Perspectives on Program Animation with Jeliot*

Mordechai Ben-Ari[1], Niko Myller[2], Erkki Sutinen[2], and Jorma Tarhio[3]

[1] Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100, Israel
moti.ben-ari@weizmann.ac.il
[2] Department of Computer Science
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, Finland
{nmyller,sutinen}@cs.joensuu.fi
[3] Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland
jorma.tarhio@hut.fi

Abstract. The Jeliot family consists of three program animation environments which are based on a self-animation paradigm. A student can visualize her Java code without inserting additional calls to animation primitives. The design of the animation environments has been guided by the analysis of feedback from high school and university students. Evaluation studies indicate the benefit of dedicated animation environments for different user groups like novice programmers. Based on the results of these studies, we present plans for a future work on Jeliot.

1 Introduction

The Jeliot family consists of three program animation environments: Eliot [1], Jeliot I [2], and Jeliot 2000 [3], which have been developed in order to improve the teaching and learning of computer science, in particular, programming. The key design principle has been learning-by-doing: a student should have an animation tool which helps him to easily construct a visual representation of a program. The two representations of a program, namely its code and its animation, should match the mental image of the student, so that he could concentrate on the comprehension process instead of being misled by disturbing visual clues.

To achieve the kind of transparency in an animation environment described above, you need a consistent technical design. The Jeliot framework is based upon *self-animation*: the syntactical structures of a programming language have built-in visual semantics. In this way, an interpreter or compiler can automatically generate the animation of a program. This means that a student can, in principle, write any program without worrying about how to visualize it. The

* The work was supported by the National Technology Agency, Finland.

learning process takes place at the level of coding a program and simultaneously studying the two representations, textual and visual, not by creating the textual representation, and then subsequently creating another—visual—one for existing code.

A danger of using an automated system in education is that the student may exhibit superficial learning, without having made the personal effort to truly understand to subject. Or worse: does a system make a student into a zombie (as a graduate student from Ekaterinburg commented), by forcing him to see the running-time behavior of a program in a predetermined way? These kinds of questions led us to create a semi-automatic implementation of the self-animation paradigm. A semi-automatic visualization environment should allow a student to define the visual semantics for each of the program structures, or at least to choose the most appropriate one for his needs.

Apparently, there is a trade-off between the speed at which a fully-automatic animation can be constructed, and the versatility of the semi-automatic paradigm. One member of the Jeliot family is fully-automatic, rather others retain the flexibility of the semi-automatic paradigm. Thus, the Jeliot framework offers an attractive platform for evaluating the effects of various animation strategies in different student populations.

2 The Development of the Jeliot Family

Jeliot is an animation tool for visualizing the execution of Java programs or algorithms. We review two versions of Jeliot: Jeliot I [2, 4] implemented in the University of Helsinki works on the Web, and Jeliot 2000 [3] implemented in the Weizmann Institute of Science is a single Java application. We also consider Eliot [1], the predecessor of Jeliot I, as a member of the Jeliot family, because Eliot is functionally similar to Jeliot I. The name Eliot was taken from the Finnish word *Eliöt*, which means living organisms. The name Jeliot stands for Java-Eliot.

2.1 Early Years

The Jeliot family is an outcome of a long process. In 1992 Erkki Sutinen and Jorma Tarhio were involved with a project [5] of implementing ready-made animations for string algorithms. They noticed that the actual process of creating animations was more useful for learning than just watching ready-made animations. Because it took up to 100 hours to create a simple animation using the tools that were available at that time, the development of new tools for creating animations was started.

The first step towards Eliot was the implementation of self-animating data types. A data type is *self-animating* if the animation system provides a selection of visual representations for the type, and predefined animations are associated with its operations. If a program uses animated data types, its animation is seen as a sequence of the visualized operations when the program is run. This paradigm is called *self-animation*.

Self-animation is similar to the interesting-events approach [6], where the events are connected with the operations of the data types. However, self-animation has several advantages over the interesting-events approach: The algorithm and animation codes are not separated and data is not duplicated, because you do not have to construct the animation by inserting calls to animation primitives within the code of the algorithm. With self-animation, the preparation of an animation for a new algorithm is fast, and code reuse is easier.

