3D GameStudio

World Definition Language Tutorial & Manual

for A4 engine 4.23
and A5 engine 5.03

Johann C. Lotter / Conitec February 2001

Partially translated by Guenther Mulder

WED level editor	Matthew Ayres, Paul Hsu, Wladimir Stolipin
MED model editor	Wladimir Stolipin
Map compiler	Gennadi Wart
WDL compiler	Volker Kleipa
Engine	Johann Christian Lotter
Example games	Czeslav Gorski, Harald Schmidt, Doug Poston
Script prefabs	Doug Poston

The latest news, demos, updates and tools, as well as the Users' Magazine, the Users' Forum and the annual Contest are available at the GameStudio main page **http://www.3dgamestudio.com**.

If you own the commercial or professional edition, you are entitled to 3 or 12 months **technical support** via email. Please mail your question to **acknex@conitec.net**, and give your edition, purchase date and serial or customer number (on the key disk or on your invoice). Try finding the answer in the manual first! Technical questions can only be answered if sent by email to that address. Normally each question is answered within one working day.

Contents

Tutorial: Teach Yourself Game Programming in 6 Days 6
Sunday afternoon: WDL Syntax 7
Variables and Strings 8
If Branching 10
Loops 12
Debugging 14
Monday: Doors and Keys 18
Actions 18
The First Movement 19
User interaction 20
Tuesday: Game Physics 23
Acceleration, Inertia, Friction 25
Falling down 27
Wednesday: Artificial Intelligence 30
The Theory of Black Boxes 30
The Table of States 31
Advanced State Machines 36
Thursday: The User Interface 37
Texts 37
Panels 38
Friday: Controlling the Game 41
Changing Levels 41
Saving and Loading Games 42
Reference: WDL Syntax 44
Functions - the brain of the game 46
Variable functions 47
Vector instructions 48
String instructions 50
File instructions 52
Control instructions 54
Entity instructions 56
Multimedia instructions 63
Input / output instructions 64
Game flow instructions 65
Plugin DLL instructions 67
Debug instructions 68
Attaching functions or actions 69
Variables, Strings, Synonyms 71
Variables and Arrays 71
Strings 72
Synonyms 72
File Objects 73
Entities - inhabitants of the game world 75
Screen control parameters 75

Position parameters 76
Visual Parameters 76
Collision detection 81
Events 82
Internal parameters 85
User interface: Panels, Texts, Views 86
Panels 86
Texte 89
Views 90
Engine-Variables 93
Predefines 104
Database and Dataview 106
Multi-User Mode 108
Starter Window Definitions 110
Reference: Prefabricated Scripts 112
Movement.wdl 112
Actors.wdl 114
Weapons.wdl 115
War.wdl 117
Doors.wdl 118
Messages.wdl 120
Particle.wdl 121
Menu.wdl 121
Venture.wdl 122
Appendix 132
WDL traps 132
The dreaded wait() instruction 132
Those bloody invalid synonyms 132
Bad timing 133
Wrong mathematics 133
The event trap 134
Attaching entities to each other 134
Appendix B: Starting the Engine 136
Command Line Options 136
D3D Considerations 138
Error Messages 139
Frequently Asked Questions 144
Appendix C: Legal stuff 146
Frequently Asked Questions 147

Congratulations: As owner of 3D GameStudio you can now create interactive realtime applications – especially, but not only 2D and 3D computer games – quick, easy and without previous programming knowledge.

"Easy" does not mean that GameStudio applications are all prefabricated and look and play similar. You can use WDL, a scripting language, for giving the user interface and the objects within your game an arbitrary behaviour, which may suit especially your needs. This way, your games will not necessarily look and feel like "made with 3D GameStudio"; they will look and feel the way you've programmed them.

Creating commerical quality games is an ambitious task. Before reading this manual, you should start WED and begin with the WED tutorial to get an idea how 3D GameStudio works. At reaching the last lesson, you're ready for WDL. This manual consists of several parts. The first part, the WDL tutorial, is an introduction into the programming language. In the following part, the WDL reference, the syntax and objects of the World Definition Language **WDL** is described. If you don't want to write your own WDL scripts, but prefer to use the prefabricated ones, read the third part to learn how to define your gameplay by setting flags and parameters of the predefined functions.

Tutorial: Teach Yourself Game Programming in 6 Days

Let's start with some simple assumptions: you have dug into the WED Tutorial and created some cool levels. Now you want to glue your levels together to a great 3D game. You don't want to used the predefined game **template**s – you want to insert monsters, effects and puzzles of your own imagination. To accomplish this, you feel that you'll need to write some WDL scripts. So you have read the WDL Reference Manual from cover to cover, learned everything by heart, but understood none of it. You have studied all the prefabricated WDL scripts in the **template** directory, and understood even less.

After finishing this tutorial, you will know how to program a game. Even though it's simple to work with, WDL has the basic features of a modern object oriented programming language. WDL is based on **Javascript**, the language for web pages. So as you learn WDL scripting, you're also learning the basics of computer programming. If you want to move on to other modern programming languages, like C, C++, or Java, WDL is a great introduction. And one of the best things about WDL is that you can do a great deal with very little programming.

Enough hype about WDL. On to hype about this tutorial.

Today, we'll be running the first part of a six-day tutorial that aims to get you game programming by writing useful WDL scripts immediately. Here's a brief outline of what you will learn each day.

Sunday: Your first WDL script (lesson based on Thau's Javascript Tutorial).
Monday: Entities, doors, elevators, keys.
Tuesday: Player movement, game physics.
Wednesday: Artificial intelligence, enemies, fighting.
Thursday: User interface, panels, menus.
Friday: Advanced stuff, saving, loading, multi-level games.

Before we jump in, here are few important things to note about WDL and this tutorial.

First: don't forget that this tutorial is not a substitute for the WDL Reference Manual. Although you'll learn most of the grammar of WDL here, you won't learn all of the functions and effects available. But you'll learn enough to continue on your own.

Second: view WDL source! The best way to learn WDL is to look at the WDL scripts in the **template** directory, and at scripts other people have written. Do it frequently!

Third: experiment freely and often. At several places in this tutorial, you'll be given the opportunity to try things out. Don't be afraid to expand beyond the exercise to try new things.

All right, enough with the disclaimers. Down to business. Let's write some WDL.

Sunday afternoon: WDL Syntax

If you are already familiar with programming and scripting languages like Javascript or even C, you may choose to skip this chapter and just start with Monday's lesson about doors and keys. If not: start WED, and open and run the **office** level (**[run]** button, but don't forget **[build]** before). While looking at the sky, press the **[Tab]** key. The game freezes, and a blinking cursor appears on the screen. Now type

```
sky_scale = 0.3;
```

and press **[Enter]**. If you now observe a change at the sky texture, you just witnessed the effect of your very first WDL instruction! Edit the 0.3 to another value using the cursor keys and **[Del]**, then press **[Enter]** again. Play around with values and observe the effect in the game. You can leave this direct entry mode through **[Esc]**.

What you've done is changing the **value** of a pre-defined numerical WDL variable named **sky_scale.** It won't surprise you that this variable is responsible for the scaling of the sky texture. WDL statements are just text that can be typed into a text editor, or – using the **[Tab]** key – into the game itself. The former method is the normal one, the latter is for testing and playing around. By the way, the **[Tab]** key one-line entering mode was itself written in WDL.

Exit the game, and now open the **office.wdl** file in the WORK directory using a plain text editor - Windows' Notepad™, for instance. A better solution is a special syntax hightlighting WDL editor – you can download one from the Conitec Website. Never use a word processor like Word™ or WordPerfect™ for scripts – they will trash the files with their formatting codes!

You should now see the beginning of the main **office.wdl** file, which looks like this (or similar):

```
////////////////////////////////////////////////////////////////////////////
// Office test level
////////////////////////////////////////////////////////////////////////////
path  "..\\template";     // path from the WORK folder to the TEMPLATE folder
include <movement.wdl>;   // libraries of WDL functions, located in TEMPLATE
include <messages.wdl>;
include <doors.wdl>;
include <particle.wdl>;
include <actors.wdl>;
include <weapons.wdl>;
include <war.wdl>;
include <menu.wdl>;

////////////////////////////////////////////////////////////////////////////
// After engine start, the MAIN function is executed, so that the 2-D engine
// can display pictures, logos, or AVI animations. To load
// a level, you must perform an explicit load_level() instruction.

function main()
{
   load_level(<office.wmb>);
   ....
```

This is a basic WDL script, like it's automatically created after clicking the **[new]** Button in Map Properties.

Everything between **//** and the end of a line is a **comment** and will be ignored by the engine. A basic rule of good scripting style is that you should always think about the next person who has to look at your script. It might be a friend, a co-worker, an employer, or it could be you in three

months. The easiest way to make sure you'll understand your own script in three months is to comment freely and often. If you want to comment a huge block of text, you can put it between **/*** and ***/** like this:

```
/* this is a block
of text that I've
commented out */
```

At the beginning of the WDL file a directory path is given, and a lot of further WDL files are **include**d from this path. **Include** means that they are handled as if their content were directly typed into the script. These WDL '**template**' files contain predefined objects and functions to be made available in the game. They are described in other chapters of the manual, and can be used for doing a simple game, like a shooter, with no or very few WDL programming. But if you intend to create a very unique game, you may decide to throw all predefined stuff out, and script everything from scratch. You'll begin with an empty WDL file in that case.

The **function main()**… below is executed directly at engine start, and contains all things to do at the begin of the game – which is, in the example, loading the **office.wmb** level. However, let's now start learning the language – how WDL stores information, how it makes decisions based on that information, and how to change information based on user interaction. Ready? It's time to learn the fundamentals of computer programming. First stop: variables.

Variables and Strings

If you've taken algebra, you've seen **variables**. If you haven't taken algebra, don't worry about it. Variables are simply the way any programming language, like WDL, stores numerical information. For example, if you write "x = 2", "x" is a variable that holds the value "2." If you then say "y = x+3", "y" will hold the value "5."

Here's an example of WDL that creates variables:

```
// define some vars
var secs_per_min = 60;
var mins_per_hour = 60;
var hours_per_day = 24;
var days_per_year = 365.25;
var secs_per_day;
var secs_per_year;
```

As you see, each WDL **statement** or instruction must end with a semicolon. If you forget it, you can be sure to receive an error message from the engine. The first line beginning with '**//**' is a comment that states the obvious. The next bunch of lines are variable **definitions**. A definition in WDL always creates something. There are a few things to notice about these lines:

Before you can use a variable, you must define it with the word "**var**", followed by its name, then by an optional initial value after a "**=**". Some variables (like our **sky_scale**) needn't have to be defined – they are already internally predefined by the engine.

Variable names must start with either a letter or the underscore character. After the first character, variables can have numbers. So **harry_23** is an acceptable name for a variable.

Variable names are not case-sensitive in WDL. This means that the variables **Loop**, **loop** and **LOOP** will be considered the same. Generally, it's a good idea to pick a naming convention and stick to it. I like having all my variables lowercase, with underscores separating words. Other people prefer using internal capitalization, like **SecsPerMin**.

Variables should describe what they are. Variable names such as **a**, **b**, or **number**, aren't very useful to a person who's trying to figure out your script. Don't make your variables so long that they take forever to type, but make them long enough to be descriptive.

You can give a variable an initial value when you define it. In the example, some of the variables were given a value the first time they were defined. You don't have to do this, and we'll see examples where it's ok to define a variable even though we don't know its value right away. A variable can represent more than one number – we'll learn later about **vectors** that contain three numbers, and **arrays** that contain as much numbers as you like.

Statements end with a semicolon. Statements are the sentences of WDL and semicolons are the end punctuation marks. Spaces and line breaks are normally ignored, so the layout of the script serves only to make it more legible for people. This example could have been written in one really long line if you take out the comments. But that would be hard to read.

Ready to do something with our variables? A WDL object that does something with something is a **function**, consisting of **instructions**:

```
// do some calculations
function some_calculations()
{
   secs_per_day = secs_per_min * mins_per_hour * hours_per_day;
   secs_per_year = secs_per_day * days_per_year;
}
```

Functions' naming rules are the same as those for variables. The first character has to be a letter or an underscore. The rest of the characters can be numbers. Also, you must make sure you don't name a function the same as a variable!

After the name comes two parantheses, between which parameters could be transferred to the function – but we aren't using this here, so the parantheses are empty. Then comes the body of the function within the curly brackets. This is the set of instructions you want to run when the function is executed. In our example we see two instructions with some basic math. The expression right of the '**=**' is calculated, and the result is placed into the variable left of the '**=**'. So after the engine has executed the WDL function, the variable **secs_per_year** will contain whatever you get when you multiply 60, 60, 24, and 365.25. From the moment the function was executed, whenever WDL sees the variable **secs_per_year,** it will substitute in that huge number.

You can learn a second thing from this function: In most cases, the order of instructions does matter. Instructions are executed from the beginning of the function to the end. If we had swapped the two instructions, the result would be nonsense, because the second instructions needs **secs_per_day** already calculated!

After we've done with variables, let's now introduce a second kind of object to store information, named **string**. Strings are somewhat similar to variables, but they contain characters instead of numbers. Any group of characters, given between quotes, is a valid content of a string. So it's legal to define:

```
// a string definition
string hello = "hello world!";
```

Sticking these statement into a WDL file defines the string **hello** and fills it with the characters given. However there are some subtle differences between strings and variables. You can't do mathematics with a string. You can change the string content like you can with a variable, but you

can't omit the initial character sequence, because this gives the maximum character number – the length – of the string defined.

Once you've defined a string, you can display it by executing a function

```
function greet_world()
{
   msg.visible = on;
   msg.string = hello;
}

on_p = greet_world;
```

Here we're using a text object, **msg**, which is already defined in one of the **template** script files that were included at the beginning of our office wdl script. Text objects are used to display strings on the screen. You'll learn more about them later. For the moment, it will be sufficient to know that if we set **msg.string** to a certain string, and have set **msg.visible** to **on**, that string will be visible on the screen. The dot after **msg** indicates that the following item, string or visible, belongs to the object **msg**. Having sub-items belonging to an object this way is typical for an object oriented programming language. Visible is a **flag** – this is something like an **on/off** switch for the text object. Strings and flags are set with the "**=**" assignment.

The **on_p** statement assigns this function to the **[P]** key. Normally we must assign a function to something that starts it – just having typed it into a WDL file is not sufficient for the function to be executed! The only exception is the **main()** function that is automatically started at game start, as mentioned before. To try this one, go to the end of the **office.wdl**, and type the string definitions, the function and the **on_p** line behind the last line. Now save the wdl file, and run the office level again.

If it won't start now, but gives an error message, you can be sure that you've forgotten a little semicolon or something like that or typed something wrong. The line in question is displayed in the error message, so you can easily see and correct your mistake. Check carefully. If the level starts at least, press **[P]**. Wow! Our first WDL function is running!

Once you've gotten this to work, it's time to learn about if-else clauses.

If Branching

Branching that involves "**if**" allows your program to behave very differently depending on conditions, like what a user inputs. For example, you could write a script that would act one way toward you and a different way toward everyone else. Here's the basic form of an If instruction:

```
if (some condition is true)
{
   do something;
   do something;
   do something;
}
```

The important parts of this structure are:
- It starts with the word "**if**" (or upper case "**IF**" if you prefer).
- There is a numerical condition in parentheses that is either true or false.
- There is a set of instructions that should be executed if the condition is true. These instructions go between curly brackets.

Remember, spacing is only there to make the script more legible. I prefer to indent the instructions between the curly brackets. But you can put the entire if instruction on one line if you want. There is another instruction, else, which is quite the opposite of if. If you place an else directly after the closing bracket of an if instruction, the instructions between the else's curly brackets are executed only if those of the if instruction's are not executed. Here's an example of an if and else instruction in action:

```
function show_wdl()
{
   msg.visible = on;
   if (key_shift == 0) { // SHIFT not pressed?
      msg.string = "i've pressed the p key!";
   } else {
      msg.string = "i've pressed shift + p!";
   }
}

on_p = show_wdl;
```

Try now holding the **[Shift]** key down while pressing **[P]**. It's a different message that is displayed depending on whether you are holding **[Shift]** or not. Guess what **`key_shift`** is? Right: A predefined variable which is non-zero as long as the **[Shift]** key is pressed. The "**`==`**" condition in the parantheses is true if the value of **`key_shift`** is equal to **`0`**.

We have entered the strings here directly, instead of defining them before. Also note that the condition is two equal signs. This is one of those things that everyone messes up initially. If you put one equal sign instead of two, it won't work. Other typical conditions are:

`(var_1 > var_2)` is true if var_1 is greater than var_2
`(var_1 < var_2)` is true if var_1 is less than var_2
`(var_2 <= var_2)` is true if var_1 is less than or equal to var_2
`(var_1 != var_2)` is true if var_1 does not equal var_2

Two ways to make your conditions fancier. If you want both of two things to be true before running the instructions in the curly brackets, you can do this:

```
if ((var_1 > 18) && (var_1 < 21))
{
   do something;
}
```

Notice the two ampersands. That's how you say "and" in WDL. Notice also that the whole clause, including the two sub-parts and the ampersands must be enclosed in parentheses.

If you want at least either one of two things to be true for running the instructions in the curly brackets, do this:

```
if ((var_1 == 7) || (var_2 == 13))
{
        do something;
}
```

Ok, now it's time to do a little review of the things we covered so far. If you feel like you didn't learn one of these things, go back and hunt for it:

- **`//`** and **`/* */`** are used for comments. Comment your scripts frequently.

- Variables can hold numbers. There are a few restrictions and rules of thumb to keep in mind when naming variables.
- Strings can hold sequences of characters.
- Functions are for doing or changing something, and consist of instructions.
- Instructions and statements end in a semicolon.
- Use if-else clauses to make your WDL functions behave differently depending on conditions.

Congratulations if you made it through all that stuff. It was a lot to learn. We looked into variables, strings, functions, and if-else clauses, which in some form are parts of all programming languages. Now it's time to learn the rest of the WDL syntax. There is only one major aspect of WDL syntax that we have yet to cover: loops. Let's start with some more complicated WDL functions, and introduce loops.

Loops

Sometimes you want to do the same thing more than once. Let's say, for example, that you wanted to get a secret code from somebody and you wanted to keep asking until he gave you the right code. If you just wanted to give him two tries, you could do something like this:

```
var the_code = 12345;     // secret code number
var entered_number = 0;
string entry_line = "                                ";   // just a long empty string

function check_code()
{
   msg.visible = on;
   msg.string = "please enter number..."; // display this on screen
   waitt(8);              // wait 8 ticks (half a second)

   msg.string = entry_line;
   inkey(entry_line);     // requests keyboard input into a string
   entered_number = str_to_num(entry_line);   // converts input from a string to a number

   if (entered_number != the_code)
   {
      msg.string = "wrong - please enter number again...";
      waitt(8);

      msg.string = entry_line;
      inkey(entry_line);     // again, user input
      entered_number = str_to_num(entry_line);   // convert entry_line to entered_number

      if (entered_number != the_code)
      {
        msg.string = "wrong again!";
        return;                     // two tries, now terminate the function
      } else {
        msg.string = "wow! you've guessed it!"; // display "right" message
        return;
      }
   } else {
      msg.string = "wow! you've guessed it!";
      return;
   }
}

on_p = check_code;
```

This is a lengthy and somewhat ugly function! However, you've learned some new instructions here. **waitt()** just waits for the given number of 1/16 seconds, and then continues the function. **inkey()** produces a blinking cursor and lets the user enter characters into the given string until he presses **[Enter]**. **str_to_num** converts a numerical content of a string – "123", for example - to the corresponding number. We have to do this because we can't compare a string with a number in the `if` condition – this would be like comparing apples with oranges. **return** just terminates the function.

As you see, this function won't work if you just want to keep asking until they get the code number right. And it's pretty ugly already - imagine if you wanted to ask four times instead of just two! You'd have four levels of if-else clauses, which is never a good thing.

The best way to do similar things more than once is to use a **loop**. In this case, you can use a loop to keep asking for the secret code until the person gives up. Here's an example of a while loop in action.

```
function check_code()
{
   msg.visible = on;
   msg.string = "please enter number...";  // display this on screen

   while (entered_number != the_code)
   {
      waitt(8);           // wait 8 ticks (half a second)
      msg.string = entry_line;
      inkey(entry_line);  // requests user input into a string
      entered_number = str_to_num(entry_line);   // converts input from a string to a number
      msg.string = "wrong - please enter number again...";
   }
   msg.string = "wow! you've guessed it!"; // display "right" message
}
```

This is a **while** loop, which really simplifies our function. While loops come in this general form:

```
while (some test is true)
{
   do the stuff inside the curly braces
}
```

So the above lines say, "While the answer isn't equal to the code, continue to get user input." The loop will keep executing the instructions inside the curly brackets until the test is false. In this case, the test will only be false when the characters the user enters is the same as the code number (that is, "12345"). The test is true if the number is wrong – that's what "**!=**" does!

Because **entered_number** is given a value by the **str_to_num** instruction inside the while loop, it will have no value the first time we hit the loop. Thus we have defined it early with an initial value of **0**, which is certainly not the code number. Although looping indefinitely is often useful, loops are more commonly used to execute a set of instructions a specific number of times. Here's another example of a while loop that shows how to do this.

We want to write a program that lets you enter a number, and then prints as many x's on the screen. Let's go over this. First, define some new strings and variables:

```
var counter = 0;
var entered_number = 0;
string empty_string = "";  // a string with nothing at all
string entry_line = "                                  ";   // a long empty string
string result_string = "                                                                    ";
// a VERY long empty string, but don't forget, the maximum string size
```

```
// is given by its initial length

function count_x()
{
   msg.visible = on;
   msg.string = "please enter number..."; // display "Please enter number..." on screen
   waitt(8);                              // wait 0.5 seconds, then continue

   str_cpy(entry_line,empty_string);   // clear the input line from any previous input
   msg.string = entry_line;
   inkey(entry_line);                  // requests user input into a string
   entered_number = str_to_num(entry_line);   // convert input to a number

   str_cpy(result_string,empty_string);   // clear the result_string
   while (counter < entered_number)  {
      str_cat(result_string,"x");      // add "x" to the result string
      counter = counter + 1;           // increase loop
   }
   msg.string = result_string;
}

on_p = count_x;
```

This says, "while the variable **counter** is less than the requested number of the row of x's, add another x to the line and then add one to the value of **counter**." **str_cpy** copies a second string into the first, while **str_cat** adds a second string to the end of the first. This loop will keep adding an "x" to the **result_string** and adding one to the value of **counter** until **counter** is no longer less than the requested number.

The increasing of a loop variable is so common that programmers have developed a shortcut. Using the shortcut, the while loop could have been written like this:

```
while (counter < entered_number)  {
   str_cat(result_string,"x");   // add "x" to the result string
   counter += 1;             // this was counter = counter + 1;
}
```

The third line, **counter += 1,** says "add 1 to myself." If you have **a_number = 5**, and you write **a_number += 3,** it's just like writing **a_number = a_number + 3**. You can use **-=, *=,** and **/=** the same way. Programmers are lazy; they're always coming up with shortcuts like this.

Try the above function. Well, it seems to work – but only the first time. If we press **[P]** key a second time, there are less 'x' displayed than we've entered – or no 'x' at all. What the hell is happening here?

Debugging

If you've written a function, you'll sometimes notice that it doesn't work, or behaves differently than expected. And often you even don't know why. You can only think of two reasons:

1) Sometimes the WDL language just doesn't work.
2) I'm too stupid for game programming.

The WDL compiler is a well tested out part of GameStudio, so normally you can be confident that 1) is not the case. And you're not too stupid – bugs happen to the most experienced programmers all the time. In fact, it's quite normal that a function doesn't work at first. And it's also quite normal that even a top programmer won't find the reason by just staring at the WDL code.

Professionals **debug** their code. Debugging is no big secret. It just means here executing the code not as a whole, but line by line, and examining in detail what's happening, in order to find possible bugs. The engine has a built-in WDL debugger. You can activate it by just adding the following WDL instruction at the beginning of the function to examine:

```
function count_x()
{
   breakpoint;         // start debugging here
   msg.visible = on;
   msg.pos_y = 40;     // place the msg text 40 pixels downwards
   msg.string = "please enter number...";
   . . .
```

The additional **msg.pos_y** line is only for changing the screen position of the **msg** text object. Otherwise the **msg** text would be displayed over the debugging output, and make it unreadable. If we now start the level and press **[P]**, the game freezes, we hear a short sound and see the following line on screen:

```
       <= msg.visible = on
```

This is the next instruction after the breakpoint. Now press **[Space].** The line scrolls upwards, and we see something like

```
16.000 <= msg.visible = on
       <= msg.pos_y = 40
```

We have now executed one instruction. The 16.000 is an internal result code for setting the visible flag, and shouldn't bother us at the moment. Next **[Space]**:

```
40.000 <= msg.pos_y = 40
       <= msg.string = @2
```

The 40.000 is the result of setting **msg.pos_y** to 40, and the strange **@2** in the line below is the internal representation of a directly entered string. Press **[Space]** some more times, thus single stepping through the function line by line until you get

```
 0.000 <= inkey(entry_line)
       <= entered_number = str_to_num(entry_line)
```

Now, with the blinking cursor below, enter a "**2**" and press **[Enter]**. The next **[Space]** gets

```
 2.000 <= entered_number = str_to_num(entry_line)
       <= str_cpy(result_string,empty_string)
```

As we can see, the **str_to_num** instruction has converted our string to the number 2, as expected. For numerical expressions and for some instructions, the result code simply shows the resulting number. If we press **[Space]** two more times, we can see another meaningful result code:

```
 1.000 <= (counter < entered_number)
       <= str_cat(result_string,@4)
```

The result code for an **if** or **while** condition (the **if** or **while** itself is not displayed) is 1.000 for true, and 0.000 for false. Therefore we can see that the while condition here was fulfilled, and we're now stepping into the loop.

Thus by stepping instruction for instruction through our function, we are getting a perfect timeline of what happens when a person chooses two x's at the prompt:

First pass
- **result_string = ""** (because we copied "" into it by **str_cpy**)
- **counter=0** (because we defined it with initial value 0)
- **entered_number = 2** (because that's what the user asked for)
- 0 is less than 2 so
- "x" is added to **result_string**, so now **result_string = "x"**
- counter += 1, so now **counter** = **1**

Back into the loop: Second pass
- **counter = 1**
- **entered_number = 2**
- **result_string = "x"**
- 1 is less than 2 so
- "x" is added to **result_string**, so now **result_string** = "xx"
- counter += 1, so now **counter** = **2**

Back into the loop: Third pass
- **counter = 2**
- **entered_number = 2**
- **result_string = "xx"**
- 2 is NOT less than 2 so
- fall out of the loop and do what follows

And what follows is:
- **msg.string = result_string;**
- Displays the resulting "xx" on the screen.

Ok – so far so good. We have now finished debugging our function. If we would continue to press **[Space],** we'd now step into other functions that may run in our level. We don't want to do that, so by **[Ctrl-Space]** we now quit debugging mode and return to the normal run mode.

The function behaved as expected, but we know that the problem only arises at the second attempt. So press **[P]** again to step into it the second time. At entering the number, we now enter "**3**". We would now expect three loops. However what we're getting when stepping through the first loop is:

```
3.000 <= counter += 1;
      <= }
```

That's it - increasing the **counter** gives 3 already even in the first loop! As a consequence, only one loop is performed and only one 'x' is produced. And now we are able to find the reason: The **counter** variable still has its value of 2 from the last function execution. Now as we've finally found the bug, correcting is easy by adding

```
counter = 0;
```

at the beginning of the function. Don't forget such details when programming a loop!

Use the debugger as much as possible. It's a great way to learn how your functions work and how WDL works at all. Professionals usually debug their code even if it seems to work – this way they can detect 'hidden bugs' even before they occur under certain circumstances. If you're really

stuck with a problem in your code, you'll find in **Appendix A** a collection of the most common WDL traps.

Having learned about debugging, you can now call yourself a junior computer programmer. For the next days, we're going to leave the dry-ish world of WDL syntax and enter the real game play. Here we will learn about how you can use WDL scripts to define the behaviour of the player, of actors, or of the user interface in powerful and interesting ways.

Monday: Doors and Keys

It always is impressive to have things that move as part of our world - like clockworks set in motion, secret portals opening, or having the player suddenly find water streaming in - at the touch of a button! Therefore in this chapter of the tutorial we are going to examine how to set in motion parts of our world in different ways. The easiest and most frequent example of movement in our world is a door opening.

We know from the WED tutorial how to do that: we just have to attach the predefined **door** action to a map entity. Then, if the player presses the **[Space]** key near the door, it will rotate horizontally around its centre. But how can this be achieved? Let's have a closer look onto entity actions.

Actions

Entities are the moving objects of our game world – like doors, actors, enemies, or the player himself. Their movement is done by functions – named **action**s here - assigned to them. And those actions normally consist of one or more huge loops with a **wait(1)** instruction at the end.

Here is an example for a loop that just lets an entity rotate by itself. Maybe you've encountered a similar example in the WED tutorial:

```
action entity_rotate
{
   while (1) {
   my.pan+= 3;
   wait(1);
   }
}
```

This is a description for "rotate me permanently with a speed of 3 degrees per frame". We can assign this action to an entity by WED. The only difference between a function and an action is that the latter is defined by the word **action**, appears in WED's action list, and does not have the parantheses after the name. Each entity's action is started once at game start. We have here simply a **1** as the while condition. **1** is considered to be always true. So **while (1) { ... }** will repeat the instructions forever. This is called an infinite loop.

Inside this loop, the value **3** is added to **my.pan**. **My** is an object like the **msg** object we're already familiar with. However, **my** is not a text object, it is an entity object. More precisely, **my** is a **synonym** for that entity the action was assigned to. A synonym is a temporary name for something. As in our language "me" or "my" refers to person who's speaking, so in WDL **my** refers to the entity who's running that action. If you'd assigned this action not to an entity, but, for example, to a user interface button, you'd receive an error message ("**empty synonym**") – because **my** is exclusively for entities, and makes no sense for a button!

Pan is the entity's horizontal angle, in degrees. This angle will permanently increase by **3.** In other words: the entity will rotate. It does not matter if the **pan** value exceeds 360 degrees – the engine can handle that.

The last instruction, **wait(1**), suspends the action for one frame cycle, before repeating the loop. So we have a rotation of three degrees per frame cycle. We have already learned **waitt()** (with two t's) that waits for 1/16 seconds, while **wait()** waits for frame cycles. On a low-end PC, running at just around 16 fps, these both are almost the same. Bear in mind that everything outside the function - rendering, global variable changing and execution of other functions - will be

performed during that little **wait()** pause! Therefore each infinite loop must always contain a **wait()** or **waitt()**. Otherwise the loop will nothing else allow to happen outside itself, and will freeze your PC (resp. produce an "**infinite loop**" error message).

The First Movement

Start WED, open one of your levels, and create a fresh new script file (**Map Properties -> Script -> New**). But this time we are not going to make use of the prefabricated functions. This time we are going to write our own one. Close WED again. Use a text editor to open your script file **name.wdl**, while ***name*** is the name of your map. Now move to the very end of the script file, and type there your first entity action:

```
action door1
{
   while (1) {
      play_entsound(my,open_snd,66);
      while (my.pan < 90) {
         my.pan += 3*time;      // rotate counterclockwise
         wait(1);
      }
      waitt(16);
      play_entsound(my,close_snd,66);
      while (my.pan > 0) {
         my.pan -= 3*time;      // rotate clockwise
         wait(1);
      }
      waitt(16);
   }
}
```

Ok, let's see what we have done here. We have defined an **action** named **door1** – an action that can be attached to a door entity, which we'll do now. For creating a door entity yourself, open a new map named **mydoor.wmp**, place a single cube, size it to door-shape, give it a wood texture, move it so that it is standing upright and the map origin is placed at its hinge position, and finally **build** it with the **Entity Map** option checked. If you don't want to do all this stuff, just use the prefabricated **porta.wmb** from the office level.

You now have a map entity which you can place somewhere into your test level through WED. Do it. Then, with the door entity in WED still selected, open **Entity Properties**, and click the action field. If you have done everything right, you can now choose your new **door1** action from the list. Rebuild the level using **Update Entities** – that builds much faster in case you haven't changed anything except entities. Now start it by **Run Level**. If you don't have a player in your level yet, you can switch on the direct camera movement mode by pressing the **[0]** key. If you've done everything right, you can now observe a door permanently opening and closing, as if by magic!

As you remember, **my** is the synonym for the entity to that the action is attached – our door. The first instruction is **while (1)**, which means: Repeat forever all instructions between the following pair of curly brackets. We have to repeat the instructions because we want this action to be performed permanently during game play. Otherwise, it would only be performed once at game start, and then end. And the door would maybe move a little step, and then stand still forever.

Within the never-ending loop, we let the door play an opening sound through the **play_entsound()** instruction, and continue with another *inner loop* (you can 'nest' loops as you see). This inner loop will increase the door's angle as long as it is still below 90 degrees. The amount to increase the angle is multiplied by **time**, the duration of the frame cycle, in order to get an angle increment out of the opening speed, independent of the frame rate.

Then the door will stand still for one second. We are using the **waitt()** instruction here, which waits a certain time, unlike **wait()**, which waits for a number of frame cycles. After the pause we let the door rotate back until its angle has reached zero, and then, after another pause, everything begins again.

We are noticing some shortcomings of our action. The door always rotates between 0 and 90 degrees, i.e. east and north, regardless of its initial orientation. And it will 'overshoot' its final position by some degrees, due to the fact that the last increment may add to a value above 90, or below zero. We have to correct this:

```
action door1
{
   my.skill25 = 0;  // degree counter, independent of initial angle
   while (1) {
      play_entsound(my,open_snd,66);
      while (my.skill25 < 90) {
         my.pan -= 3*time;      // rotate clockwise
         my.skill25 += 3*time;
         wait(1);
      }
      my.pan += my.skill25-90;  // correct the overshoot
      my.skill25 = 90;
      waitt(16);
      play_entsound(my,close_snd,66);
      while (my.skill25 > 0) {
         my.pan += 3*time;      // rotate counterclockwise
         my.skill25 -= 3*time;
         wait(1);
      }
      my.pan += my.skill25;  // correct the overshoot
      my.skill25 = 0;
      waitt(16);
   }
}
```

The door is now opening clockwise, and closing counterclockwise. An entity skill, **skill25**, is used for counting the opening angle, so the door now opens independently of its initial **pan** value. Entities have 40 such internal variables that may be used in actions. Why can't we just define a variable through a **var** statement for the counter? Because we may have more than one door in our level. Variables that we define are 'globally' shared by all actions and functions.

Ok, the door is moving now. But how can we now achieve that it reacts on us, the User?

User interaction

The door next to the camera view shall open if we press a key on the keyboard. The **scan** instruction is made for that. **Scan** will trigger an entities' **event** function, if it is within the scan cone. Event functions control an entities' reaction on something happening. First let's write an action that performs a **scan**, and assign that to a key:

```
var indicator = 0;
var my_pos[3];
var my_angle[3];

function scan_me()
{
   my_pos.x = camera.x;
   my_pos.y = camera.y;
```

```
   my_pos.z = camera.z;
   my_angle.pan = camera.pan;
   my_angle.tilt = camera.tilt;
   temp.pan = 120;
   temp.tilt = 180;
   temp.z = 200;   // scan range - 200 quants
   indicator = 1;  // this is for opening
   scan(my_pos,my_angle,temp);
}
```

The whole function just sets the vectors that are needed as the three parameters for the **scan** instruction. You should read the **scan** description in the reference manual to understand what we're doing. We have defined two **vectors** for storing a XYZ position and a three dimensional angle. A vector is just a variable that contains not one but three numbers, indicated by the **[3]** after the variable name. The three numbers can be accessed as .X, .Y, .Z, or alternatively as .**pan**, .**tilt** and .**roll** parameters of the vector. **Temp** is a predefined vector that we use for intermediate results – here for the width and range of the scan cone. We have set the centre of the scan cone to the camera position, its direction to the camera direction, its range to 200 quants, and its width to almost a semicircle.

The **indicator** variable is set to **1** as a signal for the entity whose event function may get triggered by the scan instruction. If indicator was set to **1** at that time, the entity knows that this scan was intended for operating a door. Scan instructions could be used for many other purposes also, for explosions by example, and we have to distinguish here to prevent that a detonating grenade suddenly opens all doors in the level.

