Intelligent Embodied Agents

An approach to combine Agent Technology with Virtual Environments

Zhisheng Huang
Anton Eliëns
Cees Visser

Intelligent Multimedia Technology

An Approach to combine Agent Technology with Virtual Environments

Zhisheng Huang, Anton Eliëns, and Cees Visser

Vrije Universiteit

Amsterdam, 2004

Contents

List of Figures

List of Tables

Preface

This is a preface.

What You Should Already Know

This book assumes you have the following basic background:

- A general understanding of the Internet and the World Wide Web WWW.
- A working knowledge of Virtual Reality Modeling Language VRML and HyperText Markup Language HTML.

Some programming experience with the object-oriented language JAVA and the logic programming language PROLOG, is useful, but not required.

Z. Huang, A. Eliens and C. Visser

Amsterdam, The Netherlands

Acknowledgments

To all these wonderful people We owe a deep sense of gratitude especially ...

Z. Huang, A. Eliëns, and C. Visser

Acronyms

AI	Artificial Intelligence
DLP	Distributed Logic Programming Language
EAI	External Authoring Interfaces
HTML	Hyper Text Modeling Language
MAS	Multiple Agent Systems
VRML	Virtual Reality Modeling Language
VU	Vrije University Amsterdam
WASP	The project of Web Agent Support Programs
WWW	World Wide Web

Part I

Fundamentals

1 Introduction

The World Wide Web (WWW) offers people extremely large amounts of information and data from all over the world. In a relatively short time, the Web has become a de facto standard for the dissemination and retrieval of information. Because so much information is available, the Web is expected to be easy to use and offers attracting interfaces for information presentation and retrieval. However, most existing web pages are text-based, with quite limited functionalities of multimedia, like jpeg pictures, animated gif, sound and video clips , etc. Enhancing the Web with multimedia capabilities is an attempt to improve these interfaces.

The Virtual Reality Modelling Language (VRML) [ISO, 1997] has become a standard tool to build 3D virtual worlds for the Web. Using VRML, information can be presented more attractively in virtual environments. So-called *Virtual environments* are refered to computer created scenes or environments which provide the user with the sensation of ‘being inside’ as well as the possibility to interact by using computers controlled input-output devices. *Networked virtual environments* are refered to ones which can be accessed over the computer network, in particular, the Web. In this book, we focus on networked virtual environments. VRML is the most popular tool for creating networked virtual environments. In the sequel, *virtual worlds/virtual environments* are generally used to refer to the VRML-based worlds/scenes/environments.

VRML preserves almost all the functionalities of the text-based Web. Moreover, it provides a convenient and powerful tool to build 3D virtual worlds. VRML has been applied to applications which require powerful graphical interfaces for the Web, like virtual galleries, multimedia presentations, scientific visualization, 3D object simulation and exhibition, chat arena, 3D games, virtual communities, and e-commerce applications.

However, the current version of VRML, namely VRML'97, still has limited capabilities for the development of 3D Web applications. In order to make VRML more convenient and more powerful as a tool for the development of Web applications, people expect the following improvements and enhancements for 3D virtual worlds:

- **Efficient Behavior Control**. In order to control the dynamic behavior of objects, VRML'97 provides facilities for Java script nodes and ROUTE semantics. However, it is still difficult to program a sophisticated scenario of dynamic objects by using these facilities. In order to solve this problem, VRML offers an EAI (External Authoring Interface). Java is a typical EAI language, because it is a standard programming language for the development of Web applications. VRML EAI and Java allow programmers to control 3D virtual worlds, more exactly, the contents of a VRML browser window embedded in a web page from a Java applet. EAI does this with a browser plug-in that allows embedded objects in a web page to communicate with each other. Therefore, in order to efficiently control the dynamic behaviors of the objects in 3D virtual worlds, VRML needs the Java EAI. We call that approach *VRML+JAVA.*.

- **Multiple User Support**. VRML'97 is designed for a single user framework. However, a lot of Web applications require the support for multiple users and shared events in virtual worlds; this way multiple users can interact with each other and share the same virtual world events. Supporting multiple users in virtual environments requires a more sophisticated technology for processing messages in a network. That appeals for the technology of 3D virtual communities. The most popular tools for building virtual communities are currently Activeworlds [ActiveWorld, 2000] and Blaxxun virtual community servers [Blaxxun, 2000]. Both Activeworlds and Blaxxun virtual communities use the VRML+JAVA approach.

- **Convenience in navigation**. Navigating in 3D virtual worlds is still difficult. Users often feel lost in virtual worlds, and do not know where and how to go. That applies for the development of 3D navigation assistants. One of the examples of such navigation assistants is the agent concept in Blaxxun virtual communities. The 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

	VRML	VRML+JAVA	VRML+JAVA+PROLOG
3D worlds	yes	yes	yes
Multimedia	yes	yes	yes
Dynamic Behaviors	weak	yes	yes
Multiple Users		yes	yes
Navigation Guides		weak	yes
Intelligent Agents		weak	yes

Table 1.1 Relation between the functions and the approaches

platform in itself is rather limited in functionality, simply because event-action patterns are not powerful enough to program complex behavior that requires maintaining information over a period of time. More generally, by *agent* we mean one which can be programmed to conduct complex behavior, and by *intelligent agent* we mean an agent that has the capability of knowledge representation and knowledge manipulation. We will discuss agents and agent technology in details in the part II: 3D web agents. The logic programming language (PROLOG) has become one of the most popular programming languages in artificial intelligence. Naturally, PROLOG is one of the best candidates for the implementation of intelligent agents. We call the approach which combines the VRML+Java EAI technology with logic programming the *VRML+JAVA+PROLOG* approach.

- **Intelligent agents**. A navigation assistant is just one of the examples of intelligent agents. In order to offer more efficient and powerful interfaces in 3D virtual worlds for the users, we need a lot of the functionalities from intelligent agents. Some of the examples are:

  – Information agent: An agent which can gather information over the Web according to the user profiles;

  – Presentation agent: An agent which knows how to present the information more efficiently to the users;

  All of those enhanced functionalities again require a more powerful approach which combines the technology of VRML+JAVA+PROLOG.

The summary of the relation between the functions and the approaches is shown in Figure 1. In the figure, we use ”yes” to denote that the approach can efficiently support the function, and use ”weak” to denote that although the approach may support the function, it is not efficient.

The distributed logic programming language (DLP) was first proposed in 1992 by A. Eliëns. DLP supports the Blaxxun VRML client interface library. Therefore, DLP is a programming language which supports the approach VRML+JAVA+PROLOG. Furthermore, our slogan in this book is:

$$VRML + JAVA + PROLOG = DLP + VRML$$

which states that Distributed logic programming in VRML virtual environments is the tool for the approach $VRML + JAVA + PROLOG$.

In this book, we introduce distributed logic programming and discuss how we can use DLP to develop 3D virtual world application. In particular, we investigate how DLP can be used to implement agents which use rules to guide their behaviors in virtual environments. We have organized this book into three parts:

- *Part I: Fundamentals (Chapters 1-4).* This part introduces the fundamental notions of the distributed logic programming for virtual environments, including the introduction to the logic programming language PROLOG, and the distributed logic programming language (DLP).

- *Part II: 3D Web Agent (Chapters 5-9).* Part II discusses how DLP can be used to develop and implement 3D web agents for virtual environments. Chapter 5 is a brief introduction to the agent types and their architectures. Chapter 6 overviews the notions of web agents and their taxonomy. Chapter 7 investigates how DLP can be used to develop 3D web agents. Chapter 8 discusses the issue of avatar design, and Chapter 9 introduces STEP , a scripting language for 3D web agents, and XSTEP, the XML-encoded STEP , a markup language for 3D web agents.

- *Part III: Virtual Communities (Chapters 10-12).* Part III focuses on the topics of virtual communities. Chapter 10 is an overview of the existing approaches of virtual communities. Chapter 11 discusses how DLP can be used to develop 3D virtual communities. Chapter 12 concludes this book.

2 Logic Programming

This chapter is an introduction to logic programming languages, like Prolog. In case you're familiar with logic programming, you can skip this chapter.

2.1 WHAT IS PROLOG

Prolog, which stands for PROgramming in LOGic, is based on the mathematical notions of relations and logical inference. Rather different from imperative programming languages, like C++, Java, etc., in which programmers have to describe how to compute a solution procedurally, Prolog is a declarative language, in which a program consists of a set of facts and logical relationships (rules) which describe the relationships which hold for the given application. Rather then running a program to obtain a solution, the user asks a question (or alternatively called a query, or a goal). When asked a question, the inference engine of the language would search through the set of facts and rules to find the answer.

In artificial intelligence, knowledge or beliefs are normally represented as a set of rules and facts. Prolog is widely used in artificial intelligence applications, such as natural language processing, automated reasoning systems, expert systems, and agent systems.

2.2 STRUCTURE OF PROLOG

A Prolog program consists of a set of facts, rules, and queries (goals). Rules and facts in Prolog are called clauses.

- **Fact**. A fact states a property, like *John is a boy*, or a relation, like *Jan is a parent of John*, which can be written in Prolog as follows:
  ```
  boy(john).
  parent(jan, john).
  ```

  According to the conventions of Prolog, a variable should begin with an upper case letter, whereas a constant should begin with lowercase letter, or enclosed in single quotes. Therefore, we use lower-cased item *john* to denote the name John above. If we want the name to start with an upper case letter, we can write it as follows:

  ```
  boy('John').
  parent('Jan', 'John').
  ```
  In most logic programming languages, like DLP, the following two statements are considered the same:

  ```
  boy(john).
  boy('john').
  ```

  However, note that $'John'$ and $'john'$ are considered to be two different constants in Prolog. A relation (or property), like $parent(X, Y)$, is called an atomic formula (or an *atom* for short). Therefore, a fact is an atomic formula which ends with a full-stop '.'. A relation (or property) name is often called a predicate. A term is a variable, a constant, or a compound term which is built from variables, constants, and structures.

- **Rule**. A rule states logical relations among relations or properties, which consists of an antecedent and a conclusion like :

  $$\langle Conclusion \rangle : -\langle Antecedent \rangle.$$

  The conclusion of a rule is sometime called *head of rule*, whereas the antecedent is sometimes called *body of rule*. Example: the statement, *if X is a parent of Y and X is male, then X is the father of Y*, is written as :

  ```
  father(X, Y):- parent(X,Y), male(X).
  ```

  Note that there might exist more than one atomic formula in the antecedent of a rule, however, only one atomic formula is allowed to appear in the conclusion part of a rule.

- **Query**. A query states a question, which begin with "?-", follows one or several atomic formulas (which are separated by the comma ','). Example: the question *who is the father of John* can be written as:

  ```
  ?-father(X,john).
  ```

One of main advantages of Prolog is the use of recursion. The following is one of the examples:

Example 1 Search in Directed Graph. *Figure 2.2 shows a directed graph. We use successor(X, Y) to denote that the node Y is a successor of the node X in the graph. We use path(X, Y) to denote that there exists a path from the node X reaches the node Y. The problem can be formalized in Prolog as follows:*

```
successor(a,b).
successor(a,c).
successor(a,d).
successor(b,e).
successor(b,f).
successor(c,g).
successor(d,g).
successor(e,h).
successor(g,i).

path(X,Y):-
    successor(X,Y).

path(X,Z):-
    successor(X,Y),
    path(Y,Z).

path(X,X).
```

To know which nodes are successor of node a, just make the query $?-successor(a, X)$. Prolog would answer that $X = b; X = c; X = d$. The followings are other examples of queries:

- $?-path(e, f)$: *to ask if the node e can reach the node f;*

- $?-path(c, X)$: *to ask which are nodes that can be reached from the node c.*

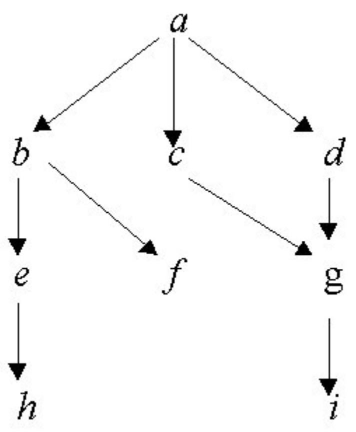


Fig. 2.1 Search in Directed Graph

2.3 HOW PROLOG WORKS: UNIFICATION AND BACKTRACKING

Prolog inference engine use a general deduction system to find solutions, which concerns the following two main notions:

- **Unification**: Unification is to find a substitution that would make two atomic formulae look the same. Sometimes the unification in Prolog is called matching. The matching operation takes two terms and tries to make them identical by instantiating the variables in both terms. Matching, if it succeeds, results in the most general instantiation of variable. For example, for the query $? - successor(a, X)$, the Prolog inference engine searches the set of facts and rules to find a unification of the conclusion of a rule (or a fact) and the query. The unification of $successor(a, X)$ and the fact $successor(a, b)$ results in the solution: $X = b$. The search procedure can be represented as a search tree, which is shown in Figure 2.2. If the inference engine can find a unification between the query and a fact, then it would result in a solution for the query. If the inference engine can find a unification between the query and a conclusion of a rule, then the current query would be replaced with the (instanted) body of the rule. The inference engine would continue to search for the solutions on the new queries. In order to find all solutions of a query, the inference engine would search in all of the databases. In the search, sometimes the Prolog engine cannot find any unification for the current query. Thus, the engine has to backtrack.

- **Backtracking**: withdraws from the part of the current search path to find other alternative path. For example, for the query $? - path(a, e)$, the engine finds a unification between the query $path(a, e)$ and the conclusion $path(X, Y)$ of the rule (p1), then the query is replaced with the body of the rule (p1), namely, the new query becomes $successor(a, e)$. After searching for all of the database, the engine cannot find any unification for the query $successor(a, e)$. Thus, the engine has to go back one step back to find a new unification for the old query $path(a, e)$. Then, the engine finds a unification between the query $path(a, e)$ and the head of the rule (p2). The current query would be $successor(a, Y), path(Y, e)$ after the instancing. The engine would continue to work on the current query $successor(a, Y), path(Y, e)$ to find the solutions. The whole search procedures is shown as a search tree in Figure 2.3.

Prolog inference engine always processes the queries by the order from left to right. For example, for the query $successor(a, Y), path(Y, e)$, the engine always tries to find the solution for $successor(a, Y)$ first, then for $path(Y, e)$. That would often result in an infinite loop in the searching if the program is not carefully written to avoid the problem. Therefore, every logic program should be carefully designed to avoid infinite loops.

Example 2 *A problem would lead to infinite loop*

```
successor(a,b).
successor(a,c).
successor(a,d).
successor(b,e).
successor(b,f).
successor(c,g).
successor(d,g).
successor(e,h).
successor(g,i).

path(X,Y):-
    successor(X,Y).
path(X,Z):-
    path(X,Y),
    successor(Y,Z).

path(X,X).
```

For the query $path(X, e)$, Prolog would give the following answers:

```
X=b;
X=a;
......
Warning: out of stack.
```

The readers can draw a search tree on the example above to see why it leads to an infinite loop.

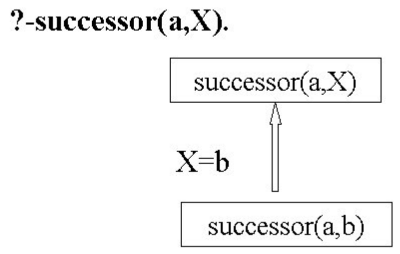


Fig. 2.2 Search in Directed Graph

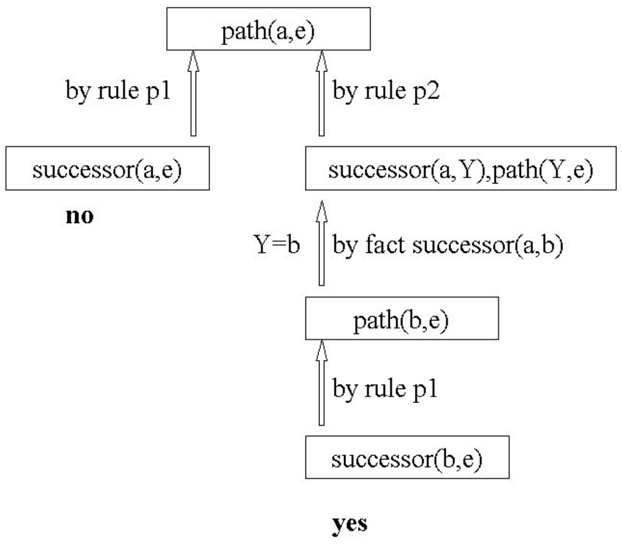


Fig. 2.3 Backtracking

2.4 WORKING ON LISTS

In Prolog, lists are the basic data structure used to represent a set of data. Here are some examples of lists:

- [ ]: an empty list;
- $[a]$: a list with one element a;
- $[X]$: a list with one element X;
- $[a, b]$: a list with two elements a and b. Note that the order of elements in a list is important. The list $[a, b]$ is different from the list $[b, a]$;
- $[a, [a]]$: a list with a as the first element, and the list $[a]$ as the second element;
- $[a|Tail]$: a list with a as the first element, and the other part of the list is a list $Tail$. A unification between the list $[a]$ and the list $[a|Tail]$ results in $Tail = [\,]$;
- $[Head|Tail]$: a list with $Head$ as its first element, and the other part of the list is $Tail$. For instance, a unification between the list $[a, b, c]$ and $[Head|Tail]$ would result in $Head = a$ and $Tail = [b, c]$;
- $[a, b|X]$: a list with a as the first element, b as the second element, and the other part of list is X. For example, a unification between $[a, b|X]$ and $[a, b]$ results in $X = [\,]$, and unification of $[a, b|X]$ and $[a, b, c]$ results in $X = [c]$.

Here are some Prolog programs on lists:

Example 3 *Element of a List.*
member$(X, List)$ *states a relation in which* X *is an element of the list List:*

```
member(X, [X|List]).
member(X, [Y|List]):-
    member(X,List).
```

Example 4 *Append.*
append$(List1, List2, List3)$ *states a relation in which* $List3$ *is a list from appending* $List2$ *to* $List1$*, namely,* $List3$ *is the concatenation of* $List1$ *and* $List2$*.*

```
append([ ],L,L).
append([H|L1],L2,[H|L3]):-
    append(L1,L2,L3).
```

Example 5 *Naive-reverse of a list.*
$naive_reverse(List1, List2)$ *states a relation in which* $List2$ *is a reversed list of* $List1$*:*

```
naive_reverse([ ], [ ]).
naive_reverse([Head|Tail], List):-
    naive_reverse(Tail, NewTail),
    append(NewTail, [Head],List).
```

Example 6 *Subset.*
$subset(List1, List2)$ states a relation in which all elements of $List1$ are elements of $List2$:

```
subset([ ], _).
subset([X|Y], Z):-
    member(X, Z),
    subset(Y, Z).
```

In the example above, the first statement can be written as $subset([\,], X)$, where X is an arbitrary variable. Note that the variable X occurs only once, which is not used anywhere else. We can use the notation '_' which denotes an arbitrary variable in the statement. Thus, the statement is rewritten as $subset([\,], \_X)$, by which we can avoid the single occurrence of variables. Although the single occurrence of variables do not harm the program, avoiding single occurence of variables sometimes helps us to find the typos in programs.

2.5 BUILT-IN PREDICATES IN PROLOG

Prolog provides many built-in predicates. The following predicates are the most useful ones. Refer to the appendix for the details of other built-in predicates. For any variable X, we use $+X$ to denote that X is an input variable, namely, before the unification, it should be instanced. We use $-X$ to denote that X is an output variable, namely, before the unification, it should be uninstantiated. Moreover, we use $?X$ to denote a variable which can be an input variable, or an output variable. A predicate (or function) is followed by a slash '/' and a number n denotes that it is an n-arity predicate (or function).

2.5.1 Arithmetic evaluation

- $?Term$ is $+Eval$ (evaluate expression)
- Basic Arithmetic functors
  - $'+'/2$: plus
  - $'-'/2$: minus
  - $'*'/2$: multiple
  - $'/'/2$: divide

Example 7 *Length of a List.*
length(List, N) states that the number of the elements in List is n:

```
length(L,N) :-
  length(L, 0, N).

length([], N, N).
length([_|L], N0, NL):-
    N1 is N0+1,
    length(L, N1, NL),
```

2.5.2 Arithmetic comparison

- $+Eval =:= +Eval$ (arithmetic equal)
- $+Eval = \backslash = +Eval$ (arithmetic not equal)
- $+Eval > +Eval$ (arithmetic greater than)
- $+Eval >= +Eval$ (arithmetic greater than or equal)
- $+Eval < +Eval$ (arithmetic less than)
- $+Eval =< +Eval$(arithmetic less than or equal)

Example 8 *Sort of a Number List.*
sort(List1, List2): List2 is sorted list of List1. For example, $sort([3, 6, 2, 4], List2)$ *would result in* $List2 = [2, 3, 4, 6]$.

```
sort([ ],[ ])$.
sort([X],[X]).
sort([X|Y],Z):-
    sort(Y,Y'),
    insert(X,Y',Z).


insert(X,[ ],[X]).
insert(X,[Y|Z],[X,Y|Z]):-
    X=<Y.
insert(X, [Y|Z],[Y|Z']):-
    X>Y,
    insert(X, Z, Z').
```

2.5.3 Cut and Fail

As discussed before, Prolog has an internal inference engine to find solutions. Although this inference mechanism is often sufficient to solve many problems, we need additional facilities to direct the search path. Built-in predicates like cut and fail are introduced to control the search space.

- **Cut**: The cut , written as '!', is used to stop the backtracking if something fails beyond that point. For instance, the predicate $memberchk(X, Y)$ succeeds if the element X is a member of the list Y.

  ```
  memberchk(X,[X|_]) :-
      !.
  memberchk(X,[_|Y]) :-
      memberchk(X,Y).
  ```

  The cut used in the first statement has the following meaning: whenever the element is found in the list, it is not necessary to keep the previous alternative solutions for backtracking.

- **Fail**: Fail is a built-in predicate which never succeeds, it starts backtracking to a previous choice point. Fail may be used to program infinite loops, or in combination with ' ! (cut) ' causes a predicate to fail.

Exercises

2.1 Draw a search tree for the example 2 to show why it leads to an infinite loop.

2.2 Implement the following relations in Prolog:
2.2.1. $factorial(N, Fac)$ where $Fac = N!$.
2.2.2. $sum(N, Sum)$ where $Sum = 1 + 2 + 3 + ... + N$.

2.3 Implement the following relations in Prolog:
2.3.1. $prefix(List1, List2)$ states a relation in which $List1$ is a prefix of $List2$, for example, $prefix([a, b], [a, b, c, d])$ holds;
2.3.2. $suffix(List1, List2)$ states a relation in which $List1$ is a suffix of $List2$, for example, $suffix([b, c, d], [a, b, c, d])$ holds;
2.3.3. $intersection(List1, List2, List3)$: $List3$ is the intersection of the set $List1$ and the set $List2$;
2.3.4. $union(List1, List2, List3)$: $List3$ is the union of the set $List1$ and the set $List2$.

3 Distributed Logic Programming

Distributed Logic Programming combines logic programming, object oriented programming and parallelism, which may be characterized by the pseudo-equation

$$DLP = LP + OO + ||$$

The language DLP can be regarded as an extension of Prolog with object declarations and statements for the creation of objects, communication between objects and the destructive assignment of values to non-logical instance variables of objects.

3.1 OBJECT DECLARATIONS

Object declarations in DLP have the following form:

```
:- object name.
var variables.
clauses.
:- end_object name.
```

where *object* and *end_object* are directives to delimit an object. Variables declared by *var* are non-logical variables; they may be assigned values by a special statement.

Objects act as prototypes in that new copies may be made by so-called *new* statements. Such copies are called instances . Each instance has its private copy of the non-logical variables of the declared object. In other words, non-logical variables act as instance variables.

Dynamically, a distinction is made between active objects and passive objects. Active objects must explicitly be created by a *new* statement. Syntactically, the distinction between active and passive objects is reflected in the occurrence of so-called constructor clauses in the declaration of active objects. Constructor clauses are clauses of which the head has a predicate name identical to the name of the object in which they occur. Constructor clauses specify an object's own activity. Other clauses in an object declaration may be regarded as method clauses, specifying how a request to the object is handled. Passive objects only have method clauses.

3.2 STATEMENTS

DLP extends Prolog with a number of statements for dealing with non-logical variables, the creation of objects and the communication between objects. These statements may occur as a goal in the body of a method.

Non-logical variables. For assigning a term t to a non-logical variable x the statement

```
x:=t
```

is provided. Before the assignment takes place, the term t is simplified and non-logical variables occurring in t are replaced by their current values. In fact, such simplifications take place for each goal. DLP also supports arithmetical simplification.

New expressions. For dynamically creating instances of objects the statement

$O := new(c)$

is provided, where c is the name of a declared object. When evaluated as a goal, a reference to the newly created object will be bound to the logical variable O. For creating active objects the statement

$O := new(c(t_1, ..., t_n))$

must be used. The activity of the newly created object consists of evaluating the constructor goal $c(t_1, ..., t_n)$, where c is the object name and $t_1, .../t_n$ denote the actual parameters. The constructor goal will be evaluated by the constructor clauses.

Method calls A method call is the evaluation of a goal by an object. To call the method m of an object O with parameters $t1, ..., t_n$ the statement

$O < -m(t_1, ..., t_n)$

must be used. It is assumed that O is a logical variable bound to the object to which the request is addressed. When such a goal is encountered, object O is asked to evaluate the goal $m(t_1, ..., t_n)$. If the object is willing to accept the request then the result of evaluating $m(t_1, ..., t_n)$ will be sent back to the caller. After sending the first result, subsequent results will be delivered whenever the caller tries to backtrack over the method call. If no alternative solutions can be produced the call fails. Active objects must explicitly interrupt their own activity and state their willingness to accept a method call by a statement of the form

$accept(m1, ..., m_n)$

which indicates that a request for one of the methods $m_1, ..., m_n$ will be accepted.

Inheritance An essential feature of the object oriented approach is the use of inheritance to define the relations between objects. Inheritance may be conveniently used to factor out the code common to a number of objects.

The declaration of an object *name* inheriting from an object *base* is:

```
:- object name : [base].
var variables.
clauses.
:- end_object name.
```

Multiple base objects are separated by "," in the declaration like:

$: -object\ \ name : [base_1, base_2, ..., base_n].$

3.3 EXAMPLES

3.3.1 Hello World

The following is a simple DLP program, which only prints the text 'hello world'.

```
:- object hello_world1.

main:-
    format('Hello, World ~n').

:- end_object hello_world1.
```

Similar to Java, the predicate 'main' is the starting point of an object. If the program above is saved in a file 'helloworld1.pl', we can use the command 'dlpc helloworld1.pl' to compile the DLP program. Note that the file extension of a PROLOG or DLP program is 'pl'. After compiling, we can use the command 'dlp hello_world1' to run the program. Note that we use the file name to compile a DLP program file, however, we use the object name to run a DLP program, for a program may consist of multiple objects.

The second 'hello world' example prints 20 times the text 'Hello World':

```
:- object hello_world2.

var count=0.

main:-
    repeat,
       format('Hello World ~w ~n',[count]),
       ++count,
    count >= 20,
    !.

:- end_object hello_world2.
```

The program above uses a non-logical variable 'count' as a counter in a repeat loop. If the goal '$count \geq 20$' fails, the program will backtrack to the start of the 'repeat' loop until the condition '$count \geq 20$' succeeds. The program above looks more like a procedural Java program instead of a declarative program. However, we can design a recursive program which behaves the same but without using the non-logical variable and the 'repeat' loop. We leave it as an exercise.

In the following, we show an object which prints the text 'hello', and another object which prints the text 'world'.

```
:- object hello.

var count=0.

hello:-
    repeat,
      format('hello ~w~n',[count]),
      sleep(500),
     ++count,
    count >= 10,
    !.

:- end_object hello.

:- object world.
```

```
var count=0.

world:-
    repeat,
      format('world ~w~n',[count]),
      sleep(250),
      ++count,
    count >= 10,
    !.

:- end_object world.

:- object hello_world3.

main:-
      _H := new(hello),
      _W := new(world).

:- end_object hello_world3.
```

The object 'hello_world3' uses the 'new' statement to create two threads. In order to see that these two threads run independently, we specify different 'sleep' times for each thread. The goal 'sleep(500)' is used to delay the thread for 500 milliseconds.

Observing the fact that the above-mentioned objects 'hello' and 'world' have the same program structure, we can implement a single object to achieve the same result. Moreover, in the following, we want to design a 'hello world' program in which multiple threads can share an independently running 'clock' thread. The 'hello' and 'world' threads print their text until the clock counts 10 :

```
:- object clock.

        var time = 0.

        get_clock(T):-
          T := time.

        set_clock(T):-
          time := T.

:- end_object clock.

:- object pulse.
```

```
pulse :-
    repeat,
      sleep(1000),
      clock <- get_clock(T),
      T1 is T + 1,
      clock <- set_clock(T1),
    T1 > 10,
    !.

:- end_object pulse.

:- object hello4.

hello4(Text):-
    repeat,
      clock <-get_clock(Time),
      format('~w ~w~ n', [Text,Time]),
      sleep(500),
    Time >= 10,
    !.

:- end_object hello4.

:- object hello_world4.

main:-
      _C := new(pulse),
      _H := new(hello4(hello)),
      _W := new(hello4(world)).

:- end_object hello_world4.
```

The object 'clock' provides the two methods *get_clock* and *set_clock* for getting and setting the clock pulse counter. The object 'pulse' simulates a clock by incrementing a counter after each repeat step.

3.3.2 File I/O

This is an example of file I/O operations in DLP.

```
:- object pxfile.

var x, y, z.
```

```
wr_file(Args) :-
      open('tmptext.1', write, Stream),
      format(Stream, 'aap noot mies, args = ~w~ n', [Args]),
      nl(Stream),
      put_char(Stream, a),
      put_char(Stream, b),
      put_char(Stream, c),
      nl(Stream),
      nl(Stream),
      close(Stream).

rd_file(_) :-
      open('tmptext.1', read, Istr),
      open('tmptext.2', write, Ostr),

      repeat,
          get_char(Istr, Char),
          write(Ostr, Char),
        Char = end_of_file,
      !,

      nl(Ostr),
      close(Istr),
      close(Ostr).

main(Args) :-
      wr_file(Args),
      rd_file(Args).

:- end_object pxfile.
```

3.3.3 Buffer Producer and Consumer

DLP multi-threaded (bounded) buffer + producer + consumer example featuring active objects, communication by rendez-vous, and non-logical variables. Execution starts at method main in object pxbuff. Method main redirects the output of this program to a browser text area and creates three active objects: active_buffer , term_consumer , and term_producer . The invocations of new/1 in main result in the creation of three independently running objects. These objects will start their execution at the specified object constructors in the corresponding new/1 goals :

```
:- object pxbuff.
```

```
 main :-
      B := new(active_buffer(10)),
      C := new(term_consumer(5,B)),
      P := new(term_producer(5,B)).

:- end_object pxbuff.
```

When an active instance of a term_producer object is created, execution starts at the term_producer object constructor. The constructor prints a message and invokes method loop/2. The 'loop' method executes N times and sends a Prolog term (in this example an integer) to the active_buffer thread, referenced by the logical variable B by means of the goal "$B < -put_term(I)$" :

```
:- object term_producer.

    term_producer(N,B) :-
           format('P ~w term_producer main/2 running~ n', [this]),
           loop(N,B).

    loop(0,_) :-
           format('P ~w end of loop ...~ n', [this]).

    loop(I,B) :-
           format('P ~w sending ~w~ n', [this, I]),
           B <- put_term(I),
           N is I-1,
           loop(N,B).

:- end_object term_producer.
```

The creation of an active term_consumer object in main/0 of object pxbuff results in the execution of the term_consumer/2 constructor. The constructor outputs a message and starts loop/2. The loop/2 method executes N times and retrieves its data from the active buffer thread by means of the "B¡-get_term(T)" goal :

```
:- object term_consumer.

  term_consumer(N,B) :-
      format('C ~w term_consumer main/2 running~ n', [this]),
      loop(N,B).

      loop(0,_) :-
```

```
            format('C ~w end of loop ...~ n', [this]).

        loop(I,B) :-
            B <- get_term(T),
            format('C ~w receiving ~w~ n', [this, T]),
            N is I-1,
            loop(N,B).

:- end_object term_consumer.
```

Object active_buffer contains four non-logical variables: head, tail, size, and count. As opposed to logical (Prolog like) variables, non-logical variables can be updated destructively. Execution starts at the active_buffer/1 object constructor. The constructor outputs a message and invokes loop/1. Method loop/1 accepts either a put_term/1 or get_term/1 method invocation request of an independently running term_producer or term_consumer object, depending on the internal state of this active_buffer object as specified by the guards: in case the current number of data items in the buffer is less than size or greater than zero, the first matching method request in the accept queue will be accepted for execution by active_buffer. If only one guard holds, the corresponding method entry will be accepted. In case there is no matching method request the thread will be blocked until such a method message arrives. A term_producer or term_consumer thread will block until active_buffer accepts a particular method invocation request and has returned its answer. When either a get_term/1 or put_term/1 method is accepted, the corresponding method in active_buffer will be executed : Method get_term/1 retrieves the first entry in the linked list and returns the corresponding term after the non-logical variable head of the linked list has been updated. Method put_term/1 will store the term in the linked list and updates the non-logical variable tail or both the head and the tail of the list. After a get_term/1 or put_term/1 method "rendez-vous", loop/1 in active_buffer prints the current state (see out_list/1) and starts the next loop iteration.

```
:- object active_buffer.

var head=null, tail=null, size=3, count=0.

active_buffer(N) :-
      format('B ~w active_buffer main/1 running~ n', [this]),
      loop(N).

loop(0) :-
      format('B ~w end of loop ...~ n', [this]).

loop(I) :-
```

```
     accept(
           put_term(P) <== [count < size],
           get_term(G) <== [count > 0]
            ),
     out_list(head),
     N is I-1,
     loop(N).

get_term(Term) :-
     -- count,
     head <- get_node_term(Term),
     head <- get_node_next(Next),
     head := Next,
     format('B ~w get term ~w~ n', [this, Term]).

put_term(Term) :-
     Node := new(buffer_node),
     Node <- set_node_term(Term),
     add_node(count, Node),
     format('B ~w put term ~w~ n', [this, Term]),
     ++ count.

add_node(0, Node) :-
     !,
     head := Node,
     tail := Node.

add_node(_, Node) :-
     tail <- set_node_next(Node),
     tail := Node.

out_list(Node) :-
    format('B~tcurrent nodes:~ n'),
    out_list(0, Node).

out_list(Curr, _) :-
    Curr = count, !,
    format('B~tend node list.~ n').

out_list(I, Node) :-
    N is I + 1,
    Node <- get_node_term(Term),
    Node <- get_node_next(Link),
    format('B~tnode no. ~w, term = ~w~ n', [N, Term]),
    out_list(N, Link).
```

```
:- end_object active_buffer.
```

Object buffer_node is a passive object (no constructor involved). It has two non-logical variables: term and next. These variables are destructively updated by the set_node_term/1 and set_node_next/1 methods, respectively. Object buffer_node is used by active_buffer to construct a linked list of buffered terms :

```
:- object buffer_node.

var term, next=null.

set_node_term(Term) :-
     term := Term.

set_node_next(Next) :-
     next := Next.

get_node_term(Term) :-
     Term := term.

get_node_next(Next) :-
     Next := next.

:- end_object buffer_node.
```

Exercises

3.1 Variants of the 'Hello World' programs.

3.1.1. Design a program which has the same functionality as program 'hello_world2', however, without using a non-logical variable as a counter and without using the predicate 'repeat'.

3.1.2. Design a progam in which a thread prints the text 'hello' several times, and another thread the text 'world'. The total number of output lines should be about 20. Analyse the difficulties if we require the number of output lines to be exactly 20, and suggest possible solutions.

4 DLP and Virtual Worlds

4.1 VRML EAI AND DLP

VRML EAI stands for the external authoring interface of the virtual reality modeling language. The EAI allows developers to control the contents of a VRML world. A VRML specification can be loaded in a browser by an application, such as a Java applet. As mentioned, DLP programs are compiled to Java classes, which can be used as Java applets in Web Browsers. DLP has been extended with a VRML EAI library, called (bcilib). For instance, in the DLP VRML library, the predicate $getSFVec3f(Object, Field, X, Y, Z)$ gets the $SFVec3f$ value (which consists of three float numbers X, Y, and Z) of the $Field$ of object $Object$, and $setSFVec3f(Object, Field, X, Y, Z)$ assigns X, Y, and Z values to the $SFVec3f$ $Field$ in $Object$, where $Object$ refers to an object in a 3D VRML world.

A DLP program can manipulate VRML virtual worlds by using the VRML EAI library predicates. Before we introduce more details of the DLP VRML predicates, we discuss first how to design VRML worlds that can be accessed from DLP programs.

4.2 DESIGN 3D VIRTUAL WORLDS FOR DLP

3D virtual worlds are implemented in VRML and the VRML EAI predicates refer to objects in the current virtual world which is loaded into a web browser. Each VRML file contains a scene graph hierarchy which consists of VRML nodes. Node statements may contain SFNode or MFNode field statements that contain other node statements. Objects which can be manipulated by DLP programs are nodes which have names defined by DEF statements in VRML. We use DEF to define object names of VRML nodes, and use DLP VRML predicates to manipulate the values of the fields of the defined nodes.

