

# Designing Games with Game Maker

version 4.1 (January 12, 2002)

Mark Overmars

## Table of Contents

<b>Chapter 1</b>	<b>So you want to create your own computer games.....</b>	<b>5</b>
<b>Chapter 2</b>	<b>Installation.....</b>	<b>6</b>
<b>Chapter 3</b>	<b>The global idea .....</b>	<b>7</b>
<b>Chapter 4</b>	<b>Let us look at an example .....</b>	<b>9</b>
<b>Chapter 5</b>	<b>The global user interface .....</b>	<b>11</b>
5.1	File menu.....	11
5.2	Edit menu .....	12
5.3	Add menu.....	13
5.4	Window menu .....	13
5.5	Help menu.....	13
5.6	The resource explorer.....	14
<b>Chapter 6</b>	<b>Defining sprites.....</b>	<b>15</b>
<b>Chapter 7</b>	<b>Sounds and music .....</b>	<b>17</b>
<b>Chapter 8</b>	<b>Backgrounds .....</b>	<b>18</b>
<b>Chapter 9</b>	<b>Defining objects .....</b>	<b>19</b>
<b>Chapter 10</b>	<b>Events .....</b>	<b>21</b>
<b>Chapter 11</b>	<b>Actions.....</b>	<b>25</b>
11.1	Movement actions.....	25
11.2	Object related actions.....	27
11.3	Miscellaneous actions .....	28
11.4	Questions .....	31
11.5	Drawing actions .....	33
11.6	Code related actions .....	34
11.7	Using expressions and variables .....	35
<b>Chapter 12</b>	<b>Creating rooms .....</b>	<b>37</b>
<b>Chapter 13</b>	<b>More about sprites .....</b>	<b>40</b>
13.1	Editing your sprites .....	40
13.2	Editing individual sub-images .....	45
13.3	Advanced sprite settings .....	46
<b>Chapter 14</b>	<b>More about sounds and music .....</b>	<b>48</b>
<b>Chapter 15</b>	<b>More about backgrounds .....</b>	<b>49</b>
15.1	Creating tiled backgrounds .....	49
<b>Chapter 16</b>	<b>More about objects.....</b>	<b>52</b>
16.1	Depth.....	52
16.2	Persistent objects.....	52
16.3	Parents .....	52
16.4	Masks .....	53
<b>Chapter 17</b>	<b>More about rooms .....</b>	<b>54</b>
<b>Chapter 18</b>	<b>Scripts.....</b>	<b>56</b>
<b>Chapter 19</b>	<b>Game information.....</b>	<b>58</b>

<b>Chapter 20</b>	<b>Game options .....</b>	<b>59</b>
20.1	Graphics options .....	59
20.2	Key options .....	60
20.3	Interaction options.....	60
20.4	Score options.....	60
20.5	Loading options .....	61
<b>Chapter 21</b>	<b>Speed considerations .....</b>	<b>62</b>
<b>Chapter 22</b>	<b>Distributing your game .....</b>	<b>63</b>
<b>Chapter 23</b>	<b>The Game Maker Language (GML) .....</b>	<b>64</b>
23.1	A program .....	64
23.2	Variables .....	64
23.3	Assignments.....	65
23.4	Expressions .....	65
23.5	Extra variables.....	65
23.6	Addressing variables in other instances .....	66
23.7	Arrays.....	67
23.8	If statement.....	68
23.9	Repeat statement .....	68
23.10	While statement.....	68
23.11	For statement.....	69
23.12	Exit statement.....	69
23.13	Functions .....	70
23.14	Scripts .....	70
23.15	With constructions .....	70
23.16	Comment .....	72
23.17	Pascal style .....	72
23.18	Functions and variables in GML .....	72
<b>Chapter 24</b>	<b>Computing things .....</b>	<b>73</b>
24.1	Constants .....	73
24.2	Real-values functions .....	73
24.3	String handling functions.....	74
<b>Chapter 25</b>	<b>GML: Game play .....</b>	<b>75</b>
25.1	Moving around.....	75
25.2	Instances.....	76
25.3	Timing .....	78
25.4	Rooms and score .....	79
25.5	Generating events.....	80
25.6	Miscellaneous functions.....	82
<b>Chapter 26</b>	<b>GML: User interaction .....</b>	<b>83</b>
26.1	Joystick support.....	85
<b>Chapter 27</b>	<b>GML: Game graphics.....</b>	<b>86</b>
27.1	Window and cursor .....	86
27.2	Sprites and images .....	86
27.3	Backgrounds.....	89

27.4	Drawing functions.....	91
27.5	Views .....	93
27.6	Transitions.....	94
27.7	Repainting the screen.....	95
<b>Chapter 28</b>	<b>GML: Sound and music .....</b>	<b>96</b>
<b>Chapter 29</b>	<b>GML: Splash screens, highscores, and other pop-ups.....</b>	<b>99</b>
<b>Chapter 30</b>	<b>GML: Files, registry, and executing programs .....</b>	<b>101</b>
<b>Chapter 31</b>	<b>GML: Multiplayer games.....</b>	<b>104</b>
31.1	Setting up a connection.....	104
31.2	Creating and joining sessions .....	105
31.3	Players .....	106
31.4	Shared data .....	106
31.5	Messages .....	107
<b>Chapter 32</b>	<b>GML: Using DLL's .....</b>	<b>109</b>

## Chapter 1 So you want to create your own computer games

Playing computer games is fun. But it is actually more fun to design your own computer games and let other people play them. Unfortunately, creating computer games is not easy. Commercial computer games as you buy nowadays typically take one to three years of development with teams of anywhere between 10 and 50 people. Budgets easily reach in the millions of dollars. And all these people are highly experienced: programmers, art designers, sound technicians, etc.

