Tutorial

Teach Yourself 3D Game Creating in 5 Days!

Translation 1996, 1997 by Guenther Mulder

Contents

This 5-day programm is about doing WDL: How to build a professional 3D game? So let us start with a simple assumption concerning your skills: you have read the reference manual from cover to cover and understood none of it. We may not be able to deal with each of the huge number of features of the 3D GameStudio, but we'll certainly give you enough to start with.

To begin with, you should have copied WED.EXE (or WEDS.EXE, WEDC.EXE resp.) complete with all its auxiliary files to a working directory of your harddisk. In order to do that first execute the INSTALL programm on the program diskette and then the one on the demo diskette; it will create the directory c:\GSTUDIO and the subdirectory DEMO. You may now include the program directory in you DOS path by adding C:\GSTUDIO to your PATH instruction of your AUTOEXEC.BAT.

Monday: Building a minimal level

On the first day you will learn - step by step - how to build a level with GameStudio. For the sake of simplicity the level should not have more than a few rooms.

For quick starting and testing of your level you should work under DOS. Once you have become a game pro you might get used to work on your level in a text editor under Windows95, on the World Editor WED in a second and a graphic program - e.g. Paintshop Pro - in a third window. But today you will start by using an everyday DOS text editor - e.g. EDIT - to edit your first WDL file, which will contain the description of or room:

EDIT MINI.WDL

All lines in the WDL file that begin with double slashes are comments. This allows you to render your definitions more structured.What you most definitely need in the file is a 'header' determining the video resolution and the name of the WMP and WDL file:

```
/////////////////////////////////////////////

// Minimum level

// created 12.8.1997 (myself)

/////////////////////////////////////////////

VIDEO              X320x400;

              MAPFILE              <miniw.WMP>;

              BIND                 <mini.WDL>;
```

The **BIND** keyword is merely a hint for the compiler of the professional version, telling him to compile the WDL file to the final game later. Although the case of the letters is not relevant, to make matters simple all keywords and predefined skills should be written in CAPITALS and the self-defined names in small letters.

Another necessary part of the ‘header’ are **INCLUDE** statements, which cause more WDL files to be integrated:

```
INCLUDE <move.WDL>;
```

The file MOVE.WDL contains the action **set_walking**, which initializes the player movement. You can find this WDL file on the DEMO disk. Its contents should not bother us at the moment, because smooth movement actions are a little bit complicated (as you'll see in the next chapter). You may simply use this action without knowing it in detail. Define it as an **IF_START** action to allow player movement in the level:

```
IF_START set_walking;
```

Now we can start with the actual definition of the level. To begin with, we specify the color palette complete with its shading ranges which is required for the representation of the textures:

```
PALETTE pal1 { PALFILE <vrpal.pcx>;

        RANGE 16,16;    RANGE 32,16;    RANGE 48,16;    RANGE 64,16;

        RANGE 80,16;    RANGE 96,16;    RANGE 112,16;   RANGE 128,8;

                                                        RANGE 136,8;    RANGE 144,8;    RANGE 152,8;
                                                        RANGE 160,16;

        RANGE 176,16;   RANGE 192,16;   RANGE 208,16;   RANGE 226,16;

        RANGE 240,16;
}
```

If you've defined more than one palette, the *last one* defined is automatically loaded at game start. The palette file you will find in the DEMO directory, as well as the bitmaps for textures.Do not forget that these files **-** as opposed to those on the SKAPH disk **-** are **not royalty free**. You may use it to practice but you may not use them in your own games!

To represent a room you need at least one wall texture and one square-shaped floor and ceiling texture:

```
BMAP arc_map,        <wandtex.lbm>,320,0,64,128;

BMAP square_map, <bodentex.lbm>,0,0,128,128;

////////////////////////////////////////////////

TEXTURE arc_tex {

SCALE_X 10;

SCALE_Y 10;

BMAPS                    arc_map;

}

TEXTURE square_ftex {

SCALE_X 16;

SCALE_Y 16;

BMAPS     square_map;

}

////////////////////////////////////////////////

WALL arc_wall {

TEXTURE                  arc_tex;

FLAGS      PORTCULLIS;

               POSITION         1;

}
```

The **SCALE_X**, **SCALE_Y** values tell the engine the ‘zoom factor’ of this bitmap **-** i.e. how many pixels fit into one **step**, the size unit of your artifical world. Here also, another convention for the purpose of clarity: Within definitions **-** and generally within winged brackets **-** all lines are indented by one tabstop to indicate that they belong to a superstructure. We have set the **PORTCULLIS** flag with **POSITION 1** in order to guarantee that the texture stops exactly at the ceiling. Alternatively you could use **ALIGN CEILING** in the WED.

At this point we are still missing something: the regions. Although we want to start by showing only one room, we need two regions: one for the room itself and one for the surrounding 'massive' border of the level:

```
REGION border {

FLOOR_HGT            40;

        CEIL_HGT              40;

        FLOOR_TEX            square_ftex;
        CEIL_TEX square_ftex;
CLIP_DIST       0;
}
REGION dungeon {
        FLOOR_HGT       0;
        CEIL_HGT        12;
        FLOOR_TEX       square_ftex;
        CEIL_TEX square_ftex;
}
```

Note that for the **border** region the heights of floor and ceiling have to be the same in order to give our level a inpenetrable closed wall on all sides. Because the interior of the **border** region will never be seen as a consequence, we may set the region's **CLIP_DIST** to 0 **-** the region will be left out of the process of rendering. It wouldn’t make much of a difference with a level this small, but will accelerate the rendering of teh screen in more complex, entangled levels.

After our WDL file **MINI.WDL** is finished, we can now create the topography:

WED MINI

(Lite version owners always write **WEDS**, commercial version owners **WEDC** instead of **WED**). By the way: if you should now get the DOS error message *‘Unknown command’*, it is because the WED directory is not in your DOS path; add it either (by editing AUTOEXEC.BAT), or always call up the programm complete with the name of the directory, e.g. **C:\GSTUDIO\WED**.

As WED can not find the topographic file **MINIW.WMP** given in the WDL header, it automatically creates a new file. At start up all you will see is an empty area with a blue point in the center: This is the player starting position.

Now move to the vertex mode with **[V]** and click several times clockwise around the player position at a distance of about 40 steps. Every click positions a vertex shaped like a green cross. After having formed a quadrangle or pentagon of vertices 'pull' a frame with pressed left mouse button around all vertices. As soon as you let go of the mousebutton, the frame disappears and green quadrangles appear around all vertices. The vertices are now marked.

Now press **[Shift]-[Ins]**. WED connects all marked vertices by a closed line and simultaneously switches into the wall mode. Now around each line you will see a green marking with a bar, which **-** if you have placed all vertices correctly clockwise **-** should point inwards.

If lines should cross, you didn’t work in sequence when placing the vertices. Go back to the vertex-mode (**[V]**) and shift the vertices with pressed left mouse button until everything is as it should be. Crossed walls cause errors in the picture! Subsequently switch back to wall mode with **[W]** and mark all lines by pulling a frame as already explained.

You now have a quadrangle or polygon consisting of lines highlighted in green, whose 'noses' point inwards. Each wall’s nose tells the wall’s right side, it does not have any furhter meaning. With the **[F]** key you may exchange the left for right side of a selected wall. Do this if any nose points in the wrong direction.

Now you have to assign the wall names and then regions to the inside and outside of your room. Select all walls, klick right, and select the only available wall name **arc_wall** from the scroll box. After assigning the walls check and scan the new region by pressing **[O] -** WED will enter the region mode now.

By moving the mouse pointer inside a closed area the area border will highlight. Just now you have only one closed area in your level. Move the mouse inside and click right. Select a region from the list by double klick or **[ENTER]**. Now you can toggle with the button **[Alternate / Default]** between regions heights previously entered with WED and default heights from your WDL region definition. Select **[Default]**, then klick **[OK]**. This way assign **dungeon** to your room and **border** to the outside.

You may afterwards check if everything is as it should be by switching to wall mode touching the walls with the mouse pointer. Especially **border** - the region surrounding everything - must have the same floor and ceiling height, otherwise the engine won't work. The wall and region parameters will be displayed on the data panel below. Run a check by selecting **[CHECK WALLS]** from the menu. If you should have forgotten or mis-assigned a wall or region, the faulty area will be marked.

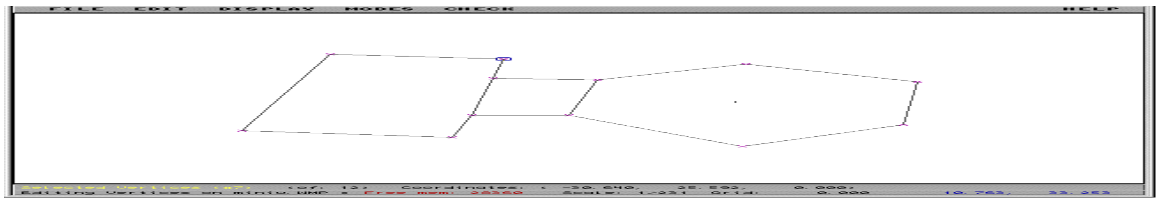
Now you have completed your first mini-level... well, almost. You still have to save it. Press **[F2]**, then intiate a trial run with **[Shift-^]**. If everything went right, you are now 'beamed' into your level. As long as you have not yet defined your own keys, you may switch the motion blur-effect with **[F5]**, save and recall your position with **[F2]** and **[F3]**, take a screenshot with **[F6]**, toggle between game and WED with **[Shift-^]** or terminate the game with **[F10]**. But take a good look at your room before doing that! First thing you will probably notice is that the wall textures are cut unsightly at the edges.

This we want to set right quickly: switch back to WED, activate the wall mode **([W])**, mark all outside walls by clicking with pressed [Shift] button or by pulling a frame around them. Them select the menu item **WALL - ALIGN HOR**.

You will notice that something happened to the value 'x y offs' down on he info panel. The walls have received a X offset, i.e. their textures were shifted horizontally. The value used for shifting was automatically calculated by WED in such a way as to assure that the textures will now fit seamlessly. Now restart your level with **[Shift-^]** and go see for yourself. Depending on how long your walls are there may still be a 'seam' where the first and last walls touch as a matter of the principle. We might compensate for that by changing the length of the last wall or the horizontal texture scaling, but let's call it quits for now.

Well, our single-room level does look a tad unsophisiticated. Ready for further derring-do? Let's ad another room! This room shall have a lower floor and a higher ceiling than the first one.

Create the new room - you should now know how to place the vertices and connect them - in some distance to the first one. Then we have to connect both rooms by a tunnel. Mark two opposite vertices (one from the first and one from the second room) and create a connection by **[Ins]**. Do the same with two neighboring vertices, this way creating a rectangular region connecting both rooms. By the way - you can create vertices in wall mode through touching a wall



```
Selected Vertices (#7)   (of: 12)   Coordinates: ( -30.640,   25.592,   0.000)
Editing Vertices on miniw.WMP x Free mem: 28360   Scale: 1/231   Grid:   0.000   10.763,   33.253
```

and pressing **[S]**. This will split the wall and create a new vertex in the middle.

This way you've got a shape of three closed regions. Mark all walls and assign the **arc_wall** to them.

Now you have again to tell the editor that you've added new regions by pressing **[O]**. You can now assign **dungeon** to all new rooms - but instead of using the predefined **default** heights, give the tunnel an **alternate** ceiling height of 8, and the new room a floor height of -5 and a ceiling height of 20 steps. The **border** region also have to be assigned again (with default heights), because you've changed some of its walls. Regions which need re-assignement are shown in red by WED.

Back inside the level (if the game was still running in the background, we will have to terminate it with **[F10]** and then restart it with **[Shift-^]** - restarting is necessary each time we've added new regions) we now face a gaping tunnel instead of a solid wall... if not, you've probably forgotten the alter the heughts of the new regions. Walls between regions of the same name and heights are shown solid by the engine.

Too bad that we now may enter our new room but can't get back out of it - the floor's threshold is too high! So either we will have to hop each time (**[Pos1]** button), or we'll define us a stair.

Enter wall mode, touch the outer wall of the tunnel and divide it by pressing **[S]**. Divide both parts again, and do the same with the other outer wall. Now both walls consist of 4 pieces. Connect each pair of opposite vertices, thus creating four regions across the tunnel. At last, give them the **dungeon** region with floor heights differing by one step... finished is the stair!

As far as the 'higher' functions for definition of menus, actors, weapons, combat and puzzle actions are concerned, you are on your own in the meantime. Study the file VRDEMO.WDL and learn how to build pillars, portals, elevators, outdoor areas and underwater regions! WDL is a powerful and flexible language, allowing you to do many things, up to the implementing artifical pseudo- intelligence in your actors. Some sample WDLs, which are meant to serve as suggestions, you'll find on the DEMO disk.

Tuesday: Moving the player

Stop! We don't want to leave you all alone yet. To give you an example for writing a WDL action we are now going to create one of the most difficult actions: The movement of the player.

Sure, you don't have to do that, because you can use the pre-fabricated move actions within the MOVE.WDL file. But if you want to become a master of WDL, here you will encounter each and every problem and prank imaginable with WDL arithmetics. So it's going to get a bit mathematical, but don't be afraid: knowledge of the four basic types of arithmetics will be sufficient. By the end of the chapter you are going to be able to infuse the player with any kind of movement behaviour whatsoever through your own actions.

The first move-action

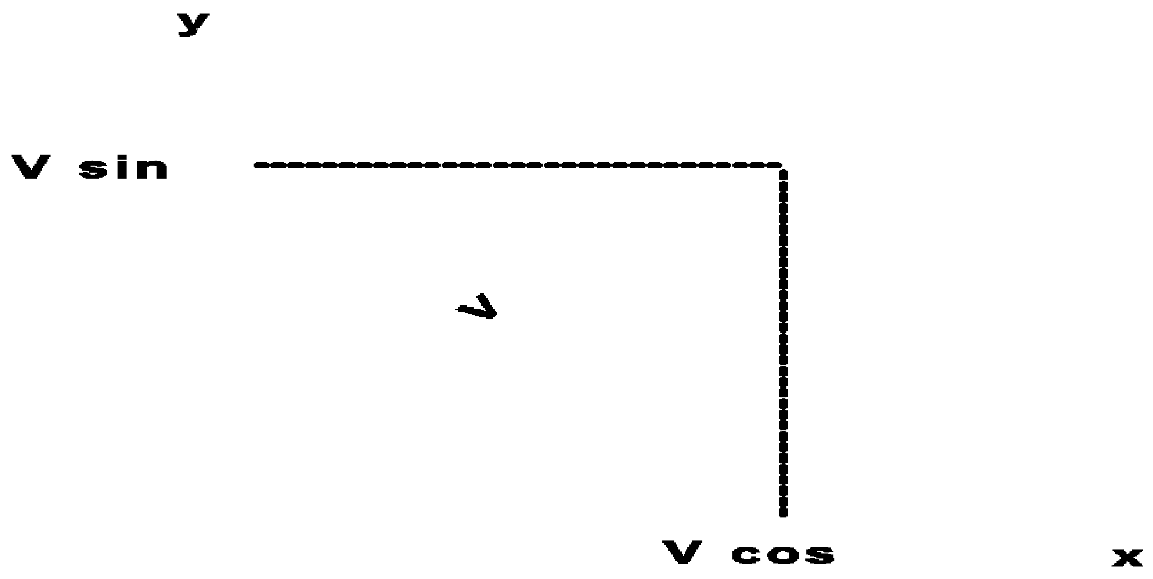
Again, we are going to use our minimal level. But this time we are not going to make use of the pre-fabricated movement actions of the **MOVE.WDL**. This time we are going to write our own ones. As the player movements are to take place continually, we shall use one of our 16 global **EACH_TICK** actions. Those 16 actions are executed after each frame cycle. First of all we are going to throw out the prefabricated **set_walking** of the **IF_START** action and insert our own action:

```
ACTION my_move {

RULE      PLAYER_VX = FORCE_AHEAD;

RULE      PLAYER_VY = FORCE_STRAFE;

RULE      PLAYER_VROT = 0.1*FORCE_ROT;

}

ACTION my_start {

SET       EACH_TICK.16, my_move;

}

IF_START my_start;
```

my_move is the simplest form of move actions: Joystick movements are directly translated into player speed. The predefined skills **FORCE_AHEAD**, **FORCE_STRAFE** and **FORCE_ROT** contain numeric values corresponding to our input for the player's forward and backward movements as well as rotation through mouse, joystick, or the cursor keys. If, for example, you push the **[↑]** key, **FORCE_AHEAD** aquires the value of **0.7**, and if you push the **[↓]** key, the value will be **-0.7**.

PLAYER_VX and **PLAYER_VY** are also predefined skills. Through those the player's speed along the x and y axis of the level's coordinate system can be defined. **PLAYER_VROT** defines the rotational speed of the player. Thus, these three **RULE** instructions supply the player with a certain speed along the axis and rotational speed that directly corresponds to the joystick input. We have multiplied the rotational speed with **0.1**, because otherwise the player would soon start to feel dizzy. To attain an input for **FORCE_STRAFE**, we use the **[,]** and **[.]** keys or move either joystick or mouse sideways while keeping **[Alt]** down.

Now what happens is that the movements do not follow the line of sight of the player. Those skills **- PLAYER_VX** and **PLAYER_VY -** defining the speed of the player refer to the same coordinates system we used when building the topography through WED. But movement along the line of sight consists of movements along the x as well as the y axis. Thus we will have to make sure that any movement of the joystick always has an impact on *both* skills **- PLAYER_VX** as well as **PLAYER_VY**, and that this impact relates to the line of sight given in **PLAYER_ANGLE**.



This we achieve by a **RULE** instruction.

