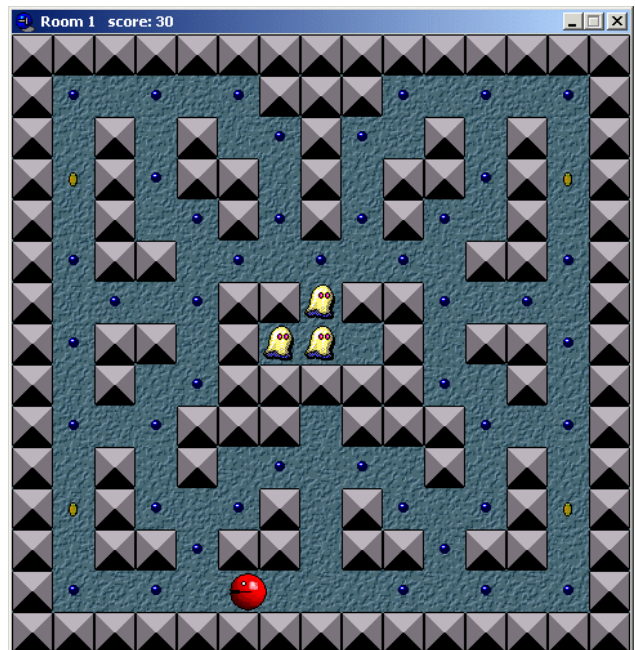


# Game Maker

Version 2.0 (September 7, 2000)

## Mark Overmars



# Table of Contents

<b><i>Part I: Using Game Maker</i></b> .....	<b>4</b>
<b><i>Introduction</i></b> .....	<b>4</b>
System requirements.....	4
<b><i>The global idea</i></b> .....	<b>4</b>
<b><i>A simple example</i></b> .....	<b>5</b>
<b><i>The main interface</i></b> .....	<b>6</b>
<b><i>Creating objects</i></b> .....	<b>6</b>
<b><i>Object image</i></b> .....	<b>7</b>
<b><i>Events</i></b> .....	<b>8</b>
<b><i>Actions</i></b> .....	<b>9</b>
Movement.....	9
Creating and destroying objects.....	10
Other actions.....	10
Conditionals.....	11
Using expressions.....	11
Using code.....	12
<b><i>Creating rooms</i></b> .....	<b>12</b>
Room settings.....	13
Background settings.....	13
View settings.....	13
<b><i>Creating sounds</i></b> .....	<b>13</b>
<b><i>Creating game information</i></b> .....	<b>14</b>
<b><i>Distributing your game</i></b> .....	<b>14</b>
<b><i>Part II: Advanced Game Design</i></b> .....	<b>16</b>
<b><i>Introduction</i></b> .....	<b>16</b>
<b><i>Creating code</i></b> .....	<b>16</b>
The code editor.....	16
Debugging the code.....	16
<b><i>The language</i></b> .....	<b>17</b>
Program.....	17
Variables.....	17
If statement.....	19
Repeat statement.....	19
While statement.....	19
Exit statement.....	20
Functions.....	20
Forall construction.....	20
Comment.....	20
Pascal style.....	20
<b><i>Game actions</i></b> .....	<b>21</b>
Moving objects around.....	21

Creating, changing, and destroying instances.....	22
Timing.....	22
Rooms and score.....	22
<b><i>User interaction.....</i></b>	<b>23</b>
<b><i>Computing things.....</i></b>	<b>25</b>
<b><i>Sounds.....</i></b>	<b>26</b>
<b><i>Game graphics.....</i></b>	<b>26</b>
Window and cursor.....	26
The background.....	27
The view.....	27
The object image.....	28
Advance drawing routines.....	28
<b><i>File IO.....</i></b>	<b>30</b>
<b><i>Part III: Tips and Tricks.....</i></b>	<b>31</b>
<b><i>General.....</i></b>	<b>31</b>
Controller.....	31
Gravity.....	31
Continues keyboard events.....	31
Joystick support.....	31
Your own score.....	31
A time limit.....	32
<b><i>Creating rooms.....</i></b>	<b>32</b>
Objects on the foreground.....	32
More precise positioning.....	32
Scrolling background.....	32
<b><i>Images and sounds.....</i></b>	<b>33</b>
Background music.....	33
Finding images and sounds.....	33
Object with multiple appearances.....	33
Your own cursor.....	33
<b><i>Clever Code.....</i></b>	<b>34</b>
Lives.....	34
Defining functions.....	34
Saving a game.....	34
<b><i>Internals.....</i></b>	<b>35</b>
Image colors and transparency.....	35
Order of events.....	35

# Part I

## Using Game Maker

### Introduction

Computer games are very popular. Playing computer games is fun, but it is actually a lot more fun to design and create computer games yourself. Unfortunately, writing computer games normally is a lot of work and requires skill in programming. This makes it something out of reach for many people. But making games can be a lot easier. *Game Maker* was designed to take away most of the burdens of writing computer games. You even don't have to know how to program. In *Game Maker* you create game characters and bring them to life with simple drag-and-drop operations. But this does not limit you. *Game Maker* allows you to make appealing games, with animated graphics, backgrounds, sounds, etc., that will be difficult if not impossible to distinguish from commercial games. All you need is your creativity. And if the standard tools and actions in *Game Maker* are not enough for you, there is a complete programming language incorporated that gives you complete control over all aspects of your game.

And what is best, *Game Maker* can be used free of charge. Games you create with *Game Maker* can be turned into full stand-alone games with which you can impress your friends and that you, if you really want to, can even sell.

This document describes all features of *Game Maker* in detail. It consists of three parts. Part I describes the basic aspects of *Game Maker*. You are strongly recommended to completely read this part. Part II describes the built-in programming language and shows how you can use it to build more advanced games. Finally, part III gives a number of tips and tricks that might help you in designing your games.

*Game Maker* comes with a collection of freeware images and sounds to get you started. These are not part of the game but were taken from public domain collections. Also there are a number of example games, in particular Pacman, Breakout and a Peg game. These games are mainly provided as examples and not as full-blown games, although they are actually quite a bit of fun to play. On the web-site

<http://www.cs.uu.nl/~markov/kids/gmaker/index.html>

some full games created with *Game Maker* are provided. Please send you own creations to [markov+games@cs.uu.nl](mailto:markov+games@cs.uu.nl) and I might add them to the site. (See below on how to do this.)

### System requirements

*Game Maker* requires a reasonable powerful computer (Pentium with 16 Mb of memory minimum; preferably Pentium 166Mhz and 24 Mb of memory or more) running Windows 95 or later. It requires at least 65000 colors (high color, 16-bits) and a screen resolution of 800x600 or more. It works best when DirectX is installed (version 5.0 or higher).

### The global idea

Games created with *Game Maker* take place in one or more rooms. (Rooms are flat, not 3D, but they can contain 3D-looking graphics.) In these rooms there are various objects. Some objects belong to the background and don't do anything, some form walls, or other static things, and others are moving around and/or act and react.

So the first thing to do is to make some objects. Below you find more information on how to do this but let me give a global description here. Objects first of all have an image such that you can see them. Objects also have a name for easy reference. You can place multiple instances of the same object in a room. So if you have e.g. three monsters you need only to define one monster object (unless you want them to have a different image or different behavior). Objects can be solid, which means that they cannot occupy the same place, or not. Also, they can be active, that is, walk around and react to each other, or passive. For example, background objects will

neither be solid, nor active. Other objects can walk over them and nothing happens. Walls are solid but not active, so other objects cannot run into them. Figures that move around in your game are active and might be solid or not, depending on their use.

Active objects can perform actions. There are different moments, called events, when actions are required. The most important ones are: creation, collision, and meeting events. When an instance of an object is created (either because it was placed in the initial room, or when it is created during the game) these actions are performed. Such actions for example put the object in motion. When an active object collides with a solid object, a collision event happens, and the object should take appropriate action (e.g. reverse direction or stop, and make a sound). When an active object meets another object, a meeting event happens. You should take action, based on the object you meet. For example, if you meet a monster you might kill yourself, and if you meet a bonus object you might add something to the score (the bonus object should in this case probably destroy itself to avoid that you keep on walking over it). Such a bonus object would typically be a non-solid active object. There are also other events: each instance of an object has an alarm clock that can generate events, and there are keyboard and mouse events such that objects can react to input from the player.

As indicated, objects can perform actions when events happen. There are many different actions possible. Objects can start or stop moving in a direction, change their speed or position, kill themselves or other objects, change into something else, create new objects, or play sound files. There is actually a complete programming language incorporated in *Game Maker* in which you can fully program the actions. But for many games you only need the standard actions and there is no need to write any line of code.

After you created the objects and specified the required actions, it is time to define the rooms. Simple game will have just one room. More complicated games can have multiple rooms and there are actions to move from one room to another. Defining rooms is easy. You specify some properties like size and color, and then you place the objects in it.

Now you are ready to run the game. Objects will come to life because of their creation actions and start reacting with each other. The user can control reactions using keyboard or mouse events.

## A simple example

Did this all sound complicated? It might at first sight, but creating a game is really easy. Let us look at a simple example. We want to make a game in which an object jumps around on the screen. The player should try to catch it by pressing the mouse on it. The game is provided under the name *Catch the Dog*. Best open it and play it to understand what I mean.

