Animation of User Algorithms in the Web*

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

Abstract

An algorithm animation environment Jeliot is introduced. Jeliot allows a web user to visualize his own algorithms, written in Java, over the Internet. Jeliot is based on self-animating data types: the user determines the visualized data objects of the source code, and Jeliot produces the animation automatically.

1 Introduction

Recently, educationalists have paid much attention to the World-Wide Web as a learning environment. 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 advertisiments 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* hyper links.

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

*The work has been supported by the Academy of Finland and the Ministry of Education.

†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`

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. Excellent as this environment is, 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 an algorithm in a text field of a web page, submits it, and gets back its animation. The operational framework of Jeliot is similar to that of Eliot [10], our earlier animation generator, which works in the X windows environment. The animation is generated automatically from the source code, and it is displayed on the user's screen. Jeliot has been implemented using the Java language. Built on an extensible architecture, Jeliot can be modified to animate most common data structures.

The contribution of Jeliot to the Computer Science education community is obvious: the student can combine the learning and research activities together, without worrying of the suitable place to do this. In addition, Jeliot also opens a new path in web-based collaborative research: the researchers can experiment with their (competitive) algorithms over the net, seeing the results online at the both ends.

At its present stage, the Jeliot algorithm animation environment serves as a proof-of-concept system. It shows that it is possible to implement a genuinely interactive learning environment also in 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.

2 Eliot and Jeliot

The Jeliot system has been implemented following the ideas of Eliot [10], our earlier animation generator. We will briefly present the key features of Eliot and explain how Jeliot differs from Eliot.

Eliot is an object-oriented tool for quick and easy production of algorithm animations. The name *Eliot* comes from the Finnish word *Eliöt*, which means living beings. The current implementation of Eliot runs on Linux workstations or on X terminals. Eliot utilizes animation primitives of the Polka animation library [12].

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 Eliot, animation is controlled by the operations of data types, and the user does not need to write any additional procedure calls.

Eliot presents a two-phase approach to 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. This paradigm is called *self-animating data types*. 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.

The design of Eliot has been strongly influenced by the theater metaphor [10] for algorithm animation. An algorithm is regarded as a script of a play. The play, or the visualized algorithm, involves a number of *roles*. An *actor* performs a role. The *performance* of each role varies according to the *direction* of the play. The roles correspond to the data objects of the algorithm. An actor is a graphical entity that has a set of visual attributes like size, shape, color, and location.

An algorithm written in C is given as input to Eliot, which extracts and displays all the data objects used in the source code. Each implemented data type has a set of ready-made visual representations from which the user picks one for each of those data objects he wants to be visualized. So far there are visualizations available for the following data types: integer, real, character, array, and tree. Based on these selected representations Eliot generates an animation program, which can be run within the user interface, producing the animation as an intentional side-effect.

Eliot supports multiple animation windows, *stages*, to be displayed simultaneously, each having its own characteristics for the same animated data objects. Eliot allows the user to define the animation characteristics independently for each data object on each stage.

Our new animation generator Jeliot (short for Java-Eliot) follows the model of Eliot and transfers it into the World-Wide Web environment. The user submits an algorithm written in Java to the system and gets back an animation of his algorithm as a Java applet.

The current version of Jeliot does not yet have all the functionality of the original Eliot, but a more sophisticated version is under construction. The current version is capable to animate integers and integer arrays in one animation window.

3 The Use of Jeliot — an Example

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

System structure. The main components of the system and their interaction are shown in Figure 1. The numbers show the order according to which the algorithm traverses through the system. The server process controls all the communication at the server side. The *main window* (1) is used for reading in the algorithm. The algorithm is then sent to precompilation, which consists of it lexical analysis (2) and *code generation* (3). These phases are required to extract the potential variables to be animated from source code and to add the animation calls for them in the code. After precompilation, a list of these variables is sent to the main window in order to let the user choose those which will actually be animated. Meanwhile the server passes the generated code to 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 a web page containing an input area for an algorithm and a selection area below it. The operation is controlled by buttons *Clear*, *Reset*, *Send*, *Help*, and *Animate*.

