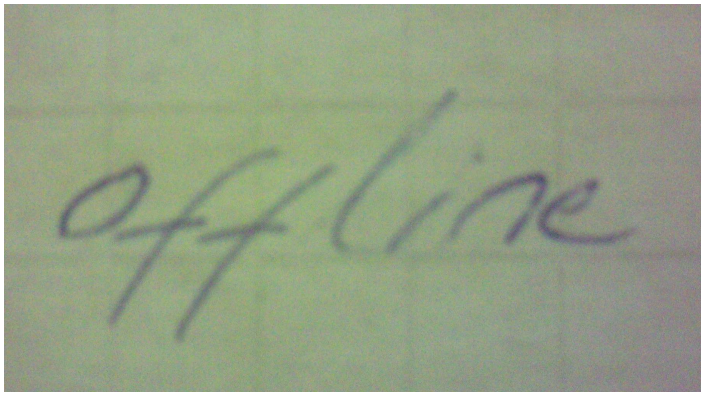
2.2 Online versus offline

When discussing character and shape recognition an essential distinction has to be made between *online* (sometimes referred to as *real time* or *dynamic*) and *offline* recognition [5, 12]. In online recognition the character or shape recognition process is executed while the user is writing or drawing (or shortly after). Thus, online recognition requires hardware that converts the strokes drawn by the user into machine interpretable signals. Electronic tablets or digitizers are used for the task. On the contrary, offline recognition can be done at any point after the writing or drawing is completed. The input is obtained by scanning the image containing the user strokes or handwriting.

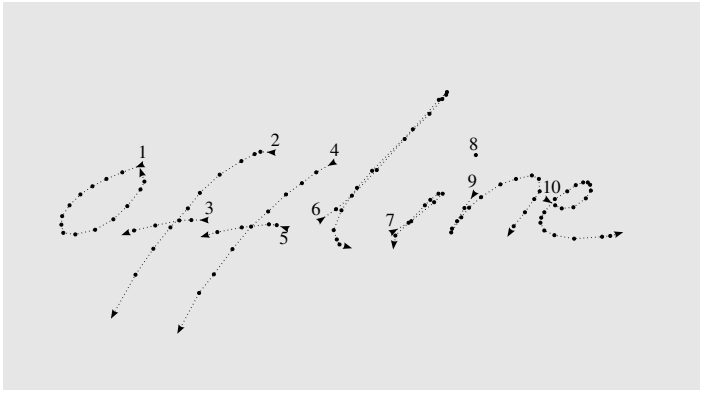
Writing — until modern times particularly handwriting — has been a key ingredient in the creation of the current level of culture and civilization [6]. Consequently, the research on the field of offline character recognition methods is abundant [6, 12, 13, 14, 15, 16]. This is due to several aspects. The development of digital computers exposed a wide range of applications for *optical character recognition* (OCR) systems [14]. The amount of manually handled paper such as bank cheques, commercial forms, credit card imprints and mail was already enormous. Other applications include aiding visually handicapped, signature verification and automatic number plate reading. There is an abundance of literature in non-digital form created before the computer era and digitization projects require automation and OCR [17, 18, 19]. A major research area is the recognition of Chinese (as well as Japanese) characters. Stallings points out that

> To make available to Westerners the culture and technology of one-quarter of the human race some form of automation must be introduced. [20, p. 87]

Although the approaches in online and offline recognition methods are different the problem set and their solutions overlap extensively [13]. Thus, it is extremely useful to understand the basic approaches, methods and applications concerned in offline recognition. Similarly, there is an inherent relation between character and shape recognition [21]. For example, mathematical formula interpretation is a recognition task that falls inevitably into both categories since it involves both symbol recognition and two-dimensional structure interpretation [22]. Furthermore,



(a) Raw (offline) image data



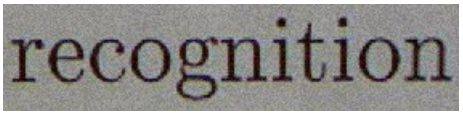
(b) Similar data as in (a) presented as point trajectory data. Information on order and drawing direction of the strokes is readily available.

Figure 2: Online vs. offline recognition.

understanding the challenges in offline recognition helps to grasp the benefits of using online methodologies.

The fundamental difference between online and offline recognition is the nature of the input data. Online recognition relies on the two-dimensional, ordered coordinate points that are recorded together with temporal data (timestamps). In offline case the input is a digital image usually obtained by scanning or by photographing analog media (see figure 2) [6]. The digital image consists only of a matrix of pixels with related color values. The image data as such is however far from usable. Several imperfect and costly *preprocessing* steps have to be executed before the shape or character recognition phase [13]. The purpose of these steps is to exclude irrelevant information and include relevant information in the input image. That is, the user strokes or printed symbols should be extracted and coffee stains, background texture and noise from other sources discarded.

Thresholding is used to differentiate objects from the background of the image. The output of this process is a binary (black-and-white) image that should include drawn or printed characters or shapes as accurately as possible. An exhaustive survey on different thresholding techniques and discussion on their performance



(a) Digitized (scanned) raw input image

(b) Thresholded black-and-white image with noise (small black dots between and inside the letters)

(c) Denoised image

(d) Thinned image

Figure 3: Preprocessing steps of offline recognition.

evaluation is provided by Sezgin [23]. The digitalization process, originally impaired image quality as well as storage and copying can introduce noise in the resulting image data. In order to improve the recognition performance some sort of *noise reduction* (or *denoising*) must be used. In addition to offline symbol recognition image denoising has various other applications and has been discussed extensively in the literature [24, 25]. Finally the black-and-white image is used as the input for *thinning* which is the process of reducing patterns to thin-line representations [26]. The thin-line or "skeleton" representation aims to retain the topological and geometrical properties of the object and makes it more suitable for feature extraction and thus further analysis. The steps involved in preprocessing are depicted in figure 3.

The preprocessing steps described above are common for all offline recognition problems whether regarding characters or graphics. The second step is *segmentation* which is sometimes considered as belonging to preprocessing. In character recognition, both machine printed and handwritten, the text has to be split (segmented) first into lines and then further into words and characters [6]. Naturally the task is easier for machine printed documents since the variations in handwriting hinder the segmentation. In some scenarios the segmentation problem can be solved by providing a grid with each character written within a box. This approach is used for example in certain forms and questionnaires that need to be analyzed automatically. Online recognition have adopted the grid approach as well [27, 28]. In the absence of the grid that guide the process, consistently spaced characters are easier to segment than cursive script [5]. Finally, the features of the segmented character are extracted and the character is assigned to one of the classes representing for example the upper and lower case letters, the ten digits and special symbols.

In the online case much more refined information can be provided to the recognition process. The input data is inherently ordered sequences of x-y-coordinate pairs with the corresponding timestamps. The number and order of strokes, the number and order of points within the strokes and the direction and speed of the stroke at any point is available. The quality and amount of input data depends on the sampling rate of the device in use. For example, the sampling rate of typical graphics tablets lie between 50–200 points per second (pps) [29, 30, 31]. Using a

computer mouse typically results in sampling rates between 10–100 pps depending on the nature of the stroke. This is due to the fact that a point is recorded only when the mouse is moved to a new location as opposed to using a constant sampling rate. In practice, modern devices are capable of providing data of a sufficiently high quality.

Besides hand-written and machine-printed character recognition there are numerous applications for offline graphics recognition. Traditional application domains for (offline) graphics recognition are proposed by Lladós et al. [21]. Automatic symbol recognition can be applied to documents with notation of variable degree of standardization such as logic circuit diagrams, engineering drawings, maps, musical scores and architectural drawings.

Different diagrams are of interest also with regard to online recognition [32]. Various other man-machine interfaces also require online graphics recognition which in that context is commonly referred to as *online sketch recognition* [33, 34, 35, 36, 37, 38, 39]. Hand-drawn sketches are used to input graphics but also control gestures to the application. Hand-held devices with touchscreen interfaces can be controlled using the gestures instead of using a set of buttons.

2.3 Problem description and objectives

Jin et al. [33] specifies the problem of online graphics recognitions as:

> Given a sketchy stroke of a closed-shape, determine the shape that the user intended to input. [33, p. 256]

More generally put, an ordered sequence of strokes (that are not necessarily closed) consisting of ordered sequence of coordinate points is to be converted into a user-intended shape such as a right-angled triangle with catheti perpendicular to the coordinate axes. Perhaps the most essential point of the problem is the user intention which should be recognized while ignoring irrelevancies caused by restrictions in hardware and human ability to draw perfect shapes especially when sketching quickly. Liu [7] divides the recognition process into four steps which are clarified in figure 4.

1. Stroke curve pre-processing

Input: a freehand stroke.

Output: a refined polyline.

Requirement: the redundancies and imperfections in the input are removed to produce the polyline.

2. Shape classification

Input: the refined polyline.

Output: a basic shape class such as line, triangle, ellipse or free curve.

Requirement: the output shape conforms to the user intention.

3. Shape fitting

Input: the shape class and the stroke (the original and refined polyline).

Output: the fitted shape.

Requirement: the fitted shape has the lowest average distance to the input stroke.

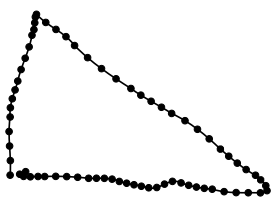
4. Shape regularization

Input: the fitted shape and the original stroke.

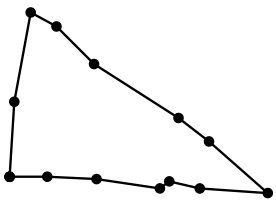
Output: the regularized shape.

Requirement: the regularized shape is similar to the original freehand stroke but also regularized (e.g. symmetrical polygon, perpendicular with the coordinate axes).

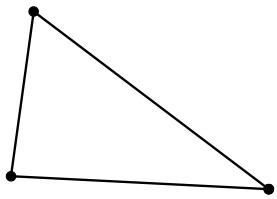
The four-step process above is called *primitive shape recognition*. Various approaches in the literature are roughly in line with the steps proposed by Liu [7] but often omit or combine some. Sometimes it is also ambiguous to categorize certain algorithms into specific steps. Liu [7] also includes two additional processes in online graphics recognition. *Composite graphic object recognition* is the process of combining two or more primitive shapes (obtained in the previous step) based on



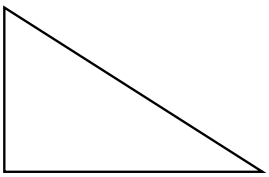
(a) Freehand stroke



(b) Preprocessing discards redundant points, removes a hooklet in the end of the stroke and closes the shape.

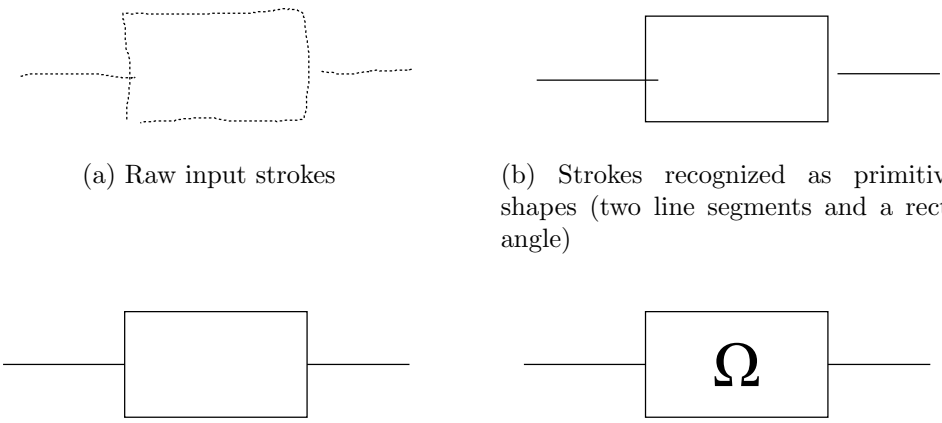


(c) Shape classified (and fitted) as a triangle



(d) Regularized triangle (a right-angle triangle with catheti collinear with the coordinate axes)

Figure 4: The four steps of primitive shape recognition



(a) Raw input strokes

(b) Strokes recognized as primitive shapes (two line segments and a rectangle)

(c) Primitive shapes combined into a composite object according to their spatial relationships

(d) The semantic of the composite object interpreted using context information (a resistor in an electronic circuit diagram)

Figure 5: The subprocesses of online graphics recognition

their spatial relationships. *Document recognition and understanding* refers to the understanding of the connections between the graphical elements as well as their semantics. An example is shown in figure 5. This thesis is focused on the primitive shape recognition process only.

Besides good recognition rate and computational performance a practical sketch recognition system should possess attributes related to user experience [34]. The usage must be as close as possible to that with traditional pen and paper. The recognition should be independent of the style of drawing. That is, a rectangle should be recognized whether it is drawn with a single stroke or four strokes. The essentially same fact is stated by Alvarado and Davis [37]. One of the most difficult problems in developing a sketch recognition system is handling the tradeoff between recognition accuracy and drawing freedom of the user. Heavy constraints on the drawing style makes the recognition easier. However, a need to sketch in a specific way contradicts with the probably most important requirement. That is, the usage was supposed to be as natural as with pen and paper.

2.4 Preprocessing

A freehand stroke drawn by human user is usually very cursive and inaccurate. Using an input method unnatural to drawing, such as computer mouse or finger on touchscreen device, further emphasizes the imperfections. Circles are not regular and lines intented to be straight might resemble arcs with considerable noise. Preprocessing aims to remove the noise and minor inaccuracies to make the strokes more similar to the user intention. This facilitates the further analysis and recognition of the sketches.

Liu et al. have divided preprocessing into four steps: polygonal approximation,

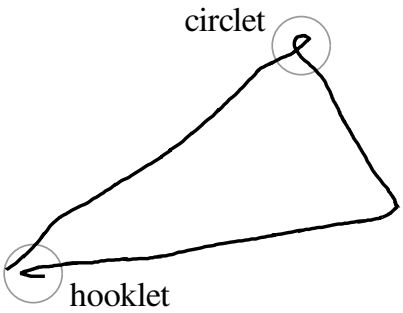


Figure 6: Unintentional noises in the sketchy stroke: hooklet and circlet.

agglomerate points filtering, end point refinement and convex hull calculation [7, 40]. The first three are discussed here since the convex hull calculation is used specifically for closed-shape recognition. The input hardware usually produces a lot more points than are necessary to accurately define the shape of the stroke. Substantial amount of points of a straight-line segment are redundant since the start and end point approximate the line segment accurately enough. Usually these non-critical points are of no use for the further recognition steps and can thus be discarded. This process is referred to as polygonal (or polyline) approximation.

Using a digitizer and a pen might produce hooklet-like segments at the ends of the stroke. Similarly, circlets may be introduced at the turning points of the stroke. These artifacts are potentially harmful in the later steps of the analysis. A hooklet might produce a false impression on to which direction the end of the stroke is pointing, where as a circlet can generate an unintentional self-intersection in the stroke (see figure 6). Agglomerate points filtering is used to remove these noises by examining the point densities of the input polyline. Segments with a hooklet or a circlet usually have much higher point densities than the average value of the whole polyline [33].

Drawing a perfectly closed shape imposes a challenge to the user. Usually a stroke intented to be for example a polygon is not properly closed or forms a cross near its endpoints. End point refinement can be used to close a nearly-closed shape properly to make the recognition easier.

2.5 Shape classification

In pattern recognition, classification is regarded as the problem of classifying an unknown input pattern into a predefined class. In sketch recognition deciding whether a user-drawn scribble represents a predefined shape (rectangle, circle, arrow) is called shape classification. Since shape classification is the most essential part of shape recognition, the terms are sometimes used interchangeably. Furthermore, in the context of shape recognition, shape classification is often the main focus of the scholarly articles [7]. The gamut of classification methods is abundant. This is due to the amount of detail in the process and the possibility to combine parts from different

approaches. However, some effort has been made to categorize the approaches.

In general, shape classification starts by extracting useful information from the input data to facilitate the classification. Lladós et al. use the traditional categorization of pattern recognition in the context of offline symbol recognition [21]. The categorization is applicable also to online recognition. In *statistical symbol recognition* the extracted information is referred to as features. The input data are represented as an n-dimensional feature vector. The criterion for selecting the features is to minimize the distance among the input patterns belonging to the same class and maximizing the distance between the patterns belonging to different classes. Once the features are extracted the classification can be done using either k-nearest neighbors, decision tree or neural networks. In k-nearest neighbor the classification is done by selecting the class the representatives of which are closest to the sample. Decision tree consists of a tree of simple decision rules that use the features to select the shape class. Training sets are used to train neural networks to reach optimal parameters for classifying future unknown input. The challenge of the statistical approach is to find relevant features that can handle noise and transformations of the shape of the same class.

In *structural symbol recognition* the extracted information are the primitive building blocks of the input pattern. For example, a rectangle consists of four lines (geometric primitives) with certain constraints. The primitives and their relations are further compared to the models for each class built using the primitives. Some approaches use the notion of formal grammars. The grammar is language that can be used to accurately describe all accepted shapes and symbols. The input is tried to parse and test whether it can be generated using the grammar. This approach is suitable for analyzing technical drawings with clearly defined symbols such as circuit diagrams. In online sketch recognition a formal, extensible grammar is developed aiming to unify the future software [41]. Hidden Markov Models (HMM) are also regarded as a structural method by Lladós. In HMM approaches the input is modeled as a sequence of states. Probabilities are related with the states and the classifying is based on finding the state sequences with high probabilities.

2.6 Direction, curvature and speed

The information on the order of the points can be used to derive more refined features of the stroke. These features can be further used to facilitate the shape classification. Yu and Cai use *direction* and *curvature* (change in direction with respect to path length) in their sketch recognition system [34]. Direction of the stroke d_n at point n is given by

$$d_n = \arctan(\frac{y_{n+1} - y_n}{x_{n+1} - x_n})$$

and curvature c_n by

$$c_n = \frac{|\sum_{i=n-k}^{n+k-1} \phi(d_{i+1} - d_i)|}{D(n-k, n+k)}$$

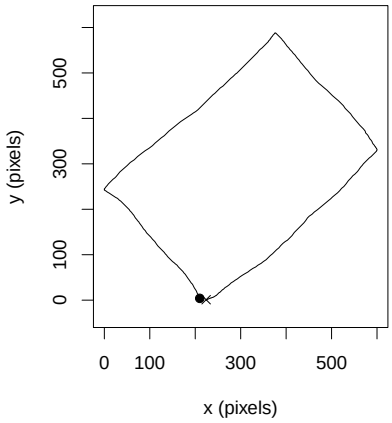
where k is a small integer (empirically set to 4) defining the neighborhood size around the n-th point and $D(n-k, n+k)$ stands for the path length between the $(n-k)$-th and $(n+k)$-th point of the stroke. The function ϕ converts the angle parameter to fall from $-\pi$ to π.

Figure 7 shows a typical input stroke alongside with its direction and curvature graphs. Curvature can be used to detect so called *feature points* which in the case of a polygon are simply the vertices. The data is usually smoothed before further analysis [35]. This is done also for the direction graph data in figure 7. The smoothing is implemented by averaging a few consecutive values in the data. Using smoothing deals with the noise caused by discrete coordinate values and the high sampling rate of the hardware. Thus smoothing also applies to the curvature graph that is derived using the direction values.

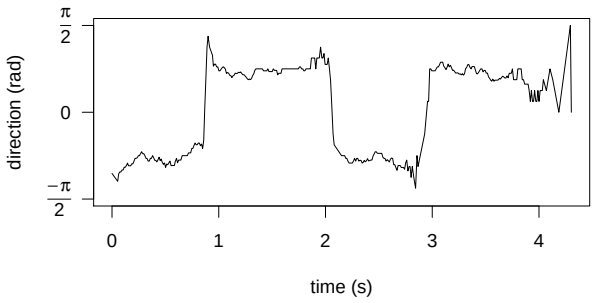
Another approach to differentiate between the correct feature points and noise is described by Sezgin [42]. He scales the mean of the curvature values and uses it as a threshold when searching for feature points.

The temporal data enables the utilization of the stroke speed information. Calhoun et al. use the speed profile of the stroke to identify *segment points*, that is the points that divide the stroke into primitives [35]. The segmentation procedure is further discussed in [43]. The observation is made that the pen speed is reduced when making intentional discontinuities like the corners of a polygon (see figure 8). However, as Yu points out the visual features of the stroke are inherently more reliable concerning the analysis [34].

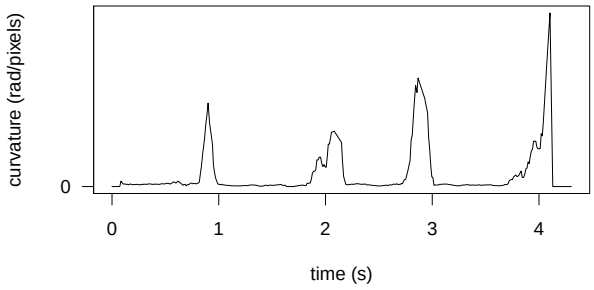
In online recognition the strokes are already separated in the input data thus making the segmentation rather trivial [44, 45]. On the other hand it is proposed that the online scheme may increase the complexity of the problem by representing irrelevant details. For example the letter "E" consists of four strokes the order of which can vary between writers. However, the same static image is still produced. Consequently, researches have tried approaches where data is converted from offline to online, online to offline and where both are used to achieve a "hybrid" recognizer [44, 45, 46, 47]. In online case inherently more information is available since the online data is easily converted into (offline) bitmap image. Thus, online recognition systems perform slightly better than offline systems [48, 49].



(a) Input stroke. The start and end point of the stroke are indicated by the dot and cross respectively.

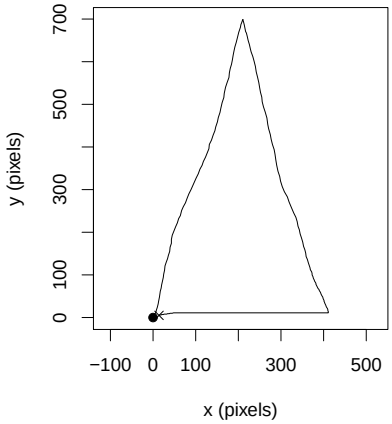


(b) In the direction graph the edges and vertices of the rectangle can be observed as horizontal "plains" and vertical "cliffs" respectively.

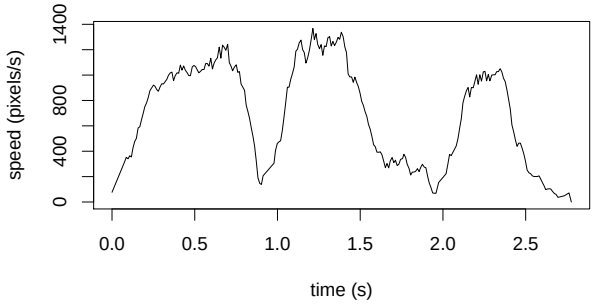


(c) Curvature graph contains inherently the same information as the direction graph. Vertices of the rectangle correspond to large (absolute) values of curvature.

Figure 7: The input stroke and corresponding direction and curvature graphs. The temporal correlation between direction and curvature is clearly visible.



(a) Input stroke. The start and end point of the stroke are indicated by the dot and cross respectively.



(b) The speed graph exposes the vertices of the triangle as local minima.

Figure 8: The input stroke and the corresponding speed graph.

2.7 Shape fitting and regularization

Once the sketch is classified, a shape from the class is fitted to the sketch. This is usually done by finding the parameters of the shape that minimize the average distance between the sketch and fitted shape. While shape fitting and regularization have gotten less focus than shape classification some approaches have been proposed.

Chen and Xie present fitting methods for lines, circles, circular arcs, ellipses and elliptical arcs [50]. Line fitting uses standard least squares method. Parameters of a circle (or a circular arc) can be deduced using three (non-collinear) points. Thus, Chen and Xie use a weighted average of centres and radii obtained by choosing all possible triplets of points in the sketch representing the circle. A more complex technique using Liming multiplier is required for ellipse fitting. Methods for ellipse and polygonal fitting are suggested also by Liu [40, 51]. Liu also discusses briefly a set of shape regularization rules that are described in more detail in [33].

Shape regularization aims to correct the drawing imperfections of a human user. Although a sketch might be correctly classified, for example as a rectangle, the edges meant to be horizontal or vertical are usually slightly skewed. Regularization (or beautification) rectifies such mistakes. This is called inner-shape regularization. Jin et al. suggest rectification rules also for producing equilateral polygons, parallel edges and special angles (multiples of 15°) [33]. Inter-shape regularization comprises making modifications to the fitted shapes according to other shapes in near proximity: their size, position and critical points.