Related Systems. The animation of Jeliot is controlled by operations on data structures. This kind of animation is closely connected with the development of debuggers and has a long history. Incense [7] was probably the first system capable of showing data structures of several kinds. Provide [8] offers alternative visual representations for variables. PASTIS [9] is an example of associating animation with a debugger. UWPI [10] introduced sophisticated automatic animation. In UWPI, a small expert system selects the visualization for a variable based on naming conventions of data types. At least Lens [11], VCC [12], and AAPT [13] are worth mentioning among other animation systems related to Jeliot.

2.2 Eliot

The key features of Eliot are self-animating data types and a user interface. Eliot extracts and displays the names of the variables of the self-animating types, which are integer, real, character, array and tree. The user decides which variables should be animated and selects their visual appearance. The user may accept the default values or change them individually for each object. In this way, constructing an animation is semi-automatic: the basic operations are defined automatically by the code, while the user can fine-tune the animation in the second phase, according to his internal view of data structures.

It is the integrated user interface of Eliot which makes self-animation practical to use. With Eliot it takes only a few minutes to design and compile a simple animation for a C program, but the same process would take about an hour without the user interface, because the set of animated variables must be programmed and the values of all visual attributes set by hand.

Eliot supports multiple animation windows called *stages*, which can be displayed simultaneously. The selection of the animated variables and their characteristics on each stage is independent. Eliot provides a feature to store the selection of variables and their visual parameters for later use.

Presentation of animation in Eliot is based on a *theater metaphor* [1], which has guided the design as well as the implementation. One can see the entire animation as a theatrical performance. The script of a play involves a number of roles, where the roles correspond to the variables of the algorithm to be visualized. An actor plays a role: in an animation, an actor is a visual interpretation of a variable. A play may have many simultaneous directions on multiple stages; similarly, an algorithm might have different visualizations on multiple animation windows.

2.3 Jeliot I

Eliot was completed in 1996. It ran under X windows and used the Polka animation library [14]. Because porting Eliot would have have been difficult, we decided to create a similar system for the World-Wide Web that would be portable. The Jeliot I system for animating Java programs was finished in 1997. Both Eliot and Jeliot I were implemented by students of University of Helsinki under direction of Erkki Sutinen and Jorma Tarhio.

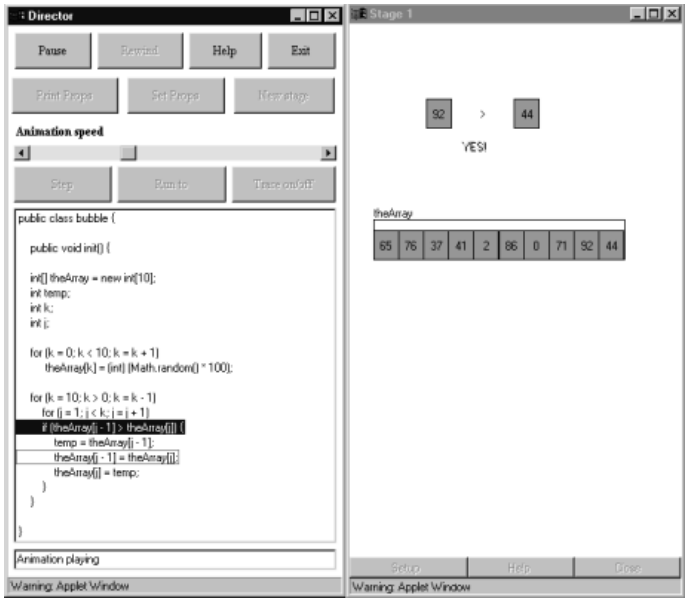

Fig. 1. A screenshot of Jeliot I

Although the functionality of Jeliot I is similar to that of Eliot, the technical design is completely different and is based on client-server architecture. Moreover, the Polka library is not any more used; instead, graphical primitives were implemented in using the standard Java libraries. The implementation of self-animation in Eliot relied on the ability to overload operators in the underlying implementation language, C++. In Jeliot I, calls of relevant animation primitives are inserted into the algorithm during preprocessing of the source code.

