In *Handbook of Virtual Humans*, N. Magnenat-Thalmann and D. Thalmann (Eds.), John Wiley & Sons, UK, 2004, 303-322

Interaction with 3-D Objects

Marcelo Kallmann

Abstract. Among the several issues related to real-time animation of virtual human actors, the ability to interact with virtual objects requires special attention. Take as example usual objects as: automatic doors, general furniture, or a lift. Interaction with such objects can easily become too complex for real-time applications. Some of the related problems involve: recognition of manipulation places, automatic arm and hand animation, and motion synchronization between actors and objects. The *smart object* approach is described here and can overcome many of these difficulties by storing all needed interaction information within the object description. Interaction information is defined during modeling phase, forming a complete "user guide" of the object. In this way, virtual actors can simply access and follow such interaction descriptions in order to accomplish some given task. Solutions to related sub-problems such as programming object's behaviors, interactions with multiple actors, or actor animation to manipulate objects are discussed here, and detailed case studies are analyzed.

9.1 Introduction

Computer graphics systems are no longer synonym of a static scene showing 3D objects. In most nowadays applications, objects are animated, they have deformable shapes and realistic movements. Such objects "exist" in virtual environments and are being used to simulate a number of different situations. For instance, costs are saved whenever it is possible to simulate and predict the result of a product before manufacture.

Although many technical issues are not fully solved, a lot of attention has been given to a next step: *lifelike behaviors*. The issue is to have virtual entities existing in virtual environments, deciding their actions by their own, manipulating virtual objects and etc. As a natural consequence, computer animation techniques today are strongly related to artificial intelligence and robotics techniques.

It is still a challenge to animate a virtual actor that can decide its motions, reacting and interacting with its virtual environment, in order to achieve a task given by the animator. This virtual actor might have its own way to decide how to achieve the given task, and so, many different sub-problems from many areas arise.

One of these sub-problems is how to give enough information to the virtual actor so that it is able to interact with each object of the scene. That means, how to give to an actor the ability of interaction with general objects, in a real-time application. This includes different types of interactions that can be considered. Some examples are: the action of pushing a button, opening a book, pushing a desk drawer, turning a key to then open a door and so on.

A human-like behavior would recognize a given object with vision and touch, and then, based on past experiences and knowledge, the correct sequence of motions would be deduced and executed. Such approach is still too complex to be handled in a general case, and not suited for interactive systems where real-time execution is required.

To avoid complex and time-consuming algorithms that try to model the full virtual actor's "intelligence", an alternate approach is to use a well defined object description where all properties, functionality features and descriptions of the steps to perform each available interaction are added to the geometrical shape description of the object. In that way, part of the most difficult thing to model, the knowledge of the virtual actor, is avoided. Instead, the designer of the object will use his/her own knowledge assigning to the object all information that the virtual actor needs to access in order to interact with the object. This more direct approach has been called the *smart object* approach [1] to animate actor-object interactions.

9.2 Related Work

The necessity to model actor-object interactions appears in most applications of computer animation and simulation. Such applications encompass several domains, as for example: autonomous agents in virtual environments, human factors analysis, training, education, virtual prototyping, and simulation-based design. A good overview of such areas is presented by Badler [2], and one example of a training application is described by Johnson and Rickel [3].

The term *object interaction* has been employed in the literature mainly for the direct interaction between the user and the environment [4], but less attention has been given to the actor-object interaction case.

Actor-object interaction techniques were first specifically addressed in a simulator based on natural language instructions using an *object specific reasoning* (OSR) module [5]. The OSR keeps a relational table informing geometric and functional classification of objects, in order to help the interpretation of natural language instructions. The OSR module also keeps some interaction information: for each object graspable site, the appropriate hand shape and grasp approach direction. This set of information is sufficient to decide and perform grasping tasks, but no considerations are done concerning interactions with more complex objects with some proper functionality.

In most of the cases, actors and objects have proper behaviors, and behavioral animation techniques [6] [7] can be employed. Behaviors can follow the *agent* approach, where actions are decided based on sensing the environment [8] [9]. These domains give techniques that can be employed to approach some issues of the general actor-object interaction problem. The term *object functionality* will be sometimes employed here instead of *object behavior*, reflecting the fact that the objects considered here have simpler behaviors than actors. The following two sections will present the related work done divided into two main categories: the definition of object's functionality, and the actor animation to perform interactions.

9.2.1 Object Functionality

In general, objects contain some proper functionality that needs to be defined. Some simple examples are: after pressing a button the lift door will open, or only after turning on the printer that it can print. Such rules need somehow to be programmed inside objects, and different techniques may be employed.

Okada [10] proposes to compose object parts equipped with input and output connectors that can be linked to achieve different functionalities. State machines are also widely used. In particular, most game engines [11] [12] use hierarchical finite state machines, defined graphically through user interfaces.

An important point to be taken into account is the ability to interpret the defined functionality in parallel, in a synchronized way. As it will be shown later, it may happen that several actors interact with a same object, e.g. to enter a lift.

Several techniques used for general behavior definition cover some of these aspects. One specific structure to define parallel behaviors is the *parallel transitions network* (PaTNets) [13] [14]. Other works [15] cover the aspect of sharing resources with concurrent state machines.

It is natural to think that the description of the object functionality should be associated with the object geometric description as well. Current standards for object description [16] are normally based on scene graphs containing some nodes to connect animation to external events, such as events from the user interface (mouse, keyboard, etc). This provides a primitive way to describe basic object functionality, but as it is not generic enough, it is always needed to make use of complete programming languages such as Java scripts. Thus, there is still place for standards on the description of functionalities.

A similar scenario appears in the *feature modeling* area, mainly on the scope of CAD/CAM applications [17] where the main concern is to represent not only the shape of the object, but also all other important features regarding its design choices and even manufacture procedures. In fact, suppliers of CAD systems are starting to integrate some simulation parameters in their models [18]. The *knowledgeware* extension of the Catia system [19] can describe characteristics like costs, temperature, pressure, inertia, volume, wetted area, surface finish, formulas, link to other parameters, etc; but still no specific considerations are done to define object functionality or interactivity.

The main point is that none of these techniques cover specifically the problem of describing object functionality for actor-object interaction purposes. As it will be presented later, object functionality, expected actor behaviors, parallel interpretation of behaviors, and resources sharing need to be solved in an unified way.

9.2.2 Actor Animation

Once the object functionality is somehow defined and available, actors need to access this information and decide which motions to apply in order to complete a desired interaction.

The actor motion control problem is very complex, and has been mostly studied from the computer graphics community, mainly targeting movie and game industries. Boulic [20] proposes a good overview of the employed techniques. Recently the motion control problem has also received contributions from the robotics and artificial intelligence domains.

For instance, from the robotics area, a classification of hand configurations for grasping is proposed by Cutkosky [21]. Following the same idea, but targeting animation of virtual actors, Rijpkema [22] introduces a knowledge-based grasping system based on pre-defined hand configurations with on-line adaptation and animation. Huang [23] also proposes a system for automatic decision of hand configurations for grasping, based on a database of predefined grasping postures.

