In Proceedings of the 11th Eurographics Workshop on Animation and Simulation (EGCAS), 2000, Interlaken, Switzerland, August, 2000.

ACE: A Platform for the Real Time Simulation of Virtual Human Agents

Marcelo Kallmann, Jean-Sébastien Monzani, Angela Caicedo and Daniel Thalmann

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

Abstract

This paper describes a system platform for virtual human agents simulations that is able to coherently manage the shared virtual environment. Our "agent common environment" (ACE) provides built-in commands for perception and for acting, while the in-between step of reasoning and behavior computation is defined through an external, extendible, and parameterized collection of behavioral plug-ins. Such plug-ins are of two types: the first type defines agent-object interactivity by using a feature modeling approach, and the second type controls the reasoning and behavior of each agent through scripted modules. Our system is analyzed in this paper and a simulation example integrating some modules with a Lisp behavioral system is shown.

Keywords: Agents, Virtual Humans, Virtual Environments, Behavioral Animation, Object Interaction, Script Languages, Python, Lisp.

1 Introduction

Virtual humans simulations are becoming each time more popular. Nowadays many systems are available to animate virtual humans. Such systems encompass several different domains, as: autonomous agents in virtual environments, human factors analysis, training, education, virtual prototyping, simulation-based design, and entertainment. As an example, an application to train equipment usage using virtual humans is presented by Johnson et al [1].

Among others, the Improv system [2] is mainly controlled by behavioral scripts designed to be easily translated from a given storyboard. Also using scripts, the Motivate system [12] is defined as a hierarchical finite state machine targeting game development.

Game engines are more and more appearing, providing many behavioral tools that can be easily integrated as plug-ins to build games. Although they offer many powerful tools, they may not be well suitable for applications different than games.

In another direction, the Jack software package [3], available from Transom Technologies Inc., is more oriented for human factors applications rather than social and behavior animation. Different systems have been built [1,14], developing their own extensions to the Jack software.

This paper describes a system platform for virtual human agents simulations that unifies the advantages of both scripts and behavioral plug-ins. The system provides the basic agent requirements in a virtual environment: to be able to perceive and to act in a shared, coherent and synchronized way. Our "agent common environment" (ACE) provides tools for the perception of the shared environment, the ability to trigger different motion motors and facial expressions, and provides ways of connection with various behavioral modules.

The central point of ACE is the easy connection of behavioral modules as plug-ins. Such plug-ins can be defined in two ways: specific modules for describing agent-object interactions using the smart object approach [4] and a behavioral library composed of modular Python scripts [5].

Virtual human agents created in ACE have automatically the ability to perform many actions, as walking, using inverse kinematics, looking at some direction, performing facial expressions, etc.

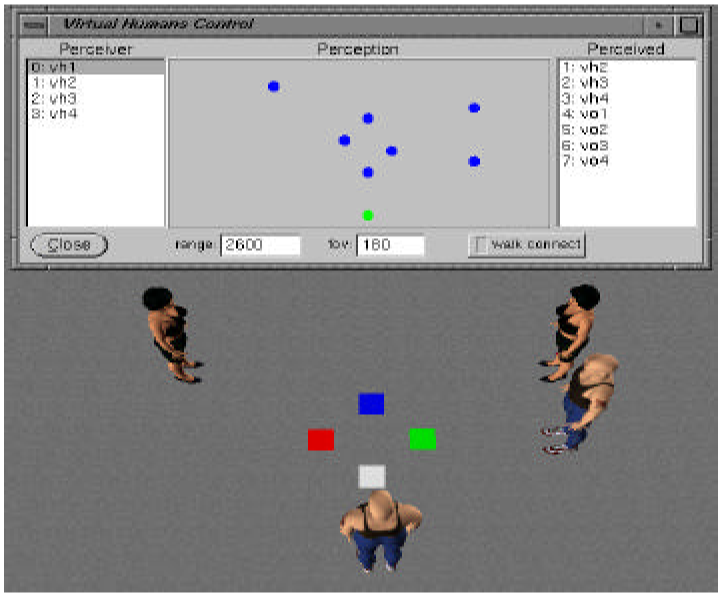
The use of behavioral plug-ins is a current trend [6] that, when well designed, can overcome the difficulty of correctly supplying all the parameters needed to initialize these actions, what can be a strenuous task.

