

Recognizing Hand-drawn Glyphs from One Example and Four Lines of Code

Rachel Blagojevic¹, Dhruv Dhir², Kapil Ranganathan², Christof Lutteroth² and Beryl Plimmer²

¹Massey University, ²University of Auckland
New Zealand

r.v.blagojevic@massey.ac.nz, dhruvdhir11@gmail.com, srkapil@gmail.com,
lutteroth@cs.auckland.ac.nz, beryl@cs.auckland.ac.nz

Abstract

The biggest challenge in the development of gesture-based user interfaces is the creation of a gesture recognizer. Existing approaches to support high-level recognition of glyphs require a lot of effort from developers, are error prone, and suffer from low recognition rates. We propose a tool that generates a recognizer for hand-drawn glyphs from one example. Our tool uses the output of a basic shape recognizer as input to the glyph recognition. The recognizer can be integrated into an app by adding only four lines of code. By reducing the development effort required, the approach makes it possible for many touch-interaction apps to take advantage of hand-drawn content. We demonstrate the tools effectiveness with two examples. Furthermore, our within-subject evaluation shows that programmers with no knowledge of gesture recognition can generate a recognizer and integrate it into an app more quickly and easily than manually coding recognition rules, and that the generated recognizer is more accurate than a manually coded one.

Keywords: Gesture recognition; gesture based interaction.

1 Introduction

Touch interfaces on phones and tablets naturally afford hand drawn input. Functional gestures such as swipe and zoom are natively supported and widely used, yet there are only a few apps that leverage hand-drawing as a form of input. In part this is because such input is of little use unless the computer can understand its meaning. This understanding is reliant on robust recognizers, which are difficult to program. As a result, it is currently too much work for general programmers to add hand-drawn input to their apps.

The field of ink and gesture recognition has developed quickly over recent years. There are now a number of easy-to-implement (Wobbrock *et al.*, 2007) and componentized solutions (Chang *et al.*, 2012; Lü and Li, 2012) for gesture or single stroke recognition. However,

there are no similarly simple solutions to creating and using high-level recognizers for glyphs comprised of several strokes.

A bottom-up approach to recognition attempts to recognize individual ink segments and then progressively group these into larger and more complex glyphs, thus developing an overall semantic understanding of the diagram. Our recognizer performs high-level recognition where glyphs comprised of more than one basic shape (i.e. line, circle, rectangle etc.) are identified. Glyph recognition is one of the last steps in a bottom-up recognition process, coming after segmentation/grouping and basic shape recognition steps have been completed.

High-level recognizers can be built via three main approaches: textual, example-based and hybrid (Johnson *et al.*, 2009). The frequently used textual approach involves describing the glyphs using rules. The formalism that is generally used for this is sketch grammars, which specify the rules for combining symbols using spatial and temporal relations (Costagliola *et al.*, 2005a; Hammond and Davis, 2005). Conventionally, the textual approach requires the manual specification of rules for each glyph using the grammar. The recognizer uses these descriptions to classify new glyphs. Defining glyphs in this manner is both time-consuming and error-prone. Additionally, it takes time for the developer to become familiar with sketch grammars.

Example-based approaches offer an alternative way to build recognizers. Instead of specifying glyphs through rules, the developer provides example glyphs; from these examples rules are automatically deduced and a ready-to-use recognizer is produced. Thus, this approach can generate a recognizer quickly without code or knowledge of a grammar, through demonstration. This saves developers of gesture-based interfaces time whilst adding flexibility and robustness. One of our study participants described it thus:

“no coding, no thinking, very easy”

In this project we demonstrate how exemplar-based high-level recognition can be drastically simplified by 1) requiring only a single exemplar glyph and 2) making it easy to add a glyph recognizer to an application. The glyph is drawn in our recognizer generator tool. This generator includes a set of features that measure spatial relationships of the strokes in the exemplar to produce a recognition matrix. The recognition matrix is used to recognize new glyphs. The application developer simply

passes the raw sketch data to the recognizer through an API and the recognition result is returned to the program. Adding the recognizer to an app only requires loading the library and calling the recognizer.

To show the feasibility of this approach we have developed two example apps (Figure 1), which include all the core spatial relationships of hand-drawn glyphs in general. They also demonstrate how hand-drawn input could be used: one app generates HTML, the other is a game.

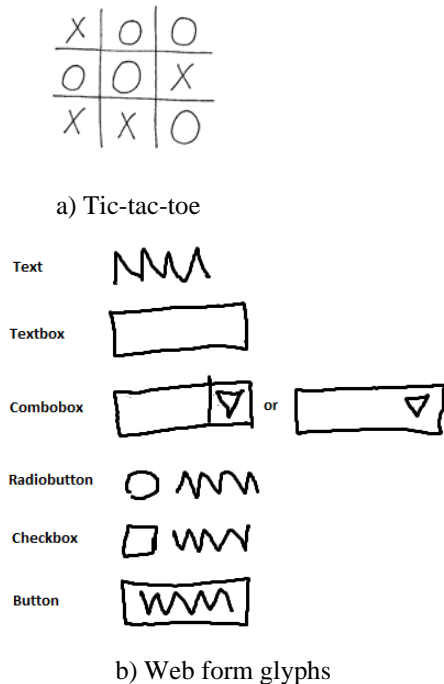


Figure 1. Glyphs recognized by each of our apps

The contributions of this project are:

- an approach that can generate a high-level glyph recognizer from one example per glyph,
- specifications of three spatial features with evidence indicating they are sufficient to recognize a range of multi-stroke hand-drawn glyphs,
- a simple API for app programmers to integrate the recognizer into their program,
- an evaluation illustrating the efficiency and accuracy of the proposed approach.

