WHIZZ'ED: A VISUAL ENVIRONMENT FOR BUILDING HIGHLY INTERACTIVE SOFTWARE

Olivier Esteban *Stéphane Chatty* *Philippe Palanque*[*]

Centre d'Études de la Navigation Aérienne
7 avenue Edouard Belin, 31055 TOULOUSE Cedex FRANCE
E-mail:{esteban, chatty, palanque}@cena.dgac.fr

KEY WORDS: interface design, visual programming, direct manipulation, animation.

ABSTRACT: This paper describes the issues raised by the development of visual tools for the construction of graphical interfaces. It presents Whizz'Ed, an experimental editor for construction of highly interactive or animated applications. In addition to the features of traditional interface builders, Whizz'Ed makes it possible to visually describe by direct manipulation the behaviour of graphical objects and their interrelations. Whizz'Ed encapsulates basic behaviours in elementary bricks that can be connected using a data-flow model. The flow graph can be structured in order to reuse complex behaviours, thus allowing the designer to create new reusable bricks at design time.

1 INTRODUCTION

Visual programming has a number of applications ranging from educational software to specialized programming tools, dedicated to domains such as signal processing or image processing. Most of these applications exercise the ease of use offered by visual programming, in order to give access to programming to non-specialists in computer science (Chang, 1987). A domain where that ease of use would be greatly appreciated is user interface design. The design of an interactive application usually involves human factors specialists, future users of the application, and occasionally graphics designers. These people usually do not have an education in computer science, and often have to rely on highly specialized programmers for building prototypes of the system they are designing. Furthermore, the design of efficient and usable interfaces requires many iterations on the design and evaluation of prototypes. As long as complex programming is necessary, the development of high quality user interfaces will be very costly; besides easier to use interface construction tools will hopefully allow to build better interfaces.

However, building visual programming tools for interface construction is still a stimulating challenge, especially when it comes to developing highly interactive, direct manipulation interfaces (Shneiderman, 1983). A number of interactive construction tools have been developed during the last years, and are now widely available. These tools, commonly known as interface builders, offer a partial solution to the problem of building graphical interfaces. They allow their users to instantiate and customize interactive objects such as buttons, dialogue boxes or menus, which are usually called interactors, or widgets. For instance, commercial interface builders such as HP Interface Architect, TeleUSE or XFaceMaker, are based on the OSF-Motif widget set (OSF, 1993). Visual Basic for Windows uses the set of widgets provided by the CUA standard (IBM, 1991). These tools can be considered as visual, in that they provide graphical representations of the objects of interest (the interactors themselves), and allow to manipulate these graphical representations in order to set the parameters of the objects.

But such interface builders can definitely not be considered as visual programming tools, since setting parameters is not programming. That lack of programming capabilities can be related to another weakness of these tools: they can only help to build what we call static interfaces. Such interfaces are able to react to user's actions, but in fact, their only difference from static figures lies in predefined behaviours, which can hardly be changed. Creating more dynamic, highly interactive interfaces, such as iconic interfaces or MacDraw™-like drawing tools, requires more than the instantiation and parameterization of predefined interactors. Some form of programming is needed to describe how users can move graphical objects around, change their shapes, or make copies of them, for instance. In fact, some of the interface builders we mentioned earlier offer facilities for reprogramming or enriching the behaviour of their widgets. But for that purpose, they provide specialized textual programming languages whose manipulation requires traditional programming skills. The chal-

[*] *also with LIS, University of Toulouse, 1 pl. Anatole France 31042 Toulouse Cedex (FRANCE)*

lenge here consists in providing easy-to-use programming facilities, so that interface designers can build highly interactive applications or prototypes.

The following section describes all the important issues that have to be addressed when building highly interactive interfaces. The next one describes previous work on user interface construction and visual programming. We then present Whizz'Ed, an experimental visual tool devoted to the construction of highly interactive or animated interfaces. Finally we present, with a simple case study, how Whizz'Ed and its conceptual model can be used for the design of an interactive application.

