RATA: Codeless Generation of Gesture Recognizers

Beryl Plimmer, Rachel Blagojevic, Samuel Hsiao-Heng Chang, Paul Schmieder, Jacky Shunjie Zhen

Department of Computer Science

University of Auckland

beryl@cs.auckland.ac.nz; {rpat088|hcha155|psch068|szhe024}@aucklanduni.ac.nz

Touch and stylus sensitive computer displays are widely available. Yet, the development of gesture sets to support these interaction methods continues to be difficult. We present RATA, a tool for interaction designers and software developers to create gesture recognizers for novel and custom gesture sets. Guided by the RATA wizard, the developer: defines their gesture set; collects example gestures; labels them with the support of an auto labeller; and generates the recognizer model file – no coding or expert knowledge of recognizers is required. Incorporating the recognizer into a program requires just two lines of code. Our evaluations show high user satisfaction and that novice software developers can design a customized gesture set and generate a recognizer in about 20 minutes.

Gesture based interaction, gesture recognition.

1. INTRODUCTION

Touch and stylus enabled hardware is increasingly available. This opens up many new interaction possibilities such as onscreen sketching and using gestures to trigger events. However many applications do not utilize these input modalities. An enabling technology that is missing is an easy to integrate, trainable and accurate gesture recognizer.

A gesture recognizer interprets the gestures made by a user, which can be for two distinctly different purposes: functional gestures and drawing. For example: simple recognizers in products like Apple's iPhone interpret gestures to scroll up and down a page and zoom in and out (functional gesture); sketch gestures are used to draw pictures and diagrams such as UML charts (drawing). If the meaning of the gesture is recognized by the computer, the software can provide sophisticated support such as intelligent editing and automated layout. In this way, gesture recognition can add richer and more natural interaction to applications.

While people are capable of intricate and precise control of tools, consider musicians' and artisans' control of their instruments, computer interaction is currently comparatively paltry. Hardware has been one of the limiting factors, but new touch and stylus displays can potentially afford much richer interaction. Another limitation is the software's ability to recognize user input. Standard recognizers, such as those imbedded in operating systems, are constrained to a limited set of predefined gestures. Thus interaction designers and software developers are limited in the gestures that can be used unless they can integrate a trainable recognizer.

There is still much to learn about designing gesture sets that are intuitive to people and recognizable by the computer. An accurate and quick to configure recognizer would enable designers to explore richer gesture interactions by supporting the quick design and testing of different gesture sets.

Numerous adaptable and configurable gesture recognizers are proposed in the literature. Some, e.g.(Fonseca, Pimentel et al. 2002; Paulson and Hammond 2008), are designed for a specific type or subset of stroke classes; previous evaluations have shown they do not work well for other sets of gestures (Schmieder, Plimmer et al. 2009). There are also a number of simple, trainable recognizers e.g. (Rubine 1991; Wobbrock, Wilson et al. 2007). These have limited accuracy and often constrain the user's drawing style. Some of these recognizers have been incorporated into toolkits in an attempt to make gesture recognition easier for non-experts.

Building an accurate recognizer continues to be a time-consuming, expert task. It requires detailed understanding of the data (i.e. gesture or input stroke features) and artificial intelligence techniques. General application programmers have neither the expertise nor the time to undertake recognizer construction. We employ an accurate gesture recognizer (Chang, Plimmer et al. 2010), that can work with both touch and pen gesture strokes, in a novel tool for designing, training and testing gesture sets.

The contribution of this paper is the extensions to the gesture recognizer (Chang, Plimmer et al. 2010). These include a wizard to guide interaction designers through the process to generate recognizers and an auto-labeller to reduce the tedium of manual labelling. Our evaluation shows

that novices can easily generate satisfactory recognizers using RATA¹.

In the following section we review related work on gesture recognizer toolkits. We then describe the target audience for RATA and provide a use case example. Following this is a description of RATA and an illustration of how it is used. We ran a usability evaluation of the RATA wizard and in addition, analysed the accuracy of the recognizers generated by the test participants. Finally, we discuss the contributions, limitations and future work intentions of this project.

