Trainable Sketch Recognizer for Graphical User Interface Design

Adrien Coyette[1], Sascha Schimke[2], Jean Vanderdonckt[1] and Claus Vielhauer[2]

1Belgian Lab. of Computer-Human Interaction (BCHI), Information Systems Unit (ISYS)
Louvain School of Management, Université catholique de Louvain,
Place des Doyens 1, B–1348 Louvain-la-Neuve (Belgium)
{coyette, vanderdonckt}@isys.ucl.ac.be – http://www.isys.ucl.ac.be/bchi
2Department of Computer Science/ITI,University Otto von Guericke,
Universitätsplatz 2, – D-39106 Magdeburg (Germany)
E-mail: sascha.schimke@iti.cs.uni-magdeburg.de

Abstract. In this paper we present a new algorithm for automatic recognition of hand drawn sketches based on the Levenshtein distance. The purpose for drawing sketches in our application is to create graphical user interfaces in a similar manner as the well established paper sketching. The new algorithm is trainable by every user and improves the recognition performance of the techniques which were used before for widget recognition. In addition, this algorithm ay serve for recognizing other types of sketches, such as letters, figures, and commands. In this way, there is no modality disruption at sketching time.

1 Introduction

Designing the right User Interface (UI) the first time is very unlikely to occur. Instead, UI design is recognized as a process that is [19] intrinsically *open* (new considerations may appear at any time), *iterative* (several cycles are needed to reach an acceptable stage), and *incomplete* (not all required considerations are available at design time). Consequently, means to support early UI design has been extensively researched [20] to identify appropriate techniques such as paper sketching, prototypes, mock-ups, diagrams, etc. Most designers consider hand sketches on paper as one of the most effective ways to represent the first drafts of a future UI [1,10,13,14]. Indeed, this approach presents many advantages over other techniques like editing in an interface builder: sketches can be drawn during any design stage [14], it is fast to learn and quick to produce [20], it lets the sketcher focus on basic structural issues instead of unimportant details (e.g., exact alignment, typography and colors) [10], it is very appropriate to convey ongoing, unfinished designs [12,16], it encourages creativity [10], sketches can be performed collaboratively between designers and end-users [15], and last but not least, it is largely unconstrained [4]. This unconstraint character turns to be a fundamental aspect to preserve in sketching tools: if for any reason, this character is disrupted, the end user may be confused or disappointed. Van Duyne *et al.* [20] reported that creating a low-fidelity UI prototype (such as UI sketches) is at least 10 to 20 times easier and faster than its equivalent with a high-fidelity prototype

(such as produced in UI builders). What is also important is that lowering the design fidelity to sketches does not reduce the design capabilities to discover usability problems. Furthermore, the end user may herself sketch to initiate the development process and when the sketch is close enough to the expected UI, an agreement can be signed between the designer and the end user, thus facilitating the contract and validation.

The idea of developing a computer-based tool for sketching UIs naturally emerged from these observations [8,15]. Such tools would extend the advantages provided by sketching techniques by: easily creating, deleting, updating or moving UI elements, thus encouraging typical activities in the design process [19] such as checking and revision. Some research was carried out in order to propose an approach combining the best of the hand-sketching and computer-assisted interface design, thus providing mixed initiative support. Among these hybrid approaches we can identify two major streams of research: *sketching only* (only a support of sketching activities is provided without interpreting them) and *sketching+interpreting* (other tools do not want to loose the effort and attempt to produce as reusable output some code). The first tools category does not endanger the unconstraint character, but the second may introduce some unexpected problems.

In order to produce the output, the system has to proceed to an analysis of the information provided; storing the input provided by the designer is then insufficient. To this end, these tools proceed to an online recognition of the input and proceed to the construction of the corresponding UI. Through the following section we will mainly focus on this second category. We consider that current restriction on the technique used in the existing tools are too strong and could be improved to unleash the power of this approach, as the actual sketching tools do not take into account the sketcher's preferences: they impose the *same* sketching scheme, the *same* gestures for all types of sketchers and a learning curve may prevent these users from learning the tool and efficiently using it. This can appear a little bit in contradiction with the main statement that would like this approach to be as easy as paper. This is also underlined in two main goals of gesture-based tools [12]: "gestures should be reliably recognized by the computer, gestures should be easy for people to learn and remember".