For example, the following specification defines a yellow cylinder in VRML.

```
#VRML V2.0 utf8

Transform {
translation 0 0 0
rotation 0 1 0 0
children [Shape {appearance Appearance {
             material Material {diffuseColor 1.0 1.0 0.0}}
             geometry Cylinder { height 2.0 radius 1.0}}]}
```

In order to manipulate the values of the field 'translation' and the field 'rotation' of the cylinder, we have to define a name for the node 'Transform' as follows.

```
#VRML V2.0 utf8

DEF cylinder Transform {
translation 0 0 0
rotation 0 1 0 0
children [Shape {appearance Appearance {
             material Material {diffuseColor 1.0 1.0 0.0}}
             geometry Cylinder { height 2.0 radius 1.0}}]}
```

Note that we do not define the name of fields. More generally, we can define a prototype of a partial hierarchy first, so we can use such a prototype to create multiple instances of 3D objects. For example, the specification of a bus whose position and orientation can be controlled by DLP programs, consists of the following three steps:

1. Design a prototype for a bus;
2. Instantiate the bus prototype;
3. Use DEF to define the name of the bus.

The corresponding VRML description is as follows:

```
PROTO Bus [
      exposedField SFVec3f translation 0 0 0
      exposedField SFRotation rotation 0 1 0 0]
```

```
{
Transform {
      translation IS translation
      rotation IS rotation
      children [
         ......
      ]}}

Transform {children [ DEF bus1 Bus {
                        translation -5 0 -1.5
                        rotation 0 1 0 0} ] }
```

Thus, $setSFVec3f(bus1, translation, 15, 0, -1.5)$ sets $bus1$ to a new position $\langle -15, 0, -1.5\rangle$. Similarly, we can use the same method to manipulate a viewpoint in a VRML world, namely, we define a viewpoint first and then use the viewpoint predicates

$$getViewpointPosition(Viewpoint, X, Y, Z)$$

and

$$setViewpointPosition(Viewpoint, X, Y, Z)$$

to set and get the values of a viewpoint that's defined as:

```
DEF myviewpoint Viewpoint { position -10 1.75 0
                            orientation 0 1 0 -1.5708
                            set_bind TRUE}
```

Since VRML allows multiple viewpoints in a virtual world, we can only get the field values of the initial viewpoint unless we use explicitly the corresponding set-predicates. In the example above, calling $getViewpointPosition(myviewpoint, X, Y, Z)$, gets the result $X = -10.0$, $Y = 1.75$, and $Z = 0.0$ no matter how the user changes the viewpoint in the virtual world. In most applications, we want to get the current position of the user's viewpoint which may be changed by using the keyboard or mouse for navigation. This can be done by adding a proximity sensor to the virtual world:

```
DEF proxSensor ProximitySensor  {center 0 0 0
                                 size 1000 1000 1000
                                 enabled TRUE
                                 isActive TRUE}
```

Here we specify a proximity sensor with the size $(1000 \times 1000 \times 1000)$. Of course, the parameters should be changed for different applications. Getting the position and the rotation of the proximity sensor means getting the current values of the user's viewpoint. Therefore, we can define the extended get-viewpoint predicates as:

```
getViewpointPositionEx(_,X,Y,Z) :-
            getSFVec3f(proxSensor,position,X,Y,Z).

getViewpointOrientationEx(_,X,Y,Z,R):-
            getSFRotation(proxSensor,orientation,X,Y,Z,R).
```

4.3 LOADING 3D VIRTUAL WORLDS

3D virtual worlds have to be loaded in a web browser for program manipulation, which can be done as follows:

1. Virtual worlds embedded in html files:

   ```
   <html>
   <title> DLP-BCI example 1</title>
   <body bgcolor=white>
   <embed src="vrml/root.wrl" width="100%" height="80%">
   <applet codebase="classes/" archive="dlpsys.jar"
   code="dlpbrow.class" width=600 height=300 MAYSCRIPT>
   <param name="mayscript" value="true">
   <param name="cols" value="60">
   <param name="rows" value="10">
   <param name="object" value="example1">
   </applet>

   </body>
   </html>
   ```

   Where *root.wrl* is the initial VRML file which is stored in the directory "*./vrml*", DLP classes directory is "*./classes*", archive="dlpsys.jar" means that the DLP system library is "dlpsys.jar", code="dlpbrow.class" says that DLP is initialized by the class 'dlpbrow', which creates a text area in the browser that serves as a message output window for DLP. The statements $< param\ name = "cols"\ value = "60" >$ and $< param\ name = "rows"\ value = "10" >$ define the columns and rows of the text area. If 'dlpbrow.class' is replaced by another class 'dlpcons.class', this message window does not appear in the browser and all messages will be forwarded to the browser's built-in Java console. MAYSCRIPT states that java scripts are enabled. $< param\ name = "object"\ value = "example1" >$ states that the DLP program which manipulates the virtual worlds is "example1".

2. Load virtual worlds for manipulation, by using the DLP VRML predicate *loadURL*(*URL*). For example, *loadURL*("*example*1.*wrl*") loads the virtual world "*example*1.*wrl*" into the web browser.

4.4 VRML PREDICATES

We call VRML predicates which can be used for getting values *get-predicates*, and predicates for setting values *set-predicates*. The DLP VRML library (bcilib) offers a complete collection of get/set-predicates for all field types in VRML. Here are some VRML predicates from the DLP VRML EAI library:

- Single Field Predicates
  - $getSFBool(+Object, +Field, -Bool)$
  - $setSFBool(+Object, +Field, +Bool)$
  - $getSFFloat(+Object, +Field, -Float)$
  - $setSFFloat(+Object, +Field, +Float)$
  - $getSFInt32(+Object, +Field, -Int32)$
  - $setSFInt32(+Object, +Field, +Int32)$
  - $getSFString(+Object, +Field, -Atom)$
  - $setSFString(+Object, +Field, +Atom)$
  - $getSFTime(+Object, +Field, -Time)$
  - $setSFTime(+Object, +Field, +Time)$
  - $getSFColor(+Object, +Field, -R, -G, -B)$
  - $setSFColor(+Object, +Field, +R, +G, +B)$
  - $getSFVec2f(+Object, +Field, -X, -Y)$
  - $setSFVec2f(+Object, +Field, +X, +Y)$
  - $getSFVec3f(+Object, +Field, -X, -Y, -Z)$
  - $setSFVec3f(+Object, +Field, +X, +Y, +Z)$

  For instance, $setSFVec3f(OtherViewpointNode, position, X, Y, Z)$ sets the position of another viewpoint node with values X, Y, Z, and $getSFVec3f(OtherViewpointNode, position, X, Y, Z)$ returns the position of the other viewpoint node in the variables X, Y, and Z respectively.

- Multi Field Predicates
  - $getMFFloat(+Object, +Field, -FloatList)$
  - $setMFFloat(+Object, +Field, +FloatList)$
  - $getMFInt32(+Object, +Field, -IntegerList)$
  - $setMFInt32(+Object, +Field, +IntegerList)$
  - $getMFString(+Object, +Field, -AtomList)$
  - $setMFString(+Object, +Field, +AtomList)$

  - $getMFColor(+Object, +Field, -RGBList)$
  - $setMFColor(+Object, +Field, +RGBList)$
    where $RGBList = [[R1, G1, B1], [R2, G2, B2], ...]$
  - $getMFVec2f(+Object, +Field, -XYList)$
  - $setMFVec2f(+Object, +Field, +XYList)$
  - $getMFVec3f(+Object, +Field, -XYZList)$
  - $setMFVec3f(+Object, +Field, +XYZList)$
    $XYList = [[X1, Y1], [X2, Y2], ...]$
    $XYZList = [[X1, Y1, Z1], [X2, Y2, Z2], ...]$

- Agent / Object Coordinates
  - $getPosition(+Object, -X, -Y, -Z)$
  - $setPosition(+Object, +X, +Y, +Z)$
  - $getRotation(+Object, -X, -Y, -Z, -R)$
  - $setRotation(+Object, +X, +Y, +Z, +R)$
  - $getViewpointPosition(+Viewpoint, -X, -Y, -Z)$
  - $setViewpointPosition(+Viewpoint, +X, +Y, +Z)$
  - $getViewpointOrientation(+Viewpoint, -X, -Y, -Z, -R)$
  - $setViewpointOrientation(+Viewpoint, +X, +Y, +Z, +R)$

In the VRML EAI library, the viewpoint predicates above manipulate a node $'Viewpoint'$, the default name of the viewpoint. However, in order to manipulate other viewpoints with non-default names, we can use generic predicates, like

$$getSFVec3f(Viewpoint, position, X, Y, Z)$$

and

$$setSFVec3f(Viewpoint, position, X, Y, Z).$$

Note that DLP programs which want to use VRML predicates to manipulate 3D objects should make the DLP VRML library *bcilib* available for it. This can be done by inheritance. Namely, the first line of DLP programs should be like:

```
:-object objectname : [bcilib].
```

4.5 MANIPULATING VRML WORLDS: EXAMPLES

In this section, we discuss how we can use DLP VRML predicates to control the contents of virtual worlds by a number of examples.

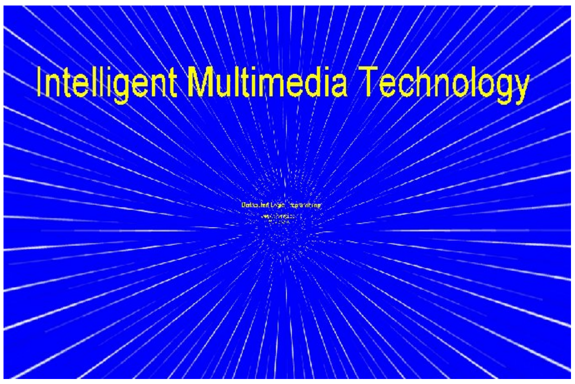


Fig. 4.1 Screenshot of title moving

4.5.1 title moving

In this subsection, we discuss several examples to achieve the effect of title moving, like those at the beginning of TV programs and movies. A screenshot is shown in Figure 4.5.1.

First we use VRML to design a virtual world in which there is a tunnel, i.e., a big cylinder with no bottom, and with the texture 'star1.jpg' as its background to simulate a star sky. Moreover, we also specify several texts which are located at the tunnel, and a music file 'mozart38.wav' to add sound. The 3D virtual world can be specified as follows in a file 'title0.wrl'.

```
#VRML V2.0 utf8

Background {skyColor [0.0 0.0 1.0] groundColor [0.0 0.0 1.0]}

DEF myViewpoint Viewpoint { position 0 3 6 orientation 0 1 0 0}

Transform { translation 0 3 0 rotation 1 0 0  1.57
      children [Transform {translation 0 0 0
      rotation 0 1 0 0
      children  Shape {appearance Appearance {
        texture ImageTexture { url "star1.jpg" repeatS TRUE repeatT TRUE }
        textureTransform TextureTransform {scale 30 10}
        material Material {diffuseColor 0.0 0.0 1.0
             emissiveColor 1.0 1.0 1.0}}
        geometry Cylinder { height 2000 radius 2.5   top TRUE bottom FALSE}}
        }]}

Transform {translation -2 3 50 scale 0.3 0.3 0.3
children [Shape {appearance Appearance {
```

```
         material Material {diffuseColor 1 1 0
         emissiveColor 1 0 0}}}
       Text { string "Intelligent Multimedia Technology" }] }


Transform {translation -2 3 30 scale 0.3 0.3 0.3
children [Shape {appearance Appearance {
          material Material {diffuseColor 1 1 0
          emissiveColor 1 0 0}}}
       Text { string "Distributed Logic Programming" }] }


Transform {translation -2 3 10 scale 0.3 0.3 0.3
children [Shape {appearance Appearance {
            material Material {diffuseColor 1 1 0
            emissiveColor 1 0 0}}}
       Text { string "VRML+JAVA+PROLOG" }] }


Transform {translation -2 3 -10 scale 0.3 0.3 0.3
children [Shape {appearance Appearance {
            material Material {diffuseColor 1 1 0
                emissiveColor 1 0 0}}}
       Text { string "Multimedia Authoring" }] }


Transform {translation -2 3 -30 scale 0.3 0.3 0.3
children [Shape {appearance Appearance {
            material Material {diffuseColor 1 1 0
              emissiveColor 1 0 0}}}
       Text { string "Thank You Very Much!" }] }




Sound {maxBack 2000 maxFront 2000
minFront 1000 minBack 1000
 intensity 20 source AudioClip { loop TRUE
url ["mozart38.wav"]}}
```

In order to achieve an animation effect, we design a DLP program so that the viewpoint can be changed regularly. Thus, we add a defined name 'myViewpoint' to define the viewpoint in the virtual world. The DLP program 'titlemove0.pl' is shown below:

```
:-object titlemove0: [bcilib].

var count = 3000.
var increment = 0.15.
var url='./title/title0.wrl'.


main :- text_area(Browser),
        set_output(Browser),
loadURL(url),
        sleep(3000),
        move_title(count).

move_title(0):-!.

move_title(N):- N>0,
        N1 is N-1,
        getSFVec3f(myViewpoint,position, X,Y,Z),
        Znew is Z - increment,
        setSFVec3f(myViewpoint, position,X,Y,Znew),
        sleep(150),
        move_title(N1).

:-end_object titlemove0.
```

In order to let the format function in DLP programs to send its output to the web browser, we use the following clauses to set a text area as the output of the program:

```
 text_area(Browser),
 set_output(Browser),
```

However, note that if we set code="dlpcons.class" in the html file, the message window is not enabled in the browser, and the two lines above should not be used in the program.

In the program, the viewpoint's position is gradually moving to the negative Z-direction by decreasing the Z value 0.15 meter each time. This is a simple example that shows how to manipulate 3D objects to achieve animation in virtual worlds.

4.5.2 Bus Driving

In this example, we design a bus driving program. Assume we have designed a bus in a VRML world, whose url is:

```
./street1.wrl
```

Driving the bus implies setting the position and rotation of the bus according to the user's viewpoint. Moreover, the position and rotation of the bus should be changed whenever the user's viewpoint changes. Here is the bus driving program, which first loads the url and moves the bus in front of the user, then starts the driving procedure.

```
:-object wasp2 : [bcilib].

var url = './street/street5.wrl'.

var timelimit = 300.

main :-
    text_area(Browser),
    set_output(Browser),
    format('Loading street1 from ~w~n', [url]),
    loadURL(url),
    format('The bus1 is going to jump
            in front of you in 5 seconds, ~ n'),
    format('then you can drive the bus
             for ~w seconds ~ n', [timelimit]),
    delay(5000),
    jump(bus1),
    drive(bus1,timelimit).

jump(Object) :-
    getSFVec3f(proxSensor,position,X,_Y,Z),
    Z1 is Z-5,
    setPosition(Object, X, 0.0 ,Z1).

drive(_,0):-!.

drive(Object,N) :- N>0, N1 is N-1,
    format('time left: ~w seconds~n', [N]),
    delay(1000),
    getSFVec3f(proxSensor,position,X,_Y,Z),
    getSFRotation(proxSensor,orientation,_X2,Y2,_Z2,R2),
    setPosition(Object,X, 0.0 ,Z),
    R3 is sign(Y2)*R2 + 1.571,
    setRotation(Object,0.0,1.0,0.0,R3),
    drive(Object,N1).
```

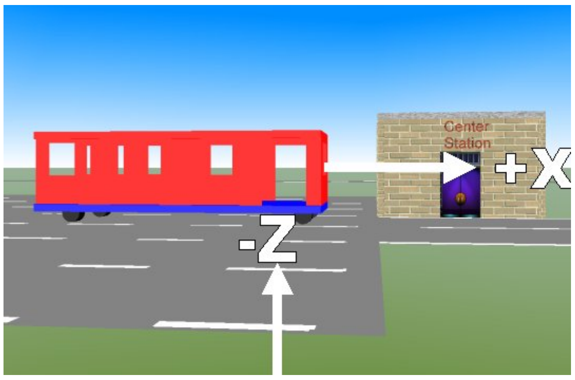


Fig. 4.2 Initial Situation of Bus Driving

```
:-end_object wasp2.
```

In the program, the *jump* rules will move the bus in front of the user, or more exactly, the viewpoint of the virtual world. The *drive* rules move the bus, i.e. the bus position and rotation are regularly updated according to the position and orientation of the user's viewpoint. The rotation of the bus is 90 degrees different from the orientation of the user's viewpoint. One of the difficulties in this program is to obtain a correct rotation value for the bus, based on the user's current viewpoint orientation. First we consider the simplest case, namely, the initial situation in which the user looks in the $-Z$ direction and the bus is positioned in the $+X$ direction, as shown in Figure 4.5.2. It is easy to see the relation between these two rotations if we have a look at Figure 4.5.2: $R3 = 1.571$. We want to obtain a general formula to calculate the new bus rotation based on the viewpoint's orientation (i.e. rotation), which is shown in Figure 4.5.2 and Figure 4.5.2. Figure 4.5.2 shows the situation in which the user turns to the right when navigating. Based on the right-hand system of rotation calculations in VRML, the value of the user's viewpoint orientation is $\langle 0.0, -1.0, 0.0, R2\rangle$. Thus, the bus rotation has to be set to $R3 = 1.57 - R2$. Figure 4.5.2 shows the situation in which the user turns to the left. The user's viewpoint orientation is $\langle 0.0, 1.0, 0.0, R2\rangle$. Therefore, $R3$ should be $R2 + 1.57$. The general formula which can be used to compute the new rotation of the bus based on the user's viewpoint orientation:

$$R3 = sign(Y2) * R2 + 1.57.$$

where $Y2$ is the Y-value of the user's viewpoint orientation.

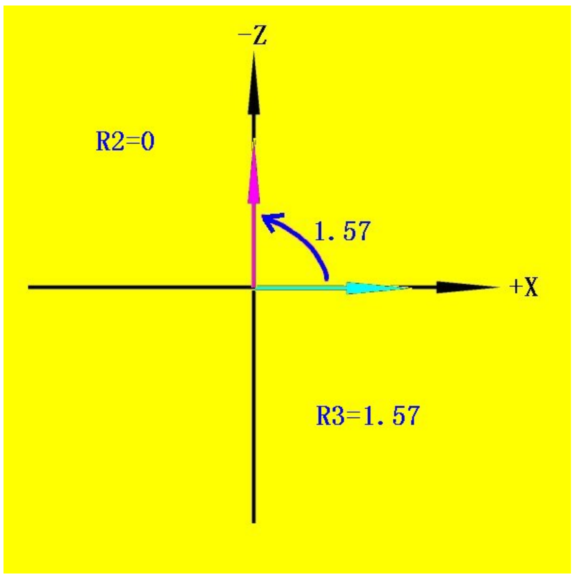


Fig. 4.3 The initial rotation values

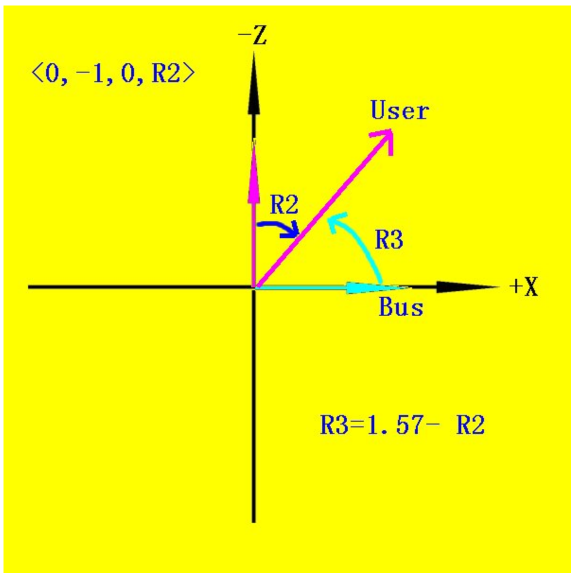


Fig. 4.4 User turns to the right

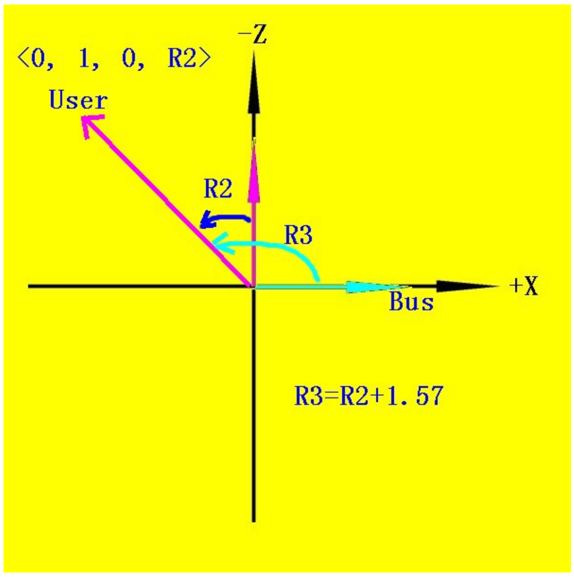


Fig. 4.5 User turns to the left

4.5.3 The Vector Library in DLP

In the bus example above, we can see that the calculation of the correct rotation values sometimes becomes a somewhat tricky task. We have to consider different situations to create a general formula which covers all cases for the calculation of the correct rotation value. Some knowledge of 3D graphics mathematics is helpful to solve this kind of problems.

DLP offers a vector library (*vectorlib*), which is useful for vector operations, in particular, for rotation calculations. Refer to a 3D graphics textbook for a general background of 3D mathematics. Several typical vectorlib predicates :

- $vector\_cross\_product(+vector(X1, Y1, Z1), +vector(X2, Y2, Z2),$ $-vector(X, Y, Z), -R)$: the vector $\langle X, Y, Z\rangle$, and the angle R are the cross product and the angle of the vector $\langle X1, Y1, Z1\rangle$ and the vector $\langle X2, Y2, Z2\rangle$, (based on the right-hand rule).

- $direction\_vector(+position(X1, Y1, Z1), +position(X2, Y2, Z2),$ $-vector(X, Y, Z))$: $\langle X, Y, Z\rangle$ is the vector with starting point $\langle X1, Y1, Z1\rangle$ and end point $\langle X2, Y2, Z2\rangle$.

- $vector\_rotation(vector(X1, Y1, Z1), rotation(X, Y, Z, R),$ $vector(X2, Y2, Z2))$: the resulting vector of a vector $\langle X1, Y1, Z1\rangle$ and a rotation $\langle X, Y, Z, R\rangle$ is $\langle X2, Y2, Z2\rangle$.

- $position\_rotation(position(X1, Y1, Z1), rotation(X, Y, Z, R),$ $position(X2, Y2, Z2))$: the resulting position of a position $\langle X1, Y1, Z1\rangle$ and a rotation $\langle X, Y, Z, R\rangle$ is $\langle X2, Y2, Z2\rangle$.

We can use the *vectorlib* predicates to compute the intended rotations in the bus example as follows:

```
:-object wasp2v : [bcilib,vectorlib].

var url = './street/street5.wrl'.

var timelimit = 300.

......

drive(_,0):-!.

drive(Object,N) :-
      N > 0,
      N1 is N-1,
      format('time left: ~w seconds~n', [N]),
      delay(1000),
      getSFVec3f(proxSensor,position,X,_Y,Z),
      getSFRotation(proxSensor,orientation,X2,Y2,Z2,R2),
      setPosition(Object,X, 0.0 ,Z),
      vector_rotation(vector(0,0,-1), rotation(X2,Y2,Z2,R2), vector(X3,Y3,Z3)),
      look_in_direction(Object,vector(1,0,0),vector(X3,Y3,Z3)),
      drive(Object,N1).

look_in_direction(Object, InitVector,DesVector):-
      vector_cross_product(InitVector,DesVector,vector(X,Y,Z),R),
      setRotation(Object,X,Y,Z,R).

:-end_object wasp2v.
```

In order to use the vector library, we add the *vectorlib* to the header of the program object. We define a new predicate *look_in_direction* which sets the object with an initial direction $InitVector$ to a destination direction $DesVector$ by using the predicate *vector_cross_product* in the vector library. We know that the user's initial orientation is oriented to the negative Z direction, namely, $\langle 0, 0, -1\rangle$ by default. After $rotation(X2, Y2, Z2, R2)$, the user looks in the direction $vector(X3, Y3, Z3)$, which can be calculated by the predicate *vector_rotation* in the vector library. The initial bus orientation is $vector(1, 0, 0)$. Therefore, during driving, the bus should keep the same orientation as the user by calling the predicate *look_in_direction*. Note that the predicate *look_in_direction* defined above can tell correct answers in most cases, however, not always. Consider the case in which $v_1 = \langle 0, 0, 1\rangle$, and $v_2 = \langle 0, 0, -1\rangle$. According to the definition of the cross product, $v_1 \times v_2$ results in the zero vector. However, we cannot use the zero vector as an axis of rotation. Try to improve the definition of the predicate *look_in_direction* to avoid the problem. We leave it as an exercise.

4.5.4 Ball Kicking

Consider a simple soccer game, in which the user is the only player in the game. If the user gets close enough to the soccer ball, the ball should move to a new position according to the position difference between the player and the ball. In the following program, we set the kickable distance to 2 meter. Namely, if the distance between the user and the ball is smaller than 2 meter, then the ball should be moved to a new position. We calculate a new position of the ball based on the position difference. If the user is at the left side of the ball, then the ball should move to the right; if the user is at the right of the ball, then the ball should move to the left. In the program, we set the move coefficient to 3: if the difference of the x parameter between the user and ball is $Xdif$, then the new position of ball of the x parameter should be increased by $3Xdif$. The same for the difference of the y parameter. Figure 4.5.4 shows the relation between the initial position of the ball and the destination position after kicking.

```
:-object wasp3 : [bcilib].

var url = './soccer/soccer1b.wrl'.

var timelimit = 300.


main :-
        text_area(Browser),
        set_output(Browser),

        format('Load the game ...~n'),
        loadURL(url),

        format('the game will start in 5 seconds,~n'),
        format('note that the total playing time
                period is ~w seconds,~n', [timelimit]),
        delay(5000),
        format('the game startup,~n'),
        play_ball(me, ball).
play_ball(Agent, Ball) :-
        -- timelimit,
        timelimit > 0, !,
        format('time left: ~w seconds~n', [timelimit]),
        delay(800),
        near_ball_then_kick(Agent, Ball),
        play_ball(Agent, Ball).

play_ball(_, _).


near_ball_then_kick(Agent, Ball):-
        getViewpointPositionEx(Agent,X,_Y,Z),
        getPosition(Ball,X1,Y1,Z1),
        Xdif is X1-X,
        Zdif is Z1-Z,
        Dist is sqrt(Xdif*Xdif + Zdif*Zdif),
        Dist < 5, !,
        X2 is Xdif*3,
        Z2 is Zdif*3,
        X3 is X2 + X1,
        Z3 is Z2 + Z1,
        setPosition(Ball,X3,Y1,Z3).

near_ball_then_kick(_, _).

getViewpointPositionEx(_,X,Y,Z) :-
        getSFVec3f(proxSensor,position,X,Y,Z).
getViewpointOrientationEx(_,X,Y,Z,R):-
        getSFRotation(proxSensor,orientation,X,Y,Z,R).

:-end_object wasp3.
```

4.5.5 Soccer Kicking with Goalkeeper

We can extend the example of ball kicking above by adding a goalkeeper to it, in such a way that the goalkeeper always looks at the ball and can check the position of the ball. If the ball is near the goalkeeper, say, within a distance of 3 meter, then the goalkeeper can move the ball to a new position. We use the vector library for the calculation of the rotation in the predicate *look_at_ball*, which simplifies the problem:

......

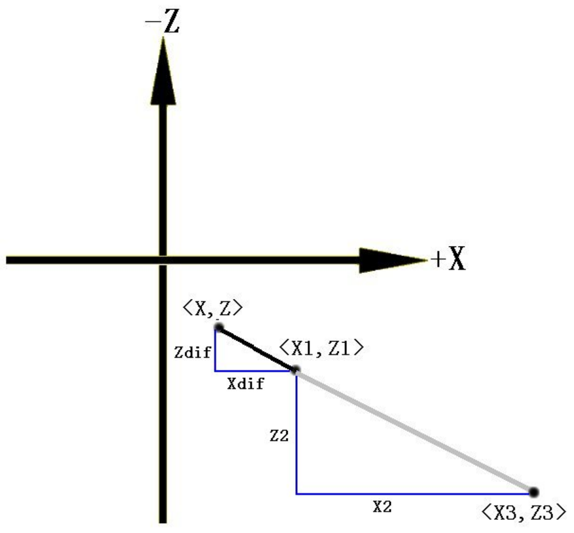


Fig. 4.6 kicking ball to a position

```
play_ball(Agent, Ball) :-
        -- timelimit,
        timelimit > 0, !,
        format('time left: ~w seconds~n', [timelimit]),
        delay(800),
        look_at_ball(goalKeeper1,Ball),
        near_ball_then_kick(Agent, Ball),
        play_ball(Agent, Ball).
......

near_ball_then_kick(Agent, Ball):-
        ......
        setPosition(Ball,X3,Y1,Z3),
        checkBallPosition(Ball,X3,Y1,Z3).

checkBallPosition(Ball, X, Y, Z):-
        getPosition(goalKeeper1,X1,_Y1,Z1),
        Xdif is X-X1,
        Zdif is Z-Z1,
        Dist is sqrt(Xdif*Xdif + Zdif*Zdif),
        Dist < 3, !,
        X2 is X - 5,
        setPosition(Ball,X2,Y,Z).

checkBallPosition(_,_,_,_).

look_at_ball(Player,Ball):-
      getPosition(Player,X,_Y,Z),
      getPosition(Ball, X1,_Y1,Z1),
      direction_vector(position(X,0,Z), position(X1,0,Z1), vector(X2,Y2,Z2)),
      look_in_direction(Player,vector(0,0,1),vector(X2,Y2,Z2)).
```

In the definiton of the predicate *look_at_ball*, we first obtain the positions of the player and the ball. We are not interested in the Y-parameters for the calculation of the rotations, because the player should not look down to the ball by rotating the whole body. This should be achieved by rotating the player's head. Based on the two positions, we can calculate the destination orientation of the player which can look at the ball by calling the predicate *direction_vector* in the vector library. We know that the initial orientation of an avatar is in the positive Z direction by default. Therefore, looking at the ball can be realized by calling the predicate *look_in_direction*.

Exercises

4.1 Variants of the title-moving.

4.1.1. Design a DLP program to implement a rolling text, namely, the text moves in the positive Y-direction.

4.1.2. Design a DLP program to implement moving titles in which the texts and the colors of the titles are changed regularly, using the following facts:

```
title_text(1, 'Intelligent Multimedia Technology', red):-!.
title_text(2, 'Moving Title Example', yellow):-!.
title_text(3, 'Changing Texts and Colors', green):-!.
......
```

4.2 Improve the example of ball kicking so that the soccer ball continuously moves to a new position. It should not simply jump to the new position.

4.3 Design a DLP program to control the bus moving so that it can move along a route which is defined by a set of facts.

4.4 Improve the example of bus driving so that the user can start and stop the bus engine. Namely, the bus moves only after the engine starts, and the bus does not move if the engine stops.

4.5 Change the definition of the predicate *look_in_direction* to avoid the zero vector as an axis of rotation.

Part II

3D Web Agents

5 Agents

The term "agents" are used by different people for different meanings. There is no real agreement on the definition of agent is. However, in general, *agents* are used to mean entities which can be programmed to conduct complex behavior. These complex behaviors can be understood as a sequence of actions which can be taken by agents. Therefore, an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors, according to [Russell and Norvig, 1995]. For software agents, the environment is the platform of software systems. The sensors are something which can get the information from the platform, whereas the effectors are something which can change the status of its environment. For web agents, i.e., software agents which are designed for web applications, the environment is World Wide Web. The sensors are something that can be used to get information from the Web, whereas the effectors are something that can be used to set information on the Web.

In the following, we will discuss the types of agents: simple reflex agents, decision-making agents, BDI agents, and extended BDI agents.

5.1 SIMPLE REFLEX AGENTS

[Russell and Norvig, 1995] offers a nice description on simple reflex agents. Simple reflex agents consist of only two components: sensors and effectors. The sensors perceive what the environment is like now, and then the effectors

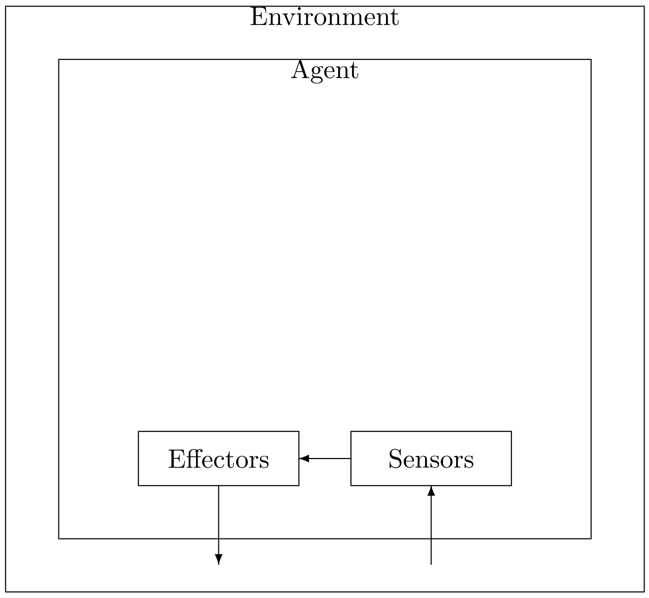


Fig. 5.1 Simple Reflex Agents

would take actions which correspond the agent's perception. The architecture of simple reflex agent is shown in figure 5.1.

Simple reflex agents have no components to reason about the perception and have no capability to know what the world is like under complicated circumstances. Moreover, simple reflex agents have no components to reason about actions and make complex decisions to take actions. They just simply perform stimulus-response behaviors.

5.2 DECISION-MAKING AGENTS

Different from simple reflex agents, decision-making agents have a component for decision making, which is shown in Figure 5.2. According to the traditional decision theories [Freach, 1988], rational agents would follow the following procedure to make decisions for their actions. The rational agents would consider a set of action alternatives (which are available with respect to the present situation). Each action alternative corresponds to an outcome state. The agents have preferences among these outcomes. The intended action alternative is the one which correspnds with the most preferred outcome.

Therefore, decision-making agents can be considered to have the following cognitive loop: sensing-thinking-acting. By *sensing*, the agents would know

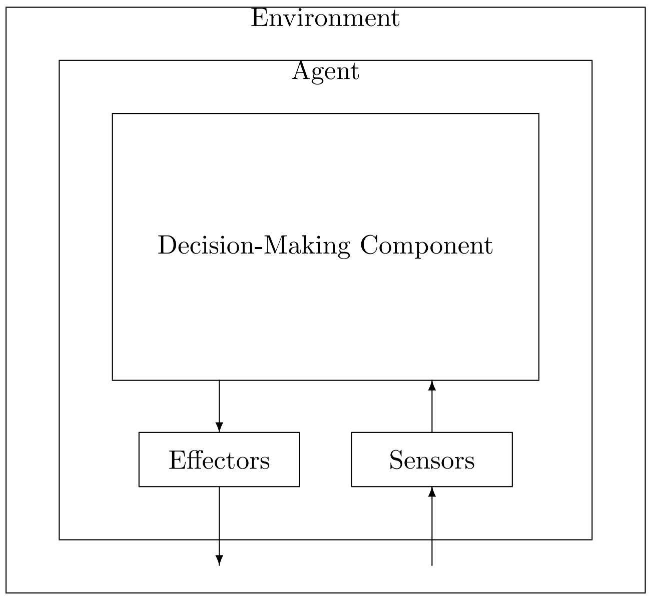


Fig. 5.2 Decision-making Agents

what the world is like now by using the sensors, thus the agents would know which action alternatives are available with respect to the current situation. By *thinking*, the agents would evaluate all of the outcomes which correspond considered alternatives, and then select the most preferred one. By *acting*, the agents would take the intended action by using their effectors.

5.3 BDI AGENTS

Decision-making agents still behave like complex stimulus-response agents. It is hard to say that they are really autonomous, for they lack a motivated component: goals, or desires. BDI Agents are based on cognitive models with the following mental attitudes: Beliefs, Desires, and Intentions. The architecture of a BDI agent is shown in Figure 5.3, which is motivated from both the traditional decision theories and the symbolic reasoning of cognitive science. [Rao and Georgeff, 1991, Rao and Georgeff, 1991] The BDI architecture actually is a general decision making model, for it is with symbolic reasoning. Therefore, BDI agents can be considered as an extension of decision making agents.

The BDI agents have the following cognitive procedure: the perceived information from the sensors are (partially) transferred into agents' beliefs. Thus,

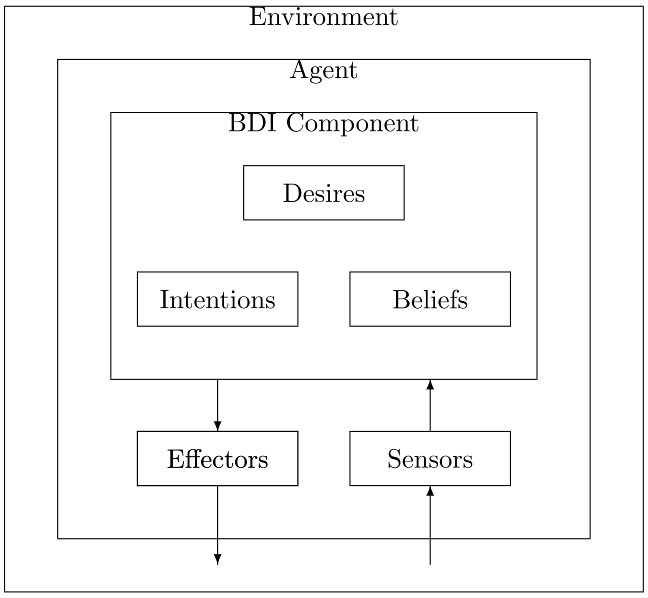


Fig. 5.3 Belief-Desire-Intention Agents

the agents are able to reason about the worlds, and know which action alternatives are available with respect to the current situations. Moreover, agents' beliefs have the information about the relations between the action alternatives and their outcomes. Furthermore, the agents' desires contain the information on the agents' preferences on these outcomes. The agents are able to deliberate about those actions, and decide which actions would be intended. The intended actions consist of the agents' intentions. Then, the agents are able to use their effectors to take the intended actions.

5.4 EXTENDED BDI AGENTS

In [Bell, 1995], Bell proposes an extened BDI architecture of rational agents, which is shown in the figure 5.4. According to Bell's architecture, a reasoning agent is situated in the real world and consists of two connected modules; a high-level (symbolic) reasoning system (or "mind") and a low-level (procedural) action system (or "body"). The symbolic reasoning system is composed of a module for theoretical reasoning and a module for practical reasoning.

