Animation of User Algorithms on the Web*

J. Haajanen† M. Pesonius† E. Sutinen† J. Tarhio‡ T. Teräsvirta† P. Vanninen†

Abstract

An algorithm animation environment called Jeliot is presented. Jeliot allows a Web user to visualize his own algorithms, written in Java, over the Internet. Jeliot is based on self-animation of data types: the user selects the data objects of the source code which he wants to visualize, and Jeliot produces the animation automatically.

1 Introduction

Recently, educationalists have paid much attention to the World-Wide Web as a learning environment [13]. Although the idea of an almost unlimited learning space seems most attractive, the learning outcomes easily remain poor: the Web pages are often used as advertisements or attractions; at most, they include material through which the student browses, with no active intention.

The same applies also to Computer Science education. To be effective, a learning environment must be linked to the learner's current problem. This is seldom the case: even the interaction between the user and the Web pages is limited to the *given* hyperlinks. The reason for this is obvious: it seems to be hard to implement truly interactive learning environments on the Web, which allow the user to learn by *doing* instead of only by surfing.

The present study focuses on animation of algorithms [3]. With an animated representation of an algorithm, the learner can — hopefully — get a fresh picture of how it works, both at an abstract (idea) level and at a concrete (implementation) level.

The work on algorithm visualization has been active since the 50's [7]. In the last few years, there has been growing interest to provide algorithm animations over the Web [1, 5, 9]. Recently, one of the most interesting innovations in the area of Web-based learning environments for Computer Science education has been the algorithm animator by Brown and Najork [4]. It provides the user with an access to an algorithm laboratory, allowing him to study an algorithm of his choice. Although this environment is excellent, it still suffers from the noted drawback: the learner has only a selection of ready-made animations.

The Jeliot environment allows the user to animate algorithms of his own over the Internet. The user writes a Java program in a text field of a Web page, submits it, and gets back its animation. The animation is generated automatically from the source code, and it is displayed on the user's screen. Jeliot has been implemented using Java. Built on an extensible architecture, Jeliot can be modified to animate most common data structures.

The client/server model of Jeliot is an example of a general framework where a Web server works as a compiler: the user submits a source program or a specification to the server which returns a Java applet or other executable program to the user.

The contribution of Jeliot to Computer Science education is obvious. The student can combine the learning and research activities together: when studying a new algorithm with the aid of Jeliot he can observe strengths and weaknesses of the algorithm and develop and test new variations of it. Thus a learner becomes a researcher.

At its present stage, the Jeliot algorithm animation environment serves as a proof-of-concept system[1]. It shows that it is possible to implement a genuinely interactive learning environment also on the Web, even for hard-to-learn topics such as algorithms. Instead of browsing just another Web service, the user of a Jeliot-like system can learn by doing, solving real problems of his own interest. The inherently open world of the Internet must provide its users with tools not only to read other users' ideas but also to develop and link one's own thinking to the whole.

*Corresponding author: E. Sutinen.

†Department of Computer Science,
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland
email: `sutinen@cs.helsinki.fi`

‡Department of Computer Science,
University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, Finland
email: `tarhio@cs.joensuu.fi`

[1] The service is available at
`http://www.cs.helsinki.fi/research/aaps/Jeliot/`.

2 Design of Jeliot

Jeliot animates algorithms (programs) written in the Java programming language by visualizing data structures as smoothly moving graphical objects. The design and the operational framework of Jeliot are similar to those of Eliot [10, 11], our earlier animation generator, which works in the X windows environment and utilizes animation primitives of the Polka animation library [14]. Jeliot (short for Java-Eliot), written in Java, implements the model of Eliot in the World-Wide Web environment. Although the use of Jeliot is similar to that of Eliot, the implementation is different because the Web environment requires an architecture of another kind.

Traditionally an animation for an algorithm is constructed by inserting calls of animation primitives to several places in the code of the algorithm. This is called an "interesting event" approach [3]. In Jeliot, animation is controlled by the operations of data types, and the user does not need to write any additional procedure calls. If the program uses the animated data types provided by Jeliot, animation of the program will be automatic because the animation is embedded in the implementation of data type operations. This paradigm is called *self-animation*. For example, a `push` operation in the user program is animated according to the predefined visualization of the stack data type.