So does this mean that it is impossible to create your own computer games? Fortunately not. Of course you should not expect that you could create your own Quake or Age of Empires within a few weeks. But that is also not necessary. A bit simpler games, like Tetris, Pacman, Space Invaders, etc. are also fun to play and a lot easier to create. Unfortunately they still require good programming skills to handle the graphics, sounds, user interaction, etc.

But here come *Game Maker*. *Game Maker* has been written to make it a lot easier to create such games. There is no need to program. An intuitive and easy to use drag-and-drop interface allows you to create your own games very quickly. You can import and create images, sprites (animated images) and sounds and use them. You easily define the objects in your game and indicate their behavior. And you can define appealing rooms with scrolling backgrounds in which the game take place. And if you want full control there is actually a simple to use programming language built into *Game Maker* that gives you full control over what is happening in your game.

*Game Maker* focuses on two-dimensional games. So no 3-D worlds like Quake. But don't let this put you down. Many great games, like Age of Empires, the Command & Conquer series, and Diablo use two-dimensional sprite technology, even though they look very 3-dimensional. And designing two-dimensional games is a lot easier and faster.

Probably the best part is that *Game Maker* can be used free of charge. And there are no restrictions on the games you create with it. No nag screen, and you can even sell them if you like. See the enclosed license agreement for more details.

This book will tell you all you need to know about *Game Maker* and how you can create your own games with it. Please realize that, even with a program like *Game Maker*, designing computer games is not completely trivial. There are too many aspects that are important: game play, graphics, sounds, user interaction, etc. Start with easy examples and you will realize that creating games is great fun. Also check the web site

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

and the forum there for lots of examples, ideas, and help. And soon you will become a master game maker yourself. Enjoy.

## Chapter 2 Installation

You probably already did this but if not, here is how to install *Game Maker*. Simply run the program `gmaker.exe`. Follow the on-screen instructions. You can install the program anywhere you like but you best follow the default suggestions given. Once installation is completed, in the Start menu you will find a new program group where you can start *Game Maker* and read the documentation. Besides the *Game Maker* program also the documentation is installed, together with the help file.

Within the installation folder (default `C:\Program Files\Game_Maker4\`) there will be a number of other folders:

- `examples`: contains a number of example games, for you to check and/or change.
- `sprites`: contains a collection of freeware (animated) sprites that can be used in your games. The following people donated sprites:
  - Primalmania, see <http://primalmania.fateback.com/>.
  - Morphosis, see <http://www.geocities.com/morphosisgames/>.
  - Billy McQuown.
- `backgrounds`: contains a number of freeware background images that you can use in your games.
- `sounds`: contains a number of freeware sound effects that you can use in your games.
- `tiles`: contains a number of tile sets that can be used when creating backgrounds (see Chapter 15). The following tiles sets were created by others:
  - `platform1_16x16.bmp` and `platform2_16x16.bmp` are tile sets for platform games. They are copyright Ari Feldman. See the website <http://www.arifeldman.com/free/spritelib.html> for more information.
  - `rpg1_24x24.bmp` and `rpg2_24x24.bmp` are useful for strategy games. They are reduced version of the sets created by Hermann Hillmann as part of the charpack. See the charpack website <http://www.vbexplorer.com/charpack1.asp> for more information.
  - `km_16x16.bmp`, `pk_16x16.bmp` and `ps_16x16.bmp` were created by e.m.b.u. See <http://www.embu.cjb.net> for conditions of use and other tile sets.
- `scripts`: contains some useful scripts (see Chapter 18).

The sprites, backgrounds, sounds, and tiles are not directly part of *Game Maker* but were taken from freeware collections.

*Game Maker* requires a modern Pentium PC running Windows 95, 98, NT, 2000, Me, or later. It requires a screen resolution of at least 800x600 and 65000 (16-bit) colors. It requires DirectX to be installed on your computer. When designing and testing games, memory requirements are pretty high (at least 32 MB and preferably more). When just running games, the memory requirements are a lot less severe and depend a lot on the type of game.

## Chapter 3 The global idea

Before delving into the possibilities of *Game Maker* it is good to first get a feeling for the global idea behind the program. 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 you place *objects*, which you can define in the program. Typical objects are the walls, moving balls, the main character, monsters, etc. Some objects, like walls, just sit there and don't do anything. Other objects, like the main character, will move around and react to input from the player (keyboard, mouse, joystick) and to each other. For example, when the main character meets a monster he might die. Objects are the most important ingredients of games made with *Game Maker*, so let us talk a bit more about them.

First of all, most objects need some image to make them visible on the screen. Such images are called *sprites*. A sprite is often not a single image but a set of images that are shown one after the other to create an animation. In this way it looks like the character walks, a ball rotates, a spaceship explodes, etc. During the game the sprite for a particular object can change. (So the character can look different when it walks to the left or to the right.) You can create your own sprite in *Game Maker* or load them from files (e.g. animated GIF's).

Certain things will happen to objects. Such happenings are called *events*. Objects can take certain *actions* when events happen. There are a large number of different events that can take place and a large number of different actions that you can let your objects take. For example, there is a *creation event* when the object gets created. (To be more precise, when an instance of an object gets created; there can be multiple instances of the same object.) For example, when a ball object gets created you can give it some motion action such that it starts moving. When two objects meet you get a *collision event*. In such a case you can make the ball stop or reverse direction. You can also play a sound effect. To this end *Game Maker* lets you define *sounds*. When the player presses a key on the keyboard there is a *keyboard event*, and the object can take an appropriate action, like moving in the direction indicated. I hope you get the idea. For each object you design you can indicate actions for various events, in this way defining the behavior of the object.

