In Journal of Visual Languages and Computing, 13(2):177-195, 2002

Modeling Behaviors of Interactive Objects for Real Time Virtual Environments

MARCELO KALLMANN AND DANIEL THALMANN

Swiss Federal Institute of Technology
Computer Graphics Lab – LIG
Lausanne, Switzerland, CH-1015
Marcelo.Kallmann@epfl.ch
Daniel.Thalmann@epfl.ch

Abstract

Real-time 3D graphics are being extensively used to build interactive virtual environments for a number of different applications. In many situations, virtual objects are required to exhibit complex behaviors, and the use of a versatile behavioral programming methodology is required.

This paper presents a feature modeling approach to define behavioral information of virtual objects, giving special attention to their capabilities of interaction with virtual human actors.

The proposed method is based on a complete definition and representation of interactive objects, which are called *smart objects*. This representation is based on the description of *interaction features*: parts, movements, graspable sites, functionalities, purposes, etc, including a behavioral automata based on scripts and graphical state machines.

Keywords: Visual Programming, Object Interaction, Autonomous Agents, Virtual Environments.

1. Introduction

Behavior animation techniques [1,2] are often used to animate virtual environments (VEs). This *bottom-up* approach for artificial intelligence appeared from the field of behavioral-based robotics [3], and the term *agent* [4] has been used to refer to behavioral entities based on perception and action.

The agent technology has been widely used together with computer graphics and virtual reality for the construction of real-time simulations with interactive and intelligent virtual entities. The necessity to model actor-object interactions in VEs appears in most applications of

computer animation and simulation. Such applications encompass several domains, as for example: human factors analysis, training, education, virtual prototyping, simulation-based design, and videogames. A good overview of such areas is presented by Badler [5], and one example of a training application is described by Jhonson et al. [6].

Commonly, simulation systems approach actor-object interactions by programming them specifically for each case. Such approach is simple and direct, but does not solve the problem for a wide range of cases. Another approach, not yet fully solved, is to use artificial intelligence (AI) techniques, as recognition, planning, reasoning and learning, in order to decide and determine the many manipulation variables during an actor-object interaction. The actor's knowledge is then used to solve all possible interactions with an object. Such *top-down AI approach* should also address the problem of interaction with more complex machines having some internal proper functionality, in which case information regarding the object functionality must be provided.

Consider the simple example of opening a door: the rotation movement of the door must be provided a priori. Following the top-down AI approach, all other actions should be planned by the agent's knowledge: walking to reach the door, searching for the knob, deciding which hand to use, moving body limbs to reach the knob, deciding which hand posture to use, grasping, turning the knob, and finally opening the door. This simple example illustrates how complex it can be to perform a simple interaction task.

This paper presents a behavioral animation approach, which includes within the object description, more useful information than only intrinsic object properties. Using feature modeling concepts (mostly from CAD/CAM applications [7]), all interaction features of an object are identified and included as part of the object description. Visual programming techniques are employed within a graphical interface system, which permits the user to interactively specify all different interaction features of an object, as its functionality, available interactions, etc. Objects modeled with their interaction features description are called *smart objects*. The smart object modeler (SOMOD) was developed for this matter and is also described in this paper.

Special attention is given for the interaction capabilities with virtual human actors. In the smart object representation, actor-object interactions are specified with pre-defined plans. Such plans access object related geometric parameters and describe both the object functionality and the expected actor behaviors during interactions. A higher layer based on a graphical state machine can be used to specify connections between interaction plans.

In this way, pre-defined interaction plans are designed during the modeling phase of the object, allowing fast execution during simulations. A plan can be seen as a scheme or program determining a sequence of actions to be taken in order to accomplish a given goal. Actions are then considered units of behavior, and thus a plan can define a behavior. In this way, the designer of the object uses his/her own knowledge about how to use the object by defining simulation parameters during the modeling phase.

In this work, both actors and objects are considered to be agents. Autonomous agents are often classified as *reactive* or *intelligent*, basically depending on the complexity of their behaviors. In the scope of this paper, an object is called *smart* when it has the ability to describe in details its functionality and its possible interactions, being also able to give all the expected low-level manipulation actions. This can be seen as a mid term classification between reactive and intelligent behaviors. A smart object is then considered to be an agent with not only reactive

behaviors, but more than that, it is able to provide the expected behaviors of its *users*, so that this extra capability makes it to achieve the quality of *smart*.

Although this paper focuses mainly on the actor-object interaction case, the general term *user of an object* is used, in order to cover interactions with different kind of entities, such as real people immersed in the VE wearing virtual reality devices, or users controlling a pointing device in a standard desktop configuration. The main idea is to provide smart objects with a maximum of information in order to deal with different possible applications for the object. A parallel with the object oriented programming paradigm [8] can be done, in the sense that each object encapsulates data and provides methods for data access.

2. Related Work

Object interaction in virtual environments is an active topic and many approaches are available in the literature. However, in most cases, the concerned topic is the direct interaction between the user and the environment [9,10,11], which does not consider objects having some proper behavior. Some systems focus on the functionality of objects [12], however actor-object interactions are rarely considered.

Actor-object interaction techniques were first specifically addressed in a simulator based on natural language instructions using an *object specific reasoning* (OSR) module [13]. However, it mainly focuses grasping tasks, and no considerations are done for the interaction with more complex objects.

Robotics techniques have been employed for specific sub-problems, as for automatic grasping of geometrical primitives [14], which is based on a pre-classification of most used hand configurations for grasping [15]. Planning algorithms for determining collision free paths of articulated arms have also been developed for manipulation tasks [16], but still, the computational cost is too high for interactive simulations.

Connecting modeling and simulation parameters is a key issue for defining object interaction and functionality, and modern CAD systems are starting to integrate simulation parameters in their models [17]. When object functionality becomes complex, specific behavioral programming techniques are used, and many approaches have been proposed. The fact is that the implementation of behavioral modules is a highly context-dependent task, so that when applying standard techniques to a specific domain, many specific issues need to be differently solved. This situation leads to many specific systems being described in the literature, but the techniques involved do not vary significantly.

The most popular techniques use *rule-based behaviors*: according to the system state, rules can be selected to evolve the simulation. Rules can be organized in different ways and their activation may be based on an *action selection* mechanism [18].

State machines are also widely used, but with the graphical representation advantage. Graphical programming methods can thus be introduced to the behavioral programming task. Initially, they seem to be a promising approach, but for most complex systems with a lot of states and transitions, they easily turn to be a difficult representation to construct, understand and maintain. For instance, only coherently drawing state machine graphs can be a complex task [19]. Many systems use some kind of finite state machine to define behaviors [20,21]. Moreover,

specific solutions for the animation of virtual actors have been proposed concerning parallel execution [22] and resources allocation [23].