The theoretical reasoning module represents the agent's beliefs, knowledge of, and reasoning about the world. The reasoning done by this module includes standard deductive reasoning (traditionally called "theorem proving")

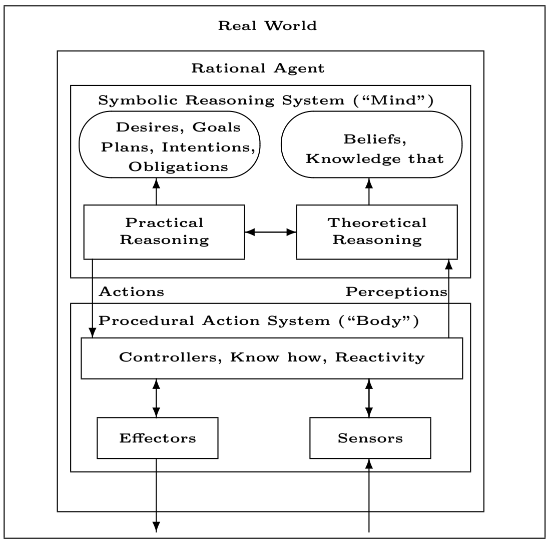

Real World
Rational Agent
Symbolic Reasoning System ("Mind")
Desires, Goals Plans, Intentions, Obligations
Beliefs, Knowledge that
Practical Reasoning
Theoretical Reasoning
Actions
Perceptions
Procedural Action System ("Body")
Controllers, Know how, Reactivity
Effectors
Sensors

Fig. 5.4 Extended-BDI Agents

as well as inductive, abductive, and probabilistic reasoning. It also performs database-type operations (lookup, update, addition, revision, deletion). The practical reasoning module represents the agent's reasoning about what it should do; and is discussed further in [Bell, 1995]. The planning system generates plans to achieve the agent's goals, schedules the resulting actions for execution, passes them to the procedural action system, and monitors their execution by it. The procedural action system (or "body") may consist of controllers, motion systems and perception systems. This component represents the agent's physical capacities and skills ("know how"). The controllers control and monitor the sensors and the effectors and mediate between them and the reasoning systems. They receive high-level action commands from the practical reasoning system, expand these to the appropriate level of detail, pass them to the effectors systems for execution, and perform low-level monitoring on them. They also pass perceptions (symbolic descriptions of the environment based on feedback from the sensors) to the theoretical reasoning system. The "mind" and "body" function as co-routines each of which is more or less active depending on the environment, the processing (reasoning) resources available, and the tasks at hand. This allows the agent to form long-term strategic plans to execute them and to react to events.

5.5 MAIN FEATURES OF INTELLIGENT AGENTS

As mentioned above, there is no real agreement on the definition of agents. However, in this book, we adopt the agent characterization given in [Jennings et al., 1998]. Namely, agents have the following main features:

- *autonomy*: the system should be able to act without the direct intervention of others and should have control over its own actions and internal state.
- *responsiveness*: Agents should perceive their environment and respond in a timely fashion to changes that occur in it.
- *pro-activity*: Agents should not simply act in response to their environment. They should be able to exhibit opportunistic, goal-directed behavior and make the initiative where appropriate.
- *socialness*: Agents should be able to interact, when they deem appropriate, with other artificial agents and humans in order to complete their own problem solving and to help others with their activities.

These characteristics, obviously, are strongly metaphorical, and are meant to stress the need for flexible, adaptive approaches to provide assistance to the user.

6 Web Agents

6.1 INTRODUCTION

Currently there are many research and development efforts going on with respect to agent technology on the Web. Many types of web agents have been proposed in recent years, which range from domain-dependent agents, like e-commerce agents[Liu and Ye, 2001], information gathering agents[Knoblock, 1997, Klusch, 1999, Klusch et al, 2003], and intelligent virtual agents[de Antonio, et al., 2001], to function-dependent agents, like negotiation agents[Beer, 1999], co-operating agents[Roth, 1999], problem solving agents. The natural questions related to that phenomenon are: what are the relations among so many different types of web agents? Are they redundant or overlapping? Is there a taxonomy to classify them?

In [Huang et al., 2000], a taxonomy of web agents has been proposed, which encompasses agents that provide a text-based interface to, for example, information retrieval services, as well as avatar-embodied guides that help visitors to navigate in virtual environments. The taxonomy must be regarded as an instrument to delineate targets for research and the realization of prototype applications that demonstrate the usefulness of agent-based intelligence on the Web. In addition, the agent-taxonomy can be deployed to establish the implications that particular target applications have on the software architecture and computational resources.

6.2 INTELLIGENT SERVICES ON THE WEB – AN OVERVIEW

Due to the exponential growth of the Web and the information it provides, finding relevant information has become more and more difficult. In particular, browsing is in many cases no longer appropriate for the user searching for specific information. It is our view that, in the near future, access to the Web will increasingly be mediated by intelligent helper applications, software agents, that assist the user in finding relevant and interesting information. As testified by a large body of literature, including [Caglayan, 1997, Cheong, 1996, Negroponte, 1995, Maes, 1997], this view is shared by many authors and developers.

During the first years of the Web it was feasible to create a fairly complete subject index of the whole World Wide Web. Yahoo was probably the most prominent one. Besides such subject indexes there were search engines that provided full text search on virtually the whole Web. Neither approach has proved sufficient. In order to better serve users, Web "portals" have emerged, combining searching and browsing, and most indexes and search engines now offer this combined access to information. Moreover, several sites are beginning to offer *personalized* portals that can be customized by the user and that may even adapt to the user's interests automatically. It is this personalization that is the step that moves portals from being general information services towards personal agents that point users towards relevant information and that hide stuff that is outside the user's field of interest.

A similar evolution exists in other areas like e-commerce. Personalized agents offer buying suggestions. Some can even automatically buy products at the best price or other conditions. Adaptive learning systems help users in studying a subject of a course by showing which pages to read in which order in order to reach that objective. More about adaptive systems can be found in [Brusilovsky, 1996].

User assistance may involve intelligent desktop support (such as the infamous paper clip), information filtering and storage (to augment search engines), the delegation of tasks (such as responding to email, or setting a shared agenda), and the maintenance of user preferences and profiles (for example to make a proper selection of musical material).

Clearly all these applications benefit from a metaphor that stresses some kind of situated intelligence, that is software processes that can interact with other such processes (and humans) and that may be adapted to the individual user's needs and preferences.

Many of the Internet/Web applications involving agents, make use of the agent-metaphor only. In our opinion, such applications could benefit from employing actual agent technology, instead of merely using the metaphor. In Section 6.5 we will briefly outline what needs to be done to make such technology available to internet-based services. Here we will merely indicate the possible scope of applications which can deploy agent technology on the Web.

In [Jennings et al., 1998] an overview is given of agent research and applications developed with agent technology. When we restrict ourselves to Web-related agent applications, we encounter information management, e-commerce and entertainment as the most dominant application areas.

We first distinguish between a variety of tasks in this application domain where agents could be deployed:

- information retrieval – to collect and filter information
- information presentation – to present the information in an understandable way
- intelligent navigation – to enable the selection of relevant information

In order to realize such functionality in a Web-based context, a framework of agent technology is needed. Apart from the programming aspects, such a framework must allow for a declarative description of agent-behavior based on user preferences, including the cooperative behavior of software agents, that is how they interact with other agents.

In Section 6.5 we will indicate the research issues involved in developing such a framework. In particular, we will concentrate on the realization of such agents in the context of virtual environments, which (as we will argue in Section 6.4) imposes additional constraints and leads to additional requirements such a framework must satisfy.

6.3 AGENTS IN VIRTUAL ENVIRONMENTS

Virtual environments, in particular 3D virtual worlds, are becoming a realistic architecture for providing services on the Web. 3D worlds offer more possibilities for presenting large information spaces in an attractive way than 2D presentations. As a consequence, agent-mediated services need an adequate representation in such environments. In this section, we will establish what possible function agents have in virtual environments, by briefly looking at the history of agent manifestations in such an environment, and by exploring how agent technology can be deployed in virtual multi-user communities for helping visitors in information retrieval tasks and navigation. In this section we will explore how agent technology may play a role in general-purpose virtual environments.

6.3.1 Agents in 3D Community Server

In many (3D) virtual environments, when entering the world, or a particular space, the visitor is welcomed by some 'officer' or guide. In most cases the welcome is reflected in the chat panel, announcing the new visitor to the other visitors. Examples of such guides can be found in AlphaWorld [AlphaWorld, 1999] and environments created with Blaxxun Community Server technology [Blaxxun, 2000]. The Blaxxun Community Server is a platform for creating virtual environments. Apart from the server, which maintains information about the state of the community and the whereabouts of its visitors, virtual worlds are defined using VRML-defined 3D worlds.

When entering the community, members or visitors may choose for 3D chat mode, which means that they find themselves in a virtual environment in 3D space, surrounded by objects, and possibly by other visitors and the agent or guide that belongs to that particular space.

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 indeed does not surpass that of simple bots and we consider the term agent to be less appropriate. Nevertheless, it is interesting to look in more detail how the behavior of Blaxxun agents can be defined, and to explore how we may enhance the behavior of these agents in such a way that we can speak of 'intelligent agents'.

The Blaxxun agent can be programmed using an event-based scripting language. An agent or bot script consists of four sections:

1. robot definition – to define the robots nickname and choose a graphical representation
2. reaction definition – to define patterns of events and actions
3. session definition – to indicate to which scenes and which server the bot belongs
4. start session statements – for initializing specific sessions

The robot definition section allows for a rich repertoire of attributes, including roles, experience, interests and even a business card.

Reactions are defined by indicating an event or event pattern and a corresponding action. The Blaxxun platform offers a rich set of built-in events, which includes both events coming from the chat channel as well as 3D events which occur in the associated 3D world. Actions include simple utterings, which appear in the chat channel as text, as well as movements of the avatar, which may beam to any desired spot in the world. Also timed actions are possible, which occur in response to a timer event, so that the bot or guide can periodically undertake some action.

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, simply because event-action patterns are not powerful enough to program complex behavior that requires maintaining information over a period of time.

Fortunately, there is a way out. That is, the Blaxxun agent platform also offers a C++ API which allows the developer to augment the functionality of a simple bot. This puts the burden on the shoulder of the developers, who are faced with the task of creating their own platform for intelligent agents in virtual environments. Still, it offers a presentation and interaction basis that may make the development of a complete "agent support system" somewhat easier.

6.3.2 Information retrieval in virtual environments

When the question arises what possible use agents may have in virtual environments, we may think of information retrieval along the lines as indicated in section 6.2 as one possible and interesting example.

Basically, the information retrieval problem in virtual environments consists of detecting containment relations between worlds, and gathering information about the objects in that world, which includes geometric shapes as well as text, image and sound objects.

We may approach this problem in three radically different ways:

- using black-box feature detectors
- meta information
- incremental knowledge gathering

Black-box feature detectors Feature detectors are a common means in multimedia database systems to obtain information about particular media items, such as images and audio in order to establish a measure of similarity between such media objects.

Despite problems with respect to selecting a relevant set of features and assigning proper weights in determining the similarity of media objects, feature detectors are effective when dealing with binary encoded material such as images and audio. When considering the material virtual environments are made of, however, we must note that much of the contents of virtual worlds is programmed content, that is the result of algorithmically generated geometries. Apart from that, it is hard to see how to obtain information about a world from a purely geometric description. In our opinion, feature detectors may be useful to inspect parts of the content of virtual worlds, in particular image or movie textures and sound objects. This information may then be used to augment the information obtained about a virtual world using any of the alternative approaches.

Meta information Meta information is an effective way to allow searching for media content. Instead of describing content by intrinsic features of the media object, meta information is used to indicate properties of media objects that can often not be extracted from the media object itself. See [Watson, 1996]. As an example consider an audio recording of an aria. Information concerning the composer and singer is most conveniently stored as meta data, although it would in principle not be impossible to extract such information algorithmically.

Meta information in virtual environments would mean additional descriptive information that describe the content or context of a particular world. In VRML/X3D, we can use the WorldInfo node to provide such information. Also it is conceivable to generate such information dynamically, based on user ratings.

Incremental knowledge gathering As noted before, a problem with virtual environments is that they are often generated in an algorithmic fashion, using information stored in a database. Moreover, the geometric information contained in a VRML/X3D file is not what makes the world interesting to a user. Rather, the interactive facilities provided by such worlds, and the navigation behavior of (other) users are what we are interested in. So, instead of a static analysis of virtual worlds, we believe that a more incremental approach is desirable, an approach where the navigational behavior of the individual user contributes to what we may call the collective awareness of a world. In other words, when we record the navigational behavior of a user and store the features of the world that are relevant to this user in a database, other users may profit from this information and query the database to be guided to the locations of interest.

6.3.3 Adaptive environments – presenting information

Given support for information retrieval in virtual environments, the question arises how to present this information to the user. One possibility, although not very appealing, would be a list of alternatives. For example, when the user asks for locations where there is information about, let's say patterns in object-oriented programming, a list of links to such information sources may be presented, as is common for most search engines. Then selecting a link might result in being ported to a world containing such information or any other location on the Web.

Alternatively, and more appealing, one may generate a virtual world that contains the answers to the query located in 3D. A clear advantage of generating a world in response to a query is that the topological properties of the world can be employed to sort the information according to some criterion of relevance.

Clearly, we have to distinguish between queries that concern information that is present within a world, queries that concern related worlds (that is worlds that are accessible from the world from which the query is posed), and queries that concern material that lies outside of the world (such as the request to locate paintings of the 17th century painted in a particular style). In the first two cases a direct transfer to a location of interest might take place, provided the location is available. In the latter case, we can only resort to generating a world based on the information that is found. Note that taking this approach to the extreme would mean to generate worlds based on the interest of visitors from the start.

Whatever approach is chosen for presenting the results from a query, the amount of information presented to the user is usually overwhelming. So, in addition to a world which contains that information, it is desirable to have a guide or agent that shows the user around, that is gives information about the results, indicates its relevance and transports the user (in a quite literal sense) to the area of interest. Such an agent could be augmented with information concerning the user's interests, (knowledge) background and preferences.

6.3.4 Discussion

Despite the availability of virtual worlds, we must remark that the notion of virtual worlds itself is somewhat elusive. Restricting ourselves to (3D) multi-user virtual communities we may distinguish between (e)commercial environments, educational environments or environments offering a mixture of commerce and entertainment.

With regard to these environments we observe that

- many environments are rather static,
- most worlds are far less immersive than desirable,
- interaction facilities are generally poor, and
- navigation is often a problematic issue.

Coming back to the question what use agents might have in virtual environments, we may take this list to develop a program of research to tackle the issues involved.

Dynamically generated worlds Virtual worlds may be generated based on a user's profile, to contain the information that corresponds with that profile in accord with the style preferences indicated by the user.

Immersiveness The lack of immersiveness is to a large part due to for example chat windows and property menus, in other words 2D gadgets that are needed for customization and interaction. Although agent technology has no contribution to make in this respect, it is conceivable that the user's avatar is made more intelligent and is augmented with features that makes the additional gadgets obsolete, for example text balloons for displaying chat and instruments for customization.

Interaction Experience shows that navigation and interaction are difficult (for the average user), simply because 3D interfaces are more complex than their 2D counterparts. To aid the user in learning how to use a world effectively, we can think of helper agents that show the user how interaction and the manipulation of objects takes place.

Navigation Due to the inherent visual complexity of (3D) virtual environments, navigation remains a problematic issue. Fixed viewpoint animations do exist, but are not sufficient to direct the user to locations of interest. Instead, we should consider navigation by query, possibly augmented with intelligent guides that accompany the visitor on touring the world.

6.4 A TAXONOMY OF WEB AGENTS

Different aspects of applications and configurations suggest different types of web agents. In this section, a taxonomy of web agents is discussed. With respect to the choice of dimensions, we must remark that even though our taxonomy has a bias towards the deployment of agents in virtual environments, the taxonomy is valid with respect to Web-based services in general.

We consider the following three dimensions of web agent types.

1. 2D versus 3D
   A 2D web agent is one which is aware of HTTP, file, and FTP protocols, whereas a 3D web agent is one which is aware of not only these protocols, but also virtual reality protocols. Typical 2D web agents provide a text-based interface, for example, information retrieval services. Typical 3D web agents are avatar-embodied guides that help visitors to navigate in virtual environments.

2. Client versus server
   As the names imply, a client web agent is on the client side, whereas a server web agent is on the server side. A typical client web agent can serve as a personal information assistant. A typical server web agent can serve as the front-end of web servers to offer information more intelligently.

3. Singularity versus multiplicity
   As the names imply, a single web agent would not consider the interface with other web agents, whereas multiple web agents would interact with each other.

There are different types of web agents based on these three dimensions, which consists of a complexity lattice of web agent types, which is shown in Figure 6.1. 3D-server-multiple-agent is on the top of the complexity hierarchy, whereas a 2D-client-single-agent is at the bottom. These dimensions are not exhaustive. In particular, in this book, we do not consider the di-

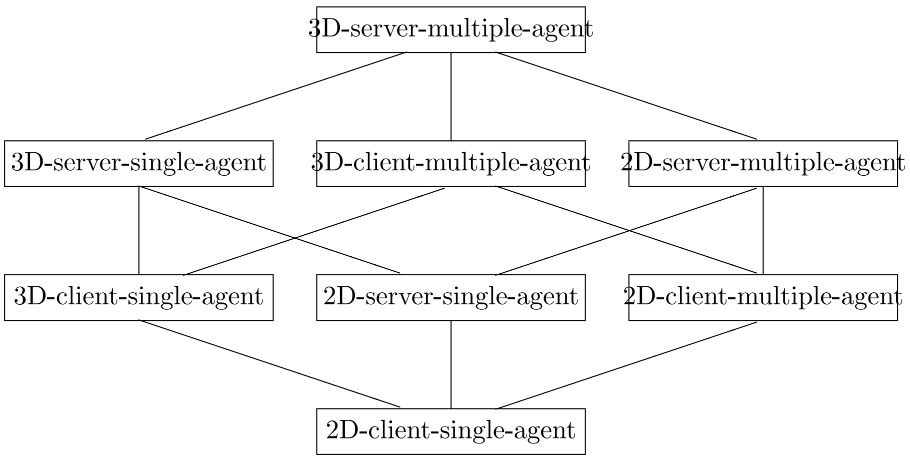


Fig. 6.1 Lattice of Web Agents

mension "stationariness versus mobility"[1]. However, the combinations based on the dimension "stationariness versus mobility" can be generalized accordingly, based on the current taxonomy. All the dimensions we consider for the taxonomy are directly web-dependent. The dimension "2D-3D" specifies the internet protocols agents have to be aware of, the dimension "client-server" is concerned with internet service modes agents have to offer, and the dimension "singularity-multiplicity" involves agent communication languages. We do not consider the functional dimension which are not directly relevant to the Web, like cooperating agents, problem solving agents, and negotiation agents. We do not discuss the dimensions which are domain dependent, like expertise seeking agents, and e-commerce agents. However, our taxonomy does cover most types of those agents. They are either 2D agent or 3D agent, either client agent or server agent, either single agent or multiple agent, or some of their combinations. Different types of web agents suggest different types of interaction modes with users and web servers. We discuss them in detail in the following section.

[1]A stationary agent is one that executes only upon the agent platform where it begins executing and is reliant upon it, whereas a mobile agent is one that is not reliant upon the agent platform where it began executing and can subsequently transport itself between agent platforms[FIPA, 1999].

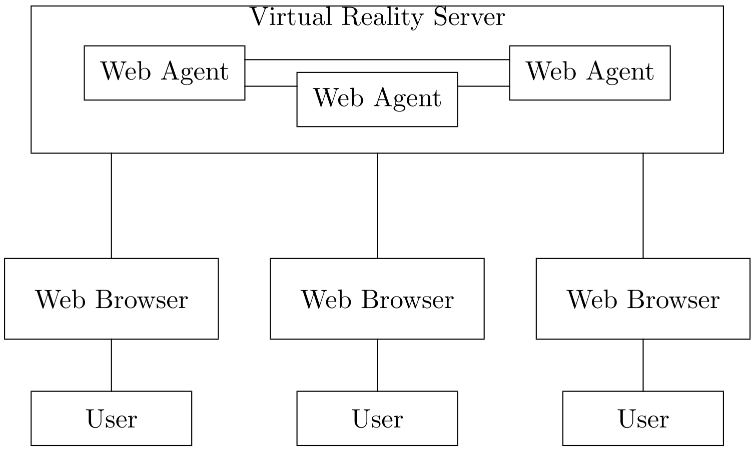


Fig. 6.2 Configuration of 3D-server-multiple-agent

6.4.1 3D-Server-Multiple-Agent

The configuration of 3D-server-multiple-agent on the Web is shown in Figure 6.4.1. The agents are part of virtual reality servers, more exactly, virtual community servers, for they interact with multiple users and agents. Users communicate with virtual reality servers via web browsers. The agents interact with each other in the virtual reality server, and communicate with web users. As agents in virtual reality, they are normally embodied by their avatars.

As discussed in Section 6.3, these kinds of web agents have a great deal of application potentials, which range from avatar-embodied guides that help visitors in information retrieval tasks and navigation in virtual environments, e-commerce shopping assistant agents in 3D-virtual shops, to intelligent agents in multi-player computer games.

6.4.2 3D-Server-Single-Agent

3D-server-single-agents and their relationship with servers and users are shown in Figure 6.3, where agents serve as the front-end of the servers. Users do not directly communicate with servers, but with the intelligent web agent which

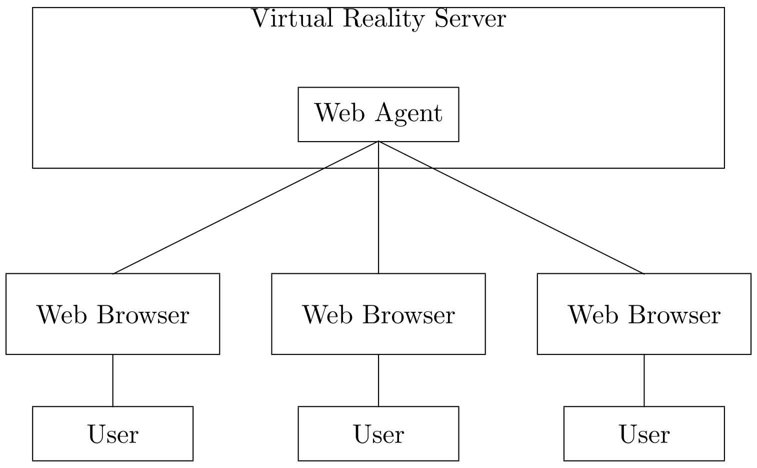


Fig. 6.3 Configuration of 3D-server-single-agent

is located on the server side. In other words, the agent may be regarded as a portal through which the server is accessed.

That kind of web agents can intelligently offer or create personalized virtual environments for users, based on users' queries or requests.

6.4.3 3D-Client-Multiple-Agent

The relation among 3D-client-multiple-agent, virtual reality server, and web users is shown in Figure 6.4. The agents are located at the client side, normally serve as users' personal information assistants. Virtual reality servers may have their own web agents. Web users communicate with servers or other users (or agents) either via web browsers or directly via web agents. Other users may also have their own web agents, which are omitted in the figure for reasons of simplicity. This type of web agent is particularly useful in the 3D chat arena, in which the agents can serve as intelligent personal assistant to help users in information gathering, and interface with other intelligent web agents.

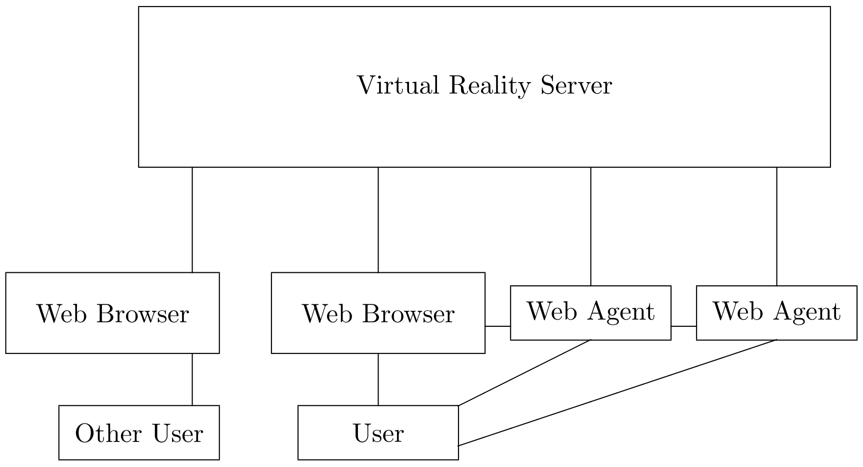


Fig. 6.4 Configuration of 3D-client-multiple-agent

6.4.4 2D-Server-Multiple-Agent

The configuration of 2D-server-multiple-agents is shown in Figure 6.5. Similar to its 3D counterparts, these kinds of web agents are a component of web servers. They can offer users a text-based or image-based interface for navigation. One useful application is to serve as an intelligent e-commerce sales-bot to offer more user-friendly and more intelligent services. This type of web agents can display intelligent behavior in text-based internet games, like MUD games [MUD, 2000].

6.4.5 3D-Client-Single-Agent

3D-client-single-agent is a simplified 3D-client-multiple-agent. Its relation with other web components is like the situation shown in Figure 6.4, except that there is only one agent available. Similar to their multiple-agent counterparts, this type of web agent normally serve as a personal information assistant in virtual environments.

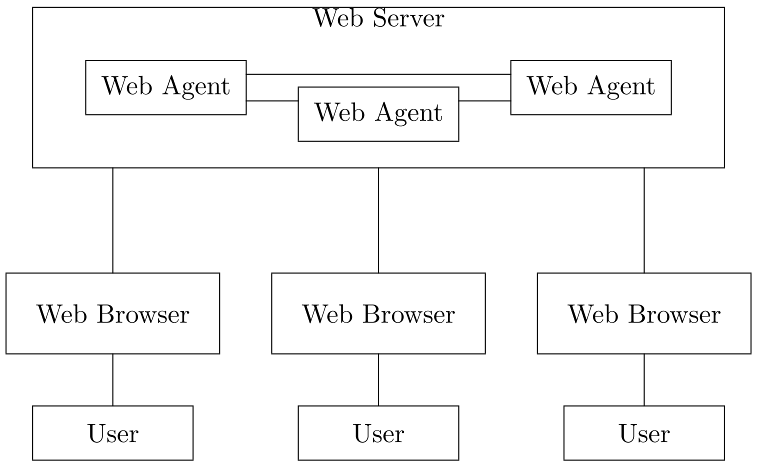


Fig. 6.5 Configuration of 2D-server-multiple-agent

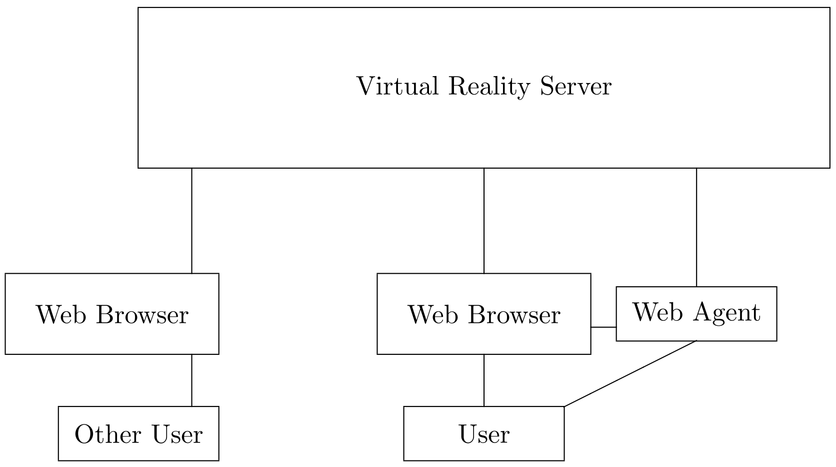


Fig. 6.6 Configuration of 3D-client-single-agent

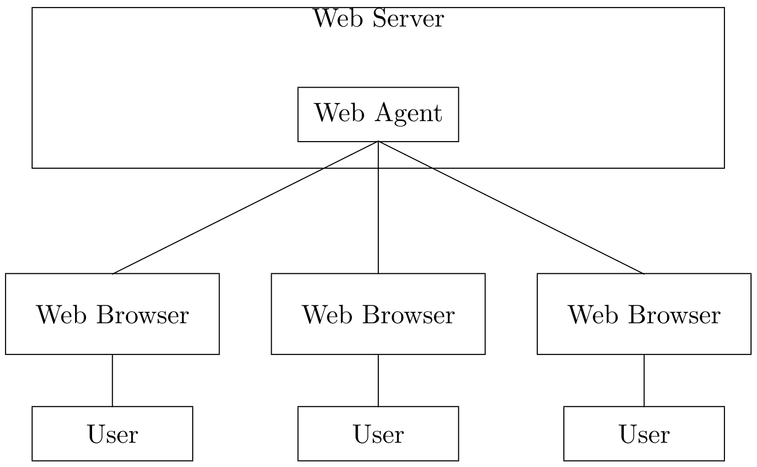


Fig. 6.7 Configuration of 2D-server-single-agent

6.4.6 2D-Server-Single-Agent

2D-server-single-agent, which is a restriction of the 3D-server-single agent, could alternatively be called *intelligent web server*. These kinds of agents normally reside at the front end of an http server to offer users more personalized messages. They can serve more efficiently and more intelligently if they are supported by powerful inference engines.

6.4.7 2D-Client-Multiple-Agent

The configuration of 2D-client-multiple-agent is analogous to the 3D-client-multiple-agent which is shown in Figure 6.4, apart from the more powerful user interface and display properties of the latter. One useful application of them is to act as an internet chat agent. They help users to record and analysis chat messages, to gather information and interact with other users or agents when requested.

6.4.8 2D-Client-Single-Agent

2D-client-single-agents may simply be called personal (text-based)-information assistants. Its relation with servers and users is analogous to the situation

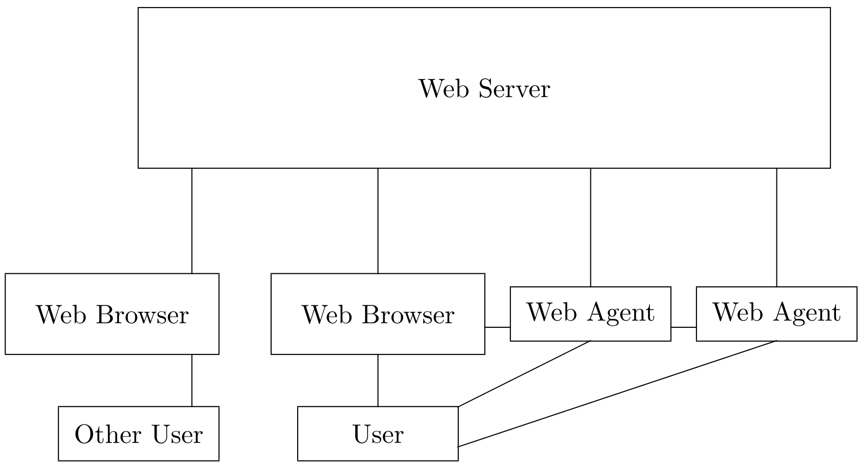


Fig. 6.8 Configuration of 2D-client-multiple-agent

shown in Figure 6.4, except that there is only one agent, which only has a 2D manifestation. Although they are among the simplest form of web agents, as they lack the facilities to interact with other web agents, they can still fulfill many interesting and useful tasks, like gathering information on request.

6.5 RESEARCH ISSUES

With the taxonomy as outlined in the previous section, we are able to indicate the research issues involved in developing an intelligent web agent (IWA).

The primary requirements IWA must fulfill may be summarized as follows:

- autonomous and on-demand search capabilities
- (user- and system) modifiable user preferences, and
- (multimedia) presentation facilities

Evidently, with regard to our taxonomy, we have a choice of where to put the functionality, i.e. either on the client-side or server-side, whether we support a 2D (text) interface or an embodiment in 3D, and whether we realize IWA as a single agent or as a multi-agent system. Each of these choices has ramifications

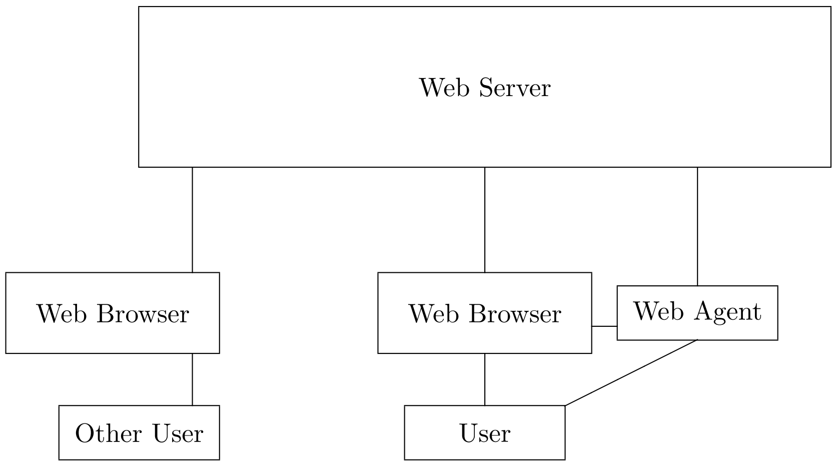


Fig. 6.9 Configuration of 2D-client-single-agent

with respect to the effort of developing the application as well as the resources needed for deploying it.

Nevertheless, irrespective of a particular choice, the following aspects play a role in developing Web-aware agents, or for that matter, a framework for developing such agents:

- generic agent software components
- user interface components for managing agents
- communication primitives for agent communication

Where we position the application in our lattice of possible agent applications will, however, affect the complexity of the components mentioned above. For example, developing 3D-client-multiple agents will require advanced interface components, as well as powerful communication primitives.

To deal with the complexities involved in developing a range of agent applications, the language DLP is designed to aim at providing support on the level of the software architecture. Architectural support for agent applications must, in our view, include

- an agent-based programming language
- a high-level API for Web-aware applications

	type	3D platform
lexicon search	2D-*-server	agent
information gathering	2D-*-server	agent/server
information presentation	3D-*-*	client/server
information visualisation	3D-*-client	client/server
navigation	3D-single-server	agent/server
brokering	*-*-*	agent

Table 6.1 Demonstrators of agent technology

- an object-based framework for distributed agent applications
- tools for agent-based information retrieval and management

Actually, in order to offer more than the agent metaphor as a guideline for developing applications, DLP is designed to deploy that kind of the agent-oriented programming languages supporting the primitives needed for expressing both the reasoning and the domain or context in which this reasoning takes place.

Demonstrators for agents in virtual environments Developing agent-technology for virtual environments is a challenging task, but probably an endeavor that is doomed to fail when one does not focus on target applications or demonstrators that deliver some useful service that can be tested in practice. To conclude our discussion of research issues, we will briefly present a list of possible target applications that may serve as the agenda for further research.

Figure 6.1 presents a number of possible applications of agent-technology in virtual environments, with an indication of the agent types involved. An asterisk (*) is used as a wildcard to indicate where each of the alternatives is possible. As an example, for information visualisation, we will very likely need an agent that operates at the client-side, even though it uses other agents that operate at the server-side. As another example, to define a brokering service that allows visitors to come in contact with another visitor or service, any choice in the agent lattice given in Section 6.4 is conceivable. However, when providing navigation services, the range of alternatives is limited to one, namely 3D-single-server agents, although also in this case a multi-agent solution would be conceivable as well.

To realize such a target applications, significant endeavor is required to model the services that are provided and for embedding these services in an intelligent multimedia platform in a more or less seamless way. As indicated in Section 6.3.1, we strive for the realization of immersive virtual environments, which means that the structure and presentation of a virtual world must not be disrupted because of the availability of some service, no matter how useful it may be. To find a seamless integration between intelligent (agent-based) services and 3D virtual worlds may, in other words, be taken as the presupposition of our research, if not a research goal in itself.

6.6 CONCLUSIONS

In this chapter we have discussed a taxonomy of Web agents. The taxonomy was geared towards the deployment of agents in virtual environments. Particularly, the agent-lattice resulting from the taxonomy implies an ordering on conceivable agents along the dimensions 2D or 3D presentation, single or multi-agent support, and client or server-side residence. This taxonomy has been applied to indicate research issues and the ramifications target applications have with respect to a platform for realizing intelligent services in a virtual environment.

7

3D Web Agents in DLP

7.1 IMPLEMENTATION OF 3D WEB AGENTS

In this chapter, we show how DLP can be used to implement 3D web agents. We discuss the problem by showing two typical examples of multiple 3D web agents: a soccer game and the simulation of a dog world. Programming 3D web agents usually involves the following main issues:

- **Agent model**: How DLP can be used to implement 3D web agents for different agent architectures, like simple reflex agents, decision making agents, BDI agents, etc. Simple reflex agents are simple in the sense that they are not powerful enough for complex tasks, like soccer game agents. The implementation of extended BDI agents involves a lot of technical details how the mental attitude components should be manipulated. In this chapter, we will start with an example based on an decision making agent model, and then discuss how the agents based on the BDI model can be implemented in DLP.

- **Multiple Agent Management**: For virtual environments, multiple agents are usually needed to fulfill complex tasks. Therefore, we have to deal with the following problems: How multiple agents can be created and how they can interact with each other by using DLP. In this chapter, we focus on the problem of non-distributed multiple agent systems, i.e. agents re located on the same computer. In the chapter 11, we will discuss how DLP can be used to deal with distributed multiple agent systems.

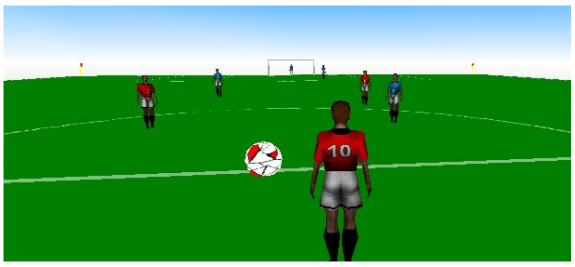


Fig. 7.1 Screenshot of Soccer Playing Game

- **Formalizing Dynamic Worlds**: 3D virtual worlds usually consist of two kinds of data: a static part, for example the geometrical data of a soccer field and soccer ball in a soccer game, and a dynamic part, describing the dynamic behavior of a soccer ball. The static component of virtual worlds is usually created by VRML. The dynamic parts are more efficiently controlled by DLP. Here, we consider how DLP can be used to formalize the dynamic behavior of virtual worlds.