How can our door detect the **scan** instruction?

```
define _counter skill25;  // use a meaningful name for SKILL25

function door_event()
{
   if (indicator != 1) { return; }  // must be right kind of scan
   if (my._counter <= 0) {          // if door was closed, open it
      play_entsound(my,open_snd,66);
      while (my._counter < 90) {
         my.pan -= 3*time;      // rotate clockwise
         my._counter += 3*time;
         wait(1);
      }
      my.pan += my._counter-90; // correct the overshoot
      my._counter = 90;
   } else {                     // otherwise close it
      play_entsound(my,close_snd,66);
      while (my._counter > 0) {
         my.pan += 3*time;      // rotate counterclockwise
         my._counter -= 3*time;
         wait(1);
      }
      my.pan += my._counter;    // correct the overshoot
      my._counter = 0;
   }
}

action door1
{
   my.event = door_event;
   my.enable_scan = on;   // make door sensitive for scanning
   my._counter = 0;       // degree counter, independent of initial angle
}
```

At first, we have used the **define** statement to give **skill25** the name **_counter** that describes what it's used for. The **template** scripts make heavy use of this way to name entity skills.

The door action is now split in two, an initialising action and an event function. The door is made sensitive for **scan** instructions, and the event function is set. This action first checks whether it was an opening **scan**, or anything else. If it was the right one, it will check whether it was already closed or open, and then open or close respectively. All we have to do is now attach the **scan_me** function to a key:

```
on_space = scan_me;
```

Now we have to walk within 200 quants of the door, and then press the **[Space]** button.

Ready for more? What about a key item we'll need to unlock the door? Let the key set a certain variable to **1** if we have picked it up, and let the door check that variable before opening. First we have to define an action for a model entity to become a key item:

```
sound key_fetch = <beamer.wav>;
var key1 = 0;

function key_pickup()
{
   if (indicator != 1) { return; }  // must be right kind of scan
   key1 = 1;
   play_sound(key_fetch,50);
   remove(my);
}

action key
{
   my.event = key_pickup;
   my.enable_scan = on;   // pick up by pressing SPACE..
}
```

As soon as we come close to the key and press the **[Space]** button, it will disappear, the sound will play, and the **key1** variable will be set to **1**. This way items can be picked up and influence the game. The door now has to check the **key1** variable upon opening:

```
function door_event()
{
   if (indicator != 1) { return; }  // must be right kind of scan
   if (key1 != 1) { return; }    // key must have been picked up already
   ...// and so on...
```

By the way, the key entity of course could also be shaped like a button or lever, and change its **frame** or **skin** parameter to show activation, instead of disappearing by **remove**. Using more variables than the one **key1** skill, we can handle different keys that open different doors.

And how can we achieve that the door opens already if an entity just comes near to it? Piece of cake: Either the player entity could perform a repeated **scan** instruction (once per second – not more often, **scan** is slow). Or we could use **event_trigger** instead of, or additional to the **scan** event. We'll leave that as an exercise to the reader… until Tuesday.

Tuesday: Game Physics

To give you an example for the simple physics used for objects in a game, we are now going to create one of the most common functions: the movement of the player entity.

Sure, you don't have to do that, because you can use the prefabricated **player_move** action with its huge set of options. But if you want to become a master of WDL, at one point you may have to write your own movement functions for special entities of your game – and here you will encounter game physics. It is going to get a bit mathematical, but don't be afraid: the four basic types of arithmetic will be sufficient. We don't introduce real physics here. We use a simplified form of physics, especially suited for a real time game where hundreds of actors perform movement arithmetic which have to be as fast and simple as possible. By the end of the day you are going to be able to infuse entities with any kind of movement behaviour whatsoever through your own functions. And even better, you don't have to wait until Tuesday to start this lesson…

Open your level WDL file. This time we are not going to make use of the prefabricated player movement functions of the **movement.wdl**. This time we are going to write our own one.

```
action move_me
{
   while (1)
   {
      my.x += 10 * key_force.x;
      my.y += 10 * key_force.y;
      move_view();
      wait(1);
   }
}
```

The first two instructions within the **while** loop assign **my.x** and **my.y** new values. **My** is the entity to which we assign the action. **My.x** and **my.y** are the X and Y coordinates of it's position, using the same coordinate system we used when building the map through WED. Changing these coordinates means moving, or more precisely, teleporting the entity. We are changing the position by adding an arithmetic expression; the adding is done by the **+=**, and the expression is **10*key_force.x** and **10*key_force.y**. In other words, "multiply **key_force** by 10 and add the result onto the entities's position".

The X and Y coordinates of the predefined vector variable **key_force** contain numeric values corresponding to our input through the cursor keys. As long as you press the **[Up]** key, **key_force.y** aquires the value of **1**, and as long as you press the **[Down]** key, the value will be **-1**. So by pressing one of the cursor keys, ten times that value will be repeatedly added to, or subtracted from the entities' position, thus moving the entity along the X or Y direction of the coordinate system.

The next instruction will execute a separate predefined function - **move_view** – to move the camera view to the entities' position. We won't analyze the **move_view** function here, but just be grateful that it exists – it is located in the **movement.wdl template** file. The last instruction, **wait(1)**, suspends this action until the next frame cycle, where all instructions will be repeated again due to the **while (1)**. If you had forgotten that **wait()** instruction, the action would never be interrupted, and thus the frame cycle would never end. That would be bad.

Enough of theory. Let's try out what we have written. If WED is already open, select **Map Properties**, and re-select the same script file – we have to do that because we have added a new entity action. Place an entity of a player's size – **guard.mdl** for example – into the level, open **Entity Properties**, and click the action field. If you have done everything right, you can now

choose your new **move_me** action from the list. Rebuild the level using **Update Entities** – that builds much faster in case you haven't changed anything except entities. Now select **Run Level**, and try the cursor keys to walk.

Hmm. Well, we must admit that your first movement action doesn't behave as well as the prefabricated one. What happens is that the movement does not follow the line of player's sight – it follows straight the X- and Y-axes of the coordinate system. In fact, you can't rotate the player at all. No wonder, you haven't provided any instruction for that. And even worse, the player won't stop at walls, he will move straight through them.

Let's fix that at first. In order to get collision detection into your entity movement, you must not change the entity coordinates directly – that would teleport the entity to its new position - but use the **move** instruction for a continuous movement over a given distance:

```
var dist[3];

action move_me
{
   while (1)
   {
      dist.x = 10 * key_force.x;
      dist.y = 10 * key_force.y;
      dist.z = 0;
      move(my,nullvector,dist);
      move_view();
      wait(1);
   }
}
```

We have defined a **vector**, **dist**, to store the distance to cover. A vector is just a variable that contains three numbers, which are used here for the X, Y, and Z coordinates. We are setting its Z coordinate to zero to prevent the player from suddenly flying into the air. The **move my**… instruction changes the player position the same way as if we had written:

```
   my.x += dist.x;
   my.y += dist.y;
   my.z += dist.z;
```

The only difference is that through **move**, the **my** entity – our player – performs a collision detection over the distance given by `dist`. **Move** takes two vector distances, of whom we've set the first one to zero using the predefined **nullvector**. The first distance is rotated according to the entities' angles, the second one (used here) is not.

Using the first rotated distance, instead of the second unrotated one, would give us the opportunity to rotate the movement direction into the player's line of sight, and use **key_force.x** to change his angle:

```
action move_me
{
   while (1)
   {
      my.pan -= 10 * key_force.x;
      dist.x = 10 * key_force.y;
      dist.y = 0;
      dist.z = 0;
      move(my,dist,nullvector);
      move_view();
      wait(1);
   }
```

```
}
```

Please note that we now use a rotated distance for **move**, while the unrotated distance is set to zero by using the **nullvector**. **dist.x** - which gives the direction ahead – is now changed by **key_force.y**.

Now we are able to make the player rotate and move in every direction. But the movement still appears to be jerky and unnatural. And our speed is dependent on the frame rate – on faster PCs the MOVE instruction is performed more often, so the player covers a larger distance in the same time. That shouldn't be. We shall now try to figure a formula that achieves smooth 'natural' movements.

Acceleration, Inertia, Friction

Let's suppose that our player is moving straight ahead at a constant speed. The distance he travels increases, according to his speed, with the time:

$$ds = v * dt$$

v = Speed in quants per tick
ds = Distance in quants
dt = Time in ticks

In our virtual world - as you will remember - distances are being measured in **quants** (= around 1 inch) and time is being measured in **ticks** (= around 1/16 sec). Thus the unit of measure for speed is quants per tick.

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 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 formulae:

$$dv = a * dt$$
$$a = F / m$$

dv = Change of speed
a = Change of speed per tick (acceleration)
F = 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 keyboard and joystick movement. In our simplified game environment this force shall be proportional to the player's mass: The bigger a guy is, the more force he can apply to accelerate himself.

$$P = p * m$$

The second force we shall call **drift**. It can be a current that carries the player in a certain direction, or the gravity, pulling him down. Drift forces may vary from place to place. In our game world, the amount of the drift also is proportional to the player's mass. This is obvious for gravity, but

can also be approximated for normal drifts who apply the more force to an object the more volume it has.

$$D = d * m$$

The third force having an effect on the player is **friction**. This force continually tries to slow him down. Unlike in realistic physics, where the decelerating force consists of friction and attenuation parts, we use an artificial force that increases with the player's mass (pressure onto the ground) and speed:

$$R = - f * m * v$$

R = friction force
f = friction factor
v = speed
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. If he is airborne, the air friction is almost zero. The negative indicator is supposed to show that this force counters the velocity. The mass m is also part of the equation, as in our game world the player will encounter greater friction and put bigger stress on the surface if he weighs more.

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

$$dv = a * dt = (P + D + R) / m * dt = (p + d - f * v) * dt$$

Thus dv must be added to the velocity each frame. p, d, and f are the propelling, drift and friction factors here. dt is the time during which the acceleration has changed the velocity – it's the time between two frame cycles here, because we are calculating a new velocity each frame cycle. As you can see, by persuading all our forces to be mass proportional, the mass is removed out of the equation, having no influence onto the movement in actor physics.

We could translate the last formula into a corresponding function (there is no drift yet):

```
var force[3];
var dist[3];

action move_me
{
   while (1) {
      force.pan = -10 * key_force.x;   // calculate rotation force
      my.skill14 = time*force.pan + max(1-time*0.7,0)*my.skill14;// rotation speed
      my.pan += time * my.skill14;  // rotate the player

      force.x = 10 * key_force.y;   // calculate translation force
      my.skill11 = time*force.x + max(1-time*0.7,0)*my.skill11; // calculate speed
      dist.x = time * my.skill11;   // distance to cover
      dist.y = 0;
      dist.z = 0;
      move(my,dist,nullvector);     // move the player

      move_view();                  // move the camera
      wait(1);
   }
}
```

Our force factors are 10 here, our friction factors 0.7, both for rotation and translation. We must use entity skills for the speeds, because they have to be preserved together with the entity; the previous speed is needed for our formula. **Skill14** stores the angular speed and **skill11** the speed ahead.

Time is the *dt* of the formula, the time of the last frame cycle. We're using the now mathematically correct formula for calculating the distance - therefore the player now moves with equal speed on each PC, almost independent of the frame rate! Note that we have converted our original, theoretical formula that gives the speed change per frame cycle

$$v \rightarrow v + dv$$
$$v \rightarrow v + (p - f * v) * dt$$

which would be expressed in WDL

```
my.skill11 = my.skill11 + (force.x - 0.7 * my.skill11) * time;
```

and transformed via the following steps

```
my.skill11 = my.skill11 + time * force.x - time * 0.7 * my.skill11;

my.skill11 = time * force.x + (1 - time * 0.7) * my.skill11;
```

to finally, somewhat more complicated

```
my.skill11 = time * force.x + max(1 - time * 0.7,0) * my.skill11;
```

Why this, and what for the **max**? **Max(a,b)** delivers the bigger number of **a** or **b**. Through **max(1-time*0.7,0)** we're limiting the result to positive values. We have to do that, because otherwise on very low frame rates – and high **time** values - **time*0.7** can be bigger than **1**, giving a negative result. This would reverse the speed – the player would move backwards! Such subtle differences between the computer simulated and the real reality must always be considered. (Normally, even an experienced game programmer would not notice such an effect before an action behaves differently than expected).

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 what happens if the player encounters a stairway, or an abyss?

Falling down

While we are moving our player horizontally by pressing keys, his vertical movement shall be caused by his environment. If he stands on solid ground, he must not perform any vertical movement at all. But if he is suddenly floating in the air – this may be the case if he were to walk over the edge into an abyss – he must fall down. For this we have to determine his height above the ground. We are using the **trace()** instruction for that.

```
action move_me
{
   while (1) {
      force.pan = -10 * key_force.x;   // calculate rotation force
      my.skill14 = time*force.pan + max(1-time*0.7,0)*my.skill14;// calculate rotation speed
      my.pan += time * my.skill14;  // rotate the player
```

```
      my.skill11 = time*force.x + max(1-time*0.7,0)*my.skill11; // calculate speed ahead
      dist.x = time * my.skill11;   // distance ahead
      dist.y = 0;

      vec_set(temp,my.x);
      temp.z -= 4000;               // calculate a position 4000 quants below the player
      // set a trace mode for using the player's hull, and detecting map entities and level surfaces only
      trace_mode = ignore_me+ignore_sprites+ignore_models+use_box;
      dist.z -= trace(my.x,temp);   // subtract vertical distance to ground

      move(my,dist,nullvector); // move the player
      move_view();              // move the camera
      wait(1);
   }
}
```

Trace() returns the distance to the next obstacle along a ray between two positions. We are using the player centre (**my.x**) as the first position, and are setting the **temp** vector to a second position some thousand quants below. Through the **use_box** trace mode we are giving the trace ray a 'thickness' of the player model's hull – which normally has 32 quants diameter – and also return the distance not to the player's centre, but to his bounding box. That's his feet in this case, because we are tracing vertically downwards. So **trace()** calculates the distance between the player's feet and the first level or map surface below. If his feet are below the floor level, **trace()** logically returns a negative value.

We have taken the easy way by simply moving the player vertically by exactly the amount of distance between his feet and the ground. Thus **dist.z** ist set to the vertical distance to cover. We are now able to walk regions of differing surface heights with our player, e.g. stairs. Please note that for simplicity we used the relative distance to move him in Z direction. Normally the absolute distance had to be used here, but as long as the player is not tilted, it doesn't matter.

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. **trace()** 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 - i.e. **trace()** returns a value below or equal to **0**. 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.

Depending on the player entities' center, the player's feet are able to even penetrate 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.

```
var friction;

action move_me
{
   while (1) {
      force.pan = -10 * key_force.x;   // calculate rotation force
      my.skill14 = time*force.pan + max(1-time*0.7,0)*my.skill14;// rotation speed
```

```
      my.pan += time * my.skill14;  // rotate the player

      vec_set(temp,my.x);
      temp.z -= 4000;               // calculate a position 4000 quants below the player
      // set a trace mode for using the player's hull, and detecting map entities and level surfaces only
      trace_mode = ignore_me+ignore_sprites+ignore_models+use_box;
      result = trace(my.x,temp);// subtract vertical distance to ground
      if (result > 5) {         // in the air?
         force.x = 0;           // no pushing force
         force.y = 0;
         force.z = -5;       // gravity force
         friction = 0.1;        // air friction
      } else {                  // on or near ground
         force.x = 10 * key_force.y;  // translation force
         force.y = 0;
         force.z = -0.5 * result;  // ground elasticity
         friction = 0.7;        // ground friction
      }

      my.skill11 = time*force.x + max(1-time*friction,0)*my.skill11; // calculate ahead speed
      my.skill13 = time*force.z + max(1-time*friction,0)*my.skill13; // calculate vertical speed
      dist.x = time * my.skill11;  // distance ahead
      dist.y = 0;
      dist.z = time * my.skill13;  // distance down

      move(my,dist,nullvector); // move the player
      move_view();              // move the camera
      wait(1);
   }
}
```

Through the **`if`** instruction we are able to distinguish whether the player is airborne or within 5 quants of the ground. In the former case he won't be pushed by the cursor keys, but drawn down by a constant gravity force. In the latter case a resilient force, being proportional to the (negative) depth of immersion (**`result`** of **`trace()`**), tries to push him out of the ground. We are using one more entity skill, **`skill13`**, for storing the vertical speed.

Well… now we could add more cool features to our action. We could use more keys to let him look up and down, or jump, or duck. Our script would grow larger and larger, towards the size of the prefabricated **`player_move`** action in the **`movement.wdl`**. We will now leave that to you. At least until Wednesday.

Wednesday: Artificial Intelligence

The most challenging thing at game developing games is how to define intelligent behaviour for the opponents. 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 normally make use of model entities. Model entities are those objects that are able to animate, and normally are used for actors, opponents, or monsters. Monsters wanting to be taken seriously 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.

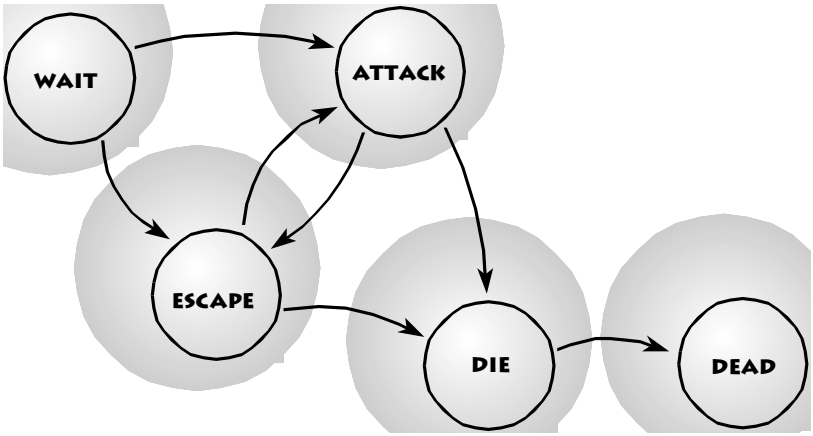
In this chapter of the tutorial you will learn to infuse your creatures with an electronic personality. We will therefor make an excursion to the theory of state machines. At the close of the chapter you will be able to create electronic beings that in their complexity are comparable to simple organisms.

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 scripting.

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 these states. The states in a way constitute the 'inner life' of the black box. For each state there exist circumstances, e.g. an external stimulus, that cause the machine to change into a different state.

For instance, an actor in a first person shooter could have the following behavioural states: **Wait**ing, **Attack**ing, **Escap(e)**ing, **D(ie)**ying, and **Dead**.



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

Event	Realised by
Actor spots Player.	Function with periodic `trace()` instruction and evaluating the `result`. Be careful, `trace()` is slow!
Player comes close to Actor (or vice versa).	`event_trigger`
Player and Actor touch.	`event_entity`, `event_impact`
Player has removed himself from the Actor beyond a certain distance.	Function that periodically calculates the distance to the player entity.
Actor is near an exploding object.	`event_scan`
Player hits Actor with the `trace()` instruction.	`event_shoot`
An animation cycle of the Actor model has passed.	Function that periodically compares the `frame` parameter.
A variable that is being changed by another function got a certain value.	Function that periodically compares that variable.
A certain amount of time has passed.	Function which performs a `waitt()`, or counts down an entity skill.
Random event.	Function that periodically compares an expression containing `random()`.

Each of these events may of course be combined with any other. Either the entities' main action or its **event** function is responsible for the transition between states.

Apart from states we are also able to define inner variables for the machine that may acquire different values and influence the behaviour. Our entity skills are inner variables. Their values can be made use of in a function and for example decide on the transition into another state.

The Table of States

The division of the behaviour of our machine into states and transitions allows us to keep the actor's action 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 one of the robots from the computer game „***Mission***".

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

At the start of the game the robot 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 him with his gun, the following transition depends on the robot's remaining health, i.e. the number of hits he had already been forced to take: the next state will either be **Attack**, **Escape**, or **Die**.

State	Player near	Health > 30	Health <= 30	Health <= 0	Dying finished
WAIT	ATTACK	ATTACK	ESCAPE	DIE	DEAD

To keep track of the health we shall employ one of the robot's entity skills. For better transparency we sum up the states in a **table of states**. We have defined five events for the flyer that may change his state. Three of those are depending on an internal health skill. We may now add the remaining states to the table.

State	Player near	Health > 30	Health <= 30	Health <= 0	Dying finished
WAIT	ATTACK	ATTACK	ESCAPE	DIE	DEAD
ATTACK	ATTACK	ATTACK	ESCAPE	DIE	DEAD
ESCAPE	ESCAPE	ESCAPE	ESCAPE	DIE	DEAD
DIE	DIE	DIE	DIE	DIE	DEAD
DEAD	DEAD	DEAD	DEAD	DEAD	DEAD

Our robot is a rather 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 robot always transitions to **Die** first. 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. We could write everything together into one big action, or define different functions for each state. We choose the latter approach – it offers the advantage that state functions can be re-used for different machines.

```
action my_robot
{
   my._walkframes = 1;
   my._entforce = 0.7;
   my._armor = 100;
   my._health = 100;

   my.enable_scan = on;       // sensible of explosions
   my.enable_shoot = on;      // sensible of gunshots
   my.event = myfight_event;  // handle hits

   mystate_wait();     // First state: watch for the player
}
```

From now on we will make heavy use of the predefined entity functions and variables in **movement.wdl** and **actors.wdl**. At first we are giving the number of frames for the walking, running, attacking and dying cycles which our robot, **enemy1.mdl**, consists of. All those parameters beginning with an underscore (**_**) are entity skills, which are given senseful names by the **define** keywords in the **movement.wd**l.

Now we have to define the event function, which has to handle hits, and the **mystate_wait**, **mystate_attack**, **mystate_escape**, and **mystate_die** functions, which represent the different states of the machine. For the **Dead** state we don't need a function: a dead enemy is supposed not to move anymore.

Let's start with the dying function. It's the simplest one:

```
function mystate_die()
{
```

```
   my._animdist = 0; // reset entity's animation time to zero
   while (my._animdist < 100)
   {
      ent_frame("death",my._animdist);    // set the frame from the percentage
      my._animdist += 5 * time; // calculate next percentage for death in ~1.25 seconds
      wait(1);
   }
   my.enable_scan = off;  // become insensible
   my.enable_shoot = off;
   my.event = null;       // and don't react anymore
}
```

This is just animation. **my._animdist** gives the percentage of the death animation of the model. It's first set to **0**, then permanently increased in the animation loop until it has reached 100%. The **ent_frame** instruction toggles through the animation frames that begin with the name "death". To test the action, let's add a simple **mystate_wai**t and an **event** function that does nothing but receiving hits:

```
function myfight_event()
{
   if (((event_type == event_scan) && (indicator == _explode)))
      || ((event_type == event_shoot) && (indicator == _gunfire))
   {
      if (my._armor <= 0) {  my._health -= damage; }
      else { my._armor -= damage; }
   }
}

function mystate_wait()
{
   while (1) {
      if (my._health <= 0) { mystate_die(); return; }
      wait(1);
   }
}
```

Take care to write those function **before** the **my_robot** action, because the latter needs the former. The event function will count the armour entity skill down if either an explosion, or a gunshot has hit our actor. The **indicator** and **damage** variables are set by the corresponding weapon function. As you've learned yesterday, **indicator** is used to distinguish the scan or shoot event from other scans which are used not for explosions, but for other purposes. And **damage** represents the strength of the weapon. If there is no remaining armour, the **health** will be reduced by further hits until the **mystate_wait** function initiates **dying** if health has reached zero.

Now place the robot and a weapon into your level. You may construct a new weapon, or use a **sparkgun** or a **flashgun** from **weapons.wdl.** Assign **my_robot** to the actor, walk into the level, grab the gun, and begin to fire onto the helpless robot. If you've done everything right, after about 10 sparkgun hits he will bite the dust.

Well, it's not fair to shoot an actor who can't shoot back. For that he has to detect us first. We'll use the **scan** instruction for watching. The player has set **enable_scan** if the **player_move** action is used, so the robot can use **event_detect** to see whether he has scanned the player.

```
function myfight_event()
{
   if (((event_type == event_scan) && (indicator == _explode))
      || ((event_type == event_shoot) && (indicator == _gunfire)))
   {
      my._signal = 1;  // by shooting, player also gives away his position
      if (my._armor <= 0) {  my._health -= damage; }
      else { my._armor -= damage; }
```

```
   }
   if ((event_type == event_detect) && (you == player)) {
      my._signal = 1;  }
}

function mystate_wait()
{
   while (1) {
      if (my._health <= 0) { mystate_die(); return; }
      if (my._health < 30) { mystate_escape(); return; }

   // scan for the player
      temp.pan = 180;
      temp.tilt = 180;
      temp.z = 1000;
      indicator = _watch;
      scan(my.x,my.pan,temp);
      if (my._signal == 1) { mystate_attack(); return; }
      waitt(8);
   }
}

action my_robot
{
   my._walkframes = 1;
   my._entforce = 0.7;
   my._armor = 100;
   my._health = 100;
   my._signal = 0;        // player not yet detected

   my.enable_scan = on;   // sensible of explosions
   my.enable_shoot = on;  // sensible of gunshots
   my.enable_detect = on; // sensible of the player
   my.event = myfight_event; // handle hits & detection

   mystate_wait(); // first state: watch for the player
}
```

Here we make use of an entity skill we've named **my._signal** (**_signal** is defined in the **movement.wdl**) for letting different entity actions exchange simple messages – in this case that the entity has detected the player. The **mystate_wait** function scans for the player within 1000 quants in front of the robot. We have to set the **indicator** variable to make certain that our scanning is not mistaken for an explosion or opening a door.

By changing **wait(1)** to **waitt(8)** we have reduced the loop rate within the **mystate_wait** function to only two **scan** instructions per second. **Scan** is slow, and simultaneous **scans** from hundreds of robots may otherwise reduce the frame rate. The robot will need half a second now to detect the player. This is his reaction time.

Please note that using **scan** the player will be detected through walls. To prevent this, **a trace()** instruction should follow to verify whether the player is really visible (we have omitted that here, but you'll find it in the WAR.WDL). Each entity found with **enable_scan** set will give a **detect** event, but only the player will set **my._signal** to 1, which is the signal for attacking:

```
function mystate_attack()
{
   my._animdist = 0;
   while (1) {
      if (my._health <= 0) { mystate_die(); return; }
      if (my._health < 30) { mystate_escape(); return; }
```

```
// turn towards player
      temp.x = player.x - my.x;
      temp.y = player.y - my.y;
      temp.z = player.z - my.z;
      vec_to_angle(my_angle,temp);
      actor_turn();

      ent_cycle("attack", my._animdist);
      my._animdist += 5 * time;
      if (my._animdist > 100) {
         my._animdist -= 100;
// fire at player at end of animation cycle
         shot_speed.x = 100;
         shot_speed.y = 0;
         shot_speed.z = 0;
         my_angle.pan = my.pan; // tilt is already set from vec_to_angle
         vec_rotate(shot_speed,my_angle);
// now shot_speed points ahead, towards the player
         damage = 20;
         fire_mode = 303.2;  // orange fireball, with smoke
         create(fireball,my.x,bullet_shot);
      }
      wait(1);
   }
}
```

This function contains a few 3-D calculations. When the robot has detected the player, he has to turn towards him, and begin to throw fireballs. At first the direction from the robot to the player is calculated. We are using the **temp** variable as a direction vector – that is a directed distance between two points in space. As you see, a direction vector can be calculated by simply subtracting the three vector components of the starting position from the target position. We won't explain vector arithmetics here - if you don't believe me, you are free to buy a linear algebra book.

The next step is converting that direction into angles, using the **vec_to_angle()** instruction. The **pan** value of the **my_angle** vector becomes the horizontal angle between robot and player, and the **tilt** value becomes the vertical angle between them. The predefined **actor_turn** function rotates an entity horizontally towards a target angle. We could directly set our robot's **pan** angle to **my_angle.pan**, but then he would turn unnaturally fast, and thus always face the player.

Next he plays the attacking animation. On reaching the last frame, he has to throw a fireball. We are using the **bullet_shot** function here. This function was intended for our **flashgun** weapon, but is a good example how you can re-use almost each function for different purposes. **bullet_shot** requires the **shot_speed** vector to be set to the right speed and direction. We'll do that by setting it at 100 quants towards X direction, then rotating it into the horizontal direction our robot is facing, while re-using the previously calculated **my_angle.tilt** value to let the fireball direction vertically point towards the player. **vec_rotate()** does the job.

At this point again our puzzle for the reader: The two very first lines within the attack loop, which both begin with **if (my._health ...** , must be given exactly in that order. Why?

The last state remaining, **Escape**, is piece of cake now:

```
function mystate_escape()
{
   while (1) {
      if (my._health <= 0) { mystate_die(); return; }
// turn away from player
      temp.x = my.x - player.x;
      temp.y = my.y - player.y;
      temp.z = my.z - player.z;
```

```
      vec_to_angle(my_angle,temp);
      actor_turn();
// walk away
      force = my._entforce * 4;
      actor_move();

      wait(1);
   }
}
```

The only new function which we are calling here is **actor_move**, which ist defined in the **template** scripts and does what you suppose.

Advanced State Machines

Intelligent and 'personalised' behaviour of opponents gives greater satisfaction especially with role-playing 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 primitive shoot -'em- up game. Such games quickly become boring.

How could we improve our robot? Most obvious are the possibilities of state enlargements, which 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. On attacking, he could advance towards the player, dodging sideways in order to avoid a direct hit. Some of these possibilities have been realised in the **war.wdl** file.

Improvements of the actor's **strategic** behaviour are a lot more complicated to realise, and not often found in action games. 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 **scan** instructions and variables) 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, which 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. If they see the player already attacked, they join the attack to combine forces.

But we never ought to overdo this. It's no chess. 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.

Thursday: The User Interface

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 the user interface functions that make communication possible between the *user*, sitting comfortably in front of his computer screen, and his imaginary counterpart the **player**, who is the entity 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.

As we remember, 'raw' text is defined as a sequence of characters with the keyword **`string`**:

```
string my_string = "this is a text!!";
```

The string may even 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 or 256 characters of the PC character set (ASCII grouping).

All in all the bitmap must leave room for 128 or 256 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, extraterrestrial characters or those stemming from your own imagination - may be represented. In our example the German 'umlauts' take the place of the special characters '**[**', '**|**' ,'**]**' and '**{**', '****', '**}**'. In the **template** directory a simple bitmap (**`ackfont.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 such:

```
font standard_font = <panfont.pcx>,8,10;
```

The final two parameters give the width and height in pixels of a single character within the bitmap set-of-characters (including the gaps). Such a 128-character bitmap has to include exactly

8*10*128 = 10240 pixels. Font bitmaps of 11, 128, or 256 characters are possible. 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 function makes our text appear on the screen:

```
function show_my_text() { my_text.visible = on; }
```

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

```
function roll_my_text()
{
   my_text.pos_y = 480;      // move the text below the screen
   my_text.visible = on;     // make it visible
   while (my_text.pos_y > 0) {   // as long as text still is on-screen
      my_text.pos_y -= 1; // roll upwards one line of pixels
      wait(1);           // then wait for one frame
   }
   waitt(16);                // time enough to read the last line
   my_text.visible = off; // shut off the text
}

on_p = roll_my_text;
```

The function is initiated by pushing the **[P]** key. We have built a **while** loop into this; the text moving and **wait()** instructions are repeated at each frame cycle. Only if the value of **pos_y** after much subtracting of **1** has finally reached **0**, i.e. if it touches the upper end of the screen, the **while** loop won't be executed any more. The function then lets the text disappear, and 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.

A **panel** on screen represents such displays. To give an example for a panel we shall now represent the values of certain variables on screen in form of numbers. It is very helpful to have those variables ever present before you eyes while testing - **debugging** - a function you have written yourself.

Panels are defined similarly to texts. Instead of a string this time there are definitions for numeric displays:

```
font standard_font = <panfont.pcx>,8,10;

panel debug_panel {
```

```
   pos_x  = 2;
   pos_y  = 2;
   flags  = visible,refresh;  // make it visible and displayable over the view
   digits 0,0,3,standard_font,16,time_fac; // frames per second
   digits 40,0,3,standard_font,1,camera.pan;
   digits 80,0,3,standard_font,1,camera.tilt;
   digits 120,0,4,standard_font,1,camera.x;
   digits 160,0,4,standard_font,1,camera.y;
}
```

We now have some 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 displays a variable or numeric parameter. **`Digits`** contain 8 parameters, that determine the X- and Y-position of the display within the panel, the number of digits, the font, a factor for multiplication, and the variable or parameter itself. The position values give the distance in pixels of the upper left corner of the numeric display from the upper left corner of the panel. The **`refresh`** flag causes the panel to be rebuilt constantly. This is the only way we can have it displayed above the camera view, 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, and four more parameters giving the camera view's position. If we would like to follow the state of other variables 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 variable **`time_fac`** for. This variable 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:

```
var fps;

function show_panel()
{
   while (1) {      // repeat forever
      fps = 0.8*fps + 0.2*time_fac;
      wait(1);
   }
}
```

Our eternally running function **`show_panel`** is responsible for 'processing' the display variables. It must be started at game start by a **`show_panel()`** instruction within the **`main`** function, and then will run forever. The **`debug_panel`** must now show the **`fps`** instead of the **`time_fac`** variable. The arithmetic expression 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 is a bitmap that can be shifted under a window according to the values of variables given. We would like to make use of a **`window`** display to represent a compass on screen. We start by painting a bitmap for the compass-bar (replace the 'O' by an 'E' to adapt it to the English language):

```
bmap compass_map = <compass.pcx>;   // 160x20 pixels
var compass_pos[2] = 0,0;

panel compass_pan {
   pos_x  = 220;
   pos_y  = 2;
```

```
   flags  = visible,refresh;
   window  0,0, 40,20,compass_map,compass_pos.x,compass_pos.y;
}
```

The compass shall always give us the camera's line of sight, 0° equalling east. The **window** line is structured similarly to **digits**; here again first the position is given, then the width and height of the window in pixels, then the bitmap to be shifted, and finally the two parameters that give the horizontal and vertical shift in pixels. We're using only a horizontal shift here, so **compass_pos.y** is always **0**. We calculate **compass_pos.x** from the camera angle:

```
function show_panel()
{
   while (1) {      // repeat forever
      fps = 0.8*fps + 0.2*time_fac;
      compass_pos.x = 120 - (camera.pan % 360)/3;
      wait(1);
   }
}
```

For scrolling around we made a compass bitmap 160 pixels wide that repeats itself after 120 pixels. Therefore our window of 40 pixels size is always filled with the compass bitmap (120+40 = 160). For a full rotation of 360 degrees the bitmap must shift bei 120 pixels, that's a third – so we divide the angle by three. We have to limit the angle range to 0..360 by the modulo (%) function, because **camera.pan** can be outside this range. And finally we have reversed the shift direction by subtracting the shift amount from its maximum value of 120 pixels.

Friday: Controlling the Game

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.

Changing Levels

There is an easy way to change levels. The player may walk through a certain door, which – maybe triggered by **event_trigger** – performs the following function:

```
string levelchange_str = "entering next world... please wait";

function moveto_level2()
{
   msg.string = levelchange_str;
   show_message();
   wait(1);        // time for message to become visible
   load_level(<level_2.wmb>);
}
```

The **show_message** function can be found in the **message.wdl**, and will show an on-screen message for some seconds. Of course, you may alternatively display a fullscreen panel, or play an AVI movie.

Loading a level will remove all entities, including the player and his weapon model, clear all entity synonyms, and terminate all entities' actions – that includes the function above, if it was triggered by a door entity. All non-entity variables however keep their values, and all non-entity functions keep running. So after loading the level, the next thing to do would be to re-create the weapon entities, who must have been stored using variables before (a synonym won't do because entity synonyms are cleared). And the function that does this must be a global function. There is an easy trick to make an entity action global:

```
action moveto_level2()
{
   msg.string = levelchange_str;
   show_message();
   wait(1);     // time for message to become visible
   my = null;   // now this action will be a global function, and keep running
   load_level(<level_2.wmb>);
   wait(1);     // time for new level to get loaded
   weapon_restore();   // function to re-create the weapon models
}
```

We had to insert the **wait()** instruction after **load_level** because the level is not loaded immediately, but after the end of the current frame cycle. To re-create all weapons the player has picked up during the game, we have to define an individual **weapon_restore** function dependent on the weapons we are using in the level. If we are using the predefined **weapons.wdl**, we know that picking up a weapon sets the corresponding **weapon1..weapon7** variable to non-zero. So that function may look like this (**gun1** etc. being the functions initialising the guns and setting them invisible):

```
function weapon_restore()
{
   if (weapon1 != 0) { create(<smallgun.mdl>,nullvector, gun1); }
   if (weapon2 != 0) { create(<biggun.mdl>,nullvector, gun2); }
// … and so on
   gun_select();// weapons.wdl; re-selects the current weapon
}
```

Saving and Loading Games

With the **save** instruction the complete state of the game is written to the harddisk all at once. The game scores saved are encrypted and compressed so that they won't use up too much room on the harddisk. With the instruction **load** a saved game score is decrypted and reloaded. All functions that were running when saving occurred will resume running after loading the score. If the player is in a different level during loading, the level will be changed automatically like with a **load_level** instruction. Loading itself - like the instruction **load_level** - will not be executed immediately in the function the corresponding instruction can be found in, but only after the end of the frame cycle.

