A Behavioral Interface to Simulate Agent-Object Interactions in Real Time

Marcelo Kallmann and Daniel Thalmann

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

Abstract

This paper shows a new approach to model and control interactive objects for simulations with virtual human agents when real time interactivity is essential. A general conceptualization is made to model objects with behaviors that can provide: information about their functionality, changes in appearance from parameterized deformations, and a complete plan for each possible interaction with a virtual human. Such behaviors are described with simple primitive commands, following the actual trend of many standard scene graph file formats that connects language with movements and events to create interactive animations. In our case, special attention is given to correctly interpret object behaviors in parallel: situation that arrives when many human agents interact at the same time with one same object.

Keywords: Virtual Humans, Virtual Environments, Object Modeling, Object Interaction, Script Languages, Parameterized Deformations.

1 Introduction

The necessity to have interactive objects in virtual environments appears in most applications of the computer animation and simulation field. The need to offer interactivity is growing, especially due to the increasing popularity of many standard graphical file formats. Users want to easily control the behavior of objects. As an example, we can consider the evolution of the *Virtual Reality Modeling Language* graphical file format [23] from the first version to the version 97 containing already options to write script commands to route user events to scene actions.

More complex situations arise when applications need to control interactions between objects and virtual human agents (here after just referred to as an agent). Some examples of such applications are: 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]. As an example, an application to train equipment usage using virtual humans is presented by Johnson et al [11].

We have modeled interactive objects following the *smart object* conception described in a previous work [13]. The adjective *smart* has been used in different contexts, as for instance, to refer to computers that can understand human actions [19]. In our case, an object is called *smart* when it has the ability to describe its possible interactions.

The key idea in the smart object concept is that objects are able to propose a detailed description of how to perform each possible interaction with them. This is done by using pre-defined plans specified through scripted commands. Such commands describe the object behaviors and have access to important 3D parameters defined during the modeling phase of the object, by means of a friendly graphical user interface application. As an example, special locations where the agent can put its hands are 3D parameters that are specified using modeling tools and saved together with the object description. Then, during the simulation phase, such parameters can be retrieved in order to feed the many available motion generators to animate agents [6, 7].

When providing pre-modeled information, we are releasing the simulation program of many difficult planning and reasoning tasks. As the real time requirement is often present in nowadays systems, we are interested in minimizing all time-consuming operations during the simulation.

For example, by giving the location of a usual graspable site of the object, an inverse kinematics algorithm can be directly used to move the agent arm in the correct direction. We adopt such solutions of having pre-defined data to gain simplicity, greater user control and real time performance, even if we may end up with some loss of individuality in interactions.

In addition, deformation algorithms that can generate parameterized data can be used to display shape changes in real time. Some examples are some morphing algorithms [12, 14] and smooth multi-resolution shape changes [10]. Deformations are included as object properties, so that we can define, for example, that some agent action will trigger an object shape change.

Some proposed systems in the literature already use similar ideas. In particular, the *object specific reasoning* [15] creates a relational table to inform object purpose and specific data of each object graspable site. Such information is then used to perform, mainly, grasping tasks [9].

In our framework, we are interested in modeling objects that may have a complex functionality, and that might interact with more than one agent at a same time. A good example of such objects is a *smart lift* that can deal simultaneously with many agents during a crowd simulation [16, 21] inside a virtual city environment [8]. This lift example (figure 1) was shown in a previous work [13] together with other objects such as automatic doors and escalators.

Figure 1. A *smart lift* that has all necessary pre-modeled information to be able to take control over agents that want to enter from one floor and go to the other floor.

Once we have connected a script language with object movements and state variables, we are able to program complex object functionality. Using script languages to control human agents is a versatile solution, as shown for example by the IMPROV system [20]. On the other hand, it is important to maintain few and intuitive script commands in order to help designers to use them. To achieve such simplicity, we designed a dedicated script language to describe a set of behaviors that can be of different types. For example, it is possible to define that an object with level of details information has the behavior to change its own current display resolution. Other behaviors can change the speed of some moving part, or can take control of an agent and make it perform some interaction.

In such a concept, during the simulation, each object has its own module of control. This leads to a virtual environment where the knowledge of how objects work is dispersed in the scene. In this case, it is also possible to have autonomous agents learning objects' purposes by retrieving the information contained inside objects.