In the figure the two fat arrows display the movements along the x and y axis necessary to achieve the shift along the player's line of sight as displayed through the broken line arrow. Thus in order to let the player move for the distance of ***V*** (length of the broken line arrow) in the direction a of his line of sight, it is necessary to shift him by ***V*** · cos a in he direction of X and by ***V*** · sin a in the direction of Y. **PLAYER_SIN** and **PLAYER_COS** supply us with two skills that automatically contain the sine and cosine of the player's line of sight. Thus our move-action now looks like that:

```
ACTION my_move {

RULE   PLAYER_VX = FORCE_AHEAD * PLAYER_COS;

RULE   PLAYER_VY = FORCE_AHEAD * PLAYER_SIN;

RULE   PLAYER_VROT = 0.1*FORCE_ROT;

}
```

Once we include the sideward movements, the first two **RULE**s result in:

```
RULE   PLAYER_VX = FORCE_AHEAD * PLAYER_COS - FORCE_STRAFE * PLAYER_SIN;

RULE   PLAYER_VY = FORCE_AHEAD * PLAYER_SIN + FORCE_STRAFE * PLAYER_COS;
```

These **RULE**s make up a *transformation of coordinates*. The player coordinate system - represented by the **FORCE_AHEAD** and **FORCE_STRAFE** skills - which was rotated by **PLAYER_ANGLE**, is being transformed into the 'resting' X/Y coordinate system, which forms the basis of the **PLAYER_VX** and **PLAYER_VY** skills.

Now we are able to make the player move in every direction. But the movements still appear to be jerky and unnatural, especially if we are using the cursor keys. We shall now try to figure out how to achieve smooth 'natural' movements.

Acceleration, Inertia, Friction

Let's suppose that our player is moving straight ahead at a constant speed. The distance he travels within a defined time increases with increasing speed and time:

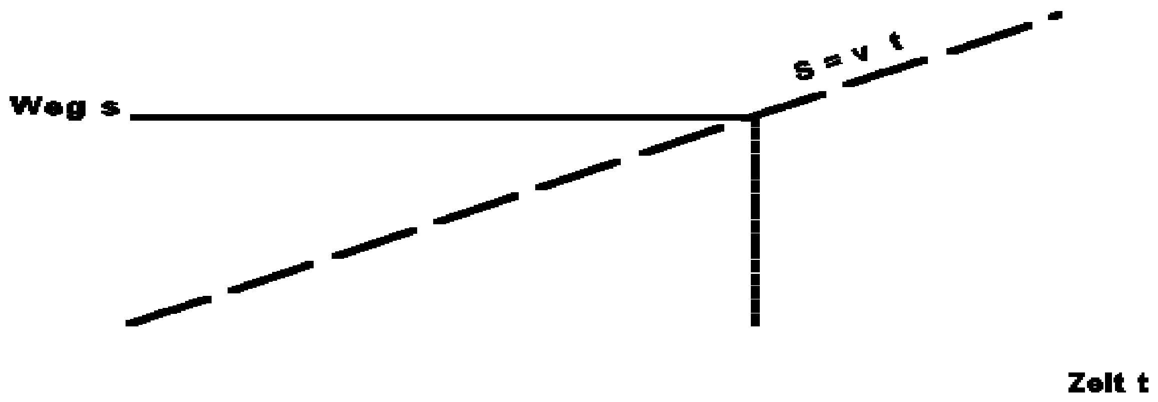
$s = v \times t$

v = Speed in steps per tick

s = Distance in steps

t = Time in ticks

In our virtual world - as you will remember - distances are being measured in Steps and time is being measured in Ticks. Thus the unit of measure for speed is Steps/Tick. If the speed v is known, we can draw a chart based on the formula above which gives the distance s for every time t. The broken line is the trajectory of the player in the space/time chart.



The faster the speed the steeper the trajectory will rise.

But what happens if we wish to change the speed? We will have to cause a *force* to make its impact on the player. The larger the force the larger the impact it has on the speed per unit of time. Yet, every body of course shows a certain resistance against such changes. This resistance, called *inertia*, is a result of the body's mass. The larger the mass of the body, the smaller the change of speed will be - supposing a constant force. This we may express in a formula:

$$a = \frac{K}{m}$$

a	= Change of speed per tick (acceleration)
K	= Force
m	= Mass

There are three basic types of forces we will have to reckon with in our virtual world. To begin with there is the *propelling force*. It is a voluntary speed variation that for example can be induced through joystick movement. The second force we shall call *drift*. It can be a drift that carries the player in a certain direction, or the gravity, pulling him down. Drift forces may vary from region to region, as it would be the case with a drift or an undertow.

The third force having an effect on the player is *friction*. This force continually tries to slow him down. Unlike the other two it increases with the players speed:

$\boldsymbol{R = -f \times m \times v}$

$\boldsymbol{R}$	= friction force
$\boldsymbol{f}$	= coefficient of friction (sliding friction)
$\boldsymbol{v}$	= speed
$\boldsymbol{m}$	= mass

The coefficient of friction ***f*** depends on the nature of the surface the player moves on: ice having a smaller coefficient of friction than stone. The negative indicator is supposed to show that this force counters the velocity. The mass ***m*** is also part of the equation, as the player will encounter greater friction and put bigger stress on the surface if he weighs more - as long as the size of his shoes remains unchanged.

All three forces - propelling force ***K***, drift ***D***, and fricition ***R*** - add up to change the player's velocity:

$$a = \frac{K + D + R}{m} = \frac{K + D}{m} - f \bullet v$$

This we may immediately translate into a corresponding action. For the variables used in the equations we will have to define new skills:

```
SKILL force_x       {}                                // force in direction of X

SKILL force_y       {}                                // force in direction of Y

SKILL drift_x       { VAL 0; }          // drift in direction of X

SKILL drift_y       { VAL 0; }          // drift in direction of Y

SKILL strength      { VAL 0.1; }        // force coefficient

SKILL fric          { VAL 0.2; }        // friction coefficient

SKILL mass          { VAL 1; }          // inertial mass of player

ACTION my_move {

   RULE force_x = strength*(FORCE_AHEAD*PLAYER_COS - FORCE_STRAFE*PLAYER_SIN);

   RULE force_y = strength*(FORCE_AHEAD*PLAYER_SIN + FORCE_STRAFE*PLAYER_COS);

   RULE PLAYER_VX = PLAYER_VX + (force_x + drift_x)/mass - fric * PLAYER_VX;

   RULE PLAYER_VY = PLAYER_VY + (force_y + drift_y)/mass - fric * PLAYER_VY;

RULE PLAYER_VROT = 0.1*FORCE_ROT;

}
```

But wait, what's that? The player won't stop any more with our new action! He goes on creeping along at minimal speed even if we have released the joystick. Are our equations faulty?

Not in theory. But unfortunately we are now confronted with the difference between theory and application - i.e. the limited accuracy of our arithmetics. Skills and **RULE** instructions are being computed with an accuracy of three decimal places. All further digits are ignored. Thus a value of less than **0.001** is automatically treated as **0** - even if it is a intermediate value within a **RULE**.

This is the case as soon as the **PLAYER_VX** drops below the value of **0.005**. If multiplied with **fric**, that is with **0.2**, a value of less than **0.001** is returned for this term (**fric * PLAYER_VX**), which is then read as **0**. **force_x** and **drift_x** also equalling **0**, no further change is made to the value of **PLAYER_VX** during the third **RULE** instruction. Thus the player moves on at the snail speed of **0.005** steps per tick 'til Kingdom come. Those developing software constantly have to reckon with this kind of insidious problems that are due to the principles of arithmetics. And sometimes those developing games as well.

The problem can be solved through a simple modification to our **RULE**s:

```
RULE PLAYER_VX = (1 - fric) * PLAYER_VX + (force_x + drift_x)/mass;

RULE PLAYER_VY = (1 - fric) * PLAYER_VY + (force_y + drift_y)/mass;
```

Mathematically this version is identical to the one before, but this one works!

Now we are able to adjust the drift, the player's strength (through **strength**), the player's mass (through **mass**), and the surface friction (through **fric**) individually according to the region. On ice for example **fric** could acquire the value of **0.05** and within a swamp the value of **0.5**. Moreover we are now moving much more fluently through the hallways thanks to the new move action. We are able to accelerate gently and to slow down the same way. But by now the action has also grown a bit larger as compared to the first version. Does that mean that we are finished now?

Sorry to say that we are not. At this point in our action we are changing velocities in both the direction of x and y by simply adding the computed acceleration after each frame cycle. But what happens if frame cycles take different amounts of time?

On fast Pentium PCs with short frame cycles the player accelerates by the same amount more often **-** and thus has an higher overall speed **-** than with slower computers. That means that the owners of slow PCs are furthermore being punished by snail-type player movements. Moreover we should not forget that within our game screen cycles of different duration may occur. Within a single level the frame rate may vary for up to three times its value depending on the location of the player.

Thus we need a kind of compensating factor **-** available to us through the predefined skill **TIME_CORR**. **TIME_CORR** has the value of **1**, if the frame rate is at exactly 16 fps, and changes in proportion to the deviation from this median. Thus at the rate of 8 fps **TIME_CORR** acquires the value of **2** and at the rate of 32 fps its value is **0.5**. With this kind of time compensation, our **RULE** would look like that:

```
RULE PLAYER_VX =(1 - TIME_CORR * fric) * PLAYER_VX + TIME_CORR * (force_x + drift_x)/mass;
```

Unfortunately we will again have to deal with special cases. If for example the product of (**TIME_CORR * fric**) approaches the value of **1**, the player won't stop properly but go on fidgeting while he is stationary. We are now approaching the limits of what can be computed by **RULE**s with a certain amount of clearness.

The equation above is therefore integrated into the **ACCEL** instruction of the program itself together with time compensation, creep suppression, supression of oscillations and compensation of all special cases. **ACCEL** accelerates a velocity by a given amount or skill. The predefined skills **FRICTION** and **INERTIA** are being used; they have to be set before each **ACCEL** instruction. Thus we get the following move action:

```
ACTION my_move {

SET    INERTIA,mass;

SET    FRICTION,fric;

                 RULE      force_x =    strength   *   (FORCE_AHEAD   *   PLAYER_COS   -
                 FORCE_STRAFE * PLAYER_SIN) + drift_x;

ACCEL  PLAYER_VX, force_x;
```

```
                RULE     force_y =   strength  *  (FORCE_AHEAD  *  PLAYER_SIN  +
                FORCE_STRAFE * PLAYER_COS) + drift_y;

ACCEL  PLAYER_VY, force_y;

SET    FRICTION,0.5;

RULE   FORCE_ROT = 0.1 * FORCE_ROT;

ACCEL  PLAYER_VROT, FORCE_ROT;

}
```

We have now added the drift directly to our propelling forces and furthermore used the **ACCEL** instruction to get a better rotational behaviour. Our move action is now finished... for the moment. The player now moves wonderfully in the direction of x and y. But what happens if he encounters a stairway? And what about those further degrees of freedom that enable a player to look up and down, to jump, to dive, and to fall down?

Vertical movement

While the horizontal movements of the player basically always remain the same **-** except for the differing amounts of friction **-** the vertical move actions differ fundamentally depending on the character of the player's movements. We shall try to implement the following types of movements in our action:

- **Driving**: In this case only the vertical intervals of the surface in the different regions will have to be compensated through the vertical movements , so that the player always trys to keep the same distance to the surface.

- **Walking**: In this case the player moves up and down rhythmically while walking. He also is to be able to jump and to duck.

Let's start with the compensation of vertical intervals. The skills **PLAYER_Z**, **PLAYER_SIZE**, **FLOOR_HGT** and **PLAYER_HGT** are responsible for that. The **PLAYER_HGT** skill returns the distance of the player's feet from the region floor surface. This distance may also be negative, if the player sinks into the ground.

We can take the easy way by simply calculating **PLAYER_Z** so that the distance of the player's eyes from the surface always equals **PLAYER_SIZE**:

```
RULE   PLAYER_Z = FLOOR_HGT + PLAYER_SIZE;
```

Once we add this instruction to our move-action we are able to pass regions of differing surface heights with our player, eg. stairs. **PLAYER_HGT** now alway returns the value of **0**. Unfortunately the player now adapts to the differing heights as unnaturally and jerky as with our initial attempts at forward movements. To attain gentler and more natural movements we again will have to take a look at the forces that have an impact on the player's vertical movements.

- As long as the player is airborne **-** i.e. **PLAYER_HGT** returns a value above **0 -** only *gravitation* and *air friction* have an impact on him. Gravitation is a drift we have already encountered: it gives the player a downward acceleration...

- ...until the player touches ground or sinks into it. In this case an additional *resilient force* takes effect. This is a new kind of force that gets stronger the deeper the player sinks into the ground. It induces him with an upward acceleration. Furthermore the friction increases significantly when the ground is penetrated.

The player must be able to even sink into surfaces of hard, impenetrable rock! In the real world this kind of surface would of course never give way, but the player's knee-joints would, which produces the same effect. So the resilient force results from the elasticity of his joints or **-** if the player is motorised **-** his vehicle's suspension. Equally we can explain the increased *ground friction* by the friction of his muscles or of the vehicle's shock-absorbers.

```
SKILL force_z       {}                              // Force in direction of Z
...
SKILL fric_air      { VAL 0.02; }       // air friction coefficient
SKILL fric_gnd      { VAL 0.97; }       // ground friction coefficient
                                            SKILL gravity           { VAL -0.2; }       // gravitational force
SKILL knee_fac      { VAL 0.3; }        // elasticity coefficient
ACTION my_move {
...                 // all the previous instructions,,,
SET                 FRICTION,fric_air;
SET                 force_z,gravity;
IF_ABOVE            PLAYER_HGT,0;                   // is the player airborne?
GOTO      airborne;
SET                 FRICTION,fric_gnd;
RULE                force_z = gravity - PLAYER_HGT*knee_fac;
airborne:
ACCEL               PLAYER_VZ,force_z;
}
```

Through the **IF_ABOVE** instruction and the following jump to the target label **airborne** we are able to distinguish whether the player is airborne or on the ground. In the latter case **FRICTION** will be altered and the resilient force, being proportional to the (negative) depth of immersion **PLAYER_HGT**, has to be added to **force_z**.

This new move action provides us with an easy way to find out what happens if we forget to remember about friction. To do this, please remove the line **SET FRICTION,fric_gnd;** from the action and move the player across a step! The result we are acquainted with from rubber balls and cars with broken shock-absorbers. It is the same result we get when we give **knee_fac** a value that is to large.

Thus we have dealt with the case of driving. In the case of walking a vertical movement is added, that corresponds to the movements of walking. The player is being compressed and stretched, so to say, depending on which phase of a step he is in. This kind of extension-dependant movements is available to us in the form of the predefined skill **WALK**. We employ this skill simply to change **PLAYER_SIZE** and add the following line at the end of our move-action:

```
RULE   PLAYER_SIZE = my_size + 0.3 * WALK;
```

The new skill **my_size** has to be defined by **VAL 4** in advance; this means that the player is usually 4 steps of height. The factor **0.3** defines the extent of the walking movements. By redefining the predefined skills **WALK_PERIOD** and **WALK_TIME** we may adjust the variables of walk time and walk period:

```
SKILL my_size             { VAL 4;     }
SKILL WALK_PERIOD   { VAL 5;     }
SKILL WALK_TIME           { VAL 10;    }
```

The **PLAYER_SIZE** skill also allows us to make the player duck through pressing the **[End]** button. This is to reduce his height. First, the simple way (to be inserted after the above line of **RULE PLAYER_SIZE=...**):

```
IF_BELOW      FORCE_UP,0;

RULE   PLAYER_SIZE = PLAYER_SIZE - 1.5;
```

Remember, **FORCE_UP** is set to **-0.7** by pressing the **[End]** key. But a superior look will be achieved if ducking and rising are accomplished fluently. A new skill is necessary for that:

```
SKILL         duck_val     { VAL 0; }

...

RULE          duck_val = 0.8*duck_val;   // reduce skill to 'fade out'

IF_BELOW      FORCE_UP,0;

RULE   duck_val = duck_val + 0.5*FORCE_UP;

RULE          PLAYER_SIZE = PLAYER_SIZE + duck_val;

...
```

To be able to understand this action you have to be aware that it will be repeated continually - each and every tick. **duck_val** usually has the value of **0**, so that **PLAYER_SIZE** is not altered. Once we push the **[End]** key, **FORCE_UP** acquires a negative value and the instruction following **IF_BELOW** is executed. This means that a certain amount is substracted from duck_val, which leads to a decrease of the player's height in the following line. Multiplying it with **0.8** in the first **RULE** instruction takes care that **duck_val** automatically moves towards **0**, the faster the larger its absolut value.

Why didn't we introduce a skill called **duck_speed** and use the familiar **ACCEL** instruction to achieve this? Because ducking in the real world has nothing to do with acceleration. You'll immediately realize that mulitplying the value with **0.8** produces a more natural look: Ducking and rising fade more gently. Theoretically we ought to install a time compensation with **TIME_CORR** at this point, but then again, there is no need to overdo it.

By the way, how deep does the player duck, given the above action? Obviously this depends on our factors **0.8**, **0.5**, and the value **FORCE_UP** acquires when **[End]** is pressed (usually **-0.7**). The ducking comes to an end when the reduction of **duck_val** through multiplication with **0.8** just about equals the negative increase through **0.5*FORCE_UP**:

$$(1 - 0.8) \times duck\_minimum = 0.5 \times 0.7 \Rightarrow \quad duck\_minimum = \underline{1.75}$$

What is still missing is the act of jumping. We could simply accelerate the player in the upward direction by a given value through pressing the **[Home]** button **-** i.e. giving **FORCE_UP** a positive value. But in the real world a jump consists of a ducking and subsequent fast upward movement, that is two 'contrary' movements. How do we integrate this in our move-action?