2. RELATED WORK

The idea of having a toolkit for non-experts to generate gesture sets and recognizers is not new. In 1999 Long et al. (1999) proposed such a toolkit for designing gestures for common functional operations. They displayed the feature values computed by the recognizer (Rubine 1991) and highlighted when different gestures had similar values. Users could then adjust their gesture set to avoid ambiguities. Long et al. concluded that it is difficult for users to design a gesture set that is well recognized – particularly for people unfamiliar with recognizers.

The Magic tool (Ashbrook and Starner 2010) is similar in intent to our work but used for 3D motion gestures, while we focus on 2D gestures. Using dynamic time warping (Fu, Keogh et al. 2005) as the underlying algorithm, gesture recognition is based on automatically generated thresholds that users can alter to achieve better recognition. The toolkit aims to improve the experience of non-expert users.

GestureLab (Bickerstaffe, Lane et al. 2007) is a tool for generating domain specific gesture recognizers using Rubine's (Rubine 1991) 13 features and a support vector machine (SVM). Other sketch toolkits provided a fixed low level gesture recognizer for primitives (e.g. lines, arcs) and expose a way for users to integrate semantic recognizers to combine the primitives into meaningful glyphs. For example: LADDER (Hammond and Davis 2005) has an interface for users in writing a grammar, consequently knowledge of grammar semantics is required; InkKit (Plimmer and Freeman 2007) derives rules for joining gestures into diagram glyphs from user examples.

iGesture (Signer, Kurmann et al. 2007) is a framework for evaluating gesture recognizers. There are 3 main parts to iGesture. First is the test bench where the user can draw a single gesture (or use a gesture from an available database) and use it to test the accuracy of different recognition

algorithms. Second is the admin interface for managing data collection. Finally the test data function exports test datasets to XML to be used for batch evaluations of the data against different algorithms. This tool is limited to standalone gestures as opposed to complete diagrams.

Many of these earlier toolkits incorporate recognizers that require the gesture set designer to understand the strengths and weaknesses of the recognizer. With expert knowledge it is possible to design a gesture set where each gesture is distinguishable, but as (Long, Landay et al. 1999) showed, non-experts cannot achieve acceptable results. In summary there is no tool available that non-experts can use to easily design and test a custom recognizer.

3. TARGET AUDIENCE AND EXAMPLE USE CASE

The target users of this work are interaction designers of touch and stylus enabled devices, for example: touchscreen graphical tablets, hand-held touch devices and pen input enabled laptops. RATA assists interaction designers of applications for these devices in designing and implementing input gesture recognition, which can then be mapped to activate various functions. The actual process of generating a recognizer requires zero coding and the implementation of the resulting recognizer requires minimal programming knowledge, thus allowing interaction designers to stay focused on the user experience aspects of interactive design rather than the detailed implementation.

As an example a custom RATA recognizer was used in SketchSet (Figure 1), a tool for Euler diagrams (Wang, Plimmer et al. 2011). In this project, a RATA recognizer is used to discriminate between ellipses, circles, blobs and text. The reported recognition rate is ellipses 100%, circles 96.15%, blobs 96.15% and text 100%.

Figure 1: Euler diagram with circle, ellipse, blob and text labels drawn with digital stylus (left) and recognized by RATA.Gesture (Chang, Plimmer et al. 2010) for automatic formalized layout (right).

Games on handheld touch devices can also benefit from RATA generated recognizers by enabling designers to explore and create game context

¹ Recognition Algorithm Tools for ink Applications

specific gestures, for example a driving simulator with specific vehicle control gestures.

Such examples require accurate recognizers to be constructed specifically for the context. While it is possible to develop individual recognizers tailored to specific domains, it is inefficient, time consuming, difficult, and results in inflexible recognizers that cannot be reused for other contexts. RATA solves these problems by providing an easy to use, quick and simple, iterative environment for interaction designers to explore gesture set alternatives and produce accurate gesture and diagram recognizers.

4. OUR APPROACH

Our work is similar in intent to that of (Long, Landay et al. 1999) which concluded it was very difficult for users to develop their own gesture recognizer. We employ a newer, more powerful recognizer generator and provide higher level user support to enable a more usable experience. We provide a wizard to guide the user through the process of creating a recognizer and a visual interface to play with the generated recognizer and heuristically investigate its accuracy. Furthermore we have packaged the recognizer so that it can be integrated into another program with just two lines of code.

