42. Building microworlds with complex, pre-defined Imagine Logo components

Andor Abonyi-Toth

Assistant professor

Eotvos Lorand University, Informatics Methodology Group, TeaM Lab, Hungary

abonyita@ludens.elte.hu

Abstract

In case of programming educational microworlds, games and simulations we often realise, that we need to implement the same functions (with minor changes) in many different applications. If we can generalise these common functions in separate components, then these building blocks would be usable in different applications, resulting in faster application development. A further advantage would be that the final applications would be uniform with regards of the appearance and usage, that eases the cognitive strain of users. Imagine Logo has already built-in components (like button, slider, panel, web browser, input box) which already are of great aids for the application developers in itself, but one can develop new components too. These new components would further aid development of educational microworld especially among teachers and learners who are not very professional in programming. The paper describes the developed components as well as illustrate through some examples their use in building more sophisticated component-based programs.

Keywords

ImagineLogo, building blocks, faster application development, new components

1. Introduction

With the Launching of ImagineLogo, students, teachers and developers got an excellent tool that greatly supports fast application development. Let us just think of the use of built-in event-controlled components (slider, web browser, multimedia player) that makes it possible for us to create quite spectacular applications, fast and simply.

However we have to see that in many cases we may need more complex components, procedures which we want to use in several applications. (e.g. a component created for plotting 2 dimensional functions could be used by teachers when developing microworlds for different subjects [mathematics, physics, chemistry, etc.] as well.

In this case we have to make sure that our components are developed in a way so that they can be inserted easily into the different applications.

If we could create a standard for this purpose then users could assemble their own applications by downloading from the Internet components necessary for their intentions. Also more advanced users could develop new components that could then be shared by others. This would provide a constantly developing component library available for ImagineLogo users.

In the following I would like to introduce a component standard used in our research group (and recommended for others as well) that offers the following opportunities for the users:

New components are accessible via toolbars in such away that they can be pasted into the program similarly to standard components.

Users can assemble the toolbars by themselves through which the components are accessible. (The order and grouping of icons can be changed easily).

"Installing" the components does not need more than unpacking a compressed file into a subdirectory and refreshing the content of the toolbar by running a given procedure.

The input/output of each component can easily be connected to the output/input of another component.

For creating new components it is only necessary to know the ImagineLogo language therefore mo re advanced users can create new components themselves.

The accessibility of the detailed documentation and the source code of the created standard will be revealed in my presentation. In the following let us see the principals used when creating components.

2. The structure of the components

While making components the most important issue us that they should be easily inserted into the different applications. For inserting new components the use of toolbar is the most convenient.

2.1. The use of toolbars

Imagine makes it possible for us to make our own toolbars and place buttons onto it. For example:

We create a toolbar called mytoolbar

```
? mainwindow'new "toolbar [name mytoolbar]
```

Onto it we place a button whose property list contains the followings: (caption "New Slider", its size is 60x30, when the button is pressed the command mainwindow'new "slider [pos (list any any)] is executed, that is a new slider object is created on our page.)

```
? mytoolbar'new "button [caption |New Slider| size [60 30] event'onpush
[mainwindow'new "slider [pos ( list any any )]]]
```

Figure 1. The result of our program code

Similarly to this we could place other buttons as well that can also have the shape of icons and thus it becomes possible for us to create an individual (graphic) toolbar.

However creation of toolbars could be automatic as well to make it easier to insert new buttons (or maybe new toolbars). For this purpose we can associate toolbars with directories, and the buttons on it with subdirectories in which text files contain the properties of buttons, (for example in which position they should be located or what command should be executed when they are pressed, etc…). In the standard we use we have created a general procedure

that assembles the toolbars based on these subdirectories. According to this, adding a new toolbar or button means nothing more that creating or copying files.

Figure 2. An example toolbar

The toolbar called c_toolbar1 on Figure 2 contains 2 icons and is associated with the subdirectory structure below.