A jump is to consist of three subsequent phases, i.e. those of ducking, upward acceleration, and an unpropelled state, where the player is airborne and entirely left to the forces of gravitation. In order to be able to distinguish between those jump phases we shall introduce the new skill called **jump_phase**. This skill usually will be set to **0**, which means that no jump is happening at the moment. By pressing **[Home]**, this skill is to be set to **1**. It must then display the current phase of the jump. During the first phase (**jump_phase** = **1**) the player is to duck, during the following one (**2**) he is to jump up. Which is followed by phase (**3**), in which the player is flying freely. After the player again touches ground after the jump, **jump_phase** is to be reset to 0, so that a further jump may be initiated by pressing the same button again.

```
SKILL     jump_phase { VAL 0; }

...

IF_ABOVE          jump_phase,0;              // jump already going on?

GOTO      jump_1;
```

```
IF_BELOW        FORCE_UP, 0.1;     // NO pressing of [Home]?
GOTO    no_jump;
SET             jump_phase,1;              // jump starts
jump_1:
IF_ABOVE        jump_phase,1;
GOTO    jump_2;
RULE            duck_val = duck_val - 0.5;            // ducking
IF_BELOW        duck_val,-0.7;      // ducked deep enough? (heed minimum)
SET     jump_phase,2;              // then jump up now
GOTO            no_jump;
jump_2:
IF_ABOVE        jump_phase,2;
GOTO    jump_3;
SET             duck_val,0;                // then jump up now
RULE            PLAYER_Z = FOOT_HGT + my_size;        // rise again
SET             PLAYER_VZ,0.5;     // leaping off speed
SET             jump_phase,3;              // and now: flying freely
GOTO            no_jump;
jump_3:
IF_ABOVE        PLAYER_VZ,0;               // player still moving upwards?
GOTO    no_jump;
IF_ABOVE        PLAYER_HGT,0;              // player still airborne?
GOTO    no_jump;
SET             jump_phase,0;              // jump finished
no_jump:
...                                // now following RULE duck_val...
```

This bit has to be inserted into the action before the **PLAYER_SIZE** is reduced via **duck_val**, because otherwise the ducking procedure won't work.

Now we need another action that enables us to let the player look upwards or downwards. The skill called **PLAYER_TILT** is used for that. The vertical tilting of the player's line of sight is to work in the same way as the ducking procedure, i.e. the line of sight is to return to it's initial position after the button is released:

```
RULE    PLAYER_TILT = 0.8 * PLAYER_TILT + 0.3 * FORCE_TILT;
```

But we may use vertical tilting for some other effects as well. We might, for example, use the **ACCELERATION** skill to let the player tilt backwards when accelerated and tilt forwards when slowed down. That looks great in racing games.

But we are going to add a different kind of effect: If the player falls of a cliff like the one in the colonade in the VRDEMO he shall have to tilt over forwards by 180 so that he will be able to see the bottom advance towards him in a very dramatic way. But this, of course, should not happen if the player is jumping at that moment:

```
IF_EQUALjump_phase,0;          // no jump?

RULE                    PLAYER_TILT = PLAYER_TILT + 0.03 * (PLAYER_HGT + 0.3);
```

At this point we added the value of **0.3** to **PLAYER_HGT**, as the player usually is going to sink into the ground a bit because of gravity and the ground's elasticity. By the way, we did not make time compensation a part of tilting the line of sight either just to keep from making things too complicated...

Our move action, complete with all features, has by now gained quite a bit lengthwise. And now, to finish things off, one more consideration: How can we get skills like **fric** or **gravity** to change automatically depending upon the region the player currently is in? There are several possible ways to acvieve this. We could for example use the eight 'universal' parameters of a region to define that **SKILL5** is to define the friction and **SKILL6** is to define the gravitational pull. In that case we would merely have to add the following line to the beginning of our **my_move** action:

```
ACTION my_move {

SET        fric, HERE.SKILL5;

SET        gravity, HERE.SKILL6;

...
```

HERE is the synonyme for the name of the region the player currently is in. Unfortunately we would have to give values for **SKILL5** and **SKILL6** for each region's definition. Furthermore, these regional parameters would then be 'used up'. Preferably one should define an **IF_ENTER** and **IF_LEAVE** action only for those regions where for example the surface friction deviates from the standard value. Remember, **IF_ENTER** is triggered once the player enters and **IF_LEAVE** once the player leaves the region:

```
ACTION set_swamp_fric          { SET fric, 0.5;          }

ACTION set_normal_fric         { SET fric, 0.2;          }

REGION swamp {

...                            // remaining region parameters...

IF_ENTERset_sumpf_fric;

IF_LEAVE set_normal_fric;

}
```

Special effects

We are now going to fit in a few special effects. They are quite welcome in any game, because they baffle the player. To begin with, we shall give the player glasses with flexible lenses, that allow him to zoom his line of sight through pressing the **[Ins]** and **[Del]** keys. But we will have to limit the predefined skill **PLAYER_ARC** beforehand, because disturbances of the image may occur during its variation:

```
SKILL PLAYER_ARC{ MIN 0.5; MAX 2; }

ACTION my_move {

...                            // all the rest...
IF_ABOVE         KEY_INS,0;                    // [Ins] pressed?
RULE     PLAYER_ARC = PLAYER_ARC - 0.1;
                        IF_ABOVE         KEY_DEL,0;                    // [Del] pressed?
RULE     PLAYER_ARC = PLAYER_ARC + 0.1;
}
```

Let’s take on an earthquake as our next special effect. It’s one of those things that always look very impressive in movies or computer games. The earthquake is going to be triggered by pressing the **[E]** key. We will control this special effect by a new action:

```
SKILL richter              { VAL 0; }  // Richter scale
SKILL random_1     { }
SKILL random_2     { }
ACTION my_move {
...                // all the rest...
RULE     PLAYER_X = PLAYER_X + richter*(RANDOM - 0.5);
RULE     PLAYER_Y = PLAYER_Y + richter*(random_1 - 0.5);
RULE     PLAYER_Z = PLAYER_Z + 2*richter*(random_2 - 0.5);
SET      random_2,random_1;
SET      random_1,RANDOM;
}
ACTION quake {
SET      richter,0.1;              // start earthquake slowly...
WAITT    16;                               // 16 ticks = 1 second
SET      richter,0.3;
WAITT    32;
SET      richter,0.5;              // climax
WAITT    48;
SET      richter,0.2;              // wait, recover your breath...
WAITT    80;
SET      richter,0.6;              // 2nd climax
WAITT    32;
SET      richter,0.1;              // fade out
WAITT    16;
SET      richter,0;                // all's calm again
}

IF_E      quake;
```

To shake up the player effectively we have produced two more random numbers **- random_1** and **random_2 -** through our random value skill **RANDOM**, which we simply put in over 'earlier' values of **RANDOM**. Through this we move the player around in a random direction. How far is defined through our new skill **richter**. By substracting **0.5** we attain a value between **-0.5** and **+0.5** for the randomizing term in question (the range of values for **RANDOM** lies between **0** and **1**).

Within the action **quake** a **WAITT** instruction is used to let the **richter** skill increase and decrease within a certain timeframe. **WAITT** causes the action to 'lay dormant' for a given number of ticks (1/16th second as you remember).

To come to an end we are now going to drop one implied restriction of our move action we have so far observed: Where is it written that it is always the player who moves? We could steer an actor by joystick or mouse just the same or **-** in reverse **-** bestow our view on an actor. The latter possibility we shall now put into action. We are going to use the VRDEMO.WDL for this purpose, where our spherical 'tour guide' hovers about. How about taking a look at the world through his eyes? The change of perspective is going to be triggered by touching the 'guide', i.e. by an **IF_NEAR** event.

```
ACTION guide_eye {

SET       PLAYER_X,MY.X;

SET       PLAYER_Y,MY.Y;

SET       PLAYER_Z,MY.HEIGHT;

ADD       PLAYER_Z,THERE.FLOOR_HGT;         // change relative height to absolute

SET       HERE,THERE;                                          // set player region

SET       FRICTION,0.5;

RULE      FORCE_ROT = 0.1 * FORCE_ROT;

ACCEL     PLAYER_VROT, FORCE_ROT;           // we'll still like to rotate

}

ACTION change_body {

SET       EACH_TICK.16,NULL;                // remove old move action

SET       MY.EACH_TICK,guide_eye;     // start new one

SET       MY.PASSABLE,1;     // or else the player will rebound on himself

SET       MY.TEXTURE,blackpixel_tex;

}

ACTOR kugel {

...

DIST                0;

FLAGS               CAREFULLY;

IF_NEAR   change_body;

}
```

The actor's **BERKELEY** flag, which was a given up to this point, may now no longer be set, because obviously he can't look himself. We put his **DIST** to **0** to trigger an **IF_NEAR** action only if he is actually touched. The new texture **blackpixel_tex** we define in advance from a bitmap of an extension of some single pixels of the transparent color 0, so that the actor's texture doesn't constantly block our view like a board nailed to our head.

There is one more small detail we have to change about the actor's **WAY**, because on his usual way through the level he would now be walking into a trap! And why's that?

At this point the time has definitely come to leave to answer this question for yourselves. It should by now be piece of cake for you to write further move actions for swimming, diving, climbing, skiing, and remote cameras. Enjoy!

Wednesday: Portals, Elevators, and Waterfalls

Up to this point we have limited ourselves to constructing static worlds.

But as far as a 3-D game is concerned it would be a lot more impressive to have things that move as part of our world - like clock-works set in motion, or secret portals opening, or the player suddenly finding himself

under water at the touch of a button! Therefore we are going to create a library of universally employable WDL-actions in this chapter of the tutorial, that will help us to set in motion parts of our world in different ways.

Portals

The easiest and most frequent example of movement in our world is a door opening. And the easiest way to open a door is by animating a texture. So let's begin with the following: we shall define a wall that performs an animation at the players approach and performs the animation in reverse at the players departure. Towards the end of the animation the player is able to pass through the wall.

As an example for this kind of texture-animated door we are going to define ourselves a four-winged pneumatic bulkhead. Unfortunately you will have to draw the bitmaps and sample the sounds for this one yourselves. To begin with we are going define sound, bitmap and texture of the door and the door-wall ourselves:

```
SOUND tuer_zisch, <tuer_auf.wav>;

BMAP gtuer_map, <grauxit2.lbm>,0,0,256,256;    // closed door

BMAP gtuer_auf1, <grauxit3.lbm>,0,0,256,256;   // opening door

BMAP gtuer_auf2, <grauxit4.lbm>,0,0,256,256;

BMAP gtuer_auf3, <grauxit5.lbm>,0,0,256,256;

BMAP gtuer_auf4, <grauxit6.lbm>,0,0,256,256;

BMAP gtuer_auf5, <grauxit7.lbm>,0,0,256,256;

BMAP gtuer_auf6, <grauxit8.lbm>,0,0,256,256;

//////////////////////////////////////////////////////////

TEXTURE grautuer_auftex {

SCALE_XY      18,18;

CYCLES        7;

BMAPS
       gtuer_map,gtuer_auf1,gtuer_auf2,gtuer_auf3,gtuer_auf4,gtuer_auf5,gtuer_auf6;

DELAY         2,2,1,1,1,2,2;
FLAGS         ONESHOT;

SOUND         tuer_zisch;

SCYCLE        2;

SVOL          0.3;

SDIST         50;

}

TEXTURE grautuer_zutex {

SCALE_XY      18,18;

CYCLES        6;

BMAPS         gtuer_auf5,gtuer_auf4,gtuer_auf3,gtuer_auf2,gtuer_auf1,gtuer_map;

DELAY         2,1,1,1,1,2,2;

FLAGS         ONESHOT;

SOUND         tuer_zisch;

SCYCLE        1;

SVOL          0.2;

SDIST         50;

}

//////////////////////////////////////////////////////////

WALL grautuer {

TEXTURE       grautuer_auftex;

FLAGS         SAVE;

DIST          22;

IF_NEAR       grautuer_auf;

IF_FAR        grautuer_zu;

}
```

Because the direction of the animation is pre-set we are going to need two textures, one for the opening and one for the closing door. Basically those two textures only differ in the sequence of the phases of animation. In order for the animation for the opening and closing of the door to happen just once at a time we make use of the **ONESHOT** flag. With the **DELAY** times we had to take into consideration that the wings of the door speed up at the beginning and slow down at the end and move the fastest in between.

Something new is to be found with the definition of the door: here, we have defined two *events* **- IF_NEAR** and **IF_FAR -** that can trigger actions. We shall have to deal with the likes of these events more often from now on. Checking the **WALL** reference chapter we can read that the **IF_NEAR** action is triggered by the player approaching beyond a boundary set by **DIST**; the opposite is true for the **IF_FAR** action. The flag **SAVE** has to be set so that the position of the door can be remembered if the current state of the game is to be saved or loaded from harddisk, because our action changes the door's **TEXTURE**:

```
ACTION grautuer_auf {

SET    MY.TEXTURE,grautuer_auftex;

BRANCH texdoor_open;

}

ACTION grautuer_zu {

SET    MY.TEXTURE,grautuer_zutex;

BRANCH texdoor_close;

}

/////////////////////////////////////////////////////

ACTION texdoor_open {

SET    MY.DIST,22;          // hysteresis

SET    MY.TRANSPARENT,1;

SET    MY.PLAY,1;

SET    MY.EACH_CYCLE,texdoor_checkopen;

}

ACTION texdoor_checkopen {

SET    MY.PASSABLE,1;       // now the player may pass the door

}

ACTION texdoor_close {

SET    MY.DIST,20;

SET    MY.PASSABLE,0;       // close door again

SET    MY.PLAY,1;

SET    MY.EACH_CYCLE,texdoor_checkclose;

}

ACTION texdoor_checkclose {

SET    MY.TRANSPARENT,0;

}
```

What strikes the mind is that the action was split up apparently for no good reason: **grautuer_auf** branches to **texdoor_open**. It has proven to be useful with the development of larger games to divide actions into *specific* and *global* actions. We may now use the global action **texdoor_open** for every door that is opened through a texture animation. The action **grautuer_auf** on the other hand does only refer to specific doors with the textures **grautuer_auftex** or **grautuer_zutex**. If our game consists of more than one level, it is advisable to provide global actions like **texdoor_open** and **texdoor_close** in a separate WDL-file which will then be inserted into the main WDL files of the different levels through the keyword **INCLUDE** .

Because the actions are triggered by wall events we may use the synonym **MY** to label the triggering wall. When the door is being approached **grautuer_auf** switches to the textures containing the opening phases and initiates the animation through the flag **PLAY**. **TRANSPARENT** is set in advance of that, making it possible to look through the gap of the opening door at this stage already. Simultaneously to the start of the texture animation an **EACH_CYCLE** action is defined which will be triggered at the final phase - when the door is already completely open. This action simply sets the wall to **PASSABLE**, so that it may be passed by the player.

There is a simple trick - called *hysteresis* - which means that the **IF_NEAR** action increases **DIST** by a few steps. Thus the **IF_FAR** action would be triggered a little later. By this we avoid that the door opens and closes constantly if the player moves about within a critical distance.

If the player again comes within a critical distance to the door the **grautuer_zu** action is triggered. Basically what now follows is the reverse of what happened before. But we must not forget to reset **DIST** to its initial value. After the animation has been completed **TRANSPARENT** will be reset as well. The final phase of texture **gtuer_map** does not contain any transparent parts, yet, if we would leave the **TRANSPARENT** flag set, the room behind the door would be rendered with the build-up of the image, which in turn uses up time for computing. Thus the **TRANSPARENT** flag should only be used when absolutely necessary!

If we place the door in form of a wall within the level there obviously has to be the same region on both sides of it, otherwise the player would not be able pass through it by **PASSABLE**. And one more hint: if we construct walls that include **IF_NEAR** or **IF_FAR** actions we must not forget that the critical distance **DIST** does only refer to the *nearest vertex* of the wall. If a wall is 50 steps long and the critical distance is set at 20 steps, the player could touch the center of the wall without triggering an **IF_NEAR** event!

Elevators

In 3-D games it is quite popular to use elevators for opening up a new point of view to the player. And they are easy to do. All we have to do is to let a certain section of the surface rise or fall at a constant speed, until a certain target height has been reached, through the use of an **EACH_TICK** action. Our action for the player movements makes sure that the player will join the ride.