2 Related Work

The value of recognition lies in being able to use the interpretation of a sketch in intelligent ways. Previous work in high-level sketch understanding, describing the way sketched glyphs are drawn and the relationships between glyphs, falls into three categories (Johnson *et al.*, 2009): textual descriptions (grammars), exemplar-based approaches, and hybrid approaches.

Grammars (Costagliola *et al.*, 2005b; Costagliola *et al.*, 2005a; Hammond and Davis, 2005; Mas *et al.*, 2005; Hammond and Davis, 2007; Brieler and Minas, 2010; Mas *et al.*, 2010) commonly use a sketch description language to define all glyphs of a domain. Recognizers are then automatically generated using these descriptions. The language can be used to define complex glyphs, often

in a hierarchical manner, using definitions of primitive shapes to describe more complex glyphs. Editing operations or gestures, relationships, and various constraints can also be defined. The drawback of this approach is that defining the grammar itself is a cumbersome task with a large potential for error.

There are several example-based recognition systems that support high-level understanding of diagrams (Plimmer and Apperley, 2003; Sharon and Panne, 2006; Avola *et al.*, 2008). Much of this work stems from Rubine's (1991) early work in example-based gesture recognition. Rules for determining semantics are extracted from sketched examples by measuring various features. The probability of a candidate glyph matching an example is then calculated. The choice of features and the available sketched example set are central to the success of these approaches. Often these high-level recognizers are for only one domain (Avola *et al.*, 2008) or can require 20+ examples per class (Sharon and Panne, 2006). In contrast our goal is to support multiple domains with only one example per class.

There are also hybrid approaches which mix textual and example based approaches. Shilman *et al.* (Shilman *et al.*, 2001) generate recognizers using textual descriptions, but use examples to generate a statistical model of the thresholds representing relationships between glyphs. Hammond *et al.* (Hammond and Davis, 2006) require a textual description of a shape which is either written by the developer or can be generated automatically. The description is checked using automatically generated near-miss examples and the developer provides feedback as to whether the example is positive or negative.

Both textual and hybrid approaches require textual descriptions of diagrams. The manual specification of each glyph in a domain is both time-consuming and error-prone. On the other hand, the example-based method has the unique potential to permit swift generation of relatively robust high-level sketch recognizers in a non-tedious manner. Thus, we decided to use the same approach for our tool.

Gesture coder (Lü and Li, 2012) is a tool that allows developers to generate example-based multi-touch gesture recognizers. The system uses a state machine approach for gesture recognition and generates developer-modifiable code by learning from examples. Gesture coder's state machines are used to handle and distinguish between different types of finger touches (swipe, pinch, pan etc.). In contrast to the previously discussed recognizers, gesture coder learns from demonstration (using gesture trajectories, distance between gestures, global gesture attributes etc.) and generates code for recognition automatically rather than asking developers to describe recognition using sketch sentences/descriptions. Reported recognition rates are about 65% for 6 or more classes. The user evaluation of Gesture Coder compared programmers integrating it into an app with coding the recognizer from the sketch primitives, an almost impossible task for the non-expert participants to complete within the timeframe of a study. Unsurprisingly, many gave up and some produced only a simple partial solution.

The \$-Family (Wobbrock *et al.*, 2007; Anthony and Wobbrock, 2012; Vatavu *et al.*, 2012) recognizers are

designed for simple, fast and accurate gesture recognition. The \$P and \$N versions are able to recognize multi-stroke gestures. The code required is minimal and pseudocode is provided to assist developers.

Gesture Coder and the \$-Family recognizers are specifically for multi-touch/pen functional gestures rather than glyph recognition. Glyphs are typically more detailed in nature than a multi-touch functional gesture, they are not always ordered by time, and require high-level recognition to piece each bit together to interpret the glyph as a whole. Multi-touch gestures can be easily grouped according to time, i.e. the gesture is performed within a certain time period and only one gesture is on the canvas at one time. For diagramming, recognizers must be able to handle many glyphs on the canvas at once.

In summary, there has been no approach so far that can build robust high-level glyph recognizers with little development effort. Existing approaches suffer from a need to manually specify the rules in a text-based grammar, or are restricted to a single stroke or gestures.

3 Our Approach

Glyphs in visual languages rely on the shapes of the individual strokes and spatial relationships between these strokes. In their seminal work on topical spatial relationships, Egenhofer and Fanzosa (1991) define the spatial relationships between a pair of ellipse shaped regions as having the following possible conditions: disjointed, touching, equal, containment, covers, overlaps. Other work (1990) extends the definition to include simple lines interspersed with the ellipses. Our case is different in that, rather than regions, glyphs are represented by a number of non-straight lines. In addition, it is difficult to draw precisely so we must cater for some fuzziness. We simplify the abovementioned set of conditions to four that are sufficient to effectively recognize a range of hand-drawn glyphs:

- Contains
- Intersects
- Adjacent
- Disjointed

Egenhofer and Fanzosa (1991) go on to show how these conditions can be represented in a binary matrix to describe a particular spatial arrangement of regions.