We shall define a function 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 screen message shall give the number of the score concerned.

```
var_info slot= 1;   // number of score (1..3)
var load_which = -1;          // which of the scores to load

savedir   "c:\\a4games";

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)?";

function qsave_game() {          // save score, with query
   yesno_txt.string = save_yesno;
   yesno_do = qsave_game1;
   yesno_show();
}

function qsave_game1() {      // save score directly
   load_status;     // load slot variable
   slot += 1;       // and increment it
   if (slot>3) {slot -= 3; } // if >3 then back to 1
   load_which = -1;// begin loading the last one
   save_status();
   save("qgame",slot); // save game
   if (result < 0) {   // error at saving?
      msg.string = save_error;
   } else {
      msg.string = ok_str;
   }
   show_message();
}

function qload_game()
{
   load_which += 1;
   if (load_which > 2) { load_which = 0; }
```

```
   if (load_which == 0) { yesno_txt.string = load_yesno1; }
   if (load_which == 1) { yesno_txt.string = load_yesno2; }
   if (load_which == 2) { yesno_txt.string = load_yesno3; }
   yesno_do = qload_game1;
   yesno_show();
}

function qload_game1 {
   msg.string = wait_str;
   show_message();
   wait(1);        // to show the message before loading
   load_status();  // load back global slot variable
   slot -= load_which;
   if (slot < 1) { slot += 3; }
   load("qgame",slot);
   wait(1);
   msg.string = load_error; // went wrong!
   show_message();
}

on_f2  = qsave_game;
on_f3  = qload_game;
```

We are going to see that the problem is to remember the number of the score last saved after loading some other new score. Because loading a game changes every variable, we have to save the number of the game last saved separately on the harddisk. The only way to do this is to use an info variable, which we have called **slot** in this case, and perform **save_info**.

We'll be using a few **message.wdl** and **menu.wdl** functions to give on screen messages or to wait for a Yes/No keystroke. The function **qsave_game** is executed through the **[F2]** key. Through pushing **[Y]** or **[Enter]** after the query we then start the proper save function **qsave_game1**. Together with other info variables we may have defined elsewhere, the variable slot is read from the file INFO0.SAV. This variable 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 sa v**. Afterwards the proper saving of the score under the name of **qgame0.sav**, **qgame1.sav**, or **qgame2.sav** is carried out - depending on which value **slot** currently has. If an error occurs during saving the predefined variable **result** automatically acquires a value not equal **0**. In this case we give out an error message, or else "OK", using the function **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 variable number and cyclically assigns one of the three functions **qload_game1**, **qload_game2**, or **qload_game3** to **[F3]**.

Pushing **[Y]** executes the function **qload_game**. This function 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. Displaying a text on-screen begins but with the next frame cycle, so without the **wait()** we would never get to see the message, because the **load** instruction stops screen refresh until the saved level was loaded completely.

In order to determine the number of the score to be loaded, **slot** is again read from the info file and the variable number is subtracted from it. The result 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 of our WDL tutorial, the last little puzzle to solve over the weekend: This time we won't have to evaluate the variable **result** in order to determine whether something went wrong with loading the file. Now, why would that be?

Reference: WDL Syntax

WDL scripts consists of **definitions** and **functions**. Definitions create objects and set their name and initial properties; functions determine the behaviour of objects by dynamically changing their properties during gameplay, depending on certain conditions. This is common for all programming language. The WDL syntax is derived from Java, Javascript, or C, but simplified. For an object to exist, it can either be placed into the map by WED – like it's done with entities - or it can be created by a script definition. The latter is done by giving its **type**, its **name,** and an initial **value** in the script file, like this:

type name* = *value;
or
type name { ... }

In the second line, between the winged brackets inital values are assigned to the single parameters of the object. A ***name*** may consist of up to 30 letters. The case does not matter. ***Names*** must not begin with numbers nor contain any special characters except the underscore _ , nor are you allowed to define the same name twice for different objects. Examples:

```
bmap sky1 = <stars.pcx>;         // defines a bitmap, named sky1
var position[3] = 10, 20, 30;                    // defines a vector named position
text mytext { font = standard_font; string = "this is a text!"; } // defines a formatted text
```

The '=' character can be omitted in such definitions, as is often done in older WDL files.

Objects consist of one or more of the following basic elements:

Variable – a fixed point number of up to six digits, with up to three digits following the decimal (eg. 123456.789). Thus variables range between 999999.999 and -999999.999. A group of three numbers (that is often used in 3D arithmetics) is called a **vector**.

Flag - a binary value, like a switch, which can be set to ***on*** (**1**) or ***off*** (**0**).

String - a text string, which must be written within quotation marks; line feeds must be written in C notation as "**\n**", back slashes as "****". Example: **"it is now time\nfor all good men…"**.

<***Filename***> - the content of a file. The file name given must be in 8.3 format. The kind of file is apparent from the extension. Valid extensions are **.pcx** and **.bmp** for bitmaps or sprite entities, **.mdl** for model entities, **.wmb** for map entities, **.mid** for songs, **.wav** for sound effects, **.avi** for animation, **.wdl** for scripts. All file names given in pointed brackets **<...>** in a **wdl** script will be packed into a world resource by the file packer. Some files must not be packed – for instance, **dll** files or **avi** movies.

In the virtual world we have a right-handed XYZ coordinate system with the Z-axis standing upright. In the 2-D case, for positioning something on the screen or within a bitmap, we use a XY coordinate system in pixel units, with the Y-axis pointing downwards, and the origin is placed in the upper left corner.

Space and time units of the virtual world are the **quant** and the **tick**. One quant is equivalent to one unit in WED and MED coordinates, and therefore one texture pixel at a 1.0 scale. How much inches a quant shall be depends on the relative size of the models. We recommend one inch per quant for person based games (shooters or adventures), and 4 inches per quant for vehicle based

games (war games or flight simulators). This way with the same level size you get much bigger worlds.

One **tick** is equivalent to 1/16 second - the average time between two frame cycles on a low-end PC. Angles are stated in degrees (0 to 360) and counted counter-clockwise. For rotations in three dimensions the so-called **Euler angles** are used: **`pan`** is the horizontal angle (0..360) about the upright Z axis, **`tilt`** is the vertical angle (-90..+90) about the rotated Y axis, and **`roll`** is the angle (0..360) about the rotated and tilted X axis. For a pan angle, 0 degrees is equivalent to the positive direction of the X-axis coordinate, which points to the east of the map. 90 degrees is north, 180 degrees west, and 270 degrees south.

Colours are given as red, green, blue component values between 0 and 255. All component values set to 0 gives black; all set to 255 gives white. The maximum colour value of 255 must not be exceeded.

The following special characters are valid within WDL:

`...;`	Semicolon terminates each instruction.
`{...}`	Winged brackets enclose assignment or instruction lists.
`"..."`	Text is given between quotation marks.
`[...]`	Array indices are giving between square brackets.
`<...>`	File names are given between pointed brackets.
`#...`	Comment until the end of the line.
`//...`	Comment until the end of the line.
`/*...*/`	Comment block.

WDL, as each other programming language, is unforgiving to syntax errors. Each forgotten or superfluous comma or semicolon will almost certainly produce an error message at start-up. So be carefully.

Some of the features described here are only available in the A5-Engine or above. They are marked with **A5** .

Functions - the brain of the game

Functions control the game and the pseudo-intelligent behaviour of the entities. An function consists of a list of **instructions,** which will be executed one after another. Gamestudio's engines, ***A4*** or ***A5***, are multitasking engines. Thus a lot of functions run simultaneously, just as in real life. The instructions represent a kind of a simple language, which allows you to influence not only the entities, but also the whole game play in numerous ways.

There are local functions and global functions. The difference is on what machine they run in a multiplayer environment. Functions attached to an entity (also named **Actions**)are **global**, and will run on the server; this way they influence the whole game world. Functions assigned to the user interface - that means functions attached to keystrokes or panel buttons - are **local** and will only run on the client. In a single-player environment – that's the normal case - the server and the client are the same PC, and you don't have to bother about where your functions run.

There are many pre-frabricated functions in the included WDL script, which should be sufficient for all kinds of simple games. You'll find a list of prefabricated functions in the appendix. However, if you want to write your own WDL functions, define them the following way:

function ***name (parameter)*** **{ .... }**
action ***name*** **{ .... }**

Creates a function or action with the given *name*. An optional numeric parameter - a variable or expression - can be handed over on calling the function, to pass information to it. The function can access that ***parameter*** under the name given between the parantheses, like a normal variable. Example:

```
function beep_times(times) {
   while (times > 0) { beep; times -= 1; wait(1); }
}
// beep_times(7) will now beep seven times
```

If a variable or another object of the same name exists, all references within the function refer to the 'local' parameter and not to the 'global' variable. The function may change the parameter, as in the example, but the original parameter in the calling function remains unchanged. Only single-number-parameters, not vectors or synonyms, may be used. The parameter can be omitted, in that case the space between the parantheses remains empty.

Actions are a special kind of functions, they don't accept parameters, but will appear in the action pop-up list in WED's entity property panel. So actions can be assigned to entities, and functions are for internal calculations. Besides that, there is no difference between functions and actions. This is our usual example for an entity action, which consists of just three instructions, and lets the entity rotate:

```
action ent_rotate {
   while (1) {
      my.pan += my.skill1*time;
      wait(1);
   }
}
```

Inside a function, the most basic type of instruction is the assignment instruction:

var = expression;
var (+-*/)= ausdruck;

Assigns a value or the result of an arithmetic expression to the given object or object parameter (object parameters are given by the parameter's name, prefixed by the object's name or synonym and a dot). The object, whose parameter is to be changed, can either be an entity placed in the map or some other object defined in the WDL script **before** the function. The arithmetic expression may be composed of any numbers, further variables or object parameters, brackets, and arithmetic operators. Examples:

```
x = (a + 1) * b / c;
z = 10;
my.tilt = asin(3*x + 0.5);
my.z += 2;     // increase the entities' z position by 2 quants
my.z += 2*time;     // move z by a time-corrected speed
my.event = react_function;       // sets the entities' event function
my.invisible = on;               // sets the entities' invisible flag
on_s = save_function;            // assigns saving to the [s] key
```

In expressions, additionally to the basic operators + - * /, the following binary operators are supported:

%	Modulo (remainder of a division)
|	Bitwise OR
^	Bitwise Exclusive OR
&	Bitwise AND

The "**=**"-character can be combined with the basic operators:

+=	adds the expression to the parameter
-=	subtracts the expression from the parameter
***=**	multiplies the parameter by the expression
/=	divides the parameter by the expression

One has to bear in mind that the fixed point arithmetic rounds all factors or intermediate values below 0.001 down to zero. Also the maximum and minimum values of 999999.999 and -999999.999 must not be exceeded. This applies to intermediate results within an expression also - so a little care has to be taken using mathematical expressions. Invalid operations - a result exceeding the maximum values, a division by zero, a square root of a negative number etc. - will cause wrong results or error messages.

In the following description of instructions, most instructions accept one or more **parameters**. Parameters can be variables or other objects. Some instructions take their parameters in parentheses and return a value that can be directly used in expressions, like the following ones:

Variable functions

The following functions just take one or two numbers as parameters, and deliver a number as result:

random(x)

Random fractional number between 0 and x. Example:

```
a = random(2*b) - b;    // sets a to a random value between -b and +b.
```

sin(x), cos(x), tan(x)
asin(x), acos(x), atan(x)
Trigonometric functions and inverse trigonometric functions.

ang(x)
Shifts the angle x into the -180 .. +180 range.

log(x)
Logarithm of x at base e.

exp(x)
e power x.

sqrt(x)
Square root of x.

sign(x)
Delivers -1 if (x<0), 1 if (x>0), 0 if (x==0).

abs(x)
Absolute amount of x.

int(x)
Integer part of x (digits after decimal are cut off).

frc(x)
Fractional part of x (digits before decimal are cut off).

max(x,y), min(x,y)
Maximum resp. Minimum of both numbers. Example:

```
a = max(0,(min(a,10)); // limits a to values between 0 and 10
```

Vector instructions

The following instructions operate on vectors, i.e. groups of three numbers. Vectors can be used for different purposes in the game, like for describing a **xyz** position, or a direction or speed, or a **pan tilt roll** 3-dimensional Euler angle. Either an array of length 3, or the first of three consecutive skills or numeric parameters of an object can be used as a vector for the following instructions. For instance, to use the **my** entity's **x y z** parameters as a position vector, just **my.x** can be given. Thus, valid vectors for the following instructions are

- any predefined vectors, and any variable that is defined as **var[3]**;
- any variable array element whose index is a multiple of 3, and at least 3 less then the length of the array (like **array[0]**, **array[3]**, **array[6]**...);
- any entity skill between **skill1** and **skill46**;
- the **x**, **pan**, or **red** parameter of any entity;
- the **x** or **pan** parameter of any view.

If anything else is given when a vector is expected, an engine crash can be the result – for speed reasons the validity of vectors are not checked.

vec_set (*vector1*, *vector2*);
Copies the second vector's three numbers to the first.

vec_add (*vector1*, *vector2*);
Adds the first and second vector, and copies the three sums to the first.

vec_sub (*vector1*, *vector2*);
Subtracts the second vector from the first, and copies the three differences to the first. Example:

```
vec_set(v1,nullvector);    // clear v1 to 0,0,0
vec_sub(v1,v2); // now v1 is the negative of v2
```

vec_scale (*vector*, *var*);
Multiplies all three numbers of the vector by the given number or expression.

vec_dist (*vector1*, *vector2*);
vec_length (*vector*);
The **vec_dist()** instruction returns the distance between two positions. The **vec_length()** instruction returns the length or magnitude of a vector, i.e. its distance to the level origin. Example:

```
  distance = vec_dist(my.x,you.x);  // calculate distance of MY and YOU entities
```

This is mathematically equivalent, but overflow-proof and faster than calculating the distance separately by:

```
  temp.x = my.x - you.x;
  temp.y = my.y - you.y;
  temp.z = my.z - you.z;
  distance = sqrt(temp.x*temp.x + temp.y*temp.y + temp.z*temp.z);
// temp.x, temp.y, temp.z  must be below 1000 for this to work!
```

vec_normalize (*vector*, *var*);
Chops the vector to the length given by the second variable, while keeping it's direction.

vec_rotate (*vectordir*, *vectorangle*);
Will rotate the the first vector by the **pan**, **tilt** and **roll** angles given trough the second vector. ***Vectordir*** has the meaning of a speed or direction here, ***vectorangle*** has the meaning of a three-dimensional angle (Euler angle). Example:

```
var direction[3] = 10, 0, 0;  // direction, pointing straight eastwards in the XYZ coordinate system
var angle[3] = 90, 45, 0;     // pan 90 degrees, tilt 45 degrees
...
vec_rotate(direction,angle);
```

After the instruction, the **direction** vector has the new values 0, 7.07, 7.07. It was rotated by 90 degrees counterclockwise and tilted by 45 degrees upwards, so it now points to the north and halfway upwards.

vec_to_angle (*vectorangle*, *vectordir*);
Computes the **pan** and **tilt** angles and of the direction given by the second vector, and places them into the **pan** and **tilt** parameters of the first vector. Also returns the length of the direction vector. Very useful for trading an angle for a direction, thus computing the angles to a target. Example for turning the **my** entity towards the **you** entity:

```
function turn_towards_target() {
// get the direction from the entity MY to the entity YOU
```

```
  vec_set(temp,you.x);
  vec_sub(temp,my.x);
  vec_to_angle(my.pan,temp);  // now MY looks at YOU
}
```

vec_to_screen (*vector, view*);

Converts the **xyz** coordinates of the given vector to screen coordinates. After the instruction, the **x** and **y** coordinates of the vector contain the **xy** screen position and the **z** coordinate contains the distance to the screen plane. The **view** must be **visible** for this instruction to work. If the resulting position is invalid, i.e. outside the screen boundaries, the instruction returns **0**, otherwise it returns **1**.

This instruction can be used to attach texts or panels to screen positions of entities. Example:

```
  panel flare_pan { bmap = flare_map; flags = transparent,d3d; }
  ...
// attach a lens flare to the MY model entity
  vec_set(temp,my.x);
  if (vec_to_screen(temp,camera)) // if visible on screen
  {
    flare_pan.pos_x = temp.x; // place the flare panel
    flare_pan.pos_y = temp.y;
    flare_pan.visible = on;
  } else {
    flare_pan.visible = off;  // otherwise disable it
  }
```

vec_for_screen (*vector, view*);

The is the opposite of the **vec_to_screen()** instruction. It converts screen coordinates, given through the vector, to a world position of the given view. Because a unique 3D position can not be determined from a **x y** screen position only, a depth must also be given in the **z** coordinate of the vector. The viewmust be visible for this instruction to work.

This instruction can be used to place entities into a level on mouse click on the screen. Example:

```
  function spawn_sprite() {
// spawn a sprite at mouse click position, 200 quants behind the screen
    temp.x = mouse_pos.x;
    temp.y = mouse_pos.y;
    temp.z = 200;
    vec_for_screen(temp,camera);
    create(<arrow.pcx>,temp,null);
  }
  on_mouse_left = spawn_sprite;
```

rel_to_screen (*vector, view*);
rel_for_screen (*vector, view*);

Like **vec_to_screen()** and **vec_for_screen()**. However they use a position relative to the view, and not to the world - the angle and origin of the view is not considered. Used for calculating the screen positions of defined entities that are attached to a certain view.

String instructions

The following instructions can be used to manipulate and evaluate character sequences (strings). If the string is not changed by the instruction, it can be given directly as argument:

str_cpy (*string1, string2*)

Copies the content of ***string2*** into ***string1***. If ***string1*** was shorter than ***string2***, the result will be truncated.

str_cat (*string1*, *string2*)

Appends a copy of ***string2*** to the end of ***string1***. If the original length of ***string1*** was shorter than the result, it will be truncated. Example:

```
string s = "                  ";

str_cpy(s,"hello");    // now s == "Hello "
str_cat(s,"world");    // now s == "Hello World"
```

str_cmpi (*string1*, *string2*)

Compares both strings without case sensitivity, and returns a value of 1 if they are equal, and 0 otherwise. Example:

```
string s1 = "hello world";
string s2 = "hello universe";

str_cmpi(s1,"hello world");    // returns 1
str_cmpi(s2,"hello world");    // returns 0
```

str_cmpni (*string1*, *string2*)

Compares both strings without case sensitivity, and returns a value of **1** if their overlapping parts are equal, and **0** otherwise. Example:

```
string s1 = "hello world";
string s2 = "hello universe";
...
str_cmpni(s1,"hello");  // returns 1
str_cmpni(s2,"hello");  // returns 1
str_cmpni(s1,s2);       // returns 0
```

str_len (*string*)

Returns the number of characters in the string. Example:

```
temp = str_len("hello ");    // now temp == 6
```

str_clip (*string*, *number*)

Clips the given number of characters from the beginning of the string. Example:

```
string s = "hello world";
...
str_clip(s,6);    // now s == "World"
```

str_trunc (*string*, *number*)

Clips the given number of characters from the end of the string. Example:

```
string s = "hello world";
...
str_trunc(s,6);   // now s == "Hello"
```

str_stri (*string1*, *string2*)

Returns the character number (beginning with 1) of the first case-insensitive occurrence of ***string2*** in ***string1***. If no occurence is found, **0** is returned. Example:

```
string s1 = "hello world";
...
temp = str_stri(s1,"world");  // now temp == 7
```

str_to_num (*string*)
Returns a number converted from the string. Example:

```
string s = "3.14";
...
pi = str_to_num(s);    // now pi == 3.14
```

str_for_num (*string, number*)
Sets the string to the character representation of the number. Example:

```
string s = "                  ";
...
str_for_num(s,sqrt(2));     // now s == "1.414"
```

str_to_asc (*string*)
Returns a numeric ASCII representation of the first character of the string. Example:

```
string s = "a";
...
temp = str_to_asc(s);    // now temp == 65
```

str_for_asc (*string, number*)
Sets the first character of the string to the ASCII character represented by the number. Example:

```
string s = "   ";
...
str_for_asc(s,65);     // now s == "A  "
```

File instructions

The following instructions write into or read out of files. They can be used for exchanging data with other programs, or for many other purposes like outputting variable contents for debugging. File instructions are available in the A5 engine only.

A5 ***handle* = file_open_read (*string*);**
Opens a file for reading. The file must exist in the **savedir**. The file name including the extension must be given by the ***string***, without a path. This instruction returns a file handle - that is a unique number to identify the opened file. The file handle is used by other instructions to access that file. If the file can't be opened - if it does not exist, for instance - **0** is returned instead. File handles become invalid after a **load()** instruction, so file operations shouldn't be interrupted by save/load operations.

A5 ***handle* = file_open_game (*string*)**
Like **file_open_read()**, but this instruction takes its file from the game directory, instead of the save directory. This way files with arbitrary data can be provided together with the game. The file mustreally exist in the game directory, it can't be encrypted within a resource. Example:

```
filehandle = file_open_game("address.txt");
```

A5 ***handle* = file_open_write (*string*);**
Opens the file for writing into. If the file does not exist in the **savedir**, it is created; otherwise it's previous content is erased.

A5 *handle* = **file_open_append** (*string*);

Opens the file for appending additional text at the end. If the file does not exist in the **savedir**, it is created.

A5 **file_close** (*handle*);

Closes the file with the given handle. A file must be closed before other programs can access it, or before it can be opened again in a different mode. As soon as the file is closed, the handle becomes invalid.

A5 **file_var_write** (*handle, var*)

Writes the given number or variable into the file with the given handle. The file must be open for writing or appending. The number is written with 3 decimals, followed by a blank. Example:

```
var filehandle;
...
filehandle = file_open_write("myfile.txt");
file_var_write(filehandle,123456.789);
file_var_write(filehandle,sqrt(2));
file_close(filehandle); // the file now contains "123456.789 1.414 "
```

A5 **var = file_var_read** (*handle*)

Returns the number that was written into the file with the given handle, and proceeds to the next position within the file. The file must be open for reading. If there are no further numbers, **0** is returned. Example (with the file created above):

```
filehandle = file_open_read("myfile.txt");
temp = file_var_read(filehandle);     // now temp == 123456.789
temp = file_var_read(filehandle);     // now temp == 1.414
file_close(filehandle);
```

A5 **file_str_write** (*handle, string*)

Writes the given string into the file with the given handle. The file must be open for writing or appending. If the string contains a '**\n**', a line feed is inserted; '****' inserts a '\'.

A5 **file_str_read** (*handle, string*)

Reads text from the file with the given handle into the given string, until a delimiter is reached. The delimiter is either a '0' byte in the file, or a line feed, or a comma (**','**). This way, comma separated strings from a database file can be read. The file must be open for reading. The original length of the string won't be overwritten.

A5 **file_chr_read** (*handle, string*)

Reads characters from the file with the given handle into the given string, until the string is full. The file must be open for reading. The number of characters read is the original length of the string.

A5 **file_asc_write** (*handle, var*)

Writes a single byte given by the number or variable (**0..255**) into the file with the given handle. The file must be open for writing or appending.

A5 **var = file_asc_read** (*handle*)

Reads a byte out of the file with the given handle, and proceeds to the next byte within the file. The file must be open for reading.

Control instructions

The following instructions control the execution of the function. Control instructions can't be used by DLL extensions.

if (*expression*) { *instructions...* }
if (*expression*) { *instructions...* } else { *instructions...* }

Will execute all instructions between the first pair of winged brackets only if the expression between the round brackets is true (i.e. evaluates to non-zero). Otherwise, the instructions between the second pair of winged brackets (after **else**) will be executed. **Else** and the second set of instructions can be omitted.

The expression can be a comparison between two sub-expressions, or a comparison of flags. Only parameters of the same type may be compared. For flag comparisons the values **on** (set) or **off** (not set) can be compared. For comparisons of non-numerical parameters the value **null** for non-existence can be used. The following operators for comparison between two parameters or sub-expressions are available:

||	true if either is true (**or**)
&&	true if both are true (**and**)
!=	true if both are not equal
==	true if both are equal
<=	true if the first is below or equal to the second
>=	true if the first is above or equal to the second
<	true if the first is below the second
>	true if the first is above the second

All comparisons evaluate to a value of **0** for false, and **1** for true. Comparisons can be combined using brackets. Examples:

```
if (my.flag1 == off) {     // if FLAG1 is not set,
  y = -1;                  // then set y and z to -1
  z = -1;
} else {
  y = 1;                   // otherwise, set y and z to 1
  z = 1;
}

if (((x+3)<9) || (y==0)) {
// set z to 10 if x+3 is below 9, or if y is equal to 0
  z = 10;
} else {
// set z to 5 in all other cases
  z = 5;
}
```

Please note that the "equals" comparison is done with "**==**", to differentiate it from the assignment instruction with "**=**". Always use round brackets, like in the example, to determine the order of evaluation within the expression. An **if** instruction may appear within the winged brackets of another **if** instruction; this way you can 'nest' several **if** instructions.

while (*expression*) { *instructions...* }
break;
continue;

The **`while`** instruction will repeat all instructions between the winged brackets as long as the expression between the round brackets is true resp. evaluates to non-zero. This repetition of instructions is called a **loop**. The expression will be evaluated again at the beginning of each repetition. If you want the loop running forever, simply use the value **`1`** for the expression.

`While` is often used to modify a value slowly during a number of frames, e.g. to open a door or to move an elevator. For this a **`wait()`** instruction (see below) has to be inserted between the winged brackets, to indicate that a frame time must pass for each repetition.

The instruction **`break`**; between the winged brackets will end the repetitions and continue with the first instruction after the closed bracket. The instruction **`continue`**; will begin immediately with the next repetition. As an example for using **`while`** see the **`ent_rotate`** function at the beginning of this chapter.

goto *label;*

Jumps to a target label in the function and proceeds from there with the subsequent instructions. ***label*** may be any name followed by a colon as target mark anywhere between two instructions in the function.

function (*number*);

Starts the function whose name is given, and delivers the given ***number*** or expression as its parameter. The parameter can be omitted, in that case the parantheses remain empty. After the end of the function, or on encountering a **`wait()`** instruction (see below) within, it returns to the current function and goes on there with the next instruction. In case of encountering a **`wait()`**, the newly started function as well as the current function may run both simultaneously from the next frame cycle on. Example:

```
function beep_times(times)
{
   while (times > 0) { beep; times -= 1; wait(1); }
}

...
beep_times(7);    // beeps seven times
...
```

return;
return (*number*);

Terminates the function. In the second case the given ***number*** or expression is returned to the calling function, where it can be evaluated in an expression. Example:

```
function distance() {
   if (my == null || you == null) { return(99999); }  // big distance indicates error
   return(vec_dist(my.x,you.x));
}

function beep_if_close() {
   if (_distance() < 100) { beep; }
}
```

exclusive_global;
exclusive_entity;

Each function can be started several times, so many instances of the same function can be running simultaneously. **exclusive_global** will terminate all other instances of the current function, and thus prevent that parallel-running functions disturb each other. **exclusive_entity** will terminate all other functions started by **my**, i.e. the current entity, at their next **wait()** instruction.

wait (*number*);
waitt (*number*);

Causes the function to pause for the given ***number*** of frame cycles or ticks. Everything outside this function - rendering, user communication, global variable changing and execution of other functions - will be performed during that pause. **Wait(1)** is often used at the end of a **while** loop, during which a value has to change smoothly over a certain amount of time. If the function with the **wait()** instruction was started from another function, this other one will also go on and execute its next instructions during the pause. **Waitt()** does the same as **wait()**, but the function is paused for a given number of ticks instead of frame cycles. This instruction is able to wait for a fixed period of time. The higher the given number is, the more precise the time will be kept. The minimum time this instruction will wait is one frame cycle.

Entity instructions

The following instructions are used for controlling entities, environment and collision detection in the game:

content (*vector*);

Checks the block material at the position given by the ***vector***. The instruction returns a value of **content_empty** (**1**) if empty space, **content_passable** (**2**) if a passable block, or **content_solid** (**3**) if a solid block is detected at the given position. Blocks in the level as well as in map entities are detected. **You** is set to the map entity detected at the position, if any. This instruction can't yet be used by DLL extensions. Example:

```
// check if the camera position is under water in the current level
   result = content(camera.x);
   if (result == content_passable) {    // under water
      fog_color = 2;  // set global fog to 2, blue fog
   } else {
      fog_color = 0;  // switch global fog off
   }
```

ent_alphaset (*frame*, *percent*);

Generates an alpha transparency map for the **my** sprite entity by multiplying the brightness of the sprite bitmap by a percent factor (-100..100). Negative percent factors produce an inverse alpha map. The alpha map can be calculated individually for each sprite frame, and gives better control than the **flare** flag. If the frame parameter is **0**, the alpha map is calculated for all frames of the sprite. Useful percent values are in the range of 5..30. Example:

```
ent_alphaset(0,-15); // creates a transparency map for thick black smoke
ent_alphaset(0,5);   // creates a transparency map for thin white smoke
```

Ent_alphaset can only be executed once per entity, further **alphaset** instructions are ignored. Entities of the same file must have the same alpha map. Alpha maps are only effective in 16- or 32 bit modes.

create (<*filename*>, *vector, action*);

Will create a new entity on the server during gameplay. Mostly used for creating bullets or explosions, or for creating players in a multiplayer game. ***Filename*** is the name of the **wmb**, **mdl**, **pcx** or **bmp** file to create the map, model or sprite entity from. The desired position where the entity will appear can be given by the ***vector***. The given ***action*** will be attached to the entity, and will start running on the server immediately after creation, with the **my** synonym set to the created entity, and **you** set to the creating (if any). **Create()** only works when the level is already loaded. So it can't be used at the very start of the game or at the very start of an entity action. This instruction can't yet be used by DLL extensions.

remove (my);

Removes the **my** entity out of the world, and terminates all functions for which that entity is **my**, except the currently running one. No function should afterwards ever try to access the removed entity (via a **my** or **you** synonym) again. This instruction can't yet be used by DLL extensions.

morph (<*filename*>, my);

This instruction morphs the **my** entity into another one, even of different type, e.g. a model into a sprite, or a sprite into a map entity. This instruction can't yet be used by DLL extensions. Example:

```
morph (<explo.pcx>,my);   // change MY into an explosion sprite
```

ent_frame (*string, var*);
ent_cycle (*string, var*);

These instructions set the **my** model entities' **frame** and **next_frame** parameters to the animation frame whose name begins with the given string. If the animation consists of several frames - e.g. "walk1", "walk2", etc. - the var gives a percentage within the whole animation cycle, whose frame numbers are calculated automatically. 0 percent gives the first frame of the animation, and 100 percent either gives the last one (**ent_frame**), or cycles to the first again (**ent_cycle**). Examples:

```
ent_frame("jump",90);   // 90 percent of jump animation
ent_cycle("jump",90);   // 90 percent within jump cycle
```

If the model has three animation frames named "jump1", "jump2", "jump3", which are frames 10, 11, and 12, the **ent_frame** instruction in the example sets **frame** to 11.8 and clears **next_frame**. The **ent_cycle** instruction however sets **frame** to 12.7 and **next_frame** to 10, in order to inbetween between the last and the first frame.

ent_vertex (*vector, number*);

Sets the given vector to the current **xyz** position of the vertex with given number of the **my** model entity. This instruction is extremely useful for composing entities from several parts. Weapons can be attached and changed without having to change the whole model, flares can be attached to headlights or jet engines, or certain parts of models can emit particle effects. The vertex number is indicated by MED on selecting a vertex. Example:

```
// let a model entities' vertex no. 248 periodically emit particles
ent_vertex(temp,248);  // get XYZ position of vertex 248
emit(1,temp,particle_flare);
```

ent_move (*vector_reldist, vector_absdist*);
move (my, *vector_reldist, vector_absdist*);

Both instructions move the **my** entity over a certain distance, and perform a collision detection during the movement (eee the entity chapter about collision detection). The first vector gives a relative distance (in rotated entity coordinates, i.e. the direction that the entity is facing), the second vector gives an absolute distance (in world coordinates). The resulting movement is a combination of both distances. Normally the first vector is used for the propelling speed of the entity, and the second one for external forces, like gravity and drift. To zero one of these vectors the predefined **nullvector** vector can be given.

ent_move returns the amount of distance covered, **move** sets the **result-**variable. If the entity could not move at all due to being blocked by obstacles, it returns **0**. The predefined **my_speed** vector is set to the resulting absolute distance. If the entity collided with something, the **normal** vector is set to the colliding surface's normal - that is a vector of length 1, pointing perpendicularly away from the surface. It can be used to determine the surface orientation. For instance, if **normal.z** is near 1, the entity collided with the ground; if **normal.z** is near -1, it collided with the ceiling; if **normal.z** is near 0, it hit a vertical wall. The surface of a sprite or model entity is considered a vertical cylinder here. The predefined **bounce** vector is set to the direction into which the colliding entity would bounce off the surface, and thus could be used to implement a bouncing behaviour.

Before doing **ent_move**, the move mode must be set through the predefined **move_mode** variable before. The following move mode combinations are available:

ignore_you	- ignores the **you** entity on collision detection.
ignore_passable	- ignores all **passable** blocks and entities.
ignore_passents	- ignores **passable** model and sprite entities.
ignore_maps	- ignores all map entities.
ignore_models	- ignores all models.
ignore_sprites	- ignores all sprites.
ignore_push	- moves over all entities with lower **push** values than **my**.
activate_trigger	- enables trigger events during the move.
glide	- glides along walls and entities on impact.

The **move** instruction can't be used in DLL plugins, and has a fixed **move_mode** combined of **ignore_you, ignore_passable, ignore_push, activate_trigger, glide**. The **ignore_passents** mode can still detect passable blocks and map entities - if the move ended up inside a passable block, the predefined **in_passable** variable is set to 1. If the move ended up inside a solid block (which only can happen if **passable** entities are moved), the **in_solid** variable is set to 1.

On collision, **ent_move** can trigger collision events on the moving as well as on other entities. If **glide** is activated, **ent_move** will itself try to move as far as possible by gliding along surfaces or making way around obstacles. Ignoring the **you** entity can be used to prevent that a bullet gets stuck immediately within the barrel or within the entity which fired it. The move modes can be combined by simple adding, like this:

```
move_mode = ignore_you + ignore_passable + ignore_push + activate_trigger + glide;
result = ent_move(reldist,absdist); // same as move(my,reldist,absdist);
```

rotate (my, *vectorangle, nullvector*);

Rotates the **my** entity by the **pan**, **tilt** and **roll** angles given by *vectorAngle*. This is not the same as changing the entity Euler angles directly. **rotate** performs a relative rotation about the

already-rotated entity axes, necessary to rotate an arbitrarily oriented object in space. Internally the rotation is performed using a quaternion into which the ***vectorAngle*** is converted. This instruction can't yet be used by DLL extensions. Example:

```
temp.pan = 3*time;
temp.tilt = 0;
temp.roll = 0;
rotate(my,temp,nullvector);
```

trace (*vectorfrom, vectorto*);

This is the general instruction that is used by entities to detect their environment. It sends a ray from the ***vectorfrom*** position to the ***vectorto*** position and checks whether this ray hits an obstacle on its way. In that case the predefined **target** vector is set to the position where the ray hits the surface of the obstacle (maybe to place a blood stain there), the predefined **normal** vector is set to the normal of that surface. The intruction returns the distance of the hit point. If nothing is hit between the two positions, it returns **0**.

If the obstacle was an entity, the **you** synonym is set to that entity; otherwise it's set to **null**. If triggering the **shoot** event is activated and the hit entities' **enable_shoot** flag was set, it's **event** function is triggered with **event_type** set to **event_shoot** (events are described in the entity chapter).

Through the predefined **trace_mode** variable the mode for the tracing can be composed by adding several mode flags. **Trace_mode** must be set immediately before the **trace()** instruction. The following flags are available:

ignore_me	-	ignores the **my** entity.
ignore_you	-	ignores the **you** entity.
ignore_passable	-	ignores all passable blocks and entities.
ignore_passents	-	ignores passable model and sprite entities.
ignore_maps	-	ignores all map entities.
ignore_models	-	ignores all models.
ignore_sprites	-	ignores all sprites.
ignore_push	-	ignores all entities with lower **push** values than **my**.
use_box	-	uses the bounding box (**min_x**, **max_x** etc.) or hull of the **my** entity.
activate_shoot	-	enables **event_shoot** triggering of the hit entity.
activate_sonar	-	enables **event_sonar** triggering of the hit entity.
scan_texture	-	sets **tex_name** and **tex_light** from the target surface.

