

Game Maker Language Guide

version 1.4 (June 1, 2000)

by

Mark Overmars

Introduction

Game Maker has a built-in interpreter. This makes it possible to create actions that execute a complicated piece of code. Basically everything you can do with the various actions can also be done in pieces of code. And with code you have many more possibilities. So if you don't like the drag-and-drop, and know a bit of programming, this is the way to go. (Realize though that the code is interpreted and, even though there is some optimisation, on a slow machine, long pieces of code might make your game slow. In particular be careful when putting long pieces of code in step events.)

This document describes the language used. It is a subset of C, with a few additions that are specific to **Game Maker**. The language contains variables (only real numbers) and control structures (conditionals, loops). There are a number of built-in variables and functions that relate to the game.

When you want to use code, there are a couple of things you have to be careful about. First of all, for all your objects and sounds use names that only consist of letters, digits and the underscore '_' symbol. Also be careful not to name objects self, other, global, or sound.

Program

A program consists of a block. A block consists of one or more statements, enclosed by '{' and '}'. Statements must be separated with a ';' symbol. So the global structure of every program is:

```
{
  <statement>;
  <statement>;
  ...
}
```

A statement can again be a block of statements. There are a number of different types of statements.

Assignment

An assignment assigns the value of an expression to a variable (for more information about variables, see below). An assignment has the form:

```
<variable> = <expression>;
```

Expression can be real numbers (e.g. 3.4), strings between single or double quotes (e.g. 'hello' or "hello") or more complicated expressions (see below).

Rather than assigning a value to a variable one can also add it using +=, subtract it -=, multiply it using *= or divide it using /=. (These only work for real valued variables and expressions, not for strings.)

Example: Some useless assignments.

```
{
  x = 23;
  str = 'hello world';
  y += 5;
```

```
x *= y;  
}
```

If statement

An if statement has the form

```
if (<expression>) <statement>
```

or

```
if (<expression>) <statement> else <statement>
```

The statement can also be a block. The expression will be evaluated. If the (rounded) value is ≤ 0 (**false**) the statement after else is executed, otherwise (**true**) the other statement is executed.

Example: The following code moves the object toward the middle of the screen.

```
{  
  if (x<200) x += 4 else x -= 4;  
}
```

Repeat statement

A repeat statement has the form

```
repeat (<expression>) <statement>
```

The statement is repeated the number of times indicated by the rounded value of the expression.

Example: The following code creates five balls at random positions.

```
{  
  repeat (5) create(random(400), random(400), ball);  
}
```

While statement

A while statement has the form

```
while (<expression>) <statement>
```

As long as the statement is true, the statement (which can also be a block) is executed.

Example: The following code tries to place the current object at a free position (this is about the same as the action to move an object to a random position).

```
{  
  while (!is_free(x,y))  
  {  
    x = random(15)*cellsize;  
    y = random(15)*cellsize;  
  }  
}
```

Exit statement

The exit statement simply ends the execution of this piece of code. (It does not end the execution of the game!)

Functions

A function has the form of a function name, followed by zero or more arguments between brackets, separated by commas.

```
<function>(<arg1>,<arg2>,...)
```

Some functions return values and can be used in expressions. Others simply execute commands. Here is a list of all functions available.

Mathematical functions

- **random(x)** returns a random real number between 0 and x
- **abs(x)** returns the absolute value of x
- **sign(x)** returns the sign of x (-1 or 1)
- **round(x)** returns x rounded to the nearest integer
- **floor(x)** returns the floor of x (largest integer smaller than x)
- **ceil(x)** returns the ceiling of x (smallest integer larger than x)
- **frac(x)** returns the fractional part of x
- **sqrt(x)** takes the square root of x
- **sqr(x)** returns $x*x$
- **power(x,n)** returns x to the power n
- **exp(x)** returns e to the power x
- **ln(x)** returns the natural logarithm of x
- **log2(x)** returns the log base 2 of x
- **log10(x)** returns the log base 10 of x
- **logn(n,x)** returns the log base n of x
- **sin(x)** returns the sine of x (x in radians)
- **cos(x)** returns the cosine of x (x in radians)
- **tan(x)** returns the tangent of x (x in radians)
- **min(x,y)** returns the minimum of x and y
- **max(x,y)** returns the maximum of x and y

String functions

Note that you can add strings and compare them (case-sensitive). The following relevant functions exist:

- **chr(val)** returns a string containing the character with code val
- **ord(str)** returns the ascii code of the first character in str
- **string(val)** turns the real value into a string

File IO

There are two functions to do file IO, that is, to save values between times you run the game. This can be used to e.g. save the room the player was last time. It is rather primitive yet.

