Visual Programming of Virtual Worlds Animation

Alberto Del Bimbo and Enrico Vicario
Università di Firenze

In virtual reality interfaces, realistic animation of virtual agents enhances human-computer interaction by supporting direct engagement in the virtual environment. Visual programming by example allows designers to define animation rules by "training" agents, thereby building behavioral rules into specification models that run automatically during the execution of the virtual environment. This allows for direct and effective replication of real-life phenomena and agent reactions to environmental stimuli.

Human-computer interaction has evolved from textual programming toward visual languages with 3D direct manipulation interfaces. In this evolution, the cognitive effort necessary for users to manage system objects has been progressively reduced by augmenting their engagement in the operation. Virtual reality environments are the ultimate stage in this trend.[1]

VR environments represent directly the world of interest. Complex interfaces engage users' 3D perceptual and spatial skills, which are largely underemployed in conventional interfaces, to accomplish a direct and natural interaction. Virtual agents that change their state in response to external and internal stimuli enhance the sense of a real-life environment. The user can interact with the system without referring to syntactic and semantic conventions of an intermediary interaction language, relying instead on empirical knowledge of the emulated environment. VR applications in fields such as personnel training, entertainment, hostile-environment exploration, and physical therapy may profit from such augmented yet simplified interaction.

Constructing and animating virtual worlds

To present an effective emulation, virtual agents must exhibit both realistic graphic shapes and realistic behaviors. Realistic shapes help the user perceive the virtual environment as a replica of a real context, thus encouraging realistic interaction. Realistic agent behaviors improve the user's engagement in the virtual world by giving a sense of interaction with a context populated by natural agents.

Due to their inherent complexity, the construction of virtual worlds requires using appropriate notations, methodologies, and tools that support the designer at all stages, from specification to implementation. Several projects have been aimed at providing toolkits for the effective design and construction of virtual worlds, such as Aviary,[2] MR,[3] World and Cyberspace, and the VR-Deck toolkit.[4] These tools all support the creation of realistic object shapes through graphic libraries, as well as effective user interaction through appropriate interface modules to I/O devices. Programmers animate virtual worlds by specifying agents' behavior through event-action rules. Each rule specifies which input events are of interest and which actions and output events are generated at their occurrence. Modules representing virtual objects communicate with each other by the exchange of events corresponding to state changes or to the occurrence of certain spatiotemporal relationships among the agents. Behavioral rules are usually defined textually, using either a conventional programming language[3] or a specific rule language.[4]

Textual programming offers the advantage that rules written directly on paper can be examined and modified at will. Its disadvantage is that it requires learning a new language and fully understanding all the conditions and reactions that characterize the behavior of the agents under development. This is often a complex task because we understand real-life phenomena in terms of empirical experience and unconscious factors, without retaining explicit knowledge of regularities and rule details. For this reason, textual specification of an agent's behavior creates a cognitive gap between what we know about the real-life entity emulated by the agent and what we can express in terms of quantitative specification rules.

Programming by example

Visual programming by example has been proposed as a means to bridge this gap, based on the rationale that most people deal better with specific concrete objects than with their abstract representations. A program can be more easily written by giving examples of what it is expected to do than through a sequence of textual rules. Specification by example appears to be particularly well suited to the development of virtual

1070-986X/96/$5.00 © 1996 IEEE

worlds, as it allows designers to express directly their knowledge about emulated realities without an explicit and complete understanding of quantitative details. Designers can rely instead on natural human capabilities such as intuition, sense of distance, orientation, and the like.

A few experiences reported in the literature apply this paradigm to the construction of advanced interfaces and graphic animations. In the *agentsheets* approach,[5] the animation of a set of typed agents moving over a discrete grid of typed cells is specified in terms of reaction rules. These determine the agent's next position as a combinatorial function of the agent type and of the cell where it presently stands. The spreadsheet's cell layout is defined visually, and the agent animations are specified through visual examples that replicate reaction rules. This approach is the basis of Kidsim,[6] designed to provide tools for young users to create graphic simulations. An environment is constructed by demonstration in terms of graphic rewriting rules that change the appearance and position of iconic agents moving over a discrete 2D grid. Rewriting rules are defined by creating a "before-picture" and an "after-picture" and by recording agent trajectories that implement the transformation between them.

The application of programming by example to virtual worlds animation has more demanding requirements than those encountered in 2D animation. The 3D reproduced spaces in virtual realities require a high degree of animation fineness in both space and time. The resulting dense representation of space and time prevents the use of discrete grids and makes ineffective the expression of triggering environment conditions in terms of exact quantitative relationships among virtual agents. Virtual reality animations also must be sensitive not only to the current environment configuration but also to the sequential evolution of agent relationships with the environment. These requirements demand that the system supporting programming by example have a more sophisticated formal nucleus to support a flexible representation of spatiotemporal conditions and sequential reaction patterns.

To investigate how visual specification by example copes with these augmented expressivity requirements, we developed a prototype tool that supports an evolutionary approach to the interactive and visual specification of virtual agent behavior. This tool employs visual behavioral training, a specification by example process that defines the behavior of each agent by visually replicating its reaction pattern to specific perceived stimuli coming from the virtual world.

The system interprets examples produced by the designer during the specification stage and automatically translates them into specification models. These models cast agent behavior into a reactive framework in which the agent switches among different operation modes based on how its spatial relationships with the other agents in the environment evolve temporally. During the operation of the virtual environment, these models adjust agents' behavior with behavioral patterns emerging from the specification examples.

In the following sections, we describe the internal formalism employed to capture perceived stimuli and reaction patterns of virtual agents. We then present the system that embeds this formalism within a visual interactive shell, discussing principles of operation and examples of use.