To make our elevator actions as universally applicable as possible we shall remove its specific parameters - i.e. speed, initial and target height - to the defined region parameters of the region concerned. There are eight such universal parameters that may be given numeric values and made into actions and be evaluated through keywords **SKILL1**..**SKILL8**. In this way an elevator could for example be defined like that:

```
REGION lift0_20 {

FLOOR_HGT    0;

CEIL_HGT     40;

FLOOR_TEX    boden_tex;

CEIL_TEX     decke_tex;

FLAGS        SAVE;

SKILL3       0;                  // Lower level of elevator in steps

SKILL4       20;         // Upper level

SKILL6       0.3;        // Elevator speed in steps per tick

IF_ENTER     lift_start;

}
```

The flag **SAVE** is important here as well, because the region will change during the game; so the position of the elevator has to be saved as well. Our elevator is supposed to go from height 0 to height 20 or vice versa, depending on at which height our player steps into it. In order for this to happen the regional parameter used **- SKILL3**, **SKILL4**, and **SKILL6 -** will have to be used in the following actions:

```
SOUND rumpel,<rumpel.wav>; // very short sound (<0.3 sec!)

SKILL lift_speed { }                    // intermediate value

ACTION lift_start {

SET           THERE.EACH_TICK,lift_downtick;

IF_BELOW      THERE.FLOOR_HGT,THERE.SKILL4;    // up or down?

SET    THERE.EACH_TICK,lift_uptick;

}

ACTION lift_uptick {

SET           lift_speed,THERE.SKILL6;

RULE          lift_speed = TIME_CORR*lift_speed;

ADD           THERE.FLOOR_HGT,lift_speed;

IF_EQUAL      THERE.HERE,1;             // is player in elevator?

RULE   PLAYER_Z = PLAYER_Z + 0.5*lift_speed;

IF_EQUAL      THERE.VISIBLE,1;          // if elevator is visible,

SET    RENDER_MODE,1;             // re-render image constantly

PLAY_SOUND    rumpel,0.3;

IF_BELOW      THERE.FLOOR_HGT,THERE.SKILL4;

END;                                           // upper Level not yet reached

SET           THERE.FLOOR_HGT,THERE.SKILL4;    // nominal value

SET           THERE.EACH_TICK,NULL;

}

ACTION lift_downtick {

SET           lift_speed,THERE.SKILL6;

RULE          lift_speed = -TIME_CORR*lift_speed;

ADD           THERE.FLOOR_HGT,lift_speed;

IF_EQUAL      THERE.HERE,1;

RULE   PLAYER_Z = PLAYER_Z + 0.8*lift_speed;   // suppress 'shaking'

IF_EQUAL      THERE.VISIBLE,1;

SET    RENDER_MODE,1;

PLAY_SOUND    rumpel,0.3;
IF_ABOVE      THERE.FLOOR_HGT,THERE.SKILL3;

END;

SET           THERE.FLOOR_HGT,THERE.SKILL3;

SET           THERE.EACH_TICK,NULL;

}
```

Because the action was triggered directly by the **IF_ENTER** event of the region we may use the synonym **THERE** for the region itself. Using the current height of the region, the action **lift_start** checks whether the lift is up or down and which way it has to go as a consequence - down or up. The movement of the region is then taken care of by the actions **lift_uptick** and **lift_downtick** respectively.

Within the action **lift_start** the same **EACH_TICK** action is changed twice by **SET**. Don't those two allocations somehow annul each other? This is of course not the case. It is very much like with life: only the last **SET** instruction rules the further proceeding of the action.

In both all but identical **EACH_TICK** actions first the time-adjusted speed **lift_speed** is calculated from the lifts speed **THERE.SKILL6** and the correctional factor **TIME_CORR**. It is then - in some cases as a negative value - added to the bottom-height of the lift. This way the region moves upwards or downwards a bit.

The two subsequent lines seem a little strange. If the player is within the lift, i.e. if the flag **HERE** of the region is set to 1, the player's vertical position **PLAYER_Z** is changed. Why? Our move action is responsible for maintaining a more or less constant distance between the player and the floor. Yet, if the floor itself moves, especially with downward movements ungainly, jerky movements ensue due to the player's elastic adjustments. By

partially moving **PLAYER_Z** along with the lift's floor these jerky movements are suppressed.

If the elevator region is visible, the skill **RENDER_MODE** will be set to 1 so that the image will be rendered even if the player does not move. Additionally the elevator sound **rumpel** will be triggered. As this sound will be replayed every tick it should not be too long - 1/3 of a second at the most - so that the sounds do not overly blend into each other.

At the end of the action the elevators position will be checked to find out whether it went past its terminal positions - **THERE.SKILL4** and **THERE.SKILL3** respectively. In that case the bottom will be reset to the exact final position and the action will be terminated. By the way: the instruction **END** does of course not terminate the **EACH_TICK** action but merely causes it to start again at the next tick! Only if the **EACH_TICK** event concerned is set to **NULL** the action will no longer be executed.

The same way as with elevators we could open or close vertical gates by moving their region's **CEIL_HGT**. But there is a more elegant way to implement doors...

Folding-doors

Instead of letting the doors open upwards we might as well get them to swing open and close horizontally, just like in real life. It is not any more complicated to do than a simple texture animation. To begin with we are going to define the door as a stand-alone rectagular region. None of the four walls of this region is allowed to be linked with any other wall, because otherwise the door region could move into one of those when opening or closing.

Apart from the four walls we'll also need a hinge. The door-region will afterwards do its turn around it. For a hinge we will use an invisible **THING** which we will place just inside the region on one of the narrow sides of the door. When positioning the door-region via WED you should keep in mind to leave the opening area unoccupied - the door may not cross any region borders!

```
                THING tuer_angel {          // must be within the region!

        TEXTURE dummy_tex;
        DIST        15;
        IF_NEAR     swing_open;
        IF_FAR      swing_close;
}
          REGION holz_tuer    {
        FLOOR_TEX   holz_tex;
        CEIL_TEX    holz_tex;
        FLOOR_HGT   10;
        CEIL_HGT    10;
        FLAGS               SAVE;
            SKILL3      0;              // initial angle with door closed (0..3.14)
            SKILL4      1.57;   // final angle with door open (1.57 = 90°)
            SKILL5      0;              // current angle of door
            SKILL6      0.1;    // speed of door in radiant per tick
        GENIUS      tuer_angle;
}
```

All walls of the door-region, too, must get the SAVE flag, because they change. By giving the intial and end angle in **SKILL3** and **SKILL4** doors may be placed in the level with whichever orientation. The speed the door opens with we define in **SKILL6**. The algebraic sign of the difference between **SKILL4** and **SKILL3** tells us whether the door is opened by increasing or decreasing the angle. **SKILL5** corresponds to the initial angel of the door. The door **swingdoor** defined above is situated parallel to the x-axis, as its intial angle is 0°.

Basically opening a door with an **EACH_TICK** action works like with the elevator. Except that this time we do not change the parameter **FLOOR_HGT** but the angle using a **ROTATE** instruction.

```
SOUND knarz,<knarz.wav>;   // very short sound (<0.3 sec!)
SKILL door_speed    { }    // intermediate values
SKILL angle         { }
ACTION swing_open {                           // open door
          SET       THERE.SKILL7,THERE.SKILL4; // set final angle
          SET       THERE.EACH_TICK,swing_inctick;
   IF_BELOW  THERE.SKILL4,THERE.SKILL3; // decrease or increase angle?
       SET    THERE.EACH_TICK,swing_dectick;
}
ACTION swing_close {                                 // close door
          SET       THERE.SKILL7,THERE.SKILL3; // set final angle
          SET       THERE.EACH_TICK,swing_dectick;
   IF_BELOW  THERE.SKILL4,THERE.SKILL3; // decrease or increase angle?
SET    THERE.EACH_TICK,swing_inctick;
}
ACTION swing_inctick {
          RULE            door_speed = TIME_CORR*THERE.SKILL6;
          ADD       THERE.SKILL5,door_speed;   // increase current angle
          ROTATE    THERE,door_speed;
             PLAY_SOUND   knarz,0.3,THERE.GENIUS;
          SET       RENDER_MODE,1;
   IF_ABOVE  THERE.SKILL5,THERE.SKILL7; // final angle reached?
SET    THERE.EACH_TICK,NULL;
}
ACTION swing_dectick {
          RULE            door_speed = -TIME_CORR*THERE.SKILL6;
          ADD       THERE.SKILL5,door_speed;   // increase current angle
          ROTATE    THERE,door_speed;
             PLAY_SOUND   knarz,0.3,THERE.GENIUS;
          SET       RENDER_MODE,1;
   IF_ABOVE  THERE.SKILL5,THERE.SKILL7; // final angle reached?
SET    THERE.EACH_TICK,NULL;
}
```

The action **swing_open** opens the door and the action **swing_close** closes it, just like with the elevator. As the door can only open through increasing or decreasing the angle depending on its position we need a few more reference points to be able to activate the correct **EACH_TICK** event. The parameter **SKILL7** of the door region will be used to temporarily save the final angle, which can either be found in **SKILL3** or **SKILL4**.

The actual moving of the door happens in the actions **swing_inctick** or **swing_dectick**. They change the current angle of the door and arrest it at the final angle in case it went beyond that angle, as with the elevator.

Underwater Regions

In the demo level we are not only able to swim around in water holes **-** we can dive into the water. How do we achieve this kind of effect?

A water region is a special case of two regions ‘stacked’ on top of each other, with the upper one lying above, the lower one below water.

```
                              REGION under_water {          // must be defined _before_ the water region!
               FLOOR_HGT    -5;
               CEIL_HGT     1;
               FLOOR_TEX    mud_tex;
               CEIL_TEX     underwater_tex;
               IF_ARISE     regio_arise;
}
REGION water {
                         FLOOR_HGT    1;                    // same as under_water's CEIL_HGT!
               FLOOR_TEX    water_tex;
               CEIL_HGT     20;
               CEIL_TEX     sky_tex;
               BELOW                  under_water;
               IF_DIVE      regio_dive;
}
```

As soon as the player enters the water region, he sinks right to the bottom of the lower region. If the region's floor has an **IF_DIVE** action it is 'permeable', and the player will find himself under water straight away. A few special effests in the **IF_DIVE** action **-** blubber sound, change of colour palette, different movemental behaviour **-** will take care of a proper underwater feeling.

```
PALETTE blue_pal { PALFILE <blue.bbm> }
                    SKILL underwater      { VAL 0; }
ACTION regio_dive {
          SET       RENDER_MODE,1;
                                   IF_EQUAL       underwater,1;         // is the player already underwater?
                                   END;           // then end
                                   SET  underwater,1;                   // underwater from now on!
                                   FADE_PAL       blue_pal,0.7;         // set underwater palette
                                             RULE           PLAYER_ARC = PLAYER_ARC + 0.3;      // change point of view
                                             RULE           my_size = my_size - 0.3;         // Hysteresis
          RULE      PLAYER_SIZE = PLAYER_SIZE - 0.3;
          RULE      PLAYER_Z = PLAYER_Z - 0.3;
                              CALL      set_diving;// diving mode
}

ACTION regio_arise{

        SET         RENDER_MODE,1;

                        IF_EQUAL        underwater,0;       // is the player already above water?

                        END;

                    SET         underwater,0;

                        FADE_PAL        blue_pal,0;                 // switch off underwater palette

                                RULE            PLAYER_ARC = PLAYER_ARC - 0.3;      // normalize point of view

                                RULE            my_size = my_size + 0.3;        // Hysteresis

        RULE        PLAYER_SIZE = PLAYER_SIZE + 0.3;

        RULE        PLAYER_Z = PLAYER_Z + 0.3;

                        CALL            set_swimming;               // swimming mode - see below

}
```

We would like to trigger a few effects while under water: all colours are to appear in a milky-bluish shade, everything distant is to disappear in a blue haze (fog effect) and the point of view is to acquire a light 'frog eye' effect. Apart from that we also want to switch the movemental behaviour of the player if necessary. For the first two effects we have defined a separate underwater palette **blue_pal** which contains only the colour blue.

In the **regio_dive** action it is checked whether the player is maybe already deep down. If not, the **FADE_PAL** instruction fades the underwater palette **blue_pal** above the normal palette by about 70% and changes the point-of-view at the same time.

Following there are three lines containing changes to the player's height and vertical position, bearing the comment 'hysteresis'. We already came across this term at the start of this chapter when dealing with the portal-action. 'Hysteresis' is a programming trick to avoid a constant 'to-and-fro' of actions. In this case that would be the constant diving and arising of the player, which we avoid by setting the height of his eyes to a lower position when he dives.

The action **regio_arise** is responsible for the player's resurfacing. It resets all underwater changes. We can trigger it by letting the player touch the surface of the water from below. The best way to do that would be with a dive move action, which we will leave to the reader as practice. The MOVE.WDL of VRDEMO does contain move actions for diving and swimming.

Thursday: Artificial Intelligence

As we have by now been told by our customers, there is one thing at developing games that always keeps the brains busy: how to define the most intelligent behaviour possible for an opponent. And thus, before we really come to a close with this tutorial, we shall busy ourselves a little bit with the creation of intelligent artificial life.

In order to people our world with living creatures we make use of the **ACTOR** definition. Actors are those objects that are able to move about independently. As we can see in our WDL reference they are defined in similar ways to Things. There are just a few additional Flags and Keywords: **TARGET**, **WAYPOINT**, **SPEED**, and **VSPEED**.

With the use of the keyword **TARGET** we may give the Actor a certain basic movemental behaviour: he is able to move along a predefined way or to either advance towards or move away from a target. Limiting ourselves to this in our game would make the whole thing become boring rather quickly. Monsters wanting to be taken seriously as opponents **-** and which monster wouldn't? - need to behave in a 'realistic' way. They have to react to the players in intelligent ways, elude him while he is still strong and hunt him down when he shows signs of weakness. A sort of artificial intelligence is needed for that.

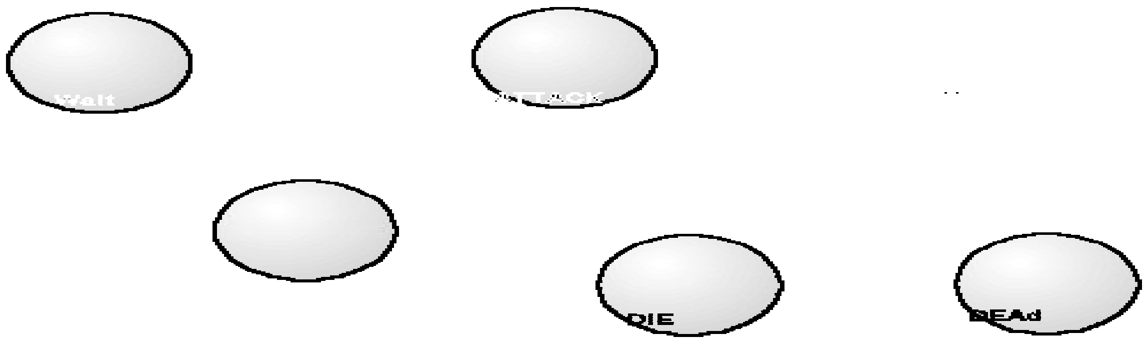
In this chapter of the tutorial you will learn to infuse your creatures with an electronic personality. We will therefor make an excursion to the theory of state machines. At the close of the chapter you will be able to create electronic beings that in their complexity are comparable to simple organisms.

The Theory of Black Boxes

The term Artificial Intelligence is synonymous to methods that give machines the ability to make sensible decisions. As far as our purposes are concerned it is of no importance whether the machine actually does posses intelligence. The aim is to make it *behave* as intelligently as possible. Whether it is actually possible to make statements as to a machine's intelligence based on it's behaviour has very much been a point of discussion between Behaviourists, Mechanists, Dualists, and the disciples of other theories for many years now, but this doubtlessly very interesting discussion does not have the slightest bit of an influence on WDL programming.

As the internal structure of the machine is of no interest to us for the time being we shall consider it a *black box* that is defined solely by it's observable behaviour, it's actions and reactions. If the behaviour of our machine is characterised by a certain simplicity it may be described as a *state machine*. The behaviour of a state machine we may consider to be a certain number of distinctly separate behavioural patterns. Each of these behavioural patterns equals an *inner state*. Thus a state machine has a limited number of states available; it is always 'in' one of theses states. These states in a way constitute the 'inner space' or inner life of the black box. For each state there is a corresponding circumstance, e.g. an external stimulus, which causes the machine to change into a different state.

With our actor-machine we could for example define the following behavioural states: Waiting, Attacking, Escaping, Dying, and Dead.



The arrows indicate the *transitions* between states. Such transitions are triggered by *Events* in the WDL. Here is a list of events that may cause an actor to respond by changing between states:

Event	**Realised by**
■ Actor views Player.	**EACH_TICK** action with periodic **SHOOT** instruction.
■ Player comes near actor	**IF_NEAR** action with **DIST**>0.

(or vice versa).	
■ Player and actor touch.	**IF_NEAR** action with **DIST**=0.
■ Player has removed himself from the actor beyond a certain distance.	**IF_FAR** action.
■ Actor is near an object that triggers an **EXPLODE** instruction.	**IF_HIT** action with **FRAGILE** flag set.
■ Player hits actor with the **SHOOT** instruction.	**IF_HIT** action.
■ The animation cycle of the actor's texture has passed.	**EACH_CYCLE** action.
■ A skill, that is being changed by another action, has reached a certain value.	**EACH_TICK** action that permanently compares this skill to its limit.
■ A certain amount of time has passed.	**EACH_TICK** action with countdown of a skill.
■ Actor has reached a certain waypoint.	**IF_ARRIVED** action with comparing of waypoint numbers.
■ Actor has reached a certain region.	**IF_ARRIVED** action with comparing a region parameter.
■ Actor has collided with an obstacle.	**IF_HIT** action with **TARGET BULLET**.
■ Random event.	**EACH_TICK** action that periodically compares the skill **RANDOM** to a numeric value.

Each of these events may of course be combined with any other. A WDL action is responsible for the transition between states which will be put into effect through a number of SET instructions.

Apart from states we are also able to define inner variables for the machine that may acquire different values and influence the behaviour. Our actor skills are equal to such variables. Their values can be made use of in an action and for example decide on the transition into another state. The theory of state machine does not include any such skills, but each of these skills could theoretically be replaced by a finite number of sub-states that equal certain values of the skill.

The Table of States

The division of the behaviour of our machine into states and transitions allows us to keep the actor's WDL somewhat transparent. To give an example we shall now define our first state machine actor. For the purpose of this we are going to make use of the Tron virus from the computer game SKAPHANDER. But in case you wish to try your hand at the example, be aware that you will have to design Tron's textures yourself as well as sample the sounds; they are not included.

Before writing the first line of WDL code we are going to create a theoretical concept. What kind of states is our Tron to have, how are these states going to be distinguished from each other, what skills, and what transitions do we need?

We define: At the start of the game Tron is to be in the WAIT state. This means that he will be lurking in some dark corner waiting for the player. If the player comes near he will change into the ATTACK state. If the player hits Tron with the SHOOT instruction, the following transition depends on the number of hits the Tron had already been forced to take: the next state will either be ATTACK, ESCAPE, or DIE.