- **write(ind,x)** writes the value x to the file and give it identifier ind (must be an integer)
- **read(ind)** returns the value of identifier ind in the file (0 if ind does not exist)

Creating and destroying

- **create(x,y,obj)** creates an instance of object obj at position (x,y)
- **change(obj1,obj2)** changes all instances of obj1 into object obj2
- **destroy(obj)** destroys all instances of object obj (if obj=self or obj=other then only the self instance or the other instance is destroyed)
- **change_at(x,y,obj)** changes all instances at position (x,y) in object obj
- **destroy_at(x,y)** destroys all instances at position (x,y)

Checking for objects

- **is_free(x,y)** returns whether the position (x,y) is collision-free for the instance
- **is_empty(x,y)** returns whether the position (x,y) is empty for the instance (the only difference with the previous function is that in this case non-solid objects are taken into account)
- **is_meeting(x,y,obj)** whether the instance meets object obj at position (x,y)
- **nothing_at(x,y)** checks whether there is no object at position (x,y) (the difference with is_empty(x,y) is that there the size of the object is taken into account)
- **object_at(x,y,obj)** returns whether there is an object of type obj at position (x,y)
- **number(obj)** returns the number of instances of object obj

Other routines

- **move_random(obj)** moves all instances of obj to a random free position
- **sleep(numb)** sleeps numb milliseconds
- **sound(numb)** plays the indicates sound number. If you want to use the name of the sound put 'sound.' in front of it, so use e.g. sound(sound.click). Use 0 to stop all sounds.
- **end_game()** ends the game
- **redraw()** redraws the field (normally you don't need this but it is useful if you e.g. sleep a while afterwards and want to make sure the new situation is visible)
- **show_cursor(show)** if show is false the cursor is made invisible inside the playing area, otherwise it is made visible
- **goto_room(numb)** go to the room number numb
- **show_highscore(numb)** shows the highscore table. Numb is the new score. If it is higher than one of the scores in the table, the player can enter his or her name. Normally you would use this as show_highscore(score), but to e.g. just show the highscore list, use show_highscore(-1)
- **show_info()** displays the info form (useful when you hide the toolbar in game runner)
- **show_message(str)** displays a dialog box with the indicated string as a message
- **show_question(str)** displays a dialog box with the string as a question; returns 1 if the user selects yes and 0 otherwise
- **get_integer(str,def)** asks the user in a dialog box for a number; str is the question, def the default value
- **get_string(str,def)** asks the user in a dialog box for a string; str is the question, def the default string
- **check_key(keycode)** returns whether the key with the particular keycode is pressed (see below for keycode constants)

Drawing commands

It is possible to let objects look rather different from their image. There is a whole collection of commands available to draw different shapes. Also there are routines to draw text. These should only be used in the advanced drawing code that you can edit by pressing the button **Advanced** in the object form. When you use them at any other place they have absolutely no effect at all. The following commands exist:

- **draw_image(x,y,obj)** draws the image of object obj with top left at position (x,y). To draw your own image use draw_image(x,y,self). But you can also draw the image of another object. This is very useful when you want objects to have different appearances.
- **draw_subimage(x,y,obj,ind)** draws the subimage ind of the object. This only makes sense when the object is an animated image with multiple subimages.
- **set_brush_color(col)** sets the brush color, that is, the color used for filling shapes. A whole range of predefined colors is available (c_aqua, c_black, c_blue, c_dkgray, c_fuchsia, c_gray, c_green, c_lime, c_ltgray, c_maroon, c_navy, c_olive, c_purple, c_red, c_silver, c_tea, c_white, c_yellow). Other values can be made using as color 256*256*blue+256*green+red where blue, green and red must be values between 0 and 255.
- **set_pen_color(col)** sets the color of the pen (used for outlines, etc.).
- **draw_line(x1,y1,x2,y2)** draws a line from (x1,y1) to (x2,y2).
- **draw_ellipse(x1,y1,x2,y2)** draws an ellipse with (x1,y1) left top and (x2,y2) right bottom.
- **draw_rectangle(x1,y1,x2,y2)** draws a rectangle.
- **draw_roundrect(x1,y1,x2,y2)** draws a rounded box.
- **draw_triangle(x1,y1,x2,y2,x3,y3)** draws a triangle.
- **set_font_color(col)** sets the color of the font used for drawing text
- **set_font_size(size)** sets the size of the font
- **set_font_style(style)** sets the style of the font (0 = normal, 1= bold, 2 = italic, 3= bold italic)
- **set_font_align(align)** sets the alignment of the font (0 left, 1= center, 2 = right)
- **set_font_name(name)** sets the name of the font, e.g. set_font_name('Times New Roman')
- **draw_text(x,y,str)** draws the string with the current font at the indicated place