In the following sections we first provide a brief overview of RATA: its component parts, the recognizer we employ, how this is packaged as a software component, and the wizard guiding process which structures the recognizer generation procedure. Next we provide a detailed description of the steps required to generate a recognizer using RATA.

5. RATA

RATA is a software toolkit designed for generating recognizers. It includes functionality for: data collection and labelling (Blagojevic, Plimmer et al. 2008); a large feature library (Blagojevic, Plimmer et al. 2011); RATA.Gesture recognizer generator (Chang, Plimmer et al. 2010); and an evaluation platform for comparing recognizers (Schmieder, Plimmer et al. 2009).

RATA integrates these components in a compact and logical manner by incorporating wizard support to the user interface. This results in an easy to use packaged tool for the purpose of recognizer generation (Figure 2).

The data collection component (Blagojevic, Plimmer et al. 2008) is designed to collect realistic data – in particular it supports drawing full diagrams instead of the isolated shapes of a diagram. For example, to collect examples for a flowchart diagram, the actual flowcharts will be obtained as a

whole, rather than drawing rectangles and arrowheads separately. This preserves the inter-gesture information between the elements of the diagram that is beneficial for sketch recognition (Field, Gordon et al. 2009). Data collection works the same regardless of input type; users may collect data samples from pen input or finger touch displays.

Figure 1: RATA component overview.

After the data is collected, labelling is required to provide the training algorithms with example data. Each stroke of a gesture set must be labelled; manually doing this is a laborious task. To reduce the effort required, we developed an auto labelling function that reduces the number of strokes which must be manually labelled (Zhen et al, 2012). The user is only required to label three or four examples of each class; the auto-labeller labels the rest of the unlabelled strokes using a recognizer generated on-the-fly from the labelled examples. The user then simply checks and corrects any errors in the auto labelling.

The features of the gestures are then calculated to form the training datasets. A feature is an attribute of a stroke that is encoded in a computer understandable way: for example the total length of a gesture, or changes of direction. These features may be either numeric or nominal values. There are currently 114 features in the feature library (Blagojevic, Plimmer et al. 2011). The dataset generator produces these features from the input data into a form usable by the recognizer for training.

The RATA.Gesture algorithm uses a training-based approach to generate customized recognizers for any 2D gesture set (Chang, Plimmer et al. 2010). It is an ensemble of four individually tuned algorithms from Weka (Hall, Frank et al. 2009): Bayesian Network, Logit Boost, Logistic Model Trees and Random Forest, combined into an ensemble using Vote also from Weka. Recognizers generated using RATA.Gesture have been evaluated against (Fonseca, Pimentel et al. 2002; Plimmer and Freeman 2007; Wobbrock, Wilson et al. 2007;

Paulson and Hammond 2008) using datasets from (Wobbrock, Wilson et al. 2007; Paulson and Hammond 2008) and a new data set, while (Paulson and Hammond 2008) and (Wobbrock, Wilson et al. 2007) marginally outperformed RATA.Gesture on their own datasets RATA.Gesture performed well on all datasets with an average recognition rate of 96.9%. RATA.Gesture is an excellent general gesture recognizer for interaction designers to use to explore gesture set design.

RATA also provides a wrapper DLL which lets programmers easily incorporate the trained recognizers into any .NET program. The model file produced by RATA.Gesture can be loaded using the DLL and enables gesture strokes to be passed to the generated recognizer for recognition. This is explained further in the following section.

5.1. Generating Recognizers using RATA

Users can create recognizers using RATA with minimal effort and no understanding of recognition techniques. To illustrate this simplicity, we present an example on how to use the tool to create a gesture set recognizer. This gesture set contains four kinds of gestures – refresh, delete, undo and redo. Throughout the process, the user is guided by the wizard tab on the right hand side of the window, which offers instructions and tips while structuring the process linearly.

After defining the project (Figure 3) there are four further steps to generate the recognizer: collect example data (Figure 4), label it (Figure 5), generate features from the data (Figure 6) and then use the generated features to train a recognizer (Figure 7). While the wizard guides the interaction designer in a linear process, it is possible to iterate and make modifications. For example, one might collect a small data set to 'have a go' and then after experimenting, collect additional examples to generate a more accurate recognizer.

Figure 2: Defining a diagram or gesture set.