In order to maximize the power of informal UI design based on sketches, the aforementioned shortcomings should be addressed. It is therefore expected that UI sketching will lead to its full potential, so as to offer the as much freedom as possible to the designer. In this paper, we consider a new kind of approach applied to SketchiXML [4] for the online processing based a combination of a multi-stroke gesture recognizer which has been developed for this purpose and the *CALI* library [6]. Indeed, most sketching tools, including *SketchiXML*, are based on a single recognition algorithm (typically, Rubine's algorithm [17]), using either a trainable gesture recognizer for gesture and shape primitive recognition or fuzzy logic for shape primitives only.

2 State of the Art

Drafting tools are used to capture the general information needed to obtain global comprehension of what is desired, keeping all the unnecessary details out of the process. The most standard approaches for such prototyping are the "paper and pencil

technique", the "whiteboard/blackboard and post-its approach" [20]. Such approaches provide access to all the components, and prevent the designer from being distracted from the primary task of design. Research shows that designers who work out conceptual ideas on paper tend to iterate more and explore the design space more broadly, whereas designers using computer-based tools tend to take only one idea and work it out in detail [8,15,19]. Many designers have reported that the quality of the discussion when people are presented with a high-fidelity (Hi-Fi) prototype was different than when they are presented with a low-fidelity (Lo-Fi) mock up. In Lo-Fi prototyping, users tend to focus on the interaction or on the overall site structure rather than details irrelevant at this level [20].

Lo-Fi prototyping offers a clear set of advantages compared to the Hi-Fi perspective [4], but at the same time suffers from a lack of assistance. For instance, if several screens have a lot in common, it could be profitable to use copy and paste instead of rewriting the whole screen each time. A combination of these approaches appears to make sense, as long as the Lo-Fi advantages are maintained. This consideration results two families of software tools which support UI sketching and representing the scenarios between them, one with and one without code generation.

DENIM [13,14] helps web site designers during early design by sketching information at different refinement levels, such as site map, story board and individual page, and unifies the levels through zooming views. DEMAIS [2] is similar in principle, but aimed at prototyping interactive multimedia applications. It is made up of an interactive multimedia storyboard tool that uses a designer's ink strokes and textual annotations as an input design vocabulary. Both DENIM and DEMAIS use pen input as a natural way to sketch on screen, but do not produce any final code or other kind of reusable output.

In contrast, SILK [10], JavaSketchIt [3], FreeForm [15,16], and SketchiXML [4] are major applications for pen-input based interface design supporting code generation. SILK uses pen input to draw GUIs and produce code for the OpenLook operating system. JavaSketchIt proceeds in a slightly different way than Freeform, as it displays the shapes recognized in real time, and generates Java UI code. JavaSketchIt uses the *CALI* library [6] for the shape recognition, and widgets are formed on basis of a combination of vectorial shapes. The recognition rate of the *CALI* library is very high and thus makes JavaSketchIt easy to use, even for a novice user. This library is able to identify shapes of different sizes, rotated at arbitrary angles, drawn with dashed, continuous strokes or overlapping lines, and use fuzzy logic to associate degrees of certainty to recognized shapes to overcome uncertainty and imprecision in shape sketches. FreeForm [15] only displays the shapes recognized once the design of the whole interface is completed, and produces Visual Basic 6 code. The technique used to build the user interface is based uses a trainable single stroke recognizer based on Rubine's algorithm [17] and dictionary for combining simple strokes into Visual Basic widgets and words. SketchiXML is another sketching tool based on the CALI library. It allows the designer to build the widgets in the same manner as JavaSketchIt, but provide coverage for a large set of widgets, and provide UI specifications instead of java. These specifications are written in UsiXML (User Interface eXtensible Markup Language – http://www.usixml.org) which are platform independent. This application is flexible and its behavior can be parameterized according to designer's preferences.

The aim of this work is thus to produce an improved version of SketchiXML so as to enable the construction of more complex widgets. Indeed the actual version based on the CALI library restraints the type of shape to be considered to a small set of shape primitive such as circle, rectangle, etc… Even if the number of widgets recognized is quite high due the possibility to build widget using a combination of more than 2 shape primitives, some widgets are still hardly "sketchable" in a natural manner. To this end we intend to develop a second type of recognition processing providing custom representations for the different kind of widget or part of widgets.

