3D Agent-based Virtual Communities

Zhisheng Huang, Anton Eliëns and Cees Visser
Vrije University of Amsterdam
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
{huang,eliens,ctv}@cs.vu.nl

Abstract

In this paper we propose an approach to 3D agent-based virtual communities, in which autonomous agents are participants in VRML-based virtual worlds to enhance the interaction with users or serve as intelligent navigation assistants. In addition, an agent communication language (ACL) is designed as a high level communication facility, in particular for the realization of shared objects in virtual communities. As a typical example of 3D agent-based virtual communities, a VRML-based multi-user soccer game has been developed and implemented in the distributed logic programming language DLP. We discuss how DLP can be used for the implementation of 3D agent-based virtual communities.

Key words: Distributed virtual environment, Virtual community, VRML, Agent

1 Introduction

3D virtual communities and in particular VRML-based multi-user virtual worlds, have been adopted in a lot of application areas like 3D virtual conferencing [21], Web-based multi-user games [18], on-line entertainment [2], and e-commerce [17]. Examples of popular 3D virtual community servers are Active World [1] and Blaxxun Interactive [2]. However, most of them do not provide support for intelligent agents. Enhancing virtual worlds with intelligent agents would significantly improve the interaction with users as well as the capabilities of networked virtual environments [3, 6, 23].

The Blaxxun community server does provide support for agents. Agents in the Blaxxun community server may be programmed to have particular attributes and to react to events in a particular way. As a remark, originally the Blaxxun agents were called bots. In our opinion the functionality of Blaxxun agents does not surpass that of simple bots and we consider the term agent to be a misnomer. Despite the large number of built-in events and the rich repertoire of built-in actions, the Blaxxun agent platform in itself is rather limited in functionality, because the event-action patterns are not powerful enough to program complex behavior that requires maintaining information over a period of time.

In this paper we propose an approach to 3D *agent-based virtual communities* with the following two shades of meaning :

- *Virtual enviroments with embedded agents*: autonomous agents are participants in virtual communities. The main advantages are: the agent can be used to enhance the interaction with users. For instance, in a multi-user soccer game it is usually hard to find enough users to join the game at a particular moment. Autonomous agents can serve as goal keepers or players whenever they are needed. Moreover, autonomous agents always possess certain background knowledge about the virtual worlds. They can serve as intelligent assistants for navigation or as masters to maintain certain activities, like a referee in a soccer game.

- *Virtual enviroments supported by ACL communication*: Agent communication languages (ACL) are designed to provide a high-level communication facility. The communication between the agents can be used for the realization of shared objects in virtual worlds. For instance, in a soccer game, whenever an agent or user kicks the soccer ball, the kicking message should be broadcast to all other agents and users. The state of the soccer ball in the user's local world can be updated after receiving the message. Such a high-level communication facility can also be used to reduce message delays, which are usually a bottleneck in networked virtual communities. We will discuss performance related details in section 5.

Intelligent agents in VRML-based virtual worlds can be considered to be what we called *3D web agents* in [11]. A VRML-based 3D soccer game with a single user has been developed and implemented in [12], supported by the distributed logic programming language DLP [7]. In this paper we discuss an example how DLP can be used for the implementation of distributed multi-user soccer games by means of a 3D agent-based virtual community.

2 Agents in Virtual Communities

The term "virtual community" is usually used to refer to the general appearance and gathering of people by means of distributed computer systems, in particular on the Internet. A typical text-based virtual community is Internet Relay Chatting [14], whereas typical 3D web-based virtual communities are VRML-based, like the Blaxxun community server and DeepMatrix [19]. In VRML-based virtual communities, virtual worlds are designed by means of VRML, whereas the VRML External Authoring Interface (EAI) is used to connect autonomous agents running in a Web Browser to the plug-ins that are required to control the virtual worlds. Virtual communities usually have a client-server network architecture. In particular, they occasionally use a centralized server architecture, because the clients are running in a remote Web Browser and the Java platform security policy allows clients only to connect to the originating host.