This paper makes an overview of our system, and a simulation example integrating some different behavioral modules is described.

2 ACE System

The core of the ACE system understands a set of commands to control a simulation. Among other features, these commands can:

- Create and place different virtual humans, objects, and *smart objects* (objects with interactivity information) [4].
- Apply a motion motor to a virtual human. Examples of such motion motors are: key-frame animation, inverse kinematics [13], a walking motor [7], facial expressions, etc. These motors can be triggered in parallel and are correctly blended, according to given priorities, by a specific internal module [8].
- Trigger a smart object interaction with a virtual human. Each smart object keeps a list of its available interactions, which depends on the object internal state. Each interaction is described by simple plans that are pre-defined with the use of a specific graphical user interface. These plans describe the correct sequence of motion motors to accomplish an interaction. The GUI is used to interactively define the 3D parameters needed to initialize the motion motors, as positions to put the hand, movements to apply at object parts, etc.
- Query *pipelines of perception* [9] for a given virtual human. Such pipelines can be configured in order to simulate, for example, a synthetic vision. In this case, the perception query will return a list with all objects perceived inside the specified range and field of view. As an example, figure 1 shows a map constructed from the results of the perception information received by an agent.



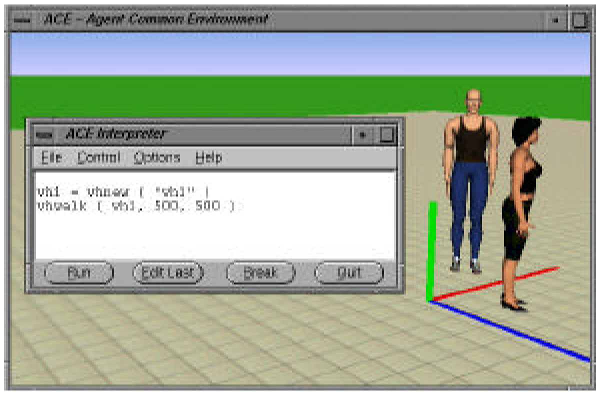
```
0: vh1
1: vh2
2: vh3
3: vh4
```

```
1: vh2
2: vh3
3: vh4
4: vo1
5: vo2
6: vo3
7: vo4
```

range: 2600   fov: 180

Fig. 1. Perception map of the lowest agent in the image. In this example, a range of 2.6 meters and a field of view of 180 is used. The darker points in the map represent the positions of each perceived agents and objects.

All previously described commands are available through simple Python scripts. When ACE starts, two windows appear. One window shows the virtual environment being simulated. The other one contains an interactive Python shell with menus to access the available dialog boxes to control and monitor the ongoing simulation (figure 2).



```
vh1 = vhnew ( "vh1" )
vhwalk ( vh1, 500, 500 )
```

Fig. 2. The ACE system with the graphical output window and the interactive Python shell.

In the interactive Python shell it is possible to load or type scripts to control the simulation. An example of a valid Python script is as simple as the following:

```
# Create a virtual human and a smart object:
bob = vhnew ( “bob”, “sports-man” )
computer = sonew ( “computer”, “linux-cdrom” )

# Query a 3 meters perception with a 170 degress field of view:
perception = vhperceive ( bob, 3000, 170 )

# If the computer was perceived, perform two interactions with it:
if computer in perception :
    sointeract ( computer, bob, “eject_cd” )
    sowait ( computer )
    sointeract ( computer, bob, “push_cd” )
```

Figure 3 shows a snapshot of the animation generated from this script. The created agent is performing the “push_cd” interaction (note that in the image other objects that were previously created are also shown).