Ignore_passent ignores model and sprite entities with set **passable** flag, but still detects passable blocks or map entities. It sets the prefined flags **in_passable**, when the starting point is within a passable block, and **on_passable** when the ray hits the surface of a passable block. It can be used to detect water below an entity.

Scan_texture can be used to give actors some kind of vision, by retrieving the texture name and the brightness of the hit surface. After the instruction, the predefined string **tex_name** gives the texture name of the surface hit by the ray, or the entity file name if a model, sprite, or terrain was hit. The variable **tex_light** gives the shadow map brightness (0..255) at the hit position in the level. The texture name can be used to check the kind of floor below an entity. Through the **str_cmpi()** instruction the floor texture name can be determined, and some different walking behaviour can be set. Another possibility is to make make a player invisible for his enemies while he hides in the shadows. If nothing was hit, **tex_name** and **tex_light** are not set.

Use_box traces not a line, but a 'thick' ray with the size of the **my** entity's bounding box for collisions with sprite or model entities, and of it's fat or narrow hull for collisions with the level or with map entities. The result delivered back gives the distance from the hit point to the nearest plane of the bounding box in that case. For the collision detection to work, the box or hull must be outside the target. A vertical trace with **use_box** is used by the **template** scripts for detecting the distance to the ground, keeping the entities' feet on the floor. Small holes or grates will appear filled out for **use_box.**

Examples:

```
// test the floor texture
   vec_set(temp, MY.x);
   temp.z -= 500; // trace downwards 500 quants below
   trace_mode = ignore_me + ignore_passable + ignore_models + ignore_sprites;
   trace (my.x,temp);  // now TEX_NAME is set to the floor name below the MY entity

// look if the YOU enemy can be shot at from my position, and if yes, hurt him
   trace_mode = ignore_me + ignore_passable + activate_shoot;
   trace (my.x,you.x); // if entity YOU was visible from MY position, its SHOOT event was triggerer
```

scan_entity (*arrayFrom, vektorWidth*);

Will scan all entities within a spherical segment of given width and direction. The first parameter is an **array** of 5 values, whose first three (**x**, **y**, **z**) give the sphere origin. The next two give the **scan** direction, i.e. the centreline of the spherical segment, as **pan** and **tilt** angles. Instead of an array, the **x** parameter of an entity or of a view can be given also. The **pan** and **tilt** values of the second vector give the horizontal and vertical width of the segment . The **Z** value of the third vector gives the scan range. If the scan sector values are set to 360, the scan will be a full sphere, like an explosion. Small scan sector values give a very small cone, like a shot from a shotgun.

If any entity with **enable_scan** set is found with it's center within the scan segment, it's **event** function is triggered with **event_type** set to **event_scan**, **result** set to the distance and **you** set to the scanning entity, if any. Also, if the **my** entity had the **enable_detect** flag set, it's **event** function is triggered with **event_type** set to **event_detect**, and **you** will be set to the scanned entity. This way for each entity detected one **event_detect** function will be triggered.

scan_entity() returns the distance of the nearest found entity, if any; otherwise it returns zero. To prevent that a scanning entity triggers himself, the scan won't detect the **my** entity. The instructions scans through walls - so you can't use it for gunshots, use **trace()** for that. The **scan** instruction, however, is very useful to open doors, set switches, detect or alert enemies and so on. Example:

```
function operate() {   // scan nearby doors or switches for operating them
  temp.pan = 120;
  temp.tilt = 120;
  temp.z = 200;
  scan_entity (camera.x,temp);
}
```

A5 scan_path (*arrayfrom, vektorwidth*);

Works like **scan_entity()**, but but finds the next path with its first waypoint in the scan cone, and attaches it to the **my** entity. If the path is not found, **0** is returned; otherwise the distance to the first waypoint is returned. The **target** vector is set to the first waypoint of the path. Example:

```
temp.pan = 360; // full sphere
temp.tilt = 180;
temp.z = 1000;
scan_path(my.x,temp);
```

A5 ent_path (*string*);

Finds the path with the given name, and attaches it to the **my** entity. If the path is not found, **0** is returned; otherwise the number of waypoints is returned. The **target** vector is set to the first waypoint of the path. Example:

```
ent_path("path_001");
```

A5 ent_waypoint (*vector, number*);

Sets the vector to the **my** entity's waypoint with the given number. The total number of waypoints is returned.

A5 ent_nextpoint (*vector*);

If the vector was set to a waypoint of the **my** entity's path, this instruction sets it to the next waypoint. Otherwise, the vector is set to the first waypoint of the path. The number of the new waypoint is returned. The following simple example lets an actor walk along the next path, of which the first waypoint is within 1000 quants of the entity's position:

```
 action walk_path  // let entity walk a circular path
{
   temp.pan = 360;  // find and attach path to entity
   temp.tilt = 180;
   temp.z = 1000;
   result = scan_path(my.x,temp);
   if (result == 0) { return; }   // no path found
   ent_waypoint(my._target_x,1);  // store first waypoint

   while (1)
   {
      temp.x = my._target_x - my.x;  // find direction to waypoint
      temp.y = my._target_y - my.y;
      temp.z = 0;
      result = vec_to_angle(my_angle,temp);
   // near target? Then find next waypoint
      if (result < 25) { ent_nextpoint(my._target_x); }
   // turn and walk towards target
      force = my._force;
      actor_turnto(my_angle.pan); // turn towards waypoint
      actor_move();              // walk straight ahead
      wait(1);
   }
}
```

emit (*var, vector, function*);

This instruction can be used to create a swarm of little moving bitmap particles, to create rocket trails, explosions, laser beams, photon torpedoes, rain, snowstorms, tornadoes or the like. The ***vector*** gives the starting position of the swarm, ***var*** gives the number of particles, and the last parameter is the ***function*** each particle will perform permanently during its lifetime. Because the function can change the position, speed, colour, size, and bitmap of each single particle - an

d can even create new 'children' particles - you have the flexibility to use particles for effects of every description. This instruction can't yet be used by DLL extensions.

The following particle variables and synonyms can be accessed and modified during the particle function (without any prefix):

my_age
Variable, the age of the particle in ticks; is zero at birth.

my_pos
Vector, the position of the particle.

my_speed
Vector, the speed of the particle.

my_map
Synonym for the bitmap of the particle (commercial edition and above). In 8-bit mode, above a certain distance the particle will be displayed as a coloured dot.

my_color
Vector, the RGB colour of the particle if a bitmap is not used. The **red**, **green**, **blue** components may range between 0 and 255. The function must take care that the maximum value of 255 is not exceeded.

my_transparent, my_flare, A5 my_bright
Flags for controlling transparency and flare effects in D3D mode (not available in all editions). They have the same effect as the **flare**, **transparent**, **bright** flags of entities.

A5 my_alpha
Sets the opacity percentage (0..100, default 50) of transparent particles in 16 or 32 bit mode (not available in all editions). At 0 the particle is totally transparent, at 100 it is totally intransparent.

my_size
Variable, the relative size of the particle (default 100). If the size is 0, the particle is invisible, but will continue to exist.

my_action
Synonym for the function of the particle. If set to **null**, the particle will die.

Because you can have many thousands of particles, the function should be as fast and as short as possible. **wait()** instructions are forbidden here.

To start a particle explosion, simply **include** the **particle.wdl** in your WDL file, and try the following from within an entities' action:

```
particle_pos.x = my.x;
particle_pos.y = my.y;
particle_pos.z = my.z;
emit 50,particle_pos,particle_trace;
```

Particle_trace is a particle function predefined in the **particle.wdl**, which can be studied as an example. See the chapter about predefined functions.

Multimedia instructions

The following instructions control the playing of sounds, music and movies. They can't yet be used by DLL extensions.

play_sound (*sound, var);*
Plays a previously defined **sound** with the volume given by the variable (0..100). After the **play_sound** instruction, the **result** variable will be set to a handle number which allows further manipulation or prematurely stopping of that sound. The handle is valid as long as this sound plays. You may create an echo effect by playing the same sound twice, with a short delay in-between. Up to 32 sounds may be played simultaneously.

play_loop (*sound, var*);
Like **play_sound**; but the sound will play continuously until it is stopped explicitly by a **stop_sound** instruction.

play_entsound *(my, sound, var);*
Plays a sound like **play_sound**, but it will be a 3D sound, played at the position of the **my** entity. The range (in quants) of an entity sound is 10 times its volume, given by the variable. The volume may be set to over 100 to give a huge range; the sound itself, of course, is not played louder than with volume 100. The sound will use the sound hardware's stereo and 3D capabilities, and will produce a doppler shift if the entity was moving towards the camera at the time the instruction was executed. Please note that the entity must already exist at the time the sound is played – that means if it was **create**d, that must be happened at least 1 frame cycle before.

stop_sound (*handle*);
Will stop the sound with the handle number given by the variable. Example:

```
sound wave = <wave.wav>;
...
play_sound(wave,50,1);
wavehandle = result;
waitt(50);
stop_sound(wavehandle);
```

tune_sound (*handle, varvol, varfreq*);
Will modify the sound with the handle number given by ***handle***. It will set a new volume (***varvol***, 1..100) and a new frequency (***varfreq***, 10..1000, percentage of the original frequency. If ***varvol*** is **0**, the volume remains unchanged; if ***varfreq*** is **0**, the frequency remains unchanged.

snd_playing (*handle*);
If the sound with the given handle is still playing, this function returns **0**, otherwise **1**. Note that sound handles are not saved, so they become invalid after a **load()** operation.

play_song (*music, var);*
Starts a new background song, which is repeated until another **play_song()** instruction. M*usic* is a previously defined name for a midi file, ***var*** gives the volume for that song between 1 and 100. By giving a zero volume the midi file playing is stopped.

play_song_once (*music, var);*
Like **play_song**, but plays the song only once.

play_cd (*varfrom, varto);*

Plays audio tracks off a CD (some editions only). ***varfrom*** indicates the first, ***varto*** and the last track to be played (min 1, max. 99). If ***varto*** has a higher value than the number of tracks on the CD, the CD will be played to its end. If ***varfrom*** is **0**, the CD currently playing will be stopped; if ***varto*** is **0**, the CD will resume at the place it was stopped. If both are **0**, nothing will happen.

After each execution of **play_cd** the predefined variable **cd_track** will assume the number of the track currently being played, or **0**, if the CD is not playing. In-between **cd_track** won't change; to display the current track continuously, a **play_cd 0,0;** instruction has to be performed repeatedly, e.g. each second.

play_moviefile (*"filename");*

Starts a full-screen movie sequence (some editions only). ***Filename*** is an AVI animation file. It can't be played from within a resource file, but must exist as an external file, so the filename should be given in double qotes instead of angular brackets. During movie play the rendering and the entity actions will be suspended. The predefined variable **movie_frame** will contain the number of the frame currently displayed, or **0** if the movie has come to an end.

stop_movie;

Stops playing the current movie.

Input / output instructions

The following instructions can be used for some special interactions of the game with the outside world:

inkey (*string);*

Gets keyboard input into the ***string*** given. The local keyboard layout will be activated automatically. The instruction then waits for the termination of the input via **[Enter]** and only then proceeds with the function, similar to **wait()**. However, no **save/load** instructions will be executed during an **inkey** input. **[Esc]**, **[Up]**, **[Down]**, **[PgUp]**, or **[PgDn]** allows you to abort the input at any time. The previous content of the string will be restored. The text of the input can be edited using **[BackSpace]**, **[Del]**, **[Right]**, **[Left]**, **[Home]** and**[End]** .

If the string appears in a **Text** displayed on screen (see below), the input as well as the cursor (flashing character #127 from the corresponding **font**) are visible. If the end of the string is reached (results from the length of the initial string of the **string** definition), no further keyboard input will be accepted. Trailing spaces will be cut off. After the termination of the input the variable **result** will be at **27** if the input was aborted with **[Esc]**, **72** with **[Up]**, **73** with **[PgUp]**, **80** with **[Down]** or **81** with **[PgDn].** If it was terminated with **[Enter]**, **result** will be 13. The string's length can be evaluated by the predefined variable **str_len**.

execute (*string);*

Executes the string content as instruction. For debugging purposes, this can be used to type WDL instructions during gameplay, like at a console, in order to change variables or parameters and observe the results immediately. Example:

```
function console() {
   inkey  exec_buffer;
   if (result==13) {          // terminated with RETURN?
      execute  exec_buffer; }   // will execute the string
}
```

exec *(string1, string2);*
Executes an external **.exe** program, which must be located within the **path** (some editions only). ***String1*** gives the name of the program, ***string2*** a command line parameter.

screenshot *("name", var);*
Grabs the content of the screen and saves it as .**pcx** file. The file name is composed from with the given ***name*** string (max. 5 characters, in parentheses) plus a 3-digit number (given by ***var***) and the extension ".**pcx**". In 8-bit mode the file will be saved as 8-bit, in 16-bit mode as 24-bit true colour **pcx**. 32 bit mode screen shots are not supported.

inport *(val, port);*
The given ***val*** is set to the content of I/O port with the address given by ***port***. Through this instruction the hardware of the PC can directly be accessed e.g. to control external devices or to implement new input devices. Note: This won't work under NT or Win2000. Example:

```
var input;
inport  input,372;     // set input to the content of port #372 decimal
```

outport *(val, port);*
The byte given by ***val*** is sent to the port with the address given by ***port***. This won't work under NT or Win2000. Only values between 0 and 255 are valid.

The following instructions can be used to select or unselect certain records of a **dataview** object (professional edition only):

select *(dataview, field name, string);*
unselect *(dataview, field name, string);*
and_select *(dataview, field name, string);*
or_select *(dataview, field name, string);*
Select selects all records of the given **dataview** object, if the first letters of the string field with the given name match the given string. If the string is empty, all records are selected. **Unselect** does the opposite, it deselects the records found and doesn't change the other ones. **And_select** effects only records already selected and **or_select** only unselected records of the **dataview**.

Game flow instructions

The following instructions control the game flow and level changing:

save *("name", var);*
Compresses, encrypts and saves the current game state under a given name plus a number in the game folder (see **savedir**). The file is written one frame cycle after the instruction. The name of the saved file is composed of the first five letters of the given string plus a three-digit number given by ***var*** plus the extension **.sav**. The following elements of the game will be saved:

- The current map,
- all variables,
- all synonyms,
- all changed strings,
- all key and button functions (**on_f1** etc.),
- the state of all currently running functions,
- all **view**, **panel**, and **text** objects,

- all entities that have an action attached.

load *("name", var);*
Loads, decompresses and decodes the game which was saved under the name ***name*** plus a number of three digits given by ***var***. The file is read one frame cycle after the instruction.

save_info *("name", var);*
Like **save**; but saves only the info variables, all bitmaps changed by **freeze_map,** and all strings changed so far.

load_info *("name", var);*
Like **load**; but loads only those values saved with **save_info**.

load_level *(<filename>);*
Level change; loads a new WMB level map from the file <*filename*>. The palette switches to that one of the new level. All current entities are removed, and all entity synonyms referring to them cannot be used any longer. All actions assigned to or functions called by entities – that means all functions whose **my** synonym is not zero - are automatically terminated at their next **wait()** instruction. However the WDL script itself will keep running and won't change, even if a different script was assigned by WED to the new level.

One frame cycle after this instruction the level is loaded, two frame cycles after the instruction the entities in the new map are created and their actions are started.

send *(entskill);*
Through this instruction a client can update an entity skill or parameter on the server, or vice versa. The parameter of this instruction must be an entity skill, prefixed by the synonym of that entity. The same skill of the same entity on the other PC is set to the received value. This instruction is normally used to send key strokes and forces to a client player, or to send events - like having picked up an item - from the server to a client. Example:

```
.....send my.skill17;
```

send_string *(string);*
Through this instruction the given string is sent to the server and to all clients, including the sender, in a multiplayer game (some editions only). The **on_string** function is triggered on each PC which has received the string, the content of that string is replaced by the received string, and the **my_string** synonym is set to the string received. This instruction may be used for exchanging messages, but also for executing instructions on all clients simultaneously. Example:

```
inkey message_str;   // lets the user type a message
if (result == 13) {  // if [ENTER] pressed
   send_string message_str; }  // send that string to all players
// After that, message_str on all PCs contains the entered message
```

freeze_map *(bmap, varsize, varpos);*
Grabs the screen into a previously defined bitmap, reduced in size. ***bmap*** is the real name of the bitmap (as an exception, here a synonym may not be given). ***varsize*** specifies the factor of reduction (**1..16**), ***varpos*** the horizontal pixel position of the bitmap where the shrunk screen will be copied to. The bitmap may then be displayed as a **window** in a panel, to indicate screenshots of one or more saved games. The modified bitmap will be saved by **save_info,** but is available one frame cycle after the **freeze_map** instruction at the earliest (insert **wait(1)** between freezing and saving!).

bmap_width (*bmap*);
bmap_height (*bmap*);
Returns the width or height of the given bitmap in pixels. Example:

```
bmap splashmap = <logodark.bmp>; // the default engine logo in templates
panel splashscreen { bmap = splashmap; flags = refresh, d3d; }
...
// center the splash screen for non-640x480 resolutions
splashscreen.pos_x = (screen_size.x - bmap_width(splashmap))/2;
splashscreen.pos_y = (screen_size.y - bmap_height(splashmap))/2
```

switch_video *(varres, varbits, varscreen);*
Switches the screen to the resolution given by ***varres*** (**1..10**), to the colour depth given by ***varbits*** (8, 16, or 32) and between fullscreen (***varscreen*** = **1**) and window mode (***varscreen*** = **2**). 0 means no change for either parameter. The maximum colour depth and resolution depends on the edition. The colour depth cannot be changed during game play (see **video_depth**). 16 and 32 bit modes need **DirectX 7.0** or above, and use 3D acceleration hardware via **Direct3D**. 8-bit colour depth, on the other hand, will even work with an old low-end PC or laptop and with **DirectX 5.0**. In 16-bit and 32-bit modes, switching to a window is only possible if the desktop resolution is the same (high or true colour). The following screen resolutions can be given:

320x200 - 1
320x240 - 2 (default mode)
320x400 - 3 (not supported by some adapters)
400x300 - 4 (not supported by some adapters)
512x384 - 5 (won't run on some laptops)
640x480 - 6
800x600 - 7
1024x768 - 8
1280x960 - 9
1600x1200 - 10

If the new video mode is not available for any reasons, nothing will happen, but the **result** variable is set to **0**; otherwise, after a successful switching, it is set to **1**. The **video_mode**, **video_depth**, **video_screen**, and **screen_size** variables are set after the instruction to their new values.

exit;
Ends the game and shuts down the engine.

Plugin DLL instructions

The following instructions are for accessing DLL plugins and engine extensions (DLL = Dynamic Link Library). DLL plugins can be used by all GameStudio versions, but can only be made with the SDK (Source Development Kit) that comes with the professional edition. Details about creating DLL plugins can be found in the GameStudio Programmer's Manual.

A5 ***handle* = dll_open (*string*)**
Opens and activates a DLL plugin whose name is given by the ***string***, and returns a DLL handle. The predefined variable **dll_handle** is also set to that handle. DLL handles are only needed if more than one DLL plugin is used at the same time. The DLL file must really exist in the game directory, it can't be encrypted in a resource file. If the DLL is not found, **0** is returned. Example:

```
var effects_handle;
...
effects_handle = dll_open("effects.dll");
```

A5 **dll_close (*handle*)**

Closes the DLL with the given handle. The handle becomes invalid after this instruction. All open DLLs must be closed upon exit.

A5 **dll_exec (*string, argument*)**

Executes a function within the DLL given by the **dll_handle** variable, and transfers the argument to it. The ***string*** contains the function's name. The ***argument*** can be a number, a variable, or any WDL object like an entity, a **view**, a **text**, a **string** etc. The structure of important WDL objects will be given in the A5 header file that comes with the SDK, and described in the Programmer's Manual. If a vector or array is given as argument, only its first number is transferred. If the DLL function returns a value, it will be returned to the WDL script. Examples:

```
dll_exec("movebounce",my);
dll_exec("playmp3","mysong.mp3");
temp = dll_exec("squareroot",temp);
dll_exec("capitalize",my_string);
```

A5 **dll_exec_vec (*string, vector*)**

Executes a function within the active DLL, and transfers a vector or array to that function. Other than **dll_exec**, not the first number itself, but a pointer to the first number of the array is transferred to the function, from which the other numbers can be determined. Example:

```
var joyffback[2];
...
joyffback[0] = 20; joyffback[1] = -50;
dll_exec_vec("doforcefeedback",joyffback);
```

Debug instructions

The following instructions are for testing purposes:

beep;

Plays a short sound. Useful during function development, as it allows to quickly find out whether a certain instruction was reached or a certain function was executed at all. The sound can be given through the **beep_sound** name, e.g.:

```
sound beep_sound = <beep.wav>;
```

breakpoint;

Stops function execution and activates the **debugger**. The game freezes. On screen the current instruction or expression that follows the breakpoint is displayed in abbreviated form. By pressing **[Space]** the instruction is executed and the result (if any) is printed at the beginning of the line. The line scrolls upwards and the next line of WDL code becomes visible. By further pressing **[Space]** the function is executed line by line the results of each expression can be examined. The debugging output is displayed by the predefined text object **_debug_txt,** which can be placed somewhere else on the screen.

On pressing **[Ctrl-Space]** the debugger is deactivated and normal function execution will continue until another or the same breakpoint is reached again. On pressing **[Esc]** debugging is terminated for the current session, and all further breakpoints are ignored. Breakpoints are also

ignored in the runtime module. Note that when placing breakpoints into an event function, or into the first part of an entity action before the first **wait()**, funny things can happen with other entities. They are not updated during debugging, and thus may appear at wrong places (and stey there until they move again). This is no problem for debugging.
Conditional breakpoints can be placed by just entering an **if** condition, like

```
if (key_b != 0) { breakpoint; }  // breakpoint only if [B] is pressed
if (my == null) { breakpoint; }  // breakpoint only if MY is an empty synonym
```

For examining variables while debugging certain lines in the script, either a **debug panel** can be used (see tutorial, an example **debug.wdl** can be found in the **work** folder), or dummy expressions can be inserted like

```
temp = my.pan;
temp = vector.x;
```

Attaching functions or actions

Functions or actions can either be attached to entities (by WED), or to key or mouse events or panels (by WDL). The following script lines may be used to assign functions to keys or other events:

main = *function;*
The function specified is performed at the start of the game (default name = **main**).

on_join = *function;*
This function is executed on the server in a multiplayer game if a client joins the game. The predefined string **message** contains the name of that client.

on_leave = *function;*
This function is executed on the server in a multiplayer game if a client leaves the game. The predefined string **message** contains the name of that client.

on_string = *function;*
This function is executed in a multiplayer game if a string sent by **send_string** was received. The predefined string synonym **my_string** is set to the string received. This instruction can be used to display that string, or execute it as an instruction, or convert it to a variable.

on_mouse_left = *function;*
The given function is executed when the left mouse button is pressed. The **on_mouse_middle** and **on_mouse_right** names can be used in a similar way.

on_click = *function;*
The given function is performed when clicking left with the mouse pointer somewhere within a **view**, without hitting any object or panel.

on_mstop = *function;*
This function is triggered when the mouse pointer is active and the mouse was held stationary for ½ second. The variable **mouse_calm** may be used to specify a maximum distance in pixels that is still considered to represent immobility (default 3). By evaluating the variable **mouse_moving** you may also find out whether the mouse is held stationary (**0**) or is moving(**1**).

on_joy1 = *function;*

The given function is performed when the first joystick button is pressed. The **on_joy2, on_joy3**, and **on_joy4** names can be used in a similar way.

on_anykey = *function;*

The function is triggered when any key of the keyboard is pressed.

on_f1 = *function;*

The function specified is triggered when the **[F1]** key is pressed. The following keys may be assigned functions by the same way: **on_f2...on_f12, on_esc, on_tab, on_ctrl, on_alt, on_shiftl, on_shiftr, on_space, on_bksp, on_cuu, on_cud, on_cur, on_cul, on_pgup, on_pgdn, on_home, on_end, on_ins, on_del, on_pause, on_car** (Full stop), **on_cal** (Comma), **on_enter, on_0...on_9, on_a...on_z**.

The function names reflect the physical mapping on the keyboard, not the character actually printed onto that key. The names given here correspond to a German keyboard. On US keyboards, the Y and Z keys are swapped. The following special keys have alternative names for US and German keyboards:

Key	US	German (alternative name)
on_grave	[~`]	[°^]
on_minusc	[_-]	[?\] (on_sz)
on_equals	[+=]	[`'] (on_apo)
on_brackl	[{[]	[Üü] (on_ue)
on_brackr	[}]]	[*+] (on_plus)
on_bksl	[|\]	-
on_semic	[:;]	[Öö] (on_oe)
on_apos	["']	[Ää] (on_ae)
on_slash	[?/]	[_-] (on_minus)

All key, mouse, or joystick functions may be changed by other functions (eg. **on_f1 = *function*;**) during gameplay, so that the same key or button may start different functions at different times. For clarity of the WDL script it is, however, recommended to do this only to let the end user change his keyboard configuration.

Variables, Strings, Synonyms

There are some basic objects that can be defined in a WDL script for using them in functions.

Variables and Arrays

Variables and **arrays** store 6-digit fixed point numbers with 3 digits after the decimal. There are predefined variables that can be used by functions to change parameters of the game or the display. You'll find a listing of these predefined variables in a following chapter. Additionally any number of variables can be defined to store numbers for arbitrary purposes - state of health, combat prowess, magic skills, blood pressure, level of alcohol and so on... all this is left to the author's imagination.

A variable is defined as follows:

var *name*;
var *name* = *value*;
Creates a variable with given name. The alternative second line sets the variable to the given default value.

var *name*[*n*];
var *name*[*n*] = *value_1, value_2, ... value_n*;
Creates a variable that contains *n* numbers. In the alternative second line all numbers are given default values. Such a multi-number variable is called an **array**. The number *n* is called the **length** of the array. An array may have any length, but a maximum of 500 default values can be given in a WDL file. An array of length 3 is called a **vector**. Example:

```
var my_array[5] = 0, 10, 20, 30, 40;
```

This creates an array of 5 numbers, that can be accessed in expressions through **my_array[0]**… **my_array[4]**. In the expression the number in the **[]** brackets – the **index** - tells which one of the 5 numbers of the array is meant. Note that there is no number **my_array[5]**, as the index starts with **0**.

Within an expression, a var given without index is considered the first number of the array (**my_array == my_array[0]**). For using arrays as vectors, some meaningful abbreviations for the first three numbers can be used: **.x, .y, .z, .pan, .tilt, .roll, .blue, .green, .red.** So the following references to numbers of arrays of length 3 are equivalent:

```
my_vec.x == my_vec.pan == my_vec.blue == my_vec[0] == my_vec
my_vec.y == my_vec.tilt == my_vec.green == my_vec[1]
my_vec.z == my_vec.roll == my_vec.red == my_vec[2]
```

Using those abbreviations produces better readable and a little faster scripts, because then the index is not examined at run time against exceeding the array length. The length of an array can be accessed through its **length** parameter. The advantage of using an array, compared to defining single variables, is that any numeric expression also can be given as index. Example:

```
temp = 0;
while (temp < my.array.length) {
   my_array[temp] = temp;   // sets the array to 0,1,2,3,... etc.
   temp += 1;
}
```

Care must be taken that the index never exceeds its maximum value, 4 in this example. Otherwise an error message will be issued.

Vars can be redefined at any place within the WDL script. Only the last definition is used:

```
var v1 = 7;
...
var v1 = 13;
```

creates the variable **v1** with an initial value of **13**. The length of redefined arrays must not change.

A special kind of variables and arrays are created through:

var_info *name;*

Creates a variable or an array that is saved by a **save_info** instruction. Info variables are suited for user-settings like volume or screen resolution that are supposed to bet loaded at game start, and must not change when loading a previously saved game. You can dynamically alter a variable's info state by setting its **info** flag to **on** or **off**:

```
my_array.info = on;
save_info "state",0;
my_array.info = off;
```

saves the **my_array** variable into the file **state0.sav.** The info flag itself is not saved or loaded, it always keeps it's last value.

Please note: Earlier WDL versions knew variables and vectors under a different name: **SKILL**. If you encounter in an old WDL file a definition like this:

```
SKILL my_skill { X 10; Y 20; Z 30; }
```

an array of length 3 with default values of 10, 20, 30 is defined.

Strings

Strings are just a plain sequence of alphanumerical characters – letters, numbers or symbols - which can be used for messages, onscreen menus or the like. They are defined this way:

string *name* **= "***text***";**

Defines a character sequence with the given *name* and the content *text* between double quotation marks. Line feeds within the text are represented in C notation as **\n**, backslashes as \\, quotation marks as **\"**. If the text spreads over several lines in the script file, line feeds are inserted automatically. Like variables, strings can be redefined.

The **string** alone is not yet visible at the screen. To make it visible, you'll need a **text** object, which is dealt with in a further chapter.

Synonyms

Synonyms are **template**s for objects. They store objects like variables store numbers. Synonyms allow, by example, the same function to do the same with different objects, depending on what object is currently assigned to the synonym given. Synonyms are similar to pointers in a

programming language. Like with skills, there are some predefined synonyms; the most frequently used is the **my** synonym. Synonyms must be defined **before** they are used or referred to.

synonym *name* **{ ... }**
Defines a synonym with the name *name*. The winged brackets may contain the following names in the sequence given:

type *name;*
Must always be given, and refers to the type of synonym. *Name* may either be **entity**, **panel**, **bmap**, **string**, or **function**.

default *name;*
The object assigned to the synonym at game start. Can be omitted.

In functions, you can use synonyms like normal object names. Function synonyms themselves can be started by other functions. Synonyms can be assigned to each other:

```
synonym ent1_syn   { type entity; }
synonym ent2_syn   { type entity; }

   ...
   ent1_syn = my;
   ent2_syn = ent1_syn;
```

Some synonyms are predefined. You'll find a list of them in a further chapter. The two most important predefined synonyms are **my** for the entity that has started the current function; and **you** for the entity that has recently caused the current function in any way, e.g. by shooting at, or colliding with the entity the function was attached to, or by creating that entity. **you** will also be set by some instructions.

Two synonyms can be compared by the **if** instruction to check whether the same object was assigned to both. However synonyms cannot be compared with the object itself.

File Objects

Files may be named by names, in order to define bitmap, font, animation, sound or music objects, and thus make them controllable by functions. All file names that belong to the game must not exceed the 8.3 format (8 characters plus extension), and must be given without path in pointed brackets **<...>**. Pointed brackets are an indicator for WED's **publish** or **resource** function to include that file into the published game. The following file objects may be defined:

bmap *name* **=** *<filename>;*
bmap *name* **=** *<filename>, x, y, dx, dy;*
Assigns a bitmap - the content of a picture file - to the name. The file may be in **pcx** or **bmp** format, either true colour (24-bit RGB) or 256 colours (8-bit palettized). The alternative second line assigns a rectangular section from the picture to the name, making it possible to have several bitmaps collected in a single file. The coordinates ***x, y*** mark the upper left corner of the section, ***dx, dy*** give its width and height in pixels. Omitting the coordinates causes the entire bitmap to be assigned to the name.

If you encounter a bitmap error message on start, you've chosen a wrong format unsupported by the engine. All palletized bitmaps of a level must share the same colour palette, which must be given via WED map properties. However, each level may have its own palette. The first

colour of the palette (colour #0) must always be black, and the last colour (colour #255) must always be white. You can use true colour bitmaps even if the video mode of the engine is switched to 8-bit colour depth; in that case, however, there will be a visible loss of quality.

Bitmaps can be used for overlays. In that case the parts with colour index #0 of palettized bitmaps, or with the R, G, B components all below 8 (in case of true colour bitmaps) are transparent. All bitmaps used for surface textures in the map must be a power of two in horizontal and vertical size (e.g. 64, 128, 256…), no less than 16, and no more than 1024 pixels. Some 3D accelerators, like the Voodoo series, don't allow textures sizes of more than 256 pixels. The only exceptions are bitmaps used for the **`scene_map`**, which have to be huge in order to give a nice looking background, and may be up to 4096 pixels in size.

font *name* **=** *<filename>, width, height;*

Defines a fixed width character font out of a picture file. For the file format, the same restrictions as for **`bmap`** apply. ***Width, height*** give the size of a single character in pixels. All characters must have the same size. The bitmap can either contain 11 characters - numbers 0..9 and space - for numerical displays, or 128 or 256 characters for alphanumerical text.
The sequence of alphanumerical characters in the bitmap must correspond to the character set used by the WDL script, be it either DOS/ASCII or Windows/ANSI. In the bitmap, the characters may be ordered in four or eight rows. The number of characters and their arrangement is automatically determined from the character size and the size of the bitmap. But the bitmap size must **exactly** be either 11 times, 128 times, or 256 times the given character size.

Abb. 1 : Fontbitmap

sound *name* **=** *<filename>;*

Assigns a sound file in **`.wav`** format to the name. A **`wav`** file can be renamed to **`.wex`**; in this case the file is first looked for outside the resource. The format may be 11 or 22 kHz, 8 or 16 bit.

music *name* **=** *<filename>;*

Assigns a song file in **`.mid`** format to the name. The instruments must correspond to the general midi standard.

Entities - inhabitants of the game world

As mentioned before, entities are the dynamic objects, the actors, monsters, vehicles and other things who can move around in our game world. Additionally to the parameters which can be set in WED's entity property panel, there are much more properties which can be set, evaluated, or changed by a WDL action. These properties influence the entities' look, its reaction to certain events, and its moving or collision behaviour.

Entities can be created in three ways:

- by placing them in the level by WED,
- by creating them in the level at runtime through the **create** instruction,
- by defining them through an entity definition in the WDL script (models and sprites only).

Because the level is running on the server, entities that are placed into the level only exist on the server and on the client who has created them (if any). They don't have a name, so their parameters can only be changed by the WDL action that is attached to or triggered by them. The parameters must be prefixed by **my** or **you**. On leaving the level all level entities are deleted automatically, and their actions are terminated.

Script defined entities exist outside a level, on the server and on all clients. They can be visible even if there is no level loaded at all. Their parameters can be changed by any external function using the entity name given in the definition. They can be used to display 3D elements of the user interface, like a rotating compass, or a weapon carried by the player. Entities are defined the following way:

entity *name* { … }

Within the definition, any entity parameter can be given an initial value. Also a list of flags can be defined that are set at game start. Example:

```
entity shotgun_onscreen
 {
   type = <shotgun.mdl>;
   layer = 2;      // display above entities with layer 1
   flags = visible;// visible on screen from the start
   view = camera;  // same camera parameters as the default view
   albedo = 50;    // gouraud shading from sun position
   x = 100;        // place 100 quants ahead of the view
   y = -50;        // 50 to the right
   z = 0;          // and center vertically
 }
```

Screen control parameters

The following parameters are used only in entity definitions:

type = <*filename*>;

Here the default name of the entities' **mdl** file is given in angular brackets. Only model entities can be defined. The file of the entity can be changed at runtime through a **morph** instruction.

layer = *number*;

Determines the order of the entity, if it is visible on screen and overlaps with other objects like panels or texts. Elements with higher **layer** value will be placed over elements with lower **layer** value. The **layer** parameter (default 0) cannot be changed at runtime.

view = *viewname*;

The **view** determines the camera parameters and clipping rectangle for the rendering the entity onscreen. If no **view** is given, the default **view** (**camera**) is used. The **view** must not necessarily be visible for that purpose. The entity position is given in quants relative to the center of the view, with the X axis pointing ahead towards the screen.

Position parameters

The following parameters are used for positioning the entity:

x, y, z

The position of the entities' centre in the map coordinate system (in quants). The **x** parameter can also be used as a position vector . An entity can be teleported or moved without collision detection by simply changing these values. For defined entities, these parameters give the position relatively to the view's eye position.

pan, tilt, roll

The entities' Euler angles that describe rotations about its Z, Y, and X axis (in degrees, 0..360). An entity can be rotated by changing these values.

Visual Parameters

The following parameters influence the entities' appearance in the level:

A5 **lod1, lod2, lod3**

These parameters control the discrete geometric levels of detail - **LOD** - of an entity (not available in all editions). LODs are used for increasing the frame rate. The rendering of a detailed model with several thousands of polygons looks quite good when the model is close to the camera. However when the same model is far from the eye point, details are less noticeably. A simpler model with few polygons will look just as good in that situation, but render faster. The difference in rendering speed is quite remarkable in huge outside levels where lots of entities are visible, like trees in a wood, or an ork army, or a city made from house map entities.