To keep track of the number of hits we shall employ the Tron's **SKILL1**. For better transparency we sum up the states in a *table of states*.

State	**Texture**	**Target**	**Player near**	**Hits so far <=4**	**Hits so far <=5**	**Hits so far >5**	**Animation finished**
WAIT	tron_tex	NULL	ATTACK	ATTACK	ESCAPE	DIE	-

We have defined five events for the Tron that may change his state. Three of those are triggered by a direct hit, each time depending on the number of previous hits. Apart from that the texture and the respective **TARGET** for each state are given. We may now add the remaining states to the table. Within the state of ESCAPE a further hit is to make the Tron attack again:

State	**Texture**	**Target**	**Player near**	**Hits so far <=4**	**Hits so far <=5**	**Hits so far >5**	**Animation finished**
WAIT	tron_tex	NULL	ATTACK	ATTACK	ESCAPE	DIE	-
ATTACK	tron_atak	FOLLOW	-	ATTACK	ESCAPE	DIE	-
ESCAPE	tron_hovo	REPEL	ATTACK	ATTACK	ATTACK	DIE	-
DIE	tron_explo	NULL	-	-	-	-	DEAD
DEAD	tron_rest	NULL	-	-	-	-	-

Empty fields (**-**) in the chart do not change the actor's state. Our actor is a simple machine: if a more complex behaviour were required our table would contain a lot more columns and lines. At closer inspection you might notice that a few unnecessary transitions were defined in the table. For example the transition between WAIT and DEAD will never be made because the Tron immediately transitions to ATTACK at the first hit. But because with more complex state machines special cases like this one are not always so easy to see any more it is always sensible to define the table as completely as possible.

Once the table of states is finished the WDL programming starts. To begin with we will define the sounds, bitmaps and textures for the different states of the Tron as well as the actor itself:

```
//////////////////////////////////////////////////
```

```
// Tron actor

////////////////////////////////////////////////////

SOUND tron_snd,<tron_vir.wav>;

SOUND tron_baller,<schuss1.wav>;

SOUND tron_aua,<tron_au.wav>;

SOUND tron_spot,<tr_tac.wav>;

////////////////////////////////////////////////////

BMAP	tron_0vh1,<v3_vhs.lbm>,20,12,92,84;

BMAP	tron_0vh2,<v3_vhs.lbm>,180,12,92,84;

BMAP	tron_0vh3,<v3_vhs.lbm>,340,12,92,84;

BMAP	tron_0vh4,<v3_vhs.lbm>,20,117,92,84;

BMAP	tron_0vh5,<v3_vhs.lbm>,180,117,92,84;

BMAP	tron_0vh6,<v3_vhs.lbm>,340,117,92,84;

BMAP	tron_0hre1,<v3_re_li.lbm>,20,12,92,84;

BMAP	tron_0hre2,<v3_re_li.lbm>,180,12,92,84;

BMAP	tron_0hre3,<v3_re_li.lbm>,340,12,92,84;

BMAP	tron_0hre4,<v3_re_li.lbm>,20,117,92,84;

BMAP	tron_0hre5,<v3_re_li.lbm>,180,117,92,84;

BMAP	tron_0hre6,<v3_re_li.lbm>,340,117,92,84;

BMAP	tron_0s1,<v3_vhs.lbm>,20,222,92,84;

BMAP	tron_0s2,<v3_vhs.lbm>,180,222,92,84;

BMAP	tron_0s3,<v3_vhs.lbm>,340,222,92,84;

BMAP	tron_0s4,<v3_vhs.lbm>,20,327,92,84;

BMAP	tron_0s5,<v3_vhs.lbm>,180,327,92,84;

BMAP	tron_0s6,<v3_vhs.lbm>,340,327,92,84;

BMAP	tron_0hli1,<v3_re_li.lbm>,20,222,92,84;

BMAP	tron_0hli2,<v3_re_li.lbm>,180,222,92,84;

BMAP	tron_0hli3,<v3_re_li.lbm>,340,222,92,84;

BMAP	tron_0hli4,<v3_re_li.lbm>,20,327,92,84;

BMAP	tron_0hli5,<v3_re_li.lbm>,180,327,92,84;

BMAP	tron_0hli6,<v3_re_li.lbm>,340,327,92,84;

BMAP	tron_an2,<v3_an.pcx>,180,12,92,84;
```

```
BMAP       tron_an3,<v3_an.pcx>,340,12,92,84;
BMAP       tron_an4,<v3_an.pcx>,20,117,92,84;
BMAP       tron_an5,<v3_an.pcx>,180,117,92,84;
BMAP       tron_an6,<v3_an.pcx>,340,117,92,84;
BMAP       tron_ex1,<v3_ex.lbm>,160,0,160,105;
BMAP       tron_ex2,<v3_ex.lbm>,320,0,160,105;
BMAP       tron_ex3,<v3_ex.lbm>,0,105,160,105;
BMAP       tron_ex4,<v3_ex.lbm>,160,105,160,105;
BMAP       tron_ex5,<v3_ex.lbm>,320,105,160,105;
BMAP       tron_ex6,<v3_ex.lbm>,0,210,160,105;
BMAP       tron_ex7,<v3_ex.lbm>,160,210,160,105;
BMAP       tron_ex8,<v3_ex.lbm>,320,210,160,105;
BMAP       tron_ex9,<v3_ex.lbm>,0,315,160,105;
BMAP       tron_ex10,<v3_ex.lbm>,160,315,160,105;
BMAP       tron_ex11,<v3_ex.lbm>,320,315,160,105;
BMAP       tron_corpse,<knallig.lbm>,437,256,47,21;
////////////////////////////////////////////////////////////
TEXTURE tron_tex {
SCALE_X 16;
SCALE_Y 16;
           SIDES              8;
CYCLES  6;
        BMAPS    tron_0vh1,tron_0vh2,tron_0vh3,tron_0vh4,tron_0vh5,tron_0vh6,
tron_0hre1,tron_0hre2,tron_0hre3,tron_0hre4,tron_0hre5,tron_0hre6,
tron_0s1,tron_0s2,tron_0s3,tron_0s4,tron_0s5,tron_0s6,
tron_0hre1,tron_0hre2,tron_0hre3,tron_0hre4,tron_0hre5,tron_0hre6,
tron_0vh1,tron_0vh2,tron_0vh3,tron_0vh4,tron_0vh5,tron_0vh6,
tron_0hli1,tron_0hli2,tron_0hli3,tron_0hli4,tron_0hli5,tron_0hli6,
tron_0s1,tron_0s2,tron_0s3,tron_0s4,tron_0s5,tron_0s6,
tron_0hli1,tron_0hli2,tron_0hli3,tron_0hli4,tron_0hli5,tron_0hli6;
           DELAY              3,3,2,2,3,3;
MIRROR  0,0,0,1,1,1,1,0;
```

```
AMBIENT 0.1;
            SOUND            tron_snd;
SCYCLE   1;
            SVOL             0.2;
            SDIST            40;
}
TEXTURE tron_hovo {
SCALE_X 16;
SCALE_Y 16;
            SIDES            8;
CYCLES   6;
    BMAPS       tron_0vh1,tron_0vh2,tron_0vh3,tron_0vh4,tron_0vh5,tron_0vh6,
tron_0hre1,tron_0hre2,tron_0hre3,tron_0hre4,tron_0hre5,tron_0hre6,
tron_0s1,tron_0s2,tron_0s3,tron_0s4,tron_0s5,tron_0s6,
tron_0hre1,tron_0hre2,tron_0hre3,tron_0hre4,tron_0hre5,tron_0hre6,
tron_0vh1,tron_0vh2,tron_0vh3,tron_0vh4,tron_0vh5,tron_0vh6,
tron_0hli1,tron_0hli2,tron_0hli3,tron_0hli4,tron_0hli5,tron_0hli6,
tron_0s1,tron_0s2,tron_0s3,tron_0s4,tron_0s5,tron_0s6,
tron_0hli1,tron_0hli2,tron_0hli3,tron_0hli4,tron_0hli5,tron_0hli6;
            DELAY            5,5,3,3,5,5;
RANDOM  0.3;
MIRROR   0,0,0,1,1,1,1,0;
AMBIENT  -0.2;
            SOUND            tron_snd;
SCYCLE   1;
            SVOL             0.2;
            SDIST            40;
}
TEXTURE tron_atak {
SCALE_XY        16,16;
CYCLES   7;
        BMAPS   tron_0vh1,tron_an2,tron_an3,tron_an4,tron_an5,tron_an6,tron_an3;
```

```
            DELAY              1,1,1,1,1,1,1;
AMBIENT 0.3;
            SOUND              tron_baller;
SCYCLE  4;
            SVOL               0.25;
            SDIST              200;
}
TEXTURE tron_explo {
SCALE_XY        18,18;
CYCLES  11;
BMAPS   tron_ex1,tron_ex2,tron_ex3,tron_ex4,tron_ex5,tron_ex6,
tron_ex7,tron_ex8,tron_ex9,tron_ex10,tron_ex11;
            DELAY              1,2,2,2,2,2,2,3,3,2,2;
AMBIENT 1;
            FLAGS              ONESHOT;
            SOUND              tron_bang;
SCYCLE  2;
            SVOL               0.5;
            SDIST              300;
}
TEXTURE tron_rest {
SCALE_XY        16,16;
            BMAPS              tron_corpse;
}
//////////////////////////////////////////////////
ACTOR   simple_tron {
TEXTURE tron_tex;
SKILL1  0;                              // initial number of hits
HEIGHT  2;                              // height hovering above ground
EACH_TICK tron_warten;      // initial state
}
```

Some of the textures are multi-sided. Because of the actors symmetrical character **MIRROR** flags may be employed to save on bitmaps. As the Tron will always run towards the player when attacking a single sided texture is sufficient; the same is true for the exploding Tron. The escape texture was darkened a bit by a negative ambient to indicate damage or energy loss.

The definition of the actor can be kept sparse because all necessary parameter are set in the action **tron_warten**. This action was defined as an **EACH_TICK** action. As a consequence of this trick the state WAIT will be activated immediately after the start of the game. Still we will have to make sure that the actions resets the **EACH_TICK** event to **NULL** on it's own, or it would repeat itself continually:

```
//////////////////////////////////////////////////

ACTION tron_warten {                                                    // WAIT state

SET             MY.TEXTURE,tron_tex;

SET             MY.TARGET,NULL;

SET             MY.DIST,60;                                      // effective range

SET             MY.IF_NEAR,tron_angriff;

SET             MY.IF_HIT,tron_anfluster;

SET             MY.EACH_TICK,NULL;

SET             MY.EACH_CYCLE,NULL;

}

ACTION tron_anfluster {                                  // transition to ATTACK, ESCAPE, or DIE

ADD             MY.SKILL1,1;                             // increase number of hits

PLAY_SOUND      tron_aua;                  // wail

IF_ABOVE        MY.SKILL1,5;

BRANCH  tron_sterben;

IF_ABOVE        MY.SKILL1,4;

BRANCH  tron_flucht;

BRANCH          tron_angriff;

}
```

Because all actions have to be triggered by the actor itself, we may use the **MY** synonym to indicate the actor. The action **tron_anfluster** is necessary to trigger either ATTACK, ESCAPE, or DIE, depending on the number of previous hits. Apart from that it is supposed to play a sound as a response to a hit.

```
ACTION tron_angriff {                      // ATTACK state

SET             MY.TEXTURE,tron_atak;

SET             MY.SPEED,0.4;

SET             MY.VSPEED,0.2;

SET             MY.CAREFULLY,1;            // avoid obstacles

SET             MY.TARGET,FOLLOW;
SET          MY.IF_NEAR,NULL;
SET          MY.IF_HIT,tron_anfluster;
SET          MY.EACH_TICK,NULL;
SET          MY.EACH_CYCLE,tron_schuss;
}
ACTION tron_schuss {                          // fire
SET          SHOOT_X,0;
SET          SHOOT_Y,-0.5;          // Tron shoots with his legs
SET          SHOOT_RANGE,200;
SHOOT        MY;
IF_EQUALHIT_DIST,0;          // no hit?
END;
ADD          health,-1;          // Player takes hit
}
```

With the attack action we have to set the horizontal as well as the vertical speed - plus the **CAREFULLY** flag, if necessary - with the target. Vertical speed **VSPEED** combined with the target **FOLLOW** causes the Tron to hover towards the player if the same is situated on a ledge above him.

The attack texture was drawn in such a way that the Tron will shoot once per cycle of animation. That is why we need the **EACH_CYCLE** action **tron_schuss** to find out whether the player was actually hit or not. The **EACH_CYCLE** event in this case does not trigger a new state.

The **SHOOT** instruction checks whether a horizontal line may be drawn between the actor and the player that does not cross a wall. If that is the case - and if the player is within the range of **SHOOT_RANGE** - the pre-defined skill **HIT_DIST** will be set according to the distance of the actor. Otherwise, **HIT_DIST** will be 0.

Through **SHOOT_Y** we shift the location of the shot slightly downwards, because the Tron is shooting with his 'legs'. If we didn't do that the player would be hit even if the Tron's energy guns were behind a wall. To make the game appear realistic we should always be aware of finer points like this one.

```
ACTION tron_flucht {                          // ESCAPE state
SET          MY.TEXTURE,tron_hovo;
SET          MY.SPEED,0.4;
SET          MY.VSPEED,0;
SET          MY.CAREFULLY,1;
SET          MY.TARGET,REPEL;
SET          MY.IF_NEAR,NULL;
SET          MY.IF_HIT,tron_angster;
SET          MY.EACH_TICK,NULL;
SET         MY.EACH_CYCLE,NULL;
}
ACTION tron_angster {                   // transition to ATTACK or DIE
ADD         MY.SKILL1,1;                // increase number of hits
PLAY_SOUND  tron_aua;           // wail
IF_ABOVE    MY.SKILL1,5;
BRANCH  tron_sterben;
BRANCH      tron_angriff;
}
ACTION tron_sterben {                   // DIE state
SET         MY.TEXTURE,tron_explo;
SET         MY.SPEED,0.2;
SET         MY.VSPEED,0;
SET         MY.CAREFULLY,0;
SET         MY.PASSABLE,1;
SET         MY.TARGET,FOLLOW;
SET         MY.IF_NEAR,NULL;
SET         MY.IF_HIT,NULL;
SET         MY.EACH_TICK,NULL;
SET         MY.EACH_CYCLE,tron_tod;
}
ACTION tron_tod {                       // DEAD state
SET         MY.TEXTURE,tron_rest;
SET         MY.PASSABLE,1;
SET         MY.TARGET,NULL;
SET         MY.IF_NEAR,NULL;
SET         MY.IF_HIT,NULL;
SET         MY.EACH_TICK,NULL;
SET         MY.EACH_CYCLE,NULL;
}
```

The action **tron_sterben** again contains something remarkable that seemingly contradicts our table of states. Even though the Tron has been finished off he again moves towards the player with target **FOLLOW**! By this trick we get the explosion's exhaust to come nearer and expand in addition to its own animation, which makes the explosion look even more realistic. Of course the exhaust is now no longer supposed to avoid obstacles, therefor **CAREFULLY** was set to 0. Additionally with the exploding as well as with the dead Tron **PASSABLE** must be set for it not present an obstacle to the player.

Advanced State Machines

Intelligent and 'personalised' behaviour of opponents gives greater satisfaction at playing especially with role games or action games. Once you have been playing a game for a certain time you will start to notice certain *characteristics* of your opponents - weaknesses or idiosyncrasies - that you may make use of. You are able to develop a *strategy*. If such characteristics are lacking, so is a strategy: the game will be a mere Shoot'em-Up game. Such games quickly become boring.

How can we improve our actor **simple_tron**? Most obvious are the possibilities of cosmetic amelioration. Simple state enlargements give the actor a more complex, more 'personal' repertoire of behaviour. He might give a cry of anger when he sees the player; he might rebound when hit, or start spinning. He doesn't advance towards the player in a straight line but dodges sideways in order to avoid a direct hit. Some of these possibilities have been realised in the ENEMY.WDL file that comes with this tutorial; when they are realised, the number of states and transitions increases significantly.

Improvements of the actor's *strategic* behaviour are a lot more complicated to realise. They clearly move into the direction of developing a higher degree of artificial intelligence and can only be realised by letting the actors communicate with one another (through **EXPLODE** instructions and skills) and/or by making them to behave in regard to the player's position relative to the level's topography. Some more suggestions for that as well:

- As soon as an actor ('guardian') views the player he alarms a fellow-actor within a certain distance. Suited to raise an alarm are objects like sensors, cameras, trip-wires, or light barriers. It is important for building suspense that the player is aware of the alarm!

- If several actors attack the player, they try to surround and encircle him.

- Actors lie in wait for the player, lurking around corners, by advancing towards the player just to the point where he can't yet see them.

- There may be canyons and rooms within the level, that function as 'traps': here, actors can attack the player from all sides. The actors try to tease the player or push him in such cases.

- Actors can deliver a prognosis on the outcome of the fight by comparing the player's strength to their own. If the player is still quite strong, they call re-enforcement, if not, they attack directly.

Some of these suggestions have been realised in some levels of the commercial game SKAPHANDER PLUS. But we never ought to overdo this. The player should always be aware that there is danger ahead. Actors that completely hide and lie in ambush for the player for quite a while in order to suddenly get at him from all sides to finish him off aren't much fun to have in your game.

Special Machines

In order to give games a higher value, objects can be placed within the levels, that react in a special way to the player, or when hit or worked. Again, two objects from the SKAPHANDER PLUS game will serve as examples on how to realise 'special effects' in a WDL, namely the *barrel* and the *mine*.