Once you have defined your objects it is time to define the *rooms* in which they will live. Rooms can be used for levels in your game or to check out different places. There are actions to move from one room to another. Rooms first of all have a *background*. This can be a simple color or an image. Such background images can be created in *Game Maker* or you can load them from files. (The background can do a lot of things but for the time being, just consider it as something that makes the rooms look nice.) Next you can place the objects in the room. You can place multiple instances of the same object in a room. So, for example, you need to define just one wall object and can use it at many places. Also you can have multiple instances of the same monster objects, as long as they should have the same behavior.

Now you are ready to run the game. The first room will be shown and objects will come to life because of the actions in their creation events. They will start reacting to each

other due to actions in collision events and they can react to the player using the actions in their keyboard or mouse events.

So in summary, the following things (often called resources) play a crucial role:

- *objects*: which are the true entities in the game
- *rooms*: the places (levels) in which the objects live
- *sprites*: (animated) images that are used to represent the objects
- *sounds*: these can be used in games, either as background music or as effects
- *backgrounds*: the images used as background for the rooms

There is actually one other type of resource: *scripts*. Scripts are small pieces of code that can be used to extend the possibilities of *Game Maker*. They will be treated in the advance chapters later in this book.



## Chapter 4 Let us look at an example

It is good to first have a look at how to make a very simple example. The first step is to describe the game we want to make. (You should always do this first; it will save you a lot of work later.) The game will be very simple: There is a ball that is bouncing around between some walls. The player should try to click on the ball with the mouse. Each time he succeeds he gets a point.

As can be seen we will require two different objects: the ball and the wall. We will also need two different sprites: one for the wall object and one for the ball object. Finally, we want to hear some sound when we succeed in clicking on the ball with the mouse. We will just use one room in which the game takes place. (If you don't want to make the game yourself you can load it from the Examples folder under the name `touch the ball.gmd`.)

Let us first make the sprites. From the **Add** menu select **Add Sprite** (you can also use the appropriate button on the toolbar). A form will open. In the **Name** field type "wall". Select the **Load Sprite** button and choose an appropriate image (you can find one in the maze folder). That is all and you can close the form. In the same way, create a ball sprite.

Next we make the sound. From the **Add** menu select **Add Sound**. A different form opens. Give the sound a name and choose **Load Sound**. Pick something appropriate and check whether it is indeed a nice sound by pressing the play button. If you are satisfied, close the form.

The next step is to create the two objects. Let us first make the wall object. Again from the **Add** menu choose **Add Object**. A form will open that looks quite a bit more complex than the ones we saw so far. At the left there is some global information about the object. Give the object an appropriate name and from the drop down menu pick the correct wall sprite. Because a wall is solid you should check the box labeled **Solid**. That is all for the moment. Again create a new object, name it ball, and give it the ball sprite. We don't make the ball solid. For the ball we need to define some behavior. In the middle you see the events. Select the creation event. At the far right you see all the possible actions, in a number of groups. From the **move** group choose the action with the 8 red arrows and drag it to the list in the middle. This action will make the object move in a particular direction. Once you drop it in the action list a dialog pops up in which you can indicate the direction of motion. Select all the 8 arrows to choose a random direction. You can leave the speed as 8. Now close the dialog. So now the ball will start moving at the moment it is created. Secondly we have to define what should happen in the case of a collision event with the wall. Find the collision event button (with the two little red arrows hitting each other) and in the drop down menu select the wall object. For this event we need the bounce action. (You can see what each action does by holding the mouse still above it.) Finally we need to define what to do when the user presses the left mouse button on the ball. Find the corresponding event and select the left mouse button from the pop-up menu. For this event we need a few actions: one to play a sound (can be found in the group of **Misc.** actions) and one to change the score (in the same group) and

two more to move the ball to a new random position and moving in a new direction (in the same way as in the creation event). For the sound action, select the correct sound. For the score action, type in a value of 1 and check the **Relative** box. This means that 1 is added to the current score. (If you make a mistake you can double click the action to change its settings.)

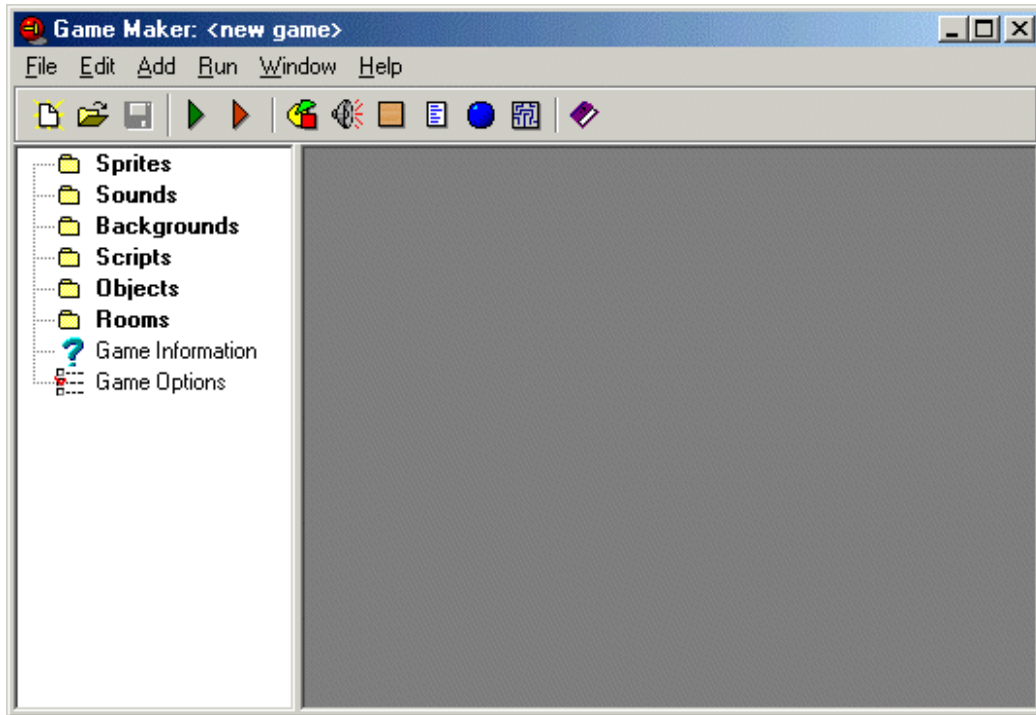
Our objects are now ready. What remains is to define the room. Add a new room to the game, again from the **Add** menu. At the right you see the empty room. At the left you find some properties that you can change, like the width and height of the room. At the bottom left you can select an object in the pop-up menu. By clicking in the room you can place instances of that object there. You can remove instances using the right mouse button. Create a nice boundary around the room using the wall object. Finally place 1 or 2 ball objects in the room. Ready.