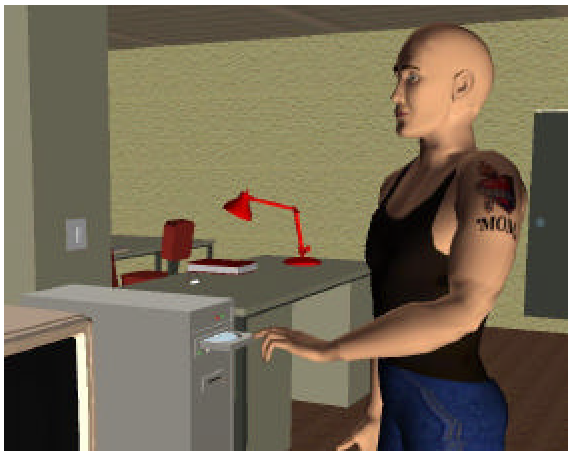


Fig. 3. An agent-object interaction being performed.

The smart object “computer” loaded in this example was defined with a specific modeler where all low-level 3D parameters, object states, needed motion motors, etc were defined (figure 4).

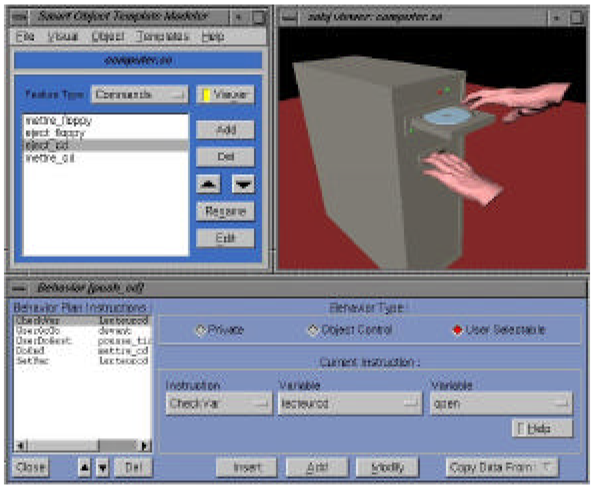


Fig. 4. Modeling phase of the smart object "computer".

In this way, the low-level motion control is performed internally in ACE by following the interaction plans defined inside each smart object description. Python scripts can then easily instruct an agent to interact with a smart object without the need of any additional information. After an interaction, the state of the smart object is updated, and the virtual human agent will wait for another Python order.

Smart objects work as state machines where transitions are interactions. For example, when an object o is in a state s, the list of available interactions to perform with o are defined through the list of transitions starting from s. The use of graphical interfaces to define such state machines is a common approach and many different visual programming techniques have been used targeting a wide range of applications [4,12,15,16]. In our approach, we use text instructions to define the transitions (or the interaction plans), and a graph to describe the connections with the states.

In order to coherently control a multi-agent simulation in ACE, each agent runs in a separate thread, handled by a common *agents controller* module. This module is responsible for transporting messages between the threads by providing a shared area of memory for communication (figure 5).

Usually, each time an agent is created, a new thread starts in order to control it. This is directly implemented in the Python layer. The display update is handled by the controller, which also provides synchronization facilities between threads. Keeping the display update into the controller ensures that no conflicts arise (this could be the case if concurrent processes update the display at a same time).

Concurrent actions (motions or facial expressions) are already handled internally in ACE. However, in some cases it may be interesting to have specific concurrent modules controlling the evolution of specific agent actions. For such cases, new threads can be created within the agent thread, as depicted in figure 3.