The design of Jeliot is based on the following view on algorithm animation: an *animation generator* consists of a *user interface* and a *visual interpreter*. The interpreter transforms the data objects of an algorithm to their visual counterparts in the animation. Hence, an animation generator presumes a *visual semantics* for the animated language. To facilitate diverse visualizations, the visual semantics must not be fixed but adjustable. The user interface allows the user to define the visual appearance of the animation, i.e. adjust the parameters of the semantics. This is important, because Ford [6] shows that programmers visualize even an integer variable very differently, not to mention more complex data types.

Presentation of animation in Jeliot is based on a *theater metaphor*, which has guided the design as well as the implementation. One can consider the entire animation to be a *theatrical performance*. The *script* of the play is the algorithm to be visualized, since it determines what happens and in what order. We need also a *stage* for the play to be performed; this refers to a window in which the animation is shown. In fact, we may even have multiple stages with the same show at the same time, but played slightly differently on each stage. The play is performed by *actors*, which are graphical entities having a set of visual attributes like size, shape, color, and location. Each actor has a *role* to play which corresponds to a data object of the algorithm. The appearance of each actor is determined by the *director*, i.e. the user, who designs the animation.

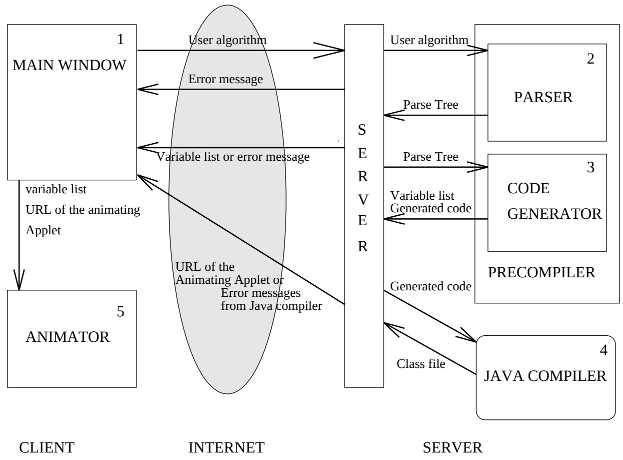

Figure 1: The overall structure of Jeliot.

3 Using Jeliot — an Example

Next, we will give an overview of producing of an executable animation of a user algorithm with Jeliot. We trace an example of the use of Jeliot by following the path of the algorithm through the system. The implementation of Jeliot will be explained in more detail in Section 4.

Overall structure. The main components of the system and their interaction are shown in Figure 1. The numbers tell the order in which the algorithm traverses through the system. The server process controls all the communication at the server side. The system uses the *main window* (1) for reading in the algorithm which is then sent to precompilation, consisting of *parsing* (2) and *code generation* (3). The latter two phases are required to add animation calls in the source code and to extract the potential variables to be animated. After precompilation, a list of these variables is sent to the main window.

The main window will open a director window from which the user can open stages for the animation. Each stage has its own setup window from which the user can choose the variables for animation and adjust their presentation. Primitive types, arrays, queue, and stack can be animated in the current version of Jeliot.

Meanwhile the server passes the generated code to

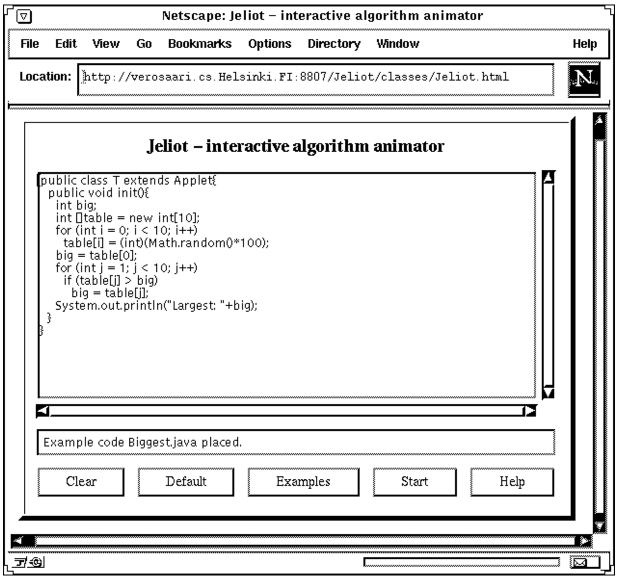

Figure 2: The main window.

the *Java compiler* (4). After compilation the server passes the name of the resulting applet class to the main window. This animating applet and the *animator* (5), the animation engine of Jeliot, together show the desired animation.