Supporting behavioral training through visual specification

In a specification by example approach, the designer constructs the behavioral models of virtual agents by replicating their actions under relevant environmental conditions. The system interprets these examples and captures them into an internal specification model that will operate during the execution of the virtual environment. Our system casts this internal model within a reactive framework made up of stimuli and reaction patterns. Stimuli occur as certain temporal evolutions of spatial relationships among the entities of the virtual environment. They are represented through an original language that extends concepts of conventional temporal logic to deal with space and time in a unified framework.

Reaction patterns are expressed through a Petri net-based model that permits representation of internal parallelism. This specification formalism has a two-fold role in the agent's training and operation. First, in the training stage, it captures the agent's behavioral responses to stimuli and trains it to act in a specified way. Next, in the execution stage, specification models replicate the agent's trained behavior. They also reproduce the evolution of spatiotemporal relationships among virtual agents to recognize the occurrence of situations to which the agent should react.

Specification of perceived stimuli

To encompass the expression of significant spatiotemporal phenomena, as encountered in the

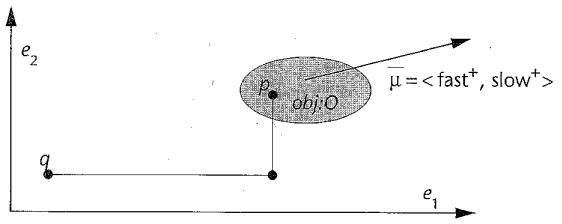

Figure 1. A sketch of a 2D scene. This illustrates that obj is an object of type O moving fast along axis e_1 and slow along axis e_2 (that is, it has a speed vector $\bar{\mu} = <fast^+, slow^+>$). The XSTL assertion (p) $\models_s <O; fast^+, slow^+>$ captures the fact that point p is occupied by an object of type O and motion status $\bar{\mu} = <fast^+, slow^+>$. The XSTL assertion (q) $\models_s \Diamond^{e1-}_{long} \Diamond^{e2+}_{short} <O; fast^+, slow^+>$ captures the fact that an object of type O and motion status $\bar{\mu} = <fast^+, slow^+>$ is reached from point q by moving for a long distance along the positive direction of axis e_1 and then for a short distance along the positive direction of axis e_2.

Spatial Logic of Frame Assertions

Spatial Logic quantitatively represents spatiotemporal phenomena in dynamic VR scenes. The sample assertions below express specific spatial relationships among elements in individual frames of a sequence.

Assertion	Meaning
$(p)\models_s \varnothing$	Point p satisfies spatial formula $\varnothing$
$\varnothing = <O; \bar{\mu}> \mid \neg \varnothing_1 \mid \varnothing_1 \wedge \varnothing_2 \mid \Diamond^{e+}_{\gamma} \varnothing_2$	Spatial formula capturing the position of a point with respect to objects in the scene
$(p)\models_s <O; \bar{\mu}>$	Point p is occupied by an object of type O which moves with speed $\bar{\mu}$
$(p)\models_s \neg \varnothing_1$	Point p does not satisfy spatial formula $\varnothing_1$
$(p)\models_s \varnothing_1 \wedge \varnothing_2$	Point p satisfies spatial formulae $\varnothing_1$ and $\varnothing_2$
$(p)\models_s \Diamond^{e+}_{\gamma} \varnothing_2$	Spatial formula $\varnothing_2$ is satisfied in some point p' reached from p moving along the positive direction of axis e such that the distance $\lvert p-p' \rvert$ satisfies constraint γ
$(p)\models_s \Diamond^{e-}_{\gamma} \varnothing_2$	Spatial formula $\varnothing_2$ is satisfied in some point p' reached from p moving along the negative direction of axis e such that the distance $\lvert p-p' \rvert$ satisfies constraint γ

perceptual process of real-life entities, we designed an original language, eXtended Spatio Temporal Logic (XSTL). This language extends the concepts of conventional Temporal Logic[7] to support a homogeneous description of both spatial and temporal phenomena.[8] The spatiotemporal evolution of elements in a dynamic scene is captured through assertions organized in nested dynamic and static levels.

At the dynamic level, sequence assertions, expressed through the conventional operators of Temporal Logic, capture the evolution over time of the spatial contents of the individual frames. These spatial contents are defined at the static level by frame assertions formed in an original language, Spatial Logic, which transposes the operators of Temporal Logic to deal with space instead of time. Mirroring concepts proposed in the literature to augment Temporal Logic with quantitative modeling capability, XSTL introduces metric expressivity in both frame and sequence assertions to permit representation of qualitative metric relationships among space points and time instants in which certain conditions hold.

Spatial Logic of frame assertions

Frame assertions capture spatial relationships between the points occupied by agents appearing in individual frames of a sequence. The simplest frame assertion states that an agent of a specific type and having a particular motion status occupies some point in the scene. If p is a point, O an object type, and $\bar{\mu}$ a speed vector, the assertion $(p) \models_s <O, \bar{\mu}>$ means that point p is occupied by an object of type O moving at speed $\bar{\mu}$ (see Figure 1). We can create more complex assertions using the operators *spatial forward eventually* ($\Diamond^{ei+}_{\gamma}$) and *spatial backward eventually* ($\Diamond^{ei+}_{\gamma}$) to express that, starting from point q and moving along an axis e_i, point q' is reached within a specified distance range γ in which a certain spatial assertion holds. The sidebar "Spatial Logic of Frame Assertions" gives examples and detailed descriptions of assertions that can be composed for each scene to describe in quantitative terms the spatial relationships among elements.