Now let us look at how it was created. Press the button **Create Objects**. You will see that there is just one object in the list: the dog. Click on it with the mouse. Suddenly a lot of information pops up. At the bottom left you see the name of the object (dog) and the image. Furthermore it is indicated that it is solid and active. At the right you see the events and after some of them some blocks that indicate actions. Only three events contain actions: the creation event, the alarm event, and the mouse event. The create event contains two actions. The first one moves the dog to a random position. The second one sets the alarm clock to 10 ticks (1 second). The alarm event does exactly the same things. The result is that every second the dog moves to a random position. Finally let's look at the mouse event. This is executed when the player manages to press the mouse on the object. There are four actions here. The first action adds one to the score. The second action plays a little sound. And the other two actions again move the dog to a random position and set the alarm clock.

Close the window and press the button **Create Rooms**. You see that there is just one boring room with the dog inside it. Just note that at the left it is indicated that the speed is 10 (so 10 ticks per second). You can change this to make the game go faster or slower.

Close the window and press the button **Create Sounds**. Here you see that there is just one sound defined for the game. This is the sound used when you manage to click on the dog.












I hope this convinces you how simple things are. You might want to play a bit with this game and change some aspects. E.g. start with two dogs in the room (select the dog from the list at the bottom of the Rooms window and click anywhere in the field). Or change the alarm clock setting in the Objects form (click with the right mouse button on the action to change the settings). You can also make the object move rather than stand still (drag the

action with the 8 arrows to the alarm event and select all arrow buttons). Finally, you might want to change the sound that the dog makes.

But maybe it is better to first read on, to understand how to use *Game Maker*.

## The main interface

When you start *Game Maker*, you are asked to select the game that you want to play or edit. (If you want to start creating a new game, click on the button labeled **Cancel**.) Now a toolbar appears at the top of the screen that might look disappointingly simple. It contains just a menu and a few buttons. But don't be fooled; a lot more awaits you. From left to right you find the following buttons:

-  **New Game** Start making a new game. After this you can press the buttons to create objects and design rooms, described below.
-  **Open Game** Use this to open an existing game. After you opened a game you can play it or change it.
-  **Save Game** Only available when you changed a game. Use this to save your game.
-  **Run Game** Runs the game. A new window appears in which the current game is being played. (You can also press <Ctrl>-R.)
-  **Pause Game** Pauses the currently running game. Press **Run Game** to continue it. (You can also press <Ctrl>-P.)
-  **Stop Game** Only available when the game is running. Stops the game. (You can also press <Ctrl>-Q or the <Esc> key.)
-  **Step** Do a single step in the game (only available when paused). (You can also press <Ctrl>-S.)
-  **Create Objects** When you press this button a large window appears in which you can create objects for your game, or edit existing objects (see below). Here you can (should) set the name of the game.
-  **Create Rooms** If you press this button you can create the rooms for your game (see below).
-  **Create Sounds** Here you can indicate the sounds you want to use in the game (see below).
-  **Create Game Info** Here you can create the game information that is shown when the player presses the <F1> key while playing the game.

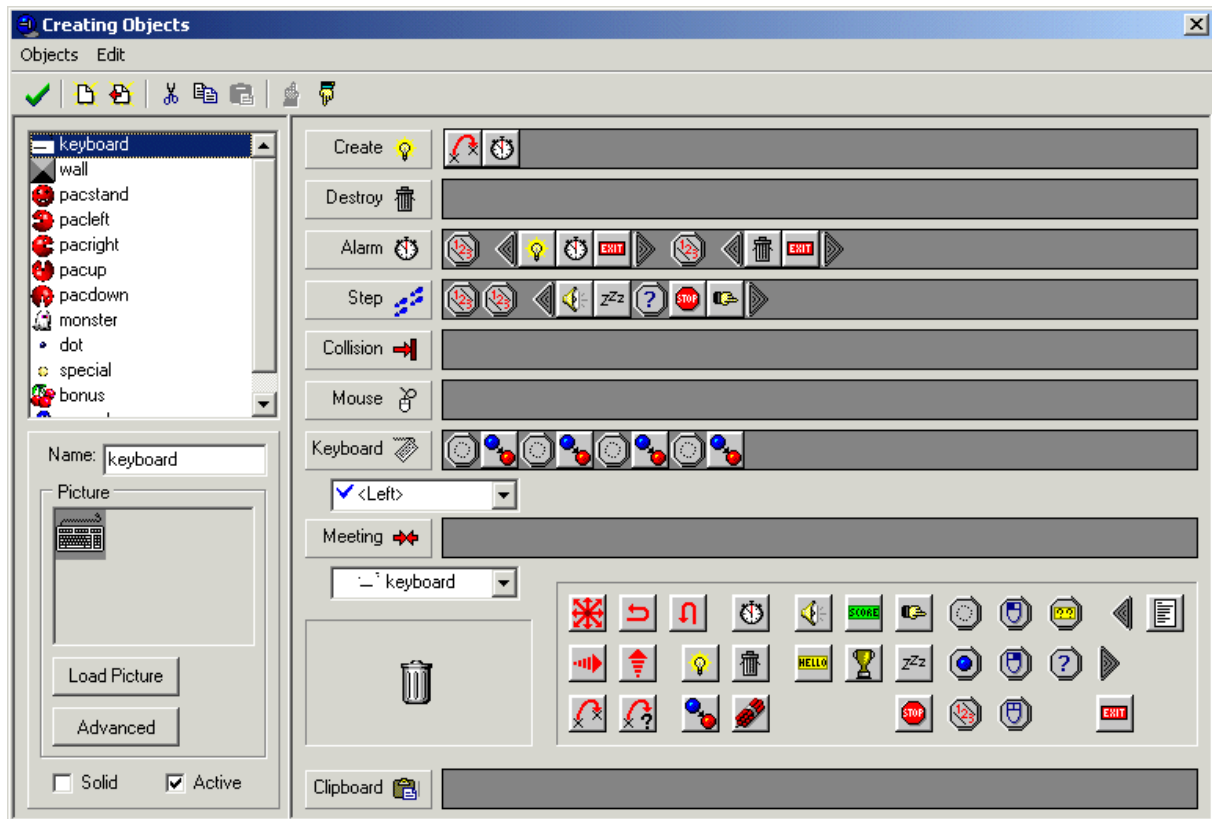
Some more commands are available in the menu. There are for example commands to rename the game, to delete the game or to save the game under a different name. For some of the other commands, see below.

So when you want to play a particular game you first press the button **Open Game**, select the game you want, and then press **Run Game**. You can press <F1> to get information about the game you play.

If you want to design a new game, press the button **New Game**. Press **Create Sounds** if you want to use any sounds, and load and name them. Next press **Create Objects** to create the objects you need. When your objects are ready, choose **Create Rooms** to design the rooms for your game. Then press **Run Game** to test your game. Press **Create Game Info** to provide information about the game you made. Finally press **Save Game** to save your game.

## Creating objects

The first step in designing a game is to create the objects that appear in the game. Typical objects you might need are walls, the figures that move, bonus items, etc. To create objects press the button with the blue ball on it. A form will open that looks as follows:



At the very top left of the form you see a list of all objects currently defined. Above it there are buttons to add a new object at the end of the list or insert an object before the currently selected object. You can also change the order of the objects in the list by pressing the buttons with the up and down pointing hands. Finally, you can cut or copy an object to the clipboard and paste it back from the clipboard. You can use this to e.g. make copies of objects (by using copy and paste). You can even copy an object from one game to another. Some words of warning are required here though. First of all, don't move an object by using cut and paste. When you cut an object, all references to it in other objects (in particular meeting events) are removed. They don't reappear when you paste the object back. Also, when you copy an object from one game to another better don't have meeting events defined or sounds in the events because they might get mixed up completely.

Once you press the **Add** or **Insert** button, or click on an object, some information about the object appears at the left bottom. First of all, there is the name. Make sure that all object have different names. Although not strictly necessary, for advanced use it is better to only use letters, digits and the underscore ' \_ ' symbol in the names. There is also a box that contains the objects image. Click on the button **Load Picture** to load a different image (see below for more information on images). There is also a button labeled **Advanced**. When you press this button you can indicate more precisely how objects should be drawn. See part II of this document for more information on advanced drawing.

You also see two boxes labeled **Solid** and **Active**. As indicated before, solid means that the object will create collisions when other objects hit it. Set it for things like walls but not for other objects. Active means that the object reacts on certain events. Once an object is **Active**, at the right a large amount of information occurs. It shows all the different events and the possible actions you can use. If you don't need any actions for the object, better make it non-active. See below for more information.

To close the object form press the button with the green checkmark.

## Object image

Objects have an image associated with them such that you can see them. An image can have any size. An image can be an icon file (\*.ico), a bitmap file (\*.bmp), a gif file (\*.gif), and even jpeg or metafiles. To pick the image for an object, click on the button **Load Picture** at the left bottom of the object form. Use the file dialog to pick the correct file. Images are considered partially transparent such that they can move over a background. The

color of the left bottom pixel of the image is the transparency color. So always make sure that this pixel is part of the background. Let me describe the possible image types in a bit more detail.

- Icon files normally have a size of 32x32 pixels. There are huge collections of them available on the web. Some of these free public domain icons are provided with this program, but they are not part of the program. Icons normally have a transparency color defined, which is used by *Game Maker*. By the way, *Game Maker* internally converts the icon file into a gif file and also stores it as a .gif file.
- You can also use a bitmap file. The left bottom pixel color is used as transparency color. Similar, you can use jpeg files and metafiles.
- But the most powerful way is to use an (animated) gif file. During the running of the game the animation is played. *Game Maker* stores all images as animated gif files. Animated gif files are available everywhere on the web. A number of them are provided with this program. You can create your own animated gif files using any of the many available gif animator programs, e.g. the free Microsoft GIF Animator at

<http://msdownload.microsoft.com/msdownload/gifanimator/gifsetup.exe>






or by using *Image Maker* which is available from the *Game Maker* web site

<http://www.cs.uu.nl/~markov/kids/gmaker/index.html>




## Events

When the game starts running, events occur. By specifying actions for some of the events, you determine what happens in the game. For example, when an instance of an object is created, a creation event occurs for this instance. You can e.g. indicate that at this moment the object should start moving to the left. When the object hits a wall, a collision event occurs. You can e.g. say that in the case of a collision the object should reverse its horizontal direction. If you now place the object in a room, with walls to the left and the right, the object will keep moving left and right between the walls.