5.1.1. Data collection (Figure 3)

Interaction designers can have multiple participants contribute to the gesture library, thus ensuring a more style tolerant recognizer, by providing custom instructions for drawing on the drawing canvas. Each participant has a separate canvas for each drawing; this simplifies the process of organizing the data. Evaluations show that about 20 examples of each class will produce a near-optimum recognizer for RATA algorithms (Chang 2010) and other trainable recognizers such as (Rubine 1991).

Figure 3: Data Collection

5.1.2. Data Labelling (Figure 4)

Training a recognizer requires examples for which classification is known. The auto-labelling function is provided to reduce the labour involved in manually labelling each stroke. The user first manually labels a subset of each gesture class using the interface shown in Figure 5. It is similar to the data collection interface, but instead of showing the diagram description, the list of classes defined during the initialization is displayed. A stroke (or strokes) can be selected by being clicked or lassoed and the corresponding label is selected from the list on the right. The stroke's colour is changed to indicate its label. The user can then activate the auto-labelling function which will label remaining unlabelled strokes based on the manually labelled examples. The user is able to correct any mistakes the auto-labeller makes and adjust options for more accurate auto-labelling.

Figure 4: Data Labelling

Figure 5 shows the result of first manually labelling some of each class (from top-down: refresh, delete, undo and redo gestures) and then applying the auto-labeller for the rest. One of the undo gestures has been wrongly auto-labelled as redo and can be

manually corrected. Note that the recognizer is normally able to differentiate the similar looking redo and undo gestures due to the difference in starting point and direction of the strokes (undo being right to left, redo left to right).

5.1.3. Feature generation (Figure 5)

Labelled strokes are ready to be converted into feature vectors. Users can select any subset of features to be used; for novice users who are unfamiliar with individual stroke features, two default subsets are provided. The first is for diagram recognition and the second is for functional gesture recognition. The latter excludes features that consider the spatial or temporal context of a gesture in relation to other gestures (e.g. time since last stroke) as it is assumed that each gesture is independent (i.e. not part of a larger diagram).

Individual participants can be removed through the list of participants. This functionality is implemented to exclude outliers or reserve some data for testing. The generated feature file is saved in CSV format.

5.1.4. Recognizer generation (Figure 6)

To generate a recognizer the user first selects the feature file (prepared in the previous step) and then simply clicks on the button to produce a new recognizer. This recognizer is saved as a Weka (Hall, Frank et al. 2009) format .model file.

The recognizer generation interface also has a drawing canvas to provide quick feedback on created recognizers. When the user loads a recognizer the information panel on the left displays the component names and colours assigned to them. Any strokes drawn on the central canvas are immediately recognized and colour coded. More thorough tests can be conducted via RATA's Evaluator interface (Schmieder, Plimmer et al. 2009).

Figure 5: Feature Generation

Adding the recognizer to a .NET program is very simple. The generated recognizer is utilized by including the RATA DLL into the program. The RATA DLL exposes an API to the recognizer. Users can perform recognition with two lines of code (Figure 7).

Figure 6: Recognizer generation and informal test panel

By providing the location of the generated model file, the first line loads the recognizer (this is a once only operation). The function "classifierClassify" classifies the given gesture and returns a result string. There are two classify methods, one classifies a single gesture, the other a collection of gestures. RATA takes an average of 0.087 seconds to classify a stroke on an Intel® Core™2 Duo Processor E8400 and 4GB of RAM.

Overall, RATA simplifies the process of generating a recognizer. All details of feature generation and recognizer training are hidden from the user while still providing the optional ability to tweak certain aspects such as selecting or omitting stroke features.

```
1. c = ClassifierCreator.GetClassifier("../rata.model");
2. string result = c.classifierClassify(targetGesture);
```

Figure 7: Code for loading the generated recognizer and recognizing a gesture.

6. USER EVALUATION

Our evaluation is primarily focused on the usability of RATA. The main goal is to show that generating a recognizer for new domains with RATA is an easy task, achievable without any background or technical knowledge of the recognition domain. We conducted an observational user study where eight programming students created their own custom recognizers. Each participant followed the steps set out by the RATA wizard: define a new domain, collect and label data, generate the feature set and train the recognizer, and heuristically evaluate their recognizers by drawing examples of their gesture set in the informal test panel.