The "Living Worlds" Working Group describes a general concept and context of VRML-based virtual communities [13]. A *scene* is used to refer to a set of VRML objects which is geometrically bounded and is continuously navigable, i.e. without "jumps". A *world* consists of one or more scenes linked together both from a technical and conceptual point of view. A *SharedObject* is an object whose state and behavior are to be synchronized across multiple clients. The SharedObject on one of these clients is called an instance of the SharedObject. In "Living Worlds" a *pilot* is used to refer to an instance of a SharedObject whose states

and behaviors are replicated by other instances, its drones. A *drone* is an instance of a SharedObject replicating the state or behavior of another instance, its pilot.

In agent-based virtual communities, a shared object is designed to be controlled by an agent. Therefore, a *pilot agent* is one which controls the states or behavior of a shared object, whereas a *drone agent* is one which replicates the state of a shared object. Based on the different types of shared objects, the agents can be further classified by means of the following three types:

- object agents: an autonomous program controls a simple shared object, like a soccer ball. Pilot object agents are usually located at the server side, whereas drone object agents are usually located at the client side.

- user agents: an autonomous program which controls a user avatar; it translates commands from users to messages for the communication between the agents. A pilot user agent is located at the user or client side. Drone user agents can be located at the server or other clients, however, they are usually not required as will be explained in section 4.

- autonomous agents: an autonomous program with its own avatar which is able to perform complex tasks, like an autonomous player in a soccer game. Pilot autonomous agents are located at the server side, whereas drone autonomous agents are located at all clients.

In addition, we also need multi-threaded, i.e. active components which are in charge of several aspects of the communication infrastructure. However, we would not call them agents, but *active communication components*, because agents are only interested in the collaboration with other agents rather than in particular aspects of the communication facilities themselves.

A programming language supporting 3D agent-based virtual communities, as described above, should have the following features:

- VRML EAI support: It should support VRML EAI, like Java does;

- Distributed communication capabilities: It should support network communication, like TCP/IP;

- Multiple threads of control: It should support multiple threads of control for the simulation of pilots and drones in both client and server sides;

- Declarative language: preferably, it should be a declarative language, like a logic programming language, which supports rule-based knowledge representation as is often necessary for the implementation of intelligent agents.

Based on the requirements above, the distributed logic programming language DLP has been extended to support 3D agent-based virtual communities.

3 Distributed Logic Programming for Virtual Environments

Distributed logic programming [7] combines logic programming, object oriented programming and parallelism. The use of DLP as a language for the implementation of agent-based virtual communities is motivated by the following language characteristics: object-oriented Prolog, VRML EAI extension, and distribution.

3.1 Object-oriented Logic Programming

DLP incorporates object-oriented programming concepts, which make it a useful tool for programming. The language accepts the syntax and semantics of logic programming languages like Prolog. It is a high-level declarative language suitable for the construction of distributed software architectures in the domain of artificial intelligence. In particular, it's a flexible language for rule-based knowledge representation [8].

In DLP, an object is designed as a set of rules and facts, which consists of a list of formulas built from predicates and terms (variables or constants). For instance, a rule like

$$
\begin{array}{l}
findHowtoReact(Agent, Ball, shooting) :- \\
\quad getPosition(Agent, X, Y, Z), getPosition(Ball, Xb, Yb, Zb), \\
\quad gatePosition(Agent, Xg, Yg, Zg), Distance(X, Y, Z, Xb, Yb, Zb, Distb), \\
\quad Distance(X, Y, Z, Xg, Yg, Zg, Distg), Distb =< kickableDistance, \\
\quad Distg =< kickableGoalDistance.
\end{array}
$$

states that *if the ball is kickable for the agent and the gate is within the kickable distance, then the agent should try to shoot.*

3.2 VRML EAI Extensions

DLP is an extensible language. Special-purpose requirements for particular application domains can easily be integrated in the existing object-oriented language framework. DLP has been extended with a run-time library for VRML EAI [16]. The following predicates are some examples of DLP VRML built-ins:

- URL-load predicate $loadURL(URL)$
  loads a VRML world at URL into the Web Browser.
- Get-position predicate $getPosition(Object, X, Y, Z)$
  gets the position values $\langle X, Y, Z \rangle$ of the Object in the VRML world.
- Set-position predicate $setPosition(Object, X, Y, Z)$
  sets the position values $\langle X, Y, Z \rangle$ of the Object in the VRML world.
- Get-property predicate $getSFVec3f(Object, Field, X, Y, Z)$
  gets a value (which consists of three float numbers X, Y, and Z) of the $Field$ of the $Object$.
- Set-property predicate $setSFVec3f(Object, Field, X, Y, Z)$
  assigns the $SFVec3f$ value X, Y, and Z to the $Field$ of the $Object$.