- **Cognitive Model**: In order to make decisions in virtual worlds, the agents should have some minimal knowledge about the virtual world they are located in and the scenarios the agents may play, which lead to the construction of cognitive models for 3D web agents. In the example of soccer games, the soccer playing agents need to know which actions should be taken under particular conditions.

7.2 SOCCER PLAYING AGENTS: AN EXAMPLE

We take the soccer playing game as an example, because soccer playing is a game that typically requires prompt reactivity and complex interaction between the multiple agents in a system.

7.2.1 General Consideration

We will design a soccer game for the Web, namely, a soccer game that can be played in web browsers, like Netscape communicator or Microsoft Internet Explorer. We consider two teams in the game, red and blue, and each team has ten players and one goalkeeper. The players and goalkeepers will be implemented as 3D web agents. Considering the problem of performance, we may reduce the numbers of players in the game. Four players and one goalkeeper in each team are normally enough to show interesting game scenarios. Being one of the players, the user can use the keyboard or mouse to play the game. A screenshot of the soccer playing game is shown in Figure7.1:

In this chapter we consider only a game in which all agents are located at the same computer, namely, a non-distributed system. Therefore, only one user is allowed to join the game. In Chapter 11, we will describe a soccer game with multiple users, that is, a distributed soccer game.

7.2.2 Design of virtual worlds

Before we start to design 3D soccer playing agents, we should design 3D virtual worlds for the soccer games by using VRML. The 3D virtual worlds of the soccer games consists of the following parts: a soccer field, a soccer ball, two goal gates, and twenty-two soccer player avatars. Moreover, in order to show the game score, we also need two score plates which locate each side of the soccer field.

Figure7.2 shows a field with a length of 100 meter and a width of 64 meter. This field is a gif file ”field1.gif”, therefore the field can be designed in VRML as follows:

```
Transform {
      translation 0 0 0
            children [
            Shape {
              appearance Appearance {
              texture ImageTexture { url "field1.gif" }
              textureTransform TextureTransform {scale 1 1}
               }
               geometry Box {size 100 .2 64}
             }
             ]
```

The soccer balls are spherical objects with a radius of 18 cm. First we design a prototype of soccer balls with the fields translation and rotation as follows:

```
PROTO Ball [
          exposedField SFVec3f translation 0 0 0
          exposedField SFRotation rotation 0 1 0 0
      ]
 {
Transform { translation IS translation
            rotation IS rotation
      children [
      Transform {
      translation 0 0 0
                       children [
      Shape {
```

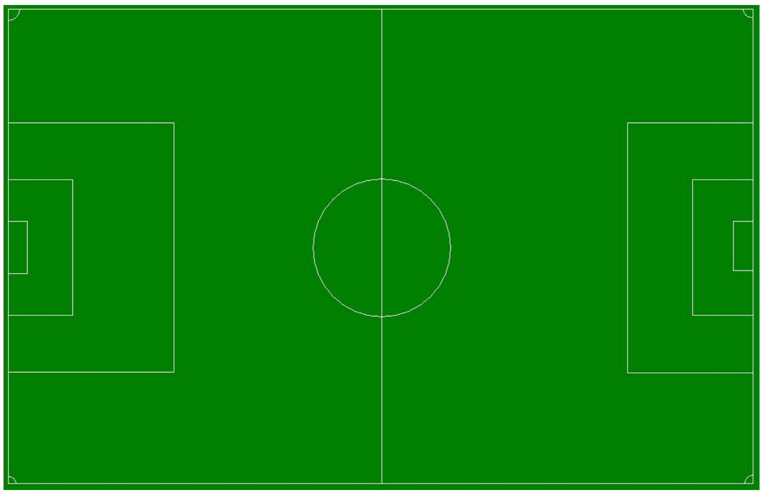


Fig. 7.2 The Field of Play

```
           appearance Appearance {material Material
         {diffuseColor .9 .9 .9
         emissiveColor .9 .9 .9}
           texture ImageTexture { url "ball1.gif" }
           textureTransform TextureTransform {scale 2 2}
                    }
            geometry Sphere {radius .18}
           }

 ] }
] } }
```

Then, we define a concrete soccer ball with the name "ball", which is located at the center of the field $\langle 0, .25, 0\rangle$, its default position [1]:

```
Transform {
    children [
     DEF ball Ball
                     { translation 0 .25 0
                      rotation 0 1 0 0
                     }
    ] }
```

[1]We make the y-dimension of the ball position a little bit higher, so that we can see it better.

Similarly we can define the prototype of goal gates with translation and rotation fields as follows. However, we will omit some details:

```
PROTO Gate [
           exposedField SFVec3f translation 0 0 0
           exposedField SFRotation rotation 0 1 0 0]
{
Transform { translation IS translation
            rotation IS rotation
      children [
......
] } }
```

Furthermore, we define two concrete goal gates: leftgate and rightgate, which are at the positions $\langle -49, 0, 0\rangle$ and $\langle 49, 0, 0\rangle$:

```
Transform {
    children [ DEF leftGate Gate
                     {translation -49 0 0
                      rotation 0 1 0 1.5708
    } ] }

Transform {
   children [ DEF rightGate Gate
                     { translation 49 0 0
                      rotation 0 1 0 -1.5708
    } ] }
```

Moreover, we need twenty-two soccer player avatars for the games. The prototype of player avatars can be designed with different complexities, which may range from a simple object like a box, to more complicated humanoids with facial expressions and body gestures. We will discuss the issues of avatar design in the next chapter. In this chapter, we just assume that the avatars can be designed with the following main fields:

- **position**: to set the position of the avatar.
- **rotation**: to set the rotation of the avatar.
- **nickname**: to give a hint to the name of the avatar.
- **whichChoice**: to indicate whether or not the avatar should appear in the virtual world, which is convenient to add or delete any avatars from the scene. If whichChoice is -1, the avatar does not appear in the scene. This is useful for a multi-user version of the soccer game; when a new user joins the game, we can set the field "whichChoice" of its avatar to 1, and set the field value to -1 when a user quits the game.

- **picturefile**: to set the clothes and the appearance of the avatar. For instance, any player in the team red should wear red clothes with a player number. The appearance can be designed in a picture file.

The prototype of player avatars is designed as follows:

```
PROTO Sportman [
    exposedField SFVec3f position
    exposedField SFRotation rotation
    exposedField SFInt32 whichChoice
    exposedField SFString nickname
    exposedField MFString picturefile
]
{......}
```

The following VRML code defines a goalkeeper and a player:

```
Transform {
    children [ DEF goalKeeper1 Sportman
       {rotation 0 1 0 -1.5708
       whichChoice -1
       position 48 1.8 0
       picturefile ["sportmanblue1.jpg"]
       nickname "blue1"
                   } ] }

Transform {
    children [ DEF blue9 Sportman
                   {
             rotation 0 1 0 -1.5708
             whichChoice -1
             position 35 1.8 4
             picturefile ["sportmanblue9.jpg"]
             nickname "blue9"
                   } ] }
```

In order to let a user join the game, we also need to add facilities in the virtual worlds to get and set the user's viewpoint position and orientation, like we discussed in Chapter 4.

7.2.3 Multiple Thread Control

Now, we are ready to desig the DLP part. The 3D soccer game is a concurrent system, which involves the dynamic activities of the soccer ball and the soccer player agents. For its realization we use the multi-threaded object facilities of DLP.

First of all, we define a game clock to control the time of the game as follows:

```
:- object game_clock.
      var time_left = 5000.

      get_time(Time) :-
            Time := time_left.

      set_time(Time) :-
            time_left := Time.

:- end_object game_clock.


:- object clock_pulse.

      clock_pulse(Clock) :-
            repeat,
             sleep(1000),
             Clock <- get_time(Time),
             Left is Time - 1,
             Clock <- set_time(Left),
        Left < 1,
       !.

:- end_object clock_pulse.
```

We set the total time of the game to 5000 seconds. The object clock-pulse repeatedly sets the game time, i.e., after sleeping 1000 milliseconds. The game clock object has the functions get_time and set_time, which can be called by any other object to get the current game time.

The soccer players can be classified into the following three kinds of agents:

- **goalkeeper**: An agent whose active area is around the goal gate of its team;
- **soccerPlayer**: An autonomous agent which plays one of the following roles: forward, middle fielder, and defender.
- **soccerPlayerUser**: An agent (more exactly, just an object), which represents the user.

Suppose that we have designed the DLP objects to control the dynamic activities of the soccer ball, goalkeeper, soccerPlayer, and soccerPlayerUser, which will be discussed in the next sections, multi-threading for the soccer game can be implemented as follows:

```
:- object waspsoccer : [bcilib].

var url = 'soccer.wrl'.

main :-
text_area(Browser),
set_output(Browser),

format('Load the game ... ~ n'),
loadURL(url),
Clock := new(game_clock),
        _Pulse := new(clock_pulse(Clock)),

        Clock <- get_time(TimeLeft),
format('the game will start in 5 seconds,~ n'),
format('the total playing time is ~w seconds,~ n', [TimeLeft]),
delay(5000),

format('game startup,~ n'),
_ball := new(ball(ball,  Clock)),
_GoalKeeper1 := new(goalKeeper(goalKeeper1, Clock)),
_GoalKeeper2 := new(goalKeeper(goalKeeper2, Clock)),
_UserMe := new(soccerPlayerUser(me_red10, Clock)),
_Blue9 := new(soccerPlayer(blue9, Clock)),
_Blue8 := new(soccerPlayer(blue8, Clock)),
_Blue7 := new(soccerPlayer(blue7, Clock)),
_Red2 := new(soccerPlayer(red2, Clock)),
_Red3 := new(soccerPlayer(red3, Clock)),
_Blue11 := new(soccerPlayer(blue11, Clock)),
_Red11 := new(soccerPlayer(red11, Clock)).

:- end_object waspsoccer.
```

The 3D virtual world "soccer.wrl" is loaded into the scene first, then we wait for five second to start the game. Furthermore, we use the method "new" to create new threads which control the behaviors of the soccer ball, two goalkeepers, one soccerPlayerUser, and seven soccerPlayer agents respectively. Note that each thread has its own information about the game clock.

7.2.4 Formalizing Behaviors Soccer Ball

In virtual worlds, we consider a three dimensional coordinate system, in which each point is represented by a vector, like $\langle x, y, z\rangle$.

Suppose that a soccer ball is kicked from an initial point $\langle x_0, y_0, z_0\rangle$ with initial velocity $v = \langle v_x, v_y, v_z\rangle$ meter/second. The acceleration due to gravity is in the negative y-direction and there is no acceleration in the x-direction and z-direction. We have the following equations:

$$
\begin{array}{ll}
(1) & x = x_0 + v_x * t \\
(2) & y = y_0 + v_y * t - 1/2 * g * t^2 \\
(3) & z = z_0 + v_z * t
\end{array}
$$

where t is the time parameter, and g is the acceleration of a body dropped near the surface of the Earth. We take $g = 9.8$. Suppose that the soccer ball is kicked from the ground with a static start. The total time T_{total} of the soccer ball taken from being kicked to falling back on the ground can be calculated from the equation (2), by letting $y = 0$ and $y_0 = 0$. Thus, $T_{total} = v_y/4.9$. The behavior of the soccer ball kicked with static start can be expressed in DLP as follows.

```
kickedwithStaticStart(Ball,X0,Y0,Z0,Vx,Vy,Vz,UpdateDelay):-
    Ttotal is Vy/4.9,
    steps := 1,
    repeat,
       delay(UpdateDelay),
       T is  steps*UpdateDelay,
       X is X0+ Vx*T,
       Y is Y0+ Vy*T-4.9*T*T,
       Z is Z0+ Vz*T,
       setPosition(Ball,X,Y,Z),
       ++steps,
    steps > Ttotal//UpdateDelay,
    !.
```

The predicate *kickedwithStaticStart* requires the information of the kick force vector $\langle Vx, Vy, Vz \rangle$. However, it is much easier to get the information of the agent's rotation along the XZ plane, which leads to a new predicate *kickedwithStaticStartRF* which is based on the information of the agent's rotation, the force in the direction the agent faces, i.e. a force in the XZ plane, and a force in the Y-dimension.

```
kickedwithStaticStartRF(Ball,Field,X,Y,Z,R1,F1,Vy):-
            Vx is F1 * cos(R1),
            Vz is F1 * sin(R1),
            kickedwithStaticStart(Ball,Field,X,Y,Z,
        Vx,Vy,Vz,ballSleepTime).
```

Thus, the object "soccer ball" can be formalized in DLP as follows:

```
:- object ball : [bcilib].

var steps.
var xmax = 50.0.
var xmin = -50.0.
var zmax = 32.0.
var zmin = -32.0.

var ballSleepTime=300.

ball(Name,Clock) :-
      set_field(Name, lock, free),
      format('~w thread active.~ n',[Name]),
      activity(Name,Clock).

activity(Name,Clock) :-
      repeat,
         sleep(5000),
         Clock <-get_time(TimeLeft),
         format(' ~w seconds left for the game.~ n', [TimeLeft]),
         getPosition(Name,X,Y,Z),
         setValidBallPosition(Name,X,Y,Z),
      TimeLeft < 1,
      format('The game is over!!!~ n'),
      game_score <- showGameScore,
      !.

lock(Name) :-
get_field_event(Name, lock, _dont_care_).

unlock(Name) :-
set_field(Name, lock, free).

validMinMax(V0, Vmin, Vmax, VN) :-
            askValid_Min(V0, Vmin, V1),
            askValid_Max(V1, Vmax, VN).

askValid_Min(X, Xmin, Xval) :-
            X < Xmin,
            !,
            Xval = Xmin.

askValid_Min(X,_Xmin, Xval) :-
            Xval = X.


askValid_Max(X, Xmax, Xval) :-
             X > Xmax,
            !,
            Xval = Xmax.

askValid_Max(X,_Xmax, Xval) :-
            Xval = X.


setValidBallPosition(_Ball,X,_Y,Z):-
            X =< xmax,
            X >= xmin,
            Z =< zmax,
            Z >= zmin,
            !.

setValidBallPosition(Ball, X, Y, Z) :-
            validMinMax(X, xmin, xmax, Xnew),
            validMinMax(Z, zmin, zmax, Znew),
            setPosition(Ball, Xnew,Y,Znew).

......

:- end_object ball.
```

The thread for the soccer ball is executes the following tasks:

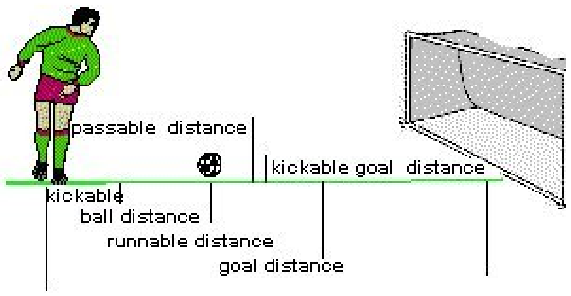
1. regularly checks the game clock.
2. regularly checks the validity of the position of the soccer ball. If the ball is kicked outside the field, the position of the ball should be set back to a proper position inside the field.
3. offer a kick predicate for players.
4. lock and unlock the status of the soccer ball to avoid strange behaviors of multiple kicks at the same time. In the chapter describing a multi-user version of the soccer game, we will see a more efficient approach to the problem of multiple kicks.

7.2.5 Cognitive Models of Soccer Players

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 [FIFA, 2001].

The agents in the soccer game use a simple cognitive model of the soccer game, in which the agents consider the information about several critical distances, then make a decision to kick. The considered critical distances are:

- *ball distance*: the distance between the ball and player.
- *goal distance*: the distance between the player and goal gate.
- *kickable ball distance*: the agent can directly kick the ball.
- *kickable goal distance*: the agent can kick the ball to the goal.
- *runnable distance*: the agent can run to the ball.
- *passable distance*: the agent can pass the ball to a team-mate.



We are now going to design the player agents, based on a decision-making model. Namely, each player agent has the following cognitive loop: sensing–thinking-acting. When sensing, agents use their sensors to get the necessary information about the current situation. The main information that's gathered: the agent's own position, the soccer ball's position, and the goal gate's position. During the stage of thinking, agents have to reason about other players' positions and roles, and decide how to react. Thinking results in a set of intended actions. By acting, agents use their effectors to take the intended actions.

A general framework of the soccer playing agents based on decision-making models can be programmed in DLP as follows:

```
:- object soccerPlayer : [bcilib].


soccerPlayer(Name, Clock) :-
      setSFInt32(Name,whichChoice, 0),
      format('~w thread active.~ n', [Name]),
      activity(Name,Clock).

activity(Name,Clock) :-
      repeat,
            sleep(2000),
            Clock <- get_time(TimeLeft),
            format(' player ~w  thread ~w seconds left~ n',
        [Name,TimeLeft]),
            getPositionInformation(Name,ball,X,Y,Z,Xball,Yball,Zball,
        Dist,Xgoal,Zgoal,DistGoal),
            findHowtoReact(Name,ball,X,Y,Z,Xball,Yball,Zball,Dist,
        Xgoal,Zgoal,DistGoal,Action),
            format('player ~w action: ~w ~ n',[Name,Action]),
            doAction(Action,Name,ball,X,Y,Z,Xball,Yball,Zball,Dist,
        Xgoal,Zgoal,DistGoal),
      TimeLeft < 1,
      quitGame(Name),
      !.

......

:- end_object soccerPlayer.
```

At the beginning of a soccer player agent thread, the avatar should be set to appear in the scene. Then, it should gather the information about the agent's own position $\langle X, Y, Z\rangle$, the ball's position $\langle Xball, Yball, Zball\rangle$, the position of the goal gate $\langle Xgoal, Ygoal, Zgoal\rangle$, and calculate the distance from the agent to the ball and the gate. Since the y-position of the goal gate is never changed, it is not used for the calculation of the goal distance, i.e. we do not need the value of the y-position. After getting the necessary information, the agents should figure out how to react, and then do the actions.

To simplify the decision-making procedure, we can carefully design the procedure, so that the agents need not to evaluate the outcomes of actions but but only have to focus on the actions themselves. For example, we consider only the following actions: shooting, passing, run-to-ball and move-to-default-position.

- shooting: the ball is kicked to the gate.
- passing: two consecutive kicks are done by two players of the same team.
- run-to-ball: as the name implies, run to the ball.
- move-to-default-position: move around the agent's active area.

We can design the decision-making procedure in such a way that there is only one possible action with respect to a particular situation. Suppose that the ball distance is $Dist$ and the goal distance is $DistGoal$, the decison tree based on the simplified model can be as follows:

If the ball is kickable (i.e, close enough to kick), and the gate is close enough, then do shooting; if the ball is kickable, but the gate is too far, then try to pass the ball to a teammate; if the ball is not kickable, and the ball is *located within the agent's active area, then run to the ball until it is kickable; if the ball is not kickable, and the ball is not located within the agent's active area, then move around in the agent's active area.*

The procedure can be programmed in DLP as follows:

```
findHowtoReact(_,Ball,_,_,_,_,_,_,Dist,_,_,Dist1,shooting):-
      Dist =< kickableDistance,
      Dist1 =< kickableGoalDistance,
      !.

findHowtoReact(_,_,Ball,_,_,_,_,_,_,Dist,_,_,Dist1,passing):-
      Dist =< kickableDistance,
      Dist1 > kickableGoalDistance,
      !.
   findHowtoReact(Player,_,_,_,_,X1,_,_,Dist,_,_,_,run_to_ball):-
      Dist > kickableDistance,
      getFieldAreaInformation(Player,_,_,FieldMin,FieldMax),
      FieldMin =< X1,
      FieldMax >= X1,
      !.

findHowtoReact(Player,_,_,_,_,X1,_,_,Dist,_,_,_,move_around):-
      Dist > kickableDistance,
      getFieldAreaInformation(Player,_,_,FieldMin,_),
      X1 < FieldMin,
      !.

findHowtoReact(Player,_,_,_,_,X1,_,_,Dist,_,_,_,move_around):-
      Dist > kickableDistance,
      getFieldAreaInformation(Player,_,_,_,FieldMax),
      X1 > FieldMax,
      !.
```

Taking actions can be simple, or complicated, which depends on the situation. For example, shooting can be programmed in DLP as follows:

```
doAction(shooting,Player,Ball,_,_,_,X1,Y1,Z1,_,
        Xgoal,Zgoal,DistGoal) :-
      ball <- isUnlocked(Ball),
      ball <- lock(Ball),
      kick_ball_to_position(Ball,X1,Y1,Z1,
        Xgoal,0.0,Zgoal,DistGoal),
      ball <- unlock(Ball),
      !.
```

Namely, first check if the ball is unlocked, if yes, then lock the ball, then kick the ball to the gate, then unlock the ball. However, just like most soccer players in real life, they cannot fully control the soccer ball to kick it into the gate, which usually involve some degree of uncertainty. Therefore, in the following, we introduce a random number to change the behavior of kick-ball-to-position:

```
kickBalltoDirection(Player,Ball,X,_,_,X1,Y1,Z1,X2,_,Z2):-
      check(Ball,position(X1,Y1,Z1)),
      look_at_position(Player,X2,Z2),
      getRotation(Player,_,_,_,R),
      getCorrectKickRotation(X,X1,R,R1),
      KickBallForceY is kickBallForceY + random*10.0,
      ball <- kickedwithStaticStartRF(Ball,translation,X1,Y1,Z1,
        R1,kickBallForce,KickBallForceY).
```

The action "passing" is more complicated, for it needs to find a proper teammate to pass the ball. An intuitive idea is to pass the ball to the nearest teammate. However, that would be expensive computationally. (Why? we leave it as an exercise.) Therefore, we need an efficient solution to reason about the teammate's position and roles to find a suitable teammate to pass the ball to. Again, we leave the problem as an exercise.

7.2.6 Controlling Goalkeepers

The behavior of the goalkeeper agent is almost the same as the soccer player agent, namely, they should have the same cognitive loop. However, the goalkeepers should have a set of different actions. For example, a goalkeeper never considers the action "shooting". They have different decision trees.

It is convenient to require the goalkeepers to check regularly whether or not a new game score should be added. If a goalkeeper finds that the ball is already inside the gate, he should add one more score to his opponent team, then throw the ball to one of his teammates, or run to the ball, then kick the ball. The action "run-then-kick" can be simply programmed in DLP as follows:

```
doAction(run_then_kick, GoalKeeper,Ball,_X,Y,_Z,X1,Y1,Z1,_):-
            format('~w action: run_then_kick.~ n',
        [GoalKeeper]),
            setSFVec3f(GoalKeeper,position,X1,Y,Z1),
            ballThrowPosition(GoalKeeper,_,X3,Z3),
            look_at_position(GoalKeeper,X3,Z3),
            getRotation(GoalKeeper,_,_,_,R),
            R1 is R - 1.5708,
            ball <- kickedwithStaticStartRF(Ball,translation,X1,Y1,Z1,R1,
        kickBallForce,kickBallForceY),
            !.
```

7.2.7 Behaviors of Soccer Player Users

Different from the soccer player agents and the goalkeeper agents, the behaviors of the object "Soccer Player User" are rather simple. They need no cognitive model for the game. They do not make autonomous decisions. The main task for them is to check whether or not the user is near enough to be able to kick the ball. If yes, then move the ball to an appropriate position. The main framework of the "soccer player user" object can be as follows:

```
:- object soccerPlayerUser : [bcilib].

var kickableDistance = 3.0.
var kickBallForce = 10.0.
var kickBallForceY =6.0.

soccerPlayerUser(Name, Clock) :-
      format('The user ~w thread active.~ n', [Name]),
      activity(Name,Clock).

activity(Name,Clock) :-
      repeat,
            sleep(1000),
            Clock <- get_time(TimeLeft),
            near_ball_then_kick(Name,ball),
      TimeLeft < 1,
      !.



near_ball_then_kick(Agent, Ball):-
      getViewpointPosition(Agent,X,_,Z),
      getPosition(Ball,X1,Y1,Z1),
      X < X1,
      distance2d(X,Z,X1,Z1,Dist),
      Dist < kickableDistance,
      getViewpointOrientation(Agent,_,_,_,R),
      R1 is R -1.5708,
      ball <- isUnlocked(Ball),
      ball <- lock(Ball),
      ball <- kickedwithStaticStartRF(Ball,translation,X1,Y1,Z1,R1,
        kickBallForce,kickBallForceY),
      ball <- unlock(Ball).
```

```
near_ball_then_kick(Agent, Ball):-
     getViewpointPosition(Agent,X,_,Z),
     getPosition(Ball,X1,Y1,Z1),
     X >= X1,
      distance2d(X,Z,X1,Z1,Dist),
      Dist < kickableDistance,
      getViewpointOrientation(Agent,_,_,_,R),
      R1 is -R -1.5708,
      ball <- isUnlocked(Ball),
      ball <- lock(Ball),
     ball <- kickedwithStaticStartRF(Ball,translation,X1,Y1,Z1,R1,
        kickBallForce,kickBallForceY),
      ball <- unlock(Ball).


near_ball_then_kick(_, _).

......

:- end_object soccerPlayerUser.
```

The predicate *near_ball_then_kick* is used to check whether or not the user is close enough to kick the ball. If yes and the ball is unlocked, then kick the ball based on the user's current orientation.

7.2.8 Discussion

In this subsection, we have discussed how DLP can be used to implement a single user / multi-player soccer game. In this game, multiple 3D web agents are based on a decision making architecture. Decision rules are used to guide their behaviors to show certain intelligence.

A problem in this game is the 'line-up' phenomenon in which several agents run to the same position without awareness of the other players' behaviors, as shown in Figure7.3. The 'line-up' phenomenon results in that the agents do not use the information of other teammates when they decide to run to the goal position. Here are several solutions for 'line-up' phenomenon:

- Solution 1: obtaining more information of players, and add more computation. For instance, in the game, we can ask the agents to find the nearest agent to the soccer ball. Only the nearest player in the team should run to the soccer ball. However, the problem of this solution is that it is unsuitable for real-time agents, like the agents in the soccer game, because the computation on the closest teammate is expensive.

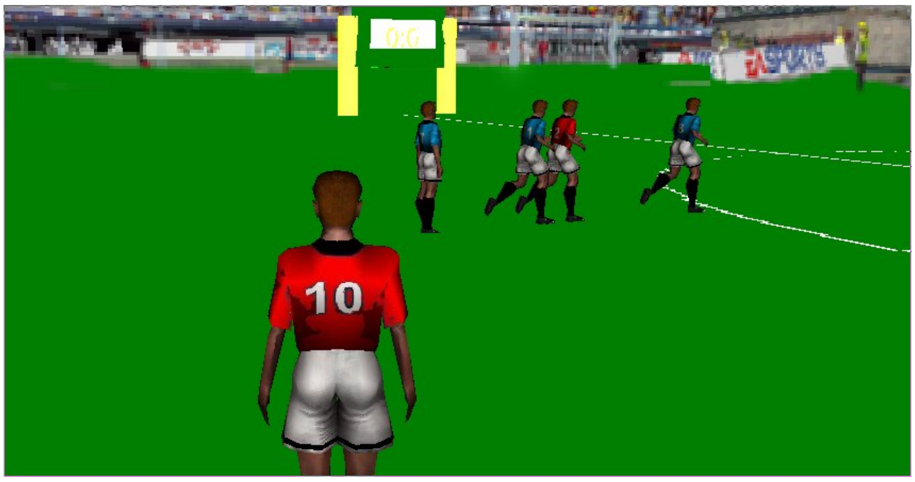


Fig. 7.3 Line-up phenomenon in Soccer Playing Game

- Solution 2: Introducing Distributed behavioral models, like those that are used in the simulation of flocks, herds, and schools in artificial life. We are going to discuss a typical example in the next subsection. The problem of this solution is that it is not intelligent enough, however, it is good enough to avoid the 'line-up' phenomenon.

7.3 DOG WORLD

In this subsection, we are going to discuss the 'dogworld' example in which several dogs can play with their master, i.e., the user. The dogs can move with the master, run to the master, and bark, without the 'line-up' phenomenon. A screen shot of the dogworld example is shown in Figure7.4.

7.3.1 Design the virtual world

First we design a virtual world of several dogs with the following fields:

- position: a position field of a dog;
- rotation: a rotation field of a dog;
- whichChoice: a field which can be used to set whether a dog appears or disappears in the virtual world, like the playing agents in the soccer game example;
- id: an identification field of the dog, which will be used in the distribution function for the simulation of the flock which is discussed later;

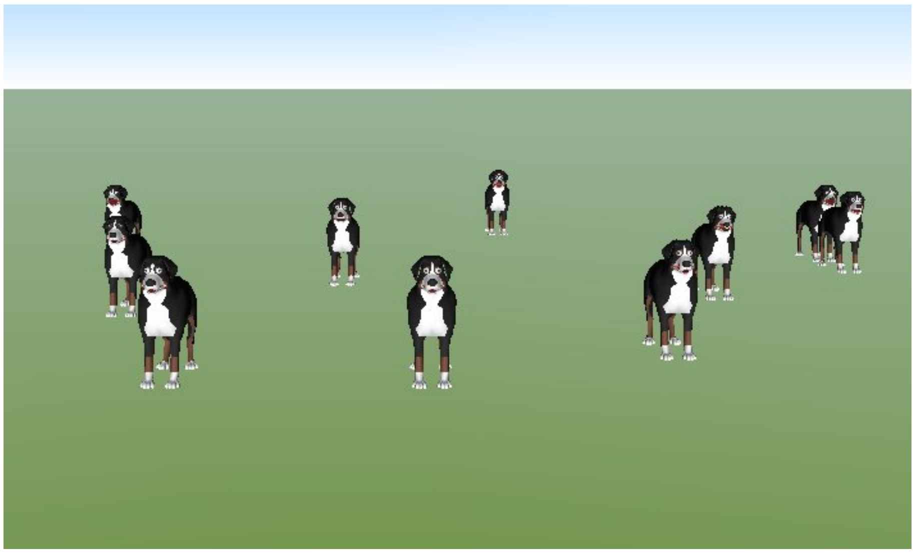


Fig. 7.4 Screenshot of the dogworld example

- bark: a field which can be used to control the bark sound in a file 'bark.wav'.

7.3.2 Behavioral model of the dogs

The behavior of the dogs are based on the following rules:

- If the master runs, then dogs run with the master.
- If the master stands, then dogs bark, and move to the master.
- If the master is slowing down, then dogs stop move, look at the master, and bark.
- If the master is too far away from a dog, the dog runs back to the master.

Of course, we should have described in more detail what means by slowing down and too far away. That can be done by different parameters, like the following:

```
var sleeptime = 250.
var small_movement = 0.20.
var big_movement = 0.40.
var max_distance = 40.
```

Namely, if the master moves more than 0.40 meter within 250 milliseconds, it means that the master is running; if the master moves less than 0.20 meter,

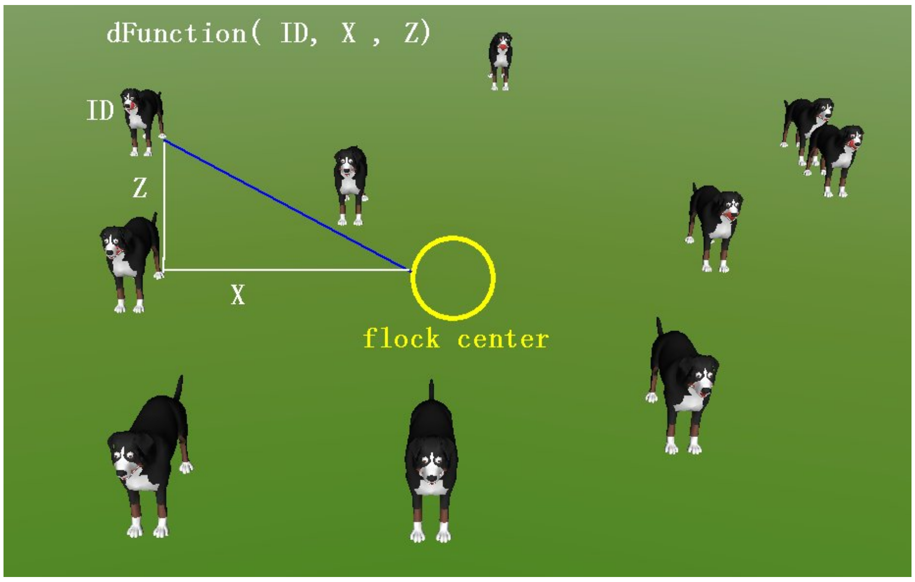


Fig. 7.5 Flock and Distributive Function

it means that the master is standing; if the master is neither running, nor standing, it means that the master is slowing down; if the master is at a distance of 40 meters, it means that master is too far away.

7.3.3 Flock and Distributive Function

In [Reynolds, 1987], Craig Reynolds discusses the problem of simulated flocks, and points out that the following issues are important for the simulation of the flocks.

1. Collision Avoidance: avoid collisions with nearby flockmates.

2. Velocity Matching: attempt to match velocity with nearby flockmates.

3. Flock Centering: attempt to stay close to nearby flockmates.

In the dogworld example, we do not try to solve all of these problems in simulated flocks. However, we borrow the idea of centering to avoid the line-up phenomenon. We design a distribution function so that each dog moves to a simulated flock center based on this distribution function, as shown in Figure7.5.

7.3.4 Implementation

We can control the bark sound by setting the field bark in the virtual world, so that when the value of the loop is 'true', then the sound file is playing repeatedly, and when the value is 'false', the barking stops:

```
bark(Dog,Time):-
      setSFBool(Dog, bark, true),
      sleep(Time),
      setSFBool(Dog, bark, false).
```

The next issue is to simulate the flock centering. When the master is standing, the flock center is the position of the master. However, when the master is running, the dogs usually run much faster then the master. We set the flock center dynamically somewhere in front of the master. Therefore, we design an enlargement parameter so that the flock center is changing dynamically. The computation of the flock center can be based on the position of the master, the movement of the master and the enlargement parameters. We need a distribution function which can be used to compute a relative position of a dog to the flock center. To simplify the problem, we design the distribution function so that the relative position of a dog to the flock center is constant. We use the predicate *dFunction* to specify the distribution function. $dFunction(ID, X, Y)$ means that the dog ID should move to the position $\langle x_0 + X, y_0 + Y\rangle$ if the position of the flock center is $\langle x_0, y_0\rangle$.

The implementation of the actions of the dogs can be described in DLP as follows. The complete source code of the dogworld example can be found in the appendix.

```
doAction(Dog, position(X,_Y,Z),position(X1,_Y1,Z1),_,_,_,
  look_at_master):-
     ook_at_position(Dog,X,Z,X1,Z1),
     bark(Dog,500).

doAction(Dog, position(X,Y,Z),position(X1,_Y1,Z1),_,_,_,
   move_to_master):-
     getSFInt32(Dog,id,ID),
     getFlockCenter(position(X2,_Y2,Z2), master_standing,[]),
     dFunction(ID, Xd,Zd),
     X3 is X2 + Xd,
     Z3 is Z2 + Zd,
     look_at_position(Dog,X,Z,X1,Z1),
     bark(Dog,500),
     move_to_position(Dog,position(X,Y,Z),position(X3,Y,Z3),5),
!.

doAction(Dog, position(X,Y,Z),position(X1,_Y1,Z1),_,
   position(X3,Y3,Z3),_,move_with_master):-
     Xdif is X3-X1,
     Zdif is Z3-Z1,
     getFlockCenter(position(X5,_Y5,Z5), master_moving,
        [position(X3,Y3,Z3),Xdif,Zdif]),
     getSFInt32(Dog,id,ID),
     dFunction(ID, Xd,Zd),
     X6 is X5 + Xd,
     Z6 is Z5 + Zd,
     look_at_position(Dog,X,Z,X6,Z6),
     move_to_position(Dog,position(X,Y,Z),
        position(X6,Y,Z6),10),
     !.

......

getFlockCenter(position(X,Y,Z),master_standing, []):-
     getSFVec3f(proxSensor,position,X,Y,Z),
     !.

getFlockCenter(position(X,Y,Z),master_moving,
   [position(X1,Y,Z1), Xdif, Zdif]):-
     X is X1 + Xdif* enlargement,
     Z is Z1 + Zdif* enlargement,
     !.

dFunction(1,0,3):-!.
dFunction(2,-1,-2):-!.
dFunction(3,1,-7):-!.
dFunction(4,-2,3):-!.
dFunction(5,2,2):-!.
dFunction(6,-3,0):-!.
dFunction(7,3,-1):-!.
dFunction(8,-4,4):-!.
dFunction(9,5,-4):-!.
dFunction(10,5,-3):-!.
```

7.3.5 Discussion

In this example, we have shown how a distribution function can be used to simulate a flock without the 'line-up' problem. The position of an agent, i.e., a dog in this example, relative to the flock center, is a constant in the distribution function. Although it is a simple solution, the behaviors are not natural enough for the simulation of a flock. That can be improved by the introduction of more flexible functions.

Exercises

7.1 Consider the following soccer game extensions:

7.1.1. Analysis the reasons why the computation on the closest teammate is expensive.

7.1.2. Find your own solution to the action "passing" and program it in DLP.

7.1.3. For the action "run to ball", a soccer player agent should not simply run to a position. Since the position of the ball is always changing, the player should "run and trace" the ball. Write a program to implement the action "run and trace".

7.1.4. Develop a cognitive model for the goalkeeper, and design a decision tree to control the behavior of the goalkeeper.

7.1.5. Improve the soccer game example to avoid the 'line-up' phenomenon.

7.2 Improve the dog world example by adding more rules on the behavioral model, like:

- If the master stands near a dog, then the dog jumps up and bark.
- If the master stands and turns slowly, then dogs lie upside down.