Whereas Egenhofer and Fanzosa considered only ellipses and lines, this project deals with a far wider set of basic strokes. For the single-stroke recognition we rely on RATA.Gesture (Chang *et al.*, 2012), it generates a stroke recognizer from a few examples. The results of the gesture recognizer are then used in our tool to build the high-level glyph recognizer.

At the core of all gesture recognizers are features. We have devised three features, one each for the conditions to be recognized. To build a recognizer we also borrow from Egenhofer and Fanzosa (1991) the idea to use a binary matrix to represent which conditions exist in the current drawing and which do not. How this recognizer is constructed is described in Section 4. Section 5 describes how the recognizer can be used by the application programmer via an API. This API also makes it possible for developers to test for features in a sketch manually.

In Section 6 we describe two proof-of-concept apps. Each requires all spatial features to be recognized and results in a different action when drawing is completed. These apps are used to test the recognizers' accuracy. Finally, to ensure that the tool is usable by app programmers and is more efficient than hand-coding the spatial relationships, we conducted a comparative study where student programmers performed two tasks: one task using this tool, and the other task coding the rules manually using the three proposed spatial features. This is a fairer evaluation than in (Lü and Li, 2012) as functions to detect spatial relationship features are made available to the participants in our study.

4 Recognizer Generation

This section describes how our approach can be used to generate a recognizer from one hand-drawn exemplar. We use the example of a combo box to explain each concept.

4.1 Features

Using knowledge of spatial properties from Egenhofer and Fanzosa (1991), we identified three key distinguishing features to include in our recognizer: intersection, containment, and adjacency. If none of these features are present, then the strokes are disjointed. These features are depicted in Figure 2 for the combo box example.

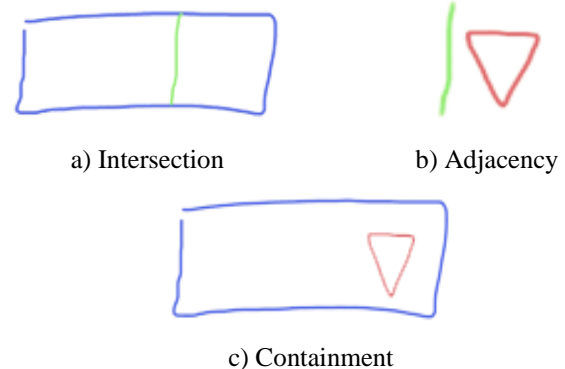


Figure 2. Features of a combo box

Intersection occurs when two strokes cross at any point. This does not include self-intersections or strokes that are close but not intersecting.

Containment is when the bounding box (axis aligned) of one stroke is inside another stroke's bounding box. If the inner bounding box is outside of the outer bounding box at any point then it is not considered containment – this would be intersection or adjacency. Note that this feature is not symmetric, e.g. the arrow of the combo box in Figure 2c is contained in the textbox but the textbox is not contained in the arrow.

Adjacency occurs when two strokes are intersecting or horizontally or vertically adjacent. Horizontal and vertical adjacency is calculated using the bounding boxes (axis aligned) of the strokes. Consider horizontal adjacency. First the maximum width of the two bounding boxes is found, which is used to calculate a threshold for "closeness". "Closeness" is calculated as a proportion of

the maximum width or height, e.g. for horizontal adjacency this is set to 50% of the maximum width, which was determined by informal testing. If the distance between the bounding boxes' left or right sides are within the threshold, they are considered horizontally "close". Then a similar calculation is completed to find whether the bounding boxes sit on the same horizontal line by checking if the top of one box is "close" to the middle of the other. The threshold used here is 30% of the maximum bounding box height. Vertical adjacency is calculated in a similar way except that it considers the top and bottom of the boxes to determine vertical "closeness" and the left and right sides to determine if they are on the same vertical line (thresholds for "closeness" used here are 10% of the maximum bounding box height and 30% of the maximum width respectively determined via informal testing).

4.2 Matrices

A matrix for each of the above features is derived to represent the relationships that exist in a multi-stroke glyph (e.g. Figure 3). These matrices encode spatial relationships between all possible pairs of strokes. Each matrix has size $n \times n$, where n is the number of single-strokes in the glyph. In Figure 3 the first matrix represents the intersections that exist between strokes in the glyph, the second is for containment relationships and the third for adjacency. The relationships are represented using Boolean values, where '1' indicates that the relationship exists between two strokes. The name of each single stroke is shown in the first row of the matrix; these are ordered alphabetically as recognition is independent of the order in which strokes are drawn. In Figure 3, the highlighted cell of the intersection matrix shows that there is an intersection between the textbox and divider strokes, as illustrated in the figure. The highlighted cell in the containment matrix shows that the arrow is contained by the textbox. For the adjacency matrix the highlighted relationship shows that the divider is adjacent to the arrow. Overall, for the combo box there are more adjacency relationships than intersection or containment.




