HandMove: a system for creating animated user interface components by direct manipulation

D. Vodislav
Conservatoire National des Arts et Métiers/CEDRIC
292, rue St-Martin, 75141 Paris Cedex 03, France, vodislav@cnam.fr

Abstract
We describe **HandMove** (**H**uman **AN**imation by **D**irect **M**anipulation of **O**bjects and **V**isual **E**lements), a highly interactive system for building animated scenes by direct manipulation. Its underlying model is based on concurrent evolution of graphical objects, position and attribute constraints, trajectory-based motion, event synchronization. Animation may be produced by time signals, user input or application values. Our objective is twofold: first, to present an animation model allowing intuitive and simple descriptions of complex animated scenes without textual programming; next, to integrate the resulting animation as dynamic elements in user interfaces built with a UIMS (User Interface Management System).

Keywords
Animation, user interfaces, direct manipulation, demonstrational techniques, constraints

1 INTRODUCTION

Classical computer animation systems use several methods to produce animation: creation of consecutive frames, behavior simulation for dynamic systems described by equations, etc. Usually they ignore user interface aspects and focus more on rendering problems, keyframing, object deformation, forward and inverse kinematics of articulated structures (Watt, 1992).

Animation was first introduced in user interfaces to visualize the dynamic behavior of the underlying application and appeared as a very suitable way to convey information. By presenting a continuous evolution instead of switching from one configuration to another, animation shifts a user's task from cognitive to perceptual activity and consequently decreases mental load (Robertson, 1993).

From a user interface perspective, animation raises two complementary issues: how to interactively describe the evolution of graphical objects (structure, relationships, motion laws) and how to use resulting animated scenes as user interface components (communication with other interface elements, with the application, with the user).

Several approaches were used to specify animation in user interfaces. Algorithm animation visualizes a program execution by representing data structures as graphical objects and data manipulation as object movements. Balsa (Brown, 1988) is a well-known algorithm animation system based on libraries of graphical objects and predefined movements. Tango (Stasko, 1991) uses a graphical editor (Dance) to create animation routines by specifying motion paths as a sequence of locations. However, algorithm animation, as most data visualisation systems controlled by the application, does not consider user interaction or building standalone user interface components.

Whizz (Chatty, 1992) creates animated modules controlled by three types of input signals: time, application values and user input. Using a musical metaphor, Whizz separates graphical objects ('dancers') from their motion laws ('notes' played by 'instruments') and represents animated scenes as data-flow graphs. Hudson and Stasko (Hudson, 1993) extend the Artkit user interface toolkit to include animation support. They improve the path-based model of the Tango system and create time-based animation modelled by 'transitions' (defined by a graphical object, a time interval, a trajectory and a pacing function).

Graphical constraints specify permanent relations between the objects in a scene. They offer a simple and declarative way to define dependencies to be maintained by the system at run-time, speeding up the prototyping process. Juno (Nelson, 1985) and Geometer's Sketchpad (Jackiw, 1993) use constraints to describe the structure of geometric figures. Animus (Duisberg, 1987), based on the Thinglab constraint management environment (Borning, 1986) uses static, time function, time differential and trigger constraints to define animation.

Direct manipulation and demonstrational techniques (Cypher, 1993) allow visual interaction between the user and the graphical representations of objects, defining their structure and behavior without textual programming. If direct manipulation interfaces for animation specification already exist (Chatty, 1992)(Stasko, 1991), programming by demonstration is more difficult to use in this field. Demonstration is very appropriate for repetitive user centred tasks such as graphical editing or text formatting, but specifying animation has a more imperative style and concerns time-dependent relationships; in conclusion animation is hard to demonstrate. However, we think various demonstrational techniques may be used in animation programming, by example for inferring position, attribute and time constraints.

To date, there has been little support for animation in user interface builders. Current UIMS create static user interfaces by interactive techniques, but in most cases illustrating the application's dynamic behavior needs significant programming. In order to reduce such programming effort, the user should be able to describe animated elements by direct manipulation and to build the user interface by combining static and dynamic components.

We consider that a system which creates animated components for user interfaces must fulfill the following requirements:

- It should be based on various animation sources in user interfaces: time, application values, user input. By mixing animation input stimuli, resulting scenes may be adapted to different application types (time-driven animation, data visualization, interactive games, etc).
- It should provide a rich scene model, able to describe object structure, inter-object relationships, various evolution laws, synchronisation, etc. Features not supported by the model usually need an important programming effort.

- It should avoid classical programming, by using direct manipulation for all scene definition aspects. Non-skilled programmers should be able to use the system with little effort. Resulting scenes must have concise but meaningful graphical representations and enable the user to understand and modify the scene's structure and behavior.
- It should enable an easy integration of animated scenes in user interfaces, as independent modules having inputs, outputs and internal dynamics.

