A Task Definition Language for Virtual Agents

Spyros Vosinakis
Department of Informatics
University of Piraeus
80 Karaoli & Dimitriou str.
Greece, 18534, Piraeus
spyrosv@unipi.gr

Themis Panayiotopoulos
Department of Informatics
University of Piraeus
80 Karaoli & Dimitriou str.
Greece, 18534, Piraeus
themisp@unipi.gr

ABSTRACT

The use of Virtual Environments as a user interface can be important for certain types of applications, especially in the fields of education and entertainment. These synthetic worlds are even more attractive for the user when they exhibit dynamic characteristics and are populated by virtual agents. There is, however, a lack of general-purpose tools for designing and implementing intelligent virtual environments, and especially in the case of defining virtual agents' tasks, where there is a strong dependence between the task execution and the context. In this paper, we present our approach towards a context-independent definition of tasks using a high-level language. With the proposed task definition language, one can combine numerous built-in functions and commands to describe complex tasks as a combination of parallel, sequential and conditional execution of actions. It can be used to program complicated virtual agent interactions with the environment without going into much detail on how these tasks are implemented and how parallelism is achieved. The main advantage of the proposed language is that it enables tasks to be easily constructed and reused by different agents and in different environments. Our approach has been based on SimHuman, a platform for rendering and animating Virtual Agents in real-time.

Keywords

virtual reality, virtual agents, virtual environments, intelligent agents, task definition, animation

1. INTRODUCTION

The enormous growth of processing power that we have witnessed in the last few years, as well as the powerful features of modern graphics cards, have enabled the development of complex three-dimensional environments that no longer need an expensive workstation or a super computer to run. Nowadays, even everyday users can have a limited virtual reality experience on their personal computers by navigating and interacting with beautiful synthetic worlds. The use of Virtual Environments as a user interface should, nevertheless, not be limited to just browsing a beautiful 3D scene and interacting with passive objects or other users. However attractive a synthetic world may be, if there is little or no autonomy at all, the immersive experience is of limited interest. Virtual worlds become more interesting if there is some user-independent action in the environment, which can be achieved with the use of virtual agents.

A virtual agent can be defined as an autonomous entity in a virtual environment. It should not only look like, but also behave as a living organism (human [Bad93], animal [Ter94], or other fictional character [Ayl99]) and be able to interact with the world and its inhabitants. To enhance a virtual agent's autonomy and add more believable characteristics, one should model the agent's functionality and behavior so as to resemble the real behavior of the creature it represents. This means that the ideal virtual human in a synthetic environment is the one that seems to behave as a real human would; the one whose actions seem to have purpose and meaning. Virtual Environments can, therefore, benefit from the advances in Artificial Intelligence, and especially in the fields of Intelligent Agents, Robotics and Artificial Life, to increase their autonomy and become more believable and interesting for the users.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Journal of WSCG, Vol.11, No.1., ISSN 1213-6972
WSCG'2003, February 3-7, 2003, Plzen, Czech Republic.
Copyright UNION Agency – Science Press

There has been significant research in the field known as Intelligent Virtual Environments [Luc00, Ayl01] or Intelligent Virtual Agents, concerning appearance [Mag96, Bre00, Aub00], motion [Tha96, Bro98, Kar98] and behavior [Per96, Sil99, Bur01], and even an integrated system has been proposed, the Agent Common Environment [Kal02b]. There is, however, a lack of standard architectures, methodologies and general-purpose tools, whilst each approach seems to be using a different configuration of a Virtual Environment as a basis. It seems that Virtual Environments can have different levels of detail and complexity concerning physical laws of the world, agent visualization and animation, and agent – object interaction. Therefore, it is hard if not impossible, to design and build tools that simultaneously satisfy all cases.

In the existing approaches one may also notice that there is a strong dependence between the action definition and the context, i.e. actions depend on both the agent's structure and the current environment settings. In most of the cases, these actions are described using specific object names and numerical data, which makes it hard to reuse them in different scenarios or to be performed by different agents. Furthermore, one may notice the absence of a tool to describe action combinations and sequences needed to achieve specific tasks. In most of the cases, actions define simple agent-object or agent-agent interactions, and complex tasks are executed as a result of higher level decision making, e.g. with the use of a planner. We believe, nevertheless, that most everyday tasks are a result of action combinations and decisions, which can be described with the use of a procedural language. Tasks could then be instantly executed avoiding the computational cost of a planner. The latter can be restricted to higher-level decisions, e.g. for switching between tasks.