!combobox				#Glyph name
Textbox-Arrow-Divider				#Single strokes
Textbox	0,	0,	1,	#Intersection matrix
Arrow	0,	0,	0,	
Divider	1,	0,	0,	
#				
Textbox-Arrow-Divider				#Containment matrix
Textbox	0,	1,	0,	
Arrow	0,	0,	0,	
Divider	0,	0,	0,	
#				
Textbox-Arrow-Divider				#Adjacency matrix
Textbox	0,	1,	1,	
Arrow	1,	0,	1,	
Divider	1,	1,	0,	

Figure 3. Feature matrices for a combo box. The highlighted matrix cells correspond to the relationships shown in the example sketches.

Recognition must be independent of stroke drawing order. For example, the strokes in Figure 1 could be added to a canvas in any order. To achieve drawing order

independence without computing all permutations of drawing order, we sort the strokes in both the recognition matrices and those on the canvas into alphabetical order by their single stroke label.

The recognizer must also recognize glyphs that are a superset of other glyphs, e.g. the textbox in Figure 1 is a valid glyph in its own right and would usually be drawn before the other parts of the combo box. The recognizer must therefore consider all previous strokes regardless of their recognition state. The recognizer tries to recognize from the glyph with the largest to smallest number of component strokes. This is computationally expensive so we provide two recognizer modes: instant (only considers strokes that are not already recognized as a part of a glyph) and iterative (considers all strokes).

Developers create a recognizer by drawing one example of each glyph they want to recognize in our tool. This requires minimal effort and no understanding of recognition techniques. They may provide more than one example glyph if required. The feature matrices are generated automatically using the examples and stored in a recognizer file. This recognizer can then be integrated into an app.

5 Recognizer Integration via the API

Once the recognizer has been generated, it can be loaded into an app to identify newly drawn glyphs using the code shown in Figure 4.

```

Helper recogHelper = new Helper(this);      (1)
recogHelper.loadModelFile(RATA_SSR_FILE);  (2)
recogHelper.loadFile("UI.txt");           (3)
...
Void onTraceRecognized(trace){
    String result=recogHelper.recognize(trace);(4)
}

```

Figure 4. Code required to integrate a recognizer

To use the generated recognizer in an app a Helper object is created (Figure 4(1)). Next, the single-stroke recognizer (Figure 4(2)) and the generated recognizer matrix file (Figure 4(3)) are loaded. With the set-up phase complete, new glyphs can be passed to the recognizer to be identified as they are drawn with the onTraceRecognized() function (this function is triggered when a new stroke is drawn). The recognize() function (Figure 4(4)) executes the recognition process. This processing involves two steps:

- 1) recognizing each stroke individually using the single stroke recognizer (Chang *et al.*, 2012), and
- 2) deducing spatial relationships between the recognized strokes using the generated recognizer.

With the results of the single stroke recognizer a matrix can be generated from the newly drawn strokes. The recognition system considers at most the n most recent strokes, where n is the number of the strokes that make up the largest glyph in the domain. This matrix is compared against the loaded matrices (Figure 4(3)) for a match. To optimize performance, the comparison is skipped if there is no glyph in the domain which is made

up of the same number of strokes as the recognizer matrix generated from user-drawn strokes. In case of a match, the recognition result is returned to the developer’s app through the `onTracesRecognized()` callback method for them to process.

For the more advanced developer we have exposed further recognition preferences that can be set through the Helper object. The developer may choose which features are considered during recognition; by default the recognizer uses all three. Additionally, the developer may choose to enable the instant recognition mode; by default the iterative recognition that considers all strokes is used.

6 Proof of Concept

We have implemented two example apps as proof of concept: sketching user interfaces and tic-tac-toe. All three spatial relationships are represented in both domains. The different domains also illustrate different post-processing of the recognition results.

A survey was conducted to observe how people draw within these domains; this aided us in identifying the form of the glyphs required for recognition. Each participant of the survey was asked to draw: a typical web form, and a tic-tac-toe playing board with a circle or cross in each grid. Participants used several user interface glyphs such as textboxes, combo boxes, radio buttons, checkboxes, buttons and labels for the web form. The most common form of glyphs drawn is shown in Figure 1b. All participants drew the tic-tac-toe playing board as shown in Figure 1a.

Using the results of our survey we generated recognizers for each of the glyphs in the domains. This involved drawing one example of each glyph with our recognizer generation tool. For glyphs that could be drawn in more than one way, i.e. combo boxes, an example for each variant was drawn.

The recognizers were integrated into an app for each domain. Once the glyphs are recognized within the app, post processing can be performed to take advantage of the recognition results. For the user interface domain we used the sketches to generate corresponding HTML code. We used the tic-tac-toe recognizer to create a game which involves turn taking and informs the users when someone has won. It also uses the recognition to perform error checking, such as ensuring there is only one symbol per cell and a symbol does not overlap other cells.

6.1 Recognizer accuracy

We also evaluated the accuracy of the recognizers (calculated as the number of correctly classified glyphs / total number of glyphs) in our proof of concept apps. To do this we collected sketches from nine people where they were asked to draw using the two apps. For the tic-tac-toe app participants played five games of tic-tac-toe against themselves. For the user interface app participants drew eight of each web form component, (see Figure 1b). Some participants drew slightly more or less than the number specified depending on the time available. The number of glyphs collected for each app and recognition results are shown in Table 1.