The previous work [13] showed in detail how smart objects are modeled, gave some examples of object behavior definitions, and discussed some applications based on smart objects. In this paper, we present some important solutions adopted by our current behavioral language, showing how far we can describe interactions still maintaining simplicity. In addition, we present how we use some concurrent programming techniques to be able to synchronize more than one agent interpreting an object behavior at the same time.

We also show new integrated behavioral options that permit creating a new class of objects, like birds and fishes, and also objects that can change their shape by using pre-calculated parameterized deformation data that do not compromise our real time requirements.

The remind of this paper is organized as follows: Section 2 gives a brief overview of how we model smart objects. Section 3 describes the solutions adopted by our current script language to define object behaviors, along with some examples. Section 4 depicts our approach to concurrently interpret behavior scripts when controlling more than one agent at the same time. Finally, section 5 presents what class of shape deformation algorithms is being considered and section 6 concludes and presents some future work considerations.

2 Modeling Smart Objects

We follow a *Feature Modeling* approach that is a vast topic in the engineering field [4]. The basic idea is to identify and define, during the modeling phase, all interesting features of the object.

In the scope of our applications goals, we define four classes of different *interaction-features*:

- *Intrinsic object properties*, for example: the movement description of parts, a semantic key text description of objects purpose or intent, etc.
- *Interaction information*: like positions of interaction parts (knobs, buttons), hand shapes to use and so on.
- *Object behaviors*, that are available depending on objects state. For example, an automatic door has the behavior *close* available only if there are no agents passing through the door, and the door is open.
- *Expected agent behaviors*: are associated with object behaviors in situations that need to describe: when the

agent should walk, which location it should reach with its hand, etc.

Intrinsic object properties and interaction information are modeled using a graphical user interface. Figure 2a shows a modeling section of a slot machine where the usual locations to get closer and grasp the handle are specified. Figure 2b shows a desk with many interaction-features identified.

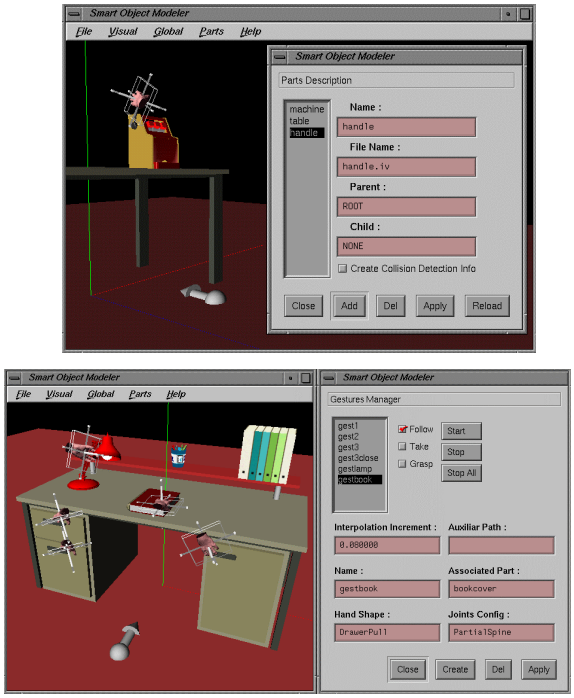

Figure 2a and 2b. Two modeling sessions of smart objects: by using an interactive user interface it is possible to define moving parts of the object, and specific information to control agents as: key positions and hand locations which are important for most interactions.

Once we have modeled all 3D information for an object interaction, we have to describe how and when such data will be used. That means, we need to synchronize object movements with agent movements also taking into account the possibility to have many agents interacting with the same object.

In order to do this, we use a dedicated script language, following a common trend in many commercial products. In the next section we show how this language is designed.

3 Defining Object Behaviors

In order to specify how the object will behave during the simulation phase, we use a script language that consists of keywords grouped into behavior definitions. These keywords are able to trigger movements of the object, give instructions to agents, change state variables and call sub-behaviors.

This approach has shown to be easy and straightforward to describe objects with common behaviors, but when objects become more complex (as the lift of figure 1), we can observe that simplicity is no longer present. To overcome this, we classify the most used behavior definitions in templates that can be used to compose behaviors that are more complex. It is always a challenge to achieve a simple, yet powerful, set of keywords. That's why we find that it is important to keep a format that can be generated directly by a user interface. This is essential if we want designers to program the behaviors of objects and not only their geometry.

