

SKETCH: Modeling Using Freehand Drawing in Eclipse Graphical Editors

Ugo Braga Sangiorgi
IDEIAS Group, Departamento de Informática,
PUC-Rio
R. Marquês de São Vicente, 225
Gávea, Rio de Janeiro, RJ, Brasil, 22451-900
+55 21 3527-1500 ext. 4353
usangiorgi@inf.puc-rio.br

Simone D.J. Barbosa
IDEIAS Group, Departamento de Informática,
PUC-Rio
R. Marquês de São Vicente, 225
Gávea, Rio de Janeiro, RJ, Brasil, 22451-900
+55 21 3527-1500 ext. 4353
simone@inf.puc-rio.br

ABSTRACT

Nowadays, with the increasing popularity of touch-enabled devices, it is getting more common to give users the flexibility to insert information directly on the screen, using their fingers or a pen, instead of using a mouse. This project aims to support a more flexible modeling activity by enriching user's interaction with Eclipse graphical editors using the capabilities provided by touch/tablet devices.

Keywords

Sketching Recognition, Eclipse, Flexible Modeling Tools

1. INTRODUCTION

Eclipse is an Open Source platform, currently widely adopted for software development and modeling. Many graphical editors were built using the Graphical Editing Framework (GEF) [1], a framework created and maintained by the community that serves many purposes in creating graphical editors for specific domains – from GUI builders to business processes.

This paper presents a tool to support a more flexible modeling activity in Eclipse, enriching user's interaction with graphical editors by taking advantage from the capabilities provided by touch/tablet devices using free-hand sketching recognition.

Sketch recognition is the automatic understanding of any hand drawn input (using an electronic pen or a finger), and it has many uses from Computer-Aided Design systems to gestural interfaces. Despite the plurality of devices supporting hand drawn input, however, few softwares out of the graphic design domain (like 3D modelers and image editing) have successfully put this technology to practical use.

Software design and modeling could benefit as well from sketching [5]. We argue that, especially in initial stages of design, **informal** tools are often preferred to register and discuss ideas because they are more flexible. What is produced with them – mockups and intermediate models – is easily changeable. However, **formal** tools can provide modeling assistance, checking and provide a trace of

the decisions made at the design – which is too hard, often impossible to obtain otherwise.

We propose Sketch API, an Eclipse plug-in to allow developers to add sketching recognition capabilities to graphical editors built with Eclipse. With the API we are willing to approach the modeling task as a twofold process – one of freely sketching models with little interruption from the system; and another more formal, “classic modeling”, recognizing the elements drawn by the user right away, having as output a model in its canonical representation. Given the wide range and heterogeneity of graphical editors built with Eclipse, the main goal now is to try to define a common set of functionalities across them.

In the next section we give a brief overview of the modeling tools associated with the Eclipse platform that might benefit from sketching recognition and also what other tools have implemented sketching in a fashion similar to our approach. Section 3 will outline the directions of the API and also show the current state of the API's implementation. We conclude in section 4, also presenting plans for future work.

2. BACKGROUND

The Eclipse Graphical Editing Framework (GEF) was created to provide a layered architecture for graphical editing of models. With an editor, it is possible to do modifications to the underlying model – like changing elements' properties and model's structure, connecting and nesting elements [1]. All these modifications to the model can be handled in a graphical editor using common functions like *drag and drop*, *copy and paste*, and actions invoked from menus or toolbars.

It is worth to mention some examples of the wide range of tools built within GEF, such as the Eclipse Visual Editor¹ – a tool for visually building Graphical User Interfaces in Java (but also extensible for other languages); Zest² – a toolkit for graph visualization and manipulation; Eclipse UML2 tools and GEF3D [6] – a 3D editor using the same principles of 2D GEF.

There are many sketch-based modeling tools, often using sketching as a way of encouraging the freedom in early design phases such as Calico [5], which presents several adjacent workspaces in which the user can sketch an object and use it multiple times in a diagram; and DEMAIS [2], a tool for multimedia design, in which the designer is able to construct storyboards by freely sketching and using temporal operators.

None of the tools, however, seem to combine sketching, beautified graphical representation and association with an underlying

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

FlexiTools '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...\$10.00.

¹<http://www.eclipse.org/vpep>

²<http://www.eclipse.org/gef/zest>

metamodel. By adding sketching capabilities to Eclipse editors we are providing potential new ways of interacting and modeling in already existing environments.

3. SKETCH API

Our goal is to provide an API that not only allows users to perform gestures such as move, resize and rotate, and also to freely draw, connect and label the model elements, but also that supports a sort of intermediate modeling, also being as ‘invisible’ as possible regarding the recognition.