Also from the robotics domain, planning algorithms are able to define collision-free paths for articulated structures. Koga [24] has applied such algorithms for the animation of the arms of a virtual actor obtaining interesting results. The main drawback of the method is conversely also its main advantage: because of its random nature, complicated motions can be planned, but with high and unpredictable computational cost, thus not currently applicable for interactive simulations. A huge literature about motion planning is available, mainly targeting the motion control of different types of robots [25] [26], and companies start to appear to provide this technology [27].

Artificial intelligence techniques such as neural networks and genetic algorithms have been applied for the control of virtual bodies [28] [29] [30] [31]. Currently, these algorithms are too costly and can be only applied to limited cases. However, artificial intelligence techniques will probably become more powerful and usable in a near future.

Inverse kinematics [32] is still the most popular technique for articulated structure animation. Some works can handle the animation of complex structures, taking into account several constraints [33]. Some works present specific implementations regarding only the movement of the actor's arm [34] [35]. Although interesting results can be obtained it is still difficult to obtain realistic postures, specially concerning the automatic full body animation for reaching objects with hands. For instance, to determine a coherent knee flexion when the actor needs to reach with its hand a very low position. For a complete overview of the possibilities of Inverse Kinematics techniques, as well as other related issues, please refer to the Motion Control chapter of this book.

In another direction, database driven methods can easily cope with full body postures. The idea is to define pre-recorded (thus realistic) motions for reaching each position in the space inside a discrete and fixed volumetric grid around the actor. Then, when a specific position is to be reached the respective motion is obtained through interpolation of the pre-recorded motions relative to the neighboring cells. This is exactly the approach taken by [36] with good results achieved, but limited to the completeness of the database. Complementary, some works [37] propose techniques to adapt pre-recorded motions for respecting some given constraints. Database methods were also successfully used to determine grasping postures [38]. The main drawback of such methods is that they are not general enough: it is difficult to adapt motions to all given cases and also to handle collisions with the environment. However, some works start to propose solutions to handle collisions [39].

As a final conclusion, table 9.1 makes a comparison of these many methods, from the point of view of animating actors for general interactions with objects.

	Realism	Real-Time	Generality	Collisions
Motion Database	+	+	-	-
Path Planning	-	-	+	+
Inverse kinematics	-	+	+	-

Table 9.1. Comparison of the many motion control methods, regarding: the realism of the generated movements, the real-time ability of computation, generality for being applied to different types of interactions, and the ability to handle and solve collisions with the environment and self collisions

9.3 Smart Objects

In order to simplify the simulation of actor-object interactions, a complete representation of the functionality and interaction capabilities of a given object is proposed to be used. The idea is that each interactive object contains a complete description of its available interactions, like forming a "user guide" to be followed by actors during interactions. Once objects contain such information, they are considered here to become "smart".

9.3.1 Interaction Features

A feature modeling approach is used, and a new class of features for simulation purposes is proposed: *interaction features*. Interaction features can be seen as 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 to be 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 a straightforward approach because then, to perform an interaction, actors 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 9.2.

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 action.
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	Object and 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 9.2. The eight types of interaction features used in the smart object description

9.3.2 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*, able to correctly interpret the behavioral plans.

There is a trade-off when choosing which features to be considered in an application. As shown in figure 9.1, 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.

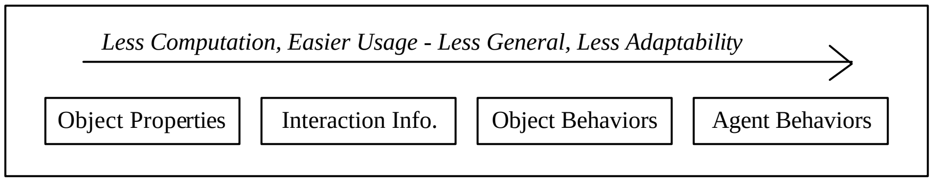


Fig. 9.1. The choice of which interaction features to take into account is directly related to many implementation issues in the simulation system

Note that the notion of a realistic result is context dependent. 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.

A design choice appears while modeling objects with too many potential interactions. Especially in the case of composed objects, 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 dashboard. 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.

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.

9.4 SOMOD

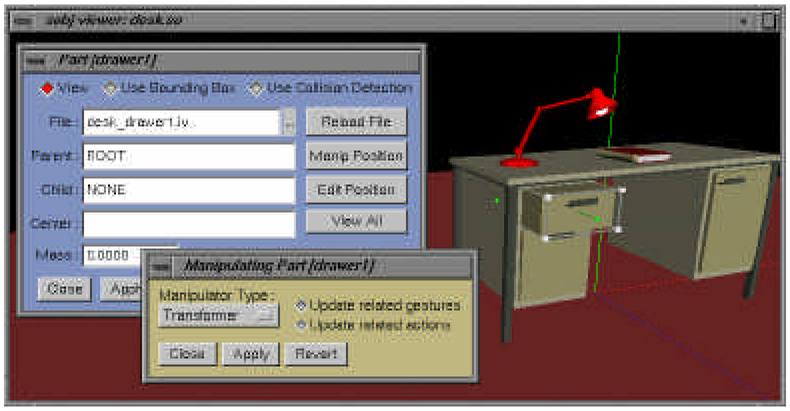
The Smart Object Modeler application (SOMOD) [40] was developed specifically to model smart objects. It was developed using Open Inventor [41] as graphics library, and FLTK [42] for the user interface.

SOMOD permits to import geometric models of the component parts of an object, and then to interactively specify all needed interaction features. Features are organized by type, and a main window permits to manage lists of features. According to the feature type, specific dialog boxes permit to edit the related parameters.

9.4.1 Object Properties

Text input windows are used to enter any text descriptions, with specific fields to describe a semantic name for the object, and its overall characteristics. These definitions can then be retrieved by simulators for any kind of processing.

An object is composed by assembling its different parts. The geometry of each part is imported from commercial modeling applications. Parts can then be positioned interactively using Open Inventor manipulators (figure 9.2). The same technique of using manipulators is adopted to define the movement actions that can be applied to each part. For example to define a translation, the user displaces a part using a manipulator, and the transformation movement from the initial position to the user-selected position is then saved as an action. Note that actions are saved independently of parts, so that they can be later parameterized differently (defining commands) and applied to different parts.