3 New Sketch Recognition Algorithm

As explained, additional to the shape recognizer based on the CALI library, we build a new, trainable recognizer to solve some of the problems of the existing recognizer, that were mentioned above. The main idea of the new sketch recognizer is to divide a hand drawn input into a sequence of line segments with a particular direction and to compare two of these sequences using the so called *string edit distance*. A similar approach has been successfully suggested in biometric user authentication, e.g. in [18].

3.1 Raw Data

The drawing input from a TabletPC, i.e. the information about the pen movement, is available as a sequence of 3-tuples (x_i, y_i, p_i), where x_i and y_i are the coordinates and p_i is the binary pen pressure. In our environment, the coordinates are available in units of screen pixels; the binary pressure is set to 1, if the pen tip is touching the drawing surface and set to 0, if the pen is lifted. While using the mouse instead of pen as drawing input device, the pen-down is simulated by pressing the left button.

3.2 Feature Extraction

The features to be extracted from the raw data are based on the idea, described in [7]. The drawing plane is superimposed with a grid and the freehand drawing input is quantized with respect to the grid nodes (Fig. 1). Each grid node has eight adjacent grid nodes and for each pair of adjacent nodes one out of eight directions can be given. So, from the sequence of successive grid nodes, a sequence of directions can be derived. This sequence can be coded using an alphabet {0-7}, each value representing one direction. This approach was first presented by Freeman in 1974 [7], where it was used for a compressed storage of line drawings. We utilize the sequence-like representation as our basis for sketch recognition, because it is a short description and location invariant description of complex drawing inputs. For each raw sampling point (x_i, y_i) ($i \in [1,\ldots,n]$ for a sequence of n raw sampling points) that closest grid node (qx_i, qy_i) is selected by the following equations:

$qx_i = round(x_i / w_g)$ and
$qy_i = round(y_i / w_g)$, where w_g is the grid width (Fig. 1).

From the sequence of successive grid nodes (qx_j, qy_j) resulting from sketch input, a

string of directions (coded as words out of $\{0\ldots7\}^*$) of adjacent grid nodes is build. If two or more successive raw sampling points are quantized as the same grid node point, then this grid node appears only once in the sequence. Depending on the grid width w_g and on the distance of the successive raw sampling points, it is possible for the respective grid nodes not to be direct adjacent to each other. In this case the gap can be filled by using the line algorithm of Bresenham [2].

The gap between two drawing partitions, i.e. the delay between a pen-up and the subsequent pen-down event can be coded with respect to the relative position of the last grid node (qx_j, qy_j) before the pen-up and the first grid node (qx_{j+1}, qy_{j+1}) after the pen-down. Dependent of the distance and the angle between (qx_j, qy_j) and (qx_{j+1}, qy_{j+1}), a different coding can be used to indicate the kind of gap. Using this method, it is possible to extract features from hand drawn inputs, which are represented as strings, consisting of codes, which describe the local direction of line segments in chronological order and the characteristic of gaps between drawing partitions.

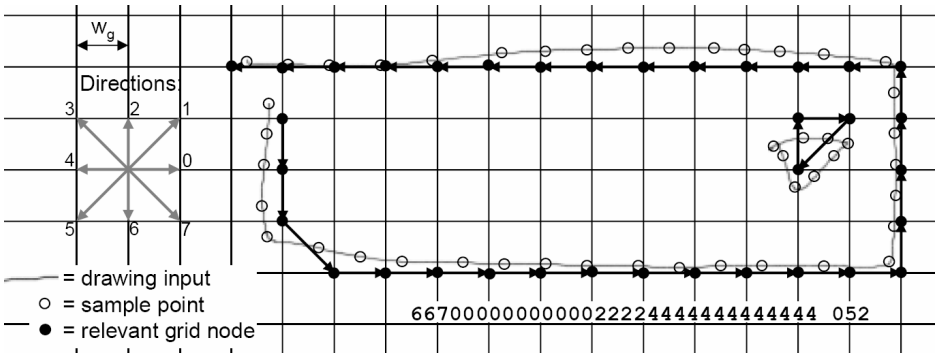


Figure 1. Square grid quantization of freehand shapes.