Existing animation systems do not satisfy (totally or partially) some of these requirements. Systems such as Balsa (Brown, 1988) or Tango (Stasko, 1991) focus on a specific aspect (algorithm animation) and do not consider user interface interaction. Other systems have no support for advanced features such as graphical constraints (Whizz (Chatty, 1992)) or synchronization (the Artkit extension (Hudson, 1993)). A system such as DataViews (from V.I. Corporation) cannot express synchronization or user-driven animation by direct manipulation, even though it offers the programming support to realize them.

This article presents HandMove, an interactive system for creating animated scenes to be integrated in user interfaces. HandMove was conceived to fulfill the four requirements above. Its scene model allows composed objects related by position and attribute constraints, with concurrent evolution based on abstract trajectories and synchronized by events. Animated scenes are adapted to time, application values and user input stimuli. The interaction model is simple and intuitive; based on direct manipulation, it does not use textual programming at all. Resulting scenes are independent modules, used by a UIMS as ordinary widgets. An implementation using the XnslDraw (NSL, 1994a) graphical widget (created by NSL) and integrating HandMove in XFaceMaker (NSL, 1994b) (the NSL's X/Motif UIMS) is currently in progress.

2 THE HANDMOVE SYSTEM

The architecture of the HandMove system is presented in Figure 1. The HandMove editor enables easy creation of complex animated scenes by direct manipulation. Roughly, a scene is a set of graphical objects (actors) with motion laws based on various input stimuli (time, application values, user input) and synchronized by events. A declarative scene description language called HandScript is used to express resulting animated scenes and to store them in external files.

The scene compiler translates HandMove scenes into the target language of the XnslDraw widget (NSL, 1994a), which manages graphical objects with dynamic behavior described by scripts. Resulting scenes are used as dynamic user interface components, integrated in applications built with the XFaceMaker X/Motif UIMS (NSL, 1994b).

It is not our intention to build whole interfaces with HandMove, but only dynamic parts that otherwise need programming. We think that current UIMS manage well enough static interface design, with the advantage of respecting standard graphic styles (e.g. OSF/Motif (Open Software Foundation, 1991)). HandMove scenes may be used by an external UIMS, while in most current systems, resulting animation can only be used inside the system. In order to integrate the animated scenes into user interfaces, such systems are forced to offer full interface design capabilities.

HandMove significantly reduces the programming effort of a UIMS user, by creating complex animated scenes without textual programming. The architecture of the HandMove system also contributes to reduce this effort. The scenes have internal dynamics and communicate with the application only by stimuli and signals; thus the application has not to control each action in the scene, but only to send/receive signals.

By considering various input stimuli, HandMove can successfully build scenes for different application types: data visualisation, simulation, animated help, algorithm animation, interactive games, standalone animated objects (e.g. a sandglass), etc.

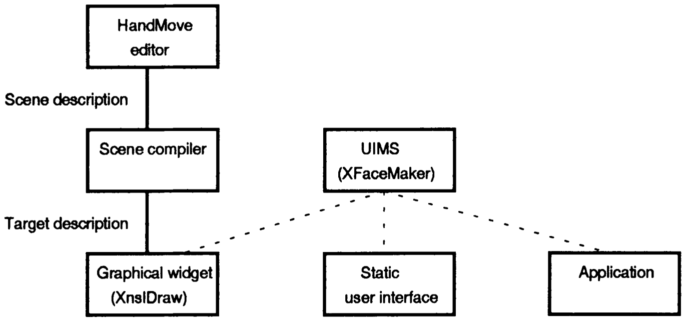


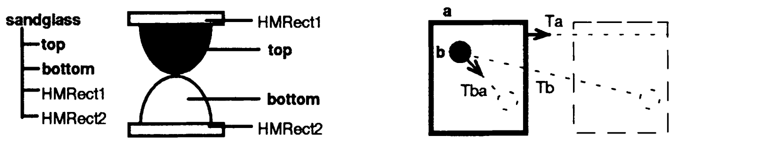
Figure 1 The HandMove architecture.

3 THE UNDERLYING MODEL

3.1 Primitive objects

Actors, decor

Graphical objects in HandMove scenes are two-dimensional; they have a name, graphical attributes (e.g. color, visibility, rotation angle, line type) and a composition list. Components may be primitive objects (e.g. lines, rectangles, labels) or other composite objects, thus an object's structure is represented by a composition tree. Objects with dynamic evolution are called **actors**. Figure 2.a displays the actor *sandglass* and its composition tree, with component actors *top* and *bottom* and static objects *HMRect1* and *HMRect2*. An actor's components are constrained to follow its motion, but they may have their own attribute evolution and a motion relative to the higher level actor. In Figure 2.a, *top* and *bottom* change their filling percentage attribute, while in Figure 2.b, the component actor *b* follows the motion of the higher level actor *a*, but also has its own relative motion (*Ta* is the trajectory for *a*, *Tba* is the trajectory for *b* relative to *a* and *Tb* is the 'global' trajectory for *b*).



(a) Composition tree (b) Relative motion for component actors

Figure 2 Actor composition.