6.1. User Testing Methodology

We conducted a pilot test with one participant to validate the test protocol and to determine whether the instructions were appropriate. This was followed by individual sessions where each participant worked alone. We collected information on participants' prior experience and opinions about the tasks using a questionnaire.

The questionnaire's first section dealt with the participants' familiarity with: digital stylus input, gesture based input and general programming skills. In the next section, participants were asked to comment on the evaluation task and the RATA environment such as task comprehension, effectiveness of RATA in assisting task completion, ease of use and other standard usability questions. Most questions were presented on a five-point Likert scale (strongly agree, agree, neutral, disagree, strongly disagree), with an open ended question and comment section at the end.

Before starting the tasks, participants were given a five minute demonstration of how to generate a recognizer, and asked to complete the first part of the questionnaire. In addition we provided a full tutorial on the recognizer creation process on a paper hand-out. Participants worked through the hand-out using a touch screen and pen input enabled Dell XT2 tablet running Windows 7, RATA and Morae usability testing software for screen recording.

Each participant had to generate two recognizers; one for a diagram and one for functional gestures. We ensured they understood the difference between the two by describing an example of each. Participants were responsible for designing their own custom diagram and gesture sets and subsequently provided all the training examples for the recognizers of these designs. Each participant followed the instructions and wizard as described previously, to first generate a recognizer for a drawing diagram set and then repeated the process for a gesture set. We fixed the order as it is more intuitive for most people to design diagram sets and we expected the experience with this would help them to design the gesture set.

We only imposed two restrictions for the user designs; one is that each diagram or gesture set must include at least five different classes in order to provide a realistic recognition task for the recognizers. The other restriction is that each class must be drawn with a single stroke, as RATA.Gesture is a single stroke recognizer. Otherwise users were open to explore various gestures and diagram semantic representations. Some examples of the participants' gesture and diagram sets are shown in Figure 8.

For drawing samples we asked participants to use pen input and for functional gestures we asked for finger touch input. Each participant was responsible for creating all of the data for each recognizer and we asked them to include at least 15 examples of each diagram or gesture class as an adequate training set for an accurate recognizer. For labelling, we asked the participants to use the

Figure 8: From top to bottom, P1 and P7 diagrams and P8 and P2 gesture sets designed and implemented during the evaluation. Colours identify stroke classifications.

auto-labelling function. First manually labelling about five examples for each class and applying the auto-labeller for the rest. The participants then manually corrected any mistakes made by the auto-labelling process.

After labelling was completed the participants generated a recognizer and played with it in the informal test panel. They were encouraged to stress the recognizer's accuracy threshold by drawing shapes in unfamiliar ways to the training set or drawing variable sized versions of the same classes. They repeated the process for the gesture recognizer.

At the end of the evaluation the participants were asked to complete the rest of the questionnaire, including the open question and comments. The participants were also asked to think about and comment on the accuracy of their recognizer, particularly on what they thought affected the success rate of recognition.

6.2. User Testing Results

Eight students, one female and seven male, participated in the study. The participants came from a mixture of undergraduate and postgraduate computer science and engineering backgrounds, thus guaranteeing familiarity with programming. A summary of their self-ratings of existing skills and knowledge is shown in Table 1.

Table 1: The students' familiarity with digital stylus, touch and general programming skills on a five-point Likert scale.

Question	Stylus	Touch	Programming
Mean	2.50	2.25	3.37
SD	1.07	1.03	1.19

All participants successfully completed the tasks. Participants reported positively for task comprehension ($M = 4.75$, $SD = 0.46$) and enjoyment ($M = 4.37$, $SD = 0.52$), tying in with understanding of the purpose of RATA ($M = 4.63$, $SD = 0.517$).

Participants strongly agreed with the ease of recognizer generation ($M = 4.38$, $SD = 0.74$) and the helpfulness of RATA in accomplishing that goal ($M = 4.88$, $SD=0.35$). The program functionality was also reported to be easy to use ($M=4.25$, $SD=0.71$) and the process was intuitive ($M= 4.5$, 0.53). This positive feedback was further reflected in open ended comments, with answers such as "very interesting and helpful program", "ease of which a concept can be taken through to a working recognizer" and "it was quite intuitive and easy to understand".