The main window, the animator, and the animating applet are run on the user's machine (the client side), other components on the provider's machine (the server side).

Main window. The main window (Figure 2) is an applet on a Web page containing an input area for an algorithm.

In the beginning an algorithm is typed or pasted to the input area. Then the algorithm is submitted to the Jeliot server for parsing and compilation phases (the *Start* button). A collection of sample algorithms is provided for the user (the *Examples* button).

Input algorithm. Algorithms written in Java must be placed inside a Java class or classes. Algorithms submitted to Jeliot are not different in this respect. As a default, Jeliot offers a skeletal class so that the user may concentrate on the code of his actual algorithm.

The algorithm should be added to such a method of a user class which is run for sure. Since the classes are made subclasses of the Applet class by the system, a good choice is the `init()` method — we use this approach in Figure 3, which shows one of the predefined example algorithms. This example algorithm searches sequentially for the largest value of an integer array, which has been initialized with random integers.

```
public class T{
  public void init(){
    int big;
    int []table = new int[10];
    for (int i = 0; i < 10; i++)
      table[i] = (int)(Math.random()*100);
    big = table[0];
    for (int j = 1; j < 10; j++)
      if (table[j] > big)
        big = table[j];
    System.out.println("Largest: "+big);
  }
}
```

Figure 3: An example algorithm.

Precompilation. When the Jeliot server receives the algorithm, it extracts the variables, which can be animated, from the source code and forms a list structure of them. This list structure is then passed back to the main window to be shown in the setup window, which can be opened from the stage window.

The code generator replaces operations on variables by method calls that besides doing the intended operation cause an animation call to be sent to the animator. After precompilation the animating applet is compiled with the Java compiler.

Director, stage, and stage manager windows. The animation control in the Jeliot is based on the concepts of director, stage and actor. In the following, the term "director" is used for the module, which helps the user to direct the play. The director is responsible for the management of the stages. A stage gives an unique view to the animation. The actors visualize the operations done on variables. Several actors can be present on a stage and a certain actor can be shown on many stages. An actor may have different presentation on each stage it is viewed on.

The animation and stage creation are controlled from the director window. When a stage is created, a stage manager window will also be opened for it. This is the window where the user can pick the actors — variables from any desired class and method of his algorithm — for the animation. The location, size, color and the representation of the actors on the screen can also be defined in this window. A variable can be visualized in various numeric formats or geometric shapes. The selections are stored into the proper actor class instance that will represent that particular

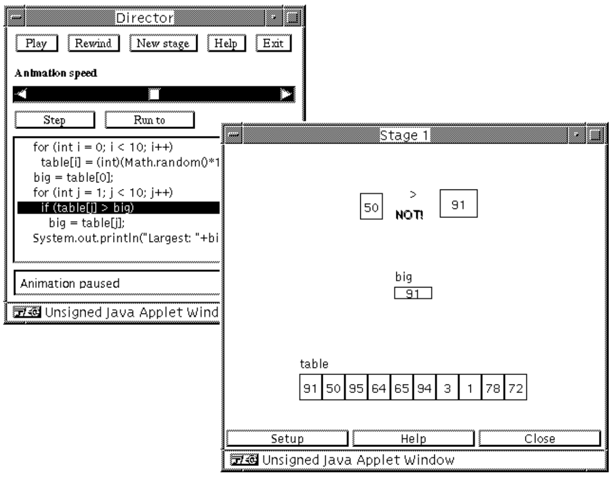

Figure 4: With the director window the user controls the animation of his code, running on Stage 1.

variable on that stage. All the operations done on a variable are acted by all the actors representing it on distinct stages.