To get the best of both worlds, each entity can be given up to 4 model or sprite files, which change automatically dependent on the distance of the entities' centre to the camera. The three further files can be assigned through the keywords **lod1**, **lod2**, and **lod3** in an entity definition, like:

```
entity lodgun {
   type = <gun.mdl>;     // used for near model
   lod1 = <gunsimp1.mdl>;  // used above 12.5% clip_range, and for the near shadow
   lod2 = <gunsimp2.mdl>;  // used above 25%, and shadow above 12.5% clip_range
   lod3 = null;  // invisible above 50%, and no shadow above 25% clip_range
}
```

The above example defines the LOD behaviour for all **<gun.mdl>** entities in the game. The entity switches to LOD level 1 at a distance of 1/8 the **clip_range**, to level 2 at ¼ the **clip_range**, and to level 3 at ½ the **clip_range**. For preventing repeated "popping" between two LOD levels at critical distances, the LOD levels change with a hysteresis value of 15%. If the entity is further away than the **clip_range**, it won't be displayed at all. LODs can also be used for creating 4 level MIP mapping for sprites.

The LOD levels are valid for all entities that use the same model. For proper animation, they should be of the same kind; an animated model should not change into a map or a sprite. The number of frames and the frame names must be the same on all LOD levels. If **null** is given instead of a file name, the entity is not visible above the LOD level. This way, sub-entities of an entity composed of several parts can be suppressed above a certain distance. If a model has several LOD levels, it's **shadow** is calculated from one further level than the displayed model. This way dynamic shadows can now be rendered remarkably faster. If the model is further away than 50% of the **clip_range**, no shadow is rendered at all.

scale_x, scale_y, scale_z

The scale factors, which determine the entities' size in quants per pixel (only valid for sprite and model entities). They can be changed during gameplay to grow or shrink the entity in real time. Their default value is 1.0, which means that a pixel of the entities' texture has the size of 1 quant. Note: Changing an entities' size will also change it's bounds, but doesn't change its collision hull (see next chapter). If scaled very small, it may be necessary to raise its center to prevent that the entity is stuck in the floor.

frame, next_frame

Model and sprite entities can consist of any number of frames in order display animation. In case of sprite entities, the animation frames must be of the same size, side by side in the graphics file. The number of sprite frames must be given at the end of the file name after a '**+**' sign, like this (the total length of the file name must not exceed the 8.3 format):

Fig. 2 : Animated Sprite EXPLO+7.PCX

There are two ways to animate a model or sprite entity. If the entity consists of several frames, but has no action attached, it will cycle automatically through its frames with a rate of 8 frames per second for models, and 12 frames per second for sprites. If it has an action attached, it won't animate by itself. Instead, it's **frame** value (1.. number of frames) gives the number of the current frame. Thus by permanently adding **time** to the **frame** parameter the model can be animated, e.g.:

```
my.frame += 1.5*time;       // gives 16/1.5 = 12 frames per second
if (my.frame > 10) {
     my.frame = 1; }        // don't exceed the frames
```

Alternatively, for a walking model, the distance covered could be added to **frame**, instead of a time factor.

If the **frame** parameter has a fractional part, the model's shape will be interpolated between the frame number given by the integer part, and a target frame given by the **next_frame** parameter. If **next_frame** is set to **0**, **frame + 1** will be taken as target frame. This way, a model can smoothly morph forwards or backwards between arbitrary frames.

Example for a loop function that uses **next_frame** to smoothly animate a model between frames 20 and 30:

```
my.frame = 20;            // set to start frame
while (1) {
   my.frame += time;         // 16 frames per second
```

```
    if (my.frame > 30) { my.next_frame = 20; } // interpolate to start frame
    else { my.next_frame = 0; }      // interpolate to FRAME+1
    if (my.frame >= 31) { my.frame -= 10; }    // if end frame reached, skip back
    wait(1);
  }
```

Normally a model entity is animated through the **ent_frame()** and **ent_cycle()** instructions.

A5 **u, v**

The pixel offsets of map entity textures, in horizontal and vertical direction. By changing these offsets in real time, effects like streaming water can be achieved. Example:

```
action water_current   // assign to a water block
{
  while (1) {
    my.u += 5*time;
    my.v += 5*time;
    wait (1);
  }
}
```

skin

If a model entity has multiple skins, they can be changed through its **skin** parameter (1...number of skins). If the entity has no action attached, it will automatically cycle through its skins at 12 cycles per second.

ambient

Through this parameter (-100..+100, default 0), an entities' texture can be made appear darker or brighter. Additionally, model entities are gouraud shaded, according to their position to the sun azimuth and elevation. Map entities are flat shaded, also according to the sun position, if they have flat albedo textures.

albedo

The reflectivity of map or model entities or terrain to sunlight (0..100, default 50). The bigger the **albedo**, the darker the shadows and the brighter the lit parts of the model. Transparent models placed in the level have a default albedo of 0.

lightrange
lightred, lightgreen, lightblue

These values are used for colouring onscreen entities, and for emitting coloured dynamic light by level entities. In the latter case the **lightrange** parameter will give the range of the light (0..2000 quants, default **0**); for onscreen entities **lightrange** must not be used. The **lightred, lightgreen, lightblue** parameters (0..255, default 0) will give the brightness of the red, green, and blue components of the light's colour. To extinguish the light emitted by a level entity, **lightrange** must be set to **0**.

The number of entities that may emit dynamic light simultaneously is restricted to 32. If more try, the 'oldest' light will be extinguished. Dynamic lights can be used for explosions, rockets, fireballs, or for monsters carrying torches, and will illuminate all shaded surfaces and entities within their range. In 8-bit mode, the light is always white, and it's brightness is calculated from the largest of its red, green, or blue colour components.

Dynamic lights will influence the frame rate, so they should only be used for temporary effects. The more lights, and the bigger their range, the slower is the rendering. This effect is much smaller in 16-bit or 32-bit modes, but in 8-bit mode dynamic lights can effectively halve the frame rate. The quality of dynamic light rendering in 16-bit or 32-bit modes can be increased by

setting the **d3d_lightres** variable to **1**. This doubles the resolution of the light spot at surfaces, but also increases the rendering time.

A5 alpha

Controls the transparency of translucent entities in 16 or 32 bit mode (not all editions). The parameter (default 50) gives a percentage. At 0 the entity is totally transparent, at 100 it is totally intransparent. This way entities can smoothly be faded in and out. The **transparent** flag must be set for **alpha** to have any effect. Example for fading in an entity:

```
my.transparent = on;
my.alpha = 0;
while (my.alpha < 100) {
   my.alpha += 5*time;
   wait(1);
}
my.transparent = off;
```

visible

Only for script-defined entities: If this flag is set to **on**, the entity will be visible onscreen.

invisible

If the **invisible** flag is set to **on**, the entity will be invisible in the level, but still be an obstacle for collision detection.

hidden

If the **hidden** flag is set to **on**, this entity is only visible on the client who has created it. For all other members of a multiplayer system it is invisible. This flag can be used for weapon models in a multiplayer system.

transparent

If this flag of a sprite, model or A5 map entity is set to **on**, the entity will be translucent, so you can see through it. Transparent water can be created by placing a transparent water map entity into a pool (and the water level can easily rise or fall this way).

overlay

If a model entities' **overlay** flag is set to **on**, the black parts of the entities' skin will not be drawn.

flare

If a sprite or A5 model entities' **flare** flag is set to **on**, the entity will assume alpha transparency. The darker parts are more transparent than the brighter parts. This way explosions, light effects or camera flares can be produced. If **transparent** is set at the same time, the transparency is reversed, i.e. the brighter parts are more transparent than the darker parts, like for black smoke. Only valid for sprite entities, and only in 16-bit or 32-bit modes. The **flare** flag must be switched on at the beginning of the first action of the entity. All entities that share the same file must have the same alpha transparency.

A5 bright

In combination with **flare** or **transparent**, blends the entity over the background instead of mixing both. This way the entity appears illuminated, like fire or sparks. In combination with **unlit** the entity receives an average environment light as by normal sunlight in the level.

nofog

If this flag is set, fog becomes much more transparent for that entity.

light

If this flag is set, the entity is illuminated by it's own dynamic light colour (**lightred**, **lightgreen**, **lightblue**) in D3D mode even if it's **lightrange** is **0**. This way entities can be dynamically colored. The restrictions of dynamic lights (max. 32) don't apply here. Example:

```
function flicker_red(value) {
  my.lightrange = 0;   // don't illuminate the environment
  my.light = on;       // illuminate myself
  while (my.light == on) {
    while (my.lightred < value) {
      my.lightred += 10 * time;
      my.lightred = min(my.lightred,255);  // clip to 255
      wait(1);
    }
    while (my.lightred > 0) {
      my.lightred -= 10 * time;
      my.lightred = max(my.lightred,0);  // clip to 0
      wait(1);
    }
  }
}
```

nofilter

If a sprite or model entities' **nofilter** flag is set to **on**, the entity is drawn without bilinear filtering and antialiasing in D3D mode. Thus the texture won't get that blurry look. This is useful to avoid the black sprite seams produced by some 3D cards (Voodoo) due to antialiasing to the black transparency colour.

unlit

By setting an entities' unlit flag, the entity is not lit by it's environment. Only it's own **ambient** value, and its **bright** flag influences the brightness. By setting the **ambient** to -100 the entity becomes totally black. Large entities placed in the level, with a diameter of 250 quants or more, get the **unlit** and **bright** flag set by default.

shadow

By setting a model's **shadow** flag, it's shadow is projected with 50% transparency onto the floor (not supported by all editions). The projection direction is given by the sun azimuth and elevation (**sun_angle**). The shadow is visible even if the model is invisible. Projecting a shadow can increase the model rendering time by around 50%. The **shadow** flag can be set by WDL as well as by WED.

Dynamic shadows only work in levels that don't contain non-shaded (flat or totally dark) surfaces. This restriction will be overcome by the next update.

oriented, facing

If a sprite entities' **oriented** flag is set, the entity will be oriented in world space according to its **pan**, **tilt**, and **roll** angles, rather than facing the player. This flag can be used to create flat objects like fences or transparent windows, or to place a shadow sprite flat onto the ground.
If a sprite entities' **facing** flag is set, the entity will always face the camera. This flag can be used for spherical objects, like fireballs or explosions.

If neither flag is set, the sprite will look like a billboard, standing upright, and horizontally always turning to the player.

near

If a model or sprite entities' **near** flag is set, the entity will much less be clipped away by walls or other entities. This flag can be used for weapons carried in front of the camera, for explosions, or for the player in 3rd person view.

Collision detection

Depending on which entity types collide with which other, two different methods of **Collision detection** are used. For this each entity has both a **Bounding Box** and a box-shaped **Hull**. The hull is used for collisions of a moving entity against the level or map entities. This kind of collisions detect each polygon of the target entity. For all other types of collisions, the bounding boxes of the two entities are tested against each other. Bounding box as well as hull won't rotate with the entity, they are always xyz axis aligned.

While the bounding box can have an arbitrary size, there are only three possible hulls: fat, narrow and point sized. The narrow hull is a cube of 32 quants edge size around the entity's centre. The fat hull is a box of 32 quants vertical and 64 quants horizontal size. These sizes match most requirements, but for some special cases they can be changed. Unlike the bounding box, they are not given individually per entity, but only per level, by entering the following options in WED's Build **Shading options** line:

-narrow *hsize vsize voffset*
-fat *hsize vsize voffset*

The first number (***hsize***) is the horizontal size of the hull, the second (***vsize***) is the vertical size, and the third (***voffset***) is the vertical offset of the hull relative to the entities' origin. For instance, **-narrow 32 32 0 -fat 64 48 8** gives the default hulls. All map entities should be built with the same hull sizes as the level. 4 predefined vectors contain the current level's hull box sizes for evaluation in WDL functions: **hull_narrowmin[3]**, **hull_narrowmax[3]**, **hull_fatmin[3]**, and **hull_fatmax[3]**. They contain the **x**, **y**, **z** minimum and maximum values of the narrow and fat hulls.

For sprite entities, its origin is the geometric centre of the bitmap. For map or model entities, the origin is its coordinate origin given by the editor. If you want the model to be able to climb stairs, simply place its origin into a higher position of its body. The highest step the model can climb is given by the difference between its lower hull border and it's feet. If you don't want the model to climb at all - if it's a car, by example - place its origin into a low position. But be careful not to place it too low, otherwise its hull may penetrate the floor, and the model won't be able to move at all. The origin is also the reference point for model rotations.

The following parameters influence individually the entities' collision behaviour.

min_x, min_y, min_z, max_x, max_y, max_z

The bounding box corners of the entity, relative to the entities' origin, used for collision detection against other sprite or model entities (for collision detection against the level or map entities not the bounds, but the hull is used). The bounds are initially set to some default values, but can be changed. Take care that when they are increased, and another entity is nearby, the latter might get stuck. Also they are reset to their default values each time the entities' model or scale is changed. Thus they can be changed one framecycle after creating the model the earliest. They are used often by functions to determine the entities' vertical size, which is (**max_z - min_z**).

fat, narrow

Flags, used for collision detection against the level or against map entities. Determine which hull, if any, is used by the entity. If an entity is larger than 64 quants, its **fat** flag automatically is set to **on** at game start. Otherwise its **narrow** flag is set. If the entity is smaller than 8 quants, like a bullet, neither of the flags is set – a point hull is used then. You can set or reset **fat** and **narrow** manually, to influence the collision hull used by the entity. In some cases you'll want the entity to choose the narrow hull instead of the fat one, to pass through small doors. **fat** and **narrow** must never be set both at the same time. They are set automatically if the entities' scale is changed, or it is **morph**ed.

Example for manually assigning a player a narrow hull so that he fits through doors:

```
my.fat = off;
my.narrow = on;
```

passable

If the **passable** flag of an entity is set to **on**, its collision detection is switched off. It will be like a ghost, moving through walls, and other entities can move straight through it. Collision events are not triggered.

push

Through the entities' **push** value you can determine more specifiaclly than with **passable** whether the entity will be an obstacle to another one or not. The player entity, by example, must not be an obstacle for an elevator cabin entity, otherwise the cabin with the player within would never move. An entity with a higher **push** value (default = **0**) will consider an entity with a lower **push** value not an obstacle, and run straight over it.

Please note that entities will not be pushed out of the way automatically. Pushing out of the way, if desired, can be performed by the event function (see below). The **push** values of doors and elevators are preset to 10 by the **doors.wdl** functions.

trigger_range

Gives the range within **event_trigger** will be triggered by a bypassing entity, which itself has **trigger_range** set to non-zero. Default = **0**.

Events

An entity can be made sensible to certain game events, like collisions, being shot, or being clicked with the mouse. If such things happen to an entity, each time it's **event** function is started.

event

The **event** function can be set by the entities' main action, e.g:

```
my.event = my_function;
```

The function starts as soon as a certain event happens with the corresponding **enable** flag set to **on**. At the beginning of the **event** function, the predefined variable **event_type** can be checked to determine which kind of event happened. So for each type of event there is an **enable** flag to make the entity sensitive for this event, and a certain value of the **event_type** variable. Depending on that, the **event** function then can let the entity react on the event, by giving way, firing back, exploding or the like.

Event functions are actually executed immediately during the instruction of another entity that caused the event, like a **move**, **scan**, or **trace** instruction. The event function itself normally should only transfer information to the entity's main function – it shouldn't perform instructions that can trigger events itself, displace entities, or change anything else in the level. Thus instructions like **move, create, remove, trace** etc. must be avoided here! Otherwise all sorts of bad things can happen, like two entities endlessly triggering each other's event (the game could freeze in that case). If for some reason the event function must perform such 'critical instructions', they must be preceded by a **wait(1)** for delaying them to the next frame. Then it's safe.

The following events can trigger the **event** function, with the variable **event_type** set to a number that indicates the type of event and can be given as one of the following pre-defined keywords for better readability.

event_block - enable_block

Collision with a surface of the level during a **move** instruction. The entity must not be **passable** and it's **push** value must be 0 or less. On start of the **event** function, the vector **normal** is set to a direction perpendicular to that surface, and the vector **bounce** is set to the direction into which the entity would bounce off. Example:

```
var angle[3];

function bounce_event()
{
  if (event_type == event_block) {
    play_entsound my,whamm,50;
    to_angle angle,bounce;  // bounce off the surface
    my.pan = angle.pan;    // let my face the new direction
    my.tilt = angle.tilt;
  }
  if (event_type == event_entity) {
     ...
  }
  // etc. ...
}

action bounceball
{
  ...
  my.enable_block = on;  // make entity sensitive for block collisions
  my.enable_entity = on; // make entity sensitive for entity collisions (see below)
  my.event = bounce_event;
  ...
}
```

event_stuck - enable_stuck

Caught in a corner during **move**, unable to move further. The entity must not be **passable**.

event_entity - enable_entity

Collision with another entity during **move**. The entity must not be **passable** and it's **push** value must be equal or less than the **push** value of the other entity. The **you** synonym is set to the other entity, **normal** and **bounce** are set as before.

event_impact - enable_impact

Hit by another entity that performed a **move** and had a lower or equal **push** parameter. Neither entity must be **passable**. **You** is set to the other entity. **Normal** and **bounce** are also set to give the direction pependicular to the moving surface and the direction into which the entity would bounce.

event_push - enable_push
Run over by another entity with a higher **push** parameter. Neither entity must be **passable**. **You** is set to the pushing entity. **Normal** and **bounce** will be set as usual. **Event_impact** won't be triggered in this case.

The event function may now control the entities' make-way behaviour. If the pushed entity now performs a **move** instruction, it will not trigger a collision with the pusher again, because the **you** entity is excluded from the **move** instruction's collision detection.

If two entities collide, each one's **event** function is triggered, the moving entities' one with **event_entity**, the hit entities' one with either **event_impact** or **event_push**. If both entities were moving, each one can get two different events in undefined order.

event_click - enable_click
Clicked on with the left mouse button.

event_rightclick - enable_rightclick
Clicked on with the right mouse button.

event_touch - enable_touch
Touched with the mouse.

event_release - enable_release
The mouse was removed from over the entity.

Mouse events will only be triggered if the entity is within a distance given by the predefined **mouse_range** variable (default 1000 quants) from the camera position.

event_scan - enable_scan
Scanned by a **scan_entity** instruction. If **scan_entity** was performed by an entity, **you** is set. **Result** is set to the distance to the centre of the scan cone.

event_detect - enable_detect
Performed a **scan_entity** instruction and found an entity with **enable_scan** set within the scan cone. **You** is set to that entity, and **result** is set to the distance to that entity. For each entity found the event is triggered separately.

event_trigger - enable_trigger
Another entity performed a **move** within a certain range. **You** is set to the triggering entity. The range is the overlap of both entities' **trigger_range** parameters. If both entities' **trigger_range** is zero, no function will be triggered.

event_shoot - enable_shoot
Hit by a **trace()** instruction with **activate_shoot**. **You** is set to the tracing entity, if any.

event_sonar - enable_sonar
Hit by a **trace()** instruction with **activate_sonar**. **You** is set to the tracing entity, if any.

event_disconnect - enable_disconnect
The entities' client has disconnected and left the multiplayer game. The **event** function can be used to remove the entity.

After start of an **event** function, all event-dependent variables and synonyms, like **NORMAL** etc., keep their values only until the next **wait()** instruction. During the wait pause they can (and

certainly will) be changed from other functions. If you want to keep them longer, copy them into entity skills. For clarity, let the entities' main action do most of the work, and keep the **`event`** function as short and simple as possible, without any **`wait()`** instructions.

Internal parameters

The following parameters have no influence on the look or collision behaviour entity itself, but can be used by functions:

client
In a multiplayer game each entity has a unique **`client`** parameter. If the entity was created by a client, the parameter is set to this client number; otherwise it is **`0`**. It can be used to determine whether two entities were created by the same client, or by the server. If an entity was created by another entity, it inherits the **`client`** parameter.

skill1 ... skill48
General-purpose numerical variables for use in functions. The first 8 entity skills can be set by WED; their meaning also depends on the entities' action. The remaining skills can be used to store internal numerical properties, like speed or target position. Each three consecutive entity skills can be used together as a vector, which is given by giving the first of the three skills.

flag1 ... flag8
General-purpose binary flags for use in actions. Can also be set by WED.

entity1, entity2
Entity synonyms for use in actions.

string1, string2
String synonyms for use in actions (eg. to display a text if the mouse touches the object).

panel1, panel2
Panel synonyms for use in actions (eg. to assign an inventory overlay picture).

User interface: Panels, Texts, Views

At game start the camera view fills the entire screen; without any WDL script, no menu or any user interface is visible. The user interface must be put together via WDL by way of panels, texts, further 3D views, and entities that are visible onscreen.

Panels

Panels are rectangular areas with a background pattern or overlay, and optional instruments or controls on it. They can be used for cockpits, dashboards, inventories and inventory items, buttons, splash screens or images. They are defined the usual way:

panel ***name*** **{ ... }**
Defines a display panel with the name ***name***. Example:

```
bmap compass_map = <compass.pcx>;
panel aircraft_pan
{
   pos_x  = 4; pos_y = 4;
   digits = 0,0,4,digit_font,1000,player._rpm;
   digits = 60,0,4,digit_font,1,player._speed_x;
   digits = 120,0,4,digit_font,1,my_height;
   window = 200,0,40,20,compass_map,compass_pos.x,compass_pos.y;
   flags = refresh,visible;
}
```

A panel definition may contain - in given sequence - the following parameters:

bmap = ***bmap*****;**
Bitmap name for the panel background. The size of this bitmap determines the size of the panel. Normally it will be drawn only once. If the panel is moved over the screen, i.e. the panel position is changed, the background bitmap will be redrawn. This way a panel can be used for a 2-D sprite.

Panels may be of any size. When the panel bitmap is larger than the restrictions given by the 3D card, it is automatically split and rendered in several parts by the engine.

layer = ***number*****;**
Determines the order of the panel, if it overlaps with other objects. Elements with higher **`layer`** value will be placed over elements with lower **`layer`** value. The **`layer`** parameter cannot be changed during gameplay.

pos_x = ***number*****; pos_y =** ***number*****;**
Distance of the upper left border of the panel from the upper left border of the screen. These values can be changed during gameplay in order to move the panel over the screen.

A5 **alpha =** ***number****;*
Determines the transparency of a translucent panel in 16 or 32 bit mode (not in all editions). At **`0`** the panel is totally transparent, at **`100`** it is totally intransparent. Panel elements like buttons inherit the **`alpha`** value from the panel. This way panels, buttons etc. can smoothly be faded in and out. The panel's **`transparent`** flag (see below) must be set. Example for fading in a panel:

```
my_panel.transparent = on;
my_panel.alpha = 0;
```

```
while (my_panel.alpha < 100) {
   my_panel.alpha += 20*time;
   wait(1);
}
my_panel.transparent = off;
```

flags = ***flag1, flag2...;***

Here a list of all panel flags may be given, which are to be set at game start. All other flags are **`off`** by default. The following flags may be set:

visible

Only by setting this flag the panel will appear on the screen.

overlay

If this flag is set, the colour 0 (black) of the **`bmap`** won't be drawn, so that the panel background appears as an overlay.

transparent

If this flag is set, the panel background and buttons will be drawn semi-transparent over the screen.

refresh

If this flag is set, the panel is redrawn at every frame cycle. This is required for displaying a panel above a view, or having a text scrolling across the panel without overwriting the background. Without **`refresh`**, the panel is only redrawn if its position is changed, and its elements are only redrawn if the corresponding variable is changed. Especially on huge panels constant redrawing costs rendering time and reduces the frame rate.

d3d

If this flag is set, the panel is rendered by hardware in 16 or 32 bit mode. Otherwise it is rendered by software. Hardware rendering is faster and looks better, but eats valuable texture memory.

A panel may be given several sub-elements, for composing cockpits or menus. Within the following element definitions the given ***`x, y`*** positions refer to the distance of the upper left corner of the respective element to the upper left edge of the panel:

button = ***x, y, bmapon, bmapoff, bmapover, actionon, actionoff, actionover;***

Defines a button on the panel. The size of the given button corresponds to the size of the bitmap with the name given by ***bmapon***. This bitmap will be visible if the left mouse button is pressed over the button. ***Bmapoff*** will be visible if the mouse is over the panel, but not over the button. ***Bmapover*** will be visible as long as the mouse is over the button.

ActionOn will be executed if the mouse is clicked over the button; ***actionoff*** will be executed if the mouse is hold for a certain time and then released over the button; and the ***actionover*** will be executed if the mouse touches the button. Real function names must be given here, not synonyms. Except for ***bmapon***, which determines the size of the button, each of the bitmap and function names can be replaced by **`null`**.

Several buttons on the same panel may share the same function, if it is defined with a parameter. The number of the button that triggered the function is delivered as parameter to that function. The number 1 corresponds to the first button in the panel definition. Example:

```
function count_up_skill(button_number)
{
   play_sound click,50;
   if (button_number == 1) { player.skill40 += 1; }
   if (button_number == 2) { player.skill41 += 1; }
   if (button_number == 3) { player.skill42 += 1; }
}

panel skill_pan {
   …
   button = 0,0,on_map,off_map,off_map,count_up_skill,null,null;
   button = 0,10,on_map,off_map,off_map,count_up_skill,null,null;
   button = 0,20,on_map,off_map,off_map,count_up_skill,null,null;
   …
}
```

vslider = ***x, y, len, bmap, min, max, var*****;**
hslider = ***x, y, len, bmap, min, max, var*****;**

Vertical or horizontal slider, which can be dragged with the mouse for entering values. The number ***len*** gives the height or width of the slider range in pixels. The bitmap ***bmap*** is used for the slider button, that is dragged with the left mouse button pressed. The ***min*** and ***max*** numbers determine the value range of the slider. The variable is set to a value within that range dependent on the current slider position. If moved to the upper or left position, ***var*** is set to the ***min*** value, at the lower or right position it is set to the ***max*** value. Example:

```
hslider = 10,10,40,slider_map,1,8,my_var;  // shifts my_var from 1 to 8 over 40 pixels
```

vbar = ***x, y, len, bmap, factor, var*****;**
hbar = ***x, y, len, bmap, factor, var*****;**

Vertical or horizontal bar graph display for the graphical representation of a non-array variable in the panel. The bitmap is shifted vertically or horizontally depending on the variable value. ***Len*** is the vertical or horizontal size of the bar in pixels. ***factor*** is a fixed point number, which when multiplied by the ***var*** variable results in the shifting of the bitmap in pixels. ***Bmap*** must be a minimum of (***len*** + ***factor****(var value maximum)) pixels in vertical or horizontal size.

window = ***x, y, dx, dy, bmap, varx, vary*****;**

Displays a cutout 'window' from a bitmap. The numbers ***dx*** and ***dy*** give the size of the cutout in pixels. The source bitmap ***bmap*** must not be smaller than ***dx*** and ***dy***. Both ***varX*** and ***varY*** give the position of the cutout window on the bitmap in pixels, relative to the upper left corner. The window may only be placed inside the bitmap's borders.

digits = ***x, y, len, font, factor, var*****;**

Numerical display. ***Font*** is a previously defined character set either consisting of 11 characters (numbers 0..9 and space) or of 128 or 256 characters in ASCII order. The integer part of the non-array variable ***var*** is shown with ***len*** digits in the panel. The variable value is first multiplied by the ***factor***. Instead of a variable, the name for any numerical parameter of any object can be given.

Leading zeros are suppressed. If the font does not contain a 'minus' character, negative values are not shown. Instead of displaying numbers, you could use a special font bitmap to display symbols with a one-digit display.

mouse_map = ***bmap*****;**

Alternative mouse pointer within the panel. If this name is not given, the mouse pointer keeps its normal look.

on_click = ***function;***
Instead of the 'global' **`on_click`** function, this given function is performed by left clicking with the mouse pointer anywhere within the panel bitmap.

Texte

Texts are used to display strings for menus, for messages or for the dialogue with actors.

text ***name*** **{ ... }**
This defines a formatted text named ***name***. A text definition may contain - in given order - the following parameters:

layer = ***number;***
Determines the order of the text, if it overlaps with other screen elements. Elements with higher **`layer`** value will be placed over elements with lower **`layer`** value. The **`layer`** parameter cannot be changed during the game.

pos_x = ***number;***
pos_y = ***number;***
Distance of the upper left border of the text from the upper left border of the screen. The position may be outside the screen, but then only the visible part of the text is diyplayed. The position of the text may be changed during the game. This way longer texts may be scrolled horizontally or vertically over the screen.

size_y = ***number;***
Height of the displayed text in pixels (default and maximum = height of screen). The text is only displayed within the vertical screen area of **`pos_y`** and **`pos_y+size_Y`**.

offset_y = ***number;***
Number of the first pixel line of the text, which will be displayed on **`pos_y`** position. This parameter allows you to scroll the text vertically pixel by pixel.

strings = ***number;***
Maximum number of strings this **`text`** may contain. This parameter cannot be changed during gameplay. The strings will be displayed one under each other. Multiple strings are especially useful for menus.

font = ***font;***
The character set for the text; must contain either 128 or 256 characters. This parameter cannot be changed during gameplay.

char_x, char_y
Size of a single character of the **`font`** in pixels. These values can't be changed, but may be used in functions to determine the bounds of the text display.

string = ***string1, string2...;***
The actual text, containing of one or more strings (define maximum number in **`string`**s beforehand!). ***String*** may be changed by functions during gameplay. If a string name **`null`** is assigned, no text is displayed. If the text contains more than one string, individual strings can be accessed through a **`[]`** index suffix, e.g.

```
my_text.string[7] = new_string;
```

dataview = *name*;
Assigns a **dataview** object to the text (professional edition only). The text contains all such string fields of the dataview's database, whose field number corresponds to the **viewpos** parameter. Only the strings of selected records are displayed, and the **index** parameter counts only the selected records.

viewpos = *number*;
Field number of the database string field to be displayed in the text.

A5 **alpha =** *number*;
Determines the transparency of a translucent text in 16 or 32 bit mode (not in all editions). At 0 the text is totally transparent, at 100 it is totally intransparent. This way texts can smoothly be faded in and out. The text's **transparent** flag (see below) must be set.

flags = *flag1, flag2...;*
Here a list of all text flags may be given, which are to be set at game start. All other flags are **off** by default. The following flags may be set:

center_x, center_y
Setting one or both of these flags causes the text to be centered around **pos_x** horizontally and/or **pos_y** vertically. If the respective flag is not set, the text will be displayed justified left or up.

condensed
Setting this flag causes the text to be 'compressed' horizontally by 1 pixel per character. Especially fonts in italics look better that way.

narrow
Similar to **condensed**, but the text is compressed further.

transparent
If this flag is set, the text will be drawn semi-transparent over the screen.

visible
Only by setting this flag the text will appear on the screen.

Views

For the creation of rear mirrors, missile cameras or the like, or for single-PC multiplayer games, multiple 3D views into the level map can be displayed. One view, named **camera**, is already predefined at game start. Each further view can be created through the **view** definition:

view *name* **{ ... }**
defines a view named *name*. Within the **view** definition the following parameters can be given:

layer = *number*;
Determines the order of the view, if it overlaps with other views. Views with higher **layer** value will be visible over views with lower **layer** value. Panels and texts will always be displayed over all views independently of their **layer** values.

pos_x = ***number*****;**
pos_y = ***number*****;**
Distance of the upper left border of the **view** from the upper left border of the screen. The position must be inside the screen, and also the size of the **view** must be chosen so that no part of it is outside the screen.

size_x = ***number*****;**
size_y = ***number*****;**
Horizontal and vertical size of the view in pixels. In software mode, the size must be a multiple of 4 in horizontal and 2 in vertical direction.

x, y, z
Position of the view in the map. Can only be set by functions. The **x** position can also be used as a vector.

pan, tilt, roll
Angles of the **view** in the map. Can only be set by functions. The **pan** angle can also be used as a vector.

arc = ***number*****;**
Angle, which gives the field of view; it's default value of 60 degrees corresponds to the human eye. By decreasing this angle you'll get a telescope effect. *A5* If the arc of a view has a negative value, the view is horizontally flipped in 16 or 32 bit mode. This way rear view mirrors can be realized.

aspect = ***number*****;**
Determines the vertical-to-horizontal pixel size ratio. The default value of 1 gives a normal ratio of 1:1.

A5 **offset_x =** ***number*****;**
A5 **offset_y =** ***number*****;**
Defines an eye point offset in horizontal and vertical direction. At **0** the eye point is in the center of the view window as before; at **-0.5** it is at the left or upper edge, at **0.5** it is at the right or lower edge of the view window. It must not be outside the view edges. The eye offset can be used for driving or flight simulators where the driver or pilot sits at the right or left side of the screen. Example:

```
camera.offset_x = -0.33; // set the eye point for a pilot in a two-seater
```

ambient = ***number*****;**
Light value (**0..100**, default **0**) added to everything which is seen in this **view**. Can be used, by example, to create an infrared or radar view, or to change the overall brightness of the map.

fog = ***number*****;**
Strength (**0..100**, default **50**) of the coloured fog effect for this **view**. Only effective within fog areas, where the surfaces got a fog value, or if global fog (see **fog_color** variable) is switched on. If this value is set to **0**, there won't be any fog. On **100**, you can hardly see the player entities' hand before his eyes. Fog will decrease the rendering speed in software mode.

diameter = number;
'Camera size' in quants, to determine the hull used for the camera collision detection. If set to 0, the view can move through walls.

genius = ***synonym*****;**

Synonym for an entity that is attached to the view. Used for 1st and 3rd person games. This entity will be invisible if the view is within its bounds. The view can still move independently of the entity, but it won't see any object that cannot be visible at the same time for the entity. By setting **`genius`** to **`null`**, no entity is attached to the view.

flags = **`visible;`**

Only by setting this flag the view will appear on the screen.

Engine-Variables

There are a lot of internally predefined variables and synonyms, which influence the rendering in many ways, or contain values which can be used in functions:

nullvector
All three numbers of this predefined vector are always **0**.

pi
This variable has always the value of 3.14159265.

time
The time of the last frame cycle, in ticks. Often used as a correction factor to compensate different frame rates for time dependent functions. A speed multiplied by **time** gives a distance. On a frame rate of 16 frames/sec. this variable has a value of 1. For preventing extreme "jerks" **time** is limited.

total_ticks
The time passed since start of the game. The value of this variable is increased every frame cycle by the **time** variable.

total_frames
The number of frames displayed since start of the game.

sys_seconds (Seconds, 0..59)
sys_minutes (Minutes, 0..59)
sys_hours (Hours, 0..23)
sys_day (Day of the month, 1..31)
sys_month (Month, 1..12)
sys_year (Year, 4 digits)
sys_dow (Day of the week, 1..7, 1 = Monday)
sys_doy (Day of the year, 1..365)

These variables contain the PC's current time and date.

sound_vol
Volume for all **wav** sound effects, range from **0 .. 100**.

midi_vol
Volume for midi songs, range from **0 .. 100**. By default, pressing **[F12]** toggles the sound and music volume on and off.

midi_playing
As long as a midi song is playing, this variable is at non-zero.

movie_frame
Gives the frame number of the currently playing movie animation file; set to **0** if no movie is playing.

cd_track
Number of the audio CD track currently playing, or **0** if nothing is played. Is only updated by the instruction **play_cd**.

freeze_mode
If this variable is set to **1**, all entity actions will stop, causing all entities in the level to freeze. If the variable is set to **2**, also all currently running functions will stop. This variable is normally used to pause a running game. Resetting the variable to **0** (default) will cause all stopped functions to resume at the point **freeze_mode** was set.

fps_max
Limits the frame rate to the given value (default 50 fps), in order to avoid speed differences and other effects that may occur on extremely high frame rates due to poor **wdl** scripting.

fps_min
This variable (default **4**) limits the range of the **time** variable to a minimum frame rate. Below that frame rate, **time** stays constant.

pos_resolution
This variable selects the resolution of entity positions. At **0** (default) the resolution is 0.125 quants. At **1** the resolution is 0.001 quants, allowing for extremley huge levels of hundreds of square miles with very low scaled models. Objects fixed very close to the camera, like a weapon, are moving smoother in high resolution mode. The drawback is a 25% increase of the response time for multiplayer games. **Pos_resolution** must be set before loading the first level, and must not be changed during gameplay.

app_name
This string contains the name of the application, for instance **"office"** when the main WDL file name is **office.wdl**.

warn_level
If this variable (default **1**) is set to **0**, warnings about texture, entity sizes and bad WDL instructions are never issued. If it is set to 2, warnings about possible texture size problems are always issued – even if the current 3D card can display them. This is useful if you want to check whether your textures are small enough to be displayed even on outdated 3D cards, like Voodoo3.

num_actions
Gives the number of currently running actions (including particle actions). It is also displayed in the **[D]** statistics panel.