Participants also rated very highly the helpfulness of the auto-labelling function in making labelling easy ($M=4.75$, $SD=0.46$), and the overall satisfaction with the accuracy of the generated recognizers was high ($M=4.38$, $SD = 0.52$).

On average, participants took 36 minutes to complete the tasks (ranging from 30 to 44 minutes). They created their diagram recognizer first, in an average of 21 minutes and the gesture recognizer second, in an average of 15 minutes.

We also investigated the ability of RATA in helping to identify problem gesture sets where the accuracy for some gestures is lower than expected. Previous research by (Long, Landay et al. 1999) provided a view into the raw attribute data of gesture classes computed by the recognizer, highlighting similarities between gesture classes, which explicitly suggested the gesture designs were too similar. Long et al. concluded this to be confusing to the user and not as effective as they envisaged.

RATA provides a visual approach to identifying and correcting gesture design problems. The first indicator is the auto-labeller function which makes a best effort attempt in auto-labelling training strokes. From this result, the user is able to see an intermediate recognition result of their gesture strokes which identifies problem classes that are consistently being confused with one another. Participants reported that the auto-labelling function helped to identify gestures/diagrams that were too similar ($M=4.13$, $SD = 0.99$).

The second indicator is when a recognizer is generated and the user is able to gauge its immediate accuracy in the informal test panel. The user is again provided with visual feedback on recognition results and on classes that may be consistently confused with each other. When asked to comment on how to improve their recognizers, participants were able to identify the problems: "Shapes were too similar", "variability in the way I draw them" and provide solutions: "make the shapes more distinguished and also draw them in different ways", "Maybe not have shapes drawn similarly together", "collect more samples".

Some of the criticisms and recommendations include: need for faster refresh of labelling results, improvements to the manual labelling interface, further "tweaks the UI" and navigational improvements. We addressed these concerns by changing the GUI layouts of windows of each component and optimising for performance. This included resizing the data collecting paint panel, changing the background and foreground window colours, resizing and changing text sizes and fonts, and moving button locations.

7. RECOGNIZER ACCURACY

Naturally, the ultimate purpose of a tool such as RATA is to generate an accurate recognizer for the gesture set. To get a more concrete idea of RATA recognizer performance on gesture sets that were quickly designed by non-experts we analysed the accuracy rate of test examples drawn in the informal test panel.

We used the test gestures participants' drew in the informal test panel as test data. Because we had asked the participants to 'stress test' their recognizers, the testing examples were cleaned of obvious mistakes and ambiguous gestures.

The test strokes drawn across all participants for the diagram task totalled 629, with an average of 15.7 stroke examples per class. The overall recognition rate from the participant generated recognizers was 97.6%.

For the functional gesture task, two participants could not get the hardware to register their touch gestures (this is discussed further in the next section). These two participants instead used pen input to train the gesture recognizer and in turn used pen input to draw test examples. Omitting these two participants, the total number of test example strokes drawn across the remaining participant gesture sets was 328 with an average of 10.93 stroke examples per class. The recognition rate was weaker than diagrams, at 94.36%. Table 2 summarizes these statistics.

Table 2: Summary of recognition result statistics

	Diagram	Gesture
Total Training Strokes	718	411
Avg Training Strokes per class	17.95	13.70
Total Testing Strokes	629	328
Avg Testing Strokes per class	15.72	10.93

The individual diagram gesture sets ranged from 90.49% to 100% and for gestures from 87% to 100%. **Error! Not a valid bookmark self-reference.** summarizes the recognition rates of each participant.

Table 3: Summary of individual participant recognition rates.

Participants	Diagrams %	Gestures %
P1	100.00	94.29
P2	97.78	91.38
P3	97.65	100.00
P4	98.82	87.00
P5	90.49	Omitted
P6	96.19	Omitted
P7	100.00	93.49
P8	100.00	100.00
Mean	97.62	94.36
SD	3.19	5.05

8. DISCUSSION

The goal of this project is to provide a toolkit that non-experts can use to generate a gesture recognizer that will produce satisfactory results. Although there have been many recognizers developed over the last few years, it is some time since a interaction designer's tool has been proposed. Long et al. (1999) noted in their study that non-experts had difficulties designing a gesture set that could be satisfactorily recognized. In contrast to their displayed feature values, which the user had to interpret, we visualize the recognition result.

