EMPOWERING LOGO THROUGH A COMPONENT-ORIENTED APPROACH[1]

Manolis Koutlis, Chronis Kynigos[2], Aggeliki Oikonomou, George Tsironis[3],
Computer Technology Institute,
P.O. Box 1122, 261 10 Patras, Greece,
tel: +30 61 273496, fax: +30 61 222086
[2]&
University of Athens, School of Philosophy,
Dept. of Philosophy, Education and Psychology, Section of Education
Panepistimiopolis, Ilissia, Athens, Greece,
tel: +30 1 7248118, fax: +30 1 7248979
e-mail: koutlis@cti.gr, kynigos@cti.gr, oikonomu@cti.gr, tsironis@cti.gr

Abstract

In this paper, we argue for the potential of a Logo - based scriptability in a component architecture for exploratory software. Logo is reincarnated as an autonomous software entity which when embedded in a component-oriented environment takes the role of the central commander, able to manipulate the other components' behavior through suitable Logo programs.

Keywords
Software components, scripting, programmability

1 Introduction

Besides the much celebrated Turtle Graphics, Logo based environments have gained recognition for their expressive power, especially regarding microworld development and the unconstrained ability for creating and manipulating entities of a wide range of complexity.

Parallel to the Logo-based culture, many successful educational pieces of software have been developed in both research and commercial contexts involving a range of subject domains and

[1] This work is being accomplished in the course of project "YDEES" (http://www.cti.gr/RD3/ydees.html), funded by the European Community Support Framework II (Greek Ministry of Industry Energy and Technology, General secretariat for R&D, Measure 1.3, Project 726.) The project's objectives include (a) the development of exploratory learning environments for mathematics, geography, physics and music based on a component-oriented software architecture (b) the development of curriculum for focused exploratory activities (microworlds) (c) concurrent development of exploratory classroom cultures in five primary schools (d) evaluation of the software, the classroom learning environments and content-specific learning processes.
[3] Names appear in alphabetic order.

pedagogical approaches (like SimCity, Interactive Physics, CamMotion, Function Probe, Tabletop and Centenia, to mention just a few). These "applications", having typically been developed by means of general purpose software development tools (like C or C++ for example), do provide the valuable functionality they were designed for, but are usually not programmable, i.e. they do not allow the user to algorithmically alter their behavior.

On the other hand, there has been recent and growing appreciation of the potential for component-oriented architectures for educational software as an alternative to the "application-oriented" model ([5], [10], [12]). In this model, suitably designed "components" are autonomous, high-level and reusable software entities, with either domain-specific or general purpose functionality (like graphs, maps, globes, tables, editors, sliders, buttons, vectors, turtles, etc.). These components act as the building blocks for the construction of compound microworlds. We thus suggest that programmability is an important feature for component architectures in that a microworld's functionality can be specified by defining the behavior of each participating component as well as the interactions among them, through a so-called "scripting language", i.e. a specially designed language that provides the syntax and mechanisms to instruct components to act in a specific way.

In this paper we present a new paradigm for creating exploratory microworlds, by providing a mechanism that allows Logo and the formalization that goes with it, to be directly embeddable in traditionally "alien" software settings. To do that, Logo is given a new face and a new role: in a component-oriented world Logo can be thought of as a component itself, at the same conceptual level as any other component, with the special characteristic that it can directly affect other components' behavior through common Logo programming. A Map component, for example, could zoom-in to a specified area by a specified factor through suitable Logo commands that are supported by the Map component. The benefits are twofold: the component-oriented approach inherits Logo's expressive power, and Logo is reincarnated as a core tool for formal expression in modern software environments. Zooming-in on the map component, for instance, can be done either visually when the focus is strictly on the effect of the action or by means of symbolic descriptions which provide accuracy, can be used as tools and as objects and can have parameters providing them with behavioral properties.

The rest of this paper is as follows: we begin by quickly reviewing some key issues of the component-oriented approach in educational contexts, then we give an example in familiar territory by describing how a traditional Logo environment could be conceived as a suite of components, and next we present the mechanism that puts Logo in the components' commander role, and some examples that demonstrate its uses.