Furthermore, DLP programs are compiled to Java class files, which makes it a convenient tool for the implementation of VRML EAI applets.

3.3 Distributed Programming Language

DLP is also a distributed programming language. DLP programs can be executed at different computers in a distributed architecture.

The following predicates are some examples of TCP/IP networking primitives in DLP:

- Server predicate $tcp\_server(ServerPort, ServerSocket)$ creates a server socket.
- Server accepting a new client: $tcp\_accept(ServerSocket, ServerStreamIn, ServerStreamOut)$ creates a message input stream and a message output stream associated with the server socket.
- Client predicate $tcp\_client(ServerHostName, ServerPort, TimeOut,$ $ClientStreamIn, ClientStreamOut)$ creates a message input stream and a message output stream, connecting a client to a DLP server running at "ServerHostName".
- Bi-Directional Client / Server Communication predicates: $tcp\_get\_term(StreamIn, Term)$ gets a message term from the stream, and $tcp\_put\_term(StreamOut, Term)$ writes a message term to the stream.

Moreover, DLP allows for multiple threads of control in a single program, which makes it a convenient tool for the implementation of autonomous agents.

4 Distributed Communication

In general, a virtual community based on a client-server network architecture works as follows: all the client processes connect with a centralized server via a Web browser, usually by means of a TCP connection. The server receives, processes, and transfers the messages concerning shared objects to the clients for the necessary synchronization.

To improve the performance, multiple threads of control are introduced in both server and clients in the virtual communities, as is shown in Figure 1. Each thread has its own message queue to store incoming-messages (sent from other threads) which have not yet been processed. Thus, sending a message to a thread means sending the message to the recipient's message queue. Each client has its own communication thread, called *client thread*, which is in charge of the network communication. In addition, for each client a special thread called *server thread* is created at the server side for the network communication with its corresponding client thread. The introduction of multiple threads leads to the following communication patterns:

- communication between internal threads: messages are sent from a thread to another thread inside server or clients. This kind of communication is done directly, either in a asynchronous or synchronous way, without the intervention of active communication components.
- communication between threads across a network: messages are sent from a thread located at the server to a thread located at a client or vice versa. Sending a message from a thread located at the server to a thread located at a client has the following procedure:
  1. the message is sent to the corresponding server thread's message queue;
  2. the server thread retrieves the message from his message queue;

3. the server thread invokes *tcp_put_term* to put the message to the stream connected with the client thread;
4. the client thread uses *tcp_get_term* to get the message from the stream;
5. the client thread stores the message in the destination thread's message queue;
6. the destination thread retrieves the message from its own message queue.

- communication between two clients: messages are sent from a client thread to a thread located at another client. Since there is no direct connection between two clients, this kind of communication has to be achieved via the server.

The clients and server are designed to consist of two main components (Figure 1) : a general component, called gg-server and gg-client, which deal with the network communication and an application specific component, called wsserver and wsclient for the soccer game, which deals with anything that is relevant for the application. Furthermore, the *gg_echo* component is used for the actual message broadcasting.

In agent-based virtual communities, each agent is represented as a thread. Considering the high degree of autonomous behavior of user agents, we don't need the drone user agent at the server side and other client sides, which will become more clear in section 5.