Events can only be specified for active objects. Once you make an object active, at the right of the form all possible events occur. Each event is followed by a dark gray rectangle. In this rectangle you can drag the actions that should happen when the event occurs (see below). The following events exist. (In most games you will indicate actions for only a few of them.)

-  **Creation Event.** A creation event occurs whenever an object is created. All instances of active objects will get a creation event when the game starts. You typically use them to give the object a direction and a speed. Also when you create an instance of an object during the game or when you change an instance into a different object, a creation event happens.
-  **Destruction Event.** This event happens when an instance of an object is destroyed. Often you don't need to do anything in this case, but you might use it to do something with the score, to end the game, or to create a new object somewhere else.
-  **Alarm Event.** Each active object has an alarm clock that you can set (see the actions below). The alarm clock counts down and when it reaches 0 an alarm event occurs. You can use this to let certain things happen from time to time. For example, an object can change its direction of motion every 10 steps (in such a case the alarm event, as one of its actions, sets the alarm again). Or you can open a door for a short period of time after which you close it again.
-  **Step Event.** This event occurs every step in the game. For simple games you don't need to do anything here but for more complicated games it is one of the most crucial events. You can for example continuously change the speed or direction of motion.
-  **Collision Event.** A collision event happens when an active object bumps into a solid object. This is not allowed so the actions in this event should take care that the collision does not occur. (If the actions do not avoid the collision, the program will try a couple of times, after which the object will stop moving. If there are no actions defined, the object will not react to the collision and might go straight through the solid object.) A typical action here is to stop the motion, reverse the direction of motion, or choose a random new direction of motion. But also more complicated actions can occur, e.g., you can push the other object out of the way. (Collision events only occur when you hit a solid object. If you want to have an action happening when you hit a non-solid object, use the meeting event below. When you specify a meeting event for a solid object, a collision event won't occur. In this way you can specify special behavior for certain objects.)



-  **Meeting Event.** For each object type you can specify a different set of actions. Indicate the object in the drop-down list and then specify the actions. You can specify meeting events for both solid and non-solid objects. Meeting events play a crucial role. When your man meets a monster he might die. When he hits a piece of gold the piece of gold is destroyed and the score is raised. And when he meets a button a door can be opened (changed from a solid closed door into a non-solid open door).
-  **Mouse Event.** A mouse event occurs when the user clicks with the mouse on the object. This can be used for user interaction. For example, you can generate some objects that look like buttons, and when the user clicks on them, something can happen in the game.
-  **Keyboard Events.** For further interaction you can specify keyboard events. In the dropdown list specify the key on the keyboard and next indicate the actions that should take place when the key is pressed (when the user keeps the key pressed, the event occurs repeatedly). You can specify actions for the arrow keys, for the numeric keypad (when <NumLock> is pressed) for the normal character keys, and for the function keys. (User interaction is also possible with the joystick. See part II or III of this document.)

Sounds like a huge list of possibilities, doesn't it. But for most games you need to specify actions for only a few events. Looking at the examples provided helps a lot in understanding the possibilities.





## Actions

For each event you can specify the actions that must happen when this event occurs. Typical actions set the direction of the motion, the speed, etc. Each action is represented by an icon in the area at the bottom right of the form. (Because only active objects can have events, this is only shown when the current object is active.) When you let your mouse pointer rest on an icon, a description is given. You add actions to events by dragging them to the dark gray bars next to the names of the events (for keyboard and meeting events, make sure you first selected the right key or object in the dropdown lists below them). You can also drag actions from one event to another. If you hold the left <Ctrl> key, the action is copied. To remove an action, drag it to the trashcan. At the bottom of the form you see another dark gray box named Clipboard. You can also drag and copy actions here. They will stay on the clipboard as long as you don't stop *Game Maker*. In this way you can easily move or copy actions between different objects or between different keyboard or meeting events.




Most actions have some parameters that you have to provide. When you place an action, a form will pop up asking you for these values. If you want to change the parameters later, click on the action with your right mouse button. This form will look similar for most actions. At the top you can indicate to whom the action must apply. The default is self, which means that the action applies to the object that created the event. But you can also indicate that the action should apply to all instances of a particular object. So, for example, when your man meets a switch object you can make all doors disappear, or you can (temporarily) stop all monsters. For collision and meeting events you can also specify that the action should be applied to the other object involved. So, for example, when you hit a coin you can indicate that the other object (the coin) should disappear. Also, for many actions, you can indicate whether the change should be relative or not. Relative means that certain values, like the position or speed or timer, are increased with the amount you specify. If you uncheck relative, the values are set to the value you provide.

## Movement





A number of actions deal with the movement of the objects.

-  **Set the direction of motion.** Click on the arrow indicating the direction you want, or click on the square in the middle to make the movement stop. You can press multiple directions. In this case a random choice is made.
-  **Reverse horizontal direction.** Reverses the horizontal direction of motion. Can e.g. be used when hitting a vertical wall.
-  **Reverse vertical direction.** Reverses the vertical direction of motion. Can e.g. be used when hitting a horizontal wall.
-  **Set horizontal speed.** Sets the horizontal speed. The number you give is the number of pixels the object moves in each step. Default, the speed is 8. Note that you set the speed relative to the current







speed, unless you unmark the box labeled **Relative**. If you, e.g., want to make an object move faster all the time you can add a small value (e.g. 0.1) to the speed in every step.

-  **Set vertical speed.** Sets the vertical speed of the object.
-  **Move to position (x,y).** Moves the object to the specified position. Normally the position is relative to the current position. So e.g. (-32,0) means that the object is moved 32 pixels to the left. (0,32) means that the object is moves 32 pixels down. If you unmark the box labeled **Relative** you can specify an absolute position. (0,0) is the top left position in the room.
-  **Move to a random empty cell.** Moves the object to a random empty cell.



## Creating and destroying objects

-  **Create a new object at (x,y).** Create a new object at position (x,y) (relative to the current object or absolute).
-  **Destroy object.** Use this action to destroy yourself or other objects. The destroy event will be generated.
-  **Change into another object.** Use this action to change the instance (or all instances of a particular type) into a different object. This is largely the same as destroying yourself and creating a new object of the new type at the current spot except that no new instance is created and things like speed, timers, etc. stay the same. But for the current object the destroy event is generated and for the new one the create event. In Pacman this event is constantly used to change the pacman object and to change the monsters into scared monsters.
-  **Destroy all objects at a position.** Destroys all the objects that exist at the indicated position. For example, if you have a bomb in the game and it explodes, you can kill all the object left, right, above and below it.

## Other actions












-  **Set the alarm clock.** Here you can set the alarm clock such that after the indicated number of steps the alarm event happens. This can be used to let things happen after certain intervals. Often the alarm event sets the alarm clock again to let the thing happen again some steps later. You can also set the alarm clock of another object.
-  **Set the score.** Here you can set the score or add to the score (relative). The score will be displayed once you set it. (So if you want the score to be displayed from the beginning, set it to 0 in the create event of some instance.)
-  **Show the highscore.** This action shows the highscore table. If the current score is higher, the player can fill in his name. Best use when the player dies in the game, that is, just before stopping the game. The highscore table is saved.
-  **Play a sound.** You are asked for the name of the sound. See below how to add sounds to your games. You can also select <stop sound> to stop the sound that is currently playing.
-  **Show a message.** Displays a message box containing the message string. Game play will be interrupted until the player presses **OK**. Can be used to give certain instructions during the game. Use # in the string to indicate a new line.
-  **Go to another room.** Indicate the number of the room (relative to the current room number or absolute). This can be used to create games with multiple rooms or levels. For example, associate such

an action with the meeting event of the man with a door. To restart the current room, move relative to room 0.

-  **Sleep a while.** Specify the time in milliseconds (i.e. 1000 is one second).
-  **End the game.** Ends the game.

## Conditionals

Conditionals might be bit more complicated to understand at first. They ask a question. When the answer is yes (true), the next action is performed. Otherwise the next action is skipped. You can also perform or skip a number of actions by putting them in a block. (Note that conditional actions have a different shape to distinguish them.)

-  **If there is no collision at (x,y).** You specify a position (relative to the current position or absolute). If moving there would cause no collision, the next action or block of actions is performed. This can e.g. be use to check whether a position is collision-free before going there.
-  **If there is a particular object at (x,y).** Again you specify a position, but also an object. If that object occurs at that position (that is, the current object would meet it if moved there) the next action is performed.
-  **If the number of instances is a value.** You provide the number and the object. If the number of instances of that object is equal to the number the next action is performed. For example, if there are no coins left (number is 0) you can do something.
-  **If question to player.** You specify a question for the player. If the player answers yes, the next action is performed.
-  **If left mouse button pressed.** If the left mouse button is pressed the next action is performed.
-  **If right mouse button pressed.** If the right mouse button is pressed the next action is performed.
-  **If no mouse button pressed.** If no mouse button is pressed the next action is performed.
-  **If expression.** Here you can type in an arbitrary expression (see below). If it is true, the next action is performed.
-  **Begin block.** Use this symbol immediately after a conditional to create a block of actions.
-  **End block.** Place this at the end of the block.
-  **Exit event.** This action exits the event, that is, no further actions in the event are performed. This can be useful after a conditional.