Now it is time to test our game. Press the **Run** button and see what happens. If you made no mistakes the ball starts moving around. Try clicking on it with the mouse and see what happens. You can stop the game by pressing the <Esc> key. You can now make further changes.

Congratulations. You made your first little game. But I think it is now time to learn a bit more about *Game Maker*.

## Chapter 5 The global user interface

When you start *Game Maker* the following form is shown:



At the left you see the different *resources*, mention above: Sprites, Sounds, Backgrounds, Scripts, Objects, Rooms and two more: Game Information and Game Options. At the top there is the well-known menu and toolbar. In this chapter I will describe briefly the various menu items, buttons, etc. In the later chapters we discuss a number of them in detail. Note that many things can be achieved in different ways: by choosing a command from the menu, by clicking a button, or by right clicking on a resource.

### 5.1 File menu

In the file menu you can find some of the usual commands to load and save files, plus a few special ones:

- **New.** Choose this command to start creating a new game. If the current game was changed you are asked whether you want to save it. There is also a toolbar button for this.
- **Open** Opens a game file. *Game Maker* files have the extension *.gmd*. There is a toolbar button for this command. You can also open a game by dragging the file to the *Game Maker* window.
- **Save.** Saves the game design file under its current name. If no name was specified before, you are asked for a new name. You can only use this command when the file was changed. Again, there is a toolbar button for this.
- **Save As.** Saves the game design file under a different name. You are asked for a new name.

- **Create stand-alone.** Use this to create a stand-alone version of your game that you can distribute to other people. You will get more information about distributing games in Chapter 22.
- **Import scripts.** Can be used to import useful scripts from files. This will be discussed in more detail in Chapter 18.
- **Export scripts.** Can be used to save your scripts in a file, to be used by others. Again see Chapter 18.
- **Preferences.** Here you can set a number of preferences about *Game Maker*. They will be remembered between different calls of *Game Maker*. The following preferences can be set:
  - **Show recently edited games in the file menu.** If checked the four most recently edited games are shown in the file menu.
  - **Use compression when saving games.** If checked images in game files are compressed. This considerably reduces the game file size but will slightly increase loading and saving times.
  - **Keep backup copies of files.** If checked the program saves a backup copy of your game with the extension .bak. Change this back to .gmd and you can use it again.
  - **Hide the designer and wait while the game is running.** If checked, when you run the game, the designer window will disappear and come back when the game is finished.
  - **Run games in secure mode.** If checked, any game created with *Game Maker* that runs on your machine will not be allowed to execute external programs or change or delete files at a place different from the game location. (This is a safeguard against Trojan horses.) Checking this might make that certain games don't work correctly.
  - **In object properties, show hints for actions.** If checked, in the object properties form, when you hold your mouse over one of the actions, a description is shown.
  - **Scripts and code.** See Chapter 18 for more information about these preferences.
  - **Image editor.** Default *Game Maker* uses a built-in editor for images. If you have a better other image editing program you can indicate here to use a different program for editing the images.
- **Recent Files.** Unless disabled in the preferences, the last four games you worked on are given here, such that you can open them with a simple mouse click.
- **Exit.** Probably obvious. Press this to exit *Game Maker*. If you changed the current game you will be asked whether you want to save it.

## 5.2 Edit menu

The edit menu contains a number of commands that relate to the currently selected resource (object, sprite, sound, etc.) or group of resources. Depending on the type of resource some of the commands might not be available.

- **Insert resource.** Insert a new instance of the currently selected type of resource before the current one. (If you selected a group of resources the resource is added

in the group.) A form will open in which you can change the properties of the resource. This will be treated in detail in the following chapters.

- **Duplicate.** Makes a copy of the current resource and adds it. A form is opened in which you can change the resource.
- **Insert group.** Resources can be grouped together. This is very useful when you make large games. For example, you can put all sounds related to a certain object in a group, or you can group all objects that are used in a particular level. This command creates a new group in the currently selected resource type. You will be asked for a name. Groups can again contain groups, etc. As indicated below you can drag resources into the groups.
- **Delete.** Deletes the currently selected resource (or group of resources). Be careful. This cannot be undone. You will though be warned.
- **Rename.** Give the resource a new name. This can also be done in the property form for the resource. Also you can select the resource and then click on the name.
- **Properties.** Use this command to bring up the form to edit the properties. Note that all the property forms appear within the main form. You can edit many of them at the same time. You can also edit the properties by double clicking on the resource.
- **Show Object Information** Using this command you can get an overview of all objects in the game.

Note that all these commands can also be given in a different way. Right-click on a resource or resource group, and the appropriate pop-up menu will appear.

### 5.3 Add menu

In this menu you can add a new resources of each of the different types. Note that for each of them there is also a button on the toolbar and a keyboard shortcut.

### 5.4 Window menu

In this menu you find some of the usual commands to manage the different property windows in the main form:

- **Cascade.** Cascade all the windows such that each of them is partially visible.
- **Arrange Icons.** Arrange all the iconified property windows. (Useful in particular when resizing the main form.
- **Close All.** Close all the property windows, asking the user whether or not to save the changes made.