Imagine root directory

```
└──c_toolbar1
    ├──1
    │   button.lgf
    │   makeclass.txt
    │   property.txt
    │
    └──2
        button.lgf
        makeclass.txt
        property.txt
```

According to created standard the following files can be found in each subdirectory:

Button.lgf: the set of pictures on the button in LGF format

Makeclass.txt: contains the definition of the special component relating to the button (the procedure that creates the toolbar also creates the new component class based on the definition found here)

Property.txt: contains the properties of the button.

Property.txt follows the structure of windows INI files so that it can be modified easier. (there are several sections in the file where we can set the parameters related to the section) The structure of the file:

```
[separator]     ; there should be a separator before/after the button
before = false
after = true
[gap]           ; there should be a gap before/after the button before =
false
after =false
[c_button]      ;description of the properties of the buttons
caption = [New Slider]
.... other button properties
```

If the user sets the value of the variable before to true in the separator section then a separator will appear before the button when the toolbar is refreshed. In this way users can change the position of the buttons as they wish by modifying the text files.

2.2. The structure of components

When creating our components we have to make sure that the different components are handled similarly, they can be connected to each other, the elements of the component can be moved together and most importantly they can be created by running a procedure (the code of this procedure is put into the makeclass.txt file in the directory structure above.)

How can we make components that fulfil the requirements above?

The best way is to create new components as classes. Let us take a simple example. Let us create a new component that differs from the slider component in that its value is shown next to the slider. (Figure 3.) This is a function which we use regularly in our applications so making a new component for this purpose is practical.

Figure 3. Modified slider component

To be able to move the elements of the component together the best solution is to place the elements on a pane that is the class of our new component (myslider) what is a descendant of the pane class. (Pane is a rectangular movable area for turtles and other objects created anywhere on a page or another pane in ImagineLogo.) Let us set the width of the pane to be 230, the height to be 35 pixels, set the namePrefix to myslider (that means that the different components will get the name myslider1, myslider2, etc…), let its origin be coordinates [0 0] and we should be able to drag it with the mouse.

```
? mainwindow'newclass "pane "myslider  [width 230 height 35 nameprefix
myslider origin [0 0] autodrag true]
```

When the new pane is created the slider and textbox components should appear as well. For this we use the event onCreate. (the event occurs when we create a new instance of the class). As an effect of the onCreate event we have to create a textbox and a slider with the appropriate properties. According to these our procedure changes as follows:

```
mainwindow'newclass "pane "myslider
 [width 230 height 35 nameprefix myslider origin [0 0] autodrag true
 event'oncreate [
 new "textbox [name (word myname "_ "value) apos [185 8] growwithvalue
true oneline true transparent true height 20]
 new "slider [name (word myname "_ "slider) apos [8 8] max 5000 min 0
size [160 23] thumblength 10 transparent true value 1 ]]]
```

In the next step we have to work on the task that when the value of the slider changes the value should appear in the textbox. For this we have to write the onChange event of the slider as well which we have to place in the property list of the slider component.

```
event'onchange [ let "my_place myplace'myname  let "my_name myname
ask (word :my_place  "_ "value) [run (word "setvalue char(32) :my_name
"\' "value)]]
```

Now we are ready, the only thing left is to create an instance for our component.

```
? new "myslider []
```

Running the program code in the different applications we can create the new class and using the command above we can also create its new instances. With this it became possible to use the new component in other applications as well. If we save the program code into a textfile (taking care not to use comments and that commands are written in one long line) then we can load the code with the load command as well that is executed by Imagine automatically. Taking advantage of this function we created our standard by place the program code necessary for creating the class into the makeclass.txt file within the directory containing the definition of buttons.

2.3. More advanced properties of components

By following the logical steps above we can build our own components to which we can add more advanced functions as well. In the following the more advanced functions of the created components are introduced by presenting the possibilities of a specific component. As an example let us see what possibilities we added to the slider component above.

Figure 4. Modified slider component

As Figure 4 shows have modified our component so that apart from the actual value, the minimum and the maximum value of the slider also shown.

The advanced properties of the component:

Components have their individual menu (contextmenu) that can be seen by clicking with the right mouse button (Figure 5)

The first three sections of the appearing menu comes from the default menu of the pane. However functions below the section ---Component Functions -- are specific to the component.