Animation. In the animation a variable is visualized as an object, which can be for example a box or a circle. The contents of the variable is visualized inside the object, and the name is shown beside the object. Operations are shown by moving these objects and possibly showing the operator between them. In comparisons, the result is given as colored text. Optionally, the director window highlights the active line of the user program.

A screen shot from the animation of our example is given in Figure 4, where a comparison of variables `big` and `table` is shown. The director window is also shown in the same figure.

4 Implementation of Jeliot

The animations produced by Jeliot are visual interpretations of the data objects of a source code. The two main parts of Jeliot are the *server* and the *animator*. The server parses the source code and recognizes the data objects which can be animated. It also modifies the source code so that animation calls will be sent to the animator. The animator (or the animation engine) constructs and shows the animation of the resulting code.

4.1 Animation classes

A data object becomes a self-animating one when it is replaced by an instance of an animation class. These instances are the visual counterparts of the actual data objects. In Jeliot the animation classes are divided into two classes: *roles* and *actors*. The variables in the user's code are replaced by a proper role class in the precompilation phase. These role class instances then send requests to the actor class instances visualizing that particular role on the stages. Both the role classes and actor classes are organized in a way that inheritance and overloading can be used as effectively as possible. At the moment there are animation classes which correspond to primitive types, array (one and two-dimensional), queue, and stack.

The conversion to self-animating types is achieved by modifying the references of the data objects (either declarations or uses). Each instance of an role class has a unique ID. These IDs link the data objects the user has chosen to animate to the instances that are created during the execution of the code.

The role classes have (visual) methods corresponding to the operations performed on the animated data objects. These methods include calls to the actor class instances. The roles do not have to know if they are animated or not. All they have to do is to send animation requests to the actors. These requests are relayed through the director to all the stages. Each stage knows which roles are acted on it and the request is passed to the actor if necessary.

Animation classes are precompiled, and they are stored as byte code. Therefore, the code generator has to add import statements to include these classes to the user code.

4.2 Server

The server process receives all incoming service requests from the main window. All requests come to a single socket. The server process then allocates a new socket and a corresponding communication thread for each new client. The communication thread creates a directory for the new client, and it is able to write required files into this directory.

Communication thread. The communication thread takes care of a single client of Jeliot by communicating with it over the Internet. The communication thread is capable of receiving two types of messages from the client and sending back three types of messages. The start of a message determines the message type.

Received messages. The communication thread listens continuously to the socket which the server has allocated for it. It controls its own operation according to the received message. A received message of the first type contains the code to be animated; the messages of the second type are stop requests. Whenever the communication thread receives a stop request, it releases the no longer needed resources and stops.

Sent messages. The communication thread can send any of the following messages to the client:

1. a list of data objects that can be animated;
2. a reference to the class to be loaded for running the animation; or
3. a compilation error which can originate from the parser, the code generator, or the Java compiler (see Fig. 1).

Processing the source code. The communication thread transforms the source code, received in a message, into a Java class file. In order to do this, the communication thread utilizes three separate modules, namely the *parser*, the *code generator*, and the *compiler*. Before compilation, some segments of the original code have to be replaced with a new code.

Parsing. First, the communication thread starts the parser to check the syntax of the source code. The parser has been constructed with the CUP tool [8] from an LALR grammar, and it applies a lexical analyzer constructed with the JLex tool [2]. The parser builds a parse tree which is returned to the communication thread.

Code generation. The second step is to modify the parsed code into a form which can be animated. This phase also produces the list of data objects from which the user can select those to be animated.

The skeleton of the list structure is a tree consisting of nodes representing class or method declarations. Each of the nodes starts a list representing data objects declared in the corresponding class or method; we call these elements the data object nodes. If a node represents a class, the code generator stores the name of the class. Into a method node, the code generator stores also the types of the arguments, the return type of the method, and the method name. Data object nodes hold the name and the type of the data object and a unique ID to be used in the animation. The code generator maintains a counter to administer the ID numbering.

The code generator writes the modified code into a file and returns the control to the communication thread. The communication thread sends the animation selection list to the client.