2 Why Components?

Component-oriented environments provide end-users with a pool of components, potentially developed from different teams in different programming environments, which can "live" in a common space and tightly cooperate one with one another. Using a kind of an editor they may assemble any necessary number of components (each one participating as many times as required) in functional configurations which, as a whole, can behave as coherent microworlds. To further define the functionality of a microworld, they may either "interconnect" the components in suitable ways, or they can use an end-user programming language to write small scripts that "talk" to components and affect their behavior. There are already a number of research results demonstrating the above, like the SimCalc ([12],

http://tango.mth.umassd.edu/) and Avakeeo ([9], http://www.cti.gr/RD3/avax.html) environments.

The advantages of such an approach include among other things:

- Large scale reusability. A component, once designed, can participate as many times as required, in as many microworld contexts as necessary (i.e. one design can accommodate many situations). On the other hand, it is for this reason that the design of a component is more difficult than the design of a simple application. Apart from the originally intended behavior of a component, the designer has the task to envision the cluster of potential behaviors of which this particular one is a case of.
- Seamless sharing and integration of components. Recent technological standards (like OpenDoc, ActiveX and JavaBeans) guarantee that components developed even by different teams, in different programming environments, can work in synergy in a plug-and-play manner. In that way, duplication of effort is minimized and reuse of brilliant software in much more contexts than those it were originally designed for, is made possible.
- Microworld reconstructibility. Microworlds can be composed, altered, extended or decomposed to their constituent components by end-users as desired, just like in Lego construction kits.
- Opportunity for novel designs. Components call for a rethinking of the nature of traditional computational tools and functionalities, leading to new designs and approaches ([9]).

3 Logo, the component way

In our design, Logo itself is viewed as a component, at the same conceptual level as any other component. The target was to embed Logo in an open component-oriented world, while at the same time retaining its traditional functionality. The questions raised by this approach involved deciding on which components Logo should consist of and what their individual functionality would be.

To address these issues we took advantage of the new functional characteristics that component-oriented development provides. Thus, a traditional Logo environment-application can now be formed by assembling three separate software components (see Figure 1):

- The Logo component, incorporating a parser of the Logo language. This is the component able to interpret the language primitives, to define new procedures and either execute the user's commands or pass them to a Logo Turtle. The parser's front-end is a text editor, where the user types the Logo commands and views their output.
- The Graphics component, which represents the window a Logo Turtle walks on. It is the canvas on which all Turtle's graphical output that results from commands, edited in a Logo component, are executed by the Turtle(s) being connected with both of these components.
- The Turtle component, which is not just the iconic representation of a Turtle on a Graphics component, but a separate entity which has its own, user-defined, characteristics and behavior, such as color, shape, pen or rubber. Moreover, a Turtle can be dynamically bound to any of the Graphics and Logo components that participate in a given microworld.

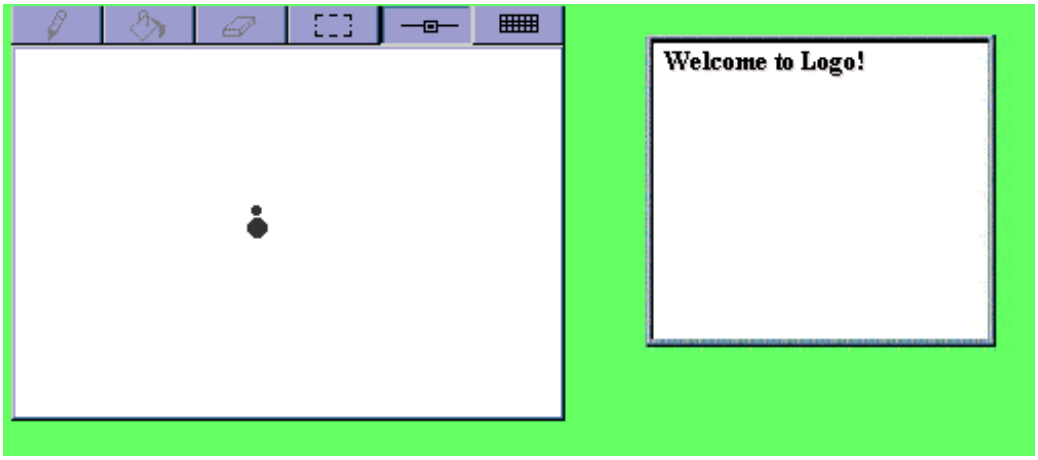

Figure 1: Logo in the form of three cooperating components: Logo, Turtle, Graphics.