With the help of the functions in the following section we can set the values of the component (minimum, maximum, value), we can make the background of the pane transparent or we can change its colour.

We can even connect components with the menu point Connect to other component.

We can disable the appearing of the context menu with the Disabled functions menu point. Thus it becomes impossible for the users to change the components placed on the pane.

Using the last menu point it is even possible to clone the component.

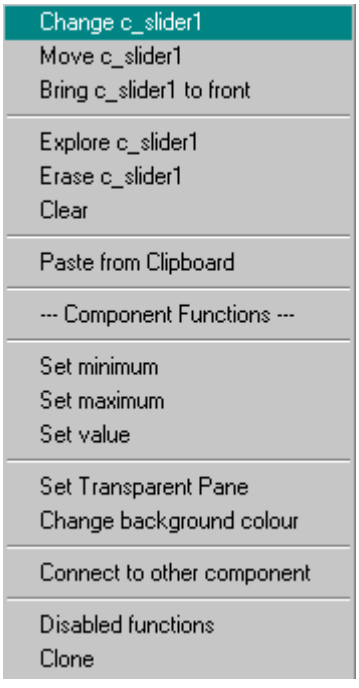

Figure 5. The context menu of the component

We can set the values of the components by using commands as well. (e.g. the c_slider1'setmin 200 command is interpreted and with this we set the minimum value of the component. This way we can give values to the components or ask them from the command line.)

The inputs/outputs of components can be connected to outputs/inputs of other components. After choosing the Connect to other component menu point we can choose from four options (Figure 6)

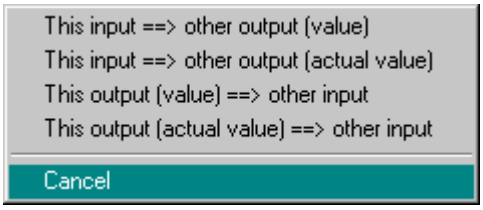

Figure 6. Menu for connecting components

As it is clear from Figure 7, the input or output can be connected to the actual value or value of the output of another component. (in the latter case the value of the first component automatically changes when the value of the second component changes.) After choosing the appropriate menu point the parameters appear between which we can create a connection (Figure 7). If we connect the actual values of the components we can choose the frequency with which values are transferred from one component to the other. (e.g. in every 2 seconds, see Figure 8).

Figure 7. Menu for choosing parameters

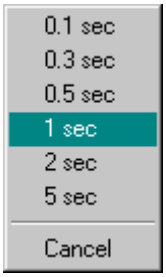

Figure 8. Choosing the timing

The way components are connected can be seen if we move the mouse over the target component, then a red border appears around the component and also a connection point. (On the connection point an O or an I is written depending on whether we can choose from the output or the input parameters - Figure 9.) By clicking on the connection point a menu appears where we can select the parameters. As a result the connection is created between the components.

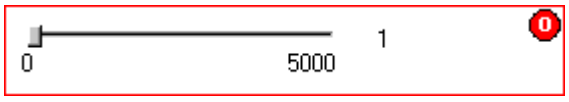

Figure 9. Figure of the component

We store the commands that connect components in a separate procedure. Therefore it is possible for the connections created to work even after saving and reloading our application.

Connections are working even when components are not located on the page.

2.4. Conclusions

Offering more complex components could primarily give help to those teachers who wish to create microworlds to be used in education with minimal knowledge of Logo programming by using the drag and drop method. An unconcealed goal of these components also to turn the interest of teachers towards programming challenging them to try to realise even the ideas for which components do not yet give solution.

The standard we introduced can be suitable in creating components that are compatible to each other, unified and can be installed easily and thus may make ImagineLogo application developers' work much easier.

We plan to create a portal on the Internet where components can be up- and downloaded according to topics. Here we also will publish support programs and descriptions necessary for making components.

3. Acknowledgement

This project has been carried out with the partial support of the European Community in the framework of the Socrates programme MINERVA 101301-CP-1-2002-1-HU-MINERVA-M.

4. References

Imagine Reference guide written by A. Blaho, I. Kalas, L. Salanci, P. Tomcsányi &Logotron Limited