The modeling activity can be conducted in one of two modes: sketched to ‘beautified’ – with the former being less intrusive than the latter. In the sketched mode, the users might draw the elements whole model and then progressively “formalize” it. In ‘beautified’, the system would try to recognize the element instantly, maybe demanding the user’s attention and intervention more often.

The project is still in a preliminary state, and is a proposal under review as an official Eclipse project³. In its current state, it provides an API and an basic sample editor for shapes such as Squares, Triangles and Circles, but shapes made up of multiple strokes are also supported. The API is targeted at developers who wish to add sketching capabilities to GEF editors, doing so by means of a small set of configurations, in which it is possible to specify, among other things, the elements that should be recognized.

Traditionally, GEF editors use one tool for each element of the domain, thus the user must select a tool from the palette and click at the editor, adding a new element to the diagram. The goal of Sketch is to enable users to draw and connect the elements inside the editor’s area, using their own gestures to represent every type of element.

Thus, the Sketch API might be applied in a wide range of graphical editors, from editors with a large set of elements, in which the user has to search through the whole GEF palette in order to add them to the editor; to editors that only needs the gesture recognition feature.

3.1 Initial Implementation

This API is currently composed of three parts: a Sketch Tool, a Recognizer and a set of stored sketches. The *SketchTool* is adapted from GEF’s *AbstractTool* to be able to give visual feedback when a gesture occurs. Figure 1 shows how the tool registers the user’s clicks and drags and provides a visual feedback of what is being drawn on the editor (a). Once the sketch is finished, a thread at the tool passes the points to the *SketchManager* (b), which in turn processes those points and updates the editor (c).

Basically, the recognition is made in a chain of algorithms (A1 to An) arranged to take part in a decision, using the ‘Chain of Responsibility’ pattern [4]. The *SketchManager* has a series of recognizers, which in turn receives a list of points captured by the SketchTool, and then are periodically probed for a result. Each recognizer has a configured chain with a set of algorithms. If the first algorithm of the chain is not ‘sure’ about what the sketch means, it can delegate the decision to the next one on the chain, returning an element or nothing, if no element could be determined.

After the decision process is done, the *SketchManager* can compare the results and create the element or ask the user what he/she meant, if the result is not accurate enough, or if there is not enough confidence in it.

Currently, we have only one algorithm and only one recognizer setup, being able to detect shapes composed of single and multiple

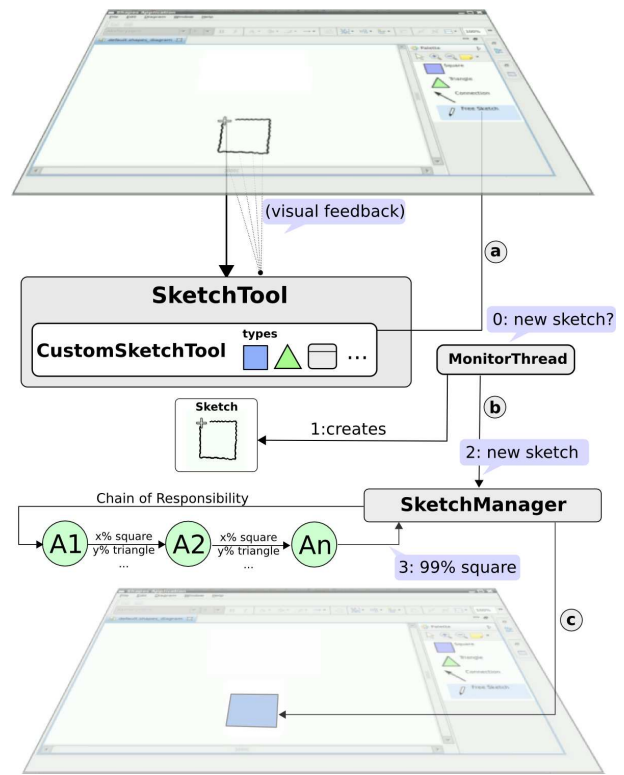


Figure 1: An overview of the current state of Sketch API.

strokes, as well as connections and dashed connections.⁴ The first algorithm on this single chain was made using an approach based on Levenshtein’s algorithm for string distance, based on the work described in [3]. It receives the user’s sketch transformed to a set of points and places them on a grid, then each point is transformed to a number according into the next point’s position.

For each element there is a set of stored words that describe its many forms, like drawn by the user. The recognizer compares each new sketch with the words of each element, calculating the distance between them – if the distance is smaller for a square than for a triangle, for example, then the element is probably a square.

Whenever it is impossible to determine which element was drawn, the *SketchManager* can ask the user, through a small dialog, what he/she meant by that sketch. Then the element is created and the information is stored for future use. In this way, the Sketch API offers a trainable recognizer that might evolve to have more sophisticated behaviors, using a background processing to do analysis of the database of stored sketches.

