SIACG 2002 - 1st Ibero-American Symposium on Computer Graphics
1-5 July 2002, Guimarães - Portugal

Sketching User Interfaces with Visual Patterns

Anabela Caetano Neri Goulart Manuel Fonseca Joaquim Jorge
Department of Information Systems and Computer Science
INESC-ID/IST/Technical University of Lisbon
R. Alves Redol, 9, 1000-029 Lisboa, Portugal
atc@mega.ist.utl.pt, nfcg@mega.ist.utl.pt,
mjf@inesc-id.pt, jorgej@acm.org

Abstract
This paper presents an approach to layout static components of user interfaces as hand-drawn compositions of simple geometric shapes, using sketch recognition and visual languages. The system uses a visual grammar built from drawings collected from users. We tried to understand how people sketch interfaces and what combinations of shapes they are more likely to used when sketching widgets. From there we implemented a prototype system, for creating user interfaces through hand-drawn geometric shapes, identified by a gesture recognizer. This prototype generates a Java interface, whose layout can be beautified using an a posteriori set of grammar rules (e.g. to align and group objects, etc.). We have conducted usability assessments with ten users to compare our approach with a commercial system (JBuilder). Besides a measurable speed advantage in drawing interfaces, users found our system more comfortable, satisfactory and efficient to use than the commercial product, as demonstrated by post–experiment questionnaires.

Keywords
Calligraphic Interfaces, Task Analysis, Usability Evaluation, Visual Parsing

1. DEVELOPING USER INTERFACES

Designing and coding the user interface represents a significant percentage of the total time spent when creating applications. Even though interface builders reduce the amount of time needed as compared to manual design, they focus on the final result, rather than allowing users to rapidly explore design ideas [9]. This emphasis on the final result inhibits the creativity of interface designers, because it suggests false commitment to a particular solution, discouraging users from exploring other alternatives. We believe, as other authors [12, 7], that better computer-based design tools should support sketching as the primary means to outline and diagram user interfaces.

Since paper and pencil are the designer's choices to quickly sketch new ideas and shapes, we try and approach this environment by proposing a visual method based on composing hand-drawn geometric shapes. In this manner, we exploit designer's natural ability at sketching and drawing.

Our approach combines the usability and flexibility of paper with the plasticity and interactivity of electronic media. To this end, we are exploring a new generation of visual interfaces organized around sketches and visual languages, which we call *Calligraphic Interfaces*.

The rest of the paper is organized as follows. In section 2 we describe related work about sketching user interfaces. The next section addresses fundamental distinctions between sketch–based and direct manipulation applications. Section 4 describes the task analysis performed to identify the "best" visual grammar. The next section presents the prototype architecture and describes the gesture recognizer, the visual language used to define widgets and the *a posteriori* grammar used to beautify the final result. Finally, we describe our experimental evaluation and we present some conclusions.

2. RELATED WORK

Calligraphic Interfaces predate some of the most established work in Graphical User Interfaces by many years. In 1963, Sketchpad, the first interactive system was developed by Ivan Sutherland [16] using a light pen to draw directly on a computer screen.

Wittenburg and Weitzman [17] presented an approach to automatic document layout based on parsing and syntax–directed translation through relational grammars. Their system included an interactive editor to capture document composition and layout styles through visual grammars.

Gross and Do [7, 8] describe a system based on sketches that are partially interpreted for use in architectural drawings, using a stroke recognizer considerably simpler and less robust than ours. Their work does not address visual ambiguity also. In general, work in this area does not prop-

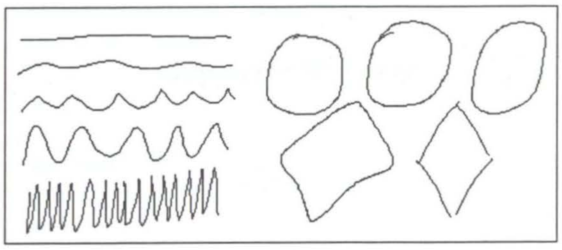

Figure 1. Ambiguity in sketched shapes

erly handle the ambiguous nature of visual input. Even when visual ambiguity is addressed, the solutions devised are often based on syntax manipulation or resort to imposing arbitrary thresholds on parameters.

Meyer describes EtchaPad [14], a tool for sketching widgets in the Pad++ system, including layout operations such as beautifying. Our system tries to accomplish much of this automatically, in the vein of ideas put forth by Pavlidis [15].

Landay [13] describes a sketch-based approach (SILK) for the early stages of user interface design. His work uses electronic sketching to allow designers to quickly record design ideas in a tangible form, obviating the need to specify too much detail too early in the process. His electronic sketches possess the advantages normally associated with computer–based tools, making them well suited to user interface design. While the main focus of our work is the static component of the interface, Landay focuses on dynamic behavior of interfaces, using storyboards to illustrate sequences between screens. SILK keeps the initial sketch visible while we identify the sketch and replace it by a beautified version (but not the final widget) while editing.

Even though Landay's tool supports the design of the static component of the user interface, it is not very clear how the grammar productions are defined for each widget and what parsing approach is used. Also, it is not clear how input errors and misrecognitions are handled, an ever–present issue in recognition–based applications.

