University of Helsinki
Department of Computer Science
Series of Publications A, No. A-1997-4

Erkki Sutinen, Jorma Tarhio
Simo-Pekka Lahtinen, Antti-Pekka Tuovinen
Erkki Rautama, Veijo Meisalo

Eliot — an Algorithm Animation Environment

Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland

Helsinki, Nov 1997

Contact Information

Postal address:

Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.helsinki.fi (Internet)

URL: http://www.cs.helsinki.fi/

Telephone: +358 9 708 51

Telefax: +358 9 708 44441

ISSN 1238-8645
ISBN 951-45-7912-7
Computing Reviews (1991) Classification: K.3.2, I.3.7, D.2.5
Helsinki 1997
Helsinki University Printing House

Eliot — an Algorithm Animation Environment

Erkki Sutinen, Jorma Tarhio, Simo-Pekka Lahtinen
Antti-Pekka Tuovinen, Erkki Rautama, Veijo Meisalo

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland

Report A-1997-4
Helsinki, Nov 1997, 45 + 4 pages
ISSN 1238-8645, ISBN 951-45-7912-7

Abstract

Eliot is an interactive animation environment, for visualizing algorithms written in the C programming language. Eliot provides a library of visual data types which are ordinary data types with a set of pre-defined visualizations. The user picks one visualization for each data object he wants to animate. Based on these selections, Eliot constructs an animation where the objects as well as their operations are animated. Examples of the use of Eliot is given and its implementation is described. Eliot has applications in algorithm design, visual debugging, and learning to program. The preliminary results of using Eliot in the data structures laboratory have been promising.

Computing Reviews (1991) Categories: K.3.2, I.3.7, D.2.5

Key Words and Phrases: animation, visualization, algorithm design, teaching of programming, debugging

Acknowledgements

We thank Arne Dybdahl, Timo Lamminjoki, Kjell Lemström, Sami Lilja, Jari Litola, Janne Markkanen, Teemu Möttönen, Vesa Ollikainen, Seppo Pekkala, Anssi Porttikivi, Kari Pursiainen, Veli-Matti Sahlberg, and Tommi Teräsvirta for their assistance.

The comments of Matti Mäkelä and Greger Lindén are appreciated. The work was financially supported by the Ministry of Education and the Academy of Finland.

Contents

Chapter 1

Introduction

Background. The intrinsic difficulty of learning, teaching, and designing algorithms seems to rely on their representation. To be useful for programming, they must be written in a concise and formal pseudocode. However, the facilities of a pseudocode are not actually oriented to present ideas and intuitions behind a certain algorithm. Animation is one way to cope with this problem.

Algorithm animation [6, 30, 36] refers to any mechanism which presents the running of an algorithm as a movie where the visual representations of objects of the program smoothly change their location and appearance, according to the script determined by the algorithm. Animation is usually realized as *event-driven*: the visualization corresponds to the operations of an algorithm on a given input.

A few years ago we implemented an animation package called Halsa[1] [38] for teaching of string algorithms. This system contained a selection of ready-made animations for a set of algorithms. Our experiences with Halsa led us to an important conclusion [15]: We noticed that the actual process of creating animations was more useful for learning than just watching ready-made animations, which are in fact similar to movies — one cannot experiment with them. Also Stasko et al. [37] referred to this phenomenon. Because it took even 100 hours to create a simple animation using the tools that were available to us at that time, we began developing new tools [14, 18] for generating animations. An outcome of this process is the animation environment Eliot, which reduces the required work time down to just a few minutes.

Main ideas. Eliot animates algorithms written in the C programming language by visualizing data structures as smoothly moving

[1] The original name Salsa has been replaced by Halsa due to a copyright conflict.

graphical objects. All the movements are connected with the operations of the data structures. Eliot offers a library of *visual data types*. It includes basic types, like integer, and structured data types, like tree. Each visual data type has a set of visualizations. The user picks one visualization for each data object he wants to animate. Based on these selections, Eliot automatically creates a full animation of the algorithm.

The system was named Eliot after the Finnish word *Eliöt*, which means living organisms. The current implementation runs on Linux workstations or on X terminals.

Eliot presents two contributions to the field of program visualization: ease of use and an innovative implementation technique. The design of Eliot was based on the following view of 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 language. To facilitate diverse visualizations, the visual semantics must not be fixed but adjustable: for example, it must permit integers to be represented either as bars or circles. The user interface allows the user to define the visual appearance of the animation, i.e. to adjust the parameters of the semantics.

This approach raises questions. One might classify it as a pure visual debugger, based on one-to-one mappings from program structures to visual representations. Any automation of algorithm animation has been contradicted e.g. by Brown [7] and Stasko [35]. They argue that in general there is no one-to-one correspondence between the statements of a program and the images of the animation. Our approach avoids the argumentation of Brown and Stasko, since we base the animation on visual data types: the algorithms are written using these data types, not plain (untyped) C. By using the abstract types instead of basic types, the level of abstraction will become higher. For example, if an algorithm uses binary trees, all the operations are written as the operations on that abstract data type. This scheme lets Eliot utilize the pre-defined visual semantics of the data types. In addition, the user can fine-tune the mappings from data objects to their visual counterparts within the user interface.

Applications. An animation environment like Eliot has naturally a variety of applications, especially in the areas of teaching, learning, research, and problem solving. Besides being a useful tool for these areas, Eliot also serves as a unifying factor for them: working with algorithm animation can be an exciting challenge, flavored at the same time by learning and discovery. The key factor of the applicability of Eliot is ease of use combined with the speed and the strength of the self-animation paradigm.

A natural application of automatic animation is debugging. Eliot can provide the programmer with a more sophisticated view of the program than standard debuggers. The animation nicely supports the fast detection of programming errors. The usefulness of Eliot in this context is not limited to C. By slightly modifying Eliot, Simo-Pekka Lahtinen [13] was able to implement a visual debugger for the Scheme programming language.

A researcher can use visualizations generated by Eliot to analyze the behavior of an algorithm with different inputs and to observe the strengths and weaknesses of the algorithm. Eliot gives him immediate feedback of the algorithm's behavior. Thus the visualization produced by Eliot reflects his understanding of the algorithm and leads hopefully to its improvement.

From a teacher's point of view, Eliot is an appropriate choice for teaching programming. The teacher can show how a particular algorithm works on various inputs. Since Eliot provides practical mechanisms for creating animations on-the-fly, the teacher can also discuss the algorithm with the students and experiment how modifications change the behavior of the algorithm. The visualization helps him to explain the reasons for odd features of a modified program.

Eliot is a useful tool in algorithm design (see Chapter 8). The actual process of constructing an animation for an algorithm leads to a profound understanding of it.

The AAPS project. The Eliot project has inspired a new research project, called AAPS (Animation-Aided Problem Solving), financed by the Ministry of Education. Both computer scientists and educationalists participate in this project. The project is based on the relationship between computational and other problems. Algorithms are solutions for the first kind of problems. While animations help a designer to work on better algorithms, there are other computer-aided techniques — yet to be discovered — which might serve solvers of other problems, e.g. those related to school work.

The motivation for the AAPS project comes from education: at school as well as in industry training, more and more attention is paid to learning metacognitive skills, like problem solving. We investigate how computers could be applied to promote this direction, especially by helping their users to visualize and animate the problem solving process. One of the promising research areas in this respect are various methods for visualizing concept maps.

Currently, we are working on algorithm animations on the Web [10]. Our goal is to support web-based cooperation in algorithm research. However, we have not forgotten light solutions at the other end of the spectrum: we have found that it is also possible to use spreadsheet software as an animation environment [28]. In cooperation with cognitive scientists, we are evaluating the power of animations as a research-like learning tool, e.g. how animation helps in analyzing an algorithm.

Outline. This report, which is based on articles [16, 17, 22], is organized as follows. We give an example of the use of Eliot in Chapter 2. Then we describe the implementation of Eliot. In Chapter 4 we give instructions of use. In Chapter 5, we consider an example of using Eliot in algorithm design. Related work is reviewed in Chapter 6. In Chapter 7 we review a case study on the evaluation of Eliot. Before conclusion we introduce Jeliot which is a sister system of Eliot, working on the Web, in Chapter 8.

In Appendix B there is a bibliography of publications and reports related to Eliot.

Chapter 2

Example of Use

We present a C program which, starting from an empty binary search tree, inserts six nodes to the tree (Fig. 2.1). The declaration[1] `tree int T` defines a visual variable `T` to hold a binary tree object with integers as key values. In addition, pointer variables `s` and `t`, pointing to a binary tree, are defined in `treeP s,t`. The insertion algorithm uses `t` to point to a parent node while `s` points to its child, in the direction defined by `branch`. — Note that the values of the nodes are only explicitly assigned to avoid unnecessary details in the resulting visualization.

The algorithm utilizes the following methods of the tree class: The method `Mattach(i)` creates the root node with the value `i`, `Mget_root()` returns a pointer to the root node, `Mget_child(t,d)` returns a pointer to the child of `t` in the direction `d`, where `d` takes one of the two values `LEFT` or `RIGHT`. Method `Madd_node(t,d,i)` creates a node with value `i` and adds it as a child of `t` in the direction `d`. The public method `Mreference()` gives the key value of a node.

Fig. 2.2 shows four snapshots of the corresponding animation.

As a matter of fact, Eliot requires the programmer to code the binary search tree algorithm with the animated tree operations, because pointer variables are not visualized. Of course, the user could implement the tree as an array, but the automatically animated array would not display as a particularly intuitive visualization of the tree! This limitation has, however, an advantage: it makes the programmer consider the algorithm at a higher abstraction level, compared to the implementation-orientation of the pointer technique. The abstraction

[1]This declaration is not legal C code. Due to a limitation of the parser analyzing the program, abstract data types must be indicated explicitly in the source code.

```
array int a[6];

main ()
{  tree int T;
   treeP s,t;
   int temp,branch,i;

   a[0] = 8; a[1] = 2; a[2] = 1; a[3] = 6; a[4] = 4; a[5] = 10;
   temp = a[0];
   T.Mattach(temp);
   for (i = 1; i < 6; i = i+1)
   {     s = T.Mget_root();
         while (s <> NULL) {
            t = s;
            if (t->Mreference() > a[i] ) {
              branch = LEFT;
              s = T.Mget_child(t,LEFT);
            } else {
              branch = RIGHT;
              s = T.Mget_child(t,RIGHT);
            }
         }
         temp = a[i];
         T.Madd_node(t,branch,temp);
   }
}
```

Figure 2.1: *A C program inserting six nodes to a binary search tree. The method* `Mget_child` *is used to traverse in the tree, depending on the inserted value* `a[i]`*: for example, if* `a[i]` *is smaller than the value of the current node, pointed by* `t`*, the algorithm branches into the left direction. Hence, the next compared node, pointed by* `s`*, is* `T.Mget_child(t,LEFT)`*. When the algorithm reaches a node* `t` *with no child in the appropriate direction, the value of* `s` *will be* `NULL`*. Then the algorithm is ready to insert a new node with value* `a[i]` *as a child of* `t`*. This is done by the method* `Madd_node`*.*

provides the algorithm with a natural visualization; at the same time, it contributes to a correct code and its self-explanatory animation.