For agent-based virtual communities, agent communication languages (ACL's) are used to serve as a high-level communication facility. KQML [9] and FIPA ACL [10], which are based on speech act theory [20], are popular agent communication languages. A message in an ACL usually consists of a *communicative act*, a sender name, a list of recipients, and additional content. Communicative acts like 'tell', 'ask', and 'reply', are used to identify the communication actions which may change the mental attitudes of the agents. Moreover, a set of agent interaction protocols based on ACL has to be defined to achieve interoperability among the agents. Agents need not to take care of the details how the messages are passed across the network, that is the responsibility of the active communication components.

5 Example: VRML-based Multiple User Soccer Game

We used the soccer game as one of the benchmark examples to test 3D agent-based virtual communities for the following reasons:

- multiple users: Multiple human users can join the soccer game, so that a virtual community is formed.
- multiple agents: Soccer games are multi-agent systems which require multiple autonomous agents to participate in the games, in particular the goalkeepers are better to be designed as autonomous agents, rather than human users, for their active areas are rather limited, i.e. only around the goal gates. The goalkeeper agents can be designed to never violate the rules of games.
- reactivity: A player (user or agent) has to react quickly in the game. Thus, it is not allowed to have serious performance problems.

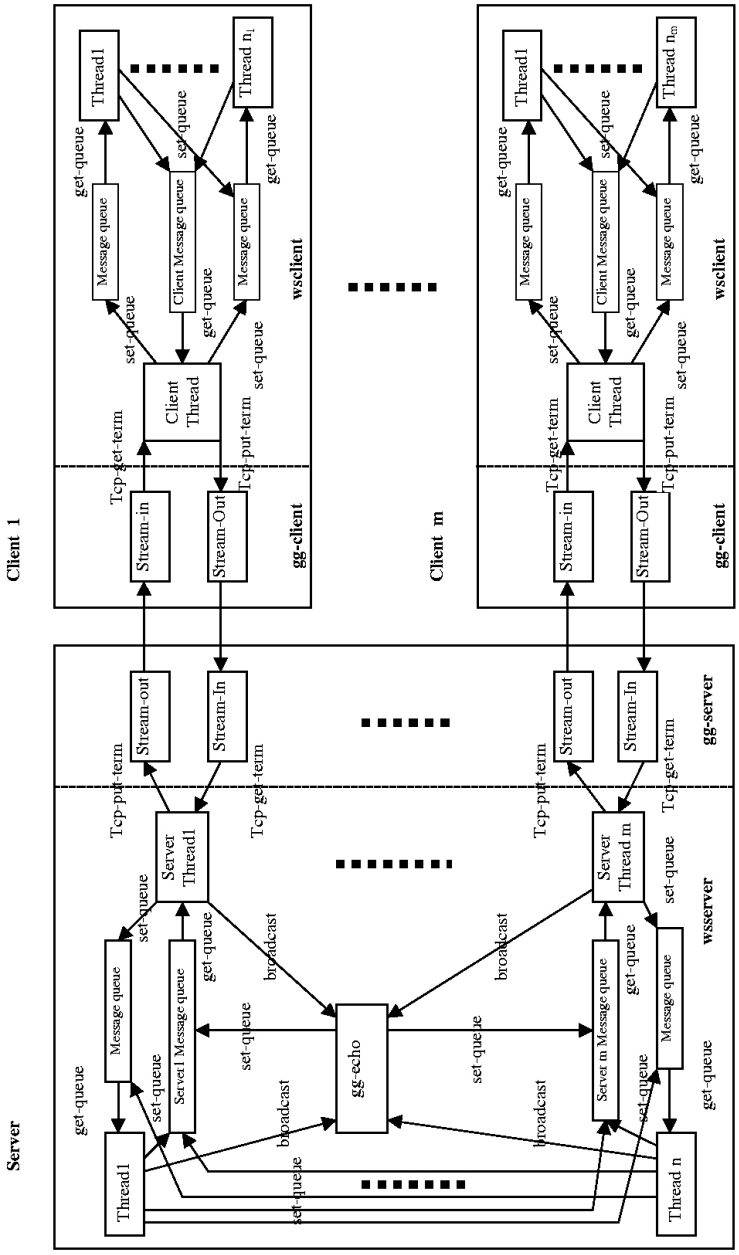

Figure 1: Communication among Multiple Threads in DLP

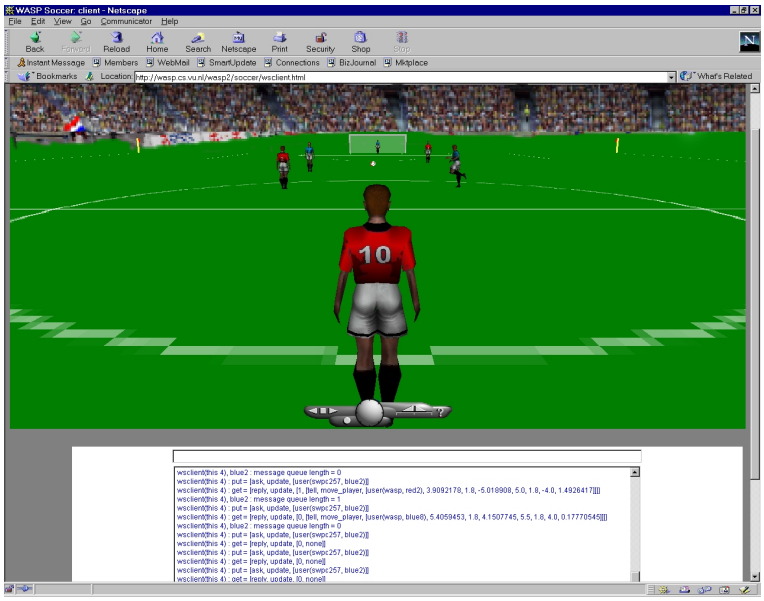

Figure 2: Screenshot of Soccer Game with Multiple Users

- cooperation/competition: Soccer games are typical competition games which require the strong cooperation among team-mates. Therefore, intelligent behavior is a necessity for agents.

- dynamic behavior: Sufficiently complex 3D scenes, including the dynamic behavior of the ball.

A screenshot of the soccer game with multiple users is shown in Figure 2.