Jeliot I is capable of animating all the primitive types of Java (boolean, integral, and floating-point types), one- and two-dimensional arrays, stacks and queues. For all types, visual representations can be selected. However, the animated tree type of Eliot is not supported. Jeliot I highlights the active line of code in the program window during execution.

In the terms of the theater metaphor, Jeliot I has an additional feature that did not exist in Eliot. Jeliot I enables *improvisations*, where the user can modify

the visual appearance on the stage while a performance is running. The modified visualization parameters have an immediate impact on the actors on stage.

Figure 1 shows a screen capture from Jeliot I. The main control panel is on the left; on the right is one stage upon which an animation is taking place. There are many control windows used to configure the animation, too many in fact for novice users. The main control panel shows the source code of the program, highlighting the statement that is currently being animated. On the stage is the animation of a table being sorted with the bubblesort algorithm. Above the table is an animation of a comparison between two values of the table: "YES!" signifies that the comparison returns true.

2.4 Jeliot 2000

The user interface of Jeliot I proved to be difficult for novices. Therefore, a new version of Jeliot was designed and developed by Pekka Uronen during a visit to the Weizmann Institute of Science under the supervision of Mordechai Ben-Ari. Jeliot 2000 [3] was specifically designed to support teaching of novice learners.

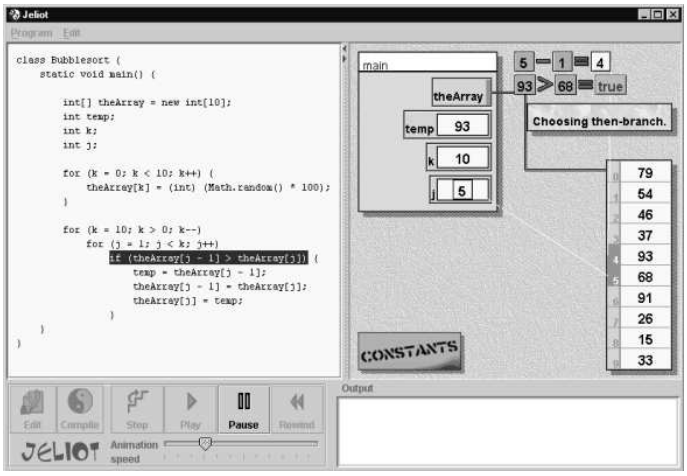

Fig. 2. A screenshot of Jeliot 2000

Two principles guided the design of Jeliot 2000: completeness and continuity. Every feature of the program must be visualized; for example, the use of a constant is animated by having the constant move from an icon for a constant store. Moreover, the animation must make the relations between actions in the program explicit; for example, the animation of the evaluation of an expression includes the animation of the evaluation of subexpressions.

As the intended users have little or no experience working with computers, the user interface of Jeliot 2000 is kept as simple as possible, without the customization abilities of Jeliot I. Jeliot 2000 displays two panels: one for the source

code, and another for the animation. The execution of the program is controlled through familiar VCR-like buttons (see Figure 2).

The implementation of Jeliot I is based on self-animating data types, but this makes it difficult to implement *visual relations* during evaluation of expressions, parameter passing and control structures. Jeliot 2000 embeds the animation system within an interpreter for Java source code, giving more power of expression at the cost of a more complicated implementation. It was a challenge to produce a smooth animation, because the visual objects that represent an expression must remain displayed for the user to examine in the context of the source code. Jeliot 2000 is written in Java like Jeliot I, but unlike the Web-based client-server architecture of Jeliot I, it is structured as a single application for simplicity and reliability in a school PC laboratory. The current implementation of Jeliot 2000 is limited in the language constructs that it supports.