Alternatively, scripts are also used to program behaviors, as in the IMPROV system [24]. Interpreted programming languages can be also used as scripting tools, as for instance the Python language [25].

3. Feature Modeling of Interactive Objects

Feature modeling is an expanding topic in the engineering field [7]. The word *feature* may suggest different ideas, and a definition can be simply: a region of interest on the surface of a part [26]. The main difficulty here is that, in trying to be general enough to cover all reasonable possibilities for a feature, such a definition fails to clarify things sufficiently.

From the engineering point of view, it is possible to classify features in three main areas: functional features, design features and manufacturing features [31]. As we progress from functional features to manufacturing features, the quality of detail that must be supplied or deduced increases markedly. In the other hand, the utility of the feature definitions to the target application decreases. For example, manufacturing features of a piece may be hard to describe and have little importance while really using the piece. A similar compromise arises in the smart object case, as explained later in figure 1.

3.1. Interaction Features

In the smart object description, a new class of features for simulation purposes is proposed: *interaction features*. In such context, a more precise idea of a feature can be given as follows: all parts, movements and descriptions of an object that have some important role when interacting with an actor. For example, not only buttons, drawers and doors are considered as interaction features in an object, but also their movements, purposes, manipulation details, etc.

Interaction features can be grouped in four different classes:

- Intrinsic object properties: properties that are part of the object design, for example: the movement description of its moving parts, physical properties such as weight and center of mass, and also a text description for identifying general objects purpose and the design intent.

- Interaction information: useful to aid an actor to perform each possible interaction with the object. For example: the identification of interaction parts (like a knob or a button), specific manipulation information (hand shape, approach direction), suitable actor positioning, description of object movements that affect the actor's position (as for a lift), etc.

- Object behavior: to describe the reaction of the object for each performed interaction. An object can have various different behaviors, which may or may not be available, depending on its state. For example, a printer object will have the "print" behavior available only if its internal state variable "power on" is true. Describing object's behaviors is the same as defining the overall object functionality.

• Expected actor behavior: associated with each object behavior, it is useful to have a description of some expected actor behaviors in order to accomplish the interaction. For example, before opening a drawer, the actor is expected to be in a suitable position so that the drawer will not collide with the actor when opening. Such suitable position is then proposed to the actor during the interaction.

This classification covers most common actor-object interactions. However, many design choices still appear when trying to specify in details each needed interaction feature, in particular concerning features related to behavioral descriptions. Behavioral features are herein specified using pre-defined plans composed with primitive behavioral instructions. This has proved to be the most straightforward approach because then, to perform an interaction, the actor will only need to "know" how to interpret such interaction plans.

In the smart object description, a total of eight interaction features are identified, which are described in table 1.

Feature	*Class*	*Data Contained*
Descriptions	Object Property	Contains text explanations about the object: semantic properties, purposes, design intent, and any general information.
Parts	Object Property	Describes the geometry of each component part of the object, their hierarchy, positioning and physical properties.
Actions	Object Property	Actions define movements, and any other changes that the object may undertake, as color changing, texture, etc.
Commands	Interaction Info.	Commands parameterize and associate to a specific part the defined actions. For example, commands *open* and *close* can use the same translation movement.
Positions	Interaction Info.	Positions are used for different purposes, as for defining regions for collision avoidance and to suggest suitable places for actors during interactions.
Gestures	Interaction Info.	Gestures are any movement to suggest to an actor. Mainly, hand shapes and locations for grasping and manipulation are defined here, but also specification of other actions or pre-recorded motions can be defined.
Variables	Object Behavior	Variables are generally used by the behavioral plans, mainly to define the state of the object.
Behaviors	Obj./Actor Behavior	Behaviors are defined with plans composed of primitive instructions. Plans can check or change states, trigger commands and gestures, call other plans, etc; specifying both object behaviors and expected actors' behaviors. These plans form the actor-object communication language during interactions.

Table 1 – The eight types of interaction features used in the smart object description.

3.2. Interpreting Interaction Features

Once a smart object is modeled, a simulation system is able to load and animate it in a VE. For this, the simulator needs to implement a *smart object reasoning module*, to correctly interpret the behavioral plans to perform interactions. For example, a VR application in which the user wears a virtual glove to press a button of a smart object will not make the same use of proposed

hand shapes. In a previous work [27], an application to test interactions between a data glove and smart objects is described.

There is a trade-off when choosing which features to be considered in an application. As shown in figure 1, when taking into account the full set of object features, less reasoning computation is needed, but less general results are obtained. As an example, minimum computation is needed to have an actor passing through a door following strictly a proposed path to walk. However, such solution would not be general in the sense that all agents would pass the door using exactly the same path. To achieve better results, external parameters should also take effect, as for example, the current actor emotional state.

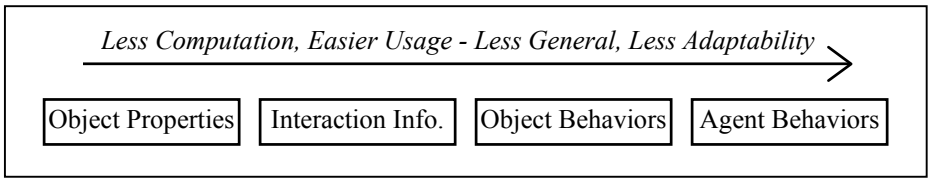

Figure 1 – The choice of which interaction features to take into account is directly related to many implementation issues in the simulation system.

Note that the notion of a realistic result is a context dependent notion. For example, pre-defined paths and hand shapes can make an actor to manipulate an object very realistically. However, in a context where many actors are manipulating such objects exactly in the same way, the overall result is not realistic.

Interaction plans form the interface between stored object's features and the application specific smart object reasoning. Figure 2 illustrates the connection between the modules. The simulation program requires a desired task to be performed. The reasoning module will then search for suitable available behaviors in the smart object. For any selected behavior, the reasoning module follows and executes each instruction of the behavior plan, retrieving the needed data from the smart object representation.

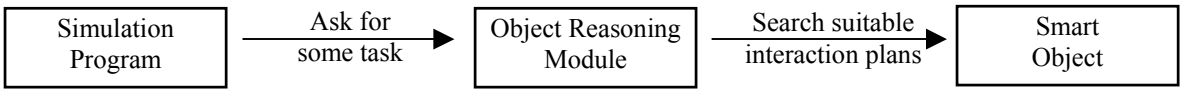

Figure 2 - Diagram showing the connection between the modules of a typical smart object application. Arrows represent function calls.