The following variables and synonyms can be used to define global settings for the rendering, the appearance of surfaces and visual effects in the level:

video_mode
The current video mode. Cannot be set directly by a function, but can be defined with a starting value to select the video mode in which the engine should start. During game play, this variable can only be changed through the **switch_video** instruction. By default, pressing the **[F5]** key toggles the video resolution. The following resolutions are available:

320x200	- 1	
320x240	- 2	(default mode for fly-thru)
320x400	- 3	(not supported by some graphics adapters)
400x300	- 4	(not supported by some graphics adapters)
512x384	- 5	(won't run on some laptops)
640x480	- 6	(default mode for the game)

800x600 - 7
1024x768 - 8
1280x960 - 9
1600x1200 – 10

video_depth

The current video colour depth (8, 16, or 32 bits). In 8 bit mode, the engine uses the internal software renderer and needs **DirectX 5.0** or above. In 16 bit and 32-bit modes, the engine will use **D3D** and needs **DirectX 7.0** or above to render the image. This variable can't be set directly by a function, but can be redefined with a starting value to select the colour depth and mode in which the engine should start. Example:

```
ifdef d3d;
   var video_depth = 16;  // D3D mode if started with -D D3D
ifelse;
   var video_depth = 8;   // 8 bit software mode otherwise
endif;
```

During gameplay this variable can only be set through the **switch_video** instruction.

video_screen

The current video screen mode (**1** = fullscreen, **2** = window). Some old 3D cards (Voodoo) are not able to render in window mode. During game play, this variable can only be set through the **switch_video** instruction. By default, pressing the **[Alt-Enter]** key toggles the screen mode.

screen_size.x, screen_size.y

The current screen resolution. Cannot be set directly, but will be changed automatically by the **switch_video** instruction.

d3d_mode

This read-only variable reflects the quality of the 3D card; **0** = incapable, **1** = weak, **2** = slow, **3** = good. It can be used to reduce the LOD on weak and slow 3D cards, increase the ambient light to compensate the low-contrast picture of weak 3D cards, or give the user advice to buy a better card.

d3d_panels

By setting this flag to **on**, all panels are drawn by D3D hardware in 16 bit mode, even if their D3D flag is not set. In 32 bit mode panels are always drawn by hardware. Drawing panels by hardware has the advantage that the frame rate increases and the panels are drawn in true colours, but the disadvantage that they eat up valuable video memory.

d3d_texmemory

The amount of video memory available on the 3D card, in KB. This is also indicated by the engine at game start. Please note that the real amount available for 3D textures is some MB less, due to the video memory needed for the screen buffers. Often, the video memory available is around 29 MB on good 3D cards (TNT2), and 5 or 9 MB on old cards (3dfx, Voodoo).

d3d_texreserved

Amount of video memory reserved for non-managed textures. Those textures are not swapped to and from virtual memory by DirectX's texture manager, and therefore can't produce a "first sight jerk" due to a hard disk access during swapping. All textures that do not fit into the reserved texture space are swapped. The default value of reserved texture space is half the amount of video memory available (**d3d_texmemory**) on good 3D cards, and 0 on worse cards.

At first the shadow maps of the level are allocated in the reserved texture space (they are the most critical for swapping), then surface textures, then entities. The sky and scene maps are allocated into reserved texture space with higher priority than the surface textures, if their synonyms are set before level loading.

The more video memory is reserved for non-managed textures, the less space is available for the texture manager, resulting in more swapping of the remaining managed textures. If there is no space left in video memory, any remaining textures can't be allocated. This is indicated by an error message, and the level won't run. Even if more video memory is reserved through **d3d_texreserved** than available on the 3D card, at least around 2 MB are always kept free for managed textures. This way, small textures can always be allocated, even with a bad **d3d_texreserved** value.

To minimize swapping jerks, set **d3d_texreserved** to the size needed for the biggest of the game levels in the **main** action before level loading. The size can be determined through **d3d_textotal** (see below). Please consider that some voodoo/3dfx cards won't start with any reserved texture space. So take care to always provide a startup option for the end user to disable reserved texture space. Example:

```
function main()
{
ifndef notex;   // disable d3d_texreserved by starting with -d notex on weak cards
    d3d_texreserved = min(12000,d3d_texmemory/2);   // biggest level of game needs 12 MB tex space
endif;
    sky_map = my_sky;     // set sky maps before level loading to give them priority
    cloud_map = my_clouds;
    scene_map = my_scene;
    load_level <office.wmb>;
    ...
  }
```

d3d_textotal

The total amount of on-board video memory (in KB), consumed in D3D mode for surface textures, entities, shadow maps, and D3D panel bitmaps of the current level. This variable is only valid after level loading. It is visible in the default debug panel (after pressing **[D]**). If this amount exceeds **d3d_texreserved**, textures can be swapped to the virtual memory, which can result in 'jerks' at certain places in the level.

d3d_texsmaps

The amount of video RAM in KB consumed by the shadow maps.

d3d_texbmaps

The amount of video RAM in KB consumed by bitmaps for D3D panels.

d3d_texlimit

Another variable for those unfortunate users of Voodoo/3dfx cards. At game start it gives the maximum texture size the 3D card can display. On normal 3D cards this is 1024 or 2048. Bad 3D cards like Voodoo3 or below have a maximum texture size of 256. All textures bigger than the size given will then be automatically shrunk to half the size. This way, Voodoo/3dfx cards can display surface textures even bigger up to 512x512, however in reduced quality.

A second purpose of this variable is to allow huge levels to run smoothly on system with very limited texture memory. **D3d_texlimit** can be set manually to a lower value. If set to a value like 64, all textures bigger than 64 will be reduced, but the smaller ones keep their original quality. This way the overall texture memory needed for the level can be reduced by over 50%.

d3d_lightres
By setting this variable to 1 the dynamic light quality in D3D mode is increased, however at the cost of decreasing the frame rate.

d3d_lines
If this variable is set to 1, all polygons will be surrounded by red lines in D3D mode. This way the splitting of surfaces by the BSP tree, and by the dynamic light tesselation process can be examined.

gamma
Can be used to brighten or darken the palette in 8-bit mode, in order to adapt the image to the monitor or to the room brightness level. By default, pressing **[F11]** toggles the `gamma` variable in several steps.
In 16-bit and 32-bit modes, where no palette exists, `gamma` can be used to brighten or darken the shadow maps, thus giving the level a brighter or darker atmosphere. In that case the `gamma` variable must be defined with a new value in the WDL script. Changing it by **[F11]** or by a function has no effect.

render_inflate
These variable (`0 .. 2.5`) can be used to compensate for little 'gaps' between polygons that are produced by inaccurate D3D drivers, especially of Voodoo, Banshee or ATI cards. When set above 0, gaps are filled, but rendering is a little slower.

logo
Variable to choose the corner where the engine watermark logo will be displayed (1..4, some editions only).

clip_range
This variable gives the maximum visibility range and the LOD distances of entities (default 100000 quants). Entities further away are not visible in the `view`. Very useful for increasing the frame rate by clipping away far entities.

floor_range
Controls the range within which the entities' light depends on the brightness and shadows of the floor area below (default 1000 quants). If the entity's height above the floor exceeds that range, the entity won't receive light, albedo and fog values from the floor texture. This way it can be prevented that aircraft models in a flight simulator suddenly change their brightness dependent on shadows on the ground.

mip_flat, mip_shaded
Use these variables (default `1.5` resp. `1.0`) to determine the relative distances for switching mipmap textures on flat or on shaded surfaces. They can also be set to different values dependent on `video_depth`. A value of 0 switches mip-mappingoff.

max_entities
The maximum amount of entities in a level; must be set before loading of thies level. The default value is `nexus` times 10.

max_particles
This variable (default 2000) determines the maximum number of particles if redefined to another value at game start.

num_particles
Indicates the number of currently existing particles.

num_vismappolys
num_visentpolys
num_visents

These variables indicate the number of visible map and level polygons, visible model and sprite polygons, and visible entities rendered per frame. They can be used for optimizing the frame rate by setting the level topography and the **clip_range** accordingly. The statistics panel, which pops up on pressing **[D]**, displays the above variables.

scene_map

Synonym for the scene bitmap wrapped around the horizon. It will appear on all sky surfaces, and can be used for mountains, or for a city landscape in a distance. It is an overlay, i.e. the colour 0 (black) parts are transparent. It must be a power of two in horizontal size, and the horizontal edges must match each other seamlessly. The **scene_map** may be bigger than the texture size limit of the 3D card. The scale and vertical position can be adjusted by the two following variables:

scene_field

Gives the horizontal sector covered by the **scene_map** in degrees (default **90**), thus determining the scale of the bitmap. If this value is below 360, the scene bitmap will be horizontally tiled.

scene_angle.tilt

The elevation angle of the lower edge of the **scene_map** in degrees (default **-5**). An angle of **0** places the scene map exactly onto the horizon.

scene_nofilter

If this flag is set to **on**, the **scene_map** is drawn without bilinear filtering and antialiasing in D3D mode. This way it won't look blurry, and has no black seams produced by some 3D cards (Voodoo) due to antialiasing to the black transparency colour.

cloud_map

Synonym for a cloud overlay bitmap, which appears on all sky surfaces behind and above the scene bitmap, and will replace the cloud overlay, which may be given by WED. It is normally used for moving clouds, and is projected onto a dome around the map. It must be a power of two in size, its colour #0 parts are transparent, and it is a tiled texture, so all edges must match seamlessly.

sky_map

Synonym for the sky bitmap, which appears on all sky surfaces behind the cloud bitmap, and will replace the sky texture originally given by WED. It can be used either for stars, or for a second layer of moving clouds. It must be a power of two in size, and is a tiled texture, so all edges must match seamlessly.

The **sky_map** and **cloud_map** can individually be switched off by setting them to **null**. If they are **null** before loading a level, they are automatically set from the sky texture in the level. If the **sky_map** is **null**, the **scene_map** is drawn in non-overlay mode. If there is neither a sky, nor a cloud, nor a scene, no sky is drawn at all. This way own sky sphere models can be activated. By using **flare** or **transparent** flags for sky spheres, an arbitrary number of sky layers can be created.

The following variables determine the appearance of the cloud and sky bitmap:

sky_clip
This elevation angle gives the lower boundary of the **sky_map** and **cloud_map** in degrees. The default value 0 corresponds to the horizon. Sky below this angle won't be drawn in D3D mode. This increases the frame rate. Examples:

```
sky_clip = -90;  // the whole sky is drawn, even below the floor
sky_clip = 45;   // only the top segment of the sky is drawn
```

sky_scale
Scale of the **cloud_map** and the **sky_map** (default **1**).

sky_curve
'Steepness' of the sky dome (default **1**).

cloud_speed.x, cloud_speed.y
Gives the speed of the moving clouds in X and Y direction (default **3**).

sky_speed.x, sky_speed.y
Gives the movement speed of the sky bitmap in X and Y direction (default **1**). For a non-moving star sky, set these values to **0**.

sun_map
Synonym for the sun overlay bitmap (not yet implemented), which appears between the clouds and the sky. It can be used to display the sun, a moon, or a planet. It may be of any size, can move over the sky during the hours of the day, can rise or set, and can change to the moon in the night. The position of the sun can be given by the following variables:

sun_angle.pan, sun_angle.tilt
Azimuth (**0...360**, default **0**) and elevation (**0..90**, default **30**) of the sun. Can be changed during gameplay and determines the light direction for the shading of flat textures, for the gouraud shading of models, and for dynamic shadows.

sun_light
Strength (**0..100**, default **50**) of the sun effect on flat textured surfaces and on models. If this variable is set to **0**, the shading of flat surfaces won't be dependent on the sun direction anymore.

fog_color
Fog colour or darkness number (**1..5**, default **0**). Fog or darkness can either be activated locally, by giving a fog colour number through the texture **albedo** value, or globally by setting this variable.

A value of **1..4** activates one of the 4 fog colors defined in **map properties**, and **5** activates darkness. The **template** scripts use the following fog numbers:

1 – distance fog (white)
2 - under water fog (blue)
3 – lava / hurt flash fog (red)
4 - user defined fog (free)
5 – darkness / blackout fog (black)

Fog or darkness lets surfaces (except the sky) and entities blur to the target color in the distance. The strength of the fog or darkness effect can be controlled by the View's **fog** parameter. In 8-bit mode and on weak or old 3D cards (Voodoo), non-black fog does not combine well with

shadows. Old 3D cards also display bright fog differently on shaded, non-shaded and totally black surfaces, which is visible in the level. So when using bright fog, take care by setting a minimum light level (**ambient** in **map properties**) that there are no totally black surfaces in the level. Normal 3D cards have no problem to display fog and shadows at the same time.

turb_speed
Gives the wave speed of the turbulence textures (default **1**).

turb_range
Gives the wave motion range of the turbulence textures (default **1**).

The following variables and synonyms are responsible for the handling of input devices:

mouse_mode
Mouse activation variable. If set to 0 (default), the mouse pointer is not visible on the screen; if set to 1 or above, the pointer appears and may be used to touch or click items. If set to 2, mouse movements don't change the **mouse_force** variable, allowing the mouse pointer to be moved independently. If the mouse pointer is active and within an active **view** window, the frame rate will noticeably go down, because additional computing power is needed to detect the entities touched by the mouse.

mouse_pos.x, mouse_pos.y
Horizontal and vertical position of the mouse pointer in pixels, relative to the upper left screen corner. If the mouse pointer is switched on through **mouse_mode**, it may be moved over the screen by changing these variables.

mouse_map
Bitmap synonym for the mouse pointer. The hot spot will be in the upper left corner by default. If set to **null**, the mouse may still be active, but is invisible.

mouse_spot.x, mouse_spot.y
The mouse pointer's hot spot coordinates to be set in pixels relative to the upper left corner of the pointer bitmap (default **0,0)**.

mickey.x, mickey.y
Movement of the mouse in pixels within the last frame.

pointer.x, pointer.y
Absolute mouse coordinates in pixels; normally used to set the variables **mouse_pos.x** and **mouse_pos.y** to move the mouse pointer over the screen. Example:

```
bmap arrow,<arrow.pcx>;

function mouse_toggle() {    // switches the mouse on and off
  mouse_map = arrow;
  mouse_mode += 2;
  if (mouse_mode > 2) {  // was it already on?
    mouse_mode=0;
  }
  while (mouse_mode > 0) {        // move it over the screen
    mouse_pos.x = pointer.x;
    mouse_pos.y = pointer.y;
    wait(1);
  }
}
```

mouse_range
Range within click, touch, or release events of entities can be triggered by the mouse (default 1000 quants).

mouse_moving
This variable indicates whether the mouse is moving (**1**) or has been immobile for ¼ second (**0**).

mouse_calm
Maximum distance in pixels still considered as immobility for **mouse_moving** (default **3**).

mouse_time
Time in ticks that is used to measure the distance **mouse_calm** in order to determine **mouse_moving** (default **4**).

mouse_left, mouse_middle, mouse_right
State of the 3 mouse buttons; **0**=not pressed, **1**=pressed.

mouse_force.x, mouse_force.y
The speed with which the mouse is currently being shifted in vertical and horizontal direction, range around **-1 ... +1**.

mouse_ent
This synonym gives the entity the mouse pointer is over (if any).

joy_raw.x, joy_raw.y, A5 joy_raw.z, joy_rot.x, joy_rot.y, joy_rot.z
Raw position of joystick axes, range **-255** ... **+255**. Up to 6 axes can be used. Through simple WDL arithmetics the range can be changed, example:

```
throttle = 50 - (50/255)*joy_raw.z;  // gives throttle a 0..100 range
```

joy_force.x, joy_force.y
The translated, calibrated position of the joystick in vertical and horizontal direction, range around **-2.0** ... **+2.0**. Automatically calibrated to zero in the centre position.

joy_1, joy_2, joy_3, joy_4, A5 joy_5, joy_6, joy_7, joy_8, joy_9, joy_10
State of the up to 10 joystick buttons; **0** = not pressed, **1** = pressed.

key_f1...key_f12, key_esc, key_tab, key_shift, key_ctrl, key_alt, key_space, key_bksp, key_cuu, key_cud, key_cur, key_cul, key_pgup, key_pgdn, key_home, key_end, key_ins, key_del, key_pause, key_enter, key_car, key_cal, key_0... key_9, key_a...key_z.
State of the keyboard keys; **0**=not pressed, **1**=pressed. The key names reflect the physical mapping on the keyboard, not the character actually printed onto that key. The names given here correspond to a German keyboard. On US keyboards, the Y and Z keys are swapped. Some special keys have alternative names to support German and US keyboards:

Key	US	German (alternative name)
key_grave	**[~`]**	**[°^]**
key_minusc	**[_-]**	**[?\] (key_sz)**
key_equals	**[+=]**	**[`'] (key_apo)**
key_brackl	**[{[]**	**[Üü] (key_ue)**
key_brackr	**[}]]**	**[*+] (key_plus)**
key_bksl	**[|\]**	**-**
key_semic	**[:;]**	**[Öö] (key_oe)**
key_apos	**["']**	**[Ää] (key_ae)**
key_slash	**[?/]**	**[_-] (key_minus)**

key_force.x, key_force.y
The smoothed force applied by pressing the cursor keys, range around **-1 ... +1.**

key_any
This variable takes the value of 1, when any key, mouse or joystick button is pressed, otherwise it is 0.

enable_key
enable_mouse
enable_joystick
If those flags are set to **off**, no keyboard, mouse, or joystick button actions are executed. This is useful, for instance, for locking the keybord during AVI playback to prevent menus popping up in the background by hitting random keys. Example:

```
  play_moviefile("intro.avi");
  enable_key = off;
  enable_mouse = off;
  enable_joystick = off;
  while (movie_frame != 0) {
// allow interrupting the intro by hitting any key (recommended!!)
     if (key_any != 0) { stop_movie(); }
     wait(1);
  }
  enable_key = on;
  enable_mouse = on;
  enable_joystick = on;
```

The following variables and synonyms are set or used by instructions:

result
This variable is set as a return code by some WDL instructions to indicate the result or success or failure.

my
Synonym for the entity the current action is attached to. It will remain valid during the whole action and all functions started by it. If the current function was not attached to an entity, this synonym is undefined.

you
Synonym for the entity that has caused or triggered the current function - be it by shooting at, or colliding with the entity the function was attached to, or be it by creating that entity, or emitting that particle. Otherwise, this synonym is undefined. It will only be valid until the first **wait()** instruction.

target, normal, bounce
Vectors, often used to indicate a target position, a surface normal, and a bounce direction by several instructions. Will only remain valid until the next **wait()** instruction.

event_type
Variable set at the beginning of an entities' **event** function to indicate the kind of event.

in_solid, in_passable, on_passable
Variables set to 0 or 1 by some instructions, describing the entities' current position.

The following predefined variables and synonyms can be used as short-term general purpose variables. They are used by some instructions. Within a particle function, they influence the position and appearance of the particle.

my_age
Variable, the age of the particle in ticks; is zero at birth, and will be incremented automatically.

my_pos
Vector, the current position of the particle.

my_speed
Vector, the speed of the particle. Also set by the **move** instruction.

my_map
Synonym for the bitmap of the particle.

my_color
Vector, the RGB colour of the particle.

my_flare, my_transparent, my_bright.
Flags that determine transparency and flare effects of particles in D3D mode. See the **flare**, **transparent**, **bright** flags for entities.

my_size
Variable, the relative size of the particle (**0 .. 200**, default **100**). If the size is **0**, the particle is invisible, but will continue to exist.

my_action
Synonym for the function of the particle. If set to **null**, the particle will die.

The following objects are used in multi-player games:

connection
This variable is set to **0** at game start. As soon as a multiplayer connection is established, it is automatically set to **1**, **2**, or **3**, dependent on the engine running in server, client, or client/server mode.

message
This **string** contains the name of the client on joining or leaving the game.

my_string
This string synonym is set to the **string** on receiving it by **send_string**.

Predefines

The following definitions can be given at the beginning of the WDL script to set some basic modes for the application and the further execution of the script:

include *<filename>;*
Reads an additional WDL script from the given file and then continues scanning the original WDL file. This way pre-defined WDL scripts can be inserted. A maximum of 40 WDL files can be given by **`include`** when generating the game by **`publish`**. The number of **`include`**s is not restricted when generating the game by **`resource`**.

bind *<filename>;*
The given file will be included in the game resource by the resource packer (professional edition only). Only for special purposes; all files between angled brackets will be included anyway. You may give any number of **`bind`** files.

savedir *"dirname";*
Names the default folder for saving games and screenshots. If this folder does not exist, it will be created when first saving a game score. Note that backslashes ("\") have to be given in C++ notation, i.e. as double backslashes (e.g. "C:\\mygame"). The folder name given here may be overridden by the command line option **`-dir`**.

path *"dirname";*
All additional files - bitmaps, sounds or entities - will be first searched for in the actual folder and then in the path given here (again, give backslashes as "\\"). You can specify up to 16 **`path`** names, which will be searched in the given sequence.

define *name;*
define *name, replacement;*
`Define`s can also be given by the **`-d`** command line option. The second line allows you to rename names and parameters in the WDL file; the first one is for including or excluding WDL lines depending on conditions (see **`ifdef`** below). Every time the ***name*** appears in the WDL script below the **`define`**, it will be replaced by the ***replacement***, which can be another name, or a number, or anything. Renaming makes functions more 'readable', for instance by giving the entities' 40 general purpose **`skill`** parameters some meaningful names. Example:

```
define mynumber,-123;
define damage,skill3;
action kill {
   result = my.damage;
   my.damage = mynumber;
}
```

Characters like "**`{} ,;()<>`**" cannot be redefined, and **`define`**s may not be nested.

undef *name;*
A name defined by **`define`** can be 'undefined' by **`undef`** for all following WDL script lines.

ifdef *name;* **ifndef** *name;* **ifelse; endif;**
These names can be used to to skip certain WDL script lines dependent on previous **`define`**s or - especially - **`-d`** command line options. All script lines between **`ifdef`** and **`endif`** are skipped if ***name*** was not **`define`**d before. All lines between **`ifndef`** and **`endif`** are skipped if ***name*** was **`define`**d before. The instruction **`ifelse`** reverses the line skipping or non-skipping. This way,

you can 'invent' new command line options for the engine, which have an arbitrary effect on the game. Example:

```
define hires_d3d;      // or start with option -d hires_d3d
...
ifdef hires_d3d;
  var video_mode { val 6; } // 640x480 high resolution
  var video_depth { val 16; } // 16 bit colour mode
ifelse;
  var video_mode { val 2; } // 320x240 low resolution
  var video_depth { val 8; } // 8 bit colour mode
endif;
```

Database and Dataview

Through external text databases, strings can be loaded dependent on external text files (professional edition only). In future versions, variables, bitmaps, sounds, and entities may be loaded as well. Such a database is defined through the following names:

database *name* **{ ... }**
Defines a database. The database is split into records (rows) and fields (columns) like a spreadsheet table. Currently all fields of the database must be ASCII strings; in future versions they may also contain numbers or file names. The following names can be given within the winged brackets of the **database** definition:

datafile **<***filename***>;**
File name of the **database** file, which contains the data as ASCII text formatted in fields and records.

separator **"***c***";**
ASCII character, which separates the fields of each record (e.g. **";"** - default: **","**). The records themselves are separates by line feeds.

string ***name;***
Defines a name for a field that contains a text string (currently the only supported field type). All fields of a record must be defined this way, in the correct order.

To select or unselect certain records of a **database**, a **dataview** is used:

dataview *name* **{ database** *name;* **}**
Creates a selection structure to a database with the given name. For each record of the database, an internal flag tells whether the record is selected or not. By default all records are selected. An arbitrary number of **dataview**s can be used for the same database.

Dataviews can be assigned to **text** objects in order to display certain strings of a database. You can use the **select**, **unselect**, **and_select** and **or_select** instructions to determine which rows of the database will be displayed and which not.

For example, a file **data.txt** contains the following three text lines:

```
Smith,John,New York
Mueller,Karl,Hamburg
Manchu,Fu,Alma Ata
```

You can use the **database** and **dataview** objects to display selected strings, and the **select** instruction for selecting certain records:

```
database people_db {
   datafile <data.txt>;
   separator ",";
   string "Name";
   string "Firstname";
   string "City";
}

dataview people_view { database people_db; }
```

```
text city_txt {
   font standard_font;
   dataview tst_view;
   viewpos 2;
   flags visible;
}

string select_str "John";

function select_people() {
   select people_view,"firstname",select_str;
}
```

Multi-User Mode

With certain GameStudio editions you can connect an arbitrary number of PCs through a local area network or the Internet, or two PCs through a modem or serial connection. One of the PCs in such a multi-user configuration runs as the **server**, each other as a **client**.

The game world exists only on the server. The server transmits the entity positions and other properties to the clients. This way each client has a consistent image of the current state of the world. It is similar to have multiple **views** defined on a single PC. However, each client himself can also transmit certain events to the server, in order to create and control one or more entities in the game world. Or for totally different purposes, like chatting or exchanging messages in an multi-user system without a game world at all.

A multi-user system must contain of one server and at least two clients. It is possible to run a server and client simultaneously, so the minimum multi-user configuration is two PCs. The server has the most work to do, and therefore should always be the fastest machine. The same WDL script must be loaded on all PCs. Within the WDL scripts the name **server** is automatically pre-defined on the server and **client** on the clients. This way, different scripts can be executed on server and clients via **ifdef server**; or **ifdef client**; statements. However care must be taken here: the number of elements like functions, variables, sounds and so on must be the same on the server as well as on the clients.

An arbitrary number of multi-user applications – named **sessions** - can run on the same local area network or the same Internet address. A client always connects to the session that runs the same WDL script. This way a multi-level game server can be realised.

The server must be started first, using the **–sv** command line option. After that, the clients can connect (**-cl** command line option). With both options given, server and client are started simultaneously. If the connection is established, the predefined variable **connection** is set to **1**, **2**, or **3**, dependent on the connection running as server, client, or both.

By default, the **IPX** protocol is used for the multiplayer network. By giving the command line option **–tcp,** the **TCP/IP** protocol can be used, as necessary for internet connections. In TCP/IP connections the user is be prompted at game start for the server's domain name or IP address to connect. If none is entered, the PC will open the session as a server, and will be waiting for other PCs to connect. If your server doesn't have a domain name, you can find your current IP address with the **WINIPCFG** program in your Windows folder. On a local network the IP address of connected PCs can be found by the **NETSTAT** program. Both programs are available on each Windows PC.

Each client has an individual name, which may be given by the **–pl** command line option. On connection of each client, the predefined **on_receive** function on the server is triggered. If the predefined **message** string synonym was set to a certain string, that string contains the name of the client which has just connected. If that client has no player name, a unique name is created by the engine.

The current status of the world is transferred to each client on connection, and will be updated after each server frame. Nothing else will be transferred automatically. So the same variables and strings may contain different values on each of the PCs, if not synchronised on purpose through the **send_string** instruction. The **send_string** and **execute** instructions can be used to send arbitrary WDL instructions, like for level change, from the server to the clients.

If an entity was created by a client through the **create** instruction, it will come into existence on the server, and its main and event functions only run there. But it will remember its creator, and if he sends entity skills through the **send** instruction, it will receive the skills from that client. This way each client can create and control his own player entity on the server. If the client disconnects, he can't send further intentions, so his player entity will remain motionless. **event_disconnect** may then be used to remove the entity.

The office level is prepared for multiplayer mode. To try it out, connect two or more PCs via a LAN with **IPX** protocol. Start a session on the fastest of the PCs in Client/Server mode (**–sv -cl**). As soon as the session is running, start the clients (**-cl**). If a client connects, his entity will appear in the office level always at the same position (given in the **main** function), a little behind the normal player position. So he should walk away a little as soon as possible to prevent further client entities sticking within each other. In a true multiplayer game this should be handled by a script. Multiplayer fighting and weapon handling also is possible, but not yet implemented in the **template** scripts. **[F4]** activates a chat mode for sending messages to the other players.

Starter Window Definitions

With the professional edition, you can change the engine's behaviour even before level start, and after finishing the game. By default, on start up a small window will be displayed, which gives some system messages, and offers a button for early abort. However, you can create your own window, with different buttons, pictures, greeting messages, your company's logo, or a progress bar.

Such a window is defined this way in the WDL script:

window winstart { ... }
or
window winrun { ... }
or
window winend { ... }

The first line defines the starting window, the second the size of the game window, the third one the window which appears after the end of the game. If the **winrun** window is not defined, the initial video resolution will be taken as the size. If the **winend** window is not defined, there will be no end window displayed.

Within the winged brackets, the following names may be given:

title "*text*";
The text which appears in the title bar of the window.

size *x*, *y*;
The window size in pixels. If the size is not given, the initial video resolution will be used.

mode *name*;
Window mode. Either **standard**, **image**, **fullscreen**, or **use_oldstyle**.

bg_color rgb(*r*, *g*, *b***);**
Window background colour; *r*, *g*, *b* = 0..255.

set font "*font*", **rgb(***r*, *g*, *b***);**
To give a font and colour for the following **text** *name*. *Font* is a windows font. If it is not available, or if "" is given, the standard font is taken.

text "*text*", *x*, *y*;
Simple text output at the given position. Line feeds within the text can be given by "**\n**".

text_stdout "*font*", **rgb(***r*, *g*, *b***)**, *x*, *y*, *width*, *height*;
Scrollable text field for the standard engine output.

frame *style*, *x*, *y*, *width*, *height*;
A frame within the window. For *style* the names **ftyp1**, **ftyp2**, **ftyp3**, or **ftyp4** can be given.

bg_pattern <*filename*>, **opaque;**
Fills the whole window background with the pattern given by the filename (.**pcx** or .**bmp** file).

pattern <*filename*>, **opaque**, *x*, *y*, *width*, *height*;
Fills the given area with the background pattern given by the filename (.**pcx** or .**bmp** file).

picture **<***filename***>, opaque,** *x, y;*
Places a picture at the given position.

progress **rgb(***r, g, b***),** *elements, x, y, width, height;*
Displays a progress bar. ***Elements*** corresponds to the number of dots otherwise displayed within the engine standard output.

The window definitions will only become valid as soon as a **starter** file was created by WED.

Reference: Prefabricated Scripts

Most of the following WDL script files are automatically **include**d to each new script. So all functions below are available for WED as well as for your own scripts, as long as you don't erase the **include** lines. The files are located in the **template** folder. If you want to change them for your own purposes, just copy them into your work folder before. Files found in the current folder will be used with priority over these with the same name in the **template** folder.

Instead of attaching an action directly to an entity, you can combine most basic entity actions (if they are of a different kind, and thus don't interfere with each other) by starting them from a main action. Then attach this main action in order to compose an actor from several behaviours. You may also set the entity skills and flags by that action, instead of the WED property panel. Example:

```
action robot1 {
   my._walkframes = 1;
   my._entforce = 0.7;
   my._firemode = damage_explode + fire_ball + hit_explo + bullet_smoketrail;
   patrol();       // set the movement behaviour
   drop_shadow();  // set the shadow
   actor_fight();   // set the fighting behaviour
}
```

At the beginning of each script file, at first the sound or graphics files needed (all located in the **template** folder) are defined, and then the starting values of some skills which may have influence on the functions within. You can change both within your main WDL script, without changing the predefined script file. To change skills, either re-define them in your main WDL file after the **include** list, or give them a different value at the beginning of your game (function **main()**).

Movement.wdl

This file contains the basic actions for the player movement.

player_move()

This action will make the entity to which it's assigned to a player. He can walk, drive, strafe, turn, look up and down, jump, duck, crawl, wade, swim, and dive. Gravity is applied, and the entity may climb up stairs. On a rather steep slope, he will be drawn downwards by gravity. If the slope is too steep, he will be pushed back so that it can't climb it. If he enters water, he will begin to swim.

For testing purposes, pressing **[0]** lets the player control the entity like a camera; pressing **[0]** again moves the camera out of the entity (development version only; the pre-defined direct camera control is disabled in the published game). Pressing **[F7]** toggles between first and third person mode. The **player** synonym is set to the entity, to be used in other functions. Be careful not to assign this action to more than one entity!

The entities' origin must be placed high enough to be able to climb stairs. The model animation can contain frame names beginning with:

"stand" - standing animation
"walk" - walking animation
"run" - running animation
"crawl" - crawling animation
"swim" - swimming animation

"dive" - diving animation
"duck" - ducking animation
"jump" - jumping animation

The number of frames per animation cycle can be arbitrary. The entity will switch from walking to running at a speed of more than 12 quants per tick. Pressing **[Home]** and **[End]** initiates ducking and jumping while on solid ground, and a vertical rotation for diving down while in water. If the entity is under water, blue fog (fog number 2) is activated.

You can change some features by setting the following skills and flags within the entity panel. The same skills and flags may also be set by functions, allowing the player to change his movement behaviour during game time. **define** statements are used to give the skills meaningful names:

_walkframes (skill1)
Must be set to **1** in order to enable animation with frame names.

_entforce (skill5)
Force factor, determines the entities' speed; default = **1**.

_banking (skill6)
Banking factor, determines the roll angle in curves. May be set to a negative (motorcycle, airplane) or positive (car) value; default = **0**.

_movemode (skill7)
Move mode; **_mode_walking** (**1**), **_mode_driving** (**2**), **_mode_swimming** (**3**) are available.

__fall (flag1)
If set, the player can take damage by falling down. If he falls, his fall time is calculated. He won't be damaged if his fall time is less then 5 ticks. Otherwise, an amount of **(10 + int((my._falltime - 5) * 1.75))** will be subtracted from his **_health (skill9)** value.

__wheels (flag2)
Prevent turns without moving, like a vehicle on wheels.

__slopes (flag3)
Adapt the tilt and roll angle to the floor slope, like a vehicle on wheels.

__jump (flag4)
Enable jumping by pressing the **[Home]** key.

__bob (flag5)
Head bobbing.

__strafe (flag6)
Enable strafing by pressing the **[,]** and **[.]** keys.

__trigger (flag7)
Sets the player's **trigger_range** for automatically operating doors or platforms on approach.

camera_move()
This function will let the player take direct control over the camera view, independent of the player actor. **[PgUp]/[PgDn]** will change the tilt angle, **[Alt+CursorLeft]** and **[Alt+CursorRight]** will change the roll angle.

client_move()
This function must be started on the client in a multiplayer game, before or after creating the player entity. It collects the movement forces, entered through the client's keyboard or mouse, and sends them to the player entity on the server which was created on the client.

drop_shadow()
This function can be used to attach a flat shadow sprite to an entity. The default shadow sprite is simply a dark semitransparent ellipse (**`shadow.pcx`**) projected onto a horizontal floor, but that can be changed by the user. The shadow appears as long as **`_movemode (skill7)`** of the entity is non-zero and the entity has no own shadow (**`shadow`** flag).

_player_intentions()
This function scans keyboard, mouse and joystick and calculates the vectors **`force`** (for player movement) and **`aforce`** (for player rotation). For redefining keys or changing the movement behaviour, an alternative **`_player_intentions`** function can be entered in the main WDL file.

move_view()
This function places the camera view at the position of the player entity, either in 1st or 3rd person mode, dependent on the content of the **`person_3rd`** variable.

attach_entity()
Can be used for creating entities that are attached to the **`my`** entity.

Actors.wdl

This script file contains the basic functions for actor (NPC) movement.

patrol
The patrol action will let a model entity walk a path between start positions, using the same skills and flags as the **`player_move`** action. At game start, the entity will look for the nearest start position within 2000 quants. It will walk over to this position. When arrived, it will look for the next position in the approximate direction the first start position is pointing to. This will be repeated until it does not find a start position any more. Then the entity will stop.

To set a circular path for the entity, place some start positions within a distance of less than 2000 steps from each other. Adjust the angles so that each position points to the next, and the last position points again to the first one.

A5 patrol_path
The patrol action will let a model entity walk a closed path, using the same skills and flags as the **`player_move`** action. At game start, the entity will look for the nearest path within 1000 quants. It will walk over to the first waypoint. When arrived, it walks to the next waypoint and so on. Have a look at this action to get the idea how write a movement function for an entity.

actor_turnto(*angle*)
This function rotates a model entity horizontally towards the given target angle.

actor_move()

This function lets a model entity move ahead at a speed depending on the **force** vector. Climbing steps, as well as switching between the walking and running animation will be handled.

Weapons.wdl

This script file contains the basic functions for carrying guns, firing them, and ammunition management.

gun

The **gun** action will make the entity a gun. It can be picked up by touching it, clicking it with the mouse, or pressing the **[Space]** key nearby. If it is picked up, it will be assigned to the camera view via the **carry** function. Different types of guns can be selected by pressing the number keys. If the **[Crtl]** key or the left mouse button is pressed, the weapon will fire. To place the muzzle flash at the right position, the weapon's centre must be inside its barrel.