Figure 2 shows a screen capture from Jeliot 2000. The left panel shows the program code that is animated. The lower-left corner contains the simple VCR-like control panel. The lower-right corner of the user interface contains a textbox that Jeliot 2000 uses to display the output. The animation is performed on the stage in the right panel. Here the animated algorithm is the same bubblesort algorithm as in Figure 1. The lower left corner of the stage displays a "source" of constants. The rest of the stage displays the animation: on the left, a box representing the main method including the variables that are declared within the method. At the moment, a comparison of two values of the table is being animated. One can see from the picture all subexpressions that are needed to evaluate the expression in the program code. In addition, an explanation of the evaluation is displayed.

2.5 Comparison and Discussion

Eliot and Jeliot I were aimed at teaching algorithms and data structures. They are more useful when the student already knows the elements of programming. Constructing of an animation is semi-automatic. Jeliot 2000 was made for novices to illustrate how a Java program works. Animation is fully automatic, and the user is not able to customize the animation. Table 1 lists the main features of the systems.

Comparing the animations of Jeliot I and Jeliot 2000 is difficult because in Jeliot I one can have several adjustable views, while Jeliot 2000 has only one fixed view. Another difference between the systems is in the level of explanation. Jeliot I does not present as many explanatory features as Jeliot 2000; these were added in the development of Jeliot 2000 as essential for novices.

Automation is one of the key features of Jeliot. Marc Brown [6] has discussed the problems of automatic animation. He argues that in general there is no one-to-one correspondence between the statements of a program and the images of the animation. An ideal animation according to him also shows synthetic metastructures of the algorithm. Of course, a fully automatic animation system cannot produce any synthetic metastructures. But in some cases they can be achieved by customizing a view, which is possible in Eliot and Jeliot I.

Table 1. Characteristics of the Jeliot family.

	Eliot	*Jeliot I*	*Jeliot 2000*
Language	C	Java	Java
Animated objects selectable	+	+	−
Visual attributes adjustable	+	+	−
Animated data types			
Number	+	+	+
Boolean	−	+	−
Character	+	+	−
Array	+	+	+
Queue	−	+	−
Stack	−	+	−
Tree	+	−	−
Active code line highlighted	−	+	+
Number of stages	many	many	1

And it is always possible to use an automatic system in an incremental way by programming synthetic metastructures as additional data structures of the algorithm and letting the animation system visualize them.

On the level of abstraction at which the programs are executed in Jeliot, informative displays are easy to construct, though the length of the program code and number of the data structures can be a problem. In Eliot and Jeliot I, one can add new stages to accommodate all the data structures that are to be animated. During the development of Jeliot I, this feature was used to debug the system, proving that even a large amount of code can be accommodated. Moreover, one can add data types of one's own inside Jeliot I and so animate even complex data structures. In Jeliot 2000, problems may arise if there are too many data structures to be simultaneously animated because Jeliot 2000 has only one stage. However, Jeliot 2000 was designed for novice users, so this limitation is not important.

3 Empirical Evaluation

There is no question that visualizations and animation of algorithms and programs is appealing, and can increase the motivation of students studying computer science. Intuitively, it would seem that they would significantly improve learning of computer science concepts; unfortunately, empirical studies do not unequivocally support this claim. From its inception, the Jeliot family has been subjected to extensive empirical evaluation; the results clearly show when program visualization can help and when not. In this section, we discuss some theoretical background, briefly describe empirical work by John Stasko, and then present details of the empirical evaluation of Eliot, Jeliot I, and Jeliot 2000.

3.1 When Does Visualization Help?

Petre and Green [15, 16] examined the use of visual programming by novices and experts. They concluded that the main advantage of graphics is the information contained in *secondary notation*, which is the informal part of the graphics: placement, color, indentation, and so on. Experts use this information efficiently to understand a graphics display; even if two experts use different secondary notation, they are able to easily decipher each others conventions and to recognize them as the work of experts. Novices ignore or misinterpret secondary notation, so their use of graphics is highly inefficient. Petre and Green conclude that: (a) the notational needs of experts and novices are different, and (b) novices must be explicitly taught to *read* graphics.

In a broader context, Mayer [17] performed a sequence of experiments on multimedia learning. He found that visualizations must be accompanied by simultaneous textual or verbal explanations to be effective. Multimedia guides students' attention and helps them create connections between text and concepts.

