In Proceedings of the 9th Eurographics Workshop on Animation and Simulation (EGCAS), Lisbon, Portugal, 1998, 73-86

Modeling Objects for Interaction Tasks

Marcelo Kallmann and Daniel Thalmann

EPFL Computer Graphics Lab – LIG
CH-1015 – Lausanne – Switzerland
{kallmann, thalmann}@lig.di.epfl.ch

Abstract. This paper presents a new approach to model general interactions between virtual human agents and objects in virtual worlds. The proposed framework is designed to deal with many of the possible interactions that may arise while simulating a virtual human performing common tasks in a virtual environment. The idea is to include within the object description, all the necessary information to describe how to interact with it. For this, a feature modeling approach is used, by means of a graphical user interface program, to identify object interaction features. Moving parts, functionality instructions and interaction locations are examples of some considered features. Following this approach, the control of the simulation is decentralized from the main animation control, in the sense that some local instructions on how to deal with the object are encapsulated within the object itself. To illustrate the approach, some examples are shown and discussed.

1 Introduction

The necessity to model interactions between an object and a virtual human agent (here after just referred to as an agent), appears in most applications of computer animation and simulation. Such applications encompass several domains, as for example: virtual autonomous agents living and working 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]. An example of an application using agent-object interactions is presented by Johnson et al [12], whose purpose is to train equipment usage in a populated virtual environment.

Commonly, simulation systems perform agent-object interactions for specific tasks. Such approach is simple and direct, but most of the time, the core of the system needs to be updated whenever one needs to consider another class of objects.

Autonomous agent applications try to model agent's knowledge using recognition, learning and understanding techniques from information retrieved through sensors [15]. Agent's knowledge is then used to solve all possible interactions with an object. Even in such a case, some information of intrinsic object functionality must be provided. Consider the example of opening a door: just the rotation movement of the door is provided a priori. All other actions should be planned by agents knowledge: walking to reach the door, searching for a knob, deciding which hand to use, moving body limbs to reach the knob, deciding which hand posture to use, turning the knob, and finally opening the door. This simple example illustrates how complex it can be to perform a simple agent-object interaction task.

To overcome such difficulties, a natural way is to include within the object description, more useful information than only intrinsic object properties. Some proposed systems already use this kind of approach. In particular, the *object specific reasoning* [13] creates a relational table to inform object purpose and, for each object graspable site, the appropriate hand shape and grasp approach direction. This set of information may be sufficient to perform a grasping task, but more information is needed to perform different types of interactions.

This paper describes a framework to model general agent-object interactions based on objects containing interaction information of various kinds: intrinsic object properties, information on how-to-interact with it, object behaviors, and also expected agent behaviors. Compared to the work of Levinson [13], our approach extends the idea of having a database of interaction information. For each object modeled, we include the functionality of its moving parts and detailed commands describing each desired interaction, by means of a dedicated script language.

A feature modeling approach [21] is used to include all desired information in objects. A graphical interface program permits the user to interactively specify different features in the object, and save them as a script file. We call such an object, modeled with its interaction features, as *Smart Object*.

The adjective *smart* has been used in a number of different contexts. For instance, Russel et al [18] and Pentland [17] discuss interactive spaces instrumented with cameras and microphones to perform audio-visual interpretation of human users. This capacity of interpretation made them *smart spaces*. In our case, an object is called *smart* when it has the ability to describe its possible interactions.

A parallel with the object oriented programming paradigm can also be made in the sense that each object encapsulates data. There is a huge literature about Object Oriented Design; an introduction to the theme is presented by Booch [4].

Different simulation applications can retrieve useful information from a Smart Object to accomplish a desired interaction task. The main idea is to provide a Smart Object with a maximum of information to attend different possible applications for the object. Each application will implement its *specific Smart Object reasoning* that will use only the applicable object features for its specific case.

The Smart Object approach introduces the following characteristics in an animation system:

- Decentralization of the animation control. By applying object and agent behaviors stored in a Smart Object, a lot of object-specific computation is released from the main animation control.
- Reusability of designed Smart Objects. A Smart Object can be modeled first for a specific application, and updated if needed for other applications without changing the usability of the original design.
- A simulation-based design is naturally achieved. The designer can take control of the loop: design, test and re-design. A designed Smart Object can be easily inserted into a simulation program, which gives feedback for improvements in the design.

Although the main focus here is to model interactions with virtual human agents, the presented Smart Object framework is also designed to be useful in interactions of other natures, as for example, 3D interactions using virtual reality devices.

In order to demonstrate some results of the proposed framework, two applications are described. One application reads simple text instructions to direct agents to interact with Smart Objects. The other application is a crowd simulation [14] in which interactions between some individual agents and Smart Objects are integrated.

The following sections are organized as follows: Section 2 gives an overview of concepts adapted from the Feature Modeling area. Section 3 describes in detail the Smart Object description and the graphical interface program used to model its features. Section 4 presents two applications using the Smart Object framework and section 5 concludes and presents some future work considerations.

