Defining the dynamic behaviour of animated interfaces

Stéphane Chatty

Centre d'Études de la Navigation Aérienne, 7 avenue Édouard Belin, 31055 TOULOUSE CEDEX, France. Phone: +33 62 25 95 42. Email: chatty@dgac.fr.

Abstract

This paper presents Whizz, a system for building animated interactive applications. We describe its musical metaphor, and its underlying model based on streams and events. We analyze the notion of animation, and show how the Whizz model can be applied to the whole dynamic behaviour of an interface, including animation, user input, and communication with the application. Such an integration leads to a homogeneous definition of that behaviour, and to interactive tools to define it. It also opens doors on new types of interaction that mix users' actions with animation.

Keyword Codes: H.5.2; D.1.7; I.3.6
Keywords: User Interfaces; Visual Programming; Computer Graphics, Methodology and Techniques.

1. INTRODUCTION

If developing the static presentation part of an interface was once a hard task because of the lack of tools, it is no longer the case. The main problem now consists in choosing the right toolkit and the right interface editor.

On the other hand, there is still little support for the dynamic part of an interface, and especially animation. Animation appeared long ago in the field of user interfaces, but animation systems are generally dedicated to simulation or some kind of algorithm animation. If you want to add motion to an interface, you have to program it.

However, animations can be useful in many kinds of interfaces. Baecker et al. [1] showed how animated icons can help understand the usage of a tool in a drawing system. The use of animation in help systems becomes increasingly popular, but it is not the whole thing. Animation can be used as a part of the data manipulation interface as well. Robertson et al. [14] assess that using an animation instead of an instant switch from one state of the display to another state transfers mental activity to the perceptual system and decreases mental load.

In fact, animation is already used in real systems. For instance in the Macintosh Finder, when double-clicking on the icon of a folder, a rectangle grows from the size of the icon to the size of the window being opened. The blinking of the cursor

in text editors is a simple animation, aimed at directing the user's attention. But such animations have to be programmed with ordinary drawing instructions and very little support for time management. The technical problems are often difficult to overcome. One can imagine that animations would be used more often if they could be programmed more easily.

This paper presents Whizz, a system which provides the integration of animation with the usual mechanisms of an interface. It allows a uniform manipulation of animation, user input and data visualization. Furthermore, its model allows the development of a direct manipulation editor for the definition of the dynamic behaviour of an interface. A prototype of such an editor has been implemented and is described at the end of the paper.

2. RELATED WORK

Animation is mainly a research theme of computer graphics. But in that domain, the interest is much more in animation rendering than in interaction or programming techniques. It is only recently that some thought has been given to these topics [18].

In the field of user interfaces, animation is most often seen as a means of presenting an algorithm or the execution of a program. Balsa [4] is the best known algorithm animation system. Pastis [12] is a recent program animation system based on a debugger. However, such systems do not directly address the issue of building animated interfaces.

There have been few attempts at providing animation toolkits. Animus [9] is one of them. It uses a constraint system to maintain the consistency across time between the display and an underlying physical model. Tango [15] is another system which implements a path-transition paradigm. One describes paths, attaches graphical objects and then plays the resulting transitions. Tango comes with an associated direct manipulation editor, Dance, which allows the creation of transitions.

A part of the stream-event model of Whizz is similar to data-flow models. Such models have already been used for the architecture of interactive systems [8, 16]. Fabrik [10] is a visual language with a data-flow model, which can be used for building interactive applications. Squeak [5] uses communicating processes to describe the processing of complex user input issued from several devices. Constraint-based systems such as ThingLab II [11] or Garnet [13] also have an underlying data-flow model induced by constraints.

Animation programming has strong similarities to other real-time applications, and particularly to music synthesis. There has been much more work in that domain, and the temporal issues generally have been addressed more thoroughly. Formes [6] is one of the best known music synthesis systems. It features a hierarchical structure of processes, used to describe the structure of music, which includes sequential, parallel, and nested patterns. Formes was also used to describe other dynamic processes, such as graphical display. Arctic [7] is a functional language which uses an event flow model to describe real-time processes. Esterel

[3] is a general purpose synchronous language, used to describe all kinds of reactive systems, among which are dialogues in interfaces.