These results directly influenced the development of Jeliot 2000, by pointing out the need to a different tool for novices, and the need to include explanatory text with the animation of control structures.

3.2 Stasko's Work

Stasko, Badre and Lewis [18] used algorithm animation to teach a complicated algorithm to graduate students in computer science, but the results were disappointing: the group that used animation did not perform better than the control group. They conjecture that the students had not used animation before and found it difficult to map the graphics elements of the animation to the algorithm. In another experiment, Byrne, Carambone, and Stasko [19] showed that students in the animation groups got better grades on challenging questions for simple algorithms, but on difficult algorithms the differences were not significant. Kehoe, Stasko, and Taylor [20] found that algorithm animation is more effective in open homework sessions than in closed examinations. As one would expect from Mayer's work, they found that animation is not useful in isolation: students need human explanations to accompany the animations.

3.3 Evaluating Eliot and Jeliot I

An empirical evaluation of Eliot was carried out in a course on data structures [21, 22], using questionnaires, video tapes, learning diaries and interviews. The studies showed that using Eliot improved the motivation and activation level of the participating students, and that students produced higher quality code and documentation. Jeliot I has also been used for cross-cultural co-operation in teaching programming [23].

Matti Lattu [24] carried out an empirical evaluation of Jeliot I primarily on two groups of high-school students. (A group of university students was also

studied, but they did not use Jeliot I in depth.) Semi-structured interviews and observations of the lectures were used. Here is a summary of the results:

- Both students and teachers tend to use continuous execution, rather than step-by-step mode. Some educators might find this result to be counter-intuitive, because step-by-step execution is more interactive and constructivist [25] than continuous execution.
- Teachers frequently used visualization during lectures *before* presenting the program source.
- Jeliot I assisted in concept-forming, especially at the novice level.
- The user interface was too complex for novices. Confirming Petre's claims, the novices had difficulty interpreting the visualization, and the grain of animation was too coarse.

In subsequent research [26], Jeliot I was evaluated for use as a demonstration aid when teaching introductory programming and Java. Several classes were observed during the year and the field notes analyzed. The observation also captured the use of traditional demonstration aids: a blackboard and an overhead projector.

Their first conclusion is that ease and flexibility of use are of paramount importance. Developers of visualization software must ensure that the software is easy to use and reliable; otherwise, low-tech materials will be preferred. Of more interest is the observation that all aspects of a program must be visualized: data, control flow, program code and objects. Jeliot I is primarily a tool for visualizing data, while Jeliot 2000 significantly improved the visualization of control flow. Perhaps the next step is to include visualization of program code and objects. The BlueJ system [27] is an excellent example of this type of visualization tool.

3.4 Evaluating Jeliot 2000

Jeliot 2000 was evaluated by Ronit Ben-Bassat Levy in an experiment [3] that is as close to a controlled experiment as one could hope for: two classes, one using Jeliot 2000 and one as a control group. The experiment was carried out on tenth-grade high school students studying an introductory course on algorithms and programming, and the results were evaluated both quantitatively and qualitatively. The classes were composed randomly, but unfortunately, the control group was better, which made interpretation of the quantitative results somewhat difficult. The experiment was run for a full year, so that the students could overcome the difficulties inherent in using a new system.

The experiment was carried out by testing learning of each new concept as it was studied during the year. In addition, an assignment at the end of the year and a follow-up assignment during the next school year were used to investigate long-term effects. The quantitative test results were supplemented with individual problem-solving sessions which were taped and analyzed. For details of the experimental setup and results, see [3]. We can summarize the results and conclusions as follows:

- The scores of the animation groups showed a proportionally greater improvement, and their average matriculation exam score was the same as that of the control group, even though the latter contained stronger students.
- Mediocre students profit more from animation than either strong or weak students, though the grades of the latter two groups do not suffer.
- Students benefit most if the animation session includes individual instruction.
- The animation group used a different and better *vocabulary of terms* than did the non-animation group. Verbalization is an important step to understanding a concept, so for this reason alone, the use of animation can be justified.
- There was significant improvement in the animation group only after several assignments; one can conclude that it takes time to learn to use an animation tool and to benefit from its use.
- Students from the animation group used a *step-by-step method of explanation*, and some even used *symbols from Jeliot 2000* in order to show the flow of values. Students from the control group expressed themselves in a generalized and verbose manner. This difference in style continued into the next year.