The barrel is meant to jump up when hit; and when it hits the ground there is to be an animated texture. This behaviour allows the player to let barrels dance in the air - as a way to practice his shooting skills, kind of.

```
BMAP fass1,<tonne.lbm>,23,254,81,174;

BMAP fass2,<tonne.lbm>,117,254,81,174;
BMAP fass3,<tonne.lbm>,236,254,81,174;

BMAP fass4,<tonne.lbm>,340,254,81,174;

BMAP fass5,<tonne.lbm>,440,254,81,174;

TEXTURE fass_tex {

SCALE_XY        35,35;

CYCLES   5;

BMAPS           fass1,fass2,fass3,fass4,fass5;

FLAGS           ONESHOT;

DELAY           3,3,3,3,3;

}

THING    fass {

TEXTURE fass_tex;

HEIGHT   0;

FLAGS           FRAGILE;                // barrel reacts to EXPLODE as well

IF_HIT   fass_hit;

}

SKILL hupf_abweich  { VAL 0; }

ACTION   fass_hit {

PLAY_SOUND      banngg,0.2;

SET             MY.SKILL1,1;                            // initial speed

ADD             MY.SKILL1,hupf_abweich;   // individual speed difference

ADD             hupf_abweich,0.05;

IF_ABOVE hupf_abweich,0.25;

SET      hupf_abweich,-0.15;

SET             MY.EACH_TICK,fass_hupf;

}
```

In the action **fass_hit** the parameter **SKILL1** is set corresponding to the speed the barrel is to move upwards with. The skill **hupf_abweich** is then added to this initial speed. The skill will subsequently be altered slightly. By this we achieve that every barrel starts jumping with a slightly different speed; whole rows of barrels that are made to jump by an **EXPLODE** instruction look better that way.

Finally, an **EACH_TICK** action is started, that is responsible for further upward movement:

```
SKILL decken_distanz        { VAL 0; }

ACTION fass_hupf {
```

```
SET             RENDER_MODE,1;   // make barrel movement visible

SET             decken_distanz,THERE.CEIL_HGT;        // calculate distance

SUB             decken_distanz,THERE.FLOOR_HGT;       // from ceiling

SUB             decken_distanz,MY.SIZE_Y;

ADD             MY.HEIGHT,MY.SKILL1;                          // move barrel vertically

IF_ABOVE        MY.HEIGHT,decken_distanz;    // does it touch the ceiling?

SET     MY.HEIGHT,decken_distanz;    // don't go further up

ADDT            MY.SKILL1,-0.15;                              // gravitation

IF_ABOVE        MY.HEIGHT,0;                                          // barrel still airborne?

END;

SET             MY.HEIGHT,0;

PLAY_SOUND      rrumms,0.3;                                   // impact on ground

SET             MY.EACH_TICK,NULL;

SET             MY.PLAY, 1;                                           // start texture animation

}
```

At first the height of the barrel when touching the ceiling is computed. Its vertical speed (**SKILL1**) is added to the current **HEIGHT** of the object and finally it is tested whether the ceiling was touched; if that is so, the barrel is set directly below the ceiling.

Now gravity exerts its pull: it makes sure that the vertical speed of an object is reduced at a certain rate per time unit. The **ADDT** instruction has to be used here in order to compensate for different computer speeds. The constant reduction of the vertical speed causes it to become negative at some point, so that the barrel falls back towards the floor. Once the floor has been reached, a sound and an animation are played and the **EACH_TICK** action is switched off. Because **HEIGHT** might have acquired an negative value during the constant reduction this parameter must explicitly be set to 0.

With the barrel we have become acquainted with a very simple state machine with only two states **-** one on ground and one in the air. With the mine, things become a bit more complicated. It can be picked up, carried around, thrown, and made to explode:

```
////////////////////////////////////////////////

// mine for plant-and-run use

////////////////////////////////////////////////

SOUND     klick,<ausklink.wav>;

SOUND     klonk,<aufsetz.wav>;

SOUND     tic_tac,<tic.wav>;

////////////////////////////////////////////////

BMAP min_ob,<mine2.lbm>,0,0,78,80;

BMAP min_ex1,<knallig.lbm>,0,42,128,128;
BMAP min_ex2,<knallig.lbm>,128,42,128,128;

BMAP min_ex3,<knallig.lbm>,256,42,128,128;

BMAP min_ex4,<knallig.lbm>,384,42,128,128;

BMAP min_ex5,<knallig.lbm>,0,170,128,128;

BMAP min_ex6,<knallig.lbm>,128,170,128,128;

TEXTURE mine_tex {

SCALE_XY            32,32;

BMAPS               min_ob;

}

TEXTURE mine_tick_tex {

SCALE_XY            32,32;

CYCLES  2;

BMAPS               min_ob,min_ob;

DELAY               4,4;

SOUND               tic_tac;

SCYCLE              1;

SVOL                0.4;

SDIST               60;

}

TEXTURE mine_explo_tex {

SCALE_XY            10,10;

CYCLES  6;

BMAPS               min_ex1,min_ex2,min_ex3,min_ex4,min_ex5,min_ex6;

DELAY               2,1,1,1,3,2;

SCYCLES 1,0,1,0,0,0;

AMBIENT 1;

FLAGS               ONESHOT;

SOUND               tron_bang;

SVOL                0.6;

SDIST               350;

}

OVLY min_ovl,<mine2.lbm>,119,0,40,31;
```

```
OVERLAY mine_ovr {

POS_X     138;

POS_Y     293;

OVLYS     min_ovl;

}

/////////////////////////////////////////

ACTOR mine {

TEXTURE mine_tex;

FLAGS               FRAGILE;

IF_NEAR   nimm_mine;

}
```

It should be remarked that with the animated texture **mine_tick_tex** one changes between two identical bitmaps. The animation here is only of importance to the acoustics, it is supposed to produce a rhythmic ticking.

```
ACTION nimm_mine {                  // take mine from ground

PLAY_SOUND          beamer,0.3;

SET                 MY.INVISIBLE,1;

SET                 LAYERS.1,mine1_ovr;

SET                 IF_CTRL,mine_werf;

SET                 IF_LEFT,mine_werf;

}
```

A classic case of transition between states, often used in games: when the mine has been touched, it disappears from the surroundings and reappears as an overlay (what's this? More about overlays in the following chapter) on the screen. By hitting either the left mouse button or the [Ctrl] key the mine can now be thrown.

```
ACTION mine_werf {                          // throw mine

SET                 IF_CTRL,NULL;

SET                 IF_LEFT,NULL;

SET                 LAYERS.1,NULL;      // let overlay disappear

PLAY_SOUND          klick,0.5;

SET                 mine.HEIGHT,1.5;

SET                 mine.DIST,0;

          DROP                mine;                         // put on player's position

SET                 mine.TEXTURE,mine_tex;

SET                 mine.INVISIBLE,0;    // visible again
SET          mine.ANGLE,PLAYER_ANGLE;
SET          mine.SPEED,2;
SET          mine.VSPEED,0.4;          // vertical speed
SET          mine.CAREFULLY,1;
SET          mine.TARGET,MOVE;         // set in motion
SET          mine.EACH_TICK,wurf_flug;
SET          mine.EACH_CYCLE,NULL;
SET          mine.IF_NEAR,NULL;
}
ACTION wurf_flug {                     // next state: flying
ADDT         MY.VSPEED,-0.15;
IF_ABOVE     MY.HEIGHT,0;
END;
PLAY_SOUND   klonk,0.5;  // hits the ground
SET          MY.HEIGHT,0;
SET          MY.VSPEED,0;
SET          MY.EACH_TICK,wurf_rutsch;
}
ACTION wurf_rutsch {                            // next state: slide on ground
ADDT         MY.SPEED,-0.3;
IF_ABOVE     MY.SPEED,0;
END;
SET          MY.TEXTURE,mine_tick_tex;
SET          MY.EACH_TICK,NULL;
SET          MY.EACH_CYCLE,mine_scharf;
SET          MY.TARGET,NULL;
SET          MY.IF_NEAR,nimm_mine;
}
```

In order to be able to throw the mine, we have to put it on the player's position by a **DROP** instruction first, because the invisible mine-thing prior to that is still situated at the position where we picked it up. The act of throwing the mine itself is split up into a flight-phase and a slide-phase. The movement during the flight resembles that of our barrel; indeed, because the mine is an actor, we may now replace the object-skill directly by the parameter **VSPEED**. Additionally there is a horizontal movement (**TARGET MOVE**). If we would watch the trajectory from the side, we would see a classic parabola, the token of trajectories within gravitational fields. **CAREFULLY** has to be set so that the mine is able to rebound from walls and things during its flight.

After it has hit the ground the mine slides on for a bit and then lies still. The mine now is armed: it is ticking and waiting for an opponent **-** an actor with his **FRAGILE** flag set **-** to turn up close to it. Yet the player may still take up the mine and place it somewhere else within the level at any time.

```
ACTION mine_scharf {                          // next state: armed

SET             SHOOT_FAC,0;

SET             SHOOT_RANGE,15;

EXPLODE MY;                           // test explosion: enemy within 15 steps?

IF_EQUALHIT_MINDIST,0;    // nothing hit?

END;

BRANCH  mine_explode;

}

ACTION mine_explode {                 // next state: explode (really!)

SET             MY.PASSABLE,1;

SET             MY.CAREFULLY,0;   // important for THERE!!

SET             MY.SPEED,0.3;

SET             MY.TARGET,FOLLOW;

SET             SHOOT_RANGE,40;

SET             SHOOT_FAC,1;

        EXPLODE MY;                           // now 'real' explosion

SET             MY.TEXTURE,min_bums_tex;

ADD             MY.HEIGHT,-2;         // new texture is larger

ADD             THERE.AMBIENT,0.2;    // illuminate region

SET             MY.EACH_CYCLE,mine_weg;

SET             MY.IF_HIT,NULL;

SET             MY.IF_NEAR,NULL;

SET             MY.EACH_TICK,NULL;

IF_ABOVE        RESULT,0;                     // was player hit as well?

RULE    health = health - 200*RESULT;   // deduct health

}

ACTION mine_weg {                     // next state: disappear

ADD             THERE.AMBIENT,-0.2;   // region dark again

SET             MY.INVISIBLE,1;       // mine is gone

}
```

At every animation cycle the mine checks via **EXPLODE** instruction, whether a victim is near. To do that it sets the skill **SHOOT_FAC** to 0, so that no-one gets hurt. This is a further example for abusing the **EXPLODE** instruction for testing purposes. As this instruction takes comparatively much time to compute it is advisable to let such cyclical test-explosions happen at larger intervals. The **EACH_CYCLE** event is suited for this.

We will see a real explosion if something was hit at the test-explosion. In this case the **SHOOT_RANGE** was selected to be distinctly larger, coming it at 40 steps, so that the actor triggering the mine might possibly also doom a few fellow fighters. The actors that were hit are responsible for damaging themselves through their **IF_HIT** actions. The player may also have to take some of it if he was dumb enough to be near the armed mine. This will be determined using the **RESULT** skill.

From our simple_tron we are already acquainted with the trick to use **FOLLOW** to cause the explosion to expand. After the explosion the mine disappears. One more small trap hides in changing the **AMBIENT** of the region to display the flash caused by the explosion. If we access a region in this way, using the keyword **THERE** for the region the action-triggerin object is in, we ought to make sure that the region that was lit up is the same region that is darkened afterwards! Therefor the actor's **CAREFULLY** flag must be set to 0 in advance, so that the **THERE** synonym does not change if the exhaust from the explosion crosses the border of a region.

Friday: Panels, Menus, Overlays

Up to now all we were exclusively concerned about constructing a virtual world and its inhabitants. In this chapter we are going to talk about such functions that, while lying outside the world, still are part of the game. These are, first and foremost, those functions that allow to exert control over the game and those that make communication possible between the *user*, sitting comfortably in front of his computer screen, and his imaginary counterpart, the *player*, who is the one who has to run through subterranean labyrinths and do the wet work.

Texts

If the player has something to tell to the user, this can be done in simple written text. Texts are used for reporting on-screen, but may also be used for a dialogue between the player and actors in an adventure game. As with bitmaps, textures, and objects there is a 'hierarchy' of texts, ranging from the unrefined raw text to the jazzed-up on-screen report.

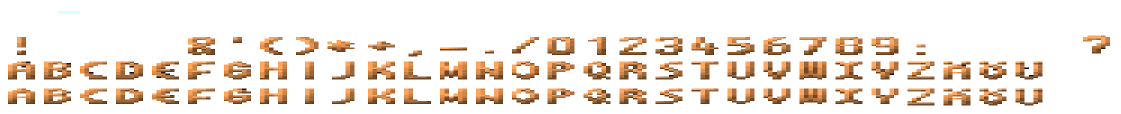
Unrefined text is defined as a sequence of letters with the keyword **STRING**:

```
STRING    my_string, "This is a text!!";
```

The string is positioned between quotation marks. It may consist of several lines, either corresponding to several lines in the WDL file or being separated from each other by the "**\n**" character string.

```
STRING    my_string_3,

"This is the first line,

this is the second,\nand this, finally, is the third line."
```

The "**\n**" between the second and third line would of course not appear on the screen. In order for the text to become visible on the screen in the first place we have to define a *character set* (font) first. A font is simply made up of little images for each letter. The images appear in a certain order within a bitmap. This order corresponds to the first 128 characters



of the PC character set (ASCII grouping).

All in all the bitmap must leave room for 128 letters, numbers and symbols, each character taking up exactly the same amount of space. The first row remains empty, the third row contains the capital letters, the fourth contains the small letters. The room between the characters can be set to the transparent colour 0. It goes without saying that every imaginable set of characters - e.g. Greek symbols or those stemming from your own imagination - may be represented. In our example the German 'umlauts' take the place of the special characters '[', '|' ,']' und '{', '\', '}'. On the demo/tutorial disk a simple bitmap (PANFONT.PCX) was added that may serve as an example for the grouping of your own set of characters.

The keyword **FONT** is used to define the set of characters as a such:

```
FONT    standard_font,<panfont.pcx>,8,10;
```

The final two parameter give the breadth and height in pixels of a single character within the bitmap set-of-characters (including the gaps). So the bitmap has to include exactly 8*10*128 = 10240 pixels. Once we have the **FONT** available we can make a **TEXT** out of our **STRING**:

```
TEXT my_text {

POS_X   20;

POS_Y   40;

FONT            standard_font;

STRING  my_string;

}
```

POS_X and **POS_Y** return the position of the text on screen in pixels, in reference to the left upper corner. But we will have to make it visible first, because the text defined above still won't show up on the screen. The following action makes our text appear on the screen:

```
ACTION show_my_text { SET my_text.VISIBLE,1; }
```

This method allows us to show several texts on-screen simultaneously. As it is possible for us to change the texts position **POS_X** and **POS_Y** during the game, the text can also be made to roll across the screen vertically:

```
ACTION roll_my_text {

SET     my_text.POS_Y, 400;         // Set text below screen

SET     MESSAGES.10, my_text;       // make visible

roll_1:                             // action is repeated from this line on

WAITT   1;                                                  // sleep for 1/16 second

ADD     my_text.POS_Y,-1;           // roll upwards for one line of pixels

IF_ABOVE        my_text.POS_Y,0;            // is the text still on-screen?

GOTO    roll_1;                             // then do it again

SET     MESSAGES.10,NULL;                   // else shut off text

}

IF_R    roll_my_text;
```

The action is initiated by pushing the **[R]** key. We have built a 'programming trick' into this yet again, the **GOTO** jumping backwards, which causes the action to 'go to sleep' each time at the **WAITT** instruction after the jump label **roll_1**. The effect of this being that not the complete action is repeated at each tick, but only a small portion of it - namely that between the label and **GOTO**. Programmers call that a 'loop'. Only if the value of **POS_Y** after much adding of -1 has finally reached 0, i.e. if it touches the upper end of the screen, the **GOTO** instruction won't be executed any more. The action then terminates.

Panels

Many games use displays showing numbers or a bar to give information on the player's state or certain variables of the game. One classical example would be the role playing game where health, combat strength, and dozens of other player-characteristics constantly change and have influence on the game.

Such displays are represented by a **PANEL** on screen. To give an example for a panel we shall now represent the values of certain skills on screen in form of numbers. It is very helpful to have those skills that play a role with the discharge of an action ever present before you eyes while testing - *debugging* - an action you have written yourself.

Panels are defined similarly to texts. Instead of a string this time there are definitions for parts of the display like **DIGITS**, **HBAR**, or **VBAR**:

```
FONT standard_font,<panfont.pcx>,8,10;

PANEL debug_panel {

POS_X     0;

POS_Y     12;

FLAGS     REFRESH;

DIGITS    18,2,3,standard_font,160,TIME_FAC;         // Frames per second

DIGITS    50,2,2,standard_font,1,ACTIVE_NEXUS;

DIGITS    80,2,2,standard_font,1,ACTIVE_OBJTICKS;

DIGITS    110,2,2,standard_font,1,ERROR;

DIGITS    150,2,4,standard_font,10,PLAYER_SIZE;

DIGITS    200,2,4,standard_font,10,PLAYER_ANGLE;

DIGITS    240,2,4,standard_font,1,PLAYER_X;

DIGITS    280,2,4,standard_font,1,PLAYER_Y;

}

STRING debug_labels,

" fps nex obj err siz ang x y";

TEXT debug_text {

POS_X     0;

POS_Y     2;

FONT      standard_font;

STRING    debug_labels;

}
```

```
ACTION panel_init {
SET	panels.10,debug_panel;
SET	messages.10,debug_text;
}
```

After the action **panel_init** was executed we now have 8 numeric displays at the upper end of the screen that inform us about the game's or player's state. Each **DIGITS** line within the **PANEL** definition shows one skill in form of a numeric value. It contains 8 parameters, that determine the X- and Y-position of the display, the number of digits, the set of characters, a factor for multiplication, and the skill itself. The values of position do not refer to the screen, but to the distance in pixels of the upper left corner of the display from the upper left corner of the panel. The flag **REFRESH** causes the panel to be rebuilt constantly. This is the only way we can have it displayed above the 3-D window, which as well is generated anew with every frame. With texts we don't have to mention the **REFRESH** flag because there it is automatically set.