Decor elements are motionless objects involved in the motion of some actors in the scene. A decor element may have one or more actors attached to it during scene evolution. Figure 3 presents a scene for a sort program, where each ball is an actor representing a sorted value and vertical sticks are decor elements with attached actors. Actor-decor attachment is useful when it is simpler to address an actor through the decor element (e.g. in the sort program we want (decor) positions 3 and 4 to interchange values). Decor elements may be used as start/end points for actor motion; in such a case they must define positioning rules for the attached actors (in Figure 3, actors are positioned at the bottom of the decor elements).

As in the example in Figure 3, in many cases we have similar actors behaving in a similar way. In such a situation it is more profitable to describe the evolution of one actor only and to treat the others by similarity. We introduce the notion of **collection of actors** (or decor elements) to respond to this need. Collections (frequently used in algorithm animation) are modelled with arrays and the only operation we consider is addressing an element by its index.

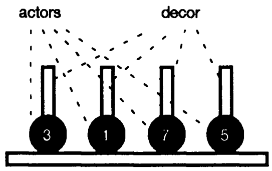
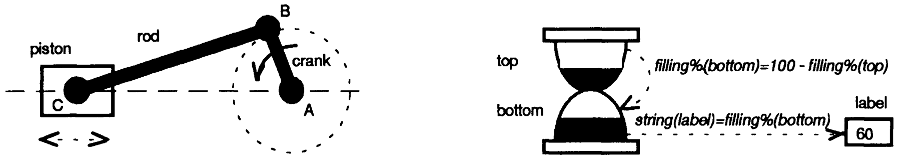


Figure 3 Actors and decor.

Constraints

Actors may be related to each other by position constraints and attribute constraints. **Position constraints** are obtained in HandMove scenes with fixed/mobile articulations and translation guides. In Figure 4.a, the crank-rod-piston mechanism is constrained by two mobile articulations (B, C), a fixed one (A) and a translation guide for the piston on the axis. The position constraints split actors into two categories: with independent motion and constrained ones. To describe the motion in a scene, one needs to specify motion for independent actors only. In Figure 4.a, the crank is the independent actor which determines the movement for the rod and the piston. Another kind of position constraints is induced by actor composition: a component's motion is relative to the position of the whole.

Attribute constraints express permanent relations between the attributes of two actors. HandMove uses binary one-way relationships for attribute constraints. In Figure 4.b, the filling percentage of *top* determines the filling percentage of *bottom*, which constrains the string displayed by *label*.


B
piston
rod
crank
C
A
<- - >
top
filling%(bottom)=100 - filling%(top)
bottom
label
string(label)=filling%(bottom)
60

(a) Position constraints (b) Attribute constraints

Figure 4 Position and attribute constraints.

3.2 Motion elements

Input stimuli and actor evolution

Actors have independent evolutions, under the influence of three types of basic input stimuli: time, values (produced by the external application) and user input (produced by mouse or keyboard events). Time signals are periodic with a given frequency; the others are sudden events.

Actor evolution concerns object motion and attribute changes. Motion is a combination of translation and rotation. HandMove pays special attention to translation (modelled with trajectories), but considers rotation as a simple attribute change (rotation angle).

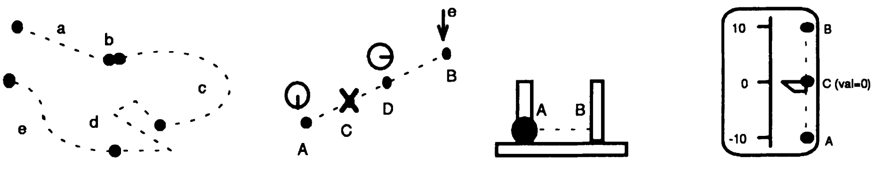
Trajectories

HandMove associates with each actor its (translation) **trajectory**. Trajectories are paths made up of consecutive **segments**. A segment has an origin, an end point, a type (point, line, polyline, ellipse arc, spline) and a geometry. The actor moves along the successive segments; on each segment it follows a different evolution law and it may have a different input stimulus source. The translation law for a segment expresses the covered distance as a function of the input stimulus. Other numerical attributes (e.g. rotation angle, filling percentage) may also vary according to continuous evolution laws. Attributes with discrete values (e.g. color, visibility) may only have discontinuous changes.

Spots are salient points on the trajectory, places where 'something happens'. Spots are the only places where discontinuous changes may happen. Segment start/end points are always spots. A spot may be defined:

- Explicitly, by a given position (pointed on the trajectory) or a decor element (only for segment start and end points);
- By a particular value of the input signal (e.g. when the application value v becomes 0);
- By a particular value of an attribute with continuous evolution (e.g. when the rotation angle is 45°);
- By an incoming event (as shown below).

Figure 5.a presents a possible trajectory, containing all segment types. Segments *a*, *b*, *c*, *d*, *e* are of type line, point, ellipse arc, polyline and spline. On a point segment there is no translation; the start and the end point have the same location, but different definitions (e.g. the end point is often defined by an incoming event). In Figure 5.b, the actor has continuous laws for translation and rotation. A is the start point, B the end point defined by the incoming event *e*, C is defined by an explicit position and D by the value of the rotation angle (90°). Figure 5.c illustrates the use of decor elements as start/end points; the exact position at destination is given by the positioning rule of the end point decor element. In Figure 5.d, the pointer moves along the AB segment, controlled by the application value *val*; spot C corresponds to *val*=0.