We consider two playing teams, red and blue, in the soccer game. Two goal keepers, a soccer ball, and several agent players are designed to be pilot agents in the server. Whenever a new user joins the game, a client thread is created for which a user avatar is created to be the pilot agent in the client.

5.1 Agent players and their cognitive models

Each agent player may play one of the following four roles: *goal keeper*, *defender*, *mid-fielder*, and *forward*. Each role has its own active area in the soccer field. Each agent player has the following cognitive loop: sensing–thinking–acting. By sensing, agents use get-predicates to retrieve the necessary information about the current situation. The main information sources are: agent position, soccer ball position, and the goal gate position. In the stage of thinking, avatars have to reason about other players' positions or roles and decide how to react based on their preferences and the information about the current situation. Thinking results in a set of intentions, more exactly, a set of intended actions. By acting, agents use the set-predicates to take the intended actions.

In the current version of the soccer game we do not require that agents know all the rules of the soccer game, like penalty kick, free kick, corner kick, etc. In the simplified

soccer game, soccer players have several kick actions, like shooting, passing, run-to-ball, move-around-default-position, etc.

The agents in the soccer game use a simplified cognitive model of soccer games [12] in which the agents consider the information about several critical distances, then make a decision to kick. Despite this simplified "cognitive" soccer game model, each player shows a remarkably intelligent behavior [12].

5.2 Distributed Soccer Game Protocol

ACL is used to design a distributed soccer game protocol which states how the message should be processed and forwarded among the agents to achieve shared objects. The messages in the distributed soccer game protocol are a 3-tuple:

$$[Act, Type, ParameterList]$$

where Act is a communicative act; like 'tell', 'ask', 'register' ; Type is a content type, like 'position', 'rotation', 'kick-ball' ; ParameterList is a list of parameters for the content type.

