Matrix – Concept Animation and Algorithm Simulation System

Ari Korhonen
Department of Computer Science and Engineering
Helsinki University of Technology
Finland
archie@cs.hut.fi

Lauri Malmi
Department of Computer Science and Engineering
Helsinki University of Technology
Finland
lma@cs.hut.fi

ABSTRACT

Data structures and algorithms include abstract concepts and processes, which people often find difficult to understand. Examples of these are complex data types and procedural encoding of algorithms. Software visualization can significantly help in solving the problem.

In this paper we describe the platform independent Matrix system which combines algorithm animation with algorithm simulation, where the user interacts directly with data structures through a graphical user interface. The simulation process created by the user can be stored and played back in terms of algorithm animation. In addition, existing library routines can be used for creating illustrations of advanced abstract data types, or for animating and simulating user's own algorithms. Moreover, Matrix provides an extensive set of visual concepts for algorithm animation. These concepts include visualizations for primitive types, arrays, lists, trees, and graphs. This set can be extended further by using arbitrarily nested visualizations

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids—*visual debugging*; E.1 [**DATA STRUCTURES**]: Arrays, Linked Lists, Trees, Graphs; K.3.2 [**Computer and Information Science Education**]: Computer science education—*data structures and algorithms*

General Terms

Algorithms, Human Factors

Keywords

algorithm simulation, algorithm animation, software visualization

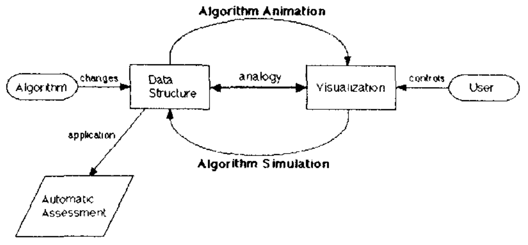

Figure 1: Algorithm animation and simulation

1. INTRODUCTION

Let us assume we have a *visualization* for some data structure and an algorithm that manipulates it. Obviously, the visualization could be used for animation purposes. During the execution of the algorithm the state of the data structure changes, and these states can be visualized for the user, one after another to animate the algorithm. Animation is thus a sequence of visual snapshots of the data structure.[1] In addition, the user may have some kind of control of the process, *i.e.*, he can interact with the system in order to stop and continue the animation. Moreover, these animation steps could be memorized in order to give the user the control of traversing the animation sequence back and forth. We call such a process simply *algorithm animation*.

On the other hand, we can allow the user to perform operations of his own. Thus, instead of letting the algorithm to execute the instructions and manipulate the data structure, we can allow the user to perform the manipulation process. If the user processes the data structure as an algorithm would do, we say that the user simulates the algorithm. We call such a process *user controlled simulation* of an algorithm or simply *algorithm simulation* (Figure 1).

Algorithm animation is a widely used method and a large number of systems and different approaches to the subject have been implemented [1–12]. However, the level of interac-

[1]Some authors define this *discrete animation* opposite to *continuous animation* in which the movement from one display to another is decorated by smooth movements of objects on the screen.

tion is typically limited to controlling the *animation display* and the algorithms involved, thus having only the influence on the input data, execution of the algorithm, and possibly the way the output is represented. In addition, some systems, *e.g.* [3, 4], allow the user to write algorithms of his own which can be animated automatically. It is, however, possible to include algorithm simulation into the method and to allow direct interaction with the data structures involved. For example, when studying balanced trees the user could perform the animations of binary search tree insertions each followed by the simulation of rebalancing operations. Such a combination of working methods is a valuable aid in learning algorithms, as well as in designing and experimenting with new algorithms.

However, in order to keep the representation coherent, this approach requires concepts which have knowledge of their internal structure and how this structure can be manipulated[2]. This has been implemented in Matrix [7, 9]; it demonstrates a way to implement the idea of combining algorithm animation and algorithm simulation.