3.2 Towards Sketched vs. Beautified modes

The current implementation might evolve to the approach described before – offering the two modes of modeling. In either way the user would be able to train the recognizer using drawings as inputs and interacting with the recognizer to correct its behavior (through a dialog or the Properties sheet – an Eclipse widget that provides a list of editable properties of any selected element on an editor).

In **Beautified** mode, the sketch is merely a gesture recognition for the element’s ‘canonical’ representation. The user sketch the element and the *SketchManager* either creates it at the editor or

³<http://www.eclipse.org/proposals/sketch>

⁴As seen on the demo at <http://vimeo.com/2252782>

gives back an immediate response, through a dialog, asking which element is intended by that sketch.

The two modes might be interchangeable, since a more preemptive recognition is desirable in a situation where the recognizer's training is at focus, so the user could use the sketched mode later, with a higher level of confidence.

In **Sketched** mode, the figure of the intended element might be replaced by an SVG figure containing the user sketch, only 'anti-aliased' to contain fewer corners – that way the user would perceive some processing was done and there was a visible output regarding the recognition.

Whenever the element cannot be recognized, no anti-aliasing takes place and the sketch remains the same. The user might focus on the unrecognized elements later and, if wanted, specify which element was meant by the sketch.

By letting users draw the model using their own graphic representation, we are allowing the editor to be more flexible regarding its visual notation. For instance, assuming a simple model with the elements named *flower*, *sun* and *cloud* – they might be connected and generate an output or might serve as input for other models. A user might draw the flower element, for instance, in infinite ways, having any number of petals, with or without stem, and so on (figure 2).



Figure 2: The same element (flower) sketched in different ways.

An underlying model will be created, once the user signifies his/her drawing as “flower”, and it is possible to have the same model represented in many different ways.

It is important to note that this allows the creation of a “graphical model”, without predefined visual counterparts to the model elements, just elements and relations – the user would choose how elements will look like. Also, this approach can also be used to make annotations on existing models, since the user might be able to create an ‘annotation’ element.

On Eclipse frameworks side, some minor modifications will need to take place in order to allow editors to have such behavior. That will imply for example of underlying model to have a generic element to serve as an unrecognized element, to be created at the model while the user does not signifies it as an element, as well as to make the editor flexible enough to hold any graphic representation for it's elements. Fortunately, those are all widespread practices often implemented among Eclipse products.

4. DISCUSSION AND FUTURE WORK

Many graphical editors were built using Eclipse platform and its frameworks, however they rely on point-and-click paradigm only, missing perhaps new ways of interactions that are becoming popular nowadays, such as pen-based and multi-touch interaction.

Given the large amount of available editors built with Eclipse, we believe a sketch and gesture recognition API would benefit several developers, helping to increase their products quality, allowing their users to (a) freely represent the model using their own visual notation (b) use gestures to help on editing (c) make annotations on models.

However, given the heterogeneity of those editors' domains, it seems logical to find a common core so that to add pen or touch ca-

pabilities would fit the needs of all of the users. In order to explore the possibilities and be able to reach a wide number of graphical editors as possible, there are immediate steps planned, including:

- To define a set of plugins to take part the recognition chain, being customized to provide gestures, sketching and/or 'beautified' modeling in group or separately;
- Build a sort of benchmarking into the *SketchManager*, in order to be able to provide a quicker response on 'beautified' immediate recognition.
- To deal with recognition's degradation, when the recognizer it is not able to properly identify forms due to successively wrong inputs or when two or more sketched forms are too alike.

5. ACKNOWLEDGMENTS

The authors thank CAPES and CNPq (for grants 479167/2008-7 and 311794/2006-8), and their colleagues at the Semiotic Engineering Research Group at PUC-Rio for ongoing support and contribution to their research. We would like to thank Christopher Aniszczyk and Mariot Chauvin and all the initial interested parties on the project.

6. REFERENCES

- [1] *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM Corp., Riverton, NJ, USA, 2004.
- [2] B. P. Bailey and J. A. Konstan. Are informal tools better?: comparing DEMAIS, pencil and paper, and authorware for early multimedia design. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 313–320, New York, NY, USA, 2003. ACM.
- [3] A. Coyette, S. Schimke, J. Vanderdonckt, and C. Vielhauer. Trainable Sketch Recognizer for Graphical User Interface Design. *Human-Computer Interaction INTERACT 2007*, pages 124–135, 2007.
- [4] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.
- [5] N. Mangano, A. Baker, and A. van der Hoek. Calico: a prototype sketching tool for modeling in early design. In *MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering*, pages 63–68, New York, NY, USA, 2008. ACM.
- [6] J. von Pilgrim and K. Duske. GEF3D: a framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 95–104, New York, NY, USA, 2008. ACM.