When a task to perform becomes more complex, it can be divided into smaller tasks. This work of dividing a task into sub-tasks can be done in the simulation program or in the reasoning module. In fact, the logical approach is to leave the reasoning module only to perform tasks that have a direct interpretation from the smart object behaviors. Then, additional layers of planning modules can be built according to the simulation program goal.

Another design choice appears while modeling objects with too many potential interactions. This issue is related to definition of the component parts of a composed object. In such cases it is possible to model the object as many independent smart objects, each one containing only basic interactions. For example, to have an actor interacting with a car, the car

can be modeled as a combination of different smart objects: car door, radio, and the car panel. In this way, the simulation application can explicitly control a sequence of actions like: opening the car door, entering inside, turning on the radio, and starting the engine. On the other hand, if the simulation program is concerned only with traffic simulation, the way an agent enters the car may not be important. In this case, a general behavior of entering the car can be encapsulated in a single smart object car. A similar design choice is presented later in the paper with a smart lift, which can be created having only the global interaction plan "enter", or many decomposed plans: "press", "go in", "go out", etc.

The smart object approach introduces the following main characteristics in a simulation system:

- Decentralization of the animation control. Object interaction information is stored in the objects, and can be loaded as plug-ins, so that most object-specific computation is released from the main animation control.

- Reusability of designed smart objects. Not only by using the same smart object in different applications, but also to design new objects by merging any desired feature from previously designed smart objects.

- A simulation-based design is naturally achieved. Designers can take full control of the loop: design, test and re-design. Designed smart objects can be easily connected with a simulation program, to get feedback for improvements in the design.

4. SOMOD Description

The SOMOD application was developed specifically to model smart objects. It permits to import geometric models of the component parts of an object, and then to specify interactively all needed interaction features, using a graphical user interface. Even for the definition of the behavioral plans, a specific dialog box was designed to consistently guide the selection of the possible parameters to specify for each primitive instruction. Also, a graphical state machine can de used for defining higher-level behaviors.

4.1. Object Properties

Features are organized by type, and specialized dialog boxes are used to edit them. Many properties can be defined with text descriptions as a semantic name for the object or any other characteristic, which can then be retrieved by simulators for any kind of processing. Geometric files to compose an object can be loaded and positioned to create a smart object (figure 3). Movements to apply in parts, i.e. actions, are also defined interactively with graphical manipulators.

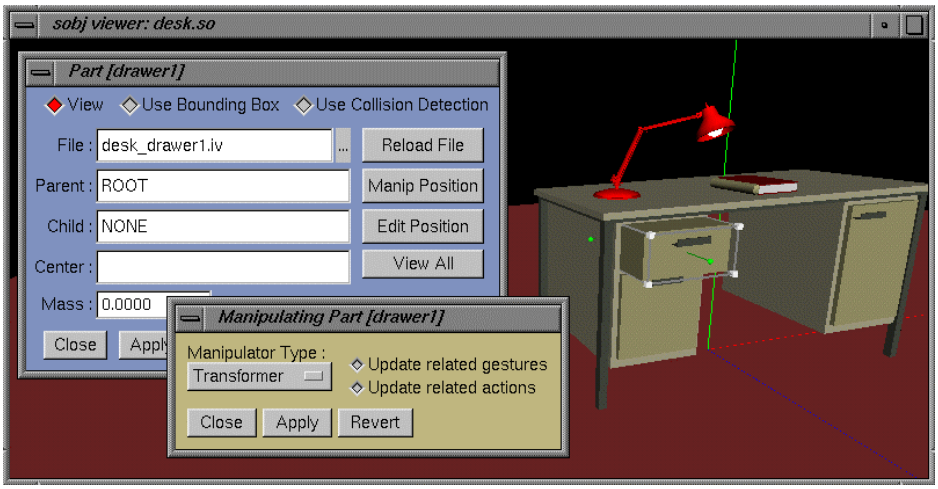

Figure 3 – Defining the specific parameters of a drawer. The drawer is a part of the smart object desk, which contains many other parts. The image shows in particular the positioning of the drawer in relation to the whole object.

4.2. Interaction Information

Commands are simply specified as a parameterization over an action, permitting to use a same action in reverse order, at different speeds, and applied to different object parts. Positions are also graphically defined and they can be used for many different purposes. Figure 4 shows as example the used positions for entering an automatic door.

The ability to interactively define graphical features is very important, as it is the only way to see and evaluate their relative location to the object. Features are defined in relation to the object's hierarchy root, so that they can be transformed to the same reference frame of the actor whenever is needed during the simulation.

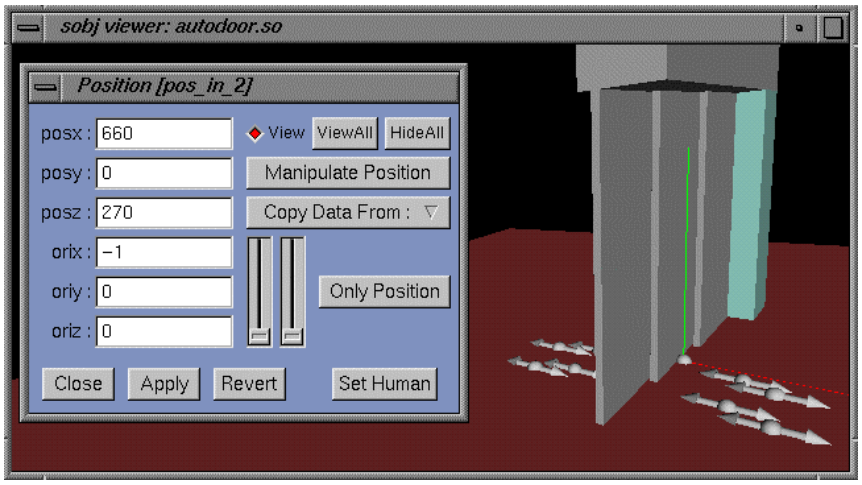

Figure 4 – Positions can be defined for any purpose. In particular, the image shows many different positions (and orientations) which are placed to propose different possible locations for actors to walk when arriving from any of the door sides.

Gestures are the most important interaction information, as they are mainly related to define manipulations with the object. We use the term gesture in order to refer to any kind of motion that an actor is able to perform. The most used gesture is to move the hand towards a pre-defined position and orientation in space, for example, to press a button (a *push* movement), or to grasp something. As example, actual possible parameters for the gesture feature are shown in figure 5.

