Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 224 (2009) 41–46
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Dynamic Evaluation Tree for Presenting Expression Evaluations Visually

Essi Lahtinen[1,2]

Department of Software Systems
Tampere University of Technology
Tampere, Finland

Tuukka Ahoniemi[3]

Department of Software Systems
Tampere University of Technology
Tampere, Finland

Abstract

Novice programmers have difficulties with their visual attention strategies when following program visualizations. This article presents work in progress on improving the user interfaces of visualization tools to support students in the visual attention problems. We introduce a user interface solution called the dynamic evaluation tree. The basic idea is to reduce the amount of separate windows of the user interface and thus make it possible to concentrate the visual attention more in one part of the screen.

Keywords: Novice programmers, Program visualization tools, Visual attention strategy, Dynamic evaluation tree.

1 Introduction

The user interfaces (UIs) of visualization tools are often build with a similar structure. Many tools seem to have the same components in their UI and similar locations for them. We feel that this is partly because the tools are offering multiple different perspectives for the example and no specific design principles are applied for the UI design. Components are in their places just because they always used to be.

However, the effectiveness of a visualization tool in its pedagogical point of view may suffer from the use of multiple components and their placement in the screen. This article references the results of an eye-tracking study by Bednarik [2]. Based on

[1] Thank you for Nokia Foundation for partly funding this work.

[2] Email: `essi.lahtinen@tut.fi`

[3] Email: `tuukka.ahoniemi@tut.fi`

doi:10.1016/j.entcs.2008.12.047

this, we suggest a new way of integrating some of the UI components and improving the user's target of visual attention.

2 A Typical Layout of a Visualization Tool User Interface

A typical visualization tool presents multiple different kinds of actions in turns and parallel during the execution of a program or algorithm. The different kinds of actions can be for instance:

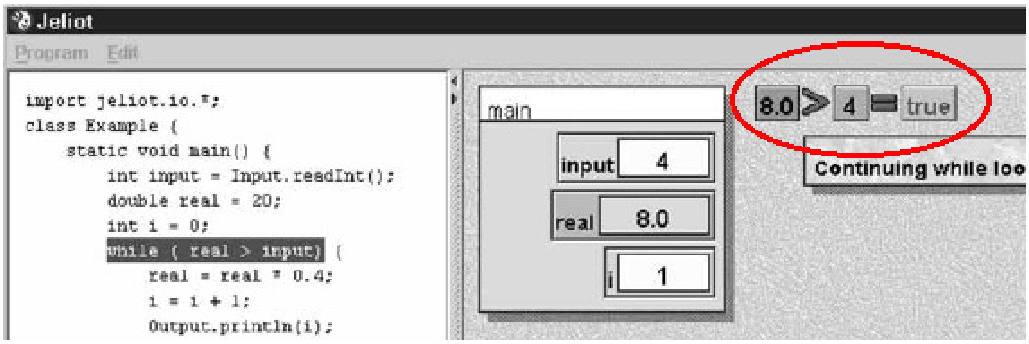
- Control reaches a new statement in the program code or algorithm.
- The values stored in the memory of the computer are referenced or changed.
- The values of expressions are evaluated.
- The program prints output and reads input.

The layout of visualization tools' UI usually presents different kinds of actions in different windows. Different tools have different names for the windows. We list some possibilities:

- *Code window:* Shows the program code or the algorithm that is executed. It typically illustrates the execution by highlighting the line of code or algorithm. It can also be named the algorithm window.
- *Memory window*: Performs most of the visual effects by drawing pictures of the variables and data structures and highlighting parts of the pictures. In the UI of Jeliot [6], this window is named *the theater*.
- *Evaluation window:* This window is activated whenever the code window executes an expression. The values of the operands, the operator, and the value of the whole expression are shown here. An example is marked with a red circle in Figure 1.
- *Console window:* Prints the input and reads the output of the program.

In addition to the most usual windows mentioned above, there can be other possibilities like *the annotation window* that explains the run of the program in writing [9]. Also the visualization tools that allow user interaction, often have a window for the control buttons.

Depending on the focus of the visualization, it is possible that some of the windows are not necessary and are thus absent. For example, algorithm visualization tools might not need the evaluation window at all since they present the algorithm on a higher abstraction level than individual expressions. Examples of tools that do not need an evaluation window are presented by Malmi et al. [5] and Naps et al. [7]. Sometimes one window of the visualization tool contains more than one kind of actions. For instance, the theater in Jeliot 3 [6] actually includes both the memory window and the execution window.



```
import jeliot.io.*;
class Example {
    static void main() {
        int input = Input.readInt();
        double real = 20;
        int i = 0;
        while ( real > input) {
            real = real * 0.4;
            i = i + 1;
            Output.println(i);
```

Fig. 1. The values of the variable in the expression executed in the code window is copied to the evaluation window for evaluation.

3 Results of an Eye-Tracking Study

A study on the methods of analyzing visual attention strategies in programming by Bednarik [2] is partly conducted by tracking the eye movements of programmers using a visualization tool Jeliot 3 [6]. The study describes the visual strategies of both expert and novice programmers.