1.1 Related work

There are many tools which are dedicated for the easy creation of algorithm animations, but which do not include the simulation functionality, for example, Amethyst [13], Jeliot [4] and UWPI [5]. These tools analyze the given Java or Pascal code to produce the visual representation. The JDSL Visualizer [3] tool uses another approach for generating visualizations; user's code should conform to special interfaces, a method also available in Matrix. Matrix, Jeliot, JDSL Visualizer and UWPI provide visualizations for high-level *conceptual displays* opposite to the *concrete data structures* which appear in the implemented code. It is also worth noting that both Matrix and JDSL Visualizer have the ultimate goal of using the system to provide feedback to students while working with assignments. We have gathered good results while using a similar system in our course on Data Structures and Algorithms in order to automatically assess students exercises [8].

Next we list some differences between Matrix and other systems. First, we do not know any other system which is cabable of algorithm simulation in the sense of direct graphical manipulation of visualized data structures. Astrachan *et al.* discuss simulation exercises while introducing the Lambada system [1, 2]. However, their context is quite different, because the students simulate models of practical applications, partly coded by themselves and the system is used only for illustrating the use of primitive data structures without much interaction with the user.

Second, our approach is also quite different from the works of Stasko, Kraemer and Mukherjea [12, 14, 15, 16] where the general method is to generate animations by annotating the actual algorithm code. In Jeliot [4] the annotation method has been automated but the starting point is essentially the given user code. This enables easy creation of a visualization for arbitrary code, but still lacks the simulation functionality provided by Matrix.

Third, Matrix is designed for students working on a higher level of abstraction than, for example, JDSL Visualizer. In other words, JDSL Visualizer is designed for a *programming course* (CS2) to provide interactive debugging tools for educational purposes while Matrix is intended for a *data structures and algorithms course* (CS7) to illustrate and grasp the logic and concepts of data structures and algorithms. Of course, both kinds of tools are fit for use.

Fourth, Matrix is implemented in Java which gives us more flexibility in terms of platform independency, compared to older systems such as Amethyst and UWPI.

2. MATRIX

In this section we present the basic ideas and functionality of Matrix. The system has been designed for two purposes. First, the user can create and manipulate data structures by using the commands available in the user interface, *i.e.* menu commands and context sensitive drag-and-drop operations. Second, the user can implement his own code (Java classes), load it to the system and let Matrix visualize the data structures. The whole simulation functionality is available in this case, too. We describe these two operation modes through some examples below. Thereafter we discuss the visualization process in more detail in Section 2.1.

From the user point of view, Matrix operates on a number of *visual concepts* which include arrays, linked lists, binary trees, common trees, and graphs, as depicted in Figure 2. Abstract data types can be visualized using different visual concepts. For example, a heap can be represented as a binary tree or as an array. Moreover, both concepts can be visible at the same time and simulation operations performed on either of them are visualized simultaneously in the other one.

Context sensitive drag-and-drop operations give versatile opportunities to work on different abstraction levels. Consider a student learning basic search trees. He can exercise on a Binary search tree (BST) by dragging new keys into the correct leaf positions in the tree thus simulating the insertion algorithm. After mastering this, he can switch to work on an ADT level and drag the keys into the title bar of the representation. Then the keys are inserted into the tree by using the pre-implemented BST insertion routine. Now, he faces the question of balancing the tree. First he can exercise on rotations on the basic level by dragging edges into new positions and let the system redraw the tree. When he masters this he can use the available rotation commands in menus directly. Finally, he can choose to work on AVL trees, create a new tree from the menu, and use the AVL-tree insertion routines. In addition, he can experiment on the behaviour of AVL trees by creating an array of keys and dragging the whole array into the title bar. Then the system inserts the keys in the array one by one into the tree using the implemented insertion routine. Moreover, the result is not a static tree, but a sequence of states of the tree between the insertions, which can be stepped back and forth to examine the working of the algorithm more closely.

[2]Actually it is a fairly complicated issue to interpret how certain user operations on the visual representation should be interpreted on the level of the implemented data structure. This is discussed more thoroughly in Section 2.2.