In the current implementation of Eliot, the type of a node of a tree is limited to simple types. For example, the nodes cannot have an array as their contents.

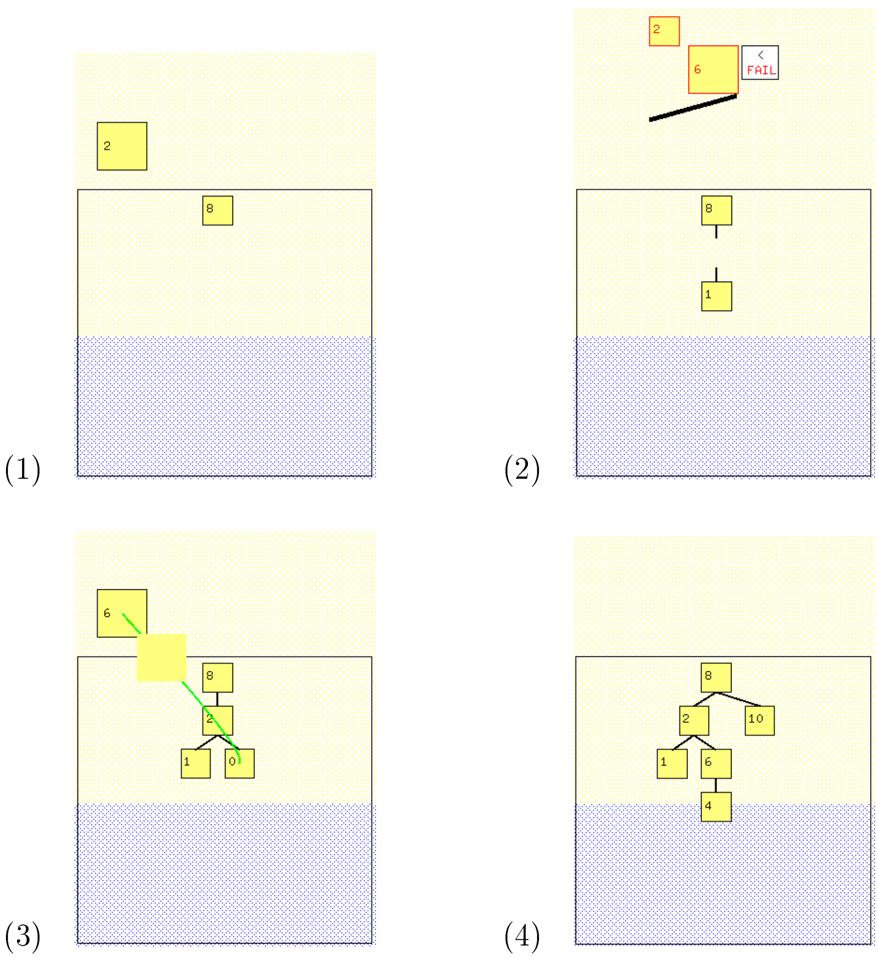

Figure 2.2: *Four phases of the binary tree animation for inserting items 8, 2, 1, 6, 4, and 10. In phase (1), the algorithm inserts the second item with value 2 into the tree. In phase (2), after inserting the first three items, the algorithm compares the fourth inserted item with value 6 to the node with value 2. The* `FAIL` *message results from the comparison* $2 > 6$ *in the algorithm. Phase (3) shows the inserted value 6 sliding into its right location. Finally, phase (4) represents the tree with all the inserted values.*

Chapter 3

Implementation

3.1 Design Principles

At our department, program visualization and algorithm animation in particular have been studied in four student projects [1, 5, 14, 18]. In designing Eliot, we have used the accumulated knowledge of algorithm animation tools and techniques acquired during the earlier projects. We believe, that Eliot presents two important contributions to the field of program visualization: ease of use, and the innovative implementation technique. In this section, we briefly present the novel ideas behind Eliot.

3.1.1 The Theater Metaphor

The predecessor of the Eliot project, Halsa++ [14], introduced a *theater metaphor* to algorithm animation. A revised version of this model was used in the design of Eliot.

We developed further the *theater metaphor* to describe algorithm animation with Eliot. According to the model, a program is regarded as a script of a play. The play, or the visualized program, involves a number of *roles*. An *actor* performs a role. The *performance* of each role varies according to the *direction* of the play. Therefore, a play may have many simultaneous directions on *multiple stages*; similarly, an algorithm might have different visualizations on multiple animation windows.

A visualization is a collection of concurrent directions of a play. The user (or the animator) is the creator of the visualization; the user may both direct the play as well as write the script. The audience consists of the viewers of the visualization, including the animator.

Roles. The roles correspond to the data objects of the algorithm; more exactly, to the data objects of the visual data types. The interactions between the roles consist of the computational operations, namely assignments and comparisons, etc., that are performed on the data objects.

All the data objects of the algorithm, like temporary index variables, need not necessarily be represented as graphical entities in the animation. The theater metaphor offers a natural fulfillment of this requirement: Following his interpretation of an algorithm, the user as the director of the play may choose to visualize only the data objects that are crucial to understanding the algorithm.

To the script writer (i.e. the programmer), the roles appear just like any other program objects. They can be used with other objects and the same operations can be applied to them. The only difference is that the user may animate the roles on the scene by directing the actors playing the roles. This difference is, however, not seen in the code of the script. It is the task of the Eliot compiler to recognize the roles of a script which can be animated and provide the user with a list of these roles.

Actors and performances. In the theater, the roles in a script are played by actors. In an Eliot animation, an actor is a graphical entity that has a set of visual attributes like size, shape, color, and location. For example, an integer variable, with a value 3, may be animated as a yellow circle with a relative radius 3 at the left bottom of a stage window. Each actor may have a different appearance, i.e. different values for the visual attributes.

The correspondences between roles and actors are defined as *associations*. In the previous example, the integer variable might also have another interpretation as a rectangle. We say that the role of the variable is associated with two actors (naturally, on different stages), the circle and the rectangle, reflecting different users' mental models of a variable; both of the actors are associated with exactly one role, namely that of the variable. Generally, an actor is associated with only one role, but a role may have multiple actors associated with it.

The user starts the animation process by writing a script. For example, an assignment $x = 2$ will be represented as 2 sliding towards an actor representing the role of x. Then he directs the play: he decides that x will be represented as an actor with a rectangular shape. The sequence of transformations of an actor, whether caused by the script writer or the director, is called the performance of an actor.

In Eliot, the constructing scheme is semiautomatic: the basic performance is defined automatically by the code while the user can fine-tune it in the second phase, according to his own mental model. In Eliot, all the simultaneous directions on different stages are synchronized. The result is a smooth animated movie advancing simultaneously on multiple windows. The movie may look different in each window, since one window may be tuned to give a good overall view, whereas another might be focused on a detailed item.

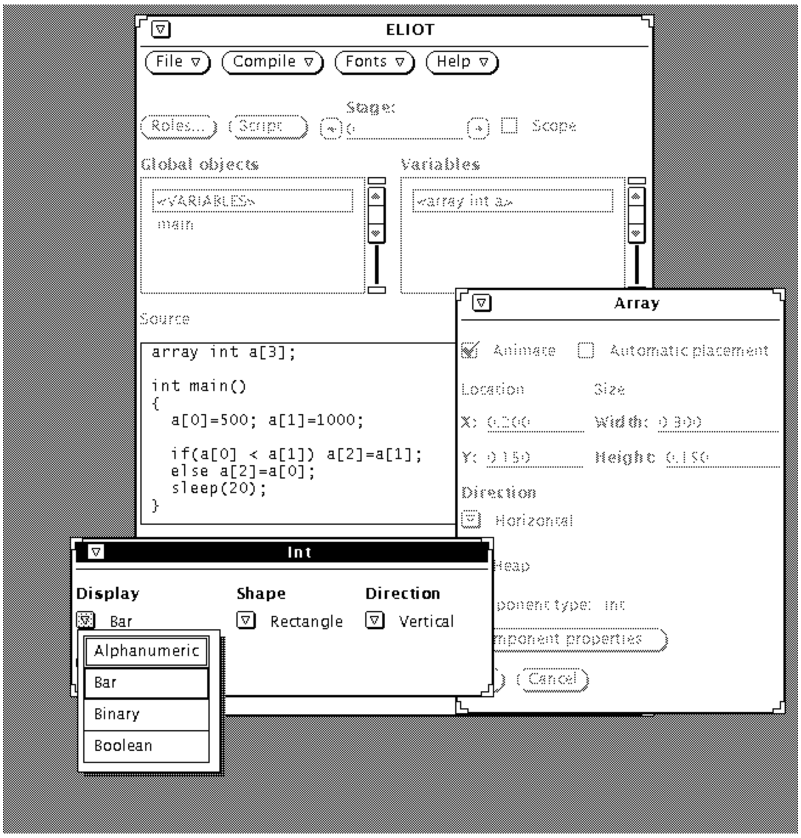

Figure 3.1: *The Eliot user interface.*

The Eliot theater. In principle, a role should be played according to the director's imagination. However, the available actors limit the director's freedom. As a theater with expressive actors, Eliot supports several object types to be animated, like simple variables, arrays, and trees. This is an essential feature for an animation environment like Eliot which gives the user no access to low level animation primitives.

The preselected set of animated object types with their operations is usually enough for the user of Eliot.