3. A MUSICAL METAPHOR

In the same way as users of interactive applications, programmers build their own mental model of the toolkit they are using. This is especially true of object-oriented toolkits. The underlying model of such toolkits is often very abstract in order to provide power of expression and flexibility. But it is useful to provide a simple conceptual model in order to facilitate comprehension. For that reason, we will first present the conceptual model of Whizz, before detailing the underlying, more abstract and more general model.

3.1. Dancers, Instruments, Rhythms, Tempos

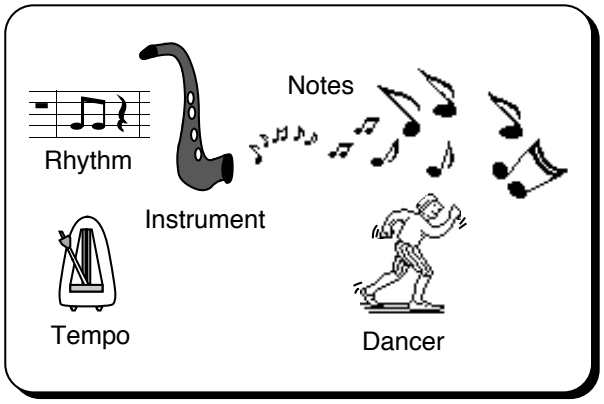

Figure 1. The musical metaphor of Whizz.

The basic conceptual model of Whizz uses a musical paradigm, summarized in Figure 1. It involves dancers, instruments, notes, rhythms and tempos. *Dancers* are the entities which produce the perceptible effects. They generally are graphical objects capable of movement or deformation. They move or change shape according to the *notes* they hear. Notes are a generalization of musical notes. They carry simple pieces of information such as points, colours, numbers, or even real musical notes. *Instruments* play these notes according to their own characteristics and state. Some instruments emit positions along a trajectory. Others emit numbers read in a file, or computed with an algorithm. They play those notes at dates determined by a tempo and a rhythm. The *tempo* determines a series of dates at regular intervals. The *rhythm* determines for which of these dates the instrument must play a note.

There is no one-to-one relation between instruments and dancers. Several dancers can listen to the same instrument, and one dancer can listen to several instruments at a time. In the latter case, the musical paradigm is a little twisted:

generally all the instruments of an orchestra play related parts, and dancers listen to the resulting music. In our case, dancers most often listen and react independently to each instrument. For instance a rectangle may move according to the positions emitted by an instrument, and change colour according to another instrument.

Tempos and rhythms can also be shared amongst instruments. Two instruments having the same rhythm and tempo will play their notes at the same time. More complex synchronization can be achieved with different rhythms and a shared tempo. For example, an icon representing a ball can move on every beat of a tempo and rotate on one beat out of three. Whereas using the same tempo allows synchronization, using tempos with different intervals allows one to express phenonema with different time scales. For instance, the smooth movement of an icon and the slow blinking of a cursor do not need the same time resolution.

The most useful instruments for graphical animations are instruments that produce points. Whizz provides paths defined on a point-per-point basis, as in Tango. It also provides more abstract movements such as circular or linear trajectories. That approach is similar to that of object-oriented graphical toolkits, which provide graphical objects like rectangles or lines. More sophisticated movements can also be added. In its current version, Whizz contains attractors: they are movements which bring an object to a desired position, whatever its initial position and speed. It also has smoothly decelerated movements and oscillating movements as produced by springs. As for graphical toolkits, it seems that a few basic movements are sufficient to describe most interfaces.

As stated before, the main dancers are graphical objects. They have a number of input slots depending on their nature. For instance a rectangle dancer may change its position, its corners, its border colour or its fill colour. A polygon dancer has an additional input slot to add new vertices. Some special dancers are made of several graphical objects (generally icons), of which only one is visible at the same time. They have an input slot that makes it possible to change the visible object. Those polymorphic dancers are useful to implement traditional animation, performed by a simple succession of images.

3.2. Unexpected events

When building an animated interactive system, specifying the movement of objects is not enough. It often happens that one wants to perform an action when a dancer has reached the end of a trajectory, or when it passes a border, or even when it meets another dancer. Such events are generally painful to compute beforehand, and it is much more pleasant to be notified when they occur. Furthermore, they are impossible to foresee when the movement of dancers depends on the user's actions.