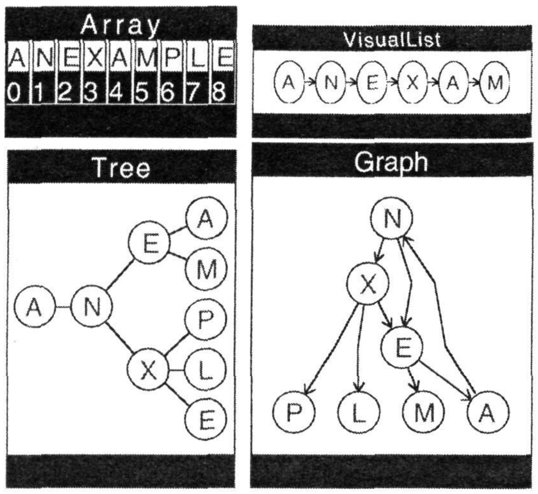

Figure 2: Fundamental data types, such as arrays, lists, trees, and graphs are important reusable abstractions regularly used in computer science. The above are printed from the Matrix system.

An important feature providing new opportunities is that visual concepts can be nested to create complex structures, for example, adjacency lists or B-trees. Matrix does not restrict the level of nesting. Moreover, context sensitive operations also apply to such structures, although the interpretation of an operation is not always simple.

As an example of the visualization of user's own code consider a case in which the student wants to implement a visualization of an AVL tree. He has three options for accomplishing this task. First, he can built the code so that it conforms to the visual concept interface Tree. Second, he can reuse the Matrix library component BinTree. Third, there exists an implementation of *binary search tree* (ADT) as a Matrix library component; so the easiest way to implement the AVL tree is to inherit the existing binary search tree class and to override the insert and delete methods to support the balancing of the tree.

The bytecode of the compiled data structure can be dynamically loaded into Matrix. The system automatically creates a new instance of the class by invoking its default constructor and opens a new frame containing the visualization of the data structure. After that, the user can, *e.g.*, insert data into the structure as described above and see the changes in the visualization immediately. This procedure allows *visual testing* of the algorithm. Instead of writing and compiling test classes with test data generation the user can simply drag-and-drop the test keys or key sets into the structure and verify the results in a visualized form. Since the visualization sequence (algorithm animation) is based on the actual states of the data structure, the user can backtrack the algorithm to observe the behaviour more closely.

As an example, consider the visualization in Figure 3. The user has loaded his own AVL-tree code into Matrix and selected a set of keys from the menu to be visualized together

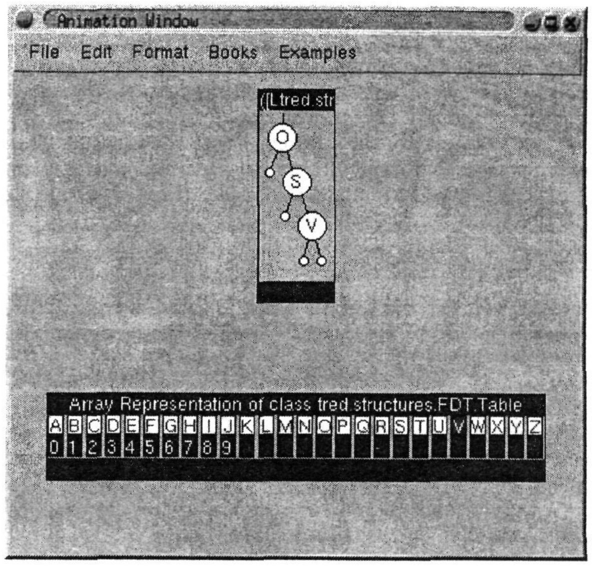
Animation Window
File Edit Format Books Examples
(fi.tred.str
O
S
V
Array Representation of class tred.structures.FDT.Table
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9

Figure 3: Concept Visualization of an Erroneous User Made AVL tree.

with the AVL tree. The search tree was initially empty. The figure shows the structure after the user has drag-and-dropped three keys into the target structure. Obviously the user made structure seems to be broken: the structure behaves like an ordinary binary search tree and no rotations are executed even though the structure is imbalanced.