Now that these components are separate instances, it is easy to construct a microworld with more than one Logo components, each one addressing more than one Turtle components, which do not necessarily have to walk on the same Graphics component. Moreover, the functionality of each of these components can be isolated and designed separately. As a result, the Graphics component is also designed and used as a general-purpose drawing environment where free-hand drawing can be performed ([4]) or representations of graphs can be visualized; thus, the same software component can be reused in many different activities.

At the same time, any other software component (like for example a Map, a Table, a Slider, etc.) can function within the same Logo environment. The new functional characteristics of such a microworld go far beyond a shallow coexistence of different kinds of software that offer a juxtaposition of various features: the component architecture brings up a new prospect for reconsidering not only the structure of a Logo application itself but also its synergy with other software components. In this context, it becomes more straightforward conceptually, and more feasible technically, for Logo to directly communicate and exchange data with other components that may have been designed for different purposes. It is up to the end-user to decide on how many of the components provided and what portion of their functionality (s)he is going to use at any time.

4 ComponentLogo: a new suit for Logo or a new prospect?

The establishment of interrelations between software components, like those described in the previous section, made us reconsider the extensibility of the software tools that had already been developed and the roles they could play in subject domains other than the ones that they were originally designed for. In the case of Logo, in particular, it became obvious that in a component-oriented environment the Turtle paradigm could be considered as just an instance of a more general approach to the way a programming language can manipulate all coexisting software components within a microworld.

From this perspective, ComponentLogo, the component flavor of Logo, is not just a revamp of Logo or just an attempt to port it to a new environment, but much more:

- ComponentLogo is no more a standalone application, but a reusable component that can be embedded in many contexts.

- Other components, with minor programming additions that give them “Logo awareness”, can now be manipulated through a formal language.
- ComponentLogo can access and manipulate internal attributes and functions of other components, or export its own internal variables to them.

The ComponentLogo component comprises of an editor and a parser, providing the facility for editing, testing, and interpreting ComponentLogo programs. The ComponentLogo language is based on Berkeley MSW Logo ([6], [11], ftp://cher.media.mit.edu/pub/logo/software /mswlogo), suitably augmented to support the additional required functionality. Other components, communicate with the ComponentLogo component through their “programming interfaces”, which include:

◊ The set of primitives that the component makes available to the ComponentLogo component. These primitives may be used in programs, in order to control the state/behavior of the particular component.

◊ A mechanism for handling each event that is generated by the ComponentLogo component, as a result of the execution of a primitive that belongs to the programming interface of the specific component. In essence, the latter triggers an appropriate internal action that corresponds to the executed primitive.

It is through this interface that programs written in the ComponentLogo language can affect the state and the behavior of other components. As programs are parsed and executed by the ComponentLogo parser, suitable “events” are propagated to the target components which in turn translate them to corresponding actions. The primitives of a component’s programming interface are dynamically “loaded” to the grammar of the ComponentLogo language as soon as the component is embedded in a microworld. This is made possible by means of an extension mechanism that allows on-the-fly registration of new primitives.

A central issue that had to be addressed concerns the syntactic additions that would allow the identification of the target component(s). When the ComponentLogo parser encounters a “foreign” primitive, registered by a component as part of its programming interface, it has to send an event to that component. ComponentLogo is aware of all other components participating in a specific microworld and also keeps track of the number of components of the same type (each foreign primitive is associated to the type of its parent component). Thus when the parser evaluates a primitive of type Map, for example, and if the current microworld contains only one instance of this type, then the parser can safely assume that the destination part is that sole instance. If however there exist more that one components of type Map, then the parser cannot judge, and the user has to explicitly specify the destination part. Two alternative but complementary syntax schemes have been developed for this:

```
a)( tell [ <ComponentA> <ComponentB>  <ComponentN> ] )
b)<ComponentName>  <primitive>  [<argument1>  <argument2>…
  <argumentN> ]
```

The first scheme initiates a block of commands. In this block, each evaluated foreign primitive that might have more than one destination components, is sent to the component(s) specified next to“tell” that have the same type as the primitive. Native ComponentLogo primitives might also be used in a tell block. A tell block is terminated either by another tell command that changes the list of destination parts or by a standalone tell, which simply resets the list of recipient parts. The second scheme specifies that the destination component of the foreign

<primitive> is the component with name <ComponentName>. The <primitive> included in such a command cannot be a native ComponentLogo primitive. Also, commands of this type may be embedded in tell blocks.