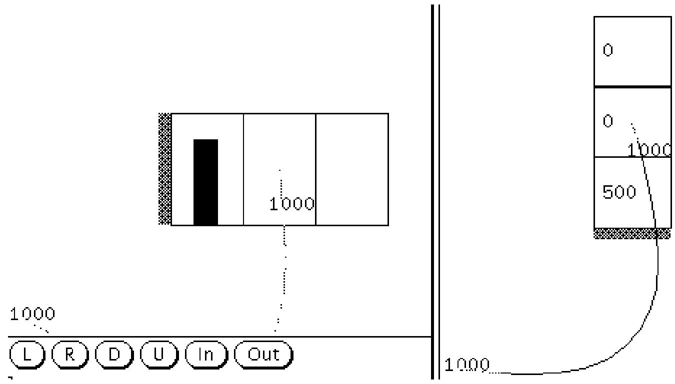

Figure 3.2: *Two different representations of an array.*

Example. To clarify our terminology, let us consider our example algorithm, which is listed in Fig. 3.1. The actual program listing is the script. It is played on two stages, although this is not indicated by the script, but rather by the direction: The user has entered parameters for two stages in the user interface. The array is the only role of the play. On one stage, it is played by an actor which displays the contents of the array as vertically aligned numbers, and on the other stage by an actor which displays the contents as horizontally aligned bars (Fig. 3.2). Note that a logarithmic scale is used in the bars. The performances of the actors include value assignments, in which the value literally enters the bounds of the actor, and comparisons, in which two actors are always involved (Figs. 3.3–3.5).

Also other authors have used the theatrical metaphor to describe a program in action. For example Agentsheets [29], a software designed for generating animated representations for various application domains, is based on a *participatory theater* model, since it lets the users direct the "play" without stopping it. The theatrical metaphor was introduced by Negroponte [27] as a replacement of the desktop metaphor: in the future, users will delegate the agents of the interface to accomplish tasks. Following these perspectives, we could equip the future Eliot with an intelligent *tutor*, guiding the user in the world of animated algorithms.

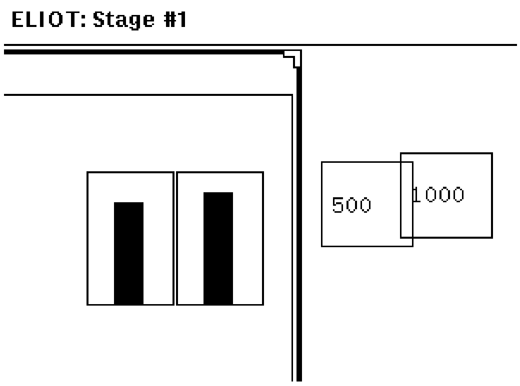

Figure 3.3: *The two first array items are brought to the upper right hand corner for comparison.*

3.1.2 Self-Animation

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 [7]. In Eliot, animation is controlled by the operations of visual data types, and the user does not need to write any additional procedure calls. Each visual data type has a collection of visual representations, and operations of a data type have pre-defined visualizations. When the animation program is run, the animation is performed as a sequence of the visualized operations. We call this approach *self-animation.*

Self-animation is a variation of the interesting event approach: the events are connected with the operations of the data types. Self-animation has several advantages. The algorithm and animation codes are not separated and data not duplicated as in the standard interesting event approach and therefore the reuse of the code is easier. The preparation of an animation for a new algorithm is fast.

The implementation of self-animation in Eliot relies on the ability to overload operators in the underlying implementation language, C++. Operators of the visual data types have been redefined to produce animation as a side-effect. Alternatively, it would have been possible to add the calls of relevant animation primitives to the algorithm by preprocessing, if the implementation language had not supported overloading of operators. VCC [3] and AAPT [32] use the latter approach.

The benefit of self-animation is significant, when the algorithm is coded using abstract visual types. Let us take an example. Suppose

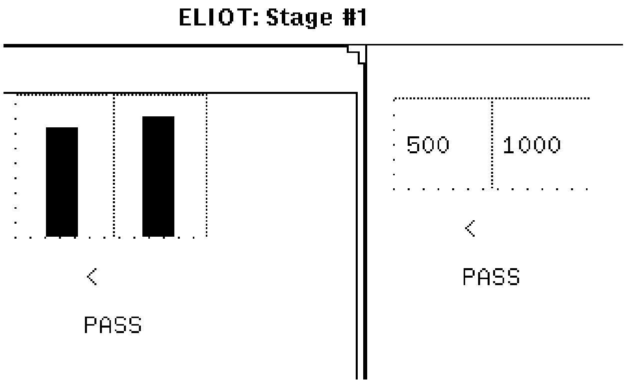

Figure 3.4: *The result of the comparison is displayed, after which the items will be brought back to their original positions. The result of the comparison is also illustrated with a color change to green or red.*

that a binary tree is implemented as an array. Instead of coding a traverse in the binary tree by adding and subtracting an index variable i, we use the binary tree data type with its operations. These operations have their conventional visual counterparts while the general-purpose arithmetic operations on i could be used to denote any book-keeping of the algorithm, without any relevance to the binary tree.

To implement a visualization for a new data type is not a trivial task. So far we have implemented the following visual data types: integer, real, character, array, and tree. More structured data types such as list, stack, and graph will be added to the future versions of the system.

3.2 Overview of Implementation

Eliot has been implemented in the X11 environment and it runs on Linux workstations. Eliot applies animation primitives of a general algorithm animation library, Polka [34]. As the back-end system, Polka provides the graphics and display facilities and high-level system functions for producing smooth and visually appealing animations.

Eliot consists of three components: a library of self-animating classes that uses the services of the Polka package, a graphical user interface, and a compiler. There is a global object in the class library that maintains internal data structures. It also provides methods for managing structured types and passing information from role class

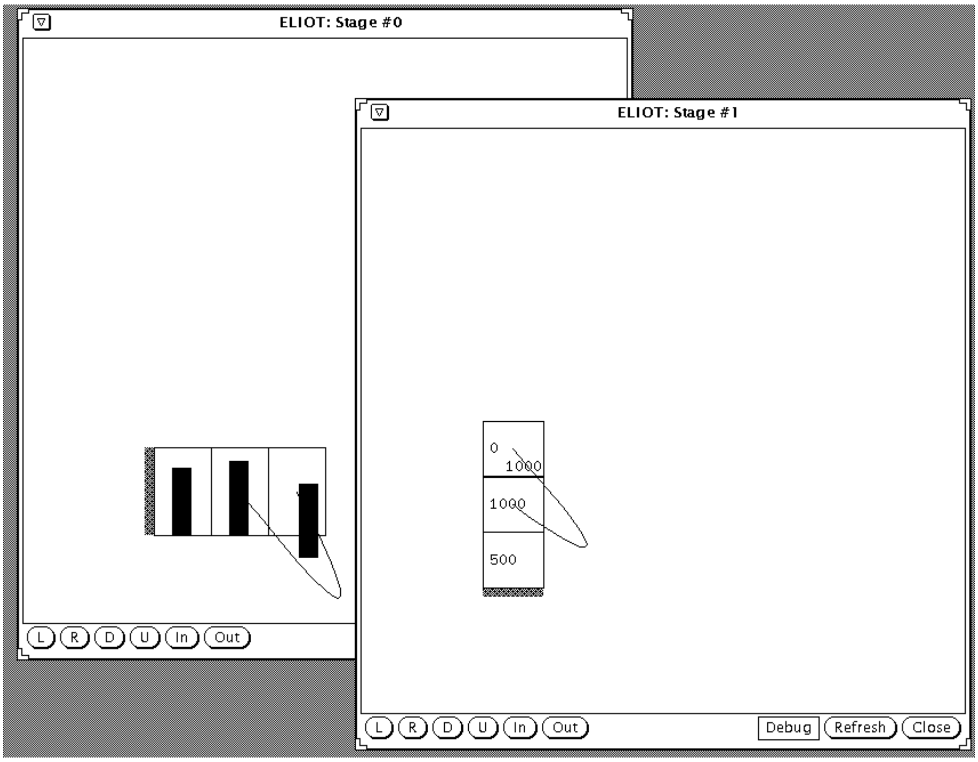

Figure 3.5: *Finally, the third array item is being assigned the value of the second item.*

instances to actor class instances.

The user interface has been implemented using the wxWindows library [33] and the compiler has been generated with lex and yacc. The compiler containing parsing and code generating modules is called by the user interface whenever necessary. The process of constructing an animation is illustrated in Fig. 3.6.

Because the compiler was generated with yacc, it is not difficult to change the language of the input algorithms from C to some other language. We have plans to make at least a Pascal version.

To clarify the characteristics of the implementation of Eliot, let us study the principle of the Polka package (Fig. 3.7). Polka gives the user tools to base the animation on the mappings between arbitrary algorithm operations and animation scenes. These mappings require explicit definitions. In practice, an animation for an algorithm is constructed by inserting calls of animation primitives to several places in the code of the algorithm.

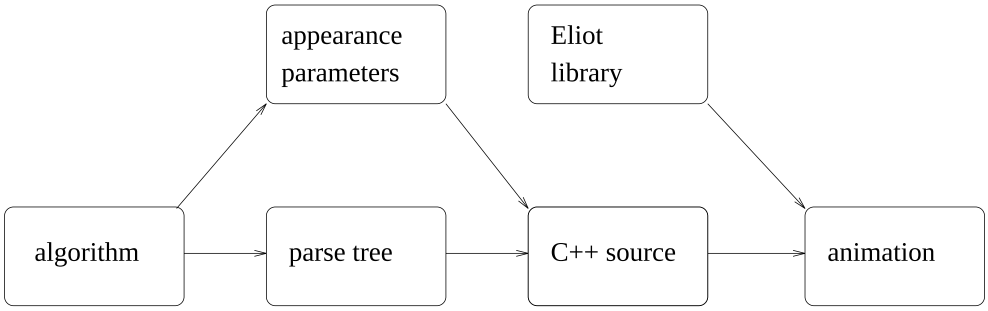

Figure 3.6: *From an algorithm to animation. The parser constructs a parse tree and default appearance parameters from the algorithm. The user modifies the appearance parameters. The code generator produces a C++ source program. The resulting animation program is compiled and linked with the Eliot library.*

In the Eliot animation environment, the user just loads the code of his algorithm and defines the appearance of the animated data objects in the user interface. Based on this information, the Eliot library produces the animation using the visual data types. Eliot needs only those routines of Polka that are needed to produce the graphics, see Fig. 3.8.

3.3 Technical issues

3.3.1 The Eliot Class Library

The Eliot class library implements several classes of visual data types, as well as the mechanism to actually produce the animation. Figure 3.9 shows the core of the library as an OMT class diagram. The class `Halsa` is the core of the animation system. All roles (or, animated instances of visual data types) use methods of a global Halsa instance named `halsa` to produce any animation they require.