In the panel we can now see - from left to right - the current frame rate in frames-per-second (multiplied by 10 in order to make the post-comma digits visible), the efficiency of the nexus (if the value here reaches 14 we have to increase it with the keyword **NEXUS**), the number of object's **EACH_TICK** actions currently in progress simultaneously, the error display (should always return 0), and four more skills giving the player's position. If we would like to follow the state of other skills during the game we may display them here instead.

There is one tiny thing amiss with the display of the frame rate we used the skill **TIME_FAC** for. This skill returns the value 1, if the frame rate is at exactly 16 images per second, and changes proportionally. That means that the display changes rather quickly and is hard to read. A little trick will make it slow down a bit:

```
SKILL fps  {}
ACTION show_panels {
RULE	fps = 0.8*fps + 0.2*TIME_FAC;
}
PANEL debug_panel {
POS_X	0;
POS_Y	12;
FLAGS	REFRESH;
DIGITS	18,2,3,standard_font,160,fps;	// frames per second
...
}
ACTION panel_init {
SET	panels.10, debug_panel;
SET	messages.10, debug_text;
SET	EACH_TICK.15, show_panels;
}
```

Our new **EACH_TICK** action show_panel is responsible for 'processing' the display skills. The **RULE** instruction makes sure that a spontaneous change of **TIME_FAC** only has an impact of 20% on the display, which causes the value to change five times slower.

A further example for displays are horizontal and vertical bars, that shift according to the values of the skills represented. We would like to make use of a horizontal bar-display to represent a compass on screen. We shall start by drawing a bitmap for the compass-bar:

```
BMAP kompass_map,<kompass.pcx>,0,0,160,22;

PANEL kompass_pan {

POS_X               140;

POS_Y               376;

FLAGS               REFRESH;

HBAR                0,0,40,kompass_map,-19.05,PLAYER_ANGLE;

}

ACTION panel_init {

SET                 panels.10, debug_panel;

SET                 panels.11, kompass_pan;

SET                 messages.10, debug_text;

SET                 EACH_TICK.15, show_panels;

}
```

The compass always gives us the player's line of sight, 0° equalling east. The line **HBAR** is structured similarly to **DIGITS**; here again first the position is given, then the breadth of the display in pixels, then the bitmap to be shifted, and finally the factor of multiplication that is to return, multiplied with the skill given at the end, the horizontal shift in pixels.

Our compass bitmap is 160 pixels large horizontally; yet the scale repeats itself after 120 pixels already. With cyclical bar-displays the breadth of the bar-bitmap has to exceed its maximum shift by the breadth of the display - in this case 40 pixels - exactly. The maximum shift results from the highest possible value the displayed skill can acquire, multiplied by the given factor. The value of **-19.05** results from our consideration that the compass-bar was drawn in such a way as to be shifted by exactly 120 pixels with one full rotation of the player - and that to the left, thus in the negative. At a full rotation the skill **PLAYER_ANGLE** acquires the value of **2p** (i.e. araound **6.3**); the desired value thus results from:

-120 = factor ×6.3* Þ *factor = -120 / 6.3 = <u>-19.05</u>

Overlays - Weapons & Tools

The third type of on-screen objects our engine is able to represent is the **OVERLAY**. An overlay might be considered to be a graphic on a transparent foil that is laid over the screen. It is particularly well suited to represent cockpits, swords, reticules, and the likes over the 3-D window.

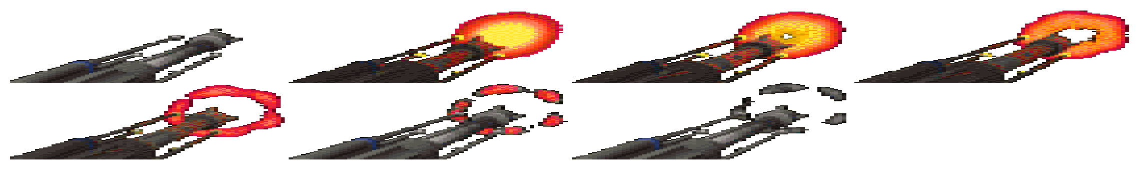
We had already to do with overlays regarding the mine in the last chapter. To give another example for a weapon we have realised using an overlay we may take a look at the laser gun, the 'debugger', from the game SKAPHANDER

PLUS:

```
/////////////////////////////////////////////

// Debugger

/////////////////////////////////////////////

OVLY    cannon1_0 <canon1.LBM>,0,0,68,82;

OVLY    cannon1_1 <canon1.LBM>,68,0,68,80;              // shift -2

OVLY    cannon1_2 <canon1.LBM>,136,0,68,79;    // -3

OVLY    cannon1_3 <canon1.LBM>,204,0,68,80;    // -2

OVLY    cannon1_5 <canon1.LBM>,68,82,68,82;

OVERLAY cannon0 { POS_X 27; POS_Y 250; FLAGS ABSPOS; OVLYS cannon1_0; }

OVERLAY cannon1 { POS_X 25; POS_Y 252; FLAGS ABSPOS; OVLYS cannon1_1; }

OVERLAY cannon2 { POS_X 24; POS_Y 253; FLAGS ABSPOS; OVLYS cannon1_2; }

OVERLAY cannon3 { POS_X 25; POS_Y 252; FLAGS ABSPOS; OVLYS cannon1_3; }

OVERLAY cannon5 { POS_X 27; POS_Y 250; FLAGS ABSPOS; OVLYS cannon1_5; }
```

Quite a lot of overlays for just a single weapon! But we want to represent the debugger complete with recoil and a flash of fire. The keyword **OVLY** has the same syntax as **BMAP**. But still, the graphics with this keyword are stored in a special way: they use up about two- to three-times the storage space of 'conventional' bitmaps, but are being drawn much faster in return. This way we are able to have even large overlays displayed over the screen without causing the frame rate to drop noticeably.

The keyword **OVERLAY** relates to **OVLY** in about the same way as **TEXTURE** does to **BMAP**. A finished overlay contains one or (in later versions) several **OVLYS**, that appear on a list in the **OVERLAY** definition similar to the **BMAPS**. Added to that are, again, the positions **POS_X** and **POS_Y**, that give, just like with **TEXT** and **PANEL**, the distance of the upper left corner of the overlay from the upper left corner of the...no, not the screen but the 3-D window (the window may be smaller than the screen!). That is a further difference between **OVERLAY** and **PANEL**. Only if the flag **ABSPOS** is set do the values refer to distance from the screen's edge and not from the 3-D window.



So now we have five different weapon-overlays (we won't use every one from the bitmap above) that we are going to set in motion with the following action:

```
SOUND twaeng,<laser.wav>;

ACTION fire {

fire_loop:
```

```
PLAY_SOUND          twaeng,0.3,-0.7;
SET                 LAYERS.13,cannon1;
ADD                 PLAYER_LIGHT,0.5; // flash of light
WAIT                1;
SET                 LAYERS.13,cannon2;
ADD                 PLAYER_LIGHT,-0.5;
WAIT                1;
SET                 LAYERS.13,cannon3;
WAIT                1;
SET                 LAYERS.13,cannon5;
WAIT                1;
SET                 LAYERS.13,cannon0;
WAIT                1;
IF_EQUALKEY_CTRL,1;                         // key still down?
GOTO      fire_loop;                // then go on firing
IF_EQUALMOUSE_LEFT,1;
GOTO      fire_loop;
}
ACTION init_fire {
SET                 LAYERS.13,cannon0;
SET                 IF_CTRL,fire;
SET                 IF_LEFT,fire;
}
```

After **init_fire** was executed the gun can be triggered by the **[Ctrl]** key or the left mouse button. Now it becomes obvious why that many overlays were needed: by cyclically changing the overlay the gun is animated. Additionally the positions differ slightly by a few pixels to indicate recoil. The list **LAYERS** has the same meaning for overlays that **PANELS** has for panels and **MESSAGES** has for texts. The **PLAY_SOUND** instruction is used with the **BALANCE** parameter to indicate that the gun is mounted to the left side.

Theoretically we would now be through with overlays if it weren't for the fact that guns often play a big role in 3D games. Therefor we shall improve our gun in such a way that it will not only shoot but also hit. We will be needing a few animations representing flashes occurring with a hit, and some smoke rising, for that:

```
BMAP      flash_1,<knallig.lbm>,0,0,16,16;
BMAP      flash_2,<knallig.lbm>,16,0,16,16;
BMAP      flash_3,<knallig.lbm>,32,0,16,16;
```

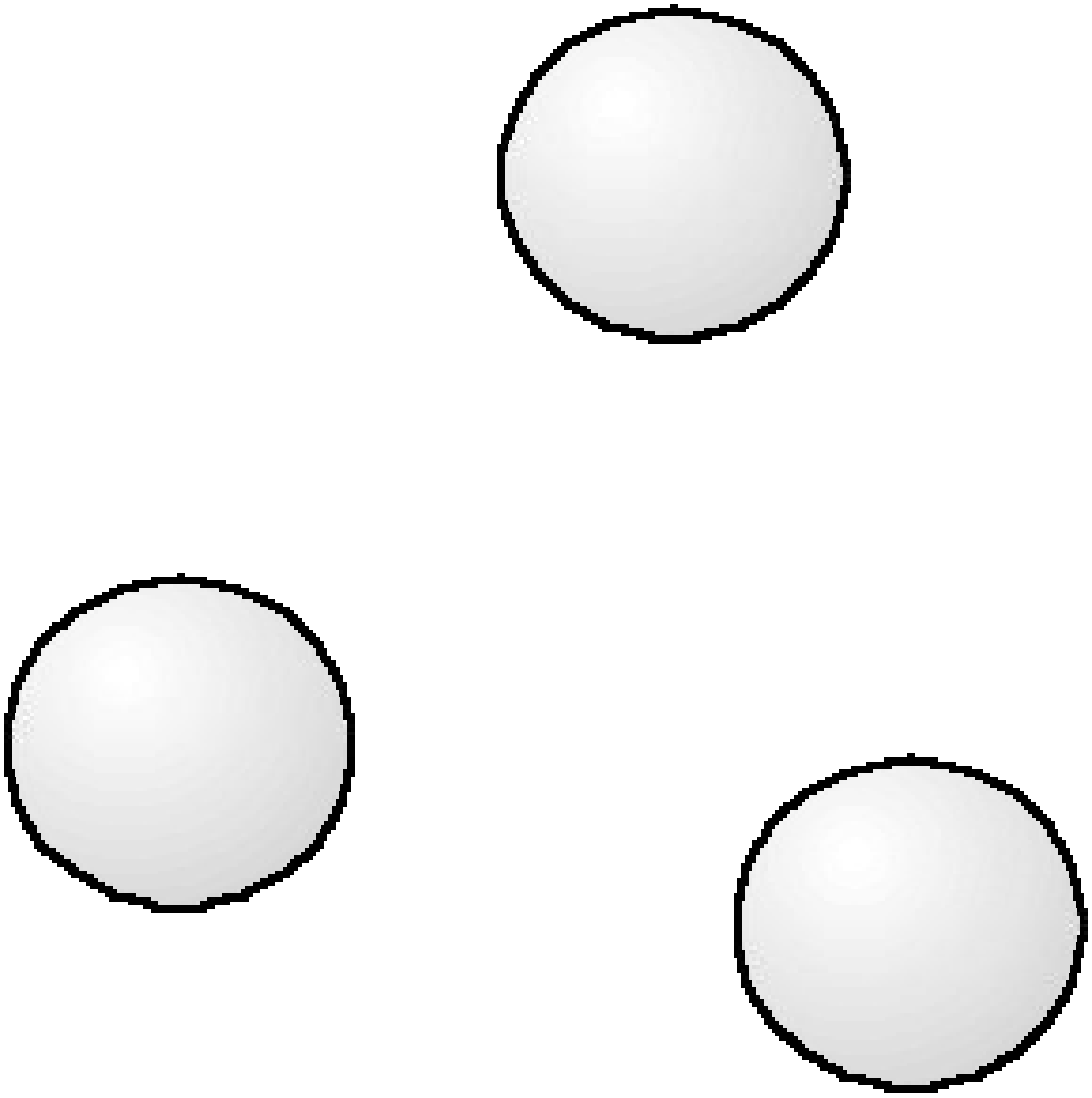
```
BMAP	flash_4,<knallig.lbm>,48,0,16,16;
BMAP	wolke_1,<knallig.lbm>,124,0,20,20;
BMAP	wolke_2,<knallig.lbm>,144,0,20,20;
BMAP	wolke_3,<knallig.lbm>,164,0,20,20;
BMAP	wolke_4,<knallig.lbm>,184,0,20,20;
BMAP	wolke_5,<knallig.lbm>,204,0,20,20;
TEXTURE flash_tex {
SCALE_XY		12,12;
CYCLES		4;
BMAPS		flash_1,flash_2,flash_3,flash_4;
DELAY		1,1,1,1;
FLAGS		ONESHOT;
AMBIENT	1;			// flash shines in dark regions also!
}
TEXTURE wolke_tex {
SCALE_XY		12,12;
CYCLES		5;
BMAPS		wolke_1,wolke_2,wolke_3,wolke_4,wolke_5;
DELAY		2,2,2,2,2;
FLAGS		ONESHOT;
AMBIENT	0.5;
}
ACTOR	flash1 {
TEXTURE flash_tex;
FLAGS		GROUND,INVISIBLE,PASSABLE;
}
ACTOR	flash2 {
TEXTURE flash_tex;
FLAGS		GROUND,INVISIBLE,PASSABLE;
}
ACTOR	flash3 {
TEXTURE flash_tex;
```

```
FLAGS            GROUND,INVISIBLE,PASSABLE;
}
ACTOR    flash4 {
TEXTURE flash_tex;
FLAGS            GROUND,INVISIBLE,PASSABLE;
}
SKILL hgt        { }
SKILL hdist      { }
ACTION init_flash {
SET      MY.TEXTURE,flash_tex;
SET      MY.PLAY,1;                // start animation
SET      MY.DIST,hdist;     // hit distance
ADD      MY.DIST,-0.5;
// compute approximate height of hit
RULE     hgt = PLAYER_Z + -0.83*hdist*PLAYER_TILT + 0.4*RANDOM - 1.2;
SET      MY.HEIGHT,hgt;
DROP     MY;                               // place flash in front of hit object
SET      MY.SPEED,0.3;             // flash approaches player
SET      MY.TARGET,FOLLOW;
SET      MY.EACH_TICK,NULL;
SET      MY.EACH_CYCLE,init_wolke;  // smoke rises after animation
}
ACTION init_wolke {
SET      MY.TEXTURE,wolke_tex;
SET      MY.PLAY,1;
SET      MY.EACH_TICK,wolke_auf;
SET      MY.EACH_CYCLE,flash_weg;
}
ACTION wolke_auf {                         // let smoke rise
ADD      MY.HEIGHT,0.4;
}
ACTION flash_weg {                         // smoke disappears
```

```
SET         MY.INVISIBLE,1;
}
ACTION fire {
fire_loop:
PLAY_SOUND          twaeng,0.3;
SET                 LAYERS.13,cannon1;
ADD                 PLAYER_LIGHT,0.5; // flash of light
SET                 SHOOT_RANGE,200;
SHOOT;                                                          // shoot from 200 steps distance
IF_EQUALHIT_DIST,0;                         // hit nothing?
GOTO      fire_weiter;
SET                 hdist,HIT_DIST;         // save distance from hit
IF_EQUALflash1.INVISIBLE,0;   // is flash1 still active anywhere?
GOTO      set_flash2;               // then use flash2
SET                 flash1.INVISIBLE,0;   // else activate flash1 now
SET                 flash1.EACH_TICK,init_flash;
GOTO                fire_weiter;
set_flash2:
IF_EQUALflash2.INVISIBLE,0;
GOTO      set_flash3;
SET                 flash2.INVISIBLE,0;
SET                 flash2.EACH_TICK,init_flash;
GOTO                fire_weiter;
set_flash3:
IF_EQUALflash3.INVISIBLE,0;
GOTO      set_flash4;
SET                 flash3.INVISIBLE,0;
SET                 flash3.EACH_TICK,init_flash;
GOTO                fire_weiter;
set_flash4:
SET                 flash4.INVISIBLE,0;
SET                 flash4.EACH_TICK,init_flash;
```

```
fire_weiter:
WAIT                    1;
SET                     LAYERS.13,cannon2;
ADD                     PLAYER_LIGHT,-0.5;
WAIT                    1;
SET                     LAYERS.13,cannon3;
WAIT                    1;
SET                     LAYERS.13,cannon5;
WAIT                    1;
SET                     LAYERS.13,cannon0;
WAIT                    1;
IF_EQUALKEY_CTRL,1;
GOTO       fire_loop;
IF_EQUALMOUSE_LEFT,1;
GOTO       fire_loop;
}
```

Given our knowledge on state machines it should not be to hard to understand this action. The flash-on-hit of the gun we define as an actor. For each actor two **ONESHOT** texture animations are available. The chart of states of such short-lived actors is fairly simple (cf. fig.)



The actor is activated when a new **SHOOT** instruction from our fire action meets with an obstacle, i.e. if the skill **HIT_DIST** is not 0. The action **init_flash** sets the textures of its first state and puts it over the spot that was hit **-** that is 0.5 steps in front of the wall of the object hit, in direction of the player and at the height of the centre of the 3-D window. It is put there by the **DROP** command.