(a) Trajectory segments (b) Spots (c) Decor start/end points (d) Input value spot

Figure 5 Trajectory and spots.

The actor's behavior in a spot is described by a set of standard **actions**. Updating an attribute and generating an event are the most common actions (see section 4).

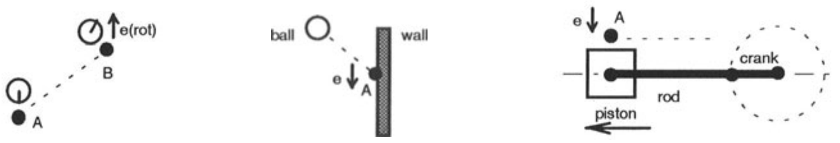
The user creates spots on the trajectory, but their position is relative. Except for the first type of spot, their exact location is known at run-time only. Moreover, spots may be local to a segment or global to the whole trajectory. Only spots defined by an attribute value or by an incoming event may be global. In the case of a global spot, the definition condition is tested all along the trajectory, thus the spot may be located in any point of any segment.

Because segment start/end points are spots, the real trajectory is also known at run time only (in most cases). A segment start point is defined by the real position of the previous end point spot. Moreover, the segment's end point position may be changed by actions at run-time. All these relativity elements define the notion of **abstract trajectory**. Roughly, this means that the path drawn at specification time is different from the real trajectory, but contains all the necessary elements to obtain the final run-time behavior.

Events

HandMove actors communicate through **events**. An actor can explicitly generate an event (only in spots) to announce a change in its status or its motion. Events are identified by their names and may carry one or more values (e.g. attribute values, index of an actor in a collection). In Figure 6.a, event *e* is generated to announce that the actor reached the end of the segment and carries the value of the current rotation angle. Other event sources are:

- The **Master**, a special object that monitors user defined relations between actors. An event is generated when the condition of such a relation is satisfied (e.g. the collision of two actors). In Figure 6.b, the Master generates event *e* (received by the ball) when the ball hits the wall.
- The **Constraint manager**, a special object that manages position constraints. An event is generated when an actor reaches a limit point. In Figure 6.c, the Constraint manager generates event *e* (received by the piston) when the piston moving to the left hits its left limit.
- User input (e.g. a mouse button click, a key stroke).
- The external application, which in addition to application values may generate events for synchronisation purposes and communication with the animated scene.



(a) Actor generated event (b) Master event (c) Constraint manager event

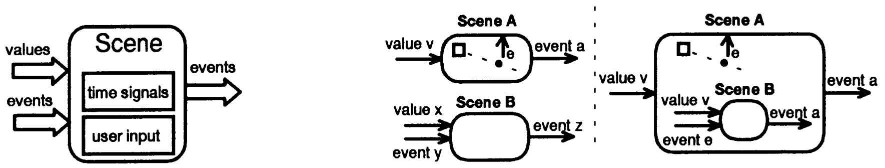
Figure 6 Event generation.

Master and Constraint manager events may be received by the actors involved in the event generation only. Event reception may be restrained to a trajectory segment by using local spots. Generally, events may be received by:

- Actors; this is the most common case.
- Decor elements with attached actors; the event is then dispatched to actors.
- Collections of actors; the event is dispatched to the specified element.

3.3 Animated scenes

A scene is a set of actors and decor elements, related by constraints, moving on trajectories under the influence of input stimuli and communicating through events. Scenes are modules with possible inputs for application values or events and outputs for events (Figure 7.a). Actually, application values may be seen as events carrying a value, so one may consider scenes as receiving and generating events. We consider as scene inputs/outputs every explicit connection which should be made between the scene and its external environment in order to exchange signals. User input stimuli (keyboard, mouse) are produced at run-time and are automatically directed to the scene. They are global to all scenes and need no explicit connection to a scene's input. Time signals are automatically produced by the system for every time-driven segment evolution inside the scene.



(a) Scene inputs and outputs

(b) Merging scenes

Figure 7 Animated scenes.

HandMove uses a scene specification language named HandScript to describe animated scenes. The HandScript format is used to save scenes in external libraries (Figure 8). A scene may be reused in standalone or merged into a new scene. In the latter case, we must indicate the correspondence between the inputs/outputs of the included scene and the existing signals in the new context. In Figure 7.b the scene B is merged into the scene A; the inputs/outputs of B are connected to signals from the context of the scene A (value *v* - the input of A; event *e* - generated by an actor of A; event *a* - the output of A).

```
Scene S (Input Value Int v, Event in(Color c); Output Event out() ){          // inputs and outputs for the scene
    Actor a{                              // an actor
        Object{                           // graphical object description
            Attributes {...}              // graphical attributes
            Geometry {...}                // if primitive graphical object
            Composition {...}             // if composed object
            Constraints {...}             // position and attribute constraints
        }
        Trajectory{                       // list of segments
            {Type: HM_line;               // segment type=line
             Geometry {...}               // line coordinates
             Stimulus = Time (100);       // stimulus signal on the segment (100 ms timer)
             Evolution {...}              // evolution laws on the segment
             Startpoint: Spot_descr{...}  // spot description for start point (actions)
             Endpoint: Spot_descr{...}    // spot description for end point (type, actions)
             Spots{...}                   // list of other spots
            },...                         // other segments
        }
    }....    // other actors, decor elements, collections, etc.
}
```

Figure 8 HandScript description of a scene.