5 ComponentLogo in action

The following examples illustrate the use of ComponentLogo. The first microworld (Figure 2) consists of a Turtle component and a ComponentLogo component. The well-known turtle navigation primitives no longer belong to the ComponentLogo grammar since they are meaningless in a microworld that contains no turtles. Instead, the primitives belong to the programming interface of the Turtle component, which declares them to the ComponentLogo grammar whenever it is embedded in a microworld. The user might use these primitives in ComponentLogo programs in the same way that native ComponentLogo primitives are used.

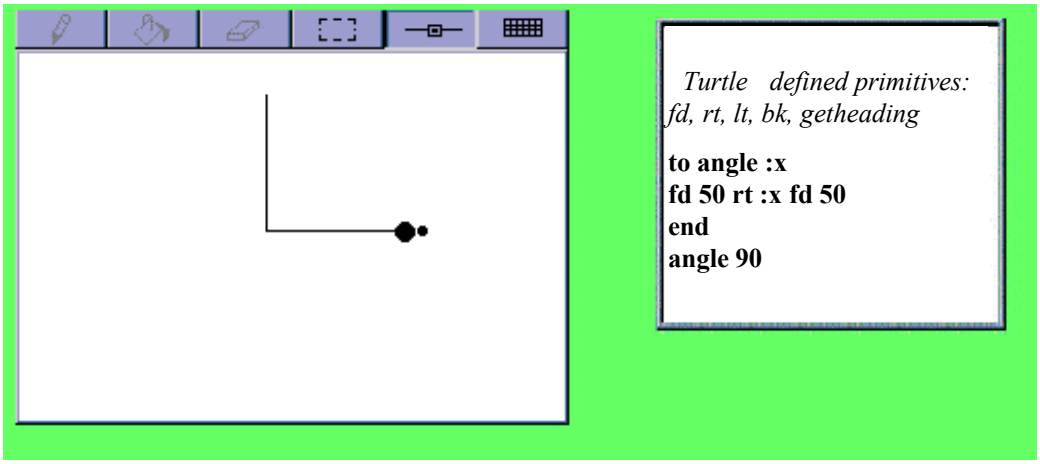

Figure 2

The next example demonstrates the use of the “tell” primitive.

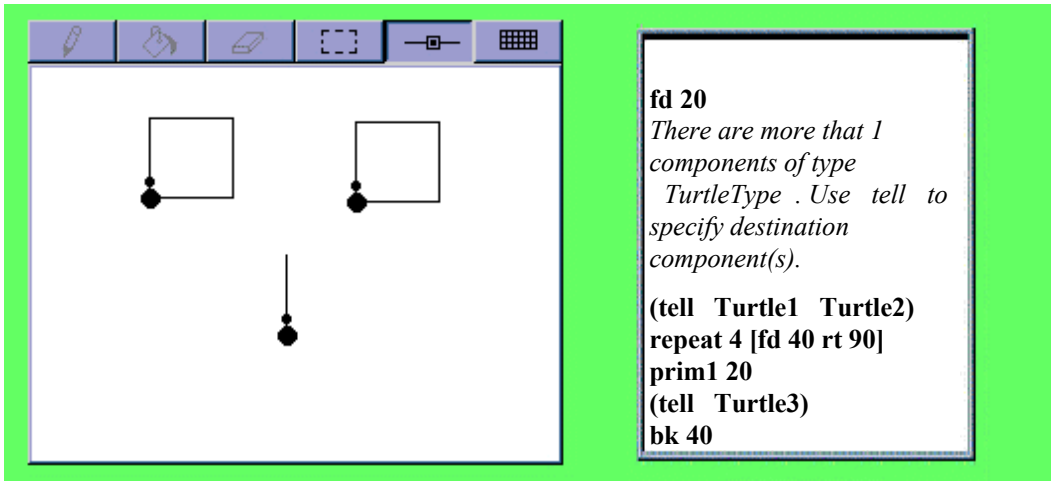

Figure 3

The ComponentLogo parser responds to the first `fd` command with a message informing the user that the destination component(s) of the command should be specified, since there are now more that one potential recipient turtles. By using "tell", the user specifies `Turtle1` and `Turtle2` as the destination components, and so the commands contained in the `repeat` loop that follows are executed by the first two turtle components. The next "tell" sets `Turtle3` as the destination component for the commands that follow.