In the beginning an algorithm is typed or pasted to the input area. The *Send* button submits the algorithm to the Jeliot server for parsing and compilation phases.

Input algorithm. Algorithms written in Java must be placed inside a Java class, or classes. Algorithms given to Jeliot are no different. As a default, Jeliot offers a skeleton class so that the user may concentrate on the code of his actual algorithm.

Technically, this class must extend the Applet class which allows executable Java programs to be added on web pages. The Applet class is part of the standard Java class library.

The algorithm should be added to some method of an user class which is surely to be run. A good choice is the `init()` method — we use this approach in our example shown in (Figure 3).

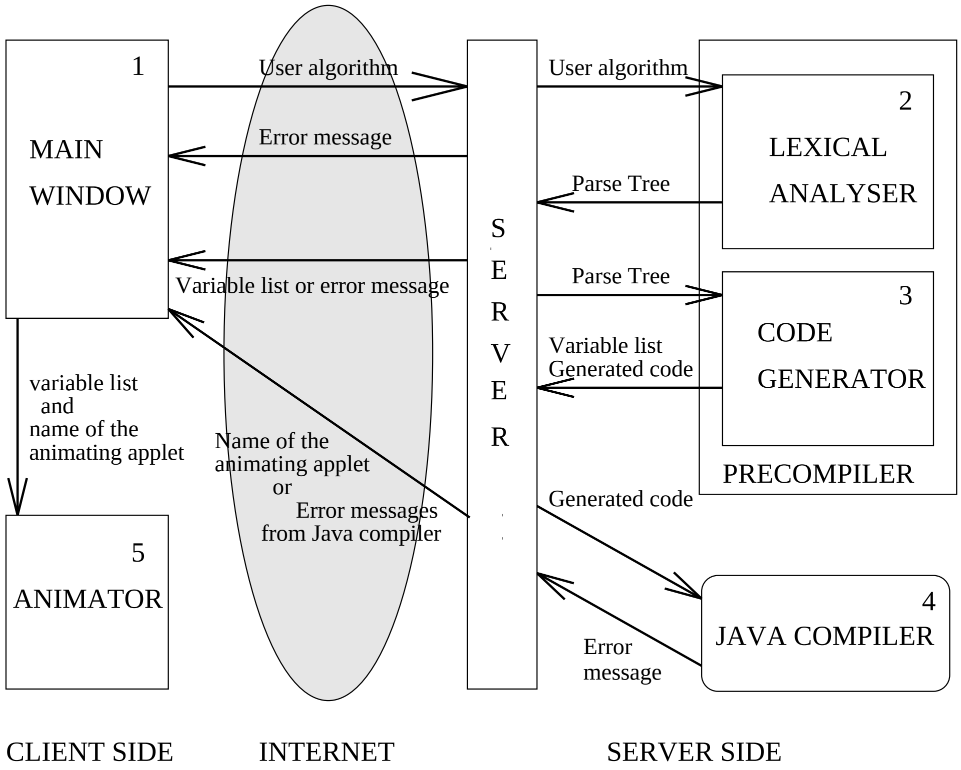

Figure 1: The system structure of Jeliot.

The given example algorithm searches sequentially for the largest value of an integer array, which is initialized with random integers.

Precompilation. When the Jeliot server receives the algorithm, it extracts the variables, which can be animated, from the source code and collects information on them. This information is then passed back to the main window to be shown in the selection area.

The Jeliot code generator replaces the type declarations of these variables by special animating types and creates instances of them. All the operations done on these variables are replaced 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.