3.3 String Edit Distance

To compare two strings, a common technique is the so called string edit distance, as a measure of their dissimilarity. The idea behind this distance is, to transform one string into another string using the basic character wise operations *delete*, *insert* and *replace*. The minimal number of these operations for the transformation of one string into another one is called the edit distance or *Levenshtein distance* [Lev65]. The smaller the minimal number of needed edit operations for a transformation from string *A* to string *B*, the smaller is the distance between these strings. Instead of only using the number of operations, in some cases it is advantageous to use weights for the different operations. One possibility to determine the edit distance between two strings *s* and *t*, with *m* and *n* being the respective lengths, is to fill a matrix *D* of the size $m+1 \times n+1$ as follows [11]:

$D_{0,0} = 0$,

$D_{i,0} = D_{i-1,0} + w_D(s_i)$,

$D_{0,j} = D_{0,j-1} + w_I(t_j)$ and

$D_{i,j} = min\ \{D_{i-1,j} + w_D(s_i), D_{i,j-1} + w_I(t_j), D_{i-1,j-1} + w_R(s_i, t_j)\ \}$,

where s_i and t_j are the i^{th} and j^{th} elements of the strings s and t. $w_D(s_i)$ is the weight for removing operation of a code s_i, $w_I(t_j)$ is the weight for inserting a code t_j and $w_R(s_i, t_j)$ is the weight for replacing a code s_i by t_j. If s_i and t_j are equal, then $w_R(s_i, t_j)$ is zero. The value $D_{m,n}$ is the weighted edit distance of the strings s and t. For a better understanding of the procedure of this computation, we illustrate the resulting matrix in Fig. 2. It is obvious, that the complexity of the straight forward computation of the edit distance is $O(m \cdot n)$. For each matrix element $D_{i,j}$, the three adjacent elements at the left side and on top (marked in Fig. 2 by bold border) are required. In practice it can be shown, that the most relevant elements of the matrix D are those around the main diagonal, so the complexity can be reduced, if the grey fields are pre-initialized with an infinite value, so the min-clause of the calculation procedure considers stronger the more relevant elements around the main diagonal. Therefore, the computational complexity can be reduced to $O(b \cdot max\{m, n\})$, where b is a constant factor.

			t_1	t_2	t_3	…	…	t_n
		0	*1*	*2*	*3*	…	…	*n*
	0	0	1	2	3	…	…	…
s_1	*1*	…	…	$D_{i-2,j-2}$	$D_{i-2,j-1}$	…		
s_2	*2*	…	…	$D_{i-1,j-2}$	$D_{i-1,j-1}$	$D_{i-1,j}$		
…	…	…	…	…	$D_{i,j-1}$	$D_{i,j}$		
…	…							
s_m	*m*							$D_{m,n}$

Figure 2. Matrix D for computation of edit distance.

3.4 Sketch Recognition using String Edit Distance

As outlined above, the string edit distance can be utilized for the purpose of shape recognition using direction-based feature strings, extracted from hand drawn inputs. The idea is to have a repository, containing a set of reference shapes. For recognition, the unknown shape is compared with all shapes in the repository, i.e. the edit distance between the feature strings of the unknown shape and all reference shapes are calculated. The type of that reference shape, having the smallest edit distance to the unknown shape, is assumed to be the type of the unknown shape. Further, to avoid erroneous recognition of unknown shapes without a representation in the reference repository, a threshold for the maximal allowed edit distance has to be defined.

Due to the nature of string edit distance, the distance value at an average is dependant on the lengths of the strings s and t – the longer the strings, the higher is the average distance value. Therefore a kind of normalization is required. The best solution for considering the lengths m and n in the calculation of edit distance $D_{m,n}$ of two strings s and t is the following:

$$dist(s, t) = D_{m,n} / max\{m, n\}$$

A second method to normalize the string length impact is to "penalize" large differences in lengths of the two feature strings. It can be assumed, that only if a shape *S* is different from another shape *T*, the lengths *m* and *n* of the respective feature strings *s* and *t* are different. (The inversion is not true – equal lengths of *m* and *n* do not imply the equality of the shape types!) By introduction of the string length difference compensation factor the adapted distance could be calculated as follows:

$$dist(s, t) = d(m, n) \cdot D_{m,n} / max\{m, n\} \quad with \quad d(m, n) = max\{m, n\} / min\{m, n\}$$

The effect of $d(m, n)$ is to increase the edit distance by the degree of relative difference of string lengths. Finally, as a third improvement, it is possible to "penalize" the operations *replace*, *insert* and *delete* for the *gap* symbol. The idea is that normally the trained sketches in the repository have the same number of strokes (and consequently the same number of gaps) as the actual drawn shape. So, by using a large weight factor for these "gap operations", an amount of misrecognitions can be avoided.

