Visual Editing of Animated Algorithms: 
the Leonardo Web Builder

Vincenzo Bonifaci¹,³ Camil Demetrescu¹ Irene Finocchi² Luigi Laura¹

¹Dipartimento di Informatica e Sistemistica 
Università di Roma “La Sapienza” 
Via Salaria 113, 00198 Roma, Italy. 
{bonifaci, demetres, laura}@dis.uniroma1.it

²Dipartimento di Informatica 
Università di Roma “La Sapienza” 
Via Salaria 113, 00198 Roma, Italy. 
finocchi@di.uniroma1.it

³Department of Mathematics and Computer Science 
Technical University of Eindhoven 
PO Box 513, 5600 MB Eindhoven, The Netherlands. 
v.bonifaci@tue.nl

ABSTRACT

Leonardo Web is a collection of tools to animate algorithms. Animations can be generated with a visual editor or directly as a trace of an algorithm’s execution. They can be visualized via a small Java player, available as an applet or as a standalone application; the player supports bidirectional continuous and step-by-step execution. Furthermore the system allows to export the animations in several formats, including Macromedia Flash, Microsoft PowerPoint and animated GIF.

In this paper we discuss the design issues of one of the component of the visual editor of Leonardo Web, called the Builder, that can be used to design an animation from scratch as well as to refine batch-generated ones.

Keywords

algorithms animation, visual interfaces

1. INTRODUCTION

Algorithm animations are an effective tool for teaching and for learning algorithms and data structures; on the other side it is usually very expensive [7, 11] to create computer based learning material. If one wants to develop algorithm animation the range of tools varies significantly; on one side we find some commercial (or not) general-purpose tool, such as Microsoft PowerPoint, Apple Keynote or OpenOffice Impress, that are versatile but even animating a simple algorithm, like a sorting one, can be time consuming; furthermore if one wants to animate the same algorithm with a different input the animation usually is to be done again from scratch; on the other side there are specialized tools, such as Macromedia Flash, that are powerful enough but may have a steep learning curve. A completely different approach is to write computer programs to draw animations, but this could be complex and, again, time consuming. Furthermore, if one wants its animations to be platform independent, like an animated GIF or a MPEG movie, this could narrow the choice. In this context, a quite natural solution involves the use of Web-based technologies [2, 6, 12, 17, 19].

The Leonardo Web system, presented in [5], is based on a scripting language which is used to represent animations. The script describes the animation in a differential way; for every frame, only the differences with the previous one are given. This is done both in order to avoid having huge scripts and also to make it easier to create a script from the trace of a running program.

The visual editor makes it possible to load these scripts, modify the animation, and save it again in scripted format. In principle, it is possible to use the system without ever resorting to the visual editor; a script can be generated by means of a trace from a running program, like in the animated ball example, or even written with a simple text editor. However, it is conceivable that many times the user would like to modify the animation in an intuitive manner without interacting with the scripting code directly. For this reason we provided a visual editor in which the scripting code is completely hidden and what the user sees is only the animation itself. The editor can be used to modify the animation, or even to create one from scratch, and is able to save it again in scripting language format. Thus it is possible to work at the animation level only, without ever seeing a single line of the script. After the animation is saved, the corresponding script, generated by the editor, is ready for use with the Java player.

The goals of the visual editor are thus the complete support of the scripting language and the ease of use. A major issue in the editor is the conversion from the differential representation of the animation in the scripting format to the frame-by-frame representation used in the editor itself, and viceversa. This is necessary because we cannot show the an-

AVI '06, May 23–26, 2006, Venezia, Italy.
imation to the user as a sequence of inter-frame differences; for the editor to be intuitive, we have to present the user with the cumulative content of every frame. Notice that this issue is also partially present in the Java player, but in that case one only needs to convert the differential representation into the cumulative one and not the other way around.

This paper is organized as follows: in the next section we discuss related work, while Section 3 presents an overview of the Leonardo Web System; the details about the visual editor are addressed in Section 4, and concluding remarks can be found in Section 5.