Selection area. In the selection area the user gets a list of the variables divided into three columns *Class*, *Method* and *Variable*. The user can pick variables from any desired class and method of his program to be animated. Class variables can be chosen with class selection. The selection is passed to the Animator which adds

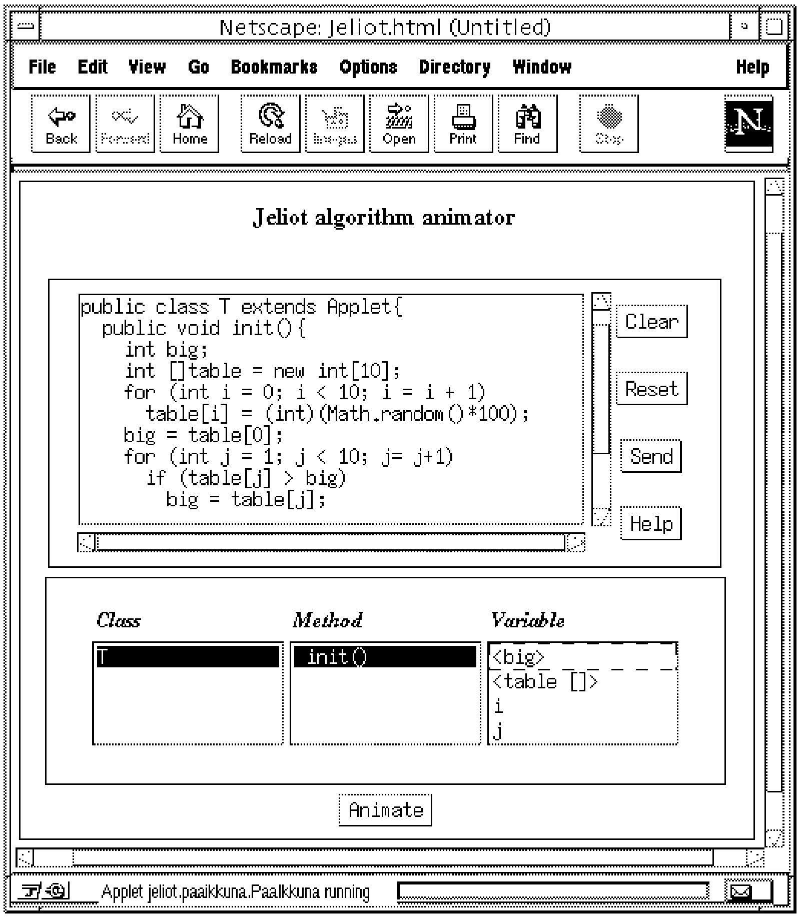

Figure 2: The main window.

the IDs of the chosen variables to its data structures. Now when the animating version of the algorithm does something with any selected variable, the operation will be animated.

The animation is launched with the button *Animate*. This will start the Animator, which initiates the animating applet, the animating version of the user algorithm. The Animator and the animating applet together create the animation.

Animation window. The *Animate* button opens an animation window. In the beginning the animation is in a halted state. The animation can be started and stopped with the *Play/Pause* button. The speed of the animation can be adjusted with a sliding control.

A variable is visualized as a box with its value in it and its name beside it.

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

Figure 3: An example algorithm.

Operations are shown by moving variable boxes and showing the corresponding operator between them. In comparisons, the result is shown by changing the color of the operator — green denotes true and red false.

The animation of our example is shown in Figure 4, the boxes of the selected variables `big` and `table` can be seen. Scalar variables are usually shown on the left and arrays in the low part of the animation window. Variables are slid to the upper part of the window for comparisons. An active comparison is shown in Figure 4.

With the Jeliot system anyone has a possibility to have an animation of his Java algorithm generated and shown in the web. The special property of Jeliot system is that there is no need to insert any special commands to the source code — animation is created automatically from a code written in Java. Although there are some minor syntactical limitations, all of the reserved words of Java are accepted, and what is more important, nothing has been added to the language.