### 5.5 Help menu

Here you find some commands to help you:

- **Contents.** Here you can access the on-line version of this document.
- **How to use help.** In case you do not know, some help on using help.
- **Web site.** Connect you to the *Game Maker* website where you can find information about the most recent version of *Game Maker* and collections of games and resources for *Game Maker*. I recommend that you check out the site at least once a month for new information.

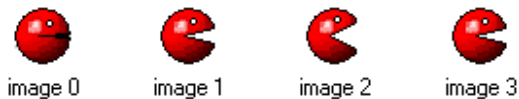
- **About Game Maker:** Give some short information about this version of *Game Maker*.

## 5.6 The resource explorer

At the left of the main form you find the resource explorer. Here you will see a tree-like view of all resources in your game. It works in the same way as the windows explorer and you are most likely familiar with it. If an item has a + sign in front of it you can click on the sign to see the resources inside it. By clicking on the – sign these disappear again. You can change a name of a resource (except the top level ones) by selecting it (with a single click) and then clicking on the name. Double click on resource to edit its properties. Use the right mouse button to access the same commands as in the **Edit** menu. You can change the order of the resources by clicking on them with the mouse and holding the mouse button pressed. Now you can drag the resource (or group) to the appropriate place. (Of course the place must be correct. You e.g. cannot drag a sound into the list of sprites.)

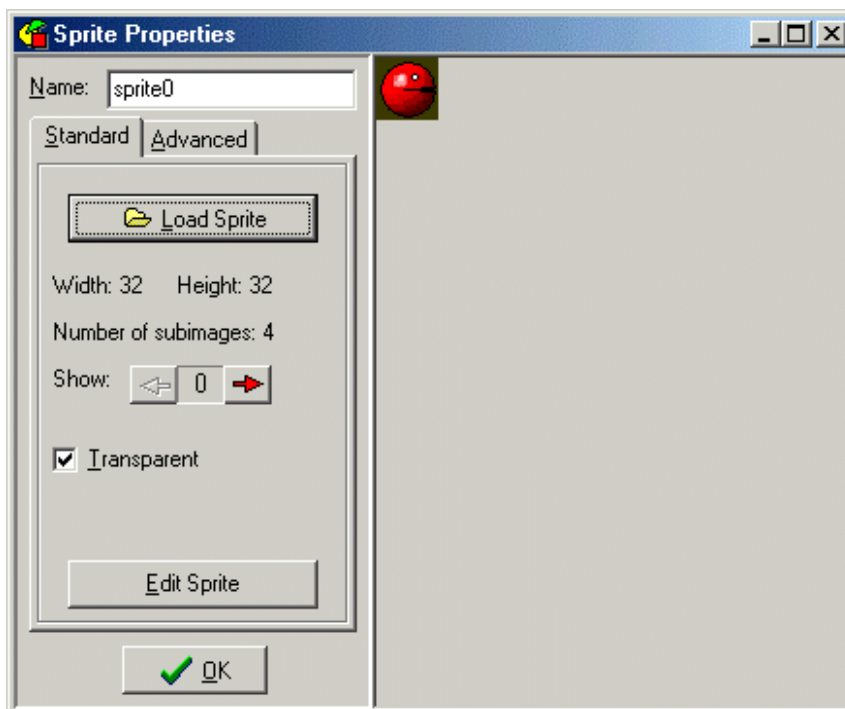
## Chapter 6 Defining sprites

Sprites are the visual representations of all the objects in the game. A sprite is either a single image, drawn with any drawing program you like, or a set of images that, when played one of the other, look like an animated motion. For example, the following four images form a sprite for a Pacman moving to the right.



When you make a game you start by collecting a set of nice sprites for the objects in your game. A collection of sprites, both static and animated, is provided with *Game Maker*. Other sprites can be found on the web, normally in the form of animated gif files.

To add a sprite, choose the item **Add Sprite** from the **Add** menu, or use the corresponding button on the toolbar. The following form will pop up.



At the top you can indicate the name of the sprite. All sprites (and all other resources) have a name. You best give each sprite a descriptive name. Make sure all resources get different names. Even though this is not strictly required, you are strongly advised to only use letters and digits and the underscore symbol (`_`) in a name of a sprite (and any other resource) and to let it start with a letter. In particular don't use the space character. This will become important once you start using code.

There are two tabs, labeled **Standard** and **Advanced**. For the moment we will only talk about the standard options. The advanced ones can be found in Chapter 13.

To load a sprite, click on the button **Load Sprite**. A standard file dialog opens in which you can indicate the sprite. *Game Maker* can load many different graphics files. When you load an animated gif, the different subimages form the sprite images. (Note that compressed animated gifs cannot be loaded.) Once the sprite is loaded the first subimage is shown on the right. When there are multiple subimages, you can cycle through them using the arrow buttons.

The checkbox labeled **Transparent** indicates whether the background should be considered as being transparent. Most sprites are transparent. The background is determined by the color of the leftmost bottommost pixel of the image. So make sure that no pixel of the actual image has this color. (Note that gif files often define their own transparency color. This color is not used in *Game Maker*.)

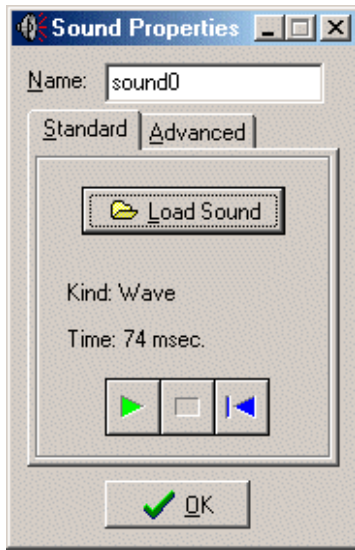
With the button **Edit Sprite** you can edit the sprite, or even create a completely new sprite. For more information on creating and changing sprites, see Chapter 13.