Another principle that we have in mind is to be able to provide high level behaviors. The fact is that in our current applications for simulating virtual city environments, the demands are always for complex behaviors. Users rarely want to have only a graspable object, but want to have a complex automatic machine to buy train tickets. And such machine would have only one main behavior to make the agent walk close to the machine, make it put the money inside, press a sequence of buttons and take the ticket.

Listing 1 shows the behaviors definitions used to model a smart table for a party simulation. The table contains many graspable fruits and has two behaviors: the main one takes an agent to the nearest fruit to grasp it and eat it. A second behavior puts back on the table all fruits that were eaten. Figure 3 shows a scene of a simulation done with this table.

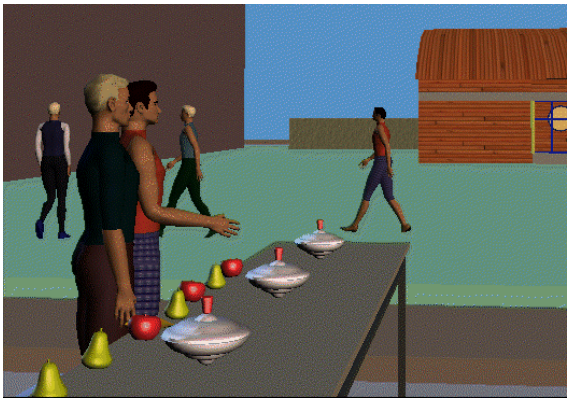

Figure 3. The smart table in a party simulation.

Doing so, a social behavioral simulation of groups of agents [16] in a party can easily integrate the option to have agents going to eat, for example, a fruit on the table.

Listing 1 shows how the two behaviors of this table example are coded. We can observe that a simple script like this can describe complex agent-objects interactions. However, this script has to be carefully designed so that it can correctly be interpreted by many agents at the same time during the simulation. We explain how we deal with such kind of parallel execution in the next section.

```
BEHAVIOR get(n)
  Subroutine
  ChangePos     pos_to_go[n]      pos_very_far
  DecreaseVar   var_num_fruits
  GotoPos       pos_near_fruit[n]
  DoGesture     gest_get_fruit[n] LeftHand
  MoveHand      own_mouth         LeftHand
END

BEHAVIOR get_closest_fruit
  GetNearestPos var_index         pos_to_go[]
  DoBehavior    get(var_index)
END

BEHAVIOR replace_fruits
  ChangeAllPos  pos_to_go[] pos_near_fruits[]
  ShowAllParts  fruits[]
END
```

Listing 1. Behavior definitions for the smart table example.

An important point we have observed is that a script of the kind of listing 1 is already complicated for designers to create it. To try to minimize this difficulty we introduce a kind of behavior template. The idea is to keep a collection of pre-defined common behavior definitions that can be included and connected in a new object from a simple dialog box.

To illustrate this feature we show two examples we have used: A first example is a simple button with the behavior definitions to be pressed. Another example is the behavior definitions of an object that can change the speed of its moving parts.

Listing 2 shows how we can use such templates to define a button behavior. During the modeling phase, the designer can interactively model the shape of the button, its movements, and a desired hand gesture to press it (this specifies a location to reach and a hand shape). Then he or she can select the template *press_button* that will open a dialog box to ask for the needed names to correctly merge the pre-defined behavior inside the list of already modeled behaviors. In this button example, the user is asked to enter four parameters.

The keyword *CheckVar*, that appears in the listing 2 makes the behavior *press_button* available or not, depending on a variable state. This state control is important for when many agents intend to press the same button at the same time: only the first one will do it. After that the behavior will be unavailable avoiding agents pressing buttons that were already pressed. This kind of consistency control is sometimes tricky to program and that's why the usage of such templates facilitates the design process. The next section explains the usage of state variables to synchronize controlled agents.

```
TEMPLATE press_button:
  "Enter the button press state variable :"
   var_button_pressed
  "Enter the hand gesture name :"
   gest_to_press
  "Enter the hand to use (righ/left) :"
   hand
  "Enter the button movement name :"
   cmd_move_button

  BEHAVIOR press_button
    CheckVar   var_button_pressed  false
    SetVar     var_button_pressed  true
    DoGesture  gest_to_press       hand
    DoCommand  cmd_move_button
  END # of behavior

END # of template
```

Listing 2. A behavior template for a button.