2 Feature Modeling of Interactive Objects

Feature modeling is an expanding topic in the engineering field [16]. The word *feature* conjures up different ideas when presented to people from different backgrounds. A simple general definition, suitable for our purposes, is “a feature is a region of interest on the surface of a part” [20].

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 to give a good mental picture.

From the engineering point of view, it is possible to classify features in three main areas: functional features, design features and manufacturing features [16]. As we progress from functional features through design 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 some piece may be hard to describe and have little importance while really using the piece. A similar compromise arises in the Smart Object case. This situation is depicted in figure 1 and will be explained later.

A huge literature is available for the feature modeling technique in the scope of engineering. A good coverage of the theme is done by Shah and Mantÿla [21].

In a Smart Object, we propose a new class of features that are interaction-features for simulation purposes. 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 agent. For example, not only buttons, drawers and doors are considered as interaction features in an object, but also their movements and purposes.

As already mentioned, different interaction features are considered:

- 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 agent to perform each possible interaction with the object. For example: the position of some interaction part (like a knob or a button), specific hand interaction information (hand shape,

approach direction) and description of object movements that may control an agent (an escalator).

- Object behavior: an object can have various different behaviors, which may or may not be available, depending on the configuration of its state variables. For example, an automatic door can close only if some state variables are true: as one indicating that no agents are passing through the door, and another indicating that the door is open.
- Agent behaviors: associated with each object behavior, it is useful to have a description of some behavior that the agent should follow. For example, one possible behavior to model in an automatic door is to open itself when an agent comes nearby, give a feed point to the agent walk (in order to pass through the door), and then close.

Each application will implement its *specific Smart Object reasoning* that will make use only of the applicable object features for its specific case. For example, a virtual reality application in which the user wears a virtual glove to press a button to open a Smart Object door, will not make use of a proposed hand shape to press the button.

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 realistic results are obtained. As an example, consider a door that has a behavior to control an agent passing through it. So, an application can easily control an agent passing through the door, by just using a supplied path. In this case, minimal computation is needed. But such solution would not be general in the sense that all agents would pass the door exactly in the same way. To reach more realistic results, external parameters should also take effect, as for example, the current agent emotional state [3].

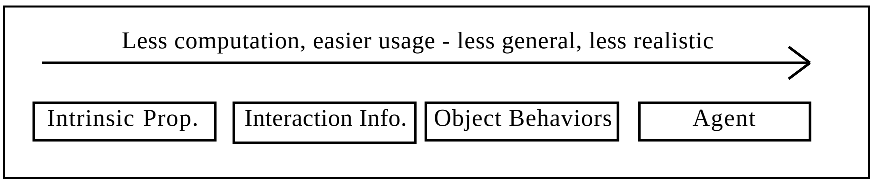

Less computation, easier usage - less general, less realistic
Intrinsic Prop.
Interaction Info.
Object Behaviors
Agent

Fig. 1. Compromise between less computation vs. realism when choosing object features to use.

A complete Smart Object proposes a detailed solution for each possible interaction with the object. In each application, the specific reasoning program can decide whether to use the proposed solution or not. Doing so, the development of a specific reasoning is simplified by starting with following all proposed solutions, and then, gradually adjusting each solution to reach desired special cases.

The next section describes in detail how, and how far, the directives showed above are accomplished by the Smart Object.

3 The Smart Object Description

All the features of a Smart Object are described in a text-based script file. The geometry of the object is saved in any desired format being just referred from the script. This keeps the script file general enough to work in a variety of applications.

As most parameters in the script file are not easily defined in a text editor, a graphical interface program called *Smart Object Modeler* is used. The modeler is implemented using the Open Inventor library as graphical toolkit, and a simplified user interface layer over Motif.

3.1 Considered Features

The considered features in a Smart Object are identified in such a way as to give direct useful information for the many motion generators available in a human agent implementation. We have been designing and testing the Smart Object framework within the HUMANOID environment [6], and the AGENTlib motion control architecture [7]. The AGENTlib provides automatic management of action combination over an agent. For example, in such architecture, motion generators from an Inverse Kinematics module, and from a walking model [5], are easily combined and assigned to a given agent. The role of a Smart Object is to provide the needed parameters to correctly initialize each motion generator.

The Modeler is organized in different dialog boxes, according to the feature type being described. The organization of these dialogs has a direct relation with the script file organization. Each moveable part of an object has a separate geometry file. All parts are loaded according to a specified hierarchy, and they can be re-positioned. Figure 2 shows the two dialogs used to load and locate the parts of an automatic door. The positioning of each part can be done interactively using one of the six manipulators available in the Open Inventor. However, for precise alignments, it is also possible to explicitly edit each value of the final transformation matrix. Other intrinsic object properties such as physical attributes and text strings indicating semantic meaning object purpose are entered in another specific dialog.