2. RELATED WORK

A common approach to animate algorithm over the Web consists of writing Java applets that display the desired visualizations (see, e.g., [1, 14, 20, 22]). Few Web-based systems can be also found in the literature with the aim of automating (or at least simplifying) the process of creating visualizations. JEllo3 [16] is well-suited at illustrating basic programming concepts: it parses the Java code and allows the user to choose a subset of variables to visualize according to built-in graphical interpretations. JDSL [3] is a Java library of data structures that features a visualizer for animating operations on abstract data types such as AVL trees, heaps, and red-black trees [8]: the visualization of these data types is embedded into the library. VEGA [13] is a C++ client/server visualization environment especially targeted to portraying geometric algorithms. WAVE [10] hinges upon a publication-driven approach: algorithms run on a developer’s remote server and their data structures are published on blackboards held by the clients, that attach visualization handlers to the public data structures.

Note that a specific programming language is at the base of most of Web-based algorithm animation tools; these tools receive as input the implementation of an algorithm, an input instance, and a specification of the visualization; their output is a visual trace of the algorithm execution, i.e., a sequence of moving images that describe the evolution of the data at-run-time. Some efforts towards language-independent and more general solutions can be found in systems such as JSamba [21], which exploits a scripting-based approach, and JAWAA [18], which features an editor to generate animation traces.

3. OVERVIEW OF LEONARDO WEB

In this section we briefly recall some of the key aspects of Leonardo Web. The interested reader can find architectural details and implementation issues in [5], and up to date information about the Leonardo Web tools can be found at the Web site http://www.dis.uniroma1.it/~leoweb, where Leonardo Web is available under the GPL licence. Leonardo Web animations are stored as plain text files written in a simple scripting language, which specifies the incremental changes between key frames. The incremental nature of the scripting language makes presentation scripts small and compact, and thus amenable, together with the 100 KB applet version of the Player, to a quick access even on slow network connections.

The Leonardo Web visual editor, called The Builder, is detailed in the next section.

The Player. The Leonardo Web Player is as available as a light Java applet or stand alone application; it has been designed to be an easy-to-use presentation viewer with a clean and simple graphic user interface, shown in Figure 1, resembling to a standard VCR control tool. It is able to interpret text files created by the Builder or by the Library. The user can start, stop, rewind, and play the presentation both forward and backward. Playing is supported either in a step-by-step fashion, or continuously. Animated transitions of graphical objects, including movements and color changes, are smoothly rendered by the system by generating sequences of interpolating interframes. To support effective on-line deployment even on slow network connections, the Player is fully multi-threaded, allowing it to start playing a presentation even if it has not been completely downloaded from the remote peer.

The Library. Imagine that one wants to animate the same algorithm against several input instances; in this scenario, it might be easier to write a program whose execution produces the desired animation script, rather than having to specify it directly in the Builder. Still, a script generated in this way can be later refined and completed using the Builder. To support this scenario Leonardo Web provides a Java library (JLeoScript) that provides primitives for creating presentation scripts. Available primitives supported by the class JLeoScript include adding graphical objects to the scene, deleting existing graphical objects, moving, grouping and resizing objects, and changing color.

Furthermore, there is a repository of animated illustrations for a textbook on algorithms and data structures edited by McGraw-Hill in 2004 [9]. The textbook is currently being used in undergraduate courses on algorithms and data structures in several Italian universities. More details about this repository and its effectiveness as an aid in teaching algorithms can be found in [4].

4. THE VISUAL EDITOR

As we already explained, the internal representation of an animation in the editor is based on a frame-by-frame structure. Thus, an animation is seen as a sequence of frames,
every frame containing a certain set of graphical objects such as rectangles, lines, etc. Between two frames, the animation seen in the player is obtained by interpolating the positions of the objects, their colors and other properties. In Figure 2 it is shown a high level scheme of the internal organization of the editor.

![Internal organization of the Builder](image1)

**Figure 2: Internal organization of the Builder**

The first functionality that the visual editor needs is the ability to import any already existing script file. Since a parser for the scripting language was already implemented in the player module, we reused that code to partly support this function. In order to correctly translate the definition commands (i.e., the graphical objects, such as “ellipse” or “polygon”) in the script into the corresponding data structures maintained by the editor for every frame, we made a heavy use of the polymorphism of Java. In this way, every definition command is automatically mapped to the right graphical object data structure.

Once the animation is loaded, or a blank one is created, the user can start modifying it. The user can browse the sequence of frames either with simple navigation buttons or by means of a component which holds thumbnail images of every frame. The central window of the editor displays the frame that is currently selected for editing.

