漢
字

Kanji Stories

Development of an Application for Learning Kanji Characters

Nils Grabenhorst

Bachelor Thesis

Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin

Supervisor: Prof. Dr. Raúl Rojas

Berlin, January 18, 2012

Abstract

This Bachelor Thesis presents an attempt to develop an application for touch screen devices that helps students of the Japanese language to memorize Japanese Kanji characters. The character drawn by the student on the touch screen will be verified by the application. The typical approach to Japanese character recognition is to use Nearest Neighbor Classifiers, while Hidden Markov Models are used less frequently. Since the purpose of the application under development is to teach the writing of the kanji, the important aspects of the feature vector have to be strictly checked, other aspects of the character shape are not essential. The approach chosen for the recognition algorithm is therefore to explicitly define each aspect to be checked. The data available to the recognizer will also give future versions of the application the opportunity to provide detailed feedback about the particular aspect violated that caused a match rejection.

Contents

1 Introduction

The Japanese language is largely considered very difficult to learn. It is very different from western languages and has some unique grammatical constructions that are unfamiliar. The cultural background the language is embedded within is also quite foreign. The Interagency Language Roundtable of the U.S. government has classified languages according to their difficulty for native english speakers to learn them, ranging from the "Category I languages that closely cognate with English" such as Spanish and French to Category III languages including Chinese and Japanese that are "Exceptionally difficult for native English speakers to learn". According to the U.S. Foreign Service Institute, *"Limited working proficiency"*[1] can be achieved for Category I languages after 575-600 class hours. To achieve the same level of proficiency in Japanese would take approximately 2200 class hours.[JK], [Rub98]

Apart from unfamiliar grammar, vocabulary, idioms, and cultural background, the Japanese writing system poses a significant challenge to any student willing to learn the language. While the "alphabets" Hiragana and Katakana are quite easy to master, the logographic Kanji characters are a formidable obstacle in itself. Thousands of seemingly abstract glyphs have to be memorized, both for writing and reading. Any help breaking down the task into manageable pieces and offering memory aids will be greatly appreciated. By guiding the student through the kanji in an order that teaches basic kanji first, then kanji that are assembled from previously memorized kanji, the task of memorizing them becomes much more manageable than classic methods of learning the glyphs by rote and repetition.

Many kanji look very similar, and for instance once the shape of 未 has been memorized, another glyph will inevitably come up causing great confusion, such as 末. The meanings are very different, the former is *un-, not yet, hitherto, still, even now, sign of the ram*; the latter is *end, close, tip, powder, posterity.* The subtle variance in shape is only noticeable if both glyphs can be compared next to each other: the length of the topmost horizontal stroke relative to the one below is different. Having memorized the first character days or even weeks before encountering the second character,

[1] "Able to satisfy routine social and limited office needs and to read short typewritten or printed straightforward texts."[JK]

the latter will inevitably overpower the memorized first character because the student will unlikely notice the subtle difference in shape and, oblivious to the problem at hand, will likely not develop a learning aid to keep the confusing characters apart. A good teacher will have to point out pitfalls such as these.

The learning technique outlined in [Hei01] deconstructs each kanji into its primitive elements; these are either strokes, or simpler kanji that have been memorized before. These primitive elements are then reassembled by creating a composite ideogram, which is an image or a story tying the primitive elements of the character together. Rather than memorizing the character itself, its ideogram is memorized. The more imaginative, vivid, charming, disgusting or shocking the ideogram is, the more memorable it will be. Recalling the ideogram will lend itself to reconstruct the writing of the associated glyph. This system does not yet take the readings into account, but associating each kanji with its meaning is a large step forward in achieving Japanese literacy.

Each time a more complex character is studied, all of its components are reviewed as well, further solidifying previously acquired knowledge. Practicing to write each glyph while thinking about its ideogram will solidify the connection between the two. Many students use flash cards for studying kanji using variations of the Leitner system, a basic spaced repetition learning system where several stacks of flash cards are used to keep track of the proficiency. Stacks with new material is reviewed more often than those that represent the better known material. After a card has been reviewed successfully it can proceed to the next stack, the ones the student has difficulties with are returned to the first stack. Each card has to progress through each stack at least once. While this works very well if done meticulously, sloppyness may cause problems such as memorizing the wrong stroke order or diminished efficiency due to not drawing each glyph upon review. Merely visualizing a glyph is by far inferior to actually drawing it.

Much research has gone into finding a good timing for spacing the intervals of reviewing information in order to maximize memorization efficiency. [JA08]

Flash cards can be simulated as a computer program. The benefits are obvious, since the computer can serve out the cards accurately, preventing cheating. Progress can be automatically kept track of even for each individ-

ual character so that characters the student has shown difficulty with may be served more often for review than others. Characters that have not been reviewed for a long time may be inserted higher up in the review queue. Many people nowadays have a computer they take anywhere they go: their smartphone. If there is a kanji studying application installed, idle times during the day can be used conveniently to quickly review a few kanji.

On the iOS platform, numerous kanji learning applications exist. Some are direct implementations of flashcards systems[2], others offer little games asking the student to select one of the shown glyphs matching the supplied meaning[3]. Some applications allow the student to write characters by tracing them on-screen or drawing them on an empty screen first, then comparing them to the correct character[4]. One application lets the student trace a character, then the program checks whether the stroke order was correct[5].

As of December 2011, no application seems to exist on iOS that offers a spaced repetition system-based kanji study application using the touch screen for kanji drawing with character recognition. The aim of this bachelor thesis is to study the feasibility of developing such an application and to implement a prototype. The emphasis will be on implementing a character recognition algorithm which will serve two purposes, one technical and one psychological. The technical advantage is that the results of the character recognition can be applied to determine the spacing and frequency of the character review, presenting characters that the student previously had dif-

[2]
- 'StickyStudy: Japanese' by Justin Nightingale
- 'Kanji' by Lima Sky
- 'Kanji Flip' by Proffitt Ink

[3]
- 'Kanji Pop' by Lima Sky
- 'JLTP Study' by Mathias Navne

[4]
- 'Kanji LS Touch' by Jan Bogner
- 'Remembering the Kanji' by Mirai LLP

[5]
- 'iKanji touch' by ThinkMac Software

ficulties with earlier and more often. On the other hand trying to draw a character with almost instant feedback will likely be more enjoyable than a static flashcard-based system.

The scope of the application is as follows:

1. The application presents an area for drawing kanji characters using a finger on a capacitive touch screen.

2. The application asks the student to draw a specific kanji by displaying the keyword meaning of the kanji.

3. If the kanji is presented to the student for the first time, a different view is shown, displaying an animation of the writing of the glyph. If the kanji is composed of other kanji, the keyword meanings for these are also shown. The student will have the opportunity to enter an ideogram that is helpful in memorizing the kanji.

4. After the student has drawn the glyph, the software checks if the input is correct. Each kanji object stores a *level* value. The number of the level value correlates to the number of times the kanji has been drawn successfully in a row. If the input is correct, the kanji gets promoted by incrementing the level value, otherwise it will be set back to 1.

5. If the student does not know how to write the kanji, he or she may press a *"peek"* button to see the same view as in 3 above. Upon doing so, the level of the character is set back to 1.

6. Characters of levels 0 to 5 are *active* and presented to the student. Characters of level 0 are new and will be introduced as described in 3. Characters of level 6 are considered *"known"* and are currently not presented.

7. If the number of active characters drops under a threshold, new ones are taken into the active set. New characters will never contain components that have not been introduced before. Additionally a few of the known characters are randomly chosen and added to the active set by setting their levels to 5. This will ensure they are reviewed from time to time. Instead of choosing them randomly, more elaborate ways of selecting them may be introduced later, such as the longest time

since the last review, or the least favorable ratio of successful to failed reviews.

2 The Japanese Writing System

2.1 Kanji, Hiragana, Katakana, Rōmaji

The Japanese writing system uses four distinct sets of characters: Kanji, Hiragana, Katakana and Rōmaji. The following sentence[6] is a simple example of how each set of characters is used for a specific purpose:

kin yō bi mi ra sa n to de Tō kyō e i ki ma shi ta
金曜日ミラさんとJRで東京へ行きました。

This sentence means, "I went to Tokyo by train with Mr. Miller.":

kin yō bi	mi ra	sa n	to	de	Tō kyō e	i ki ma shi ta
金曜日	ミラ	さん	と	JRで	東京へ	行きました。
Friday	Miller	Mr.	with	JR by	Tokyo to	went

Japanese writing rarely makes use of spaces to separate words, however the use of different character sets helps breaking sentences apart.

Rōmaji

Rōmaji are letters of the Roman alphabet. They are used to transliterate Japanese to target non-Japanese readers. Occasionally Rōmaji is used in conjunction with the other Japanese character, typically for acronyms and company or brand names as is the case here for *JR*, the Japanese railway company.

kin yō bi mi ra sa n to de Tō kyō e i ki ma shi ta
金曜日ミラさんとJRで東京へ行きました。

Kana

Kana are two sets of syllabic characters, *Katakana* and *Hiragana*, containing 48 basic characters each. While both have been derived from Kanji, none of these characters carry any *meaning*. Instead, each character is a

[6] The small type above each Japanese glyph is a transliteration to Roman writing. Sometimes similar transliterations to hiragana glyphs can be found above rare and obscure kanji. This reading aid is known as *Furigana* or *Ruby*.

phonetic code not unlike the characters of the Roman alphabet. Being *syllabic* however, each character roughly equates a phonetic syllable, with the occasional second or third character used to modify the pronunciation of the syllable. The correlation between sound and symbol is in fact even closer than between letters and sounds of the English or German language. [Got05, p.81]

Katakana

Katakana are most commonly used for loan words and foreign names. Other uses are for onomatopoeia[7] or for adding emphasis not unlike the use of italics in western writing. In the sample sentence, katakana is used for writing an english surname, *Miller*. Since the amount of available syllables in Japanese is considerably lower than in European languages, perfect transliterations rarely exist as can be observed in this example.

kin yō bi mi ra sa n to de Tō kyō e i ki ma shi ta
金曜日ミラさんとJRで東京へ行きました。

Hiragana

Particles and inflections are written using hiragana. Hiragana are phonetically equivalent to katakana. Occasionally nouns and the stems of adverbs, adjectives and verbs of Japanese origin are also written using hiragana, either if no kanji exists, or if the characters are rare and unknown.

kin yō bi mi ra sa n to de Tō kyō e i ki ma shi ta
金曜日ミラさんとJRで東京へ行きました。

Kanji

Kanji are logographic characters. Each kanji is composed of a number of strokes and denotes *meanings* or *concepts* rather than a phonetic property. Sometimes one character correlates to one word, but they are often combined with one or even more other kanji to form a compound word. The kanji system was borrowed from China in the 6th century and the characters have largely remain unchanged, however there are a few rare cases of char-

[7] Words imitating sounds

acters unique to Japan (Kokuji). While many thousands of kanji exist[8], a much smaller subset is in common use. The Japanese Ministry of Education has assembled a list of 2,136 *jōyō kanji*[9] for regular use. The jōyō kanji are the ones that Japanese students have to learn in elementary, junior high and high school. The 1,006 characters taught in elementary school comprise about 90 percent of the characters found in newspapers. [Got05, p.82]

kin yō bi mi ra sa n to de Tō kyō e i ki ma shi ta
金曜日ミラさんとJRで東京へ行きました。

2.2 Readings

Kun'yomi and On'yomi

Each Kanji character typically has several readings, of which *on'yomi* are the readings borrowed from the Chinese language; *kun'yomi* are the Japanese readings. The reading to be chosen for a particular instance of a character is to be determined by the reader taking into account the group of kanji and kana that form the kanji compound for a word. Context may also play a role.

2.3 The Anatomy of Kanji Glyphs

2.3.1 Strokes

Each kanji glyph is composed of strokes that can be further broken down to basic stroke shapes and more complex strokes. The basic stroke shapes are:

丶 **Dot** A very short dash

一 **Horizontal** A stroke that is written from left to right

丨 **Vertical** A stroke that is written downward


Rise Written left to right, rising up


Press Down Written left to right, falling

[8] The kanji dictionary *Daikanwajiten* lists more than 50,000 kanji characters.[Wik]
[9] as of 2010

㇒ **Throw Away** Written right to left, falling.

More complex strokes are combinations of basic stroke shapes, using four ways to combine them:

㇄ **Break** A sharp turn, most often at a right angle. The direction is either to the right or down. The stroke to the left is composed of three basic strokes with two breaks.