Part [drawer1]: [✓ View] Use Bounding Box Use Collision Detection
File: desk_drawer1.iv
Parent: ROOT
Child: NONE
Center:
Mass: 0.0000

Manipulating Part [drawer1]
Manipulator Type: Transformer
Update related gestures
Update related actions

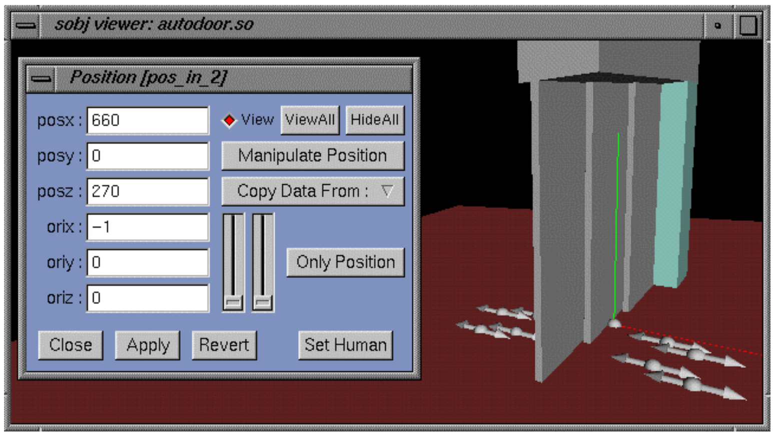
Fig. 9.2. Defining the specific parameters of a drawer. The drawer is a part of the smart object desk, which contains many other parts. The image shows in particular the positioning of the drawer in relation to the whole object

9.4.2 Interaction Information

Commands are specified simply by associating actions to parts, and giving a parameterization of how the motion should be applied to that part. Commands fully specify how to apply an action to a part and are directly referenced from the behavioral plans whenever a part of the object is required to move.

Positions and directions (figure 9.3) can be specified for different purposes. Each position (as each feature) is identified with a given name for later referencing from the interaction plans.

Note that all features related to graphical parameters can be defined interactively, what is important in order to see their location in relation to the object. Positions are defined in relation to the object skeleton's root, so that they can be transfomed to the same reference frame of the actor whenever is needed, during the simulation. Note that smart objects can be loaded and positioned anywhere in the virtual environment and all associated information need to be transformed accordingly.



posx : 660
posy : 0
posz : 270
orix : -1
oriy : 0
oriz : 0

Fig. 9.3. Positions can be defined for different purposes. In the figure, many different positions (and orientations) are placed to propose possible places for actors to walk when arriving from any of the door sides

Gestures are normally the most important interaction information. Gestures parameters are defined in SOMOD and proposed to actors during interactions. The term gesture is used to refer to any kind of motion that an actor is able to perform. A gesture can only reference a pre-recorded animation to be played, but the most used gesture is to move the hand towards reaching a location in space using inverse kinematics. Figure 9.4 illustrates the process of defining hand configurations and locations to be reached. Such information can then be used during simulations to animate the actor's arm to perform manipulations with the object. It is left to the simulator to decide which motion algorithm should be applied. As it will be shown in section 9.5, inverse kinematics was used for the examples shown here.

Some extra parameters can be set in order to define if after the reaching motion is completed, the associated part should be taken, put, or just followed. Following is used to simulate interactions such as pressing buttons or opening drawers.

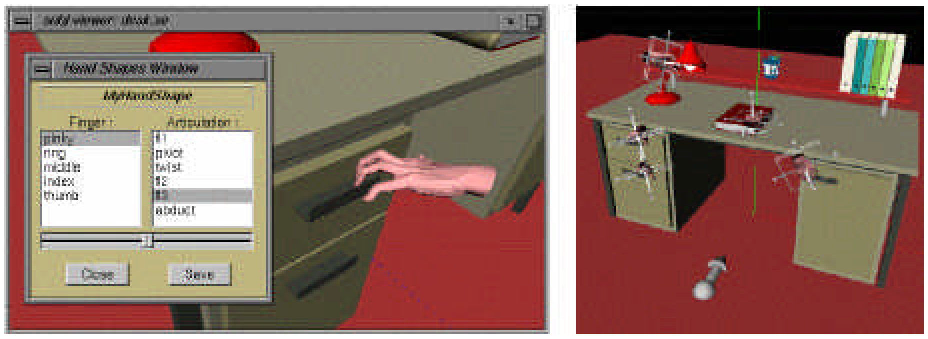


Fig. 9.4. The left image shows a hand shape being interactively defined. The right image shows all used hand shapes being interactively located with manipulators

9.4.3 Behaviors

The behavioral animation solutions presented here are specific for the actor-object interaction problem. For a more general and complete overview on behavioral animation techniques, please refer to the chapter Behavioral Animation of this book.

In smart objects, 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, and 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 that non-programmers are able to create and test it.

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 drawers, 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. Variables are defined by assigning a name and an initial value, and can be also used for other purposes from the interaction plans, as for instance to retrieve the current position of the actor in order to decide from which side of a door the actor should enter.

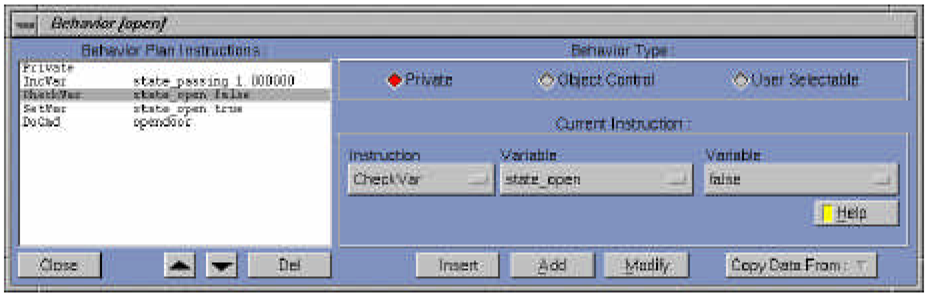
Interaction plans are defined using a specific dialog box (figure 9.5), which guides the user through all possible primitive instructions to use. The following key concepts are used for the definition of interaction plans:

• An interaction plan describes both the behavior of the object and the expected behavior of actors. Instructions that start with the keyword "user" are instructions that are proposed to the user of the object, i.e., the actor. Examples of some user instructions are: `UserGoto` to propose it to walk to some place, `UserDoGest` to tell it to manipulate some part using pre-defined gesture parameters (figure 9.5), `UserAttachTo` to say that the actor should be attached to some object part, as when entering a lift, etc.

• In SOMOD, an interaction plan is also called 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 a virtual environment. They enable objects to act like agents, for example sensing the environment to trigger some motion, or to have a continuous motion as for a ventilator. Object control behaviors cannot have user-related instructions. Finally, user selectable behaviors are those that are visible to be selected by actors, in order to perform a desired interaction.

• 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. Behavior availability is controlled with an instruction `CheckVar`. Its use is exemplified in figure 9.5, and will be explaining in section 9.6 (case studies).



```
Private
IncVar        state_passing 1.000000
CheckVar      state_open false
SetVar        state_open true
DoCmd         opendoor
```

Behavior Type: [✓ Private] Object Control User Selectable

Instruction: CheckVar
Variable: state_open
Variable: false

Fig. 9.5 Defining interaction plans: menu-buttons are used to list all possible instructions to use, and for each instruction, the possible parameters are listed in additional menu-buttons. In this way complex behaviors can be easily created.

Multi-Actor Interactions. If the object is required to interact with many actors at the same time, synchronization issues need to be taken into account. There are two levels of synchronization. The first one is to guarantee the interaction coherence. For this, interaction plans are responsible to correctly make use of variables to, for example, count the number of actors currently interacting with the object, correctly setting states and testing conditions to ensure correctness when multiple actors are accessing the same object, etc. Section 9.6.2 details a smart object door that is able to deal with several actors at the same time.