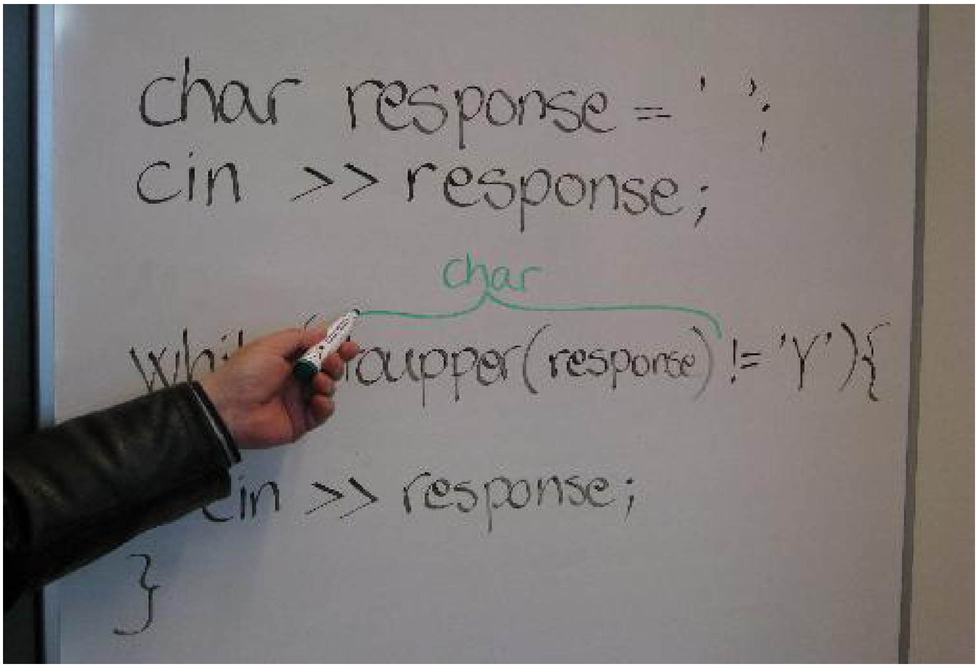
According to the study, expert programmers can better follow the information shown parallel in different windows of the visualization tool. They are able to change their visual attention strategy during a session of using the visualization tool. At the beginning of a session, they often concentrate on the code window and later on in the session on relating the code with the presentations in the other windows. Specifically, the experts follow the code window of the visualization tool more comprehensively than novices.

In contrast, novice programmers use only a couple of visual attention strategies. They either switch their visual attention repeatedly between different windows or concentrate all the time on one of the windows. Since the target audience of visualization tools are mainly novice programmers, this kind of a visual attention strategy should be taken into account when designing the UI of the tools.

4 Dynamic Evaluation Tree

When teachers explain the execution of program code to students, they tend to annotate the program code using curly brackets above or below the code line as seen in Figure 2. It is an easy way to mark the value or the type of an expression. This kind of annotations can be used in many different ways. Table 1 gives some examples. The same notation has also been used by Kumar [4] in an applet that generates problems related to expression evaluation. Since this has proven to be a good way to illustrate the execution to students, we suggest it should be tried in a visualization tool too.

The expressions that the evaluation window presents are also shown in the code window since they are, of course, part of the statement in execution. Instead of separating the expression in the evaluation window, we suggest that the evaluation tree could be integrated into the code window. This would reduce the need to switch



```
char response = ' ';
cin >> response;
while(toupper(response) != 'Y'){
cin >> response;
}
```

Fig. 2. Using curly brackets to explain on a white board.

Purpose	Example
1. To show the value of an expression	$\texttt{double width = }\underbrace{\underbrace{\texttt{max(}\,\underbrace{\texttt{x}}_{1.5}\texttt{, 0)}}_{1.5}\texttt{ + }\underbrace{\texttt{marginal}}_{2.5}}_{4.0}\texttt{;}$
2. To show the type of an expression, especially useful in case of hierarchical data structures like a vector containing structs	$\underbrace{\underbrace{\underbrace{\underbrace{\texttt{studentlist}}_{\text{vector<Student>}}\texttt{.at(i)}}_{\text{Student}}\texttt{.name}}_{\text{string}}\texttt{.at(j)}}_{\text{char}}$
3. To explain some error situations	$\texttt{if(}\underbrace{\underbrace{\texttt{response}}_{\text{char}}\texttt{ != }\underbrace{\texttt{'Y'}}_{\text{char}}}_{\text{bool}}\texttt{ }\vert\vert\texttt{ }\underbrace{\texttt{'N'}}_{\text{char}}\texttt{)}$

Table 1
Some examples on the use of curly brackets.

the focus of visual attention to the other side of the screen and thus should be easier to use for novice programmers.