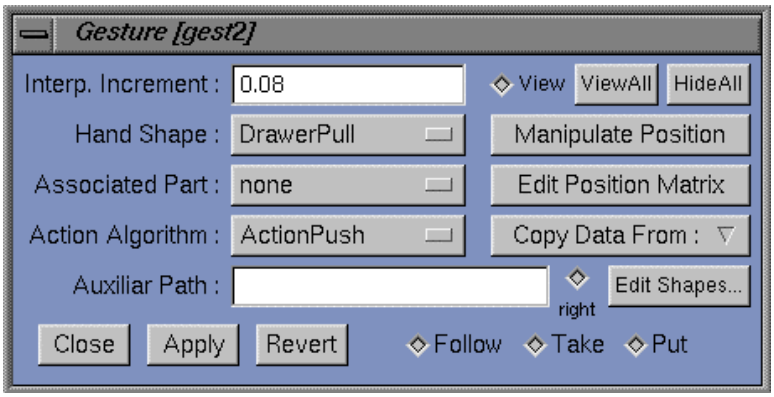

Figure 5 – Gestures parameters dialog box. Depending on the action algorithm chosen, some parameters may be used differently. In the image, the action algorithm *push* is selected.

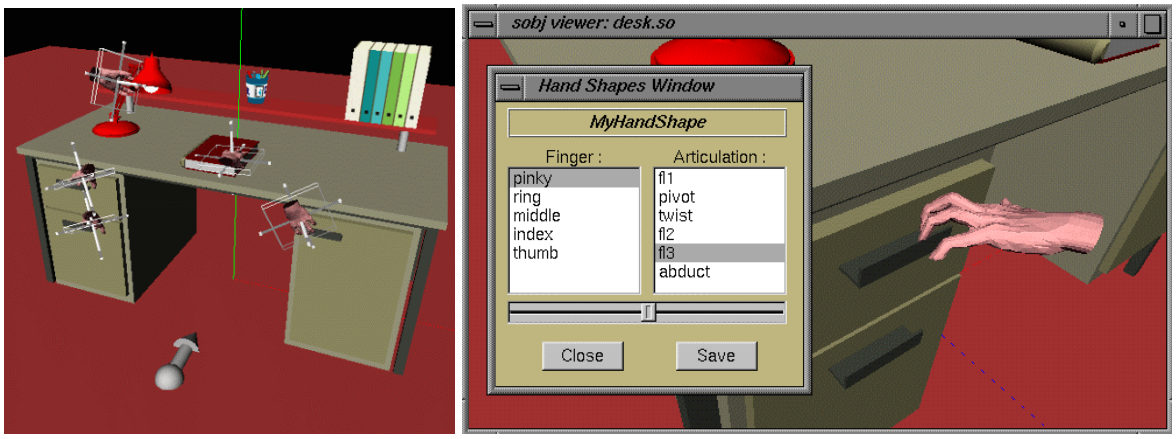

Figure 6 –The left image shows all used hand shapes being interactively located with manipulators. The right image shows a hand shape being interactively defined.

For each gesture, a hand shape (see figure 6), a positioning matrix for the hand, and the desired action algorithm must be supplied. Depending on the action algorithm, some extra parameters may be required. As the possible actions depend directly on the capabilities of the animation system, configurable descriptive files are used: when SOMOD starts, a special folder is scanned where files define each supported action algorithm in the target animation system. For example, the ACE system [28], which is based on the agentlib [29], has full animation support for smart objects, and the most used gesture actions are: reach, push, and sit; all of them based on an inverse kinematics module specifically designed for the animation of virtual human actors [30].

In short, the action reach only animates an actor's arm to put its hand in a given location. Depending on the state of extra parameters, after the hand has reached the defined goal, the actor can then take or put an associated part. The push action differs in two aspects: it is able to animate the whole actor's body in order to achieve better postures, and the actor's hand can then follow a movement of an associated part in order to simulate the movement of opening, pressing, pushing, etc. The sit action will animate the actor to sit, using a target position to reach with a column central skeletal joint. All these actions use inverse kinematics as the motion motor. These motion motors are correctly initialized and blended during transitions by the simulation system.

4.3. Behaviors

As already explained, behaviors are defined using pre-defined plans formed by primitive instructions. It is difficult to define a closed and sufficient set of instructions to use. Moreover, a complex script language to describe behaviors is not the goal. The idea is to keep a simple format with a direct interpretation to serve as guidance for reasoning algorithms, and which non-programmers can create and test.

A first feature to recognize in an interactive object is its possible states. States are directly related to the behaviors one wants to model for the object. For instance, a desk object will typically have a variable state for its drawer, which can be assigned two values: "open", or "close". However, depending on the context, it may be needed to consider another midterm state

value. Variables are used to keep the states of the object and can be freely defined by the user to approach many different situations. Interaction plans can then be created (figure 7), following some key concepts:

• An interaction plan describes both the behavior of the object and the expected behavior of its user. Examples of some user instructions are: UserGoto, UserDoGest, UserAttachTo, etc. In a previous work [31], a complete list of the available primitive instructions in SOMOD is given.

• In SOMOD, an interaction plan is also called as a behavior. Many plans (or behaviors) can be defined and they can call each other, as subroutines. Like programming, this enables building complex behaviors based on simpler behaviors.

• There are three types of behaviors (or plans): *private*, *object control*, and *user selectable*. Private behaviors can only be called from other behaviors. An object control behavior is a plan that is interpreted all the time since the object is loaded in the virtual environment. This permits having objects acting like agents, for example sensing the environment to trigger some other behavior, or to have a continuous motion as for a ventilator. Finally, user selectable behaviors are those that can be selected by users, in order to perform a desired interaction. Only selectable behaviors can be accessed by users.

• Selectable behaviors can be available or not, depending on the state of specified variables. For example for a door, one can design two behaviors: to open and to close the door. However, only one is available at a time depending on the open state of the door. The instruction CheckVar is used to control the availability of behaviors. When the CheckVar test is false the behavior is not available for selection.

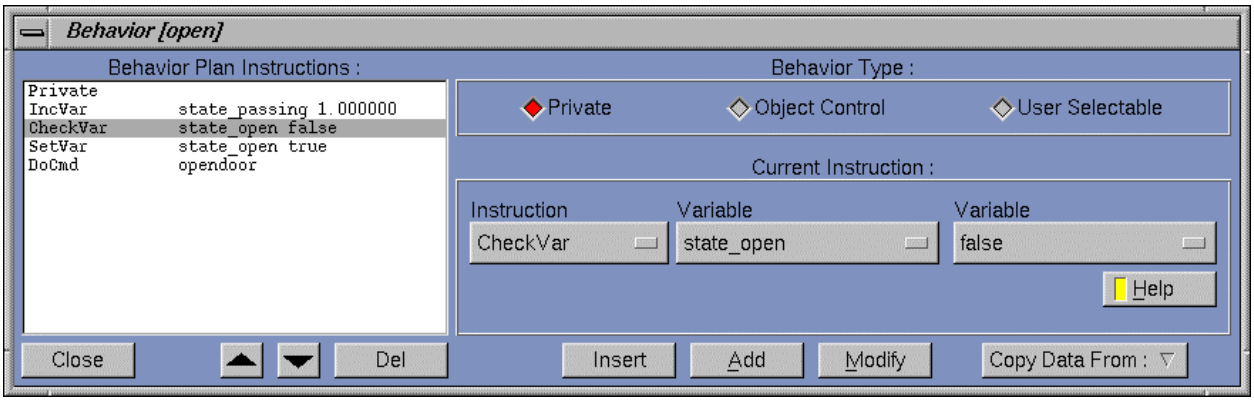

Figure 7 – Plans can be defined with a menu-based interface where all possible instructions and parameters are easily chosen.