Listing 3 shows a template used to model birds and fishes that have the behavior to move its parts continuously to simulate flying and swimming.

```
TEMPLATE fly_movement:
  "Enter the command movement to open  :"
   open_wing_mov
  "Enter the command movement to close :"
   close_wing_mov

  BEHAVIOR fly_movement
    DoAlways
    DoCommand   open_wing_mov
    DoCommand   close_wing_mov
  END # of behavior

  BEHAVIOR fly_change_speed(n)
    ChangeCmdInterpIncrem open_wing_mov   n
    ChangeCmdInterpIncrem close_wing_mov  n
  END # of behavior

END # of template
```

Listing 3. Behavior template definitions used for birds and fishes.

The behaviors shown in listing 3 were made to model a bird with moving wings. The designer can model the geometry of the bird body and wings, and four rotational movements to open and close each wing. The rotational movements to open the wings receive a common name, so that they are executed in parallel when triggered. The

same is done to the rotations used to close the wings. Doing so, when the template is called, the designer has just to say the names of modeled rotations and the behaviors definitions can be correctly merged into the objects design.

Using this same template, it is possible to define similar objects, for example, a fish. For that, the designer only needs to define different rotation movements that are linked to the fishes' fins. And then the same behaviors are correctly merged.

Figure 4a shows a flock of birds in a real time simulation. In this simulation, the application controls the velocity to move the wings and the position and orientation for each bird. Figure 4b shows the modeling phase of the fish that can be inserted in the same type of simulation as it has the same behavioral interface as the bird.

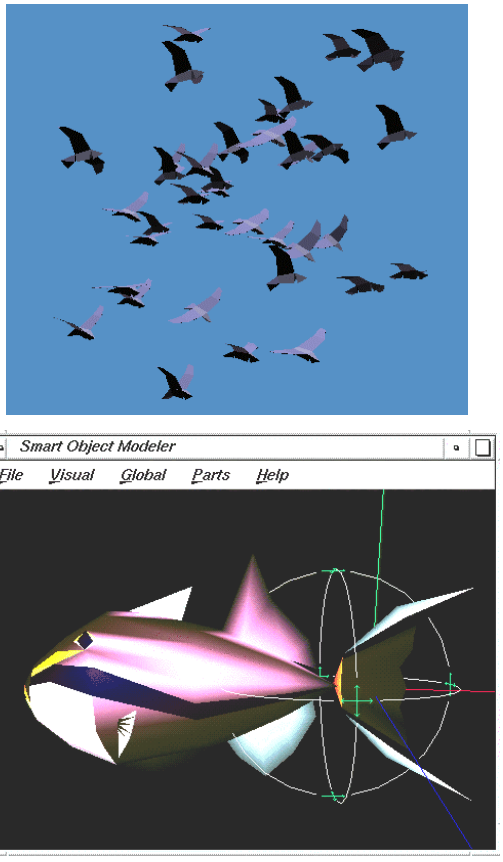

Figure 4a and 4b. The upper figure (4a) shows a simulation of birds that have their wings movement controlled as behaviors. The lower figure (4b) shows the modeling phase of a fish that has the same behavioral control.

4 Concurrently Interpreting Object Behaviors

Regarding agent-object interactions, when we have more than one agent interacting with a single object at a same time, we have to correctly synchronize the interpretation of the behavioral scripts. Moreover, we have to deal with problems like having simultaneous access to more than one resource. Such difficulties are well studied in the areas of concurrent and parallel programming [1].

Let's use as an example the behavior *get_closest_fruit* (listing 1) of our party table (figure 3). Each time the simulation program sends an agent to perform this behavior, a new process is created that will interpret the behavior script, triggering agent actions, and objects commands (that is the shared resource). Figure 5 illustrates this process creation.

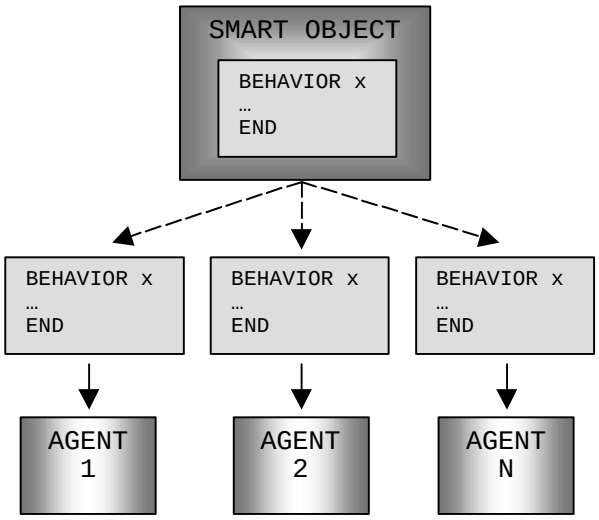

Figure 5. When each agent starts to perform some interaction with an object, a new process (simulated in the same executable) is created to interpret the behavioral script. Synchronization is done by checking state variables global to all processes.