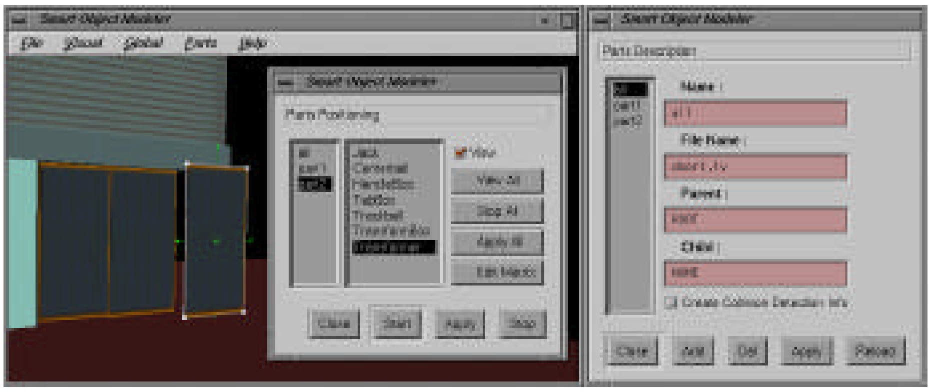


Fig. 2. Specifying part positioning (left) and part hierarchy (right).

Another intrinsic property is the definition of how each part of the object can move. Movements are called by *actions*, and are in fact, rotations or translations. To define an action, the same system of manipulators is used: the user selects some part of the object and presses the start button. The chosen manipulator will appear to let the user move the selected part. When the part moves to the desired final position, the user presses the stop button. The action is then calculated as the equivalent transformation matrix. Each action is independent of a part, so that the same defined movement can be used for different finalities.

The features described above constitute all considered intrinsic object features. Other features are used to define behaviors and interaction information: *positions* and *gestures.* A position is, in fact, a vector in space. Each position is identified by a name and can be later referenced from different behaviors. Figure 4 shows 3d vectors indicating key positions for the agent to reach for various Smart Objects.

A hand gesture-feature defines mainly an approach vector and a hand shape to aid agents to perform some dexterous activities, such as: grasping, button pressing, pulling a drawer, etc. Many gestures can be defined, and later referenced by their name in the behavior definitions.

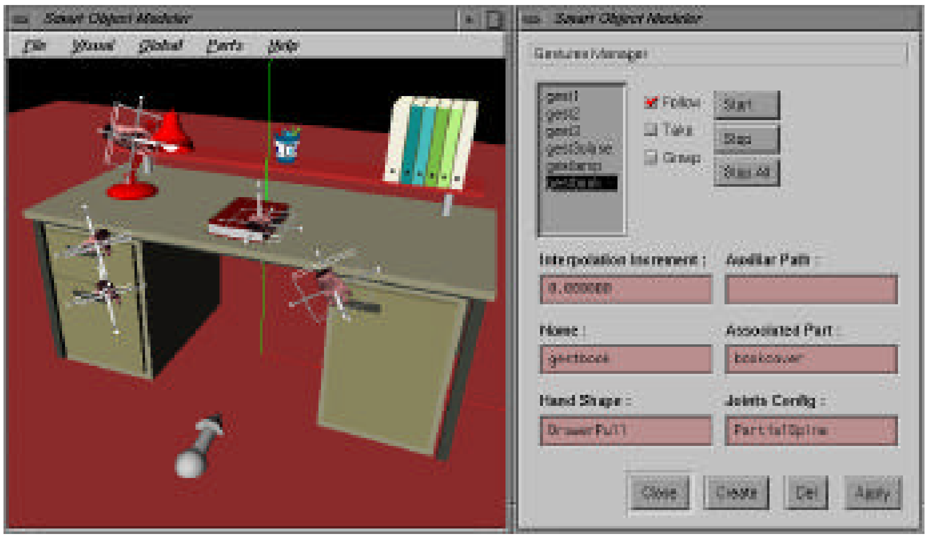
Figure 3 shows the dialog used to define gestures. Each gesture can be located in space also by using manipulators. The hand shape of a gesture is defined by choosing one pre-defined hand posture file. Posture files can be updated using a dedicated interface program. Some pre-defined postures follow the classification given by Cutkosky [8]. This classification envisages grasping tasks, and hence, other postures such as button pressing, pulling and pushing are added.

Another parameter defined for a gesture is a joint configuration. A joint configuration supplies necessary parameters to the Inverse Kinematics motor, when the agent performs the gesture. Some parameters are joints to use (e.g., whether the arm alone or also some spinal joints) associated with weights to indicate which joint angles have more priority to move in relation to others.

A gesture has three flags that are applied in conjunction with an associated part. One indicates whether the gesture movement can be integrated with a grasping of the associated part. The second flag indicates if the part can be taken by the agent. The last flag says if the part has to follow the hand movement (such as a drawer being drawn by the hand).