Whizz provides a number of *active zones* (or fields) in order to detect such events. They range from linear borders to elliptic fields or grids. They can emit events such as crossing, entering, leaving, etc. Similar events may also be emitted by instruments when their part is finished, or when a particular time is reached. This allows to use the event-based structure of programs that has proven to be useful for interactive applications.

We have just described the musical metaphor of Whizz, which features tempos, rhythms, instruments and dancers, with an additional notion of unexpected event. We will now describe the more general model that underlies this metaphor.

4. THE STREAM-EVENT MODEL

The underlying model is based on two types of information propagation: streams and events. This stems from the fact that dynamic processes, and especially graphical animation, have two modes of evolution. The first are evolutions which represent a continuous phenomenon. Such evolutions are indeed discretized because of the constraints of digital computing, but it is useful to keep their continuous nature in mind. A good example of a continuous evolution is the movement of an object along a path. But there are also sudden evolutions, such as a light turning from red to green, or the disappearance of an object. Such evolutions can be represented as special cases of continuous evolutions (with only one significant step), but it is generally awkward. It is much simpler to manipulate them as small and independent pieces of information. Such a distinction between continuous and instantaneous information has sometimes been made for input devices [17]: buttons produce instantaneous information, whereas a mouse produces a flow of positions.

Whizz models continuous evolutions with a data-flow model. Streams of data (the notes) are emitted and filtered by *modules* organized in a graph. Every module has a number of input and output *plugs* that can be dynamically connected to other plugs. Every plug has a type that describes what kind of data can circulate through it, and can be connected only to compatible plugs. The tempos, rhythms, instruments and dancers that were previously described are such modules. Tempos are special modules which emit notes at regular intervals, initiating the propagation of information. They are the only parts of Whizz that have to take real-time into account. All the subsequent propagation and handling of notes are supposed to be instantaneous compared to the interval between two notes. The other modules act as filters: rhythms let only certain notes through, instruments enrich them, and dancers handle them and move.

Modules can be grouped in *scenes*. A scene is a group of inter-connected modules; it is itself a module, and it may export some of its modules' plugs. This mechanism allows to build and reuse parts of interfaces. Scenes can also be stored in files and retrieved, which makes it possible to design interactive scene editors.

Instantaneous evolutions are modelled by events, similar to those used in many graphical toolkits. They represent a less structured way of communication than streams. Events are mostly characterized by their types, in the same way as input events in window systems. Events are emitted by various sources, such as modules or active zones. They are caught by modules, which have predefined reactions such as self-destruction. They may also be attached to callback functions, like in traditional user interface toolkits. One may define callback functions that instantiate and connect modules, so that events result in the beginning of new animations.

We believe that this communication based on streams and events generalizes the architecture that is proposed by most user interface toolkits. We will now show that at least, it allows a good integration of animation with usual user interface technology.

5. ANIMATION AND INTERFACE BEHAVIOUR

The word *animation* is often used without any precise definition. In fact its meaning depends on the context. From a perception point of view, we say that something is animated when its appearance changes with time. The animations that are generally studied are implemented as explicitly time-based processes, like the animations described earlier in this paper. But the same perceptive effects can be achieved differently. The active values used in several UIMS offer an animated visualization of variables. Their evolution is controlled by the application execution and not by a clock in the interface, but it makes no difference to the user. Similarly, a user dragging an icon with a mouse is achieving a certain kind of animation. The technical issues are the same, and even the perceptive effect is the same when several users look at the same display.

Consequently, we consider that the interface behaviour can be described in three categories :

- time-driven animation
- application-driven animation
- user-driven animation

The stream-event model covers all three cases. Regarding the stream-based communication, one only needs to provide special modules attached to input devices or application data. They can be connected to the modules described above, and produce the same animations. Whereas tempos initiate the propagation of information by impulses that need to be enriched by instruments, those modules initiate it with more significant information. For instance, Whizz has a module for the mouse, which emits points and can be connected to a dancer in order to move it. Active values are also provided: they are modules that emit integers, for instance. As in Fabrik, filters (instruments) may be used to transform integers into points, and produce various data representations, like the speedometer of Figure 2.

The event-based communication is also useful for the integration of our three types of animation. Synthetic events caused by time-based animations and user-input events are naturally handled the same way: they are both very similar to the traditional notion of event. The notion of event from the application is less traditional, but it proved to be useful. Balsa and other algorithm animation systems use the notion of "interesting event". Interesting events correspond to key steps in the execution of the algorithm, where an animation should be performed. They are generally associated to annotations in the source code of the animated program, and emitted when the annotations are encountered.