Another level of synchronization is to correctly manage the interpretation of several interaction plans in parallel. This is independent of the object type and should be done by the simulator. A proposed procedure for this kind of synchronization is described in section 9.5.1.

Graphical State Machines. SOMOD plans are very simple to use 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.

To simplify modeling the behaviors of more complex objects, SOMOD proposes a dialog box to graphically design finite state machines. The proposed solution is to start designing basic interaction plans, using the standard behavior editor (figure 9.5). Then, when all components have their functionality defined, the state machine window is used, permitting to define the states of the whole object, and the connections between the components. At the end, designed state machines are automatically translated into interaction plans so that, from the simulator point of view, a single representation is kept, and all behaviors are treated as plans. Section 9.6.3 exemplifies how a state machine is used to simplify the creation of a smart lift.

Templates. One important concept is to reuse previously defined objects and functionalities, and a specific template loader was designed for this goal. The template loader window can import any kind of features from other smart objects. Note that some features have dependencies on other features. These dependencies need to be tracked and coherently loaded. In addition, names are automatically updated whenever conflicts with previously created names appear.

Extensions. SOMOD has been used for different purposes and different types of extensions have been integrated. Positions can be automatically generated to delimitate regions for collision avoidance when actors are walking. Behavioral instructions can simply call a user-defined function in Python language [43], virtually enabling any complex behavior to be coded. A VRML [16] exporter was also developed, however with limited animation capabilities.

9.5 Interacting with Smart Objects

When a smart object is loaded in a virtual environment, actors are able to access available interaction plans and interpret them. Two main issues should be considered when interpreting plans: how to synchronize the interpretation of plans in parallel, and how to animate the actor's skeleton whenever a manipulation action is required.

9.5.1 Interpretation of Plans

Each time an actor selects an interaction plan to perform, a specific thread is created to follow the instructions of the plan (figure 9.6). The situation is that a simultaneous access to the smart object is done when interpreting instructions, for example, that access variables or trigger motions.

A simple synchronization rule to activate and block the many processes interpreting plans in parallel is adopted. However, interaction plans still need to be well designed in order to cope with all possible combinations of simultaneous access. For example, complex situations as the *dining philosophers* problem [44] are not automatic solved.

Long and local instructions. A simple built-in synchronization rule between threads is used. For this, plans instructions are grouped into two categories: *long* instructions, and *local* instructions. Long instructions are those that cannot start and complete in a single time step of the simulation. For example, instructions that trigger movements will take several frames to be completed, depending on how many frames the movement needs to finish. All other instructions are said to be local.

Plans are interpreted instruction by instruction, and each instruction needs to be finished before the next one is executed. When a plan is being interpreted by some thread *T*, all other threads are suspended until a long instruction is found. In this way, *T* will fully execute sequences of local instructions, while all other threads remain locked. When a long instruction is reached, it is initialized, the other threads are activated, and *T* stays observing if the instruction has finished. This scheme results with the situation where all activated threads are in fact monitoring movements and other long instructions, and each time local instructions appear, they are all executed in a single time step, while other threads are locked.

This approach automatically solves most common situations. For example, suppose that a smart lift has an instruction like: "if state of calling button is *pressed* do nothing; otherwise set state of calling button to *pressed* and press it". Suppose now that two actors, exactly at the same time, decide to call the lift. The synchronization rule says that while one thread is interpreting local instructions, all others are locked. In this way, it is guaranteed that only one actor will actually press the button. Without this synchronization, both actors would press the button together at the same time, resulting serious inconsistent results.

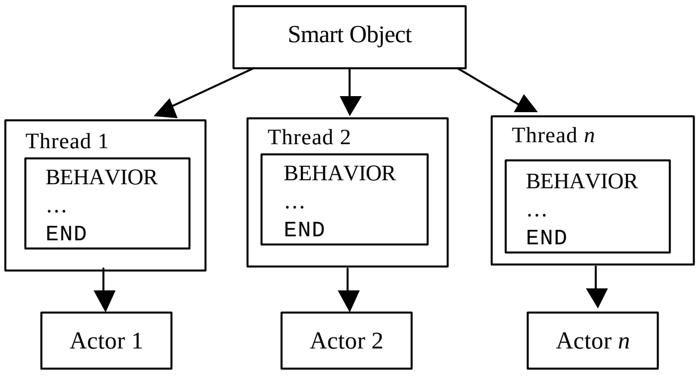


Fig. 9.6. For each actor performing an interaction with an object, a thread is used to interpret the selected interaction plan. Each thread accesses and controls, in a synchronized way, its related actor and object, according to the plan's instructions

9.5.2 Manipulation Actions

To animate the actor's skeleton for manipulation actions, Inverse kinematics is used, with different constraints based on the type of manipulation and on the goal location to reach with the hand.

An unified approach was designed targeting general manipulations with one arm. First of all, a manipulation movement is divided in three phases: reaching, middle, and final phases (figure 9.7).

All manipulation movements start at the reaching phase. In this phase, inverse kinematics is used in order to animate the actor's skeleton to have its hand in the specified position. Then three cases can happen, depending on the parameters retrieved from the smart object: *follow*, *take*, or *put*. Parameters *follow* and *take* are used to specify the attachment of objects to the actor's hand. The *follow* parameter indicates that the actor's hand should then follow a specified movement. This is the case for example to press buttons and open drawers: the specified translation movement to animate the object part is followed by the actor's hand, while inverse kinematics is used in order to adjust the posture of the actor's skeleton. The final phase has the default behavior to either keep the actors posture unchanged or to change it to a standard rest posture. This can be configurable and permits to chain other actions instructions or to left the actor in the rest posture. In this way long and complex manipulations can be subdivided in small pieces, each piece matching the schema of figure 9.7.

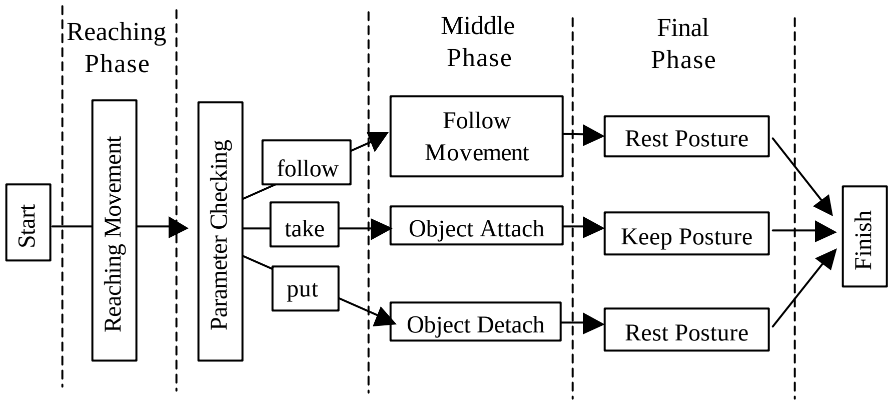


Fig. 9.7. Considered phases for a manipulation instruction

Additionally to the inverse kinematics motion to control the actor's arm towards the specified location, the current hand fingers configuration is interpolated towards the specified final configuration, and the actor's head is set to look to the place of manipulation.