Although we talk here about processes, we simulate such concurrent interpretation of scripts in the same executable program. This is done by calling a *perform* function for each process from an infinite main loop of the application. So, the solutions we describe here are used to correctly decide, for each time step, if the process's *perform* function keeps going interpreting its script or not, allowing other processes to advance their script interpretation.

If we take no special care in interpreting the scripted commands, serious problems arise. Consider that we execute one line of each script being processed, in each time step of the simulation. In this way, at time *t*, an agent a_1 may interpret the command *GetNearestPos* (listing 1). This command will search for the *pos_to_go* that is closer to a_1 current position. At the same time *t*, we can have another process controlling agent a_2 that may also execute the same command and can find that the nearest position to go is the same. If this happens, we will have a_1 and a_2 colliding each other when they arrive in the same position. Worse results will happen when a_2 will try to grasp a fruit that was already taken by a_1.

To avoid such problems, we could use special keywords to specify when a process should stop interpreting its script to leave others processes work. For example, by placing such a keyword after the *ChangePos* instruction, we would force that the process of agent a_1 would execute *ChangePos* before leaving the control to agent a_2. Doing so, when the process of a_2 interprets *GetNearestPos*, it will certainly find a different *pos_to_go*, because the process of a_1 has already invalidated its used position by changing it to *pos_very_far*.

However, a simpler solution was adopted. Each keyword of the script has a fixed property to either stop the execution to allow other processes to run, or to jump directly to the following command of the same script.

In general, when a keyword that triggers some *long action* is found, the process sends the related action to the agent or object and leaves the control to other processes until the action is not completed. These *long actions* are all agent-related actions (walk, arm gesture, etc) and also object movements. All other keywords (check states, call sub-behaviors, compare positions, etc) are executed in sequence, locking the main loop until a *long action* is found again or the script is finished. This criterion has shown to solve all conflicts of our current objects.

This way of synchronization relies mainly on sharing global variable states with all processes. Another example of synchronization is shown in the button example of listing 2. In this case, the global variable *var_button_pressed* is used to avoid an agent pressing a button that can be already pressed by another agent.

Even using all solutions described above, we can still find examples where avoiding a deadlock situation would be difficult. For example, we could not solve all situations during a simulation of the well-known example of simultaneous access of resources that is called the *dining philosophers* [1].

Nevertheless, we still can deal with some complicated examples such as the lift shown in figure 1. In all agent-object interactions we are modeling, we use two main agent-related actions: an inverse kinematics control [3] and a walk motor [5]. For each case, we have a script command to trigger the motion motor.

The correct management of the motion control techniques that can be applied to agents is guaranteed by the *agentlib* environment [7]. The main task of the smart object control module is so to synchronize objects movements with agentlib actions.

All the parameters needed to initialize the motions are specified during the modeling phase of the object. In particular, the *DoGesture* command in the behavior script will activate a main inverse kinematics action (called *reach*) to make the hand move to the pre-defined location. *DoGesture* will also activate two complementary actions: *LookAt* to make the agent look at the reach goal position, and a *HandShape*, that will interpolate the current hand shape of the agent to the pre-defined one (figure 6).

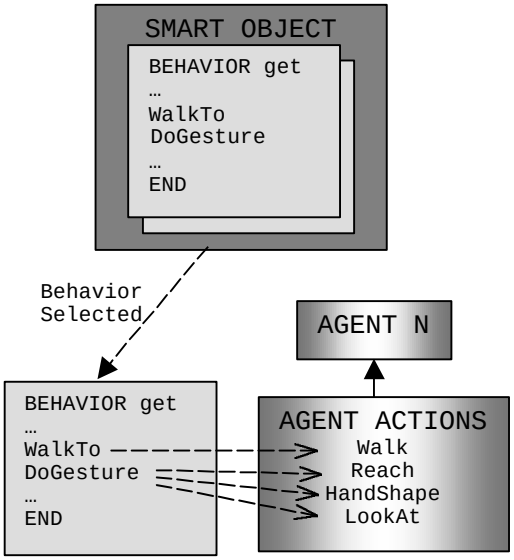

Figure 6. When a behavior script is interpreted, each agent-related command is converted into agentlib [7] actions for a coherent management and blending with other actions, as for example, facial animation expressions controlled by any other module.