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 recognises 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

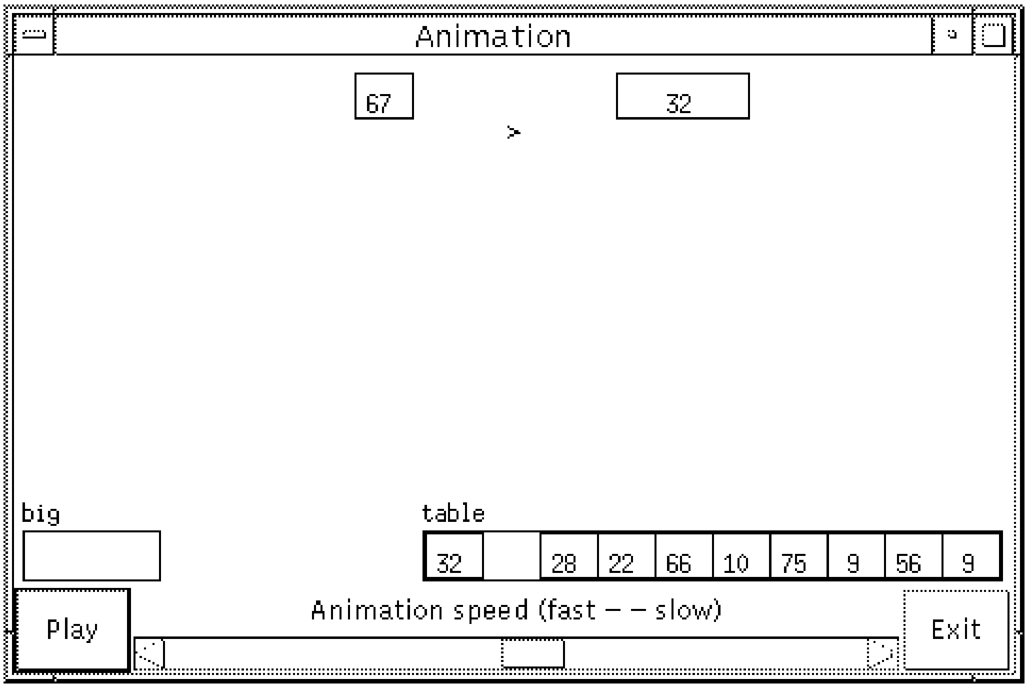

Figure 4: An animation.

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. Animation classes are subclasses of one special animation class (Jobject). At the moment there are animation classes which correspond to Java's int (Jint) and array (Jarray) types. The conversion to self-animating types is achieved by modifying the references of the data objects (either declarations or uses). Each instance of an animation 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 animation classes have (visual) methods corresponding to the operations performed on the animated data objects. These methods include calls to the animation engine. When created, each instance of an animation class checks with the Animator if it has to send animation calls. This forms a connection between the Animator and the animating applet.

Animation classes are precompiled, and they are stored as byte code. Therefore, the server has to generate the import statements to include these classes to the user code. This is carried out in the code generation phase.

4.2 Server

The server process receives all incoming service requests from web pages. 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 more 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 be animated;

2. a reference to the class to be loaded for running the animation; or

3. a parser or compiler error.

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 different modules, namely the *parser*, the *code generator*, and the *compiler*. Before compilation, some of the original code segments have to be replaced with a new code.

Parser. First, the communication thread starts the parser to check the syntax of the source code. The parser also creates a parse tree from the code. The parser is constructed with the CUP tool [8], and it applies a lexical analyzer constructed with the Java-Lex tool [2]. The parser has been constructed from an LALR grammar. The constructed parse tree is returned to the communication thread.

Code generator. The second step is to modify the parsed code into a form which can be animated. This phase produces information to both the client and the Animator. The client needs information to present a list of data objects from which the user can select the ones to be animated. On the other hand, the Animator has to be able to link the user choices to the visual objects on the screen.

The code generator modifies the parsed code. It also collects all data objects that can be animated into a hierarhical structure, called the *animation selection list*, which is sent to the client. The skeleton of this structure is a tree consisting of nodes representing class or method declarations.

Each of the nodes starts a list of elements 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 store information about 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.

Compiler. The final task is to compile the file. The communication thread starts the Java compiler which compiles the generated code file. After the compilation is finished, the communication thread is able to send the message about the class to be loaded in the web page. When this class is run in the page, the animation calls are produced as a side effect and the web user can see the animation.