Early work on shape regularization has been made by Pavlidis and Wyk [52]. They focus on imposing certain constraints on lines obtained by using edge extraction algorithms on digitized images. A clustering scheme is used to group the lines and corrections are made to their slopes, collinearity and length.

The work of Revankar and Yegnanarayana is also based on offline recognition that requires digitization and binarization of an image [53]. They construct threshold measures for line connectivity, relative orientation, equality and parallelism. The thresholds are used to create a set of rules for geometric shapes containing special forms of polygons (e.g. equilateral triangle, square) to facilitate the beautification task.

Igarashi et al. take an interactive approach where the recognizer constructs multiple beautified candidate shapes for the user to choose from [54]. Their system considers connecting stroke ends to a vertex or line segment, line parallelism, perpendicularity, alignment, congruence, symmetry and interval equality.

Paulson and Hammond's *PaleoSketch* system integrates beautification procedures to their primitive recognition [38]. The authors note that the endpoints of a stroke are significant when deducing the user's intention and thus adopt an "endpoint-significance theme". For example, a stroke recognized as a line is not fitted using the popular least squares method but by simply connecting the endpoints.

2.8 Sketch recognition systems

This section presents briefly various sketch recognition systems that have been proposed. The systems differ not only in their methods but their specificity, generalizability and constraints to the user. Some approaches are extremely domain-specific while others try to provide more domain-independent, general recognition framework.

Jin et al. propose "a novel and fast shape classification and regularization algorithm for on-line sketchy graphics recognition" meant for pen-based user interfaces [33]. The steps of the approach conform strictly to the ones depicted in figure 4. The preprocessing step consists of polygonal approximation, agglomerate points filtering, end points refinement and convex hull calculation. The preprocessing yields a convex polygon with n vertices. *Attraction force model* is used to combine adjacent vertices under a certain threshold to decrease the number of vertices from n to m. The shape is classified as a polygon or an ellipse using an intuitive, rule-based approach. There are independent shape fitting procedures for polygons and ellipses. For polygons, the least squares method is used to obtain the fitted edges and vertices that are further adjusted using the original stroke. For ellipses, the sampling points of the stroke are used to obtain the axes and the gravity centre of the ellipse. Minimizing the difference to the input stroke yields the fitted ellipse. Finally a set of regularization rules are applied to the fitted shape to obtain the user-intended shape. A set of inter-shape regularization rules are also presented. Shape classification precision of over 90% is reported.

Yu and Cai aim to use only low-level geometric features and no domain-specific knowledge to achieve "a domain-independent system for sketch recognition" that could be used as a foundation for higher-level applications [34]. Their approach tries to parse a freehand sketch into primitive shapes and simple objects that can be further used in domain-specific applications. The two-stage process first approximates the stroke with one primitive shape or a combination of primitive shapes. In a recursive algorithm the stroke is segmented using direction and curvature information. It is analyzed whether the segments can be represented by any of the primitive shapes. Differentiation between primitives (line, arc, circle, helix) is done based on geometrical features and stroke direction data. The post-process stage analyzes connectivity between the strokes, cleans up redundant primitives and executes basic (domain-independent) object recognition. Recognition rate for primitive shapes and polylines was reported to be nearly 98% and for hybrid shapes between 70–90%.

Calhoun et al. provide recognition system that allows symbols to be composed of multiple strokes [35]. The strokes are first divided into geometric primitives (lines and arcs). The primitives are obtained by segmenting the strokes using speed and smoothed curvature information. Shape classification takes a structural approach. A semantic network composed of geometric primitives and their relationships is used as a description for each shape. The set of shapes does not have to be defined in advance. On the contrary, new shape models can be created by providing a few examples for which a corresponding semantic network is created. Recognizing

a shape is a matter of comparing the semantic network obtained from the input srokes with the network obtained from the training examples. Informal recognition rate of 95% (with certain manual corrections) is reported but formal recognition accuracy statistics are not provided. The domain-independent recognition engine is used in further work where information on the relationships of the symbols as well as domain information are used [55]. The additional information is utilized to interpret schematic sketches of physical devices.

Landay and Myers emphasize the importance of sketching in the early phases of design and try to reduce the gap between an early, sketched design of a user interface (UI) and a rough, working prototype of it [36]. They propose a system called SILK (Sketching Interfaces Like Krazy) that enables designers to quickly sketch UIs and convert them into working prototypes. SILK recognizes user sketches as UI elements such as text fields and scrollbars. The recognized primitive components must be drawn with a single stroke and they consist of rectangle, squiggly line (representing text), straight line and ellipse. These primitives can be combined and further recognized as UI components. In addition to the primitives, SILK recognizes editing gestures such as *delete* and *group* or *ungroup objects*. The underlying recognition engine of SILK is based on the work by Rubine [56]. His recognition system, GRANDMA (Gesture Recognizers Automated in Novel Direct Manipulation Architecture), is a single-stroke gesture recognition engine. The gestures are interpreted using a statistical, trainable recognizer for which new gestures can be constructed by providing a set of training samples. A set of geometric features are extracted from the sketched gesture and a linear classifier is used to select the correct gesture class. The single-stroke limitation was tackled in SILK by using a specific timeframe or spatial connectivity of the sketches. Recognition rates of 89% for the editing gestures, 93% for the primitives and 69% for the UI components were reported.

SketchREAD (Sketch Recognition Engine for mAny Domains) developed by Alvarado and Davis aims to address several drawbacks in earlier systems [37]. First, the recognition engine separates geometric information about shapes from their semantic interpretation. Hence, the engine can be used as the basis for recognition systems on multiple domains. Second, the recognition process uses both bottom-up and top-down approach. That is, low level interpretation is done first (bottom-up) but the context information can be used to correct possible low-level interpretation errors (top-down). Third, the approach imposes as few constraints as possible on the drawing style and the set of recognized symbols. Furthermore, the engine uses LADDER (A Language to Describe Drawing, Display, and Editing in Sketch Recognition), a formal language by Hammond and Davis [41], which is used to describe how sketched diagrams in a domain are drawn, displayed and edited. The usage of hierarchical representation such as LADDER enables others to extend the set of recognizable symbols and add domain-specific knowledge to the recognition system. The performance evaluation with sketches of family trees and circuit diagrams shows a substantial performance enhancement when compared to a baseline system that uses only bottom-up technique.

Another, more recent work that integrates with LADDER was suggested by

Paulson and Hammond [38]. Their PaleoSketch system is reported to utilize more geometric primitives than previous systems while still achieving a high (primitive) recognition rate (98.56%). The authors note the drawback of Rubine's feature-based classifier: it performs poorly on natural sketch data since the sketches are assumed to be drawn in the same way as the sample sketches. PaleoSketch places very few constraints on the drawing style, returns multiple interpretations (for the user to choose from) and takes a hierarchical, and thus extensible, approach. Two new features presented are reported to facilitate distinguishing polylines from curved strokes, a problem which has been difficult for other systems. *Normalized distance between direction extremes* (NDDE) is the proportion of the stroke between the highest and lowest direction values. *Direction change ratio* (DCR) is the maximum change in direction divided by the average change in direction. Polylines typically have low NDDE and high DCR as opposed to curved strokes with high NDDE and low DCR.

Sezgin and Davis base their work on a user study indicating that people are likely to use consistent stroke ordering when drawing certain shapes [39]. They argue that the otherwise exponential complexity of the recognition task can be reduced to polynomial time since the approach is interactive and incremental. Furthermore, the observation of stroke ordering allows them to use a technique based on the Hidden Markov Model (HMM), where the probability of a sketch belonging to a symbol class can be efficiently computed. Depending on the method used recognition accuracies from 81% to 97% are reported. Also, a promising performance statistics is reported compared to a simple, feature-based recognizer. However, it is noted that the technique cannot properly handle objects drawn using a single stroke. The method relies also extensively on the assumption made about the consistent drawing style.

2.9 Global geometric feature extraction

A somewhat different approach from the ones above is suggested in a series of papers by Fonseca and Jorge [57, 58, 59, 60]. They have valued simplicity of the method over robustness with their sketch recognition system. Their shape classification strategy is peculiar in that it uses only global geometric features of the input strokes together with fuzzy logic. That is, the strokes are not tried to be segmented into smaller and simpler subshapes which would then be recognized separately as geometric primitives. One or more strokes are collected within a timeout period. The strokes are considered to belong together and analyzed as a whole to obtain the recognized shape. The benefit of using only global geometric features is the invariance with rotation and scale of the shapes.

The recognition process is started by calculating the convex hull of the input points (deriving from one or more strokes). Then one calculates the maximum-area triangle and quadrilateral enclosed in the convex hull as well as the minimum-area rectangle enclosing the convex hull. The areas and perimeters for the three special polygons are calculated.

The essential idea of the classification is to combine the areas and perimeters of

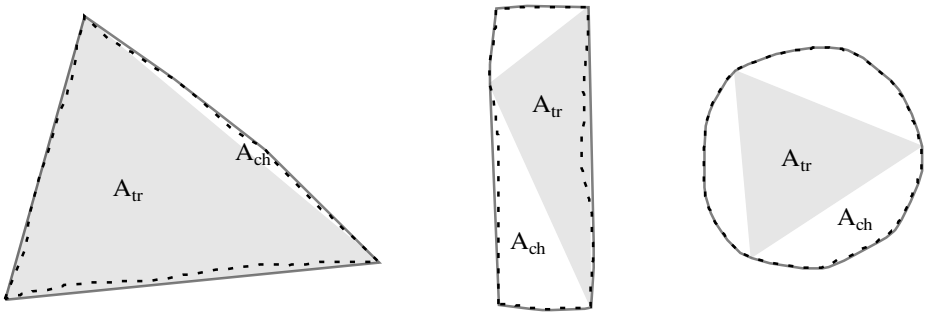


Figure 9: Freehand-drawn triangle, rectangle and circle (dashed lines) with their respective convex hulls (solid lines) and maximum-area triangle (light gray) enclosed in the convex hull. The ratio $R_{tr/ch} = A_{tr}/A_{ch}$ (area of the enclosed triangle divided by the area of the convex hull) is close to unity for triangles but substantially smaller for rectangles and circles: triangle $R_{tr/ch} = 0.95$, rectangle $R_{tr/ch} = 0.51$, circle $R_{tr/ch} = 0.44$.

the polygons to yield representative values for each shape. For example, the ratio of the areas of the convex hull and the enclosed triangle is intuitively very close to one for triangles but substantially less for other shapes such as rectangles or circles (see figure 9). The distinction between different shapes is obtained by analyzing the statistical distributions of a set of similar values for each shape class. More specifically, the distributions are used to construct fuzzy sets for each feature. The fuzzy sets give probabilities that given a set of strokes belongs to a certain shape class. This approach is used to address the inherent ambiguity in the input sketches (for example differentiating between circles and ellipses).

The manually constructed fuzzy sets are embedded in a decision tree that uses them as criteria to prune away unlikely shape classes. If more than one shape is possible for a given input the recognizer can return multiple suggestions for the user to choose from. Apart from basic geometric shapes (triangle, rectangle, circle, line, rhombus, ellipse) their system differentiates between solid and dashed lines and recognizes certain gestures bound to actions such as “delete” or “select”.

The approach based on the geometric features alone is intuitive and easy to understand. The drawback lies on the poor extensibility since only primitive shapes are recognized. The lack of hierarchical structure prevents the recognition of highly complex shapes.

Somewhat more complex shapes like crosses are recognized by adding simple heuristics on top of the geometric features: “number of strokes is 2 AND both strokes are lines AND strokes intersect”. However, the heuristics need to be manually constructed. A trainable version of the recognizer uses Naive Bayes classifier together with a feature vector of 24 geometric shapes [61]. Although the trainable recognizer does not require manual construction of the fuzzy sets and the decision tree the same primitive shapes are used.

The method described above has been named CALI (Calligraphic Interfaces). Others have taken CALI as the basis of their work in related pattern and sketch

recognition problems. Jota et al. describe a system for recognizing hand gestures with the help of a video camera [62]. Images of human hand in different poses are processed to first yield the silhouette and finally the contour of the hand. The contour is then used as an input for CALI to distinguish between gestures like *point*, *click* and *scroll*. Caetano et al. aim to create prototypes of user interfaces by recognizing UI components drawn freehand [63]. They construct heuristics on top of CALI to recognize sketched UI components like textfields, radio buttons and menus.

3 Preprocessing algorithms

Preprocessing the user-drawn sketches is the first part of the sketch recognition process and thus extremely important. The quality of the preprocessing procedures affects the performance of the system in every subsequent step of the system. This section describes in detail a set of algorithms for preprocessing the freehand strokes. An overview of the problem and an illustration of the input and output parameters of the algorithm are presented. A thorough explanation of the inner workings of each method is provided in the form of pseudocode implementation. Furthermore, the computational complexity of the technique, as well as alternative approaches to the problem, are discussed.

3.1 Douglas-Peucker

3.1.1 Overview

The Douglas-Peucker algorithm [64] is used to solve the *curve simplification* problem. According to Heckbert the problem is to take

> ... a polygonized curve with n vertices (a chain of line segments or "polyline") as input and produce an approximating polygonized curve with m vertices as output. A closely related problem is to take a curve with n vertices and approximate it within a specified error tolerance. [65, p. 4]

The problem has been studied by various researches and is of importance in the fields of digital cartography, GIS (georaphic information system) applications and CAD (computer-aided design) systems [66]. The heuristic approach called Douglas-Peucker calculates an approximation for a polyline using an error tolerance given as a parameter. The algorithm was developed independently by various researchers in the beginning of 1970's. In addition to Douglas and Peucker the algorithm was published by Ramer and is hence occasionally called Douglas-Peucker-Ramer [64, 67].

3.1.2 Input & output

The input for the algorithm is a list of n points $P = p_1, \ldots, p_n$ representing a polyline and an error tolerance ϵ. The output is a list of m points $Q = p_1, \ldots, p_n$. Usually $m < n$. However, depending on the nature of the input polyline and the error tolerance, m might be equal to n if the input polyline cannot be approximated without losing too much of its details. Due to the nature of the Douglas-Peucker algorithm every point in Q is also a point in P. In general, polyline simplification algorithms produce a point set that is not necessarily a subset of the input points. For example a variation of algorithm presented by Rosensaft produces output points that are obtained by averaging a pair of input points [68]. The input and output of the algorithm is depicted in figure 10.

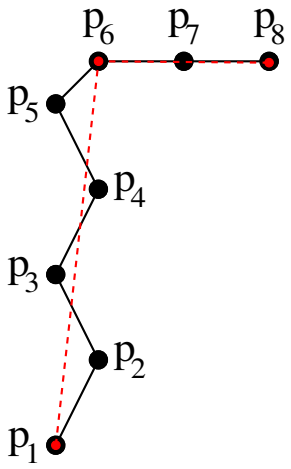
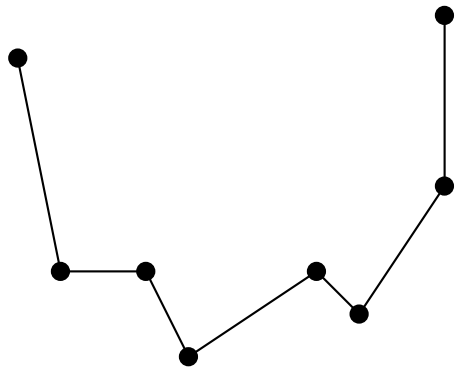


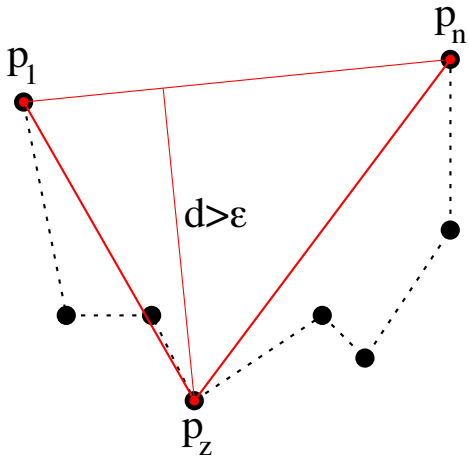
Figure 10: Input: p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8 Output: p_1,p_6,p_8

3.1.3 Algorithm overview

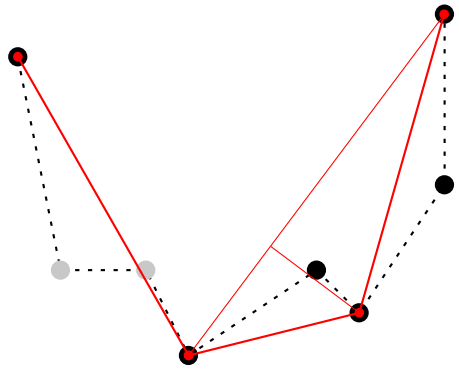
The Douglas-Peucker algorithm recursively splits the input polyline and handles the parts separately. The algorithm starts by constructing a line segment between the start and end point of the input (p_1 and p_n respectively). Then point p_z, namely the point between p_1 and p_n, furthest away from the line segment is searched by iterating through the points. If p_z is closer than ϵ (error measure) from the line segment, all the points between the start and end point can be discarded. Finding the furthest point from the line segment ensures that no point in between is further than ϵ away from the approximation. If p_z is further than ϵ it is included in the approximation. Algorithm recursively handles the polylines $p_1, \ldots, p_z$ and $p_z, \ldots, p_n$ (see figure 11 and algorithm 1).



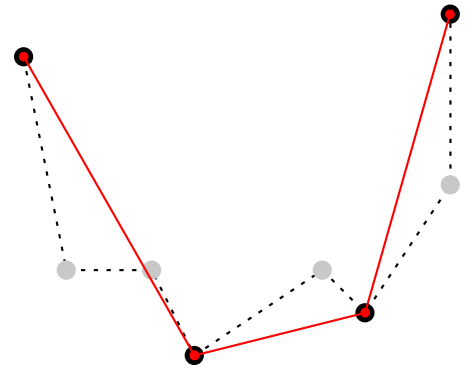
11.1



11.2



11.3



11.4

Figure 11: 11.1 The original polyline. 11.2 The point furthest away from the line segment between the end points is included in the polyline approximation. 11.3 Algorithm recursively handles the left and right side: on the left side the two points (in gray) are close enough to the approximating line segment and thus are discarded; on the right side the point furthest away from the line segment between the end points is selected. 11.4 The gray points are close enough to the approximating line segment and thus are discarded. The resulting polyline approximation is marked with red points and line segments.

3.1.4 Computational complexity

The high — though not optimal — quality of Douglas-Peucker comes with the cost of $O(n^2)$ upper bound. Consider the worst case when the approximated polyline is identical to the input polyline and the middle points are added to the approximation in their natural order $(p_2, p_3, \ldots, p_{n-1})$. First p_1 and p_n are the only points included in the approximation. It takes $n-2$ steps in the for-loop to find the furthest point from the polyline (lines 8–13). Recursive handling of $\{p_1, p_2\}$ takes constant time and handling of $\{p_2, \ldots, p_n\}$ takes $n-3$ iterations in the loop. In the subsequent, recursive invocations the number of iterations in the loop are $n-4, n-5, \ldots, 2, 1$. The consequent series gives the quadratic computational cost of the algorithm.

$$(n-2) + (n-3) + \cdots + 2 + 1 = \frac{n^2 - 3n + 2}{2}$$

The best case of $\Omega(n)$ is achieved when all the points lie on the same line segment between the start and end point. Expected time cost is approximately $\Theta(n \log n)$ [65].

3.1.5 Alternative approaches

A modified Douglas-Peucker algorithm has been proposed which has an upper bound of $O(n \log n)$ [69]. However, the algorithm is not general since it requires the input to be a simple polyline (polyline without self-intersections).

Approaches to the problem are always trade-offs between computational cost and the quality of the approximating polyline. Kolesnikov provides classification and thorough listing of different approaches [66]. Naive, optimal algorithms would have exponential cost but with certain optimizations can be executed in quadratic or cubic time. Optimum approaches are discussed and one with $\Theta(mn^3)$ running time (m denotes the number of output points) is proposed by Perez and Vidal [70]. Faster heuristic algorithms do not achieve optimality but often run in linear time.

Naive, yet not particularly elegant $O(n)$ solution would be to include every kth vertex in the output. Algorithms of better quality with decent running time are proposed, among others, by Leu et al [71]. Their algorithm of $\Theta(n)$ running time is based on iteratively examining the polyline and on each iteration simultaneously approximating arcs with chords with locally minimum deviation from the arcs. Boxer et. al [72] modified the algorithm to improve the quality of the approximating polyline. The modified version of the algorithm has a running time of $O(n + r^2)$ where r is the number of removed vertices. Algorithms with linear running time are proposed also in [73],[74],[75] and [76].

The optimization problem of approximating contour of n vertices with a subset of size m is an NP-hard problem [77]. Hence, various approaches to solve the problem in reasonable amount of time have been presented. These include dynamic programming, Newton's method, iterative point elimination, sequential methods, split-and-merge methods, dominant points or angle detection, k-means-based methods and evolutionary algorithms.

Algorithm 1 douglasPeucker(P, ϵ)

Input: points $P = p_1, \ldots, p_n$ (polyline), ϵ (error tolerance)
Output: approximation of input polyline

```
1: if n < 3 then
2:    return P // Return the input points as such (end of recursion)
3: end if
4:
5: Construct line segment l between p_1 and p_n
6: // Find the point furthest away from the line segment
7: maxDistance = −1, p_z = null
8: for i = 2 to n − 1 do
9:    if distance(l, p_i) > maxDistance then
10:      maxDistance = distance(l, p_i)
11:      p_z = p_i
12:   end if
13: end for
14:
15: if maxDistance ≤ ε then
16:   return p_1, p_n // Discard all other points but start and end point
17: else
18:   // Recursively handle the parts
19:   left = douglasPeucker({p_1, ..., p_z}, ε)
20:   right = douglasPeucker({p_z, ..., p_n}, ε)
21:   Merge left and right into result and remove the duplicate p_z
22:   return result
23: end if
```

3.2 End point refinement filter

3.2.1 Overview

Whether the user stroke is input with a mouse or a pen-based interface the start and end of the stroke might contain hooklet-like segments. These distortions can be the result of hardware noise or careless drawing by the user. Especially with pen-based systems the touching of the tablet and lifting the pen-tip off the tablet can produce hooklets [33, 78]. Although the hooklets present only a minor deviation from the intended stroke they impose difficulties in the later steps of the preprocessing and shape classification. A major implication of a hooklet in a stroke is that it might change dramatically the direction to which the end of the stroke is pointing (see figure 12). The direction is essential for example for algorithms that try to deduce whether the stroke is intented to present a closed shape or not (see section 3.3).

3.2.2 Input & output

The input for the algorithm is a list of n points $P = p_1, \ldots, p_n$ representing a polyline, bin size B and maximum number of points in a bin M. The output is a list of m points $Q = A_{start}, \ldots, p_j, p_{j+1}, \ldots, A_{end}$. The stroke represented by points P might have a hooklet in one or both ends. In the output Q any hooklet is removed. If there are no hooklets in the stroke the input and output might be identical in which case $A_{start} = p_1$ and $A_{end} = p_n$. However, if a hooklet is removed it holds that $m < n$. Furthermore, the set of output points is not necessarily a subset of the input points. The input and output of the algorithm is depicted in figure 13.

3.2.3 Algorithm overview

The algorithm works symmetrically for the start and end of the stroke. Hence, only the start of the stroke is considered here. The hooklet or some other form of unintended scribble is removed by averaging the k first points of the stroke. The value of k is dynamically constrained by two criteria: size of the averaging bin and

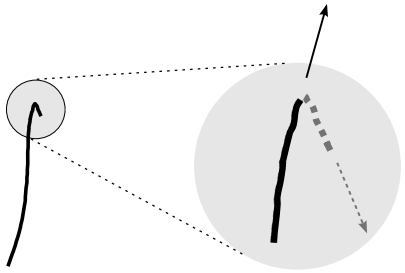


Figure 12: A hooklet can dramatically distort the end direction of a stroke. The arrows illustrate the intended (solid arrow) and distorted (dashed arrow) end directions.

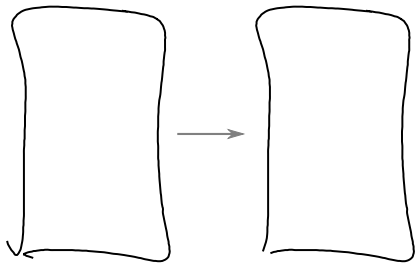


Figure 13: The input (left) and output (right) stroke of the end point refinement filter.

maximum number of points in the bin (parameters B and M for the algorithm respectively).

A square-shaped bin (with the side length of B) functions as a window that contains the points that are to be averaged. The start of the stroke is examined point at a time while adding each point to the bin. In case a point would not fit inside the bin or the maximum number of points in the bin is reached, the average is calculated for the points already in the bin. The x- and y-coordinates of the point are rounded to the closest integers and the resulting point is denoted by A_{start}. In the final step the points that were added to the bin are replaced by their average, A_{start}. The execution of the algorithm is illustrated in figure 14 and algorithm 2.