3. SKETCHING DOCUMENTS

Most systems surveyed in the previous section combine sketch recognition with direct manipulation commands to achieve their results. In our approach direct manipulation commands are only used to carry out the most infrequent tasks (opening a file, saving, generating code, etc.). Almost everything else is achieved through gestures and sketch recognition.

However, this poses problems from a visual language parsing standpoint in that conventional parsing cannot be applied throughout the system. Indeed, visual grammars require that a start symbol be eventually "recognized" to accept its corresponding visual sentence. This is not adequate for interactive input, in which users construct many different individually valid visual sentences but no coherent whole (visual sentence) until the very last stages of a given design. The idea of partial grammars was suggested by

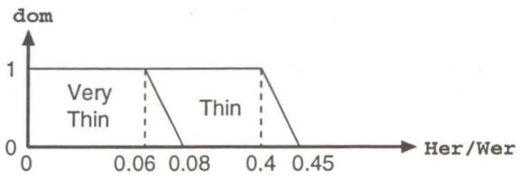

Figure 2. Fuzzy sets representing *Thinness.*

Wittenburg and Weitzman [17] who noted this fenomenon. In the same vein, we decided to use what we call *partial goals* or fragments of grammars that can be accepted separately. Parsing becomes as a two step procedure. To recognize shapes and combine these into widgets we use a *bottom–up* approach. To recognize higher–level constructs (and beautify layouts as side effect) we use a *top–down* pos–processing step before generating Java code.

One of the problems recognizing hand drawn sketches lies in identifying individual shapes. This requires resolving ambiguous shapes from a set of visually related symbols. Ambiguity exists, for instance, between `Rectangles` and `Diamonds`, `Ellipses` and `Circles`, `Lines` and `WavyLines`, as Figure 1 shows.

Composing geometric shapes using visual relationships and spatial constraints illustrates another source of ambiguity. Quite often the assessment of qualitative properties of shapes yields ambiguous results. An example, arises when we use the slope of a sketched line to classify it as vertical, oblique or horizontal. Traditional reasoning tends to assign one and only one meaning to each drawn line. However, hand–drawn sketches present less clear–cut choices, due to the sloppy nature of informal hand–drawings. Clearly, a more graduated approach is required to process these kinds of information.

Humans solve the natural ambiguity associated with sketched visual symbols, their properties and visual arrangements, by identifying more than one possible characterization and using context cues and external explanations to select one possible interpretation.

To address these problems we use Fuzzy Relational Grammars. These are described in [11] and in [1]. Their main advantage is to combine fuzzy logic and spatial relation syntax in a single unified formalism. We address imprecision and ambiguity both at the lexical and syntactical levels. At the lexical level, our recognizer uses fuzzy logic to deal with ambiguous shapes as illustrated in Figure 1. For example the production below describes how to identify `Lines`, regardless of rotation and scale:

```
IF Stroke IS VERY THIN
THEN
     Shape IS Line
```

Fuzzy grammar rules allow us to *quantify* attributes on objects such as `VERY THIN` through *Linguistic Variables* [18]. This particular instance describes the fact that lines are geometrically thin (i.e. their enclosing rectangles

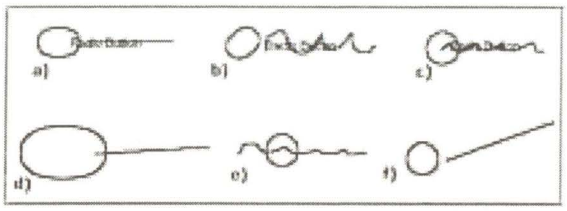

Figure 3. Valid and invalid sketches for RadioButtons

have a very low Height (Her) to Width (Wer) ratio). Figure 2 illustrates this. Contrarily to crisp logic rules which depend on arbitrary thresholds, linguistic variables allow us to model user and context variations in an elegant and flexible way.

We deal with visual ambiguity in a similar way, i.e. when it is not possible to univocally identify a shape, the recognition subsystem returns a list of possible candidates. The application can then choose the "best" candidate using context information.

Finally, at the syntax (grammar) level, fuzzy spatial relations allow us to deal with imprecise spatial combinations of geometric shapes. E.g. the rules below describe productions used to identify the `TextField` and `RadioButton` widgets.

```
IF X-Line INSIDE Rectangle
   AND Rectangle IS THIN
   AND X-Line IS HORIZONTAL
WHERE X-Line IS Line OR WavyLine
THEN
     Widget IS TextField

IF X-Line RIGHT-OF X-Circle
   AND Circle IS SMALL
   AND X-Line IS HORIZONTAL
WHERE X-Line IS Line OR WavyLine
   AND X-Circle IS Circle OR Ellipse
THEN
     Widget IS RadioButton
```

In the productions above the spatial relation `RIGHT-OF` maps spatial arrangements of objects into fuzzy sets to allow imprecisely drawn arrangements to be flexibly recognized. Figure 3 shows some of these arrangements, some of which are recognized (a, b, c) and some (d, e, f) are not. d) fails, because the Ellipse is not small. e) fails, because the `RIGHT-OF` relation does not hold, contrary to c) since the degree of spatial overlap is excessive. Finally, the line in f) is too slanted for the production to fire. Fuzzy logic allows us to percolate degrees of certainty from the right–hand–sides to the left–hand–side of productions and up a parse tree to reflect a "goodness–of–fit" of sketches to grammar entities. Recognition of compound objects happens as a side–effect of certain non–terminals being recognized by the parser. We can handle ambiguity at the user interface level by backtracking on the choice of symbols