The actual recognition of hand drawn inputs can be done by parallel using a set of different grid widths for the quantization while features string extraction. Here, for each single grid width setting, that shape from the reference repository is obtained having the smallest edit distance to the features in the corresponding grid size of the unknown input. So, for a set of different grid widths, a number of decisions for possible types of shape references can be achieved. From this set of decisions a degree of certainty can be derived by dividing the number of matches for each reference type by the number of decisions at all.

4 Integration into the Existing System

4.1 Implementation

As presented in [4], the SketchiXML's architecture is based on a set of collaborative agents where each agent is in charge of a specific part of the recognition/interpretation process. In order to meet previously elicited requirements, we have thus developed a new set of agents for the shape recognition process. Indeed, this role was held by a single agent in the first version. The new version is more sophisticated as several agents are participating in the process. A minimum of four agents are now participating in this process, two agents are providing the shape recognition for the shapes primitives and the gestures, a third agent is dedicated to coordination and the integration of the result of these two agents, and the last agent is responsible for dispatching this information to the system. The role of such agents is defined in [5] with the virtual mediator pattern definition. The *virtual mediator* defined in [5] is responsible for the following action:

- Decomposing the client request into sub requests, and then…
- Sending each of these sub requests to the relevant Service Providers.

When receiving the answer coming from each service provider, the mediator is responsible for:

- Integrating answers from the Service Providers to formulate final result, and then…
- Sending this result back to the Client.

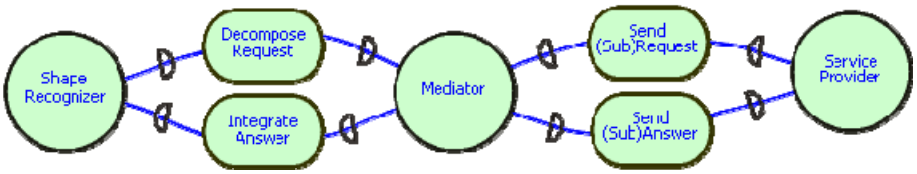
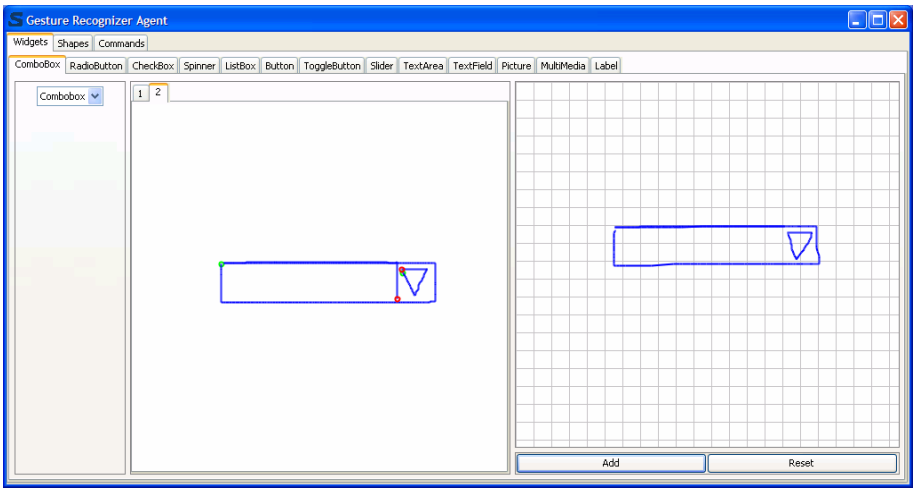


Figure 3. i* representation of Virtual Mediator.

In SketchiXML, the mediator would then be responsible to handle the data provided by the shape recognizer (agent collecting the raw data online) and to decide which agent to invoke. Even if both agents can be called simultaneously, the mediator can decide that only one of the agents is likely to provide the answer. As an example, if the designer is using a tablet pc and draws a sketch with the pen button pressed, then this sketch must be considered as command, and commands are only associated with gesture, it's thus useless to recognize it with the vectorial shape recognizer.

Another possible situation is the reception of a sketch to recognize by the mediator, but as a part of the current user interface. In this situation the mediator does not know the type in advance, as the sketch can be a vectorial shape or a widget. Then, the mediator sends a request to both agent and wait for their answers. If the answer provided by the first agent to reply has a very high degree a certainty then the mediator does not wait for the reply of the other agent and provide the result directly to the interpreter agent, otherwise the mediator wait for all the answers and select be most appropriate answer.