3.2.4 Computational complexity

The computation time of the algorithm is dominated by the for-loop in lines 1–8 since other operations can be done in constant time. The operations within the for-loop are executed at most M times. In some implementations M could be chosen to be a certain percentage of the total number of points in the input polyline. In this case the computational complexity would be $O(n)$. However, usually M is chosen to be fixed and hence the algorithm can be executed in constant time.

3.2.5 Alternative approaches

Jin et al. use a measure of point density in their agglomerate points filtering [33]. Their approach is computationally slightly more expensive ($O(n^2)$) but removes also circlet-like distortions around the turning points of the stroke.

Huang et al. propose a set of preprocessing techniques for online handwriting recognition [79]. They describe a three-step process for eliminating hooks not only in the ends of the stroke. First, the stroke is interpolated in order to make the stroke points evenly distributed along the length of the stroke. Second, sharp points are detected by analyzing the slopes of consecutive points in the stroke. Finally, the hooks are detected by using certain changed-angle and length thresholds.

PaleoSketch developed by Paulson and Hammond analyzes the first and last 20%

of the stroke before the recognition process [38]. Within the 20% of the stroke they search for the highest curvature value. If the value is above a threshold, the stroke is broken at that point and the tail is removed. The removal is not performed if the stroke has a low number of points. This criterion is intuitively reasonable for any end point refinement algorithm.

Algorithm 2 refineEndPoints(P, B, M)

Input: points $P_n = p_1, \ldots, p_n$ (polyline), averaging bin size B, maximum number of points in the bin M

Output: polyline with hooklets and scribbles removed from the ends

1: **for** $i = 1$ to M **do**
2: **if** p_i can be added to the bin without exceeding the bin size B **then**
3: Add p_i to the bin
4: **else**
5: // Break out of the loop (to line 10)
6: **break**
7: **end if**
8: **end for**
9:
10: $A_{start} = average(p_1, p_2, \ldots, p_k)$ // p_k is the last point added to the bin
11: $A_{start} = closestDiscretePoint(A_{start})$
12: Replace $p_1, p_2, \ldots, p_k$ with A in the output
13:
14: Repeat the process to the end of the polyline $(p_n, p_{n-1}, \ldots)$ to obtain A_{end}
15: **return** polyline $Q = A_{start}, p_{k+1}, p_{k+2}, \ldots, A_{end}$

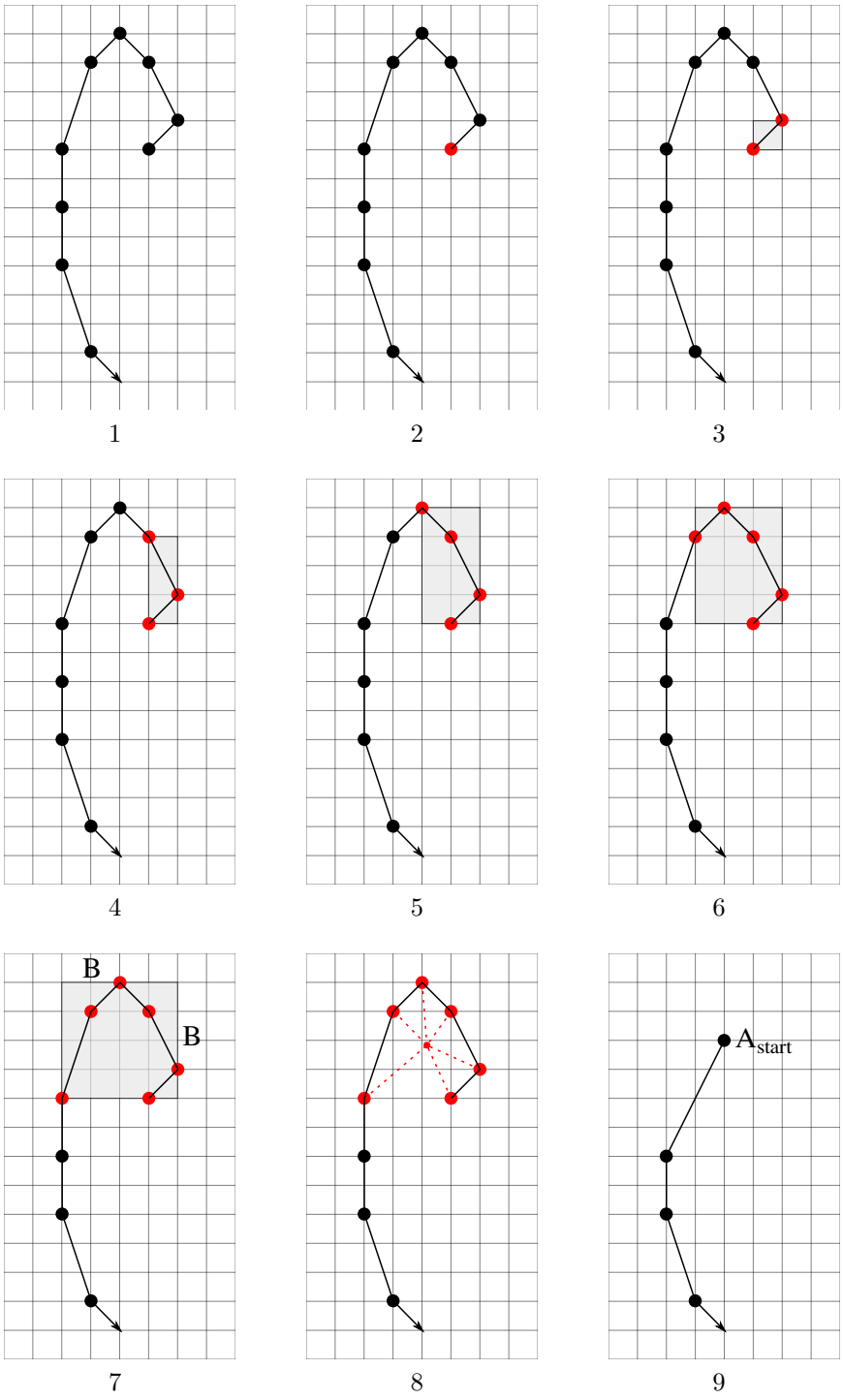


Figure 14: The execution of the end point refinement algorithm applied to the start of the stroke (parameters $B = 4, M = 10$). 1. Start of the original stroke (arrow indicating the drawing direction). 2 – 7. Points are added to the averaging bin until its maximum size ($B \times B$) or maximum number of points (M) is reached. 8. Points in the bin are averaged. 9. The point with discrete (integer) coordinates, closest to the average (A_{start}) is used to replace the averaged points. Consequently, the hooklet in the original stroke is removed.

3.3 Stroke closing filter

3.3.1 Overview

Closed shapes (polygons, circles, ellipses) are common in many sketching application domains such as UML (unified modeling language). When drawing closed shapes the user-drawn stroke is usually not precisely closed. Either the end of the stroke does not reach the start, or the head and tail of the stroke form a cross. Since many recognized shapes are closed (circle, polygons) refining these imperfections facilitate the subsequent classification steps. Jin et al. have presented a simple preprocessing method for solving the problem [33].

3.3.2 Input & output

The input for the algorithm is a list of n points $P = p_1, \ldots, p_n$ representing a polyline (single stroke). The output is a list of m points Q. The stroke represented by points P might be improperly closed. In the output Q the stroke is perfectly closed (the start and end point of the stroke coincide). Due to the nature of the algorithm the set of output points is not necessarily a subset of the input points. The input and output of the algorithm is depicted in figure 15.

3.3.3 Algorithm overview

The algorithm is designed to handle two kinds of imperfections: an un-closed shape (figure 15a) and a closed shape with a cross with redundant tails (figure 15b). In the case of an unclosed shape the head and tail of the stroke are extended along their end directions. If the extensions intersect the intersection point is used as the start and end point of the stroke thus effectively closing the stroke (figure 16a). The algorithm must also handle the case where one of the extensions intersects not with the other intersection but with the tail of the original stroke (figure 16c). When there is a cross formed by the head and tail of the stroke, the redundant tails are removed similarly. The intersection point is retained but the rest of the head and tail are spliced out (figure 16b). The algorithm presented in section 4.4 is used to find the self-intersections in the stroke. The details of the stroke closing filter are presented in algorithms 3, 4 and 5.

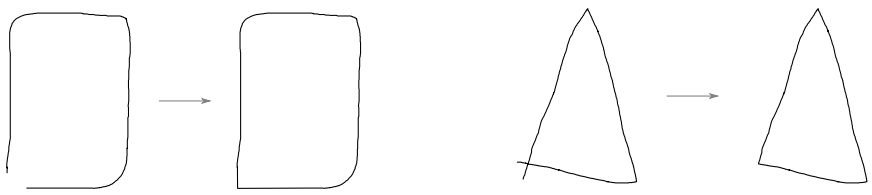


Figure 15: Examples of input and output of stroke closing filter. The improperly closed stroke ends are corrected in the output.

For simplicity a set of experimentally configured constants are omitted from the algorithm description. The *extension length* (EL) is used when extending the head and tail of the stroke. Using too small a value might prevent the method from properly closing the stroke. However, too big a value could lead to unwanted behaviour where a stroke intended to be un-closed is mistakenly closed. The *maximum cutted path length* (MCPL) defines the maximum length than can be spliced out from the stroke. The value must be adequately small as otherwise the algorithm might mistakenly consider a self-intersection in a stroke as a closing point. The *maximum distance from endpoint* (MDE) is checked in the case of a cross. An intersection is not considered as a closing point if it is further away than MDE from either the start or end point of the stroke.

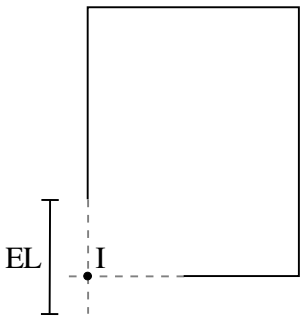
3.3.4 Computational complexity

The computational complexity of calculating the intersections in line 1 in algorithm 3 is $O(n \log n + k \log n)$ where k is the number of intersections in polyline P (see section 4.4.4). The computationally most complex part of the algorithm 4 is the removal of redundant points in line 7. Other computations can be done in constant time. Thus, the running time of the subprocedure is $O(n)$ since the removal of points is ultimately bound by the number of points in the polyline. Algorithm 5 also has to iterate over the intersection points in line 1. The running time of $O(n)$ applies both to lines 4–6 and to line 9. Thus, the running time of the subprocedure is $O(n+k)$. By combining the results we can see that the running time of the algorithm is dominated by the calculation of intersections. Hence, the overall computational complexity of the algorithm is the one of intersection calculations, $O(n \log n + k \log n)$.

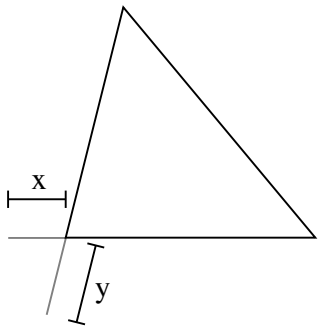
In practise the algorithm is computationally relatively cheap. First, usually the number of intersections is very small and thus the time requirement for calculating the intersections is close to $O(n \log n)$. Second, usually the number of points to remove from the polyline is only a fraction of n due to the various constraints discussed above.

3.3.5 Alternative approaches

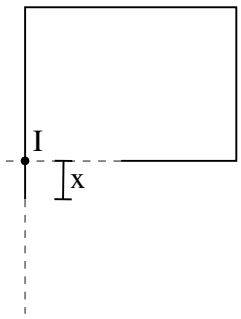
The technique presented above is very intuitive since it closely resembles the steps that a human would take manually to solve the problem. There is very little literature discussing the problem. A naively simple approach would be merely to check the distance of the first and last point in the polyline. If the distance is under certain threshold the points are connected with a line segment. However, this would work only for an un-closed shape and would not consider the human intention aspect embedded in the end directions of the head and tail of the polyline. Furthermore, in the case of a cross, the preprocessed polyline might be even messier than the original.



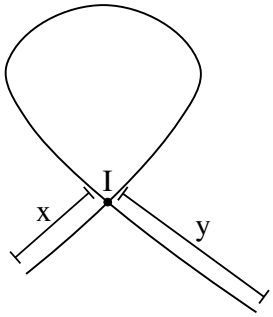
(a) Closing of the stroke by extending the ends (dashed lines). *EL* extension length, *I* resulting intersection point.



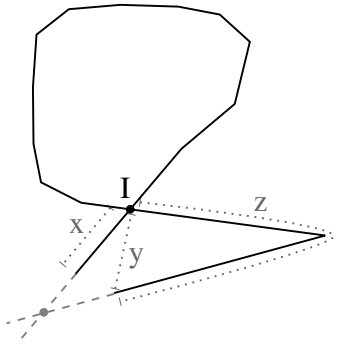
(b) Closing of the stroke by removing the ends (gray lines). The distances x and y must be below the thresholds *MCPL* and *MDE*.



(c) Closing of the stroke when the end extension intersects with the stroke tail. The length of the cut-out part (x) must be below *MCPL*.



(d) The self-intersection is retained because x and y are larger than the thresholds.



(e) The stroke ends are near the self-intersection point I: $x < MDE$ and $y < MDE$. However, the cutted path length z would be larger than the threshold *MCPL*. Thus, the algorithm continues by searching for the intended closing point by extending the ends of the stroke (dashed lines).

Figure 16: Various scenarios faced by the stroke closing algorithm.

Algorithm 3 closeStroke(P)

Input: points $P_n = p_1, \ldots, p_n$ (polyline)
Output: polyline properly closed (if necessary)

$L = intersections(P)$ // Calculate list of self-intersections in P.
if $L = \emptyset$ **then**
  return $handleNoIntersections(P)$
else
  return $handleIntersections(P, L)$
end if

Algorithm 4 handleNoIntersections(P)

Input: points $P_n = p_1, \ldots, p_n$ (polyline)
Output: polyline properly closed (if necessary)

Construct line segments $head_{ext}$ and $tail_{ext}$ extending the head and tail of P
if $head_{ext}$ and $tail_{ext}$ intersect in point I **then**
  // See figure 16a
  Add I as the first and last point in P
else if Either $head_{ext}$ or $tail_{ext}$ intersect with a line segment in P in point I **and** I satisfies MCPL and MDE constraints **then**
  // See figure 16c
  Remove points in P that come after I
  Add I as the first and last point in P
end if
return P

Algorithm 5 handleIntersections(P, L)

Input: points $P_n = p_1, \ldots, p_n$ (polyline), list of intersections $L = I_1, \ldots, I_k$
Output: polyline properly closed (if necessary)

Find the intersection I closest to p_1 and p_n
if I satisfies MCPL and MDE constraints **then**
  // See figure 16b
  Remove points in P that come after I
  Add I as the first and last point in P
  return P
else
  // See figure 16e
  return $handleNoIntersections(P)$
end if

4 Feature extraction algorithms

Although the raw points of the user-drawn sketch define the input accurately and completely, they are as such of little use to the shape recognition process. To differentiate between the shape classes one must select a set of representative features and extract those features from raw data. This section lists a collection of algorithms for computing certain geometric features of the input sketch. As with preprocessing algorithms the subsections discuss each algorithm in detail and provide an overview of the problem and proposed approach, the input and output of the algorithm, as well as a pseudocode implementation. The computational complexity of the algorithm and alternative approaches in the literature are also discussed.

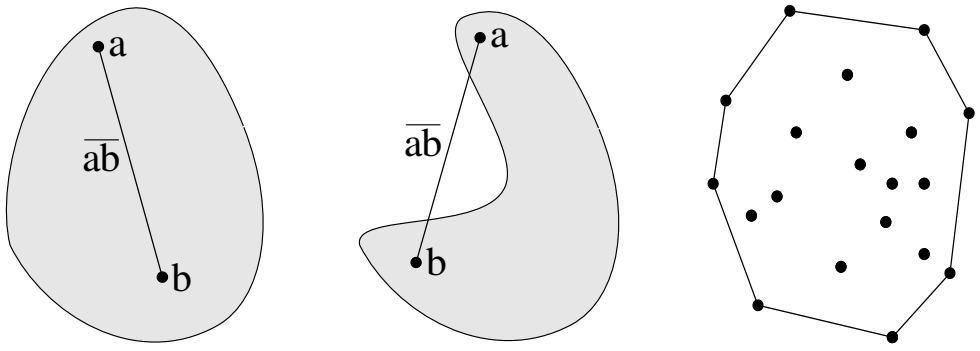
4.1 Convex hull

4.1.1 Overview

A subset S of the plane is called *convex* if and only if for any pair of points $a, b \in S$ the line segment $\overline{ab}$ is completely contained in S [80, 81]. The *convex hull* $CH(S)$ of a set S is the smallest convex set that contains S. A special case concerns only a finite set of points P where the convex hull $CH(P)$ can be defined as the unique convex polygon whose vertices are points in P and that contains all points in P. See figure 17.

4.1.2 Input & output

The convex hull algorithm is supposed to solve the problem of finding the convex hull $CH(P)$ for a finite set of points P. Thus, the input is n points in $P = p_1, p_2, \ldots, p_n$ in arbitrary order. The output is a subset of P: the vertices of the unique, convex polygon that is the convex hull. To specify the output more accurately we require that the output points are listed in clockwise order starting with the leftmost point. See figure 18.



(a) convex (b) non-convex (c) convex polygon

Figure 17: Convex and non-convex planar sets and convex hull (polygon) for a finite set of points.

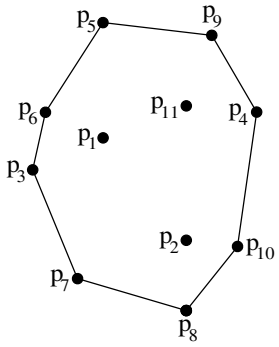


Figure 18: Input: p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9,p_{10},p_{11} Output: p_3,p_6,p_5,p_9,p_4,p_{10},p_8,p_7

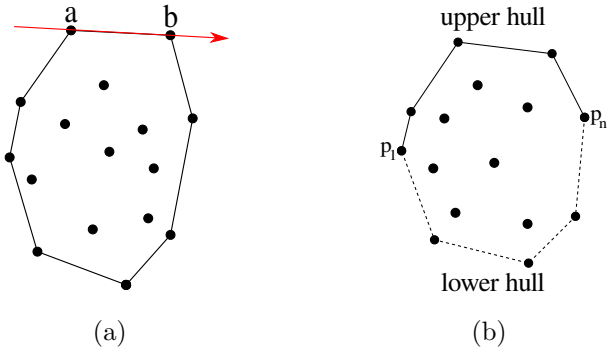


Figure 19: Convex hull definitions

4.1.3 Algorithm overview

The idea of the algorithm relies on the observation that if $\vec{ab}$ is an edge of the convex hull all the points in P lie to the right of a line that coincides with the edge (illustrated in figure 19a). First, the input points are sorted by ascending x-coordinate (ascending y-coordinate if x-coordinates are the same) giving a sequence $p_1, \ldots, p_n$. Thus, the convex hull can be divided into *upper hull* and *lower hull* that both have p_1 and p_n as their first and last points (figure 19b). Determining the convex hull is done by constructing the upper and lower hull separately.

The computation of the upper hull can be thought of as walking around the boundary of the convex hull from p_1 to p_n. We observe that when moving to the next point we are always making a right turn. Otherwise the previous point would not be a part of the upper hull. The algorithm incrementally adds points to and updates the upper hull. The updating step checks whether the last three points make a right turn, in which case a new point is added. However, if the last three points make a left turn, the second last point has to be removed from the upper hull and the check repeated. Finally the rightmost input point is added to the upper hull. The lower hull is computed similarly but this time proceeding from p_n to p_1. The upper and lower hull are combined and since p_1 and p_n are included in both lists, the duplicates are removed. See algorithm 6 and figure 20 for the details.

4.1.4 Computational complexity

The sorting of points in line 1 can be done in $O(n \log n)$ time (for example with heapsort [81]). The for-loop is executed $n - 2$ times. Within the loop, in line 5, a point is deleted from L_{upper}. For each point this can happen once at most. Thus, the number of executions of line 5 is bounded by n and the for-loop takes $O(n)$ time. The computation of the lower hull also takes $O(n)$ time. Therefore the overall time requirement for the algorithm is determined by the time requirement of the initial sorting of the points $O(n \log n)$.

4.1.5 Alternative approaches

Computing the convex hull for a set of points is a classic topic in computational geometry and is discussed extensively in the literature. The algorithm above is based on an algorithm presented by Graham, commonly known as *Graham's scan* [81, 82]. The approach above, sorting the points on x-coordinate and computing upper and lower hull separately, is a modification presented by Andrew [83]. The computational complexity of the algorithm presented above is in fact also the lower bound ($\Omega(n \log n)$) for all algorithms [84]. However, algorithms of improved upper bound performance have been introduced by many authors. The method presented by Jarvis, known as *Jarvis's march*, compares the polar angles of the points to incrementally achieve the solution [81, 85]. The algorithm achieves an output-sensitive computational complexity of $O(hn)$ where n is the number of input points and h is the number of vertices in the resulting convex hull. Eddy achieves the same worst-case performance with divide-and-conquer type of approach [86]. Finally, Kirkpatrick and Seidel have presented an $O(n \log h)$ algorithm to the problem [87]. The method is based on dividing the set of points with a vertical line to the right and left part and handling the parts recursively.

Algorithm 6 convexHull(P)

Input: points P
Output: convex hull $CH(P)$ as a list of polygon vertices in clockwise order

```
1: Sort the points by ascending x- and y-coordinate resulting in P = p_1, ..., p_n
2: Initialize list L_upper = (p_1, p_2)
3: for i = 3 ... n do
4:     while L_upper contains more than two points and the last three points in L_upper
       do not make a right turn do
5:         Delete the middle one of the last three points from L_upper
6:     end while
7: end for
8:
9: Repeat for lower hull resulting in list L_lower
10: Remove the first and last point from L_lower
11: Append L_lower to L_upper → L
12: return L
```

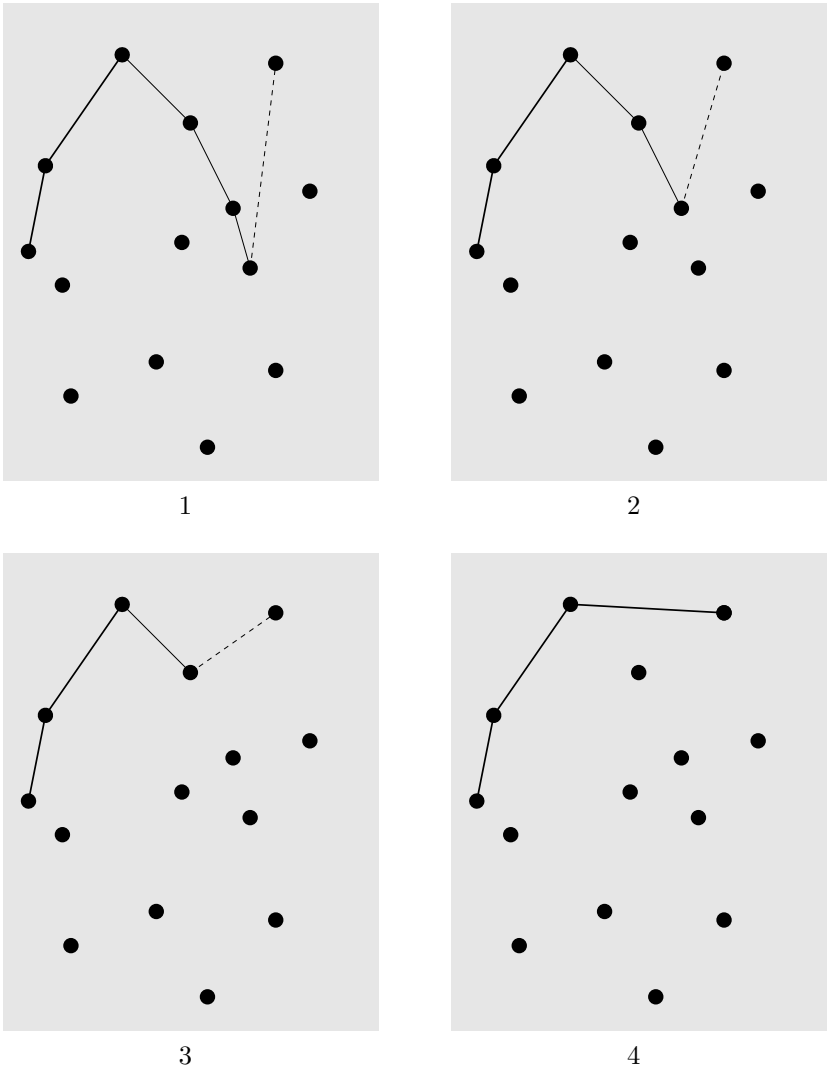


Figure 20: The execution of the convex hull algorithm. In 1, 2 and 3 the last three points do not form a right-turn. Hence, the middle point is removed from the list. In 4 a right-turn is formed and the algorithm moves to the next candidate point.