## Chapter 7 Sounds and music

Most games have certain sound effects and some background music. Some useful sound effects are provided with *Game Maker*. Many more can be found on the web.

To add a sound resource to your game, use the item **Add Sound** in the **Add** menu or use the corresponding button on the toolbar. The following form will pop up.



As before, at the top you can change the name of the sound resource. Again you have two tabs, **Standard** and **Advanced**. We only discuss the standard option here. For more information on the advance tab, see Chapter 14.

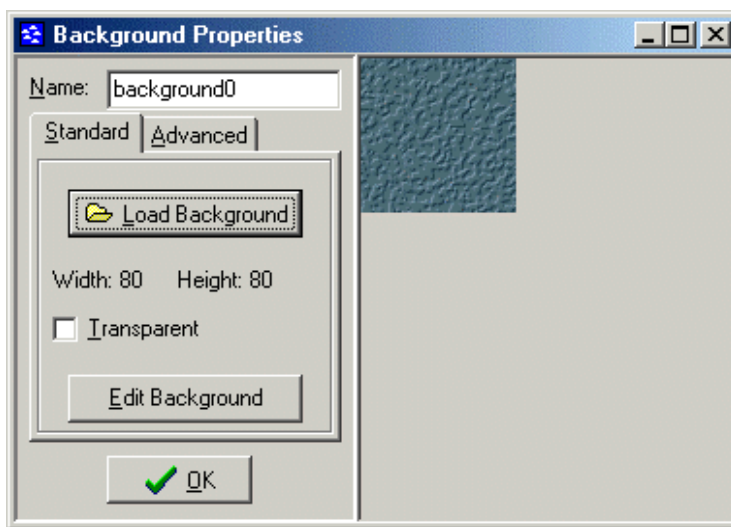
To load a sound, press the button labeled **Load Sound**. A file selector dialog pops up in which you can select the sound file. There are two types of sound files, wave files and midi files. (For information on mp3 files see Chapter 14.) Wave files are used for short sound effects. They use a lot of memory but play instantaneously. Use these for all the sound effects in your game. Midi files describe music in a different way. As a result they use a lot less memory, but they are limited to instrumental background music. Also, only one midi sound can play at any time.

Once you load a music file its kind and length are shown. You can listen to the sound using the play button.

## Chapter 8 Backgrounds

The third type of basic resources is backgrounds. Backgrounds are usually large images that are used as backgrounds (or foregrounds) for the rooms in which the game takes place. Often background images are made in such a way that they can tile an area without visual cracks. In this way you can fill the background with some pattern. A number of such tiling backgrounds are provided with *Game Maker*. Many more can be found on the web.

To add a background resource to your game, use the item **Add Background** in the **Add** menu or use the corresponding button on the toolbar. The following form will pop up.



Again you can provide a name, and again there are **Standard** and **Advanced** tabs. For more information on the advanced options, see Chapter 15. Press the button **Load Background** to load a background image. *Game Maker* supports many image formats. Background images cannot be animated! The checkbox **Transparent** indicates whether or not the background is partially transparent. Most backgrounds are not transparent so the default is not. As transparency color the color of the leftmost bottommost pixel is used.

You can change the background or create a new one using the button **Edit Background**. For more information, see Chapter 15.

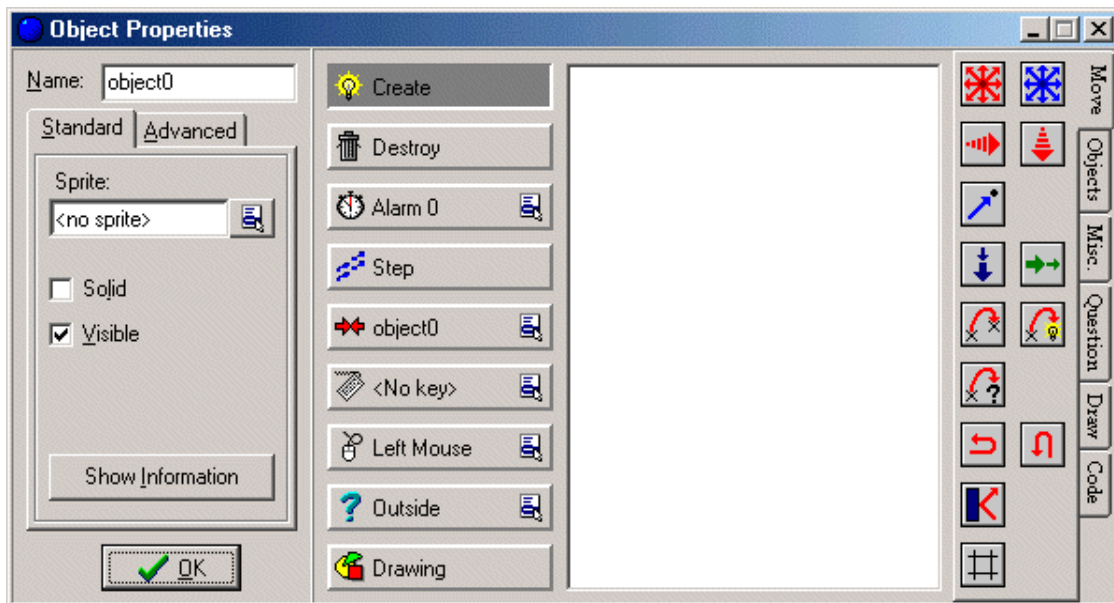
## Chapter 9 Defining objects

We might have added some nice images and sounds to the game, but they don't do anything. We now come to the most important resource of *Game Maker*: the objects. Objects are entities in the game that do things. They most of the time have a sprite as a graphical representation such that you see them. They have behavior because they can react to certain events. All things you see in the game (except for the background) are objects. (Or to be more precise, they are instances of objects.) The characters, the monsters, the balls, the walls, etc. are all objects. There might also be certain objects that you don't see but that control certain aspects of the game play.