Figure 1 shows how composing multiple *eventually* operators that refer to multiple axes of the scene lets us capture the displacement of point q with respect to an object of a given type and motion status.

Recursive combination of multiple spatial *eventually* operators referring to different axes produces an assertion that basically captures the structure of a walkthrough leading from point q to object *obj*. By combining several such assertions through the Boolean connectives, the position of object *obj* with respect to point q can be characterized as a set of walkthroughs leading from q to any of the points of *obj*. The position of an observed object *obj* with respect to an observing object obj_o can then be characterized through the set of walkthroughs leading from the points of obj_o to the points of *obj*. Objects represented as generic polygonal shapes can be delineated by a finite set of points; different selections of representative points can accomplish different levels of detail

and accuracy in representing objects' shapes and spatial relationships.

Operatively, if we approximate the observed object with its minimum bounding rectangle, we can obtain the possible walkthroughs by extending these bounds along the axes of the observing object and then see which regions of the resulting partition include part of the observing object. As shown in the 2D scene depicted in Figure 2, each such region is characterized by a specific assertion capturing a walkthrough from that region to a region of the observed object. Combined, these assertions depict the mutual position of the two objects as a set of walkthroughs such that (1) each walkthrough is possible from at least one point in the observing object, and (2) from each point in the observing object there exists at least one possible walkthrough to the observed object.

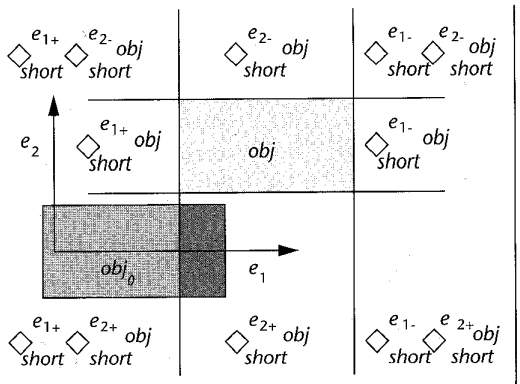

Figure 2. A 2D scene allowing for two different walkthroughs from the observing object obj_0 to the observed object obj. The spatial position of obj_0 with respect to obj is expressed by two possible walkthroughs: if O and μ are the type and the speed of obj, the walkthrough assertion $(obj_o) \models_s \Diamond^{e1+}_{long} \Diamond^{e2-}_{short} <O, \bar{\mu}>$ is satisfied by all the points in obj_o, while $(obj_o) \models_s \Diamond^{e2+}_{short} <O, \bar{\mu}>$ is possible only for the points of obj_o in the dark gray area of the figure.

Temporal Logic of scene sequences

Sequence assertions capture temporal ordering relationships among the scenes in which certain frame assertions hold. The simplest possible sequence assertion is expressed in the form $(j) \models_t \Phi$ and means that the frame assertion Φ holds in the jth scene of the sequence. (See the "Temporal Logic of Scene Sequences" sidebar for more examples and detailed descriptions.)

The temporal operator *backward eventually* ($\Diamond^t_\gamma$) expresses assertions involving more than one scene of the sequence. Moving back along the time axis, a scene is eventually encountered, within a specified temporal qualitative distance, in which a certain sequence assertion holds. For instance, given a sequence assertion θ and a scene index j, the sequence assertion $(j) \models_t \Diamond^{t-}_{soon} \theta$ means that there exists a scene j' that precedes j such that the distance $j - j'$ falls in the range of *soon* and that θ holds in scene j'.

We can express complex time-ordering relationships among scenes with different spatial conditions by composing frame assertions through the Boolean connectives and with multiple temporal operators. In general, XSTL permits representation of stimuli in terms of relationships among agent trajectories, with the possible use of quantitative spatial, temporal, and speed parameters. This expressivity could be extended, without changing the essence of the formalism, by augmenting the syntax of the basic scene assertion with additional motion parameters (for example, acceleration or angular parameters) or with parameters characterizing relevant aspects of the instantaneous status of the agent. This enables the representation of stimuli depending on nonmotion conditions, such as the appearance or the spirit of a living agent.

Temporal Logic of Scene Sequences

Temporal Logic assertions express time-ordering relationships among phenomena in dynamic VR scenes. Below are some examples that express how specific spatial relationships evolve among elements as frames of a sequence unfold.

Assertion	Meaning
$(j) \models_t \theta$	The *jth* scene satisfies temporal formula θ
$\theta = \Phi \mid \neg\, \theta_1 \mid \theta_1 \wedge \theta_2 \mid \Diamond^t_\gamma \theta_2$	Temporal formula capturing the temporal position of a frame in the sequence featured by the environment
$(j) \models_t \Phi$	The *jth* scene satisfies spatial assertion Φ
$(j) \models_t \neg\, \theta_1$	The *jth* scene does not satisfy temporal formula θ_1
$(j) \models_t \theta_1 \wedge \theta_2$	The *jth* scene satisfies temporal formulas θ_1 and θ_2
$(j) \models_t \Diamond^t_\gamma \theta_2$	The temporal formula θ_2 is satisfied in a frame with index $k \leq j$ such that the temporal distance $j - k$ satisfies constraint γ

Specifying reaction patterns

Reaction patterns are specified through a reaction sequential logic that defines how the agents' elementary moves are engaged in response to internal control commands. The agent engages these, in turn, when it perceives stimuli from the virtual environment. For each agent, the reaction logic comprises a set of execution modes formed