㇃ **Hook** Similar to the break, this is a sharp turn. The direction is either to the left or down. The hook shown is attached to a vertical stroke, therefore it points to the left.

㇇ **Bend** A stroke section typically curving counter-clockwise to the left or clockwise to the right.

㇌ **Slant** A stroke section typically curving clockwise to the left or counter-clockwise to the right.

Fig. 1 lists the basic and compound strokes used for CJK[10] characters. Even though the number of characters is very large, they are composed of a very limited number of stroke shapes. The strokes are conceptually very regularly shaped and distinctive.

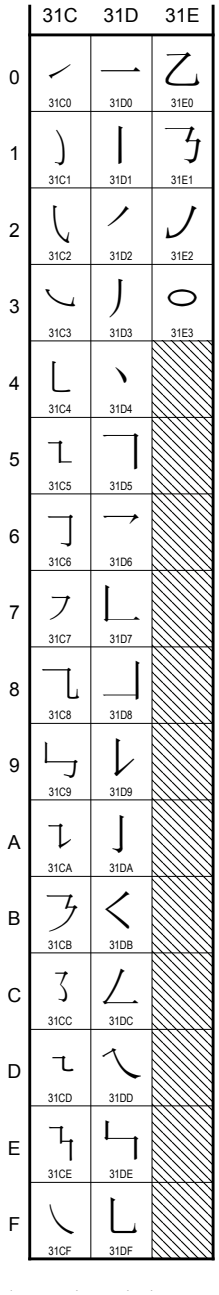

Figure 1: *CJK Strokes. 31E3 is not used in modern Japanese kanji; 31CB and 31CC are counted as two strokes each. [uni10]*

2.3.2 Glyph Components

Each kanji is composed of a number of strokes. Each stroke is in a well defined location relative to other strokes. The strokes are written in a well-defined order, even if writing them in a different order would result in a glyph of the same shape. Similarly, horizontal strokes are always written left-to-right, vertical strokes downward. Combined

[10] CJK is short for Chinese, Japanese and Korean.

strokes such as angled strokes or hooked strokes have to be written in one go. One might argue that not adhering to these rules may result in a glyph that looks exactly the same. However, complying with these rules not only ensures efficient handwriting; the stroke order rules also increase legibility in hand-written characters. Experienced writers will not always fully raise the pen or brush from the paper when moving from the end of one stroke to the beginning of the next one, hence generating an extraneous line or curves connecting the two (connective strokes). If the writer would not observe the stroke order, these extraneous curves would appear in a configuration that is unexpected to the reader, therefore obscuring the character written.

In order to briefly introduce an example for the composition of kanji characters, the character for *mouth* is 口 and written using three strokes: The first stroke is the vertical one on the left-hand side. After that, an angled stroke starting horizontally at the top left, then proceeding vertically to the bottom right corner is drawn. The horizontal stroke on the bottom comes last. In order to memorize this character, one can view this as a pictograph of a *mouth* wide open.

As another example, consider the character for *to say*, 言. Since strokes are usually written in the order left-to-right, top-to-bottom, the first four strokes can be written intuitively beginning at the top, working downwards. The bottom part is actually a character that has been memorized before: 口. This can be used to a great effect in helping to memorize many characters, since complex characters are composed of simpler characters. Here we can literally see four sound waves rising out of a talking mouth.

Characters often appear as glyph components in more than one other character. 語 is the character for *word*, composed of three components: 言, 五, and 口. If one has learned these characters before, a *"story"* can be attached to the character and its meaning, stringing the components together. The more memorable the story is, the easier it is to memorize meaning and character. The stroke order of the compound character is almost always keeping the stroke order of the components intact. Compared to less structured learning approaches, the effort to learn a large number of characters can be significantly reduced by studying them in an order that takes the fact that most kanji are composed of simpler components into account. By attaching memorable stories to each character, the retention rate can be further improved. [Hei01]

3 Recognizing Characters

3.1 A Brief Overview of Recognition Strategies

Today, two dominant approaches exist for on-line handwritten character recognition. For languages based on the roman alphabet or similar alphabets, most recognition engines use a Hidden Markov Model for each character to be recognized. Recognition engines for Japanese characters predominantly use nearest neighbor classifiers. These calculate the distance of the input data to all of the reference data sets stored in a database. The shorter the distance is, the more likely a match has been found. The distance to the reference data, or template, is determined using a set of features, the *feature vector*. Geometrical aspects of data such as length, angle, curvature may be extracted from the input data in order to obtain a feature vector. A popular feature is the directional feature, where local directional information is extracted from the data for the four major directions: horizontal, vertical, and the two diagonals. The area of the training data is then divided into a regular pattern of regions such as squares. For each region and each each major direction, the number of directional features in that region is counted. The result is a histogram of training data that can be compared to the histogram of input data.[JLN]

Other features include the *directional segment strength feature*, the *stroke count feature*, as well as feature points of interest including intersections, endpoints and points of maximum curvature, among others. [Hil93]

Elastic matching may be used to further improve the recognition rate. Elastic matching is an optimization problem in trying to best match the input data to the template data by warping the input data in both dimensions. This process allows to compute a distance that is invariant to deformation. [US05]

Using Hidden Markov Models (HMMs) for recognizing kanji characters is possible and has been done successfully, but the sheer amount of characters poses a challenge in that for each character an HMM has to be generated and supplied with sufficient training data. The feasibility of HMMs improves if each kanji character is considered a 'word' composed of a number of strokes that act as 'characters'. The number of different stroke shapes is very manageable. The strategy is to have a HMM for each possible stroke shape instead of one HMM out of thousands for each character. For typical

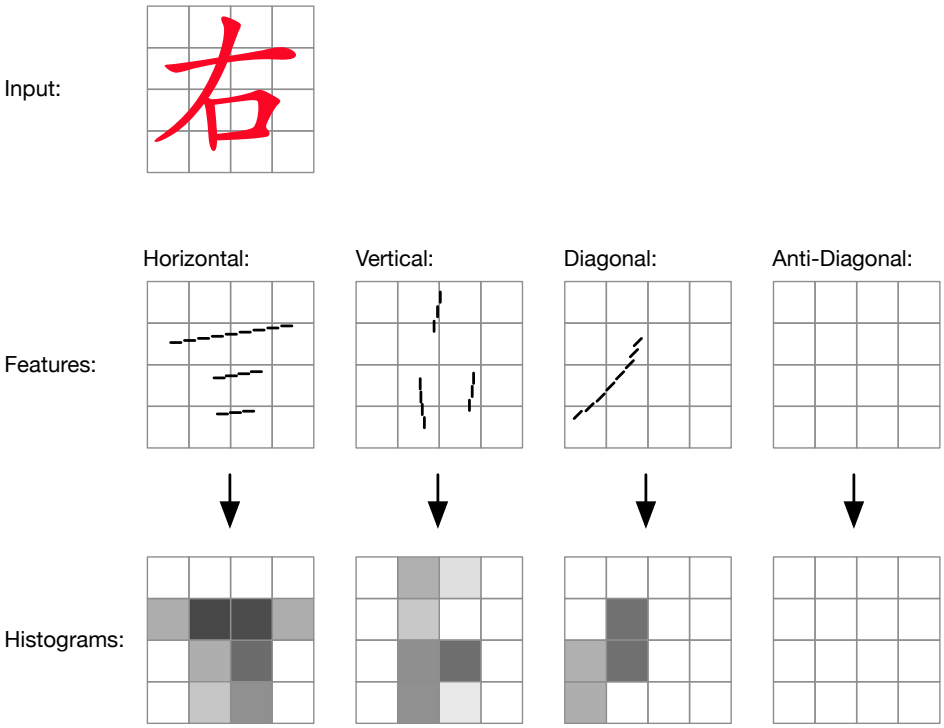

Figure 2: *Directional Features are often used in kanji recognizers. The directional features are the converted into a vector of histograms that can be compared against. Normalizing during preprocessing has to take care of ensuring correct positioning and proportioning of the input data.*

applications, challenges are the variance of stroke order and stroke count in japanese input that is much higher than the variance in letter order and letter count in western writing[11]. Stroke count and stroke order are an important feature in the scope of this thesis, since the purpose of the application is to teach correct writing. [JLN]

The approach chosen for the recognizer is not to have a machine learning approach. Instead, the feature data is entered explicitly. This makes it possible to ensure that a particular feature that is very subtle and may be missed by a learning algorithm due to in-class variance being considerably larger than the between-class variance. The fact that stroke shapes are very well defined raises the hope that manually constructing recognizers for each glyph is feasible, if laborious.

The option of manually constructing HMM parameters was dismissed because the states of an HMM cannot be inspected by design. Instead, a state machine was developed that allows inspection in order to find out the reason for a stroke not matching. This information is currently logged to the console, but may be used in the future to provide more fine-grained feedback

[11] A variance in letter order or letter count is a misspelling. A variance in stroke order or count in a kanji may result a perfectly legible glyph shape not discernible from a kanji written correctly.

to the student as for the reason *why* the character drawn is wrong[12].

Neither handwriting nor calligraphy is a scope of this project, hence strokes are to be strictly separate from each other, simplifying the recognition problem significantly.

3.2 Data Available

In typical OCR applications the data available to the recognition algorithm is limited to a raster image of color values, usually black and white pixels. Other applications have access to on-line data recorded by a touchscreen, graphic tablet or similar device, simplifying the preprocessing significantly and also adding temporal data to the input. This project has access to on-line data due to touchscreen input method processing a series of touch events. Touch events are received in the order they are drawn and can easily be attributed to a certain stroke, since each stroke consists of exactly one touch-down, exactly one touch-up and optionally one or more touch events in between.

The goal of the recognizing algorithm is not to recognize a character, in fact the algorithm already *knows* the character the student is supposed to draw. Instead, the algorithm has to check the character drawn by the student against the template character and decide whether they match or not.

The data available is therefore,

1. The geometry of each stroke, given in the order drawn, and
2. The defined properties of the template character.

3.3 Variance

Recognizing Kanji characters can be challenging due to the fact that two distinct characters may have a small between-class variance, but the within-class variance between the same character handwritten twice or characters written in different fonts can be very high. Compare the entirely different

[12] Feedback may include:

- The angle of a section of a stroke is wrong.
- A hook is missing on a particular stroke.
- A certain point on a stroke should align with another stroke.

characters 未 and 未, yet 未 and 未 are both the same character in different fonts. It will be essential for the recognizer to be aware of subtle differences such as these since it is the goal to teach the student to take care about such important aspects when writing kanji. In the above example, if the student is asked to write 未, yet the top horizontal stroke written extends beyond either side of the horizontal stroke below, the recognizer must detect this and report that the glyph does not match.

3.4 Avoiding Frustration

The recognition algorithm needs to be tuned to avoid causing frustration. Each time a character is recognized as incorrect is a very jarring experience for the user. An input that was correct but nonetheless causing the algorithm to return a wrong verdict will be a significant cause of frustration, particularly since the user does not understand why the verdict was wrong and what part of the input triggered the error. The user will lose faith in the accuracy of the application, even if the accuracy is in fact quite good. Therefore the accuracy needs to be as close to 100% as possible for *correct* input.

Incorrect input has to be detected in most cases as well in order for the application to be useful. However, for incorrect input it is not quite as important to achieve an extremely high success rate of recognizing a character as being wrong, since the application can give the user feedback by other means, such as displaying the correct character alongside the character drawn by the user. This is also important because it presents the student an opportunity to develop a sense for the aesthetics of his or her handwriting.

The algorithm has to check for the correctness of the most important aspects of a character: stroke count, stroke order, stroke shapes and the location of each stroke relative to other strokes.

3.5 Character Composition

Each character is defined as a tree of glyph components. Each component can either be a stroke or another tree of glyph components. This data structure facilitates reuse of existing characters as components within other, more complicated characters. Each component can be addressed by its component number within that character. Since a component can be located in

different positions of different characters, each component is wrapped into an instance of a numberedComponent, which ensures the proper ordering of components within a character.

The tree describing a character contains the information necessary for recognizing the drawn glyph. Each glyph component within the tree may have one or more instances of recognizer objects attached. If the glyph component is a leaf and therefore a stroke, it must have at least one recognizer which is the *stroke shape recognizer*[13]. One or more *location recognizers* can be added to any glyph component, whether it is a stroke component or not.