7.3 Design a 3D web agent for the navigation assistant in virtual worlds, like a guide in a building. The 3D web agent should be able to fulfill the following tasks:

7.3.1. Send greeting messages whenever she finds a new visitor.

7.3.2. Show a brief introduction to the configuration of the building.

7.3.3. Guide the visitor to look around in a (virtual) building.

8
Avatar Design

8.1 AVATARS

In vitual worlds, avatars are used to represent human users or autonomous agents. The complexities of avatars can range from the simple form, which may just consists of a group of boxes and spheres, to the most complicated ones, like humanoids with facial animation and sophisticated gestures. Based on their functionalities, avatars can be classified into the following different types:

- *Animated versus Non-animated*: Animated avatars have their own animation data. These animations are usually controlled by using ROUTE semantics to express built-in gestures. Non-animated avatars have no built-in animation data. However they may be controlled by using external facilities (i.e., programs) to make the animation. DLP can be used to control the animation if the body components of the avatars are properly defined by using DEF and those defined nodes have geometrical fields, like position and rotation. In this chapter, we are more interested in the design of non-animated avatars. However, we will discuss how DLP can be used to create and control the gestures of the avatars.

- *Texture-based versus non-texture-based*: Texture-based avatars use textures to cover or present parts of their bodies, especially, the face and clothes. These textures are usually presented by using picture files, like gif or jpeg. Non-texture-based avatars do not use any picture files. They use their own built-in appearance data to present special effects. The file sizes of the texture-based avatars are usually small, for they have embedded picture files. However, they are hard to be controlled by programs. Non-texture-based avatars have their own appearance data, which would increase the file size of the avatars, however, they provide the possibilities for the control from external programs. In this chapter we will discuss how DLP can be used to control the facial expression of non-texture-based avatars.

- *audio-embedded versus non-audio-embedded*: Audio-embedded avatars have their own embedded audio data in their files, whereas non-audio-embedded avatars have not any voice/sound data on them. The file sizes for good quality voice are usually extremely large. They are seldom embedded into avatars and virtual worlds.
- *H-anim compliant versus non-H-anim compliant.* H-anim1.1 is a specification for standard humanoids by the Humanoid animation working group.[H-anim, 2001] As the name implies, H-anim compliant avatars are designed according to the H-anim specification, whereas non-H-anim compliant avatars are not. In this chapter we will focus on the design and the control of H-anim compliant avatars.

8.2 H-ANIM 1.1 SPECIFICATION

As claimed by Humanoid animation working group in [H-anim, 2001], goals of H-anim specification are the creation of libraries of humanoids for reusable in Web-based applications, as well as authoring tools that make it easy to create humanoids and animate them in various ways. H-anim specifies a standard way of representing humanoids in VRML97. This standard will allow humanoids created using authoring tools from one vendor to be animated using tools from another. H-Anim humanoids can be animated using different animation systems and techniques.

An H-Anim file contains a set of Joint nodes that are arranged to form a hierarchy. Each Joint node can contain other Joint nodes, and may also contain a Segment node which describes the body part associated with that joint. Each Segment can also have a number of Site nodes, which define locations relative to the segment. Sites can be used for attaching accessaries, like hat, clothing and jewelry. In addition, they can be used to define eyepoints and viewpoint locations. Each Segment node can have a number of Displacer nodes, that specify which vertices within the segment correspond to a particular feature or configuration of vertices.

The Joint PROTO looks like this:

```
PROTO Joint [
    exposedField     SFVec3f      center             0 0 0
    exposedField     MFNode       children           []
    exposedField     MFFloat      llimit             []
    exposedField     SFRotation   limitOrientation    0 0 1 0
    exposedField     SFString     name                ""
    exposedField     SFRotation   rotation            0 0 1 0
    exposedField     SFVec3f      scale               1 1 1
    exposedField     SFRotation   scaleOrientation    0 0 1 0
    exposedField     MFFloat      stiffness           [ 0 0 0 ]
    exposedField     SFVec3f      translation         0 0 0
    exposedField     MFFloat      ulimit              []
]
```

The meanings of most fields of the Joint PROTO, like scale, translation, are straightforward from the names, however, the following fields need a further explanation

- ulimit and llimit: gives the upper and lower joint rotation limits. The ulimit field defines the maximum values for rotation around the X, Y and Z axes. The llimit field describes the minimum values for rotation around those axes.
- limitOrientation: describes the orientation of the coordinate frame in which the ulimit and llimit values are to be interpreted. This field specifies the orientation of a local coordinate frame, relative to the Joint center position described by the center exposedField.
- stiffness: specifies values ranging between 0.0 and 1.0 which give the inverse kinematics system hints about the "willingness" of a joint to move in a particular degree of freedom.

The segment PROTO look like this:

```
PROTO Segment [
    field           SFVec3f     bboxCenter        0 0 0
    field           SFVec3f     bboxSize          -1 -1 -1
    exposedField    SFVec3f     centerOfMass      0 0 0
    exposedField    MFNode      children          [ ]
    exposedField    SFNode      coord             NULL
    exposedField    MFNode      displacers        [ ]
    exposedField    SFFloat     mass              0
    exposedField    MFFloat     momentsOfInertia  [ 0 0 0 0 0 0 0 0 0 ]
    exposedField    SFString    name              ""
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
]
```

An explanation on some of the fields above:

- mass: the total mass of the segment, however, it is usually not necessary.
- centerOfMass: the location within the segment of its center of mass.

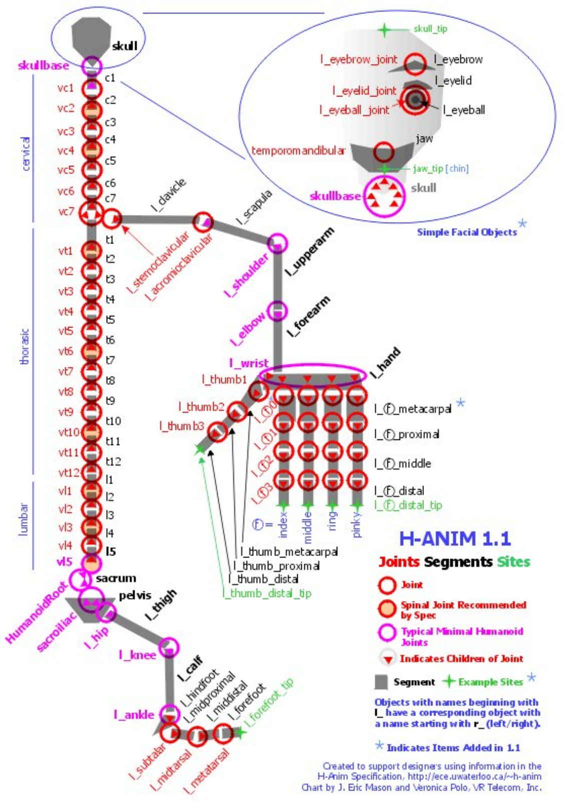


Fig. 8.1 A Standard Joints/Segment Diagram of H-anim 1.1

- momentsOfInertia: the moment of inertia matrix. The first three elements are the first row of the 3x3 matrix, the next three elements are the second row, and the final three elements are the third row.

A standard joints/segment diagram of H-anim 1.1 specification is shown in Figure 8.2

In H-anim specification, site nodes are designed for the following three purposes: First, it can be used to be an "end effector" location for an inverse kinematics system. Next, it defines an attachment point for accessories such as hat and clothing. Third, it provides a location for a virtual camera in the reference frame of a Segment (such as a view "through the eyes" of the humanoid for use in multi-user worlds).

Sites are located within the children exposedField of a Segment node. The children field of a site node is used to store any accessories that can be attached to the segment.

The Site PROTO looks like this:

```
PROTO Site [
    exposedField    SFVec3f     center           0 0 0
    exposedField    MFNode      children         []
    exposedField    SFString    name             ""

    exposedField    SFRotation  rotation         0 0 1 0
    exposedField    SFVec3f     scale            1 1 1
    exposedField    SFRotation  scaleOrientation 0 0 1 0
    exposedField    SFVec3f     translation      0 0 0
    eventIn         MFNode      addChildren
    eventIn         MFNode      removeChildren
]
```

According to H-anim specification, if used as an end effector, the Site node should use the following consisting naming system: the name of the site should be with with the name of the segment which attached, like a "_tip" suffix appended. The end effector Site on the right index finger should have a name like "r_index_distal_tip", and the Site node would be a child of the "r_index_distal" Segment. Sites that are used to define camera locations should have a "_view" suffix appended. Sites that are not end effectors and not camera locations should have a "_pt" suffix. Sites that are required by an application but are not defined in this specification should be prefixed with "x_".

Sometimes the application may want to identify some specific vertices within a Segment. That would require a Displace to store the hint message. The Displacers for a particular Segment are stored in the displacers field of that Segment.

The Displacer PROTO looks like this:

```
PROTO Displacer [
    exposedField MFInt32  coordIndex     [ ]
    exposedField MFVec3f  displacements  [ ]
    exposedField SFString name           ""
]
```

The coordIndex field shows the indices into the coordinate array for the Segment of the vertices that are affected by the displacer. The displacements field describes a set of 3D values that should be added to the neutral or resting position of each of the vertices referenced in the coordIndex field of the Segment. These values correspond one to one with the values in the coordIndex array.

The H-anim file also has a single Humanoid node which stores human-readable data about the humanoid such as author and copyright information. That also stores additional information about all the Joint, Segment and Site nodes, and serves as a "wrapper" for the humanoid. Of course, it is used to describe the top-level Transform for positioning the humanoid in virtual worlds.

```
PROTO Humanoid [
    field           SFVec3f    bboxCenter          0 0 0
    field           SFVec3f    bboxSize            -1 -1 -1
    exposedField    SFVec3f    center              0 0 0
    exposedField    MFNode     humanoidBody        [ ]
    exposedField    MFString   info                [ ]
    exposedField    MFNode     joints              [ ]
    exposedField    SFString   name                ""
    exposedField    SFRotation rotation            0 0 1 0
    exposedField    SFVec3f    scale               1 1 1
    exposedField    MFNode     segments            [ ]
    exposedField    MFNode     sites               [ ]
    exposedField    SFVec3f    translation         0 0 0
    exposedField    SFString   version             "1.1"
    exposedField    MFNode     viewpoints          [ ]
]
```

8.3 CREATING H-ANIM COMPLIANT AVATARS

Based on H-anim 1.1 specification, we can design any humanoid with different body geometrical data and different levels of articulation, which can be anything we like. The appendix of the H-anim 1.1 specification also provides a suggest on the body dimension and three levels of articulation. The level of articulation zero is the minimum legal H-Anim humanoid, with the node HumanoidRoot and several default site translations. The level of articulation one is a typical low-end real-time 3D hierarchy. The level of articulation two is a more complex one, a body with simplified spine. Take the VRML files of the level of articulation, which are available from the H-anim web site, as templates for building H-anim 1.1 compliant avatars, we can design our own ones.

Here are some examples: First we can design a simple H-anim 1.1 compliant avatar, which just use simple geometrical data, like box, or sphere, to be the body parts, which is shown in Figure 8.3.

For the convenience of the test on the construction, we can design each body part as a seperated VRML file. For instances, the left-hand can be like this:

```
\#VRML V2.0 utf8

DEF hanim_l_hand Transform {
            translation 0.15 0.7 -0.025
            rotation 0 0 1  0
               children [
                     DEF hanim_l_hand_shape Shape {
```

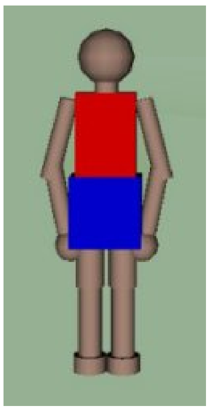


Fig. 8.2 A Simple H-anim 1.1 Compliant Avatar

```
appearance Appearance {
material  Material {
ambientIntensity 0.200
shininess 0.200
diffuseColor 0.76863 0.61961 0.54902
emissiveColor 0.0 0.0 0.0
specularColor 0.0 0.0 0.0
        }}
    geometry Sphere {
       radius 0.07
      } } ]}
```

Namely, the left-hand (without the left forearm and left upper-arm, which would be defined in a separated file), is just a sphere with a defined name "hanim_l_hand". Moreover, we need a translation field and a rotation field to put the body-part to an appropriate position. Suppose that this VRML file is saved with the file name "l_hand0.wrl". We can add it into the left-hand segment in the template file like this:

```
......
children [
               DEF hanim_l_hand Segment {
               name         "l_hand"
               children [

            Inline { url "l_hand0.wrl"}
.....
```

We use *Inline* to add the VRML file into the avatar file, however, note that VRML EAI does not support any referred nodes which is defined in Inline files. If we want to control the defined nodes in DLP, we should add the corresponding lines into the avatar file. Similarly, we can define other body parts, like forearm, thigh, upper-arm, skull, calf, etc.

In order to obtain more realistic humanoid avatars, we need more sophisticated geometrical data on the body parts and some necessary accessories, like hairs, clothes, etc. The upper body and the clothes are normally located at the joint *vl*5. Suppose that the corresponding VRML file is stored as "l5.wrl". Add the data of upper body and the clothes can be like this:

```
......
DEF hanim_l5 Segment {
       name           "l5"
       children [
         DEF hanim_l5 Inline { url "l5.wrl" }
......
```

Moreover, the hairs should be located at the skull_tip site of the skull base joint as follows:

```
.....
 DEF hanim_vl5 Joint {
       name           "vl5"
      center         0.0028 1.0568 -0.0776
       children [
       DEF hanim_skullbase Joint {
         name         "skullbase"
         center       0.0044 1.6209 0.0236
         children [
          DEF hanim_skull Segment {
           name         "skull"
          children [
          DEF hanim_skull
            Inline { url "skull.wrl" }

            DEF hanim_skull_tip Site {
             name         "skull_tip"
             translation    0.0050 1.7504 0.0055
             children [
            Inline { url "hair.wrl" }
             ]
          }
```

A H-anim 1.1 compliant avatar with hairs and clothes is shown in Figure 8.3.

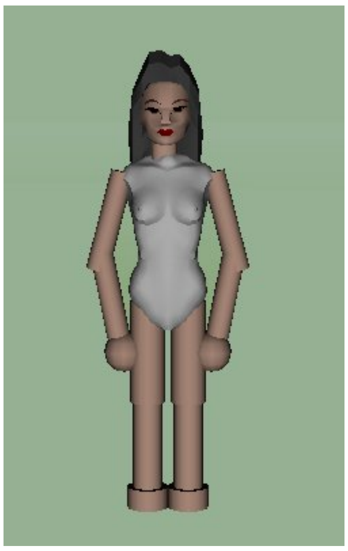


Fig. 8.3 A H-anim 1.1 Compliant Avatar with Hairs and Clothes

8.4 AVATAR AUTHORING TOOLS

8.4.1 Curious Labs Poser 4

Poser 4 by Curious Labs is a 3D-character animation and design tool for avatars design. [Curious Labs] The program provides plentiful libraries of pose settings, facial expressions, hand gestures, and swappable clothing. Users can create images, movies, and posed 3D figures from a diverse collection of fully articulated 3D human and animal models. More usefully, poser 4 supports the export of avatar files with VRML format or H-anim format. However, the sizes of the exported files from poser are usually too large, say, 2 MB or more, which would be a problem for any significant use over the Web. A screenshot of Poser 4 is shown in Figure 8.4.1.

8.4.2 Blaxxun Avatar Studio

Blaxxun Avatar Studio is a tool for design of animated and texture-based VRML avatars. Blaxxun Avatar studio has not yet support of the export of the avatars file with H-anim compliant format. The avatars designed by Avatar Studio are non H-anim compliant ones. Using Avatar Studio, avatars can be designed with large range of body sizes, individual proportions, skin, hair and eye color. Once the avatar's basic properties are determined, she/he can be dressed from a large wardrobe and furnished with an extensive selection

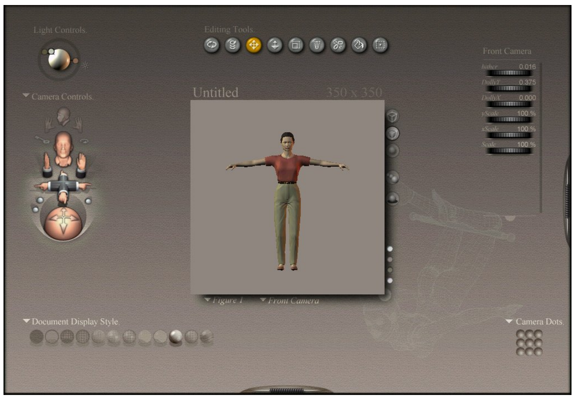


Fig. 8.4 A Screenshot of Poser 4

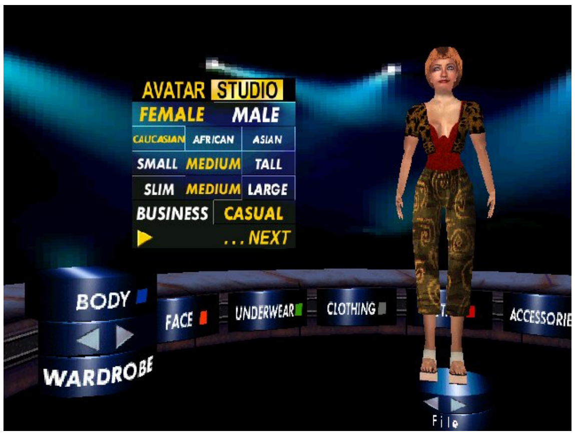


Fig. 8.5 A Screenshot of Avatar Studio

of accessories, like sunglasses or handbags. Avatar Studio also supports the "animations editor," a tool for creating gestures and movements which are then assigned to certain key-words. The typical keys for the gestures are: hello, hey, yes, smile, frown, no, and bye. A screenshot of Blaxxun Avatar Studio is shown in Figure 8.4.2. Avatars designed by Blaxxun Avatar Studio are texture-based ones. Therefore, the faces and clothes of the avatars can be changed by editing the corresponding texture files. For example, Figure 8.4.2 shows the soccer player avatar "blue2", whose texture file for the face and the clothes is shown in Figure 8.4.2.

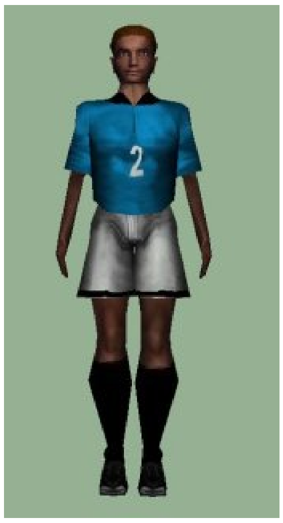


Fig. 8.6 Soccer Player Avatar blue2

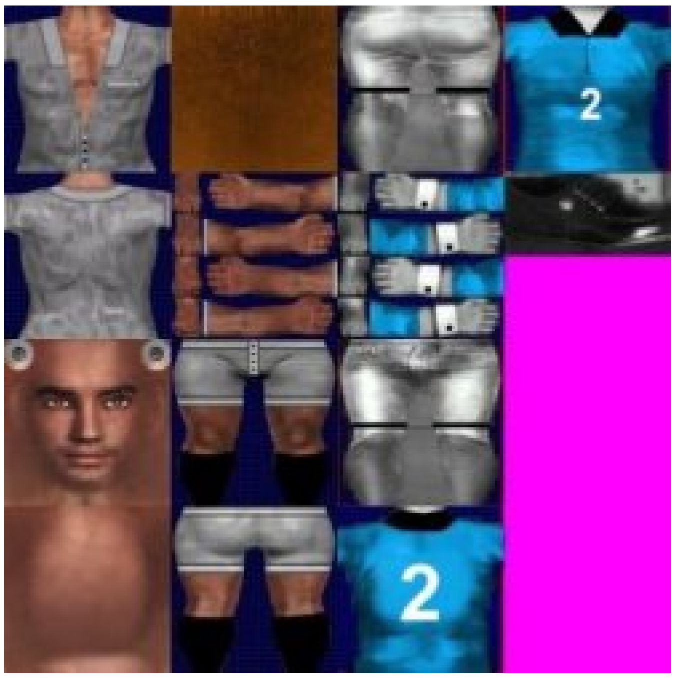


Fig. 8.7 Texture of Soccer Player Avatar blue2

8.5 AVATAR ANIMATION CONTROL IN DLP

Humanoid avatars can be controlled by using the get/set-predicates in DLP, like those are shown in WASP soccer games. They can move to certain positions by the set-position-predicates, or turn to certain orientation by the set-rotation-predicates, on those humanoid avatars. These humanoid avatars can be built based the H-anim specification. The avatar animation can be achieved by setting the positions/rotations of the body parts of the humanoid avatars with different time intervals.

Consider a humanoid avatar which is based on H-anim specification. Turning the left arm of the avatar to front can be realized by simplely setting the rotation of the left shoulder joint *hanim_l_shoulder* to $\langle 1, 0, 0, 1.57\rangle$ as follows,

```
setRotation(hanim_l_shoulder, 1, 0, 0, 1.57)
```

However, in order to achieve a smooth change of movement of the body part, we have to introduce several interpolations between two rotations with certain time interval control. Turning Object to a rotation $\langle X, Y, Z, R\rangle$ within $Time$ milliseconds with I interpolation can be achieved by the following DLP program[1]:

```
turn_object(Object, rotation(X,Y,Z,R), Time, I):-
            getRotation(Object,X1,Y1,Z1,R1),
            count := 0,
            incrementr := (R-R1)/I,
            incrementx := (X-X1)/I,
            incrementy := (Y-Y1)/I,
            incrementz := (Z-Z1)/I,
            sleeptime := Time*1000/I,
            repeat,
                 Rnew is R1+incrementr*(count+1),
                 Xnew is X1+incrementx*(count+1),
                 Ynew is Y1+incrementy*(count+1),
                 Znew is Z1+incrementz*(count+1),
                 setRotation(Object,Xnew,Ynew,Znew,Rnew),
                 sleep(sleeptime),
                 ++count,
            abs(Rnew-R) =< abs(incrementr),
            setRotation(Object,X,Y,Z,R).
```

Animation of avatars usually involves the movements of several body parts simultaneously. That would make the maintenance of the timing and sycroniza-

[1] In order to achieve a natural transition between two rotations, we need the slerp interpolation on quatenions, which is explained the section 9.6.3

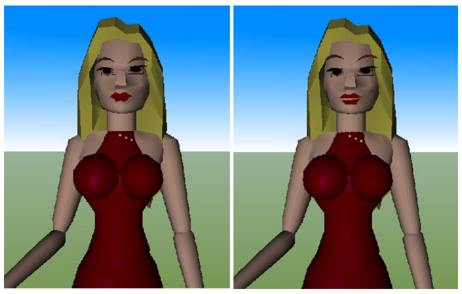


Fig. 8.8 Facial Animation

tion of the multiple threads which controls the body movements more complicated. In Chapter 9 we will discuss a scripting language which can be used to simplize the control of the animation of humanoid avatars.

One of the important issues of the avatar animation is the facial expression. H-anim specification adopts the facial animation parameters (FAP), which are first proposed in MPEG4. The following is a simple example which shows how some facial expression, like eyebrow movement and smile can be realized by basing on some ad hoc facial geometrical data. The facial expression is shown in Figure 8.8.

```
eyebrow_move(l,Range):-
getSFVec3f(l_eyebrow,translation,X,Y,Z),
Y1 is Y + Range,
setSFVec3f(l_eyebrow,translation,X,Y1,Z).
eyebrow_move(r,Range):-
getSFVec3f(r_eyebrow,translation,X,Y,Z),
Y1 is Y + Range,
setSFVec3f(r_eyebrow,translation,X,Y1,Z).


smiling(Time):-
smile_point(2, Point2),
setMFVec3f(lower_lip_coordinate, point, Point2),
sleep(Time),
smile_point(1, Point1),
setMFVec3f(lower_lip_coordinate, point, Point1).

smile_point(1,SmilePoint) :-
SmilePoint = [[0.0381, 0.0312, -5.0E-4],
   [0.015, 0.0162, -0.0204],
[0.015, 0.03, 0.0],
[0.0215, 1.0E-4, -8.0E-4],
 [-0.015, 0.0162, -0.0204],
[-0.0381, 0.0312, -5.0E-4],
 [-0.015, 0.03, 0.0],
[-0.0215, 1.0E-4, -8.0E-4]].

smile_point(2,SmilePoint):-
SmilePoint = [[0.04810,0.03920,-0.00050],
[0.01500,0.00020,-0.02040],
[0.01500,0.01400,0.00000],
[0.02150,-0.01610,-0.00080],
[-0.01500,0.00020,-0.02040],
[-0.04810,0.03920,-0.00050],
[-0.01500,0.01400,0.00000],
[-0.02150,-0.01610,-0.00080]].
```

Exercises

8.1 Design a H-anim 1.1 compliant avatar for the soccer playing agents. The avatar should be able to be controlled in DLP to show the following gestures: kicking, greeting, shouting, and ball-holding.

8.2 Design a texture-based humannoid avatar, by using your own photo as the texture of the avatar's face.

8.3 Extend the facial animation example with more facial expressions.

9

STEP : a Scripting Language for Embodied Agents

9.1 MOTIVATION

Embodied agents are autonomous agents which have bodies by which the agents can perceive their world directly through sensors and act on the world directly through effectors. Embodied agents whose experienced worlds are located in real environments, are usually called *cognitive robots*. *Web agents* are embodied agents whose experienced worlds are the Web; typically, they act and collaborate in networked virtual environments. In addition, *3D web agents* are embodied agents whose 3D avatars can interact with each other or with users via Web browsers[Huang et al., 2000].

Embodied agents usually interact with users or each other via multimodal communicative acts, which can be non-verbal or verbal. Gestures, postures and facial expressions are typical non-verbal communicative acts. In general, specifying communicative acts for embodied agents is not easy; they often require a lot of geometrical data and detailed movement equations, say, for the specification of gestures.

In [Huang et al., 2002b] we propose the scripting language STEP (Scripting Technology for Embodied Persona), in particular for communicative acts of embodied agents. At present, we focus on aspects of the specification and modeling of gestures and postures for 3D web agents. However, STEP can be extended for other communicative acts, like facial expressions, speech, and other types of embodied agents, like cognitive robots. Scripting languages are to a certain extent simplified languages which ease the task of computation and reasoning. One of the main advantages of using scripting languages is that the specification of communicative acts can be separated from the programs which specify the agent architecture and mental state reasoning. Thus, changing the specification of communicative acts doesn't require to re-program the agent.

The avatars of 3D web agents are built in the Virtual Reality Modeling Language (VRML). These avatars are usually humanoid-like ones. We have implemented the proposed scripting language for H-anim based humanoids in the distributed logic programming language DLP.

In this chapter, we discuss how STEP can be used for embodied agents. STEP introduces a Prolog-like syntax, which makes it compatible with most standard logic programming languages, whereas the formal semantics of STEP is based on dynamic logic [Harel, 1984]. Thus, STEP has a solid semantic foundation, in spite of a rich number of variants of the compositional operators and interaction facilities on worlds.

9.2 PRINCIPLES

We design the scripting language primarily for the specification of communicative acts for embodied agents. Namely, we separate external-oriented communicative acts from internal changes of the mental states of embodied agents because the former involves only geometrical changes of the body objects and the natural transition of the actions, whereas the latter involves more complicated computation and reasoning. Of course, a question is: why not use the same scripting language for both external gestures and internal agent specification? Our answer is: the scripting language is designed to be a simplified, user-friendly specification language for embodied agents, whereas the formalization of intelligent agents requires a powerful specification and programming language. It's not our intention to design a scripting language with fully-functional computation facilities, like other programming languages as Java, Prolog or DLP. A scripting language should be interoperable with a fully powered agent implementation language, but offer a rather easy way for authoring. Although communicative acts are the result of the internal reasoning of embodied agents, they do not need the same expressiveness of a general programming language. However, we do require that a scripting language should be able to interact with mental states of embodied agents in some ways, which will be discussed in more detail later.

We consider the following design principles for a scripting language.

Principle 1: Convenience As mentioned, the specification of communicative acts, like gestures and facial expressions usually involve a lot of geometrical data, like using ROUTE statements in VRML, or movement equations, like those in computer graphics. A scripting language should hide those geometrical difficulties, so that non-professional authors can use it in a natural way.

For example, suppose that authors want to specify that an agent turns his left arm forward slowly. It can be specified like this:

```
turn(Agent, left_arm, front, slow)
```

It should not be necessary to specify it as follows, which requires knowledge of a coordination system, rotation axis, etc.

```
turn(Agent, left_arm, rotation(1,0,0,1.57), 3)
```

One of the implications of this principle is that embodied agents should be aware of their context. Namely, they should be able to understand what certain indications mean, like the directions 'left' and 'right', or the body parts 'left arm', etc.

Principle 2: Compositional Semantics Specification of composite actions, based on existing components. For example, an action of an agent which turns his arms forward slowly, can be defined in terms of two primitive actions: turn-left-arm and turn-right-arm, like:

```
par([turn(Agent, left_arm, front, slow),
    turn(Agent, right_arm, front, slow)])
```

Typical composite operators for actions are sequence action *seq* , parallel action *par*, repeat action *repeat*, which are used in dynamic logics [Harel, 1984].

Principle 3: Re-definability Scripting actions (i.e.,composite actions), can be defined in terms of other defined actions explicitly. Namely, the scripting language should be a rule-based specification system. Scripting actions are defined with their own names. These defined actions can be re-used for other scripting actions. For example, if we have defined two scripting actions *run* and *kick*, then a new action *run_then_kick* can be defined in terms of *run* and *kick* :

```
run_then_kick(Agent)=
   seq([script(run(Agent)), script(kick(Agent))]).
```

which can be specified in a Prolog-like syntax:

```
script(run_then_kick(Agent), Action):-
   Action = seq([script(run(Agent)),script(kick(Agent))]).
```

Principle 4: Parametrization Scripting actions can be adapted to be other actions. Namely, actions can be specified in terms of how these actions cause changes over time to each individual *degree of freedom*, which is proposed by Perlin and Goldberg in [Perlin and Goldberg, 1996]. For example, suppose that we define a scripting action *run*: we know that running can be done at different paces. It can be a 'fast-run' or 'slow-run'. We should not define all of the run actions for particular paces. We can define the action 'run' with respect to a degree of freedom 'tempo'. Changing the tempo for a generic run action should be enough to achieve a run action with different paces. Another method of parametrization is to introduce variables or parameters in the names of scripting actions, which allows for a similar action with different values. That is one of the reasons why we introduce Prolog-like syntax in STEP.

Principle 5: Interaction Scripting actions should be able to interact with, more exactly, perceive the world, including embodied agents' mental states, to decide whether or not it should continue the current action, or change to other actions, or stop the current action. This kind of interaction modes can be achieved by the introduction of high-level interaction operators, as defined in dynamic logic. The operator 'test' and the operator 'conditional' are useful for the interaction between the actions and the states.

9.3 SCRIPTING LANGUAGE STEP

In this section, we discuss the general aspects of the scripting language STEP.

9.3.1 Reference Systems

Direction Reference The reference system in STEP is based on the H-anim specification: namely, the initial humanoid position should be modeled in a standing position, facing in the +Z direction with +Y up and +X to the humanoid's left. The origin $\langle 0, 0, 0\rangle$ is located at ground level, between the humanoid's feet. The arms should be straight and parallel to the sides of the body with the palms of the hands facing inwards towards the thighs.

Based on the standard pose of the humanoid, we can define the direction reference system as sketched in figure 9.1. The direction reference system is based on these three dimensions: front vs. back which corresponds to the Z-axis, up vs. down which corresponds to the Y-axis, and left vs. right which corresponds to the X-axis. Based on these three dimensions, we can introduce a more natural-language-like direction reference scheme, say, turning left-arm to 'front-up', is to turn the left-arm such that the front-end of the arm will point to the up front direction. Figure 9.2 shows several combinations of directions based on these three dimensions for the left-arm. The direction references for other body parts are similar. These combinations are designed for convenience and are discussed in Section 9.2. However, they are in general not sufficient for more complex applications. To solve this kind of problem, we introduce interpolations with respect to the mentioned direction references. For instance, the direction 'left_front2' is referred to as one which is located

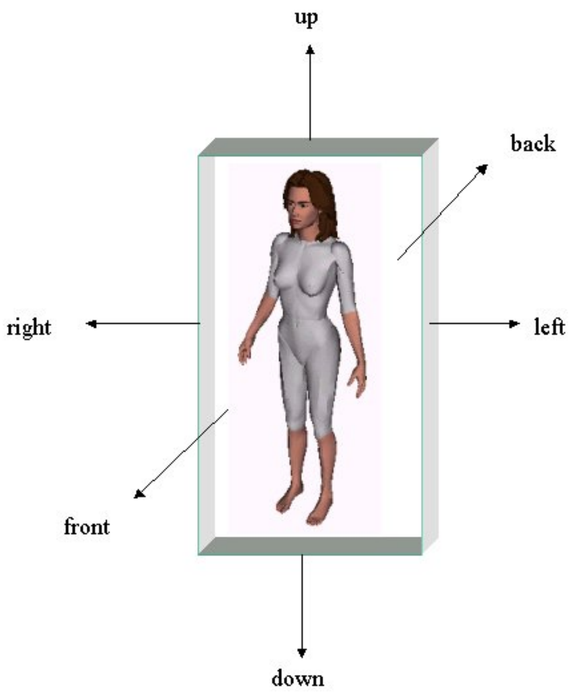


Fig. 9.1 Direction Reference for Humanoid

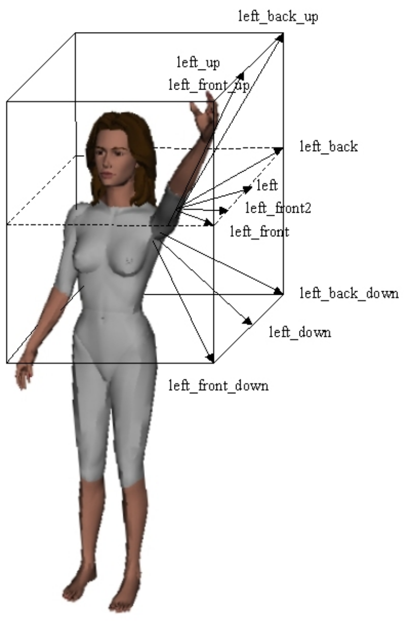


Fig. 9.2 Combination of the Directions for Left Arm

between 'left_front' and 'left', which is shown in Figure 9.2. Natural-language-like references are convenient for authors to specify scripting actions, since they do not require the author to have a detailed knowledge of reference systems in VRML. Moreover, the proposed scripting language also supports the orginal VRML reference system, which is useful for experienced authors. Directions can also be specified to be a four-place tuple $\langle X, Y, Z, R\rangle$, say, $rotation(1, 0, 0, 1.57)$.

Body Reference An H-anim specification contains a set of *Joint nodes* that are arranged to form a hierarchy. Each Joint node can contain other Joint nodes and may also contain a *Segment node* which describes the body part associated with that joint. Each Segment can also have a number of Site nodes, which define locations relative to the segment. Sites can be used for attaching accessories, like hat, clothing and jewelry. In addition, they can be used to define eye points and viewpoint locations. Each Segment node can have a number of *Displacer nodes*, that specify which vertices within the segment correspond to a particular feature or configuration of vertices.

Figure 9.3 shows several typical joints of humanoids. Therefore, turning body parts of humanoids implies the setting of the relevant joint's rotation. Body moving means the setting of the HumanoidRoot to a new position. For instance, the action 'turning the left-arm to the front slowly' is specified as:

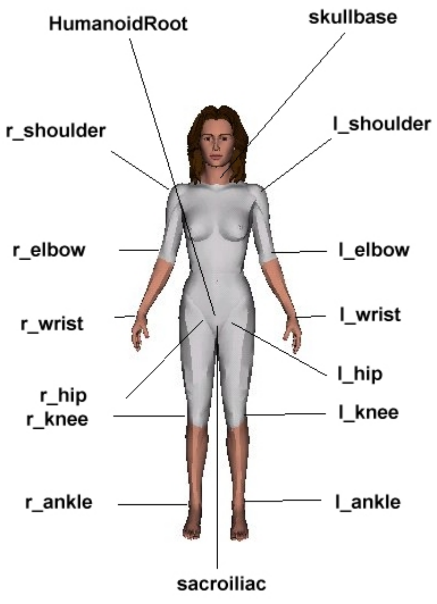


Fig. 9.3 Typical Joints for Humanoid

```
turn(Agent, l_shoulder, front, slow)
```

Time Reference STEP has the same time reference system as that in VRML. For example, the action *turning the left arm to the front in 2 seconds* can be specified as:

```
turn(Agent, l_shoulder, front, time(2, second))
```

This kind of explicit specification of duration in scripting actions does not satisfy the parametrization principle. Therefore, we introduce a more flexible time reference system based on the notions of beat and tempo. A *beat* is a time interval for body movements, whereas the *tempo* is the number of beats per minute. By default, the tempo is set to 60. Namely, a beat corresponds to a second by default. However, the tempo can be changed. Moreover, we can define different speeds for body movements, say, the speed 'fast' can be defined as one beat, whereas the speed 'slow' can be defined as three beats.

9.3.2 Primitive Actions and Composite Operators

Turn and move are the two main primitive actions for body movements. Turn actions specify the change of the rotations of the body parts or the whole body over time, whereas move actions specify the change of the positions of the body parts or the whole body over time. A turn action is defined as follows:

```
turn(Agent,BodyPart,Direction,Duration)
```

where `Direction` can be a natural-language-like direction like 'front' or a rotation value like 'rotation(1,0,0,3.14)', `Duration` can be a speed name like 'fast' or an explicit time specification, like 'time(2,second)'.

A move action is defined as:

```
move(Agent,BodyPart,Direction,Duration)
```

where `Direction` can be a natural-language-like direction, like 'front', a position value like 'position(1,0,10)', or an increment value like 'increment(1,0,0)'.

Here are typical composite operators for scripting actions:

- Sequence operator 'seq': the action seq([$Action_1$, ...,$Action_n$]) denotes a composite action in which $Action_1$, ...,and $Action_n$ are executed sequentially, like:

  ```
  seq([turn(agent,l_shoulder,front,fast),
  turn(agent,r_shoulder,front,fast)])
  ```

- Parallel operator 'par': the action par([$Action_1$, ...,$Action_n$]) denotes a composite action in which $Action_1$, ...,and $Action_n$ are executed simultaneously.