The recognition results are good, with the tic-tac-toe app reaching an almost perfect recognition rate, and the

user interface recognition achieving 85.9% accuracy. The recognizer for the user interface domain had more glyphs to interpret than tic-tac-toe which may account for the difference in accuracy.

	# Glyphs	% total correct	% correct (RATA)	% correct (glyph rec)
User Interface	496	85.9	94.5	91.3
Tic-tac-toe	435	99.5	100.0	99.5

Table 1. Recognition accuracy of each app.

The last two columns of Table 1 show the source of errors made, with the percentage of glyphs that were correctly classified by RATA (the single-stroke recognizer) and by the generated glyph recognizer. The glyph recognizer is responsible for a larger proportion of the errors than RATA, particularly for the user interface domain. On closer inspection we found two main sources of misclassification for user interfaces: 45% of the comboboxes (type 1 in Figure 1b) were misclassified; and 32% of the radiobuttons were misclassified. The combobox was often identified as a textbox; we believe this is because the inner line was not found to intersect with the outer box. The radiobutton was commonly classified as a label, most likely because the circles were not found to be adjacent to the label. This indicates some room for improvement for our glyph recognition strategy, which could include allowing for more fuzzy conditions to be applied, especially when determining thresholds for “closeness”. It also highlights the need for extra measures to be applied to minimise the effect of errors from previous recognition steps, RATA in this case, on glyph recognition. These issues are discussed further in Section 8.

7 User Study

The goal of the user study was twofold: to test whether ordinary programmers could generate and integrate the recognizer into an app; and to test if this is more efficient in terms of time and accuracy than hard-coding a recognizer for the chosen app. The task was to create a high-level recognizer for a non-trivial domain. Ten 4th-year students who were well-versed in Java programming were recruited.

In order to make the comparative evaluation as fair as possible, our evaluation makes two important differences from the study by Lü and Li (2012), who performed a similar evaluation for multi-touch functional gesture recognition. First, we made sure all participants were familiar with the domain. We chose the UI domain as all of the participants had previous experience with UI development. Second, we provided the participants with adequate technological support when hard-coding a recognizer. That is, the participants had access to our features for detecting spatial relationships. There are a number of feature libraries publically available that an application programmer could employ, so realistically developers would not code features.

7.1 Methodology

We conducted pilot tests with three participants to determine whether the instructions and tasks were appropriate. This was followed by individual sessions during which each of the participants worked alone. We collected information on participants' prior experience and opinions about the tasks through a questionnaire.

Each participant had to generate two recognizers for the web UI domain: one generated using our tool (Tool) and the other created through the hard-coding approach (Hard-code). The glyphs they focused on are those shown in Figure 1b. Participants were given a maximum of 30 minutes for each task. For both the tasks, participants were provided with the User Interface app which generated HTML code (on one half of the screen) for the web UI glyphs sketched (on the other half of the screen). The bits of the source code responsible for recognition were omitted from the app, but where they should be added was clearly marked with comments. The participants followed the provided instructions to work through both parts. To aid with the hard-coding task, we particularly pointed out the functions to check for spatial relationships (intersection, containment and adjacency) between a pair of strokes, and also ensured they understood how to use these functions to hard-code a high-level recognizer.

In the questionnaire's pre-task section participants rated their familiarity with: touch interfaces, sketching apps on touch-screen devices, Java programming, Android programming and programming of sketch recognizers. In the two post-task sections of the questionnaire, participants were asked to rate task comprehension, ease of using the approach to generate a recognizer, perceived recognizer accuracy and enjoyment. All these ratings were performed using a five-point Likert scale. In the last section of the questionnaire, participants were asked to rank the two approaches for generating recognizers for ease of use and accuracy; they were also asked to provide an overall ranking. Lastly, they were asked to comment on what they liked/disliked about the two approaches considered.

Half the participants were asked to generate a recognizer using our tool first whereas the other half were asked to create one using the hard-coding approach first, to balance any order effects. Before starting the tasks, participants were given a two minute demonstration of how to use our tool to generate a recognizer by example, and asked to complete the first part of the questionnaire. In addition, we provided a hand-out containing instructions for both the tasks. The participants were asked to fill out a post-task section of the questionnaire following each completion of each task. Participants performed the tasks on an ASUS tablet running Android 4.2.1 and a DELL core i5 PC running the Eclipse IDE (with ADT installed) on Windows 7.

After the participants implemented the recognisers in the apps, they were asked to run them on the Android tablet and test the recognition. They were encouraged to stress test the recognizer by varying the order in which the glyphs were sketched (the glyphs tested are the same as in Figure 1b). At the end of the evaluation, the participants were asked to complete the rest of the

questionnaire, including the open-ended questions and comments. We used each participant's test data to measure the accuracy of the recognizers they produced. A Wilcoxon Signed Rank Test was used to test for significant differences in the results (as they were not normally distributed), unless stated otherwise.

7.2 Results

Participant's self-ratings of existing skills and knowledge were recorded on a 1-5 scale, with 5 being "expert". All participants rated themselves as 5 regarding touch interface use, but the mean was 3.2 for using sketching apps on touch devices. For programming skills, all rated Java programming as 5, with the Android programming mean 4.3 and gesture recognition programming mean just 2, indicating none had experience in coding recognizers.