Please realize the difference between sprites and objects. Sprites are just (animated) images that don't have any behavior. Objects normally have a sprite to represent them but object have behavior. Without objects there is no game!

Also realize the difference between objects and instances. An object describes a certain entity, e.g. a monster. There can be multiple instances of this object in the game. When we talk about an instance we mean one particular instance of the object. When we talk about an object we mean all the instances of this object.

To add an object to your game, choose **Add Object** from the **Add** menu. The following form will appear:



This is rather complex. At the left there is some general information about the object. In the middle there are the various possible events that can happen to the object. See the next chapter for details. At the right there are the different actions the object can perform. These will be treated in Chapter 11.

As always, you can (and should) give your object a name. Next you can indicate the sprite for the object. To this end, click with the left mouse button on the sprite box or the menu button next to it. A menu will pop-up with all the available sprites. Select the one you want to use for the object. Below this there are two check boxes. The box labeled **Solid** indicates whether this is a solid object (like a wall). Collisions with solid objects are treated differently from collisions with non-solid objects. See the next chapter for more information. **Visible** indicates whether instances of this object are visible. Clearly, most objects are visible, but sometimes it is useful to have invisible objects. For example, you can use them for waypoints for a moving monster. Invisible objects will react to events and other instances do collide with them.

The button **Show Information** gives an overview of all information for the object that can also be printed. This is in particular useful when you loose overview of all your actions and events.

## Chapter 10 Events

*Game Maker* uses what is called an event driven approach. That is, in all sorts of cases the instances of the objects get events (kind of messages that something has happened). They can then react to these messages by executing certain actions. For each object you must indicate to which events it responds and what actions it must perform. This may sound complicated but is actually very easy. First of all, for most events the object does not have to do anything. For the events where something must happen you can use a very simple drag-and-drop approach to indicate the actions.

In the middle of the object property form you see a number of buttons labeled **Create**, **Destroy**, etc. These are the different events. Below I will give a complete list plus descriptions. One of them is dark gray. This is the event we are currently changing. At the right there are all the actions represented by little icons. They are grouped in a number of tabbed pages. In the next chapter I will describe all the actions and what they do. Between the events and the actions there is a list. This list contains the actions for the current event. To add actions to the list, drag them with your mouse from the right to the list. They will be placed below each other, with a short description. For each action you will be asked to provide a few parameters. These will also be described in the next chapter. So after adding a few actions the situation will look as follows:



Note that the word **Create** has become bold. This indicates that it contains action. Each event that contains actions will have its name in bold. Now you can start adding actions to another event. Click on the correct event with the left mouse button to select it (the button becomes dark gray) and drag action in the list.

You can change the order of the actions in the list again using drag-and-drop. If you hold the <Ctrl> key while dragging you make a copy of the action. You can even use drag-and-drop between action lists for different objects. When you click with the right mouse

button on an action, a menu appears in which you can delete the action (can also be done by using the <Del> key) or copy and paste actions. When you hold your mouse still above an action, a longer description is given of the action. See the next chapter for more information on actions.

There exists a rather large collection of different events. As you will notice some of the event names have a menu symbol next to it. This means that there is a collection of events here. When you click with the mouse button on the menu, or right-click on the event name, a menu appears from which you can pick the event you want to change. Here is a description of the various events. (Again remember that you normally use only a few of them.)



### **Create event**

This event happens when an instance of the object is created. It is normally used to set the instance in motion and/or to set certain variables for the instance.



### **Destroy event**

This event happens when the instance is destroyed. To be precise, it happens just before it is destroyed, so the instance does still exist when the event is executed! Most of the time this event is not used but you can e.g. use it to change the score or to create some other object.



### **Alarm events**

Each instance has 8 alarm clocks. You can set these alarm clocks using certain actions (see next chapter). The alarm clock then ticks down until it reaches 0 at which moment the alarm event is generated. To indicate the actions for a given alarm clock, you first need to select it in the menu. Alarm clocks are very useful. You can use them to let certain things happen from time to time. For example a monster can change its direction of motion every 20 steps. (In such cases one of the actions in the event must set the alarm clock again.)



### **Step event**

The step event happens every step of the game. Here you can put actions that need to be executed continuously. For example, if one object should follow another, here you can adapt the direction of motion towards the object we are following. Be careful with this event though. Don't put many complicated actions in the step event of objects of which there are many instances. This might slow the game down. To be more precise, there are three different step events. Normally you only need the default one. But using the menu you can also select the begin and the end step event. The begin step event is executed at the beginning of each step, before any other events take place. The normal step event is executed just before the instances are put in their new positions. The end step event is executed at the end of the step, just before the drawing. This is typically used for e.g. changing the sprite depending on the current direction.



### **Collision events**

Whenever two instances collide (that is, their sprites overlap) a collision event appears. Well, to be precise two collision event occur; one for each instance. The instance can react to this collision event. To this end, from the menu select the object with which you want to define the collision event. Next you place the actions here.

There is a difference in what happens when the instance collides with a solid object or a non-solid object. First of all, when there are no actions in the collision event, nothing happens. The current instance simply keeps on moving; even when the other object is solid. When the collision event contains actions the following happens:

When the other object is solid, the instance is placed back at its previous place (before the collision occurs). Then the event is executed. Finally, the instance is moved to its new position. So if the event e.g. reverses the direction of motion, the instance bounces against the wall without stopping. If there is still a collision, the instance is kept at its previous place. So it effectively stops moving.