The class `Eliot` is the base class of all other Eliot classes. The class `Role_a` is the base class of all animated classes, implementing some basic visualization routines, which may be overridden by the inheriting classes. Each role knows its number, or numeric identifier allocated by the global `halsa` instance. Currently implemented visualized classes that inherit the `Role` class include `Int_a` (integer), `Char_a` (character), `Array_a` (array, compound object class), `HRole_a` (a base class for

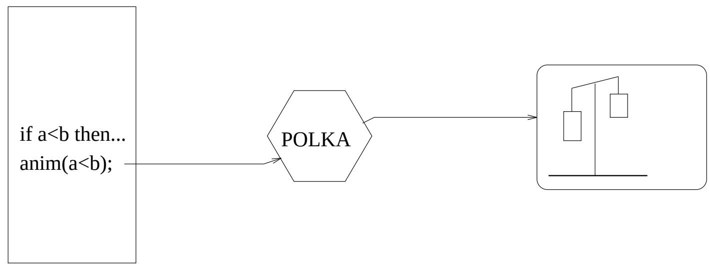

Figure 3.7: *The user of the Polka animation library inserts calls of the animation routines into the animated program. The animation routines are implemented as the Polka library. The architecture provides the user with a large set of options. However, the process of developing an animation takes too long for teaching or learning purposes.*

objects that can be pointed to by handle pointers), etc.

The management of stages is handled by the class `Stage`. A Stage object keeps e.g. track of which locations of the stage are currently occupied, and allocates an unused location to an actor. The class `Actor` objects are generic, and have no idea of which role object they represent. The role itself produces the visual representation of its current state. The actor class methods implement routines such as `move, resize, compare` with another role, etc.

3.3.2 Starting a Visualized Program

A global `Halsa` object, named `halsa` is defined by the Eliot library. When the program is started, the object is created, causing its constructor routine to be executed. It generates a requested number of stages, initializes its internal arrays, etc. What can be seen on screen is just a number of empty windows. Then, the visualization is displayed in the windows while the program executes.

3.3.3 Creating an Animated Object

The process of creating a self-animated object is illustrated in the upper section of the message sequence chart shown in Figure 3.10. When a visualized object (any class inherited from class `Role_a`) is created in the program, its constructor function calls a `Mregister` method of the `halsa` object. The registration method allocates an

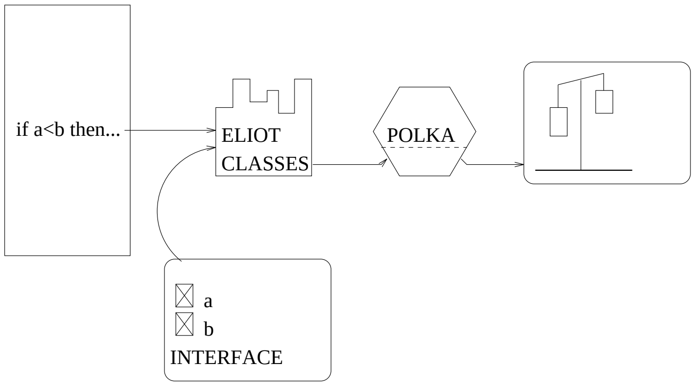

Figure 3.8: *The user of Eliot writes only the algorithm, without any additional animation calls. In the user interface, the user selects the animated objects. The animations are produced by using the self-animated classes of the Eliot library. Eliot applies only a subset of Polka, the graphics routines.*

identifier number (or, role number) for the role, and enters it into the role array. Then, the `halsa` object creates one `Actor` object for each existing stage.

The constructor function of an actor then asks the actual role object to provide a visual representation of its current value. Should the role be an integer object with a value of 10, the representation might be a number 10 surrounded by a bounding box. Naturally, the representation may be more complex than this. When all actors have been created, the animation is advanced by one frame, causing the actors to appear on each existing stage.

If the role is a compound object (e.g. an array consisting of a bounding box and a number of component objects), the constructor function of the role has to create the component objects as well. These will in turn have their own constructor functions executed, and a visual representation is thus created for them as well. After this, the host object will allocate a space for each component object by using the `assign_slice` method of the global `halsa` object. The request is forwarded to each actor. The actors then place and resize the component object actors to proper locations.

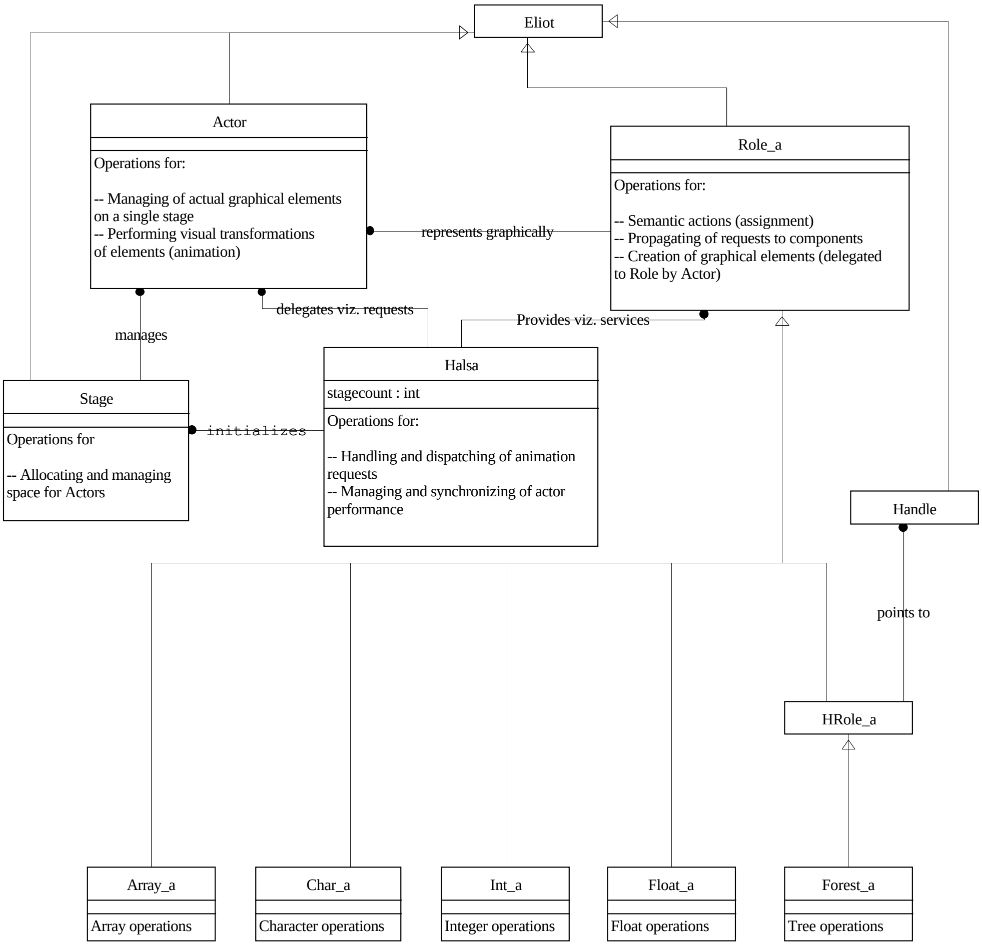

Figure 3.9: *Eliot class hierarchy.*

3.3.4 Animating an Operation

Assume that we have a class that implements a regular integer type, but that it has a visual representation of a thermometer. E.g. when an instance of the class is assigned the value 10, the visual thermometer on each stage begins to show a temperature of 10 degrees. Then, we have a line of code which assigns the thermometer object a new value. The class `Thermo_a` should have an overloaded assignment operation, which would be invoked. See Figure 3.10.

First, the thermometer object would request the global `halsa` object to visualize the assignment. By default, it would cause the `Actor` object representing the value being assigned to move along a curved path to its destination — on top of the `Actor` of thermometer object. The `halsa` object would then request the actors on each stage to do the actual job.

Second, the new value would actually be stored in the private storage (a regular integer variable in this case) of the thermometer object holding the value. No visualization is produced in this step.

Third, the thermometer object (which is a role object) asks the global `halsa` object to update the visual representation of the role. This request would also be forwarded to actors on each stage representing the role.

A visualized operation might also be a change of visualization attributes. The programmer might decide to change the color of a visualized object, or move it to a different location. A request is handled by the global `halsa` object, which then forwards it to each actor. After the actors have completed the operation, the `halsa` object uses a callback function of the role object, requesting it to forward the request to its component objects, if any. This way, for example when an array is resized, its component objects will also be handled accordingly. However, these operations have no counterpart in the original C program on the *role level* — they merely improve the visual appearance of a program on the *actor level*.

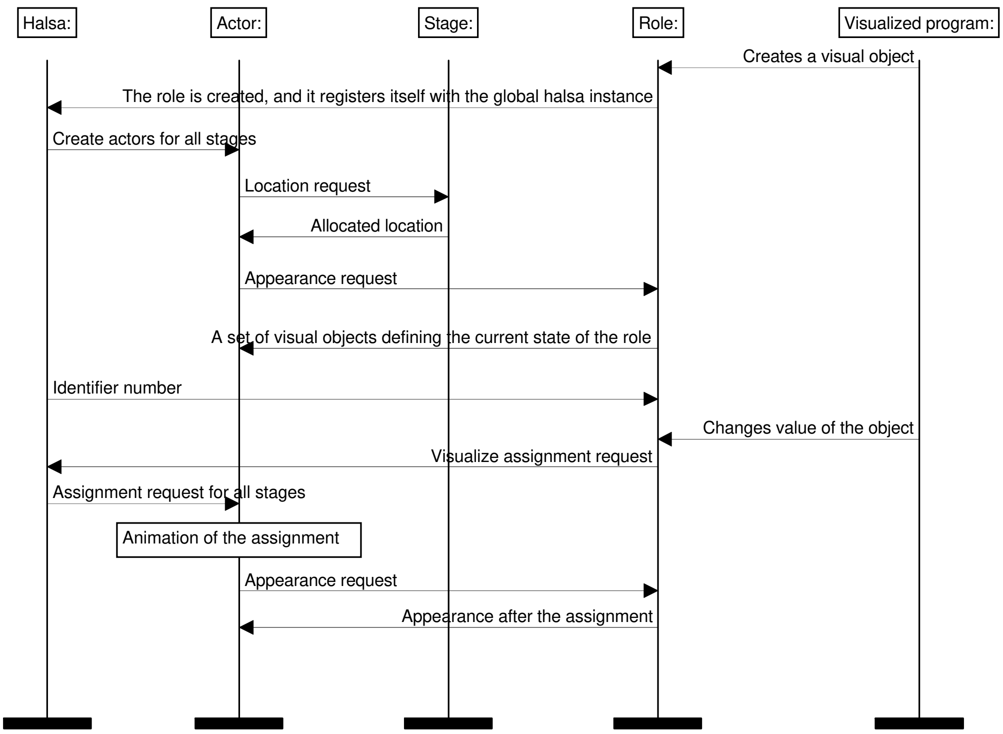

Figure 3.10: *The interactions between class instances when an animated object is created and a value is assigned to it.*

Chapter 4

The Use of Eliot

We will present an example of producing an animation with Eliot in order to introduce the features and possibilities of Eliot.