The action also handles transferring guns into another level. In each level, all gun models that may be already picked up by the player must be placed somewhere. The **gun** action checks whether the player has picked up a gun with the same weapon number before, and if so, automatically transfers it into the player's possession at level start.

Individual guns can be build by setting the following flags and skills and then calling the **gun** action:

__rotate (flag1)

If this flag is set, the gun model rotates before being picked up.

__silent (flag2)

If this flag is set, the gun pickup message will be suppressed.

__bob (flag5)

If this flag is set, the gun sways a little if the camera is moving. The sway period is the same as used for head bobbing in the **movement.wdl.** If the flag is not set, the gun stands still.

__repeat (flag7)

If this flag is set, it's a machine gun; otherwise a single action gun.

skill1 ... skill3

Ahead, right and down position of the gun, while carried, in quants relative to the camera.

_ammotype (skill4)

Ammo type (1..4) the gun needs. If **0**, the gun doesn't consume ammunition. The part after the decimal, times 100, gives the amount of ammo to be added when the gun is picked up. E.g. 2.30 = Ammo type 2, and 30 rounds are already in the gun.

_bulletspeed (skill5)

Speed of the bullet, default = 200 quants / tick. If the gun emits particles or no bullets at all, then its range in quants is twice the bullet speed. The part after the decimal, times 100, gives the strength of the recoil. If **0**, then there is no recoil. If above **0**, the recoil is done by sliding backwards by the given amount in quants. If below **0**, then the recoil is done by swinging upwards by the given amount in degrees.

_weaponnumber (skill6)

Weapon number that determines the key (**[1]...[7]**) to be pressed to select that gun, if it was picked up before.

_firetime (skill7)

The time (in ticks) the gun needs to reload. Includes the time for the gun recoil and animation, if any.

_firemode (skill8)

Fire mode and damage effect of the gun. It can be composed by adding the following numbers that are defined as names:

damage_shoot (1) - Damage is applied by **event_shoot**, without bullet.
damage_impact (2) - Damage is applied by the **event_impact** of the bullet.
damage_explode (3) - Damage to several targets by **event_scan** at hit point.
fire_particle (4) - the gun produces a vapour trail to the target.
fire_ball (12) - fires orange fireballs which radiate light.
fire_rocket (16) - fires a rocket **<rocket.mdl>** that leaves a smoke trail.
fire_laser (20) - a laser beam **<lbeam.mdl>** is emitted to the target.
bullet_smoketrail (32) - the fireballs leave spark trails.
hit_flash (128) - at hit point there will be a light flash.
hit_explo (256) - at hit point there will be an explosion.
hit_smoke (512) - at hit point a cloud of smoke will ascend.
hit_scatter (1024) - there will be multiple hit points, as by a shotgun.
gunfx_brass (2048) - gun ejects cartridge cases **<gbrass.pcx>**.
hit_sparks (8192) - at hit point a shower of sparks is produced.
hit_hole (16384) - a bullet hole **<bulhole.pcx>** appears in map walls.

The part after the decimal, times 100, gives the amount of damage the bullet produces (default = **10**). If the bullet explodes, the explosion radius is five times the damage value.

Example:

```
action launcher {   // build a gun subspecies
   my.__bob = on;   // swaying
   my.__repeat = on;      // machine gun
   my._ammotype = 3.20    // ammo type 3, 20 rounds
   my._bulletspeed = 100.05; // bullet speed 100, recoil 5 quants
   my._weaponnumber = 4;  // press 4 to select
   my._firetime = 5;      // around 3 shots per second
// exploding fireball with spark trail, damage 40, explosion range 200 quants
   my._firemode = damage_explode+fire_ball+hit_explo+bullet_smoketrail+0.40;
   gun();
}
```

ammopac

This action will make the entity an ammunition pack. It can be picked up by touching it, clicking it with the mouse, or pressing the **[Space]** key nearby. The ammo pack's properties are set through the following flags and skills:

_rotate (flag1)

If this flag is marked, the entity rotates before being picked up.

_silent (flag2)

If this flag is set, the item pickup message will be suppressed.

_ammotype (skill4)
Ammo type (**1..4**).

skill5
Amount of ammo to be added when picking up the pack.

medipac
This action will make the entity a first aid kit. It can be picked up by touching it, clicking it with the mouse, or pressing the **[Space]** key nearby. The medipac's properties are set through the following flags and skills:

_rotate (flag1)
If this flag is marked, the entity rotates before being picked up.

_silent (flag2)
If this flag is set, the item pickup message will be suppressed.

skill5
Amount of health to be added when picking up the pack.

bullet_shot
This action can be attached to a bullet entity upon creating. It runs the bullet into the direction given by the **shot_speed** vector. The damage the bullet effects on impact is given by the **damage** variable. Its effects – explosion or not, smoke trail or not and so on – can be given by the **fire_mode** variable, which accepts the same values as a gun's **skill8** (see **gun**).

War.wdl

This script file contains the basic functions for fights between player and actors.

actor_fight
This action can be assigned to an entity that fights the player using a gun. It handles watching for the player, attacking him, managing armour and health, trying to escape, and dying.

The cycles of the model animation must contain frames for the attack ("**attack**") and death ("**death**") animation. Most skills and flags have the same meaning as for **player_move**, with the following exceptions:

_hitmode (skill6)
Defines the behaviour on receiving hits and dying. Currently only the values **0** and **256** can be given, the latter meaning that the entity will explode after playing its dying animation, and be shattered into a lot of **<gibbit.mdl>** parts that fly through the air and fall to the ground.

_firemode (skill8)
Defines the fire mode of the entities' gun, and the damage a hit will produce. The same **firemodes** as for **gun**s can be given.

_health (skill9)
Initial health amount of the entity. Decreases on hits if the entity has no armour anymore. If this value reaches zero, the entity dies.

_armor (skill10)
Initial armour amount of the entity. Decreases on hits. If this value reaches zero, the entity begins to loose **health**.

_muzzle_vert (skill32)
Vertex number of the gun muzzle for shooting enemy models. Used to let the shoots emit from the muzzle position rather than the center of the model.

For testing purposes, the **freeze_actors** variable may be set to **1** to prevent the actors from attacking. Setting it to **2** makes the actors passable.

player_fight
This action can be used for an entity that serves as player in an unfriendly environment. It handles loosing armour and health, and dying. The skills and flags have the same meaning as for **player_move**.

Doors.wdl

This script file contains all functions for all moving parts of a level, like door, elevator, or drawbridge entities.

send_handle()
This function is assigned to the **[Space]** key by default, and will send a handle signal from the client to the server. The player entity that received this signal will use the **scan** instruction to open or close any door, or operate any elevator, within a 200 quants range from the entity.

door
This action can be assigned to door map entities. The door will open by horizontally rotating around its hinge, which is located at its center. The door opens when a **scan** instruction was performed nearby through **send_handle**, or when it was clicked on with the mouse, or when an entity approaches which has the **__trigger** flag set, or when a remote switch was operated. The door's properties are set through the following flags and skills:

_endpos (skill3)
The door's opening angle is given by its **skill3** (in degrees). If none is given, the default is **90** degrees.

_keytype (skill4)
Key number (**1..8**). If a key is needed to operate the door, **skill4** must be set to the number of that key.

_force (skill5)
Gives the door speed (degrees per tick). If **skill5** is negative, the door will open counterclockwise.

_trigger_range (skill7)
Gives the range in quants within which an entity must be to open the door automatically on approach, and close it behind him. The entities' **__trigger** flag must be set. If no range is given, the door won't be triggered automatically.

_switch (skill8)
Determines which switch entities will open this door, if any. The skill is composed from a sum of powers of 2 (1, 2, 4, 8, 16, 32, 64...). The switches whose **_switch** value contains one of those powers of 2 will open the door on operating.

gate
The gate action can be assigned to vertically moving gates (map entities). The gate will move upwards under the same conditions as described for the door entity; its movement range is pre-set to 90% of its length. The gate speed is given by **skill5** (quants per tick), the time it waits at the upper position before closing again is given by **skill6**. If **skill6** is zero, the gate will only close if again a **scan** instruction was received. Further gate properties are set through the following flags and skills:

_keytype (skill4)
Key number (**1..8**). If a key is needed to operate the door, **skill4** must be set to the number of that key.

_trigger_range (skill7)
Gives the range in quants within an entity must be to trigger the gate automatically. If no range is given, the gate won't be triggered automatically.

lid
This action can be assigned to lid or trapdoor map entities, as well as for drawbridges. The lid will open by rotating vertically around its hinge under the same conditions as described for the door entity. The lid entities' centre must be placed at its hinge position. The lid properties are the same as for a door entity.

elevator
This action can be assigned to platform map entities, which may move horizontally, vertically, or in any direction. The elevator will carry passengers between the position it's placed at, and an arbitrary target position. The target XYZ coordinates are given by **skill1, skill2,** and **skill3**. If any of these skills is 0, the elevator will not move along that axis. If the elevator collides with an entity or the player, it will move straight through it - crushing it or pushing it aside can be handled by a separate function using **event_push**.

If a key is needed to start the elevator, **skill4** must set to the numer of that key (1..8). The elevator speed is given by **skill5** (quants per tick), the time it waits at each end position is given by **skill6**. If **skill6** is zero, the elevator will only move if a **scan** was received (by pressing the **[Space]** key nearby) or if it was clicked on with the mouse; otherwise it will permanently move as a paternoster lift. Further elevator properties are set through the following flags and skills:

_keytype (skill4)
Key number (**1..8**). If a key is needed to operate the elevator, **skill4** must be set to the number of that key.

_force (skill5)
Elevator's speed in quants per tick.

_trigger_range (skill7)
Gives the range in quants within an entity must be to call the elevator automatically from its end position. If this value is set to **1** or above, the elevator starts automatically on stepping onto it.

_switch (skill8)
Determines which switch entities will start this platform, if any. The skill is composed from a sum of powers of 2 (1, 2, 4, 8, 16, 32, 64...). The switches whose **_switch** value contains one of those powers of 2 will start the platform on operating.

__remote (flag6)
If this flag is set, the elevator can be called from its end positions by receiving a **scan**.

teleporter
This action can be assigned to map entities, that teleport other entities on touching to a target position whose XYZ coordinates are given by **skill1**, **skill2,** and **skill3**.

ent_rotate
This action is useful for ventilators or water wheels. It will permanently rotate the entity to which it is assigned around its Z-axis by its **skill1** value (degrees per tick), around its Y-axis by its **skill2** value and around its X-axis by its **skill3** value.

key
This action can be assigned to key entities, which can be picked up and used for unlocking doors. Of course, they don't have to look like keys. A key entity can be picked up by touching it, clicking it with the mouse, or pressing the **[Space]** key nearby. If it is picked up, one of the **key1..key8** variables will be set to 1, dependent on the key number given by the key's **skill4**. The key's properties are set through the following flags and skills:

__rotate (flag1)
If this flag is set, the key item rotates before being picked up.

__silent (flag2)
If this flag is set, the item pickup message will be suppressed.

_keytype (skill4)
Key number (**1..8**).

doorswitch
This action can be attached to switches for doors and elevators. Its **_switch** value **(skill8)** has to be set to a power of two (1,2,4,8,16, etc.). When clicking onto a switch, all doors open that have the same power of 2 set in its **_switch** value.

Messages.wdl

The functions in this script file are responsible for showing panels, and displaying messages on the screen.

show_message(), blink_message()
The **show_message** function will display the current **msg.string** for five seconds at the upper left screen corner. The **blink_message** function will do the same, but the text will blink. Example:

```
string this_string,"this is my message!"
...
msg.string = this_string;
show_message();
```

scroll_message()

This function can be used to scroll multiple line messages over the screen. To add a new line at the bottom, use the following instructions:

```
set_string scroll.string,this_string;
scroll_message();
```

The maximum number of lines visible is given by the **scroll_lines** definition at the beginning of **message.wdl**.

enter_message()

By starting this function a player can onscreen enter a one-line message in a multiplayer game, and send it to all other players. By default, this function is set to the **[F4]** key.

show_panels()

Activates a panel at the lower left screen corner, where the player's armour, health and the ammunition remaining in the current weapon are displayed.

Particle.wdl

This script file is responsible for handling particle explosions, smoke trails and other particle effects.

particle_scatter()

This function can be assigned to particles to create an explosion. The bitmap **<particle.pcx>** is used for close particles.

particle_range()

Same as particle_scatter, but the particles will show a colour, instead of a bitmap, and will change their colour during their lifetime. The starting colour is given by the **red**, **green**, **blue** values of the **scatter_color** colour vector, the colour steps to change is given by the **scatter_range** colour vector.

particle_trace()

Same as **particle_scatter,** but the particles leave little smoke trails, made from further particles.

particle_line()

Generates a fading vapour trail between the positions given by the vectors **p** and **p2**.

snowfall()

Starts a snowfall around the camera.

Menu.wdl

This script file is responsible for handling the user interface.

menu_main()

This function will display a standard game menu. The menu can be operated by mouse as well as by keyboard. There are options to start a new game, save and load a game, set the sound and music volume and the video resolution, invoke a help panel, or quit. This menu function can be

used as a template for individual game menus. By default, the game menu will be activated by pressing the **[Esc]** key.

yesno_show()

This function will display an onscreen panel, where a **yes** or **no** button can be selected via mouse or keyboard. By default, it's a simple black and white bitmap. Touching a button with the mouse will highlight it. Clicking the **yes** button or pressing the **[Y]**, **[Z]** or **[Enter]** key will execute the function; clicking the **no** button or pressing **[N]** or **[Esc]** will remove the panel. The text string and the function must be set before via **`yesno_txt.string`** and **`yesno_do`**.

menu_show()

This function will display an onscreen menu, where up to seven topics can be selected via mouse or keyboard. As long as the menu is visible, the game will be frozen. Touching an item with the mouse or the cursor **[up/down]** keys will highlight it. Clicking it or pressing the **[Enter]** key will execute the function assigned to that item. By default, the menu will disappear by pressing the **[Esc]** key.

The text strings and corresponding functions for the menu must be set before via **`menu_txt1.string .. menu_txt7.string`** and **`menu_do1 .. menu_do7.`** The **`menu_max`** variable must be set to the number of items in the menu.

console()

The **`console`** function will display a cursor on-screen where you can type in WDL instructions, which will be executed during gameplay. Multiple instructions can be typed, up to 80 characters on a single line. Each instruction must be terminated by a semicolon. By default, the console will be activated by pressing the **[Tab]** key.

mouse_on(), mouse_off(), mouse_toggle()

These functions will switch the mouse pointer on or off. By default, the mouse pointer will appear by pressing the right mouse key.

Venture.wdl

This script file contains functions that can be used to help create adventure (RPG) type games. It is not automatically included into a new script file, but must be **`include`**d manually. This script is used by the Adeptus game. **Venture** can be customized to better fit your needs by using the following:

Predefines

`Venture.wdl` contains several default values. Many of these bitmaps, sounds, variables, etc can be changed by 'predefining' them in your game script before you **`include venture.wdl`**. To predefine a set of values you must first **`define`** the group's name, then you must predefine all the values in that group (if you don't want to change all the values, cut and paste the section between "**`define adv_def`**"... and "**`endif`**" into your script, and edit the values you want to change).

For example, to redefine the combat sounds type the following in you game script before "**`include <venture.wdl>;`**":

```
define adv_def_combat_snd;// predefine combat sounds
  sound weapon_swing_snd, <my_swing.wav>;
  sound weapon_hit_snd, <my_hit.wav>;
  sound player_hurt_snd, <my_pain.wav>;
```

Overrides

These are functions that are designed to be redefined in your game script after **venture.wdl** is **include**d. These functions replace default functions in venture (which are usually blank).

Managers

Venture functions are broken down into groups called Managers. Each manager handles an aspect of venture (inventory, combat, player stats, etc). All functions that belong to a manager start with that manager's prefix. The prefixes are as follows: Dialog (**vdia_**), Input (**vinp_**), Inventory (**vinv_**), Screen (**vscr_**), Picture Save (**vpic_**), Player Stats (**vpst_**), Items (**vitm_**), Combat (**vcom_**), Spells (**vspl_**), Touch Text (**vttx_**), and Miscellaneous (**vmsc**_).

Screen Manager

Used to manage the player's screen. This is where the player's interface panels are defined (**statbar_pan**, **screen_pan**, **blood_pan**, **blacktop13_pan... blacktop16_pan**).

Screen Predefines

The following strings can be predefined:

```
define adv_def_adventure_text;
  string wrong_key_str, "this key doesn't fit!";
  string wrong_item_str, "try another item!";
  string wrong_type_str, "nothing happened!";
  string level_info_str, "you made a level!  ...";
  string start1_str, "...";
  string bonus_info_str, "\nyou may share   bonus points!      finished";
  string char_pan_str, "             character skills ...";
  string help_str"keyboard commands ...";
  string menu_str,"...";
  string admenu_str, "...";
  string exit_str, "...";
  string vstr_exit_txt,"thank you ...";
  string credit_str, "...";
  string vent_blank_str, "...";
```

The following panels can be predefined:

```
define adv_def_screen_maps;
  bmap button_map, <ventbutt.pcx>;         // button bitmap
  bmap char_button_map, <ventback.bmp>, 535,0,60,100;    // icon for character button
  bmap inv_button_map, <ventback.bmp>, 530,120,70,85;    // icon for inventory button
  bmap menu_button_map, <ventback.bmp>, 570,210,70,70;     // icon for menu button
  bmap screen_map, <ventback.bmp>;                // the entire screen map
  bmap hpbar_map, <redbar.pcx>;                   // hitpoint bar map
  bmap strbar_map, <grnbar.pcx>;                  // strength bar map
  bmap manabar_map, <bluebar.pcx>;                // mana bar map
  bmap info1_map, <ventinfo.bmp>;    // info1 panel bitmap (used for save and load)
  bmap info2_map, <ventdia.bmp>;    // small info-text panel

  font panel_font, <ventfont.pcx>,8,10;       // panel font bitmap

  define  screen_stat_hp_bar_x     540; // X offset for the hitpoints bar
  define  screen_stat_hp_bar_y      10; // Y offset for the hitpoints bar
  define  screen_stat_str_bar_x    555; // X offset for the strength bar
  define  screen_stat_str_bar_y     10; // Y offset for the strength bar
  define  screen_stat_mana_bar_x   570;// X offset for the mana bar
  define  screen_stat_mana_bar_y    10; // Y offset for the mana bar
  define  screen_stat_bar_height    80; // the height of the stat bars
  define  screen_stat_bar_factor  1.0;  // vertical shifting factor

  bmap blood_splash_map <redf.pcx>;    // blood red splash (shown when player is damaged)
  bmap black_map <blackf.pcx>;         // blackout bitmap (used for fades)
```

Screen functions

vscr_fade_out
Fades entire screen to black.

vscr_fade_in
Fades entire screen in from black.

vscr_black_out
Black out entire screen.

vscr_toggle_menu
vscr_toggle_help
vscr_toggle_credits
Toggle the menu, help, and credit screens on and off.

vscr_show_menu
Display the main menu (in game menu).

vscr_show_startmenu
Display the start menu (menu displayed at the start of the game).

vscr_show_admenu
Display the 'after death' menu (menu does not let player continue the current game).

vscr_show_help
Display the help screen.

vscr_show_credits
Display the credits screen.

vscr_show_exit
Display the exit screen.

vscr_show_info
Display info screen. **`my`** must have a valid **`string2`** (this string will be displayed in the window).

vscr_close_all
Hide all panels, init the player's input keys (see **`vinp_init_keys`**), and enable the player's icons.

vscr_close_all_at_start
Hide all panels, disable the player's input keys (see **`vinp_reset_keys`**), and disable the player's icons.

vscr_close_all_at_end
Hide all panels, disable the player's input keys (see **`vinp_reset_keys`**), and enable the player's icons.

Picture Save Manager
Manages the save/load functions. Each game state is saved with a screen-shot and an option line of text. Up to four games can be stored.

Picture Save Predefines

The following can be predefined:

```
define adv_def_save_load;
  define picsize_x 64; // 320x240 shrinked by 5
  define picsize_y 48;

  bmap slot1_map <white.pcx>,0,0,picsize_x,picsize_y;  // default save image
  bmap slot2_map <white.pcx>,0,0,picsize_x,picsize_y;
  bmap slot3_map <white.pcx>,0,0,picsize_x,picsize_y;
  bmap slot4_map <white.pcx>,0,0,picsize_x,picsize_y;

  define picsave_bkgd; // panel has background
  bmap picsavebk_map <black.pcx>,0,0,170,140;
  define picloadbk_map picsavebk_map;
  define panel_font,standard_font;

  define picslot1_x 10;
  define picslot1_y 10;
  define picslot2_x 84;
  define picslot2_y 10;
  define picslot3_x 10;
  define picslot3_y 68;
  define picslot4_x 84;
  define picslot4_y 68;
  define picexit_x  160;
  define picexit_y  130;
// DEFINE PICSAVE_TEXT;   // uncomment to have titles for each slot
```

Picture Save functions

vpic_save

Toggle the save game panel on and off. When the save game panel is on, the player can select a slot by clicking on it. A screen-shot is automatically generated, the player can type in a short text line (in '**`picsave_text`**' is defined), and the game state is saved.

vpic_load

Toggle the load game panel on and off. When the load game panel is on, the player can select a saved game by clicking on it's slot.

NPC Manager

Used to manage non-player characters (**NPC**s).

NPC Predefines

The following can be predefined:

```
define adv_def_npc;
  var vsk_nav_speed[3] = 10, 0, 0; // speed vector used in random navigation
```

NPC functions

vnpc_random_navigation

Used to control the invisible navigation target. Have your NPC follow this target if you want them to 'wander' around.

Player Stat Manager

Used to manage the player statistics (hitpoints, mana, strength, agility, bravery, intuition, experience, bonus points, and level).

Player Stat Predefines

The following can be predefined:

```
define adv_def_playerstats;

define  player_level_two_exp 1000;   // experience need for first five levels
define  player_level_three_exp   2500;
define  player_level_four_exp 5000;
define  player_level_five_exp 8000;
// this value is then doubled for each level after this one
// (ex. 6 = 16000, 7 = 32000, 8 = 64000)

var player_hp = 20; // player's base hitpoints
var player_mana = 0; // player's base mana level
var player_str = 15; // player's base strength
var player_gew = 15; // player's base agility
var player_mut = 15; // player's base bravery
var player_int = 15; // player's base intuition

define  player_rand_hp 5; // the random amount of hitpoints added
define  player_rand_mana 0;  // the random amount of mana added
define  player_rand_str 25;  // the random amount of strength added
define  player_rand_gew 25;  // the random amount of agility added
define  player_rand_mut 25;  // the random amount of bravery added
define  player_rand_int 25;  // the random amount of intuition added

define player_bonus_per_lvl  10; // number of bonus points awarded each level

define player_periodic_hp 0.008;    // number of hitpoints points regenerated every tick
define player_periodic_str0.050;    // number of strength points regenerated every tick
define player_periodic_mana  0.008; // number of mana points regenerated every tick
```

Player Stat functions

vpst_init_stats
Initialize the player's base stats.

vpst_show_char
Show the player's stats panel.

vpst_show_bonus
Show the player's stats panel and allow them to spend their bonus points to increase their skills.

vpst_show_create_character
Initialize the character (**`vpst_init_stats`**) and show the bonus panel (**`vpst_show_bonus`**).

vpst_toggle_char
Toggle the player's stats panel on and off.

vpst_enable_periodic_update
Regenerate stats over time (strength, hit points, and mana) and update the bars in the life panel.

vpst_check_exp
Add the given experience (stored in the variable **`give_exp`** before calling this function) to the player's experience (**`player_exp`**) and check to see if the player has reached the next level.If the player has reached the next level, increase the player's level (**`player_lvl`**) and set the next level's experience goal (**`next_lvl_exp`**), give the player bonus points (**`player_bonus_per_lvl`**) for stats, and show the bonus panel.

Item Manager

Used to manage items.

Item Predefines

The following can be predefined:

```
define adv_def_items;
  sound item_hit_snd,<sack.wav>;   // sound played when item hits ground
```

Item Overridden functions

vitm_create_item

Create an item depending on the current **mouse_object** variable. Every inventory item should be included in this function. Example:

```
function vitm_create_item()
{
   if (mouse_object == m_fish)
   {
      create <fish_ov.pcx>, my_pos, entity_fish_event;
   }
   if (mouse_object == m_key)
   {
      create <key_ov.pcx>, my_pos, entity_key_event;
   }
   ...
}
```

Item functions

vitm_init_item

Initialize an object so it can be used in **venture**. Call after you've set up all the item's parameters. Example:

```
// Desc: fish entity action (attached to fish)
action entity_fish
{
   my.ent_id = m_fish;  // set entity id to fish
   my.facing = on;      // always faces player
   my.ent_scale = 1.0;  // set scale (no pun intended :) )
   MY.AMBIENT = 60;     // light value
   MY.STRING1 = fish_str;  // touch text
   vitm_init_item(); // init this item (venture.wdl)
}
```

vitm_throw_item

Throw an item in the direction of the mouse pointer. **my** must be set to an item before calling.

vitm_drop_item

Drop an item at the player's feet. **my** must be set to an item before calling.

vitm_release_item

Release the item that the player is currently holding (**mouse_object**).

Inventory Manager

Used to manage the player's inventory.

Inventory Predefines

The following can be predefined:

```
define adv_def_invent_p;

bmap invent_map, <invent.bmp>;     // 12 box inventory panel
bmap invent_item_map, <items.bmp>;// inventory items

bmap i1_map, <invent.bmp>, 2, 2, 62, 70;   // 1 box of the 12 box inventory panel
bmap i2_map, <invent.bmp>, 66, 2, 62, 70;
bmap i3_map, <invent.bmp>, 2, 72, 62, 62;
bmap i4_map, <invent.bmp>, 66, 72, 62, 62;
bmap i5_map, <invent.bmp>, 2, 135, 62, 62;
bmap i6_map, <invent.bmp>, 66, 135, 62, 62;
bmap i7_map, <invent.bmp>, 2, 197, 62, 62;
bmap i8_map, <invent.bmp>, 66, 197, 62, 62;
bmap i9_map, <invent.bmp>, 2, 260, 62, 58;
bmap i10_map, <invent.bmp>, 66, 260, 62, 58;
bmap i11_map, <invent.bmp>, 2, 320, 62, 62;
bmap i12_map, <invent.bmp>, 66, 320, 62, 62;

define invent_p_invent_temp_item_dx 70;// width of each item in invent_item_map

define invent_p_x      387;   // inventory panel upper left border position
define invent_p_y        4;

define invent_p_win_dx 70;    // cutout window size
define invent_p_win_dy 70;

define invent_p_row_1       2;   // row offsets for inventory panel
define invent_p_row_2      72;
define invent_p_row_3     135;
define invent_p_row_4     197;
define invent_p_row_5     260;
define invent_p_row_6     320;

define invent_p_col_1      2;   // column offsets for inventory panel
define invent_p_col_2     66;
```

Inventory Overridden functions

vinv_reset_critical_items

Reset the game critical items. Example:

```
function vinv_reset_critical_items()
{
   // start without sword or diamond
   __has_sword = false;
   __has_diamond = false;
}
```

vinv_critical_items

Check to see if a critical item has been picked up. Example:

```
function vinv_critical_items()
{
   if (invent_box_content == m_sword)
   {
      __has_sword = true;
      on_space = vcom_create_sword;
   }
   if (invent_box_content == m_gem)
   {
      __has_diamond = true;
   }
```

```
}
```

Inventory functions

These functions can be used to handle the player's inventory.

vinv_reset_inventory

Reset the player's inventory (zero out all values).

vinv_show_inventory

This function makes the inventory panel visible to the player.

vinv_hide_inventory

This function hides the inventory panel from the player.

vinv_toggle_inventory

Toggle between **show_inventory** (if the inventory panel is currently hidden) and **hide_inventory** (if the inventory panel is currently visible).

vinv_put_item_inventory

Put the item currently stored in '**invent_temp_item'** into the next empty box in the player's inventory. If no slot is available the item is released (**vitm_release_item**).

vinv_delete_item_inventory

Remove the first item in the player's inventory that matches the item value currently stored in '**invent_temp_item**'. '**venture_return**' is set to 1 if the item was found, 0 otherwise.

Combat Manager

Used to manage close combat.

Combat Predefines

The following can be predefined:

```
define adv_def_combat_snd;

sound weapon_swing_snd, <swing.wav>;
sound weapon_hit_snd, <hit.wav>;
sound player_hurt_snd, <ahh.wav>;      // sound played when player takes damage
```

Combat functions

vcom_player_defense

This function is called to handle attacks on the player. Use the **vent_npc_str**, **vent_npc_mut**, and **vent_npc_gew** variables (set by the attacking entity before calling this function) to calculate an attack on the player. The player uses his **player_mut, player_gew**, and **player_int** variables to defend. Damage to the player is calculated (**player_current_hp** is reduced) and (if damage is greater than zero) **vcom_player_hit** is called.

vcom_player_attack

This function is called to handle attacks by the player. Use the player's **player_mut, gew**, and **str** to calculate the attack. Use the **vent_npc_int**, **vent_npc_mut**, and **vent_npc_gew** variables (set by the defending entity before calling this function) to calculate the defense from the player's attack. Damage is calculated and applied to the calling entity's **_health** value. If the entity's value **< 0**, its **_state** is set to **_state_dead**.

vcom_player_hit

This function is called to handle when the player gets hit. If the hit has reduced the player's hit points to zero or less, it calls **vcom_player_dead**. Else, it plays the hurt sound, and flash the blood panel.

vcom_player_dead

This function is called to handle when the player dies. Reset the keys, show blood panel, freeze the game, and show the 'after death' menu.

vcom_create_sword

Create a sword object at **my_pos**. Set the **[Space]** key to remove the sword.

vcom_direct_attack

Apply damage direct to a target by reducing **my._health** by the value currently stored in **hp_minus**. If it kills the target, set its flag to **_npc_dead**.

Dialog Manager

Used to manage NPC dialog.

Dialog Overridden functions

vdia_make_talk

Used to branch through the various NPC-to-player dialogs. Example:

```
function vdia_make_talk()
{
   vscr_close_all();
   vinp_reset_keys();

   __icon_active = off;     // don't let player use icon buttons

   if (npc_entity.ent_id == m_testnpc)
   {
      if (npc_entity._dialog_state == 11)
      {
         dialog_txt.string = testnpc11_str;
         dialog1_button_panel.visible = on;
         actor_dialog_b1 = 12;
         actor_dialog_b2 = 13;
         exit1 = off;
         exit2 = off;
      }
   ...
   }
   vdia_show_dialog();
}
```

Dialog functions

vdia_show_dialog

Show the NPC dialog panel.

Input Manager

Used to manage the player's input.

Input Predefines

The following can be predefined:

```
define adv_def_cursor_map;
  bmap mouse_l_map, <arrow.pcx>;    // mouse cursor image map
```

Input Overridden functions

vinp_set_mouse_map_to_item

Used to set the mouse pointer to the current '**mouse_object**'. Example:

```
functon vinp_set_mouse_map_to_item()
{
   // set the mouse cursor's "hot spot"
   mouse_spot.x = 4;
   mouse_spot.y = 4;
   if (mouse_object == m_obj_1)
   {
      mouse_map = object1_map;
   }
   if (mouse_object == m_obj_2)
   {
      mouse_map = object2_map;
   } ...
}
```

Input functions

vinp_clear_mouse_to_inventory

If there is an object on the player's mouse cursor, remove it and place it in the first empty slot in the player's inventory.

vinp_mouse_init

Initialize the mouse settings.

vinp_mouse_toggle

Switches the mouse pointer on and off

vinp_show_mouse

Switch mouse pointer on.

vinp_hide_mouse

Switch mouse pointer off.

vinp_reset_keys

Deactivate keyboard input.

vinp_init_keys

Set keys to default key mapping.
Note: you can override this function if you want different key mappings in your game.

touch text manager

Used to manage 'touch text'.

vttx_show_touch_text

Show touch text.

vttx_hide_touch_text

Hide touch text.

vttx_show_wrong_item_text

Used to indicate that the item being used is the wrong one.
Hide the current touch text, display the wrong item text for 2-seconds, and then redisplay old touch text.

Misc Manager

Used to manage miscellaneous functions.

vmsc_exit

Fade to black and exit the game. Display the exit text ('**vstr_exit_txt**').

Appendix

WDL traps

WDL is an easy language - usually. But creating a real time game sometimes produces problems that you normally won't encounter in simple programming tasks, like doing some web page effects with javascript. For example, in a commercial game thousands of functions and actions are running at the same time and can interfere with each other. If you don't keep that in mind, you can run into trouble. In the following we've collected some common problems happening not only to beginners, but also to experts!

The dreaded wait() instruction

The following code that should produce seven sounds won't work. However it works if we delete the **wait(1)** line! Why?

```
function seven_chimes()
{
   temp = 0;
   while (temp < 7) // repeat until temp is 7
   {
      play_sound chime_snd, 50; // play a chime
      temp += 1;       // increase temp
      wait(1);
   }
}
```

The **temp** variable is used by a lot of functions for intermediate results. All those other functions run during the **wait()** time. At first encountering the **wait(1)**, **temp** has certainly a value of **1** – but after that, it can have any value, depending on what the other actions did with it.

Note: Look on **wait()** as an instruction that definitely can change **all** external variables and synonyms. Only the **my** synonym is guaranteed to keep its value within an action.

Those bloody invalid synonyms

The following two actions are assigned to two entities in the level. Yesterday they worked perfectly. But today, after changing something in the level at a totally different place, suddenly an "**invalid synonym usage**" error message is spat out by the second action! How come?

```
synonym monster { type entity; }
```

```
action become_monster {
   monster = my;   // set a global synonym, that can be used by other functions
}

action get_monster_height {
   my.z = monster.z;  // use the synonym to set my vertical positin to the monster's
}
```

The second action requires the **monster** synonym being already set. However this depends on whether the first action was started before the second – and there's only a 50% chance. It is not defined in which order the actions assigned to entities are started at level loading. Replace your second action by:

```
action set_monster_height {
   while (monster == null) { wait(1); }// wait until there is a monster
   my.z = monster.z;
}
```

Note: If using a synonym – even a simple one like **my** or **you** – always think about the possibility that it could be undefined at that time!

Bad timing

Using a variable for passing information to a function sometimes does not work. I want to emit a blue and a green particle, but I always get two green particles! What is wrong in the following code?

```
var particle_mode = 0;

...
particle_mode = 2;  // this should be a blue particle
emit 1,my.x,particle_start;
particle_mode = 1;  // this should be a green particle
emit 1,my.x,particle_start;
...

function particle_start()
{
   if (my_age == 0)
   {
      if (particle_mode = 0) { my_color.red = 255; my_color.green = 0; my_color.blue = 0; }
      if (particle_mode = 1) { my_color.red = 0; my_color.green = 255; my_color.blue = 0; }
      if (particle_mode = 2) { my_color.red = 0; my_color.green = 0; my_color.blue = 255; }
   ...
}
```

Unlike calling a function directly, functions indirectly started by instructions like **emit**, **create**, **load_level**, **load** etc. are not guaranteed to start immediately after that instruction. They can start up to two frame cycles later, and sometimes not even on the same PC in multiplayer games! Particle functions, for example, run on the clients when **emit** is performed on the server. In the above example, both particle actions start with **particle_mode** set to **1** (or to any other value **particle_mode** may get afterwards). The solution would be to use different particle functions.

Wrong mathematics

I found the following formula for calculating a two dimensional distance to the origin in my mathematics book – however in WDL, it sometimes delivers a wrong result!

```
dist2d = sqrt(my.x*my.x + my.y*my.y);
```

The absolute value of all variables and intermediate results must lie between 0.001 and 1000000. So the above formula becomes invalid if **my.x** or **my.y** exceeds 1000 (because then the intermediate result **my.x*my.x** exceeds 1000000). Solution: If you expect the intermediate results of formulas to exceed 1000000, limit them by multiplication by a carefully choosen factor, and then correcting the whole result:

```
dist2d = 10000 * sqrt((0.01*my.x)*(0.01*my.x) + (0.01*my.y)*(0.01*my.y));
```

This mathematically identical formula gives **my.x** and **my.y** a valid range between 10 and 100000, and delivers zero if they are below 10. In normal cases the game variables won't exceed the critical boundaries, the only exception can happen when multiplying two distances.