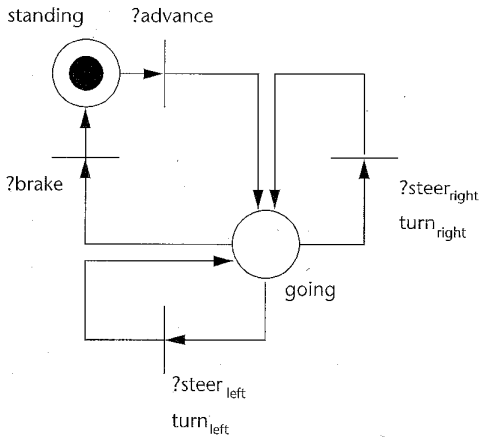

Figure 3. A condition-event representation for the reaction logic of a car agent. The car agent has two operation modes, standing and going, activated by four control commands: advance, brake, $steer_{left}$, and $steer_{right}$.

by recursively combining basic actions through a set of trace operators such as concatenation, termination, and selections driven by the occurrence of environmental stimuli. At the start of behavioral training, the reaction logic defines how the agent responds to control commands, but does not specify the conditions under which these commands are to be issued. The behavioral training completes the reaction logic by defining which spatiotemporal stimuli activate control commands.

For the sake of concreteness, consider the case of a car agent provided with three basic actions: *go*, $turn_{left}$, and $turn_{right}$; two execution modes, *standing* and *going*; and four control commands, *advance*, *brake*, $steer_{left}$, and $steer_{right}$. In *going* mode, the car repetitively executes the *go* action; different control commands may force the car to execute actions $turn_{left}$ and $turn_{right}$ or to enter *standing* mode. In *standing* mode, the agent does not execute any action until the user engages the control command *advance* to return it to *going* mode.

We can describe such reaction logic suitably using a condition-event Petri net model,[9] in which mode activations are associated with conditions, and changes in the activation scheduling are represented as events. For the car agent, the net has two conditions corresponding to *going* and *standing* execution modes, and four events (transitions) associated with the control commands. Figure 3 shows the token in *standing* condition, which indicates that initially this is the only active mode. Engaging the control command *advance* moves the token from *standing* to *going* mode; this sets the *standing* condition to "false" and the *going* condition to "true." Upon engagement of the *brake* command, the agent leaves *going* mode and reenters *standing* mode. The commands $steer_{left}$ and $steer_{right}$ remove the agent from *going* mode, execute the basic actions $turn_{left}$ and $turn_{right}$, respectively, and again return the agent to *going* mode.

The condition/event model enables designers to represent synchronization (by allowing the explicit sequencing of events), parallelism (by allowing tokens in multiple places), environment choice (by allowing concurrent events associated with different control commands), and non-deterministic selection (by allowing concurrent events associated with the same control command). Canonical Petri net extensions can further augment the model's expressive power. Static priorities associated with transitions can be used to support deterministic selection among conflicting events.[9]

Timing assumptions, expressed by associating transitions with a time duration or with an execution delay, create time-triggered actions (for example, a car starting after a 30-second stop). Behavioral patterns that change with continuity (for example, a car changing its speed within a dense range) can be represented by replacing the low-level condition-event net with a high-level Petri net provided with functional modeling capability.[10] This permits representation of dense-valued status information (for example, the speed of the car), which, applied to basic actions (for example, the method of advancing the car), obtains a continuously changing behavior.

Colored Petri nets enable designers to synthesize complex parametrized behaviors by allowing them to specialize agent states and transitions with attributes. By associating such state and transition parameters with stimuli dependent on non-motion conditions, designers can also describe interactions based on non-visual relationships.

Interactive training of virtual agents

We describe here operating principles of the system supporting visual programming by example and automatic operation of virtual agent behaviors. The system runs on an IBM RISC 6000 machine with a 7234-GTO graphics accelerator for real-time interaction with the 3D virtual world. Visualization is based on GraPHIGS under X-Windows. The system derives agents' behavioral models by interpreting training examples that specify

1. which spatiotemporal stimuli are of interest for each agent, and
2. what control actions the agent takes in response to their occurrence.

These examples are interpreted and represented according to the internal formalism described earlier. After this training, the system automatically operates the new behavioral models to feature a virtual environment where agents behave according to rules derived from the specification examples.