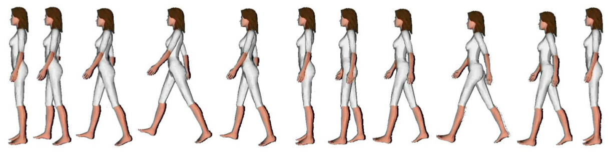


Fig. 9.4 Walk

- non-deterministic choice operator 'choice': the action `choice([`$Action_1$`, ...,`$Action_n$`])` denotes a composite action in which one of the $Action_1$, ...,and $Action_n$ is executed.

- repeat operator 'repeat': the action `repeat(Action, T)` denotes a composite action in which the *Action* is repeated T times.

9.3.3 High-level Interaction Operators

When using high-level interaction operators, scripting actions can directly interact with internal states of embodies agents or with external states of worlds. These interaction operators are based on a meta language which is used to build embodied agents, say, in the distributed logic programming language DLP. In the following, we use lower case Greek letters ϕ, ψ, χ to denote formulas in the meta language. Examples of several higher-level interaction operators:

- test: `test(`ϕ`)`, check the state ϕ. If ϕ holds then skip, otherwise fail.

- execution: `do(`ϕ`)`, make the state ϕ true, i.e. execute ϕ in the meta language.

- conditional: `if_then_else(`ϕ`,`$action_1$`,`$action_2$`)`.

- until: `until(action,`ϕ`)`: take action until ϕ holds.

We have implemented the scripting language STEP in the distributed logic programming language DLP.

9.4 EXAMPLES

9.4.1 Walk and its Variants

A walking posture can be simply expressed as a movement which exchanges the following two main poses: a pose in which the left-arm/right-leg move forward while the right-arm/left-leg move backward, and a pose in which the right-arm/left-leg move forward while the left-arm/right-leg move backward.

The main poses and their linear interpolations are shown in Figure 9.4. The walk action can be described in the scripting language as follows:

```
script(walk_pose(Agent), Action):-
   Action = seq([par([
       turn(Agent,r_shoulder,back_down2,fast),
       turn(Agent,r_hip,front_down2,fast),
       turn(Agent,l_shoulder,front_down2,fast),
       turn(Agent,l_hip,back_down2,fast)]),
    par([turn(Agent,l_shoulder,back_down2,fast),
       turn(Agent,l_hip,front_down2,fast),
       turn(Agent,r_shoulder,front_down2,fast),
       turn(Agent,r_hip,back_down2,fast)])]).
```

Thus, a walk step can be described to be as a parallel action which consists of the walking posture and the moving action (i.e., changing position) as follows:

```
script(walk_forward_step(Agent),Action):-
  Action= par([script_action(walk_pose(Agent)),
          move(Agent,front,fast)]).
```

The step length can be a concrete value. For example, for the step length with 0.7 meter, it can be defined as follows:

```
script(walk_forward_step07(Agent),Action):-
   Action= par([script_action(walk_pose(Agent)),
         move(Agent,increment(0.0,0.0,0.7),fast)]).
```

Alternatively, the step length can also be a variable like:

```
script(walk_forward_step0(Agent,StepLength),Action):-
  Action = par([script_action(walk_pose(Agent)),
     move(Agent,increment(0.0,0.0,StepLength),fast)]).
```

Therefore, the walking forward N steps with the *StepLength* can be defined as follows:

```
script(walk_forward(Agent,StepLength,N),Action):-
   Action = repeat(script_action(
      walk_forward_step0(Agent,StepLength)),N).
```

As mentioned above, the animations of the walk based on these definitions are just simplified and approximated ones. As analysed in [Faure, 1997], a realistic animation of the walk motions of human figure involves a lot of computations which rely on a robust simulator where forward and inverse kinematics are combined with automatic collision detection and response. We do not want to use the scripting language to achieve a fully realistic animation of

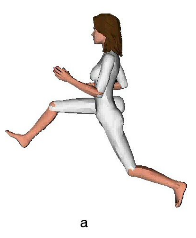


Fig. 9.5 Poses of Run

the walk action, because they are seldom necessary for most web applications. However, we would like to point out that there does exist the possibility to accommodate some inverse kinematics to improve the realism by using the scripting language.

9.4.2 Run and its Deformation

The action 'run' is similar to 'walk', however, with a bigger wave of the lower-arms and the lower-legs, which is shown in Figure 9.5a. As we can see from the figure, the left lower-arm points to the direction 'front-up' when the left upper-arm points to the direction 'front_down2' during the run. Consider the hierarchies of the body parts, we should not use the primitive action $turn(Agent, l\_elbow, front\_up, fast)$ but the primitive action $turn(Agent, l\_elbow, front, fast)$, for the direction of the left lower-arm should be defined with respect to the default direction of its parent body part, i.e., the left arm (more exactly, the joint l_shoulder). That kind of re-direction would not cause big difficulties for authoring, for the correct direction can be obtained by reducing the directions of its parent body parts to be the default ones. As we can see in Figure 9.5b, the lower-arm actually points to the direction 'front'.

The action 'run_pose' can be simply defined as an action which starts with a basic run pose as shown in Figure 9.5b and then repeat the action 'walk_pose' for N times as follows:

```
script(basic_run_pose(Agent), Action):-
 Action=par([turn(Agent,r_elbow,front,fast),
      turn(Agent, l_elbow, front, fast),
      turn(Agent, l_hip, front_down2, fast),
      turn(Agent, r_hip, front_down2, fast),
      turn(Agent, l_knee, back_down, fast),
      turn(Agent, r_knee, back_down, fast)]).

script(run_pose(Agent,N),Action):-
    Action = seq([script_action(basic_run_pose(Agent)),
      repeat(script_action(walk_pose(Agent)),N)]).
```

Therefore, the action running forward *N* steps with the *StepLength* can be defined in the scripting language as follows:

```
script(run(Agent, StepLength,N),Action):-
 Action=seq([script_action(basic_run_pose(Agent)),
   script_action(walk_forward(Agent,StepLength,N))]).
```

Actually, the action 'run' may have a lot of variants. For instances, the lower-arm may point to different directions. They would not necessarily point to the direction 'front'. Therefore, we may define the action 'run' with respect to certain degrees of freedom. Here is an example to define a degree of freedom with respect to the angle of the lower arms to achieve the deformation.

```
script(basic_run_pose_elbow(Agent,Elbow_Angle),Action):-
   Action = par([
     turn(Agent,r_elbow,rotation(1,0,0,Elbow_Angle),fast),
     turn(Agent,l_elbow,rotation(1,0,0,Elbow_Angle),fast),
     turn(Agent,l_hip,front_down2,fast),
     turn(Agent,r_hip,front_down2,fast),
     turn(Agent,l_knee,back_down,fast),
     turn(Agent,r_knee,back_down,fast)]).

script(run_e(Agent,StepLength,N,Elbow_Angle),Action):-
   Action = seq([script_action(
      basic_run_pose_elbow(Agent,Elbow_Angle)),
      script_action(walk_forward(Agent, StepLength, N))]).
```

9.4.3 Tai Chi

In this subsection, we will discuss an application example, the development of an instructional VR for Tai Chi, illustrating how our approach allows for the creation of reusable libraries of behavior patterns.

A Tai Chi exercise is a sequence of exercise stages. Each stage consists of a group of postures. These postures can be defined in terms of their body part movements. Figure 9.6 shows several typical postures of Tai Chi. For instance, the beginning-posture and the push-down-posture can be defined as follows:

```
script(taichi(Agent,beginning_posture),Action):-
    Action =seq([
              turn(Agent,l_hip,side1_down,fast),
              turn(Agent,r_hip,side1_down,fast),
            par([turn(Agent,l_shoulder,front,slow),
        turn(Agent,r_shoulder,front,slow)])]).

script(taichi(Agent,push_down_posture),Action) :-
```

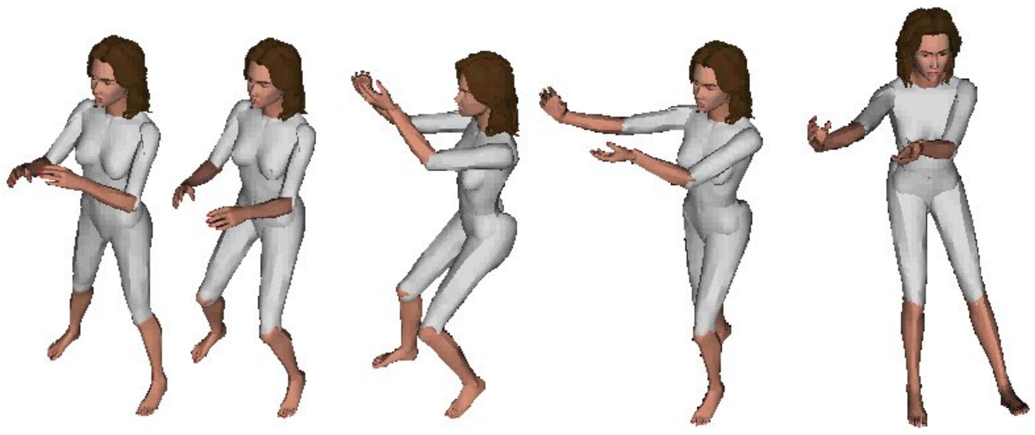


Fig. 9.6 Tai Chi

```
   Action =seq([
       par([turn(Agent,l_shoulder,front_down,slow),
            turn(Agent,r_shoulder,front_down,slow),
turn(Agent,l_elbow,front_right2,slow),
turn(Agent,r_elbow,front_left2,slow)]),
 par([turn(Agent,l_hip,left_front_down,slow),
      turn(Agent,r_hip,right_front_down,slow),
turn(Agent,l_elbow,right_front_down,slow),
turn(Agent,r_shoulder,front_down2,slow),
turn(Agent,l_knee,back2_down,slow),
turn(Agent,r_knee,back2_down,slow)])]).
```

Those defined posture can be used to define the stages like this:

```
script(taichi(Agent, stage1), Action) :-
   Action =seq([
         script(taichi(Agent,beginnin_posture),
         script(taichi(Agent,push_down_posture),
         ......
            ]).
```

Furthermore, those scripting actions can be used to define more complex Tai Chi exercise as follows:

```
script(taichi(Agent), Action) :-
    Action = seq([
        do(display('Taichi exercise ...~n')),
  script(taichi(Agent, stage1)),
        script(taichi(Agent, stage2))
        ......
              ]).
```

Combined with the interaction operators, like do-operator, with built-in predicates in mete-language, the instructional actions become more attractive. Moveover, the agent names in those scripting actions are defined as variables. These varibales can be instantiated with different agent names in different applications. Thus, the same scripting actions can be re-used for different avatars for different applications.

9.4.4 Interaction with Other Agents

Just consider a situation in which two agents have to move a 'heavy' table together. This scripting action 'moving-heavy-table' can be designed to be ones which consist of the following several steps (i.e., sub-actions): first walk to the table, then hold the table, and finally move the table around. Using the scripting language, it is not difficult to define those sub-actions, they can be done just like the other examples above, like walk and run. A solution to define the action 'moving-heavy-table' which involves multiple agents can be as follows [1]:

```
script(move_heavy_table(Agent1,Agent2,Table,
  NewPos), Action):-
       Action = seq([par([
            script(walk_to(Agent1,Table)),
            script(walk_to(Agent2,Table))]),
       par([script(hold(Agent1,Table,left)),
            script(hold(Agent2,Table,right))]),
       par([move(Agent1,NewPosition,slow),
            move(Agent2,NewPosition,slow),
            do(object_move(Table,NewPos,slow))
                  ])]).
```

The solution above is not a good solution if we consider this action is a co-operating action between two agents. Namely, this kind of actions should not be achieved by this kind of pre-defined actions but by certain communicative/negotiation procedures. Hence, the scripting action should be considered as an action which involves only the agent itself but not other agents. Anything the agent need from others can only be achieved via its communicative actions with others or wait until certain conditions meet. Therefore, the cooperating action 'moving-heavy-table' should be defined by the following procedure, first the agent walks to the table and holds one of the end of the

[1]We omit the details about the definitions of the actions like walk_to, hold, etc.

table[2], next, wait until the partner holds another end of the table, then moves the table. It can be defined as follows:

```
script(move_heavy_table(Agent,Partner,
    Table, NewPos), Action):-
       Action=seq([script(walk_to(Agent,Table)),
         if_then_else(not(hold(Partner,Table,left)),
                    script(hold(Agent,Table,left)),
                    script(hold(Agent,Table,right))),
       until(wait,hold(Partner,Table,_)),
       par([move(Agent,NewPos,slow),
            do(object_move(Table,NewPos,slow))]
                )]).
```

9.5 XSTEP: THE XML-ENCODED STEP

We are also developing XSTEP, an XML encoding for STEP. We use *seq* and *par* tags as found in SMIL[3], as well as action tags with appropriate attributes for speed, direction and body parts involved. As an example, look at the XSTEP specification of the *walk* action.

```
<action name="walk(Agent)">
      <seq>
        <par>
          <turn actor="Agent" part="r_shoulder">
            <dir value="back_down2"/>
            <speed value="fast"/>
          </turn>
          <turn actor="Agent" part="r_hip">
            <dir value="front_down2"/>
            <speed value="fast"/>
          </turn>
          <turn actor="Agent" part="l_shoulder">
            <speed value="fast"/>
            <dir value="front_down2"/>
          </turn>
          <turn actor="Agent" part="l_hip">
            <dir value="back_down2"/>
            <speed value="fast"/>
          </turn>
```

[2]Here we consider only a simplified scenario. We do not consider the deadlock problem here, like the that in the *philosopher dinner problem*.
[3]http://www.w3.org/AudioVideo

```
      </par>
       ......
    </seq>
  </action>
```

Similar as with the specification of dialog phrases, such a specification is translated into the corresponding DLP code, which is loaded with the scene it belongs to. For XSTEP we have developed an XSLT stylesheet, using the Saxon[4] package, that transforms an XSTEP specification into DLP. We plan to incorporate XML-processing capabilities in DLP, so that such specifications can be loaded dynamically.

9.6 IMPLEMENTATION ISSUES

We have implemented the scripting language STEP in the distributed logic programming language DLP. In this section, we discuss several implementation and performance issues. First we will discuss the module architectures of STEP . Scripting actions are defined as a sequence or parallel set of actions. One of the main issues is how to implement parallel actions with a satisfying performance. Another issue is which interpolation method should be used to achieve smooth transitions from an initial state to a final state.

9.6.1 STEP Components

STEP is designed for multiple purpose use. It serves as an animation/action engine, which can be embodied as a component in embodied agents, or can also be located at the controlling component at XSTEP, the XML-based markup language.

STEP consists of the following components:

- **Action library**: The action library is the collections of the scripting actions, which can be of user defined or of system built-in.
- **STEP ontology**: The STEP ontology component defines the semantic meanings of the STEP reference systems. So-called *Ontology* is a description of the concepts or bodies of knowledge understood by a particular community and the relationships between those concepts. The STEP body ontological specification is based on H-anim specification. The STEP ontology component also defines the semantic meaning of the direction reference system. For instance, the semantic interpretation of the direction 'front' can be defined in the ontology component as follows:

[4]http://saxon.sourceforge.com

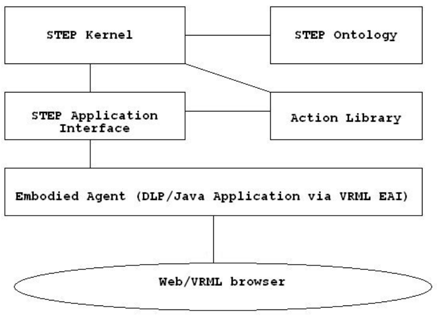

STEP Kernel
STEP Ontology
STEP Application Interface
Action Library
Embodied Agent (DLP/Java Application via VRML EAI)
Web/VRML browser

Fig. 9.7 STEP and its interface with embodied agents

```
rotationParameter(_,front,rotation(1,0,0,-1.57)).
```

Namely, turning a body part to 'front' is equal to setting its rotation to (1, 0, 0, -1.57). Separated ontology specification component would make STEP more convenient for the extension/change of its ontology. It is also a solution to the maintenance of the interoperability of the scripting langauge over the Web.

- **STEP Kernel**: The STEP kernel is the central controlling component of STEP . It translates scripting actions into executable VRML/X3D EAI commands based on the semantics of the action operators and the ontological definitions of the reference terms. The STEP kernel and the STEP ontology component are application-independent. The two components together is called the *STEP engine.*

- **STEP application interface**: The STEP application interface component offers the interface operators for users/applications. The scripting actions can be called by using these application interface operators. They can be java-applet-based, or java-script-based, or XML-based.

The avatars of VRML/X3D-based embodied agents are displayed in a VRML/X3D browser. These embodied agents are usually designed as java applets which are embodied in the browser. They interact with virtual environments via VRML/X3D External Application Interface (EAI). STEP is also designed as the part of the java applets, which can be called by embodied agents via the step interface component. STEP module architecture and its interface with embodied agents is shown in Figure 9.7.

9.6.2 Parallelism and Synchronization

How to implement parallel actions with a satisfying performance is an important issue for a scripting language. A naive solution is to create a new thread

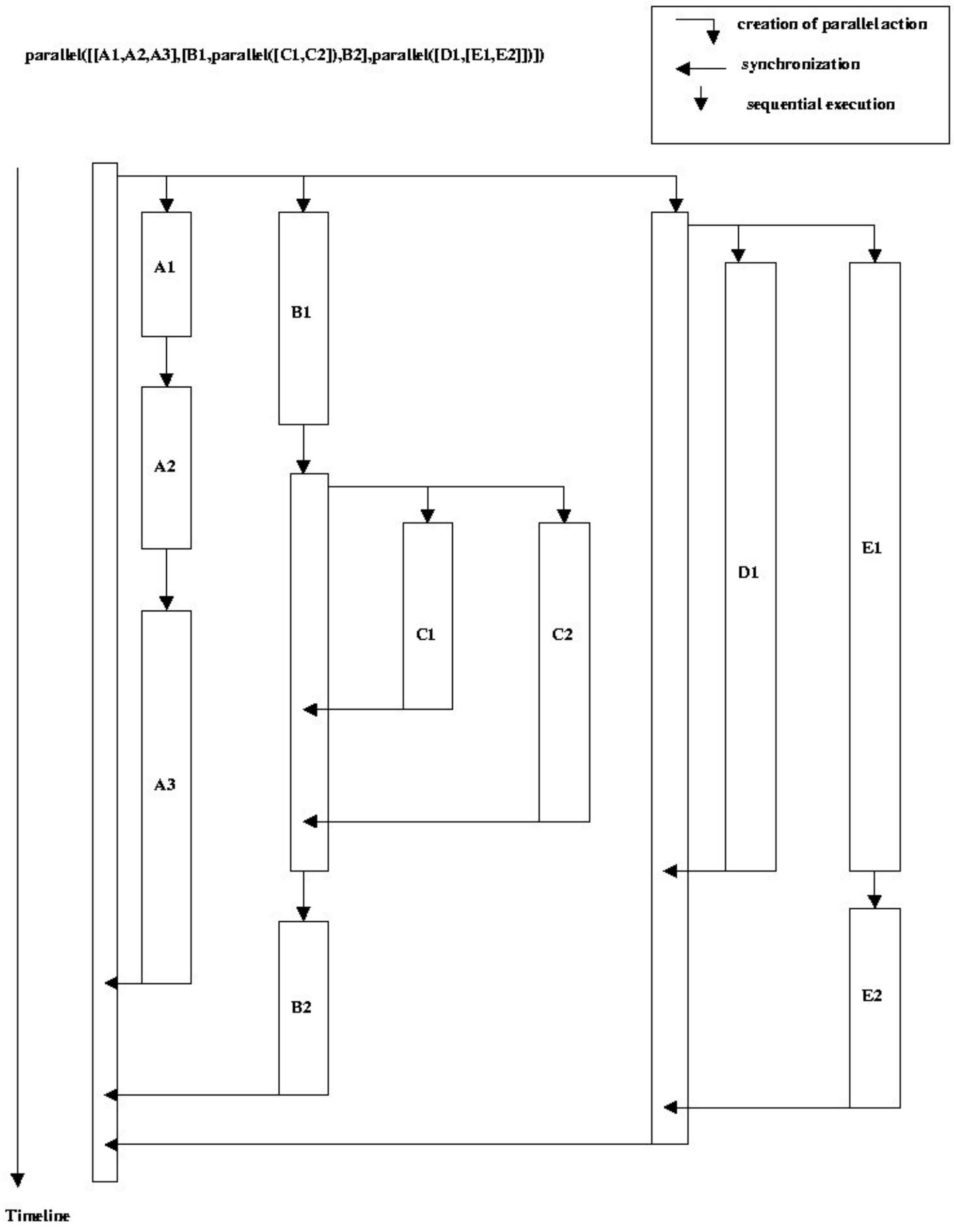

parallel([[A1,A2,A3],[B1,parallel([C1,C2]),B2],parallel([D1,[E1,E2]])])
creation of parallel action
synchronization
sequential execution
A1
A2
A3
B1
B2
C1
C2
D1
E1
E2
Timeline

Fig. 9.8 Processing Parallel Actions

for each component action of parallel actions. However, this naive solution will result in a serious performance problem, because a lot of threads will be created and killed frequently. Our solution to handle this problem is to create only a limited number of threads, which are called *parallel threads*. The system assigns component actions of parallel actions to one of the parallel threads by a certain scheduling procedure. We have to consider the following issues with respect to the scheduling procedure: the correctness of the scheduling procedure and its performance. The former implies that the resulting action should be semantically correct. It should be at least order-preserving. Namely, an intended late action should not be executed before an intended early action. In general, all actions should be executed in due time, with the sense that they are never executed too early or too late.

In general, a set of composite sequential and parallel actions can be depicted as an execution *graph*; each node in a graph represents either a sequential or a parallel activity. Execution graphs indicate exactly when a particular action is invoked or when (parallel) actions synchronize. Parallel actions can be nested, i.e. a parallel action can contain other parallel activities. Therefore, synchronization always takes place relative to the parent node in the execution graph. This way, the scheduling and synchronization scheme as imposed by the execution graph preserves the relative order of actions.

From a script point of view, a parallel action is finished when each of the individual activities are done. However, action resources are reclaimed and re-allocated incrementally. After synchronization of a particular activity in a parallel action construct its resources can be re-used immediately for other scripting purposes. Just consider a nested parallel action which consists of several sequential and parallel sub-actions, like,

$par([seq([A1, A2, A3]), seq([B1, par([C1, C2]), B2]),$
$par([D1, seq([E1, E2])])])$

The procedure of the scheduling and synchronization is shown in Figure 9.8.

9.6.3 Rotation Interpolation

We use DLP to implement the scripting language STEP . One of the issues on the implementation is to achieve the function of turn-object by introducing an appropriate interpolation between the starting rotation and the ending rotation.

Suppose that the object's current rotation is

$$R_s = \langle X_0, Y_0, Z_0, R_0 \rangle$$

and the ending rotation of the scripting action is

$$R_e = \langle X, Y, Z, R \rangle,$$

and the number of interpolations is I. In STEP , we use slerp (spherical interpolation) on unit quaternions to solve this problem. Let $Q_s = \langle w_0, x_0, y_0, z_0 \rangle$ and $Q_e = \langle w, x, y, z \rangle$ be the corresponding quaternions of the rotations R_s and R_e. The relation between a rotation $\langle X, Y, Z, R \rangle$ and a quaternion $\langle w, x, y, z \rangle$ is as follows:

$w = cos(R/2)$.
$x = X \times sin(R/2)$.
$y = Y \times sin(R/2)$.
$z = Z \times sin(R/2)$.

The function slerp does a spherical linear interpolation between two quaternions Q_s and Q_e by an amount $T \in [0, 1]$:

$slerp(T, Q_s, Q_e) = R_s \times sin((1 - T) \times \Omega)/sin(\Omega) +$
$R_e \times sin(T \times \Omega)/sin(\Omega)$

where $cos(\Omega) = Q_s \cdot Q_e = w_0 \times w + x_0 \times x + y_0 \times y + z_0 \times z$. See [Schoemake, 1985] for more details of the background knowledge on quaternions and slerp.

One of the requirement to achieve a natural transition of the two rotations is to introduce the non-linear interpolation between two rotations. STEP also allows users to introduce their own non-linear interpolation by using the enumerating type of interpolation operator. An example:

```
turnEx(Agent,l_shoulder,front,fast,
       enum([0,0.1,0.15,0.2,0.8,1]))
```

would turn the agent's left arm to the front via the interpolating points 0, 0.1, 0.15, 0.2,0.8,1.

Users can use the interaction operators *do* to calculate interpolating point lists for their own interpolation function. Therefore, the enumerating list is powerful enough to represent arbitrary discrete interpolation function.

9.7 CONCLUSIONS

In this chapter we have discussed the scripting language STEP for embodied agents, in particular for their communicative acts, like gestures and postures. Moreover, we have discussed principles of scripting language design for embodied agents and several aspects of the application of the scripting language.

STEP is close to Perlin and Goldberg's **Improv** system. In [Perlin and Goldberg, 1996], Perlin and Goldberg propose **Improv**, which is a system for scripting interactive actors in virtual worlds. STEP is different from Perlin and Goldberg's in the following aspects: First, STEP is based on the H-anim specification, thus, VRML-based, which is convenient for Web applications. Secondly, we separate the scripting language from the agent architecture. Therefore, it's relatively easy for users to use the scripting language.

Prendinger et al. are also using a Prolog-based scripting approach for animated characters but they focus on higher-level concepts such as affect and social context[Prendiner et al., 2002]. STEP shares a number of interests with the VHML(Virtual Human Markup Language) community[5], which is developing a suite of markup language for expressing humanoid behavior, including facial animation, body animation, speech, emotional representation, and multimedia. We see this activity as complementary to ours, since our research proceeds from technical feasibility, that is how we can capture the semantics of humanoid gestures and movements within our dynamic logic, which is implemented on top of DLP.

[5]http://www.vhml.org

Part III

Virtual Communities

10 Virtual Communities

10.1 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 [VESL], Web-based multi-user games [MiMaze], on-line entertainment [Blaxxun, 2000], and e-commerce [Messmer]. Examples of popular 3D virtual community servers are Active World [ActiveWorld, 2000] and Blaxxun Interactive [Blaxxun, 2000].

The term "virtual community" is usually used to refer to the general appearance and gathering of people over distributed computer systems, in particular, on the Internet. A typical text-based virtual community is Internet Relay Chatting [Liu, 1999], whereas typical 3D web-based virtual community are the VRML-based ones, like the Blaxxun community server, DeepMatrix [Reitmayr et al., 1999], VLNet [Capin et al., 1997]. In VRML-based virtual communities, virtual worlds are designed by using VRML, whereas the VRML External Authoring Interface (EAI) connects the Java Virtual Machine running in a Web Browser to execute applets and plug-ins are used to control the virtual worlds.

Virtual communities usually have a client-server network architecture. In particular, they occasionally use a centralized server architecture, for the clients are Java applets running in a remote Web Browser, and the Java platform security policy allows clients only to connect to the originating host.

10.2 LIVING WORLDS

The "Living Worlds" Working Group describes a general concept and context of VRML-based virtual communities [Living Worlds]. In Living worlds, a component called *MUTech*, or alternatively called *Multi-User Technology*, is used to implement shared behaviors/states across the network. MUTechs provide all network communication needed for multi-user interaction beyond that provided by the Browser itself. 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 *zone* is a contiguous portion of a scene. A *SharedObject*, or shared object, is an object whose state and behavior are to be synchronized across multiple clients. Thus, a zone is a container for SharedObject. The SharedObject on one of these clients are 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, Namely, a *drone* an instance of a SharedObject replicating the state or behavior of another instance, its pilot. In Living Worlds, a avatar is defined as a SharedObject whose pilot is under real-time user control. However, in agent-based virtual communitity, which would be discussed in the next section, an avatar can be a representation of a human user, or an autonomous agent.

10.3 AGENT-BASED VIRTUAL COMMUNITY

Most existing virtual community 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 [Broll et al., 2000, Earnshaw, et al., 1998, Watson, 1996].

In [Huang et al., 2002], an approach to 3D agent-based virtual communities has been proposed. So-called *agent-based virtual community* has the following two shades of meaning :

- *Virtual environments 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 environments supported by ACL communication*: Agent communication languages 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. Moreover, 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 11.3.

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 distinguished into 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. Of course, 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 the section 11.2.
- 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 components which are in charge of passing communication messages. However, we would not call them agents, but *communication components/threads*, for the agents are interested in the communications only with other agents, rather than the communication facilities themselves.

11

DLP for Virtual Communities

11.1 DLP NETWORKING PREDICATES

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

The followings are TCP/IP networking predicates in DLP:

- Host-identification predicate $host_address(HostName, InternetAddress)$ gets HostName with the name of the host from which the current program objects are loaded.
- Server predicate $tcp_server(ServerPort, ServerSocket)$ creates a server socket on a server port.
- Server acceptance predicate $tcp_accept(ServerSocket, ServerStreamIn, ServerStreamOut)$ create a stream for messages input and a stream for message output on the server socket.
- Client predicates $tcp_client(ServerHostName, ServerPort, TimeOut, ClientStreamIn, ClientStreamOut)$ create a stream for message input and a stream for message output from the client on the host at the server port.
- Bi-Directional Client / Server Communication predicates: $tcp_get_term(StreamIn, Term)$ gets a message term from the stream, and $tcp_put_term(StreamOut, Term)$ puts a message term on the stream.

The TCP networking predicates are primarily intended for internal , i.e. DLP run-time system use only. The primitives are used in the current DLP run-time system to implement a high-level distributed communication framework. However, occasionally it will be useful to implement a dedicated distributed communication infrastructure. The following source code example illustrates how TCP predicates can be used to build special purpose distributed processing systems in DLP itself. The program example consists of two objects: pxserver and pxclient.

Object pxserver is used as a stand-alone program; once invoked by a "dlp pxserver" command, it will handle multiple pxclient connections. Each client connection will be processed by a separate pxserver thread :

```
:- object pxserver.

    var port = 4321.

    main :-
     tcp_server(port, ServerSocket),
     format('Server ~w : socket = ~w~ n', [this, ServerSocket]),
     new(pxserver(ServerSocket), _),
     format('end of ~w main~ n', [this]).

   pxserver(ServerSocket) :-
    server_init(ServerSocket, ServerStreamIn, ServerStreamOut),
    server_loop(ServerStreamIn, ServerStreamOut).

   server_init(ServerSocket, ServerStreamIn, ServerStreamOut) :-
    tcp_accept(ServerSocket, ServerStreamIn, ServerStreamOut, ClientHostNa
    format('Server ~w : accepting new client connection from ~w~ n',
           [this, ClientHostName]),
     new(pxserver(ServerSocket),_).

   server_loop(ServerStreamIn, ServerStreamOut) :-
    repeat,
     tcp_get_term(ServerStreamIn, ClientRequest),
     process_mesg(ClientRequest, ServerReply),
     tcp_put_term(ServerStreamOut, ServerReply),
    ServerReply = thread_shutdown,
    !,
    tcp_close(ServerStreamIn),
    tcp_close(ServerStreamOut).

    process_mesg(Input, Reply):-
     format('Server ~w : tcp get = ~w~ n', [this, Input]),
                message_type(Input, Reply),
                format('Server ~w : tcp put = ~w~ n', [this, Reply]).
```

```
        message_type(mul_int(I0,I1), ReplyTerm) :-
                      integer(I0),
                      integer(I1),
                      !,
                      IR is I0 * I1,
                      ReplyTerm = mul_val(IR).

        message_type(InputTerm, ReplyTerm) :-
                      InputTerm = [client, [_]],
                      !,
                      ReplyTerm = [server, [ServerHost]],
                     local_host(ServerHost, _).

        message_type(client_shutdown, thread_shutdown) :- !,
                       format('~ nServer Thread ~w shutdown~ n', [this]).

        message_type(Input, Reply) :-
                       Reply = unknown_message_type(Input).

:- end_object pxserver.
```

A pxclient object can execute as a stand-alone client or can execute in a browser. Typically, several active pxclient objects will connect to a single multi-threaded pxserver object simultaneously. In case one or more pxclient objects are running in a browser on different machines, the pxserver should run on the same machine from which the pxclient code is loaded.

```
:- object pxclient.

        var port = 4321, loop = 5 .

        main :-
                       text_area(BrowserTextArea),
                       set_output(BrowserTextArea),
                       code_base_host(ServerHost),
                       pxclient(ServerHost, port),
                       format('end of ~w main~ n', [this]).

        pxclient(ServerHost, ServerPort) :-
                       format('Code base host = ~w~ n', [ServerHost]),
                       tcp_client(ServerHost, ServerPort, 10, StreamIn, StreamOut),
                       format('client socket = ~w, ~w~ n', [StreamIn,StreamOut]),
                       client_init(StreamIn, StreamOut),
                       client_loop(StreamIn, StreamOut),
                       client_exit(StreamIn, StreamOut),
                       tcp_close(StreamIn),
                       tcp_close(StreamOut).

        client_init(StreamIn, StreamOut) :-
                       local_host(ClientHost,_),
                       format('Client host = ~w~ n', [ClientHost]),
                       tcp_put_term(StreamOut, [client, [ClientHost]]),
                       tcp_get_term(StreamIn, OK),
                       format('server answer = ~w~ n', [OK]).

        client_loop(StreamIn, StreamOut) :-
                       repeat,
                         ServerInput = mul_int(loop, 100),
                         format('Client ~w : tcp put = ~w~ n', [this, ServerInpu
                         tcp_put_term(StreamOut, ServerInput),
                         tcp_get_term(StreamIn,  ServerReply),
                         format('Client ~w : tcp get = ~w~ n', [this, ServerRepl
                         -- loop,
                       loop =:= 0,
                       !.

        client_exit(StreamIn, StreamOut) :-
                       tcp_put_term(StreamOut, client_shutdown),
                       tcp_get_term(StreamIn, thread_shutdown).

:- end_object pxclient.
```

Another (somewhat unusual) example of running the client - server objects is shown below. When the object pxsock is executed by means of a "dlp pxsock" command, it will create an active pxserver as well as several active pxclient threads. These active threads will communicate within the context of a single program by means of the established socket connections.

```
:- object pxsock.

        var port = 4321.

        main:-
                        tcp_server(port, Socket),
                        local_host(Host, _),
                        new(pxserver(Socket), _),
                        new(pxclient(Host,port), _),
                        new(pxclient(Host,port), _),
                        new(pxclient(Host,port),_),
                        format('end of ~w main~ n', [this]).

:- end_object\ pxsock.
```

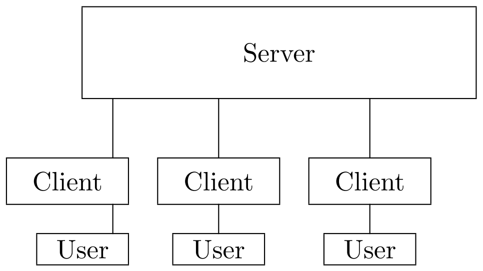


Fig. 11.1 virtual community based on server-client mode

11.2 DISTRIBUTED COMMUNICATION

In general, a virtual community based on server-client network architecture works like this: all of the clients/users connect with a centralized server via a Web browser, usually by means of a TCP connection. The server receives, processes, and transfers the messages about shared objects to the clients for the necessary update. The network architecture of the server-client based virtual community is shown in figure 11.1.

To improve the performance, multiple threads of control are introduced in both server and clients for the virtual communities, which is shown in Figure 11.2. Each thread has its own message queue to keep the coming-messages (sent from other threads) which have not yet been processed. Thus, sending a message to a thread means just putting the message to the recipient's message queue. The predicates *get_queue* and *set_queue* are used to get and set a message from the queue respectively. Each client has its own communication thread, called *client thread*, which is in charge of the network communication. Moreover, for each client a special thread called *server thread*, is created for the network communication with its corresponding client thread. The introduction of multiple threads leads to the following more sophisticated communication types:

- communication between internal threads: messages are sent from a thread to another thread inside server or clients. This kind of communication is done by directly calling one of the predicates *get_queue* and *set_queue*.
- communication between threads across the 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 put to the corresponding server thread's message queue;
2. the server thread uses *get_queue* to get the message from the message queue;
3. the server thread used *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 uses *tcp_put_term* to put the message to the destination thread's message queue;
6. the destination thread uses *get_queue* to get 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 are no direct connection between two clients, this kind of communications have to be achieved via the server.

Moreover, server/clients are designed to consist of two main components: a general component, called gg-server and gg-client, which deals with the network communication and an application specific component, say, 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 to achieve 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 11.3.

For agent-based virtual communities, agent communication languages (ACL) are used to serve as a high level communication facility. KQML [Finin and Fritzson, 1994] and FIPA ACL [FIPA, 1999], which are based on speech act theory [Searle, 1969], are popular agent communication languages. A message in 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 the agent interaction protocols based on ACL have to be defined to achieve interoperability among the agents.

The agents need not to take care of the details how the messages are passed across the network, that is done by communication threads. The configuration between multiple agents at the server or client is shown in figure 11.3.