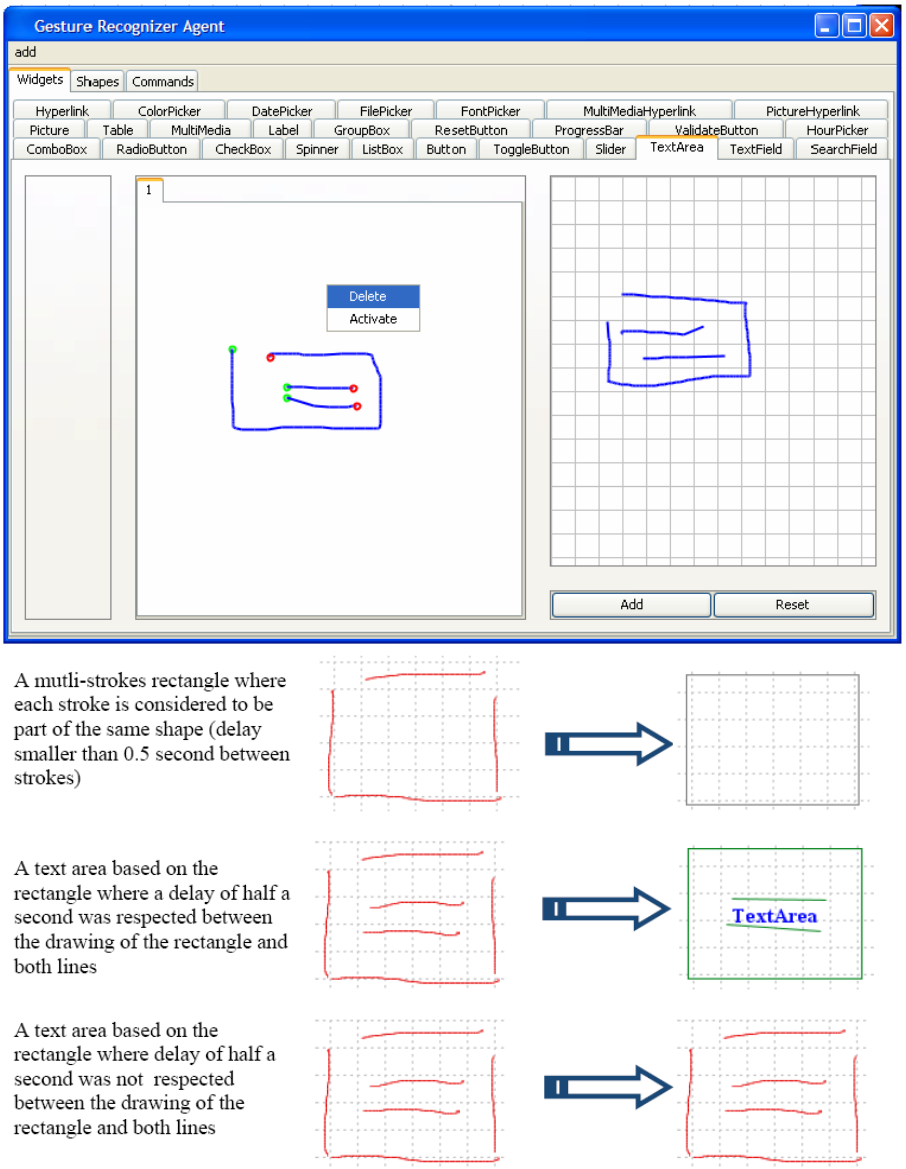


Figure 4. Management of user trainable shape references.

Moreover, the role of the gesture recognizer agent consists in three different tasks. Firstly the agent is responsible for managing the reference repository of hand drawn GUI widgets, geometric primitives and command gestures. To this aim the agent displays a training module (Fig. 4) allowing the designer to add, remove and visualize the current repository. The second task consists in processing the gesture recognition. So the agent is responsible for feature string extraction of hand drawn inputs from users, comparison of the feature strings with those in the reference repository. The last

task of the agent consists listing all the widgets and shapes candidate, annotated with a degree of confidence. This list is then send to the mediator agent presented previously in order to compare the results with the results of the shape recognizer agent, based on geometric primitives. This mediator implements a fusion strategy for the outputs of these two different recognizer modules.