4.2 Minimum-area enclosing rectangle

4.2.1 Overview

Minimum-area enclosing rectangle for a set of points P is the smallest, unique rectangle that contains all the points in the set. The difficulty of the problem lies in the fact that in a general case, the sides of the resulting rectangle are not parallel with the coordinate axes. The problem can be simplified by requiring that the set of points P forms a convex polygon. For an arbitrary set of points this can be achieved by computing its convex hull using the algorithm described in section 4.1. In addition to theoretical interest the solution to the problem has applications in certain packing and layout problems, as well as in determining a suitable package size in goods transport [88, 89].

4.2.2 Input & output

The input for the algorithm is a convex polygon represented as a list of points $P = p_1, p_2, \ldots, p_n$ sorted in clockwise order. The output is a rectangle of minimum area enclosing the input points. The rectangle is represented as a list of points $R = r_1, r_2, r_3, r_4$ sorted in clockwise order. See figure 21.

4.2.3 Algorithm overview

The algorithm to solve the problem is based on the theorem 4.1 proved by Freeman and Shapira.

Theorem 4.1 *The rectangle of minimum area enclosing a convex polygon has a side collinear with one of the edges of the polygon. [88, p. 411]*

By using the theorem 4.1 a naive approach would be to iterate over each edge of the polygon, constructing the corresponding rectangle and computing its area. This

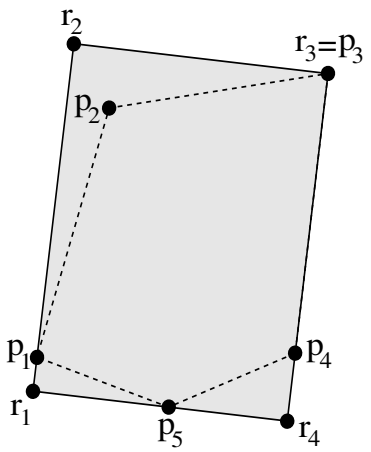
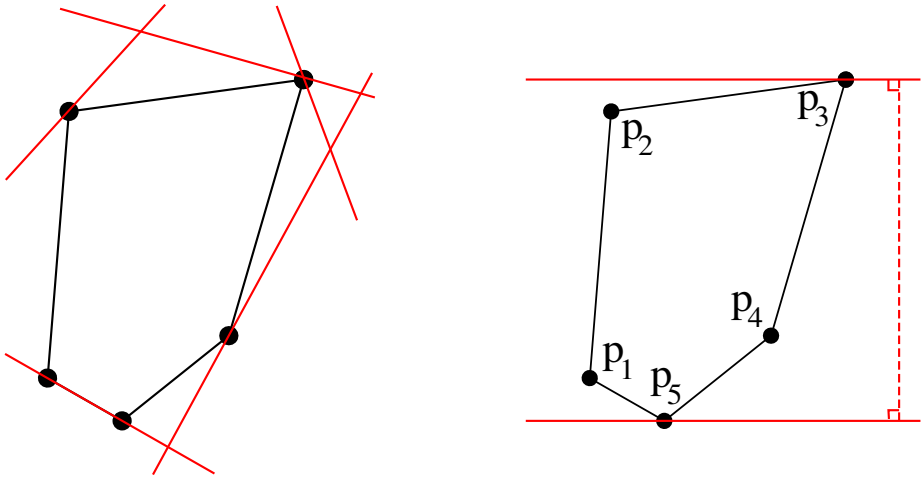


Figure 21: Input: polygon p_1,p_2,p_3,p_4,p_5 Output: rectangle r_1,r_2,r_3,r_4



(a) Some lines of support for a polygon

(b) p_3 and p_5 are an anti-podal pair. For example p_1 and p_4 are not since they do not admit parallel lines of support.

Figure 22: Lines of support and anti-podal pairs.

is in fact the approach adopted in [88] and results in running time of $O(n^2)$. It is obvious that the lower bound for the problem is $\Omega(n)$ since the algorithm needs to examine each point at least once. Based on the theorem 4.1 Toussaint achieved also the same upper bound in [90]. Toussaint's algorithm utilizes the method of *rotating calipers* that was first presented by Shamos in [91]. Pirzadeh gathers the results above and presents Toussaint's algorithm for computing the minimum-area enclosing rectangle in [92].

The idea of rotating calipers is based on *lines of support* and *anti-podal pairs* (see figure 22).

Definition 4.1 *A line L is a line of support for a convex polygon P if it intersects P and the interior of P lies on one side of L. [92, p. 10]*

If L intersects P at a vertex v (or an edge e), v (or e) is said to admit L.

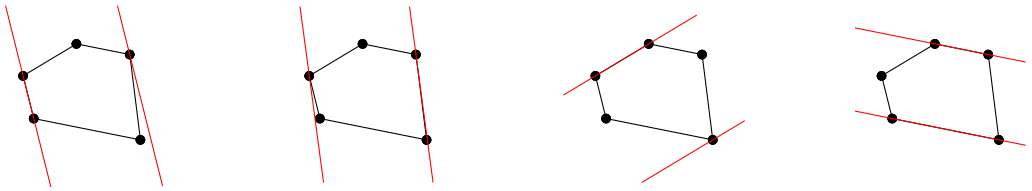
Definition 4.2 *Given a convex polygon P, a pair of vertices $p, q \in P$ is called an anti-podal pair if p and q admit parallel lines of support. [92, p. 10]*

Shamos used two parallel lines of support to iterate over all the anti-podal pairs of a convex polygon in order to calculate the diameter of the polygon (the distance between the vertices that are farthest apart). The iteration is done by "rotating" the lines of support (see figure 23) around the polygon. Hence the term "rotating calipers". Toussaint added another pair of parallel lines of support to efficiently iterate over all the edges of the convex polygon given as an input. See algorithm 7 and figure 24 for details.

Algorithm 7 minimumAreaEnclosingRectangle(P)

Input: points P (convex polygon with n vertices)
Output: minimum-area enclosing rectangle $ER(P)$

Find vertices with minimum and maximum x and y-coordinates (p^x_{min}, p^x_{max}, p^y_{min} and p^y_{max})
Initialize two sets of calipers (parallel lines of support) coinciding p^x_{min}, p^x_{max}, p^y_{min} and p^y_{max} and parallel to the coordinate axes. The calipers form a rectangle enclosing P and angles $\theta_i, \theta_j, \theta_k$ and θ_l (see figure 24.1).
while calipers have been rotated in total less than $\pi/2$ (90°) **do**
    $\theta = \min(\theta_i, \theta_j, \theta_k, \theta_l)$
    Rotate calipers by θ, thus making the rectangle coincide with another edge of the polygon
    Compute A, the area of the rectangle
    if $A <$ the current minimum (area of the rectangles seen so far) **then**
        Store A as the current minimum
    end if
    Compute $\theta_i, \theta_j, \theta_k$ and θ_l
end while
return the rectangle that had the smallest area



23.1 23.2 23.3 23.4

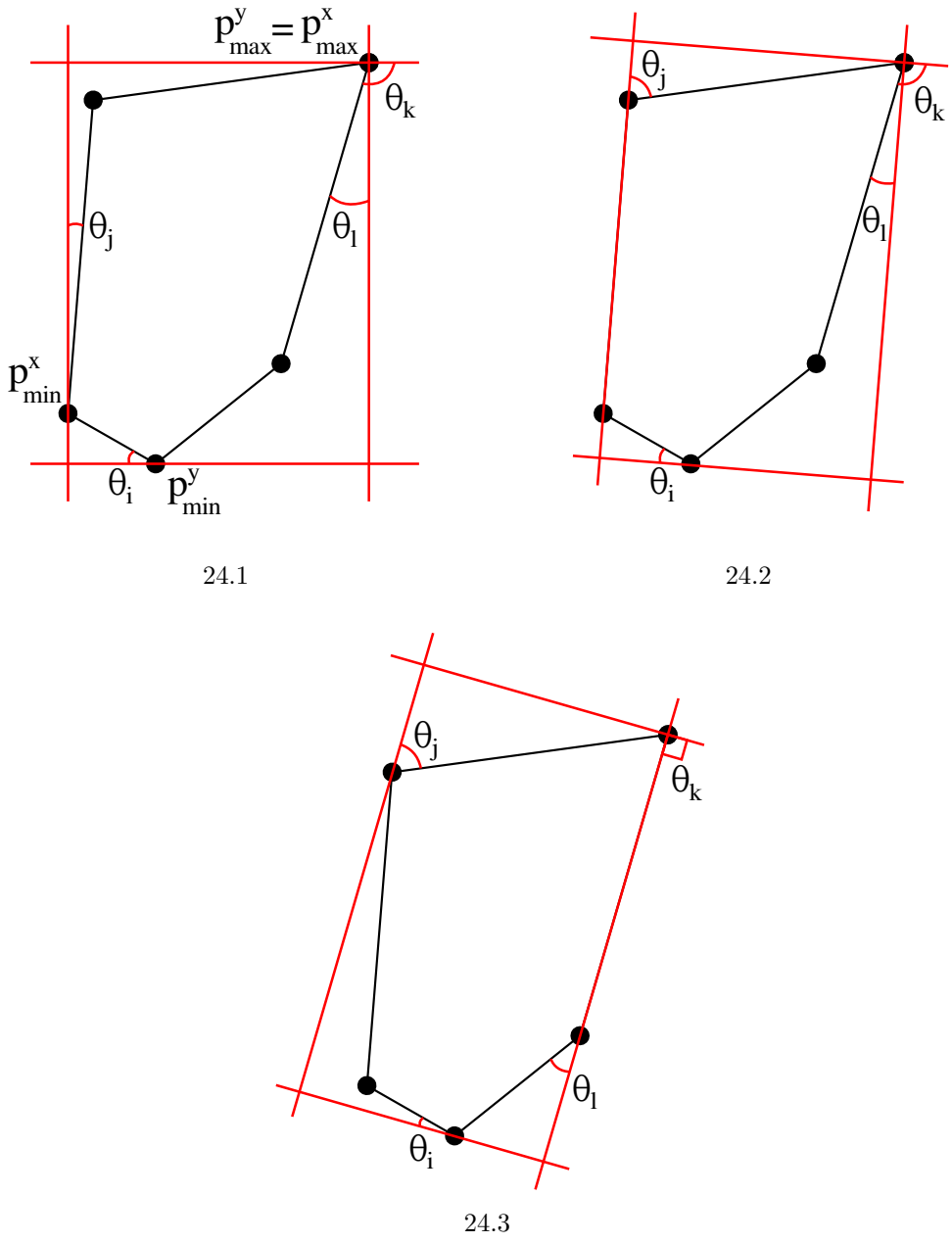
Figure 23: The idea of *rotating calipers*: edges of the polygon are iterated by rotating parallel lines of support around a polygon.

4.2.4 Computational complexity

The operations in lines 1, 2 and 4–10 can be executed in constant time. The while loop is executed n times. This is evident since every rotation of the calipers results in one new edge of the convex polygon to coincide with one of the calipers. Once the calipers have rotated an angle of over $\pi/2$, all edges of the polygon have been examined. Hence, the computational complexity of the algorithm is $O(n)$ where n is the number of vertices of the input polygon.

4.2.5 Alternative approaches

Essentially the same algorithm inspired by the work of Shamos was described also by Arnon and Gieselmann [93]. The problem can be generalized into three dimensions, the solution for which resembles the principles of rotating calipers in 2D [94]. Perhaps surprisingly the problem of the minimum-perimeter enclosing rectangle is not equivalent with the minimum-area case. The two problems can be solved in a similar manner but in rare cases the solutions do not coincide [95].



24.1 24.2

24.3

Figure 24: Computing the minimum-area enclosing rectangle for a convex polygon using rotating calipers. The algorithm starts with the rotating calipers parallel to the coordinate axes. In figure 24.1 $\theta_j = \min(\theta_i, \theta_j, \theta_k, \theta_l)$ and thus the calipers are rotated by θ_j. In figure 24.2 the minimum angle is θ_l.

4.3 Maximum-area enclosed triangle

4.3.1 Overview

The maximum-area enclosed triangle for a convex polygon is the largest triangle, the vertices of which lie inside the polygon. The triangle is not unique in a general case because of symmetry. This is obvious for example in the case of regular polygons with more than three vertices (see figure 25a). The problem falls into the category of finding inscribing or circumscribing polygons that maximize a certain measurement such as area or perimeter.

4.3.2 Input & output

The input for the algorithm is a convex polygon represented as a list of vertices $P = p_1, p_2, \ldots, p_n$ where $n \geq 3$ sorted in clockwise order. The output is a triangle of maximum area that is fully contained by the polygon. The triangle is represented as a list of three points $T = A, B, C$ sorted in clockwise order. See figure 25b.

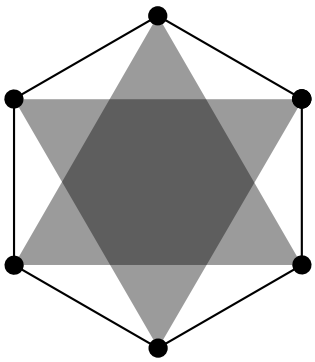
4.3.3 Algorithm overview

The algorithm described here was presented by Dobkin and Snyder [96]. They try to solve a family of geometric optimization problems that are of form:

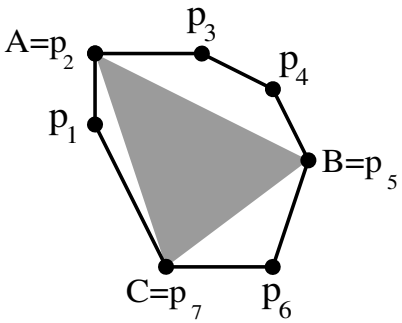
> If P is an n sided convex polygon, what is the largest $k$gon having specified properties which may be embedded in P. [96, p. 9]

The algorithm is based on the theorem 4.2 that the authors prove by induction.

Theorem 4.2 *Given an n-gon $P_n = p_1, \ldots, p_n$, and an integer $k > 1$, there is an area maximizing inscribed k-gon, $P_k = p'_1, \ldots, p'_k$ such that $\{p'_1, \ldots, p'_k\} \subseteq \{p_1, \ldots, p_n\}$. [96, p. 10]*



(a) In some cases there are many triangles with the same maximum area.



(b) Input: polygon p_1,p_2,p_3,p_4,p_5, p_6,p_7 Output: triangle A, B, C

Figure 25

That is, the vertices of the maximum area triangle are also vertices of the convex polygon which implies that there is an algorithm that can solve the problem in finite time.

Iterating through all combinations of three vertices would naturally yield the solution but the computational complexity of the approach would be cubic. However, Dobkin and Snyder present a proof that the problem can be solved faster by using three points which define an inscribed triangle and are moved along the vertices of the polygon. In brief, the algorithm starts by setting the three points to three consecutive vertices of the polygon. The first point is moved along the vertices of the polygon as long as the area of the triangle does not decrease. Next the procedure is repeated to the second point moving it as long as the area does not decrease. Finally, the third point is moved to the next vertex of the polygon and the algorithm continues moving the first and second point. The vertices of the polygon are iterated in this manner and the maximum-area triangle is kept in memory. The method is described in more detail in algorithm 8 and illustrated in figure 26.

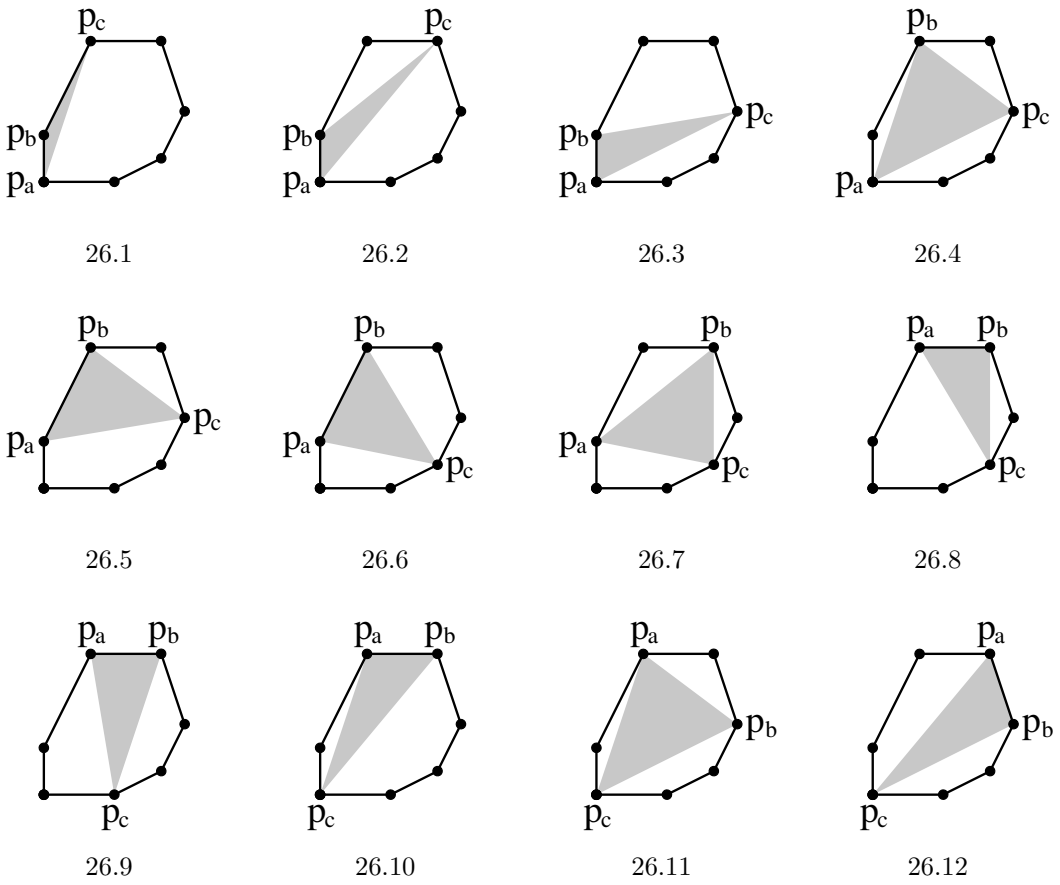


Figure 26: First steps in computing the maximum-area enclosed triangle for a convex polygon.

4.3.4 Computational complexity

In algorithm 8, function *area* denotes the calculation of the area of a triangle. The calculation can be done in constant time. The computational complexity of the algorithm can be determined by considering how many times *area* is executed. The reasoning is based on the following observations: *area* is executed at most six times before some of the vertices (p_a, p_b, p_c) advances in line 7, 10 or 19. Point p_a visits exactly n vertices. The lines 6, 9 and 20–21 ensure that p_a, p_b and p_c maintain their order. Consequently, neither p_b nor p_c can overtake p_a and therefore they visit fewer than $2n$ vertices. This reasoning yields the total number of executions of *area*: $2n(6 + 6) + 6n = 30n$. The average execution is likely to require fewer executions but the worst case scenario gives the linear upper bound $O(n)$.

4.3.5 Alternative approaches

The upper bound of the algorithm is clearly also the lower bound since an optimal algorithm has to go through all the vertices of the polygon in order to find the maximum-area triangle. In addition to the triangle algorithm Dobkin and Snyder provide a k-gon version of their procedure. Both algorithms rely on the fact that the input polygon is convex. Boyce et al. continue by finding maximum perimeter and area convex k-gons with vertices k of the given n points [97]. Specially, it is noted that the n points are totally arbitrary. The authors prove that the vertices of the k-gon are points on the convex hull of the set of n points and provide an algorithm with $O(kn \log n + n \log^2 n)$ running time and linear memory requirement.

Algorithm 8 maximumAreaEnclosedTriangle(P)

Input: points $P_n = p_0, \ldots, p_{n-1}, n \geq 3$ (convex polygon)
Output: vertices A, B, C of the maximum-area enclosed triangle

```
1:  // All additions are performed modulo n
2:  A = p_0, B = p_1, C = p_2
3:  a = 0, b = 1, c = 2
4:  repeat
5:    while true do
6:      while area(p_a, p_b, p_{c+1}) ≥ area(p_a, p_b, p_c) do
7:        c = c + 1
8:      end while
9:      if area(p_a, p_{b+1}, p_c) ≥ area(p_a, p_b, p_c) then
10:       b = b + 1
11:     else
12:       // Break out of the loop (to line 16)
13:       break
14:     end if
15:   end while
16:   if area(p_a, p_b, p_c) > area(A, B, C) then
17:     A = p_a, B = p_b, C = p_c
18:   end if
19:   a = a + 1
20:   b = (a == b) ? b + 1 : b
21:   c = (b == c) ? c + 1 : c
22: until a = 0 // We have iterated through all vertices of the polygon
23: return A, B, C
```

4.4 Intersections

4.4.1 Overview

Few sketch recognition approaches employ the detection of intersection points of the strokes. This is due to the fact that a vast majority of the recognized shapes contain no intersections at all (excluding the closing point of closed shapes). Furthermore, the shapes that do, can usually be recognized by other means and thus the intersection information is of no value. However, in certain heuristics the information on self-intersections of a stroke can prove useful. The heuristics might be used to facilitate the recognition of not only visual shapes but control gestures that do not necessarily resemble any visual objects.

The algorithm presented is based originally on the work by Shamos and Hoey [98]. They developed so called *sweep line algorithm* (or *plane sweep algorithm*) the idea of which is based on a conceptual sweep line that is swept over a surface. The sweep line is stopped at certain points for calculations and condition evaluations. The benefit of using the sweep line is the reduction in the number of calculations required to produce the output. The technique has many applications in computational geometry where the computational complexity can be dramatically reduced compared to naive algorithms.

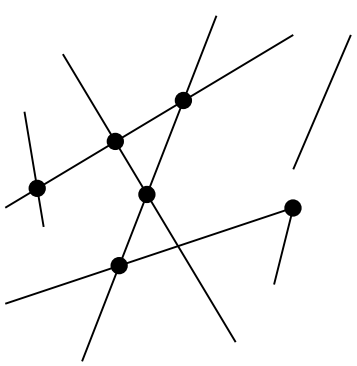
Building upon the work of Shamos and Hoey the sweep line algorithm was extended by Bentley and Ottmann [99]. Their algorithm calculates the intersection points of a set of line segments and cleverly avoids exhaustive checking for intersection between each segment. The geometrical basis of the algorithm is highly intuitive and the technique is described below together with a series of illustrating figures.

However, some degenerative cases of the input, as well as implementation details, are omitted for clarity. The algorithm is described in detail in [80] including proofs for correctness and running time as well as some implementation pitfalls. The outline of the approach is presented below.

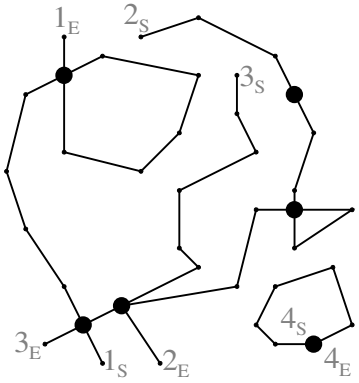
4.4.2 Input & output

The input for the algorithm is a set of n line segments $S = s_1, \ldots, s_n$, each defined by two points. In the context of hand-drawn strokes, the lines are obtained by splitting each polyline into its parts. The output is a set of m intersection points $P = p_1, \ldots, p_m$ between the lines. The line segments are considered to include the start and end point. Thus two lines with end points coinciding are considered to intersect (even if they are collinear).

However, in the case of lines obtained from a single polyline it is noted that all consecutive lines intersect. These intersections can simply be removed from the output. The input and output of the algorithm is depicted in figure 27.



(a) Input: line segments. Output: intersection points between the line segments.



(b) Input: each line segment of each polyline. Polylines are numbered for clarity (x_S and x_E denote the start and end of stroke x respectively). Note that the output includes self-intersections (polylines 1 and 2), a self-intersection which closes the polyline (polyline 4) and an intersection at which the lines only "touch" each other (intersection between polylines 2 and 3).

Figure 27: Input and output of the algorithm.

4.4.3 Algorithm overview

The algorithm is based on a horizontal sweep line l that is thought of as starting the sweep above all line segments. Gradually l is moved downwards in discrete steps. Each step moves the sweep line to a position where it coincides with an *event point*. At each event point calculations are made to find possible intersections and to update the *status of the sweep line*.