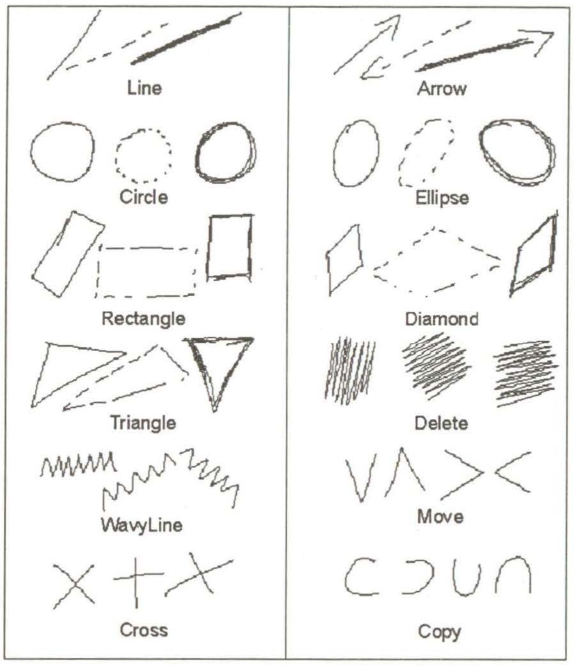

Figure 4. Shapes identified by the recognizer

although this currently only works at the recognizer interface.

Fuzzy Logic and parsing alone cannot guarantee success in addressing users' drawing styles. To identify the "right" visual grammars, we asked likely users of our system, as described in the next section.

4. STUDYING USERS and TASKS

Specifying user interfaces using a paper like interface can be fast, easy and natural. However, to capitalize on users' acquired experience using familiar metaphors and principles to make the interface simple and easy to learn, we must know how users sketch interface widgets. Only then we can define a visual grammar which provides a good match with users' mental models.

We started our task analysis by inquiring ten likely users of our system (undergraduate Comp Sci students) about the best way to represent a widget using a combination of sketches. The inquiry was composed of two parts. In the first, we asked users to sketch a representation for each widget without any restriction. In the second part, we asked the same thing, but now restricted to a set of geometric shapes (see Figure 4). From the responses we selected the two or three most drawn combinations of figures for each widget and we defined our visual grammar, which is depicted in Figure 5.

After this experiment we counted how many of the user–drawn widgets matched entries in our lexicon. The results are shown in percentage in Figure 6. A cursory examination of those results reveals that over half of the widgets were drawn in a way that matched our preset configurations at least 50% of the time. Further, we found out that

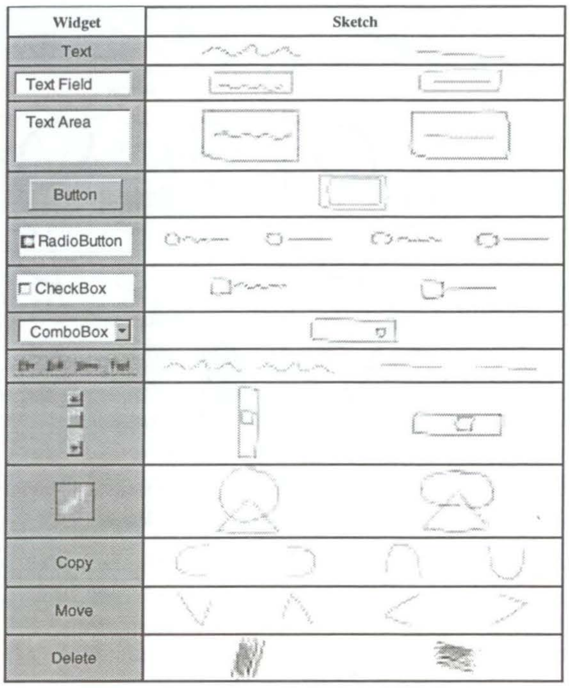
Widget
Sketch
Text
Text Field
Text Area
Button
RadioButton
CheckBox
ComboBox
Copy
Move
Delete

Figure 5. Visual constructs used to define widgets

this set is easily memorable in another experiment. Although we retained the lexicon thus developed for the remainder of this work, it is easy to add more productions to the basic grammar to accommodate different ways of drawing the same data. Moreover, most subjects commented that they appreciated the economy of strokes and gestures afforded by our vocabulary.

Even though some combinations from our grammar were different from users' selections, they found our constructs easy to learn. We consider this result to be encouraging, proving there is a reasonable "user–independent" set of representations. Another usability test conducted with four users found out that they retained over 90% of the "correct" shape combinations after one month not using our system.