We fixed Bubblesort as the example algorithm to be animated. Although Bubblesort is a simple algorithm, it allows us to demonstrate the use of Eliot from various aspects. The source code — written in Eliot-C — is shown in Figure 4.1. This code is given as a file to Eliot.

Note that the only difference of this code from the standard C is the way an array is declared. A description of the Eliot-C language is given in Section 4.7.

4.1 Loading the Algorithm

After launching Eliot it opens the main window, shown in Figure 4.2. We load our algorithm into Eliot by choosing the Load item in the File menu. The Load operation opens a file selection window seen in Figure 4.3. While loading, Eliot parses the code. In the case of parsing errors, Eliot refuses to finish loading and explains the reason in a dialog window. The code should be revised before trying again.

4.2 The Properties

Next the variables to be animated should be selected. For each animated variable the user may set the values of visual parameters or accept the default values. From the list labeled *Global objects* we choose the item <*VARIABLES*>. This brings up an item *array int table* in the *Variables* list. Animated variables are shown surrounded by angle brackets. Clicking that item a new window opens where we can choose

```
#include <stdlib.h>
#include <time.h>

array int table[12];

void init_table() {
    int i;
    srand(time(NULL));
    for (i = 0; i < 12 ; i = i + 1)
        table[i] = rand() % 200;
}

void main() {
    int i, j, temp;
    init_table();

    for (i = 11 ; i > 0 ; i = i - 1)
        for (j = 0 ; j < i ; j = j + 1)
            if (table[j] > table[j+1]) {
                temp = table[j]; table[j] = table[j+1];
                table[j+1] = temp;
            }
}
```

Figure 4.1: *The bubblesort algorithm*

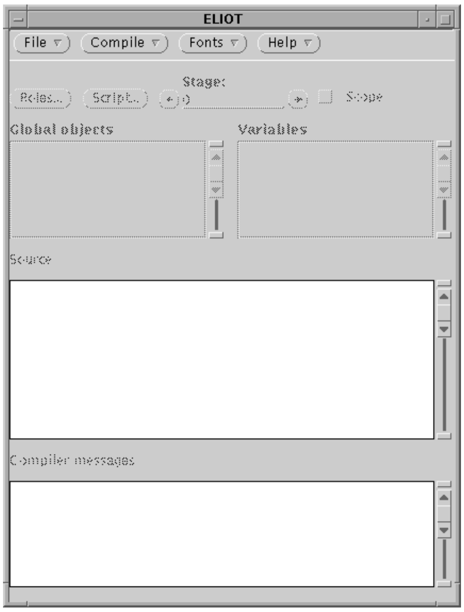

Figure 4.2: *The main window*

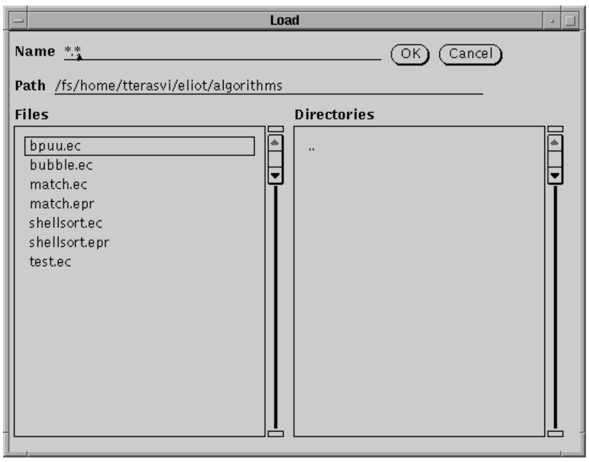

Figure 4.3: *Loading a file*

the properties for that variable. We decide to *animate* the array and let Eliot use *Automatic placement* to place it on the stage.

Next we click the *Component properties* button. A new window opens up. In this window we set the appearance of the components of the array. This time we make changes. We choose *Bar* from the

Display list and then change the *Direction* to *Vertical*. Then we click the *Ok* button twice to close the windows.

We choose other variables to be animated by selecting the item *main* from the *Global objects*-list. The *Variables* list shows the variables in that function: *int i*, *int j* and *int temp*. If we choose to animate the variables *i* and *j*, we click those and just tick the *Animate* box.

Now we are ready to generate our first version of the animation. The animation will run on a stage mostly based on the default settings of the chosen variables.

To get more stages we could change the number labeled *Stage* by clicking on the appropriate enlarge/decrease buttons. Animated variables and their animation proprerties should be set on each stage.

4.3 Saving the Properties

After carefully designing an animation it is often useful to be able to preserve all the properties to edit the animation later on. With Eliot this can be done in two ways: Saving only the properties, or saving the properties with the code. Either of these operations can be selected from the menu under the submenu *File*.

4.4 Compiling

After the properties have been set, it is time to generate the C++ file which has the variables that are to be animated as visual data types, and then compile that file with the C++ compiler.

The compilation can be done by choosing the menu item *Compile-Compile*, which first generates the C++ file and a makefile to be used in the compilation, and then calls the compiler. The compiling can also be done by generating the necessary files by *Compile-Generate* and then compiling with the Unix command

```
make -f Makefile.bubble
```

Which ever way the compilation is done, the result is a binary file which includes animation code.

4.5 Animation

The animation can be run from within Eliot by choosing the item *Compile-Run* from the menu, which – if needed – first generates the

C++ file and the makefile, then makes the compilation and starts the animation. If the program is already compiled, the animation can be started from the terminal – in our examplecase it is named *bubble*.

When the animation is finally started two windows appear: *The Polka Control Panel* and a stage with *number 0*. The Polka Control Panel is used to control the speed of the animation, while the actual animation takes place on the stage(s).

At first the animation is in *pause* state. It is to be noted that regardless of how many stages there are going to be, only the first one is visible in the beginning. Pressing the *Step*-button makes the rest of the stages visible.

The Run button starts the animation. The speed of the animation can be adjusted using the slidebar. When the animation is paused the *Step* button advances it by one animation frame at each click.

On each stage, the variables to be animated are shown, as well as the assignment and comparisons performed on those variables. When the animated program ends, all the stages are closed immediately and the Control Panel disappears.

4.6 Changing the Animation

The first version of the animation is usually not the final version. Either the positions or appearances of some variables are not appropriate or there may even be something wrong in the algorithm itself. Changing the properties is done as explained in section 4.2.

If there is a need for changing the code, it should be done with an external editor, because there is no editor in Eliot. After the changes have been made, the new code is loaded into Eliot using the *File-Load* menu selection.

When loading the code, all the selections for the properties of the variables are lost, unless they are saved with the *File-Save Properties* menu selection. Then they can be retreived by *File-Load Properties*. In case of new or missing variables, Eliot either sets their properties to default values or discards the missing properties.

4.7 Eliot-C

Eliot-C is the dialect of the C programming language accepted by Eliot. It differs only a little from the standard C. In the following we describe some features and limitations of Eliot-C.

4.7.1 The Data Types of Eliot-C

In addition to standard C data types, Eliot-C includes animated array and tree data types. An array can contain integers, floats or chars. In the current implementation tree can contain only integers.

An array is declared as:

```
array basetype a[size];
```

where `basetype` can be of the type `int, float` or `char`. For example, an integer array of size 10 is declared like this: `array int numbers[10];`

An animated array cannot be dynamic so it's size is fixed.

A tree is declared as:

```
tree basetype a;
```

At the moment the basetype can only be of type `int`. Pointers to the tree are of type `treeP`.

4.7.2 Operations

In Eliot-C, an animated array can be used just like a standard C array. The only limitation is that the size of an array is fixed, so e.g. the `realloc()`-function cannot be used to change it.

Because C does not have a tree data type, Eliot-C is extended to contain operations that can be used to manipulate trees. A node in a tree can have an unlimited number of subtrees (of course very wide trees do not fit on the scene window).

The tree operations include:

Operation	**Purpose**
Mreference(*node*)	Returns the value of the *node*.
Mattach(*value*)	Creates the root node of a tree and assigns *value* to it.
Mget_child(*node, n*)	Returns a pointer to the n'th child of the *node*.
Mget_root()	Returns a pointer to the root of the tree.
Madd_node(*node, n, value*)	Creates the n'th child of the *node* and assigns *value* to it.
Mremove_node(*node*)	Removes the subtree with root *node* from the tree.

Chapter 5

Algorithm Design with Eliot

Sorting a given material to a particular order is a well-known algorithmic problem, which is fairly easy to describe and to understand. Although there is hardly anything new to be discovered in this problem, we will use it as an example for describing how to solve algorithmic problems with Eliot. We will make an attempt to produce a new sorting method.

The first question to be answered is "which general approach to use". Each one would produce a different kind of algorithm. Examples of possible algorithms are listed to illustrate this. However, not all alternatives produce good sorting algorithms. Listed below are a few examples of possible methods.

- Divide-and-conquer: Divide the items into two groups, then sort both groups separately, and merge them together by taking always the smallest of the first items of each group.

- Greedy algorithm: Take the first five items, then go through the rest of the material. If any of the items that you are holding is smaller than the one that you encounter, toss the smaller one out and pick the larger one. When you have gone through the items, you have found the five biggest items. Sort them, and repeat the process until no more items are left to pick.

- Stack of playing cards: Pick one card. Go through the rest of the cards, and find the card that is the greatest card in value below the card you picked, and the card that is the smallest card in value above the card you picked. Then, glue the three cards together, and repeat the process until you have 52 cards glued together.

```
#include <unistd.h>

array int a[10];                      // The numbers to be sorted.

void sort(int left, int right)
{
  int i;
  int temp;

  if(right > left) {                          // Have we more than one?
    for(i=left;i<=(right-left)/2+left;i++) { // Loop to do the comparisons.
      if(a[i] > a[right-(i-left)]) {         // Are these two numbers in
                                              // wrong order?
        temp=a[right-(i-left)];               // If yes, then switch them.
        a[right-(i-left)]=a[i];
        a[i]=temp;
      }
    }
    sort(left, (right-left)/2+left);          // Sort from left to center.
    sort((right-left)/2+left+1,right);        // Sort from center to right.
  }
}

int main()                              // The program actually begins here.
{
  int i;

  for(i=0;i<10;i++) a[i]=10-i;          // Generate a reverse-order
                                        // list of numbers.

  sort(0,9);                            // Sort zeroth through ninth number.
  sleep(20);
}
```

Figure 5.1: *The code of Stupidsort in Eliot-C.*

The algorithm we will create is based on the divide-and-conquer technique. The task is to sort a list of numbers in increasing order. We will call the new algorithm "Stupidsort", which name has, of course, nothing to do with the actual performance of the algorithm. We try to get a working algorithm by including the following steps:

- Check if you have more than one item to sort. If not, the list is obviously already in order, so do not continue with this process.
- Compare the first item with the last item. If the first item is smaller than the last item, do nothing. If the first item is greater than the last item, switch the items.
- Do the same to the second and second-last item, third and third-last item, etc. until you reach the middle item.
- Now all small items are in the first half of the list, and all large items are in the second half of the list. First, put away the second half of the list, and order the first half using this procedure. Then, put away the first half of the list, and order the second half using this procedure. Then, put the first half on top of the second half. The list is now in order.