Table 2 shows the results of the questionnaire particularly for questions asked about both methods of recognizer creation.

Participants reported positively on task comprehension (Table 2 Q1) for both tasks (Tool: $M = 4.5$ $SD = 0.71$, Hard-code: $M = 4.4$ $SD = 0.70$). All participants either completed the tasks or coded till their time expired. On average participants took 11.60 minutes to complete the Tool task ($SD = 3.13$), whereas hard-coding took an average of 27.40 minutes ($SD = 3.60$). A paired t-test showed that there was a significant difference in the time that participants took to generate and integrate a recognizer ($t = -9.488$, $p < 0.001$, Cohen's $d = -3.00$).

Participants wrote 10 lines of code on average when they used our tool and API; when hard-coding participants added an average of 45 lines. Participant 2 produced one of the better results, shown in Figure 5 and Figure 6. The code in Figure 6 shows the main recognition method `doRecognition()`, which gets three strokes as arguments. The method is dominated by many conditional statements that consider the different orderings in which the strokes making up a glyph are given. For example, when recognizing a combo box in the first lines of the method, each of the given `StrokeEvents` could contain the text box that needs to be recognized as part of a combo box. When hard-coding, many participants did not complete the recognizer as they did not include code to handle all permutations of strokes that a glyph could be made up of. This was either

Question	Method	SD	D	N	A	SA
Q1. I understood the task	Tool			1	3	6
	HC			1	4	5
Q2. Using the tool/hardcoding was easy	Tool				7	3
	HC		2	2	6	
Q3. I enjoyed using the tool/hardcoding	Tool			1	4	5
	HC	2		1	5	2
Q4. The recognition was accurate	Tool				5	5
	HC		2	3	3	2

Table 2. Results of comparative questionnaire (HC = hard-code)

because of time restrictions or they did not consider these cases.

Participants found generating a recognizer by example using our tool to be very easy ($M = 4.4$, $SD = 0.52$). Integrating the recognizer with the app was found to be equally easy. The participants found using our tool (Table 2 Q2) significantly easier ($Z=-2.251$, $p = 0.024$, $r = -0.50$) than doing the same by hard-coding (Tool: $M = 4.3$ $SD = 0.48$, Hard-code: $M = 3.4$ $SD = 0.84$). Also, the participants found our tool to be more enjoyable (Table 2

Q3) than hard-coding (Tool: $M = 4.4$ $SD = 0.70$, Hard-code: $M = 3.5$ $SD = 1.43$), although this was not statistically significant ($Z=-1.476$, $p=0.14$, $r = -0.33$).

Since our tool considers all possible permutations in which the strokes of a diagram can be drawn, participants rated it as significantly more accurate (Table 2 Q4) than the hard-coded recognizer ($Z=-2.232$, $p=0.026$, $r=-0.50$, Tool: $M=4.5$ $SD=0.53$, Hard-code: $M=3.4$ $SD=1.07$).

```

Helper msrHelper = new Helper(this);
msrHelper.loadModelFile(RATA_SSR_FILE);
msrHelper.loadFile(RATA_MSR_FILE);

void onTraceRecognized(TraceEvent stroke, String result) {
    msrHelper.recognize(stroke);
}
void onTracesRecognized(RecogResult r) {
    String msrResult = r.getResult();
}

```

Figure 5. Participant 2's code using our tool

```

// TODO: Implement the method.
// In this method:
// 1) Recognize 2-stroke components i.e. Button, Checkbox and Radiobutton.
// 2) Recognize the 3-stroke component i.e. Combobox.

private String doRecognition(StrokeEvent strokeEvent1, StrokeEvent strokeEvent2, StrokeEvent
strokeEvent3){
    if(strokeEvent3.exists() == true){
        if(strokeEvent1.isRecognisedResult("Textbox") && strokeEvent2.isRecognisedResult("Divider")
&& strokeEvent3.isRecognisedResult("Arrow") && isContained(strokeEvent1, strokeEvent3)
&& isIntersecting(strokeEvent1, strokeEvent2) && isAdjacent(strokeEvent2, strokeEvent3))
            return "combobox";
        if(strokeEvent2.isRecognisedResult("Textbox") && strokeEvent1.isRecognised...
            return "combobox";
        if(strokeEvent2.isRecognisedResult("Textbox") && strokeEvent3.isRecognised...
            return "combobox";
        if(strokeEvent3.isRecognisedResult("Textbox") && strokeEvent1.isRecognised...
            return "combobox";
        if(strokeEvent3.isRecognisedResult("Textbox") && strokeEvent2.isRecognised...
            return "combobox";
        if(strokeEvent1.isRecognisedResult("Textbox") && strokeEvent3.isRecognised...
            return "combobox";
    }
    if((isContained(strokeEvent1, strokeEvent2) && (strokeEvent1.isRecognisedResult("Textbox")
&& strokeEvent2.isRecognisedResult("Label")) || (isContained(strokeEvent2, strokeEvent1) &&
strokeEvent2.isRecognisedResult("Textbox") && strokeEvent1.isRecognisedResult("Label")))){
        return "Button";
    }
    if(isAdjacent(strokeEvent1, strokeEvent2)){
        if((strokeEvent1.isRecognisedResult("Circle") && strokeEvent2.isRecognisedResult("Label"))
|| (strokeEvent2.isRecognisedResult("Circle") &&
strokeEvent1.isRecognisedResult("Label"))
            return "radiobutton";
        else if((strokeEvent1.isRecognisedResult("Textbox") &&
strokeEvent2.isRecognisedResult("Label"))
|| (strokeEvent2.isRecognisedResult("Textbox") &&
strokeEvent1.isRecognisedResult("Label"))
            return "checkbox";
    }
    return "NO_MATCH";
}

public boolean isIntersecting(StrokeEvent strokeEvent1, StrokeEvent strokeEvent2){
    ... code was provided }
public boolean isContained(StrokeEvent strokeEvent1, StrokeEvent strokeEvent2){
    ... code was provided }
public boolean isAdjacent(StrokeEvent strokeEvent1, StrokeEvent strokeEvent2){
    ... code was provided }