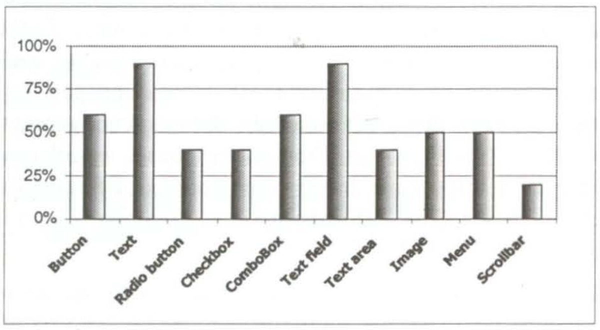

Figure 6. Percentage of sketches by users similar to our constructs

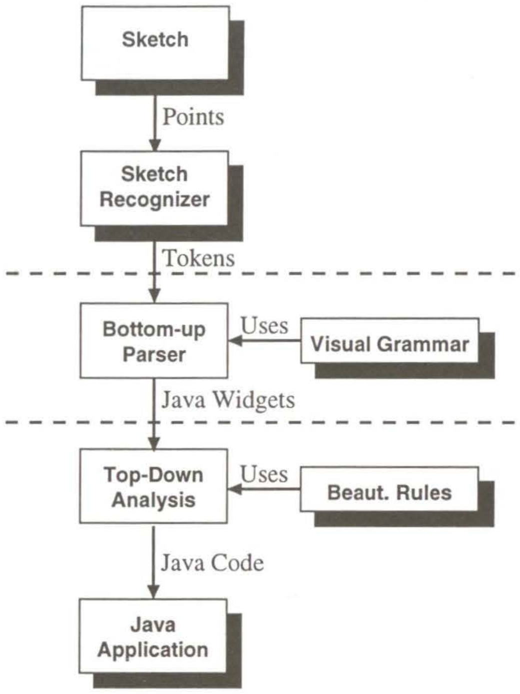

Figure 7. JavaSketchIt architecture

5. OUR SYSTEM

The prototype we built is able to identify ten different user interface widgets and supports three editing commands (copy, move and delete), using a subset of the gestures recognizable using CALI (see next section). To this end users sketch shapes using a pen on a digitizing tablet. The system tries to identify each shape as soon as it is drawn. After identifying a shape, the system tries to combine it with previously inserted shapes, using basic spatial and adjacency relationships.

The information about shapes and their spatial relationships is then fed to a bottom–up parser system, which uses basic knowledge about the structure and patterns of user interface layout to infer which graphic element was intended. The inference rules are grouped into a grammatical specification of a visual language for user interface design. In this visual language symbols are sketches built by combining simple shapes from a subset of those presented in Figure 4.

Finally, when the user requests to see the generated Java code, the user interface just sketched is submitted to a top–down beautification stage, which applies a set of beautification rules to group, align or center widgets.

The system components and their relations are depicted in Figure 7 and described in the following sections.

5.1. Individual Shape Recognizer

The individual shapes drawn by the user are processed by a simple shape classifier. This was developed using CALI [5, 6], a software library for the development of Calligraphic Interfaces, organized around a simple Shape Recognizer. The recognizer uses a fast, simple and com-

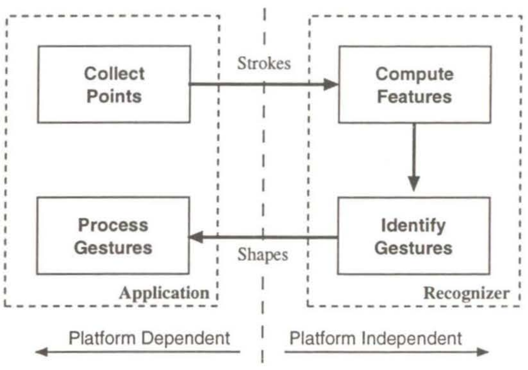

Figure 8. A simplified version of the CALI architecture

pact approach to identify Scribbles (multi-stroke geometric shapes) drawn with a stylus on a digitizing tablet. The method is able to identify shapes of different sizes and rotated at arbitrary angles, drawn with dashed, continuous strokes or overlapping lines.

The recognition process is based on three main ideas. Firstly, it uses entirely global geometric properties extracted from input shapes. Since we are mainly interested in identifying geometric entities, the recognizer relies mainly on geometric information. Secondly, to enhance recognition performance, we use a set of filters either to identify shapes or to filter out unwanted shapes using distinctive criteria. Finally, to overcome uncertainty and imprecision in shape sketches, we use fuzzy logic to associate degrees of certainty to recognized shapes, thereby handling ambiguities naturally.

This algorithm recognizes elementary geometric shapes, such as `Triangles`, `Rectangles`, `Diamonds`, `Circles`, `Ellipses`, `Lines` and `Arrows`, and five gesture commands, `Delete`, `Cross`, `WavyLine`, `Move` and `Copy`, as depicted in Figure 4. Shapes are recognized independently of changes in rotation, size or number of individual strokes. Commands are shapes drawn using a single stroke, except the `Cross` which requires two strokes. The recognizer works by looking up values of specific features in fuzzy sets associated to each shape and command. This process yields a list of plausible shapes ordered by degree of certainty.