4 DESCRIBING EVOLUTION

4.1 Principles

HandMove focuses the description of an animated scene on actors and their trajectories. Unlike the Whizz model (Chatty, 1992), which separates actors from their motion sources (position/attribute generators), HandMove collects all actor evolution elements at one level. Conceptually, by handling all actor features together, it becomes simpler to express internal relationships, thus to represent more complex and natural evolution.

At the user perception level, this actor focusing policy favours locality: when describing an actor's evolution, the user doesn't need to concentrate on the rest of the scene. The trajectory, the motion laws, the received events, the response actions are all described in the actor's context. For the user describing a scene, with few exceptions, actors are loosely coupled: they interact by events or by constraints (maintained by the system).

Trajectory is the fundamental element in describing actor evolution. Its spatial representation allows a simple and intuitive support for human-system interaction. Actor evolution has two distinct aspects: continuous and sudden changes. Position (obtained from the covered distance on the trajectory) and possibly some actor attributes (e.g. rotation angle, filling percentage) have continuous evolution laws expressed as a function of the input stimulus. Each continuous law gives to a trajectory point a new dimension: the attribute value corresponding to this point. Therefore, we say that continuous evolution laws enrich trajectory semantics. In Figure 9, the point C has four different meanings and is perfectly defined by any of the four values.

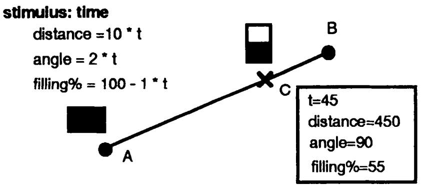


Figure 9 Continuous evolution laws

Figure 10 Position constraints solving

Discrete or sudden changes are described in HandMove by spots, more precisely by the set of actions in each spot. The actions in a spot concern the current actor only and their execution order does not matter. The most important action types are: update an attribute, generate an event, stop/continue evolution, change the current position, restart trajectory/segment, update the end point position, start an alternative segment. Repetitive evolution is obtained by using the restart trajectory/segment actions.

Even though trajectory-oriented motion is appropriate for interactive specification, it has the drawback to fix in advance the actor's route. With the notion of abstract trajectory (defined above) HandMove proposes a solution to this problem by significantly increasing flexibility. By using event-defined end points and end point updates, trajectories dynamically change at run time following the scene status. Thus the trajectory traced by the user is an abstract representation of the real trajectory (at run time). The difference between the two trajectories is the same as the distinction between a program and its execution.

Using constraints represents an appropriate way to reduce the programming effort, by decreasing the number of independent actors. One of the most difficult tasks for HandMove is to manage these constraints. Figure 10 shows an example of determining the movement of a constrained actor. A, B and C are articulations to other objects, A has a known motion law, B is fixed and C is free. The actor's position is completely determined by two points. We know the position of A and B at any moment, so the actor's motion is also known. Thus the motion of C can also be determined and now C will behave as a point with known motion law for the other object articulated in C. This method is used by the scene compiler to determine the motion of all constrained actors and to verify if the position constraints are not contradictory. The compiler finds the sequence of direct calculation formulas to be computed at run time for each constrained actor.

Most of the previous examples use time because the reader has a better perception of time-driven animation. The other input stimuli types may be used in a similar manner. Motion and continuous evolution laws can be based on numerical application values or the mouse pointer position. A special (and frequent) case is when the actor is constrained to follow the mouse position. The main difference between time and the other stimuli is that time signals are periodic and cumulative. While for application values or user input the stimulus value is carried by the signal, the time stimulus value used is given by the total number of time signals.

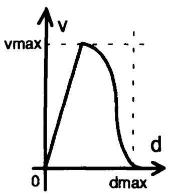
4.2 The HandMove editor

The HandMove editor allows to create animated scenes by direct manipulation. Actors, decor and trajectories are described using graphical primitives (e.g. lines, arcs) and interaction techniques typical of graphical editors. All actors in a scene share the same space for their evolution. Actors/decor and trajectories are organized in separate layers.

To define attribute constraints, the user points the related actors, specifies the attributes and the dependency formula (among some standard dependency types). To define an articulation, the user specifies the joined actors and the articulation point for each one. A guide constraint is defined by attaching a trajectory to the constrained actor.