Constraints Distribution. The used inverse kinematics module was developed by Paolo Baerlocher [33] (see the Motion Control chapter of this book), and is able to control the full actor body with different types of constraints. The skeleton's root is a node between the pelvis and the spine, and which separates the hierarchies of the legs and feet from the hierarchies of the spine, head and arms.

At the beginning of a manipulation, the actor's skeleton sizes and the task position to reach with the hand are analyzed and different constraints are set.

• First, the inverse kinematics module is set to only animate the joints of the arm, shoulder, clavicle, and the upper part of the spine. This set of joints makes the reach volume space larger, as the actor can reach further positions by flexing the spine. However, a side effect is that even for closer positions to reach, the spine can move, generating weird results. To overcome this, two new constraints are created. Positional and orientation constraints, with low priority, are used in order to keep the spine straight as long as it is possible (figure 9.8).

• Secondly, if the goal position to reach with the hand is lower than the lowest position achievable with the hand in a straight rest position, a special knee flexion configuration is set. The joints of the hip, knee, and ankle are added to the allowed joints to be animated by the inverse kinematics module, and two new constraints, with high priorities, are added to keep each foot on its original position and orientation. This configuration makes the actor to flex the knees, keeping its feet attached to the ground, while the actor's skeleton root is gradually lowered (figure 9.9).

After the initial phase of constraints and joints control distribution is done, the hand task is set with highest priority. In order to animate the arm from the initial configuration to the desired one, the initial hand position and orientation are extracted and interpolated towards the goal hand position and orientation, generating several interpolated sub-goals. Interpolations can be simply linear, but should consider biomechanical properties. Inverse kinematics is then used to generate the final actor posture for each interpolated sub-goal.

After the reaching phase is completed, a *follow* movement might be required (figure 9.7). In this case, the hand keeps its orientation, and only its position is updated to follow the movement of the object being manipulated. Figures 9.10 and 9.11 exemplify both reaching and middle phases.

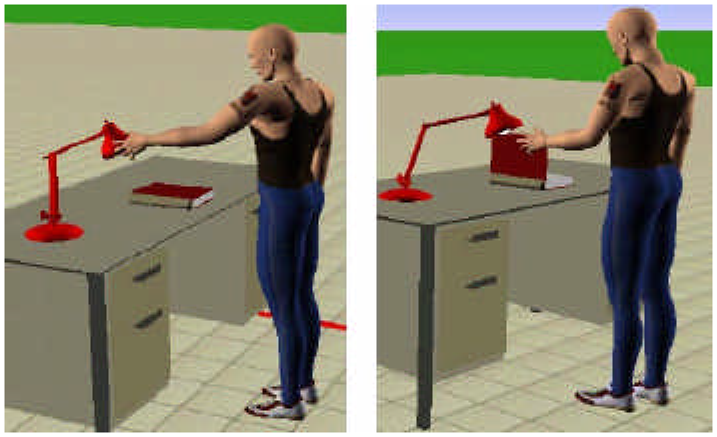


Fig. 9.8. Specific constraints are used to keep the actor’s spine as straight as possible

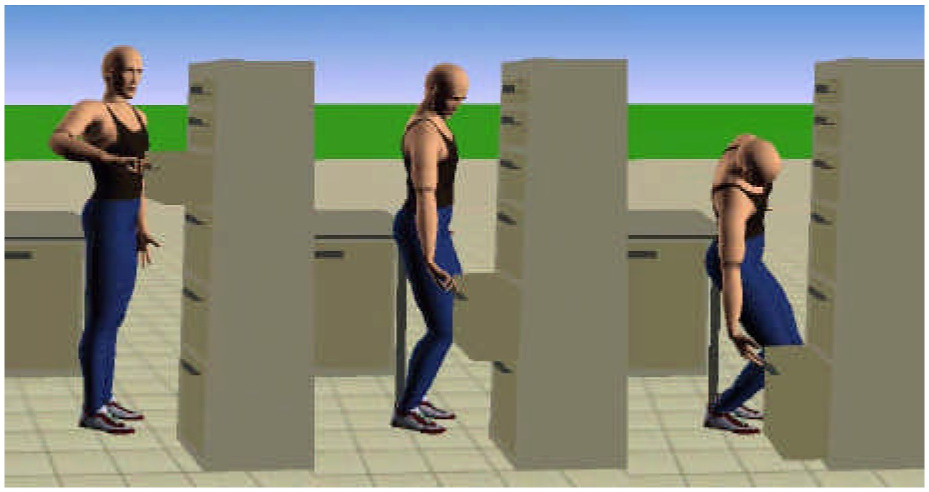


Fig. 9.9. When the position to reach with the hand is too low, additional constraints are used in order to obtain knee flexion. The images show, from left to right, the postures achieved when reaching each time lower positions

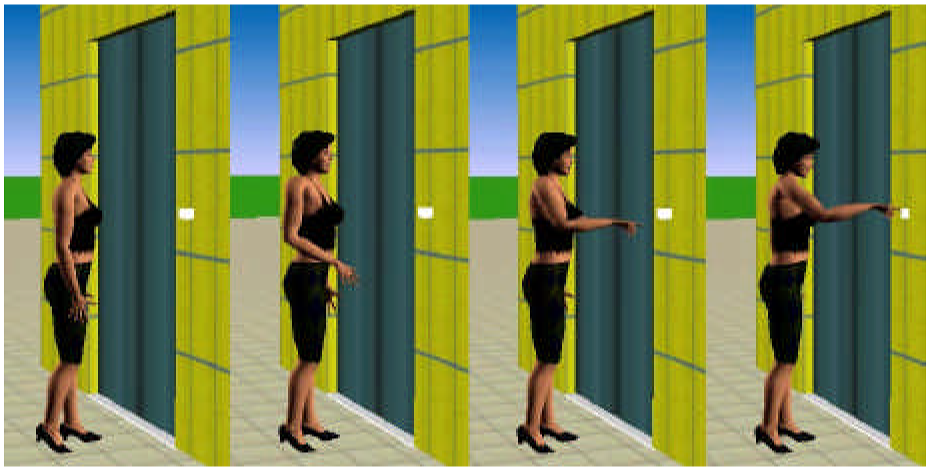


Fig. 9.10. The reaching phase of a button press manipulation. Note that the actor's head is also controlled to look to the button, and the hand configuration is gradually interpolated towards the final button press shape

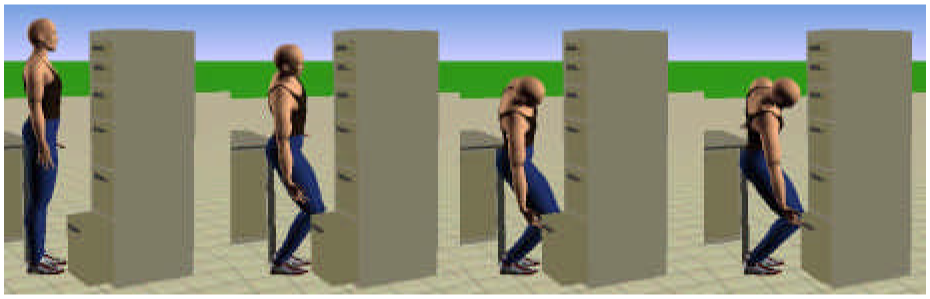


Fig. 9.11. The reaching and follow movements used to close a drawer. Note that during the "follow phase" (closing the drawer) the knee flexion is maintained

9.6 Case Studies

This section details how interaction plans can be actually coded. The focus is on explaining how different issues can be solved using simple scripted plans, and not to give a complete description of SOMOD instructions. The first case shows how simple interactions such as opening a drawer can be defined. The second example shows how concurrent object access can be handled when multiple actors want to interact with a same automatic door. And a final example shows how the graphical state machine editor can be used to simplify the definition of more complex interactions.