Here is a summary of a problem-solving session on nested if-statements that demonstrates how the above conclusions were arrived at:

- In the control group, only the stronger students could answer the questions, and only after many attempts. They were not sure of the correctness of their answers and had difficulty explaining them.
- The stronger students of the animation group also had difficulties answering this question! They did not use Jeliot 2000 because they believed that they could understand the material without it.
- The weaker students of the animation group refused to work on the problem, claiming that nested if-statements are not legal, or that they did not understand the question.
- The mediocre students of the animation group gave correct answers! They drew a Jeliot 2000 display and used it to hand simulate the execution of the program.

4 Future Plans

The knowledge that has been collected through empirical evaluation of Jeliot has already changed the development of the Jeliot family. Here we present suggestions for the further development of Jeliot.

4.1 Visualization Techniques and User Interface Issues

In automatic program visualization, the animation is performed at constant speed, even though some parts of the program are more difficult to understand

than others. The ability to specify varying animation speeds for different parts of the program would make it easier to concentrate on difficult parts of the program. For example, initialization could be run at a higher speed than the statements in the inner loops of a sorting algorithm. The question arises: How does the user specify such difficult parts? The user would have to specify such parts through special comments in the program or using the user interface. The next paragraph suggests that semi-automated visualization could help with this specification.

For a novice user who has never programmed, automatic animation is essential. However, as the user becomes more experienced, he or she will want to control the configuration of the animations, for example, to select the speed of animation of different parts of the program, or even to skip the animation of parts like initialization. The user will also want to configure the visual elements for color, form and placement as was done in Jeliot I. Jeliot I also showed that storing configurations is important, because it fosters reuse animations, making them easier to share between a teacher and a student, or among the students themselves.

While forcing users to shift their gaze from one point to another on the screen is not recommended [28], it is important to guide the user in focusing on significant elements of the animation. One possibility would be to use sound [17]: the user would come to recognize specific sounds as guiding focus to specific locations.

4.2 Visualization with Jeliot

Structures of the programming language. The animation of method calls and array access is not entirely transparent in any of the systems of the Jeliot family, even though precisely these elements can be difficult for novices. It is important to find better ways to illustrate how a method gets its parameters and how multi-dimensional arrays are accessed. New self-animating data types such as lists and graphs would extend the applicability of Jeliot. Furthermore, Jeliot should make it easy for the user to create a new self-animating data type.

Object-oriented programming. Jeliot uses Java, a popular object-oriented language, but it can not handle objects or user defined classes. The next version of the Jeliot should provide better support for animating aspects of objects, such as object creation and method calling. Jeliot is quite good at animating the dynamic aspects of program execution, but to support object-oriented programming, it should also visualize the class structure in order to show the *uses* and *inherits from* relationships among the classes (cf. [27]).

Other programming languages. Currently Jeliot supports only programs written in the Java language. A visual debugger for Scheme [29] was implemented by slightly modifying Eliot, showing that the Jeliot could be modified to support other programming languages.

Visualizations of other subjects. Many dynamic phenomena can be described as algorithms, and therefore visualized by Jeliot [30]. For example, it would be possible to visualize the Mendelian rules of inheritance for a biology class.

4.3 Integration with Other Environments

We would like to integrate Jeliot into the learning environment so that metadata [31] could be collected about the students. For example, if Jeliot could collect metadata about the difficulties that an individual student has, this could be used both by the teacher and by Jeliot itself to adapt the animation speed as described above.

Jeliot could be integrated with other program visualization tools such as BlueJ [27] to provide a richer variety of views of the program and its execution.