The case of pulling a drawer exemplifies a situation where the reasoning algorithm faces the compromise between practicality and realism depicted in figure 1. A simple way of making an actor opening a drawer is to apply inverse kinematics to make the hand follow the opening motion of the drawer. The object provides all necessary parameters, including the action matrix to open the drawer that can be interpolated. But for more realistic results, the drawer action matrix can be used only to restrict its movement, and physical forces applied from the agents hand should be calculated in order to make the drawer move.

Figure 3 shows some gestures being modeled in a desk. Each gesture can be viewed as a hand object in the defined position and orientation. For example, in a gesture to open a drawer, the inverse kinematics motor can use the final hand location as end effector and the set of proposed joints to move. Another motion generator is concomitantly applied to interpolate the current hand shape to the final shape. After the hand position is reached, local geometric information can be used to perform subsequent movements, according to the gesture flags.



gest1
gest2
gest3
gest3close
gestlamp
gestbook

[✓ Follow] Take Grasp

Interpolation Increment : 0.000000
Auxiliar Path :
Name : gestbook
Associated Part : bookcover
Hand Shape : DrawerPull
Joints Config : PartialSpine

Fig. 3. Defining hand gestures (see Appendix).

Figure 4 shows three Smart Objects being modeled: an automatic door, a table with graspable fruits, and a door that opens after a button is pressed. In each case, some defined gestures and positions are shown.

The next subsection describes how the features modeled until now are referenced by behavior descriptions.

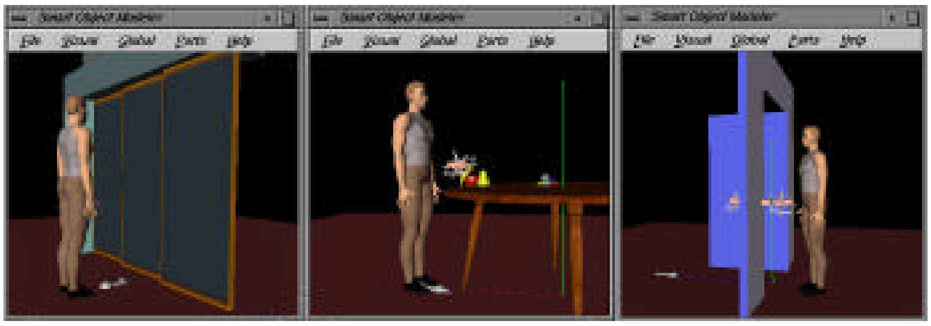


Fig. 4. Gestures and Positions being defined in different situations (see Appendix).

3.2 Behavior Features

To rule how the object will interact with an agent, three more types of features are used: commands, variable states, and behaviors.

A command is the full specification of some movement. It connects an action with an object part, and defines how the action is to be parameterized. Many commands can have the same identifying name, meaning that all should be performed in parallel.

Variables can be declared to define an object state. Instructions to check, set a value, or add a value to a variable are provided to change and query a current state.

Finally, a number of behaviors can be defined. Each behavior is given a name, and is composed of a sequence of instructions, called as *behavior items*. It is possible to view a behavior as a subroutine in a script program, where each behavior item is an instruction. Each behavior can be classified as either available or unavailable, depending on the current object state. For example, a behavior to open a door will be available only if the door state *open* is false, i. e., the door is closed.

Many behavior items are provided, and it is better to illustrate them with a concrete example. Listing 1 shows the commands, variables and behaviors used to define the behavior “enter” in the automatic door example. The following paragraphs explain the behaviors presented in listing 1.

```
 COMMANDS
 # name          action         part     ini    end    inc
 cmd_open_door   translation1   part1    0.00   1.00   0.05
 cmd_open_door   translation2   part2    0.00   1.00   0.05
 cmd_close_door  translation1   part1    1.00   0.00   0.05
 cmd_close_door  translation2   part2    1.00   0.00   0.05
 END # of commands

 VARIABLES
   open          0.0
   passing       0.0
 END # of variables
BEHAVIOR go_out1                      BEHAVIOR go_out2
   Subroutine                           Subroutine
   addvar      passing  1.0             addvar      passing  1.0
   gotopos     pos_out1                 gotopos     pos_out2
   addvar      passing -1.0             addvar      passing -1.0
 END # of behavior                    END # of behavior

 BEHAVIOR open_door                   BEHAVIOR close_door
   Subroutine                           Subroutine
   checkvar    open     0.0             checkvar    open     1.0
   changevar   open     1.0             checkvar    passing  0.0
   docmd       cmd_open_door            changevar   open     0.0
 END                                    docmd       cmd_close_door
                                      END
BEHAVIOR enter1
  Subroutine                           BEHAVIOR enter2
  gotopos    pos_enter1                  Subroutine
  dobh       open_door                   gotopos    pos_enter2
  dobh       go_out1                     dobh       open_door
  dobh       close_door                  dobh       go_out2
END # of behavior                        dobh       close_door
                                       END # of behavior
BEHAVIOR enter
  Dobhifnear enter1 pos_enter1
  Dobhifnear enter2 pos_enter2
END # of behavior
```

Listing 1. Some behaviors used in the automatic door example.