The status of the sweep line is defined as the ordered sequence of segments intersecting the sweep line (see figure 28). The order of the line segments is resolved by their intersection points with the sweep line. The line segments with the intersection points on the left come first in the ordering. When the sweep line is moving downwards line segments are added to the status, removed from the status and their order is changed. The rationale behind maintaining the status is to reduce the number of intersection checks. The data structure used to hold the status is denoted by τ.

In addition to the sweep line status the algorithm needs to maintain an *event queue* Q. The event queue contains the event points ordered by descending y-coordinate. If two points have the same y-coordinate, the one with smaller x-coordinate comes first. Figure 29a illustrates the event queue ordering. There are three types of event points: *upper point* of a line segment, *lower point* of a line segment and an *intersection point*. Upper and lower point of a line segment is distinguished based on the same criteria as with the ordering of Q (see figure 29b).

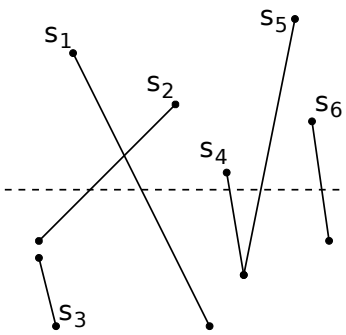
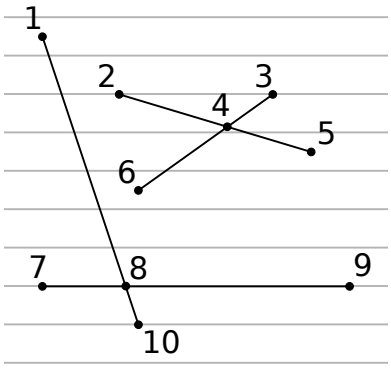
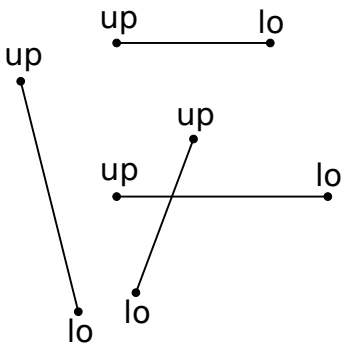


Figure 28: The status of the sweep line (dashed): s_2, s_1, s_4, s_5, s_6.



(a) Ordering of points in the event queue. Points with bigger y-coordinate and smaller x-coordinate come first. If points lie on the same horizontal line the ones with smaller x-coordinate come first. Horizontal lines (gray) added for clarity.



(b) Distinction of upper (up) and lower (lo) points of line segments. In the case of horizontal lines the left and right end points are denoted upper and lower points respectively.

Figure 29

The sweep line iterates through the event points and at each point the type of the point determines the actions to take. At an upper point a new line segment is intersecting with the sweep line (figure 30a). The line segment must be added to the status of the sweep line while maintaining the order constraint of the status. Next at most two intersection checks are made since the new segment might intersect with the ones already in the status. The lines to check are the one to the left and the one to the right of the new line segment. If any intersections are found (below the sweep line) they are added to the event queue. The new line might have intersection points with other lines in the status as well, but these are detected in later steps of the algorithm.

At an intersection point, two line segments (the ones that are intersecting) in the status switch their order (figure 30b). This means that each of them gets at most one new neighbor — "At most" since the segment can end up being the leftmost or rightmost segment in the status. Again, the two lines are checked for intersections with their new neighbors and any intersection points found are added to the event queue.

Facing an end point of a line segment means that the line segment is removed from the sweep line status (figure 30c). Thus, the left and right neighbor (if there are any) of the segment now become adjacent and are checked for an intersection. Once again, the possible intersection point is added to the event queue.

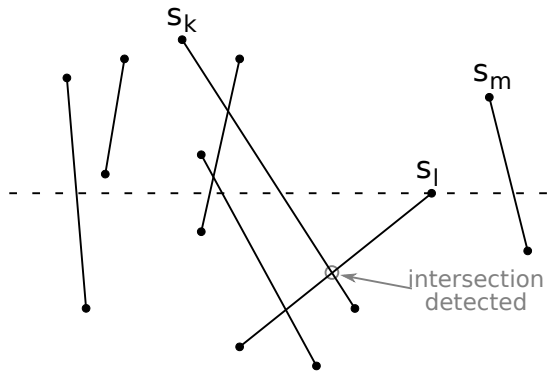
Advancing in this manner has two implications. First, at each point only the neighboring line segments are checked for intersections which is a tremendous gain in the running time of the algorithm. Second, the correctness of the algorithm is guaranteed by the invariant: all intersection points above the sweep line have been found. The invariant is based on the fact that a pair of intersecting line segments are adjacent in the sweep line status at some point of the execution of the algorithm. Since at some point they are adjacent they are ensured to be checked for intersection.

4.4.4 Computational complexity

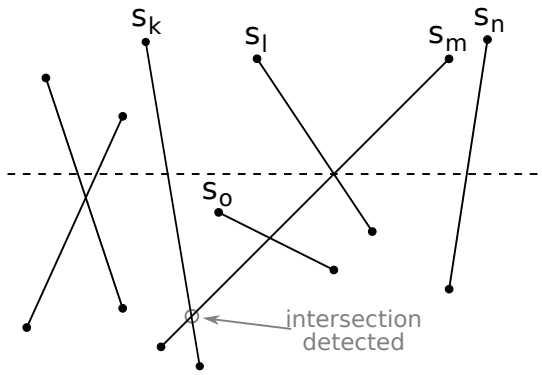
The naive approach to the problem would be to check every line segment against each other for intersections. The implementation of the algorithm is very simple and it works for small input size n. However, with larger input the $O(n^2)$ running time is highly unfavorable. The problem of the algorithm is that in most cases each line segment is far away from the majority of other line segments. Intuitively it would be more efficient to check for intersections only between the line segments that lie close to each other. The sweep line and its status data structure is designed to tackle this deficiency in the brute force algorithm.

In the original work of Shamos and Hoey an $O(n \log n)$ algorithm is presented for determining whether n planar line segments intersect [98]. They also prove that the problem of finding all k intersections has the lower bound $\Omega(n \log n + k)$. The finding of such algorithm is left as an open problem. Bentley and Ottmann address the problem indirectly by providing the algorithm presented above. It has the running time of $O(n \log n + k \log n)$. The algorithm is clearly output-sensitive since it has to iterate over each intersection point.

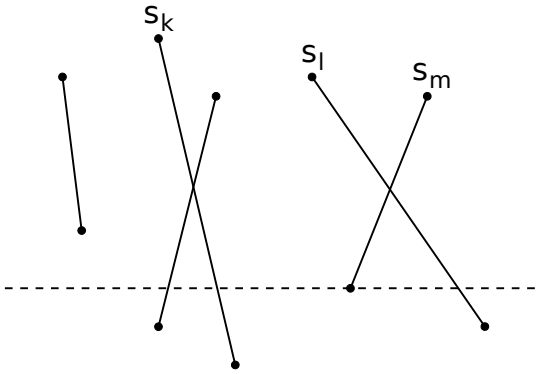
Certain constraints on the data structures used in the algorithm must be set in order to achieve the $O(n \log n + k \log n)$ running time [80]. The event queue Q must support operations for fetching the next event point, inserting a new one and testing whether a point is already present in Q. The event queue can be implemented as a balanced binary search tree which takes $O(\log m)$ time for fetch, insert and lookup operations (m is the number of points in the queue) [81]. A variation of a balanced binary search tree can also be used as the data structure of the sweep line status τ. Consequently, neighbor search and update operations take $O(\log n)$ time. An $O(n)$ storage requirement can be achieved by storing in Q only the intersection points among adjacent segments. This can be achieved by modifying the algorithm to



(a) At an upper point (of line segment s_l) the line segment is added to the status of the sweep line and is checked for intersections with its neighbors in the status (s_k and s_m) and possible intersection points are added to the event queue.



(b) At an intersection point the intersection is recorded. Intersecting line segments (s_l and s_m) switch their order in the status of the sweep line. Thus, they have to be checked for intersections with their new neighbors (s_l is checked against s_n and s_m is checked against s_k). Possible intersection points are added to the event queue.



(c) At a lower point, the line segment (s_m) is removed from the status of the sweep line. This requires the checking of new neighbors (s_k and s_l) for intersections. A possible intersection point is added to the event queue.

Figure 30: The three types of event points and the corresponding action taken by the algorithm when sweeping over them.

delete an intersection point when the intersecting lines stop being adjacent and re-adding it when the lines become adjacent again.

4.4.5 Alternative approaches

The question whether the lower bound of $\Omega(n \log n+k)$ can be achieved was left open in Bentley and Ottmann's algorithm [100]. The question was answered by Chazelle and Edelsbrunner who presented an optimal algorithm for solving the problem [101]. Their approach is also based on the sweep line technique but employs additional ideas and requires a rather complex proof for the running time. The storage requirement of the algorithm is $O(n + k)$.

Thus, the question remained whether an algorithm exists that runs in optimal time and also has the optimal storage requirement of $O(n)$. A few years later Balaban introduced an algorithm with the same time complexity but with the optimal $O(n)$ space requirement [102]. In addition to the deterministic algorithms above, techniques using randomized approaches were presented by Clarkson and Shor in [103] and by Mulmuley in [104]. Their algorithms achieve the optimal $O(n)$ spare requirement and the *expected* running time of $O(n \log n + k)$.

Algorithm 9 intersections(S)

Input: INPUT a set of line segments $S = s_1, \ldots, s_n$
Output: OUPUT intersection points between segments in S

Initialize an empty event queue Q and sweep line status τ
Insert both end points of each segment into Q
while $Q \neq \emptyset$ **do**
    pop (get and remove) the next event point P in Q
    handleEventPoint(P)
end while
return all intersection points found when executing *handleEventPoint*

Algorithm 10 handleEventPoint(P)

Input: INPUT event point P
Output: OUPUT (No output. The function stores possible intersection points.)

// $U(P)$: set of segments whose upper point is P
// $C(P)$: set of segments that have P in their interior
// $L(P)$: set of segments whose lower point is P
Find $U(P)$, $C(P)$ and $L(P)$

if $U(P) \cup C(P) \cup L(P)$ contains more than one segment **then**
    Add P to the set of intersections points
end if
Delete the segments in $L(P) \cup C(P)$ from τ
Insert the segments in $U(P) \cup C(P)$ into τ
// Deleting and re-inserting the segments in $C(P)$ ensures their correct order

if $U(P) \cup C(P) = \emptyset$ **then**
    Get s_l and s_r (left and right neighbor of P) from τ
    if s_l and s_r intersect below the sweep line **then**
        Insert the intersection point into Q
    end if
else
    // Let s_{lm} and s_{rm} be the leftmost and righmost neighbors of $U(P) \cup C(P)$ in τ
    // Let s_l be the left neighbor of s_{lm}
    // Let s_r be the right neighbor of s_{rm}
    if s_{lm} and s_l intersect below the sweep line **then**
        Insert the intersection point into Q
    end if
    if s_{rm} and s_r intersect below the sweep line **then**
        Insert the intersection point into Q
    end if
end if

4.5 Compound features

The geometric properties of the input strokes described above are used to obtain a set of representative features to be used in shape classification. First, the area and perimeter are calculated for the convex hull, enclosing rectangle and enclosed triangle. The values are not usable as such since they are dependent on the size of the drawn shape. Certain combinations of the values are of more use.

For example, if we denote the perimeter and area of the convex hull with P_{ch} and A_{ch} respectively the *thinness ratio* P_{ch}^2/A_{ch} is especially useful in distinguishing circles from other shapes [59]. Since a circle covers the maximum area with a constant perimeter, its thinness is the smallest ($4\pi \approx 12.57$) of all planar figures (see figure 31). The other compound features utilized in the recognition process are presented in the next section.

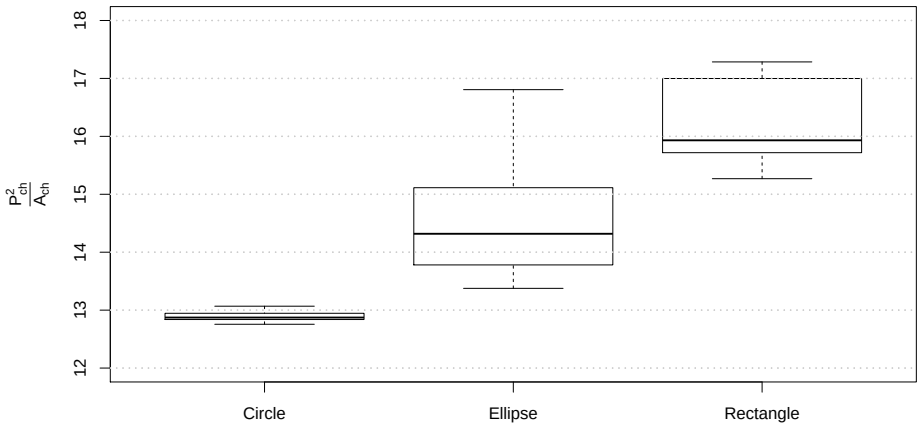


Figure 31: Thinness ratio for three shapes. The thinness of circles is the smallest.

5 Sketch recognition using geometric features and heuristics

Various approaches to sketch recognition have been presented in the past 15 years. They differ mostly in their level of domain specificity, extensibility and hierarchicality of recognition. The majority of the articles focus heavily on the shape classification. Some also include techniques for preprocessing, shape fitting and regularization.

This section describes an approach to sketch recognition inspired by previous work. The focus of this thesis is on the first two steps of shape recognition, namely curve preprocessing and shape classification. Shape fitting is implemented with simple rules to complete the system. Shape regularization is beyond the scope of the thesis.

The previous section gave a detailed analysis of the algorithms used for preprocessing and shape classification. This section covers the integration of those algorithms into a working sketch recognition system. Furthermore, the shape classification method is presented. Also, the nature of the user interface and essential usability matters are briefly discussed.

5.1 Approach step-by-step

The system is used by drawing a set of strokes with a mouse or a digitizing tablet. A timeout value is used to decide which strokes belong to the same shape. That is, after the last stroke has ended and no new strokes have been started within a given time, the strokes drawn so far are collected and recognized as a whole. The next stroke starts a new shape. This timeout-based approach is used in [59]. Additionally, one could define a threshold value for the distance between strokes. If a stroke (within the timeout value) is far away from the previous ones, the previous strokes are input to the recognition system and the new stroke is considered to start a new shape. Alvarado and Davis point out that the requirement of completing a shape with consecutive strokes imposes a restriction on the drawing style of the user [37]. While strictly true, the restriction is quite minimal in this context since a vast majority of simple shapes are drawn in this manner in any case. The input is thus an ordered collection of strokes that contain the list of input points together with their *timestamps* captured by the input device. XML can be used to present an example of the input data (see listing 1).

Listing 1: Example input for the recognizer

```
<Strokes expectedshape="Rectangle">
    <Stroke> <-- First stroke -->
        <Point x="830" y="783" time="1377360941"/> <!-- First point -->
        <Point x="826" y="780" time="1377361029"/> <!-- Second point -->
        ...
        <Point x="781" y="739" time="1377361090"/> <!-- Last point -->
    </Stroke>
    <Stroke> <!-- Second stroke -->
        ...
    </Stroke>
</Strokes>
```

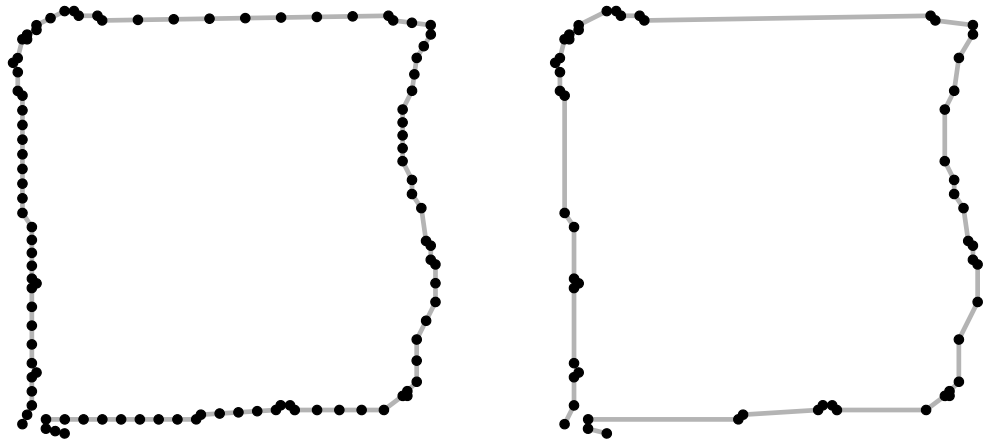
The raw input points are first preprocessed in four steps as described in figure 32. The preprocessing is performed similarly for each stroke. For example, the preprocessing closes only shapes that consist of a single stroke. Thus, the benefit of preprocessing is merely to clean each individual stroke rather than search for interstroke relations. In the majority of cases the polyline approximation step also dramatically reduces the number of points that need to be processed by other algorithms.

In figure 32 one can see the co-operative nature of the end refinement algorithm and stroke closing algorithm. Without the end refinement step the stroke closing would not work, since in the polyline approximation the ends of the stroke are pointing away from each other. Thus the method of extending the ends of the strokes described in section 3.3 would fail.

The course of feature extraction, shape classification and shape fitting is depicted in figure 33. First the convex hull is calculated for the preprocessed input points. Next the minimum-area enclosing rectangle and the maximum-area enclosed triangle are calculated for the convex hull. The properties of the three polygons together with other values, such as the total number and length of strokes, are used to obtain likelihood for each shape. The shape with the highest likelihood is selected and a fitted shape is created to replace the original user sketch.

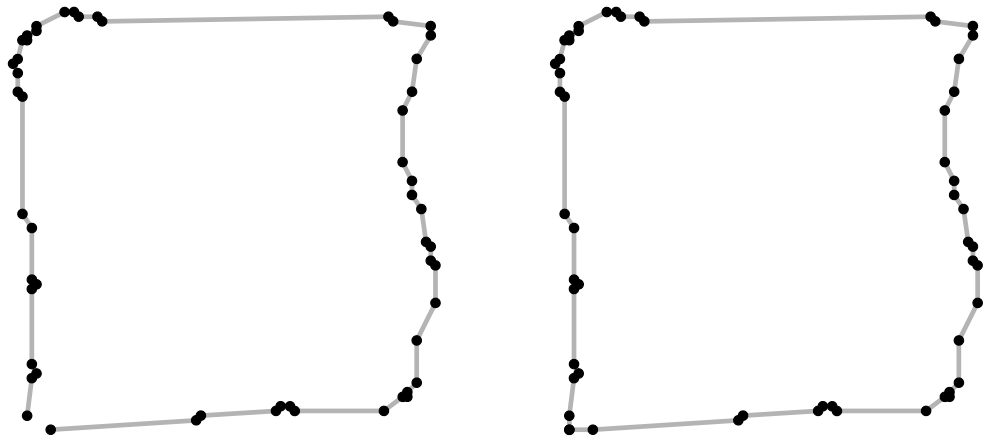
5.2 Shape decision process

To identify the drawn shape, a classification method similar to the one employed by Fonseca et al. is used [60]. First, a set of 50 training samples are drawn for each shape. Second, the obtained training material is used to construct a boxplot for each feature. A boxplot graph is used to depict numerical data graphically to indicate certain statistical features of the data (see figure 34a for explanation) [105, 106]. Third, the boxplot figure is used to obtain a set of so called *fuzzy sets* for each shape-feature pair. Finally, certain manual corrections are made to the fuzzy sets. The corrections account for information that cannot be embedded in the training samples. For example, the length of a line can be infinitely large and thus the information cannot be present in the training data.



(a) Original input polyline

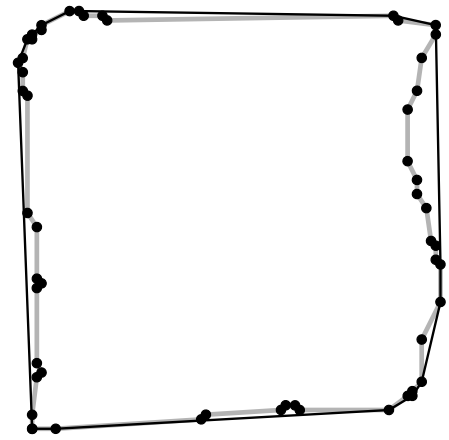
(b) Polyline approximation (filtered for redundant and duplicate points)



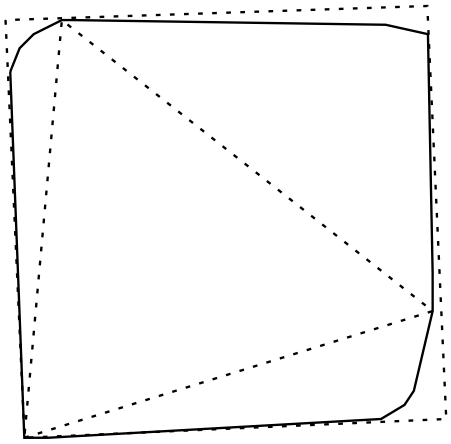
(c) Stroke ends refined

(d) Stroke ends closed

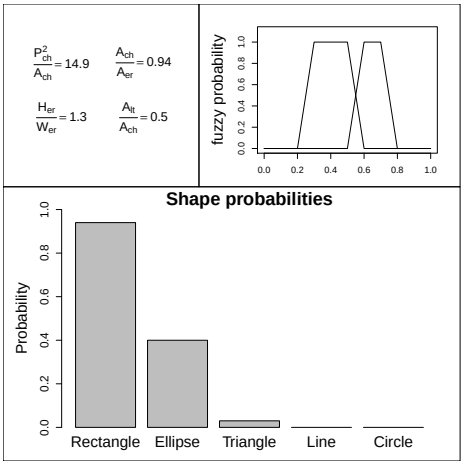
Figure 32: The preprocessing steps of the recognition system. First Douglas-Peucker algorithm is used to filter out redundant points. Duplicate points (consecutive points with the same coordinates) are removed as well in (a)–(b). Stroke ends are refined with the averaging based method in (b)–(c). Finally, stroke ends are closed properly in (c)–(d).



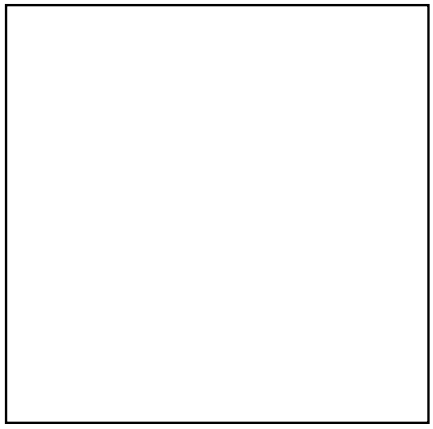
(a) Convex hull calculation



(b) Feature polygon calculation



(c) Feature value and probability calculation and shape classification



(d) Rectangle fitted to the strokes

Figure 33: The feature extraction, shape classification and shape fitting steps. First, the convex hull of the input is calculated in (a). Second, enclosed triangle and enclosing rectangle are calculated in (b). A set of feature values are calculated which are then used to calculate likelihood (probability) for each shape (c). Fitted rectangle in (d).

5.2.1 Fuzzy sets

Fuzziness is a concept of vagueness concerning the description of the semantic meaning of events and statements [107]. Fuzzy sets are a way of handling the vagueness (uncertainty) in grouping items in the data. In a deterministic context an item either belongs to a set or not: "the scribble drawn by the user *is* a rectangle". In fuzzy set theory the relation is more vague: "the scribble is a rectangle with 60% probability". In the field of sketch recognition fuzzy sets can be used to handle the inherent uncertainty in the data.

The figure 34b illustrates a fuzzy set obtained from the boxplot. The fuzzy sets attaches a probability to the value of the feature. The probability is called *degree of membership*. Several fuzzy sets are used to define each shape. Some features are used to identify the correct shape, while others can be used to filter out mismatches.

5.2.2 Recognized shapes

The recognition system implemented can distinguish between eight shapes: *arrow*, *circle*, *diamond* (or *rhombus*), *ellipse*, *line*, *rectangle*, *star* and *triangle*. Arrow and star are recognized by using simple heuristics that are described later. The classification of rest of the shapes rely solely on the geometric features. The ninth shape, *scribble*, is used as the "unknown shape" when the system cannot recognize the strokes as any other shape. The recognized shapes are depicted in figure 35.