5 Behaviors to Change Shape Appearance

In order to enhance the possibilities of a smart object, some shape deformation capabilities are being incorporated. The idea is to have the same behavior interface to trigger also some deformation effects on the object shape.

We do not want to have heavy deformation calculations during a simulation. We are not interested in incorporating, for example, some specific skin deformation module [22] for our fish example (figure 4) to obtain a smooth body motion simulation. Our target applications deal with complex scenes with many objects, and where it is important to keep real time frame rates.

For such reasons, we are interested in deformation algorithms that can be run offline to generate a data file that will describe a parameterized result of the pre-calculated deformation. For example, there is a huge literature about algorithms to calculate smooth changes of level of details (or multi resolution) that can be parameterized. One example is the progressive mesh representation presented by Hoppe [10].

Another deformation algorithm that can be parameterized in most cases is a polyhedral morphing. Commonly, morphing algorithms are divided in solving two phases: the correspondence problem, and the interpolation problem [14]. Once the correspondence is solved, the interpolation can be parameterized between 0 and 1 and evaluated in real time.

Figure 7 shows a morphing result obtained in a previous work [12]. The output of this algorithm is a set of trajectories passing through the models vertices. Then, to obtain the morphing effect, we make each vertex move along the trajectories (together with a gradual global scaling), giving the shape transformation effect.

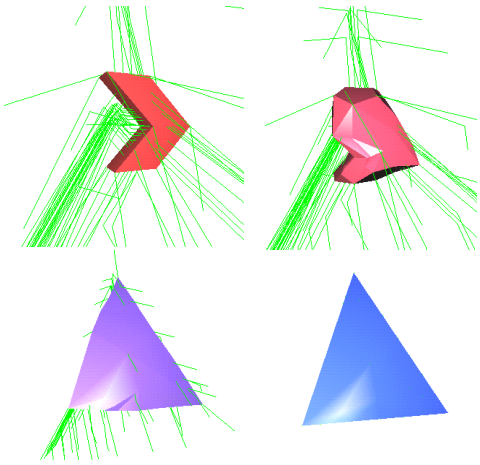

Figure 7. A morphing transformation that can be obtained by moving vertices along pre-calculated trajectories, together with a gradual global scaling.

Any other deformation algorithm that can be parameterized can be easily integrated. As another example, we took some models of muscles that were deformed using a spring-mass system [17]. The muscle deformation is calculated according to changes in the size of the muscle main axis. So that we can get intermediate models, and generate trajectories that passes through corresponding vertices. Doing so, we just need to parameterize the linear piece-wise trajectories to the maximal and minimal axis length changes (figure 8).

These two examples illustrate some possibilities of shape deformation that can be used at a very low computational cost. During the simulation, we only need to perform basic interpolation techniques to get an updated position of the model vertices.

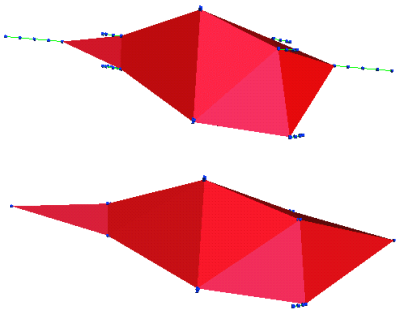

Figure 8. A deformable muscle that changes its shape by moving its vertices along the pre-calculated trajectories. The deformation is parameterized in relation to the length of the muscle main axis.

Connecting such deformations is trivial: once they can be parameterized between 0 and 1, a simple command call in a behavior definition with the desired parameter can trigger a shape change. With this feature, it is easy to connect shape changes to agents movements or other actions.

For example we can think about an object that changes its shape after an agent touches it, and an object that changes its resolution of appearance based on the distance from the camera.

Of course this approach has its limitations. For example, it is possible to “record” the shape deformation of some soft surface after some force is applied to a region of the surface. Then, we can parameterize this and “play” it back whenever the agent touches that region. But this will work only for that region, and the deformation will always be the same. This is similar to having a keyframe animation of the evolution of the surface vertices.