2.1 Visualization method

In this section we consider the visualization principles more closely. The creation of a visualization is divided up into three phases that are *Adaption*, *Validation* and *Laying out* the representation. In addition, the GUI provides a way to influence on the functionality of these phases.

2.1.1 Adapter

Matrix provides several methods for creating visualizations. The degree of hand coding needed to produce a visualization depends on the method used. First, it is possible to produce the visualization manually in terms of user controlled simulation, as explained above. No actual coding of programs is needed. We discuss this topic further in Section 2.2. Second, it is possible to reuse the *Matrix library components*. These include several implementations for all supported concepts, called *Fundamental Data Types, FDTs*. Some authors call this the probing technique [10, 11]. Third, each FDT has the corresponding visual concept interface that could be implemented directly. Thus, it is possible to implement a data structure which directly conforms to one or several visual concept interfaces. Thereafter the visualization of the structure and the basic algorithm animation and simulation features are automatically available in Matrix. It should be noted, however, that this method requires extra work to gain the possibility to reverse the temporal direction of the animation.

The system tries to treat each of the visualized data structures similarly through the visual concept interface. Sometimes this may lead to difficulties for the user in implementing the required interface efficiently. The system therefore provides adapters for some of the concepts in order to make it easier to adapt the data structure into the concept. This adaption is transparent for the programmer, thus forcing him to choose from a range of visual concept interfaces in order to make his structure visible.

2.1.2 Validation

For each concept there exists a single visual concept interface that is used to prepare the visualization. For example, every data structure D that should have the option to be visualized as a tree should conform to the corresponding visual concept interface Tree. The next challenge is to ensure that the procedure which displays trees actually takes one as its input. We should therefore validate D and remove all cycles. Thus, even if the input is a graph, we can still accept the original structure and visualize it as a tree. In practice this means that the visualization only covers the connected nodes of the graph starting from a given root node in the depth first search order. All back, forward and cross edges are visualized as connections to special terminal nodes such that their layout differs from the ordinary nodes. Moreover, the user can modify the behaviour of the validation process in order to prevent the visualization of such edges.

2.1.3 Layout

The final phase of creating the representation produces the layout for a concept. The layout procedure may assume that the input it receives really follows the conceptual principles, as we have discussed above. Thus, the input for the procedure is a set of nodes and connections between them which obey the rules of the concept. The output is a hierarchy of these nodes that should appeal to the intuition what the user expects to see while working with the corresponding metaphor.

Obviously there is not only one single layout that is adequate for, let us say, all trees. At the moment, the system provides two layouts for trees that differ slightly from each other. Similarly, several layouts for any concept can be implemented into the framework. The GUI provides a way to change the layout on run-time.

2.2 User Interface Objects

There are four different kinds of interactive components in the GUI. The first three components behave similarly compared with each other. The fourth component, the binary relation between two other components, has slightly different behaviour.

The principles how the interactive components interact with each other are discussed below. The interpretation of the semantic meaning of various user operations is, however, often a difficult or even an ambiguous question when considered in general. We omit it here for brevity. For the future development, however, the following guidelines are provided.

2.2.1 Hierarchy, Node, and Key

A *Hierarchy* refers to a set of *Nodes*. A node could be identified by a unique *Key*. A key could be a primitive or some more complex structure. If the key is a primitive, it has no internal structure that is a subject of further examination. If the key is some more complex structure, it is treated recursively as a hierarchy.

All these three entities can be moved around the display. The simulation consists of drag-and-drop operations which can be carried out by picking up the *source* entity and moving it onto the top of the *target* entity. After the drop action, the framework, by default, invokes the corresponding *target.insert(source)* operation for the underlying data structure. Several other methods could also be invoked. For example, the user could invoke *delete* for a single key by using the pop-up-menu or by drag-and-dropping the key into a special *trash* structure.