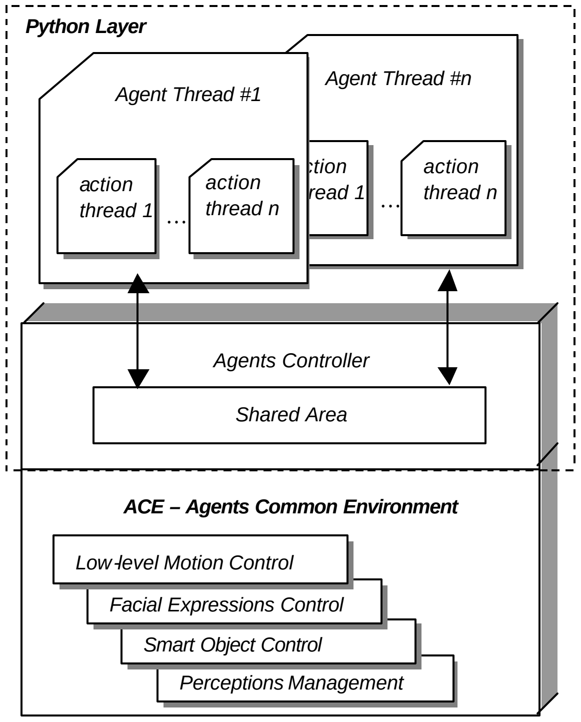


Fig. 5. ACE system architecture.

Inside an agent's thread, the user of the system can ask for agent-object interactions to be performed, and also initialize any motion motor directly. Note that an agent-object interaction may trigger many motion motors sequentially or in parallel, so that all current motions being applied to a virtual human agent need to be correctly blended, generating coherent skeleton joint angles. The blending of motions is done using the AgentLib [8] framework, which is linked inside ACE.

Whenever an object interaction is asked, a special *Object Interaction Thread* (figure 6) is created to monitor the execution of the needed motions until completion. This module is implemented internally in ACE (not in the Python layer) and can be seen as the agent's capability to interpret object interaction instructions; like reading the user's manual of a new object to interact.

In this way, at the Python layer, an object interaction is seen as any other primitive action. Motion blended is supported in all cases, but the user is responsible to coherently start the motions and object interactions. For instance, when an object interaction to push a button with the right hand is requested, the object interaction thread will be active until the hand reaches the button. If, at the same time, another module is controlling the right arm towards a different position, a deadlock may happen.

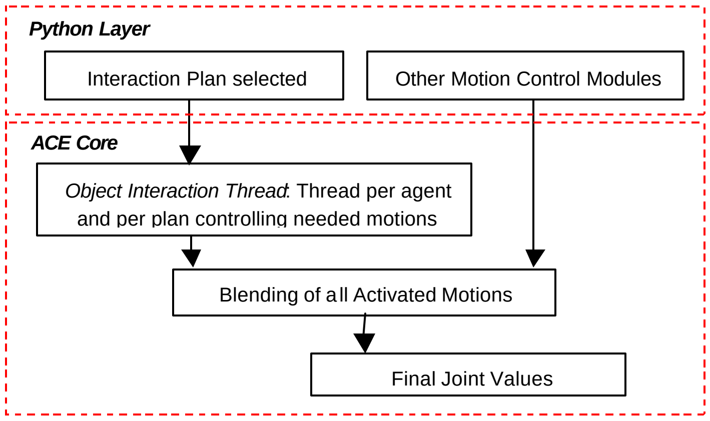

Python Layer
Interaction Plan selected
Other Motion Control Modules
ACE Core
Object Interaction Thread: Thread per agent and per plan controlling needed motions
Blending of all Activated Motions
Final Joint Values

Fig. 6. Motion blending permits other control modules to run in parallel with an object interaction, for example, to control body parts that are not used during the interaction.

Although object interactions are defined with pre-defined plans, a lot of issues still need to be solved during run time. We keep minimal information inside the plans in order to leave to the agent's autonomy space to generate personalized motions during the interactions. For example, for a simple interaction like opening a drawer, the related interaction plan defines a position to stand near the drawer, a position for the end effector (for the right hand) and a suitable hand shape to use. But where to look and if it is needed to bend the knees or not are decisions taken by the agent during run time (figure 7).