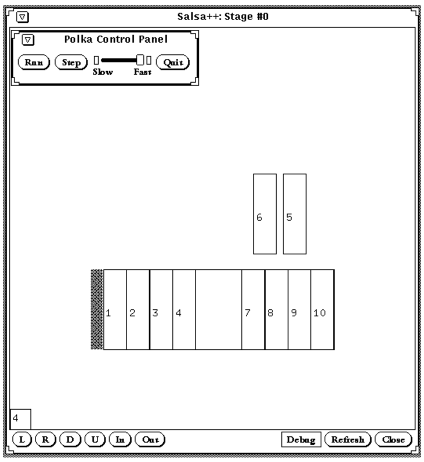

Figure 5.2: *The sorting algorithm is processing a reverse-order input. The array is drawn automatically by Eliot.*

We programmed the code shown in Fig. 5.1 and run it. When run, Eliot will automatically produce the animation on the screen, which gives more information on the operation of the algorithm than the mere output or the listing of the code.

In Fig. 5.2 there is a snapshot of the animation, when the program is processing a reverse-order list of numbers. The resulting list is clearly in order.

When we try to order another kind of input list, we encounter a situation shown in Figures 5.3, 5.4, and 5.5. We now draw a conclusion that the sorting method we have designed apparently does not work correctly with *all* inputs. Yet it does work with *some* inputs. We could try different lists, such as 7 3 2 9, and test how they will be ordered. Eventually, we might find out that this method is such that not *all* numbers in the first half of the list are smaller than *all* numbers in the second half. E.g. in the list 7 3 2 9, number 7 would be compared with number 9, and as they are already in order, nothing would be done. Then, number 3 would be compared with number 2, and they would be switched. We could continue by testing a few solutions. The first one is that if we manage to get the halves in order first, then we could merge the two halves together by comparing the first elements and taking the smaller one first. It turns out that this will lead to a working algorithm.

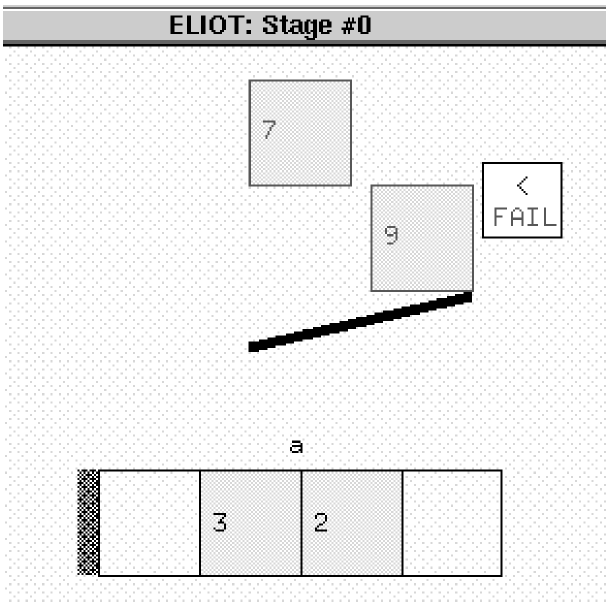

Figure 5.3: *The sorting algorithm is processing input 7 3 2 9.*

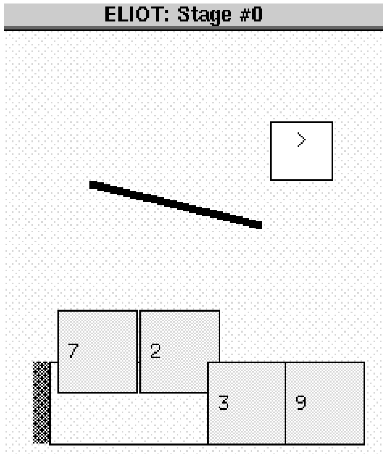

Figure 5.4: *The algorithm is proceeding; numbers 3 and 2 have been switched.*

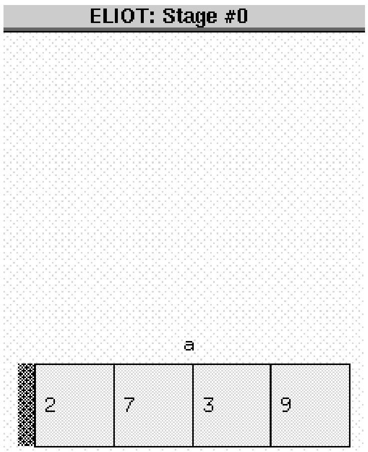

Figure 5.5: *The algorithm has finished; the result is invalid.*

Another solution is that we pick one item, and compare all other items to that one. The ones that are smaller will go to the left, and all others will go to the right. Then we sort both parts using the same method, and then append the parts together. As a result, the list gets sorted. Actually, this modification is a well-known sorting method quicksort [2], which is generally agreed to be one of the best.

The time required to develop the animations of this example was less than one hour.

Chapter 6

Evaluating Eliot in a Laboratory Course

To study the usefulness of Eliot we experimented with it in a real situation in a data structures laboratory course. This laboratory is associated with a lecture course (Data Structures) which covers basic data structures such as lists, heaps, trees, hash tables and stacks as well as some algorithms like search and sorting algorithms [2].

We experimented with Eliot in one group in Fall 1996. The group consisted of eight students, six of which used Eliot in a programming assignment on data structures. Our study is qualitative.

6.1 Arrangements

Traditionally, the motivation of the students in a data structures laboratory has been low. This might be explained by two factors: first, the assignments have had a rather theoretical character, and second, the students have been working on an individual basis, with only incidental co-operation with each other.

To get the students interested in their assignments, we organized the group into pairs. Moreover, we used assignments with practical relevance: the students had to apply the theory of the lecture course into the practice of the laboratory. In rethinking the teaching of the laboratory, we have got new insights into how to transform the course into an activating and creative learning process.

Three pairs of students were given an assignment to be done with Eliot. The other two students took individual assignments not related to Eliot.

Pair A studied two information retrieval schemes. In the first scheme records were stored in a sorted array and binary search was used for searching. In the second scheme a binary search tree was applied. The task was to compare these two approaches and their effectiveness in various situations.

Pair B compared sorting methods. The task was to find out which sorting method is appropriate in a given situation and how the algorithms could be improved. To find new ways to apply Eliot, a secondary task for pair B was to design parallel sorting animations with Eliot. The motivation for this task was that parallel animations of different algorithms are not directly supported by Eliot.

Pair C implemented and tested the LZW compression algorithm. One student implemented compression and the other decompression.

The pairs studied algorithms and data structures using Eliot. In our experiment we focused on two things: How does Eliot help a user to understand the behavior of an algorithm and what is the role of Eliot as a group member in the learning and problem-solving process.

The pairs worked mainly individually. The whole group had two weekly meetings, where the students also discussed their work and learning process. In the last session, each student presented a poster of his or her assignment.

6.2 Methods

We collected several kinds of data on the test group: question forms, video taping of students' work, learning diaries, interviews, portfolios and evaluation of students' productions.

Questionnaires. While our study is qualitative, three questionnaires were designed so that they offer a view of the students' learning processes. Most questions were open ended to give the students freedom to report what they thought was important and students were encouraged to answer them as carefully as possible. To support these open ended questions, there were also some multiple choice questions, which can be used to verify the answers.

The first questionnaire consisting of open ended questions was administered at the beginning of the course. It included questions which were meant to evaluate students' motivation and previous knowledge on algorithm animation. The second one measured the influence of Eliot on their understanding an algorithm. It also included questions about the usability and development of Eliot. The third was a multiple choice questionnaire, which covered the user interface, the usability, the quality of produced animations, and how Eliot has influenced the learning process.

Video taping. Since the number of students was small, we were able to video tape students while they were working. This was done in four stages:

- When students were getting familiar with the Eliot system. This gave us first-hand information on the usability of Eliot in situations where the user is new to the system.
- When students were working as pairs on their subject. We focused on the role of Eliot as third group member — how Eliot assisted the group work and which kind of new innovations it brought up. The students showed how they had used the system and what they had learned while using it.
- And finally, when students presented their posters. After this, they were interviewed individually to assess the learning outcomes of the course.

Learning diaries. To follow the students' learning process all the time, we asked them to keep learning diaries of their own. Every student kept record of everything he or she had learned, things they did not learn and possible reasons for not learning them.

Interviews. It is difficult to get highly informative answers with questionnaires only. So we decided to make individual interviews which were recorded for later evaluation. The interviews gave detailed information about the students' experiences and opinions. Although it is important to let the interviewee talk about what he or she wants, the interviewer may affect answers with leading questions.

Feedback discussion. When the course was over, we arranged a feedback discussion. Students presented their opinions on Eliot and the course overall.

Research team. Besides the present authors, also the teaching assistant of the laboratory course participated in belonged to the research team. In this kind of study we thought that it is necessary to evaluate also the advancement of the research project using diaries.

6.3 The Nature and Interpretation of the Collected Data

The collection of data by questionnaires and diaries was planned originally for a larger group of students and therefore the result was not deep enough to make any final conclusions possible. It is also natural that individual differences were rather large in the data. Anyway, we will briefly describe our main findings, which were also used in the preparation of the next phase of our project. Free observations and videotapes offered additional information.

The students had chosen our group simply due to the timing of the meetings. Their success in the previous courses had been in general lower than average, but there were no specially problematic cases among them. Only one of them stated that she had had problems relevant to the course in her work so that she expected the course to be very useful to her also in her future work. The others considered that the course was necessary for obtaining an academic degree, but of little use otherwise. It is apparent that more time should be allocated to strenghten the motivation of the students at the beginning of the course.