As an example, the behaviors used in the automatic door example of figure 7 are reproduced below:

```
BEHAVIOR open
  Private
  IncVar   state_passing 1.00
  CheckVar state_open false
  SetVar   state_open true
  DoCmd    opendoor

BEHAVIOR close
  Private
  IncVar   state_passing -1.00
  CheckVar state_open true
  CheckVar state_passing false
  SetVar   state_open false
  DoCmd    closedoor
```

```
BEHAVIOR go_1_2
  Private
  UserGoTo pos_in
  DoBh     open
  UserGoTo pos_out
  DoBh     close

BEHAVIOR go_2_1
  Private
  UserGoTo pos_in_2
  DoBh     open
  UserGoTo pos_out_2
  DoBh     close
```

```
BEHAVIOR enter
  UserGetClosest tmp pos_in,pos_in_2
  If             tmp==pos_in
  DoBh           go_1_2
  Else
  DoBh           go_2_1
  EndIf
```

In this example, the user of the object has access only to the behavior “enter”, which detects in which side of the door the actor is, and calls the correct private behavior accordingly.

In order to cope with many actors at the same time, two details are taken into account by these behavioral plans: First, the “state_passing” variable counts the number of actors actually passing through the door, and is used to avoid closing the door while an actor is still passing. Another thing is that many different positions were defined for this example but having the same reference name. So that, for instance, when the instruction “UserGoTo pos_out” is interpreted, the system is able to choose, among all positions with the same name “pos_out”, one that is not yet occupied. In figure 8, two actors are interpreting in parallel the same interaction plan, in order to correctly enter the door.

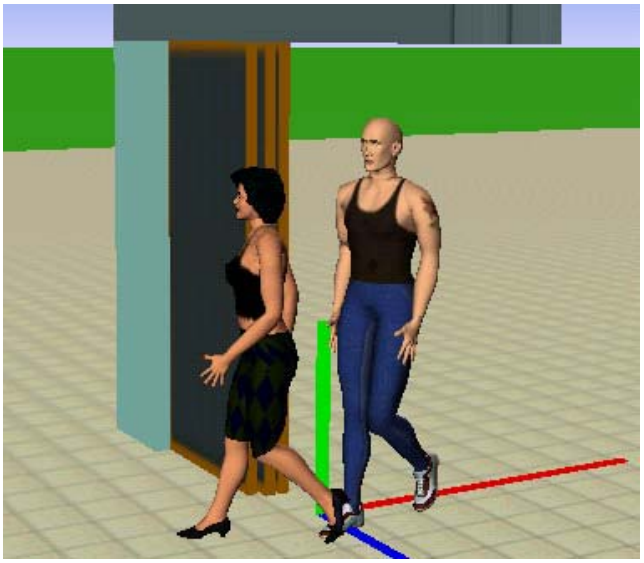

Figure 8 – Two actors passing through an automatic door. The used interaction plans can correctly manage more than one actor interaction at the same time.

SOMOD plans are suitable for describing simple interactions and functionalities. They can still cope with much more complex cases, but then plans start to get more complex to design. It is like trying to use a specific purpose language to solve any kind of problems.

As an example, consider the case of modeling a two-stages lift, composed of many parts as: doors, calling buttons, the cabin, and the lift itself. These parts need to have synchronized movements, and many details need to be taken into account in order to correctly control actors interacting with the lift.

To simplify modeling the behaviors of such complex objects, SOMOD has the extra ability to graphically design finite state machines. The proposed solution is to start designing basic interaction plans for each components of the lift, using the standard behavior editor of figure 7. Then, when the components have their functionality defined, a state machine can be used, permitting to define the states of the whole object lift, and the connections between the component behaviors.

Figure 9 shows a state machine example for this lift case. The user has first designed the plans for the functionality of each component part in particular. For example, there are behaviors to open and close each door of the lift, to press each calling button, to move the cabin, and so on. Then, the user defines in the state machine editor only two global states: “floor_1” and “floor_2”. When the lift is in “floor_1”, the only possible interaction is “enter_12”, that will call a private behavior which calls the full sequence of instructions to perform the full interaction: pressing the calling button, opening the door, entering inside, closing the door, move the cabin up, opening the other door, and going out.

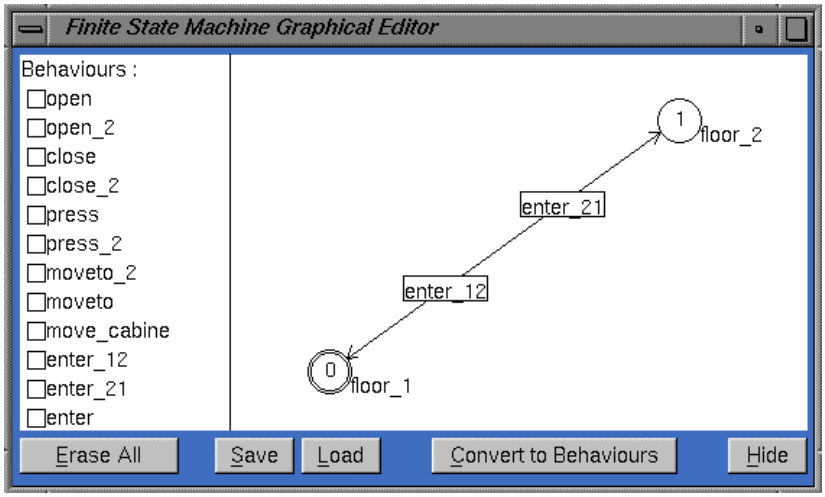

Figure 9 – A state machine for a lift functionality where all interaction during the process of taking the lift are programmed inside plans enter_12 and enter_21. In the image, the double circle state is the current state, and the rectangular boxes show the interaction needed to change of state. For example, to change from floor_1 to floor_2 state, interaction enter_12 is required.