4.3 Animator

Next we will give an overview of the Animator, the animation engine of Jeliot. The Animator is the module of Jeliot which constructs and 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. Typical animated operations are comparison between two variables and initialization of an array. 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 Animator is started by the main window. At that point, the variables to be animated have been selected and the animating applet has been compiled at the server side. Next 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 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 Animator 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. This feature makes Jeliot quite versatile. For example Jeliot can be used to debug even (other) animation algorithms.

Data structures. To manage the animated variables, the Animator keeps track of them. All potential variables for animation are stored in a list (including those which have not been selected). This approach make it easier to add new features in the future. The record of a variable in the list contains its current value, its ID, and the graphical properties: location, size, and color. The IDs of the variables are used as hash keys to access quickly the objects in a window.

User interface. The user interface of the Animator consists of three parts: an animation window, a control bar and animated variables, see Figure 2. The control bar is used to start or pause the animation and to adjust the animation speed.

Animation. The animated variables are represented as colored boxes. The value of the variable is drawn inside the box. When a variable takes part in some animated operation, a temporary copy is made of the corresponding graphical object. The copy is moved on the window and the original one stays at a fixed position. This simplifies considerably the managing of the stage because the positions of the actual objects in the window remain unchanged.

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

The animation requests are made by calling a set of public methods of the Animator. At present, the Animator supports two types of requests, 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 in the upper part of the animation window.

2. If the result of the comparison is true, a green comparison operator is drawn between them. Otherwise, a red operator 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.

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 platform with method overloading, inheritance etc. For example, to include the char type to the a nimated types of system is straightforward. The only change in the Animator is to overload the interface methods of the Animator by char type of arguments.

5 Future development

We are gradually adding new features to Jeliot. We expect that Jeliot will soon reach the level of its predecessor Eliot. We will concentrate especially on the animation and better user control; the client side of Jeliot as well as the parser and code generator have been finished.

In the next version, the user as the director of the animation should be able to use multiple views to represent different aspects of a certain play. We expect to implement soon multiple windows which show different animations in parallel, based on one code. This means that there has to be a separate control panel through which the viewer can control each of the stages.

The restricted number of the animated data types in the current version of Jeliot is only temporary. We decided to start with the server and precompilation since they were necessary to test the concept. Now, we are ready to implement new animating data types. We will include more primitive types as well as some complex data types, for example Java's reference types. Some of these new structures will undoubtedly be new classes such as binary trees and lists.

Animated objects need to have more diversity in their appearance. Ford [6] shows that different programmers visualize even an integer variable very differently, not to mention more complex data types. This puts pressure to an animation generator like Jeliot where the user can only choose from a selected list of data types, provided by the program. — The authors got a nice confirmation of Ford's observation when they argued whether to use a concept "box" or "rectangle" to refer to the visualization of a variable.

Meisalo et al. [11] 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 user interface of Eliot and Jeliot are similar to each other, the criticism from the students applies also to developing Jeliot further.

At the moment the user has only two alternatives to determine the role of a certain self-animating data object in the play: appear or not appear. A wider range of possibilities to influence on how each actor performs its role should be given to the director. As mentioned before, the appearance of an actor with multiple choices and also the position on the stage are needed. We have plans to take this even further so that the roles could be changed also while the play is running. New roles could be added on one or more stages, or the way a role is acted could be changed.

The viewer should be able to know which part of the code is being visualized, whether he is learning an algorithm or debugging its code. One of the solutions is to highlight a line while animating it. It is useful to let the user to view the animation incrementally, line by line or frame by frame.

References

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

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

[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. Proceedings 1996 IEEE Symposium on Visual Languages, 1996, 266 – 275.

[5] J. Domingue and P. Mulholland: The Internet software visualization library. To appear in: Proc. PPIG 9, Psychology of Programming Interest Group, 9th Annual Workshop, 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] 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.

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