The observed students' views of the Data Structures laboratory might also be a sign of a more general attitude towards theoretical studies in Computer Science: theory is regarded as an isolated topic, unrelated to practical challenges. This phenomenon sets new requirements for any Eliot-like learning environment, to provide the students with motivating links to *see* an abstract idea working in practice. In fact, the word "theory" comes from a Greek word, meaning seeing.

The teaching assistant and other members of our research group made efforts to motivate the students to participate in this kind of group work and working in the Eliot environment. The students started their work somewhat hesitatingly, but soon it appeared that there were many bugs in the prototype version of Eliot they used. Even the more or less normal interruptions in the services of the local computer network made it rather slow and frustrating for the students to work on their tasks. It was valuable for our project that the students identified bugs in Eliot, although they themselves did not understand this as a positive outcome of their work. Instead they were worried about the slow progress of their own programming work.

In fact, the difficulties in using a prototype version of Eliot had also its positive side. Normally, the students either use software with very few bugs or test programs written by themselves. However, to understand the significance of thorough debugging, students of Computer Science should experience the role of an *end user* who utilizes random features of a new software package, not aware of its weak points. This aspect of a user-oriented debugging will be pointed out in the following phases of the project.

Towards the end of the course, when the students felt their individual tasks would soon be completed, the students, to our satisfaction, found their work very motivating. They prepared their posters eagerly and they presented their results in the final session with enthusiasm seldom seen in these groups. Furthermore, the lab course flavored the learning process with research-like questions. As a result, the students discussed their tasks and findings as experts and insiders, not mere course participants. All of this indicated clearly that the students had got a deeper understanding of their algorithms compared to the learning outcomes of an ordinary laboratory.

Chapter 7

Related Work

Algorithm animation focuses on visualizing complex computations from a semantic viewpoint, and understanding the flow of control and (the meaning of) sequences of operations is the main intellectual challenge. The key is abstraction; although many algorithm animations show manipulations of complex data structures, they try to provide an abstract view of the data to concentrate on the algorithmic aspects.

The work done in algorithm animation includes general algorithm animation systems by Brown and Herschberg [9] and Stasko [34, 36], and more instructionally oriented visualization packages integrated in computer aided education environments. For instance, Ho et.al. [11] describe an advanced visually oriented classroom computing environment where program visualizations are utilized on various courses. Also, other authors [20, 21, 31] report on computer aided instruction systems utilizing program visualization.

The Zeus animation system [8] is a representative example of state of the art algorithm animation systems. Zeus makes it possible to create presentations with several concurrent views of the execution of an algorithm. The user writes the algorithm to animate with Modula-3, and annotates the code with dispatching of events at interesting points, where animation views should be updated to reflect changes in the state of the computation. Zeus provides a set of preselected graphical objects, effects, and animation events. It is also possible to define new graphical objects and to program new animation events.

We mentioned earlier Polka [34], a general purpose algorithm animation package. The main contribution of Polka and its predecessor, TANGO [36], is the incorporation of aesthetically pleasing, smooth transitions of graphical objects in an animation view, i.e. “real” animation, with routines for constructing complex animations of combined, simultaneous transformations (e.g. movement) of several objects in a view. As in Zeus, the interaction between the algorithm part and the animation part of a visualization is carried out by generating events during the execution of an algorithm. With Polka, there is not much predefined animation material to be used right away. The user is required to define (i.e. to code) the animation views and the animation sequences corresponding to the events generated by the algorithm part. In practice, the algorithm data must often be duplicated in the animation views.

Three systems for visualizing programs, GAIGS [26], VCC [3, 4], AAPT [32] are similar to our work in some ideas: the goal is to allow users to create visualizations easily themselves. However, GAIGS and AAPT focus mainly on introductory programming and VCC on debugging. VCC generates the animation program from an input program. The user can change the appearance of visualized objects. Only arrays, records, and single variables are visualized. VCC applies a preprocessor which adds the calls of animation primitives to the code. VCC provides an additional window for the tree of subroutine calls. AAPT can also generate animations for short algorithms. The system displays the lines of code as they are executed and shows the effect of the operations on simple data structures selected by the user. AAPT utilizes a preprocessor in a similar way to VCC.

Another program visualization is the University of Washington Illustrating Compiler [12] which produces discrete visualizations of Pascal programs. It is able to visualize abstract data types by using a small expert system. The quality of the visualization for a new algorithm depends on how well the expert system knows the combination of the data structures present in the algorithm.

PASTIS [25] can visualize an algorithm without any additional information in the code. The animation is controlled by the debugger, and therefore the system works on a lower level than Eliot. The user is required to write special scripts to define animation views and to bind the views to the program running the algorithm.

London and Duisberg [19] report an interesting early object-oriented approach to program visualization. They describe an algorithm animation system implemented in Smalltalk. The system extends the well-known Model-View-Controller framework to algorithm animation. However, the system implements the traditional algorithm animation approach based on annotating program code with interesting events that invoke animation scenes. The authors stress the benefits of object-orientation in reusing animation code. Interestingly, the main advantage of object-orientation according to the authors is the clean separation of program and animation code. However, the approach chosen in [19] is to first write the program and then design the animation with no indication of automation except of reusing existing animation views when defining the animation.

As an attempt of utilizing algorithm animation techniques in program debugging and understanding, the prototype implementation of the Lens system by Mukherjea and Stasko [24] assists the user in building an animation by providing a set of predefined animation operations which can be applied to animation objects bound to the program variables. The animation system is coupled with a normal debugger to facilitate step-wise execution. Although the intentions of Lens are similar to our work, the approach adopted in Lens focuses on operations and data as separate concepts in the spirit of the procedural paradigm. Consequently, the programmer is still required to annotate program code with animation operation invocations and to declare the operations, which are rather low-level animation primitives. The effort required by the user is clearly larger than with Eliot.

Compared to the previous work on algorithm animation, our system provides a more limited but a much more easy-to-use visualization tool. Focusing on abstract data types and automation brings our work closer to architecture visualization than traditional algorithm animation techniques. Also, this makes it easier to design a library of reusable visualizations (animation scenes) using our approach than using the traditional approaches.

Chapter 8

Eliot on WWW: Jeliot

We have also implemented another algorithm animation environment based on ideas of Eliot. This system called *Jeliot* (short for Java-Eliot) has been programmed in Java and it works on the World-Wide Web. Jeliot allows any Web user to visualize his own algorithms, written in Java.

The operational framework of Jeliot is similar to that of Eliot. The user writes an algorithm in a text area of a Web page, submits it, and Jeliot returns an animation of the algorithm.

Jeliot is based on the client/server architecture. The two main parts of the system are the *server* running at the server side and the *animator* running at the client side. When the server receives the algorithm, it extracts all the variables which can be animated from the source code and constructs the animating applet from the source. This applet controls the presentation of the animation by calling the animation routines of the animator.

Before the animation can be run, the user must select the variables of the algorithm to be animated from a list provided by the server. Here the approach differs from Eliot, in which the selection is done before the generation of the animation program, whereas the selection is independent of the generation in Jeliot. Thus in Jeliot the selection serves as a dynamic parameter of the animation, and can be changed even while the animation is run.

The current version of Jeliot differs from Eliot in many respects. Jeliot is able to visualize Java's primitive types, array (one and two dimensional), queue, and stack. Jeliot highlights the active row of the algorithm during execution. A shortcoming of Jeliot is that it is not capable of storing animation parameters.

See Haajanen et al. [10] for more details.

Chapter 9

Conclusion

We have presented Eliot, a system which helps a student, a teacher, or a researcher to animate an algorithm of his own. The user of Eliot writes the code of the algorithm in the C language and defines certain animation parameters, specifying which data objects to animate and in which way. As a result, he can observe the visual representation of the algorithm in action on the screen.

The Eliot project has a problem-based educational background: How can an interactive, computer-aided learning environment support students to learn the basic idea of an algorithm? In this respect, Eliot has served as a pioneer: as a computerized tool, it helps the user to build an animation of his own algorithm, not just to see a ready-made one.

Originally, the intended customer of Eliot was a student of Computer Science. The educational goal provided the student groups, designing and implementing Eliot, with an inspiring starting point. As computer scientists, they had a shared role of a producer and a consumer. This twofold role of the students might have been one of the reasons for the successful Eliot project: the students really struggled for innovative software. They realized a working algorithm animator could be a useful tool also for themselves. An anecdote tells that the students even used Jeliot, the successor of Eliot, to debug Jeliot itself!

The problem-based approach has characterized the Eliot project all the way. The successive student groups, each responsible for a certain phase in the development of Eliot, have been briefed on quite a general level, without going deeply into the specifications. This strategy has not only proved to be fruitful for Eliot itself, but also opened horizons for new research areas.

The close connection to educationalists, motivated by the need to develop Eliot from a learner's and a teacher's perspective, has also had consequences on teaching practices and methods of Computer Science. We have designed a new teaching method for laboratory courses on data structures and algorithms. We call our method CLAP [23], since it underlines creativity, laboratory-like investigation (with Eliot as an instrument of animation), active learning, and participation in a group. We also organized a course on problem solving for fourth-year students of Computer Science.

To conclude, for us Eliot has served as an example of the multidimensionality of Computer Science. It has also shown the power of interdisciplinary research as a door to an inspiring land of new questions. As far as we understand, the new questions need multidisciplinary research, but at the same time, even strengthen our role and identity as computer scientists.

References

[1] Aarnikoivu M., Antikainen S., Huttegger N., Kyllönen H., and Wuolijoki M.: Algorithm Animator Animal (in Finnish). Report C-1994-26, University of Helsinki, Department of Computer Science, 1994.

[2] Aho, A., Hopcroft, J., and Ullman, J.: *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass., 1983.

[3] Baeza-Yates, R., Jara, L., and Quezada, G.: VCC: Automatic Animation of C Programs. In: COMPUGRAPHICS'92. Lisboa, Portugal, 1992, 389–397.

[4] Baeza-Yates, R., Quezada, G., and Valmadre, G.: Visual Debugging and Automatic Animation of C Programs. Draft, University of Chile, Department of Computer Science, 1996.

[5] Britschgi, J., Joutsenvirta, T., Järvenranta, K., and Tuovinen, A.-P.: SALSA Animator (in Finnish). Report C-1993-14, University of Helsinki, Department of Computer Science, 1993.

[6] Brown, M.: *Algorithm Animation.* ACM distinguished dissertation, MIT Press, 1988.

[7] Brown, M.: Perspectives on algorithm animation. In: Proc. CHI '88: Human Factors in Computing Systems, ACM, 1988, 33–38.