The next example demonstrates the alternative way of defining the recipient components.

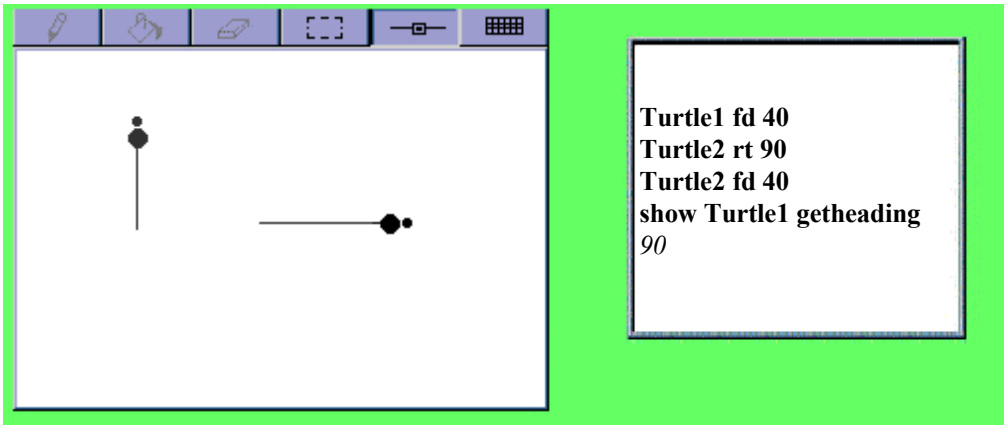

Figure 4

The first command addresses `Turtle1,` while the next two commands, `Turtle2.` The last command can be decomposed into two sub-commands: `show`, which is a native ComponentLogo command and simply prints out the value of the single argument it accepts, and, `Turtle1 getheading`. The latter is executed by `Turtle1`. The result of the execution is fed back to the parser and provided as an argument to the `show` command.

The next example demonstrates a mixture of the two syntax schemes.

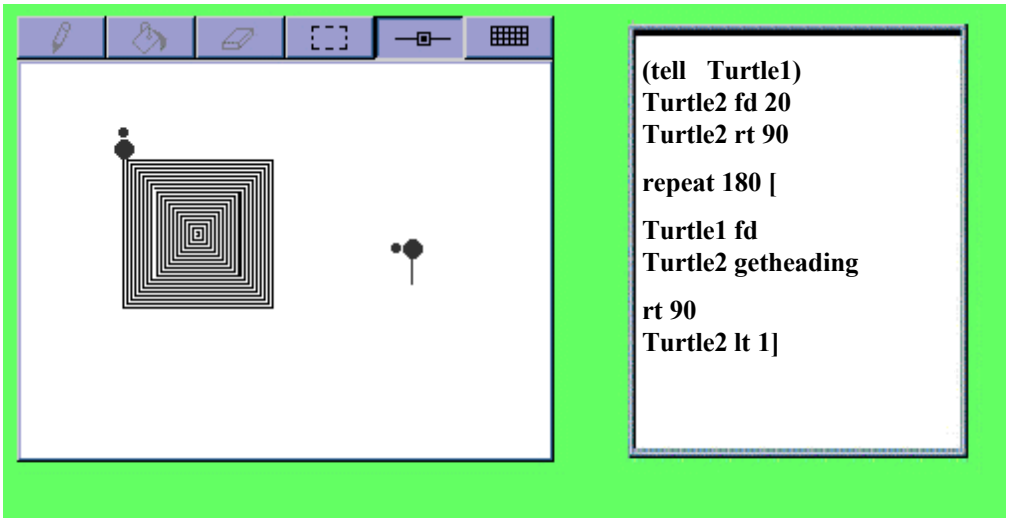

Figure 5

"Tell" specifies `Turtle1` as the default recipient of commands, originating from the programming interface of `TurtleType` components. `Fd` and `rt` address `Turtle1`, moving it 20 steps forward and turning it 90 degrees to the right. The heading of turtle `Turtle2` is now 0. The following command is a compound `repeat` statement. The first command in the `repeat` block is of particular interest. Actually, it is two commands, the second one supplying an argument to the first one. The second is executed by `Turtle2`, whereas the first is executed by `Turtle1`. So this command forwards `Turtle1` as many steps as is the heading of the turtle of `Turtle2`. The next `rt` is posted to `Turtle1`, because of the preceding `tell`. The last command turns `Turtle1` 1 degree to the left, increasing its heading by 1.