Designed state machines are automatically translated into interaction plans so that, from the simulator point of view, all behaviors are treated as plans. When the smart lift is loaded in a simulation environment, the created available behaviors can then be selected.

4.4. Templates

Another important utility available in SOMOD is to load template objects. The idea is that any pre-modeled smart object can serve as a template to model new smart objects. A specific window to load templates was designed permitting to scan directories containing smart objects and to choose the desired features to import (figure 10).

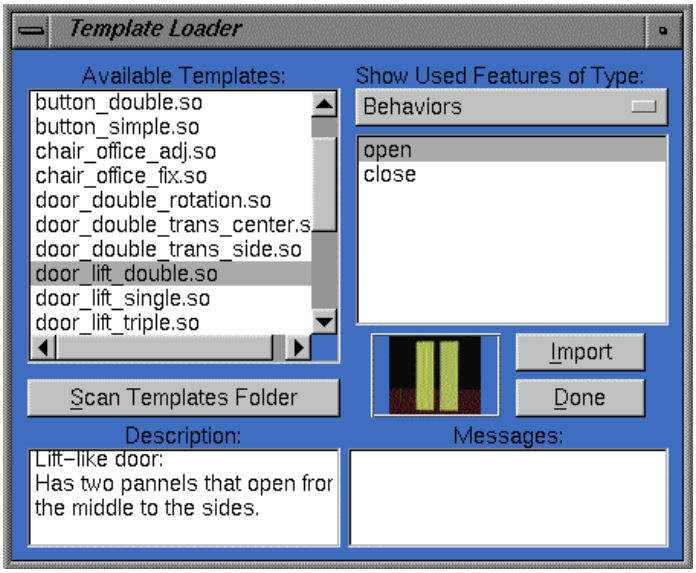

Figure 10 – The template loader permits to scan directories with previously modeled objects and components, visualize their internal features, and import any selected set of features. In the image, a set of different types of doors can be selected, each one having a different geometry or functionality, as double or single panels, center or side opening, translation or rotation opening, etc. For instance, these doors can be easily imported to test different configurations for the lift shown in figure 12.

The template loader functionality associated with the graphical state machine editor and with ACE, forms an effective way of definition, test and reuse of object interactivity. Sets of components can be maintained in proper folders, allowing testing different combinations of components, and achieving a simulation-based design system.

5. Analysis

SOMOD proposes to program low-level object behaviors using programmed plans, which can be graphically composed to create more complex functionalities. However, there is always a design choice regarding how far behaviors should go in detail to describe interactions or not.

In the lift behavior example of figure 9, there is a single interaction of entering the lift, which can be too long, giving no options to the actor during the interaction. Figure 11 shows a more complex state machine, which models the functionality of the lift by taking into account possible intermediate states. In this case, the actor needs to select, step by step, a sequence of interactions in order to take the lift to the other floor. Figure 12 shows some snapshots of the animation sequence of an actor entering the lift.

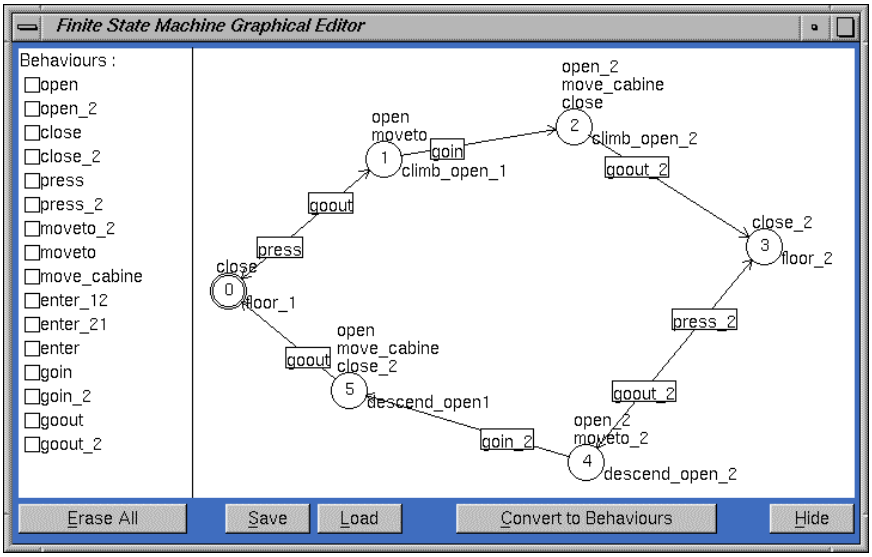

Figure 11 – A more complex state machine for the lift where intermediate states are considered. The figure also shows that behaviors are associated with states, to be triggered whenever the object enters that state.

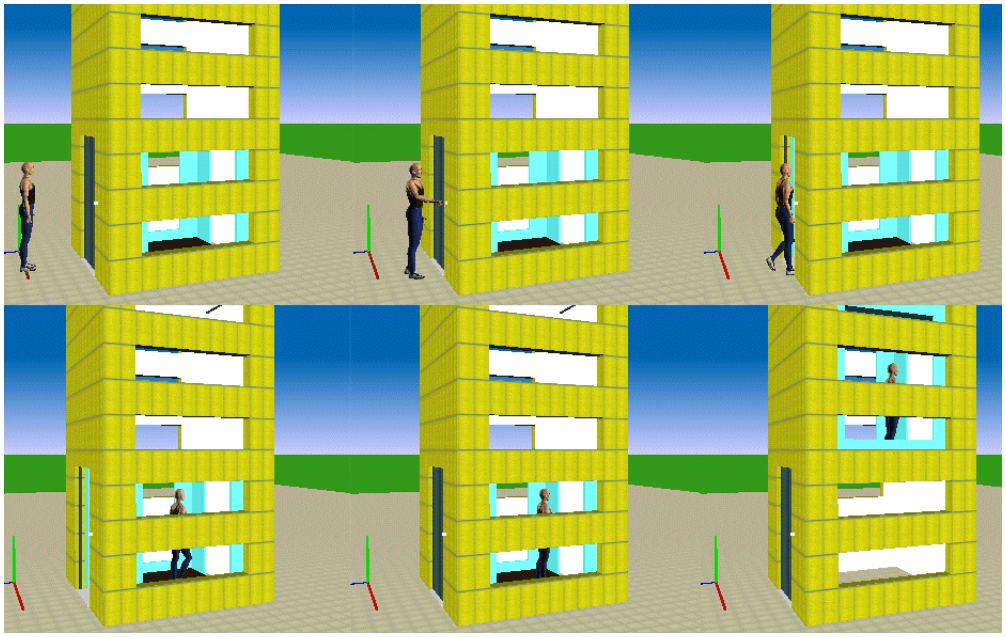

Figure 12 – An actor entering the smart lift.