2 ISSUES RAISED BY INTERFACE BUILDING

The complexity of user interface software is not related to algorithmic complexity: pictures used in today's graphical interfaces are fairly simple, and only simple geometric computations are necessary. Furthermore, such computations can be encapsulated in reusable graphical objects (rectangles, splines, polygons, etc.) or geometry managers (alignment constraints, graph managers, etc.) that are nowadays well mastered. The complexity of an interactive application actually lies in the behaviour of graphical objects, and in the many interrelations between them (Myers et al., 1990). As a consequence, a visual tool for interface construction does not need to provide representations for sophisticated numeric loops and tests. The need is rather to identify the correct building blocks for describing behaviours, interrelations between objects, to offer structuring mechanisms to help managing the remaining complexity, and to allow the dynamic management of objects, as it is common to have very short-lived objects e.g. ghosts used for feedback information in direct manipulation interfaces. The behaviour is characterized by the reactive nature as each object is always idle and waiting for events to react to, according to events that may be initiated either by user's actions or by time pulses. The graphical interrelations between objects describe relations in the behaviour of different objects (e.g. two objects moving according to the same trajectory) and are called *geometrical constraints*, while the interrelations between time and objects are called *temporal constraints*.

3 RELATED WORK

Providing tools that are able to take into account the characteristics described above is still a challenge. However, some research has been done in that direction. The three main approaches are constraint based systems, the path-transition paradigm, and data-flow systems.

SketchPad (Sutherland, 1963) was the first drawing system that used constraints. SketchPad allowed lines to be constrained by relationships with other lines (perpendicular, parallel, etc.). ThingLab extended that notion by providing a general simulation environment (Maloney et al., 1989). Constraints in Sketchpad and ThingLab are bi-directional and allow objects to be attached and updated simultaneously. Garnet (Myers et al., 1990) is another constraint based system that contains a set of tools to assist the design of user interfaces. However, constraint based systems do not allow the description of behaviours in a very natural way. Indeed, it is more difficult for designers to express behaviours in an abstract way (such as needed with constraints) rather than describing causality links between objects (such as needed with data-flow based systems). Moreover, usually constraints are to be expressed in a textual and thus hard to understand language (e.g. temporal logics). Indeed, the use of graphical notations for describing behaviours (as in path-transitions or data-flow diagrams) is easier and thus such systems can be used by a broad range of people with different programming levels. Tango (Stasko, 1989) is an algorithm animation system based on the path-transition paradigm in which one describes paths, attaches graphical objects to them, and then plays the resulting transitions. Tango contains a WYSIWYG demonstrational design tool called Dance for designing animations. But using the path-transition paradigm makes difficult the expression of temporal constraints, as well as geometrical constraints. In order to allow natural description of behavioural constraints (both geometrical and temporal), the data-flow paradigm has been introduced.

Systems based on data-flows make programs easier to construct due to the natural understandability of data-flow diagrams. NL (Harvey and Morris, 1993) is a visual programming language, based on a data-flow programming model. A NL data-flow program is a directed graph. The arcs represent the paths over which tokens move between nodes, where they may be transformed into other tokens. NL uses a data-driven firing rules: a node is fired when each of its input ports holds a token. NL provides composite nodes which enable programmers to recode groups of nodes into comprehensible chunks. InterCONS (Smith, 1990) is a visual data-flow language in which certain primitives are associated with interactors like buttons or sliders. A similar approach can be found in Prograph (Cox et al., 1989) and Fabrik (Ingalls et al., 1988). Fabrik enhances the traditional data-flow model with a bidirectional data-flow. This extension permits the use of nodes that combine several functions (typically a function and its inverse).

4 WHIZZ'ED

We implemented a visual programming tool called Whizz'Ed that allows the visual programming of highly interactive interfaces. The purpose of such a tool is to allow the creation (by direct manipulation) of interactive objects to build such an interface. This section is dedicated to the presentation of Whizz'Ed. We will present the conceptual model of Whizz'Ed, its functioning and its graphical representation.

4.1 The conceptual model

The conceptual model of Whizz'Ed is based on data-flows and a set of predefined elementary components. The editor is built on top of an underlying library called Whizz (Chatty, 1992) dedicated to the programming of highly interactive and animated interfaces. In order to explain the data-flow model of Whizz, we usually exploit a musical metaphor. In this metaphor, graphical objects are called dancers and non-graphical objects can be either instruments or tempos. Flows between these objects are made of simple pieces of data such as integers, colors, positions, etc. Referring to the musical metaphor, these pieces of data are called notes, and are produced by instruments. In order to allow communication, each object holds several input and output slots called plugs. Dancers listen to instruments and their graphical appearance (shape and aspect) changes according to the notes they hear. There are different kinds of instruments, some of which will be described in the next section. Among them, for instance, instruments called rotors, each producing positions bound on a circle, thus allowing dancers connected to them to move along that circle.