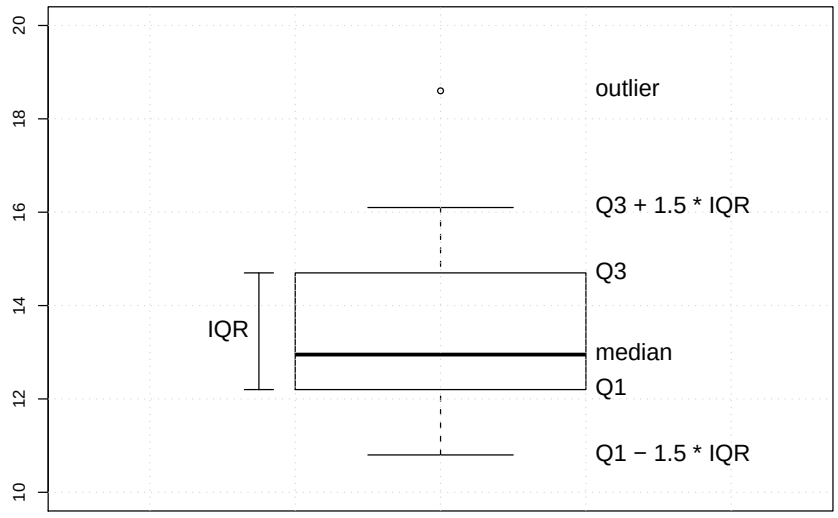
The line is recognized by using the *thinness ratio* which is defined as P_{ch}^2/A_{ch} where P_{ch} and A_{ch} denote the perimeter and area of the convex hull respectively. The convex hull of a line is "thin" in the sense that it is little in area but big in perimeter. Thus, the thinness ratio for line is significantly greater than for any other shape (see figure 36).

The thinness ratio can also be used to distinguish circles. Since a circle covers the maximum area with a constant perimeter, its thinness is the smallest ($4\pi \approx 12.57$) of all planar figures. Figure 37 illustrates the distinctive values for circles.

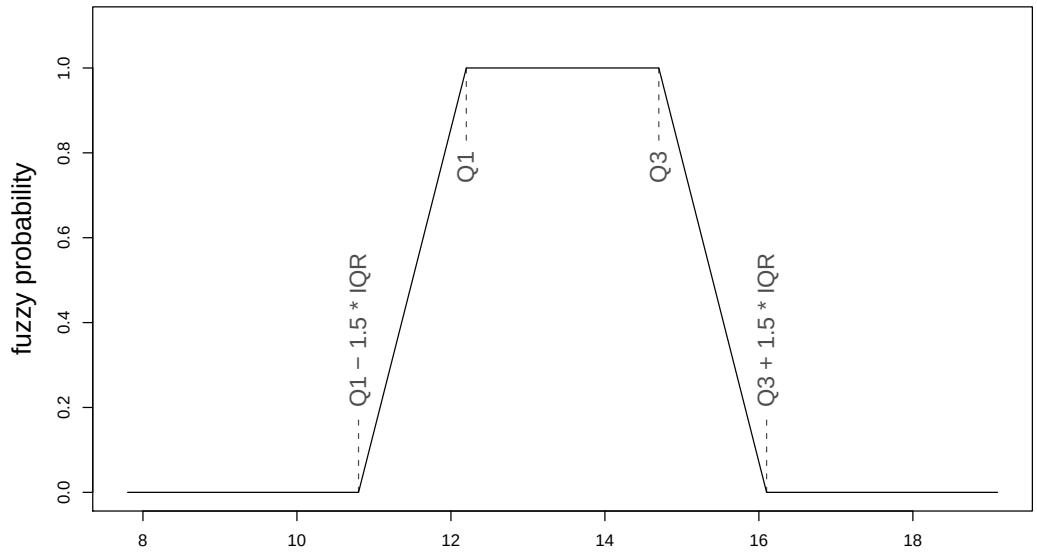
The recognition of triangles can be achieved by using the convex hull and the maximum-area triangle enclosed within it. The area of the triangle is usually only slightly smaller than the area of the convex hull. Thus, one uses the ratio A_{tr}/A_{ch}, where A_{tr} and A_{ch} denote the areas of the triangle and convex hull respectively. The ratio is near unity for triangles and smaller for other shapes (see figure 38).

The rectangle is recognized similarly to the triangle. One utilizes the fact that the convex hull and minimum-area rectangle enclosing it are almost identical for rectangles. Thus, the ratio P_{ch}/P_{re} is used, where P_{ch} and P_{re} denote the perimeters of the convex hull and enclosing rectangle respectively. The ratio is near unity for rectangles and smaller for other shapes, excluding lines (see figure 39).

Distinguishing the diamond and the ellipse from each other and the rectangle imposes another challenge. The distinction can be achieved by employing two new features: $A_{ch}^2/(A_{tr}A_{re})$ and A_{tr}/A_{re}. Symbols A_{ch}, A_{tr} and A_{er} denote the areas of the convex hull, enclosed triangle and enclosing rectangle respectively. The values of both features for the three shapes are illustrated in figures 40 and 41. Diamond can be distinguished by using the first ratio and rectangle by using the second ratio.



(a) The bottom and top of the box represent $Q1 = 25^{th}$ and $Q3 = 75^{th}$ percentile respectively. The vertical band inside the box is the median (50^{th} percentile). *Interquartile range* is defined to be the difference between the third and first quartiles: $IQR = Q3 - Q1$. In this thesis the whiskers are set to the lowest data point within $1.5 \times IQR$ and the highest data point within $1.5 \times IQR$. Data point that does not fit within the whiskers is shown as an outlier. The boxplot was produced using the following data: (10.8, 11.0, 11.1, 11.9, 12.2, 12.2, 12.3, 12.4, 12.5, 12.6, 12.9, 13.0, 13.6, 14.2, 14.3, 14.6, 14.7, 15.1, 15.5, 15.5, 16.1, 18.6).



(b) Fuzzy set derived from the boxplot above. Values outside the whiskers result in probability 0, values between the interquartile range in probability 1, and values between the whiskers and *Q1* and *Q3* in probabilities between 0 and 1.

Figure 34

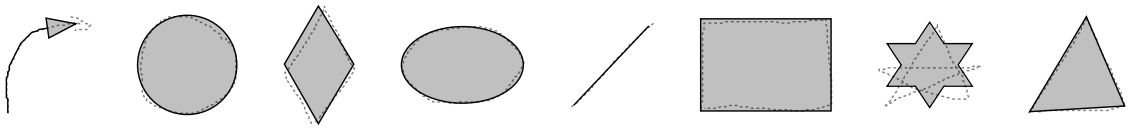


Figure 35: The shapes recognized by the system together with the scribble that produced them. Arrow, circle, diamond, ellipse, line, rectangle, star and triangle.

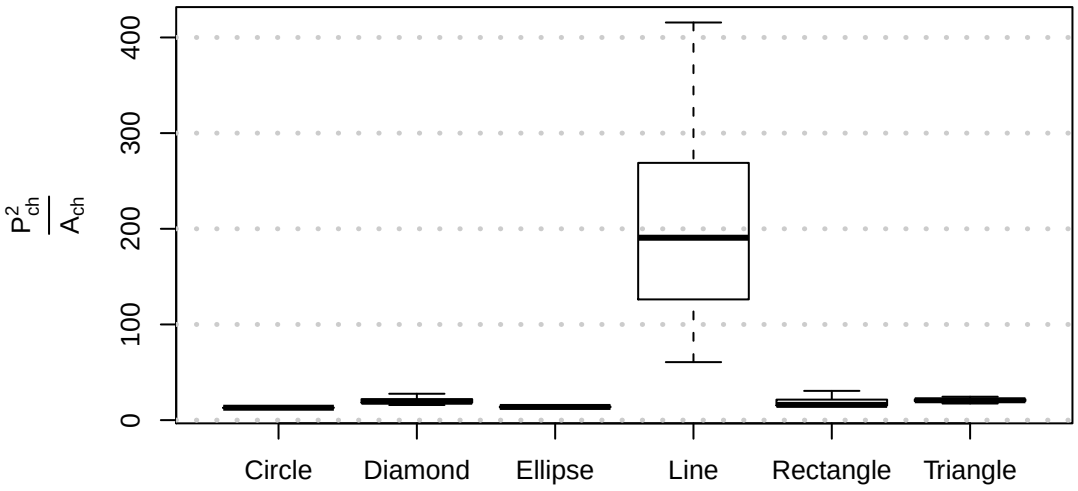


Figure 36: The line is easily distinguished from other shapes by its thinness ratio.

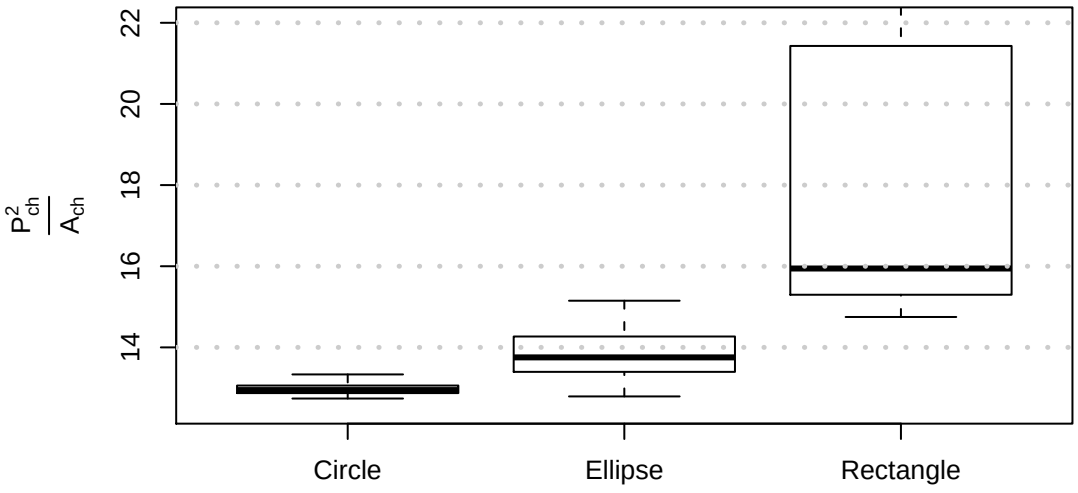


Figure 37: Thinness ratio can be used to recognize circles.

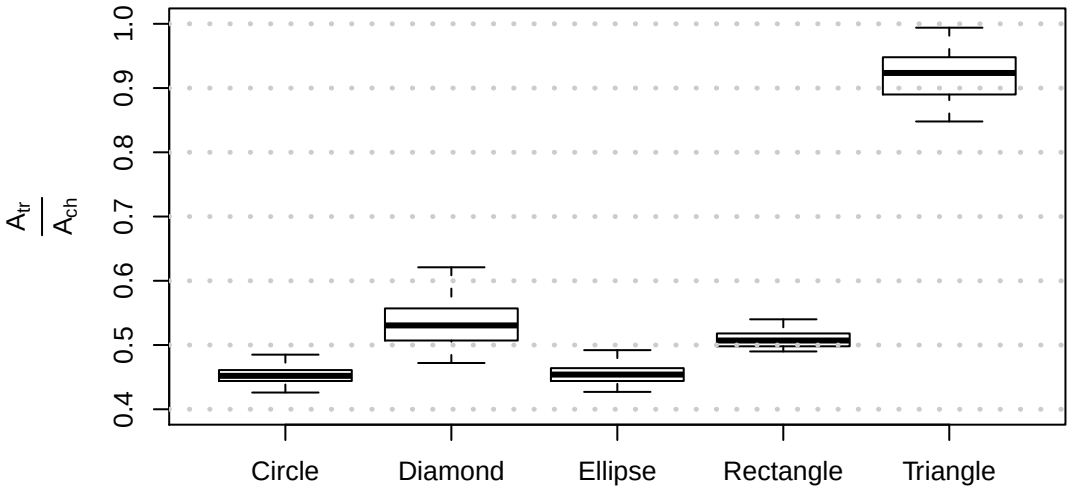


Figure 38: The ratio of areas of the maximum-area enclosed triangle and convex hull is near unity for triangles.

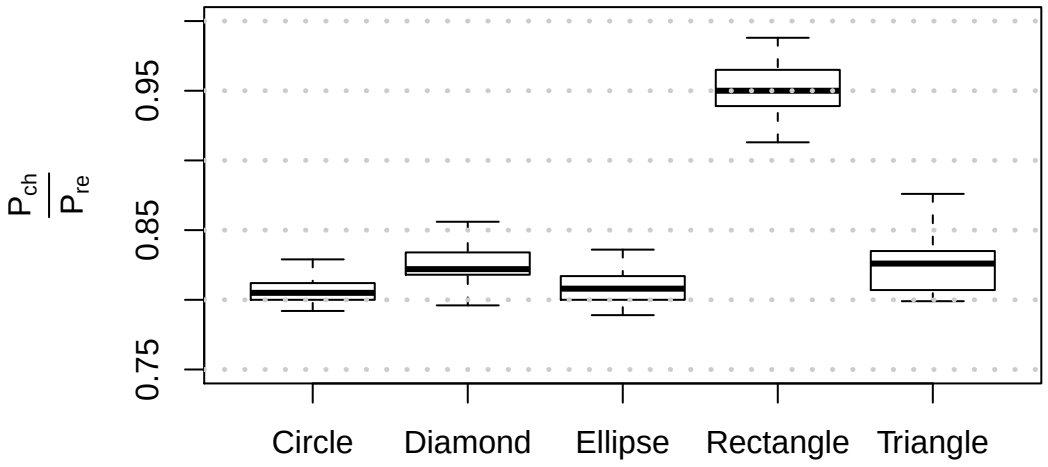


Figure 39: The perimeters of the convex hull and minimum-area rectangle enclosing it are roughly the same.

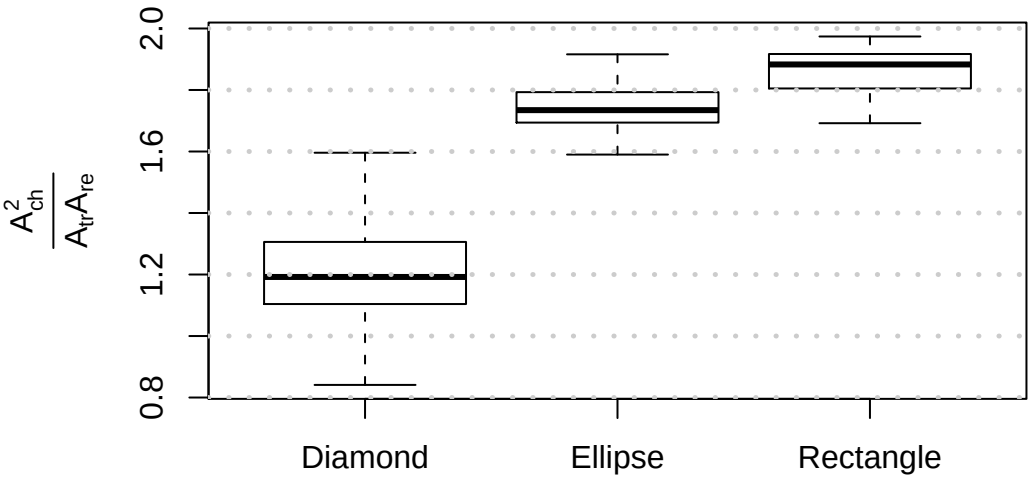


Figure 40: The diamond has slightly smaller values than ellipse and rectangle for $A_{ch}^2/(A_{tr}A_{re})$.

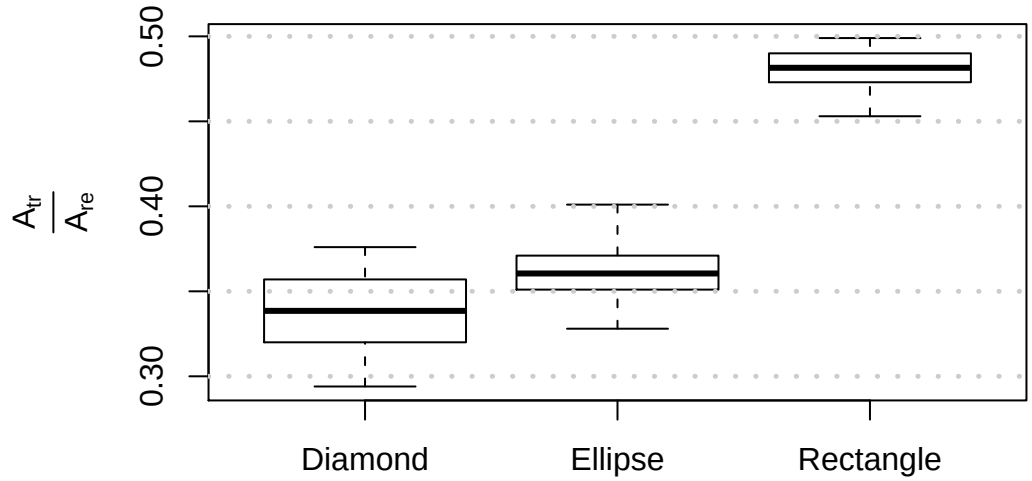


Figure 41: The area of the enclosed triangle is roughly half of the enclosing rectangle for rectangles and smaller for diamonds and ellipses.

Furthermore, to distinguish ellipses from circles one calculates the bounding box (smallest rectangle with sides collinear with the coordinate axes) for the input and compares the ratio of its width and height. Ellipses that have the width-height ratio near unity are recognized as circles (see figure 42).

For all closed, convex shapes one utilizes the ratio of total stroke length (sum of all stroke lengths) and perimeter of the convex hull, l_s/P_{ch}. The ratio can be used to filter out unclosed and concave scribbles since the ratio is near unity for closed, convex shapes (see figure 43). The recognized shapes and properties used to distinguish them are gathered in table 1.

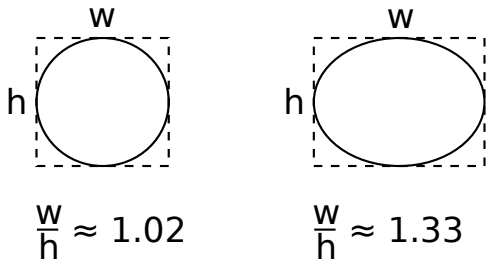


Figure 42: Circle (left) and ellipse. Bounding box of a set of points is the smallest rectangle that contains the points and has sides collinear with the coordinate axes. The bounding box dimensions can be used when distinguishing circles from ellipses. Width-height ratio near unity implies a circle.

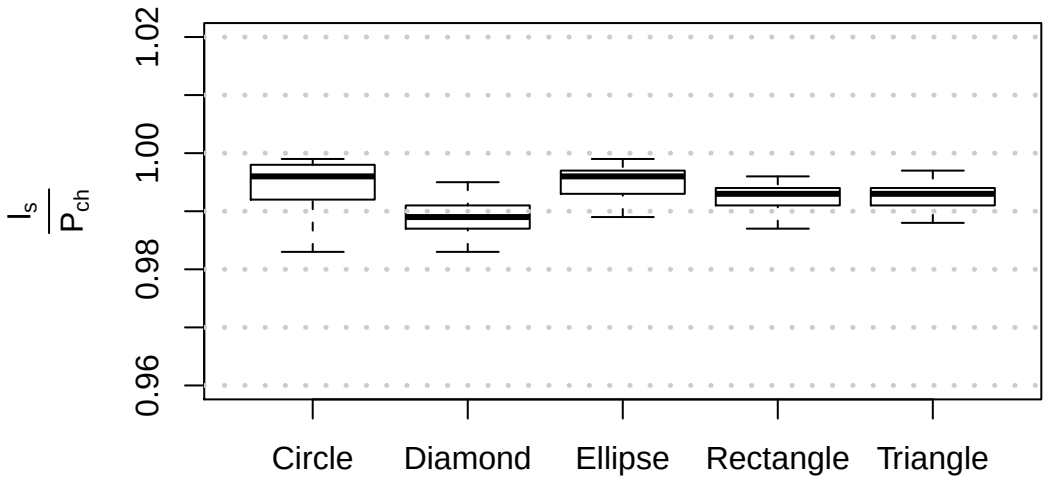


Figure 43: The length of strokes for closed convex shapes is roughly equal to the perimeter of their convex hull. l_s and P_{ch} denote the length of all strokes (sum of lengths) and perimeter of the convex hull respectively.

Table 1: Features used to recognize each shape. Abbreviations: l_s total length of strokes, P perimeter, A area, ch convex hull, tr enclosed triangle, re enclosing rectangle, w, h width and height of the bounding box.

	$\frac{P_{ch}^2}{A_{ch}}$	$\frac{l_s}{P_{ch}}$	$\frac{w}{h}$	$\frac{A_{ch}^2}{A_{tr}A_{re}}$	$\frac{A_{tr}}{A_{re}}$	$\frac{A_{tr}}{A_{ch}}$	$\frac{P_{ch}}{P_{re}}$
Circle	×	×	×				
Diamond		×		×	×		
Ellipse		×	×	×			×
Line	×						
Rectangle	×	×		×			×
Triangle		×				×	

5.3 Recognizing arrow and star

To detect more complex shapes such as arrows and stars the system has to use simple heuristics to aid recognition. The heuristic rules utilize the lower level geometric features. An *arrow* can be drawn with two strokes. The first stroke constitutes the *shaft* and the second stroke the *head*. Furthermore, the shaft is usually drawn from tail to head. This pattern can be utilized in the heuristics as is done by Fonseca and Jorge [59]. A modification of their heuristics for recognizing arrows is clarified in figure 44 and algorithm 11. Particularly, the restriction that the convex hull of the second stroke should contain the last point of the first stroke, is omitted in Fonseca and Jorge's method.

The *star* is also recognized by using a certain convention of drawing. Star can be drawn with one stroke consisting of five line segments. Figure 45 illustrates this style of drawing and clarifies the heuristics employed. The recognition is based on the observation that the stroke forms five self-intersections in the middle of the shape. The number of self-intersections is used as the basis for recognition. However, there are often other self-intersections as well. They are usually located in the spikes of the star and imply that the mere counting of the intersections would produce suboptimal results. Consequently, the algorithm only counts the intersections that lie closer to the center of the shape. In practice this is achieved by calculating the centre of gravity point of the stroke and generating a scaled convex hull for the input. A higher recognition rate can be achieved by considering only the intersection points that lie inside the scaled convex hull.

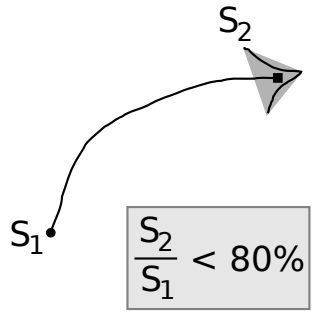


Figure 44: Heuristics for recognizing arrows. First, an arrow should consist of two strokes. The length of the second one (head) should be smaller than certain proportion of the length of the first one (shaft). Finally, the convex hull (gray) of the second stroke should contain the last point of the first stroke (indicated by the black square).

Algorithm 11 isArrow(S)

Input: INPUT a list of n strokes $S = s_1, \ldots, s_n$
Output: OUPUT boolean indicating whether the strokes constitute an arrow

1: **if** $n = 2$ **and**
   $length(S_2)/length(S_1) < 80\%$ **and**
   convex hull of S_2 contains the last point of S_1 **then**
2: **return true**
3: **else**
4: **return false**
5: **end if**

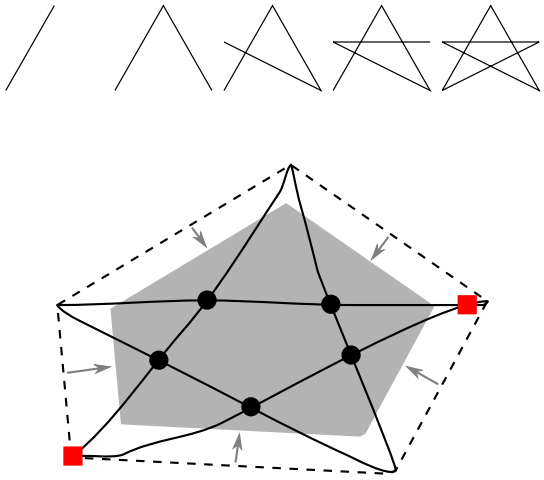


Figure 45: Step-by-step drawing convention for a star (above) and heuristics for recognizing it. The algorithm first calculates the convex hull (dashed line) and intersection points (dots) for the original stroke (solid line). For each intersection point, it is checked whether it is contained in the shrunk convex hull (gray area). If there are five intersections inside the shrunk convex hull, the stroke is recognized as a star. The scaling of the convex hull excludes "false" intersection points (red squares). The one on the left is the self-closing point of the stroke. The other results from a circlet in the spike of the star.