4.2 Interpretation

Previously, when a new shape was recognized, the sketch was replaced by its corresponding vectorial shape if the recognition was enabled. But the extension to cover gesture is not straightforward. Indeed, with the previous version of SketchiXML it was quite natural to replace the sketch by its corresponding vectorial shape, as the sketch was supposed to be similar to its corresponding shape. But, using the gesture recognizer, such an approach does not hold. Even if there is no reason to provide a gesture for the triangle, that is completely different from a triangle, it is possible and the decision belongs to the designer. Another example that is more likely to happen, is a situation where the designer provides a gesture representation for a widget, then it is not possible to replace the sketch by its equivalent in term of vectorial shapes since this sketch may not contain any vectorial shape. The solution would be then to replace, the sketch by its corresponding widget or an informal representation of the widget. But on the other hand we want to keep the informal and unfinished aspect of the user interface, so as to encourage checking and revision. We have thus opted for an alternate solution that just consists in using the sketch provided without any transformation.

The leftmost part of Fig. 5 gives an illustration of a checkbox representation, where we can see that the shape recognized as vectorial shape look very sharp (the last widget) while the representation provided for the three first widgets look imprecise since they are based on the sketch provided by the designer. It is important to maintain this level of uncertainty in order not to give the impression to the end user that it is already a final UI. In contrast, the unconstraint character should be maintained throughout the recognition process without giving the impression that the level of fidelity suddenly changed. Fig. 5 shows other levels of fidelity for the same example.

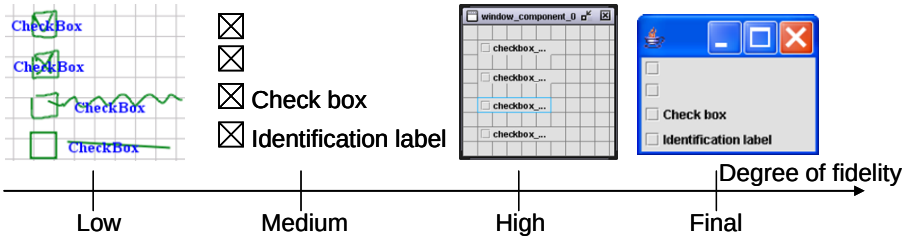


Figure 5. Sketched widget and related textual label.

Another problem faced with the interpretation of such gesture lies in their geometrical properties. When a new shape is handled the interpreter agent extract all the possible candidates from its knowledge set, in order to evaluate if a new widget can be built using this new shape. But, since the gestures do not have the same properties, the

constraints set had to be extended so as to cover the entire possible situations. Obviously, the solution adopted is less precise than the situation where only vectorial shapes are used. As an example, if a text area is recognized as a gesture and is displayed on the screen, then if the designer draws a horizontal line inside this widget, then the system should consider the line as a part of the text area rather than a new label. But, as long as the designer is free to define a custom representation of this component, we cannot predict the geometric properties of the widget, we have thus no other choice than to consider an approximation using bounding boxes coupled with Monte Carlo simulations.

5 Conclusion

Through this paper, we have presented an innovative contribution to the domain of sketch based design tools. Most of the existing tools only allow using mono-stroke gestures, and introducing, as a matter of fact, a strong constraint on the number of possible representations. We have proposed in this paper an alternative allowing to recognize multi-stroke gesture combined with the CALI library. Even, if a larger scale study would be appropriate to evaluate the benefit, the results observed shows better results, in any cases, than the previous version of SketchiXML based on the CALI library. Indeed, if the CALI library fails to recognize a scribble, the gesture recognizer may be able to recognize it since its processing is completely different.

Acknowledgements

We gratefully acknowledge the support of the SIMILAR network of excellence (http://www.similar.cc), the European research task force creating human-machine interfaces similar to human-human communication of the European Sixth Framework Programme (FP6-2002-IST1-507609) and the ReQuest project, funded by the Walloon Region (WIST 1).

References

[1] Bailey, B.P. and Konstan, J.A., *Are Informal Tools Better? Comparing DEMAIS, Pencil and Paper, and Authorware for Early Multimedia Design*, Proc. of the ACM Conf. on Human Factors in Computing Systems CHI'2003, ACM Press, 2003, pp. 313-320.

[2] Bresenham, J.E., *Algorithm for Computer Control of a Digital Plotter*, IBM Systems Journal, Vol. 4, No. 1, 1965, pp. 25-30.