We believe that events from the application can be used in more general circumstances. For instance, they can be used in any interface that needs to visualize the

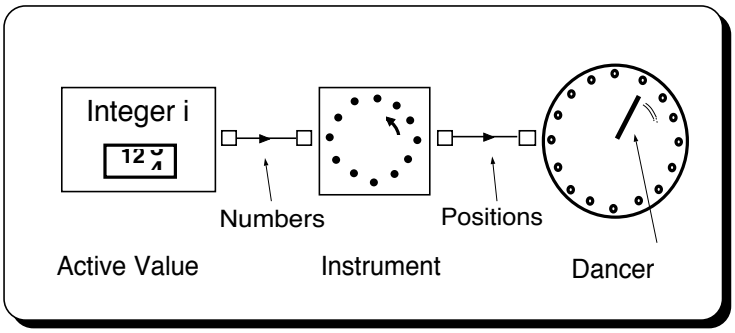

Figure 2: A speedometer connected to an integer. The instrument transforms the integer notes into positions, and emits them to the input slot that controls the extremity of the hand.

operations performed in the application, and not only data. It is especially true of animated interfaces, where animations can be attached to operations, in the same way as representations are attached to data. Let us consider an iconic interface for a file system. A file is represented by an icon. An operation on such a file can be represented by an animation of that icon. For instance, if the file system sends an event to notify the destruction of the file, the icon could blow up and vanish.

6. MIXING USER INPUT WITH ANIMATION

The integration of the different kinds of dynamic behaviour in Whizz makes it possible to mix them in the same interface. Such a mixing is often difficult to achieve. For instance, most algorithm animation systems freeze user input while they are animating objects. With the stream-event model, the mixing is natural because user input and animation are parts of a unique information propagation scheme. Some bursts of information propagation are initiated by tempos, and some others by the user's actions.

That interleaving of user's actions and animation has other consequences. It implies that the paradigm of playing predefined scenes is no longer valid. For instance, it is impossible to foresee the moment when an object will cross a border, because the user's actions may modify its evolution at any time. This is why Whizz provides synthetic events such as border-crossing. The evolution of dancers is partly independent from the programmer, in the same way as the user is.

Games are good examples of applications that make use of animation and user input. A prototype of a breakout game has been implemented with Whizz, and is fast enough to be played on a standard workstation. In that game, the ball is a dancer connected to an instrument managing a trajectory, and to a tempo. The paddle is also a dancer, connected to a module emitting the position of the mouse. Collisions between the paddle and the ball produce events, as well as collisions between the ball and the bricks, which are active zones.

6.1. An air traffic control example

We will now detail an example from Erato, an experimental project aimed at improving the services offered by the French air-traffic management systems. In the current system, planes are represented as glyphs on the controller's screen, and they move according to the information obtained by radars. Using this representation, controllers have to avoid collisions. A new service offered by Erato consists in extrapolating the future trajectories of planes, so that the controller may detect problems earlier.

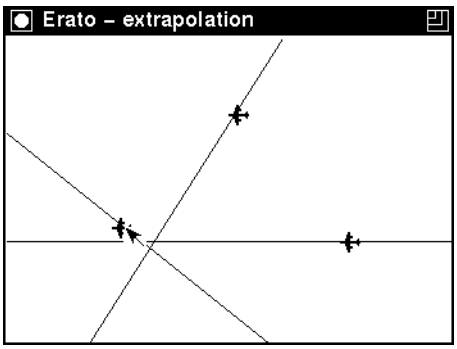

Figure 3: The air traffic control example. A plane is dragged to the possible point of collision, and the position of other planes is extrapolated.

The interface to that service works as follows. During normal operation, planes move regularly along their trajectories. When the controller wants to figure out whether two or more planes are a potential problem, he picks one plane and drags it along its future trajectory. The other planes will then move as though time was driven by the dragged plane. Possible collisions are thus directly visible (Figure 3).