In this paper we present our approach towards a context-independent definition of tasks for virtual agents, using a high-level language that enables them to be reused by different agents and in different environments. We discuss about the syntax and functionality of the implemented language and present detailed examples of task definitions using everyday scenarios. We also review the related work concerning programmable agent behaviors.

2. A 3D ENVIRONMENT WITH VIRTUAL AGENTS

In this section we will discuss about the structure and functionality of the virtual environment that our research is based on, since it will aid to the better understanding of how the proposed task definition language is working and how it can be used to describe the agents' interaction with the virtual world.

The Task Definition Language is based on SimHuman [Vos01], a tool for the creation of 3D environments with virtual agents. SimHuman consists of a programming library and two utilities for designing the environment and the agents' animation sequences. The library allows users to define and animate three-dimensional scenes with an arbitrary number of objects, virtual agents and user-controlled avatars, and has embedded characteristics such as Inverse Kinematics, Physically Based Modelling, Collision Detection and Response, and Vision. The geometry models for the virtual agents can be imported from Curiouslabs Poser, a commercial program for modeling, posturing and animating synthetic characters.

The main object in SimHuman is the world, which represents the virtual environment itself. It contains the set of all entities that exist in the environment, which are either objects or agents. The difference between an object and an agent is that the latter can perform actions and perceive the current state of the world and, therefore, it can have an autonomous operation. Agents are not necessarily life-like; objects with reactive behavior (a computer, a car, etc.) could also be declared as agents.

Entities can be arranged in a tree-structured hierarchy, where the geometric transformation of an entity is affected by that of its parent, and affects that of the children. All entities have a geometry, some common geometric properties (translation, rotation, size, etc) and a set of attributes, which are user-defined variables.

An attribute's use depends on the environment and the respective laws, e.g. it could hold the current speed of an object, percentage of liquid in a bottle, the state of a button, etc. In the case of human-like agents, attributes can be used to model emotions, biological properties, mental states, etc. The world has also a set of attributes that model the global properties and can be perceived by all agents, e.g. the time of the day, the weather, etc. Each attribute is of the form **<name, type, value>**, where *name* is its unique name, *type* is the type of data that it holds and *value* its current value.

3. THE TASK DEFINITION LANGUAGE

Our Task Definition Language's purpose is to fill the gap between higher-level decision processes and the agent's motion and interaction in the environment. Using this language, one can combine numerous built-in functions and commands to describe complex

tasks as a combination of parallel, sequential and conditional execution of actions. Its main advantage is that it makes it easier for the user to specify action combinations and scenarios for virtual agents, without having to explicitly program all implementation details. Furthermore, it is a way to define these tasks in a context-independent manner and, thus, to easily reuse them with different agents and/or in different environments.

The Task Definition Language consists of a number of user-defined tasks, each of which has a unique name and a number of arguments. As a task we define the combination of virtual agent actions that achieves a specific result, e.g. having lunch, going to bed, driving to work, etc.

Actions

An action is a process that causes changes to the world and its entities. It has a duration, either fixed or variant, and is executed in continuous timeframes. We call *primitive actions* the set of commands that an agent can perform in one timeframe. In SimHuman these actions are: changing the geometric properties of entities (rotation, translation, place in the hierarchy), adding new entities or removing entities from the world, and sending messages to other agents. It is, of course, the world's responsibility to apply these actions and, through its laws, to determine their effects on the entities. Note that an agent can execute more than one primitive actions in one timeframe, e.g. rotate both arm and leg, but these will be visualized simultaneously.

An action is implemented as a sequence of primitive action sets that are executed in continuous timeframes. This sequence can be:

- **predefined** : a_1, a_2, … a_n , with duration = $n*\Delta t$, where a_i is a set of primitive actions performed in the timeframe i and Δt is the duration of a timeframe
- **goal-oriented** : $a_i = f(a_{i-1},I,G)$, where a_i is the set of primitive actions performed in the timeframe i, I is the set information about the objects and their properties that the agent receives from its senses, G is the goal, and f is a function that returns the next primitive action set based on the agent's current action and senses. The sequence terminates when the agent's senses indicate that the goal has been reached.