By elaborating on the variations of talking to Turtles, as was presented in the previous examples, we do not of course intend to repeat well known issues, but rather to draw from familiar turtle-based scenarios to others involving more general and complex components like those described below.

The next example illustrates the use of ComponentLogo to manipulate a Table component. Table defines the primitives: `getcell`, `setcell` and `sum` to the ComponentLogo grammar, which can be used to retrieve the values of any cell or combination of Table cells, and utilize Table's functionality for calculating new values and/or storing these values into other cells.

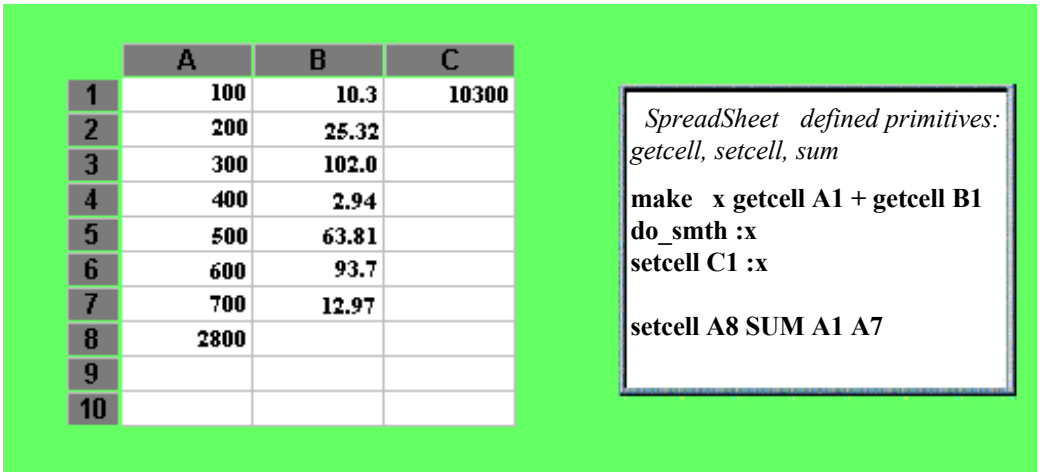

Figure 6

Finally, the next example[1] demonstrates the use of ComponentLogo in a geography microworld consisting of a ComponentLogo, Slider, Turtle Globe, WorldMap and Clock components. These are configured in a way that the WorldMap's day-night areas are also reflected to the Globe's view (achieved by just synchronizing the GMT-times of the two components), the Turtle's positioning is reflected on both the WorldMap and the Globe, and its corresponding local-time on the Clock's reading. In that way, any change in a component's state is automatically reflected on any other dependent components, accordingly.

[1] This example has been adapted from [10].