The examples shown here were generated with the Agents Common Environment (ACE) system [45], which integrates smart objects with higher-level behavioral control modules. For more information about ACE, and for higher-level animation examples obtained with ACE, please refer to the chapter Behavioral Animation of this book.

9.6.1 Opening a Drawer

Interaction plans are grouped as a set of behaviors, each one containing a list of instructions, similarly to procedures of a program. The following code exemplifies interactions of opening and closing a drawer, as showed in figures 9.9 and 9.11:

```
BEHAVIOR open_drawer
  CheckVar    var_open false
  UserGoTo    pos_front
  UserDoGest  gest_drawer
  DoCmd       cmd_open
  SetVar      var_open true
END

BEHAVIOR close_drawer
  CheckVar    var_open true
  UserGoTo    pos_front
  UserDoGest  gest_drawer
  DoCmd       cmd_close
  SetVar      var_open false
END
```

The instruction `CheckVar` determines if the behavior is available or not. For example, the behavior `open_drawer` is only available if the state variable `var_open` is false. In this way, at any time, actors can ask for a list of available interaction plans, i.e, available object behaviors according to the objects state.

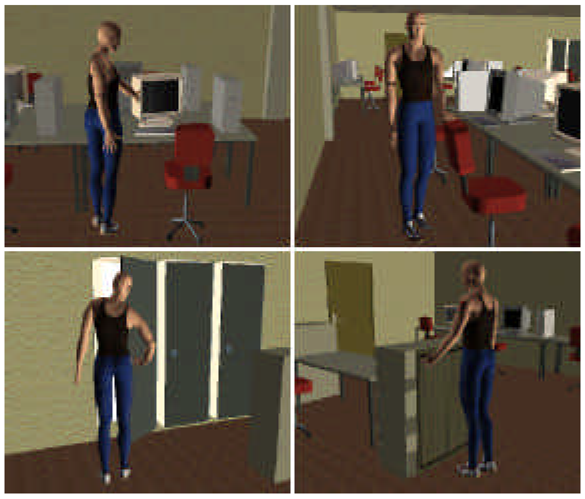


Fig. 9.12. Several interactions similar to the opening drawer case: all are based on reaching a position and following an object's movement

In the example, when an actor receives the plan `open_drawer`, it will execute each primitive instruction in the plan starting with the `User` keyword. That is, it will walk to the proposed position and then it will apply inverse kinematics to animate the actor to perform the specified motion in the gesture `gest_drawer`. In this case, the motion includes a *follow* movement that is set to follow the movement of the drawer part. The drawer itself will be animated by the smart object, according to the specified `cmd_close` parameters. At the end, the object state is changed, allowing the `close_drawer` behavior to be available. Note that both instructions, which are interpreted by the object and by the actor, are mixed in the same behavior.

This same kind of solution can be applied to all interactions consisted of reaching a place and then following an object's movement. This is the case of interactions such as: pushing a button, opening the cover of a book, opening doors, etc (figure 9.12).

9.6.2 Multiple Actors Interaction

The following code exemplifies how state variables are used to synchronize multiple actors passing a same automatic door:

```
BEHAVIOR open
  Private
  IncVar   var_passing 1
  CheckVar var_passing 1
  DoCmd    opendoor
END


BEHAVIOR close
  Private
  IncVar   var_passing -1
  CheckVar var_passing 0
  DoCmd    cmd_close
END
```

```
BEHAVIOR enter_l_r
  Private
  UserGoTo pos_l_in
  DoBh     open
  UserGoTo pos_r_out
  DoBh     close
END

BEHAVIOR enter_r_l
  Private
  UserGoTo pos_r_in
  DoBh     open
  UserGoTo pos_l_out
  DoBh     close
END
```

```
BEHAVIOR enter
  UserGetClosest pos pos_l_in, pos_r_in
  If             pos==pos_l_in
  DoBh           enter_l_r
  Else
  DoBh           enter_r_l
  EndIf
END
```

Each time an actor desires to pass the door, it will interpret the behavior `enter`, that will give it the correct positions to pass the door without colliding with other actors. The behavior `enter` is the only one visible to the actor, as all others are declared private. It starts by deciding from which side the actor will enter, using the `UserGetClosest` instruction that asks if the actor is closest to a position on the left side (`pos_l_in`), or on the right side (`pos_r_in`). Then, according to the result, the behavior to enter from the correct side is called.

Behaviors `enter_l_r` and `enter_r_l` simply: send the actor to an initial position, open the door, send the actor to the final position on the other side, and close the door. But some more things are needed in order to cope with multiple actors. When modeling this smart object, several positions were created for each of four cases: going in from the left and right side, and going out from the left and right sides. Positions of a same case are given a same name, so that when the instruction `UserGoTo` is called, the object is able to keep track, among all defined positions with the same name, those that currently have no actor using it. Figure 9.3 illustrates this automatic door and the used positions. The figure shows four positions for each case, which are associated to orientations and are represented as arrows. Note however that such strategy relies entirely on the correct definition of the positions, for a more general solution, actors should be also equipped with sensors to avoid colliding with other entities which are out of the smart object control.

Behaviors `open` and `close` are called to actually close or open the door. These behaviors maintain a state variable that counts how many actors are currently passing the door (`var_passing`) to correctly decide if the door needs really to be open or closed. Remember that actors interpret the same script concurrently, so that all actors will ask to close the door, but only the last one will actually close it.

9.6.3 Complex Behaviors

Consider the case of modeling a two-stage lift. Such a lift is composed of many parts: doors, calling buttons, a 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.

Similarly to the examples already explained, the modeling of the lift functionality is done in two phases. First, all basic behaviors are programmed, e.g., opening and closing doors, pressing the calling button, and moving the lift from one stage to another. If the lift is to be used by multiple actors, then all corresponding details need to be taken into account. Then, a higher-level interaction of just entering the lift from the current floor to go to the other floor is coded using calls to the basic behaviors.

To simplify modeling behaviors of such complex objects, a state machine can be used permitting to define the states of the object as a whole, and the connections between the component behaviors.

Figure 9.13 shows a state machine example for the lift case. This state machine has 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: pressing the calling button, opening the door, entering inside, closing the door, move the cabin up, opening the other door, and going out.