One possibility for integrating the evaluation of expressions in the code window is to use a presentation similar to the curly brackets in Table 1. In this kind of illustration, the evaluation of an expression would of course not stay in the code window all the time, but appear step by step when the expression is executed and disappear when the execution proceeds to the next line. Thus, we call it *the dynamic evaluation tree.*

An alternative way of presenting the data dynamically inside the code window would be to show it in a tooltip window when the cursor of the mouse is placed on an expression in the code window. This could be used in any visualization tool regardless of the use of the dynamic evaluation tree. However, this solution does not help if you want the evaluations to be shown as the user clicks the step button.

The idea of using curly brackets is just to present the values of expressions in program code. In addition to this, also the other functionalities of the visualization tool are needed to present the execution of the program as a whole. For example on a line that contains a function call the tool can first present that the control changes to the definition of the function and use the highlighting of the executed line for this. After the execution of the function the tool can use a curly bracket to express that the return value of the function evaluates to the function call.

5 Discussion and Conclusions

Some tools for functional programming, e.g., WinHipe by Pareja-Flores et al. [8] and DrScheme by Felleisen et al. [3], use a similar idea of presenting evaluation of expressions dynamically. It is called the rewriting model of evaluation. In these tools the evaluation is presented as a sequence of rewritten expressions that shows the same information than the curly brackets in the dynamic evaluation tree. Actually, the idea of using curly brackets presents exactly the same information as rewriting. Both rewriting and use of curly brackets are ways of presenting deduction.

The rewriting model works in a very natural way in functional programming languages. It also provides a handy solution for non-trivial evaluations like function calls within imperative program languages. When using curly brackets instead of rewriting, function calls can be handled with other features than embracing as explained in previous chapter.

WinHipe has also been evaluated and the students have experienced that the tool is easy to use [1]. This is encouraging and we think that also the use of curly brackets is worth trying with students.

The reason why we prefer the use of curly brackets from rewriting is that we want the original program code to stay in the code window exactly as it is and only add annotations inbetween the lines. Thus, the curly brackets used in a similar way than the rewriting model, suits our needs better than rewriting the expressions.

If the code window includes the dynamic evaluation tree, it is actually no longer merely a code window but more like a multipurpose window. This should not only reduce the constant switching of the focus of visual attention but also relate the evaluation directly to the code. When the evaluation is presented directly inside the actual program code, the user may be able to form a stronger mental association between the code and what it actually does. This way the user could hopefully learn how to read the code better than when using a separate evaluation window.

Since novice programmers have most problems with their visual attention strategies, the dynamic evaluation tree should be most helpful for them. After all, the biggest target audience of visualization tools is novice programmers. Thus, we feel that the idea is worth trying.

The dynamic evaluation tree has not been implemented yet but we are charting the possibilities to add it to the next version of an existing visualization tool, VIP [9]. There will be some technical challenges in the implementation: the code window needs to be "stretched" vertically to make space for the curly brackets and the text above or below them. An other possibility could be to show the curly brackets in tooltip windows on top of the code window. However, this would obscure some code and could thus make the use of the tool difficult.

When the dynamic evaluation tree is implemented, it should be evaluated with an eye-tracking study to determine the possible aid with the visual attention strategies. The student still has at least the code window and the memory window to follow. An interesting possibility for further research could be to study whether it is possible to guide the student to develop better visual attention strategies by using the dynamic evaluation tree and other similar solutions in the UI.

One window fits all!

References

[1] Ángel Velázquez-Iturbide, C. Pareja-Flores and J. Urquiza-Fuentes, *An approach to effortless construction of program animations*, Computers & Education **50** (2008), pp. 179–192.

[2] Bednarik, R., "Methods to Analyze Visual Attention Strategies: Applications in the Studies of Programming," Ph.D. thesis, University of Joensuu, Joensuu, Finland (2007).

[3] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi, *The DrScheme project: an overview*, SIGPLAN Not. **33** (1998), pp. 17–23.

[4] Kumar, A. N., *Results from the evaluation of the effectiveness of an online tutor on expression evaluation*, in: *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education* (2005), pp. 216–220.

[5] Malmi, L., V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä and P. Silvasti, *Visual algorithm simulation exercise system with automatic assessment: TRAKLA2*, Informatics in Education **3** (2004), pp. 267–288.

[6] Moreno, A., N. Myller, E. Sutinen and M. Ben-Ari, *Visualizing programs with Jeliot 3*, Proceedings of the International Working Conference on Advanced Visual Interfaces AVI 2004 (2004).

[7] Naps, T. L., J. R. Eagan and L. L. Norton, *JHAVÉ - An environment to actively engage students in web-based algorithm visualizations*, ACM SIGCSE Bulletin , Proceedings of the thirty-first SIGCSE technical symposium on Computer science education SIGCSE '00 **32** (2000), pp. 109–113.

[8] Pareja-Flores, C., J. Urquiza-Fuentes and J. Ángel Velázquez-Iturbide, *Winhipe: an ide for functional programming based on rewriting and visualization*, SIGPLAN Not. **42** (2007), pp. 14–23.

[9] Virtanen, A. T., E. Lahtinen and H.-M. Järvinen, *VIP, a visual interpreter for learning introductory programming with C++*, Proceedings of the Fifth Finnish/Baltic Sea Conference on Computer Science Education (2005), pp. 129–134.