## Using expressions

In many actions you need to provide values. Rather than just typing a number, you can also type a formula, e.g.  $32*12$ . But you can actually type much more complicated expressions. For example, if you want to double the horizontal speed, you could set it to  $2*hspeed$ . Here  $hspeed$  indicated the current horizontal speed. There are a number of other values you can use. The most important ones are:

- **x** the x-coordinate of the instance
- **y** the y-coordinate of the instance
- **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)

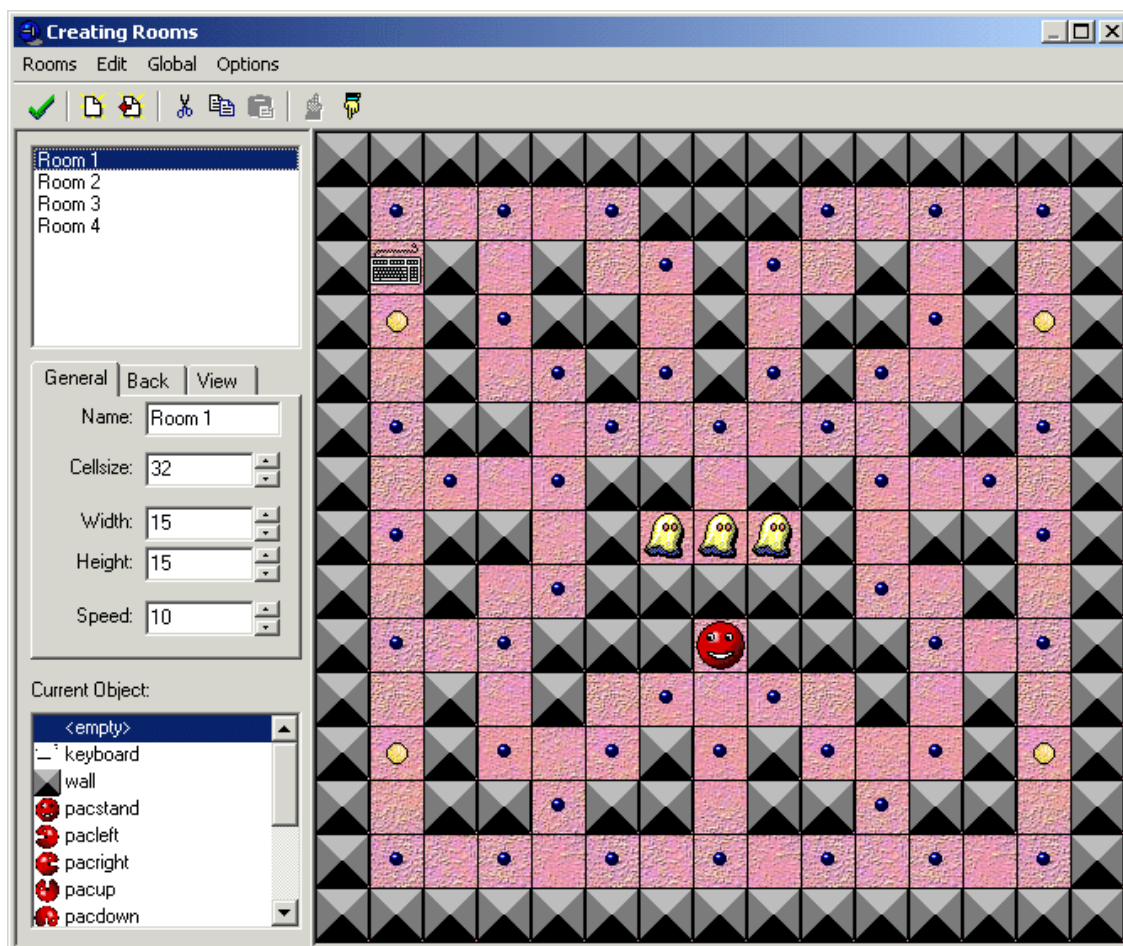
You can also refer to these values for other objects by putting the object name and a dot in front of it. So e.g. if you want a ball to move to the place where the coin is you can set the position to (coin.x,coin.y). In the case of a collision or meeting event you can refer to the x-coordinate of the other object as other.x. In conditional expressions you can use comparisons like < (smaller than), >, etc. For complete information on expressions see part II of this document.

## Using code

Even though you can create quite elaborate games using the standard actions, at some stage you might want further control. To this end *Game Maker* has a complete built-in programming language and interpreter. You can create actions that contain pieces of code. Within this code you can actually do almost everything you can do in the actions, but you can do a lot more. For a detailed description of the language, see part II of this document.

## Creating rooms

After you defined the objects you need it is time to create the room(s). Click on the **Create Rooms** button and the following form will pop up.



At the top left you find the list of rooms. When you are creating a new game, there is only one in it, labeled <new>. There is a toolbar with a number of buttons. These can be used to add rooms, insert them, cut, copy, and paste them, and move them up and down in the list. (You can copy rooms between games, but this only works if they use the same objects!) You can also give these commands from the menu. You can always use **Undo** in the **Edit** menu (or press <Ctrl>-Z) to restore the changes you made to the current room. If you really messed things up, choose **Exit discarding changes** from the **Rooms** menu. This will discard all changes you made to the collection of rooms.

At the right there is the room. You place objects in the room by choosing the object in the list at the bottom left, and then clicking with your mouse in the correct cell. With the right mouse button you can clear cells. Normally, each cell can contain only one object. You can change this using a setting in the **Options** menu. But be careful. Because the objects are drawn on top of each other you might not see them.

In the **Global** menu you find some addition commands, e.g. to clear the room completely, and to move all instances in the room left, right, up, or down.

## Room settings

Once you added a room or selected a room, at the middle left a number of settings about the room are shown that you can change.

- **Name.** The name of the room that will be shown in the caption, when running the game.
- **Cellsize.** The size of a cell in the room in pixels. Normally this is 32 but if you want to use smaller images you might want to change this (some of the provided examples use 24).
- **Width.** The horizontal number of cells. Change it using the up and down arrow buttons.
- **Height.** The vertical number of cells.
- **Speed.** The game speed, that is, the number of steps per second. (If your machine is slow this number might not really be achieved.)

## Background settings

If you click on the tab labeled **Background** you set a number of aspects of the background.

- **Color.** Click here to have a colored background. Click on the box to change the color.
- **Image.** Click here to have a background image. Click on the box to indicate the filename for the image. You can use many different types of images (icons, bitmaps, jpeg files, gif files, and metafiles). Click **Tiled** to have copies of the image placed next to each other. Click **Stretched** to have the image stretched over the background. If you click **Scrolling** you can make a scrolling background. In this case further fields appear in which you can set the horizontal and vertical scrolling speed.

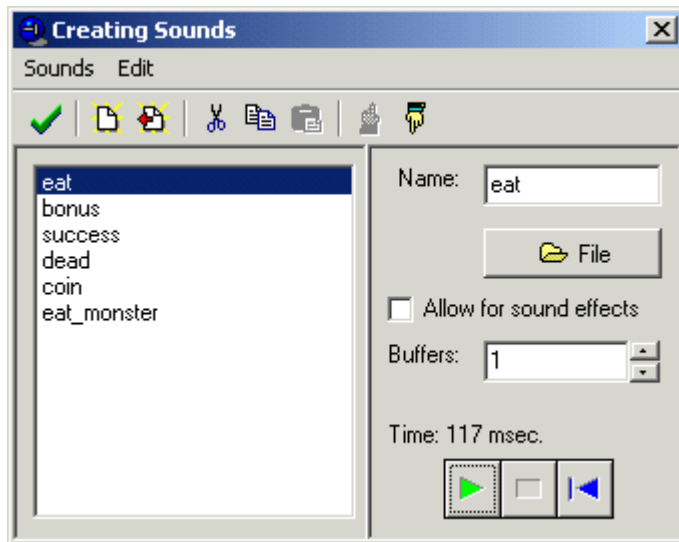
## View settings

In many games you do not see the whole playing field but only part of it. You explore the area by moving some object through the room. The part of the room you see scrolls automatically with the moving object. You can achieve this in *Game Maker* in a very simple way. First click on the **View** tab. Indicate that you want to restrict the view by clicking on the check mark. Now you can indicate the following:

- **Width.** The horizontal number of cells in the view. Change it using the up and down arrow buttons. Make sure it is no larger than the width of the room.
- **Height.** The vertical number of cells in the view.
- **Border.** The number of cells that must remain visible around the object. If this is 0 the room will start scrolling only when the object reaches the boundary. Otherwise, it starts scrolling earlier. Normally, a setting of 1 or 2 is the best.
- **Object.** The object that must always remain visible. The view scrolls when this object moves around. You can set it to none, but this is only useful if you control the view from within a piece of code (see part II).

## Creating sounds

A nice game should definitely have some sounds in it. To add sounds to your game, press the button **Create Sounds**. The following form will show:



At the left you see the list of sounds. In the toolbar and menu you find buttons to add or insert a sound, change the order of the sounds, or cut or copy a sound to the clipboard and paste it back. (Realize that if you cut a sound all sound actions that refer to it are removed.) When you click on a sound or add a sound, you can change its name and choose the sound file. You can also test the sound using the buttons. When you are done click the button labeled **Close**.

There are two types of sound files that can be used: midi files (extension .mid) or wave files (extension .wav). Midi files are typically used for background music. They will loop forever. Wave files are for short sound effects. To add a sound effect to a particular event, use the sound action. (To create background music, use a sound action in the creation event in some object.)