Compilation. As the final task, the communication thread starts the Java compiler for the generated code file. After the compilation, the communication thread is able to send a message about the class to be loaded on the Web page. When this class is run, the animation calls are produced as a side effect and the Web user can see the animation.

4.3 Animator

The animator is the module of Jeliot which shows the actual visualization of an algorithm. The animator can be thought as a small server, which handles requests of animated operations sent by the animating applet. The animator basically includes the director, the stages, the communication between them, and the GUI. The term animator is used here to describe these parts as a whole. A typical animated operation is a comparison between two variables or an assignment. The animator animates the requested operation and returns the control to the animating applet, which is blocked while the animator is active. As described earlier, the animation requests are embedded in the code of the animating applet during code generation.

Start of animation. The director is started by the main window. At that point, the list of variables that can be animated has been received, but the animating applet is still being compiled at the server side. The stages and the properties of the variables on them can be set at this time. When the compilation has been successfully finished, the animating applet class is loaded from the server. This class has been stored in the directory of the original Web page containing the main applet. This makes it possible to use the default class loader of the Java virtual machine, which searches for the classes in the home directory of the Web page. After the class has been loaded, the animation is started in the director window and the director opens a new window and starts the animating applet in it. Because the animating applet runs in a separate window, the user may utilize any graphical capabilities of Java in his algorithm to be presented in that window. This feature makes Jeliot quite versatile. For example Jeliot can be used to debug even (other) animation algorithms.

Linking of actors with roles. To manage the animated variables (i.e. actors), the animator keeps track of them. Each stage keeps a list of all potential variables for animation (including those which have not been selected). Each variable in the list has a reference to its role counterpart in the play, and a record of the graphical properties: location, size, color, etc. The opposite does not hold, the roles do not know their graphical representations (actors).

User interface. The user interface of the animator consists of two parts: stages where the animation takes place, and the director window (Figure 4) where the user can control the animation and its speed on all the

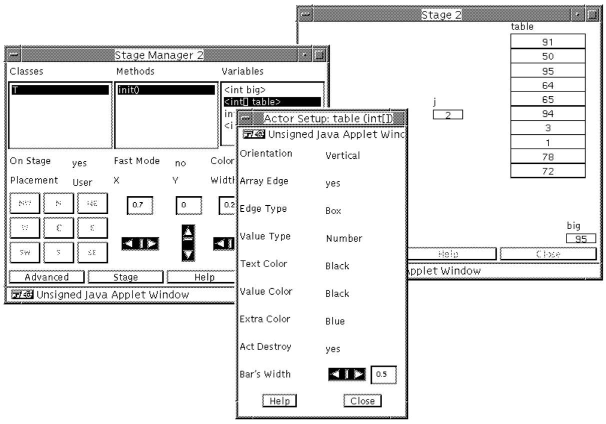

Figure 5: The stage setup window.

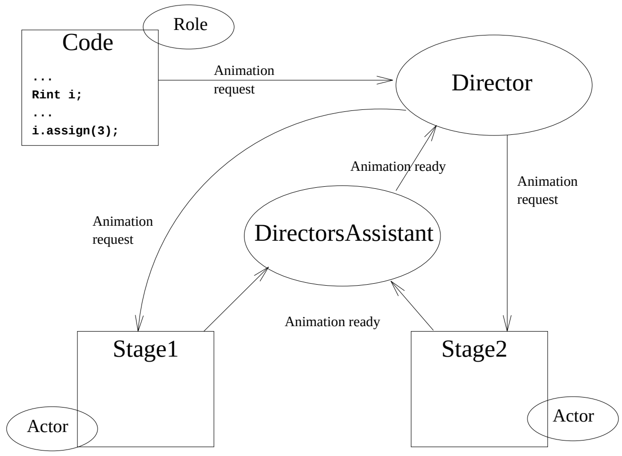

Figure 6: Messages during animation.

stages. Every stage has a setup window (Figure 5), called a *stage manager*, for adjusting the properties of the actors on that stage as well as the properties of the stage itself. The Figure 4 shows a screen shot of the director window.