As an example, consider Figure 2. The letter symbols (keys) in Array (hierarchy) at positions (nodes) $0-5$ could have been drag-and-dropped one at the time into an initially empty list (hierarchy), thus producing the structure VisualList. Moreover, one could continue the simulation by inserting the rest of the keys into the list by drag-and-dropping the keys at positions $6-8$ into the target list title bar.

2.2.2 The connection entity

The framework allows reconnections of binary relations. The target node of a pointer could be changed to another node. For example, the successor of the VisualList node N in Figure 2, could be set to point into the node labeled X. Several pointer updates can be performed simultaneously in order to avoid situations in which part of the underlying structure becomes obsolete. Thus, simple target node updates do not change the underlying structure until the user explicitely finalizes the update operation. In our example, the successor of M can be set to point into the otherwise disappearing node E, thus producing the list A, N, X, A, M, E after the final update.

2.2.3 Frames

A collection of hierarchies could be combined to a single frame. Within a frame it is possible to move entities from one hierarchy to another as described above. The GUI also provides additional functionality for manipulating hierarchies such as opening a (sub)hierarchy on other frame, disposing a hierarchy, minimizing or hiding an entity or some part of it, (re)naming or resizing an entity, changing the orientation of a hierarchy (flip, rotate), and changing the representation (concept or layout) for a hierarchy. These operations influence on the animation process by making adjustments in the modules of the visualization assembly-line in order to achieve the desired outcome.

There is also a built-in mechanism for adding one's own functionality into the representation. For example, it is possible to include *rotate*-methods into a binary tree implementation and allow these operations to appear in the pop-up-menu. Thereafter the operations are included among other operations and can be accessed through the GUI.

2.2.4 *Animator*

In addition to frames, there is a special window that contains the operational interface of the system: the *animator* window. All build-in probes are implemented in such a way that every update operation, *i.e.*, any method invocation that changes the structure, is stored into an additional queue and could be revoked later on. This technique allows the animator to rewind the queue of operations and replay the sequence again. Naturally, each update operation could also be revoked and invoked again one at a time.

In order to gain the possibility to fully animate user made data structures and algorithms, the easiest way is to reuse the ready-made probes. If, however, this is not adequate, it is obviously possible to add new animated components [7] into the system, but this procedure is much more difficult, in general, and is not discussed here any further.

3. CONCLUSION

In this paper, we have presented the Matrix system for visualizing data structures and for simulating and animating algorithms. Matrix provides new flexibility for creating visualizations. It combines ordinary code-based algorithm animation with algorithm simulation, in which the user directly manipulates data structures by using the graphical user interface facilities without writing any code.

Matrix has been designed to support easy building of concept visualizations and concept animations similar to textbook examples. Our experience based on the TRAKLA-system [8] supports our view that this approach, especially algorithm simulation exercises, improves the learning curve of students. The students can achieve a clear understanding of data structures and algorithms without pondering sometimes blurring implementation details. It is interesting to note that our observations resemble the findings of Hundhausen [6]. He performed a study how different algorithm visualization strategies effect on learning. He showed in his survey that students who build animations of their own learned the algorithms better than students learning on animations created by experts. The Matrix algorithm simulation is an easy method for creating student-made animations, where the focus of working is on the conceptual level and not on the implementation or representation level details.

However, Matrix supports also automatic generation of algorithm animation on courses that concentrate on implementation issues. Users can write code which reuses such fundamental data types that can visualize themselves automatically. Alternatively, their own data structure classes can implement one or more interfaces which provide the automatic visualization for the corresponding visual concepts. Such code can be dynamically imported into Matrix and used in simulation operations in the same way the standard system components operates. Thus, the user can implement a data structure of his own and survey its behaviour using the simulation tools. This feature can be used, for example, in *visual debugging* of an algorithm in which the user can debug his own code by examining the corresponding visualization. The visualization may reveal errors in the code which could otherwise be difficult to observe for a novice programmer as we have demonstrated in Figure 3.