The application selects a character from the database of template characters. The component tree is copied, but the strokes are omitted. The result is a component tree with empty positions for strokes at its leaves. When the student draws a stroke, a new stroke object is generated with the polyline generated from the touch events. This stroke is then inserted at the next available position in the component tree, which is the position of the stroke expected to be drawn at this point in time. If the student draws not enough strokes, the tree will have remaining empty positions. If more strokes than the character's stroke count are drawn, these excess strokes will be inserted as children of the root, numbered as if they were valid strokes.

Once the student asks the program to check the drawn glyph, the number of the strokes drawn is evaluated. If the stroke number does not match the stroke count of the template character, the algorithm terminates with a failed match. Otherwise the component tree is traversed. Any recognizers attached to a component are evaluated. All recognizers have to match in order for the drawn glyph to be considered correct.

[13]Stroke shape recognizers inherit from the abstract superclass DHLShapeRecognizer_Super.

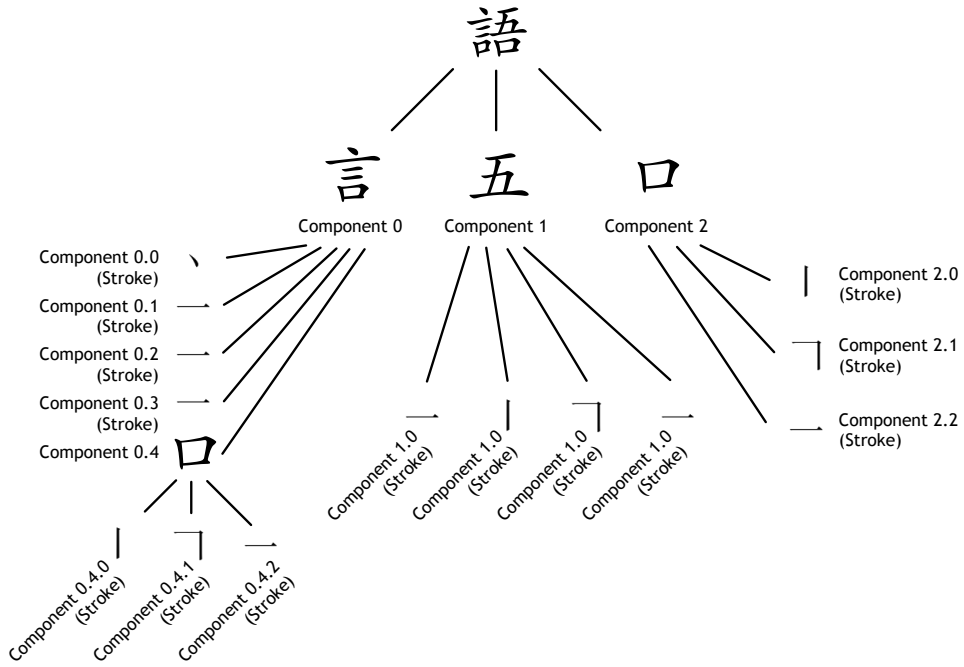

Figure 3: *Character composition. The simplified object graph for the character* 'word' *is shown. Note that* component 2 *and* component 0.4 *are the same instance of* 口*, hence all of their subcomponents are also the same.*

4 Stroke Shapes

A polyline is a piecewise linear curve given by an array of vertices defining the endpoints of the polyline's edges. The stroke polyline is generated by taking the filtered array of touch event coordinates from the touch input between two consecutive touchDown/touchUp events. The filtering[14] just blocks any touch events that are too close to the last registered touch event, and also removes touch events that would result in a colinear sequence of vertices. The stroke polyline therefore represents the stroke as drawn by the student, albeit slightly smoothed due to the filtering.

Stroke shapes are the first important feature used in the recognizing algorithm. The two aspects of the shape tested for are a sequence of angles as well as optionally a sequence of curvatures.

The simplest strokes are straight strokes such as Horizontals, Verticals and Rises. Each one of these is defined by just one angle for the main direction. Other strokes are combinations of simpler strokes, with each one of the stroke sections adding an additional angle to be recognized. Some strokes have one or more sections that are curved either in a clockwise or counter-clockwise direction. The stroke shape feature therefore consists of a series of one or more directions given by their angles. Optionally one or more curvatures are specified. If a stroke section is not curved, it will not

[14]The filtering is performed in the DHLStandardTouchesFilter.h class.

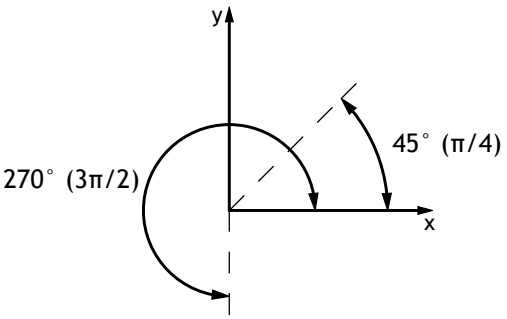

Figure 4: *Angles in coordinate space.*

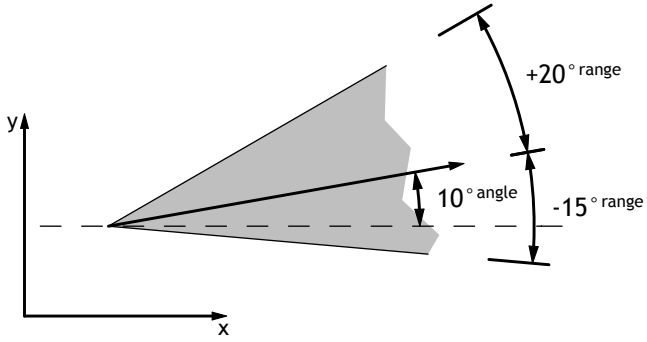

Figure 5: *An angle object represents an angle relative to the x-axis as well as upper and lower bounds for deviation from that angle. The angle shown is 10°, but accepted angles may range from -5°to 30°.*

be verified that the section is strictly straight, since it is very difficult to write a straight stroke. In fact, handwritten straight strokes have a slight curve in many cases. Very exaggerated curves on non-curved sections will be rejected by the recognizer when checking the angle of the section.

4.1 Angles, Curvatures

Angles are measured relative to the x-axis of the coordinate system of the draw area, rotating counter-clockwise (Fig. 4)[15]. A horizontal stroke section may be intuitively considered to have an angle of 0°. That kind of precision is not very helpful since it is not humanely possible to draw that precise a stroke without drawing aids. Furthermore, strokes are often drawn slanted for aesthetic reasons. While the character for *mouth* idealistically is a rectangle (口), it is often drawn with slightly slanted strokes ( ). Hence the angle information needs to be modeled allowing for some degree of deviation. The angle class[16] stores the angle as well as rangePlus and rangeMinus values for the upper respectively lower bounds of the deviation allowed (see Fig. 5).

Due to the kanji being composed of predominantly very few sub-strokes, very few standard angles are sufficient to specify the majority of strokes. In order to be able to globally adjust one of these standard angles later when testing shows that the ranges specified are too narrow or too wide to enable reliable recognition, angles can be named rather than be supplied with explicit degree and range values. This name is used to look up the angle and

[15]The coordinate system of iOS views is flipped, i.e. the y-axis points downward. For illustrative purposes a non-flipped coordinate system is shown.

[16]DHLAngle.h

▼ angled stroke	Diction...	(8 items)
▼ 270-0	Diction...	(2 items)
▶ anchor weights	Array	(3 items)
▼ angles	Array	(2 items)
▼ Item 0	Diction...	(3 items)
degrees	Number	270
-	Number	25
+	Number	15
▼ Item 1	Diction...	(3 items)
degrees	Number	0
-	Number	20
+	Number	35
▶ 0-270	Diction...	(2 items)
▶ 0-250	Diction...	(2 items)
▶ 0-225	Diction...	(2 items)
▶ 250-330	Diction...	(3 items)
▶ 270-350	Diction...	(3 items)
▶ 330-60	Diction...	(3 items)
▶ 225-315	Diction...	(3 items)
▶ angled stroke hook	Diction...	(3 items)

Figure 6: *An entry in the XML file specifying the standard angles. Two angles are given for the angled stroke named,* "270-0".

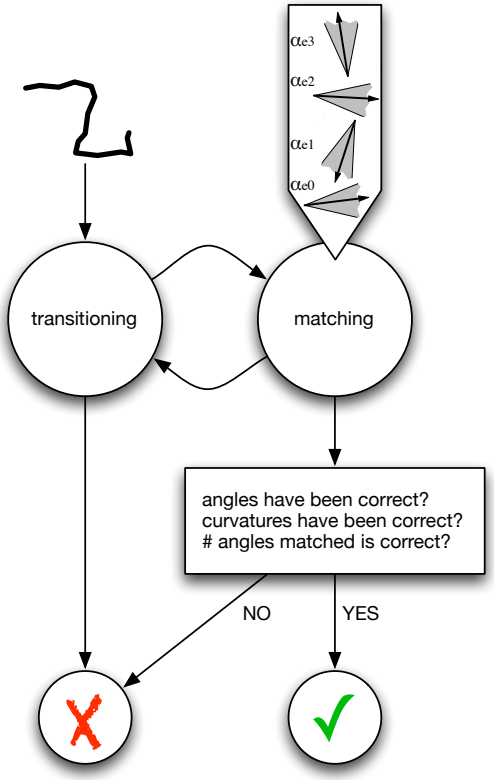

Figure 7: *Stroke shapes are matched by a simple automaton. The input is the polyline drawn by the student. Its edges are traversed and depending wether they match the expected angles α_{e_i} provided, the automaton alternates between the two states.*

range values from an XML file[17]. The angle's name is given hierarchically[18], consisting of a category name such as *"angled stroke"*, a stroke name such as *"270-0"*, and an index giving the specific angle of that stroke (See Fig. 6). Only the numbers given at the "angles"-entry referred to by category name and stroke name have to be modified in order to adjust a defined angle for that specific stroke type without affecting other stroke types. This avoids the need to fine-tune many thousands of angles individually.

4.2 Recognizing Stroke Shapes: Angle Automaton

The recognizing algorithm makes use of a deterministic automaton[19] in order to check the stroke polyline for the presence of the expected angles and curvatures (see Fig. 7). The input consists of the stroke polyline drawn by the student as well as the strokeData stored in the shape recognizer. The stroke

[17] StandardAngles.plist
[18] The class storing the hierarchical angle name is DHLStandardStrokePlistPath.h
[19] DHLAngleAutomaton

data contains the expected angles and the curvature information. The output is a boolean value[20]; `YES` if the stroke shape was determined to match the strokeData, or `NO` if the shape does not match. The automaton consists of two basic states, *inMatchingState* and *transitioningState*. Two methods take care of transitioning between the states, *moveToInMatchingState* and *moveToTransitioningState*.

The automaton iterates through the edges of the stroke polyline drawn by the student. The polyline edges are accessible by an edgeIndex. A counter named numberOfExpectedAnglesMatched keeps track of the number of matched stroke sections. The first step before entering the first state of the automaton is to set up the environment. The indices are set to the first angle and the first edge. No section has been matched yet, so the counter is set to zero. A flag is set to notify the automaton that it is dealing with the beginning of the stroke. Finally the first expected angle is loaded from the stroke data's set of angles[21]:

```
1 edgeIndex = 0;
2 expectedAngleIndex = 0;
3 numberOfExpectedAnglesMatched = 0;
4 isCheckingBeginningOfStroke = YES;
5 expectedAngle = [expectedAngles objectAtIndex:expectedAngleIndex];
```

Next, the first state of the automaton is entered. Since no angle has been examined yet, the first state is the one transitioning to a matching state. The automaton stays in this state until an angle of an edge matches the expected angle. Note that an additional check has to make sure that the beginning of the stroke is not too deformed, as described later.

```
7 [self polylineIsTransitioningToNextMatchingState];
```

Upon leaving the automaton, the result has to be interpreted. The automaton has stored its result in the `verdict` variable, but the automaton does not have the global information necessary to conclusive decide whether the stroke matches. The automaton has to terminate in a matching state, otherwise the end of the stroke cannot have matched the last expected angle.

[20] Objective-C terminology is used. `YES` is equivalent to `true`, `NO` to `false`.