The event trap

The following function is intended to teleport the player to the location (100,200,0) if he touches a teleporter entity which the **teleport1** action is attached to. However it doesn't work:

```
function tp1_event()
{
   you.x = 100;
   you.y = 200;
   you.z = 0;
}

action teleport1 {
   my.enable_impact = on;
   my.event = tp1_event;
}
```

The event function is triggered by **event_impact** from within a **ent_move** instruction. So the **ent_move** was not finished at the time the event function was executed. The **event** function changes the position of the entity, but **ent_move** does the same, and positions the entity to its calculated target position upon termination. This way the displacement by the event function has no effect. Solution: insert a **wait(1)** at the begin of the event function, so that **ent_move** can be finished before:

```
function tp1_event()
{
   wait(1);
   you.x = 100;
   you.y = 200;
   you.z = 0;
}
```

Note**:** **Never** use instructions that change something in the level, or can produce events themselves, in the first cycle of an event function! This also means instructions like **create**, **remove**, **ent_move**, **trace**, **shoot**, **scan**... always insert a **wait(1)** when using them in events.

Attaching entities to each other

We have a model entity with a bright red light at it's very centre. For improving the light effect, we've attached a flare sprite to that model:

```
action red_model {
   create <light.pcx>,my.x,flare_light;// attach light sprite
   patrol(); // perform move behaviour
}

action flare_light {
   my.flare = on;  // set the flare flag
   my.bright = on; // make light more brilliant
   my.facing = on; // always face the camera
   while (1) {
      my.x = you.x;// place the sprite at the xyz position of the entity who created it
      my.y = you.y;
      my.z = you.z;
      wait(1);
   }
}
```

We expected that the light sprite is fixed to the center of the model, and moves together with it. However it always seems to lag behind by one frame. And even worse, the model now moves at only half the speed as before! What's happening?

The second problem is easy to fix: We've forgotten to make the light passable. So the model permanently collides with it's own light, which halves its speed. Entities attached to other ones must get the **passable** flag:

```
action flare_light {
   my.passable = on;
   my.flare = on;  // set the flare flag
   ... // etc.
```

But what produces the strange time lag? If two entities influence each other, it's important to keep in mind the order of their two simultaneously running actions. They won't run really at the same time. In the example, during one frame the WDL function scheduler runs first the **flare_light** action, and then the **red_model**'s **patrol** action. Functions that are started first will run first. So the **flare_light** is always placed to the position the **red_model** had in the frame before. The solution is simple: By changing the order of the actions ...

```
action red_model {
   patrol(); // perform move behavior first
   create <light.pcx>,my.x,flare_light;// attach light sprite and start it's action after that
}
```

...the light stays perfectly at the entities' centre. For the same reason, the **drop_shadow()** function must always be called after the movement function of an entity.

Appendix B: Starting the Engine

The development version of the A4 or A5 engine – **acknex.exe** - is located in the BIN subfolder. Except from within WED, you can also start it directly by giving the command

acknex.exe *mapname*.**wmp** [*options*]
or
acknex.exe *scriptname*[.**wdl**] [*options*]

from within the folder where your map or script file is located. Or you can create a windows shortcut and assign that command line to it. Except from the game itself, there are three more files required for the engine to run. These are also normally located in the BIN folder, and can either be copied from there into the game folder, or re-created directly:

acknex.mdf – the engine error message file
acknex.wdf – the starter definition file, which can also be customised by WED professional version,
palette.raw – the palette file, which can also be produced by simply renaming the ***levelname*.raw** file that is created by the **build** process.

Acknex.exe is part of the development system, and must never be given away. For distribution of a game, WED's **publish** and **resource** functions will create a EXE distribution version of the engine, named after the application.

Command Line Options

The same command line options are valid for the EXE runtime engine as well as for the **acknex.exe** development engine. Thus the published game can be started with the following command line options, given behind the name of the EXE module:

-nx *number*
Size of the nexus in megabytes. The nexus is the internal data structure for picture rendering. The nexus size depends on the size and complexity of the biggest level of the game. The bigger the nexus, the more complex scenes may be shown, and the more entities can be used - but the more virtual memory is used - if real memory is not available, it's taken from the harddisk. The default value for the nexus is 40 megabytes.

A nexus too small for a scene can cause a sudden frame rate drop, or even give an engine error message. If you ever encounter a "**nexus too small**" or "**too much entities**" message, the engine must be started by giving a nexus size higher than that you've used before (e.g. **–nx 80** for 80 MB nexus size and 800 entities).

-dir *name*
Gives the folder defined for saving games and screenshots; will override any **savedir** name defined in the WDL script.

-d *name*
Defines the name *name* for further evaluation in the WDL script through **ifdef**. This has the same effect as giving **define** *name*; within the WDL script. This way, arbitrary game options, like starting screen resolution, D3D or not, difficulty grade etc., can be set via the command line.

-diag
Outputs diagnostic messages into the **`acklog.txt`** file in the current folder.

-wnd
Will start in window mode instead of fullscreen mode. As long as you haven't redefined the **`on_enter`** function, you can always switch between window and fullscreen mode by **[Alt-Enter]**. Please note that using 16-bit or 32-bit video depths (D3D mode), window mode only works if the desktop colour depth is the same (high or true colour). In 8-bit video depth the window can be opened in all desktop resolutions.

-n3d
Will force the engine internal software renderer in 8 bit video mode, and not use D3D, even if 16-bit or 32-bit video mode is given within the WDL script.

-s3d
Forces the external D3D software renderer in 16-bit or 32-bit modes. The D3D software renderer is dead slow, but can be used for taking screenshots, or for determining whether a certain visible fault is caused by the engine or by the D3D card drivers.

-w3d
Will force weak D3D mode, even when the 3D card pretends to support multiple textures. This may be necessary for some old 3D cards. See the following section.

-crd *number*
The 3D card with the given number will be selected, instead of the default one. This is useful for systems with more than one 3D card. Only active 3D devices will be selected. The order or the cards is not determined, so for selecting a second card sometimes "**`1`**" and sometimes "**`2`**" must be given. The startup message will give the name of the card.

-nj
Disables the joystick.

-nm
Disables the mouse.

-sv
Starts a new multiplayer session in server mode (some editions only).

-cl
Joins to a multiplayer session in client mode (some editions only).

-pl *name*
By this option an individual name can be given to each client (up to 16 characters). Otherwise the engine will generate a client's name.

-com *number*
Uses a serial link with the given COM port number (**`1...4`**) for 2-player sessions(some editions only).

-modem
Like **`-com`**, but uses a modem connection (some editions only). At game start you will be prompted which of your modems to use, and which phone number to be dialled. If you don't

enter a phone number, the connection will start as a server, and will be waiting for another PC to call and connect.

-ipx
Like **-com**, but uses a local area network (LAN) with IPX protocol for multiplayer sessions (some editions only).

-tcp
Like **-ipx,** but uses either a local area network (LAN) with TCP/IP protocol, or an active Internet connection (some editions only). At game start you will be prompted for the server's domain name or IP address to connect. If you don't enter one, the PC will open the session as a server, and will be waiting for other PCs to connect. If your server doesn't have a domain name, you can find your current IP address with the **WINIPCFG** program in your Windows folder. On a local network the IP address of connected PCs can be found by the **NETSTAT** program. Both programs are available on each Windows PC.

If not changed by WDL, the following keys are active within the game or withing a level walkthrough:

[F2]	Saves the game into the file "**save_0.sav**".
[F3]	Loads the last saved game.
[F5], [Shift-F5]	Toggles the video resolution.
[Alt-Enter]	Toggles window and fullscreen mode.
[F6]	Takes screenshots into the file "**shot_n.pcx**".
[F10]	Quits the game.
[F11]	Toggles Gamma (8-bit mode only).
[F12]	Toggles music and sound in three steps.

The following keys are active in the development engine, but not in the distribution engine:

[0]	Toggles the direct movement mode for the player entity and the camera in three steps.
[G]	Toggles camera collision detection.
[D]	Activates a default **debug panel**, which displays for the current fps, camera XYZ position, camera angle, and the D3D video memory amount consumed by the current level.
[Tab]	Activates a blinking cursor where WDL instructions can be entered. Some instructions like **exit** or **load_level()** can't be entered directly.

D3D Considerations

In 16- and 32-bit modes a 3D accelerator is activated, if any is found. For using the full D3D feature set, **DirectX 7.0** is required (if not already installed, the latest DirectX version can be downloaded from the Microsoft site). 3D accelerators differ in features, speed and in the maximum texture size they can display. Depending on the characteristics of the hardware, the engine uses different algorithms to generate textures, shadows, and effects. The engine always tries to squeeze the best performance and image quality out of the hardware. An indicator for the performance of your hardware and the maximum texture size is the message in your starting window:

no d3d device – 16/32 bit modes disabled
D3D is not properly installed on your PC, or the DirectX version is too old. The engine can only run in 8-bit mode.

rgb emulation device detected - 1024x1024 textures supported
D3D is installed, but no 3D hardware was found. This is normally the case with most laptops. 16-bit modes are emulated by software, 32-bit modes are not available. 16-bit rendering is slow and only useful for screenshots.

d3d hal incapable device detected - 1024x1024 textures supported
A very old D3D accelerator was detected, which lacks essential 3D features. 16-bit modes are possible, but not recommended – the pictures will lack shadows and some other effects. 32-bit modes are not available.

d3d hal weak device detected – 256x256 textures supported
d3d hal slow device detected – 256x256 textures supported
A cheap, outdated 3D device was detected, mostly a Voodoo/3dfx card. 'Slow' however is better than 'weak'. Textures, sprites or model skin triangles larger than 256x256 pixels won't be displayed (but there are also slow or weak devices that support 1024x1024). 16-bit and 32-bit modes are remarkably slower, and look worse as in non-weak mode.

d3d hal good device detected – 1024x1024 textures supported
A modern 3D device, capable of multitexture blending operations, was detected. Your application may run with over 65 fps on a 1024x768 screen.

Some old 3D accelerators pretend multitexture capability, but don't really support it. This can also happen if you change your 3D card and mix drivers of your old an new card by improper deinstallation. Such cases are normally detected by the engine, and it will run in weak mode. But if such a card succeeds in deceiving the engine, D3D mode won't display lights and shadows properly, or won't run at all. In such cases the **`–w3d`** command line option must be used to force weak mode. Please note that some old Voodoo/3dfx versions can only run in fullscreen mode (max. 640x480) and do not support reserved texture space.

Error Messages

Quite often you will encounter error or warning messages on compiling or running a level. Most error messages occur during scanning the WDL file, and indicate syntax errors. Your have mistyped something, or referred to a file or object that does not exist. The WDL file and line in question (if a any) is given, so you can easily correct the mistake. All other errors that are not self-explaining and obvious are listed here. Warnings and error messages are only issued by the development system, not by the runtime module of the final game.

Errors during engine startup

error e351: corrupted engine (possible virus!)
An attempt to hack or modify the runtime module was detected, probably due to a virus infection.

error e356: problem with wdl script
The WDL script could not be executed. Either it contains a syntax error that is indicated. Or it was not found in the current directory. Or it was modified after publishing.

error e1003: missing basic palette (palette.raw)
For some reason the initial palette file is missing. You can replace it by renaming the palette ***levelname*.raw** that was created by the last **`build`** process.

error e1004: insufficient memory or disk space
Your hard disk is full. Erase some stuff, or buy a bigger one.

error e1005: invalid level
The runtime module was created for a different game - the resource file doesn't fit. Create it anew.

error e1241: d3d device open failure
error e1242: video device open failure
DirectX is not properly installed on your PC. Install DirectX 7.0 or above.

Errors during loading levels, entities or animation

error e1100: can't open level
error e1101: bad wmb format
error e1103: bad colormaps
error e1109: level contains no palette
The given WMB file is not found or is corrupted. Re-build it.

error e1301: can't open file
error e1302: bad mdl format
error e1303: bad mdl bounds
The given model file is not found, corrupted or has a wrong format.

error e1199: unknown entity type
error e1321: invalid entity
You've tried to load an entity that is neither a sprite, nor a map, nor a valid model.

error e1104: invalid texture size
error e1105: bad texture animation
The WMB file contains textures that have an invalid size, or their animation cycle does not begin with **+0** or is interrupted.

warning w1107: texture size no power of 2
This warning is only issued if the **warn_level** variable is set to **2** or above. The texture is displayed, but in reduced quality.

warning w1108: texture too big for 3d card
warning w1114: sprite too big for 3d card
This warning is only issued when using a Voodoo 3D card, or if the **warn_level** variable is set to **2** or above. Voodoo cards are not able to display textures of more than 256 pixels in size. Through a software trick the engine can get Voodoo to display up to 1024x1024 surface textures (not sprites), but in reduced quality.

error e1111: invalid pcx/bmp format
error e1112: invalid size
error e1115: bad tga format (32 bit rle only)
The bitmap file contains an invalid format or a wrong texture size. The colour depth must be 8 bit or 24 bit on normal textures, and 32 bit on textures with alpha channel.

error e1113: bitmap too big for 3d card
The bitmap is so large that it does not fit into the video memory (e.g. bitmaps of 4096x4096 or larger).

warning w1305: triangle too big for 3d card
This warning is only issued when using a Voodoo 3D card, or if the **warn_level** variable is set to **2** or above. Voodoo cards are not able to display model triangles of more than 256 pixels in size.

warning w1306: entity too big for this level
This warning is only issued if the **warn_level** variable is set to **2** or above. The entity is too big and split in too many parts by the BSP tree at it's current position. Therefore the BSP culling can not be used for that entity, resulting in a slower frame rate. The number of BSP splits for an entity can be minimized by placing its origin at its center position.

error e1310: bad scale value
You gave a negative or otherwise strange entity scale.

error e1320: action not found
You've renamed or erased the entity behaviour action out of the WDL file.

warning w1351: can't open file
warning w1352: can't play sound
warning w1361: not found or no valid avi
The sound or animation file was not found or has an invalid format.

Runtime Errors

error e1201..1211: nexus too small
What it says: You have to increase the nexus (**-nx** command line option) for this level.

error e1220: general error
An internal crash happened. Note the circumstances, and ask Conitec for help.

error e1230: insufficient memory for d3d textures
Your 3D card has by far not enough memory for even considering running this level.

error e1231: insufficient disk space left
Your hard disk it full. Erase some stuff.

error e1240: palette missing for transparency
error e1243: palette missing for fog
You tried transparency or fog in 8-bit mode without having loaded a level with a valid fog palette.

error e1244: 3d card requires weak mode (-w3d)
Your 3D card does not support a feature from it's own device capability list. You have to force weak mode by the **–w3d** command line option.

error e1245: 3d card internal memory failure
Your 3D card encountered an internal driver bug. Get fixed drivers from the manufactorer's site. Inbetween, starting in weak mode helps sometimes.

error e1299: demo time expired. order the full version at www.conitec.net
What it says.

error e1340: more entities than max_entities
What it says: You have to increase either the nexus or **max_entities**.

Client/Server communication errors

warning w1401: client/server communication trouble
warning w1402: client/server buffer overflow
The client/server connection is broken, or too much entities were created at the same time.

warning w1403: too much clients
More clients try to connect than entities are available in the level. The client is rejected. Increase `max_entities`.

warning w1404: incompatible engine versions
The client uses an engine with a different protocol version, and is rejected.

warning w1405: session already exists - can only join it as client
A server tried to open a session that already existed on the net.

warning w1406: session doesn't exist - can only open it as server
A client tried to join a nonexistent session.

Bad WDL code

warning w1501: empty synonym used
You are using a synonym that was not initialized, and therefore does not exist in the scope of this function. Refer to the manual about synonyms and how to use them.

warning w1502: endless loop
You have probably forgotten a `wait()` in a loop of the given action. Read the WDL tutorial about loops.

warning w1503: invalid array index
You gave an index that exceeds the array length (can crash the engine).

warning w1504: negative log
warning w1505: negative square root
warning w1507: division by 0
You used an arithmetic function with a wrong argument (can crash the engine).

warning w1506: too much actions
More actions are running than the scheduler can handle. Probably you are creating actions in an infinite loop.

warning w1508, w1510: can't load from file
You're trying to load a game score or info file that was saved by a different game or WDL file.

warning w1509: avoid save/load in same frame
You're trying to perform several `load` or `load_level` operations at the same time. Only one load operation is possible per frame cycle – the others won't be executed.

warning w1511 - dangerous instruction in event
This runtime warning message will be issued when the given event function contains bad WDL code -a forbidden instruction that changes something in the level or can trigger an event itself, like `move`, `remove`, `trace`, `scan` etc. in the first cycle of the event. Insert a `wait(1)` before

executing such instructions in an event function. This warning is only issued if the **warn_level** variable is set to **2** or above.

WED / Map Builder warnings and errors

load error: error before line xxx
The file opened by WED contains no valid WMP content. There can be a couple of reasons, like a PC crash during saving, a harddisk fault, or manual editing / overwriting of the WMP file. If the WMP file is destroyed, it can be recovered: WED automatically produces two backup files, a **.bak** file at each save operation, and a **.$$m** file at each **build** operation. The latter is only for emergency cases, as the group information is lost.

warning w010: vertex out of plane
The compiler does not like the shape of a particular block. Mostly harmless.

warning w011, w201, w202: null length edge
A block has an edge that is too short - two vertices are too close to each other. If the block number is given, find that block (**Edit -> Find Block**) and erase it.

warning w013: can't create block
The block is ill-shaped, or has duplicate planes or some other problems. It won't be visible in the level, and can produce portal errors. Find that block and erase it.

warning w022: map must be rebuilt
You have tried to update the textures of a map, but the level itself or a texture size has been changed. The map must be rebuilt.

warning w064: block with inconsistent surfaces
You have tried to apply passable (liquid) and solid textures to the same block. You have to decide whether the whole block should be liquid or not.

warning w071: duplicate plane
The given block has two identical, or very similar sides. Find and erase it.

warning w074: duplicate blocks
Two identical blocks at the same position, probably due to a failed copy operation. Erase one of them.

warning w090: region leaking
The sky box is not tight, or an object is outside, or there is no sky box at all. This is normally not a problem. However the outer surfaces of the sky box should have the **none** rendering mode set in such a case to prevent a frame rate decrease due to scanning invisible surfaces.

warning w149: invalid surface size
The texture is applied to a surface that is too large or too small, or belongs to an ill-shaped block. The surface will be displayed with a default texture.

warning w177: can't create portal
warning w110: misplaced portal
An internal error occured that could lead to a clipping problem in the level at a certain place. Mostly due to block errors that were indicated in previous warning messages. Mostly harmless.

critical error: map size
critical error: ...overrun
critical error: too much ...
The map is way too big or contains too much or too complicated polygons to be compiled. Erase a lot of blocks. Replace block objects or prefabs by map entities.

critical error: can't create portals
The map contains bad blocks or other problems. If block errors were indicated before, erase those blocks. Other problems might be too elongated or acute blocks.

Frequently Asked Questions

Q. I've created a runtime module for my game, using the **publish** or **resource** function, but it won't start.

A. Check whether you've got a missing file error messaged during the **publish** or **resource** process. If files are missing, no runtime module will be created. Another possibility is that you've changed one of your files in the CD directory afterwards. In that case you have to **publish** or **resource** again.

Q. My game seems to start in software mode. How can I get the engine to start in 16 bit D3D Mode?

A. In your WDL file, define the **video_depth** variable to 16. Using walkthrough, start with the option line **-d3d**.

Q. The engine shows a remarkable frame rate drop if the camera faces a certain wall. Behind this wall lies a very complex room, but it is not visible from here, and should not influence the frame rate.

A. You have compiled the map with **preview** switched on. In **preview** mode, the frame rate is heavily dependent on the map complexity (whether visible or not).

Q. In 16-bit or 32-bit D3D mode, some surfaces are pitch black, or aren't displayed at all.

A. Your textures are larger than 256x256. Some 3D hardware, Banshee or Voodoo3 by example, doesn't support textures sizes above 256x256.

Q. I have tried to build my own **wad** files by adding 8-bit textures I've created, but I keep getting wrong colors in the level, as well as in WED.

A. You've used the wrong palette in your paint program to create your own textures with. Or you have converted them to 8-bit palettized format without specifying a palette. Adapt your textures to the right level palette. In Paint Shop Pro, it's simply "Load Palette". It must be the same palette as you've assigned in Map Properties.

Q. I have created a new palette, and adapted my textures to it. But now the textures have got little black and white dots in fullscreen mode. In window mode they look all right.

A. Your palette violates the colour 0 = black, colour 255 = white restriction. Edit your palette, change colour 0 and colour 255, and adapt all your textures to it anew.

Q. Why does the frame rate drop when I use small texture scales (<0.1 quants/texel)?

A. This happens on shaded surfaces. The smaller the scale, the more cache surfaces are created, the bigger the cache you'll need, and the more often it needs to be refilled if your **nexus** is too small.

Appendix C: Legal stuff

You have now finished your game, finished the beta test, found a publisher or distributor, and are awaiting your game to appear in the stores. But haven't you forgotten something? What about license fees to pay for distributing the runtime module with the A4 or A5 game engine inside?

Nope. By having legally purchased 3D GameStudio, you have the right to create as many games as you want, distribute them, earn money and become wealthy, without having to bother about paying royalties or license fees. You also have the right to use all textures, maps, models, weapons and sounds delivered as part of the development system.

In exchange for these rights we require three things from you:

- Send us a copy of your published game.
- Display an engine logo for one second at game start.
- Do not use 3D GameStudio for a game that intentionally propagates genocide against a certain group, nation, race or religion existing in today's real world.

In the standard or extra editions, there is a tiny engine 'watermark' corner logo that will be displayed automatically in a screen corner of your choice. It's semitransparent, and you can easily integrate it into a panel to hide it.

It's up to you to create the splash screen with the engine logo. You may add your own game logos and text. The logo must be displayed for at least 1 second (16 ticks) at the beginning of the game. There is a choice of two logos – a dark one (**`logodark.bmp`**) and a bright one (**`logolite.bmp`**). The display the engine version – A4 or A5 – in a cube. You'll find them in the **template** folder.

The logo must be placed near the centre of the splash screen. No part of the cube may be covered by other graphics or text. It may be adapted to your level palette, and resized to match you starting screen resolution, but the cube must cover at least one third of the horizontal screen size. You can display more splash screens with your own logos, but the engine logo splash screen must be the first one of your game.

Except from the Three Requirements, there are some further legal things to bear in mind.

Together with the software you'll receive a password that allows free updates from our web site. You are now allowed to give this password to other persons.

Maybe you're not working alone at the game, but together with a team. You have the right to make copies of 3D GameStudio for backup purposes, but it is illegal to work with them. If several members of your team are working with 3D GameStudio, you do not have to buy a copy for each member – you can buy a team license. For developer groups or school/college classes, team licenses are a way to save costs. All members of the team must use the software for the same project, and the software must not be used for separate projects or outside the team. The team license includes one copy of 3D GameStudio, which may be installed on the PC of each team member. The purchaser of a team license is responsible for updating the software, and must not give away the serial number and update password to anyone else including the team members. For details see the online price list on our internet order form.

If you are using textures, sounds, models or other material from the demo levels of the development system, please note that you must use them for GameStudio applications only. It is illegal to use them for any other purpose. It is also illegal to distribute any other part of the development system, or to modify the distributable runtime module in any way.

The right for royalty free distribution does not include the right to use or broadcast GameStudio applications for commercial purposes at public places, like TV shows, arcade machines, commercial online multiplayer game sites and so on. If you are planning such an event, please contact support to find out whether a license in your case is required or not. For using GameStudio applications in public at places like exhibitions, it is normally sufficient to display the engine logo in regular intervals between the games.

For professional game development companies, there are some advantages to buy a license even for normal distribution. The one-second splash screen logo is not required for licensed games. Licensed runtime modules are only available for the professional edition. To purchase one, you'll need the Magic Key that WED displays for your resource file. We'll send or email the module to you normally within one working day. The one-time license fee depends on the end user price and distribution width, and ranges from $1000 for a shareware game, up to $10,000 for a full-size game distributed world-wide. For details, and for multiple game licenses, please contact support.

Frequently Asked Questions

Q. I need a feature or effect for my game, which seems not to be available in the engine. Can I pay for it to get it implemented?

A. We'll take your money with pleasure. Just contact support. However, maybe the feature you need is already scheduled for the next free update, or an interim version is already available for evaluation. You'll find a list of upcoming features on the user forum

Q. Why does the software consist of a CD-ROM plus a key disk? Why not a CD-ROM alone?

A. To prevent software piracy. The key disk contains your personalized key file with burned-in serial number. This way we could trace any illegal copy – or any game made with it - back to the customer who purchased it, persuade him with strong arguments not to commit piracy any more in his life, and thus fight the evil in the world.

Q. I've sent a technical question to support, but got a cheeky and vague answer.

A. Probably you've asked: “Send me a WDL program that does this or that.” Support then normally answers: “Please read the manual, and program it yourself.”

Index

__bob 113, 115
__fall 113
__jump 113
__remote 120
__repeat 115
__rotate 115, 120
__silent 115, 120
__slopes 113
__strafe 113
__trigger 113, 118
__wheels 113
_ammotype 115f
_armor 117
_banking 113
_bulletspeed 115
_debug_txt 68
_endpos 118
_entforce 113
_firemode 116f
_firetime 116
_force 118f
_health 113, 117
_hitmode 117
_keytype 118ff
_movemode 113
_muzzle_vert 118
_rotate 116f
_silent 116f
_switch 118ff
_trigger_range 118f
_walkframes 113
_weaponnumber 115
&& 54
*= 47
+= 47
< 54
<= 54
!= 54
-= 47
== 54
> 54
>= 54
|| 54
–pl 108
–sv 108
–tcp 108
–w3d 141
/= 47

A

abs 48
acknex.exe 136
acos 48
action 46, 73, 100, 104
activate_shoot 59, 84
activate_sonar 59, 84
actor_fight 117
actor_move 114
actor_turn 114
actors.wdl 114
albedo 78, 99
alpha 79, 86, 90
ambient 78, 80, 91
ammopac 116
ammunition 115
and_select 65, 106
ang 48
animation 77
app_name 94
arc 91
array 71
asin 47f
aspect 91
atan 48
avi 102

B

beep 68
beep_sound 68
bg_color 110
bind 104
bitmap 86, 100
blink_message 120
bmap 44, 73f, 86, 100
bmap_height 67
bmap_width 67
bmp 44, 73, 110
bounce 58, 83f, 103
bounding box 81
break 55
breakpoint; 68
bright 79f
bullet 115
bullet_shot 117
bullet_smoketrail 116
button 87

C

call 112, 120
camera 76, 90
camera_move 114
carry 115
cd_track 64, 93
center_x 90
center_y 90
char_x 89
-cl 108, 137
client 85, 108
client_move 114
clip_range 76, 97
cloud_map 98f
cloud_speed 99
-com 137f
condensed 90
connection 103, 108
console 122
content 56
content_empty 56
content_passable 56
content_solid 56
continue 55
cos 48
-crd 137
create 57, 75, 109

D

d3d 87, 95, 138
d3d_lightres 79, 97
d3d_lines 97
d3d_mode 95
d3d_panels 95
d3d_texbmaps 96
d3d_texlimit 96
d3d_texmemory 95
d3d_texreserved 95
d3d_texsmaps 96
d3d_textotal 96
damage_explode 116
damage_impact 116
damage_shoot 116
database 106
datafile 106
dataview 65, 90, 106
debugger 68
debug panel 138
default 73
define 104f, 136
-diag 137
diameter 91
digits 88
-dir 104, 136
directx 67, 95
dll_close 68
dll_exec 68
dll_exec_vec 68
dll_handle 67
dll_open 67
door 118
doors.wdl 118
doorswitch 120
drawbridge 119
drop_shadow 112, 114
dynamic light 78

E

elevator 82, 118f
else 54
emit 61f
enable_block 83
enable_click 84
enable_detect 60, 84
enable_disconnect 84
enable_entity 83
enable_impact 83
enable_joystick 102
enable_key 102
enable_mouse 102
enable_push 84
enable_release 84
enable_rightclick 84
enable_scan 60, 84
enable_shoot 59, 84
enable_sonar 84
enable_stuck 83
enable_touch 84
enable_trigger 84
endif 104f
ent_alphaset 56
ent_cycle 57
ent_frame 57
ent_move 58
ent_nextpoint 61
ent_path 61
ent_rotate 46, 55, 120
ent_vertex 57
ent_waypoint 61
enter_message 121
entity 73, 75
error messages 139
euler angles 45, 76
event 47, 82, 84, 103, 147
event_block 83
event_click 84
event_detect 84
event_detect □, and you□ xe 60
event_disconnect 84, 109
event_entity 83f
event_impact 83f
event_release 84
event_rightclick 84

event_scan 60, 84
event_shoot 59, 84
event_sonar 59, 84
event_stuck 83
event_touch 84
event_trigger 82, 84
event_type 59f, 82f, 103
exclusive_entity 56
exclusive_global 56
exec 65
execute 64, 108
exit 67
exp 48

F

facing 80
fat 82
file_asc_read 53
file_asc_write 53
file_chr_read 53
file_close 53
file_open_append 53
file_open_game 52
file_open_read 52
file_open_write 52
file_str_read 53
file_str_write 53
file_var_read 53
file_var_write 53
fire_ball 116
fire_laser 116
flag 44
flags 87, 90, 92
flare 56, 79
floor_range 97
fog 91, 99
fog_color 91, 99
font 64, 74, 89, 110
fps_max 94
fps_min 94
frame 57, 77
frc 48
freeze_actors 118
freeze_map 66
freeze_mode 94
function 46

G

gamma 97
gate 119
genius 92
goto 55
gun 115f
gunfx_brass 116

H

handle 63
hbar 88
hidden 79
hit_explo 116
hit_flash 116
hit_hole 116
hit_scatter 116
hit_smoke 116
hit_sparks 116
hslider 88
hull 81
hull_fatmax 81
hull_fatmin 81
hull_narrowmax 81
hull_narrowmin 81

I

if 54, 77, 100
ifdef 104f, 136
ifelse 104f
ifndef 104
ignore_maps 59
ignore_me 59
ignore_models 59
ignore_passable 59
ignore_passents 59
ignore_push 59
ignore_sprites 59
ignore_you 59
in_passable 58f, 103
in_solid 58, 103
include 104
index 71
info 72
infrared 91
inkey 64
inport 65
int 48
invisible 47, 79
ipx 138
-ipx 138

J

joy_ 101
joy_1 101
joy_force 101
joy_raw 101
joy_rot 101
joystick 70, 101f, 137

K

key 120
key_any 102
key_f1 101
key_force 102
key disk 147

L

layer 75, 86, 89f
length 71
lid 119
light 80
lightblue 78
lightgreen 78
lightrange 78, 80
lightred 78, 80
load 64, 66
load_info 66
load_level 66
local 65
lod1 76
log 48
logo 97

M

magic key 147
main 69, 112
map 99
max 48
max_entities 97, 141
max_particles 97
max_x 81
mdl 44, 57, 75
me 57, 63, 73
medipac 117
menu_main 121
menu_show 122
menu.wdl 121
message 69, 103, 108
messages.wdl 120
mickey 100
mid 44, 74
midi_playing 93
min 48
min_x 59, 81
mip_flat 97
mip_shaded 97
mip-mapping 97
-modem 137
morph 57, 75
mouse_calm 69, 101
mouse_ent 101
mouse_force 100f
mouse_left 101
mouse_map 88, 100
mouse_middle 101
mouse_mode 100
mouse_moving 69, 101
mouse_off 122
mouse_on 122
mouse_pos 100
mouse_range 84, 101
mouse_right 101
mouse_spot 100
mouse_time 101
mouse_toggle 122
move 58, 83f, 103
move_mode 58
movement.wdl 112
movie 93
movie_frame 64, 93
multiplayer 94
music 74
my 46f, 57f, 62, 73, 75, 77, 102, 104
my_action 62, 103
my_age 62, 103
my_alpha 62
my_color 62, 103
my_flare 62, 103
my_map 62, 103
my_pos 62, 103
my_size 62, 103
my_speed 58, 62, 103
my_string 66, 69, 103
my_transparent 62, 103

N

-n3d 137
narrow 82, 90
near 81
next_frame 57, 77
nexus 97, 136, 141, 145
-nj 137
-nm 137
nofilter 80
nofog 79
normal 58f, 83f, 103
null 54, 62, 87, 89, 92, 100, 103
nullskill 58
nullvector 93
num_actions 94
num_particles 97
num_visentpolys 98
num_visents 98
num_vismappolys 98
-nx 136

O

offset_x 91
offset_y 89, 91
on_click 69, 89

on_enter 70, 137
on_f 65
on_f1 70
on_join 69
on_joy 70
on_leave 69
on_mouse_left 69
on_mouse_middle 69
on_mouse_right 69
on_mstop 69
on_passable 59, 103
on_receive 108
on_string 66, 69
or_select 65, 106
oriented 80
outport 65
overlay 79, 87

P

palette 73, 97, 144
pan 45, 58
pan_map 87
panel 65, 73, 86
parameter 47
particle_line 121
particle_range 121
particle_scatter 121
particle_trace 62, 121
particle.wdl 62, 121
particles 61f, 121
passable 58f, 82
path 104
patrol 112, 114
patrol_path 114
pcx 44, 57, 65, 73, 110
pi 93
-pl 137
play_cd 64, 93
play_entsound 63
play_loop 63
play_moviefile 64
play_song 63
play_song_once 63
play_sound 63
player_fight 118
player_move 112, 114
pointer 100
pos 76, 86, 89ff, 100
pos_resolution 94
pos_x 86, 89ff
pos_y 86, 89ff
preview 144
publish 104, 144
push 82, 84

Q

quant 44, 77

R

radar 91
radiance 78
random 47
recoil 115
refresh 87
rel_for_screen 50
rel_to_screen 50
remove 57
render_inflate 97
resource 104, 144
result 63f, 67, 102, 104
return 55
roll 45, 58
rotate 58
run 84

S

-s3d 137
save 64ff
save_info 66, 72
savedir 65, 104, 136
scale_x 77
scan 83, 118f
scan_entity 60
scan_path 61
scan_texture 59
scene_angle 98
scene_field 98
scene_map 74, 98
scene_nofilter 98
screen_size 67, 95
screenshot 65
scroll_message 120
select 65, 106
send 66, 109
send_handle 118
send_string 66, 69, 108
separator 106
server 108
session 108
set 47, 73, 100, 110, 120
shadow 77, 80
show_message 120
show_panels 121
sign 48
sin 48
size 89, 91, 95, 110
size_x 91
size_y 89, 91
skill 44, 72, 104f
skin 78
sky_clip 99
sky_curve 99
sky_map 98
sky_scale 99
sky_speed 99
sky sphere 98
snd_playing 63
snowfall 121
software piracy 147
sound 63, 74, 93
sound_vol 93
sqrt 48
stereo 63
stop_movie 64
stop_sound 63
str_cat 51
str_clip 51
str_cmpi 51
str_cmpni 51
str_cpy 50
str_for_asc 52
str_for_num 52
str_len 51, 64
str_stri 51
str_to_asc 52
str_to_num 52
str_trunc 51
stricmp 59
string 44, 64, 72f, 89, 106, 120
sun_angle 80, 99
sun_light 99
sun_map 99
-sv 137
switch_video 67, 94f
synonym 73
sys_day 93
sys_dow 93
sys_hours 93
sys_minutes 93
sys_month 93
sys_seconds 93
sys_year 93

T

tan 48
target 59, 61, 103
-tcp 138
teleporter 120
tex_light 59
tex_name 59
text 64f, 72f, 89, 106, 110
tick 44
tilt 45, 58
time 46f, 77, 93f
title 110
total_frames 93
total_ticks 93
trace 59, 83f
trace_mode 59
transparent 79, 87, 90
trigger_range 82, 84, 113
tune_sound 63
turb_range 100
turb_speed 100
type 44, 73
type" here the name of the entities' mdl xe 75

U

undef 104
unlit 80
unselect 65, 106
use_box 59

V

val 105
var 71
var_info 72
variable 44
vbar 88
vec_add 49
vec_dist 49
vec_for_screen 50
vec_length 49
vec_length() 49
vec_normalize 49
vec_rotate 49
vec_scale 49
vec_set 49
vec_sub 49
vec_to_angle 49
vec_to_screen 50
vector 44, 48, 50, 71
venture.wdl 122
video_depth 67, 95, 105
video_mode 67, 94, 105
video_screen 67
video memory 95
view 65, 69, 76, 90, 97
viewpos 90
visible 79, 87, 90
voodoo 94
vslider 88

W

-w3d 137, 139
wait 56
waitt 56
war.wdl 117
warn_level 94, 140f, 143
wav 44, 74, 93

weapons.wdl 115
wex 74
while 46, 55f, 100
window 66, 88, 110
window mode 110
winend 110
winipcfg 108, 138
winrun 110
winstart 110
-wnd 137

Y

yesno_show 121
you 57ff, 73, 75, 83f, 102
your 57