The Task Definition Language supports a number of standard actions, both predefined and goal-oriented, which can be performed by the agent. Of course, all available primitive actions can be solely called as well. Supported actions have a unique name and a number of arguments.

Probably the most important feature an animated agent should have is keyframing, i.e. the ability to execute predefined animation sequences. We have developed a utility to design, test and store keyframed animations in a visual environment, by selecting the agent's body parts and adjusting their rotation. The set of the user-defined animation sequences forms an *animation library*, which is stored in a file and is part of the agent's configuration. During runtime, an agent can execute any of the existing animations with the action: **anim <name>**, where *name* is a string containing the unique name of the animation sequence. Before executing the animation, the agent performs a transition with constant speed from its existing pose to the initial pose of the animation, and then starts executing the animation sequence. Global rotation and translation changes can also be included in an animation, but they are treated as relative values, e.g. the agent moves one meter forward and turns 30 degrees to the left.

Another important feature of virtual agents is locomotion, i.e. the ability to walk inside the environment. The Task Definition Language provides a number of actions for the agent's locomotion. The agent is able to walk up to a given position, to rotate its body until it reaches a certain orientation, to walk to a certain direction, or to follow a path. Positions and orientations are given as vectors and a path is a list of vectors. The locomotion engine is not using standard limb rotations, because we wanted to support any type of agent and not just human-like ones. Therefore, the engine works using a state machine that switches between several user-defined animation sequences. The use of the state machine is to ensure that the movement and rotation of the body takes place in a correct and believable manner, e.g. in the case of virtual humans, the agent starts rotating only when one of its legs is on the ground, and it rotates around that particular leg.

The interaction between agents and objects cannot always be based on predefined animations, because there might be cases, where the agent may have to rotate its limbs to reach a certain point in space. To solve such problems, the virtual agent has to use a form of inverse kinematics. In our approach, it can perform the action: **ik <chain> <position>**, where *chain* is a kinematic chain of entities and *position* is a vector containing the target position. To make the agent-object interaction easier, the user is able to define a number of spots (positions in space) in the object configuration, and assign an object's spot as a target position in inverse kinematics actions. This could assist the agent on how to catch or how to use an object. After catching an object, one can always add it to the agent's hierarchy using the respective

primitive action, so that the object remains attached to the agent's hand.

The inverse kinematics action animates the chain so that the last joint's position is equal to the target. Instead of using a generic inverse kinematics solver, we are using an approach, which tests at every step the best rotation for each joint to achieve the target. This continuous correction sequence has the advantage that the animation looks more natural and human-like compared to applying a transition from the current state of the chain to the solution of the problem. Additionally, it works with moving targets and can avoid collisions with objects that lie between the agent and the target.

Instead of calling a single action, it is also possible to call a full task using the proper arguments. This can result to higher-level tasks that combine several subtasks to achieve a goal. Finally, there are actions that have no effect on the environment, such as those intended to copy values between variables and to manipulate the agent's knowledge base by adding or removing beliefs.

Literals, Variables and Functions

When an action is called, it should be followed by a number of arguments, each of which should be of a certain type according to the action definition. Arguments can be literals that provide straightforward values, variables, or even functions. The user should, of course, ensure that the variable's type or the function's return type is the appropriate. The available types are: boolean, integer, float, string, entity, list of entities, vector, list of vectors and relation. A relation is a composite type, which uses a string as a name and a list of values of any type, e.g. person('John', 28, 1.80, 'single'). Relations are useful for message exchange and knowledge representation.

There are four different variable sets that can be used. These are the agent's attributes, the task arguments, the task's internal variables and other entities' attributes. To refer to a member of one of the first three sets, one can simply use its name. On the other hand, to refer to an attribute of another entity, one should use the notation **[<entity name>]<variable name>**.

A very important feature of the task definition language is the ability to use functions as arguments, because it allows actions to be called with values that are adapted to different environments. In that way, one can define tasks that may be executed even in highly dynamic worlds, because with the use of variables and functions, the agent can track the current state of the world and possible relations between entities, and use this knowledge to apply the proper actions. Functions use arguments themselves, and, therefore, they can be nested, producing even more useful combinations, as we shall see in the examples.