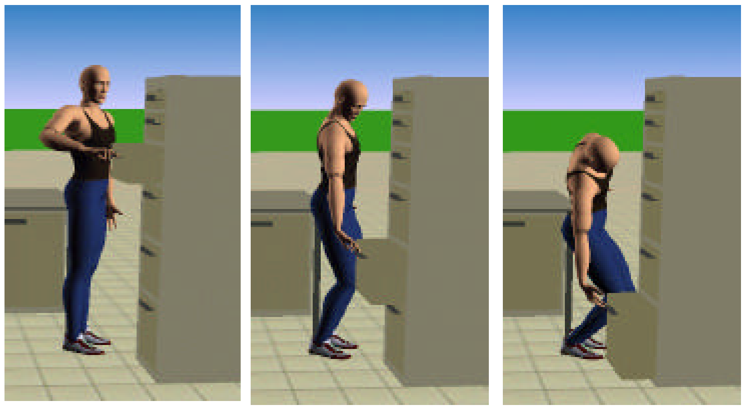


Fig. 7. Agent's autonomy decides where to look and whenever to bend the knees.

3 Using Python Scripts

Python scripts can be organized in modules, which are dynamically loaded from other scripts. Many available modules exist for different purposes, as graphical user interface generation, image processing, mathematical computation, threads creation, TCP/IP connection, etc. The Python interpreter, together with such modules, is available for most computer platforms, including Unix systems and PC Windows. Moreover, if required, new modules can be implemented in Python that might also access methods in C/C++ to achieve better performance.

As shown in the previous section, threads creation is a key issue as we want agents to be able to run their own behavioral modules in an asynchronous environment. The use of such behavioral Python modules is straightforward: the animator chooses one module from a library of pre-programmed modules and runs it inside its agent thread. However, such modules need to be carefully designed in order to avoid conflicts and to guarantee a correct synchronization between them.

The module for TCP/IP connections is used whenever one wants to control the simulation with messages generated from another application. This is the case for the example showed in the following section, where we use a behavioral module written in Lisp sending orders to ACE threads in order to simulate a predefined scenario.

4 A Simulation Example

We have created a virtual computer lab with around 90 smart objects, each one containing up to four simple interactions. When we put some virtual human agents in the environment, we end up with a lot of possible actions combinations to choose.

In this environment, each created agent has internal threads to specifically control its navigation, gestures played as key-frame sequences, smart object interactions, and an *idle state*.

The navigation thread controls the walking motion motor along given collision-free paths. Key-frame animation gestures and object interactions are performed and controlled when it is required. And whenever the agent is detected to stop acting, the idle thread is activated, sending specific key-frames and facial expressions to the agent, simulating a human-like idle state.

The idle state thread is a parameterized behavioral Python module based on some agent emotional states. For example, when the agents anxiety grows, the frequency of small and specific body posture and facial animations (as eye blinking) increases.

We have then translated a simple storyboard into Lisp plans inside *IntelMod* [10], an agent-based behavioral Lisp system. This system communicates with the ACE agent threads by means of a TCP/IP connection, as shown in figure 8.

The scenario is as follows: a woman that has access to the lab comes in a day-off to steal some information. So she enters into the room, turns on the lights, read in a book where is the diskette she would like to steal, then she takes the diskette, turns off the lights and go out of the room. During all the simulation, the woman is nervous about being discovered by someone, and so the idle state module was set to synchronize a lot of head movements and some small specific facial expressions to demonstrate this state.