The door was designed to correctly deal with many agents passing through it at the same time. So it has two state variables: *open*, indicating if the door is open or not, and *passing*, indicating how many agents are currently passing through it. All behaviors are declared as subroutines, which enforces them to be always unavailable. Only exception is behavior *enter* which is always available, as it does not have a *checkvar* item to test its availability.

When the application wants to control an agent entering the door, it searches for available behaviors in the Smart Object, and identifies the *enter* behavior as the desired one. This behavior simply calls another behavior, *enter1* or *enter2* depending whether the current agent position is near *pos_enter1*, or *pos_enter2* (pre-defined positions). This enables agents entering from any side of the door. Suppose that the behavior *enter1* is called. Then, each item of this behavior is carried out. The *gotopos* item gives a final position to the agent walk just before starting entering the door. At this point, if the door was not automatic, a gesture command *dogest* could be inserted to direct some specific agent action to open the door. Once the actor is ready to enter the door, three other behaviors are sequentially called: *open_door*, *go_out1*, and *close_door*.

The behavior *open_door* checks if the state variable *open* is false (zero), and if this is the case, the command to open the door is called and the state is changed. Note that the command *cmd_open_door* (similarly *cmd_close_door*) has two entries with the same name that makes the two parts of the door laterally translate at the same time. After the door is open, the behavior *go_out1* is called. This behavior updates the *passing* state variable value and supplies the position *pos_out1* to the agent to walk to the other side of the door. Finally, the behavior *close_door* is called. Then, the command to close the door is called if the variables *open* is true (one) and *passing* is zero, meaning that the door is open and no agents are currently walking trough the door. Only in that case, the door will then be closed.

It is difficult to define a closed and sufficient set of behavior items to use. Moreover, a complex script language to describe behaviors [19] is not the goal. The idea is to keep a simple script format with a direct interpretation to serve as guidance for reasoning algorithms, which any designer can create and test.

In the other hand, there are cases where a more complex functionality requires a complete programming language to be described. The functionality of a complete elevator (figure 8) is an example of how far the actual script language can go. To model more complex cases we intend to propose in the script language the possibility to call extern functions written in a C-like programming language. In this way, designers will still be able to model a basic functionality. Only some external functions will be provided by programmers in some specific cases. In section 4, other considerations about these objects are discussed.

Behavior definitions form an interface between stored objects features and the application-specific object reasoning. Figure 5 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 command of the behavior definition, retrieving the necessary data from the Smart Object.

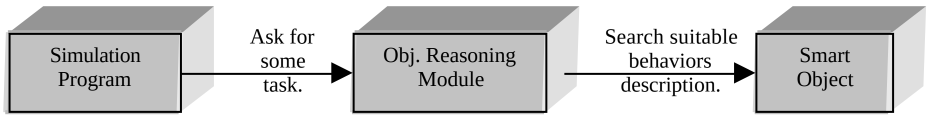


Fig. 5. Diagram showing the connection between the modules of a typical Smart Object application. Arrows represent function calls.

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 must be made while modeling objects with too many potential interactions. In such cases, in order to exercise a greater control over the interactions, it is better to model a Smart Object for each part of the object, containing only basic behaviors. For example to model an agent 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, thus permitting more personalized interactions. 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 the Smart Object.

In the current approach, object behaviors and agent behaviors are mixed in the same script language. In this way, whenever an agent starts an interaction, a process is created to interpret both agent and object behaviors. This approach is more direct to model, but for more complex objects, we intend to have one process always performing object behaviors, and whenever an agent starts an interaction, another process starts performing only agent behaviors. Processes can synchronize by checking global state variables.

3.3 Analysis of the Considered Features

By defining a general and complete set of behaviors, it is always possible to have a complete solution to perform a desired interaction.

By interpreting behavior definitions, the animation control is decentralized. Suppose the case of a complex simulation environment where many semantic rules are being processed all the time. One approach to direct such high level tasks is based on PaT-Nets [22]. In this case, when an agent is directed to go from one room to another, all local instructions on how to deal with encountered doors are stored inside each Smart Object door. This approach separates the high level planing from the specific low level Smart Object reasoning.

Another key characteristic is reusability. Reusability is reached in two levels. A first level is by using the same Smart Object in different applications. For this, different specific behaviors can be added and called when needed. Another level of reusability is achieved while designing a new Smart Object. In this case, the designer can merge any desired feature from some previously designed Smart Object.

It is also important to have the Smart Object specific reasoning easily connected with high level planners. This is possible by observing behavior names, and the object purpose description. This point is addressed in the next sub-section.

3.4 Connecting Behaviors to High Level Planners

There is a vast literature involving high level planners to guide an animation, and much of it focuses on connecting language to the animation, by interpreting and expanding given text instructions [1, 23, 10, 11]. In particular, Webber et al [23] identify the limited perception of agents as a main limitation to correctly interpret such text instructions, resulting in a poor knowledge construction. The proposed Smart Object framework can minimize this difficulty by providing in each Smart Object purpose-features that the agent can easily access to update its knowledge.