Animation. The outlook of the animated variables can be selected from many predefined visualizations. The value of a variable can be seen for example as an alphanumerical value drawn inside a box or as a bar without a bounding box. The same variable can have different appearances on separate stages. When a variable takes part in some animated operation, a temporary copy is made of the corresponding graphical object. The copy is moved in the window while the original object stays at a fixed position. This simplifies the stage management because the positions of the actual objects in a window remain unchanged.

The animator uses double buffering scheme to get a flickerless and continuous animation.

The animation requests are made by the instances of the Role classes that call the director which then multicasts these requests to all the stages (Figure 6). Each stage checks if the role that sent the request is visualized on that stage. If a proper actor is found, the request is passed to the actor, which then calls the animation routines that perform the visualization. All the animation routines needed — for example drawing a box or a circle to a specific location on the stage — are implemented in a class called JAPI, short for the Jeliot Animation Primitive Interface. At present, the animator supports several types of requests, for example comparison and assignment. Furthermore, these are divided in subtypes by overloading the method names with different arguments. For example, the animating applet may request to show the comparison between an integer constant and an integer variable or two integer variables. The actual animation action depends on the arguments given in the request. The comparison between two integer variables is animated as follows:

1. The copies of the corresponding objects in the window are slid to the comparison area, by default in the upper part of the animation window.
2. If the result of the comparison is true, the green text *Yes* is drawn between them. Otherwise, the red text *Not* is drawn.
3. The copies are slid back on the original objects and deleted.

The comparison between an integer constant and integer variable is carried out in the same way, except that a temporary object is first made for the constant. Arrays are handled as a set of scalar objects. When an animation request is associated with an array item, a copy of the item is made and it is animated as a scalar variable.

When a stage has finished an animation request, it sends a message to the *director's assistant* which forwards this message to the director. The director's assistant works as an asynchronous mailbox for the director. The director blocks the the animating class until it has received the messages from all of the stages, and then the execution of the code can continued.

The animator is designed to be flexible. The needs of the future development are kept in mind. The object-oriented nature of Java gives us a solid development framework with method overloading, inheritance etc. The animation routines offered by the JAPI class supports the building of new animated data types. Furthermore, dividing the implementation of the animated variables in roles and actors makes it possible to develop the visual aspects just by concentrating on the actual animation without modifying the server, for example the code generation. Also the director, relaying messages between roles and actors, does not know what types of animating classes exist in the system, and therefore adding new types would mean no changes there.

4.4 EJava

The dialect of Java accepted by Jeliot is called EJava. There are minor differences between EJava and Java(tm). EJava includes two animated abstract data types: Rstack and Rqueue for presenting stacks and queues. Rstack resembles the Stack class in the standard java.util package.

There are some limitations in EJava concerning arrays: When declaring an array, it cannot be assigned to another array or null. Arrays should be declared using notation `Type [] name` instead of `Type name []`. Animated arrays must be one or two-dimensional. ArrayIndexOutOfBoundsException cannot be caught.

EJava does not properly support inheritance of user classes. New variables cannot be declared in while or do-while statements without making an explicit block for the statement. Class attributes are not initialized. Error checking is weaker in EJava than in Java.

EJava contains more reserved words than Java. Jeliot does not handle unicode characters properly. Compound assignment operators `|=`, `&=`, `<<=`, `>>=`, and `<<<=` cannot be used with animated variables.

We are working on reducing the differences between EJava and Java. Up to date list can be found with the on-line user's manual that can be found at our server.

5 Future development

We are gradually adding new features to Jeliot. The storing of the values of visual attributes is on the top of our wish list. We have also plans of improving the space allocating heuristic for actors on stages.

We will implement new animated data types, including more complex data types like binary trees and lists. Also the visualization of instances of classes will be studied.

Meisalo et al. [12] performed a case study of Eliot as a learning environment in a laboratory course offered at the University of Helsinki. It gave valuable feedback from intermediate level Computer Science students. The overall experience of using Eliot as a learning aid was positive. Because the use of Jeliot is similar to that of Eliot, this evaluation applies also to developing Jeliot further.