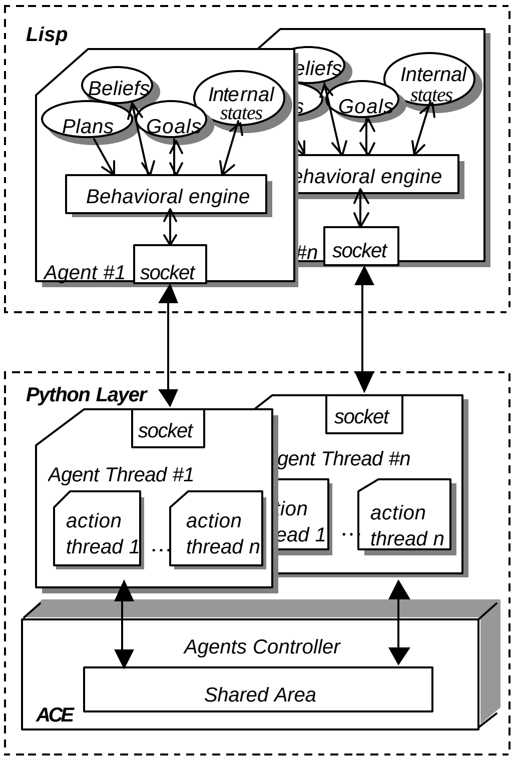


Figure 8. Agent-based behavioral system with ACE.

The behavior of the stealer woman has been modeled in IntelMod. The agent's behavioral engine is in charge of deciding which is the next action to take relying on the agent's current beliefs, internal agent' states and goals. All this information is used to trigger one of the agent's plans available. When a plan is triggered, some post-conditions are reached, some updates in the internal agent' structures are done and finally the corresponding actions are sent through the TCP/IP socket to be performed. Inside ACE, the corresponding agent action thread is activated, and will later send a feedback to the IntelMod's correspondent agent once the action has finished.

An example of a Lisp plan used in this simulation is as follows:

```
(newPlan  'enter-to-place
    '(  (tiredness 90 <)
        (nervosity 30 <) )
    '(  (needs steal info (? company))
        ((? company) has his info (? place))
        (! (is inside (? place)))
    '(  (Act (sointeract door(? place) open))
        (Add (opening (? place) door)))
```

)

The first goal of the stealer is to enter the room where the information resides. Then the action applied by the agent is to open the door and enter. Some important internal states in this case are checked. The agent should not be too tired but it is very nervous and these states are sent to the idle state thread. Some snap shots of this simulation is shown in figure 9.

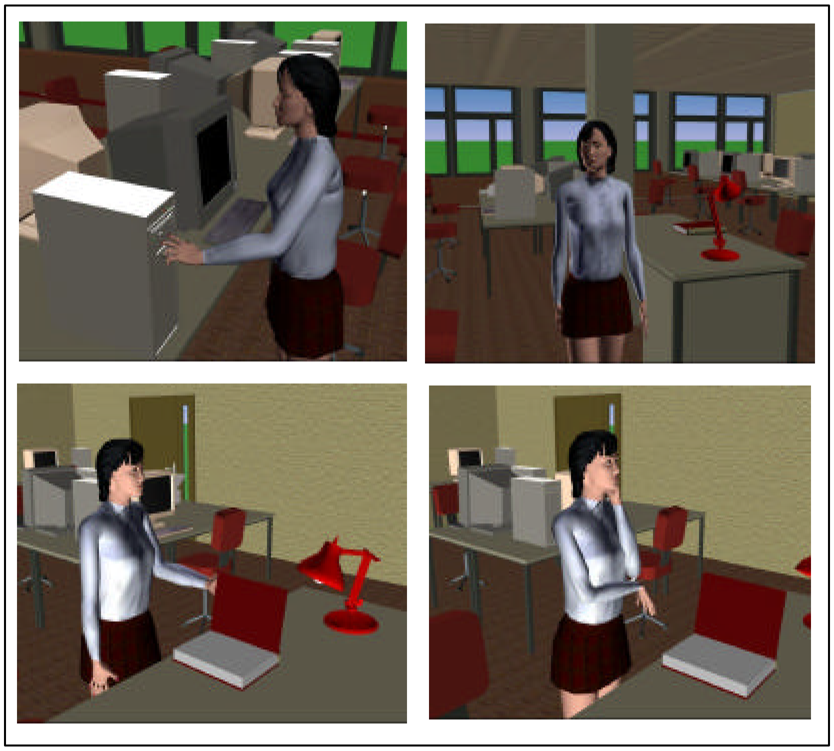


Fig. 9. Some snapshots of the simulation.