In the lift behavior example of figure 9.13, there is a single interaction of entering the lift, which can be too long, giving no options to the actor during the interaction. Figure 9.14 shows another solution, 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. As illustration, figure 9.15 shows some snapshots of the animation sequence of an actor entering the lift.

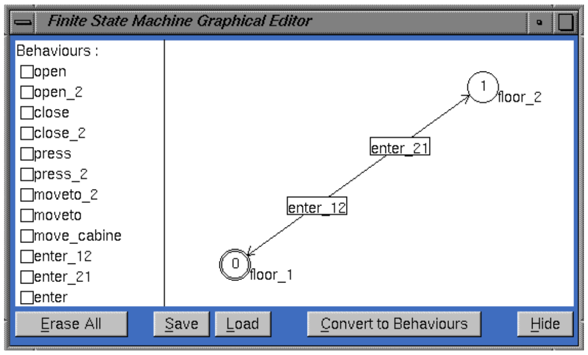


Fig. 9.13. A state machine for a two-stage lift functionality. The double circle state is the current state, and the rectangular boxes show the interaction needed to change of state. For example, to change from floor_1 to floor_2 state, interaction enter_12 is required

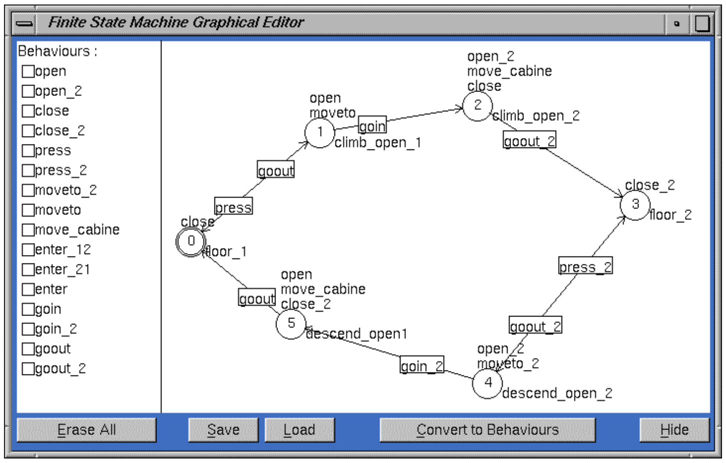


Fig. 9.14. A state machine considering intermediate states. The figure also shows behaviors associated with states, to be triggered whenever the object enters that state

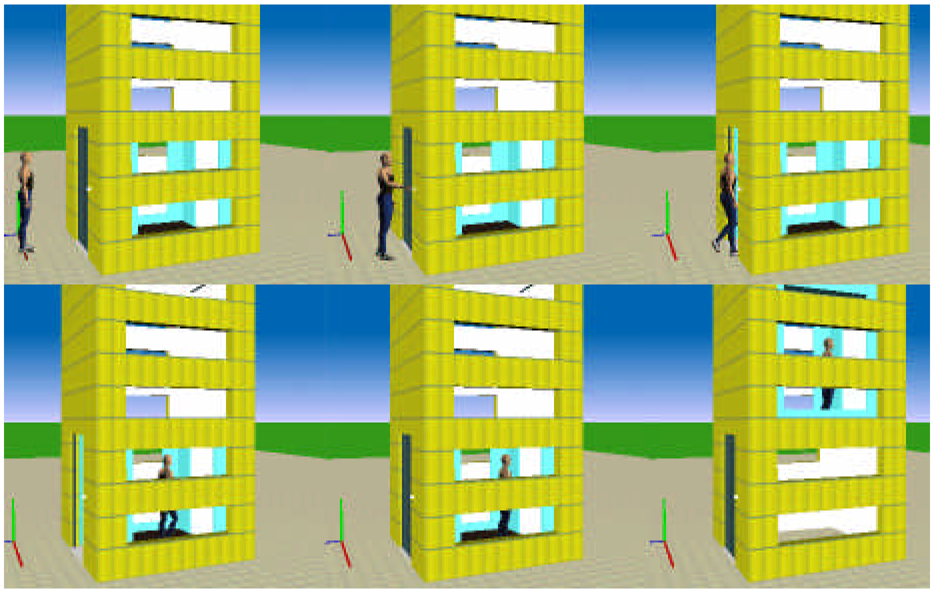


Fig. 9.15. An actor entering the smart lift

9.7 Remaining Problems

The smart object approach proposes software architecture able to coherently control actor-object interactions for real-time applications. However, this architecture relies that difficult sub problems can be solved:

• Simple yet powerful definition of concurrent behaviors. The smart object approach proposes a simple script language organized as interaction plans to describe the object functionality. Even if such solution can cope with complex cases, a more general, intuitive and simple way to define functionalities and behaviors is still a topic of intense research.

• Planning and learning low-level manipulation procedures. Rather then always using predefined geometric parameters for general manipulations, robust algorithms still need to be developed in order to calculate feasible motions, taking into account collision detection with the manipulation space, and automatic decision and dynamic update of hand configurations and placements. For instance, the actor's hand configuration should change during opening a door, and the same for the whole skeleton configuration. Opening a door realistically would also involve a combination of walking and hand manipulation.

• Realistic human motion. Even when a feasible and correct motion is calculated, how to make this motion display natural and realistic? This is a difficult issue, which involves psychological states, and comfort issues. The most reasonable direction to research is to fill the gap between motion-capture animation and simulation/procedural animation. Some ideas are to mix pre-recorded motion (database driven or not), corrected with simple interpolation or inverse kinematics methods. The goal is to reuse realistic recorded human movements, parameterized for a wide range of object manipulations, and optimized to reach comfort criteria.

• Learning and adaptation. Virtual actors should be able to learn the motions, and also the usefulness of objects, while interacting with them. This would an adaptation of the motions towards more realistic results. Learning and adaptation processes from the Artificial intelligence field are still to be applied to enhance actor-object interactions. Smart objects have been used connected to rule-based systems (figure 9.16) and or virtual life algorithms (figure 9.17) controlling the decisions of which interactions to perform, but still nothing has been done on learning and adaptation on the motion level.

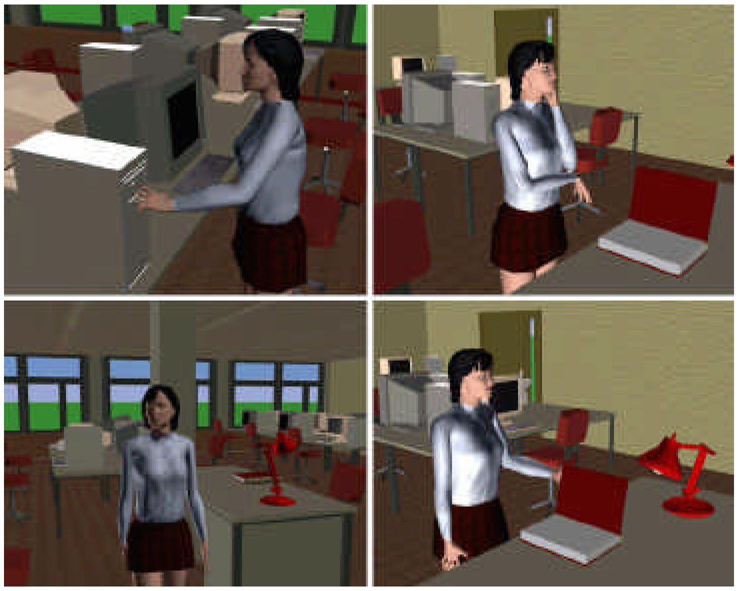


Fig. 9.15. Snapshots of a simulation controlled with Lisp plans, in order to follow a given storyboard [45].