The set of shapes selected and presented in Figure 4 are the basic elements to construct technical diagrams, such as electric or logic circuits, flowcharts or architectural sketches. These diagrams also require distinguishing between solid, dash and bold depictions of shapes in the same family. Some authors notice that the majority of diagrams were built of ellipses, rectangles and lines [2]. Another author [4] states that designers use a small set of symbols and that they share drawing conventions, so there is coherence between symbols used by different designers.

The recognizer has a recognition rate of 96%. It is fast: each scribble requires, on average, less than 50 ms (using a Pentium II @ 233 MHz) to be recognized, from feature vector computation to final classification. The fast response characteristic makes it very usable in interactive applications. We are currently working on a trainable version of the recognizer. The approach presented here extends and improves Kimura's work [2], which recognized a small number of shapes and did not distinguish rotated, open/closed, dashed and bold shapes.

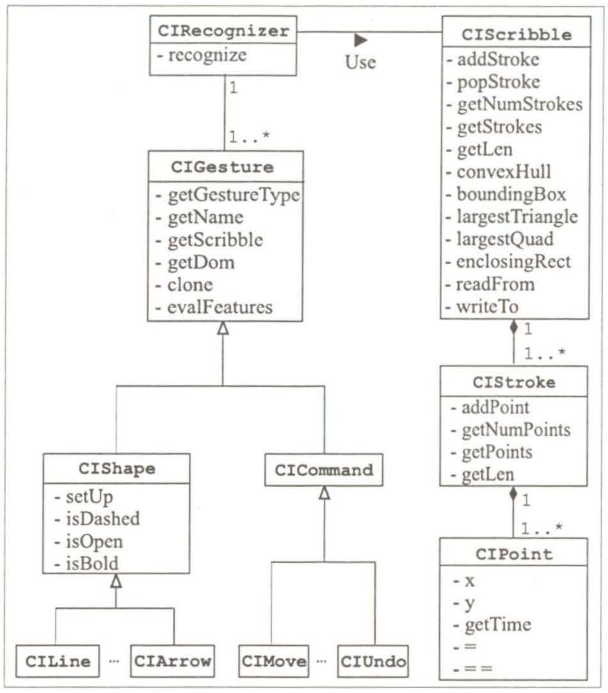

Figure 9. Class diagram

5.1.1. CALI architecture

The CALI library was developed to be platform independent. Actually there are two packages available [5], one for Linux and another for MS Windows.

Figure 8 shows the main blocks of the recognizer as well as the blocks to develop on the application side. One of the blocks, on the application side, is responsible for collecting the individual points of the strokes, while the other is responsible for receive and manipulate the gestures returned by the recognizer. The code developed on the application side is machine dependent.

The first block of the recognizer receives the sketch from the application and computes the corresponding geometric features. The second identifies the correct gesture or gestures based on the values computed before. The recognized gestures are inserted in a list order by degree of certainty, and returned to the application.

5.1.2. Class description

CIRecognizer Main component of the library, that interact directly with calligraphic applications, identifying hand drawn scribbles. This class implements a recognizer of geometric shapes and gesture commands based mainly on geometric information.

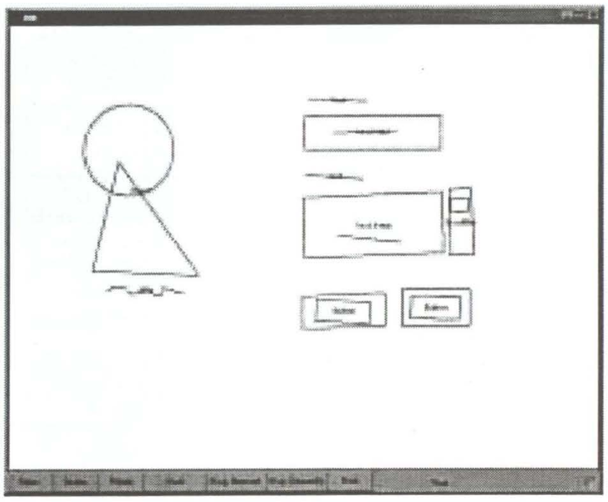

Figure 10. A sketch of an user interface

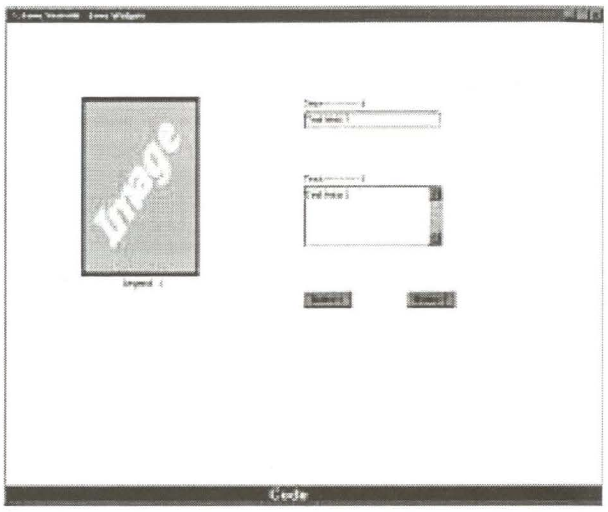

Figure 11. The user interface in Java

ClGesture Defines all the recognized entities, shapes and commands. The objects of this class have associated to them the original scribble and the recognition degree of certainty.