In describing a scene, the user first specifies scene's inputs/outputs, then actors, decor, constraints, trajectory, etc. For actor evolution, he describes each segment's characteristics (input stimulus, continuous laws). Continuous laws may be chosen among some standard types and usually are linear dependencies. The most interesting case is in describing actor translation laws in time-driven animation. Besides the standard motion types (uniform, accelerated, 'slow-in/slow-out' (Hudson, 1993), etc), the user may specify arbitrary motion laws, by using a special editor (Figure 11). The user traces the graph of velocity (*v*) as a function of the covered distance (*d*), by using lines, ellipse arcs, splines, etc. We use distance instead of time, because it is difficult to estimate the total time necessary to cover the segment, while the total distance can be easily determined.

To create a spot, the user indicates a point on the trajectory and its meaning (exact location, attribute or stimulus value, incoming event). Related actions are selected among the limited set of choices. All these elements (scene's inputs/outputs, stimuli sources, segment characteristics, spots) are displayed with dedicated graphical notations. This graphical feedback provides to the user a synthetic view of the scene. Figure 12 shows an example for spots: we describe spot definition, spot actions, restart condition (if one of the actions is 'stop') and restart actions. Spot A is defined by its exact position on the abstract trajectory; when the spot is reached, the actor generates event *s*. Spot B is defined by the value of the rotation angle (45°) and the actor arrived in B stops. It will restart only when it receives event *g* and it will change its color at that moment.



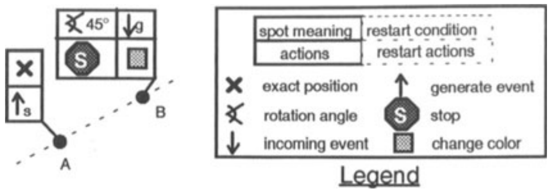
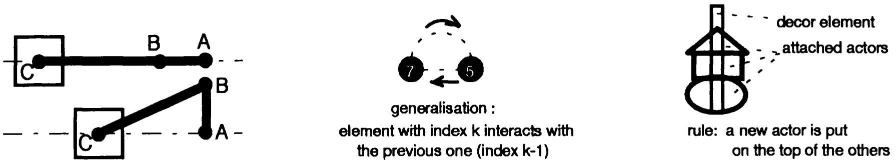


Figure 11 Arbitrary motion law. **Figure 12** Graphical feedback for spots.

Animated scenes can be entirely described by using the direct manipulation techniques presented above. In certain cases, demonstrational techniques may be used in order to facilitate the scene definition process. Future work will consider the construction of a demonstration layer on the top of the HandMove model. The visual representation of animation provided by our model offers an elegant solution to the problem of program visualisation/update, typical for programming by demonstration. We present below some examples of using demonstration to infer relationships.

- Constraint inference, by using different configurations of the related objects (in Figure 13.a, the system receives the two actor configurations and infers the existence of articulations in A, B and C).
- Generalisation of collection elements behavior (in Figure 13.b, the system infers that each collection element switches with the previous one).
- Inference of positioning rules for decor elements with attached objects (in Figure 13.c, the position of the attached actors enables the system to infer the positioning rule).



(a) Position constraints inference (b) Collection generalisation (c) Rule for decor element

Figure 13 Demonstrational techniques.

5 AN EXAMPLE

Let's assume the four-stroke engine in Figure 14. We define the following actors:

- *Piston*, *rod* and *crank*, related by position constraints (fixed point articulation for *crank* in A; *rod* and *crank* articulated in B; *rod* and *piston* articulated in C). *Piston* has a linear translation inside the cylinder, *rod* and *crank* are constrained by *piston*.
- *Admission* and *leakage*, the valves for admitting fuel and evacuating exhaust gas.
- *In* and *out*, arrows that are visible only when *admission*, respectively *leakage* are open.
- *Spark*, which is visible a brief moment when *piston* produces a maximum fuel gas pressure.

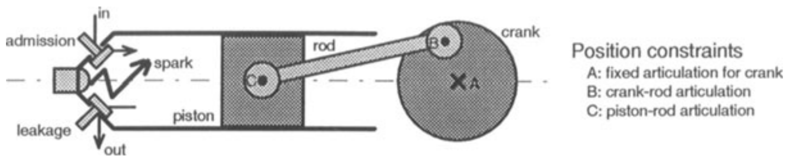

in
admission
spark
leakage
out
piston
rod
crank
B
C
A
Position constraints
A: fixed articulation for crank
B: crank-rod articulation
C: piston-rod articulation

Figure 14 Example: a four-stroke engine.

The evolution description for the main independent actors is illustrated in Figure 15. *Leakage* and *out* have similar evolution to *admission* and *in*; *rod* and *crank* are constrained by *piston*. We only illustrate start/end point definitions using the graphical representation described above; continuous evolution laws are not shown.