6 Conclusions and Future Work

We showed in this article the smart object approach to model interactive objects for real time simulations. The essential idea is that smart objects contain a series of pre-

programmed behaviors that can control complex agent-object interactions, moving parts, and shape changes. Also, the necessary details to control more than one agent at a same time during an interaction were discussed.

This approach always provides a complete pre-defined plan for each desired interaction with the object, and an easy usage is obtained by means of a graphical interface program to model objects interactions and to program objects behaviors. Because behaviors can become complex, we have adopted a template-based approach to select pre-programmed sub-behaviors, allowing designers to also define the scripted part of the model.

Although this approach seems to take out the autonomous aspect of the agents because they are basically following a sequence of pre-programmed commands, two levels of autonomy can be explored. A first level is not to directly interpret the proposed sequence of commands, but to analyze them and decide with external planning and reasoning algorithms what actions to perform, for example, with the aid of synthetic vision [18]. Another level is to use agents' autonomous capacity to decide which interactions to select and from which objects, leaving only the low-level actions to be described by the pre-programmed plans.

Our experience shows that when the simulation environment grows we have an increasing number of simulation parameters to control. And that is when it is preferred to concentrate on the high-level parameters, leaving the low level interaction information to the smart objects. This situation commonly occurs in virtual cities simulation applications [8] [21].

As future work, we are investigating in two main directions:

To perform simulations with autonomous agents that use their own perception modules to identify which objects (and with which behaviors) they need to interact with in order to achieve some high level given goal.

To extend this smart object conception for interactions with a real human using VR devices, e. g., with a data glove. In this case, the user would only place the hand near pre-defined places in the virtual object to trigger some scripted behavior.

7 Acknowledgments

The authors are grateful to Dr. L. P. Nedel, for providing the data for the deformable muscle example, to S. R. Musse for the simulations involving crowds and smart objects, and to T. Michello for the fish model design. This research was supported by the Swiss National Foundation for Scientific Research and by the Brazilian National Council for Scientific and Technologic Development (CNPq).

8 References

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

[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. Baerlocher, and R. Boulic, "Task Priority Formulations for the Kinematic Control of Highly Redundant Articulated Structures", IEEE IROS'98, Victoria (Canada), 323-329.

[4] 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.

[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] 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.

[9] 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.

[10] H. Hoppe, "Progressive Meshes", Proceedings of SIGGRAPH'96, 1996, 99-108.

[11] 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.

[12] M. Kallmann and A. Oliveira, "Homeomorphisms and Metamorphosis of Polyhedral Models Using Fields of Directions Defined on Triangulations", Journal of the Brazilian Computer Society ISSN 0104-6500 vol. 3 num. 3, april 1997, special issue on Computer Graphics and Image Processing, 52-64.

[13] M. Kallmann and D. Thalmann, "Modeling Objects for Interaction Tasks", EGCAS'98 - 9th Eurographics Workshop on Animation and Simulation, 1998, Lisbon, Portugal.

[14] J. R. Kent, W. E. Carlson and R. E. Parent, "Shape Transformation for Polyhedral Objects", Proceedings of SIGGRAPH'92, 1992, 47-54.

[15] L. Levison, "Connecting Planning and Acting via Object-Specific reasoning", PhD thesis, Dept. of Computer & Information Science, University of Pennsylvania, 1996.

[16] 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.

[17] L. Nedel and D. Thalmann, "Real Time Muscle Deformations Using Mass-Spring Systems", Proceedings of CGI'98, IEEE Computer Society Press, 1998.

[18] 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.

[19] A. Pentland, "Machine Understanding of Human Action", 7th International Forum on Frontier of Telecom Technology, 1995, Tokyo, Japan.

[20] K. Perlin, and A. Goldberg, "Improv: A System for Scripting Interactive Actors in Virtual Worlds", Proceedings of SIGGRAPH'96, 1996, New Orleans, 205-216.

[21] E. Schweiss, S. R. Musse, F. Garat and D. Thalmann, "An Architecture to Guide Crowds using a Rule-Based Behaviour System", Proceedings of Autonomous Agents' 99, Whashington, May, 1999.

[22] X. Tu, and D. Terzopoulos, "Artificial Fishes: Physics, Locomotion, Perception, Behavior", Proceedings of SIGGRAPH'94, July, 1994.

[23] VRML web address: http://www.vrml.org