There are a number of functions that can detect spatial relations between entities, a feature that is important both for conditional execution of actions, and for managing the agent's beliefs about the world. These functions use two entities as arguments and can be used for the following relations: *near*, *on*, *front_of*, *behind*, *left_of*, *right_of*, *above* and *below*. They are evaluated using the current geometric properties of entities (position, size, orientation) and return a boolean value, e.g. above(e1, e2) returns true if the entity named 'e1' is above the entity named 'e2'. The relations front_of, behind, left_of and right_of are relative to the second entity's local coordinate system. All entities have user-defined front and up vectors, which are used in this case for detecting if such relations are true. Another important function is *intersect(entity1, entity2)*, which checks if two entities collide with each other using the environment's built in collision detection engine.

There are also functions that can generate a new position relative to a given entity or a given position. This relative position can be one of the following: *left*, *right*, *front*, *behind*, *above*, *below* and *on*. In the case of a given entity, the appropriate function uses the entity's own coordinate system, while in the other case, it uses the agent's one. There are of course infinite vectors that satisfy such relations, e.g. there are infinite places above an entity, but the language is using a user-defined default distance value to create the appropriate return vector. Finally, one can use functions that return the distance between two entities or the middle position between them.

Besides the functions that deal with spatial properties, there are also logical ones, such as *and*, *or* and *not*, which are very useful for defining complex conditions. Another important function is *exists(ent, f)*, where *ent* is a variable of type entity, and *f* a function (or a combination of functions) with return type boolean that uses *ent* in its definition. The function returns true if there is at least one entity in the environment for which *f* returns true, and in such a case the variable *ent*'s value is the first such entity found. For example, if there is a variable *e* declared as entity and the agent's structure contains a sub-entity called 'Hand', the function *exists(e, intersects(e, Hand))* returns true if there is an entity that intersects with the agent's hand, and the value of *e* is that entity.

Finally, the task definition language provides functions for all basic arithmetic operations and

relations between floats, integers and vectors, as well as a number of utilities for type conversions and list iterations. At the moment there are 85 predefined functions already implemented.

Defining Tasks

A task consists of three parts: the task definition, the variable declaration and the body. The syntax is:

```
<task definition>
#Variables
<variable declaration>
#Body
<block of commands>
#end
```

The task definition is as follows: TASK name($type_1$ arg_1, $type_2$ arg_2, ..., $type_n$ arg_n), where name is the task's name and $type_i$ and arg_i are the type and name of the i-th argument respectively. In the variable declaration, the user defines the local variables that will be used by the task by writing their type, name and initial value. Finally, the body contains a block of task commands, which is declared in the form c_1; c_2; ... c_n , where c_1, c_2, ..., c_n are task commands. Possible task commands are:

- **<action>** : a single action
- **PAR(<block b1>, <block b2>)** : Blocks b1 and b2 are executed in parallel
- **DO(<block b>) UNTIL c** : Block b is executed until condition c is true
- **IF <bool c> THEN (<block b1>) ELSE (<block b2>)** : If condition c is true, block b1 is executed, else block b2 is executed

The simplest body declaration is to have a series of actions separated with semicolon. The actions are then executed sequentially and the task execution terminates when the last action has finished. In the case of an action failure, e.g. because the inverse kinematics cannot be solved for the given chain, or because an unknown animation is requested, the whole task fails.

Parallel execution of actions is supported, and it can be achieved using the PAR command. The user defines two blocks that are executed in parallel until they are both successfully terminated. The action parallelism works with resource allocation and de-allocation. Each of the possible agent actions animates a number of limbs, some of which may be critical for the action's success, while others may not. When an action starts executing, it allocates the critical body parts, and no other action is allowed to use them, until it is over and de-allocates them. Other actions, may, however take control of the non-critical parts of that particular action. For example, a walking animation allocates the legs, feet and global translation and rotation, but it does not allocate the arms, although it animates them. It is then possible for an agent to walk and scratch its head in parallel, but it is not possible to kick something while walking. In the second case, where an action tries to allocate a resource that has already been allocated, the PAR command fails.

There are cases, where one may wish the agent to repeat executing the same block of actions until a certain condition is met. This can be achieved with the DO – UNTIL command. The difference between this command and similar ones in classic procedural programming languages is that the termination condition value is not only checked at the end of the block, but also during its execution. This means that one can instruct an agent to execute a number of actions to reach a goal, but if that goal condition is reached earlier, the rest of the actions will be skipped. The frequency of the condition checking during the command execution is user-defined.