Let us now detail the realization of that interface with Whizz. Every plane is represented by a scene made of a graphic dancer and a "glider" instrument (Figures 4 and 5). Both modules are predefined ones. The glider manages a straight trajectory. It has input plugs for stepping, random access by a numeric parameter, and random access by a position. Notes emitted through the first plug make the glider emit successive positions on the trajectory. A number passed through the second plug is considered as a number of steps from the initial point. A point passed through the third plug make the glider emit the projection of that point onto the trajectory. The glider has an output plug that emits the current position and one that emits the corresponding number of steps. The position output is connected to the position input of the dancer. The numeric output of the glider is exported by the scene, as well as its three inputs.

We can now examine the structure of the interface (Figures 6 and 7). Two situations are possible. During normal operation, all planes are connected to a tempo through their "step" input. When a plane is picked, an event is emitted, and the connections are changed: the dragged plane is connected to the mouse module through its "projection" input plug, so that it will stay on its trajectory

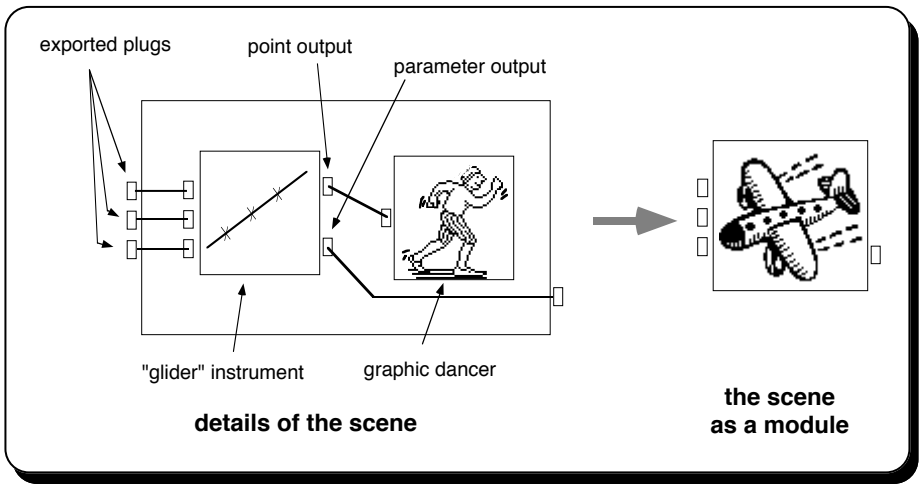

Figure 4. The structure of the "plane" scene.

```
/* A plane is a scene composed of a dancer and an instrument. */
/* It exports three input plugs and one output plug. */
class Plane : public Scene {
   Dancer* Image;
   Instrument* Speed;
   InPlug Step, Random, Projection;
   OutPlug OutParameter;
   OutPlug* NormalSource;
};

/* Constructor of planes. */
Plane :: Plane (Point& initial, Point& speed) {
   Image = new Polymorph;
   /* The polymorphic dancer has only one appearance. */
   Image->AddAppearance (new Icon ("plane"));
   Speed = new Glider (initial, speed);
   /* The positions of the dancer are given by the instrument. */
   Image->Position.Connect (Speed->OutPosition);
   /* Now export the plugs of the instrument. */
   ExportInPlug (Speed->Step, Step);
   ExportInPlug (Speed->Random, Random);
   ExportInPlug (Speed->Projection, Projection);
   ExportOutPlug (Speed->OutParameter, OutParameter);
}
```

Figure 5. The definition and initialization of a plane.