The basic message formats for the distributed soccer game protocol are:

- register game: [register, game_name, from(Host)].
- register accept: [tell, accept, user(Host,Name)].
- register wait: [tell, wait, []].
- new player: [tell, new_player, user(Host,Name)].
- tell position: [tell, position, [user(Host,Name),position(X,Y,Z)]].
- tell rotation: [tell, rotation, [user(Host,Name),rotation(X,Y,Z,R)]].
- text chat: [tell, text, [user(Host,SenderName), RecipientNameList, Text]].
- text chat broadcast: [tell, text,[user(Host,SenderName), [all], Text]].
- kick ball: [tell, kick_ball, [user(Host,Name),force(X,Y,Z)]].
- tell game score: [tell, game_score, user(Host,Name)].
- ask game score: [ask, game_score, user(Host,Name)].
- reply game score: [reply, game_score, score(score1,score2)].
- unregister game: [unregister, game_name, user(Host,Name)].
- reply unregister game: [reply, unregister, done(Host,Name)].
- player gone: [tell, player_gone, user(Host,Name)].

The meaning of the message formats above are straightforward. For instance, the message [tell,position,[host(swpc257,red10),position(0,0,10)]] states that the current position of player red10 at host swpc257 is $\langle 0, 0, 10 \rangle$.

The distributed soccer game protocol for pilot agents are straightforward; they should regularly tell their position and rotation to the communication components if the position or rotation is changed, so that their information can be updated by their drones. Moreover, for the player agents, if they kick the ball, the kicking message has to be passed to the server.

received message	**condition**	**reply message**	**broadcast**
register game	player name available	register accept	tell new_player
register game	player name not available	register wait	
tell position			tell position
tell rotation			tell rotation
tell kick	legal kick		tell kick
tell kick	illegal kick		
text chat	recipient list	text chat	
text chat	broadcast		text chat
ask, game_score		reply game_score	
unregister game		reply unregister	player gone

Table 1: Distributed Soccer Game Protocol

The server decides which one is a legal kick and takes certain actions. Thus, the server plays a central role for the synchronization between pilot and drone agents. The distributed soccer game protocol for the server thread is a set of 4-tuples with the following format: $\langle M, C, RM, B\rangle$, which means that if Message M is received and Condition C holds, then reply the message RM and broadcast the message B. The protocol is shown in table 1.

In theory, the protocol above is sufficiently expressive to realize shared objects in the soccer game. However, in practice it results in several performance problems. Consider a problem caused by autonomous player agents: players may continuously run to the ball or other positions. If the players regularly send messages about their positions and rotations, the message queues grow rapidly, which results in serious message delays. In a worst case scenario, a user will never be able to kick a ball because its local world isn't updated.

5.3 Performance Improvement

In order to improve the performance and decrease the message delays, new message formats are required in the protocol so that the drone agents can simulate the behavior of their counterpart pilot agents at a high level, i.e. the behavior can be computed locally. However, note that the high level simulations are suitable only for autonomous agents and object agents, for their pilots are controlled by DLP programs; their behaviors are to some extent predicatable. Because of the high autonomy of the human users, user agents are usually hard to be simulated at a high level. Thus, the high level message formats are used only for autonomous player agents and objects agents.

One example of a high-level simulation is: if an agent wants to run to the position $\langle X1, Y1, Z1\rangle$ from the initial position $\langle X0, Y0, Z0\rangle$, then he sends a move-player message. Other high level message formats are:

- run and trace: the payer runs and trace the ball until it can kick the ball.
- move ball: the ball is moving to a new position.

Introducing the high level message formats significantly reduces the message delays. Assume that in the game there are u users, a autonomous player agents, including object

agents. Compared to autonomous agents, human agents are relatively slow to change their position or rotation. Assume also that each autonomous agent creates m_a messages per second and each human agent creates m_u messages per second. There are $M = a \times m_a + u \times m_u$ message per second in total. That means that each communication thread has to process M messages. If a communication thread is able to process M_c messages and $M_c < M$, then the message queue length becomes $t \times (M - M_c)$ after time t. Now, suppose that introducing a high-level message format f for which the average time period of the action is $at(f)$ and the probability is $p(f)$. A single high-level message m with format f corresponds to $at(f) \times m_a$ messages for a period $at(f)$. It reduces $m_a - 1/at(f)$ messages per second for a single occurrence of message m. In general, the reduced number of messages $M_r(f)$ per second by introducing f is as follows:

$$M_r(f) = (at(f) \times M \times p(f) + 1)/at(f)$$

The improved performance ratio $R(f)$ is defined as:

$$R(f) = M_r(f)/M \approx p(f)$$

The improved performance is mainly determined by the probability of the high-level message. For example, if the average time period for the action move_player is 4 seconds and the probability is 0.15, the move_player message results in a 15 percent performance improvement.