A natural application of automatic animation is debugging [1]. The debugging abilities of Jeliot could be improved by implementing all of the Java language, and also by more efficient highlighting of the code. Errors found during compilation should be highlighted and partial animation performed if possible. Perhaps even common syntax errors could be animated. Thus, Jeliot could form part of a semi-automated assessment system for programming exercises [32].

4.4 Jeliot in a Learning and Development Community

Users should be able to interact with each other. Jeliot could be integrated with Internet communication tools to facilitate students working together on the same project, by enabling all participants to view the same animation.

Many of the proposed extensions to Jeliot could be implemented independently. The Jeliot source code could be licensed as free software, perhaps under the GNU general public license (GPL), with coordination coming from the Department of Computer Science at the University of Joensuu.

5 Conclusion

The phases of the history of Jeliot reflect different trends or approaches in computer science education, especially in teaching how to program. The predecessor Salsa was a package of ready-made animations for teaching string algorithms: it emphasized the instructive perspective. The Eliot, Jeliot I, and Jeliot 2000 systems, with their fully or semi-automatic animation tools, are examples of constructive learning environments. The future platforms will be worked out by extended and networked teams: they represent the idea of a learning community. In these communities, one can no more make a distinction between a teacher, a learner, and a designer.

One of the main lessons learned during the development and evaluation cycle of Jeliot is that of different learners and learner groups. An animation system should always offer a solution to a certain learner group's needs. Therefore,

an evaluation is not just another stage in the design and implementation of an environment, but should be carried out simultaneously during the whole process. Moreover, there is seldom one single best application for all animation or program comprehension needs, but rather a bunch of components of which a learner can pick up the ones she needs.

To sum up, what we have learned during the close to ten years of working with program animation, is that animation as well as apparently other learning tools should help a learner at his individual learning difficulties adaptively, distance independently, and taking into account diverse learning and cognitive styles.

References

1. Lahtinen, S., Sutinen, E., Tarhio, J.: Automated animation of algorithms with Eliot. J. Visual Languages and Computing **9** (1998) 337–349.
2. Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., Vanninen, P.: Animation of user algorithms on the Web. In: Proceedings of VL' 97 IEEE Symposium on Visual Languages. (1997) 360–367.
3. Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P.A.: The Jeliot 2000 program animation system. Journal of Visual Languages and Computing (2001 (submitted)) Preliminary version in E. Sutinen (ed.) *First Program Visualization Workshop*, pages 131–140, University of Joensuu, 2001.
4. Sutinen, E., Tarhio, J., Teräsvirta, T.: Easy algorithm animation on the Web. Multimedia Tools and Applications (2001) (to appear).
5. Sutinen, E., Tarhio, J.: String matching animator Salsa. In Tombak, M., ed.: Proceedings of Third Symposium on Programming Languages and Software, University of Tartu (1993) 120–129.
6. Brown, M.: Perspectives on algorithm animation. In: Proceedings of CHI '88. (1988) 33–38.
7. Myers, B.: Incense: a system for displaying data structures. ACM Computer Graphics **17** (1983) 115–125.
8. Moher, T.: Provide: a process visualization and debugging environment. IEEE Transactions on Software Engineering **14** (1988) 849–857.
9. Müller, H., Winckler, J., Grzybek, S., Otte, M., Stoll, B., Equoy, F., Higelin, N.: The program animation system pastis. Journal of Visualization and Computer Animation **2** (1991) 26–33.
10. Henry, R., Whaley, K., Forstall, B.: The University of Washington illustrating compiler. In: Proceedings of ACM SIGPLAN '90 Symposium on Compiler Construction. Volume 25(6) of SIGPLAN Notices. (1990) 223–233.
11. Mukherjea, S., Stasko, J.: Toward visual debugging: integrating algorithm animation capabilities within a source level debugger. ACM Transactions on Computer-Human Interaction **1** (1994) 215–244.
12. Baeza-Yates, R., Fuentes, L.: A framework to animate string algorithms. Information Processing Letters **59** (1996) 241–244.
13. Sanders, I., Harshila, G.: AAPT: Algorithm animator and programming toolbox. SIGCSE Bulletin **23** (1991) 41–47.
14. Stasko, J.: Polka Animation Designer's Package. (1994) Animator's Manual, included in Polka software documentation.
15. Petre, M.: Why looking isn't always seeing: Readership skills and graphical programming. Communications of the ACM **38** (1995) 33–44.