In our study participants reported high user satisfaction. The average time for the recognizer generation process was less than 20 minutes and this included the time required to design the gesture set. In a real-world scenario we would expect that interaction designers would spend more time to iteratively design, evaluate and collect data for a gesture set. Yet RATA provides the ability to quickly get an idea of whether the gesture set is viable.

The indicative recognition rates for diagram and gestures are 97.62% and 94.36% respectively. We did not insist users improve their gesture designs or collect training and testing input of multiple users to increase drawing style tolerance. While we use the RATA.Gesture algorithm, RATA includes implementations of (Fonseca, Pimentel et al. 2002; Plimmer and Freeman 2007; Wobbrock, Wilson et al. 2007; Paulson and Hammond 2008) and a recognizer using any of these algorithms could be generated with RATA.

In comparison to (Long, Landay et al. 1999) where some of the participants had prior experience designing gesture sets, all of our participants were novices unfamiliar what makes gestures easy or difficult to differentiate. This resulted in some set designs being difficult for a recognizer to

differentiate (Figure 10). In comparison Figure 11 shows a gesture set where the classes were visually very distinguishable and provided higher recognition rates. Although it is not always the case that gestures that are visually similar to the human eye are classified similarly by the recognizer, visual similarity does suggest the stroke features generated will be similar and thus recognized similarly.

The approach we have taken with RATA makes it possible for interaction designers to easily explore different gesture sets and programmers to include an ‘out of the box’ recognizer in applications. With little modification RATA would also be able to support on-going training inside another program. This would certainly increase recognition rates as individuals do have unique drawing styles (Field, Gordon et al. 2009). This would allow not only software designers, but also end users, to create their own gesture sets.

Figure 9: Above: example of P4’s gesture set. Below: The most common misclassifications. Note the similarity between gestures ‘close’ and ‘back’.

A limitation of this study is that we have considered only single gestures. Diagram shapes (such as

rectangles) are often drawn in multiple strokes. The primitives that make up a shape can be combined by adding simple rules to the application program. We are currently working on an extension to RATA so that multi-stroke shapes can be recognized (e.g. a rectangle drawn in one, two, three or four strokes). This will remove the need to either restrict drawing practice or add hard coded rules.

RATA can be used to design drawing or functional gesture sets, the primary difference between them is that functional gestures are generally contextually independent – thus we have two predefined feature sets in RATA. We also evaluated functional and drawing gestures with their respective natural input methods (finger touch and stylus pen respectively) in order to gather more realistic data. Finger input is less precise than stylus input, with the point of contact being much larger than a sharp stylus point and there are differences in the capacitive screen hardware detection mechanisms. Therefore it is not prudent to mix data and compare results of gestures and diagrams.

Figure 10: Examples of sets with highest recognition rates (100%), P8’s ‘rocket’ diagram and P3’s ‘marking’ gestures.

The problem of two participant’s inability to use finger touch input is a curiosity. The touch screen did not consistently register their finger touches and therefore produced ragged multiple strokes for a

single gesture. We speculate it is a difficulty in the capacitive touchscreen hardware in registering the participant's electrical signal, but this is beyond the scope of the evaluation. We decided to proceed with the evaluation using pen input for gestures for these two participants but omitted their gesture data from the analysis.

9. CONCLUSION

The RATA toolkit provides interaction designers and software developers with a complete solution to generate gesture recognizers. It is quick and easy to learn how to generate a recognizer and this simplicity makes it easy to iterate through the design process. It is also simple to integrate the generated recognizer into a program. We used the SSR algorithm which is flexible and sophisticated, resulting in an accurate gesture recognizer for different types of gesture data, but other algorithms could also be used. Our evaluations show novice users are able to design gesture sets and generate accurate recognizers without difficulty and are very satisfied with the usability of RATA.

10. ACKNOWLEDGMENTS

This work is partially funded by a Royal Society of New Zealand Marsden Grant.