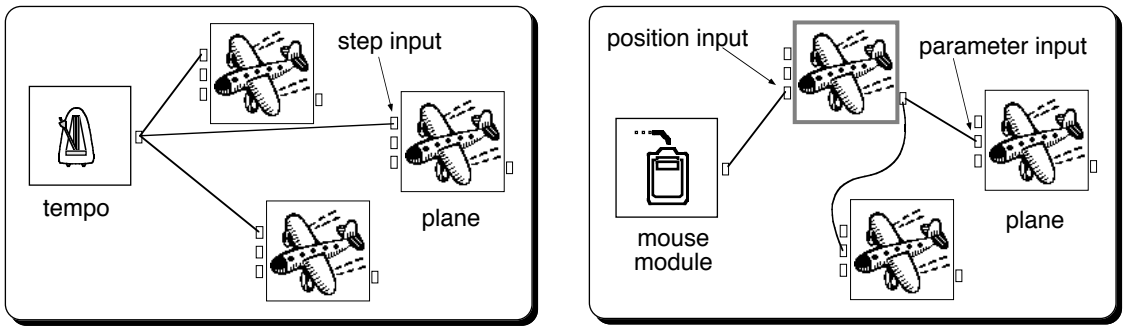

Figure 6: The structure of the flow graph during normal operation, and when a plane is dragged.

```
/* When a plane is connected to a tempo, */
/* the output plug of the tempo is stored for future reference. */
Plane :: ProceedWith (Tempo& t) {
   Step->Connect (t.Out);
   NormalSource = &t.Out;
}

/* When dragging begins, the normal source is disconnected */
/* and the plane is connected to the mouse module. */
Plane :: BeginDrag (MouseModule& d) {
   Step->Isolate ();
   Projection->Connect (d->OutPosition);
   /* Then, all other planes are connected to this one. */
   ListIterator other_plane (AllPlanes);
   while (++other_plane)
      if (other_plane != this) {
         other_plane->Step->Isolate ();
         other_plane->Random.Connect (this->OutParameter);
      }
}

/* When dragging ends, all planes are connected back to their normal source. */
Plane :: EndDrag () {
   ListIterator any_plane (AllPlanes);
   while (++any_plane) {
      any_plane->Random->Isolate ();
      any_plane->Step.Connect (any_plane->NormalSource);
   }
}
```

Figure 7: The possible actions on a plane: make it move with a tempo, or drag it. When it is dragged, all the other planes are moved accordingly.

while moving. All other planes are connected to the dragged plane through their numeric input plug, so that they keep at pace with it. When the plane is released, all connections are restored to their previous states.

6.2. Throwing icons

The mixing of animation and user input can lead to unusual kinds of interaction. One can imagine an iconic interface where one could throw icons into the trash can. Such an experiment is feasible with our model. One just needs to handle the event emitted by the drag module when dragging is over, and to connect an instrument and a tempo in its place (Figure 8). Depending on the type of the instrument, the

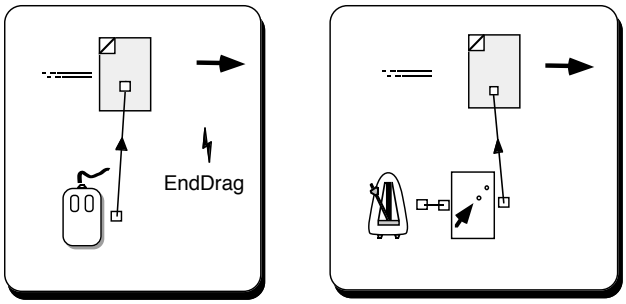

Figure 8: Throwing an icon. When dragging is over, the motion is continued as a time-driven animation.

dragged object will slow down until it stops, for instance. If it is thrown in the right direction, the object may enter the field of the trash can as in Figure 9. That will produce an event, and the icon will switch from its decelerated movement to an attraction towards the trash can. The last event will then be the collision, and the object will be destroyed. The interaction technique demonstrated by this example

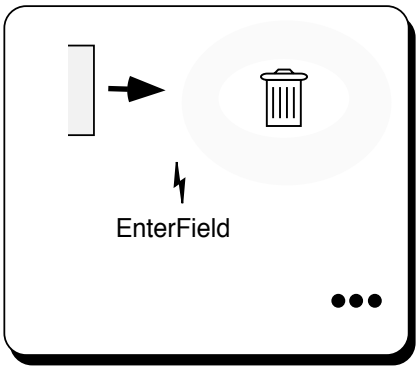

Figure 9. The icon is captured by the trash can.

is rather new. We believe that it can be useful in the same way as animations can be useful for data visualization.

7. A DIRECT MANIPULATION TOOL

Whizz was designed and implemented to allow the building of direct manipulation tools for the definition of animated interfaces. We present here the prototype that has been implemented. The purpose of such a tool is to allow the creation of scenes. Scenes are resources that can be stored in files, retrieved, and instantiated. This allows us to design animations and graphical representations independent of the application, in the same way as most interface builders do. One only needs to load a scene from a file, instantiate it, and establish the necessary connections with existing modules.