ClShape A Shape is a special case of gesture, that models all geometric shapes (Line, Circle, Rectangle, etc.). All instances of this class have attributes, like open, dashed or bold, and a geometric definition (a set of points).

ClCommand A special case of gesture, but without attributes nor geometric definition. Commands do not have a visual representation and usually trigger an action. In that they are different from shapes, since often there are not tokens in the grammar ascribed for commands. These tend to be directly executed after recognition. However, interface builders can choose to incorporate command processing in their grammars. We tend to look on this as a matter of policy, which should be the concern of the client application, rather than mechanism, which CALI provides.

ClScribble This class represents a scribble, which is build from a set of strokes. From a scribble we can compute some special polygons, like the Bounding Box, the Convex Hull, the Largest Triangle, etc., used during the recognition process.

ClStroke Defines a stroke composed by a set of points. It has methods to add points, to know the number of points, to get the points and to compute the total length of the stroke.

ClPoint Models a bidimensional point with a time stamp.

5.2. Visual Parsing

Individual shapes recognized by CALI, are then assembled by a bottom–up parsing procedure. A visual grammar defines rules of composition of individual elements as described in [11]. We use a simplified version of the bottom–up parsing approach described in [10]. With the exception of depictions of text widgets, all other widgets are defined by grammatical rules joining two tokens (recognized shapes). Whenever a shape is recognized, we check to see whether there is a grammar production that uses that figure. For each such production, we add an item to a list of candidate rules to fire. When the next shape is drawn by the user and recognized we check it to see if it matches the "missing elements" in one or more candidate rules. If it does, we check the constraints associated with each candidate rule that can fire. From here, three things can happen:

1. The newly sketched shape does not satisfy the constraints associated with any of the productions. In this case, we create a new candidate list for productions which contain this shape in their right–hand–side.

2. A single rule matches both the shapes and constraints. We create its corresponding widget and show its representation on the screen.

3. More than one production matches. For each production that can fire, we compute its degree of likelihood as the least of the degrees of likelihood of the right–hand–side symbols and the constraints associated with this production. The system then alerts the user that there are several candidates to match the input thus drawn and then shows the interpretation with highest degree of likelihood. The user can then either accept this interpretation or cycle through the other interpretations by taping on the figure displayed.

5.3. Beautification

Besides the composition rules defined for the interface layout, we also have a set of "beautification" rules to be applied *a posteriori*. These rules are defined also as grammar productions but they are parsed top–down rather than bottom–up (as the grammar rules that identify widgets and primitives).

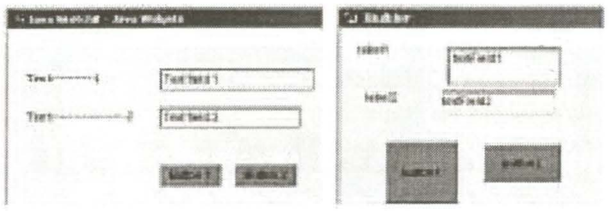

Figure 12. Simple task: Final result with JavaSketchit (left) and JBuilder

These rules are used to group widgets such as `radio buttons` or `check boxes` (if they are near each other), to align `text` and `text fields` or `text areas`, to center a caption below its associated `image` or to align `buttons`. Figure 11 shows the final result of an user interface sketched (see Figure 10) using our prototype, where the beautification rules were applied.

Again these rules exemplify the flexibility and expressive power afforded by Fuzzy Logic and approximate matching in that we can naturally express concepts such as "A and B are approximately aligned" or "B and C are approximately concentric" where A and B are right–hand–side symbols in a grammar production. The beautification occurs as a side–effect of parsing after the left–hand–side is reduced, by *enforcing* the spatial relationships to hold *exactly*, through changing attributes of right–hand–side items. Code generation happens also as a side effect of parse–tree traversal: the portion of the visual sentence which forms a correct visual sentence (i.e. is reachable from the start symbol) is traversed to produce Java code for each non-terminal node in its corresponding parse–tree.

6. EXPERIMENTAL EVALUATION

We performed an usability assessment to check user acceptance and performance of our prototype and to compare it with the JBuilder editor. The study was done by ten subjects (seven males and three females) using a pen-based computer SONY VAIO LX900, which features an integrated digitizing tablet and display screen. While the user sample is small, we believe that it adequately represents the target group (Computer Science students and interface developers). The goal of the experiment was to evaluate user acceptance and to check to see how quickly and accurately users would be able to create simple interface designs using our system as compared to a more conventional approach.