Finally, the Task Definition Language provides an IF-THEN-ELSE statement to switch between different blocks of commands if needed, e.g. the task *sit(entity e)* may use different commands according to the type of e (couch, chair, stool, etc).

4. EXAMPLES

We will present a number of examples to show how the Task Definition Language can be used to achieve certain agent – object interactions, or even to describe more complex agent behaviors.

Interacting with Objects

The first example is an agent that uses its hand to catch an object. This action can be performed in various ways, according to the desired level of detail and complexity, both of the agent and the environment.

The simplest way of doing it is without grasping. Supposing that the agent's fingers do not move, one can define an end-effector on the surface of the hand as a dummy joint, and a spot on the surface of the object. One can then use the inverse kinematics action with a joint chain that includes the shoulder, the elbow, the wrist and the defined end-effector, and use as target position the spot on the object. If the action succeeds, the next one will be to assign the object as the end-effector's child, so that it remains attached to the agent's hand. This approach may not be the most elegant one, but it is suitable for large environments that may need simplified agent models. Furthermore, if the spots on the objects have been

carefully chosen, it can achieve pretty good visual results.

A more elegant way of having an agent catch an object is to use its fingers and have them grasp the object. One approach is to define as many spots on the surface of the object as the fingers that will be used. An inverse kinematics action can then bring the hand at the desired position and after that, a number of parallel IK motions using the PAR command will move the fingers concurrently towards the defined spots. Another approach is to constantly rotate all fingers towards the surface of the object in parallel, and stop rotating each one when it collides with the object (using the DO - UNTIL command). Both approaches need complex agent models and skeletons, but achieve much better results compared to the previous one. The second case avoids the definition of spots, so it is more general, but it has to make use of constant collision detection checks, which will increase the computational cost.

Finally, one can use the Task Definition Language to build more complex interactions, such as to use one object with another. For example, an agent could catch a coin and use it on the slot of a machine, just by using the coin itself as an end-effector of the inverse kinematics chain (since it has been added to the agent's structure after its catching) and use the slot's center position as a target.

Observing the Environment

There may be cases, where the agent may not initially know which object to interact with. In such cases, the agent may have to observe the state of the world and select the appropriate objects before performing a task. Consider the example where an agent walks into a restaurant and checks for a free table. Let us assume that there are no reserved tables, so a table is considered free if all chairs around it are not occupied. In such a case, the agent can use a combination of conditions to examine if there is a free table in the world. First of all, it could examine if a chair around a table is occupied by checking if there is an entity Chair of class 'chair', which is near the table, and there is an entity of class 'Human' that intersects with it, so it is probably sitting on it. Using the above condition, to check if a table is free, one has to check if there exists an entity Table of class 'table', around which no chair is taken. The syntax of such a task will be:

```
TASK sit_on_free_table()
#Variables
   entity Table ''
   entity Chair ''
   entity Human ''
#Body
DO (
   task walk_around()
) UNTIL exists( Table, and(
   eq( [Table]class, 'table' ),
   not ( exists ( Chair, and(
      eq ( [Chair]class, 'chair' ),
      and(
         near ( Chair, Table ),
         exists( Human, and(
           eq( [Human]class, 'human' ),
          intersect( Human, Chair ) ) )
      ) ) )
   )
   )
);
task go_and_sit(Table)
#end
```

In the above code, we suppose that there are two additional tasks: *walk_around()*, which lets the agent follow a predefined path and *go_and_sit(Entity e)*, where the agent walks to the table *e* and sits on one of its chairs.

With this approach, the agent can directly draw conclusions using only the partial information it receives by its senses. This fact makes even more sense in highly dynamic environments, where changes in the world are so rapid that it is almost impossible to keep track of every one of them. Consider for example a soccer game played with agents. It is not effective to constantly add and remove global beliefs concerning which players are near the ball and who has taken control of it. Using the task definition language, one can easily detect if an opponent has the ball and who that person is, using a simple condition like:

```
exists ( Human, and (
   and(
      eq ( [Human]team, 'opponent'),
      and( near ( Human, Ball ),
         front_of ( Ball, Human ) )
      ),
      not( exists ( Other, and (
         eq( [Other]team, 'our' ),
         near ( Other, Ball ) ) ) )
   )
)
```

which means that there is a person from the opponent team, who is near the ball, the ball is in front of him, and no person from the agent's team is near the ball.