You will also see a line reading **Buffers**. This value can normally be kept to 1. It is useful to set **Buffers** to a higher value if the same sound must be plays multiple times simultaneously. For example, a shooting sound might overlap with itself when the player fires the gun in rapid succession. In this case you need a value higher than 1. (Only use this when strictly necessary because it uses resources.) There is also the check box labeled **Allow for sound effects**. When you check this box, from within code you can use sound effects like changing the volume or the frequency. For most games this is not necessary. See part II of this document for more information about sound effects.

## Creating game information

A good game provides the player with some information on how to play the game. This information is displayed when the player presses the <F1> key during game play. To create it, press the button **Create Game Information**. A little build-in editor is opened where you can edit the game information. You can use different fonts, different colors, and styles. A good advice is to make the information short but precise. Of course you should add your name because you created the game. All example games provided have an information file about the game and how it was created.

If you want to make a bit more fancy help, use e.g. Word. Then select the part you want and use copy and paste to move it from Word to the game information editor.

## Distributing your game

Of course you would like others to be able to play your games as well. This is very easy. Load the game you want to distribute. In the **File** menu there are three important items: **Import**, **Export**, and **Create stand-alone**. To distribute a game, click on the item **Export**. You are asked for a filename where to store the game. (The file name will end with .zip, because the games are stored as compressed zip files.) Best store it at a place where you can find it back. Now give this file to your friend. To put the game into *Game Maker*, such that you can play it, use the menu item **Import**, choose the correct zip file, and you are done. The game can now be played and is added to your list of games (so you don't have to import it again later).

Please mail your creations to [markov+games@cs.uu.nl](mailto:markov+games@cs.uu.nl) such that I can place them on the web site.

If you want to create a version of the game that does not require *Game Maker* to be installed on your computer, open the game and choose the menu item **Create stand-alone**. You are asked for the place where to create the stand-alone game. Indicate the directory and you are done. If your game was named XXX, at the indicated place a new directory XXX has been created in which you will find a program called XXX.exe (plus a directory called Games that stores the games information). Executing XXX.exe will play your game. Note that in the stand-alone version the menu and toolbar are not shown. The player can view the help file by using <F1>. If the game ends, the program ends as well.

# Part II

## Advanced Game Design

### Introduction

Using the standard commands in *Game Maker* you can make rather interesting games. But to make more complicated games you will need to do a little programming. *Game Maker* has a built-in programming language in which you can control all aspects of the game. Once you familiarize yourself with this language there is no need to use any of the other commands anymore. Moreover, using the language, you have much more precise control over the game graphics, over the way objects interact and over the way your game interacts with the player. In this way, with relatively little effort, you can make games that can hardly be distinguished from professional games.

To use the programming language, you drag the code action into one of the events. A little editor pops up in which you can type in your code. Each event for each object can have its own piece of code that is executed whenever this event occurs.

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 (real numbers or strings) and control structures (conditionals, loops). There are a large number of built-in variables and functions that relate to the game. First the basic syntax of the language is described. Next we delve into the details of how to control the game and the graphics.

One word of caution is in place. Interpreting pieces of code takes quite some computer time. So when you use large pieces of code that are executed very often (e.g. in the step event of an object of which there are many instances) it might slow down the game a bit.

### Creating code

To create code, drag the code action to an event of one of the objects. The code editor will show up in which you can type in the code. If you want to change code later, right click with the mouse on the action.

### The code editor

The code editor is an editor that is directed towards writing code for *Game Maker*. It has the standard features of an editor, like cut (<Ctrl>X), copy (<Ctrl>C) and paste (<Ctrl>V) within the editor or with other applications, and Undo (<Ctrl>Z). You can print the code, save it to a file, etc. There are though a few additional aspects. First of all, there is a button to check your code. Errors in the syntax (like missing brackets) are reported and also non-existing functions or a wrong number of parameters are reported. The same applies to references to non-existing objects or sounds. (At this stage it is impossible to know whether variables exist. This is only known at run time. So you won't get a warning if you mistyped a variable name.)

Another important aspect is that you can indicate whether the code should apply to the current instance, to the other instance in the case of a collision or meeting, or to all instances of a particular object (like with most other actions).

### Debugging the code

It is easy to make errors when writing code. *Game Maker* helps you in a number of ways to find errors. First of all, inside the code editor you can check your code. To this end press the button **Check the code**. It will detect a number of errors in your code, like wrongly typed keywords, missing brackets, or non-existing functions (or functions with the wrong number of arguments). Always check your code when you are ready with it. It helps a lot.



Some errors can only be detected when the program is running. A form will show up indicate the error and where it occurred. You can either abort execution of the game, or continue to maybe gather some more information. (The information provided is rather limited at the moment. This will be improved in future versions.)

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, choose **Debug** from the **Play** menu. 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 instances of this object, 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). You can watch the variables change.

## The language

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 you must use names that start with a letter and only consist of letters, digits and the underscore ‘\_’ symbol. Otherwise you cannot refer to them from within the code. Also be careful not to name objects self, other, global, or sound because these have special meaning in the language.

### 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, which will be discussed below.

### Variables

Like any programming language there are variables. Variables can store either real values or strings. Variables do not need to be declared. There are a large number of built-in variables. Some are general, like mousex and mousey that indicate the current mouse position, while all others are local to the object instance for which we execute the code, like x and y that indicate the current position of the instance. A variable has a name that must start with a letter and can contain only letters, numbers, and the underscore symbol ‘\_’. When you use a new variable it is local to the current instance and is not known in code for other instances (even of the same object). You can though refer to variables in other instances; see below.

### Assignments

An assignment assigns the value of an expression to a variable. An assignment has the form:

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

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

### Expressions

Expressions can be real numbers (e.g. 3.4), strings between single or double quotes (e.g. ‘hello’ or “hello”) or more complicated expressions. For expressions, 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. All operators work for real values. Comparisons also work for strings and `+` concatenates strings.

### Example

Here is an example with some useless assignments.

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

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

### Variables in other instances

As stated above, variables you create or refer to are local to the current instance (except for some predefined global variables). You 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.hspeed = self.hspeed;
  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 functions. E.g. you can use `destroy(self)` to destroy yourself.)

### Extra variables

You 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;
  }
}
```

### 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  $\leq 0$  (**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! For this you need the function `end_game()`; see below.)

## Functions

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

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

Note that for a function without arguments you still need to use the brackets. Some functions return values and can be used in expressions. Others simply execute commands. Below you find a list of all functions available.

## 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` (see below). 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 behavior. (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;

```

## Game actions

From your code you can perform all actions that are available. And in many cases you have additional control.

### Moving objects around

Obviously, an important aspect of games is the moving around of object instances. Each instance has two built-in variables *x* and *y* that indicate the position of the top-left corner of the object. Position (0,0) is the top-left corner of the room. You can change the position of the object instance by changing its *x* and *y* variables. If you want the object to make complicated motions this is the way to go. You typically put this code in the step event for the object.

If the object moves with constant speed and direction, there is an easier way to do this. Each object instance has a horizontal speed (*hspeed*) and a vertical speed (*vspeed*). Both are indicated in pixels per step. Secondly, there is a horizontal direction (*hdir*) and a vertical direction (*vdir*). After each step the program sets  $x = x + \text{hspeed} * \text{hdir}$  and  $y = y + \text{vspeed} * \text{vdir}$ . So you have set these variables only once (for example in the creating event) to give the object instance a constant motion.

To summarize, each instance has the following 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).

To help you keep your objects at the right places, the following read-only variables exist:

- **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).

When you want to move your object instances around you might want to stay away from other objects. There are a number of function for this:

- **is\_free(x,y)**: Returns whether the position (x,y) is collision-free for the instance, that is, when we place the instance at position (x,y) no collision with a solid object occurs.
- **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)**: Returns whether the instance meets object obj when placed at position (x,y).
- **nothing\_at(x,y)**: Returns whether there is no object at position (x,y). The difference with *is\_empty(x,y)* is that there the size of the current instance is taken into account.
- **object\_at(x,y,obj)**: Returns whether there is an instance of object obj at position (x,y).

Finally there is a function to move an object to a random free position. It will be aligned with the cell boundaries.

- **move\_random(obj)**: Move all instances of object obj to a random free position. Use self to move only the instance itself.

## Creating, changing, and destroying instances

In most games it is necessary to create objects, destroy them or change them into other objects (e.g. a plane changes into an explosion). There exist a number of functions that deal with the creation and destruction of object instances.

- **create(x,y,obj)**: Creates an instance of object obj at position (x,y).
- **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(obj1,obj2)**: Changes all instances of obj1 into object obj2.
- **change\_at(x,y,obj)**: Changes all instances at position (x,y) in object obj, that is, all object instances whose bounding box contains position (x,y).
- **destroy\_at(x,y)**: Destroys all instances at position (x,y).
- **number(obj)**: Returns the number of instances of object obj.

There are also a number of variables associated with objects that change aspects of the object:

- **object**: The object type of the instance. If you change this, the instance changes into the other object. There is though a big difference with the change() function. When you use the change function the destroy and creation events are executed. When you change the object variable this does not happen.
- **active**: This variable indicates whether the instance is active. Be careful when you change this: an object can make itself inactive but it can never make itself active, because no events are processed for it anymore.
- **solid**: This variable indicates whether the instance is solid. You can change it temporarily make an instance not solid.

## Timing

Good games required careful timing of things happening. Fortunately *Game Maker* does most of the timing for you. It makes sure things happen at a constant rate. This rate is defined when defining the rooms. But you can change it using the global variable gamespeed. So for example, you can slowly increase the speed of the game, making it more difficult, by adding a very small amount (like 0.001) to gamespeed in every step.

- **gamespeed**: The speed of the game in steps per second (change this to speed up or slow down the game).