Algorithm 12 isStar(S)

Input: INPUT a list of n strokes $S = s_1, \ldots, s_n$
Output: OUPUT boolean indicating whether the strokes constitute a star

```
1: if n ≠ 1 then
2:     return false
3: end if
4: Calculate the self-intersection points of the stroke.
5: Calculate the convex hull for the stroke points.
6: Generate a scaled (shrunk) convex hull CH_scaled.
7: if CH_scaled contains exactly 5 intersection points then
8:     return true
9: else
10:    return false
11: end if
```

5.4 Problems of the approach

Some of the major problems of the presented approach have been reported in earlier studies. Using only global geometric properties makes it hard to distinguish ambiguous shapes such as $k$gons with $k \geq 5$ [33, 40, 51]. Furthermore, the approach is inherently an ad hoc one. Extending the recognition system is cumbersome. The geometric primitives (circles, ellipses et cetera) can be drawn with as many strokes as possible but each shape has to be drawn with consecutive values within the timeout value [35, 34]. In addition to these drawing style restrictions the system is incapable of identifying the constituent parts of the recognized shapes.

The recognition rates of the system are good for part of the primitive shapes drawn in a certain manner. However, this comes with the cost of a few fundamental deficiencies. First, the approach is ad hoc, tailored for a specific purpose with a pre-defined shape set and lacks hierarchical nature. Second, it follows that extending the system with new shapes and functionality cannot be done in a consistent manner. This, in turn, leads to even more ad hoc solutions. Third, the approach is inherently unpredictable considering the possibility of false positives. Some scribble, not intended to be recognized as any shape, might have the same geometric features as one of the shapes in the pre-defined shape set (see figure 46). In some cases the problem can be solved easily by introducing new features that filter out the false positives. For example, the figure 46 depicts a situation that can be handled by introducing *hollowness* [58]. Hollowness counts the proportion of points near the centre of the scribble. For hollow shapes, such as polygons and ellipses, hollowness is zero.

The lack of hierarchy in handling the shapes prevents the approach from being applicable in recognizing more complex shapes. For example, the diagram notations for a database (network diagram) or a transistor (circuit diagram) are impractical to recognize using geometric features alone, even though the parts of the shapes could be easily recognized (see figure 47). For working recognition one needs information on individual parts of the shape as well as their interrelational constraints. LADDER (A Language to Describe Drawing, Display, and Editing in Sketch Recognition) could be used for the purpose [41].

The previous section presented two extensions to the geometric feature-based recognition. The arrow and the star are recognized using heuristics that analyzes the number of strokes as well as relations (spatial and intersections) between the strokes. The heuristics uses lower level features provided by the base recognition system. However, introducing the new shapes requires domain insight and manual programming. This is far more laborious than for example the method presented by Calhoun et al., where new symbols can be added by merely providing a few examples of the new shape [35].

Finally, the restrictions on the drawing style degrade the usability of the system. Recognizing an arrow and a star relies strictly on a specific manner of drawing. The geometric primitives can be drawn more freely but nevertheless with consecutive strokes.

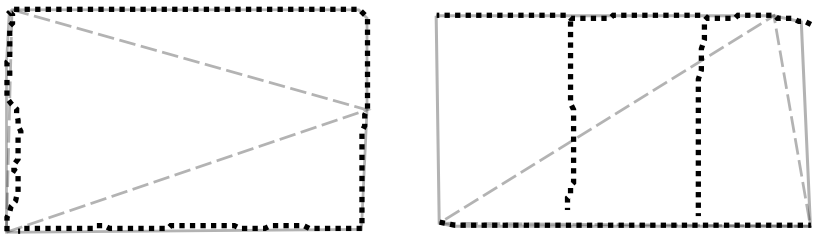


Figure 46: Two shapes recognized as rectangles. The original strokes are shown as black dotted lines. Convex hull and maximum-area triangle within it are shown in gray solid and dashed lines respectively. Although the strokes on the right does not represent a rectangle it is recognized as one since its properties are virtually the same as with the shape on the left: total length of strokes, convex hull and the ratio of areas of the enclosed triangle and convex hull.

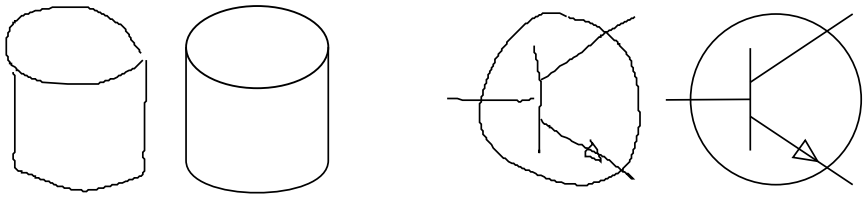


Figure 47: Diagram notation for database (left) and transistor (right). The primitive parts of the database are ellipse, two lines and an arc. The primitive parts of the transistor are four lines, a triangle and a circle. Especially in the case of the transistor hierarchical recognition methods are required.

6 Test setting and results

The development and verification of the algorithms in the system required the implementation of a simple drawing application. The application was further utilized in analyzing the recognition accuracy and computational performance of the approach. This section briefly describes the implemented software and the test setting used for the performance analysis. Results on the recognition rates as well as the computation times for individual steps of the recognition are reported based on the real input data gathered from a group of subjects.

6.1 Test framework

A computer program with a graphical user interface was implemented in order to test the recognition accuracy of the approach in practice. The program was written in Java programming language and it includes a drawing area, control buttons and a settings panel. The GUI is shown in figure 48. To facilitate the development of the algorithms and heuristics, the settings panel contains a variety of configuration options. The intermediate phases of the recognition (various preprocessing filters, feature polygon and intersection calculations) can be presented on the drawing area together with the original strokes as well as the final recognition result.

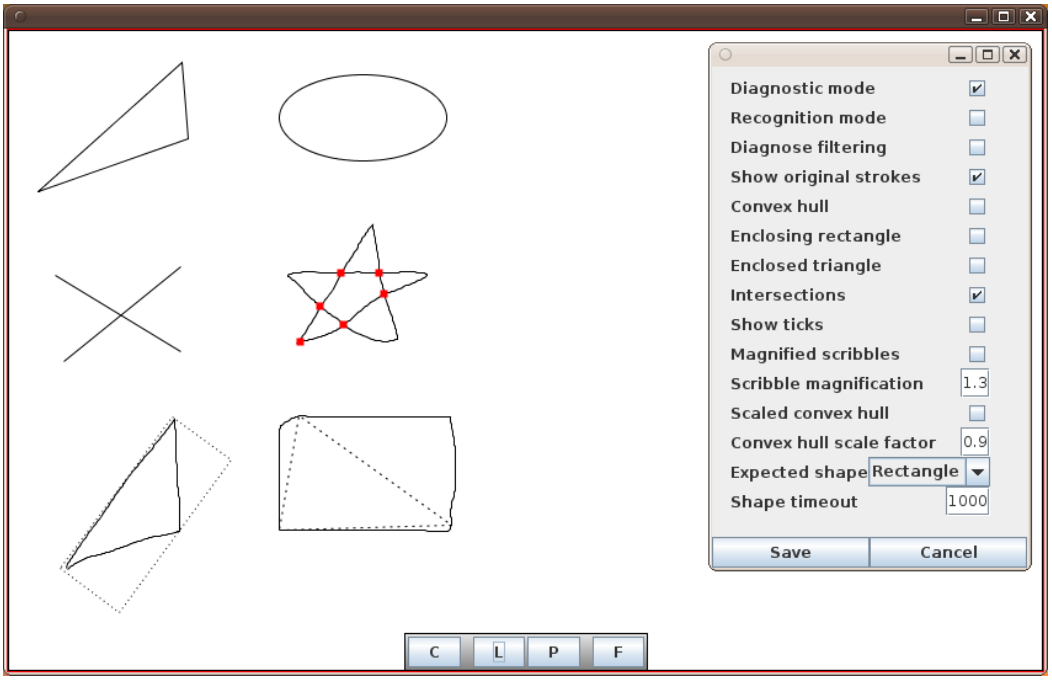


Figure 48: Main panel with recognized shapes (triangle, ellipse and crossing lines) and scribbles together with diagnostic artifacts (enclosing rectangle, enclosed triangle, intersections). Control buttons on the bottom: **(C)**lear drawing area, **(L)**oad scribble from file, **(P)**rint SVG source, **(F)**ull screen toggle. Settings panel (on the right) with various configuration options for helping development and analysis tasks.

The sketches were produced using a standard computer mouse. Using a digitizer tablet or a touch-based device is just a matter of plugging in a device that can be used to control the mouse pointer. Furthermore, functionality was implemented to save scribbles into files and to later load them for analysis. The files contained the raw points recorded by the program to enable the precise repetition of the recognizition process. Scribbles saved into files could also be batch processed to analyze the performance of the system with different configurations. For documentation purposes the shapes visible on the drawing area could be saved into file in vector format (SVG, scalable vector graphics).

6.2 Test setting overview

A group of 10 subjects was gathered to obtain realistic and varied enough test data for the analysis. The subjects were aged between 20–32 and hence accustomed to using a computer mouse. However, the subjects were in no way particularly experienced in drawing with a computer mouse. The subjects were told that they are going to test a sketch recognition system that can recognize and rectify certain shapes. First the subjects were allowed to briefly play around with the system to get comfortable with the system and drawing with a computer mouse. After the "warming-up session" the subjects were asked to draw 15 of each geometric shape (circle, diamond, ellipse, line, rectangle, triangle). The number of shapes actually drawn varied between $15 - 20$ for each subject.

The subjects were adviced to draw the shapes as they would draw them when using a pencil and paper, i.e., with no specific instructions on the style of drawing. The subjects were asked to draw the sketches "without too much accuracy as they would be merely sketching something rather than producing a final picture". There was no feedback given by the recognition system during the test setting. Only the user-drawn strokes were visible on the drawing area. These measures were taken to prevent the subjects from adjusting to a certain drawing style, speed or accuracy merely to improve the recognition.

Next the recognition performance of arrow and star was investigated. First the subjects were asked to "draw 5 arrows and 5 stars" with no other instructions to find out the variation among the subjects on how they would normally draw the shapes. This was done to find out the validity of the assumptions on the drawing style utilized in the recognition heuristics for arrows and stars. It was concluded that the subjects had a variety of different styles for drawing arrows and stars. Some subjects draw the shapes exactly as required by the recognition heuristics. However, majority employed one or several styles not recognizable by the heuristics. Hence, the "correct" drawing style needed to be explained to the subjects thus adding more restrictions for the approach.

Thereafter the assumed drawing style was explained to the subjects but the underlying recognition intricacies were not revealed since the users should not be concerned about the technical details to achieve adequate recognition rates. Then the users were asked to draw 15 arrows of varying length and 15 stars of varying size.

Table 2: Confusion matrix of the recognition results. The drawn (expected) shape is shown in the left and the recognized shape on the top. The percentage of correctly recognized shapes can be seen on the diagonal. The rightmost column (*Scribble*) represents the proportion of shapes that was not recognized as any shape.

	Arrow	Circle	Diamond	Ellipse	Line	Rectangle	Star	Triangle	Scribble
Arrow	**74%**								26%
Circle		**71%**		28%					1%
Diamond			**45%**	17%		22%		1%	15%
Ellipse		6%	10%	**81%**		1%			2%
Line					**100%**				
Rectangle				2%		**89%**			9%
Star							**66%**		34%
Triangle			1%					**94%**	5%

6.3 Recognition accuracy

Table 2 gathers the results of the recognition accuracy of the system. The overall recognition rate is 78% with a total number of 1316 input shapes. The overall recognition rate is satisfactory. However, the variation is considerable between the recognition rates for different shapes.

The line is the easiest shape to recognize since its thinness ratio is especially distinctive. Thus, quite expectedly the recognition rate is perfect. However, the majority of the lines drawn by the subjects were relatively long which eased the recognition. Noise with shorter lines is likely to introduce recognition errors.

Triangle was also almost perfectly recognized. The good results stem from the fact that no other shape is easily mixed with triangle. The failures resulted from "too round" triangles that resulted in too small minimum-area enclosed triangles in relation to the convex hull (see figure 49a).

The results on recognizing the rectangle are also good. The majority of cases where the sketch was not recognized at all were due to the rectangle being drawn slightly skewed (the opposite sides are not of equal length)(see figure 49b). A small proportion of roundish rectangles were also mistakenly recognized as ellipses.

Distinguishing between a circle and an ellipse was expectedly difficult. The recognition rates are satisfactory but recognizing circles suffered particularly from mixing the two shapes. Problems in recognizing ellipses varied more. Much less ellipses were recognized as circles than vice versa. However, ellipses not drawn round enough were often recognized falsely as diamonds. The ratio $A_{ch}^2/(A_{tr}A_{re})$ was closer to typical values of diamonds (see figure 40).

On the one hand, differentiating circles from ellipses can be seen as choosing certain threshold value for width-height ratio, since drawing a perfect circle will always be impossible for human users. On the other hand, one might need to

deduce the user intention using other than geometric information (possibly pen speed, context). Utilizing only the geometric features will not suffice since roughly half of the shapes intended to be circles would most likely be interpreted as ellipses also by a human (see figure 49c).

The recognition rate of diamonds was the worse of all shapes. Diamonds were often recognized as ellipses, rectangles or not at all. Recognition as rectangle resulted often from the fact that the input sketch resembled a square very closely (see figure 49d). The shapes intended to be diamonds were actually merely squares that were rotated 45°. The shapes misrecognized as ellipses resembled quadrilaterals that had angles that deviated only slightly from right angles. Shapes not recognized at all usually did not resemble diamonds as their sides were not of equal length.

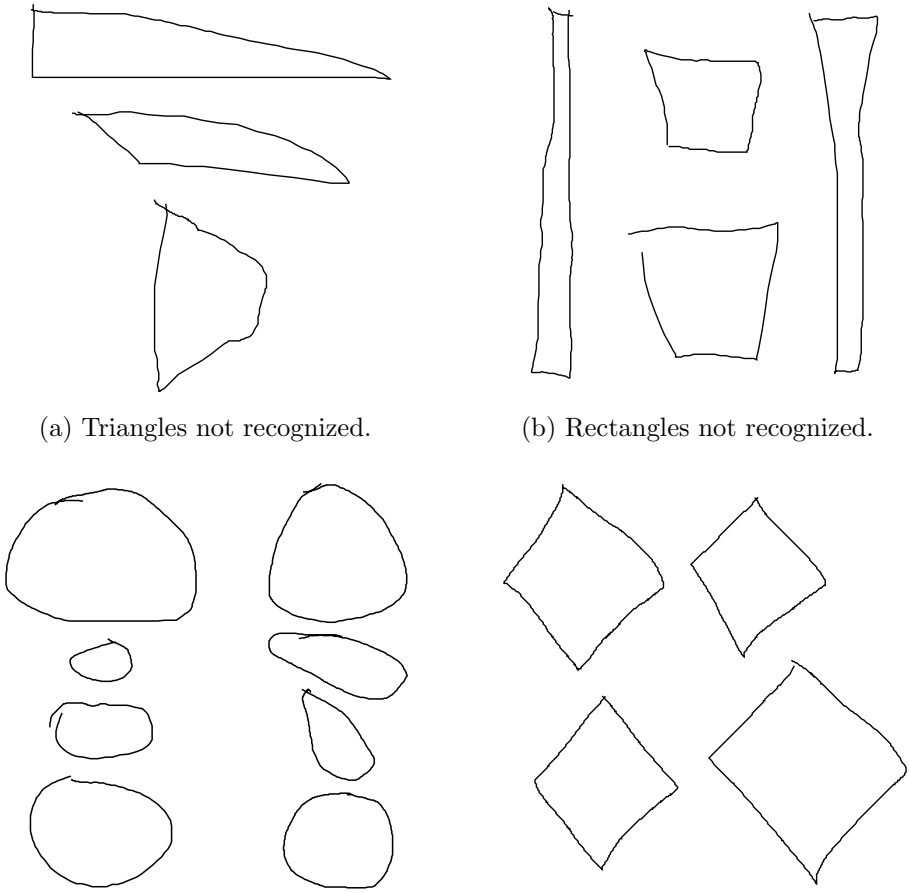
Of the shapes that used additional heuristics for the recognition process, arrows were recognized better than stars. Of the misrecognized shapes, some were drawn in a way that left the last point of the first stroke outside the convex hull of the second stroke (see figure 44). By scaling the convex hull one can slightly improve the recognition rate but this also makes false positives more likely. For other arrows the restriction, that confined the length of the second stroke to be a proportion of the first one at maximum, was violated. The violation occurred especially for arrows with short shafts that resulted in relatively long strokes for the arrow heads (see figure 50a). To improve the recognition one could take into account the absolute length of the arrow shaft: for very short arrows the length of the arrow head stroke can easily be larger than the length of the shaft stroke.

The heuristics for recognizing stars lacked robustness. Almost all of the failed shapes had 6 intersections inside the scaled convex hull (see figure 45). The additional intersection was either the closing intersection of the stroke or one resulting from a circlet in one of the spikes of the star (see figure 50b). Changing the scaling factor of the convex hull cannot be used to improve the recognition rate since making the convex hull smaller also excludes some of the "correct" intersection points. Possibly a better approach would be to analyze each intersection point. One notes that the intersecting line segments are relatively far away from each other, if a distance along the stroke path is used (the smaller of the two distances can be used). The distance is much smaller in the case of intersections resulting from closing the stroke or a circlet in a spike of the star.

The test setting measured only the sensitivity of the system. In normal use the input is likely to contain scribbles that are not intended to represent any of the shapes recognized by the system. In those cases, the number of possible false positives affects the usability of the system. Consequently, the approach should be tested in real life usage scenarios to assess its specificity.

Using a similar approach Fonseca and Jorge achieved a considerably higher recognition rate (95.8%) [59]. Their system had a larger set of predefined shapes (excluding *star*) and a more versatile set of geometric features. There were similarities in the results: Line, rectangle, triangle and circle were the easiest for their system to recognize. Furthermore, diamonds were the most difficult and were also often recognized as rectangles.

Besides the more thorough use of geometric features two differences might explain



(a) Triangles not recognized.

(b) Rectangles not recognized.

(c) (Intended) circles recognized as ellipses on the left and (intended) ellipses incorrectly recognized (as circles or other shapes) on the right.

(d) (Intended) diamonds recognized as rectangles.

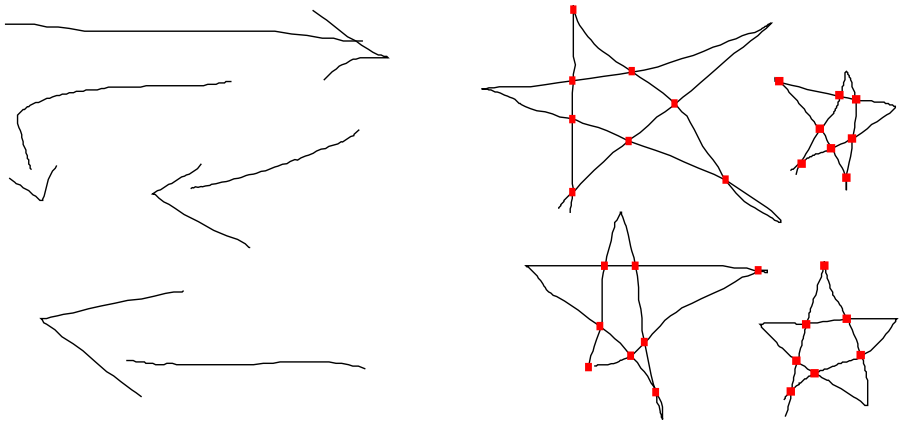
Figure 49

the gap in recognition accuracies between the two systems. First, the subjects of Fonseca and Jorge's experiment used a digitizing tablet instead of a computer mouse. Hence, they were provided with a more natural equipment for drawing. Additionally, two of the nine subjects were experienced in using the equipment. Second, the subjects of my experiment were specifically asked not to draw too carefully to imitate a normal sketching scenario.

6.4 Computational performance

The tests were executed on PC (AMD Athlon^{TM} 64 Processor 3000+, 1GB of memory). The used hardware represented a relatively fast PC in 2005. In today's standards it is clearly outdated.

The computational performance of the system was analyzed by recording the times between the start and end of different steps in the recognition process. This



(a) Arrows not recognized.

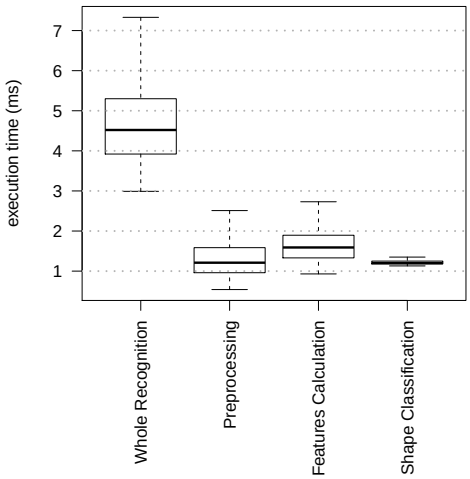
(b) Stars not recognized. Red squares indicate the stroke intersection points.

Figure 50

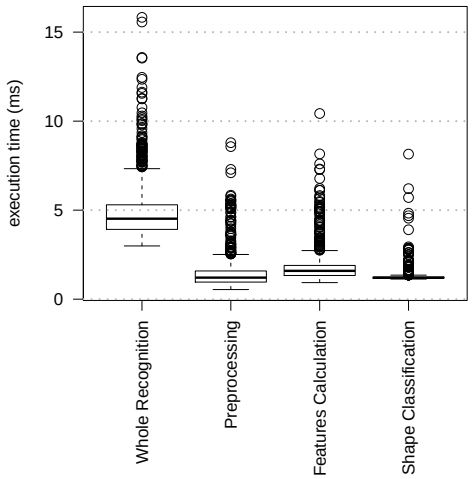
introduces some variation resulting from other computational load on the test system but illustrates best the performance in practice. The recognition process was divided into preprocessing, feature calculation and shape classification. Preprocessing was further divided into polyline approximation (Douglas-Peucker filter), stroke ends refinement and stroke end closing (division depicted in figure 32). Feature calculation was further divided into individual feature calculations: convex hull, minimum-area enclosing rectangle, maximum-area enclosed triangle, intersections, bounding box, centroid (center of gravity), center (of bounds).

Figures 51 and 52 summarize the performance results. One set of figures exclude the outliers to better illustrate the most common values and relative differences between the measures. The time taken to execute the whole recognition process is of most importance to the user. The median execution time of below $10ms$ is more than adequate for good user experience. In usability engineering, a system response time below $100ms$ is considered an instant response [108, 109]. Hence also the maximum time taken for recognition (less than $20ms$) can be considered very good. The results indicate that the algorithms used in the system have a well-defined behaviour also in the worst case. Performance of this level also enables the use of extensive feedback to the user. The recognition process or parts of it can be executed while the user is drawing in order to guide the user.

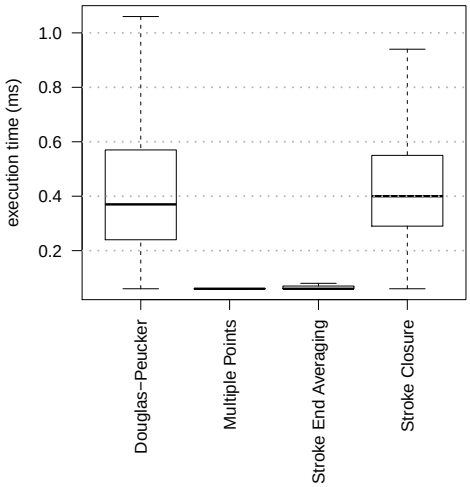
Of the three steps (preprocessing, features calculation, shape classification) preprocessing depends most on the number of input points. More specifically it is the Douglas-Peucker algorithm that has to handle all the input points whereas the subsequent steps have considerably less points to analyze. The average reduction of points in Douglas-Peucker algorithm was 66%. Figure 51c illustrates how the more complex algorithms (Douglas-Peucker $O(n^2)$ and stroke closure $O(n \log n + k \log n)$) are also more expensive in practice. As can be seen in figure 52 the execution times are very small for calculating different features as well as shape probabilities.



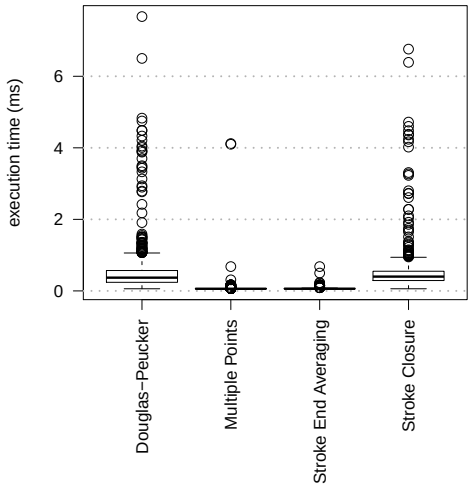
(a) Execution times for different steps of the recognition process.



(b) Execution times for different steps of the recognition process including outliers.