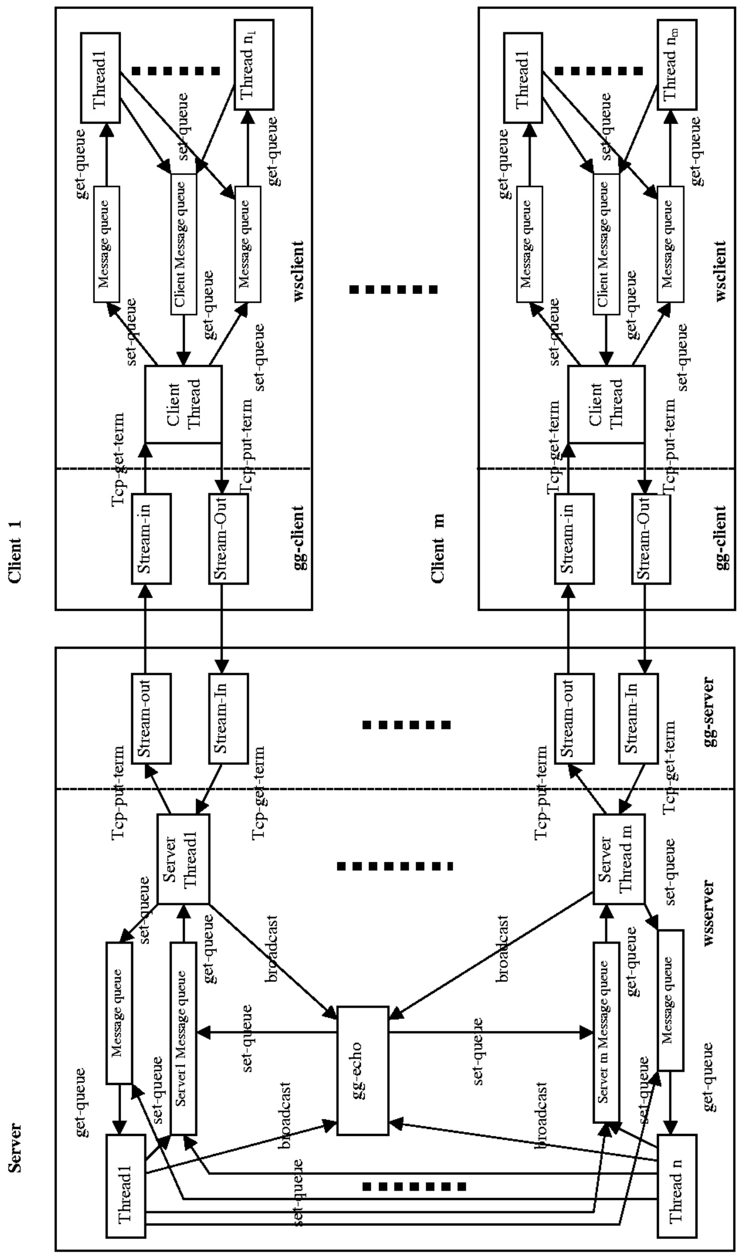


Fig. 11.2 Communication among Multiple Threads in DLP

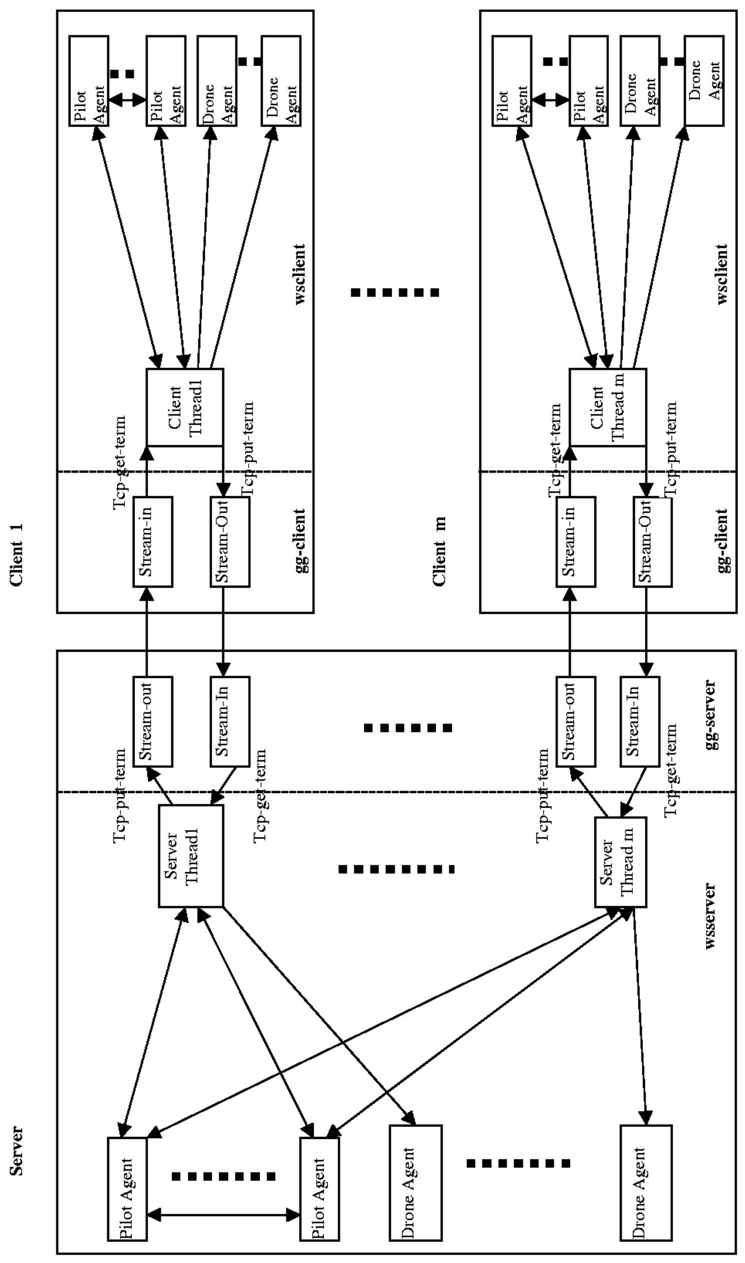


Fig. 11.3 Communication among Agents in DLP

11.3 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 be ones which never violate the rules of games.

- reactivity: Any player (user or agent) has to react very quickly in the game. Thus, it is not allowed to have serious performance problems.

- cooperation/competition: Soccer games are typical competition games which require the strong cooperation among teammates. 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 11.4.

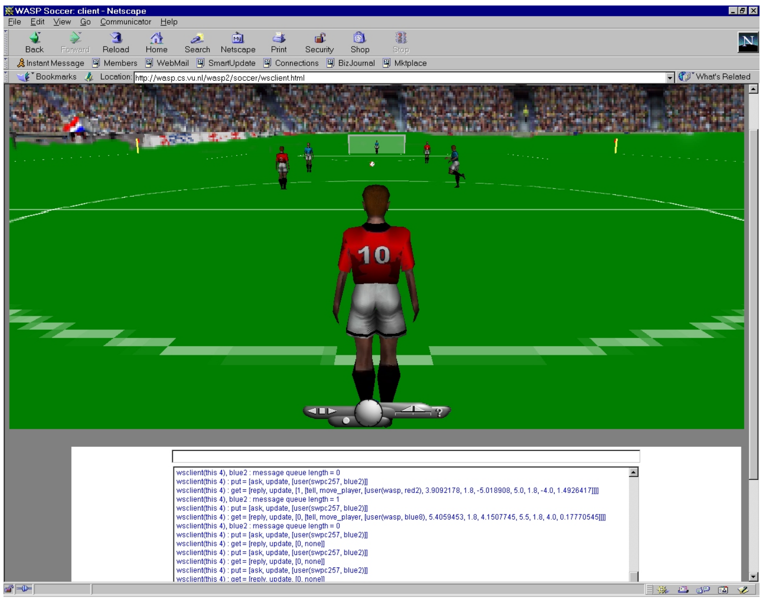
We consider to extend the soccer game example with a single user to the multiple user version. Namely, there are two playing teams: red and blue, in the soccer game. Two goal keepers, a soccer ball, and several agent players are designed to be pilot agents in the server. Whenever a new user joins the game, a client is created in which a user agent is created to be a pilot agent in the client.

Of course, the game would establish the balance of the number of players for two teams by considering the assignment of the player names to new users. This is achieved by take the first element of a free name list as a new player name:

$$FreeList = [blue2, red4, blue6, red8, blue10, red12]$$

11.3.1 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


WASP Soccer: client - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Instant Message Members WebMail SmartUpdate Connections BizJournal Mktplace
Bookmarks Location: http://wasp.cs.vu.nl/wasp2/soccer/wsclient.html
What's Related
10

```
wsclient(this 4), blue2 : message queue length = 0
wsclient(this 4) : put = [ask, update, [user(swpc257, blue2)]]
wsclient(this 4) : get = [reply, update, [1, [tell, move_player, [user(wasp, red2), 3.9092178, 1.8, -5.018908, 5.0, 1.8, -4.0, 1.4926417]]]]
wsclient(this 4), blue2 : message queue length = 1
wsclient(this 4) : put = [ask, update, [user(swpc257, blue2)]]
wsclient(this 4) : get = [reply, update, [0, [tell, move_player, [user(wasp, blue8), 5.4059453, 1.8, 4.1507745, 5.5, 1.8, 4.0, 0.17770545]]]]
wsclient(this 4), blue2 : message queue length = 0
wsclient(this 4) : put = [ask, update, [user(swpc257, blue2)]]
wsclient(this 4) : get = [reply, update, [0, none]]
wsclient(this 4) : put = [ask, update, [user(swpc257, blue2)]]
wsclient(this 4) : get = [reply, update, [0, none]]
wsclient(this 4) : put = [ask, update, [user(swpc257, blue2)]]
wsclient(this 4) : get = [reply, update, [0, none]]
```

Fig. 11.4 Screenshot of Soccer Game with Multiple Users

shared objects. The messages in distributed soccer game protocol is a 3-tuple: [1]

$$[Act, Type, ParameterList]$$

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 the parameters for the content type.

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

- register game: [register, game_name, from(Host)].
- register accept: [tell, accept, [Name, BallPosition, Score, OtherPlayers]].
- 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]].

[1]Consider the fact in the game most messages are sent either only to the server for further processing, or all of the clients, namely, there is only one world in the soccer game. We omit the sender name and the recipient names in most message formats. However, note that they can be added in the parameterList if necessary.

- 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 the player red10 at the host swpc257 is $\langle 0, 0, 10\rangle$.

The distributed soccer game protocol for pilot agents are straightforward, namely, they should regularly tell their position and rotation to the communication threads if the position and 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 communication thread. The server communication thread would decide which one is legal kick and takes certain actions. Thus, the server communication thread plays a central role for the communication among pilot/drone agents. The distributed soccer game protocol for the server communication thread is a set of 4-tuples with the following format: $\langle M, C, RM, B\rangle$, which states that if the Message M is received and the Condition C holds, then reply the message RM, and broadcast the message B. The protocol is shown in the table 11.3.1.

11.3.2 Performance Improvement

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

In order to improve the performance and decrease the message delays, new message formats are needed 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, thus, their behaviors are somehow predictable. Because of the high autonomy of the human users, user agents

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 11.1 Distributed Soccer Game Protocol

are usually hard to be simulated in a high level. Thus, the high level message formats are used only for autonomous player agent and objects agents. Furthermore, consider the cost of creation of a thread, drone user agents are not needed, for their drones can be controlled directly by the communication threads.

One of the high-level simulations is that: 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, like [tell, move_player,[user(Host,Name),X0,Y0,Z0,X1,Y1,Z1, Distance]].

Here is a list of high level message formats:

- move player: [tell, move_player,[user(Host,Name),X0,Y0,Z0,X1,Y1,Z1, Distance]].
- run and trace: [tell, run_and_trace, [user(Host,Name), Ball,X,Y,Z, KickableDistance,Dist,Runstep,RunSleepTime]].
- move ball: [tell, move_ball, [Ball, X0,Y0,Z0, X1,Y1,Z1, Distance]].

Introducing the high level messages formats significantly reduces the message delay. Suppose 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 and rotations. Suppose that each autonomous agent creates m_a messages per second, and each human agent create m_u message 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 for the action is $at(f)$ and the probability is $p(f)$. A single high-level message m with the format f would be equal to $at(f) \times m_a$ messages for a period $at(f)$. Namely, it would reduce $m_a - 1/at(f)$ message 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:

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

Namely, the improved performance is mainly determined by the probability of the message format. 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 would improve 15 percent of the performance.

11.3.3 Implementation

The whole implementation of the soccer game consists of the following five programs:

- gg_server deals with the network communication in the server;
- gg_client deals with the network communication in the clients;
- gg_echo deals with the network broadcasting in the server;
- *wssever* deals with the soccer game relevant issues in the server;
- *wsclient* deals with the soccer game relevant issues in the clients.

The first three programs deal with the general issues of the network communication, which can be done, like the example is already discussed in Section 11.1. In this section, we discuss the implemantation of the programs *wsserver* and *wsclient*.

11.3.3.1 ***wsserver*** The main tasks of the program *wsserver* are:

- message passing: Based on the distributed soccer game protocol discussed above, the server communication thread should control how the messages should be passed, forwarded, or broadcast to the receivers.
- realization of pilot agents: Most pilot agents would be located at the server, like autonomous player agents, pilot goalkeepers, and the pilot soccer ball, in a soccer game.

For the message passing, the object *wsserver* uses the predicate $server_reply$ to decide a replying term $ReplyTerm$ for an incoming message term $InputTerm$ as follows:

```
server_reply(InputTerm, ReplyTerm) :-
                      InputTerm = [register, soccer, from(ClientHost, ClientAddr
                      !,
                      format('~ n~w new client from ~ q~ n~ n', [this, ClientHos
                      get_field(used_name_list, Others),
                      gg_register_client(ClientHost, ClientName),
                      client_host := ClientHost,
                      client_name := ClientName,
                      client_addr := ClientAddr,
                      get_field(ball, position, BallPosition),
                      game_score <- getGameScoreStruct(Score),
                      ReplyTerm = [reply, accept, [client_name, BallPosition, Sc
                      Message = [tell, new_player, [client_name]],
                      gg_echo <- broadcast(ClientHost,ClientName, Message).

server_reply(InputTerm, ReplyTerm) :-
                      InputTerm = [unregister, soccer, [user(Host,Name)]],
                      !,
                      gg_unregister_client(Host, Name, Length),
                      ReplyTerm = [reply, unregister, [done, Length]],
                      Message = [tell, unregister, [client_name]],
                      gg_echo <- broadcast(client_host, to_all_clients, Message)

server_reply(InputTerm, ReplyTerm) :-
                      InputTerm = [tell, position, [user(Host,Name), Position]]
                      !,
                      set_field(Name, position, Position),
                      gg_echo <- broadcast(Host, Name, InputTerm),
                      gg_echo <- get_broadcasted_data(user(Host,Name), Attribute
                      ReplyTerm = [reply, update, Attributes].

......
```

If the server communication thread receives a register request from a client, then it would register the client by calling $gg_register_client$, and reply the client a message about the player name, the current position of the ball, the current game score, and the positions and rotations of the other players. Moreover, a message, which announces that the new player has joined, should be broadcast to all other players. If the server communication thread receives a unregister message from a client, it would reply a unregister message to the client, and broadcast the unregister message to all other players. If the server communication thread receives a message about the position of a player, then it should update the state of the drone by calling set_field. Moreover, the message should be broadcast to all other players and ask them to get the broadcast message by calling $get_broadcasted_data$ in the object gg_echo.

For the realization of the pilot agents in the server, *wsserver* should locally create the behaviors of the agents, like their counterparts in the soccer game with the single user version. In order to improve the performance, the VRML EAI interface is not necessary in the server, for nobody is actually watching at them and the system manager can always monitor the 3D-game by either watching the text-based message logs, or join in the game as a player. Therefore, we need some machinery which can be used for non-3D interface, i.e, text-based interface, to simulate the 3D VRML predicates, like setPosition and getPosition. These predicates are called *Get / Set Field Predicates*:

- set_field(+GlobalFieldName, +Term)
- get_field(+GlobalFieldName, ?Term)
- get_field_block(+GlobalFieldName, ?Term)
- get_field_reset(+GlobalFieldName, ?Term)
- get_field_event(+GlobalFieldName, ?Term)
- set_field(+NodeName, +FieldName, +Term)
- get_field(+NodeName, +FieldName, ?Term)
- get_field_block(+NodeName, +FieldName, ?Term)
- get_field_reset(+NodeName, +FieldName, ?Term)
- get_field_event(+NodeName, +FieldName, ?Term)

The NodeName and FieldName predicate arguments are atoms. Predicate set_field sets the associated value of the specified FieldName or NodeName and FieldName combination to the term as mentioned in the Term argument. Predicate get_field returns the stored Term value or returns the atom undefined when not yet defined by a set_field predicate. The get_field predicate is non-blocking, it always returns immediately. Predicate get_field_event blocks until a new set_field operation sets the corresponding value, after which it returns that value and "resets" the field. This blocking and reset behavior can be independently selected by the get_field_block (no reset) or get_field_reset (no blocking) predicates respectively.

Summary: get_field and get_field_reset are non-blocking : they allow for polling a particular field value. Predicates get_field_block and get_field_event always block until a new field value has been defined by the set_field predicate. Blocking and unblocking are (internally) wait / notify driven.

The soccer ball thread should regularly get the message from its own message and process the received messages as follows:

```
ball_action(Name) :-
          get_queue(ball, Message),
          ball_activity(Name,Message).

ball_activity(Ball, Message) :-
          Message = [ball_position, Data],
          Data = [Ball, Position],
          !,
          set_field(Ball, position, Position).

ball_activity(Ball, Message) :-
          Message = [kick_ball, Data],
          !,
          check_kick_ball(Ball, Data).

ball_activity(Ball, Message) :-
          Message = [move_ball, Data],
          Data = [Ball, XB,YB,ZB, XE,YE,ZE, Dist],
          !,
          move_to_position(Ball, translation, stepUnit,
          ballSleepTime,XB,YB,ZB,XE,YE,ZE, Dist).

ball_activity(Ball, Message) :-
           format('~ nUNKNOWN ~w MESSAGE = ~w~ n', [Ball, Message]).
```

The pilot player agents in the server behaves like those in the single user version, however, they have to regularly broadcast the behavior messages. For instance, if a player wants to run-and-trace, he should send the high level message to everyone, like this:

```
run_and_trace(Player,Ball,X,Y,Z,KickableDistance,Dist,
            Runstep,RunSleepTime):-
     Data = [user(ServerHost,Player), Ball,X,Y,Z,KickableDistance,
            Dist,Runstep,RunSleepTime],
     Message = [tell, run_and_trace, Data],
     gg_echo <- broadcast(ServerHost, to_all_clients,  Message),
     xold := X,
     zold := Z,
     repeat,
            sleep(RunSleepTime),
            getPosition(Ball,X1,_,Z1),
            look_at_position(Player,X1,Z1),
            getRotation(Player,_,_,_,R),
            R1 is R + 1.5708,
            Xnew is xold-Runstep * cos(R1),
            Znew is zold+Runstep * sin(R1),
            setSFVec3f(Player,set_position, Xnew, Y, Znew),
            xold := Xnew,
            zold := Znew,
            distance2d(Xnew,Znew,X1,Z1,Distball),
            balldistance := Distball,
    end_of_run_and_trace(balldistance, KickableDistance, Dist),
  !.
```

11.3.3.2 ***wsclient*** The task of the program *wsclient* is to process the received messages in the client. The client communication thread in *wsclient* uses the predicate $server_reply$ to handle the messages based on the different message formats.

If the message is 'tell position' or 'tell rotation', then set the player's position or rotation by directly calling the predicate *setPosition* like this:

```
server_reply(ReplyTerm):-
           ReplyTerm = [tell, position, Data],
           Data = [user(_Host,Name), position(X,Y,Z)],
           client_name \= Name,
           !,
           setPosition(Name, X,Y,Z).

server_reply(ReplyTerm) :-
            ReplyTerm = [tell, rotation, Data],
            Data = [user(_Host,Name), rotation(X,Y,Z,R)],
            client_name \= Name,
            !,
            setRotation(Name, X,Y,Z,R).
```

If the received message is of the high level format, then the client communication thread would forward the message to the corresponding drone agent, like this:

```
server_reply(ReplyTerm) :-
             ReplyTerm = [tell, move_player, Data],
             Data = [user(_Host,Player), _X,_Y,_Z, _X1,_Y1,_Z1, _Dist],
             !,
             set_queue(Player, [move_player, Data]).

server_reply(ReplyTerm) :-
             ReplyTerm = [tell, run_and_trace, Data],
             Data = [user(_ServerHost,Player),
                     _Ball,_X,_Y,_Z,_KickableDistance,_Dist,_Runstep,_RunSleepTime],
             !,
             set_queue(Player, [run_and_trace, Data]).
```

```
server_reply(ReplyTerm):-
             ReplyTerm = [tell, kick_ball, Data],
             !,
             set_queue(ball, [kick_ball, Data]).

server_reply(ReplyTerm):-
             ReplyTerm = [tell, move_ball, Data],
             !,
             set_queue(ball, [move_ball, Data]).
```

The behaviors of the ball drone agent is like those of the counterpart in the server, like this:

```
ball_activity(Mesg):-
            Mesg = [ball_position, Data],
            Data = [Ball, position(X,Y,Z)],
            !,
            setPosition(Ball, X, Y, Z).

ball_activity(Mesg):-
            Mesg = [kick_ball, Data],
            Data = [_Name, BallName, X,Y,Z,R1, KickBallForce, KickBallForceY],
            !,
            kickedwithStaticStartRF(BallName,translation,X,Y,Z,R1, KickBallForce
                             KickBallForceY).

ball_activity(Mesg):-
            Mesg = [move_ball, Data],
            Data = [Ball, X,Y,Z, X2,Y2,Z2, Dist],
            !,
            move_to_position(Ball,translation, stepUnit,ballSleepTime, X,Y,Z,X2
```

Similarly, the drone player agents in the clients should behave like this:

```
player_activity(Mesg):-
            Mesg = [move_player, Data],
            Data = [user(_Host,Player), X,Y,Z, X2,Y2,Z2, Dist],
            !,
            move_to_position(Player,set_position, runStep, runSleepTime, X,Y,Z,X

player_activity(Mesg) :-
            Mesg = [run_and_trace, Data],
            Data = [user(_ServerHost,Player), Ball,X,Y,Z,KickableDistance,
             Dist,Runstep,RunSleepTime],
            !,
            run_and_trace(Player,Ball,X,Y,Z,KickableDistance,
              Dist,Runstep,RunSleepTime).
```

The pilot user agents in the clients behave like those in the single user version, however, they should report their positions and rotations to the server if the positions and rotations are new, which is mainly done by the predicate $near_{b}all_{t}hen_{k}ick$ as follows:

```
near_ball_then_kick(Agent, Ball):-
            getViewpointPosition(Agent,X,Y,Z),
            local_host(Host, _),
            TellData = [user(Host,client_name), position(X,Y,Z)],
            TellMessage = [tell, position, TellData],
            getPosition(Ball,X1,Y1,Z1),
            distance2d(X,Z,X1,Z1,Dist),
            Dist < kickableDistance,
            !,
            getViewpointOrientation(Agent,_,_,_,R),
            R1 is sign(X1 - X) * R -1.5708,
            Data = [client_name, Ball, X1,Y1,Z1,R1, kickBallForce,
             kickBallForceY],
            KickMessage = [tell, kick_ball, Data],
            set_queue(client_name, TellMessage),
            set_queue(client_name, KickMessage).
```

Exercises

11.1 Design a virtual gallery for multiple users, in which an intelligent agent serve as a guide.

12 Conclusions

Appendix A
DLP Built-in Predicates

Arithmetic comparison

```
+Eval =:= +Eval (arithmetic equal)
+Eval =\= +Eval (arithmetic not equal)
+Eval > +Eval (arithmetic greater than)
+Eval >= +Eval (arithmetic greater than or equal)
+Eval < +Eval (arithmetic less than)
+Eval =< +Eval (arithmetic less than or equal)
```

Arithmetic evaluation

```
LogicalVar is Expression (evaluate expression)
Arithmetic functors :
    '+'/2
    '-'/1
    '-'/2
    '*'/2
    '/'/2
    '//'/2
    '<<'/2
    '>>'/2
    '**'/2
```

```
        abs/1
        mod/2
        rem/2
        acos/1
        asin/1
        atan/1
        cos/1
        sin/1
        tan/1
        exp/1
        log/1
        random/0
        round/1
        sign/1
        sqrt/1
        truncate/1
```

Atomic term processing

```
    atom_chars (+Atom, ?CharList)
    atom_chars (?Atom, +CharList)
    atom_codes (+Atom, ?CodeList)
    atom_codes (?Atom, +CodeList)
    atom_concat (+Atom1, +Atom2, ?NewAtom)
    atom_list_concat (+AtomList, ?NewAtom)
    atom_length (+Atom, ?Length)
    atom_number(+Atom, ?Number)
    atom_number(?Atom, +Number)
    char_code (+Char, ?Code)
    char_code (?Char, +Code)
    number_chars (+Number, ?CharList)
    number_chars (?Number, +CharList)
    number_codes (+Number, ?CodeList)
    number_codes (?Number, +CodeList)
```

Character input/output

```
    get_char(?Char)
    get_char(+Stream, ?Char)
    nl
    nl(+Stream)
    put_char(+Char)
    put_char(+Stream, +Char)
```

Logic and control

```
    ',' /2 (conjunction)
```

```
'!' /0 (cut)
fail /0
halt /1
not /1, '\+' /1
once /1
repeat /0
true /0
```

Stream selection and control

```
at_end_of_stream/0
at_end_of_stream(+Stream)
close(+Stream)
open(+Source_Sink, +IOmode, -Stream),
atom stream (TBD) + set_output
IOmode = {read | write | append}
set_input(+Stream)
set_output(+Stream)
```

Term comparison

```
Term1 == Term2
Term1 \== Term2
Term1 @< Term2
Term1 @> Term2
Term1 @=< Term2
Term1 @>= Term2
```

Term creation and decomposition

```
arg (+ArgI, +CompoundTerm, ?ArgV)
copy_term (?Term, ?Copy)
functor (+Term, ?Name, ?ArgC)
functor (?Term, +Name, +ArgC)
+Term =.. ?List
?Term =.. +List
```

Term input/output

```
display (?Term)
display (+Stream, ?Term)
format (+Format, +ArgList)
format (+Stream, +Format, +ArgList)
format specifiers : ~k ~n ~q ~t ~w
write (?Term)
write (+Stream, ?Term)
write_canonical (?Term)
write_canonical (+Stream, ?Term)
writeq (?Term)
writeq (+Stream, ?Term)
```

Term input from constant terms

```
read_from_atom (+Atom, ?Term)
read_from_atom (+Atom, ?Term, ?VarList)
unifies Term with the term representation
of Atom or an error/2 term.
```

Term output to constant terms

```
format_to_atom (?Atom, +Format, +ArgList)
format_to_chars (?CharList, +Format, +ArgList)
format_to_codes (?CodeList, +Format, +ArgList)
```

Term unification

```
?Term1 = ?Term2 (unify)
?Term1 \= ?Term2 (not unifiable)
```

Type testing

```
atom (?Term). Term is an atom.
atomic (?Term). Term is atomic (atom or number).
compound (?Term). Term is a compound term.
float (?Term). Term is a float.
integer (?Term). Term is an integer.
nonvar (?Term). Term is instantiated.
number (?Term). Term is a number (integer or float).
var(?Term). Term is uninstantiated.
```

List Processing

```
append / 3
insert(+OldSortedList, +Term, ?NewSortedList) (ascending order)
length / 2
member / 2
memberchk / 2
reverse / 2
select / 3
selectchk / 3
```

Miscellaneous

```
compare(R, Term1, Term2).
```

```
    The first argument R will be unified with
    '<', when Term1 'term-precedes' Term2
    '=', when Term1 is identical to Term2
    '>', when Term2 'term-precedes' Term1
    Predicate compare/3 is used by the term comparison predicates.

link_object(ObjectName). Load, link and initialize ObjectName and
     all its base objects.

memory(-TotalKBs, -FreeKBs). Set the total number of KBytes and the
      currently available free KBytes.

sleep(+Msecs). Suspend the execution of the current thread for Msecs
       milliseconds.

get_system_property(+Key, -Value)

set_system_property(+Key, +NewValue, -OldValue)
```

Java run-time system property predicates

```
stack_trace/0. Show the method/predicate continuations until the
       current point of execution.

text_area(BrowserStream). Argument BrowserStream will be unified with
     a Java TextArea like stream. Combining this predicate with
     set_output(BrowserStream) redirects the output of a program
     to the corresponding BrowserStream, otherwise program
     output will be written to the browser "Java Console".

text_field(InputQueueName). Asynchronous browser input from a
     TextField like input line is sent to the queue as an atom.
     Input in DLP (and Java), like browser or socket input, is usually
     processed by a separate thread. Instead of all kinds of detailed
     I/O control facilities as found in programming languages like C
     or C++, I/O in DLP (and Java) is handled by a thread that
     communicates its I/O results to other threads in a multi-threaded
     safe way.

trace/0. Turn trace mode on. All method calls will be displayed
     (non-interactively).

notrace/0. Turn trace mode off.
...
```

TCP / IP Networking

```
Host Identification
    host_address(+HostName, -InternetAddress).
    local_host(-HostName, -InternetAddress).
    code_base_host(-HostName).
       Unifies HostName with the name of the host from
       which the current program objects are loaded. When
       running in a browser. HostName will be unified with
       the name of the originating host.

Server Predicates
    tcp_server(+ServerPort, -ServerSocket).
    tcp_accept(+ServerSocket, -ServerStreamIn, -ServerStreamOut).
    tcp_accept(+ServerSocket, -ServerStreamIn, -ServerStreamOut,
           -ClientHostName).

Client Predicates
    tcp_client(+ServerHostName, +ServerPort, -ClientStreamIn,
           -ClientStreamOut). (To be done)
    tcp_client(+ServerHostName, +ServerPort, +TimeOut,
           -ClientStreamIn, -ClientStreamOut).

Bi-Directional Client / Server Communication
    tcp_get_term(+StreamIn, ?Term).
    tcp_put_term(+StreamOut, +Term).
    ....
Closing Client / Server Connection
    tcp_close(+Socket).
    tcp_close(+Stream).
```

Node::Field Attribute Storage and Retrieval Predicates
DLP can be used to develop multi-threaded stand-alone client / server systems as well as multi-threaded programs running in the context of browsers, VRML / EAI frameworks, or other special purpose environments that support JVM based execution. When developing distributed VRML / EAI Web based DLP systems it's often convenient to provide the server part of the system with similar capabilities as available in the VRML/EAI embedded clients of the system in order to maintain the (distributed) state of particular nodes. This allows for a similar modeling approach of several storage and retrieval aspects in both the client and server parts. The EAI-like NodeName :: FieldName storage and retrieval predicates (see below) provide such a functionality. Although they are primarily intended for stand-alone servers, i.e. servers not embedded in a VRML / EAI framework, they can also be used for other purposes. Typically, such a server will handle multiple independently running

client threads, therefore the storage and retrieval predicates are extended to deal with event driven, i.e. (internally) wait /notify based, processing.

Get / Set Field Predicates

```
set_field(+GlobalFieldName, +Term)
get_field(+GlobalFieldName, ?Term)
get_field_block(+GlobalFieldName, ?Term)
get_field_reset(+GlobalFieldName, ?Term)
get_field_event(+GlobalFieldName, ?Term)

set_field(+NodeName, +FieldName, +Term)
get_field(+NodeName, +FieldName, ?Term)
get_field_block(+NodeName, +FieldName, ?Term)
get_field_reset(+NodeName, +FieldName, ?Term)
get_field_event(+NodeName, +FieldName, ?Term)
```

The NodeName and FieldName predicate arguments are atoms. Predicate set_field sets the associated value of the specified FieldName or NodeName and FieldName combination to the term as mentioned in the Term argument. Predicate get_field returns the stored Term value or returns the atom undefined when not yet defined by a set_field predicate. The get_field predicate is non-blocking; it always returns immediately.

Predicate get_field_event blocks until a set_field operation sets the corresponding value, after which it returns that value and "resets" the field. This blocking and reset behavior can be independently selected by the get_field_block (no reset) or get_field_reset (no blocking) predicates respectively. An implicit (get_field_event) or explicit (get_field_reset) field reset doesn't set the field value to undefined, but only flags that the value has been retrieved. This allows non-blocking predicates to get the latest field value. Summary: get_field and get_field_reset are non-blocking : they allow for polling a particular field value. Predicates get_field_block and get_field_event always block until a new field value has been defined by the set_field predicate. Blocking and unblocking are (internally) wait / notify driven.

Get / Set Queue Predicates Queue predicates provide a flexible way to construct special purpose asynchronous interaction patterns or protocols between active objects. All queue primitives are safe in multi-threaded execution contexts (atomic queue update).

```
new_queue(+GlobalQueueName, +MaxSize)
set_queue(+GlobalQueueName, +Term)
get_queue(+GlobalQueueName, ?Term)

queue_full(+GlobalQueueName)
queue_empty(+GlobalQueueName)
```

```
queue_length(+GlobalQueueName, -CurrentLength)

new_queue(+NodeName, +QueueName, +MaxSize)
set_queue(+NodeName, +QueueName, +Term)
get_queue(+NodeName, +QueueName, ?Term)

queue_full(+NodeName, +QueueName)
queue_empty(+NodeName, +QueueName)
queue_length(+NodeName, +QueueName, -CurrentLength)
```

NodeName and QueueName predicate arguments are atoms. Predicate new_queue creates a unique queue. The maximum number of queue elements is defined by MaxSize. When a queue isn't defined yet by new_queue upon the first invocation of a set_queue or get_queue predicate, a queue descriptor will automatically be created with a default maximum size of 100. Predicate set_queue appends a new Term to the queue. If the queue contains MaxSize elements this operation will block until a get_queue operation removes an element from the queue. Predicate get_queue removes the first element from the queue and unifies this element with Term. In case the queue is empty, this operation will block until a set_queue operation adds an element to the queue.

Conditional Queue Lookup Predicates

```
accept_queue_term ( +GlobalQueueName, +AcceptExpressionList,
     -Term )

accept_queue_term ( +NodeName, +QueueName,
      +AcceptExpressionList, -Term )
  Accept the first entry in the queue that satifies one of the
  AcceptExpressions in the AcceptExpressionList.
```

Get / Set Array Predicates

```
new_array(+NodeName, +ArrayName, +Dim1)
get_array(+NodeName, +ArrayName, +Idx1, ?Elem)
set_array(+NodeName, +ArrayName, +Idx1, +Elem)

new_array(+NodeName, +ArrayName, +Dim1, +Dim2)
get_array(+NodeName, +ArrayName, +Idx1, +Idx2, ?Elem)
set_array(+NodeName, +ArrayName, +Idx1, +Idx2, +Elem)

new_array(+NodeName, +ArrayName, +Dim1, +Dim2, +Dim3)
get_array(+NodeName, +ArrayName, +Idx1, +Idx2, +Idx3, ?Elem)
set_array(+NodeName, +ArrayName, +Idx1, +Idx2, +Idx3, +Elem)
```

```
    NodeName and ArrayName are atoms.
    Array elements must be explicitly initialized.
```

DLP Regular Expression Library Predicates The rexlib requires Java SDK 1.4.X. See the java.util.regex package for more information : each Pattern or Matcher predicate mentioned below corresponds to a single Java method as defined in the java.util.regex package.

Pattern Predicates

```
        compile_pattern ( +RegExpPattern , -PatternRef )
        RegExpPattern is an atom.

        PatternRef is a foreign language object reference term.
        Compiles the given regular expression into a pattern.

        compile_pattern ( +RegExpPattern , +PatternFlagList , -PatternRef )
         PatternFlagList is a list of one or more pattern
         compilation options: [ canon_eq, case_insensitive,
         comments, dotall, multiline, unicode_case, unix_lines ]
           Compiles the given regular expression into a pattern
           with the given flags.

        pattern_matches ( +RegExpPattern , +InputAtomOrString )
        pattern_matches ( +RegExpPattern , +InputAtomOrString ,
            -Boolean )
          Compiles the given regular expression and matches the given
          input against it. Predicate pattern_matches / 2 either
          succeeds or fails, pattern_matches / 3 returns the atom
          true or false. Predicate pattern_matches / [2,3] is defined
          as a convenience for when a regular expression is used only once.

        split_input ( +PatternRef , +InputAtomOrString , -AtomList )
        split_input ( +PatternRef , +InputAtomOrString , +MaxListLength ,
              -AtomList )
        Splits the given input sequence around matches of this pattern.
```

Pattern Matcher Creation

```
        pattern_matcher ( +PatternRef , +InputAtomOrString , -Matcher )
          Creates a matcher that will match the given input against this
          pattern, many matchers can share the same pattern specification.
          Matcher is a matcher foreign language object reference term.
```

URL predicates

```
        url_open ( +URL , -URLRef )
```

```
url_read_line ( +URLRef , -LineString )
url_close ( +URLRef )
  Predicates are part of dlplib, not rexlib.
```

Matcher Predicates

```
replacement_buffer ( +BufferSize, -BufferRef)

append_replacement ( +Matcher , +BufferRef ,
       +Replacement )

append_replacement ( +Matcher , +BufferRef ,
       +Replacement , -MatcherOut )
   Non-terminal append-and-replace.

append_tail ( +Matcher , +BufferIn , -AtomOut )
Terminal append-and-replace.

end_index ( +Matcher , -Index )
    Returns the index of the last character matched,
    plus one.

end_index ( +Matcher , +GroupNumber , -Index )
   Returns the index of the last character, plus
   one, of the subsequence captured by the given
   group during the previous match operation.
   GroupNumber = [ 0 ... GroupCount - 1 ]. See also
   group_count / 2.

find_next ( +Matcher )
find_next ( +Matcher , -Boolean )
   Finds the next subsequence of the input sequence
   that matches the pattern.

find_starting_at ( +Matcher , +Index )
find_starting_at ( +Matcher , +Index , -Boolean )
   Finds the next subsequence of the input sequence
   that matches the pattern, starting at the specified
   index.

group ( +Matcher , -Atom )
   Returns the input subsequence matched by the
   previous match.

group ( +Matcher , +GroupNumber , -Atom )
```

```
   Returns the input subsequence captured by the
   given group during the previous match operation.

group_count ( +Matcher , -GroupCount )
   Returns the number of capturing groups in this
   matcher's pattern

looking_at ( +Matcher )
looking_at ( +Matcher , -Boolean )
   Starting at the beginning, match the input sequence
   against the pattern.

matches ( +Matcher )
matches ( +Matcher , -Boolean )
Matches the entire input sequence against the pattern.

replace_all (+Matcher , +Replacement , -Atom )
       Replaces every subsequence of the input sequence that
       matches the pattern with the given replacement.

replace_first ( +Matcher , +Replacement , -Atom )
       Replaces the first subsequence of the input sequence
       that matches the pattern with the given replacement.

reset_matcher ( +MatcherIn , -MatcherOut )
Resets this matcher.
reset_matcher ( +MatcherIn , +InputAtomOrString , -MatcherOut )
Resets this matcher with a new input sequence.

start_index ( +Matcher , -Index )
Returns the start index of the previous match.
start_index ( +Matcher , +GroupNumber , -Index )
        Returns the start index of the subsequence captured by the given
        group during the previous match operation.
```

Atom / String Predicates In logic programming languages, atoms are typically used for the representation of symbolic constants. Usually they have a relatively modest size and they can be used efficiently in many contexts. However, the implementation of atoms is optimized for unification; all identical atoms have a unique reference, which reduces the atom related string comparison operations to a single object reference comparison. Although atoms may be used for several text-like manipulation and inspection issues, a number of applications require sometimes a representation that is more appropriate for manipulating text in general.

```
Atom/String Predicates
    atom_to_string ( +Atom , ?String )
    string_to_atom ( +String , ?Atom )

    new_string_buffer ( -StringBuffer )
    new_string_buffer ( +Length , -StringBuffer )

    string ( ?Term )
    Succeeds when Term is a string.
    string_char_at ( +String , +Index , ?Char )
    Char is a single character atom.
    string_code_at ( +String , +Index , ?Code )
    Code is an integer.
    string_compare_to ( +String1 , +String2 , ?Result )
    Result: < 0 | == 0 | > 0
    string_buffer_to_atom ( +StringBuffer , ?Atom )
    string_buffer_to_string ( +StringBuffer , ?String )
    string_length ( +String , -Length )
    substring ( +String , +BeginIndex , +EndIndex , -SubString )
    ...

    strings should be built-in :
        <@ , >@ , =<@ , >=@ , == / 2 , \== / 2
        = / 2 , \= / 2
        "write_term" output method
        type testing: string(Term)
        copy_term / 2 .
        FLI : StrTerm
    ...
```

DLP / JavaScript Interface

```
DLP methods (object jsilib)
    get_window ( -JSObject )
    call ( +JSObject, +JSMethodName, +ArgList, -Result )
    (t.b.d. arrays < = > lists)
    eval ( +JSObject, +Evaluate, -Result )
    see member example w.r.t. netscape/explorer remarks.
    get_member ( +JSObject, +Member, -Result )
    set_member ( +JSObject, +Member, +Value )
    get_slot ( +Member, +Index, -Value) (netscape)
    set_slot ( +Member, +Index, +Value) (netscape)
    (use call/4 for a portable solution)
JS method (part of object dlpbrow / dlpcons)
    applet.set_field("namespace", "queuename", term) (t.b.d.)
    applet.set_queue("namespace", "queuename", term)
    var applet = window.document.applets['dlpbrow'];
    var applet = window.document.applets['dlpcons'];
    Parameters namespace, queuename are strings.
    Parameter term is an integer, float, double, or string :
    a string is either :
        "term:" + "aValidTermString"
        "atom:" + "anAtom" (explicit atom)
      or any other string (implicit atom)
```

JS / DLP interface is based on the JS / Java LiveConnect interface.