Time-based behaviours are implemented by tempos that produce notes called pulses at regular time intervals. Tempos can be used to drive instruments. For each beat of a tempo, the instruments connected to it produce a note, then received by the dancers connected to them. There is no one-to-one relation between instruments and dancer as several dancers can listen to the same instrument, and one dancer can listen to several instruments at a time. Tempos can also be shared amongst instruments. This makes it possible to easily describe synchronized behaviours. For instance, in a text editor, when a user depresses one of the arrows at the ends of a scrollbar, the cursor of the scrollbar moves while the text is scrolled. These two synchronized behaviours can be obtained by connecting the text and the cursor, which are two dancers, to two instruments that describe their movements, and that are connected to the same tempo. Finally, other sources of movement than tempos can be used. Special instruments spontaneously emit the successive positions of the mouse, or the characters typed on the keyboard. Other instruments behave as active values, and can be used to communicate between Whizz constructions and other pieces of software: they are variables, that can be modified in a program, as well as instruments, that emit their new values when modified. Flows from different sources can be combined, for instance when using several input devices at a time: Whizz has been successfully used to implement two-handed input (Chatty, 1994).

When building highly interactive or animated systems, specifying the movement of objects is not enough. It often happens that one wants to perform an action when an object has finished its movement, or when it passes a border, or even when it meets another object. Such events are generally difficult to compute beforehand, and it is much more pleasant to be notified when they occur. The model provides a number of active zones (or fields) in order to detect such events. They range from linear borders to elliptical fields or grids. They can emit events such as crossing, entering, leaving, etc. This allows for example the easy building of a labyrinth interface. Walls are built from these kinds of active zones and are automatically impassable by user's actions. Similar zones can be used for the building of a slider to represent the interface of a thermometer. Events may also be emitted by instruments when their part is finished, or when a particular time is reached. This enables us to use the event model that has proven to be useful for interactive applications.

The underlying data-flow model of Whizz is based on two types of information propagation: streams, used for evolutions which represent a continuous phenomenon and events, used for isolated evolutions. Events are similar to these used in many graphical toolkits and represent a less structured way of communication than streams. With this stream-event model, user's actions and animation can be mixed since the model has a unique information propagation scheme. Events may also be attached to callback functions that may reconfigure the flow graph, by creating or destroying bricks, or changing connections. This allows events to change at run-time the behaviour of an object, or to fire animations.

Whizz'Ed provides a kit of components that can be wired together to build new components. This technique is usually called the building game metaphor. This name comes from the Lego-Logo (Resnick, 1993) construction kit which is a rich construction environment which allows the building of creatures with electronic bricks like sensors, motors, lights, and-gates, flip-flops, etc. Several visual programming systems use this efficient building concept linked with a direct manipulation editor (for example see Fabrik (Ingalls et al., 1988)). Whizz'Ed provides a set of elementary bricks that can be dancers, instruments or tempos and allows the building of higher level bricks that can be seen as reusable components. When two bricks are graphically wired by the designer, these two bricks are connected and the data-flow is either created or dynamically reconfigured.

Graphically Whizz'Ed proposes a visual language representing objects (instruments, tempos and dancers) by icons that can be animated and data-flow by lines connecting these icons. Plugs are graphically represented by small rectangles coupled with the icon representing the object. The notes are put in these plugs, and the type of the note must correspond to the type of the plug. Graphically, the shape of the plug is different according to its type.

4.2 The interactive editor

Whizz'Ed consists of three main parts: the palette, the editing area and the simulating area.