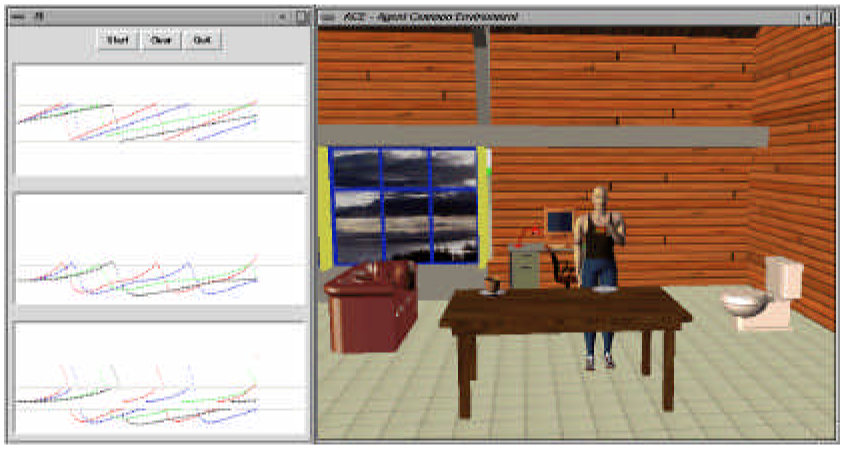


Fig. 9.16. A snapshot of a virtual life simulation based on smart objects. The curves on the left show the variation of the internal motivational parameters of the virtual human, at different levels in the action selection model. When actions are selected a corresponding object interaction is retrieved and executed [46]

References

[1] M. Kallmann, “Object Interaction in Real-Time Virtual Environments”, DSc Thesis 2347, Swiss Federal Institute of Technology - EPFL, January 2001.

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

[3] 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.

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

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

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

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

[8] M. Wooldridge, and N. R. Jennings, “Intelligent Agents: Theory and Practice”, Knowledge Engineering Review, 10(2), June, 1995.

[9] 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.

[10] 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.

[11] Motivate game engine from motion Factory (www.motion-factory.com).

[12] NeMo game engine (www.nemosoftware.com).

[13] 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.

[14] R. Bindiganavale, W. Schuler, J. Allbeck, N. Badler, A. Joshi, and M. Palmer, “Dynamically Altering Agent Behaviors Using Natural Language Instructions”, Proceedings of the 4th Autonomous Agents Conference, Barcelona, Spain, June, 293-300, 2000.

[15] F. Lamarche and S. Donikian, “The Orchestration of Behaviours using Resources and Priority Levels”, EG Computer Animation and Simulation Workshop, 2001.

[16] Virtual Reality Modeling Language - 39L (www.39l.org).

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

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

[19] Catia V5 Product from IBM (www-3.ibm.com/solutions/engineering/escatia.nsf/ Public/ know).

[20] R. Boulic, Z. Huang, and D. Thalmann, “A Comparison of Design Strategies for 3D Human motions”, in Human Comfort and Security of Information Systems, ISBN 3-540-62067-2, 1997, 305pp.

[21] 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.

[22] H. Rijpkema and Michael Girard, “Computer Animation of Knowledge-Based Human Grasping”, Proceedings of ACM SIGGRAPH '91, 25(4), July 1991, pp. 339-348.

[23] 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.

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

[25] J.-C. Latombe, “Robot motion Planning”, ISBN 0-7923-9206-X, Kluwer Academic Publishers, Boston, 1991.

[26] “Robot motion Planning and Control”, Jean-Paul Laumond, Ed., Lectures Notes in Control and Information Sciences 229, Springer, ISBN 3-540-76219-1, 1998, 343p.

[27] Kineo company (http://www.kineocam.com/).

[28] Meyer, J.A., Doncieux, S., Filliat, D. and Guillot, A. Evolutionary Approaches to Neural Control of Rolling, Walking, Swimming and Flying Animats or Robots. In Duro, R.J., Santos, J. and Graña, M. (Eds.) Biologically Inspired Robot Behavior Engineering. Springer Verlag.

[29] S. Nolfi and D. Floriano, “Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines”, ISBN 0-262-14070-5, 384p, November 2000.

[30] K. Sims, “Artificial Evolution for Computer Graphics”, Proceedings of ACM SIGGRAPH '91, 25(4), July 1991, pp. 319-328.

[31] M. de Panne, and E. Fiume “Sensor-actuator networks”, Proceedings of ACM SIGGRAPH '93, August 1993, pp. 335-342.

[32] A. Watt and M. Watt, “Advanced Animation and Rendering Techniques”, Wokingham: Addison-Wesley, 1992.

[33] P. Baerlocher, “Inverse kinematics Techniques for the Interactive Posture Control of Articulated Figures”, DSc Thesis 2393, Swiss Federal Institute of Technology - EPFL, 2001.

[34] D. Tolani, and N. Badler, “Real-Time Inverse kinematics of the Human Arm”, Presence 5(4), 393-401, 1996.

[35] X. Wang, and J. Verriest, “A Geometric Algorithm to Predict the Arm Reach Posture for Computer-aided Ergonomic Evaluation”, The Journal of Visualization and Computer Animation, 9, 33-47, 1998.

[36] D. Wiley, and J. Hahn, “Interpolation Synthesis for Articulated Figure motion”, Virtual Reality Annual International Symposium, Albuquerque, New Mexico, March, 1997.

[37] R. Bindiganavale, and N. Badler, “motion Abstraction and Mapping with Spatial Constraints”, Proceedings of CAPTECH’98, Lecture Notes in Artificial Intelligence 1537, Springer, ISBN 3-540-65353-8, 70-82, 1998.

[38] Y. Aydin, and M. Nakajima, “Database Guided Computer Animation of Human Grasping Using Forward and Inverse kinematics”, Computers & Graphics, 23, 145-154, 1999.

[39] J.-C. Nebel, “Keyframe interpolation with self-collision avoidance”, Eurographics Workshop on Computer Animation and Simulation, 77-86, Milan, Italy, September 1999.

[40] M. Kallmann and D. Thalmann, “Modeling Behaviors of Interactive Objects for Real Time Virtual Environments”, Journal of Visual Languages and Computing, Volume 13 Number 2, April 2002.

[41] J. Wernecke, “The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor Rel. 2”, Addison-Wesley, ISBN 0-201-62495-8, 1994.

[42] Fast Light Toolkit (www.12.org).

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

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

[45] 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.

[46] E. de Sevin, M. Kallmann and D. Thalmann, “Towards Real Time Virtual Human Life Simulations”, Submitted to Computer Graphics International Symposium 2001.