![The interface of the Builder](image2)

**Figure 3: The interface of the Builder. The storyboard, including all the frames, is in the bottom part of the interface.**

Through a toolbox representing the different kinds of graphical objects available, the user can add elements to the current frame. As in professional drawing software, there is also a special tool for selecting objects. Multiple selection is allowed, in which case many objects can be moved and modified together. The basic information about the selected objects are shown in dedicated text boxes; these boxes can be clicked to modify some of the properties of the objects, such as their color. The same selection tool allows it to move objects by simply dragging them. Finally, other operations that can be performed on selected objects are duplication (a copy of the object is inserted near the original) and removal from the entire animation.

Thus, through an intuitive interface, shown in Figure 3, the user can insert new objects, move them around, change their colors and other properties, etc. However, notice that even though the interface is intuitive, sometimes it is not entirely obvious what the behavior of the editor should be. For example, say that the user is at the sixth frame of an animation which has in total ten frames, and that he inserts a red ellipse into the frame. Should the ellipse be inserted in the sixth frame only? This would be a conceivable behavior; however, if the user intended to use that ellipse also in a later frame, which is likely, he would need to insert it again there. In order to avoid this problem, we decided to make this kind of changes influence not just the current frame, but also the subsequent ones; in our example, the ellipse would be inserted into the frames from six up to ten. In this way, the user will not need to reinsert the same object again and again. Notice that if the user did not want the ellipse to show up in frames nine and ten, he could make the ellipse invisible in frame nine and this would also affect frame ten. Similarly, moving an object causes the same movement to apply also to successive frames, unless the object is already in a different position in these frames; in that case, the change only affects the current frame. Though we think this behavior is good enough for a basic usability, one could think of making it more flexible by allowing the user to supersede it in some way, for example through an hot-key, or by modifying a preference setting. We plan to allow such a mechanism in the next versions of the visual editor.

When necessary, new frames can be added to the animation sequence. When the user requests a new frame, a copy of the current one is created and inserted into the sequence. Of course, all the objects are duplicated and their properties can be modified independently. The editor also allows it to delete individual frames.

In a way similar to well-known presentation software such as Microsoft Powerpoint, the graphical objects are drawn on different layers and these layers can be individually adjusted through the appropriate commands in the Edit menu, such as “Bring Forward” or “Send to Back”. Since the order in which the objects are drawn on the screen is defined by the order in which they appear in the data structure representing the frame, this feature is implemented by modifying the positions of the objects in this data structure.

As an aid to drawing and positioning the objects, the editor allows it to activate a grid that is drawn into the working
window. When the grid is active, the coordinates of new objects are rounded to the nearest point in the grid. The size of the grid is customizable.

In order to save the modified animation into the differential script format, the editor analyses the animation frame by frame and, by looking at the differences of every frame with its successor, generates the right sequence of commands. For example, if an object exists in both frames but is positioned differently, a move command will be generated. Similarly, if a frame contains an object that does not appear in the previous frame, a definition for that object is saved in the script file. Here, again, the polymorphism of Java helps in associating to each object the corresponding definition command. The saving procedure also takes care of more subtle issues, such as determining the correct sequence of commands that changes the layers of the objects in one frame into those of the next frame. Once saved, the animation can be viewed in the standalone player by launching it directly from the editor. In this way, the user can have an immediate feedback on the quality of the animation.

Finally, we mention that the editor also includes an help system, to get first-time users quickly acquainted with the basic procedures. The help documentation is in HTML format. This format was chosen because of its support for images and because HTML documents can be easily produced and modified with standard tools.

5. CONCLUSIONS

In this paper we presented the interface of the Leonardo Web editor, the Builder, and discussed the related issue of visually editing animated algorithms. Leonardo Web, including the Builder, is freely available under the GPL license and can be downloaded from [15]; we would like to emphasize that, besides the Leonardo Web script language, the Builder allows it to export the animations in several formats, including Macromedia Flash, Microsoft Powerpoint and animated GIF; thus making Leonardo Web an effective tool to produce portable algorithm animations.

As we mentioned in the previous section, we are currently working on implementing in the Builder the opportunity, for the users, to specify which frames are affected by the insertion or deletion of an object.

Acknowledgments. We wish to thank Benedetto Colombo, Paolo Coleccia, Mauro Guerrieri, Marco Longano, and Erin Mulla for their contribution to Leonardo Web.

6. REFERENCES