6 Conclusions

The two main extensions to VRML97 are expected to be: multi-user interaction and autonomous creatures [4]. In this paper we proposed an approach to 3D agent-based virtual communities, which is an attempt to provide a general framework to deal with these two issues at a VRML EAI level. Virtual environments embedded with intelligent agents offer a general solution to shared objects and autonomous creatures in VRML worlds. The interaction supported by agent communication languages provides a high-level multi-user interaction in virtual environments. We have developed and implemented a VRML-based multi-user soccer game, which illustrates that the Distributed Logic Programming language (DLP) is a high level tool for the development of 3D agent-based virtual communities.

References

[1] ActiveWorlds, `http://www.activeworlds.com`.

[2] Blaxxun Interactive Inc. `http://www.blaxxun.com`.

[3] W. Broll, E. Meier, and T. Schardt, *Symbolic Avatars Acting in Shared Virtual Environments*, `http://orgwis.gmd.de/projects/VR`, 2000.

[4] G. Carson, R. Puk, and R. Carey, Developing the VRML 97 International Standard, *IEEE Computer Graphics and Applications* 19(2), 1999, 52-58.

[5] DLP web site: http://www.cs.vu.nl/∼eliens/ projects/logic/index.html.

[6] R. Earnshaw, N. Magnenat-Thalmann, D. Terzopoulos, and D. Thalmann, Computer Animation for Virtual Humans, *IEEE Computer Graphics and Applications* 18(5), 1998, 24-31.

[7] Anton Eliëns, *DLP, A Language for Distributed Logic Programming*, Wiley, 1992.

[8] Eliëns, A., *Principles of Object-Oriented Software Development*, Addison-Wesley, 2000.

[9] T. Finin and R. Fritzson, KQML as an agent communication language, *Proceedings of the 3rd International Conference on Information and Knowledge Management*, 1994.

[10] FIPA web site:http://www.fipa.org.

[11] Z. Huang, A. Eliëns, A. van Ballegooij, P. de Bra, A Taxonomy of Web Agents, *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, IEEE Computer Society, 765–769, 2000.

[12] Z. Huang, A. Eliëns, and C. Visser, *Programmability of Inteligent Agent Avatars*, *Proceedings of the Autonomous Agents'01 Workshop on Embodied Agents*, Montreal, Canada, 2001.

[13] Living Worlds Working Group, http://www.web3d.org/WorkingGroups/living-worlds/.

[14] Z. Liu, Virtual Community Presence in Internet Relay Chatting, *Computer-Mediated Communication* 5(1), 1999.

[15] ISO, *VRML97: The Virtual Reality Modeling Language, Part 1: Functional specification and UTF-8 encoding*, ISO/IEC 14772-1, 1997.

[16] ISO, *VRML97: The Virtual Reality Modeling Language, Part 2: External authoring interface*, ISO/IEC 14772-2, 1997.

[17] E. Messmer, E-commerce yet to embrace virtual reality, http://www.idg.net/english/crd_commerce_441283.html

[18] MiMaze, http://www-sop.inria.fr/rodeo/MiMaze/

[19] G. Reitmayr, S. Carroll, and A. Reitemeyer, DeepMatrix – An Open Technology Based Virtual Environment System, *Visual Computer* 15, 395-412, 1999.

[20] J. R. Searle, *Speech Acts. An Essay in the Philosophy of Language*. Cambridge, 1969.

[21] Virtual European Statistical Lab, Conferencing using Vnet, http://vesl.jrc.it/en/comm/eurostat/research/supcom.97/01/conf/mainvnet.htm.

[22] WASP project home page: http://wasp.cs.vu.nl/wasp.

[23] M. Watson, *AI Agents in Virtual Reality Worlds – programming intelligent VR in C++*, Wiley, 1996.