The data-flow paradigm is well suited for that purpose. Firstly, the paradigm of connecting modules is easy to map onto a graphical interaction. Another reason is that it allows to split the editor in a number of specialized editors for each type of module. For instance, the rhythms of Whizz are defined with a specific editor similar to bitmap editors (see Figure 10). Specific editors are also provided for all

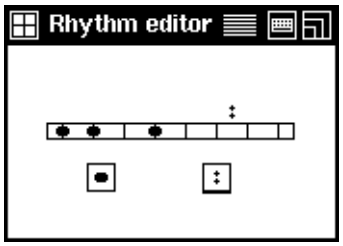

Figure 10: A simple rhythm editor, hybrid of musical scores and bitmap editors. Each vertical bar represents an instant. When a black dot is present, a beat will be emitted at that instant. The colon indicates a replicated pattern. The rhythm being edited will emit beats at instants 0, 1 and 3 and loop at instant 5.

kinds of instruments.

The main part of the editor shows a scene in a window. The representation is organized in layers superimposed like transparencies. The first layer contains the graphical objects that will appear in the final application. They may be ordinary static objects, but they are generally dancers. Another layer contains the representation of the modules and their connections. Finally a third layer visualizes active zones and the trajectories generated by instruments Figure 11 represents a simple scene being edited, with all layers visible. That scene is a standalone time-based animation. When it is activated, the rectangle and the end of the segment will begin a motion along a complex trajectory made of the composition of two rotations.

The editor allows the testing of animations while they are being built. It has been constructed with Whizz, and all icons representing modules indeed have a real module attached to them. When the editor is set in test mode, the modules are activated: notes are emitted by tempos, filtered by other modules, and handled by dancers, in the same way as they will be when the scene will be used. An additional feedback of the activity of the modules is provided. The icons representing modules

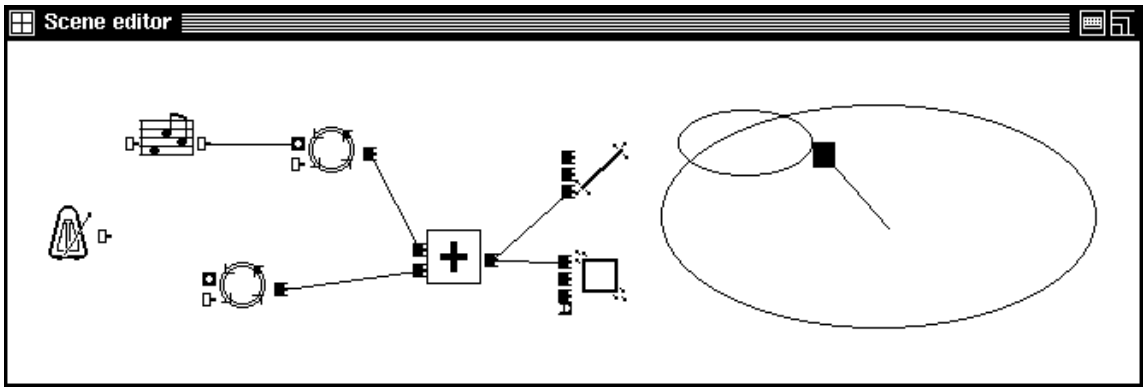

Figure 11: The creation of a scene with the direct manipulation editor. On the left, the icons represent modules. Plug types are visualized by different shapes. From the left to the right are a tempo, a rhythm, two elliptic trajectories, an adder, a segment and a rectangle. The segment is connected by a plug representing one of its ends, and the rectangle by the plug representing its position. On the right, the rectangle, the segment and the two trajectories are visible.

are implemented as dancers with several images, and they are animated by the notes their associated modules receive. With that feedback, animation designers are provided with a visualization of the working of the animation they are building. It has also proved to be useful for understanding the underlying model of the system.

8. EXTENSIBILITY

The stream-event model is extensible. New functionalities can be added by programming new types of modules, which may immediately be used in association with the predefined ones. For instance, the rhythm modules implemented in Whizz are satisfactory for simple graphical animation, but they certainly would not be enough in the eyes of a music composer. One only needs to program a more sophisticated class of rhythms, and the music composer will be able to use them instead of the standard rhythms.

Extensions to new dynamic processes can also be added with new types of notes and modules to manipulated them. Such extensions may be useful for multi-media interfaces. For instance, musical notes and the instruments that produce them have been implemented in Whizz. They make it possible to add simple sounds or tunes to interfaces. For instance, one can link the pitch of emitted notes to an integer active value, providing an audio feedback of that value.