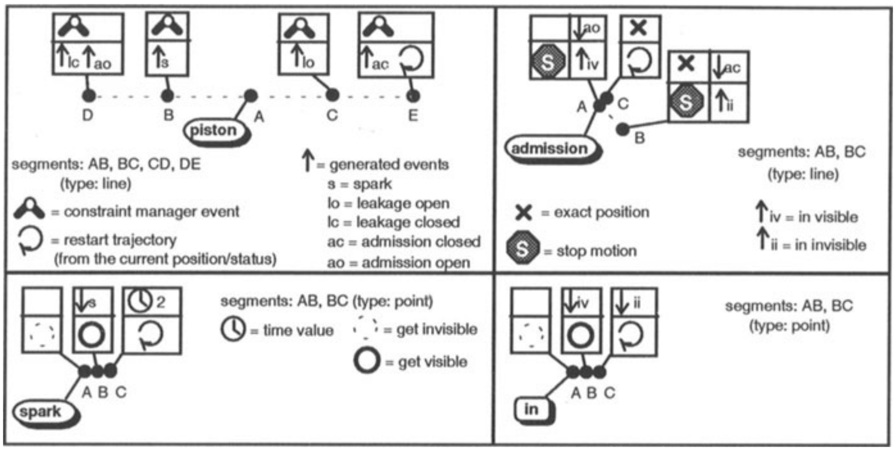


Figure 15 Actor evolution description.

Piston trajectory has four (collinear) segments, which end points are defined by Constraint manager events indicating limit positions. Events generated by *piston* control other actors (*spark* in B, *leakage* opening in C, *leakage* closing and *admission* opening in D, admission closing in E). At the end, motion is restarted considering the current position as start point. *Spark* has no motion, so we use point segments. The action in A establishes the initial status (invisible). The first segment ends when the *s* event from *piston* is received; *spark* becomes visible for 20 seconds, then its evolution is restarted (thus by becoming invisible again).

Admission begins by stopping its motion and waiting for event *ao* (from *piston*) to start the first segment. When starting, it generates an event to make *in* visible. In end point B (defined by an exact position), *admission* stops again; to start the second segment, it waits for the *ac* event. At the end point C (defined by an exact position, the same as for A), the motion is restarted from the current position (thus by waiting). *In* acts like *spark*, excepting that its visibility is controlled by the events generated by *admission*.

6 IMPLEMENTATION ISSUES

As mentioned above, our implementation uses the XnslDraw widget, created by NSL. XnslDraw is an XWindow (Xt Intrinsics) widget managing graphical objects (e.g. lines, ellipses, text). Low-level scripts may be attached to objects, describing behavior in reaction to external events. Function calls available in scripts are: move object, change an attribute, call another script, etc. The HandMove scene compiler translates actors to XnslDraw objects and their evolution to XnslDraw scripts. The integrated constraint compiler transforms constraint resolution in direct calculation formulas.

The contents of a compiled scene is kept in a XnslDraw file. The dynamic element to be included into a user interface will be a XnslDraw widget instance connected to this file. Thus, we do not create a special widget for each scene; we simply use the XnslDraw widget with different description files. However, it is possible to transform a couple XnslDraw widget - file into a new independent widget.

XnslDraw animated scenes are integrated in interfaces created with XFaceMaker, the NSL's X/Motif UIMS. XFaceMaker provides an interactive editor for interface layout design and a high-level language (Face) for user interface behavior specification. XnslDraw widgets containing animated scenes are added to the user interface as any other static Motif widget.

To communicate with the application and with the rest of the interface, HandMove scenes use two methods. In order to receive input signals (which can all be seen as events carrying values), HandMove scenes define a reduced functional interface (Figure 16). Sending a signal to the scene is equivalent to a function call. To send signals to the rest of the interface, HandMove scenes use the active values mechanism of XFaceMaker (Figure 16). Roughly, an active value is an interface object defined by a variable and two Face scripts (get script and set script) to manipulate this variable. For each outgoing event, a HandMove scene uses an active value. At run time, in order to send the signal, the scene calls the set script of the active value. This script can be handled by the XFaceMaker user to program the reaction of the application to the signal.

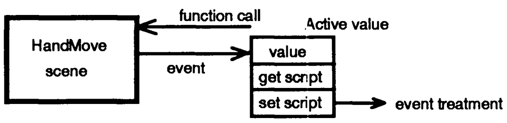


Figure 16 Integration of HandMove scenes in XFaceMaker applications.

Animated scenes may be used outside the XFaceMaker UIMS (e.g. in C programs), with one restriction. Currently, there is no mechanism for transmitting events generated by the scene toward the external application, thus HandMove scenes may only receive signals.

To respect time requirements, HandMove uses the script scheduling mechanism of XnslDraw, which allows to arm a script to be launched after a specified time period. This method cannot guarantee exact timing, due to the script treatment policy of XnslDraw and to the asynchronous character of the XWindow system. Timing problems in Unix and XWindow systems are well presented in (Hudson, 1993). On the other hand HandMove provides a powerful mechanism to describe asynchronous scene evolution (by event-based communication).

7 CONCLUSIONS AND FUTURE WORK

HandMove is an interactive system for creating animated scenes, used afterwards as dynamic user interface components. Resulting animation can be integrated in applications built with the XFaceMaker UIMS, but it may be easily adapted to other user interface builders. HandMove combines various input stimuli (time, application values, user input), so it can be used for very different dynamic application types (e.g. standalone animation, algorithm animation, interactive games).