- The palette contains the graphical representation of the objects (cf. Figure 1). This part is devoted to the elementary bricks that are supplied to the designer. The chaining of these elementary bricks allows to build complex behaviours avoiding the limitations introduced by the use of complex bricks. However it is possible for the designer to build compound bricks and to declare them as elementary bricks, thus increasing the set of bricks primarily proposed. The palette is divided into three parts: the set of instruments, the set of dancers and the set of other bricks that are only used by designers to relate instruments or dancers.
- The editing area allows to program the highly interactive or animated interfaces using elementary bricks (cf. Figure 1). The interface designer can use the palette (as in classical drawing tools) by direct manipulation, selecting an icon in the palette, dragging it to the working area and dropping it at the desired place. Data-flow between bricks is also built using direct manipulation, by selecting a plug of a brick and dragging and dropping it over another one. The arc is then automatically represented. The editor checks automatically that the type of the plugs are compatible. The direct manipulation of the bricks to assert relationships prevents many errors from the development process. The bricks can be moved to simplify wiring.
- The simulating area. This area aims at graphically representing the execution of the visual program built in the editing area (cf. Figure 2). The simulating area guards against structural errors since the building of interactive objects is immediately visible and the appearance and behaviour of these objects is directly simulated. This area can be used either for debugging, rapid prototyping or simulating the program. At this time, it is not possible to interact with the execution but it will be soon possible to act on the execution by direct manipulation. These interactions will be automatically analyzed by a module and reintroduced in the visual program, thus allowing designers to modify their program at run time.

The elementary bricks are the foundation of the building of components of the interface. Some bricks are computational, while others provide user interface functions. The designer defines his application by directly manipulating the iconic representations of these bricks. The designer selects iconic representations of appropriate bricks, places them in the editing area and connects them to achieve the objects with the desired functionality and appearance. This reusable composite brick is immediately provided to the designer in the tools area. From this state, the designer can use the new brick as a classical elementary brick. Some elementary bricks provide a specific editor to visually set some variables. We will not describe all elementary bricks that are provided by Whizz'Ed, but some of them will be detailed as they are used in the example of the next section.

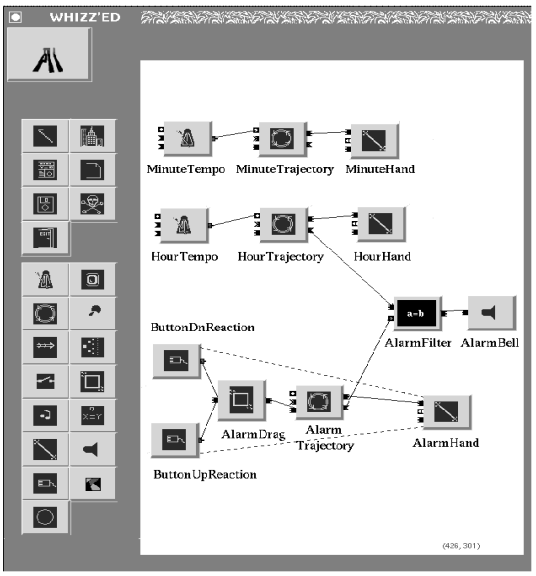

Figure 1: The Palette and Editing Areas.

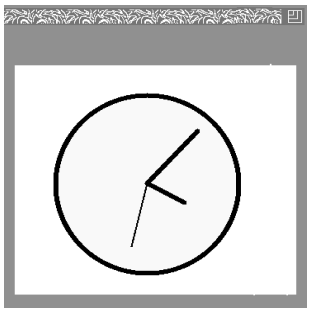

Figure 2: The Simulation Area

5 THE BUILDING OF AN ALARM CLOCK

We present in this section an example of the use of Whizz'Ed for building the behaviour of an alarm clock. The behaviour of such a clock is as follows: when both the alarm hand and the hour hand of the clock match, the bell rings. The user can directly manipulate the alarm hand of the clock in order to set it. The behaviour of the clock is quite complicated as it combines graphical interactions such as user's actions (the setting of the alarm hour), temporal aspects (clock hands moving according to time stamps) and semantical behaviour (the bell rings when the alarm hour is reached).

In spite of this relative complexity, modelling of this behaviour is quite simple, using the conceptual model of Whizz. The behaviour is described by connecting several elementary bricks together. This shows the

modelling power of Whizz as well as the efficiency of Whizz'Ed as a visual programming environment. First we will present the set of items (bricks and connections) of the visual program, then we will show how the behaviour of this program corresponds to the desired behaviour of the alarm clock.