Our usability study was performed in three steps. First, a preliminary questionnaire allowed us to get some information about the users, their background and experience with pen-based interfaces and to see how they sketched each of the different widgets using the set of shapes shown in Figure 4. In the second part of the experience, users were asked to perform the same task using our system and JBuilder. To this end, we provided them with basic instructions about our prototype and about JBuilder. After the training stage, users were asked to draw the same user interface using both systems. We selected randomly which system should be used first, in order not to bias the timing results. Figure 12 shows the user interface that users were asked to create using both systems. On the left we see a typical interface created with JavaSketchit. On the right we see the a typical result achieved with JBuilder. As we can see, the results are not exactly similar, in that the layouts are qualitatively the same, but the figures differ in minor details. We decided to accept as a success user interfaces in which the correct elements were present and the spatial relations matched the drawing presented as an example. Since we wanted to evaluate ease of prototyping, we decided that minor details were not relevant. Moreover, users were happy with the possibility to have low–level layout details handled by the beautifier. We feel that automatic beautification accounted for about 50% of the performance advantage we witnessed. Finally, we asked test subjects to fill in a post-experiment questionnaire, to get feedback on diverse items such as, satisfaction, preferences, comparative advantages, disadvantages, etc.. During our study we measured the time needed to complete each task, the number of errors and the number of times a users had to consult

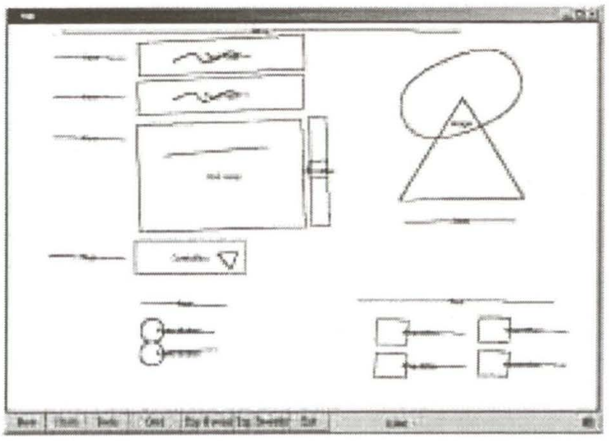

Figure 13. Complex task (sketch)

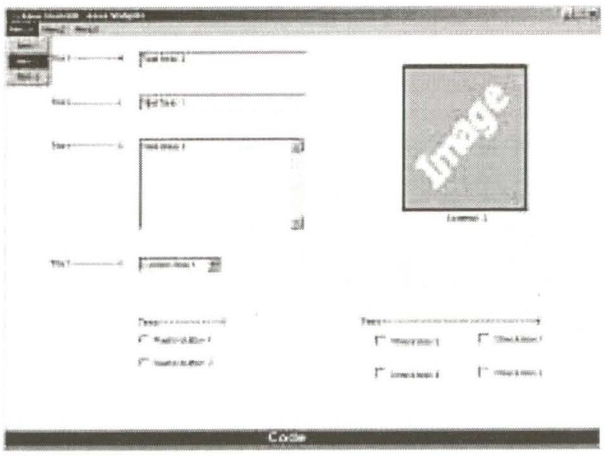

Figure 14. Complex task (result in Java)

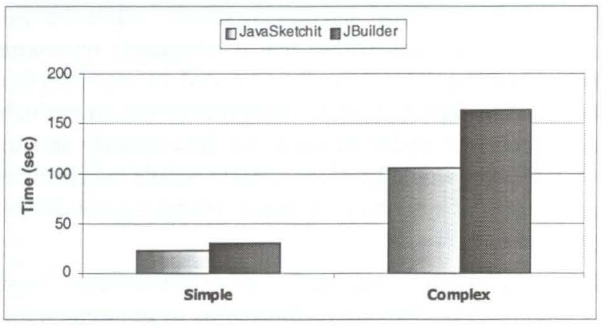

Figure 15. Time spent building simple and complex user interfaces

the manual.

From the first questionnaire we learnt that half of the users were not familiar with pen-based computers and that they usually design user interfaces. Additionally, we confirmed that our representation for widgets is reasonably familiar, since more than 50% of the sketches spontaneously drawn matched ours (see Figure 6).

The practical part of our usability test revealed that for simple tasks (see Figure 12) our system was as fast as JBuilder, but for more complex interfaces (as the one in Figure 14) users took less time to finish the task with JavaSketchIt. Figure 15 shows the time spent on each task using both applications.

We applied t–Student tests to experimental data and found out that the difference in average times to complete the simple task are not statistically significant. However, the differences for the more complex case are statistically significant with at least 90% confidence, that is our system performed significantly better than JBuilder for more complex tasks, which is encouraging even with the small population sample we used. Furthermore, by looking at Figure 12 we can observe that JavaSketchit seems to help building better–looking layouts than JBuilder with a comparable effort.

We also did a test to check the ease of learning (*learnability*) of our system. This test consisted in comparing the

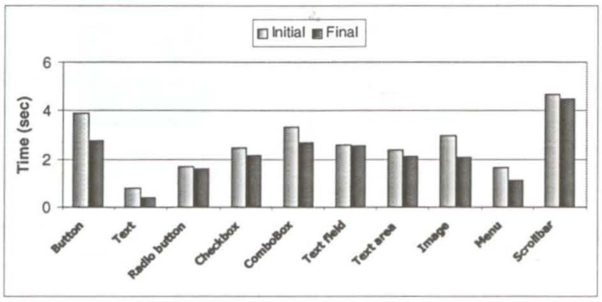

Figure 16. Time spent on the first task and last task, showing the learning evolution

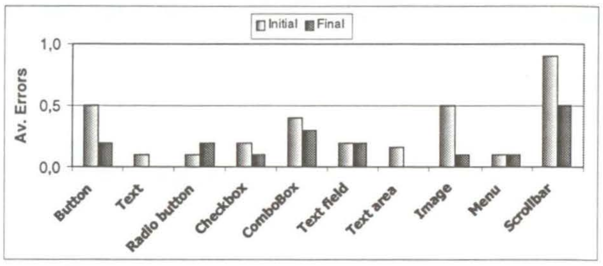

Figure 17. Errors made during the construction of an interface

time and the number of errors from Task 1 and Task 4. The results, as we can see in Figures 16 and 17, show that users took less time and made less errors in the last task. Though, after using our system just for a while, users seemingly had not trouble learning the visual grammar and were able to create user interfaces without too many errors.