16. Petre, M., Green, T.R.: Learning to read graphics: Some evidence that 'seeing' an information display is an acquired skill. Journal of Visual Languages and Computing **4** (1993) 55–70.
17. Mayer, R.E.: Multimedia learning: Are we asking the right questions? Educational Psychologist **32** (1997) 1–19.
18. Stasko, J., Badre, A., Lewis, C.: Do algorithm animations assist learning: An empirical study and analysis. In: Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems, Amsterdam, The Netherlands (1993) 61–66.
19. Byrne, M., Catrambone, R., Stasko, J.: Do algorithm animations aid learning? Technical Report GIT-GVU-96-19, Georgia Institute of Technology (1996).
20. Kehoe, C., Stasko, J., Taylor, A.: Rethinking the evaluation of algorithm animations as learning aids: An observational study. Technical Report GIT-GVU-99-10, Georgia Institute of Technology (1999).
21. Sutinen, E., Tarhio, J., Lahtinen, S.P., Tuovinen, A.P., Rautama, E., Meisalo, V.: Eliot—an algorithm animation environment. Technical Report A-1997-4, University of Helsinki (1997). http://www.cs.helsinki.fi/tr/a-1997/4/a-1997-4.ps.gz.
22. Meisalo, V., Sutinen, E., Tarhio, J.: CLAP: teaching data structures in a creative way. In: Proceedings Integrating Technology into Computer Science Education (ITiCSE 97), Uppsala (1997) 117–119.
23. Järvinen, K., Pienimäki, T., Kyaruzi, J., Sutinen, E., Teräsvirta, T.: Between Tanzania and Finland: Learning Java over the Web. In: Proceedings Special Interest Group in Computer Science Education (SIGCSE 99), New Orleans, LA (1999) 217–221.
24. Lattu, M., Meisalo, V., Tarhio, J.: How a visualization tool can be used: Evaluating a tool in a research and development project. In: 12th Workshop of the Psychology of Programming Interest Group, Corenza, Italy (2000) 19–32. http://www.ppig.org/papers/12th-lattu.pdf.
25. Ben-Ari, M.: Constructivism in computer science education. Journal of Computers in Mathematics and Science Teaching **20** (2001) 45–73.
26. Lattu, M., Meisalo, V., Tarhio, J.: On using a visualization tool as a demonstration aid. In Sutinen, E., ed.: First Program Visualization Workshop, University of Joensuu (2001) 141–162.
27. Kölling, M., Rosenberg, J.: Guidelines for teaching object orientation with Java. In: Proceedings Integrating Technology into Computer Science Education (ITiCSE 01), Canterbury, UK (2001) 33–36. www.bluej.org.
28. Saariluoma, P.: Psychological problems in program visualization. In Sutinen, E., ed.: Proceedings of the First Program Visualization Workshop. Volume 1 of International Proceedings Series., Department of Computer Science, University of Joensuu (2001) 13–27.
29. Lahtinen, S.P.: Visual debugger for Scheme. Master's thesis, Deparment of Computer Science, University of Helsinki (1996) (in Finnish).
30. Meisalo, V., Sutinen, E., Tarhio, J., Teräsvirta, T.: Combining algorithmic and creative problem solving on the web. In Davies, G., ed.: Proceedings of Teleteaching '98/IFIP World Computer Congress 1998, Austrian Computer Society (1998) 715–724.
31. Markus, B.: Educational metadata. In: Proceedings of Qua Vadis-International. FIG Working Week, Prague (2000).
32. Higgins, C., Suhonen, J., Sutinen, E.: Model for a semi-automatic assessment tool in a web-based learning environment. In Lee, C.H., ed.: Proceedings of ICCE/SchoolNet 2001 Conference, Seoul, Korea (2001) 1213–1220.