5.1 The items of the visual program

The Reaction Brick A Reaction brick is reactive to user's actions (mouse click, mouse move, etc). It is associated with an object defined as being a sensor of the user's action and transmits this action to a connected object. These bricks are aimed to allow user's actions to interfere with the behaviour of the system. A Reaction brick has one output plug for sending a note to the connected object when a user's action is performed on the sensor object. The designation of the sensor object is achieved by a semantic connection represented by a virtual temporary link between the reaction brick and the object which will be used as sensor.

The Rotor Brick A Rotor Brick is a point instrument that sends positions on an ellipsis. The rotor has three input plugs used for the calculation of the next position and two output plugs. The first input plug called "Step" is used to receive pulses. Each time a pulse is received by this input plug, the rotor computes the next position on the ellipsis according to the previous one. The second input plug called "Random" aims at receiving pulses too, but the next position is computed randomly on the ellipsis. In both cases the computed position is sent to the output plug. The third input plug called "Projection" aims at receiving directly the next position on the ellipsis. One may visually specify the center and the radius of the ellipsis, the number of steps per turn, and the initial position (in term of steps). The elliptic trajectory of the rotor could be shown but it could be hidden if necessary by visually setting a parameter. The first output plug called "Position" is used to send the next position on the ellipsis to the connected object. The second output plug called "Parameter" is used to send the angle of the director vector.

The Tempo Brick Tempos are designed to synchronize animated actors. There is no synchronization between different tempos. The tempo features three input plugs: the first one (of type date) corresponds to the interval of time between two notes it must produce, the second one (of type integer) represents the number of notes it has to produce (these two parameters may be also directly visually specified) and the last one (of type boolean) represents its state: active (default) or idle. When an event is received, a tempo emits a note on its output plug.

The Segment Brick Segments are deformable segments. They have five specific input plugs. The Begin, End and CenterPosition that control the position of the begin and end extremities, and the center. The type of these plugs is a position type. The two other plugs concern the color of the segment, the first one is the BorderColor while the other one is the FillColor that control respectively the color of the border and the filling. The type of these plugs is RGB (three integers for the Red Green and Blue colors).

The Drag Brick This brick allows the drag of a graphical object. It has one input plug and one output plug. The input plug called BeginEnd (of type boolean) aims at receiving a pulse to begin or end the drag. When the drag is performed, the output plug emits the successive positions of the drag, that is to say the successive positions of the mouse while the user moves it.

The Bell Brick The Bell Brick emits a sound when a pulse is received on its input plug. One may visually specify the volume of the sound.

The Filter Brick The Filter Brick is used to check the equality between two values. It has two input plugs which get the values to compare, and an output plug to emit a pulse if the equality is checked.

5.2 The visual program of the alarm clock

The visual program presented in Figure 1 must be read as follows. The hour and minute hands of the clock are displayed using Segment bricks called HourHand and MinuteHand. Each of these hands move according to a trajectory defined by two elementaries Rotor bricks called MinuteTrajectory and HourTrajectory. The move to the next position of the hands is triggered according to temporal events produced by two Tempo bricks called MinuteTempo and HourTempo The HourTempo (resp. MinuteTempo) is connected to the input plugs of the HourTrajectory (resp. MinuteTrajectory) Rotor brick itself connected to the Segment brick HourHand (resp. MinuteHand). The alarm hand of the clock has to be reactive to the events mouse-down and mouse-up in order, for the user, to be able to drag it. This sensitivity to user's actions is managed by the elementary Reaction bricks called ButtonDnReaction and ButtonUpReaction. The dotted lines in the model represent the flow of events between the bricks responsible for the reaction to user's actions and the graphical object where this user's action can take place (the Segment brick called Alarm hand). The alarm hand can only be dragged inside the clock, the end of the alarm hand which is at the center of the clock is fixed and its outer end can only be moved according to a circular trajectory. In order to produce this drag we use the data flow. Thus, the output of the Reaction bricks are connected to the Begin/End input plug of the elementary Drag brick called DragAlarm. DragAlarm is connected to the input plug "Projection" of the elementary Rotor brick AlarmTrajectory (which compute the next position of the outer end of the alarm hand according to the current position of the mouse). The output plug "Position" of

AlarmTrajectory is connected to the input plug "End" of AlarmHand in order for the next position of the alarm hand to be displayed on the screen. The output plug "Parameter" of AlarmTrajectory, is connected to the first input plug of the elementary Filter brick called AlarmFilter while the output plug "Parameter" of the brick HourTrajectory is connected to the second one. This Filter brick will trigger the bell (represented by a brick called AlarmBell) when the value in each of its input plugs is the same.