5 Conclusions and Final Remarks

We showed in this article how different types of plug-ins can be used in the ACE system, having the Python script language as the main interface layer between the low-level motion animation control, and the high-level behavioral control.

The Python layer can be also seen as the boundary between general-usage animation modules and application-specific modules. For example, ACE has a built-in walking motor but without a navigation control, as navigation requirements can change drastically depending on many issues as: real time interactivity, human-like navigation and exploration, optimized path planning, etc.

The extensibility via Python scripts allows the plug-in of behavior modules, and also of any kind of utilities, as to monitor agents state, or to let the user control and interact with the environment. Actually we make extensive use of many control dialog boxes (written in Python or in C/C++) to inspect agents perceptions, place objects, test agent-object interactions, etc. Another type of user interactivity has been tested through a natural language interpreter, which translates simple English sentences into Python scripts to direct the animation as an interactive shell.

The smart object approach used in the system implies interesting characteristics, as the easy creation of new interactive objects, and the fact that objects' semantics stay distributed within the objects of the scene being accessed through perception queries.

With this architecture, our system has successfully achieved important requirements: extensibility, coherent low level motion control, perception of the environment, and easy creation of agent-object interactions. The user of the system can thus concentrate in the higher-level behavior implementation for the virtual human agents.

The ACE architecture is currently being integrated with the *virtual human director* software [11] developed in our lab in order to merge the capabilities of both systems.

6 Acknowledgments

The authors are grateful to Eric Devantay for the modeling of the virtual lab and its smart objects used to test our system. This research was supported by the Swiss National Foundation for Scientific Research and by the Brazilian National Council for Scientific and Technologic Development (CNPq).

7 References

1. 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.
2. K. Perlin, and A. Goldberg, “Improv: A System for Scripting Interactive Actors in Virtual Worlds”, Proceedings of SIGGRAPH’96, 1996, New Orleans, 205-216.
3. N. Badler, R. Bindiganavale, J. Bourne, J. Allbeck, J. Shi, and M. Palmer, “Real Time Virtual Humans”, International Conference on Digital Media Futures, Bradford, UK, April, 1999.
4. M. Kallmann and D. Thalmann, “A Behavioral Interface to Simulate Agent-Object Interactions in Real-Time”, Proceedings of Computer Animation 99, IEEE Computer Society Press, 1999, Geneva, 138-146.
5. M. Lutz, “Programming Python”, Sebastapol, O’Reilly, 1996.
6. N. Badler. "Animation 2000++", IEEE Computer Graphics and Applications, January/February 2000, 28-29.

7. 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.

8. 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.

9. C. Bordeux, R. Boulic, and D. Thalmann, “An Efficient and Flexible Perception Pipeline for Autonomous Agents”, Proceedings of Eurographics '99, Milano, Italy, 23-30.

10. A. Caicedo, and D. Thalmann, “Intelligent Decision making for Virtual Humanoids”, Workshop of Artificial Life Integration in Virtual Environments, 5th European Conference on Artificial Life, Lausanne, Switzerland, September 1999, 13-17.

11. G. Sannier, S. Balcisoy, N. Magnenat-Thalmann, and D. Thalmann, “VHD: A System for Directing Real-Time Virtual Actors”, The Visual Computer, Springer, Vol.15, No 7/8, 1999, 320-329.

12. Motivate product information, Motion Factory web address: http://www.motion-factory.com.

13. P. Baerlocher, and R. Boulic, “Task Priority Formulations for the Kinematic Control of Highly Redundant Articulated Structures”, IEEE IROS’98, Victoria, Canada, 1998, 323-329.

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

15. A. Scholer, R. Angros, J. Rickel, and W. L. Johnson, “Teaching Animated Agents in Virtual Worlds”, Proceedings of Smart Graphics, March 20-22, Stanford, USA, 2000.

16. C. Barnes, “Visual Programming Agents for Virtual Environments”, Proceedings of Smart Graphics, March 20-22, Stanford, USA, 2000.