A Complete Scenario

From the above mentioned examples it becomes pretty clear that using the language to construct some basic, parameterized tasks and combining them in higher level tasks, one can easily build agent behaviors for complicated scenarios, avoiding the computational cost of using more complex decision techniques, such as planning.

Figure 1: The agent is walking in the bar

We have built an example of an agent visiting a bar (Figure 1) by defining several subtasks and a global task that controls them. The subtasks are: walking around until there is a free table, sitting on a chair, calling the waiter, ordering drink and drinking from a bottle. The communication between the agent and the waiter takes place through predefined messages and information is exchanged between subtasks (e.g. which table is currently free) with the user-defined agent attributes. The drinking task also uses messages. There is an attribute in the bottle configuration that holds the liquid percentage, which is decreased every time the agent brings the bottle to its mouth. To do so, the last action in the drinking task is sending a message to the world that the agent drinks from the bottle, and the world is responsible for subtracting the proper value from the bottle's liquid percentage.

5. RELATED WORK

The task execution of agents in intelligent virtual environments is an important issue and there are a number of approaches in the literature. In most of them the agent-environment interaction is context-dependent and hardwired, and there is little chance of programming new tasks and adjusting them to different scenarios. There are, however, three important approaches that let the user define new agent interactions in a programmable way: the Parameterized Action Representation [Bad00], the Smart Objects approach [Kal02a] and the Improv system [Per96].

The Parameterized Action Representation is a language designed to bridge the gap between natural language instructions and the agents who are to carry them out. One can use it to give a complete description of an action by specifying the applicability conditions, the execution steps and the termination conditions. Actions can be chained together using the preparatory specifications, a set of <condition, action> statements.

The Parameterized Action Representation is a successful way to design complex action sequences and thus to define agent behavior. It seems, nevertheless, that condition checking is restricted to variable values and symbolic relations, whilst in our approach one can use spatial relations and inter-object collisions for action termination and / or initialization. Furthermore, the ability to use and manipulate variables and to call tasks from within other tasks in our language, makes it easier to define higher-level tasks as combinations of more primitive ones using separated code, and to adjust them without having to readjust the preparatory specifications of all subtasks.

In the smart objects approach, all interaction features of an object are included within the object description. Besides the intrinsic object properties (movement description, physical properties, etc.), smart objects include information to aid actors to perform each possible interaction, description of the object's reaction for each performed interaction, and expected actor behavior in order to accomplish the interaction.

Smart objects are very important for the reusability of designed objects and the decentralization of the animation control, but they have the drawback that all agents interact with an object in the same way. Furthermore, it restricts the behavioral capabilities of agents to single agent-object interactions, while a complex task may involve more than one objects and agents. It is also specifically designed for human-like agents, which could be a drawback in some cases.

Improv consists of an Animation Engine that uses procedural techniques to generate layered, continuous motion and transition between them, and a Behavior Engine that is based on rules governing how actors communicate and make decisions. The combined system provides an integrated set of tools for authoring the 'minds' and 'bodies' of interactive actors. Improv seems, however, to offer little chance for action reusability, since all animation scripts are specified in a low-level manner by explicitly stating the translation and rotation values.

6. CONCLUSIONS AND FUTURE WORK

The proposed Task Definition Language can be used to program complex actions for virtual agents without going into detail on how these actions are implemented and how parallelism is achieved. One can easily program the agent to dynamically check for environmental changes using the built-in functions, and to adapt its behavior accordingly. One has also the freedom to define agent – object and agent – agent interactions at both symbolic and

physical levels through messages and action execution, and update the agent's beliefs to maintain a coherent world representation. These features make it easier to define complex agent behaviors in complicated worlds, and to reuse tasks in different environments and with different agents.

There are, nevertheless, some issues that still need improvement, such as the addition of more complex object interactions (e.g. buttons, handles, etc), as well as the use of facial animation for expressing the agents' emotions and for synchronizing their lips when speaking. While building the Task Definition Language, we tried to make it as simple and open as possible, and it is, therefore, easy to add support for additional actions, once our agents are enhanced with new capabilities.