The **EACH_CYCLE** action **init_wolke** is initiated after the flash animation is through. The actor is turned into a gust of smoke that rises heavenwards at the speed of 0.4 steps per tick. After the cloud animation is through the actor is set to **INVISIBLE** and thus ready for the next hit.

We will need several **flash** actors, as our gun is firing continually. While the first gust of smoke is still rising new hits must light up next to it. Thus within our **fire** action we first check which flash is currently unemployed in order to initialise it for the next flash.

Automapping

To come to an end we would now like to turn our attention to a special kind of overlay: the automap. This map may be displayed constantly over the 3-D window through automapping. By a predefined skill one of the 16 available layers may be defined as the map layer:

```
SKILL MAP_LAYER  { VAL 14; }
```

The overlays on layers 1-14 would now be displayed *below* the map, those in layers 15 and 16 would be displayed *above* the map. In order for the map to become visible in the first place we have to set the skill **MAP_MODE**. This we do by adding to our **show_panels** action above:

```
SKILL MAP_SCALE              { VAL 3; MIN 0.1; MAX 30; }      // limit area

ACTION show_panels {

RULE                 fps = 0.8*fps + 0.2*TIME_FAC;

SET                  MAP_MODE,0;

IF_EQUALKEY_TAB,0;          // [TAB] not pressed?

GOTO      pan_nomap;

SET                  MAP_MODE,0.5;      // automapping

IF_EQUALKEY_SZ,1;                        // change map scale in steps of 5%

RULE      MAP_SCALE = MAP_SCALE*0.95;

IF_EQUALKEY_MINUS,1;

RULE      MAP_SCALE = MAP_SCALE*0.95;

IF_EQUALKEY_APO,1;          // that's [+] on US keyboards

RULE      MAP_SCALE = MAP_SCALE*1.05;

IF_EQUALKEY_PLUS,1;

RULE      MAP_SCALE = MAP_SCALE*1.05;

pan_nomap:

}
```

The map will now be projected onto the screen when the **[TAB]** key is hit. The scale can be changed using the **[+]** and **[-]** keys. **MAP_MODE 1** would display the complete level, **0.5** merely the area seen.

Changing Levels

Usually a game would consist of several *levels* the player has to fight his way through one after the other. This confronts us with a new problem. We have to create a passage between those levels. If the player on a lower level has collected features or weapons he must be allowed to take them with him to higher levels. Apart from that you have to be able to stop playing at any time and later resume playing in the same level at the same spot. This demands of us the ability to either completely or partially save the current situation within the game.

There are three ways to change levels. On the one hand we could accommodate different separate levels within the same WMP file. Between these sub-levels there would be **border** regions to stop the player from simply walking from one level to the other in the usual way. Apart from that there would have to be a sufficient distance between the levels in order for the levels to lie outside the **CLIP_DIST** of the other levels.

To be able to 'beam' the player from one place in the topography to another one without having to fiddle about with co-ordinates we had best define a pseudo-object as the beam-objective and place it on the position of the destination in the level concerned via WED:

```
THING transporter_1 {                   // one transporter per level (or several?)

TEXTURE pseudo_tex;         // some texture

FLAGS     INVISIBLE;                    // so that the Thing won't sit on our face

}

ACTION beam_1 {

SET       PLAYER_X,transporter_1.X;

SET       PLAYER_Y,transporter_1.Y;

SET       PLAYER_ANGLE,transporter_1.ANGLE;

SET       HERE,zielregion_1;    // region the transporter_1 is in

SET       PLAYER_Z,zielregion_1.FLOOR_HGT;

RULE      PLAYER_Z = PLAYER_Z + PLAYER_SIZE;

SET       PLAYER_HGT,0;

}
```

The action **beam_1** causes the player to appear at the transporter-objective immediately. It may of course also be used to beam about within the same level! It must be remembered also to re-assign the player-region **HERE**, or else the collision-detection wouldn't work any more.

This method of putting several levels inside the same WMP file bears some advantages as well as disadvantages:

- levels may be changed in a jiffy without accessing the disk
- the maximum size of the WMP file (app. 5000 objects) limits the number and size of levels
- starting up the game takes longer because textures for all levels have to be loaded at the start of the game.

One further way to do it is to create a separate WMP file for every level. In that case the keyword **MAPFILE** would only define the topography of the start-level; further levels may then be accessed with the instruction **MAP** within an action:

```
ACTION beam_2       { MAP <level_2.wmp>; }
```

By **beam_2** the player is put directly onto the player start-position within the file LEVEL_2.WMP (the name of the file we put in pointed brackets so that it may be included in the game resource later). Walls, Things, and Actors are read from the new WMP file; the remaining parameters - skills, panels etc. - stay the same. The drawback of this method is that the disk is accessed for several seconds in order to load and build the new level. As a consequence the game 'freezes' for a short interval. Because the WDL file is not changed, all walls, regions, and other objects of the old as well as of the new level have to defined there.

This disadvantage may be avoided by loading the complete WDL to begin with. This would be the third and most extravagant (because you'll need the professional version) way of changing levels that basically equals restarting the game. The instruction **LEVEL** is used for that:

```
ACTION beam_3       { LEVEL "level_3.wdl","level_3.wrs"; }
```

The file LEVEL_3.WDL is loaded from the compiled World Resource LEVEL_3.WDL and started as a new game. The file names by the way here - unlike with the **MAP** instruction - are given with quotation marks because they shouldn't be included in the level by the compiler. All the objects, positions, skills, and current actions from the old level are 'forgotten'; it goes on with the **IF_START** action in the new WDL file. Yet there is one exception differentiating the **LEVEL** instruction from a complete start-up: the behaviour of a few special skills, namely those of the **GLOBAL** type.

Global Skills

GLOBAL skills are simply defined through the keyword **TYPE GLOBAL**; within the curved brackets of the skill definition. Global skills are meant for 'universal' parameters that are to be valid in all levels, e.g. sound and music settings or the player's strength and equipment. They are not 'forgotten' when changing the level but keep their old values in the new level, even if they were redefined there, assigning them different values (**VAL**). And this they have to because the old as well as the new WDL file *must* include exactly the same **GLOBAL** skill definitions with the same skill names in the same order!

The values of all **GLOBAL** skills may be separately written to the disk and reloaded. The instructions **SAVE_INFO** and **LOAD_INFO** are used for that. These instructions allow you to write general parameters - like sound and music settings, skill level, etc. - to the harddisk independent of game scores.

```
////////////////////////////////////////////////////////////

SKILL MOTION_BLUR          { TYPE GLOBAL;     VAL 0.5; }

SKILL SOUND_VOL            { TYPE GLOBAL;     VAL 1; }

SKILL MUSIC_VOL            { TYPE GLOBAL;     VAL 0.5; }

////////////////////////////////////////////////////////////

SAVEDIR  "C:\\MYGAMES";

ACTION global_save {

SAVE_INFO          "test",0;

}

ACTION global_load {

LOAD_INFO          "test",0;

}
```

SAVE_INFO saves a file containing the skill values in the directory referenced in **SAVEDIR** under the name given in the first parameter; the additional number or skill adds a number of not more than three digits. The file name is completed by the ".SAV" extension. Given the above actions the three global skills will be written to and read from the file C:\MYGAMES\TEST0.SAV.

Should the directory referenced (heed the double backslash \\ !) not yet exist when writing to the disk it will be created. And if the instruction **LOAD_INFO** is unable to find the file referenced it simply leaves the **GLOBAL** skills the way they are.

It may sound like a good idea to execute the action **global_load** at the start of the game and **global_save** at the end. Then the game will 'remember' all changes made to the global skills brought about by changing the volume slide-adjuster on the panel and keeps the new volume when starting anew.

How by the way can we achieve that only a few but not all global skills are read from the hard disk by **LOAD_INFO**? Simple: we copy the values of those skills not to be changed to 'shadow-skills' in order to get them back after

LOAD_INFO. So if we wouldn't want the skill **MOTION_BLUR** to change after the read action above, it would look like this:

```
SKILL motion_blur_s { }          // shadow skill for MOTION_BLUR

ACTION global_load_vols {                    // read just SOUND_VOL and MUSIC_VOL

SET                motion_blur_s,MOTION_BLUR;  // scrap old value

LOAD_INFO          "test",0;

SET                MOTION_BLUR,motion_blur_s;  // get back old value

}
```

One more advice: when fiddling about with saving and loading of global skills experience tells us that once in a while terrifying crash-like situations will arise. That will always happen if the number of global skills was changed since they were last saved. Thus: better delete all old skill files after editing the WDL **-** just to be on the safe side!

Saving and Loading Game Scores

With the **SAVE** instruction the complete state of the game is written to the harddisk all at once. This concerns the following parameter:

- Names of the current .WDL and .WMP files,
- value of all normal and **GLOBAL** skills,
- content of all synonyms (**MY**, **THERE**, etc.),
- state of the lists **EACH_TICK**, **EACH_SEC**, **LAYERS**, **PANELS**, **MESSAGES**,
- all key-actions (**IF_F1** etc.)
- positions and other modifiable parameters of every **PANEL**, **TEXT**, or **OVERLAY**,
- coordinates and other parameters of all actors,
- flags of all regions, walls, and things,
- coordinates and other parameters of every region, wall, and thing whose flag **SAVE** was set. This flag has to be set with all objects that may alter in any way during the game.

The scores saved are keyed and compressed so that they won't use up to much room on the harddisk. With the instruction **LOAD** a saved score is uncoded and reloaded. All **EACH_TICK** actions that were running when saving occurred will start again at the beginning after loading the score. If the player is in a different level during loading, the level will be changed automatically like with a **LEVEL** instruction. Loading itself **-** like the instructions **LEVEL** and **MAP -** will not be executed immediately in the action the corresponding instruction can be found in but only after the end of the action.

As for the scores the same is true as what was said about global skills above: loading an old score after changing the number of objects in the WDL file leads to...no, not exactly to a crash, but in any case to an abortion of the game complete with the corresponding error message.

We shall now use all the skills we have acquired so far to define an action for saving and loading scores. We want to save a score through the **[F2]-**key and reload it through the **[F3]-**key. That sounds fairly simple, doesn't it? In order to keep things interesting then we want to be able to chose between the three scores last saved by repeatedly pressing **[F3]**. A corresponding on screen message is to give the number of the score concerned.

```
//////////////////////////////////////////////////////////
// Game score management
//////////////////////////////////////////////////////////
SKILL slot { TYPE GLOBAL; VAL 1; }          // number of score (..3)
//////////////////////////////////////////////////////////
SKILL number            { VAL 0; }
//////////////////////////////////////////////////////////
SAVEDIR  "C:\\MYGAMES";
STRING leer," ";
STRING ok,"OK";
STRING save_yesno,"Save game (Y/N)?";
STRING load_yesno1,"Load last score (Y/N/F3)?";
STRING load_yesno2,"Load last-but-one score (Y/N/F3)?";
STRING load_yesno3,"Load 2nd-to-last score (Y/N/F3)?";
STRING wait_txt,"Please wait...";
STRING save_nix,"ERROR - nothing saved!\nHard disk already full?";
STRING load_nix,"Sorry - no score found!
Remember:\nFirst save - then load!";
//////////////////////////////////////////////////////////
FONT        standard_font,<panfont.pcx>,8,10;
TEXT screen_txt {
POS_X       30;
POS_Y       24;
FONT        standard_font;                   // s.o.
STRING      leer;
}
//////////////////////////////////////////////////////////
// Message actions
//////////////////////////////////////////////////////////
ACTION show_message {                        // show message
SET         MESSAGES.14,screen_txt;
SET         EACH_TICK.14,show_message;
```

```
WAITT      80;                                          // wait 5 seconds

SET        MESSAGES.14,NULL;

SET        EACH_TICK.14,NULL;

}

ACTION wait_yesno {                           // wait for Y/N keystroke

SET        MESSAGES.14,screen_txt;            // show query

SET        MOVE_MODE,0;                                 // freeze player

SET        IF_N, clear_yesno;     // assign keys

SET        IF_ESC, clear_yesno;

}

ACTION clear_yesno {                                    // end Y/N query

SET        MESSAGES.14,NULL;                  // text gone

SET        MOVE_MODE,1;                       // de-frost player

SET        IF_J, NULL;                        // unassign keys

SET        IF_Y, NULL;

SET        IF_Z, NULL;

SET        IF_N, NULL;

SET        IF_ESC, NULL;

SET        IF_F3,qload_game1;  // F3 reloads last score

}

////////////////////////////////////////////////////////////

// Saving & loading

////////////////////////////////////////////////////////////

ACTION qsave_game {                                     // save score, with query

SET        screen_msg.STRING,save_yesno;

SET        IF_J, save_game;

SET        IF_Y, save_game;

SET        IF_Z, save_game;

CALL       wait_yesno;

}

ACTION load_slot {                            // reads slot skill off hard disk

LOAD_INFO            "INFO",0;
```

```
}
ACTION save_game {                          // save score directly
CALL            clear_yesno;
CALL            load_slot;  // load slot skill
RULE            slot = slot + 1;            // and increment it
IF_ABOVE        slot,3;         // if >3 then back to 1
SET     RULE slot = slot - 3;
SAVE_INFO       "INFO",0;       // save new slot skill again
SET             RESULT,0;                   // to detect errors
SAVE            "GAME",slot;    // save game
SET             screen_msg.STRING,ok;
IF_BELOW        RESULT,0;                   // error at saving?
SET     screen_msg.STRING,save_nix;
CALL            show_message;   // show error message
}
ACTION qload_game1 {
SET             number,0;                   // take last score
SET             screen_msg.STRING,load_yesno1;
SET             IF_Y,load_game;
SET             IF_Z,load_game;
SET             IF_J,load_game;
SET             IF_F3,qload_game2;
CALL            wait_yesno;
}
ACTION qload_game2 {
SET             number,1;                   // take last-but-one score
SET             screen_msg.STRING,load_yesno2;
SET             IF_Y,load_game;
SET             IF_Z,load_game;
SET             IF_J,load_game;
SET             IF_F3,qload_game3;
CALL            wait_yesno;
```

```
}

ACTION qload_game3 {

SET             number,2;                   // take 2nd-to-last score

SET             screen_msg.STRING,load_yesno3;

SET             IF_Y,load_game;

SET             IF_Z,load_game;

SET             IF_J,load_game;

SET             IF_F3,qload_game1;

CALL            wait_yesno;

}

ACTION load_game {

CALL            clear_yesno;

SET             screen_msg.STRING,wait_txt;

SET             MESSAGES.14,screen_msg;

SET             EACH_TICK.14,load_game;

WAIT            1;                                  // to show the message before loading

CALL            load_slot;  // load slot skill

RULE            slot = slot - number;

IF_BELOW        slot,1;

RULE     slot = slot + 3;

LOAD            "GAME",slot;

SET             EACH_TICK.14,NULL;

SET             screen_msg.STRING,load_nix;   // went wrong!

CALL            show_message;               // error message

}

////////////////////////////////////////////////////////////

IF_F2    qsave_game;

IF_F3    qload_game1;
```

So for this seemingly simple job we need lots and lots of tricky actions. We are going to see that one of the problems is to remember the number of the score last saved after loading some other new score. Because loading a game changes every skill, we have to save the number of the game last saved separately on the harddisk. The only way to do this is to use a global skill, which we have called slot in this case.

To begin with we'll define ourselves a few actions to give on screen messages or to wait for a Yes/No keystroke. **show_message** shows the text **screen_txt** for 5 seconds on screen. **wait_yesno** presents the same text, but waits for a keystroke as an answer from the user. In the meantime the game is 'frozen' by setting the predefined skill **MOVE_MODE** to 0. The action resulting from a 'yes' answer has to execute a **CALL clear_yesno** first so that the player may be 'defrosted' and the keys unassigned. Pushing of **[N]** or **[Esc]** results in immediate execution by **clear_yesno**. The 'Yes' action will not only be assigned to the [Y] key, but also to [J] and [Z] in order to allow for european keyboards.

The action **qsave_game** is executed through the **[F2]** key. Through pushing **[Y]** after the security query we then start the proper save action **save_game**. To begin with the global skill slot is read from the file INFO0.SAV. This skill contains the number of the score last saved. It is increased by 1 and reset to 1 in case its maximum value of 3 is exceeded and then saved again in the file INFO0.SAV. Afterwards the proper saving of the score under the name of GAME0.SAV, GAME1.SAV, or GAME2.SAV is carried out **-** depending on which value slot currently has. If an error occurs during saving the predefined skill **RESULT** automatically acquires a value not equal 0. In this case we give out an error message, or else "OK", using the action **show_message**.

With loading through **[F3]** as well there is first the question to the user. In this case you may answer not only through **[Y]** or **[N]** but also through pressing **[F3]** again. Each pushing of **[F3]** prompts a different message on screen, changes the skill number and cyclically assigns one of the three actions **qload_game1**, **qload_game2**, or **qload_game3** to **[F3]**.

Pushing **[Y]** executes the action **load_game**. This action is meant to prompt the message "Please wait...." on screen before loading the game, which may take quite a while in case a change of levels is necessary. Unfortunately the representation of a text begins but with the next building of the screen. So we would never get to see the message, because the **LOAD** instruction stops screen refresh until the level was built anew completely. That is why **load_game** puts itself on the **EACH_TICK** list after the tried-and-true fashion and executes a **WAIT** instruction. The **wait_text** is now on screen.

In order to determine the number of the score to be loaded, slot is again read from the global skill file and the skill number is substracted from it. The value number states whether the last (0), last-but-one (1), or second-to-last (2) score is to be loaded. At that point **-** finally **-** loading itself can ensue. And now, as a conclusion, a little riddle: This time we won't have to evaluate the skill **RESULT** in order to determine whether something went wrong with loading the file. Now, why would that be?

Index