```
:-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).
```

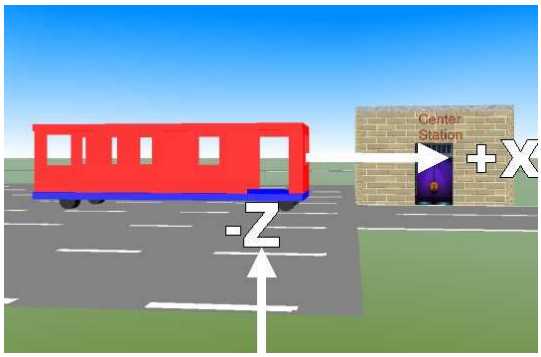

Fig. 4.2 Initial Situation of Bus Driving

```
:-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.

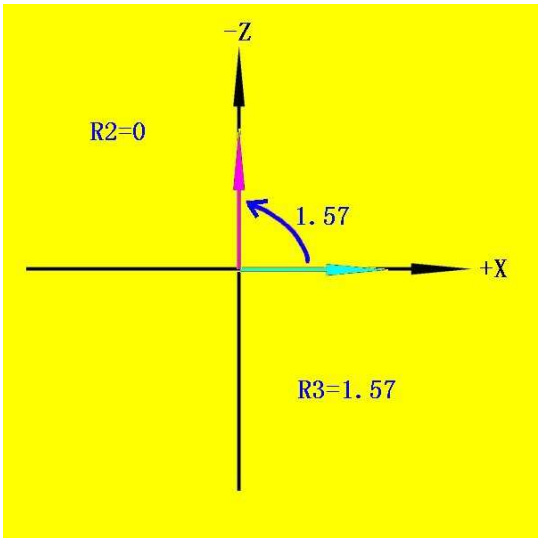

Fig. 4.3 The initial rotation values

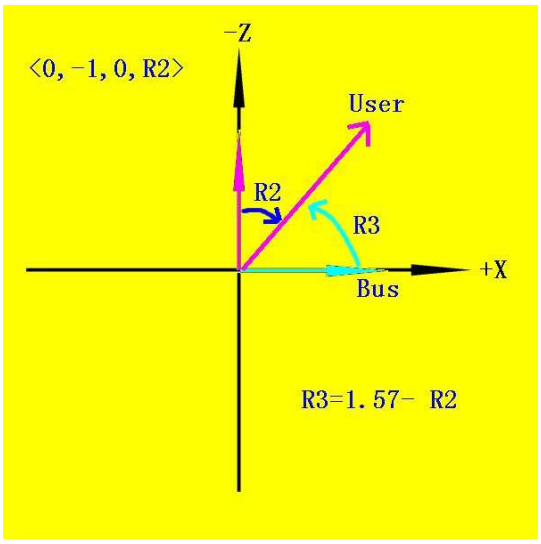

Fig. 4.4 User turns to the right

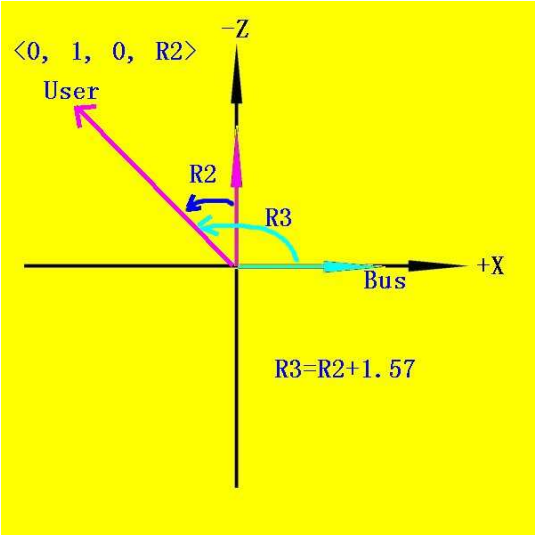

Fig. 4.5 User turns to the left

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:

```
......
```

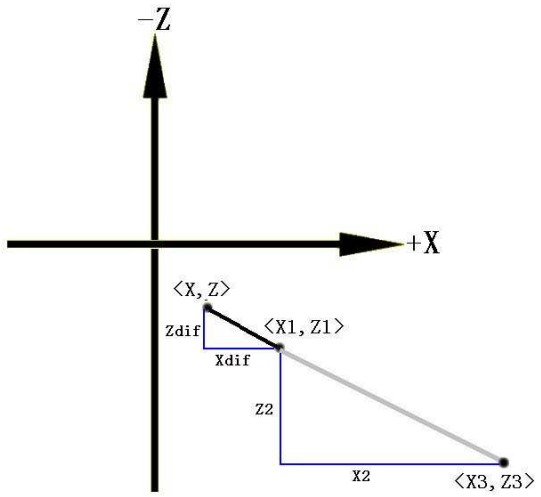

Fig. 4.6 kicking ball to a position

```
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.