[21] The pseudocode presented here is heavily influenced by Objective-C. An expression within square brackets is a message sent to an object, causing the method specified by the method's *selector* to be called:

```
returnValue = [addressee selectorWithArgument1:someValue argument2:someOtherValue] .
```

```
9  if (!isInMatchingState) {
10     verdict = NO;
11 }
```

The automaton keeps track of the number of matching states it encounters. This number must be equal to the number of sections of the stroke, which is equal to the number of expected angles. This number may optionally be one less if there is an optional section at the end of the template stroke.

```
12 maxNumAnglesToMatch = [expectedAngles count];
13 minNumAnglesToMatch = maxNumAnglesToMatch - numberOptionalExpectedAngles;
14 if (NOT (numberOfExpectedAnglesMatched <= maxNumAnglesToMatch
15         AND numberOfExpectedAnglesMatched >= minNumAnglesToMatch) ) {
16     verdict = NO;
17 }
```

The transitioning state of the automaton iterates through the edges of the drawn polyline until an edge matches the current expected angle α_e. If a matching angle has been found (see line 20), the special case of the beginning of the stroke may have to be handled. Since otherwise the transitioning state would just advance through the edges of the polyline until the first edge matches the expected angle, the student would have the opportunity to draw a stroke that is outside the matching angle for a long distance with the shape recognizer still matching (see Fig. 8). In the current implementation this extra verification takes the number of edges outside the first matching state into account and limits these to two. This has proven to be a good compromise in my testing, but further data from other testers have to verify this. A certain amount of edges have to be allowed to be outside the first matching state because of the lack of precision of touch screen input, especially initially after the touch down event if there is no initial velocity of the finger. Future implementations may take the euclidian distance traveled between the touch down event and the first matching state into account.

Other than checking whether an edge matches α_e, no further validity testing is performed in the transitioning state after the first matching state. This will give the student the opportunity to 'cheat' by drawing any shape as long as α_e is avoided (see Fig. 9). In practice this will not be a big problem since the student will try to match the character to be drawn as close as possible. The transitions between two matching states (between two stroke sections) is always an angle that may be rounded slightly, and any other shape between sections will feel unnatural when drawing. A future implementation might include a test whether the successive edges are tran-

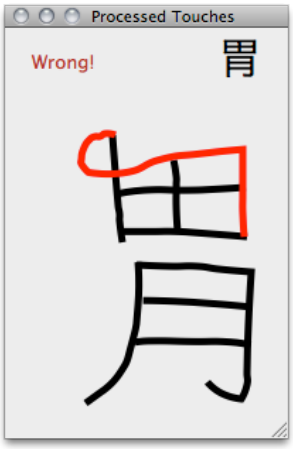

Figure 8: *The beginning of the second stroke is outside the first matching state (roughly horizontal) for a large number of edges. The automaton checks for these cases in order to correctly recognize the stroke as wrong.*

Figure 9: *The student has drawn a loop between the two stroke sections of the second, angled stroke. The recognizer in its current implementation ignores that and reports a matching stroke shape. This problem does not have a very high priority, since this is a very unnatural way to write a kanji. This may be easily addressed in the future.*

sitioning towards α_e in the most direct way. That can be easily achieved by making sure the angles of the edges only rotate in the most direct way towards α_e.

```
18 - (void) polylineIsTransitioningToNextMatchingState {
19     while (edgeIndex < [polyline edgeCount]) {
20         if ([self shouldEnterMatchingState]) {
21             if (isCheckingBeginningOfStroke) {
22                 verdict = [self firstPartOfStrokeIsNotTooFarOff];
23                 isCheckingBeginningOfStroke = NO;
24             }
25             [self moveToInMatchingState];
26             break;
27         }
28         edgeIndex = edgeIndex + 1;
29     }
30 }
```

When entering the matching state, the counter keeping track of the number of sections matched is incremented.

```
31 - (void) moveToInMatchingState {
32     isInMatchingState = YES;
33     numberOfExpectedAnglesMatched++;
34     [self polylineIsInMatchingState];
35 }
```

The matching state is again iterating through the edges of the polyline drawn. Each edge is checked if it matches the current expected angle (line 39). If that is the case, the next edge will be inspected. Otherwise, three

cases may be the cause:

1. The current edge already matches the next expected angle. In this case the moveToTransitioningState method is called, since this method will take care of transferring to the next matching state. See lines 40f.

2. The current angle neither matches the current nor the next α_e. Since glitches due to the inaccuracies of the touchscreen input are possible, especially if the student briefly pauses mid-stroke, a single non-matching edge shorter than a certain threshold value between two matching edges of the current section is considered a *glitch*. This will not cause the matching state to be left. The call to the `maybeGlitch` method on line 44 checks for this condition.

3. Any other case will cause the current matching state to be left by calling `moveToTransitioningState` .

```
36 - (void) polylineIsInMatchingState {
37     while (edgeIndex < edgeCount) {
38         edgeAngle = [polyline angleOfEdgeAtIndex:edgeIndex];
39         if ( NOT expectedAngleMatchesAngle:edgeAngle ) {
40             if ([nextExpectedAngle isApproximatelyEqualToAngle:edgeAngle]) {
41                 moveToTransitioningState;
42                 break;
43             }
44             if (maybeGlitch) {
45                 // ignore a single edge that does not match the current angle:
46                 edgeIndex = edgeIndex + 1;
47                 continue;
48             }
49             moveToTransitioningState;
50             break;
51         }
52         edgeIndex = edgeIndex + 1;
53     }
54     checkCurvatureOfCompletedSection;
55 }
```

Before entering the transitioning state, the last section is analyzed for adherence to the curvature if there is one specified for that section. This test takes the first and the last vertices of the last section of the polyline as well

as a vertex between the two and performs a leftOfTest[22]. If that test does not give the expected result, the `verdict` variable will be set to `NO`.

```
56 - (void) moveToTransitioningState {
57     checkCurvatureOfCompletedSection;
58     expectedAngleIndex = expectedAngleIndex + 1;
59     expectedAngle = [expectedAngles objectAtIndex:expectedAngleIndex];
60     polylineIsTransitioningToNextMatchingState;
61 }
```

Checking Angles of Curved Sections

The expected angle α_e of a curved section is specifying the expected angle of the directed line through the first and the last vertex of that stroke section. When looking at the individual edges of a curved section it is apparent that testing each edge against α_e is problematic. If the section is curved in a clockwise direction, the first edge of the section will be rotated counter-clockwise away from α_e, the last edge clockwise away from α_e (see Fig. 10). A counter-clockwise curve is an analogous case. Those edges may be rotated too much, hence the automaton may not match these edges without rotating α_e according to the expected curvature.

Before α_e is compared to the angle of the first edge e_0, it has to be rotated by the angle γ_0. Let v_0 be the first vertex and v_n be the last vertex of the section. Let $-1.0 \leq x \leq 1.0$ be a value stored in the stroke shape recognizer defining the amount of the curvature, with $x = -1.0$ defining a counter-clockwise half circle, $x = 1.0$ defining a clockwise half circle. Let h be the maximum distance of the expected curve from $\overline{v_0 v_n}$:

$$h = |\overline{v_0 v_n}| \cdot x$$

Let c be the circle given by v_0, v_n and h. γ_0 and γ_3 are the angles between c and $\overline{v_0 v_4}$ at their intersections.

[22] Let $p = (p_x, p_y)$ and $q = (q_x, q_y)$ be points defining a directed line l. Let $r = (r_x, r_y)$ be another point. Let D be the determinant given by

$$D = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}.$$

Then r is to the left of L if $D > 0$, to the right of l if $D < 0$ and on l if $D = 0$. [dBCvKO08]

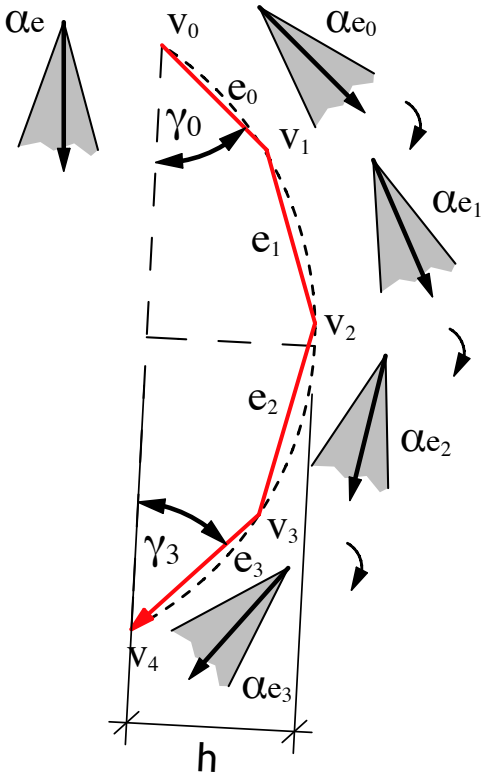

Figure 10: *Rotating the expected angle α_e on a curved section. The edges of the polyline drawn are shown red. Without rotation, edges e_0, e_1 and e_4 would not match. The maximum rotation added to α_e is γ_0. After that, the expected angle is only allowed to rotate in one direction until a maximum rotation given by $\alpha_e - \gamma_3$ is reached. γ_0 and γ_3 are calculated as follows: Let c be the circle given by v_0, v_4 and h. γ_0 and γ_3 are the angles between c and $\overline{v_0 v_4}$ at their intersections.*

When the automaton is in the transitioning state and the next α_e is the angle of a curved section, α_e is rotated by γ_0:

$$\alpha_{e_0} = \alpha_e + \gamma_0$$

For each edge e_i in the stroke section the angle can be rotated until it matches the edge's angle. This rotation is limited to only clockwise rotation if the section is a clockwise curve, or to counter-clockwise rotation if the section is a counter-clockwise curve. The rotation is further limited to a maximum given by

$$\alpha_{e_n} = \alpha_e - \gamma_0.$$

Both constraints ensure that only curved strokes within the bounds given by the original angle α_e are positively matched, while still giving enough flexibility for the curve. Because sections of kanji strokes are clearly separated by angles of typically 90°or more, the next section is clearly detectable even with the last adjusted angle α_{e_n} being closer to the next expected angle than the original α_e.

```
62 - (BOOL) expectedAngleMatchesAngle:angle {
63     if (sectionIsCurved) {
64         setupAngleRotationIfNecessary;
65         rotateExpectedAngleWithinCurvatureToAngle:angle;
66     }
67     return [expectedAngle isApproximatelyEqualToAngle:angle];
68 }
```

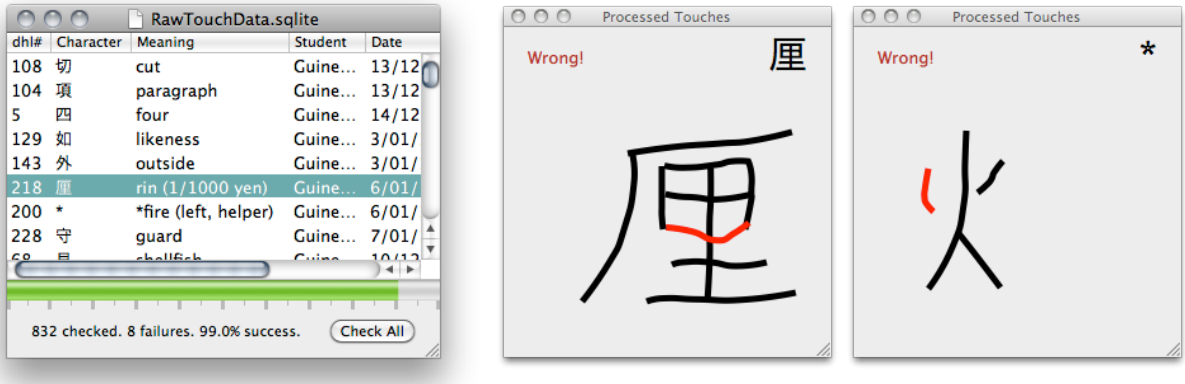

Figure 11: *Two characters determined wrong due to straight strokes not being straight when ANGLE_ACCURACY_STRAIGHT_STROKE_EDGES is set to 25. At a value of 26, both are deemed correct.*

4.3 Straight Strokes

During testing it was shown that it is surprisingly difficult to enter straight strokes on the capacitive touch screen. If the overall angle of a straight stroke is biased towards either side of the expected angle's range, angles of the individual edges within the stroke's polyline are more restricted towards that side and have more room on the other side. For example, let expected angle be $\alpha_e = 0°^{+10°}_{-10°}$ and let the observed overall angle be $\alpha_{overall} = 9°$. In this case only edges that deviate by at most 1° counter-clockwise from $\alpha_{overall}$ are acceptable by the automaton. The maximum deviation on the other side is 19°. The desired behavior would be to check whether $\alpha_{overall}$ matches α_e, then make sure that the edges of the polyline do not deviate too much from $\alpha_{overall}$ of the input stroke.

