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.

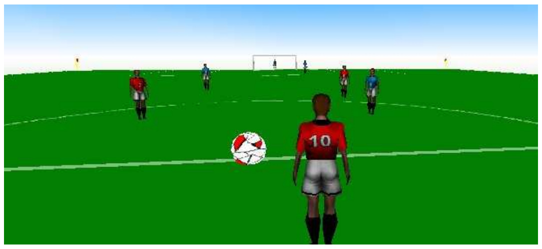


Fig. 7.1 Screenshot of Soccer Playing Game

- **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 {
```

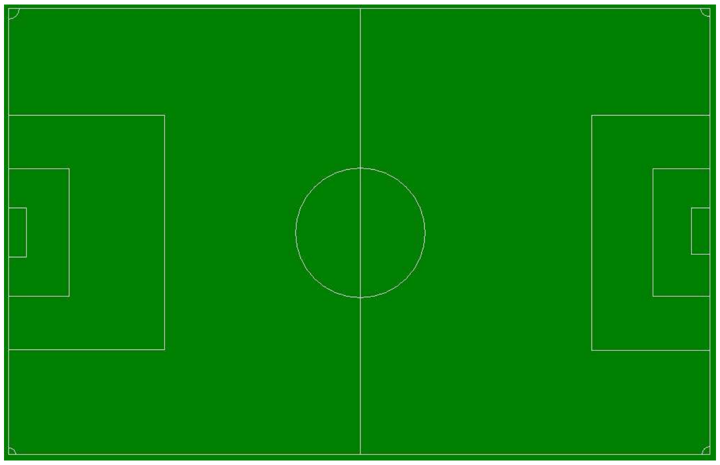


Fig. 7.2 The Field of Play

```
            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:

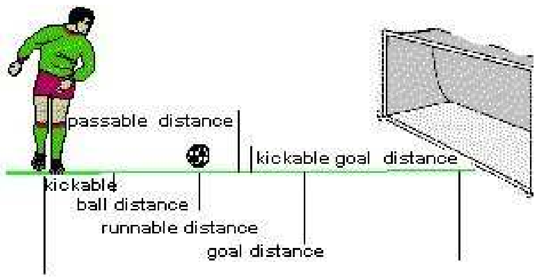
1. regularly checks the game clock.
2. regularly checks the validity of the position of the soccer ball. If the ball is kicked outside the field, the position of the ball should be set back to a proper position inside the field.
3. offer a kick predicate for players.
4. lock and unlock the status of 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.

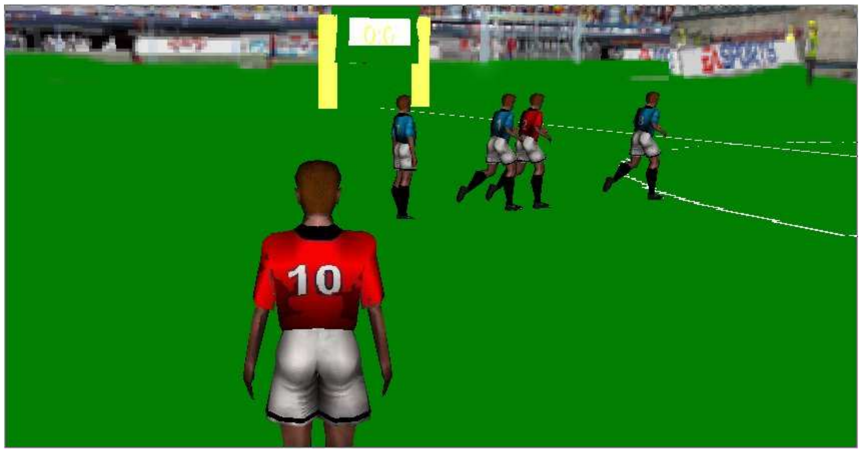


Fig. 7.3 Line-up phenomenon in Soccer Playing Game

- 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;

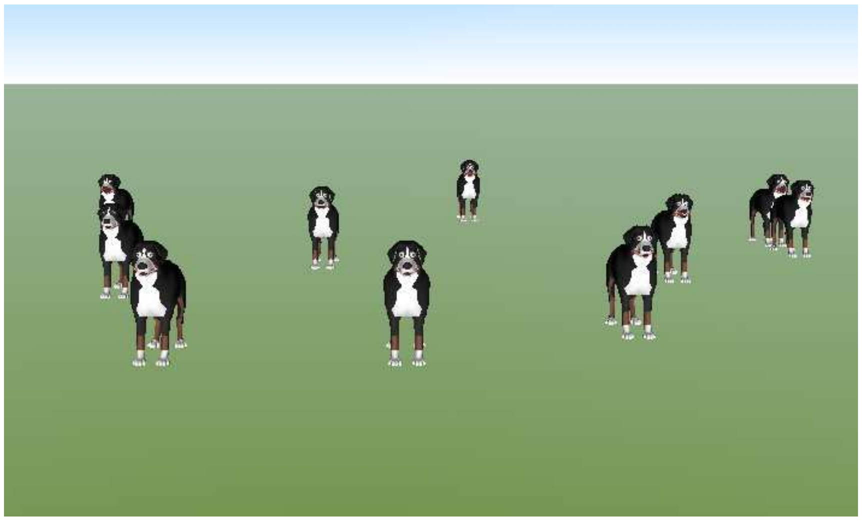


Fig. 7.4 Screenshot of the dogworld example

- 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,

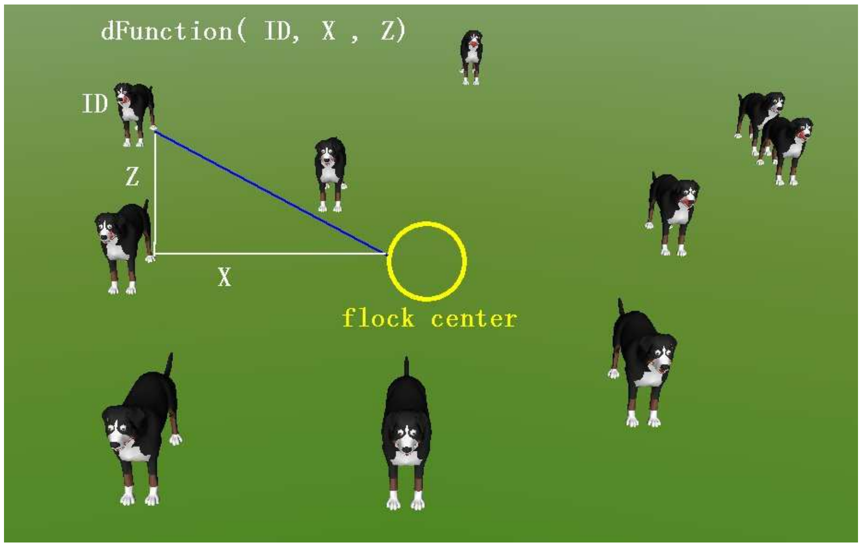


Fig. 7.5 Flock and Distributive Function

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.