11. REFERENCES

- Ashbrook, D. and T. Starner (2010) MAGIC: a motion gesture design tool. Proceedings of the 28th international conference on Human factors in computing systems, Atlanta, Georgia, USA, 10-15 April, pp. 2159-2168. ACM, New York, USA.
- Bickerstaffe, A., A. Lane, et al. (2007) Developing Domain-Specific Gesture Recognizers for Smart Diagram Environments. GREC-IAPR Workshop on Graphics Recognition, Curitiba, Brazil, 20-21 September, pp. 145-156. Springer-Verlag, Berlin.
- Blagojevic, R., B. Plimmer, et al. (2008) A Data Collection Tool for Sketched Diagrams. Sketch Based Interfaces and Modeling (SBIM '08), Annecy, France, June 11-13, pp. 73-80. Eurographics, Aire-la-Ville, Switzerland.
- Blagojevic, R., B. Plimmer, et al. (2011) Using data mining for digital ink recognition: Dividing text and shapes in sketched diagrams. Computers & Graphics 35(5): pp. 976-991.
- Chang, S. H.-H. (2010). Applying Data Mining for the Recognition of Digital Ink Strokes. Computer Science. Auckland, University of Auckland. MEng: 176.
- Chang, S. H.-H., B. Plimmer, et al. (2010) Rata.SSR: Data Mining for Pertinent Stroke Recognizers. Sketch-based interfaces and modeling (SBIM '10), Annecy, France, 7-10 June, pp. 95-102. Eurographics, Aire-la-Ville, Switzerland.
- Field, M., S. Gordon, et al. (2009) The effect of task on classification accuracy: Using gesture recognition techniques in free-sketch recognition. CAD/GRAPHICS 2009 34(5): pp. 499-512.
- Fonseca, M. J., C. e. Pimentel, et al. (2002) CALL: An Online Scribble Recogniser for Calligraphic Interfaces. AAAI Spring Symposium on Sketch Understanding, Palo Alto, California, 25-27 March, pp. 51-58. AAAI Press, Menlo Park, California.
- Fu, A. W.-c., E. Keogh, et al. (2005) Scaling and time warping in time series querying. Proceedings of the 31st international conference on Very large data bases, Trondheim, Norway, 30 Aug - Sept 2, pp. 649-660. VLDB Endowment.
- Hall, M., E. Frank, et al. (2009) The WEKA data mining software: An update. SIGKDD Explorations 11(1): pp. 10-18.
- Hammond, T. and R. Davis (2005) LADDER, a sketching language for user interface developers. Computers & Graphics 29(4): pp. 518-532.
- Long, A. C., J. A. Landay, et al. (1999) Implications for a gesture design tool. CHI 99, Pittsburgh PA, 15-20 May, pp. 40-47. ACM, New York.
- Paulson, B. and T. Hammond (2008) PaleoSketch: Accurate Primitive Sketch Recognition and Beautification. Intelligent User Interfaces (IUI '08), Gran Canaria, Canary Islands, Spain, 13-16 January, pp. 1-10. ACM, New York.
- Plimmer, B. and I. Freeman (2007) A Toolkit Approach to Sketched Diagram Recognition. BCS-HCI '07, Lancaster, UK, 3-7 Sept, pp. 205-213. British Computer Society Swinton, UK.
- Rubine, D. H. (1991) Specifying gestures by example. Proceedings of Siggraph '91, Las Vegas, USA, 28 July - 2 Aug, pp. 329-337. ACM, New York.
- Schmieder, P., B. Plimmer, et al. (2009) Automatic Evaluation of Sketch Recognizers. Sketch Based Interfaces and Modelling (SBIM '09), New Orleans, USA, 1-2 Aug, pp. 85-92. ACM, New York.
- Signer, B., U. Kurmann, et al. (2007) iGesture: A General Gesture Recognition Framework. 9th International Conference on Document Analysis and Recognition (ICDAR), Curitiba, Brazil, 23-26 Sept, pp. 954-958. IEEE Computer Society Washington, DC, USA.
- Wang, M., B. Plimmer, et al. (2011) SketchSet: Creating Euler diagrams using pen or mouse. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Pittsburgh, PA 18-22 Sept, pp. 75 - 82 IEEE Computer Society Washington, DC, USA.
- Wobbrock, J. O., A. D. Wilson, et al. (2007) Gestures without libraries, toolkits or training: a \$1 recognizer for user interface prototypes. User interface software and technology, Newport, Rhode Island, USA, 7-10 Oct, pp. 159-168. ACM, New York.