Setting the alarm is done by direct manipulation when the user clicks on the alarm hand and drag it. At that moment, the Reaction brick ButtonDnReaction emits on its output plug a pulse towards the Drag brick DragAlarm to start the drag. This DragAlarm emits positions to the Rotor brick AlarmTrajectory which emits the new position of the end extremity of the alarm hand. Concurrently, the output plug "Parameter" of AlarmTrajectory emits the angle value to be compare to the Filter brick AlarmFilter. When the user releases the mouse button, the Reaction brick ButtonUpReaction emits on its output plug a pulse towards the Drag brick DragAlarm to stop the drag. Then the alarm hour is set. In parallel, the two Tempo bricks, MinuteTempo and HourTempo, emit pulses on their output plugs towards respectively the Rotor bricks MinuteTrajectory and HourTrajectory. MinuteTrajectory emits positions to the outer end of the minute hand. HourTrajector emits positions to the outer end of the hour hand and the angle value to the Filter brick AlarmFilter. If the two values received from HourTrajectory and AlarmTrajectory match, then AlarmFilter emits a pulse to AlarmBell.

This alarm clock can be encapsulated in a composite brick featuring a set of input and output bricks that can be directly manipulated by the visual programmer, and can be reused in other applications as the predefined elementary bricks.

6 CONCLUSION

In this paper, we have presented Whizz'Ed, a visual programming tool for highly interactive interfaces. Whizz'Ed is based on a data-flow model of Whizz. Whizz'Ed allows to describe the behaviour of graphical objects in a visual way through the use of a direct manipulation editor, with a building block metaphor. A kit of elementary bricks oriented toward user graphical interaction is provided. To illustrate our work, we have presented an example of an animated object built with a few elementary bricks.

ACKNOWLEDGEMENTS

The authors would like to thank the HCI group of the Department of Computer Science of the University of York (U.K.) where Ph. Palanque was a visiting researcher during the development of this paper. The authors also wish to thank Michelle Jacomi and the anonymous reviewers for their useful comments.

REFERENCES

Chang, S. K. (1987). Visual languages: A tutorial and survey. *IEEE Software*.

Chatty, S. (1992). Defining the behaviour of animated interfaces. In *Proceedings of the IFIP WG 2.7 working conference*, pages 95–109. North-Holland.

Chatty, S. (1994). Extending a graphical toolkit for two-handed interaction. In *Proceedings of the ACM UIST*.

Cox, P., Giles, F., and Pietrzykowski, T. (1989). Prograph: A step towards liberating programming from textual conditioning. In *IEEE Workshop on Visual Languages*, pages 150–156.

Harvey, N. and Morris, J. (1993). NL: A generic purpose visual dataflow programming language. Technical report, University of Tasmania, Australia.

IBM (1991). *Common User Access. Advanced Interface Design Guide*. IBM Corp.

Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., and Doyle, K. (1988). The Fabrik programming environment. In *IEEE Workshop on Visual Languages*, pages 222–230.

Maloney, J. H., Borning, A., and Freeman-Benson, B. N. (1989). Constraint technology for user-interface construction in ThingLab II. In *OOPSLA'89 Proceedings*, pages 381–388.

Myers, B. A. et al. (1990). Garnet, comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, pages 71–85.

OSF (1993). *OSF/Motif Programmer's Reference Release 1.2*. Prentice Hall Inc.

Resnick, M. (1993). Behavior construction kits. *Communications of the ACM*, pages 66–71.

Shneiderman, B. (1983). Direct manipulation: a step beyond programming languages. *IEEE Computer*, pages 57–69.

Smith, D. N. (1990). The interface construction set. In *Visual Languages and Applications*. Plenum Pub.

Stasko, J. T. (1989). *TANGO: A Framework and System for Algorithm Animation*. PhD thesis, Brown University.

Sutherland, I. E. (1963). Sketchpad: a man-machine graphical communication system. In *AFIPS Spring Joint Computer Conference*, pages 329–346.