```

Figure 6. Participant 2s Hard-coded implementation (note: many lines truncated as indicated with ...)

For the same reason participants agreed that hard-coding a recognizer was tedious ($M = 3.8$ $SD = 0.92$). Typical participant comments for hard-coding were “need to spend too much time”, “less flexible” and “very tedious”.

The accuracy of the recognizers built during the evaluation was calculated by the number of correctly recognized UI glyphs drawn by the participants / total number of glyphs drawn by the participants. The accuracy for recognizers generated using our tool was 100% ($SD = 0$), whereas the mean accuracy for recognizers generated through hard-coding was 75.10% ($SD = 35.77$).

Each and every participant rated our tool better than hard-coding for accuracy as well as ease of use. The overall ranking also showed our tool to be their preferred approach. This positive feedback was further reflected in the open-ended comments with answers such as: “the tool aspires to the ideal of a recognition library by removing the need to do any stroke processing”, “no coding, no thinking, very easy”, “easy for lots of diagrams at once”.

8 Discussion

The goal of this project is to provide a tool for generating high-level recognizers and an API to allow developers to easily integrate these recognizers into their apps. Our evaluation shows that developers unfamiliar with building recognizers are able to quickly generate a high-level recognizer with accurate results. The study participants were unfamiliar with sketch recognition, yet they reported the tool was easy as well as enjoyable to use. It was also considerably quicker: generating a recognizer using the tool took 12 minutes, as opposed to 27 minutes using the spatial features we provided to hard-code a recognizer. In addition, the recognizer generated using the tool was superior to its counterpart in terms of accuracy. In a real world scenario we would expect the diagrams to be larger in terms of number of strokes. In such a situation the benefits of using our tool (versus hard-coding) would become even more apparent.

The spatial features of glyphs we have considered, although not exhaustive, have produced good results for the user interface and tic-tac-toe domain. In the future we plan to extend these to consider the orientation relationship between strokes, e.g. where a line attached to the top or bottom of a circle has a different meaning. In many cases, for example node-and-edge diagrams, this is not important. However, there are situations where it may be appropriate to make such differences. In addition, our recognizer currently uses features that return a Boolean value (either the feature is present for the given strokes or it is not) and corresponding recognition matrices. This simple approach has worked well for user interface and tic-tac-toe domains. However, it could be expanded to provide a continuous measure for domains such as set diagrams, e.g. “50% overlap”, or could include machine learning classifiers such as those that other recognizers employ.

The use of any high-level recognizer relies on the results of the recognition steps that precede the current step in the process. A bottom-up approach to recognition begins by recognizing single ink segments and then progressively groups these into more complex glyphs. The ultimate goal is to develop an overall semantic

understanding of the diagram. Our recognizer performs high-level recognition where glyphs comprising of more than one basic shape are identified; this is one of the last steps in the recognition process. To account for the preceding recognition steps we have used a single-stroke recognizer (Chang *et al.*, 2012). In future work we intend to investigate ways of minimising the effect of errors from earlier stages of recognition on the later stages such as glyph recognition.

We are yet to optimise the runtime performance of the recognizer. With larger sketches it would benefit from partitioning the canvas using techniques such as (Moran *et al.*, 1997).

There are a number of single-stroke recognizers available. The one we selected (Chang *et al.*, 2012), has the advantage of being example-based, requiring no coding on the part of the programmer and having a simple API. Thus it meets our goal of a no-coding solution to glyph recognition. We have hidden the API of (Chang *et al.*, 2012) inside our own recognizer, but this could easily be exposed to app programmers if this was deemed helpful.

The sample apps that we developed show how our tool can be applied in different domains. The UI tool converts the sketch to HTML code that can be readily rendered in a browser. This is in the spirit of many early sketch tools such as (Landay, 1995). Tic-tac-toe is an example of a mini game app. There are many possible mini games that could be created using our tool.

9 Conclusion

We presented a novel approach that allows software developers to create a high-level recognizer from one example glyph, and integrate it into their app with as little as four lines of code. The effectiveness of this tool has been demonstrated in two apps employing recognizers built using our tool and API. The user study shows that developers unfamiliar with creating recognizers are able to use the approach to generate accurate high-level recognizers very easily, and are satisfied with the usability of the tool.