In its present form, Jeliot offers a platform for a Web-based learning environment. Its benefit is increasing with the general acceptance of Java. Therefore, one of its potential applications is a guided tour of Java. Along with getting acquainted with different aspects of the language, the learners could practice programming using Jeliot. This approach could serve an environment of several concurrent learners, each of which visualizing his own algorithm and comparing it with the results of the others. This is how the Web would not only offer a substitute for a traditional learning environment, following its methods, but add value by supporting a collaborative learning process.

The future Jeliot could, thus, provide a common ground for human communication and cooperation, helping learners to represent their ideas on the screen but also understanding in others' ways to do the same thing. In the same way, Jeliot opens new directions to Web-based collaborative research: the researchers can experiment with their (competitive) algorithms over the net, passing the code with for example email, and using Jeliot to visualize the results online.

One of the aims of Eliot was to bring a student's learning process closer to research activity. Jeliot adds a new dimension to this. In the future, the Web-based Jeliot hopefully brings learners, teachers, and researches together, working actively on a topic, independently on local shortcomings on teaching or research resources.

Acknowledgments. We thank Tanja Bergius, Arne Dybdahl, Pasi Huhtiniemi, Soile Huttunen, Jari Juslin, Marko Kivi-Koskinen, Pirkka Liukkonen, Juhani Peltola, Mikko Pettilä, Joonas Reynders, and Ari Vähä-Erkkilä for their assistance. The work was financially supported by the Ministry of Education.

References

[1] J. E. Baker, F. Cruz, G. Liotta, and R. Tamassia: Algorithm animation over the World-Wide Web. In: Proc. AVI '96, Int. Workshop on Advanced Visual Interfaces, 1996, 203–212.

[2] E. J. Berk: JLex: A lexical analyzer generator for Java. `http://www.cs.princeton.edu/...` `~appel/modern/java/JLex/`

[3] M. Brown: Perspectives on algorithm animation. In: Proc. CHI '88, Human Factors in Computing Systems, ACM, 1988, 33–88.

[4] M. H. Brown and M. A. Najork: Collaborative active textbooks: a Web-based algorithm animation system for an electronic classroom. In: Proc.

VL '96, Symposium on Visual Languages, IEEE, 1996, 266–275.

[5] J. Domingue and P. Mulholland: Staging software visualizations on the Web. To appear in: Proc. VL '97, Symposium on Visual Languages, IEEE, 1997.

[6] L. Ford: How programmers visualize programs. Research Report No. 271, Department of Computer Science, Old Library, The University of Exeter, Exeter, EX4 4PT, 1993.

[7] L. M. Haibt: A program to draw multi-level flow charts. In: Proc. of The Western Joint Computer Conference, volume 15, 1959, 131–137.

[8] S. Hudson: CUP User's Manual. `http://www.cc.gatech.edu/gvu/people/...` `Faculty/hudson/java_cup/manual.v0.9e.html`

[9] B. Ibrahim: World-wide algorithm animation. In: Proc. 1st World-Wide Web Conference, Geneve, Switzerland, May 25–27, 1994, 305–316.

[10] S.-P. Lahtinen, T. Lamminjoki, E. Sutinen, J. Tarhio, and A.-P. Tuovinen: Towards automated animation of algorithms. In: Proceedings of Fourth International Conference in Central Europe on Computer Graphics and Visualization 96 (ed. N. Thalmann and V. Skala). University of West Bohemia, Department of Computer Science, 1996, 150–161.

[11] S.-P. Lahtinen, E. Sutinen, and J. Tarhio: Automated animation of algorithms. Report C-1997-38, Department of Computer Science, University of Helsinki, 1997.

[12] V. Meisalo, E. Rautama, E. Sutinen, and J. Tarhio: Teaching algorithms with animation – a case study using Eliot. To appear in: Proc. of LeTTET '96, Learning Technology and Telematics in Education and Training, Joensuu, Finland, 1997.

[13] T. Reeves and P. Reeves: The effective dimensions of interactive learning on the WWW. In: B. Khan (ed), Web-based instruction, Englewood Cliffs, NJ: Educational Technology, 1997, 59–66.

[14] J. Stasko: Polka Animation Designer's Package. Animator's Manual, included in Polka software documentation, 1994.