The Smart Object description provides different text statements that can be analyzed and used to update agents knowledge: text strings identifying behaviors, a semantic name, and general purposes description. For example, if a command like “Go out of the room” is given, the high level planner can map the action “go out” with a list of key names as “doors, windows, elevators, escalators”. Then, it can search for the nearest Smart Object having one of these key names as its semantic name. Once the Smart Object is found, the next step is to choose the correct behavior by analyzing the names of each available behavior.

Section 4 presents some results obtained using a prototype of a system to control agents by giving simple text instructions based in the Smart Object behaviors.

4 Example of Smart Object Applications

Two applications have been developed using the Smart Object framework. One interprets simple text instructions to choose the closest Smart Object behavior to perform a desired task, and the other is a crowd simulation [14] that uses automatic doors and escalators as Smart Objects.

Figure 6 shows the layout of the application that interprets simple text instructions. It consists of a text entering area, a main graphical output window, and some auxiliary dialogs (as one describing available behaviors of a selected Smart Object). In the application, the user enters a command. If this is not recognized as a primitive command, the program looks into all Smart Objects in the scene to identify some text string.

In the example of figure 6, a simple command “close” can be given. After a string search, the program identifies the Smart Object having the closest available behavior, and executes it.

This application does not interpret natural language instructions, but makes a smart comparison of strings based on pre-designed strings contained in the Smart

Objects. An advantage of such an approach is that the designer can incorporate all strings that will be searched into the objects. Thus, Smart Objects can have their design fine tuned during the simulation to achieve the best match. This is achieved by having two applications running at the same time: the modeler and the simulator. Each time a Smart Object script is updated, a simple *reload* command is performed in the simulator to update objects information.

It is important to model behavior names and all other text information in a consistent way. So that many other high-level planning algorithms can be based on these identifiers. For example, a real natural language interpreter can be built on top of the primitive command interpreter. Agent perceptions can also be used to filter the information. For example, an agent willing to go out of a room can start walking randomly in a room until discovering a Smart Object with a semantic name *door*.

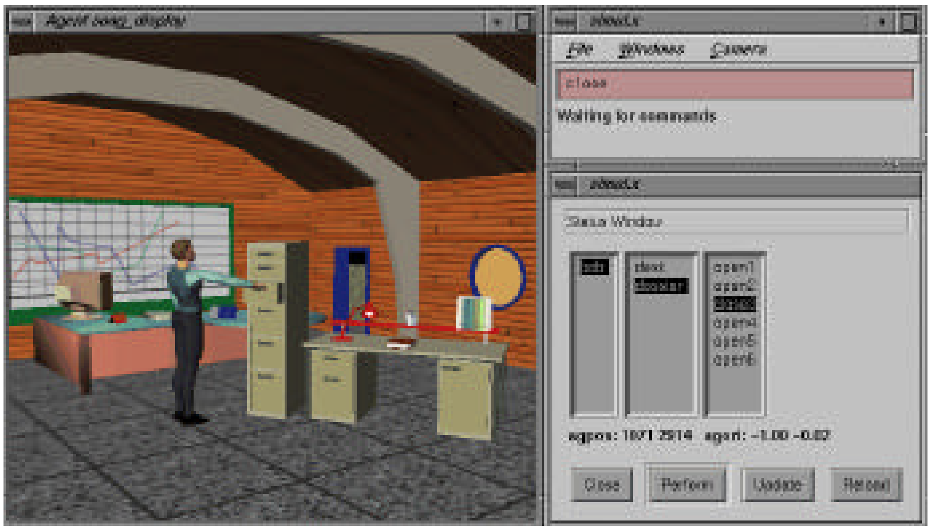


Fig. 6. A text based simulation application (see Appendix).

Another application using Smart Objects is a crowd simulation [14] inside a train station (figure 7). In this example, the decentralization of the animation control is an important characteristic.

The crowd simulator has its own control procedures and parameters for each controlled agent. One parameter supplied to a group of agents is a path containing “interest points” to visit. The crowd simulator checks whether an interest point contains some information pertaining to the Smart Object type. In this case, when such an interest point is reached, the control of the agent is released from the crowd simulator and passed to the Smart Object reasoning. The crowd simulator waits until the agent completes its interaction and regains control over it.