Please realize that drawing some shape does not change the bounding box of the object used for determining collisions and mouse events. The bounding box is determined by the initial image. You can though change the bounding box yourself by changing the following variables bb_l, bb_r, bb_b and bb_t (see below).

Expressions

Complicated expressions can be used. The following operators exist (in order of priority):

- `&&`, `||`: combine Boolean values (`&&` meaning and, `||` meaning or)
- `<`, `<=`, `==`, `!=`, `>`, `>=`: comparisons, result in true (1) or false (0)
- `+`, `-`: addition, subtraction
- `*`, `/`: multiplication, division

As values you can use number, variables, or functions that return a value. Sub-expressions can be placed between brackets.

Examples: some complicated expressions

```
{
  x = 23*((2+4) / sin(y));
  b = (x < 5) && !(x==2 || x==4);
}
```

Note that `!` in the last expression means not.

Variables

Variables are used to store real values. A variable has a name that must start with a letter and can contain only letters, numbers, and the underscore symbol `'_'`. There are a number of built-in variables that you can check or change in your programs:

Constants

General:

- **true** 1
- **false** 0
- **pi** 3.1415...

Colors:

- **c_red** red color
- **c_blue** blue color
- **c_green** green color
- **etc.**

Virtual keycodes:

- **vk_left** keycode for left arrow key
- **vk_right** keycode for right arrow key
- **vk_up** keycode for up arrow key
- **vk_down** keycode for down arrow key
- **vk_enter** enter key
- **vk_escape** escape key
- **vk_space** space key
- **vk_shift** shift key
- **vk_control** control key
- **vk_alt** alt key
- **vk_f1** F1 key (for the other function keys, just add the difference, e.g. F5 is `vk_f1+4`)
- **vk_numpad0** zero key on the numeric keypad (for the others, just add the numbers)
- for the letter key use the ascii value of the capital, e.g. `ord('D')`
- for the number keys, just use the ascii value of the number, e.g. `ord('5')`

General variables

- **room** the number of the current room (use `goto_room` to change it)
- **lastroom** the number of the last room (you cannot change this)
- **gamespeed** the speed of the game in steps per second (change this to speed up or slow down the game)
- **score** the current score (increase the score with e.g. `score += 10;`)
- **roomwidth** the width of the room in pixels (cannot be changed)
- **roomheight** the height of the room in pixels (cannot be changed)
- **cellsize** the size in pixels of a cell (cannot be changed)
- **back_color** background color (if there is no background image)
- **back_x** horizontal offset of the background image

- **back_y** vertical offset of the background image
- **back_hspeed** horizontal speed of a scrolling background image
- **back_vspeed** vertical speed of a scrolling background image
- **mousex** x-coordinate of the mouse
- **mousey** y-coordinate of the mouse
- **mousebutton** currently pressed mouse button (0=none, 1=left, 2=right, 3=middle)
- **lastkeypressed** the keycode of the last key pressed on the keyboard that has not yet been handled by some keyboard event
- **keypressed** the keycode of the currently pressed key (0=no key pressed)

You can set the last five variables. In general this does not make sense but there are cases where it helps. For example, when you want to react to a mouse click only once and not as long as it is pressed, you can set `mousebutton` to 0 once you handled it.

Instance variables

Each instance of an object has its own built-in variables.