The scene model is rich, containing some novel features as: composed actors, decor elements with attached actors, collections of actors, position and attribute constraints, event-based communication, abstract trajectories. The interaction model is based on direct manipulation and favours a local view on actor evolution. The use of dedicated graphical symbols offers a comprehensive view on the programmed behavior.

Future work will consider the following problems:

- Extending the scene model to include new features as object creation/destruction, shape changes, etc.
- Formalizing the HandMove scene model.
- Minimizing the interaction between the scene and the application. A scene's status is given by the position and the attribute values of each actor. Other status information must be kept by the application and exchanged by events with the scene. We want to extend our model in order to reduce such exchanges.
- Building a demonstrational layer on the top of the HandMove model.

Acknowledgements

I am grateful to Ion Filotti, Michel Scholl, Stéphane Chatty and the whole NSL team for stimulating discussions and helpful comments.

8 REFERENCES

Borning, A., Duisberg, R. (1986) Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, October, pp. 345-374.

Brown, M. (1988) Exploring algorithms using Balsa II. *Computer*, 21(5), May, pp. 14-36.

Chatty, S. (1992) Defining the dynamic behavior of animated interfaces, Proceedings EHCI'92, *Engineering for human-computer interaction*, IFIP Transactions, pp. 95-109.

Cypher, A. (1993) *Watch what I do. Programming by demonstration*. MIT Press.

Duisberg, R.A. (1987) Animation Using Temporal Constraints: An Overview of the Animus System. *Human-Computer Interaction*, vol. 3(3), 1987/1988, pp. 275-307

Hudson, S.E., Stasko, J.T. (1993) Animation Support in a User Interface Toolkit: Flexible, Robust and Reusable Abstractions. *Proceedings UIST'93*, Atlanta GA, pp. 55-67.

Jackiw, R.N., Finzer, W.F. (1993) The Geometer's Sketchpad. Programming by Geometry. in Cypher, A. *Watch what I do. Programming by demonstration*, MIT Press, 1993.

Nelson, G. (1985) Juno, a constraint-based graphics system. *Proceedings SIGGRAPH'85*, San Francisco, pp. 235-243.

NSL (1994a) *The NSL Widget Library*.

NSL (1994b) *XFaceMaker - User's/Reference Guide*.

Open Software Foundation (1991) *OSF/Motif Style Guide*, Prentice Hall.

Robertson, G.G., Card, S.K., Mackinlay, J. (1993) Information Visualization using 3D Interactive Animation. *Communications of the ACM*, Vol. 36, No. 4, April, pp. 57-71.

Stasko, J.T. (1991) Using direct manipulation to build algorithm animations by demonstration. *Proceedings CHI'91*, New Orleans, pp. 307-314.

Watt, A., Watt, M. (1992) *Advanced Animation and Rendering Techniques.* Addison-Wesley.

9 BIOGRAPHY

The author is a PhD student at the Conservatoire National des Arts et Métiers in Paris, France. He graduated from the Polytechnic Institute of Bucharest, Romania and the 'Pierre et Marie Curie' University in Paris, France. His research interests include human-computer interaction, animation in user interfaces, visual programming, programming by demonstration and geographical information systems.

Discussion

Michel Beaudouin-Lafon: Is your system extensible, for example how can I include audio output and synchronise it with the graphics?

Dan Vodislav: We haven't considered sound, but as the model is clear and simple, it should be easy to insert new elements. For example, we added key-frame animation actors.

Claus Unger: How do you check and prove that all the user-defined constraints are consistent with each other?

Dan Vodislav: Constraint consistency is checked at specification time, but the fact that an anchor goes beyond its limit position has to be checked at run-time. This cannot be checked at specification time because of the abstract nature of the user-specified trajectories. These constraints are geometric and are easy to check at run-time.

Pedro Szekely: What can HandMove do that systems like Alias Wavefront and similar commercial tools cannot do?

Dan Vodislav: The main contribution of HandMove is visual specification, especially the visual representations of evolution. The model was intended to offer a single virtual representation and it cannot describe any dynamic evolution. In such cases, some actions must be delegated to the application, which communicates with the animated scene through events.

Jim Larson: What has been the experience of users with HandMove?

Dan Vodislav: The system is not yet completed, so we have no user experiences .

Joëlle Coutaz: How do you, firstly, specify the types of events and secondly, perform type checking when composing scenes?

Dan Vodislav: There is a special panel for event parameter (type) specification. The type of a merged scene must also be explicitly specified by the user.

Joëlle Coutaz: How do you choose the master object? Does everything change if you change the master object, for example, taking the crank instead of the piston as the master object for your example?

Dan Vodislav: The piston is the natural choice for this example, but it is true that changing the master can require changes to the whole scene evolution, but generally we obtain a different behavior by changing the master.

Bernard Merialdo: Your example of a ball bouncing off a wall had only one ball. How would you model several balls colliding with each other and hitting the wall?

Dan Vodislav: We may use a collection of balls and describe the trajectory as the motion between two collisions, then restarting this trajectory with a new direction after each collision. The Master has to detect any collision between the ball and the wall or the rest of the ball collection.