We are working towards a generic platform for designing and running real-time virtual environments with virtual agents, and we are planning to use the task definition language as a medium to connect a spatio-temporal planner with virtual agents. Furthermore, we are trying to improve both the action execution and the world functionality, so that in the future we will be able to add more bio-mechanical characteristics to the agents, and test the task definition language in real world simulation environments.

7. REFERENCES

[Aub00] Aubel, A., Boulic, R., and Thalmann, D. Real-time Display of Virtual Humans: Level of Details and Impostors. IEEE Trans. Circuits and Systems for Video Technology, Special Issue on 3D Video Technology, 2000.

[Ayl99] Aylett, R., Horrobin, A., O'Hare, J., Osman, A., and Polshaw, M. Virtual Telebubbies: reapplying a robot architecture to virtual agents. Proc. of the Third International Conference on Autonomous Agents, New York, pp. 514-515, 1999.

[Ayl01] Aylett, R., and Cavazza, M. Intelligent Virtual Environments, - A State-of-the-art Report. Eurographics 2001, pp. 87-109, 2001.

[Bad93] Badler, N., Phillips, C., and Webber, B. Simulating Humans: Computer Graphics Animation and Control. Oxford University Press, 1993.

[Bad00] Badler, N., Bindiganavale, R., Bourne, J., Palmer, M., Shi, J., and Schuler, W. A parameterized action representation for virtual human agents. Embodied Conversational Agents, MIT Press, pp 256-284, 2000.

[Bre00] Bret, M. Virtual Living Beings. Lecture Notes in Artificial Intelligence, vol. 1834, pp. 119-134, 2000.

[Bro98] Brogan, D., Metoyer, R., and Hodgins, J. Dynamically Simulated Characters in Virtual Environments. IEEE Computer Graphics and Applications, vol. 15, no.5, pp. 58-69, 1998.

[Bur01] Burke, R., Isla, D., Downie, M., Ivanov, Y., and Blumberg, B. CreatureSmarts: The Art and Architecture of a Virtual Brain. Game Developers Conference, San Jose, CA, pp.147-166, 2001.

[Kal02a] Kallmann, M., and Thalmann, D. Modeling Behaviors of Interactive Objects for Real-Time Virtual Environments. Journal of Visual Languages and Computing vol. 13, pp. 177-195, 2002.

[Kal02b] Kallmann, M., Monzani, J.S., Caicedo, A., and Thalmann, D. A Common Environment for Simulating Virtual Human Agents in Real Time. Proc.Workshop on Achieving Human-Like Behavior in Interactive Animated Agents, Barcelona, Spain (to appear).

[Kar98] Karla, P., Magnenat-Thalmann, N., Moccozet, L., Sannier, G., Aubel, A., and Thalmann, D. Real-time Animation of Realistic Virtual Humans. IEEE Computer Graphics and Applications, vol.18, no.5, pp. 42-55, 1998.

[Luc00] Luck, M., and Aylett, R. Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments. Applied Artificial Intelligence, 14:3-32, 2000.

[Mag96] Magnetat-Thalmann, N., Carion, S., Courchesne, M., Volino, P., and Wu, Y. Virtual Clothes, Hair and Skin for Beautiful Top Models. Proc. Computer Graphics International ' 96, Pohang, Korea, pp. 132-141, 1996.

[Per96] Perlin, K., and Goldberg, A. Improv: A system for scripting interactive actors in virtual worlds. Proc. of ACM Computer Graphics Annual Conf, pp. 205-216, 1996.

[Sil99] Silva, D., Siebra, C., Valadares, J., Almeida, A., Frery, A., and Ramalho, G. Personality-Centered Agents for Virtual Computer Games. Proc. of Virtual Agents 99, Salford, UK, 1999.

[Ter94] Terzopoulos, D. , Rabie, T., and Grzeszczuk, R. Artificial Fishes: Autonomous Locomotion, Perception, Behavior, and Learning in a simulated physical world. Artificial Life, vol. 1(4), pp. 327–351, 1994.

[Tha96] Thalmann, D., Shen, J., and Chauvineau, E. Fast Human Body Deformations for Animation and VR Applications. Proc. Computer Graphics International 96, pp.166-174, 1996.

[Vos01] Vosinakis, S., and Panayiotopoulos, T. SimHuman: A Platform for Real-Time Virtual Agents with Planning Capabilities. Lecture Notes in Artificial Intelligence, vol. 2190, pp.210-223, 2001.