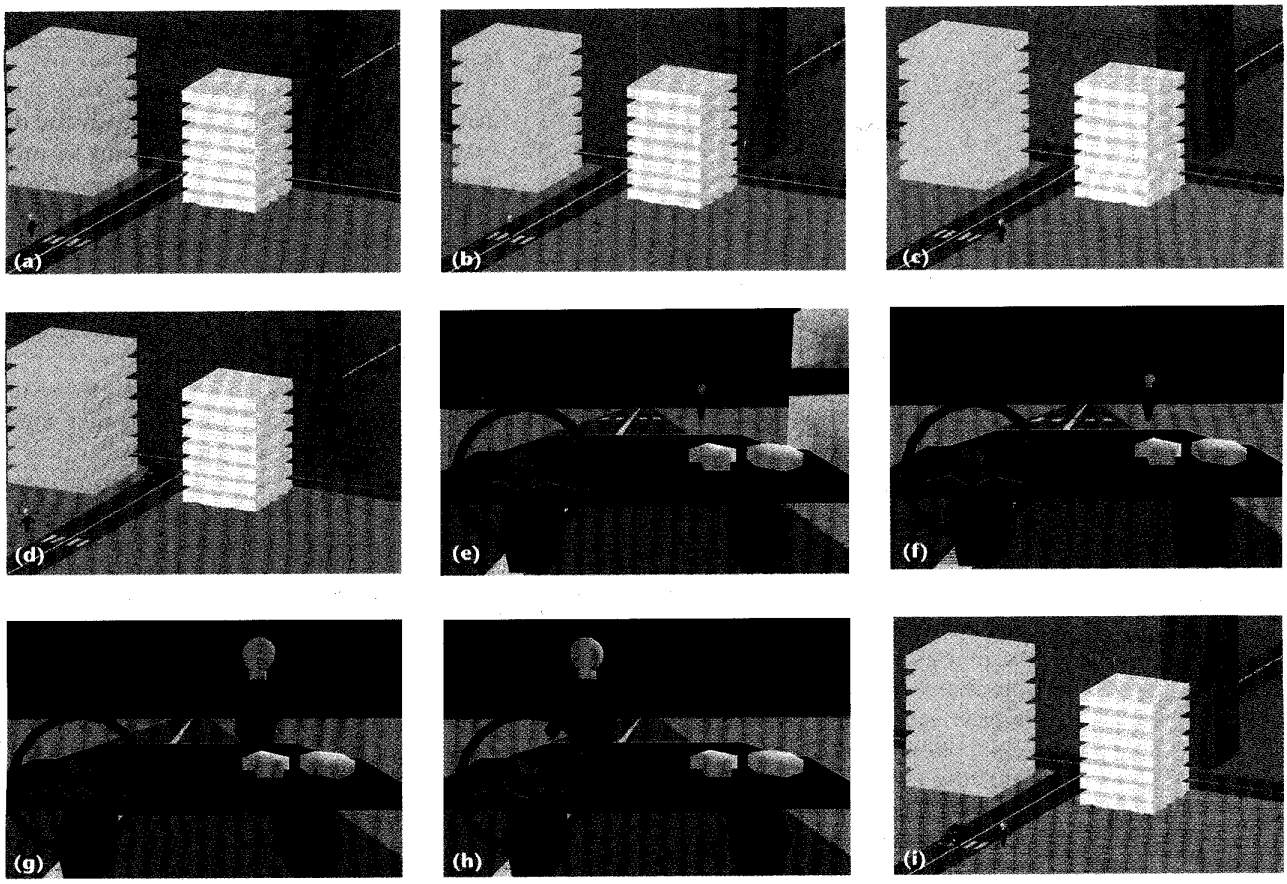

Figure 4. Visual programming of a car agent: behavior in the presence of a pedestrian. The background pedestrian agent is first made to cross the road (a through c). During the playback of this motion, the designer "gets in" the car (d and e) and drives it until she gets close to the pedestrian agent (f and g), then engages the brake command (h) and does not advance until the pedestrian has crossed (i).

Creating training examples

To create training examples, the designer visually composes dynamic scenes populated by a number of standing or moving background agents. These produce spatiotemporal situations to which the agent should react. The concurrent motion of multiple agents involved in such conditions is composed according to a multi-track recorder model: Each agent's motion pattern is separately recorded and played back during the recording of subsequent agents' motion using the rewind and playback buttons available in the interface. The designer can then synchronize the patterns of multiple agents simply by adjusting their relative speeds.

Once a training scene has been created, the designer specifies through example how the agent will respond to the spatiotemporal stimuli occurring within that scene. To enhance designer involvement, icons representing agents have an internal appearance showing environment contents perceived from the agent's point of view. This internal appearance also features a set of control buttons corresponding to the control commands accepted by the agent.

As an example, Figure 4e shows a car agent's control buttons, including the wheel for steering, the polyhedron for braking, and the arrow for advancing. By assuming the internal appearance of the agent to be trained, the designer "gets in" the agent and drives it through the training scene playback, using the control buttons in response to his perception of environmental conditions. In doing so, the designer specifies by example when control commands are issued in reaction to environmental stimuli occurring in the scene.

A spatiotemporal parser automatically interprets the spatiotemporal contents of training, then translates them into XSTL assertions. To permit different levels of precision in the specifica-

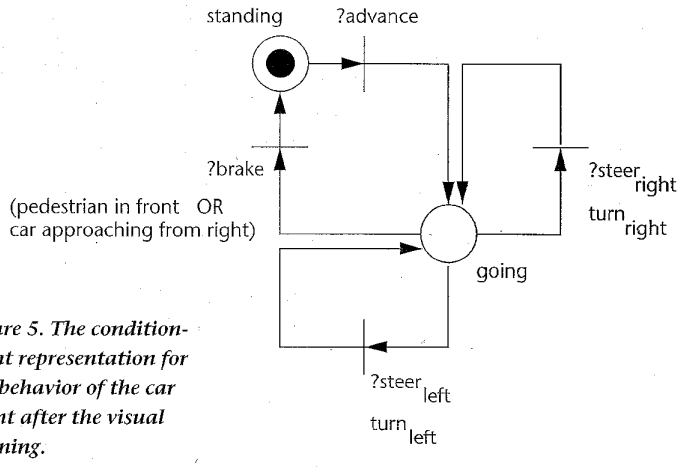

Figure 5. The condition-event representation for the behavior of the car agent after the visual training.

tion, the designer can control the level of detail accomplished in the parsing. She first clicks on the agent icons to be considered in the scene's interpretation (some agents could be included only to achieve more realistic visualizations). For each relevant object, she may then define a set of representative points to be taken into account in the evaluation of spatial relationships (the vertices of the minimum enclosing rectangle being assumed by default). Finally, a separate button interface allows the designer to define whether the scene must be interpreted considering only spatial or both spatial and temporal relationships, with or without the use of metric qualifiers.