- **x** the x-coordinate
- **y** the y-coordinate
- **hdir** the horizontal direction (-1 = left, 0 = no motion, 1 = right)
- **vdir** the vertical direction (-1 = upwards, 0 = no motion, 1 = downwards)
- **hspeed** the horizontal speed (in pixels per step)
- **vspeed** the vertical speed (in pixels per step)
- **alarm** the value of the alarm clock (in steps)
- **object** the object type of the instance
- **visible** whether the instance is visible (invisible object still are active and create collision and meeting events; you only don't see them)
- **active** whether the instance is active (be careful when you change this: an object can make itself inactive but it can never make itself active)
- **solid** whether the instance is solid (so you can temporarily make an object not solid)
- **w** the width of the image (cannot be changed)
- **h** the height of the image (cannot be changed)
- **bb_l** left side of the bounding box (with respect to the top left corner of the object, indicated by position (x,y))
- **bb_r** right side of the bounding box
- **bb_t** top side of the bounding box
- **bb_b** bottom side of the bounding box

So, for example, to set the alarm for the object whose event you are dealing with, give as code

```
alarm = 25;
```

You often might want to use and set variables in other instances. This can be achieved by preceding the variable name by the name of the object and a dot. So, for example, to address the x-coordinate of the ball use

```
ball.x = 25;
```

Now you should wonder what happens when there are multiple instances of the object ball. Well, all have their x-coordinate set to 25. If you read the value of a variable in another object you get the value of the first instance of that object. Now consider the piece of code

```
ball.x += 32;
```

At first you might think that for each ball the x-coordinate is increased with 32. Unfortunately this is not true. The above statement is the same as

```
ball.x = ball.x + 32;
```

So we first take the x-coordinate of the first ball, add 32 to it, and set this value in the x-coordinate of all other balls. This example should make clear that you have to be very careful in using and setting variables in objects of which there are multiple instances. To achieve the result you want you should use the `forall` construction

```
forall (ball) x += 32;
```

See below for more information on this construction.

There are two special object names that you can use: `self` refers to the instance itself; `other` refers to the other instance involved in a collision or meeting event. So for example you can use a piece of code like

```
{
  other.hspeerd = self.hspeerd;
  other.vspeed = self.vspeed;
}
```

Note that you hardly ever need to use `self` because you can simply use the variable names without it. (It is though useful in routines. E.g. you can use `destroy(self)` to destroy yourself.)

Extra variables

You can also create new variables by assigning a value to them (no need to declare them first). If you simply use a variable name, the variable will be stored with the current object instance only. So don't expect to find it when dealing with another object (or another instance of the same object) later. You can also set and read variables in other objects by putting the object name with a dot before the variable name.

To create global variables, that are visible to all object instances, precede them with the word `global` and a dot. So for example you can write:

```
{
  if (global.doit)
  {
    // do something
    global.doit = false;
  }
}
```

Forall construction

As indicated above, you have to be very careful when using variables in other objects for which there are multiple instances. To do this in a safe way, use the `forall` construction:

```
forall (<expression>) <statement>
```

The expression should indicate an object, so preferably, only use this with an object name or the variable `object`. The statement is executed for each instance of this object, as if the current (`self`) instance is that instance. Let me give some examples. To move all balls to the left write:

```
forall (ball) x -= 32;
```

To count the number of balls that lie above the current object use:

```
global.yy = y;
global.nn = 0;
forall (ball)
{
  if (y < global.yy) global.nn += 1;
}
```

Note the use of global variables here. You cannot use local variables because they won't exist in the ball instances and hence have no value within the `forall` statement. The `forall` construction is very powerful, but use it with care. You easily get unexpected behaviour. (E.g., never use `self` or `other` in the `forall` expression; they refer to instances, not objects! But you can use `self.object`.)

Comment

You can add comment to your code. Everything on a line after // is not read. You cannot use the symbol # in your comment. It is used internally to indicate an end of line.

Pascal style

The interpreter is actually pretty relaxed. You can also use code that looks a lot like pascal. You can use begin and end to delimit blocks, := for the assignment, and even add the word then in an if statement or do in a while loop. For example, the following piece of code is also valid:

```
begin
  x := 10;
  while x>0 do
    begin
      if x=5 then x:=x-5 else x:=x-1;
    end;
  end;
end;
```

Debugger

If your game is not working the way you expect, you often like to check certain variables. For this **Game Maker** contains a little debugger. To start it, press <Ctrl>-D in the main window. A form opens in which you can see various types of information: some global information, the value of all global variables (both the general ones and the ones you defined yourself) the variables of a particular object (choose the object you want; if there are multiple, you get the values of the first one), and some messages (hardly any at the moment). To see the various pieces of information, click on the checkmarks. (The more information you show, the slower the game runs.)

When the debugger is shown, you can also step through your game. To this end, pause the game and then repeatedly click the step button (or use <Ctrl>-S in the main window).