[3] Caetano, A., Goulart, N., Fonseca, M., and Jorge, J., *JavaSketchIt: Issues in Sketching the Look of User Interfaces*, Proc. of the 2002 AAAI Spring Symposium - Sketch Understanding (Palo Alto, March 2002), AAAI Press, 2002, pp. 9-14.

[4] Coyette, A. and Vanderdonckt, J., *A Sketching Tool for Designing Anyuser, Anyplatform, Anywhere User Interfaces*, Proc. of 10th IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT'2005 (Rome, 12-16 September 2005), Lecture Notes in Computer Science, Vol. 3585, Springer-Verlag, Berlin, 2005, pp. 550-564.

[5] Do, T.T., *A Social Patterns Framework for Designing Multiagent Architectures*, Ph.D. thesis, Université catholique de Louvain, IAG, Louvain-la-Neuve, June 2005.

[6] Fonseca, M.J. and Jorge, J.A., *Using Fuzzy Logic to Recognize Geometric Shapes Interactively*, Proc. of the 9th Int. Conf. on Fuzzy Systems FUZZ-IEEE'00 (San Antonio, 2000), IEEE Computer Society Press, Los Alamitos, 2000, pp. 191-196.

[7] Freeman, H., *Computer Processing of Line-Drawing Images*, ACM Computing Surveys, Vol. 6, No. 1, 1974, pp. 57-97.

[8] Hong, J.I., Li, F.C., Lin, J., and Landay, J.A., *End-User Perceptions of Formal and Informal Representations of Web Sites*, Proc. of ACM Conf. on Human Aspects in Computing Systems CHI'2001, Extended Abstracts, ACM Press, 2001, pp. 385-386.

[9] Kolp, M., Giorgini, P., Mylopoulos, J., *An Organizational Perspective on Multi-agent Architectures*, Proc. of the 8th Int. Workshop on Agent Theories, Architectures, and Languages ATAL'01 (Seattle, 2001).

[10] Landay, J.A. and Myers, B.A., *Sketching Interfaces: Toward More Human Interface Design*, IEEE Computer, Vol. 34, No. 3, March 2001, pp. 56-64.

[11] Levenshtein, V.I., *Binary codes capable of correcting deletions, insertions, and reversals*, Doklady Akademii Nauk SSSR, Vol. 163, No. 4, 1965, pp. 845-848 [in Russian]. English translation in Soviet Physics Doklady, Vol. 10, No. 8, 1966, pp. 707-710.

[12] Long, A.C., Landay, J.A., and Rowe, L.A., *Implications For a Gesture Design Tool*, Proc. of ACM Conf. on Human Factors in Computing Systems CHI'2001 (Seattle, 2001), ACM Press, New York, 2001, pp. 40-47.

[13] Lin, J., Newman, M.W., Hong, J.I., and Landay, J.A., *DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design*, Proc. of ACM Conf. on Human Factors in Computing Systems CHI'2000 (The Hague, April 2000), ACM Press, pp. 510-517.

[14] Newman, M.W., Lin, J., Hong, J.I., and Landay, J.A., *DENIM: An Informal Web Site Design Tool Inspired by Observations of Practice*, Human-Computer Interaction, Vol. 18, 2003, pp. 259-324.

[15] Plimmer, B.E. and Apperley, M., *Software for Students to Sketch Interface Designs*, Proc. of INTERACT'2003, IOS Press, Amsterdam, IOS Press, 2003, pp. 73-80.

[16] Plimmer, B.E. and Apperley, M., *Interacting with Sketched Interface Designs: An Evaluation Study*, Proc. of CHI'2004, ACM Press, New York, 2004, pp. 1337-1340.

[17] Rubine, D., *Specifying Gestures by Example*, *Computer Graphics*, Vol. 25, No. 3, July 1991, pp. 329-337.

[18] Schimke, S., Vielhauer, C., and Dittmann, J., *Using Adapted Levenshtein Distance for On-Line Signature Authentication*, Proc. of ICPR'04, 2004.

[19] Sumner, T., Bonnardel, N., and Kallag-Harstad, B., *The Cognitive Ergonomics of Knowledge-based Design Support Systems*, Proc. of ACM Conf. on Human Aspects in Computing Systems CHI'97 (Atlanta, April 1997), ACM Press, 1997, pp. 83-90.

[20] van Duyne, D.K., Landay, J.A., and Hong, J.I., *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*, Addison-Wesley, New York, 2002.