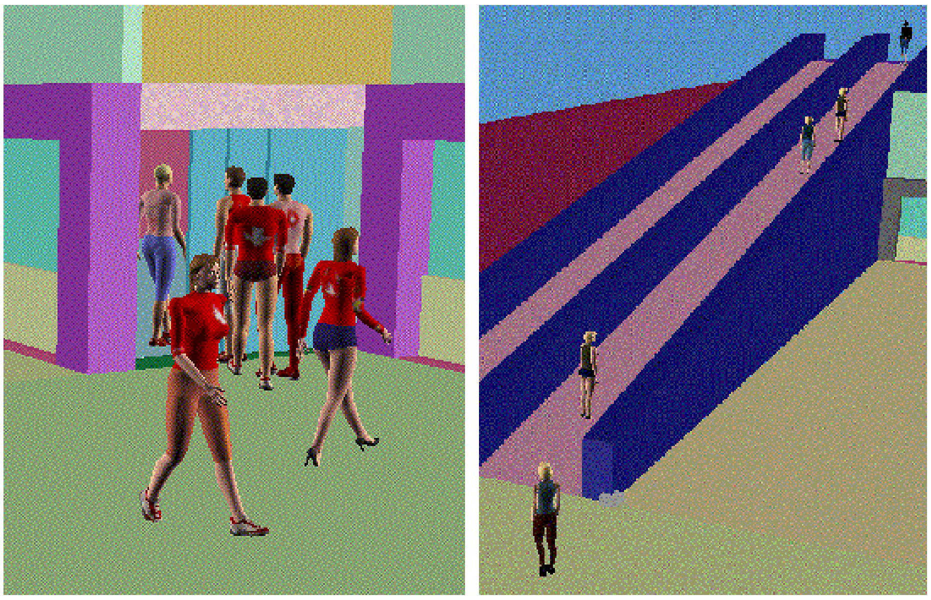


Fig. 7. Agents of a crowd interacting with an automatic door and an escalator.

Different Smart Objects are used in this simulation: a clock, automatic doors, elevators and escalators. All of them possess default behaviors to perform an interaction. In particular, the clock has only one behavior to continuously move its pointers, so that it does not request an agent to interact.

Note that, in this kind of simulation, each Smart Object must be modeled to be able to interact with more than one agent at the same time. In addition, the specific reasoning module must keep track of all current agents being controlled.

The default behavior of the automatic door is the same behavior *enter* shown in the listing 1, but considering more pre-defined positions to avoid collisions by always sending agents to an unused position. For the escalator, the default behavior is to translate the agent to the other side of the escalator, following an action-feature (section 3.1) that defines the translation.

Figure 8 shows agents interacting with an elevator. The default behavior of the elevator describes a sequence of commands to move the agent from its current floor to the other one. Special considerations are needed to deal with more than one agent at the same time. For example, different pre-defined positions are used to arrange more than one agent inside the elevator.

Other details are handled. Only the first agent arriving will press the call button while the others will only wait the elevator door open. Special considerations are also done for agents arriving from different floors at the same time. Such cases give an idea of the complexity of the model. The actual version deals correctly with three agents at the same time, has 18 pre-defined positions, 19 behaviors and 10 state variables.

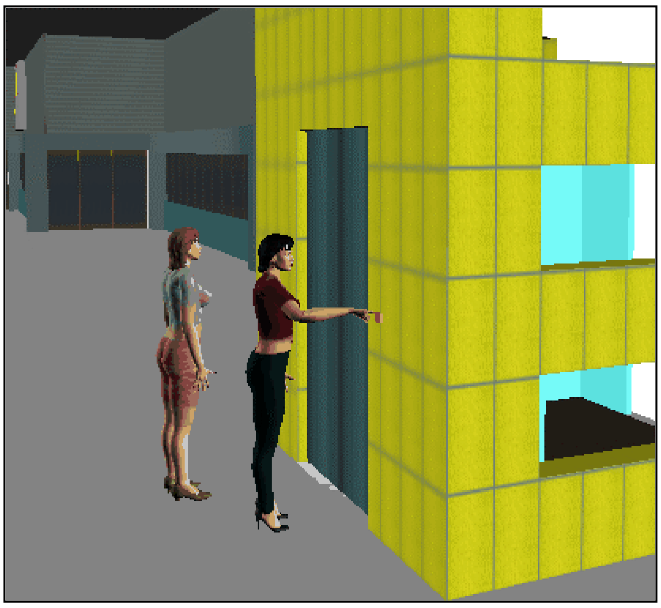


Fig. 8. Agents interacting with an elevator (see Appendix).

5 Conclusions and Future Work

This paper describes the Smart Object framework used to model general agent-object interactions. The approach used is based on a feature modeling of different kinds of interaction features, using a graphical interface program. This framework minimizes many difficulties encountered when using planning and object reasoning algorithms by achieving the following characteristics:

- Easy simulation-based design of Smart Objects.
- Easy reusability of Smart Objects to suit different applications.
- Decentralization of the animation control.
- Easy connection with high level planners.

Two applications using the proposed framework are described. These applications are still being developed and they use the same Smart Object reasoning algorithm. Many enhancements in this reasoning algorithm are being made, and this will certainly influence the creation of new interaction-features.

Both applications are being developed under the framework of AGENTlib [7] where it is also considered as an agent. A general reasoning system to interact object-agents with human-agents is being developed.

The main goal of the proposed framework is to be able to model the many agent-object interactions encountered in an urban environment simulation of a virtual city [9]. This implies connecting Smart Objects with a rule-based high-level planner, a crowd displacement simulation control, and environmental information.