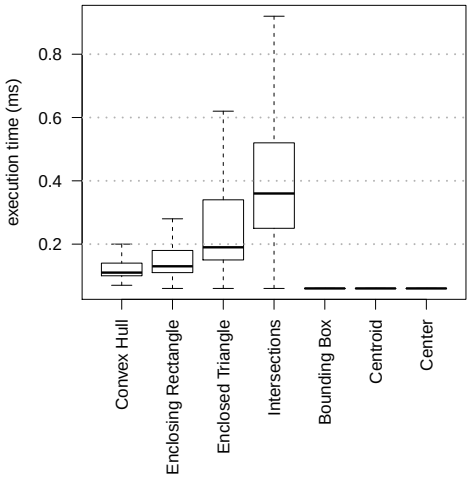


(c) Execution times for the preprocessing steps. Stroke closure is the most time consuming step since it has to calculate the intersections of the input stroke(s).

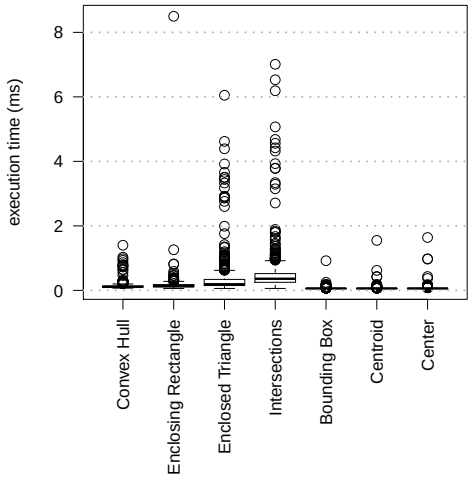


(d) Execution times for the preprocessing steps including outliers.

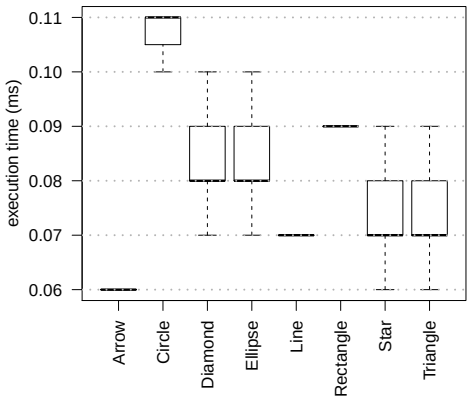
Figure 51



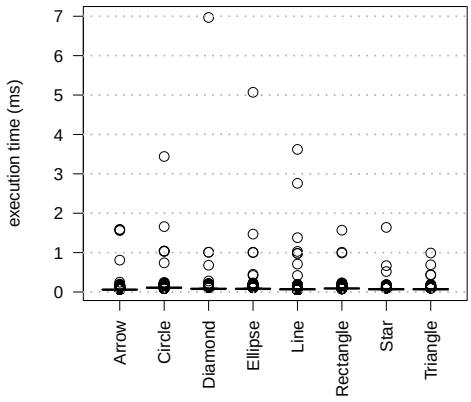
(a) Execution times for the feature calculations.



(b) Execution times for the feature calculations including outliers.



(c) Execution times for shape probability calculations.



(d) Execution times for shape probability calculations including outliers.

Figure 52

7 Conclusions

This thesis presented in detail a set of geometry algorithms for preprocessing and analyzing polylines drawn with a computer mouse. The algorithms were used to obtain a representative set of features for hand-drawn scribbles. The features were further utilized to classify the scribbles into predefined shape classes to facilitate more convenient graphics input. A graphical user interface was constructed around the algorithms to enable realistic testing of the implemented sketch recognition software. A group of subjects was gathered to obtain a sufficient amount of variable scribble data in order to test the real life accuracy and performance of the approach.

The test results on the recognition accuracy were ambiguous. The overall recognition rate was 78% but the performance varied considerably between different shapes and subjects. The simple geometric approach was fairly accurate in recognizing ellipses, lines, rectangles and triangles. Circles and especially diamonds were more difficult to recognize.

The thesis also presented method for recognizing arrows and stars. The method used simple rules that utilize the geometric features of the input strokes. The recognition accuracy for the two shapes was adequate, at most. Improvement of the accuracy would require more robust heuristics to handle the different drawing styles and the imperfections in the user-drawn strokes.

The most severe restriction of the approach is its poor extensibility. Some level of hierarchicality would have to be introduced to make the system applicable to more complex domains. The current implementation requires a lot of manual work and high level of expertise when adding new shapes. For easier extensibility some sort of description language would have to be used to encode additional shapes as well as the spatial relations between the shapes. Furthermore, utilizing the restrictions of different application domains requires embedding semantic information into the higher levels of the recognition process.

The applications for online sketch recognition include simple, gesture based user interfaces and more complex graphics inputting and editing software for domains such as electronic circuit design. The restrictions of the presented approach as such make it cumbersome to apply to the more complicated tasks. However, the presented system is highly applicable to simple gesture or shape recognition. They do not require recognition of highly sophisticated symbols but control gestures that are drawn quickly and typically with a single stroke. Extending the set of geometric features calculated for the input would be likely to provide a robust framework for recognizing an adequate number of gestures (or shapes) for applications such as web browsers.

The applicability of the approach to gesture recognition is supported also by the performance results. One of the goals of the thesis was to assess the real life performance of the algorithms used. The recognition process was very fast with all inputs, even without the newest and most powerful hardware. That is, choosing theoretically efficient algorithms also resulted in very good computational performance in practice. The overall performance enables the approach to be used also in hand-held devices with a touchscreen interface and without a more convenient

input method.

Further investigation could be done to assess whether using machine learning methods improves the recognition accuracy. The naive Bayes classifier, neural networks or k-nearest neighbors are possible alternatives to the fuzzy sets used. In addition, a comparison could be made between different input equipment and the effect of practice. The difference in recognition accuracy is left unknown between using a computer mouse, a digitizing tablet and a touchscreen device (with a finger as the pointing device).

Also, unveiling the usability implications of the system requires further work. The mere technical evaluation of recognition accuracy does not give an insight to the question, whether the recognition system achieves its ultimate goal. That is, whether the sketch recognition actually makes the user interface more convenient to use compared to the traditional approach. Assessing the usability aspects would require improvements to the user interface as well as a different and, particularly, a more realistic test setting, interviewing the subjects and monitoring their behaviour.

References

[1] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 395–410, New York, NY, USA, 1968. ACM.

[2] Stanford MouseSsite. Web document. Updated 2005. Cited February 2011. `http://sloan.stanford.edu/MouseSite/`.

[3] M. R. Davis and T. O. Ellis. The RAND tablet: a man-machine graphical communication device. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part I*, AFIPS '64 (Fall, part I), pages 325–331, New York, NY, USA, 1964. ACM.

[4] C. Machover. A brief, personal history of computer graphics. *Computer*, 11:38–45, 1978.

[5] C. C. Tappert, C. Y. Suen, and T. Wakahara. The state of the art in online handwriting recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12:787–808, August 1990.

[6] Réjean Plamondon and Sargur N. Srihari. On-line and off-line handwriting recognition: A comprehensive survey. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22:63–84, January 2000.

[7] Liu Wenyin. On-line graphics recognition: State-of-the-art. In *GREC 2003: 5th IAPR International Workshop on Graphics Recognition, 2003*, pages 291–304. Springer, 2003.

[8] Jo Twist. Law that has driven digital life. *BBC News* web document. Updated April 2005. Cited February 2011. `http://news.bbc.co.uk/2/hi/science/nature/4449711.stm`.

[9] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[10] G. E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.

[11] Ira Greenberg. *Processing: Creative Coding and Computational Art (Foundation)*. Friends of ED, 1st edition, May 2007.

[12] C. Y. Suen, M. Berthod, and S. Mori. Automatic recognition of handprinted characters – the state of the art. *Proceedings of the IEEE*, 68(4):469–487, April 1980.

[13] N. Arica and F. T. Yarman-Vural. An overview of character recognition focused on off-line handwriting. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 31(2):216–233, May 2001.

[14] V. K. Govindan and A. P. Shivaprasad. Character recognition – A review. *Pattern Recognition*, 23(7):671–683, 1990.

[15] Shunji Mori, Kazuhiko Yamamoto, and Michio Yasuda. Research on machine recognition of handprinted characters. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-6(4):386–405, July 1984.

[16] Alessandro Vinciarelli. A survey on Off-Line Cursive Word Recognition. *Pattern Recognition*, 35(07):1433–1446, 2002. IDIAP-RR 00-43.

[17] Google Books, web site. Cited February 2011. `http://books.google.com/intl/en/googlebooks/about.html`.

[18] Project Gutenberg, web site. Updated January 2011, Cited February 2011. `http://www.gutenberg.org`.

[19] Mark Milian. Project gutenberg on quest to digitize 1 billion books. *Los Angeles Times*, web document. Updated August 2010. Cited February 2011. `http://latimesblogs.latimes.com/technology/2010/08/project-gutenberg.html`.

[20] William Stallings. Approaches to Chinese character recognition. *Pattern Recognition*, 8(2):87–98, 1976. Pattern Recognition Society Monographs.

[21] Josep Lladós, Ernest Valveny, Gemma Sánchez, and Enric Martí. Symbol recognition: Current advances and perspectives. In *Selected Papers from the Fourth International Workshop on Graphics Recognition Algorithms and Applications*, GREC '01, pages 104–127, London, UK, 2002. Springer-Verlag.

[22] U. Garain and B. B. Chaudhuri. Recognition of online handwritten mathematical expressions. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 34(6):2366–2376, December 2004.

[23] Mehmet Sezgin and Bülent Sankur. Survey over image thresholding techniques and quantitative performance evaluation. *Journal of Electronic Imaging*, 13(1):146–168, 2004.

[24] D. H. Rao and P. P. Panduranga. A survey on image enhancement techniques: Classical spatial filter, neural network, cellular neural network, and fuzzy filter. In *Industrial Technology, 2006. ICIT 2006. IEEE International Conference on*, pages 2821–2826, December 2006.

[25] Mukesh C. Motwani, Mukesh C. Gadiya, and Rakhi C. Motwani. Survey of image denoising technique. In Wizard V. Oz and Mihalis Yannakakis, editors, *Proc. Global signal processing expo and conference, September 27, Santa Clara, CA*, September 2004.

[26] L. Lam, S.-W. Lee, and C. Y. Suen. Thinning methodologies – a comprehensive survey. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 14(9):869–885, September 1992.

[27] CellWriter, web site. Cited March 2011. http://risujin.org/cellwriter/.

[28] Michael Levin. Cellwriter: Grid-entry handwriting recognition, Dec 2007.

[29] Aiptek home page, web site. Cited May 2011. http://www.aiptek.com.

[30] Genius home page, web site. Cited May 2011. http://www.geniusnet.com.

[31] Wacom home page, web site. Cited May 2011. http://www.wacom.eu.

[32] Tracy Hammond and Randall Davis. Tahuti: a geometrical sketch recognition system for UML class diagrams. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[33] Jin Xiangyu, Liu Wenyin, Sun Jianyong, and Zhengxing Sun. On-line graphics recognition. In *Computer Graphics and Applications, 2002. Proceedings. 10th Pacific Conference on*, pages 256–264, 2002.

[34] Bo Yu and Shijie Cai. A domain-independent system for sketch recognition. In *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, GRAPHITE '03, pages 141–146, New York, NY, USA, 2003. ACM.

[35] Chris Calhoun, Thomas F. Stahovich, Tolga Kurtoglu, and Levent Burak Kara. Recognizing multi-stroke symbols. In *2002 AAAI Spring Symposium, Sketch Understanding*, pages 15–23. AAAI Press, 2002.

[36] J. A. Landay and B. A. Myers. Sketching interfaces: toward more human interface design. *Computer*, 34(3):56–64, March 2001.

[37] Christine Alvarado and Randall Davis. SketchREAD: a multi-domain sketch recognition engine. In *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, pages 23–32, New York, NY, USA, 2004. ACM.

[38] Brandon Paulson and Tracy Hammond. PaleoSketch: accurate primitive sketch recognition and beautification. In *Proceedings of the 13th international conference on Intelligent user interfaces*, IUI '08, pages 1–10, New York, NY, USA, 2008. ACM.

[39] Tevfik Metin Sezgin and Randall Davis. HMM-based efficient sketch recognition. In *Proceedings of the 10th international conference on Intelligent user interfaces*, IUI '05, pages 281–283, New York, NY, USA, 2005. ACM. [Extended version available.].

[40] Liu Wenyin, Xiangyu Jin, and Zhengxing Sun. Sketch-based user interface for inputting graphic objects on small screen devices. In Dorothea Blostein and Young-Bin Kwon, editors, *Graphics Recognition Algorithms and Applications*, volume 2390 of *Lecture Notes in Computer Science*, pages 67–80. Springer Berlin / Heidelberg, 2002.

[41] Tracy Hammond and Randall Davis. LADDER: a language to describe drawing, display, and editing in sketch recognition. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 461–467, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[42] Tevfik Metin Sezgin. Feature point detection and curve approximation for early processing of free-hand sketches. Master's thesis, Massachusetts Institute of Technology, May 2001. Department of EECS, MIT.

[43] James Herold and Thomas F. Stahovich. SpeedSeg: A technique for segmenting pen strokes using pen speed. *Computers & Graphics*, 35(2):250–264, 2011. Virtual Reality in Brazil; Visual Computing in Biology and Medicine; Semantic 3D media and content; Cultural Heritage.

[44] Alessandro Vinciarelli and Michael Perrone. Combining online and offline handwriting recognition. *Document Analysis and Recognition, International Conference on*, 2:844, 2003.

[45] R. Seiler, M. Schenkel, and F. Eggimannn. Off-line cursive handwriting recognition compared with on-line recognition. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 4, pages 505–509, August 1996.

[46] S. Manke, M. Finke, and A. Waibel. Combining bitmaps with dynamic writing information for on-line handwriting recognition. In *Pattern Recognition, 1994. Vol. 2 - Conference B: Computer Vision Image Processing., Proceedings of the 12th IAPR International. Conference on*, volume 2, pages 596–598, October 1994.

[47] H. Tanaka, K. Nakajima, K. Ishigaki, K. Akiyama, and M. Nakagawa. Hybrid pen-input character recognition system based on integration of online-offline recognition. In *Document Analysis and Recognition, 1999. ICDAR '99. Proceedings of the Fifth International Conference on*, pages 209–212, September 1999.

[48] P. M. Lallican, C. Viard-gaudin, and S. Knerr. From off-line to on-line handwriting recognition. In *Proceedings of the Seventh International Workshop on Frontiers in Handwriting Recognition*, pages 303–312, 2000.

[49] Cheng-Lin Liu, Fei Yin, Da-Han Wang, and Qiu-Feng Wang. Chinese handwriting recognition contest 2010. In *Pattern Recognition (CCPR), 2010 Chinese Conference on*, pages 1–5, October 2010.

[50] C. L. Philip Chen and Sen Xie. Freehand drawing system using a fuzzy logic concept. *Computer-Aided Design*, 28(2):77–89, 1996.

[51] Liu Wenyin, Wenjie Qian, Rong Xiao, and Xiangyu Jin. Smart sketchpad — an on-line graphics recognition system. In *Proceedings of the Sixth International*

Conference on Document Analysis and Recognition, Washington, DC, USA, 2001. IEEE Computer Society.

[52] Theo Pavlidis and Christopher J. Van Wyk. An automatic beautifier for drawings and illustrations. *SIGGRAPH Comput. Graph.*, 19:225–234, July 1985.

[53] S. Revankar and B. Yegnanarayana. Machine recognition and correction of freehand geometric line sketches. In *Systems, Man, and Cybernetics, 1991. 'Decision Aiding for Complex Systems, Conference Proceedings., 1991 IEEE International Conference on*, volume 1, pages 87–92, October 1991.

[54] Takeo Igarashi, Satoshi Matsuoka, Sachiko Kawachiya, and Hidehiko Tanaka. Interactive beautification: a technique for rapid geometric design. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, UIST '97, pages 105–114, New York, NY, USA, 1997. ACM.

[55] Tolga Kurtoglu and Thomas F. Stahovich. Interpreting schematic sketches using physical reasoning. Technical report, Mechanical Engineering Department, Pittsburgh, Pennsylvania, USA, 2002.

[56] Dean Rubine. Specifying gestures by example. *SIGGRAPH Comput. Graph.*, 25:329–337, July 1991.

[57] Joaquim A. Jorge and Manuel J. Fonseca. A simple approach to recognise geometric shapes interactively. In *Selected Papers from the Third International Workshop on Graphics Recognition, Recent Advances*, GREC '99, pages 266–276, London, UK, 2000. Springer-Verlag.

[58] Manuel J. Fonseca and Joaquim A. Jorge. Using fuzzy logic to recognize geometric shapes interactively. In *Fuzzy Systems, 2000. FUZZ IEEE 2000. The Ninth IEEE International Conference on*, volume 1, pages 291–296, May 2000.

[59] Manuel J. Fonseca and Joaquim A. Jorge. Experimental evaluation of an online scribble recognizer. *Pattern Recognition Letters*, 22(12):1311–1319, 2001. Selected Papers from the 11th Portuguese Conference on Pattern Recognition - RECPAD2000.

[60] Manuel J. Fonseca, César Pimentel, and Joaquim A. Jorge. CALI: An online scribble recognizer for calligraphic interfaces. In *Sketch Understanding, Papers from the 2002 AAAI Spring Symposium*, pages 51–58, 2002.

[61] César F. Pimentel, Manuel J. Fonseca, and Joaquim A. Jorge. Experimental evaluation of a trainable scribble recognizer for calligraphic interfaces. In *Selected Papers from the Fourth International Workshop on Graphics Recognition Algorithms and Applications*, GREC '01, pages 81–91, London, UK, 2002. Springer-Verlag.

[62] *Recognizing Hand Gestures with CALI.* The Eurographics Association and Blackwell Publishing, 2006.

[63] Anabela Caetano, Neri Goulart, Manuel J. Fonseca, and Joaquim A. Jorge. Sketching user interfaces with visual patterns. In *Proceedings of the First Ibero-American Symposium in Computer Graphics (SIACG02)*, pages 271–279, 2002.

[64] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, October 1973.

[65] P. S. Heckbert and M. Garland. Survey of Polygonal Surface Simplification Algorithms. Technical report, Carnegie Mellon University, Pittsburgh, PA 15213, 1997.

[66] Alexander Kolesnikov. *Efficient Algorithms for Vectorization and Polygonal Approximation.* PhD dissertation, University of Joensuu, Computer Science, 2003.

[67] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972.

[68] Marcelo Rosensaft. A method for removing roughness on digitized lines. *Computers & Geosciences*, 21(7):841–849, 1995.

[69] John Hershberger and Jack Snoeyink. Speeding up the Douglas-Peucker line-simplification algorithm. Technical report, DEC Systems Research Center, Vancouver, BC, Canada, Canada, 1992.

[70] Juan-Carlos Perez and Enrique Vidal. Optimum polygonal approximation of digitized curves. *Pattern Recognition Letters*, 15(8):743–750, 1994.

[71] Jia-Guu Leu and Limin Chen. Polygonal approximation of 2-D shapes through boundary merging. *Pattern Recognition Letters*, 7(4):231–238, 1988.

[72] Laurence Boxer, Chun-Shi Chang, Russ Miller, and Andrew Rau-Chaplin. Polygonal approximation by boundary reduction. *Pattern Recognition Letters*, 14(2):111 – 119, 1993.

[73] Jack Sklansky and Victor Gonzalez. Fast polygonal approximation of digitized curves. *Pattern Recognition*, 12(5):327–331, 1980.

[74] L. P. Cordella and G. Dettori. An O(N) algorithm for polygonal approximation. *Pattern Recognition Letters*, 3(2):93–97, 1985.

[75] Bimal Kr. Ray and Kumar S. Ray. A new approach to polygonal approximation. *Pattern Recognition Letters*, 12(4):229–234, 1991.

[76] Bimal Kr. Ray and Kumar S. Ray. A non-parametric sequential method for polygonal approximation of digital curves. *Pattern Recognition Letters*, 15(2):161–167, 1994.

[77] Shinn-Ying Ho and Yeong-Chinq Chen. An efficient evolutionary algorithm for accurate polygonal approximation. *Pattern Recognition*, 34(12):2305–2317, 2001.

[78] Heng Li, Hongfeng Shao, Jing Cai, and Xinyu Wang. Hierarchical primitive shape classification based on cascade feature point detection for early processing of on-line sketch recognition. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–397–V2–400, April 2010.

[79] B. Q. Huang, Y. B. Zhang, and M.-T. Kechadi. Preprocessing techniques for online handwriting recognition. In *Intelligent Systems Design and Applications, 2007. ISDA 2007. Seventh International Conference on*, pages 793–800, October 2007.

[80] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, April 2008.

[81] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[82] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[83] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979.

[84] Andrew Chi-Chih Yao. A lower bound to finding convex hulls. *J. ACM*, 28:780–787, October 1981.

[85] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, 2(1):18–21, 1973.

[86] William F. Eddy. A new convex hull algorithm for planar sets. *ACM Trans. Math. Softw.*, 3:398–403, December 1977.

[87] David G. Kirkpatrick and Raimund Seidel. The ultimate planar convex hull algorithm. *SIAM J. Comput.*, 15:287–299, February 1986.

[88] H. Freeman and R. Shapira. Determining the minimum-area encasing rectangle for an arbitrary closed curve. *Commun. ACM*, 18:409–413, July 1975.

[89] F. C. A. Groen, P. W. Verbeek, N. de Jong, and J. W. Klumper. The smallest box around a package. *Pattern Recognition*, 14(1-6):173–178, 1981. 1980 Conference on Pattern Recognition.

[90] Godfried Toussaint. Solving geometric problems with the rotating calipers. In *Proceedings of IEEE MELECON '83, Athens, Greece, May*, 1983.

[91] M. I. Shamos. *Computational Geometry*. PhD dissertation, Yale University, May 1978.

[92] Hormoz Pirzadeh. Computational geometry with the rotating calipers. Master's thesis, McGill University, School of Computer Science, November 1999.

[93] Dennis S. Arnon and John P. Gieselmann. A linear time algorithm for the minimum area rectangle enclosing a convex polygon. Technical Report 463, Purdue University, Computer Science Department, December 1983.

[94] Joseph O´Rourke. Finding minimal enclosing boxes. *International Journal of Computer and Information Sciences*, 14(3):183–199, 1985.

[95] N. A. A. DePano and T. Pham. Minimum enclosing rectangles: a comparative investigation of two optimizing criteria (reprise). In *Southeastcon '91., IEEE Proceedings of*, volume 1, pages 60–64, April 1991.

[96] David P. Dobkin and Lawrence Snyder. On a general method for maximizing and minimizing among certain geometric problems. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 9–17, 1979.

[97] James E. Boyce, David P. Dobkin, Robert L.(Scot) Drysdale, III, and Leo J. Guibas. Finding extremal polygons. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, STOC '82, pages 282–289, New York, NY, USA, 1982. ACM.

[98] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 208–215, October 1976.

[99] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *Computers, IEEE Transactions on*, C-28(9):643–647, September 1979.

[100] David M. Mount. Geometric intersection. In Jacob E. Goodman and Joseph O'Rourke, editors, *The Handbook of Discrete and Computational Geometry, 2nd Edition*, chapter 33, pages 857–876. CRC Press LLC, Boca Raton, FL, 2004.

[101] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In *Foundations of Computer Science, 1988, 29th Annual Symposium on*, pages 590–600, October 1988.

[102] Ivan J. Balaban. An optimal algorithm for finding segments intersections. In *Proceedings of the eleventh annual symposium on Computational geometry*, SCG '95, pages 211–219, New York, NY, USA, 1995. ACM.

[103] K. L. Clarkson. Applications of random sampling in computational geometry, II. In *Proceedings of the fourth annual symposium on Computational geometry*, SCG '88, pages 1–11, New York, NY, USA, 1988. ACM.

[104] K. Mulmuley. A fast planar partition algorithm, II. In *Proceedings of the fifth annual symposium on Computational geometry*, SCG '89, pages 33–43, New York, NY, USA, 1989. ACM.

[105] Robert McGill, John Wilder Tukey, and Wayne A. Larsen. Variations of box plots. *American Statistician*, 32(1):12–16, 1978.

[106] D. L. Massart, J. Smeyers-Verbeke, X. Capron, and Karin Schlesier. Visual presentation of data by means of box plots. *LC-GC Europe*, 2005.

[107] Hans-Jürgen Zimmermann. *Fuzzy Set Theory and its Applications*. Springer, 4th edition, October 2001.

[108] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM.

[109] Jakob Nielsen. *Usability Engineering*. Academic Press, 1993.