Therefore straight strokes are handled slightly differently. First, the overall angle $\alpha_{overall}$ of the line through the first and last vertex of the polyline entered is compared to the expected angle α_e stored in the recognizer. If the $\alpha_{overall}$ is wrong, the stroke will be considered incorrect. If the $\alpha_{overall}$ is correct, the rest of the polyline entered has to be checked. The same algorithm as for other strokes is used, however as a new expected angle $\alpha_{e'}$ the $\alpha_{overall}$ is used as a base angle. The $^+/_-$ range is set to a value shown in experiments to result in a reasonably strict criterion. This gives the student a bit more leeway for error within the stroke towards both sides of the angle while still applying the more strict expected angle criterion to the overall angle. An accuracy range value of 20° has been determined to be effective by trying several values on test data[23]. Set to 15, 9 out of 832 checked characters were determined to be incorrect due to straight strokes not being

[23] See ANGLE_ACCURACY_STRAIGHT_STROKE_EDGES in DHLConstants.h.

sufficiently straight, for a value of 20 this dropped to only two characters (see Fig. 11).

4.4 Defining Stroke Shapes

Each stroke that has to be defined explicitly has to be provided with the data needed by the stroke shape recognizer. Necessary is the sequence of angles defining the stroke sections. Optionally, one or more sections may be given a clockwise or counter-clockwise curvature.

In order to streamline the process of the stroke shape definition, the user interface offers a range of basic stroke shapes (see the buttons on the left-hand side and the top right in Fig. 12). Selecting one of these buttons will add a stroke to the character. A subclass of *DHLDefineTouchesFilter_Super* is also instantiated and added as the receiver of touch events generated by the draw area. Once a touch-up event is received, the most appropriate stroke data is selected from StandardAngles.plist according to the angles of the section drawn (see Fig. 15). The touches filter is also responsible for limiting the vertices that can be entered by drawing. A DHLDefineStraightFilter instance will only generate straight strokes having two vertices corresponding to the locations of the touch-down and the touch up event. A DHLDefineMultiAngledFilter analyses the touch events for significant changes in direction of consecutive touch events. Each time a major change of direction is detected, a vertex is generated. This way strokes with multiple sections can be drawn when defining the stroke shape. These are the only two methods used, all other subclasses of *DHLDefineTouchesFilter_Super* employ one of the two methods, but use different sections in the StandardAngles file to look up the stroke data.

For the large majority of strokes this is all that has to be done in order to define the stroke shape. In a few cases individual sections of the stroke can be edited by selecting the stroke angles entry in the table (the first entry when opening the disclosure triangle, compare the entry *Straight Stroke* in Fig. 12). Any value of any angle may be adjusted. The curvature information for any stroke section can be entered here as well.

There are a few cases where a stroke section does not have to be drawn. This is occasionally the case for a hook on a stroke of a kanji that is used as a component inside another kanji. Compare the two-sectioned stroke of the left component of 切 to the stroke at the same location in 切. These

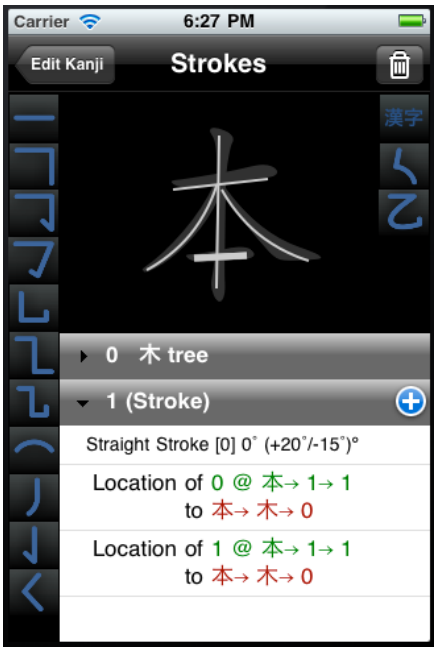

Figure 12: *Strokes are defined by selecting one of the stroke shape buttons, then drawing the stroke on the screen. The stroke information (angles, curvatures) is dependent on which stroke shape button has been pressed as well as by the angles of the stroke drawn.*

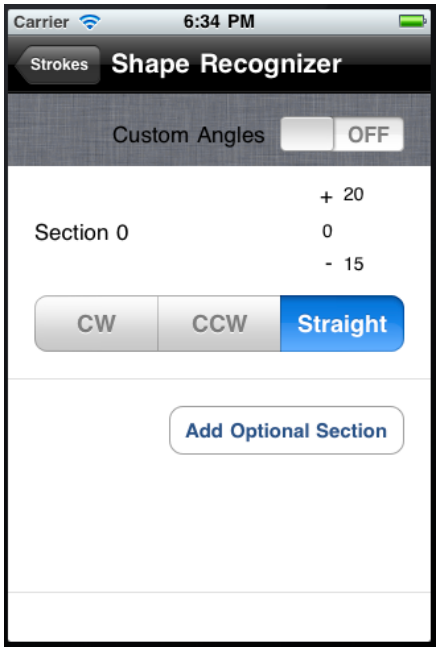

Figure 13: *The stroke sections can be further edited by selecting the stroke angles entry in the table (the first entry when opening the disclosure triangle). This stroke has one straight section with a standard angle of $\alpha_e = 0°^{+20°}_{-15°}$. The angle can be customized by applying different values. The section can be turned into a curved section by selecting one of the CW/CCW buttons. An additional optional section may be added.*

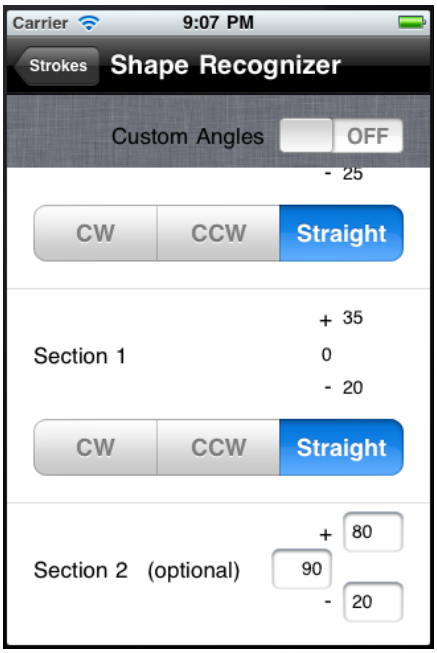

Figure 14: *An optional third section with an angle of $\alpha_e = 90°^{+80°}_{-20°}$ has been specified.*

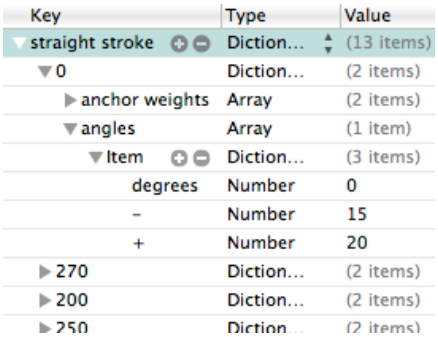

Figure 15: *The standard angle selected when specifying a straight horizontal stroke.*

Figure 16: *Several affine transforms applied to three glyph components based on the same character. The* 漢字*-button in the top left corner adds additional components selected from a list of previously defined kanji.*

Figure 17: *The left element version of* 木 *is defined separately from the character* 木*, since the shape subtly differs when used as a component on the left-hand side of a character:* 林*. Components marked as* "Element Only" *will never be presented to the student as characters on its own.*

are both the same character, yet one stroke may be written either with or without a hook on its end. Even though these are rare cases, they have to be modeled for correct recognition of either writing style. An optional section can be specified as shown in Fig. 14.

4.5 Adding Glyph Components to a Glyph

In order to significantly speed up the process of defining kanji characters, other characters can be inserted as glyph components in addition to the basic strokes. The additional benefit is that even after a character A has been added to another character B as a component, if character A has to be adjusted either due to an error or in order to improve the recognizer data, the improvements will automatically apply to B and all other characters that contain A. The added glyph component is inserted into the top level of the glyph tree at the next available index position. The student will have to write all components in the order of their indices.

The shape of subcomponents can be adjusted by applying an affine transform. The transform is saved with each numbered glyph component, the standard transform is the identity transform *id*. This transform can be

edited by using multitouch gestures. Dragging will add a displacement, pinching along the x- or y axes will add scaling along the respective axis. Skew is supported by rotating two fingers.

These affine transforms only affect the drawing of the template characters on screen, not the way the student writes the character. Since a glyph component often contains subcomponents that have transforms attached themselves, the drawing algorithm has to concatenate all transforms when descending the glyph tree. In Fig. 16, the component 杏 contains the subcomponents 木 and 口. Hence the transform applied to each stroke drawn is a concatenation of the transform applied to 杏 and the transform of the subcomponent applicable.

None of the location recognizers or shape recognizers are altered by these transforms. Even though the angles of some strokes may change significantly when displayed (compare the curved strokes in Fig. 16), these variances are accounted for by the upper and lower bounds of the specified angle. These bounds are very wide in the case of diagonal and counter-diagonal strokes, addressing the problem in almost all cases. The skew transform however also affects the horizontal strokes, whose angle bounds are typically much tighter. Therefore it is advisable to apply the skew transform very carefully. If a larger skew value is needed, a new character should be defined instead, having the desired shapes specified in its strokes in order to avoid the need for an extreme transform. This extra character does not exist in reality as a kanji, hence it can be marked as *"element only"*, causing it to never be presented to the student as an individual character outside its role as a subcomponent (see Fig. 17).

5 Component Locations

Component locations are the second feature analyzed by the recognizing algorithm. Absolute locations are not considered, since absolute positioning is very hard to achieve on a touchscreen without being given any reference such as grids or guide lines. Instead, the location of one component is defined relative to another component. This will also allow the student to vary the position and size of the character.

A *location recognizer* is saved with a glyph component and determines if a specific vertex of a stroke ($stroke_{this}$) is inside an area constrained by

a *target polygon* that is computed by considering a specific stroke section of another stroke ($stroke_{other}$). Therefore the location recognizer needs information encoded that lets it address $stroke_{this}$ and $stroke_{other}$ relative to the glyph component the location recognizer is stored within. For $stroke_{this}$ it needs to store which vertex of the stroke it is addressing; for $stroke_{other}$ a reference to a specific stroke section is needed.

Additionally the location recognizer needs information about how to compute the *target polygon* relative to the section of $stroke_{other}$.

5.0.1 Addressing Strokes

A *location recognizer* is saved with a *numbered glyph component*. A numbered glyph component takes care of storing glyph components inside a glyph tree in the correct order. This component may be a stroke, but also may be a component that contains any number of strokes and glyph components. The reason is that while inside a character, all locations have been well defined. Once this character becomes a subcomponent of another, more complicated character, any of the strokes inside that subcomponent may need another location recognizer in order to specify its location relative to the supercomponent. The $stroke_{this}$ is always a stroke within the same glyph component, while $stroke_{other}$ is always specified relative to the direct supercomponent of the glyph component. Note that the glyph component that stores the location recognizer can be a stroke, since strokes are also glyph components. This is often the case in basic characters that are assembled of strokes only.

A *component path*[24] is a sequence of indices unambiguously specifying a particular glyph component within a glyph. The angled stroke of the bottom left 口 in 語 can be addressed with the component path [0,4,1]; the same stroke inside the bottom right 口 has the component path [2,1] (comp. Fig. 3).

Each location recognizer stores a component path to $stroke_{this}$ and another one to $stroke_{other}$. The path to $stroke_{this}$ is given relative to the component path the location recognizer is located in. The path to $stroke_{other}$ is relative to the supercomponent, which the numbered glyph component has a reference to.