Enhancements in the description of the object functionality are being done by including an optional and more powerful programming language, and by separating agent behaviors from object behaviors (see section 3.2). These improvements envisage also the usage of the Smart Object paradigm in applications of 3D interaction and manipulation of objects by real humans using virtual reality devices.

6 Acknowledgments

The authors are grateful to Dr. R. Boulic, P. Baerloch, S. R. Musse and Dr. S. Bandi for fruitful discussions, and to the anonymous referees for valuable suggestions. This research was supported by the Swiss National Foundation for Scientific Research and the Brazilian National Council for Scientific and Technologic Development (CNPq).

7 References

1. N. N. Badler, B. Webber, J. Kalita, and J. Esakov, "Animation from Instructions", In N. N. Badler, B. Barsky, and D. Zeltzer, eds, "Making Them Move: Mechanics, Control, and Animation of Articulated Figures", 51-93, Morgan-Kaufmann, 1990.
2. N. N. Badler, "Virtual Humans for Animation, Ergonomics, and Simulation", IEEE Workshop on Non-Rigid and Articulated Motion, Puerto Rico, June 97.
3. P. Bécheiraz and D. Thalmann, "A Model of Nonverbal Communication and Interpersonal Relationship Between Virtual Actors", Proceedings of Computer Animation'96, Geneva, 58-67, 1996.
4. G. Booch, "Object Oriented Design with Applications", The Benjamin Cummings Publishing Company, Inc., ISBN 0-8053-0091-0, 1991.
5. R. Boulic, N. Magnenat-Thalmann, and D. Thalmann, "A Global Human Walking Model with Real Time Kinematic Personification", The Visual Computer, 6, 344-358, 1990.
6. R. Boulic, T. Capin, Z. Huang, P. Kalra, B. Lintermann, N. Magnenat-Thalmann, L. Moccozet, T. Molet, I. Pandzic, K. Saar, A. Schmitt, J. Shen, and D. Thalmann. "The HUMANOID Environment for Interactive Animation of Multiple Deformable Human Characters", Proceedings of EUROGRAPHICS 95, Maastricht, The Netherlands, August 28 - September 1, 337-348, 1995.
7. R. Boulic, P. Becheiraz, L. Emering, and D. Thalmann, "Integration of Motion Control Techniques for Virtual Human and Avatar Real-Time Animation", In Proceedings of the VRST'97, 111-118, 1997.
8. M. R. Cutkosky, "On Grasp Choice, Grasp Models, and the Design of Hands for Manufacturing Tasks", IEEE Transactions on Robotics and Automation, Vol. 5, No. 3, 1989, 269-279.
9. N. Farenc, S. R. Musse, E. Schweiss, M. Kallmann, O. Aune, R. Boulic, and D. Thalmann, "One Step towards Virtual Human Management for Urban Environments Simulation", ECAI'98 Workshop of Intelligent Virtual Environments, 1998.
10. C. W. Geib, L. Levison, and M. B. Moore, "SodaJack: An Architecture for Agents that Search for and Manipulate Objects", Technical Report MS-CIS-94-13, University of Pennsylvania, 1994.

11. C. W. Geib, “The Intentional Planning System: ItPlanS”, Proceedings of AIPS, 1994.

12. W. L. Johnson, and J. Rickel, “Steve: An Animated Pedagogical Agent for Procedural Training in Virtual Environments”, Sigart Bulletin, ACM Press, vol. 8, number 1-4, 16-21, 1997.

13. L. Levison, “Connecting Planning and Acting via Object-Specific reasoning”, PhD thesis, Dept. of Computer & Information Science, University of Pennsylvania, 1996.

14. S. R. Musse and D. Thalmann, “A Model of Human Crowd Behavior: Group Inter-Relationship and Collision Detection Analysis”, EGCAS’97, Eurographics Workshop on Computer Animation and Simulation, 1997.

15. H. Noser, O. Renault, D. Thalmann, “Navigation for Digital Actors Based on Synthetic Vision, Memory, and Learning”, Computer & Graphics, volume 19, number 1, 7-19, 1995.

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

17. A. Pentland, “Machine Understanding of Human Action”, 7th International Forum on Frontier of Telecom Technology, 1995, Tokyo, Japan.

18. K. Russell, T. Starner, and A. Pentland, “Unencumbered Virtual Environments”, International Joint Conference on Artificial Intelligence Entertainment and AI, ALife Workshop, 1995.

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

20. M. J. Pratt, and P. R. Wilson, “Requirements for Support of Form Features in a Solid Modeling System”, Report R-85-ASPP-01, CAM-I, 1985.

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

22. B. Webber, and N. Badler, “Animation through Reactions, Transition Nets and Plans”, Proceedings of the International Workshop on Human Interface Technology”, Aizu, Japan, October, 1995.

23. B. Webber, N. N. Badler, B. Di Eugenio, C. Geib, L. Levison, and M. Moore, “Instructions, Intentions and Expectations”, Artificial Intelligence Journal, 73, 253-269, 1995.