9. STATUS AND FUTURE WORK

A first version of Whizz was implemented in C++ as an extension to the X_{TV} toolkit [2], on top of the X Window System. It is currently used for experiments on animation-based interaction techniques, and interface architectures based on the stream-event model. It is also used as the data presentation module for our graphical debugger Witness. In that debugger, the graphical representations of variables are designed by end users with the animation editor, and the operations on those variables are also represented by animations designed with the editor.

Future work will take several directions:

- Refinement of the interactive editor. The present prototype has proved to be useful, but it also has evidenced the need of specific presentation paradigms, that still are to be defined.
- Extensions to be able to define whole interfaces with Whizz and its editor. Our next goal is the user being able to modify variables in Witness through actions on their representations.
- Integration with constraint specification, so as to be able to declare properties that should be kept throughout movements.

10. CONCLUSION

We have presented Whizz, a system that allows the building of animated interfaces. It contains a number of novel features, including the expression of time patterns and synchronization, objects that describe abstract trajectories, and events generated by the animated objects. Furthermore, its stream-event model offers the integration of animation with data visualization and user input, allowing an homogeneous definition of the whole dynamic behaviour of an interface. Whizz and its associated editor have demonstrated the possibility to use animation in user interfaces. They have also demonstrated that the behaviour of an interface can be defined with interactive systems similar to those used in the definition of presentation layers.

Acknowledgements

This research work was done while at the Laboratoire de Recherche en Informatique (University of Paris Sud and CNRS/URA 410). Many people have read and commented this article. I especially wish to thank Michel Beaudouin-Lafon, Thomas Baudel and Jean-Daniel Fekete for stimulating discussions. Heather Sacco, Chris Weikart and Balachander Krishnamurthy proofread various versions of it.

11. REFERENCES

[1] R. Baecker, I. Small, and R. Mander. Bringing icons to life. In *Proceedings of the ACM*

CHI'91, pages 1–6. Addison-Wesley, May 1991.

[2] M. Beaudouin-Lafon, Y. Berteaud, and S. Chatty. Creating direct manipulation interfaces with X_{TV}. In *EX'90, London, England*, 1990.

[3] G. Berry, P. Couronné, and G. Gonthier. The Esterel synchronous programming language and its mathematical semantics. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency, LNCS 197*, pages 389–448. Springer-Verlag, 1985.

[4] M. H. Brown. *Algorithm Animation*. PhD thesis, Brown University, 1987.

[5] L. Cardelli and R. Pike. Squeak: A language for communicating with mice. In *Proceedings of the ACM SIGGRAPH 1985*, pages 199–204, July 1985.

[6] P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the ACM Conference on Lisp and Functional Languages*, 1984.

[7] R. B. Dannenberg. Arctic: A functional language for real-time control. In *Proceedings of the ACM Conference on Lisp and Functional Languages*, pages 96–103, 1984.

[8] J. F. DeSoi, W. M. Lively, and S. V. Sheppard. Graphical specification of user interfaces with behavior abstraction. In *Proceedings of the ACM CHI'89*, pages 139–144. Addison-Wesley, 1989.

[9] R. A. Duisberg. Animated graphical interfaces using temporal constraints. In *Proceedings of the ACM CHI'86*, pages 131–136, 1986.

[10] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, and K. Doyle. Fabrik: A visual programming environment. In *OOPSLA'88 Proceedings*, pages 176–190, Sept. 1988.

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

[12] H. Müller, J. Winckler, S. Grzybek, M. Otte, B. Stoll, F. Equoy, and N. Higelin. The program animation system PASTIS. Technical report, Universität Freiburg, Institut für Informatik, 1990.

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

[14] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of the ACM CHI'91*, pages 189–194. Addison-Wesley, May 1991.

[15] J. T. Stasko. Using direct manipulation to build algorithm animations by demonstration. In *Proceedings of the ACM CHI'91*, pages 307–314. Addison-Wesley, May 1991.

[16] K. Tatsukawa. Graphical toolkit approach to user interaction description. In *Proceedings of the ACM CHI'91*, pages 323–328. Addison-Wesley, 1991.

[17] J. U. Turner. A programmer's interface to graphics dynamics. In *Proceedings of the ACM SIGGRAPH 1984*, volume 18, pages 263–270, July 1984.

[18] R. C. Zeleznik and al. An object-oriented framework for the integration of interactive animation techniques. *Computer Graphics*, 25(4):105–113, July 1991.