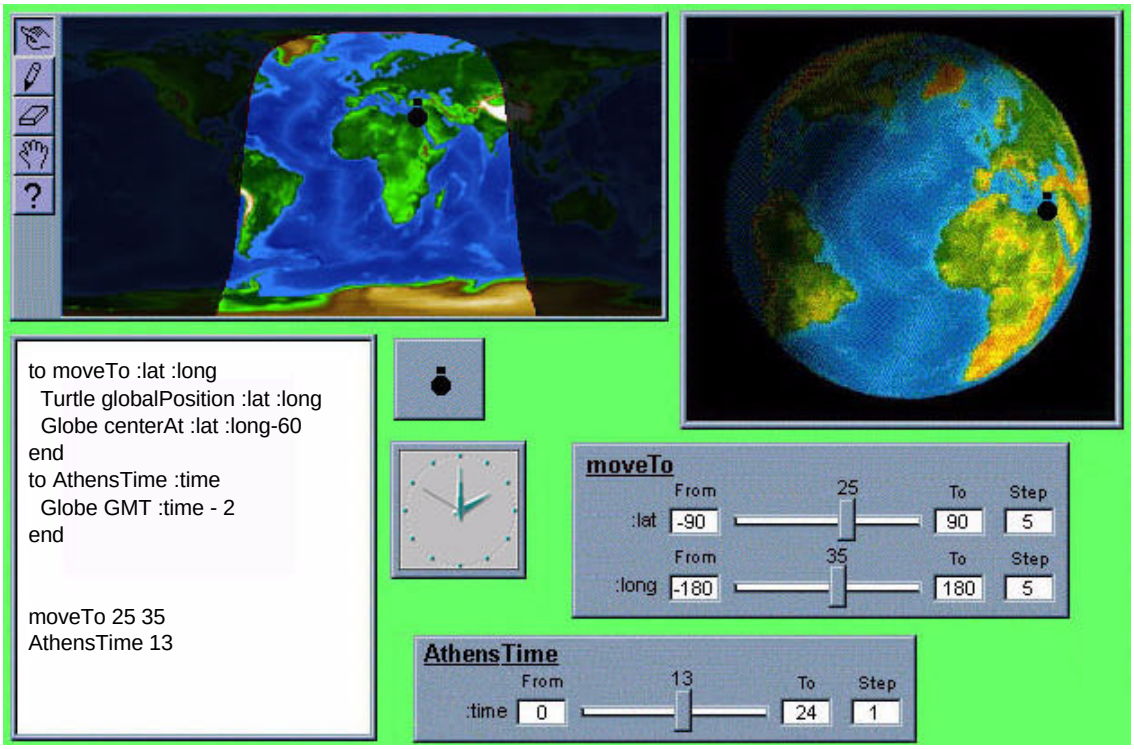

Figure 7. A geography-microworld utilizing two Slider components

The ComponentLogo component now supports primitives inherited from the newly introduced components. For example, the Globe component provides the "`GMT :time`" function that adjusts the globe's GMT-time to the specified value affecting the displayed day-night areas, and also the "`centerAt :x :y`" function that centers the globe's view around the specified location. On the other hand, the Turtle component is now augmented with the "`globalPosition :latitude :longitude`" function that moves the turtle on the specified position (reflected on the Globe and WorldMap components).

6 Conclusions

In this paper, we have proposed an ambitious scenario for Logo's future by outlining a new role for it in the modern software engineering world. In that way, both worlds are benefited and the tendency to underrate formal, text-based language as something technically outdated is surmounted: the Logo community gains the opportunity to incorporate and control complex and rich functionalities offered by the latest software technologies, while the latter are provided with an exploratory vehicle in the educational context. We think that not only does programmability provide expressive power not achievable by other means, but also that the availability of new media can enhance the scope of formal expression. Provided with components which can have a wide range of behaviors, Logo becomes a powerful means of controlling behaviors of a wide range of entities. The component architecture through upcoming commercial standards provides the required technological infrastructure to support this promising merge.

7 References

[1] Apple Inc. 1996, OpenDoc Human Interface Guidelines.

[2] Apple Inc. 1996, OpenDoc User's manual.

[3] DiSessa, A., Abelson, H.:1986, "BOXER: A Reconstructible Computational Medium." Communications of the ACM vol. 29 no 9, pp. 859-868.

[4] Eisenberg, M.:1995, "Programmable Applications. Interpreter Meets Interface". SIGCHI Bulletin, 27(2), April 1995, 68-93.

[5] Hadzilacos, Th., Koutlis, M.: "A framework for the Computer Aided Spatial Education through Geographic Microwords", Workshop in Advances in Geographic Information Systems, ACM, Washington DC November 1993.

[6] Harvey, B.: 1985, "Intermediate Programming", Computer Science Logo Style, Volume 1, MIT Press.

[7] Harvey, B.: 1993, "Symbolic Programming vs. Software Engineering - Fun vs. Professionalism - Are These the Same Question?" in P. Georgiadis, G. Gyftodimos, Y. Kotsanis and C. Kynigos, (eds.), Logo-Like Learning Environments: Reflection and Prospects, Proceedings of the Fourth European logo Conference, University of Athens, Department of Informatics, August 1993, Athens (Greece), pp. 28-31.

[8] Hoyles, C. and Noss, R.: 1992, Learning Mathematics and Logo, MIT Press.

[9] Koutlis M., Hadzilacos, Th.: 1996, "Avakeeo: the construction kit of geography microworlds", in press.

[10] Kynigos, C., Koutlis, M., Hadzilacos, T.: 1997, "Mathematics with component-oriented exploratory software". To appear in: International Journal of Computers for Mathematical Learning.

[11] Logo (Berkeley) for Windows: 1996, Users' Guide.

[12] Roschelle, J., Kaput, J.J.: 1996, "SimCalc Mathworlds for the Mathematics of Change", Communications of the ACM, August 1996/Vol. 39, No. 8, pp. 97-99.