[24] DHLComponentPath.h

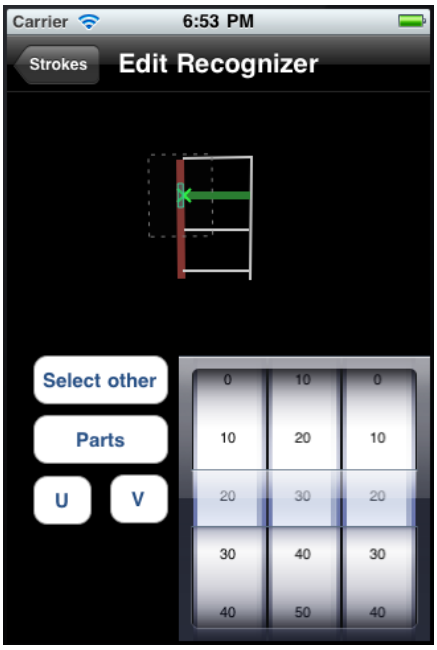

Figure 18: *Defining the location of the first vertex of the green stroke relative to the red stroke. The first vertex has to be inside the area marked green for the location recognizer to match. This area is given relative to the red stroke, in this case it will be* on *the red stroke, 30% down from the top, stretching 20% up and down.*

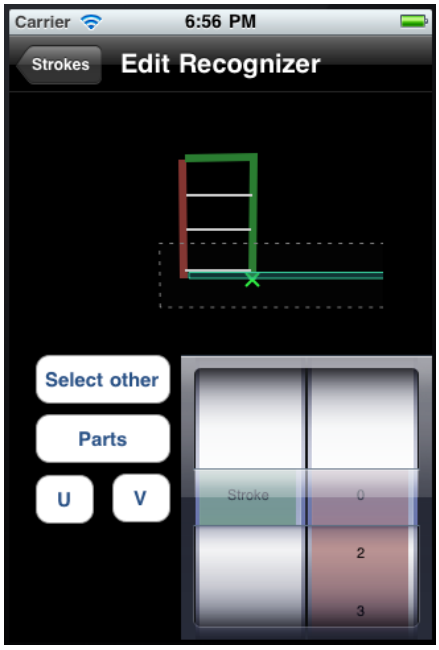

Figure 19: *Another location recognizer defined as follows: The last vertex of the second stroke (green) has to align horizontally to the bottom of the first stroke (red), keeping to the right.*

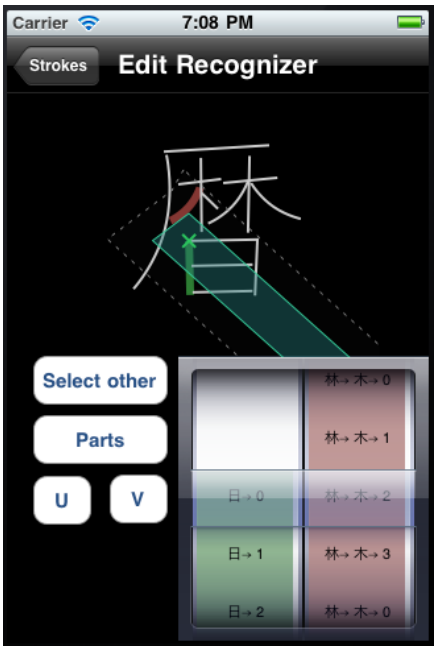

Figure 20: *Another location recognizer defined as follows: The first vertex of the first stroke of component* 日 *has to be below the third stroke of the left-hand side sub-component of component* 林.

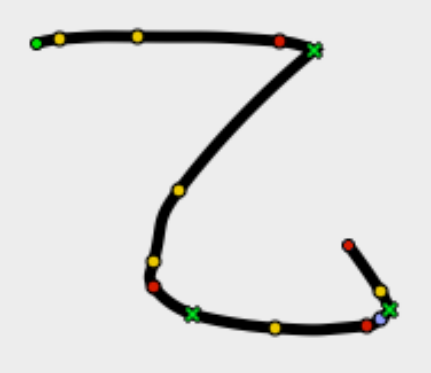

Figure 21:
A stroke with four sections:

1. *Straight horizontal*
2. *Diagonally curved*
3. *Horizontally curved*
4. *Hook up*

The vertices are marked by the angle automaton depending on the current state. Each matching state is delimited by red and/or green vertices with yellow vertices in between.

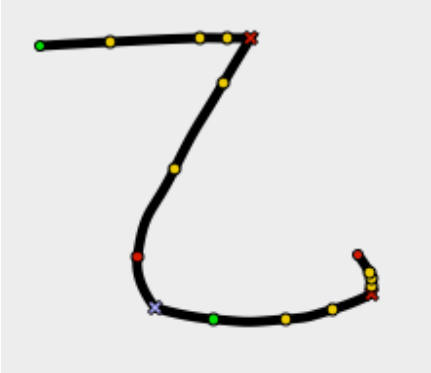

Figure 22:
The vertices in the top right and bottom right corners are vertices shared by edges of two consecutive stroke sections. These vertices are marked red as a representation of an `EXITING_MATCHING_ANGLE_STATE`.

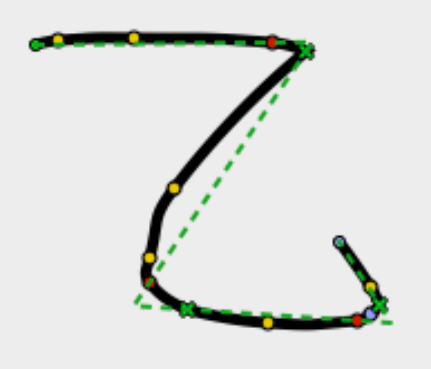

Figure 23: *The cage polyline for the stroke shown in Fig. 21 drawn as a green dashed line.*

Figure 24: *The cage polyline for the stroke shown in Fig. 22 drawn as a green dashed line.*

The vertex within $stroke_{this}$ as well as the section in $stroke_{other}$ is accessed by their indices. See Fig. 20 for an example: The location recognizer specified in component 日 has a reference to the first stroke of component 日 for adressing $stroke_{this}$; $stroke_{other}$ is the third stroke of the left-hand side sub-component 木 of component 林 in supercomponent 曆[25].

[25] 曆: calendar; 林: grove; 木: tree; 日: sun.

5.1 Stroke Sections

The algorithm that computes the *target polygon* relative to a stroke section needs access to that stroke section. A stroke section is a series of consecutive vertices of the stroke polyline drawn by the student. In the case of single-sectioned strokes such as straight strokes or simple curved strokes, the full polyline drawn can be considered to be the stroke section. The challenge is to find the delimiters of a stroke section of a multi-sectioned stroke.

The angle automaton described in section 4.2 generates important information about the sections of the stroke. The automaton marks each vertex with information about what state of the automaton it was related to. The information attached to the vertex may be one of the following states:

- `ENTERING_MATCHING_ANGLE_STATE`
- `IN_MATCHING_ANGLE_STATE`
- `EXITING_MATCHING_ANGLE_STATE`
- `IN_TRANSITIONING_STATE`

These markings make it easy to identify the vertices that certainly belong to a stroke section. Since the matching state of the automaton is entered only if an edge's angle matches the expected angle and the earliest edge exiting is the next one, a section is at least delimited by a pair of vertices marked with entering/exiting states. If an edge in one matching state id directly followed by an edge entering the next matching state without any intermediate edges, the common vertex between the two edges will be marked with an entering state. The algorithm interpreting these states considers an exiting state without previous matching entering state as being a vertex shared by edges in two consecutive stroke sections.

The *cage polyline* of a stroke is an idealized representation of the stroke to be drawn. Let v_0 be the first and v_n be the last vertex of a stroke polyline. Let (v_i^{in}, v_i^{out}) be the pair of vertices delimiting the i^{th} sequence of edges in a matching state. Let l_i be the line through v_i^{in} and v_i^{out}. Let x_i be the intersection between l_{i-1} and l_i. The cage polyline pl_{cage} of a stroke with m sections is given by the sequence of vertices

$$pl_{cage} = (v_0, x_1, \cdots, x_{m-1}, v_n)$$

(see Fig. 25).

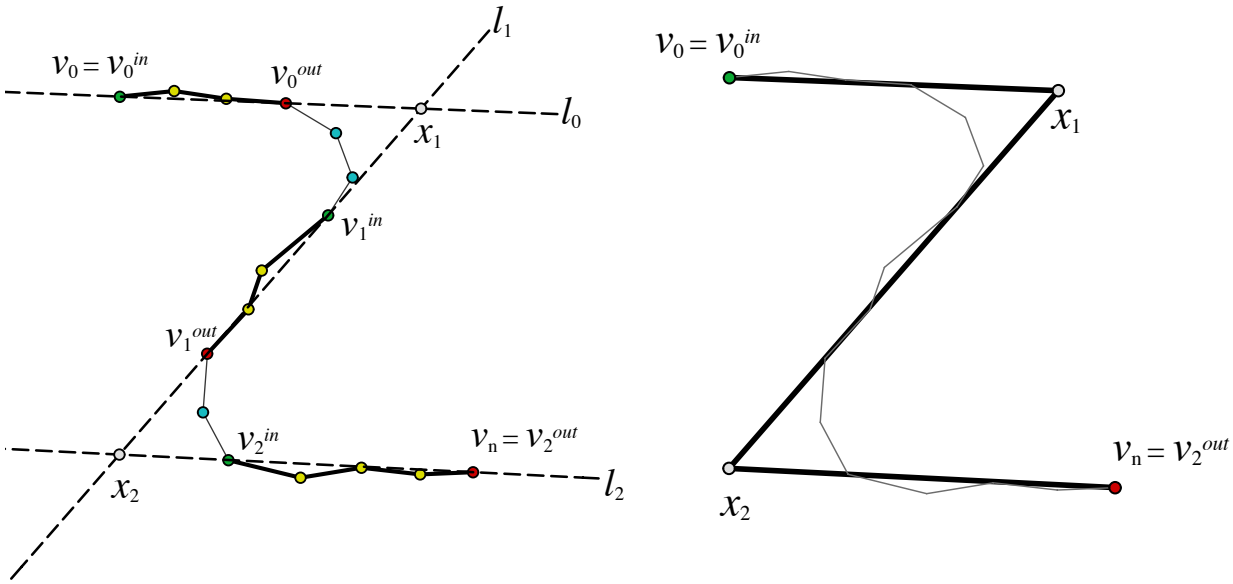

Figure 25: *The lines through the vertices marked as the boundary vertices of the automaton's matching state give the cage polyline. (continued in Fig. 26)*

This cage polyline is used to divide the stroke polyline into its section polylines. The angular bisectors of each two consecutive edges of the cage polyline is computed in order to find the intersections of these bisectors with the original stroke polyline. Let e_i^c be the i^{th} edge of the cage polyline. Let y_i be the intersection of the angular bisector between e_{i-1}^c and e_i^c that is closest to the vertex shared by e_{i-1}^c and e_i^c. The intersections $(y_1, \cdots, y_{m-1})$ divide the stroke polyline into its section polylines (see Fig. 26).

5.2 Target Polygon

Dividing a stroke polygon up into its section polygons is an essential step before recognizing a location. The target location is a polygon that is computed using the section polyline referred to by the location recognizer. Let v_l be the vertex the recognizer is examining the location of.

In addition to a reference to the stroke section of a $stroke_{other}$, the location recognizer stores the following values:

- u**:** The relative location of the target polygon along the length of the stroke section. This value is specified as a percentage of the overall length of the section.
- *plus* (*minus*)**:** The length of the target polygon given as offsets along the length of the stroke section added (subtracted) from the u value. These values are also given as a percentage of the overall length of the section.

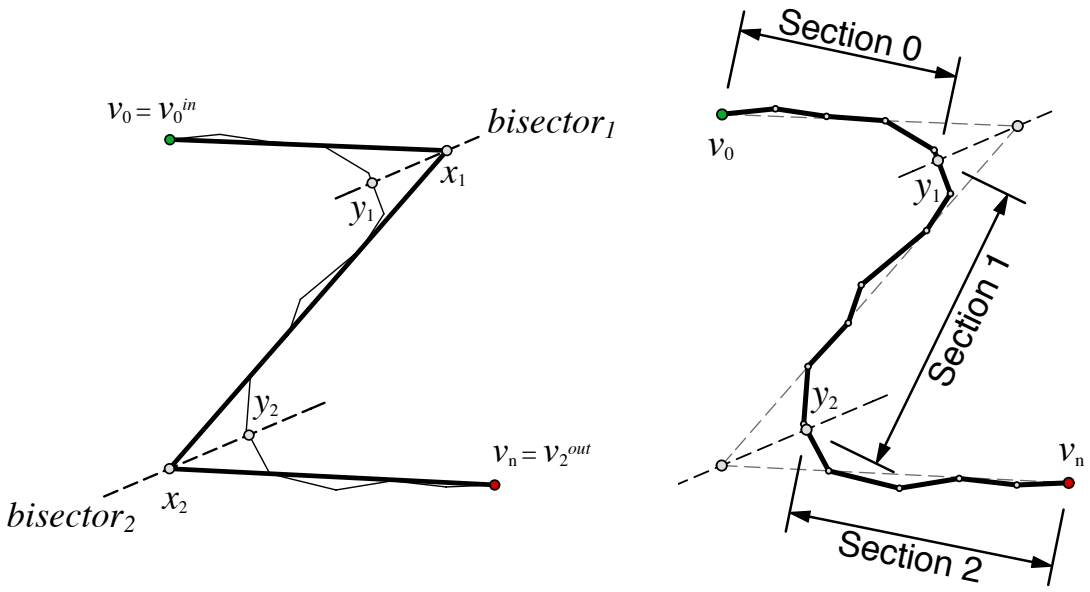

Figure 26: *(continued from Fig. 25) The intersections of the bisectors with the stroke polyline nearest to the cage polyline divide the stroke polyline into its section polylines.*

- v**:** The extend of the target polygon away from the stroke section (see Fig. 28). Possible values are:
  - *On***:** v_l must be *on* the section polyline.
  - $\leftarrow x$**,** $\rightarrow x$**:** v_l must be either on the left or on the right hand side of the section polyline, on the same x-axis value as given by $u, plus, minus$.
  - $\leftarrow y$**,** $\rightarrow y$**:** v_l must be either on the left or on the right hand side of the section polyline, on the same y-axis value as given by $u, plus, minus$.
  - $\leftarrow perpendicular$**,** $\rightarrow perpendicular$**:** v_l must be either on the left or on the right hand side of the section polyline, inside a polygon projecting away from the polyline perpendicularly.