Sometimes you might want to stop the game for a short while. For this, use the sleep function.

- **sleep(num)**: Sleeps numb milliseconds.

Finally, you can set the alarm clock for an instance using its alarm variable.

- **alarm**: The value of the alarm clock (in steps).

## Rooms and score

Games work in rooms. Rooms are numbered from 1 to lastroom. The current room is stored in variable room. You cannot change this variable directly. Instead you should use the function goto\_room(). To restart the same room just call goto\_room(room). Make sure rooms you go to exist. So a typical piece of code you will use is:

```
{
  if (room < lastroom)
  {
    goto_room(room+1);
  }
  else
  {
    end_game();
  }
}
```

Summarizing, the following variables and functions exist.

- **room**: The number of the current room. You cannot change this variable but you can use the function `goto_room` to change the current room.
- **lastroom**: The number of the last room. You cannot change this variable.
- **goto\_room(numb)**: This function makes the game go to the room number numb.
- **end\_game()**: Ends the game.
- **gamename**: The name of the game. You cannot change this variable.
- **roomname**: The name of the current room. This name is displayed in the window caption. You can change this to change the window caption!

Another important aspect of many games is the score. *Game Maker* keeps track of a score in a global variable `score`, which is shown in the window caption. You can change the score by simply changing the value of this variable. If you don't want to show the score in the caption, set the variable `showscore` to false. For more complicated games you better create your own score variable, and display it yourself (see the *Tips and Tricks* for more information on this).

There is also a built-in mechanism to keep track of a highscore list. It can contain up to 10 names. There are routines to add players to the list, to clear the list and to show the list. To make your highscore list look fancy, you can place a bitmap named `highscore.bmp` (or `highscore.gif`, or `highscore.jpg`) in the folder for the game. This bitmap will be used for the background of the highscore list. Also you can change font and color.

- **score**: The current score (increase the score with e.g. `score += 10`);).
- **showscore**: Whether to show the score in the caption.
- **highscore\_show(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 `highscore_show(score)`, but to e.g. just show the highscore list, use `highscore_show(-1)`. The highscore list is automatically saved with the game.
- **highscore\_clear()**: Clears the highscore list.
- **highscore\_add(str,numb)**: Adds a player with name `str` and score `numb` to the highscore list (without showing the list).
- **highscore\_setcolor(col1,col2)**: Use `col1` for the color of the first player and `col2` for the other players in the list (see below for the way to indicate colors).
- **highscore\_setfont(str)**: Use `str` as fontname in the highscore list. (Make sure it exists.)

## User interaction

There is no game without interaction with the user. The standard way of doing this in *Game Maker* is to put actions in mouse or keyboard events. But sometimes you need more control. From within a piece of code you can check whether certain keys on the keyboard are pressed and you can check for the position of the mouse and whether its buttons are pressed. Normally you check these aspects in the step event of some controller object and take action accordingly. The following global variables exist:

- **mousex**: x-coordinate of the mouse.
- **mousey**: y-coordinate of the mouse.
- **mousebutton**: Currently pressed mouse button. As value use `mb_none`, `mb_left`, `mb_middle`, or `mb_right`.
- **lastkeypressed**: The keycode of the last key pressed on the keyboard that has not yet been handled by some keyboard event. See below for keycode constants.
- **keypressed**: The keycode of the currently pressed key (0=no key pressed).

Note the special role of the variable `lastkeypressed`. It only keeps its value until it is handled by the system. Normally you don't want to check this. You can set these 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. Also, to make sure that keyboard events are constantly executed as long as the key is pressed, set

```
lastkeypressed = keypressed;
```

in the step event of some object.

When the user holds multiple keys simultaneously, the variable `keypressed` will only show the last one. But sometimes this is not good enough. In this case you can use the function `check_key`:

- **check\_key(keycode)**: Returns whether the key with the particular keycode is pressed.
- **check\_mouse\_button(numb)**: Returns whether the mouse button is pressed (use as values `mb_left`, `mb_middle`, or `mb_right`).

For example, assume you have an object that the user can control with the arrow keys and you want the object to move diagonal when two keys are pressed simultaneously. You can put the following piece of code in the step event of the object:

```
{
  if (check_key(vk_left))  x -= 4;
  if (check_key(vk_right)) x += 4;
  if (check_key(vk_up))    y -= 4;
  if (check_key(vk_down))  y += 4;
}
```

The following constants for virtual keycodes exist:

- **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\_backspace** backspace key
- **vk\_tab** tab key
- **vk\_home** home key
- **vk\_end** end key
- **vk\_delete** delete key
- **vk\_insert** insert key
- **vk\_pageup** pageup key
- **vk\_pagedown** pagedown key
- **vk\_pause** pause/break key
- **vk\_printscreen** printscreen/sysrq key
- **vk\_f1 ... vk\_f12** keycodes for the function keys F1 to F12
- **vk\_numpad0 ... vk\_numpad9** number keys on the numeric keypad
- 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')`

If you need the keycode for some other key, start the debugger, make the global variables visible and hold the key. You will see the correct value for `keypressed`.

Normally, the `<Esc>` key ends the program, the `<F1>` key displays the help file, and the `<F4>` key toggles between full screen and windowed mode. So you cannot create your own behavior for these keys. You can change the help key, quit key, and screen key, using the following variables (set them to 0 to disable them):

- **quitkey**: Keycode that indicates the quit key.
- **helpkey**: Keycode that indicates the help key.
- **screenkey**: Keycode that indicates the screen key.

There are also some high-level interaction functions. They temporarily stop the running of the game and show a dialog box with a message, question, or the request to enter a string or value:

- **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 true if the user selects yes and false otherwise.



- **get\_integer(str,def)**: Asks the user in a dialog box for a number; str is the question, def the default value. It returns the value given by the user.
- **get\_string(str,def)**: Asks the user in a dialog box for a string; str is the question, def the default string. It returns the string typed in.
- **show\_info()**: Displays the info form for the game.

Though this might not be obvious, *Game Maker* actually has joystick support. Movements of the joystick create keyboard events <NUMPAD>1 to <NUMPAD>9 as in the numeric keypad. The four buttons generate keyboard events for the letters A, B, C and D. So you can react on these. Please realize that you don't get this information with the check\_key() function because that function checks the keyboard hardware. Instead use one of the following routines:

- **check\_joystick\_direction()**: Returns the keycode corresponding to the current joystick direction (vk\_numpad1 to vk\_numpad9).
- **check\_joystick\_button(numb)**: Returns whether the joystick button is pressed (numb in the range 1-4).

## Computing things

At many places in your code you have to compute things. A large number of mathematical functions and constants are available to help you:

### Constants

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

### 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 or equal to x)
- **ceil(x)** returns the ceiling of x (smallest integer larger than or equal to 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)
- **degtorad(x)** converts degrees to radians
- **radtodeg(x)** converts radians to degrees
- **min(x,y)** returns the minimum of x and y
- **max(x,y)** returns the maximum of x and y
- **mean(x,y)** returns the average of x and y

### String functions

Note that you can add strings and compare them (case-sensitive) using the standard operators. The following related functions exist:

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

## Sounds

Sound plays a crucial role in computer games. Normally there are two different types of sounds: background music and sound effects. Background music normally consists of a long piece of midi music that is infinitely repeated. Sound effects on the other hand are short wave files. To have immediate effects, these pieces are stored in memory. So you better make sure that they are not too long.

You need to define the sounds before using them. They have a name and an associated file. Make sure that the names you use are valid variable names. There is one aspect of sounds that might be puzzling at first, the number of buffers. The system can play a wave file only once at the same time. This means that when you use the effect again before the previous sound was finished, the previous sound is stopped. This is not very appealing. So when you have a sound effect that is used multiple time simultaneously (like e.g. a gun shot) you need to store it multiple times. This number is the number of buffers. The more buffers for a sound, the more times it can be played simultaneously, but it also uses more memory. So use this with care. *Game Maker* automatically uses the first buffer available, so once you indicated the number you don't have to worry about it anymore.

There are three basic functions related to sounds, two to play a sound and another to stop a sound. All take the number of the sound as argument. Rather than using the number, you of course want to use its name. To make it clear that you give the name of a sound, rather than a variable, precede it with 'sound.'. So, for example, to play the click sound once use `sound_play(sound.click)`.

- **sound\_play(numb)**: Plays the indicates sound number once.
- **sound\_loop(numb)**: Plays the indicates sound number. Loop the sound forever.
- **sound\_stop(numb)**: Stops the indicated sound from playing. (Normally you only use this for sounds that loop.)

It is possible to use further sound effects. These only apply to wave files, not to midi files. When you want to use special sound effects, you have to indicate this in the sound form by checking the box. Note that sounds that enable effects take more resources than other sounds. So only check this box when you use the calls below. There are three types of sound effect. First of all you can change the volume. A value of 0 means no sound at all. A value of 1 is the volume of the original sound. (You cannot indicate a volume larger than the original volume.) Secondly, you can change the pan, that is, the direction from which the sound comes. A value of 0 is completely at the left. A value of 1 indicates completely at the right. 0.5 is the default value that is in the middle. You can use panning to e.g. hear that an object moves from left to right. Finally you can change the frequency of sound. This can be used to e.g. change the speed of an engine. A value of 0 is the lowest frequency; a value of 1 is the highest frequency.

- **sound\_volume(numb,value)**: Changes the volume for the indicates sound number (0 = low, 1 = high).
- **sound\_pan(numb,value)**: Changes the pan for the indicates sound number (0 = left, 1 = right).
- **sound\_frequency(numb,value)**: Changes the frequency for the indicates sound number (0 = low, 1 = high).