DLP / XML : Extensible Markup Language Predicates under construction.

Multi-threaded Objects (Syntax Summary)

```
program examples :
    multi-threaded (binary) semaphore objects
    multi-threaded producer / buffer / consumer objects
declaration of objects :
    :-object name .
    :-object name : [ base ].
    :-object name : [ base1, base2, ... ].
    :-end_object name .
declaration non-logical variables (nlv's) :
var i=0, j=[1,2,3], k=f(a,b,c).
destructive assignment:
nlv := Term
simplification:
    nlv := Expression
    other nlv occurrences are replaced by their current value.
object creation:
ObjectRef := new(ObjectNameOrConstructor)
method invocation:
ObjectRef <- method(...)
synchronuous accept statement :
accept (AcceptExpression1, AcceptExpression2, ...)
    accept expression:
        method(...) <== [Guard] ==> Body
        method(...) <== [Guard]
        method(...) ==> Body
        method(...)
        any
    accept guard:
        callable term
    accept body:
```

```
                callable term
        ...
```

DLP Foreign Language Interface FLI General Remarks
All Term classes, as mentioned below, are derived from class Word (and not class Term) because run-time structures (that are not Term's) should have the same machine word oriented base class. The C like macro style of the described interface hides many aspects of the actual implementation characteristics of Terms. It allows for a more convenient conversion to different Term implementation schemes.

```
Term Dereferencing

    Word dt = Word.deref (Word t)

Term Creation

    IntTerm it = Word.new_int (int)
    FltTerm ft = Word.new_flt (double)
    ForTerm ft = Word.new_for (Object ref)
    foreign object reference term
    FunTerm ft = Word.new_fun (SymTerm func, Word [ ] args)
    FunTerm ft = Word.new_fun (SymTerm func, Word arg1)
    FunTerm ft = Word.new_fun (SymTerm func, Word arg1,
           Word arg2)
    FunTerm ft = Word.new_fun (SymTerm func, Word arg1,
           Word arg2, Word arg3)
    LstTerm lt = Word.new_lst (Word head, Word tail)
    NilTerm nt = Word.new_nil ()
    SymTerm st = Word.new_sym ("String".intern())

Term Type Tests

    boolean it = Word.int_term (Word w)
    boolean ft = Word.flt_term (Word w)
    boolean ft = Word.for_term (Word w)
    boolean ft = Word.fun_term (Word w)
    boolean lt = Word.lst_term (Word w)
    boolean nt = Word.nil_term (Word w)
    boolean st = Word.sym_term (Word w)
    boolean vt = Word.var_term (Word w)

Term Type Casts

    IntTerm it = Word.int_cast (Word w)
    FltTerm ft = Word.flt_cast (Word w)
    ForTerm ft = Word.for_cast (Word w)
    FunTerm ft = Word.fun_cast (Word w)
    LstTerm lt = Word.lst_cast (Word w)
    SymTerm st = Word.sym_cast (Word w)
    VarTerm vt = Word.var_cast (Word w)
```

```
Term Value Retrieval
```

```
    int i = Word.int_val (Word it)
    double f = Word.flt_val (Word ft)
    Object o = Word.for_val (Word ft)
    String n = Word.fun_name (Word ft)
    int argc = Word.fun_argc (Word ft)
    Word argv [ ] = Word.fun_argv (Word ft)
    Word argv = Word.fun_argi (Word ft, int argi)
    Word head = Word.lst_head (Word lt)
    Word tail = Word.lst_tail (Word lt)
    String s = Word.sym_val (Word st)
```

Vectors / Quaternions (vectorlib)

```
    Predicates
        vector_dot_product(vector(X1,Y1,Z1), vector(X2,Y2,Z2),
             V1DotV2)

        vector_cross_product(vector(X1,Y1,Z1), vector(X2,Y2,Z2),
             vector(XN,YN,ZN))

        quaternion_to_rotation(quaternion(W1,X1,Y1,Z1),
             rotation(X,Y,Z,R))

        rotation_to_quaternion(rotation(X1,Y1,Z1,R1),
             quaternion(W,X,Y,Z))

        unit_quaternion(quaternion(W1,X1,Y1,Z1),
             quaternion(W,X,Y,Z))

        quaternion_product(quaternion(W1,X1,Y1,Z1),
            quaternion(W2,X2,Y2,Z2), quaternion(W,X,Y,Z))

        slerp(F, quaternion(W1,X1,Y1,Z1),
           quaternion(W2,X2,Y2,Z2), quaternion(W,X,Y,Z))
```

DLP / EAI: VRML External Authoring Interface Library (Summary)

```
VRML Browser Predicates

    loadURL( +URL )
    getWorldURL( -URL )
    setDescription( +Description )
    Description is an atom
    setTimerInterval( +NewMsecs , -OldMsecs )
    beginUpdate / 0, endUpdate / 0

    createVrmlFromString( +VrmlAtom , -ObjectRefList)
    createVrmlFromString( +VrmlAtom , +ParentObject ,
           -ObjectRefList)
    ObjectRefList is a Foreign Object Reference List.
    See also the addChildren and removeChildren predicates.
    createVrmlFromURL( +URL , +NotifyNode , +NotifyField )

    addRoute( +FromObject, +EventOutFieldName ,
          +ToObject , +EventInFieldName )

    deleteRoute( +FromObject, +EventOutFieldName,
          +ToObject, +EventInFieldName )

VRML Event Observers

    eventObserverQueue ( +Object, +FieldName,
           +QueueName )
    Object/FieldName events are sent to QueueName as a term:
    FieldName(EventValue, EventTime, Object)
    Not available (in the current EAI SDK : EventOut.unAdvise) :
    removeEventObserverQueue ( +Object, +FieldName, +QueueName )

Agent / Object Coordinates

    getPosition( +Object, -X, -Y, -Z )
    setPosition( +Object, +X, +Y, +Z )
    getRotation( +Object, -X, -Y, -Z, -R )
    setRotation( +Object, +X, +Y, +Z, +R )
    getViewpointPosition( +Agent, -X, -Y, -Z )
    setViewpointPosition( +Agent, +X, +Y, +Z )
    getViewpointOrientation( +Agent, -X, -Y, -Z, -R )
    setViewpointOrientation( +Agent, +X, +Y, +Z, +R )
    X, Y, Z, and R values are integers or floats
```

```
Agent / Object Distance

    distance2D(+X1, +Z1, +X2, +Z2, -Distance)
    distance3D(+X1, +Y1, +Z1, +X2, +Y2, +Z2, -Distance)

Vector Products

    vector_dot_product(+X1,+Y1,+Z1, +X2,+Y2,+Z2,
         -Angle12, -V1dotV2)
    V1 dot V2 = | V1 | . | V2 | . cos (Angle12)

    vector_cross_product(+X1,+Y1,+Z1, +X2,+Y2,+Z2,
       -XN,-YN,-ZN, -Angle12, -VNSize)
    | VN | = | V1 cross V2 | = | V1 | . | V2 | . sin (Angle12)

Single Field Predicates


    getSFBool(+Object, +Field, -Bool)
    setSFBool(+Object, +Field, +Bool)
    Bool is an atom : 'true' or 'false'
    getSFFloat(+Object, +Field, -Float)
    setSFFloat(+Object, +Field, +Float)
    getSFInt32(+Object, +Field, -Int32)
    setSFInt32(+Object, +Field, +Int32)

    getSFString(+Object, +Field, -Atom)
    setSFString(+Object, +Field, +Atom)
    setSFString(+Object, +Field, +Format, +ArgList)

    getSFColor(+Object, +Field, -R,-G,-B)
    setSFColor(+Object, +Field, +R,+G,+B)
    R, G, and B values are integers or floats
    getSFNode(+Object, +Field, -ObjectRef)
    setSFNode(+Object, +Field, +NameOrRef)
    getSFRotation(+Object, +Field, -X,-Y,-Z, -R)
    setSFRotation(+Object, +Field, +X,+Y,+Z, +R)
    getSFTime(+Object, +Field, -Time)
    setSFTime(+Object, +Field, +Time)

    getSFVec2f(+Object, +Field, -X,-Y)
    setSFVec2f(+Object, +Field, +X,+Y)
    getSFVec3f(+Object, +Field, -X,-Y,-Z)
    setSFVec3f(+Object, +Field, +X,+Y,+Z)
```

```
Multi Field Predicates

    getMFFloat(+Object, +Field, -FloatList)
    setMFFloat(+Object, +Field, +FloatList)
    getMFInt32(+Object, +Field, -IntegerList)
    setMFInt32(+Object, +Field, +IntegerList)

    getMFString(+Object, +Field, -AtomList)
    setMFString(+Object, +Field, +AtomList)
    setMFString(+Object, +Field, +Format, +ArgList)

    getMFColor(+Object, +Field, -RGBList)
    setMFColor(+Object, +Field, +RGBList)
    RGBList = [ [R1,G1,B1], [R2,G2,B2], .... ]
    getMFNode(+Object, +Field, -ObjectRefList)
    setMFNode(+Object, +Field, +ObjectRefList)
    TBD : (mixed) Object or ObjectRef List
    getMFRotation(+Object, +Field, -XYZRList)
    setMFRotation(+Object, +Field, +XYZRList)
    XYZRList = [ [X1,Y1,Z1,R1], [X2,Y2,Z2,R2], .... ]

    getMFVec2f(+Object, +Field, -XYList)
    setMFVec2f(+Object, +Field, +XYList)
    XYList = [ [X1,Y1], [X2,Y2], .... ]
    getMFVec3f(+Object, +Field, -XYZList)
    setMFVec3f(+Object, +Field, +XYZList)
    XYZList = [ [X1,Y1,Z1], [X2,Y2,Z2], .... ]

(Incremental) MFNode Updates

    addChildren( +ParentObject, +ObjectRefList )
    removeChildren( +ParentObject, +ObjectRefList )
```

Appendix B
Source Codes

B.1 SOCCER GAME: SINGLE USER/MULTIPLE AGENTS

Figure B.1 shows the field of play with the length 100 meter and the width 64 meter, which is designed as a gif file ”field1.gif”. Therefore, the field can be designed in VRML as follows:

```
Transform {
      translation 0 0 0
            children [
            Shape {
              appearance Appearance {
              texture ImageTexture { url "field1.gif" }
              textureTransform TextureTransform {scale 1 1}
               }
               geometry Box {size 100 .2 64}
             }
             ]
```

Similarly we can define goal gates and the soccer ball. However, we will omit the details.

The prototype of player avatars is designed as follows:

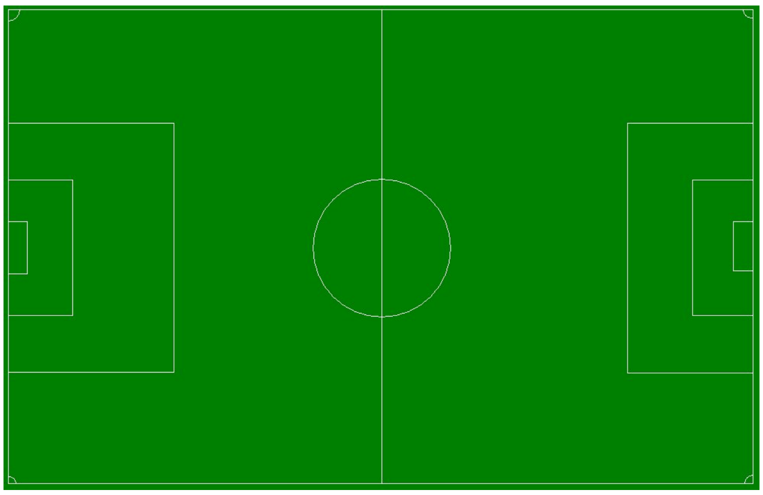


Fig. B.1 The Field of Play

```
PROTO Sportman [
    exposedField SFVec3f position
    exposedField SFRotation rotation
    exposedField SFInt32 whichChoice
    exposedField SFString nickname
    exposedField MFString picturefile
]
{......}
```

The following VRML code defines a goalkeeper and a player:

```
Transform {
    children [ DEF goalKeeper1 Sportman
       {rotation 0 1 0 -1.5708
       whichChoice -1
       position 48 1.8 0
       picturefile ["sportmanblue1.jpg"]
       nickname "blue1"
                     } ] }


Transform {
    children [ DEF blue9 Sportman
                     {
             rotation 0 1 0 -1.5708
             whichChoice -1
             position 35 1.8 4
```

```
              picturefile ["sportmanblue9.jpg"]
              nickname "blue9"
                      } ] }
```

the multiple thread control for the soccer game can be implemented as follows:

```
:- object waspsoccer : [bcilib].

var url = 'soccer.wrl'.

main :-
text_area(Browser),
set_output(Browser),

format('Load the game ... ~ n'),
loadURL(url),
Clock := new(game_clock),
      _Pulse := new(clock_pulse(Clock)),

      Clock <- get_time(TimeLeft),
format('the game will start in 5 seconds,~ n'),
format('the total playing time is ~w seconds,~ n', [TimeLeft]),
delay(5000),

format('the game startup,~ n'),
_ball := new(ball(ball,  Clock)),
_GoalKeeper1 := new(goalKeeper(goalKeeper1, Clock)),
_GoalKeeper2 := new(goalKeeper(goalKeeper2, Clock)),
_UserMe := new(soccerPlayerUser(me_red10, Clock)),
_Blue9 := new(soccerPlayer(blue9, Clock)),
_Blue8 := new(soccerPlayer(blue8, Clock)),
_Blue7 := new(soccerPlayer(blue7, Clock)),
_Red2 := new(soccerPlayer(red2, Clock)),
_Red3 := new(soccerPlayer(red3, Clock)),
_Blue11 := new(soccerPlayer(blue11, Clock)),
_Red11 := new(soccerPlayer(red11, Clock)).

:- end_object waspsoccer.
```

A general framework of the soccer playing agents based on decision-making models can be programmed in DLP as follows:

```
:- object soccerPlayer : [bcilib].
```

```
soccerPlayer(Name, Clock) :-
      setSFInt32(Name,whichChoice, 0),
      format('~w thread active.~ n', [Name]),
      activity(Name,Clock).

activity(Name,Clock) :-
      repeat,
            sleep(2000),
            Clock <- get_time(TimeLeft),
            format(' player ~w  thread ~w seconds left~ n',
        [Name,TimeLeft]),
            getPositionInformation(Name,ball,X,Y,Z,Xball,Yball,Zball,
        Dist,Xgoal,Zgoal,DistGoal),
            findHowtoReact(Name,ball,X,Y,Z,Xball,Yball,Zball,Dist,
        Xgoal,Zgoal,DistGoal,Action),
            format('player ~w action: ~w ~ n',[Name,Action]),
            doAction(Action,Name,ball,X,Y,Z,Xball,Yball,Zball,Dist,
        Xgoal,Zgoal,DistGoal),
      TimeLeft < 1,
      quitGame(Name),
      !.

......

:- end_object soccerPlayer.
```

The decision tree can be programmed in DLP as follows:

```
findHowtoReact(_,Ball,_,_,_,_,_,_,Dist,_,_,Dist1,shooting):-
      Dist =< kickableDistance,
      Dist1 =< kickableGoalDistance,
      !.

findHowtoReact(_,_,Ball,_,_,_,_,_,_,Dist,_,_,Dist1,passing):-
      Dist =< kickableDistance,
      Dist1 > kickableGoalDistance,
      !.
   findHowtoReact(Player,_,_,_,_,X1,_,_,Dist,_,_,_,run_to_ball):-
      Dist > kickableDistance,
      getFieldAreaInformation(Player,_,_,FieldMin,FieldMax),
      FieldMin =< X1,
      FieldMax >= X1,
      !.

findHowtoReact(Player,_,_,_,_,X1,_,_,Dist,_,_,_,move_around):-
```

```
      Dist > kickableDistance,
      getFieldAreaInformation(Player,_,_,FieldMin,_),
      X1 < FieldMin,
      !.

findHowtoReact(Player,_,_,_,_,X1,_,_,Dist,_,_,_,move_around):-
      Dist > kickableDistance,
      getFieldAreaInformation(Player,_,_,_,FieldMax),
      X1 > FieldMax,
      !.
```

" soccer player users" can be as follows:

```
:- object soccerPlayerUser : [bcilib].

var kickableDistance = 3.0.
var kickBallForce = 10.0.
var kickBallForceY =6.0.

soccerPlayerUser(Name, Clock) :-
      format('The user ~w thread active.~ n', [Name]),
      activity(Name,Clock).

activity(Name,Clock) :-
      repeat,
            sleep(1000),
            Clock <- get_time(TimeLeft),
            near_ball_then_kick(Name,ball),
      TimeLeft < 1,
      !.



near_ball_then_kick(Agent, Ball):-
      getViewpointPosition(Agent,X,_,Z),
      getPosition(Ball,X1,Y1,Z1),
      X < X1,
      distance2d(X,Z,X1,Z1,Dist),
      Dist < kickableDistance,
      getViewpointOrientation(Agent,_,_,_,R),
      R1 is R -1.5708,
      ball <- isUnlocked(Ball),
      ball <- lock(Ball),
      ball <- kickedwithStaticStartRF(Ball,translation,X1,Y1,Z1,R1,
        kickBallForce,kickBallForceY),
      ball <- unlock(Ball).
```

```
near_ball_then_kick(Agent, Ball):-
     getViewpointPosition(Agent,X,_,Z),
     getPosition(Ball,X1,Y1,Z1),
     X >= X1,
      distance2d(X,Z,X1,Z1,Dist),
      Dist < kickableDistance,
      getViewpointOrientation(Agent,_,_,_,R),
      R1 is -R -1.5708,
      ball <- isUnlocked(Ball),
      ball <- lock(Ball),
     ball <- kickedwithStaticStartRF(Ball,translation,X1,Y1,Z1,R1,
        kickBallForce,kickBallForceY),
      ball <- unlock(Ball).


near_ball_then_kick(_, _).

......

:- end_object soccerPlayerUser.
```

The predicate *near_ball_then_kick* is used to check whether or not the user is close enough to kick the ball. If yes and ball is unlocked, then kick the ball based on the user's current orientation.

B.2 DOG WORLD

The virtual world of the dogworld can be specified in VRML as follows.

```
PROTO Doggie [
     exposedField SFVec3f position 0 0 0
     exposedField SFRotation rotation 0 1 0 0
     exposedField SFInt32 whichChoice 0
     exposedField SFInt32  id      0
     exposedField SFBool  bark  FALSE]
{Transform { translation IS position
rotation IS rotation
children [ Dog {whichChoice IS whichChoice id IS id}
 Sound {minFront 10 minBack 10 intensity 20
       source AudioClip { loop IS bark url ["bark.wav"] }} ]}}
```

```
Transform { children [
DEF dog1 Doggie {position 0.0 0.0 -10.0 whichChoice 0 id 1}

DEF dog2 Doggie {position -1.0 0.0 -15.0 whichChoice 0 id 2}

...
DEF dog10 Doggie {position 5.0 0.0 -16.0 whichChoice 0 id 10}
]}
EXTERNPROTO Dog [
 exposedField SFRotation rotation
 exposedField SFInt32 whichChoice
 exposedField SFVec3f position
 exposedField SFInt32  id
]["./dog.wrl"]


:-object dogworld : [bcilib].

var url = './dog/dogworld.wrl'.

main :-
text_area(Browser),
set_output(Browser),

format('Load the game ...~n'),
loadURL(url),
Clock := new(game_clock),
      _Pulse := new(game_clock_pulse(Clock)),
      Clock <- get_game_time(TimeLeft),
format('the game will start in 5 seconds,~n'),
format('the total playing time is ~w seconds,~n', [TimeLeft]),
      sleep(5000),
format('the game startup,~n'),
_dog1 := new(dog(dog1, Clock)),
_dog2 := new(dog(dog2, Clock)),
_dog3 := new(dog(dog3, Clock)),
_dog4 := new(dog(dog4, Clock)),
_dog5 := new(dog(dog5, Clock)),
_dog6 := new(dog(dog6, Clock)),
_dog7 := new(dog(dog7, Clock)),
_dog8 := new(dog(dog8, Clock)),
_dog9 := new(dog(dog9, Clock)),
_dog10:= new(dog(dog10, Clock)).
```

```
:-end_object dogworld.

:-object game_clock.

var time_left = 5000.

get_game_time(Time) :-
Time := time_left.
set_game_time(Time) :-
time_left := Time.

:-end_object game_clock.


:-object game_clock_pulse.

game_clock_pulse(Clock) :-
repeat,
sleep(1000),
Clock <- get_game_time(Time),

Left is Time - 1,
Clock <- set_game_time(Left),
Left < 1,
!.

:-end_object game_clock_pulse.




:-object dog : [bcilib].

var sleeptime = 250.
var small_movement = 0.20.
var big_movement = 0.40.
var enlargement = 5.
var max_distance = 40.

dog(Dog, Clock) :-
setSFInt32(Dog,whichChoice, 0),
format('~w thread active.~n', [Dog]),
activity(Dog,Clock).

activity(Dog,Clock) :-
```

```
repeat,
sleep(500),
Clock <- get_game_time(TimeLeft),
getInformation(Dog, DogPosition, MasterPosition1,MasterRotation1, MasterPosition2, MasterRota
findHowtoReact(Dog,DogPosition,MasterPosition1, MasterRotation1, MasterPosition2, MasterRotat
nonvar(Action),
format('player ~w action: ~w~n',[Dog,Action]),
doAction(Dog,DogPosition,MasterPosition1,MasterRotation1,MasterPosition2, MasterRotation2,Act
TimeLeft < 1,
!.

getInformation(Dog,position(X,Y,Z),position(X1,Y1,Z1),rotation(X2,Y2,Z2,R2),position(X3,Y3,Z3
getSFVec3f(Dog,position,X,Y,Z),
getSFVec3f(proxSensor,position,X1,Y1,Z1),
getSFRotation(proxSensor,orientation,X2,Y2,Z2,R2),
sleep(sleeptime),
getSFVec3f(proxSensor,position,X3,Y3,Z3),
getSFRotation(proxSensor,orientation,X4,Y4,Z4,R4).


findHowtoReact(_,position(X,_Y,Z),_,_,position(X3,_Y3,Z3),_,move_to_master):-
distance2D(X,Z,X3,Z3,Dist),
Dist > max_distance,
!.


findHowtoReact(_,_,position(X1,_Y1,Z1),_,position(X3,_Y3,Z3),_,move_to_master):-
distance2D(X1,Z1,X3,Z3,Dist),
Dist < small_movement,
!.

findHowtoReact(_,_,position(X1,_Y1,Z1),_,position(X3,_Y3,Z3),_,move_with_master):-
distance2D(X1,Z1,X3,Z3, Dist),
Dist > big_movement,
!.

findHowtoReact(_,_,position(X1,_Y1,Z1),_,position(X3,_Y3,Z3),_,look_at_master):-
distance2D(X1,Z1,X3,Z3,Dist),
Dist < big_movement,
Dist > small_movement,
!.
```

```
findHowtoReact(_,_,_,_,_,_,look_at_master):-!.



bark(Dog,Time):-
setSFBool(Dog, bark, true),
sleep(Time),
setSFBool(Dog, bark, false).



doAction(Dog, position(X,_Y,Z),position(X1,_Y1,Z1),_,_,_,look_at_master):-
look_at_position(Dog,X,Z,X1,Z1),
bark(Dog,500).

doAction(Dog, position(X,Y,Z),position(X1,_Y1,Z1),_,_,_,move_to_master):-
getSFInt32(Dog,id,ID),
getFlockCenter(position(X2,_Y2,Z2), master_standing,[]),
dFunction(ID, Xd,Zd),
X3 is X2 + Xd,
Z3 is Z2 + Zd,
look_at_position(Dog,X,Z,X1,Z1),
bark(Dog,500),
move_to_position(Dog,position(X,Y,Z),position(X3,Y,Z3),5),
!.

doAction(Dog, position(X,Y,Z),position(X1,_Y1,Z1),_,
    position(X3,Y3,Z3),_,move_with_master):-
Xdif is X3-X1,
Zdif is Z3-Z1,
getFlockCenter(position(X5,_Y5,Z5), master_moving, [position(X3,Y3,Z3),Xdif,Zdi
getSFInt32(Dog,id,ID),
dFunction(ID, Xd,Zd),
X6 is X5 + Xd,
Z6 is Z5 + Zd,
look_at_position(Dog,X,Z,X6,Z6),
move_to_position(Dog,position(X,Y,Z),position(X6,Y,Z6),10),
!.


look_at_position(O,X,Z,X1,Z1):-
Xdif is X-X1,
Zdif is Z1-Z,
Xdif =\= 0.0,
R is atan(Zdif/Xdif) - sign(Xdif)*1.57,
setRotation(O, 0.0, 1.0, 0.0, R).

look_at_position(_,_,_,_,_):-!.

move_to_position(O,_,position(X,Y,Z),0):-
            setSFVec3f(O,position,X,Y,Z).

move_to_position(O, position(X1,Y1,Z1),position(X2,Y2,Z2),C):-
C > 0,
C1 is C-1,
Xdif is X2 -X1,
Zdif is Z2 -Z1,
X is X1 + Xdif/C,
Z is Z1 + Zdif/C,
setSFVec3f(O,position,X,Y1,Z),
            sleep(500),
move_to_position(O,position(X,Y1,Z),position(X2,Y2,Z2),C1).

getFlockCenter(position(X,Y,Z),master_standing, []):-
getSFVec3f(proxSensor,position,X,Y,Z),
!.

getFlockCenter(position(X,Y,Z),master_moving, [position(X1,Y,Z1), Xdif, Zdif]):-
X is X1 + Xdif* enlargement,
Z is Z1 + Zdif* enlargement,
!.


dFunction(1,0,3):-!.
dFunction(2,-1,-2):-!.
dFunction(3,1,-7):-!.
dFunction(4,-2,3):-!.
dFunction(5,2,2):-!.
dFunction(6,-3,0):-!.
dFunction(7,3,-1):-!.
dFunction(8,-4,4):-!.
dFunction(9,5,-4):-!.
dFunction(10,5,-3):-!.



:-end_object dog.
```

Appendix C

This is an appendix without a title.

Here is the content.

References

[ActiveWorld, 2000] ActiveWorlds, http://www.activeworlds.com.

[AlphaWorld, 1999] Alpha Enterprises, World of Alpha, http://www.alphaworld.com, 1999.

[de Antonio, et al., 2001] A. de Antonio, R. Aylett, D. Ballin (Eds.): *Intelligent Virtual Agents, Proceedings of Third International Workshop, IVA 2001*, Springer LNAI 2190, 2001.

[Bandai, 1997] Bandai America Incorporated website, 1997. http://bandai.com.

[Beer, 1999] M. Beer, M. d'Inverno, N. Jennings, M. Luck, C. Preist and M. Schroeder, Negotiation in Multi-Agent Systems, *Knowledge Engineering Review*, 14(3), 285-289, 1999.

[Bell, 1995] Bell, J., A Planning Theory of Practical Rationality. Proc. AAAI'95 Fall Symposium on *Rational Agency: Concepts, Theories, Models and Applications*, 1-4.

[Bell and Huang, 1997] Bell, J., and Huang, Z., Dynamic goal hierarchies, in: L. Cavedon, A. Rao, W. Wobcke (eds.), *Intelligent Agent Systems, Theoretical and Practical Issues* , LNAI 1209, Springer, 1997, 88-103.

[Blaxxun, 2000] Blaxxun Interactive Inc. web site http://www.blaxxun.com.

[Broll et al., 2000] Wolfgang Broll, Eckhard Meier, Thomas Schardt, *Symbolic Avatars Acting in Shared Virtual Environments*, http://orgwis.gmd.de/projects/VR, 2000.

[Brusilovsky, 1996] Brusilovsky, P., Methods and Techniques of Adaptive Hypermedia, *User Modeling and User-Adapted Interaction*, 6, 87-129, 1996.

[Caglayan, 1997] Caglayan, A., and Harrison, C., *Agent Source Book – a complete guide to Desktop, Internet and Intranet Agents* , Wiley, 1997.

[Capin et al., 1997] Tolga K. Capin, Hansrudi Noser, Daniel Thalmann, Igor Sunday Pandzic, Nadia Magnenat Thalmann, Virtual Human Representation and Communication in VLNet, *IEEE Computer Graphics*, March-April 1997 (Vol. 17, No. 2)

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

[Cheong, 1996] Cheong, F-C.,*Internet Agents: Spiders, Wanderers, Brokers and Bots*, New Riders, 1996.

[Cohen and Levesque, 1990] Cohen, P., and Levesque, H., Intention is choice with commitment. *Artificial Intelligence*, 42 (3), 1990.

[Creature Labs, 1999] Creature Labs website, 1999. http://www.creaturelabs.com.

[Curious Labs] Curious Labs: http://www.curiouslabs.com/.

[DLP web site, 2001] DLP web site: http://www.cs.vu.nl/∼eliens/projects/logic/index.html.

[Earnshaw, et al., 1998] 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.

[Eliëns, 1992] Anton Eliëns, *DLP, A language for distributed logic programming*, Wiley, 1992.

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

[Eliëns et al., 2002] A. Eliëns, Z. Huang, and C. Visser, A platform for Embodied Conversational Agents based on Distributed Logic Programming, Proceedings of AAMAS 2002 WORKSHOP: Embodied conversational agents - let's specify and evaluate them, 2002.

[Faure, 1997] F. Faure, et al., Dynamic analysis of human walking, Proceedings of the 8th Workshop on Computer Animation and Simulation, Budapest, 1997.

[FIFA, 2001] FIFA, Laws of soccer games, http://www.fifa.com, 2001.

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

[FIPA, 1999] FIPA, FIPA Content Language Library, Foundation for Intelligent Physical Agents, 1999.

[Freach, 1988] French, S., *Decision Theory, an Introduction to the mathematics of Rationality*, Ellis Horwood Limited, 1988.

[Graham, 1996] Graham, M., and Wavish, P., A situated approach to implementing characters in computer games. *Applied Artificial Intelligence*, 10(1):53-74, 1996.

[H-anim, 2001] Humanoid animation working group: http://h-anim.org/Specifications/H-Anim1.1/, 2001.

[Harel, 1984] D. Harel, Dynamic Logic, *Handbook of Philosophical Logic*, Vol. II, D. Reidel Publishing Company, 1984, 497-604.

[Huang et al., 2000] Zhisheng Huang, Anton Eliëns, Alex van Ballegooij, Paul de Bra, A Taxonomy of Web Agents, *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, IEEE Computer Society, pp. 765–769, 2000.

[Huang et al., 2001] Zhisheng Huang, Anton Eliëns, and Paul de Bra, *An Architecture for Web Agents*, to appear in: *Proceedings of EUROMEDIA 2001*, 2001.

[Huang et al., 2001b] Zhisheng Huang, Anton Eliëns, and Cees Visser, Programmability of Inteligent Agent Avatars, *Proceedings of the Autonomous Agents'01 Workshop on Embodied Agents*, Montreal, Canada, 2001.

[Huang et al., 2002] Zhisheng Huang, Anton Eliëns, and Cees Visser, 3D Agent-based Virtual Communities, *Proceedings of the 2002 Web 3D Conference*, ACM Press, 2002.

[Huang et al., 2002b] Z. Huang, A. Eliëns, and C. Visser, STEP : a Scripting Language for Embodied Agents, Proceedings of the Workshop of Lifelike Animated Agents, Tokyo, 2002.

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

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

[Liu and Ye, 2001] Jiming Liu, Yiming Ye (Eds.): *E-Commerce Agents, Marketplace Solutions, Security Issues, and Supply and Demand*, Lecture Notes in Computer Science 2033, Springer 2001.

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

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

[Jennings et al., 1998] Jennings, N.R., Sycara, S., and Wooldridge, M., A Roadmap of Agent Research and Development, *Autonomous Agents and Multi-Agent Systems I*, Kluwer, 275-306, 1998.

[Klusch, 1999] Klusch, M., (Ed) *Intelligent Information Agents - Agent-Based Information Discovery and Management on the Internet*, Springer, 1999.

[Klusch et al, 2003] Klusch, M., Bergamaschi, S., Edwards, P., Petta, P. (Eds) *Intelligent Information Agents: An AgentLink Perspective*, Springer LNAI 2568, 2003.

[Knoblock, 1997] Knoblock, G.A., Ambite, J.L., 1997, Agents for Information Gathering , *Software Agents*, pp. 347 - 373, 1997.

[Maes, 1997] Maes, P., Humanizing the global computer, Interview in: *IEEE Internet Computing*1(4), 1997.

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

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

[MUD, 2000] Mud Connector, http://www.mudconnect.com, 2000.

[Negroponte, 1995] Negroponte, N., *Being Digital*, New Riders, 1995.

[Perlin and Goldberg, 1996] K. Perlin, and A. Goldberg, Improv: A System for Scripting Interactive Actors in Virtual Worlds, *ACM Computer Graphics*, Annual Conference Series, 205-216, 1996.

[Petrie, 1997] Petrie, C., What's an agent ... and what's so intelligent about it?, WebWord column, *IEEE Internet Computing* 1(4), 1997.

[Prendiner et al., 2002] H. Prendinger, S. Descamps, and M. Ishizuka, Scripting affective communication with life-like characters in web-based interaction systems, *Journal of Applied Artificial Intelligence*, [Rao and Georgeff, 1991] A. Rao, and M. Georgeff, Modeling Rational Agents within a BDI-Architecture, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, pp. 473–484, Morgan Kaufmann Publishers, 1991.

[Rao and Georgeff, 1991] A. Rao, and M. Georgeff, BDI Agents: From Theory to Practice, *Proceedings of the 1995 International Conference on Multiple Agent Systems*, pp. 312–319,1995.

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

[Reynolds, 1987] Reynolds, C.W., Flocks, Herds, and Schools: A Distributed Behavioral Model, *Computer Graphics*, 21(4), July 1987, pp. 25-34.

[Reynolds, 1999] Reynolds, C.W., Steering Behaviors for Autonomous Characters. March 19, 1999. http://www.red.com/cwr/steer/gdc99.

[Robocup, 1999] RoboCup, The Robot World Cup Initiative, http://www.csl.sony.co.jp/person/kitano /RoboCup/RoboCup.html, 1999.

[Roth, 1999] Volker Roth, Mutual protection of co-operating agents, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects* , Lecture Notes in Computer Science 1603, pages 275-285, Springer-Verlag Inc., 1999.

[Russell and Norvig, 1995] Russell, S., and Norvig, P., *Artificial Intelligence; A modem approach*, Prentice Hall, New Jersey, 1995.

[Schoemake, 1985] K. Shoemake, Animating Rotation using Quaternion Curves, *Computer Graphics* 19(3), 245-251, 1985.

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

[Subrahmanian, 1998] Subrahmanian, V.S., *Principles of Multimedia Database Systems*, Morgan Kaufmann, 1998.

[Takahashi, 1999] Tomoichi Takahashi, *LogMonitor: from player's action analysis to collaboration analysis and advice on formation*, Robocup 99, 1999.

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

[WASP, 2000] WASP project home page: $http://www.cs.vu.nl/\sim huang/wasp.$

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

Glossary

Agent	Agent is ...
Artificial Intelligence	Artificial Intelligence is a study on ...

Index