This lift model can be much more complex in order to cope with many actors at the same time, entering from any floor, etc. The lift model has been extended in order to correctly cope with up to three actors entering from each floor at a same time.

The simulator program, which manages the animated VE, should correctly treat the case of parallel interpretation of interaction plans. In the VE simulator ACE [28,31], whenever an actor selects an interaction plan, a new process is opened in order to interpret it. This process can be seen as the actor skill to interact with objects, and is part of the smart object reasoning module.

However, it is difficult to evaluate if the programmed behaviors can correctly solve all possible combination of cases. Moreover, it is possible to find examples where avoiding a deadlock situation would be difficult. For example, a deadlock can easily occur when trying to solve the classical problem of simultaneous access of resources that is called the *dining philosophers* [32].

When behaviors get complex, the amount of internal private behaviors will increase, and the coherence of the final object may be difficult to guarantee. Complex behaviors can be theoretically programmed with the available plan instructions, but it may not be a simple task in many cases. Although it should be possible, it may not be straightforward to handle a complex situation like, for instance, programming the necessary behaviors for people taking the lift while carrying big objects, also paying attention to let women and kids to have priority over entering the lift. However, in any case, complex behaviors should be defined as plans. Graphical state machines are not hierarchical and were designed to be used as the last layer, just to link previously defined plans.

Difficulties like multiple access of object resources or limitations when modeling complex behaviors, are issues that should be solved with external and dedicated programming languages. The smart object plans were designed to easily solve general and simple cases, and whenever complex solutions are needed, more powerful programming languages should be used. Currently, SOMOD has the extra ability to link behavioral plans with external Python scripts [25].

It is difficult to measure how far the modeling techniques used in SOMOD are effective. Users which experienced the system found the control interface very simple to assimilate and easy to use for programming simple interactions. However, when starting to build more complex objects, it becomes difficult to coherently program and interconnect all needed behavioral plans. This difficulty is inherent to computer programming in general, and one solution to minimize this point is to build complex objects from simpler ones. This kind of construction is already possible to do with the current template loader, however solutions are still to be found regarding the automatic connection of loaded modules. Such issue is important in order to minimize the needed low-level programming capabilities of the user.

6. Final Remarks

SOMOD has been used to model smart objects for different purposes. Some times objects have simple geometry but a lot of semantic information, and some times objects simply don't offer interaction, and SOMOD is used, for instance, to define relative positions around the object for collision avoidance. Some examples of interactive applications built using smart objects are: urban environment simulations [33], object prototyping with augmented reality [34], and behavioral animation of virtual actors [28]. Figure 13 shows a typical working session and figure 14 shows some snapshots of different object interactions during the simulation of an interactive computer room.

Some extensions were done in SOMOD targeting the simulation system ACE [28,31]. For example, we have integrated behavioral instructions to user-defined functions in Python [25]. This enables writing any kind of complex behaviors, as Python is a complete interpreted and object-oriented language. Some smart objects can also be converted to VRML. A simple

translator was developed, however with several limitations. Only simple behaviors can be correctly translated, and interaction is limited to mouse clicks.

From the simulator point of view, the most important aspect of the smart object description is the fact that any user of the object can ask for a list of available interactions, which is generated in run time, depending on the current state of the object. This list of possible interactions forms the communication language between actors and objects, as a behavioral interface to coherently manage any number of users interacting with objects.

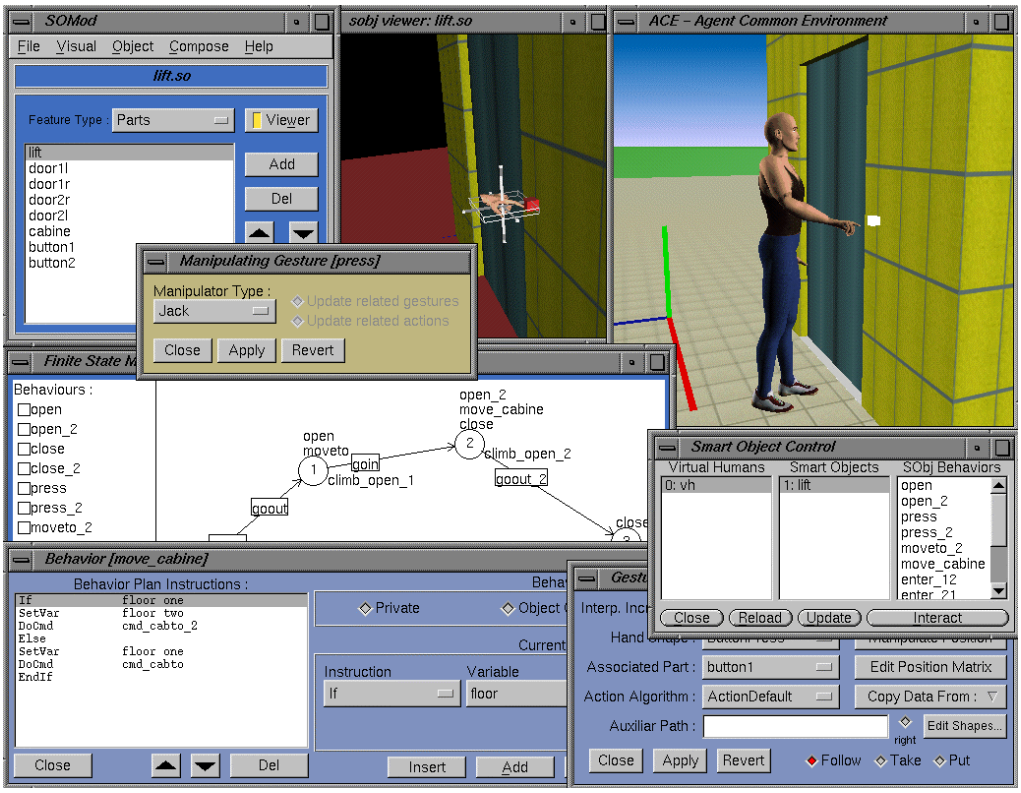

Figure 13 – A snapshot of a typical behavioral modeling session with SOMOD and ACE. The simulation window is connected to the modeling window, closing the loop of a productive simulation-based design.

Figure 14 - Four images showing different simple smart object interactions. Such objects are part of an interactive virtual lab simulated in ACE [33], which contains around 90 interactive smart objects. In this example, available behaviors of each object are identified by names with semantic meaning, permitting an automatic behavior selection mechanism from natural language instructions.