By comparing the relation between environmental stimuli and designer-issued control commands to the agent operation model, the system completes the model with those stimuli that caused control commands to be engaged. To modify agent reaction to the occurrence of a certain stimulus, the designer simply produces a new reaction through a new visual example and selects between the change with deletion and change with addition modes. The first mode associates the stimulus with the new reaction pattern only; the second mode associates the stimulus with the new reaction pattern without removing any previous association. The latter may, however, result in a single stimulus being associated with conflicting reactions. The present version resolves these concurrences on a random basis to augment the variability of the environment. If such a random selection is not appropriate to specific purposes of the virtual environment, conflicting conditions could be detected and resolved as deterministic choices with the introduction of static priorities associated with event transitions.[9]

Figures 4, 6, and 7 present simple examples that address visual specification and operation of a virtual environment that gives driving lessons to a beginner. In these examples, a beginner drives a car within a virtual environment populated by car and pedestrian agents. Car agents are trained to behave according to some rules of "good driving," specifically, to stop when they find a pedestrian in a crosswalk and to give precedence to cars coming from their right-hand side.

Figure 4 depicts the training scene created to teach virtual car agents to stop when they encounter a pedestrian in a crosswalk.

This training results in engagement of the *brake* command and the agent's transition from *going* to *standing* mode on perceiving a pedestrian at a near distance. Figure 5 depicts this schematically.

Modifying training examples

Figure 6 shows the car agent being trained to yield to other cars coming from its right-hand side. This training modifies the behavioral model to cause the car agent to leave *going* mode and enter *standing* mode whenever it detects another car approaching its right-hand side at a near distance.

The behavioral model now includes this stimulus as a disjunctive condition associated with the brake event and leads to a transition from *going* to *standing* mode (see Figure 5). The system operates these behavior specifications automatically, letting agents run autonomously within the virtual environment. To this end, agents are associated with passive execution automata controlled by a spatiotemporal parser and an agent scheduler.

Verifying scenarios

The spatiotemporal parser observes the contents of the virtual environment, evaluates frame-by-frame the positions and instantaneous displacements of the agents, and detects the occurrence of spatiotemporal stimuli relevant to the animation. This detection is carried out through the subsequent application of a spatial and a temporal model checker, which basically follows the steps of Clarke's algorithm.[11]

At each new scene, every spatial assertion appearing in a stimulus expression is checked against spatial relationships among the objects in the scene. This checking involves a worst-case computational complexity that is linear with respect to the length of the assertion itself and cubic with respect to the number of objects referenced in the

stimulus. The cubic dependency arises from the fact that, in a 3D space, the presence of n objects determines a partitioning of the space into $(2n+1)^3$ regions, each having a different combination of walkthroughs leading to the objects in the scene.

The temporal checking is carried out by comparing sequence assertions representing agent stimuli with the scene assertions satisfied in the present scene and with the history of the environment (within a finite memory window). Since temporal operators of XSTL are oriented to past time, the contents of the present scene do not change assertions satisfied in the history. This permits temporal checking of a stimulus to be completed with a worst-case complexity that is linear in both the length of the history window and the length of the sequence assertion.

The agent scheduler interleaves repetitive calls to the agents' execution automata. On each call, the agent execution automaton checks whether the spatiotemporal parser has detected the occurrence of some stimulus relevant to the agent. It executes the possible corresponding events and then executes the active modes of the agent one step further.

Combining scenarios to create interactive sessions

Figure 7 (next page) shows scenes from an interactive session of a beginning driver. The virtual environment enforces the rules defined by the two previous specification examples.

Conclusions

Our framework supports the use of visual programming by example in the creation of VR agent animations. The designer replicates virtual agents' expected responses to stimuli that occur in defined spatiotemporal relationships with other