## Game graphics

An important aspect of a computer game is the graphics. Standard, each object has an image (or series of images) associated with it. This image is drawn in each step. Also there is a background (either a single color or an image) that can even scroll with fixed speed. But if you want to make a more interesting looking game you might want much more control. E.g. you might want to draw text and shapes, you might want to control which image is shown for an object, and you might want to control the scrolling of the background. Using a piece of code, all of this is possible.

## Window and cursor

Default the game runs inside a centered window. The player can change this to full screen by pressing the <F4> key. You can also do this from within the program using the following function:

- **fullscreen(full)**: If full = true the game runs in full screen mode, otherwise it runs in windowed mode.

Note that in full screen mode the color of the region outside the playing area is the room color.

Default each game runs with a visible cursor. For lots of games you don't want this. To remove the cursor, use the function:

- **show\_cursor(show)**: If show = false the cursor is made invisible inside the playing area, otherwise it is made visible.

By the way, note that it is very easy to make your own cursor object. See the *Tips and Tricks* part on how to do this.

Sometimes you don't want to wait for the program to redraw the window. For example, when you want to do some animation from within a piece of code. For this there is the following function:

- **redraw()**: Redraws the field.

(Note that the sleep function automatically does a redraw to make sure that the screen shows the most recent state of the instances.)

## The background

The background of a room either has a color or an image. When you indicate the background image for the room it is aligned with the top left corner of the room. Sometimes you might want to change this. There are two global variables, `back_x` and `back_y`, that give the offset of the background image. (Note that the background image will be tiled so if you change the offset, still the whole room will be covered.) You can also indicate that the background should be scrolling with a fixed speed. For many scrolling games though you want to control the scrolling speed from within the game. For this there are the variables `back_hspeed` and `back_vspeed`. In summary, there are the following variables that control the background:

- **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.

## The view

As you probably know you can define a restricted view when designing rooms. The result is that only part of the room is shown and that the indicated object always stays visible. You can control the view partially from within code. You cannot change the size of the view but you can change the position of the view (which is in particular useful when you indicated no object to be visible), you can change the size of the border around the visible object, and you can indicate which object must remain visible in the view. The latter is very important when the visible object changes during the game. For example, you might change the main character object based on its current status. Unfortunately, this does mean that it is no longer the object that must remain visible. This can be remedied by one line of code in the creation event of all the possible main objects:

```
{
  view_object = self.object;
}
```

The following variables exist that influence the view:

- **view\_restrict**: Whether the view is restricted (cannot be changed).
- **view\_x**: Horizontal offset of the view.
- **view\_y**: Vertical offset of the view.
- **view\_w**: Width of the view in pixels (cannot be changed).
- **view\_h**: Height of the view (cannot be changed).
- **view\_border**: Size of the minimum border around the visible object.
- **view\_object**: Object that must remain visible.

## The object image

Each object has an image associated with it. This is either a single image or it consists of multiple images in an animated GIF file. For each instance of the object the program draws the corresponding image on the screen, with its top-left corner at the position (x,y) of the object. When there are multiple images, it cycles through the images to get an animation effect. There are a number of variables that effect the way the image is drawn. These can be used to change the effects.

- **visible**: If visible is true (1) the image is drawn, otherwise it is not drawn. Invisible object still are active and create collision and meeting events; you only don't see them. Setting the visibility to false is useful for e.g. controller objects (make them non-solid to avoid collision events) or hidden switches.
- **image\_width**: Indicates the width of the image. This value cannot be changed but you might want to use it.
- **image\_height**: Indicates the height of the image. This value cannot be changed but you might want to use it.
- **image\_scale**: A scale factor to make larger or smaller images. A value of 1 indicates the normal size. Changing the scale also changes the values for the image width and height and influences collision events as you might expect. (It does not change the values of the bounding box variables!) Realize that scaled images (both smaller and larger) take more time to draw. Changing the scale can be used to get a 3-D effect.
- **image\_number**: The number of subimages for the object (cannot be changed).
- **image\_index**: When the image has multiple subimages the program cycles through them. This can be avoided by setting this variable to the index of the subimage you want to see (first subimage has index 0). Give it a value -1 to cycle through the subimages. This is useful when an object has multiple appearances.
- **image\_speed**: The speed with which we cycle through the subimages. A value of 1 indicates that each step we get the next image. Smaller values will switch subimages slower, larger values will actually skip subimages to make the motion faster.
- **image\_depth**: Normally images are drawn in the order in which the instances are created. You can change this by setting the image depth. The default value is 0. The higher the value the further the instance is away. (You can also use negative values.) Instances with higher depth will lie behind instances with a lower depth. Setting the depth will guarantee that the instances are drawn in the order you want (e.g. the plane in front of the cloud). Background instances should have a high (positive) depth, and foreground instances should have a low (negative) depth.

## Advance drawing routines

It is possible to let objects look rather different from their image. There is a whole collection of functions available to draw different shapes. Also there are functions to draw text. To use these, you press the button labeled **Advanced** in the object form next to object image. Here you indicate a piece of code that should be executed when the object must be drawn, instead of drawing the object image. (Note that a checkmark is shown on the button if drawing code is defined.) You can put any code her, but in particular you can use the following drawing functions. Note that these functions don't make any sense anywhere else in code.

- **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. It can also be used to put different images for the same object in one animated image.
- **draw\_line(x1,y1,x2,y2)**: Draws a line from (x1,y1) to (x2,y2).
- **draw\_circle(xc,yc,r)**: Draws a circle with (xc,yc) as center and r as radius.
- **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\_button(x1,y1,x2,y2,down)**: Draws a button-like shape; down indicates whether the button is down (true) or up (false).
- **draw\_triangle(x1,y1,x2,y2,x3,y3)**: Draws a triangle.
- **draw\_text(x,y,str)**: Draws the string at the indicated place

For example, if you want to display some value on the screen, e.g. the number of lives left, you can proceed as follows. Use a global variable `lives` that indicates the number of lives. Now make an object with the following piece of code for advanced drawing:

```
{
    draw_text(x,y, 'Lives: ' + string(global.lives));
}
```

Place this object at the correct place in each room and you are done. (You best still give the object an image to be able to place it in the room, but this is not strictly necessary.) If you want to make it nicer, you can display the number of lives by a number of images. Give the life object a nice little image and use the following code:

```
{
    draw_text(x,y, 'Lives:');
    xx = string_width('Lives:') + 10 + 24*global.lives;
    while (xx >= 0)
    {
        draw_image(xx,y,self);
        xx -= 24;
    }
}
```

You might have to change the number 24, based on the width of the image. Note that the function `string_width()` gives the width of the string argument. Similar, `string_height()` will give the height of the string argument.

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 object variables:

- **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.

You can change a number of settings, like the color of the lines (pen), region (brush) and font, and many other font properties. The effect of these functions is global! So if you change it in the drawing routine for one object it also applies to other objects being drawn later. You can also use these functions in other event. For example, if they don't change, you can set them once at the start of the game (which is a lot more efficient).

- **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** (default)
  - **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 colors can be made using the routine `make_color(red, green, blue)`, where red, green and blue must be values between 0 and 255.

- **set\_pen\_color(col)**: Sets the color of the pen (used for outlines, etc.).
- **set\_brush\_style(style)**: Sets the fill style. The following styles are available:
  - **bs\_hollow**
  - **bs\_solid** (default)
  - **bs\_bdiagonal**
  - **bs\_fdiagonal**
  - **bs\_cross**
  - **bs\_diagcross**
  - **bs\_horizontal**
  - **bs\_vertical**
- **set\_pen\_size(size)**: Set the size (width) of the pen.
- **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\_name(name)**: Sets the name of the font, e.g. `set_font_name('Times New Roman')`.
- **set\_font\_angle(angle)**: Sets the angle with which the font is drawn in radians; for example, for vertical text use `set_font_angle(pi/2)`.
- **set\_font\_style(style)**: Sets the style of the font. The following styles are available:
  - **fs\_normal** (default)
  - **fs\_bold**
  - **fs\_italic**
  - **fs\_bolditalic**
- **set\_font\_align(align)**: Sets the alignment of the font with respect to the indicated position. The following alignments exist:
  - **fa\_left** (default)
  - **fa\_center**
  - **fa\_right**

## File IO

Your game might want to write some data in a file and read it back some other time. This is in particular useful to let a player save the game at some point and continue later. The following routines exist:

- **file\_exists(fname)** returns whether the file exists (true) or not (false).
- **file\_open\_read(fname)** opens the indicated file for reading.
- **file\_open\_write(fname)** opens the indicated file for writing.
- **file\_close()** closes the current file (don't forget to call this!).
- **file\_write(x)** writes the value x to the currently opened .
- **file\_read()** returns the next value in the file.

Note that you can only write to files that are stored in the directory in which the game is stored. So you are not allowed to include a path in the file name. If you want to create your own file for reading by the game, the file format is as follows: each line consists of one value. It starts with a number indicating the type (0=real, 1=string) followed by a space, followed by the value or the string.

# Part III

## Tips and Tricks

The possibilities of *Game Maker* are much larger than you might think at first sight. In this part I will give some ideas that might help you in making your games nicer.

### General

#### Controller

It is often useful to add a controller object to your game. This object can control certain general things, like playing background music, initializing certain variables, checking whether the game is over, creating random monsters, etc. The first thing such an object should do in its creation event is to move itself to a position outside the visible room, e.g. to position (-100,-100). I always give the controller a computer image. (Don't forget to place one in each room.)