10 References

- Anthony, L. & Wobbrock, J. O. 2012. \$N\$-protractor: a fast and accurate multistroke recognizer. Proceedings of Graphics Interface 2012. Toronto, Ontario, Canada: Canadian Information Processing Society.
- Avola, D., Ferri, F., Grifoni, P. & Paolozzi, S. 2008. A Framework for Designing and Recognizing Sketch-Based Libraries for Pervasive Systems. UNISCON. Springer.
- Brieler, F. & Minas, M. 2010. A model-based recognition engine for sketched diagrams. Journal of Visual Languages & Computing, 21, 81-97. Available: DOI <http://dx.doi.org/10.1016/j.jvlc.2009.12.002>.
- Chang, S. H.-H., Blagojevic, R. & Plimmer, B. 2012. RATA.Gesture: A gesture recognizer developed using data mining. AI EDAM, 26, 351-366. Available: DOI [doi:10.1017/S0890060412000194](https://doi.org/10.1017/S0890060412000194) [Accessed 2012].
- Costagliola, G., Deufemia, V. & Risi, M. Sketch Grammars: a formalism for describing and recognizing

- diagrammatic sketch languages. *Document Analysis and Recognition*, 29 Aug.-1 Sept. 2005 2005a. 1226-1230.
- Costagliola, G., Deufemia, V. & Risi, M. A trainable system for recognizing diagrammatic sketch languages. *Visual Languages and Human-Centric Computing*, 2005 IEEE Symposium on, 20-24 Sept. 2005 2005b. 281-283.
- Egenhofer, M. J. & Franzosa, R. D. 1991. Point-set topological spatial relations. *International Journal of Geographical Information System*, 5, 161-174.
- Egenhofer, M. J. & Herring, J. 1990. Categorizing binary topological relations between regions, lines, and points in geographic databases. University of Maine, Orono, Maine, Dept. of Surveying Engineering, Technical Report.
- Hammond, T. & Davis, R. 2005. LADDER, a sketching language for user interface developers. *Computers & Graphics*, 29, 518-532. Available: DOI 10.1016/j.cag.2005.05.005.
- Hammond, T. & Davis, R. 2006. Interactive learning of structural shape descriptions from automatically generated near-miss examples. *Proceedings of the 11th international conference on Intelligent user interfaces*. Sydney, Australia: ACM.
- Hammond, T. & Davis, R. 2007. LADDER, a sketching language for user interface developers. *ACM SIGGRAPH 2007 courses*. San Diego, California: ACM.
- Johnson, G., Gross, M. D., Hong, J. & Do, E. Y.-L. 2009. Computational Support for Sketching in Design: A Review. *Foundations and Trends in Human-Computer Interaction*, 2, 1-93.
- Landay, J. Interactive sketching for user interface design. *Chi '95 Mosaic of Creativity*, Doctoral Consortium, May 7-11 1995 ACM. Photocopy on file, 63-64.
- Lü, H. & Li, Y. 2012. Gesture coder: a tool for programming multi-touch gestures by demonstration. *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*. Austin, Texas, USA: ACM.
- Mas, J., Lladós, J., Sanchez, G. & Jorge, J. A. P. 2010. A syntactic approach based on distortion-tolerant Adjacency Grammars and a spatial-directed parser to interpret sketched diagrams. *Pattern Recognition*, 43, 4148-4164. Available: DOI <http://dx.doi.org/10.1016/j.patcog.2010.07.003>.
- Mas, J., Sánchez, G. & Lladós, J. 2005. An Adjacency Grammar to Recognize Symbols and Gestures in a Digital Pen Framework. In: Marques, J., Pérez de la Blanca, N. & Pina, P. (eds.) *Pattern Recognition and Image Analysis*. Springer Berlin Heidelberg. Available: DOI 10.1007/11492542_15.
- Moran, T. P., Chiu, P. & van Melle, W. Pen-Based interaction techniques for organizing material on an electronic whiteboard. *10th Annual Symposium on User Interface Software and Technology*, 1997 Banff, Canada. ACM, 45-54.
- Plimmer, B. E. & Apperley, M. Software for Students to Sketch Interface Designs. In: Rauterberg, M., Menozzi, M. & Wesson, J., eds. *Interact*, 2003 Zurich. Acceptance Rate, 73-80.
- Rubine, D. H. Specifying gestures by example. *Proceedings of Siggraph '91*, 28 July - 2 Aug 1991 Las Vegas, USA. ACM, New York, 329-337.
- Sharon, D. & Panne, M. V. D. 2006. Constellation models for sketch recognition. *Proceedings of the Third Eurographics conference on Sketch-Based Interfaces and Modeling*. Vienna, Austria: Eurographics Association.
- Shilman, M., Pasula, H., Russel, S. & Newton, R. 2001. Statistical visual language models for ink parsing. *Proceedings of the AAAI Spring Symposium on Sketch Understanding*.
- Vatavu, R.-D., Anthony, L. & Wobbrock, J. O. 2012. Gestures as point clouds: a \$P recognizer for user interface prototypes. *Proceedings of the 14th ACM international conference on Multimodal interaction*. Santa Monica, California, USA: ACM.
- Wobbrock, J. O., Wilson, A. D. & Li, Y. 2007. Gestures without libraries, toolkits or training: a \$1 recognizer for user interface prototypes. *CHI*. ACM.