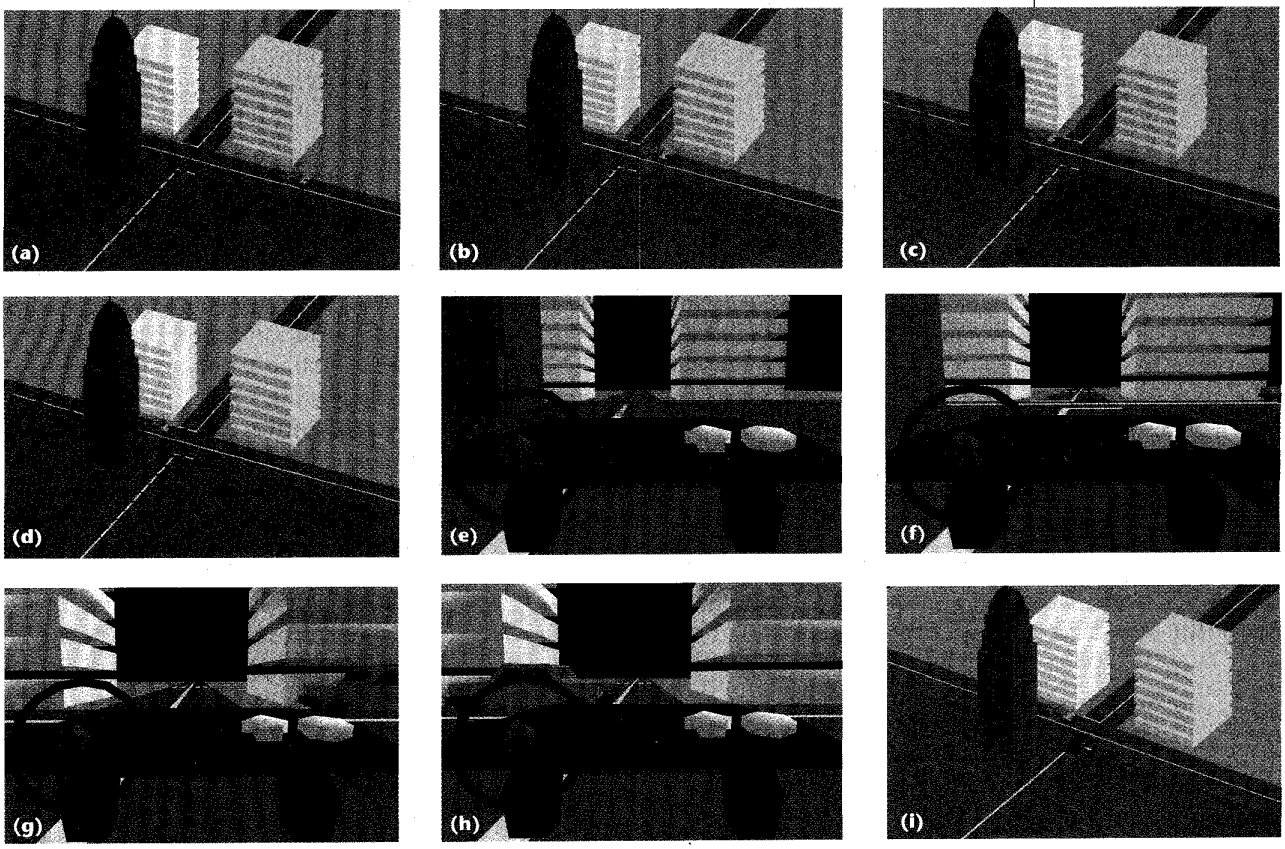

Figure 6. Visual programming of a car agent: behavior in the presence of a car at a crossroad. A training scene is composed in which a background car agent goes through a crossroad (a through c). Again, during the playback of the background scene, the designer "gets in" the car (d and e) and drives it toward the crossroad (f). When she sees the background car getting close to the crossroad on her right-hand side, she pushes the brake button (g), lets the car pass (h), and advances again (i).