#### Gravity

Assume you want to add gravity to your game. So objects should fall down. The speed with which they fall should increase every step. First give the objects a downward direction but, at the start, set the vertical speed to 0. Now we make a step event that increases the vertical speed with, say, 0.25 in each step. When the object hits the floor it should either set the vertical speed to 0 or bounce, by setting the vertical speed to the value  $-v_{speed}$  (don't change the vertical direction; this will cause problems cause then you suddenly have to decrease the vertical speed rather than increase it). Now whenever there is no floor below an object, it will start falling down with increasing speed. (See the Falling Balls demo for an example.) This works very well for platform games.

#### Continues keyboard events

Keyboard events are generated when a key is pressed (and when it is repeated by the system). It is not a continuous event as long as the key is being pressed. In some games you want this continuous behavior. E.g. you want an object to move as long as the key is pressed. There is a very simple way to achieve this behavior. In the step event of some existing object (e.g. your controller object if you have one) place a code action with only the line

```
lastkeypressed = keypressed;
```

keypressed is the currently pressed key. lastkeypressed was the key pressed last, which is used to generate keyboard events and which is then reset to 0. By the above line of code it is set back to the currently pressed key such that a new keyboard event will be generated.

#### Joystick support

Though this might not be obvious, the program actually has joystick support. Movements of the joystick create keyboard events <NUMPAD>1 to <NUMPAD>9 as in the numeric keypad. The four buttons generate keyboard events for the letters A, B, C and D. Let some object in your game react to these events and you are done. You can actually use the joystick in the provided Pacman game.

#### Your own score

Scores are normally displayed in the window caption. If you want to make the layout of your game a bit more fancy, you might prefer to put the score somewhere in the playing area itself. This can be done as follows. First of all, you should not use the score system of *Game Maker*. Instead, use your own global variable global.myscore. Add and subtract score using this variable. Now make an object that is going to display the

score. Give it an icon to be able to place it when designing rooms, but this is not important cause we are not going to draw it. Instead, use the advanced drawing feature for the object and as code use something like:

```
{
    draw_text(x,y,'score: ' + string(global.myscore));
}
```

That is all. You might want to choose a more fancy font, size, style and color for it. Part II for the functions for this.

## A time limit

In some games you might want to give the player a limited amount of time to finish a job (e.g. to catch all the monsters). This can easily be done as follows. Create an object called e.g. timeobject. Give it some image to see it. In its creation event move it out of sight (e.g. move it to position (-100,0)) and set the alarm clock to the time limit you want. In the alarm event end the game (probably with a message, a sound and/or showing the highscore table when you give points for catching monsters).

You might also want to show the time left. This is more difficult and requires some programming. Again we create an object as above. In the creation event use a piece of code that says:

```
{
    timeleft = 1000;
}
```

where you replace 1000 with the number of steps of time you allow. In the step event you put the following code:

```
{
    timeleft -= 1;
    if (timeleft == 0)
    {
        // play some sound, etc.
        end_game();
    }
}
```

Finally, you need to use advanced drawing. Here you put the following code:

```
{
    draw_text(x,y,'Time left: ' + string(timeleft));
}
```

You can also make it nicer. For example draw a line with a length depending on the amount of time left.

## Creating rooms

### Objects on the foreground

In a number of cases, you might want objects to move over other objects. Objects are drawn in the order in which they are added to the room. So make sure that you add the objects that should go on top last. (You can also change the variable `image_depth` for the objects. See Part II of this documentation for details.) But what if you need two objects on top of each other at the start? There is a very simple way to solve this problem. In the **Options** menu in the room form, indicate that you allow objects on top of each other. Not place them in the correct order in the cell. (Make sure you switch of the option immediately once you are done because you very easily make mistakes with this.)

### More precise positioning

If you want to position your objects more precisely in the room, set the cell size to half the actual object size. Now you can place objects at many more places. (You can also make objects partially overlap this way.)

### Scrolling background

Scrolling backgrounds are a nice way of adding a feeling of motion to your games. You can use them to e.g. have a plane fly forward while avoiding objects (and shooting other planes), to have a car drive over a road, etc. Make sure some other objects move with the same speed as the background, to increase the illusion. For smooth scrolling, make sure the background picture can nicely be tiled. You can also use a big background picture, e.g. a long road.



You can set the moving speed when creating the room, but you can also control the speed during the game using pieces of code. This makes it possible to e.g. leave an object in the middle of the room but have the world move when you press an arrow key.

## Images and sounds

### Background music

To create background music for your game, find a nice (and long) midi file. Now add this file to your list of sounds. Create a special object called music that, in its creation event, plays the sound and destroys itself. Place it in the first room, and you are done. The midi file will keep repeating itself until you stop the game.

### Finding images and sounds

If you cannot find the images, backgrounds, or sounds you like in the provided collections, the best way is to search the web. On the web huge collections of free images, sounds, backgrounds, animations, etc. are available. To get you started, here are some of the sites that store such material:

- <http://www.coolarchive.com/index.cfm>
- <http://www.arcadia-animations.com/frameindex.htm>
- [http://www.developer.com/downloads/d\\_images.html](http://www.developer.com/downloads/d_images.html)
- <http://www.clipart.com/>
- <http://www.kidsdomain.com/icon/index.html>

These sites again have many further links.

### Object with multiple appearances

In some games you have objects that should have a different appearance (image) in certain situations, but should have the same behavior. For example, in Pacman, the image looks different depending on the direction of motion. To this end you can create multiple copies of the same object with a different image and for the rest the same behavior, but this is a lot of work. Using advanced drawing you can solve this problem in a much easier way. You create separate objects with the different images but without any behavior. There is just one active object. In the advanced drawing code you draw the correct image, depending on the situation. For example, for the Pacman game the advance drawing code could look as follows:

```
{
    if (hdir < 0) draw_image(x,y,pacleft)
    else if (hdir > 0) draw_image(x,y,pacright)
    else if (vdir < 0) draw_image(x,y,pacup)
    else if (vdir > 0) draw_image(x,y,pacdown)
    else draw_image(x,y,self);
}
```

If the image is not animated, you can also use sub images for this and choose the right sub image.

### Your own cursor

Wouldn't it be nice to replace the dull cursor by some nice (animate) creation of your own? This is very easy to achieve. Create an object with the right image. Make is non-solid but active. In the create event put the following code:

```
{
    show_cursor(false);
    image_depth = 1000;
}
```

The first line makes the normal cursor invisible. The second line makes sure that the new cursor lies on top of everything else. In the step event put the following piece of code:

```
{
    x = mousex-16;
    y = mousey-16;
}
```

This puts your mouse image at the correct place (the -16 works for a 32x32 image with the important spot in the center. If you have a different sized image or want the hotspot of the cursor to be somewhere else, change these values).

Finally, make sure every room has one of your cursor objects in it.

## Clever Code

### Lives

Many games have multiple lives. To use this in your games you need to write a little bit of code. First of all, you need a global variable called e.g. lives. The controller object should set this in the first room of the game using in its creation event a piece of code like:

```
{
  if (room==1) global.lives = 3;
}
```

To show the number of lives to the player, you need an object, e.g. called life, whose image shows the lives. Give it the following advanced drawing code:

```
{
  draw_text(x,y,'Lives:');
  xx = string_width('Lives:') + 10 + 24*global.lives;
  while (xx >= 0)
  {
    draw_image(xx,y,self);
    xx -= 24;
  }
}
```

Now when the person dies you check whether global.lives is 0. If so, the game ends. Otherwise, decrease global.lives.

### Defining functions

The built-in programming language does not allow you to define functions. But there is a trick to do it. Make an active, non-solid object called e.g. function\_xxx. In its creation event put the piece of code that you want to use as a function. Also, put in it an action to kill itself. Now to call the function in another piece of code, use

```
create(-100,-100,function_xxx);
```

That will do the trick. When creating the object, the creation event is executed which contains the code that you want to be executed. Because the creation events destroys the object again it is immediately gone. You can pass parameters through global variables. (You can even create a recursive function this way but make sure the recursion depth is small.)

### Saving a game

You can use the file IO to save a game for later use. Let us simply record the last room the player was in, such that the next time, the player can start at this room again. To this end, create a controller object in each room that executes the following piece of code in its creation event:

```
{
  if (room==1 && file_exists('savegame'))
  {
    file_open_read('savegame');
    theroom = file_read();
    file_close();
    if (theroom>1) goto_room(theroom);
  }
  else
  {
    file_open_write('savegame');
    file_write(room);
    file_close();
  }
}
```

```
}  
}
```

You might want to precede this with a question to the user whether he or she wants to continue a saved game.

## Internals

### Image colors and transparency

As described in the documentation, object images are stored as gif images. This means that they will be reduced to 256 colors. If you load a bitmap image with more colors, this means that colors can change. Images are always considered to be partially transparent. For the transparency color the left bottom pixel of the image is used. (If you want an image that fully covers the area (e.g. a brick) simply make it one pixel higher and add there a line with a different color. Because the bounding box will be based on the non-transparent area this works fine.)

*Game Maker* must choose a color that it uses for the background when you it saves the image. For this a dark green color is used (very ugly). This might cause a problem when the image itself uses an almost similar dark green color. In this case holes might appear in the image or, worse, in some cases, the background stops being transparent. The solution here is to choose a slightly different color in your image.

### Order of events

In some cases it is important to know the order in which the events in the game are being processed. This order is as follows:

1. Handle the alarm events.
2. Handle the keyboard and mouse events.
3. Handle the step events.
4. Set each active object in its new position, based on speed and direction of motion.
5. Handle the collision events until no collisions occur anymore (when a collision occurs, the object is put back into its previous position to let the event change the motion after which it is moved again). If there is still a collision after 16 tries the program gives up and leaves the object at its previous position.
6. Handle the meeting events.