In addition to this functionality, there are certain features in Matrix which we consider important because they clarify the relations between the concepts of data structures and algorithms. First, one data structure can have many visualizations. For example, a binary heap implementation can be represented both as a tree and as an array. Second, one visualization can be used for displaying many different data structures. The design of the system is based on reusable visual concepts that can be mapped to any proper data structure to be visualized. Third, the visual concepts can be nested to visualize arbitrarily complex structures.

3.1 Future Directions

Matrix currently includes the properties described above. However, many new ideas and features are still under development. In the future releases at least some of the following aspects should be developed further.

Currently the animations consist of a sequence of data structure snapshots, in which appropriate highlighting can be applied to emphasize the transition from one state to another. This method could be improved by allowing smooth transitions (continuous animation) between the states.

The key idea in the TRAKLA system is the automatic assessment of simulation exercises. This feature has been included in the recent release of Matrix, but there is still work to do to bring this feature generally available with all kinds of algorithms. However, TRAKLA assesses only some predefined states of the data structure whereas Matrix provides the possibilities to assess the complete sequence of states generated by the user. Assessing is a complicated issue in many ways, but it provides excellent tool for giving guidance and feedback about the user-generated simulation sequence and possible errors in it.

Finally, as the field of algorithm visualization seems to be open ended, the system could be developed to handle visualization of concurrent algorithms, provide synchronized animations of multiple algorithms simultaneously, and use 3D-animations and sound to enhance the comprehensibility of the animations.

4. REFERENCES

[1] O. Astrachan and S. Rodger. Animation, visualization, and interaction in cs1 assignments. In *The proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 317-321, Atlanta, GA, USA, 1998. ACM.

[2] O. Astrachan, T. Selby, and J. Unger. An object-oriented, apprenticeship approach to data structures using simulation. In *Proceedings of Frontiers in Education*, pages 130-134, 1996.

[3] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. In *Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education*, pages 261-265, New Orleans, LA, USA, 1999. ACM.

[4] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of user

algorithms on the web. In *Proceedings of Symposium on Visual Languages*, pages 360–367. IEEE, 1997.

[5] R. Henry, K. Whaley, and B. Forstall. The university of washington illustrating compiler. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.

[6] C. D. Hundhausen and S. A. Douglas. Using visualizations to learn algorithms: Should students construct their own, or view an expert's? In *IEEE Symposium on Visual Languages*, pages 21–28, Los Alamitos, CA, September 2000. IEEE Computer Society Press.

[7] A. Korhonen. *Algorithm Animation and Simulation.* Licenciate's thesis, Helsinki University of Technology, 2000.

[8] A. Korhonen and L. Malmi. Algorithm simulation with automatic assessment. In *Proceedings of The 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pages 160–163, Helsinki, Finland, July 2000.

[9] A. Korhonen, J. Nikander, R. Saikkonen, and P. Tenhunen. Matrix - algorithm simulation and animation tool. http://www.cs.hut.fi/Research/Matrix/, November 2001.

[10] J. Korsh and R. Sangwan. Animating programs and students in the laboratory. In *Proceedings of Frontiers in Education*, pages 1139–1144, 1998.

[11] P. LaFollette, J. Korsh, and R. Sangwan. A visual interface for effortless animation of c/c++ programs. *Journal of Visual Languages and Computing*, 11(1):27–48, 2000.

[12] S. Mukherjea and J. Stasko. Toward visual debugging: Integrating algorithm animation capabilities within a source level debugger. *Transactions on Computer-Human Interaction*, 1(3):215–244, 1994.

[13] B. Myers, R. Chandhok, and A. Sareen. Automatic data visualization for novice pascal programmers. In *IEEE Workshop on Visual Languages*, pages 192–198. IEEE, 1988.

[14] J. Stasko. Tango: a framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, 1990.

[15] J. Stasko. Using student-built algorithm animations as learning aids. In *The Proceedings of the 28th SIGCSE Technical Symposium on Computer Science Education*, pages 25–29, San Jose, CA, USA, 1997. ACM.

[16] J. Stasko and E. Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18(2):258–264, 1993.