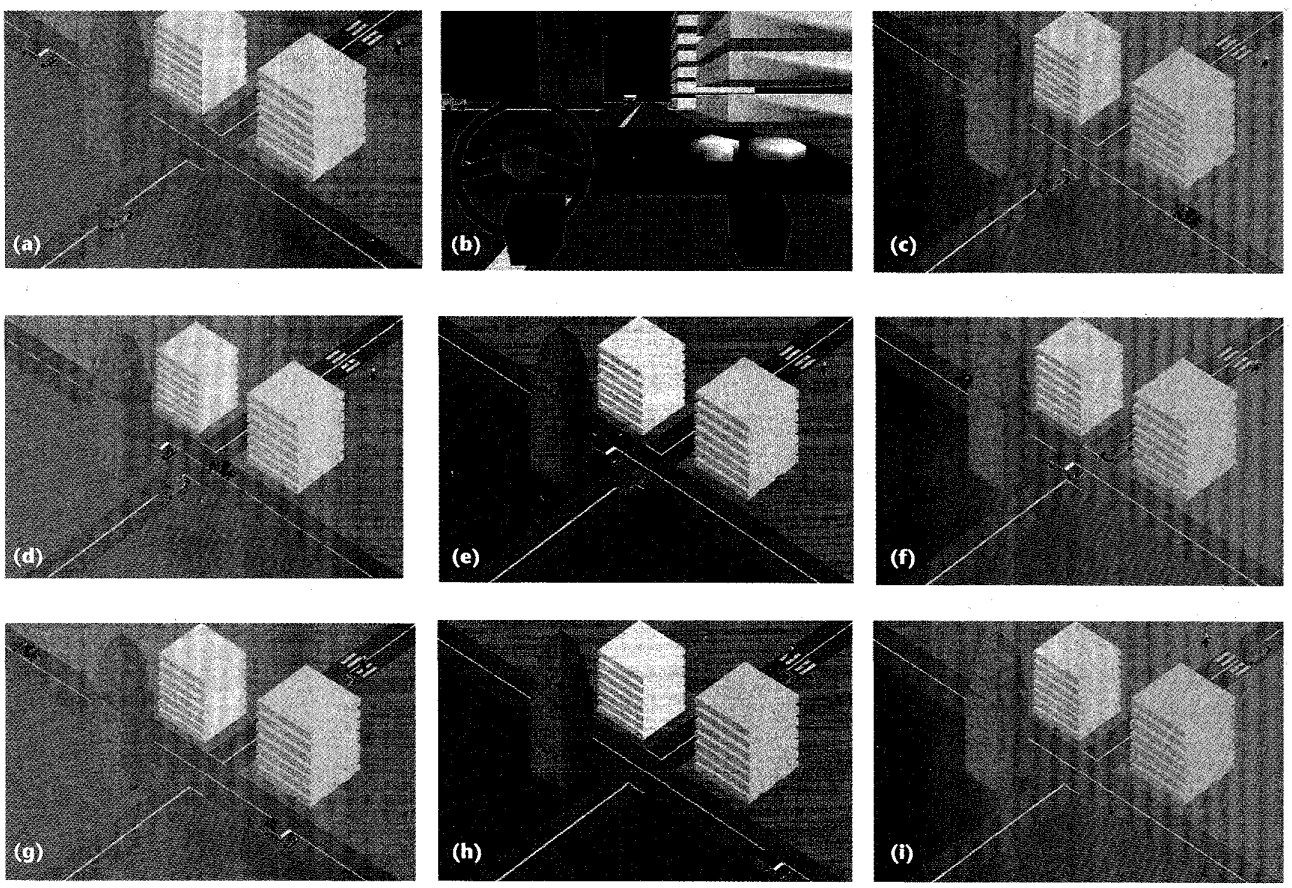

Figure 7. Interaction with trained agents in the virtual environment. The beginner drives the red car along the road on the right-hand side of the screen; a trained blue car comes from the opposite direction, and a trained grey car comes from the left-hand side (a and b). When the trained grey car reaches the crossroad, it perceives the red beginner car approaching from its right-hand side and stops to give precedence (c), as learned during the previous training stage. While the grey car waits for the red car to advance, the trained blue car approaches the crossroad and stops to give precedence to the grey car (d). After the beginner car has passed through the crossroad, the trained grey car advances through the crossroad (e), then the blue car advances when the crossroad is clear (f). Meanwhile, the trained grey car continues until it perceives a pedestrian in a crosswalk (g). Again, according to previous training, the car stops and does not restart until the pedestrian has left the crosswalk (h and i).

agents in the virtual world. Stimuli and responses occurring in the training examples are captured into an internal representation, which combines a Petri net operational model of reaction sequences with a spatiotemporal description of environmental stimuli.

During execution of the virtual environment, each agent's internal representation is automatically operated by an interpreter that executes the operational part of the specification and a model checker that compares the featured scenes against descriptions of stimuli the agents perceive. This approach permits direct engagement of human 3D perceptual and spatial skills without requiring designers to cast their empirical knowledge of the emulated environment into a textual specification. It also supports a more effective design of agent animations and permits even nonprogrammer end users to modify agent behaviors and tailor applications to specific needs.

System developments can focus on augmenting the expressivity of the internal representation so that designers can expand the range of agent behaviors with explicit priorities, timing constraints, and parametrized actions. While timing assumptions do not create problems from a formal point of view, their enforcement requires a real-time execution platform to operate specification models.

Similarly, parametrized actions basically do not affect the automatic operability of specification models. The finer granularity of behavior repre-

sentation, however, hinders derivation of general rules from interpretation of single examples. Finally, effective parameter setting requires that specification be completed with an annotation stage. This might reduce the immediacy of specification by example, but it also enables more expressivity within the formalism. **MM**

Acknowledgments

This work was partially supported by the Italian Ministry of the University and of Scientific and Technologic Research (MURST) under the project "Sviluppo di una Workstation Multimediale ad Architettura Parallela."

References

1. *Virtual Reality: Theory, Practice and Promise*, S.K. Helsel and J.P. Roth, eds., Meckler Pub., Westport, London, U.K., 1991.
2. A.J. West et al., "Aviary—A Generic Virtual Reality Interface for Real Applications," Tech. Report 1992, AIG Dept. Computer Science, Univ. of Manchester, Manchester, U.K., 1992.
3. C. Shaw et al., "Decoupled Simulation in Virtual Reality with the MR Toolkit," *ACM Trans. Information Systems*, Vol. 11, No. 3, July 1993, pp. 287-317.
4. Virtual Worlds Group, IBM T.J. Watson Research Center, *Virtual Reality Distributed Environment and Construction Kit (VR-DECK): User's Guide*, Yorktown Heights, N.Y., May 1993.
5. A. Repenning and W. Citrin, "Agentsheets: Applying Grid-Based Spatial Reasoning to Human-Computer Interaction," *Proc. IEEE Workshop on Visual Languages*, CS Press, Los Alamitos, Calif., 1993, pp. 77-82.
6. D. Canfield Smith, A. Cypher, and J.C. Spohrer, "Kidsim: Programming Agents Without a Programming Language," *Comm. ACM*, Vol. 37, No. 7, July 1994, pp. 54-67.
7. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, New York, 1992.
8. A. Del Bimbo, E. Vicario, and D. Zingoni, "Symbolic Description and Visual Querying of Image Sequences Using Spatio Temporal Logic," *IEEE Trans. Knowledge and Data Engineering*, Vol. 7, No. 4, Aug. 1995, pp. 609-622.
9. T. Murata, "Petri Nets: Properties, Analysis, and Applications," *Proc. IEEE*, Vol. 77, No. 4, Apr. 1989, pp. 541-580.
10. K. Jensen, "Coloured Petri Nets: A High Level Language for System Design and Analysis," in *High-Level Petri Nets: Theory and Applications*, K. Jensen and G. Rozenberg, eds., Springer Verlag, Berlin, 1991, pp. 44-119.
11. E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems*, Vol. 8, No. 2, Apr. 1986, pp. 244-263.

Alberto Del Bimbo is a professor of computer systems at the Università di Brescia and at the Università di Firenze, Italy. He received a doctoral degree in electronic engineering from the Università di Firenze, Italy, in 1977. His research interests and activities include image analysis, image databases, visual languages, and virtual reality. He is a member of the IEEE and of the International Association for Pattern Recognition (IAPR). He is on the board of the IAPR Technical Committee No. 8 (Industrial Applications) and is vice president of the IAPR Italian Chapter. He presently serves as associate editor of *Pattern Recognition Journal* and of the *Journal of Visual Languages and Computing*.

Enrico Vicario received a doctoral degree in electronic engineering and a PhD in computer science from the Università di Firenze, Italy, in 1990 and 1994, respectively. He is currently a researcher at the Dipartimento di Sistemi e Informatica of the Università di Firenze, Italy. His research activities include software engineering, with a particular interest in visual formalisms, specification languages, and validation techniques for time-dependent systems.

Contact the authors at the Dipartimento di Sistemi e Informatica, Università di Firenze, 3 via Santa Marta, 50139, Firenze, Italy, e-mail {delbimbo,vicario}@aguirre.ing.unifi.it.