[8] Brown, M.: Zeus — a system for algorithm animation and multi-view editing. In: Proc. of the 1991 IEEE Workshop on Visual Languages. IEEE Computer Society Press, 1991, 4–9.

[9] Brown M. and Hershberger J.: Color and sound in algorithm animation. *Computer*, December, 1992, 52–63.

[10] Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., and Vanninen, P.: Animation of user algorithms on the Web.

In: *Proc. VL '97, IEEE Symposium on Visual Languages*, IEEE, 1997, 356–363.

[11] Ho C. F., Morgan C. L., and Simon I.: An advanced classroom computing environment and its applications. In: *The Papers of the 24th SIGCSE Technical Symposium on Computer Science Education* (J. E. Miller ed.), *ACM SIGCSE BULLETIN* **25**, 1, 1993, 229–231.

[12] Henry, R., Whaley, K., and Forstall, B.: The University of Washington illustrating compiler. In: Proc. ACM SIGPLAN '90 Symposium on Compiler Construction, *SIGPLAN Notices* **25**, 6, 1990, 223–233.

[13] Lahtinen, S.-P: Visual Debuggers (in Finnish). Report C-1996-60, University of Helsinki, Department of Computer Science, 1996.

[14] Lahtinen, S.-P., Lemström, K., Litola, J., and Porttikivi, A.: Salsa++ Animator (in Finnish). Report C-1994-66, University of Helsinki, Department of Computer Science, 1994.

[15] Lahtinen, S.-P., Porttikivi, A., Sutinen, E., and Tuovinen, A.-P.: Animation as a method of learning algorithms. In: *Uudet menetelmät ja mahdollisuudet matemaattis-luonnontieteellisten aineiden oppimisessa* (ed. J. Enkenberg and K. Sormunen). University of Joensuu, Faculty of Education, 1994, 65–70.

[16] Lahtinen, S.-P., Sutinen, E., and Tarhio, J.: Automated Animation of Algorithms with Eliot. To appear in: *Journal of Visual Languages and Computing*, 1997.

[17] Lahtinen, S.-P., Sutinen, E., Tarhio, J., and Tuovinen, A.-P.: Object-oriented visualization of program logic. In: *Proceedings of Technology of Object-Oriented Languages & Systems (TOOLS 23)*, ISE Inc., Santa Barbara, California, 1997, 93–107.

[18] Lamminjoki, T., Lilja, S., and Ollikainen, V.: Eliot (in Finnish). Report. University of Helsinki, Department of Computer Science, 1995.

[19] London, R. L., Duisberg, R. A.: Animating programs using Smalltalk. *Computer* **18**, 8, 61–71 (1985).

[20] Mayerhofer, M. A., Pereira de Lucena, C. J.: Design of an algorithm simulation and animation Environment (ASA). *ACM SIGCSE BULLETIN* **24**, 2, 7–14 (1992).

[21] Mayerhofer, M. A., Pereira de Lucena, C. J., and Cavalcanti M. R.: Experience using the ASA Algorithm teaching system. *ACM SIGCSE BULLETIN* **26**, 4, 45–50, (1994).

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

[23] Meisalo, V., Sutinen, E., and Tarhio, J.: CLAP: teaching data structures in a creative way. In: *Proc. ITiCSE '97, Integrating Technology into Computer Science Education*, Uppsala University, 1997, 117–119.

[24] Mukherjea, S., Stasko, J.: Applying algorithm animation techniques for program tracing, debugging, and understanding. In: Proc. 15th International Conference on Software Engineering, IEEE, 1993, 456–465.

[25] Müller, H., et al.: The program animation system PASTIS. *The Journal of Visualization and Computer Animation* **2**, 1, 26–33 (1991).

[26] Naps, T., Swander, B.: An object-oriented approach to algorithm visualization – Easy, extensible, and dynamic. In: *The Papers of the 25th SIGCSE Technical Symposium on Computer Science Education* (D. Joyce ed.), *ACM SIGCSE BULLETIN* **26**, 1, 46–50 (1994).

[27] Negroponte, N.: Beyond the desktop metaphor. In: Research Directions in Computer Science: An MIT Perspective (A. Meyer et al., eds.) The MIT Press, Cambridge, Massachusetts, 1991, 183–190.

[28] Rautama, E., Sutinen, E., Tarhio, J.: Excel as an algorithm animation environment. Proc. ITiCSE '97, Integrating Technology into Computer Science Education, Uppsala University, 1997.

[29] Repenning, A., Sumner, T.: Agentsheets: a medium for creating domain-oriented visual languages. *Computer* **28**, 17–25 (1995).

[30] Roman, G., Cox, K.: A taxonomy of program visualization systems. *Computer*, December, 11–24 (1993).

[31] Ross, R. J.: Experience with the DYNAMOD program animator. In: *The Papers of the 22nd SIGCSE Technical Symposium on Computer Science Education* (B. B. Owens ed.), *ACM SIGCSE BULLETIN* **23**, 1, 35–42 (1991).

[32] Sanders, I., Harshila, G.: AAPT: Algorithm animator and programming toolbox. *ACM SIGCSE BULLETIN* **23**, 4, 41–47 (1991).

[33] Smart, J.: Reference manual for wxWindows 1.60: a portable C++ GUI toolkit. University of Edinburgh, Artificial Intelligence Applications Institute, 1994.

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

[35] Stasko, J.: TANGO: A framework and system for algorithm animation. Ph.D. Thesis, Technical Report CS-89-30. Brown University, Department of Computer Science, 1989.

[36] Stasko, J.: Tango: A framework and system for algorithm animation. *Computer*, September, 27–39 (1990).

[37] Stasko, J., Badre, A., and Lewis, C.: Do algorithm animation assist learning? An empirical study and analysis. In: *Proc. of INTERCHI '93* (ed. S. Ashlund et al.), IOS Press, 61–66 (1993).

[38] Sutinen, E. and Tarhio, J.: String matching animator SALSA. In: *Proc. Third Symposium on Programming Languages and Software Tools* (ed. M. Tombak), University of Tartu, 120–129 (1993).

Appendix A

The Limitations of Eliot-C

Eliot-C has the following limitations:

- Eliot-C programs cannot consist of more than one module.
- Animated variables cannot be manipulated with operations like ++ and `--` that increase or decrease the value of a variable with one.
- Pointer variables cannot be animated.

Appendix B

Bibliography on Eliot

1. Aarnikoivu, M., Antikainen, S., Huttegger, N., Kyllönen, H., and Wuolijoki, M.: Algorithm Animator Animal (in Finnish). Report C-1994-26, University of Helsinki, Department of Computer Science, 1994.

2. Bergius, T., Haajanen, J., Juslin, J., Pesonius, M., Pettilä, M., and Vanninen, P.: Java-Eliot (in Finnish). Report, University of Helsinki, Department of Computer Science, 1996.

3. Britschgi, J., Joutsenvirta, T., Järvenranta, K., and Tuovinen, A.-P.: SALSA Animator (in Finnish). Report C-1993-14, University of Helsinki, Department of Computer Science, 1993.

4. Dybdahl, A., Eloranta, J., Hietanen, K., Makkonen, P., and Markkanen, J.: Eliot/Windows - algoritmien visualisointijärjestelmä Windows-ympäristöön. Report, University of Helsinki, Department of Computer Science, 1996.

5. Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Teräsvirta, T., and Vanninen, P.: Animation of user algorithms on the Web. In: *Proc. VL '97, IEEE Symposium on Visual Languages*, IEEE, 1997, 356–363.

6. Huhtiniemi, P., Huttunen, S., Kivi-Koskinen, M., Liukkonen, P., Peltola, J., Reynders, J., and Vähä-Erkkilä, A.: Jeliot - interaktiivinen algoritmiteatteri (in Finnish). Report, University of Helsinki, Department of Computer Science, 1997.

7. ahtinen, S.-P: Visual Debuggers (in Finnish). Report C-1996-60, University of Helsinki, Department of Computer Science, 1996.

8. Lahtinen, S.-P., Lamminjoki, T., Ollikainen, V., Sutinen, E., Tarhio, J., and Tuovinen, A.-P: Animation as a method of learning to write algorithms. Research Report 162, University of Helsinki, Department of Teacher Education, 1995.

9. Lahtinen, S.-P., Lamminjoki, T., Sutinen, E., Tuovinen, A.-P, and Tarhio, J.: 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.

10. Lahtinen, S.-P., Lemström, K., Litola, J., and Porttikivi, A.: Salsa++ Animator (in Finnish). Report C-1994-66, University of Helsinki, Department of Computer Science, 1994.

11. Lahtinen, S.-P., Porttikivi, A., Sutinen, E., and Tuovinen, A.-P.: Animation as a method of learning algorithms. In: *Uudet menetelmät ja mahdollisuudet matemaattis-luonnontieteellisten aineiden oppimisessa* (ed. J. Enkenberg and K. Sormunen). University of Joensuu, Faculty of Education, 1994, 65–70.

12. Lahtinen, S.-P., Sutinen, E., and Tarhio, J.: Automated Animation of Algorithms with Eliot. To appear in: *Journal of Visual Languages and Computing*, 1997.

13. Lahtinen, S.-P., Sutinen, E., Tarhio, J., and Tuovinen, A.-P.: Learning and problem solving assisted with animated objects. In: *Abstracts of European Conference on Educational Research (ECER 95)*, University of Bath, p. 153, 1995.

14. Lahtinen, S.-P., Sutinen, E., Tarhio, J., and Tuovinen, A.-P.: Object-oriented visualization of program logic. In: *Proceedings of Technology of Object-Oriented Languages & Systems (TOOLS 23)*, ISE Inc., Santa Barbara, California, 1997, 93–107.

15. Lamminjoki, T., Lilja, S., and Ollikainen, V.: Eliot (in Finnish). Report, University of Helsinki, Department of Computer Science, 1995.

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

17. Meisalo, V., Sutinen, E., and Tarhio, J.: CLAP: teaching data structures in a creative way. In: *Proc. ITiCSE '97, Integrating Technology into Computer Science Education*, Uppsala University, 1997, 117–119.

18. Möttönen, T., Pekkala, S., Pursiainen, K., Rautama, E., Sahlberg, V.-M., and Teräsvirta, T.: Eliot, pcEliot — algoritmien visualisointijärjestelmä. Report, University of Helsinki, Department of Computer Science, 1996.

19. Sutinen, E. and Tarhio, J.: String matching animator SALSA. In: *Proc. Third Symposium on Programming Languages and Software Tools* (ed. M. Tombak), University of Tartu, 120–129 (1993).