Let pl_s be the section polyline. Let l_{pl_s} be the line through the first and last vertex of pl_s. First, the orientation of pl_s is normalized depending on the value for v as follows:

- v is *On* or *perpendicular*: pl_s is rotated until l_{pl_s} is horizontal.
- v is y: pl_s is rotated by 90°.
- v is x: pl_s is not rotated.

The normalization ensures that the rectangle r can be generated in the same orientation in any of the seven cases of v-values. r is the rectangle the target polygon is baset on; r is delimited in width (i.e. x-axis coordinates) by the position and the width of the subsection computed as shown in Fig. 27. The height of the r is set to a very high value, stretching well offscreen.

The next important step is to ensure that pl_s intersects r correctly, since that is the precondition for successfully cutting away one half.

- If the number of intersections is not even, one of the two end vertices must be inside r. The incident edge of that vertex is extended until it intersects r.
- If the number of intersections is 0, one or both of the end edges is extended until it intersects r twice.

Finally, r can be cut by the extended pl'_s. The unnecessary half is discarded, leaving a polygon p. If v is On, p is displaced and cut again, leaving a narrow polygon matching the shape of pl'_s. A displacement transform is used to move p onto pl'_s if v is On, or slightly away from the stroke otherwise. The last step is to reverse the rotation of the normalization step, the result is the target polygon.

Finally, the vertex v_l is tested whether it is in the correct position by testing if it is inside the target polygon.

5.3 (Very) Big Finger Assistant

Writing a kanji character precisely enough that all location recognizers match would be very challenging without the software subtly giving a helping hand. Touch targets need to be large enough in order to be consistently touchable[26].

The solution to this problem employed here is for each target polygon p to calculate a uniformly larger version, p^+. Immediately after the student has finished drawing a stroke, the vertices with location recognizers are checked if they are within their respective p^+, but outside p. If that is the case, a stretch transform is applied to the stroke, moving the affected vertex into p. The result is that kanji glyphs can be drawn much more precisely.

[26] Apple recommends a minimum touch target size of 44 by 44 points in its Human Interface Guidelines.

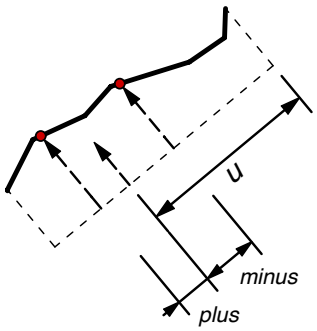

Figure 27: *A subsection of the section polyline is computed given the u value, which is the distance from the first vertex of the section specified as a percentage of the overall length of the section. The length of the subsection is determined by subtracting/adding the plus and minus values, also given as a percentage relative of the overall length.*

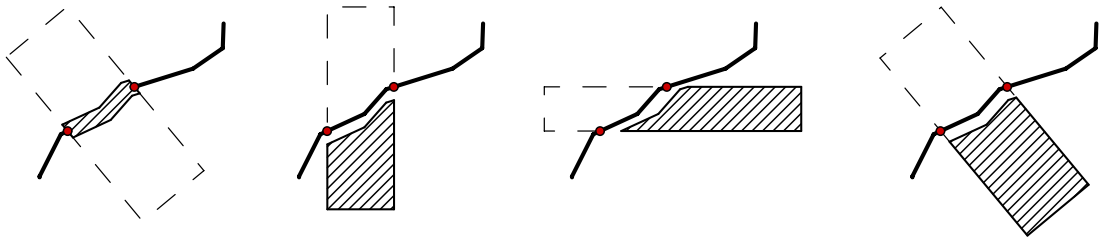

Figure 28: *The target range polygon is computed by fitting a rectangular polygon to the subsection, then performing a sequence of cutting and displacement operations. From left to right the v-values are: On, ←x, ←y, ←perpendicular. Not shown are the possible polygon locations to the right of the stroke section. The base rectangle r is shown dashed.*

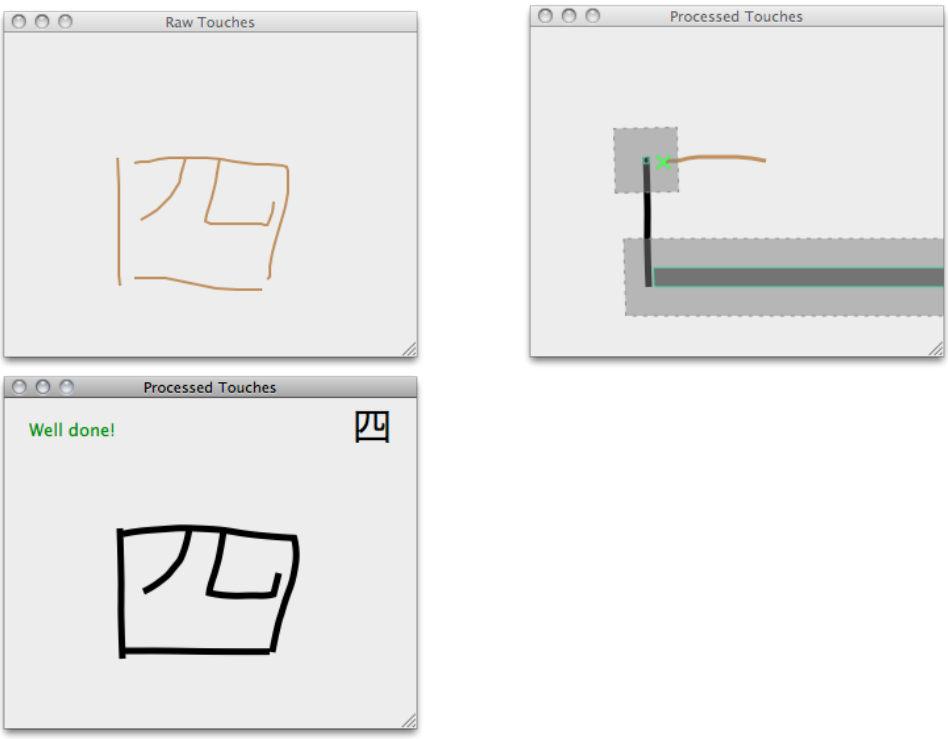

Figure 29: *The (Very) Big Finger Assistant at work. The student enters* 四*, but is challenged by the lack of precision of the touchscreen. The raw touch input on the top left shows the difficulties in aligning the stokes on the corners.*

(Top right) The second stroke in the process of drawing. The first vertex marked with a green 'x' is outside the target polygon, a very small area on the top of the first stroke. However, because the first vertex is in the enlarged polygon p^+*, it will be moved into the target polygon upon completion of the stroke.*

(Bottom) The finished character. The corners are aligned neatly.

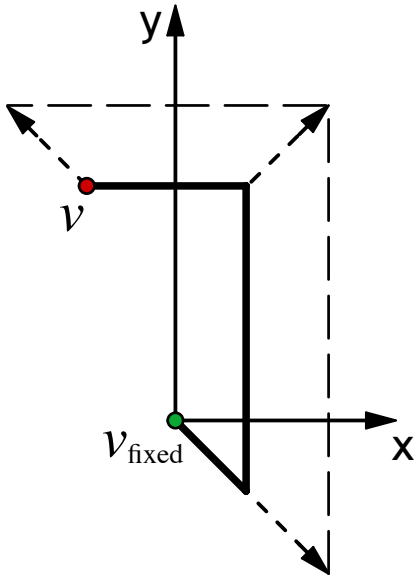

Figure 30: *Stretching unexpectedly.*

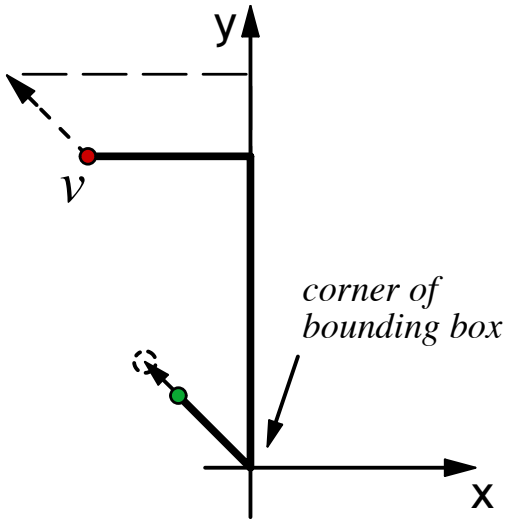

Figure 31:
An alternative stretching strategy.

Currently only the first and the last vertex of a stroke are moved in the way described above since these are typically to be placed more precisely than the intermediate vertices. If more extensive testing shows the need for intermediate vertices to be moved as well, the intersection vertices y_i of stroke polyline and the angular bisectors taken from the cage polyline (as described in Section 5.1) are the intermediate vertices to be moved.

Currently the algorithm for stretching a vertex into a new position starts with applying an affine transform to the polyline in question in order to move the vertex that is supposed to stay fixed into the origin. An affine transform stretching the polyline is applied so that after moving the line back into the original position, the vertex to be moved will end up in the desired location. At the moment, the fixed vertex is the one at the opposite end of the stroke, the last vertex if the first one is to be moved and vice versa. This approach works quite well in most cases, however in some cases the stretch feels counterintuitive to the student:

Let v be the vertex to be moved and v_{fixed} the vertex that is to stay in its current position. Let v be to the top left of v_{fixed} and be v_{fixed} to the top left of the bottom right corner of the bounding box (all other cases are analogous). The first affine transform applied moves v_{fixed} to the origin. If the stretch transform moves v to the top left, all points on the stroke that have a positive x-value will be moved to the right; and all points with a negative y-value will be moved down (see Fig. 30). This can be quite noticeable in a few cases. A better approach would be to move the stroke

to place the bottom right corner of the bounding box on the origin before applying the stretch transform (see Fig. 31. Other cases are analogous).

6 Evaluation

Applications written in Objective-C using the Cocoa frameworks use the same frameworks for both iOS and Mac OS X, with the exception of the user interface frameworks (App Kit for Mac OS, UI Kit for iOS). This offered the opportunity to develop a Mac OS application by compiling the same exact code as the iOS version, adding a desktop-specific user interface. The goal was to have a test application rather than having a fully usable learning application. The test application is named KanjiStories MacCompanion, the name of the iOS application is Kanji Stories.

In Kanji Stories, a setting can be enabled that causes all raw touch data to be recorded to a file in addition to be processed normally. This file, *rawTouchData.sqlite*, can be found when downloading the app data from the device using the XCode organizer. The raw touch data file can be opened, saved, or imported into another set of data in the MacCompanion application. Each set of raw touches is presented in a table. Upon selecting an entry, the touch events are supplied to the same back-end system as in the iOS application, causing the character to be drawn and recognized exactly the same way as on the device. Additionally the raw strokes are shown in another window.

Each touch event stored in the raw data file has a time stamp. This allows near real-time playback, simulating the student drawing the character by adding an appropriate pause before supplying the coordinates of the touch event to the back-end algorithm. The writing speed can be adjusted, or sped up to be processed as fast as the computer can by unchecking the *Real Time* box. Optionally, vertices, cage polylines and recognizers can be shown, however recognizers are only visible if *Real Time* is enabled. Logging can be disabled in order to avoid the speed penalty.