Finally, the last questionnaire revealed that on a subjective assessment, users considered our application very easy to use and to learn and that the combination of figures (grammar) used to define widgets is reasonably understood by users, matching their expectations about half of the time, which we think is a good score. The comparison of both applications revealed that JavaSketchIt provides a familiar way of sketching interfaces and feels more friendly and simpler to use than its commercial counterpart.

7. CONCLUSIONS and FUTURE WORK

We described a prototype to sketch user interfaces using combinations of simple shapes identified by a gesture recognizer. Before specifying the grammar that defines widgets, we used task analysis to check how users combine shapes to draw widgets. From the results of this study we built our grammar and implemented a prototype. To validate our approach, we made a usability assessment and compared our system with a commercial product, JBuilder. The results shown that our system was very well accepted by users and that they felt comfortable using it. Further, our sketch–based approach was considerably faster than its commercial counterpart for the more complex drawings which we consider to be an encouraging omen for calligraphic interfaces. We plan to expand our system in the future to use richer shape combinations, improve interactive parsing of ambiguous constructs through backtracking at the parser level. Finally we want to extend the beautification approach by a richer set of productions, operators and the possible inclusion of constraint satisfaction mechanisms [3].

ACKNOWLEDGMENTS

This work was funded in part by the Portuguese Foundation for Science and Technology and the European Commission.

References

[1] Maria P. Albuquerque, Manuel J. Fonseca, and Joaquim A. Jorge. Visual Languages for Sketching Documents. In *Proceedings of the IEEE Symposium on Visual Languages (VL'00)*, Seattle, USA, September 2000.

[2] Ajay Apte, Van Vo, and Takayuki Dan Kimura. Recognizing Multistroke Geometric Shapes: An Experimental Evaluation. In *Proc. ACM UIST'93*, pages 121–128, Atlanta, GA, 1993.

[3] Alan Borning, Richard Lin, and Kim Marriott. Constraints for the Web. In *ACM Multimedia 97*. ACM, 1997.

[4] Ellen Y. Do. *The right tool at the right time*. PhD thesis, Georgia Institute of Technology, September 1998.

[5] M. Fonseca and J. Jorge. CALI : A Software Library for Calligraphic Interfaces. INESC-ID, available at `http://immi.inesc-id.pt/cali/`, 2000.

[6] Manuel J. Fonseca and Joaquim A. Jorge. Experimental Evaluation of an on-line Scribble Recognizer. *Pattern Recognition Letters*, 22(12):1311–1319, 2001.

[7] Mark D. Gross. The Electronic Cocktail Napkin - A computational environment for working with design diagrams. *Design Studies*, 17(1):53–69, 1996.

[8] Mark D. Gross and Ellen Yi-Luen Do. Drawing on the back of an envelope: a framework for interacting with application programs by freehand drawing. *Computers & Graphics*, 24(6):835–849, 2000.

[9] Marti A. Hearst, Mark D. Gross, James A. Landay, and Thomas F. Stahovich. Sketching Intelligent Systems. *IEEE Intelligent Systems*, 13(3):10–19, 1998.

[10] J. Jorge and E. P. Glinert. Online Parsing of Visual Languages Using Adjacency Grammars. In *Proceedings of the IEEE Symposium on Visual Languages (VL'95)*, Darmstadt, Germany, September 1995.

[11] Joaquim A. Jorge. *Parsing Adjacency Grammars for Calligraphic Interfaces*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York - USA, December 1994.

[12] J.A. Landay and B.A. Myers. Interactive Sketching for the Early Stages of User Interface. In *Proceedings of the Conf. on Human Factors in Comp. Syst. (CHI'95)*, pages 43–50. ACM Press, 1995.

[13] James A. Landay and Brad A. Myers. Sketching interfaces: Toward more human interface design. *IEEE Computer*, 34(3):56–64, March 2001.

[14] Jonathan Meyer. Etchapad - disposable sketch based interfaces. In *Proc. ACM SIGCHI'96 - short papers*, 1996.

[15] Theo Pavlidis and Christopher J. Van Wyk. An automatic beautifier for drawings and illustrations. In *SIGGRAPH'85 Proceedings*, pages 225–234. ACM, 1985.

[16] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *Spring Joint Computer Conference*, pages 2–19. AFIPS Press, 1963.

[17] Louis Weitzman and Kent Wittenburg. Automatic presentation of multimedia documents using relational grammars. In *ACM Multimedia 94*. ACM Press, 1994.

[18] Lofti A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.