7. Acknowledgments

This research was supported by the Swiss National Foundation for Scientific Research and by the Brazilian National Council for Scientific and Technologic Development (CNPq).

8. References

1. R. J. Millar, J. R. P. Hanna, and S. M. Kealy, “A Review of Behavioural Animation”, Computer & Graphics, 23, 127-143, 1999.

2. T. Ziemke, “Adaptive Behavior in Autonomous Agents”, Presence, vol. 7, no. 6, 564-587, december 1998.

3. R. A. Brooks, “A Robust Layered Control System for a Mobile Robot”, IEEE Journal of Robotics and Automation, 2(1), 14-23, 1986.

4. S. Franklin, and A. Graesser, “Is it an Agent, or Just a Program?: a Taxonomy for Autonomous Agents”, Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springler Verlag, Berlin/Heidelberg, Germany, 1996.

5. N. N. Badler, “Virtual Humans for Animation, Ergonomics, and Simulation”, IEEE Workshop on Non-Rigid and Articulated Motion, Puerto Rico, June 97.

6. W. Johnson, and J. Rickel, “Steve: An Animated Pedagogical Agent for Procedural Training in Virtual Environments”, SIGART Bulletin, ACM Press, 8(1-4), 16-21, 1997.

7. J. J. Shah, and M. Mäntylä, “Parametric and Feature-Based CAD/CAM”, John Wiley & Sons inc. ISBN 0-471-00214-3, 1995.

8. G. Booch, “Object Oriented Design with Applications”, The Benjamin Cummings Publishing Company, Inc., ISBN 0-8053-0091-0, 1991.

9. C. Hand, “A Survey of 3D Interaction Techniques”, Computer Graphics Forum, 16(5), 269-281, 1997.

10. D. Bowman, and L. Hodges, “Formalizing the Design, Evaluation, and Application of Interaction Techniques”, Journal of Visual Languages and Computing, 10, 37-53, 1999.

11. I. Poupyrev and T. Ichikawa, “Manipulating Objects in Virtual Worlds: Categorization and Empirical Evaluation of Interaction Techniques”, Journal of Visual Languages and Computing, 10, 19-35, 1999.

12. Y. Okada, K. Shinpo, Y. Tanaka and D. Thalmann, “Virtual Input Devices based on Motion Capture and Collision Detection”, Proceedings of Computer Animation 99, Geneva, May, 1999.

13. L. Levinson, “Connecting Planning and Acting: Towards an Architecture for Object-Specific Reasoning”, PhD thesis, University of Pennsylvania, 1996.

14. Z. Huang, R. Boulic, N. Magnenat-Thalmann, and D. Thalmann, “A Multi-Sensor Approach for Grasping and 3D Interaction”, Proceedings of Computer Graphics International, Leeds, UK, June, 1995.

15. M. Cutkosky, “On Grasp Choice, Grasp Models, and the Design of Hands for Manufacturing Tasks”, IEEE Transactions on Robotics and Automation, 5(3), 269-279, 1989.

16. Y. Koga, K. Kondo, J. Kuffner, and J. Latombe, “Planning Motions with Intentions”, Proceedings of SIGGRAPH’94, 395-408, 1994.

17. J. Berta, “Integrating VR and CAD”, IEEE Computer Graphics and Applications, 14-19, September / October, 1999.

18. T. Tyrrel, “Defining the Action Selection Problem”, Proceedings of the Fourteen Annual Conference on Cognitive Science Society”, Lawrence Erlbaum Associates, 1993.

19. G. Battista, P. Eades, R. Tamassia, and I. Tollis, “Graph Drawing – Algorithms for the Visualization of Graphs”, Prenticce Hall, ISBN 0-13-301615-3, 432pp, 1999.

20. A. Schoeler, R. Angros, J. Rickel, and W. Johnson, “Teaching Animated Agents in Virtual Worlds”, Smart Graphics, Stanford, CA, USA, 20-22, March, 2000.

21. Motivate product information, Motion Factory web address: www.motion-factory.com.

22. J. Granieri, W. Becket, B. Reich, J. Crabtree, and N. Badler, “Behavioral Control for Real-Time Simulated Human Agents”, Symposium on Interactive 3D Graphics, 173-180, 1995.

23. F. Lamarche, and S. Donikian, “The Orchestration of Behaviors using Resources and Priority Levels”, EGCAS’2001 - 12th Eurographics Workshop on Animation and Simulation, Manchester, UK, 2001.

24. Perlin K., and A. Goldberg, “Improv: A System for Scripting Interactive Actors in Virtual Worlds”, In Proceedings of SIGGRAPH’96, 205-216, 1996.

25. M. Lutz, “Programming Python”, Sebastapol: O’Reilly, 1996. (see also: www.python.org)

26. S. Parry-Barwick, and A. Bowyer, “Is the Features Interface Ready?”, In “Directions in Geometric Computing”, Ed. Martin R., Information Geometers Ltd, UK, Cap. 4, 129-160, 1993.

27. M. Kallmann, D. Thalmann, “Direct 3D Interaction with Smart Objects”, Proceedings of ACM VRST'99, London, December, 1999.

28. M. Kallmann, J. Monzani, A. Caicedo, and D. Thalmann, “ACE: A Platform for the Real Time Simulation of Virtual Human Agents”, EGCAS’2000 - 11th Eurographics Workshop on Animation and Simulation, Interlaken, Switzerland, 2000.

29. R. Boulic, P. Bécheiraz, L. Emering, and D. Thalmann, “Integration of Motion Control Techniques for Virtual Human and Avatar Real-Time Animation”, In Proceedings of VRST’97, ACM press, 111-118, September 1997.

30. P. Baerlocher, R. Boulic, “Task-Priority Formulations for the Kinematic Control of Highly Redundant Articulated Structures”, In Proceedings of IROS, Victoria, Canada, 323-329, 1998.

31. M. Kallmann, “Object Interaction in Real-Time Virtual Environments”, PhD thesis, Swiss Federal Institute of Technology – EPFL, Lausanne, 2001.

32. G. Andrews, “Concurrent Programming: Principles and Practice”, The Benjamin/Cummings Publishing Company, Inc., California, ISBN 0-8053-0086-4, 1991.

33. N. Farenc, S. R. Musse, E. Schweiss, M. Kallmann, O. Aune, R. Boulic, and D. Thalmann, “A Paradigm for Controlling Virtual Humans in Urban Environment Simulations”, Applied Artificial Intelligence Journal 14(1), ISSN 0883-9514, January, 69-91, 2000.

34. S. Balcisoy, M. Kallmann, P. Fua, and D. Thalmann, "A Framework for Rapid Evaluation of Prototypes with Augmented Reality", Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST), 2000.