The checkbox in the *Correct* column of the table can be enabled or disabled in order to mark whether the raw touch data set should be recognized by the algorithm as a correct character or not. This setting can be saved with the data file. Pressing the *Check All*-button will caus the test application to iterate through all available data sets. The check result of each data

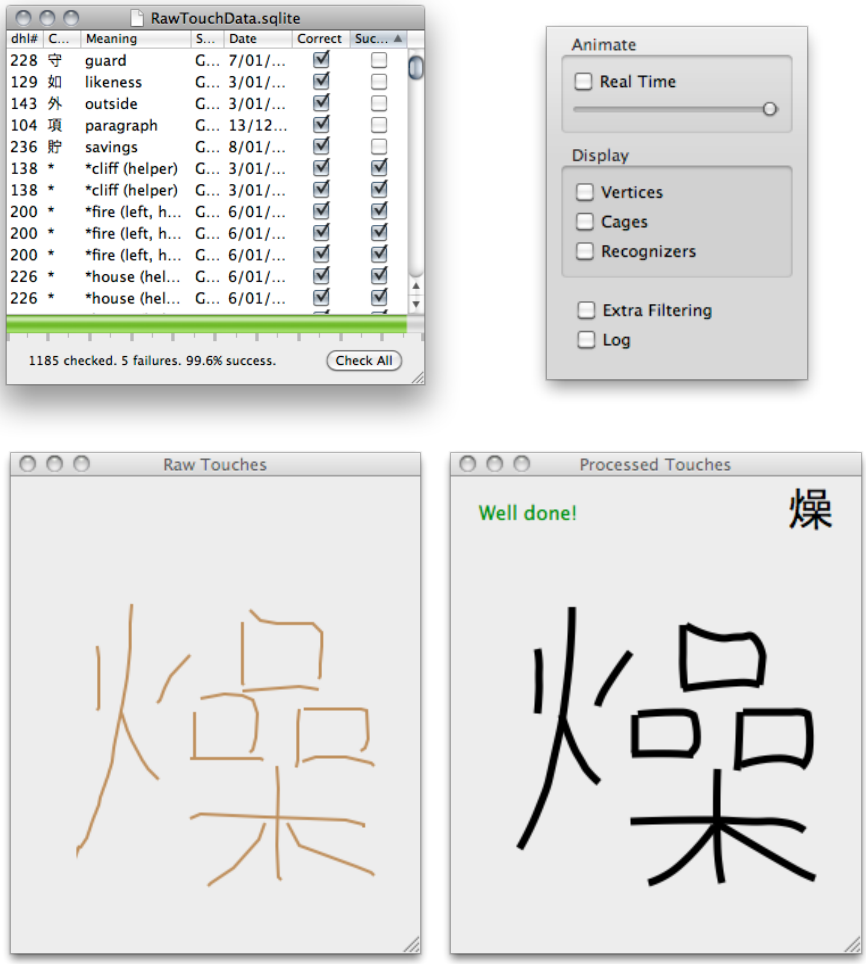

Figure 32: *The Kanji Stories MacCompanion application used for testing raw touch data collected from the mobile device.*

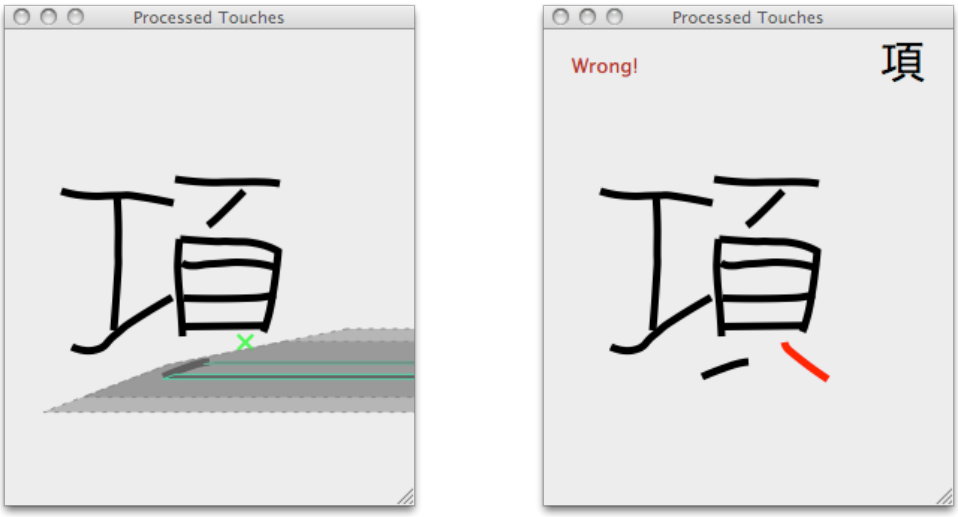

Figure 33: *Target polygons that are at a sharp angle relative to their stroke section are cut off at that sharp angle. Since the corresponding larger touch areas are cut off the same way, they also have that same steep angle. This may cause the problem shown here: The target polygons are the very narrow, dark areas protruding away to the right from the top and the bottom of the angled stroke. The touch area polygons are shaded gray, since two of them overlap in this case there is a darker area in the center. Due to the way these polygons are computed, they do not extend as far to the top left as they should, causing the first touch of the next stroke (the green cross) to be outside its touch area polygon. Therefore, the assistant will not move it onto the target polygon, the stroke is recognized as not correctly located.*

set tested is compared to the expected check result set by the checkboxes of the Correct-column. If the actual matches the expected check result, the corresponding check box in the *Success*-column will be enabled. By sorting the data set on the *Correct*-column, all failed recognition attempts can be inspected conveniently.

To date, a total of 234 Kanji have been specified as template characters. The total number of templates including glyph components that are not kanji characters themselves is 281. The total number of raw touch data collected so far amounts to 1185. Of these, 5 are not recognized correctly, a success rate of 99.6%. Of these 5 failed recognition attempts, two[27] are actually performing exactly as desired, both fail due to a hook on a stroke where no hook belongs. I have marked these as unsuccessful anyway because future testing needs to show whether it might be necessary to filter these inadvertent hooks out. Another failed recognition attempt[28] is due to an angle defined too restrictively, which can be corrected by editing the corresponding angle of the shape recognizer.

Fig. 33 shows a problem that has to be addressed by redesigning the way the touch area polygon is computed. This can be corrected if the touch

[27] 'likeness' and 'outside'
[28] 'savings'

area polygon is calculated by adding an offset outward to every edge of the target area polygon, instead of the approach described in 5.3.

The way the location can be specified is limited to the options available. These are sufficient to specify locations that are on a particular section of a stroke, off to either side of a stroke in x-axis, y-axis direction or aligned perpendicularly to the referred stroke section. By narrowing down the target range, specific vertices of a stroke can be defined as the target, or alignment horizontally or vertically can be modeled. This allows a very detailed specification of a location. A possible improvement would be to also allow specifying a location relative to the bounding box of a glyph component rather than a stroke section. The implementation of such improvement would allow reuse of the existing code, instead of computing the section polyline an edge of the bounding box can easily be provided. This would allow additional options for defining location recognizers.

Location Recognizers can be combined. If a vertex has two location recognizers, each referring to one of two stroke sections that are at an angle to each other, both location recognizers must match. In effect, the target range is the intersection of both polygons. In the current implementation, the big-finger-assistant uses only one of the locations in order to move the vertex if necessary (see 5.3). This can be improved by computing the intersection of the target polygons and touch area polygons involved, thus taking all locations into account when moving a vertex.

Raw touch data has been predominantly collected from the author, with only a few kanji written by two other persons on the author's device. More data has to be collected from other tester's devices to conclusively evaluate the success rate of the recognizer. The data collected so far is very encouraging, the high success rate is certainly in no small part due to the fact that the recognizer knows the template it is supposed to match (by design of the application).

Entering Kanji Data

Entering kanji data is a laborious process due to the sheer amount of kanji. In the author's testing, an average of 12 characters was entered per hour. A significant portion of that time is spent looking up the character, meaning and readings in dictionaries. This process can be (and should be) automated by parsing a dictionary file such as the kanjiDic. Even if the

kanji Definition speed might be doubled, a total time of almost 100 hours is necessary for kanji definition work for the full set of 2,136 jōyō kanji.

6.0.1 Additional Future Work

The following is a short list of future work in addition to the suggestions outlined in the section above. This work is not essential to the recognition algorithm.

Two features of the application should be implemented before more widespread testing commences. A 'Dispute'-functionality should let a student, who disagrees with the verdict of the character recognition, send the raw data to the author. This data will be valuable in improving the recognizer. Also, in order to improve memorizing the kanji, the student should be able to enter a story for each character.

Improved feedback when a character is wrong if the information is available would be very valuable to the student. For example, if an endpoint of a stroke is outside of a target range, an arrow pointing into the target range could be displayed. Or if a hook on a stroke is missing, a red hook could be added to the stroke.

An interesting question is, whether the recognizer could be adapted for search applications. Since the template characters are organized as a tree, the recognizer could filter the template characters with each stroke entered, until there are only a few or ideally just one matching template left when the character is fully drawn. This approach only works well if the character is drawn correctly since there is no distance function evaluated. An application as a postprocessing step of a different recognizer may be possible, where the other recognizer narrows the possible matches down to a small set, and if the algorithm developed for this thesis matches any of those characters it will be ranked at the top of the list of possible matches. Further research would be necessary to find out if that approach can improve the recognition rate of existing solutions.

7 Conclusion

A prototype has been developed for an application teaching students of the Japanese language the meaning and the writing of Kanji characters. The student can write kanji characters on a touch screen, getting feedback

about the correctness. Template data for a subset of the jōyō kanji has been generated offering a wide range of possible stroke shapes. The recognition algorithm has shown to perform well on kanji of this subset, although future improvements are possible.

References

[CPL+] Wing Hang Cheung, Ka Fai Pang, Michael R. Lyu, Kam Wing Ng, and Irwin King. Chinese optical character recognition for information extraction from video images. Available online at http://www.cse.cuhk.edu.hk/~lyu/paper_pdf/ocr.pdf.

[dBCvKO08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry*. Springer-Verlag, Berlin, 2008.

[Got05] Nanette Gottlieb. *Language and Society in Japan*. Cambridge University Press, 2005.

[Hei01] James W. Heisig. *Remembering the Kanji, Vol. I.* Japan Publications Trading Co, Ltd., 1-2-1 Sarugaku-cho, Chiyoda-ku, Tokyo 101-0064, Japan, 2001.

[Hil93] Optical recognition of handwritten chinese characters: Advances since 1980. *Pattern Recognition*, 26(2):205 – 225, 1993.

[JA08] Philip I. Pavlik Jr. and John R. Anderson. Using a model to compute the optimal schedule of practice. *Journal of Experimental Psychology: Applied*, Vol. 14(No. 2):101–117, 2008. Available online at http://www.apa.org/pubs/journals/features/xap142101.pdf.

[JK] Frederick H. Jackson and Marsha A. Kaplan. Lessons learned from fifty years of theory and practice in government language teaching. Available online at http://aladinrc.wrlc.org/bitstream/1961/3455/8/gurt_1999_07.pdf; visited January 5th 2012.

[JLN] S. Jaeger, C.-L. Liu, and M. Nakagawa. The state of the art in japanese online handwriting recognition compared to techniques in western handwriting recognition. *International Journal on Document Analysis and Recognition*, 6:75–88.

[KMCK00] Nishiguchi Koichi, Shinya Makiko, Koga Chiseko, and Mikogami Keiko. *Minna no Nihongo Shokyuu I Kanji Eigoban*. 3A Corporation, Shoei Bldg., Sarugaku-cho 2-chome, Chiyoda-ku, Tokyo 101-0064, Japan, 2000.

[Rub98] Jay Rubin. *Making Sense of Japanese*. Kodansha International Ltd., 1998.

[uni10] Cjk strokes; the unicode standard, version 6.0, 2010. Available online at http://unicode.org/charts/PDF/U31C0.pdf; visited January 3rd 2012.

[US05] Seiichi Uchida and Hiroaki Sakoe. A survey of elastic matching techniques for handwritten character recognition. *IEICE - Trans. Inf. Syst.*, E88-D:1781–1790, August 2005.

[Wik] Wikipedia. Dai kan-wa jiten. Available online at http://en.wikipedia.org/wiki/Daikanwa_Jiten; visited January 2nd 2012.

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Berlin, Januar 2012