When the other object is not solid, the instance is not put back. The event is simply executed with the instance at its current position. Also, there is no second check for a collision. If you think about it, this is the logical thing that should happen. Because the object is not solid, we can simply move over it. The event notifies us that this is happening.

There are many uses for the collision event. Instances can use it to bounce against walls. You can use it to destroy object when they are e.g. hit by a bullet, etc.

### **Keyboard events**

When the player presses a key, a keyboard event happens for all instances of all objects. There is a different event for each key. In the menu you can pick the key for which you want to define the keyboard event and next drag actions there. Clearly, only a few objects need events for only a few keys. There are two different ways in which you can receive keyboard events. These are either continuous, that is, as long as the key is pressed you get a keyboard event in each step, or not continuous in which case you get just one keyboard event when the key is pressed (and more once the key start repeating). You can set this in the game options (see Chapter 20). There are two special keyboard events. One is called <No key>. This event happens in each step when no key is pressed. The second one is called <Any key> and happens whenever some key is pressed. By the way, when the player presses multiple keys, the events for all the keys pressed happen. Note that the keys on the numeric keypad only produce the corresponding events when <NumLock> is pressed.

### **Mouse events**

A mouse event happens for an instance whenever the mouse cursor lies inside the sprite representing the instance. Depending on which mouse buttons are pressed you get the no button, left button, right button, or middle button event. Note that these events only occur when the mouse is above the instance. If the player presses a mouse button on a place where there is no instance, no event is generated. Still it is sometimes important to react

to any mouse click. This can be achieved by creating a sprite with the size of the room. Now create an object with this sprite that covers the whole room. You can make the object invisible. Place it in each room, and it will get events whenever the player presses his mouse.

### Other events

There are a number of other events that can be useful in certain games. They are found in this menu. The following events can be found here:

- **Outside**: This event happens when the instance lies completely outside the room. This is typically a good moment to destroy it.
- **Boundary**: This event happens when the instance intersects the boundary of the room.
- **Game start**: This event happens for all instances in the first room when the game starts. It happens before the room start event (see below) and even before the creation events for the instances in the room. This event is typically defined in only one "controller" object and is used to start some background music and to initialize some variables, or load some data.
- **Game end**: The event happens to all instances when the game ends. Again typically just one object defines this event. It is for example used to store certain data in a file.
- **Room start**: This event happens for all instances initially in a room when the room starts. It happens before the creation events.
- **Room end**: This event happens to all existing instances when the room ends.
- **No more lives**: *Game Maker* has a built-in lives system. There is an action to set and change the number of lives. Whenever the number of lives becomes less than or equal to 0, this event happens. It is typically used to end or restart the game.
- **End of animation**: As indicated above, an animation consists of a number of images that are shown one after the other. After the last one is shown we start again with the first one. The event happens at precisely that moment. This can be used to e.g. change the animation, or destroy the instance.
- **User defined**: There are eight of these events. They normally never happen unless you yourself call them from a piece of code.

### Drawing event

Instances, when visible, draw their sprite in each step on the screen. When you specify actions in the drawing event, the sprite is not drawn, but these actions are executed instead. This can be used to draw something else than the sprite, or first make some changes to sprite parameters. There are a number of drawing actions that are especially meant for use in the drawing event. Note that the drawing event is only executed when the object is visible. Also note that, independent of what you draw here, collision events are based on the sprite that is associated with the instance.



## Chapter 11 Actions

Actions are the things that happen in *Game Maker*. Actions are placed in events. Whenever the event takes place these actions are performed. There are a large number of different actions available and it is important that you understand what they do. In this chapter I will describe all of them.

All the actions are found in the tabbed pages at the right of the object property form. There are six sets of actions. You get the set you want by clicking on the correct tab. When you hold your mouse above one of the actions, a short description is shown to remind you of its function.

Let me briefly repeat: To put an action in an event, just drag it from the tabbed pages to the action list. You can change the order in the list, again using dragging. Holding the <Ctrl> key while dragging makes a copy of the action. (You can drag and copy actions between the lists in different object property forms.) Use the right mouse button to remove actions (or use the <Del> key) and to copy and paste actions.

When you drop an action, a window will pop-up most of the time, in which you can fill in certain parameters for the action. Two types of parameters appear in many actions. At the top you can indicate to which instance the action applies. The default is self, which is the instance for which the action is performed. Most of the time this is what you want. In the case of a collision event, you can also specify to apply the action to the other instance involved in the collision. In this way you can e.g. destroy the other instance. Finally, you can choose to apply the action to all instances of a particular object. In this way you can e.g. change all red balls into blue balls. The second type of parameter is the box labeled **Relative**. By checking this box, the values you type in are relative to the current values. For example, in this way you can add something to the current score, rather than changing the current score to the new value. The other parameters will be described below. You can later change the parameters by double clicking on the action.

### 11.1 Movement actions

The first set of actions consists of those related to movement of objects. The following actions exist:



#### **Start moving in a direction**

Use this action to start the instance moving in a particular direction. You can indicate the direction using the arrow keys. Use the middle button to stop the motion. Also you need to specify the speed of the motion. This speed is given in pixels per step. The default value is 8. Preferably don't use negative speeds. You can specify multiple directions. In this case a random choice is made. In this way you can e.g. let a monster start moving either left or right.



#### **Set direction and speed of motion**

This is the second way to specify a motion with the blue arrows). Here you can indicate a precise direction. This is an angle between 0 and 360 degrees. 0 mean to the right. The direction is counter-clockwise. So for example 90 indicates an upward direction. If you want an arbitrary direction, you can type `random( 360 )`. As you will see below the function `random` gives a random number smaller than the indicated value. As you might have noticed there is a checkbox labeled **Relative**. If you check this, the new motion is added to the previous one. For example, if the instance is moving upwards and you add a bit of motion to the left, the new motion will be upwards to the left.



### Set the horizontal speed

The speed of an instance consists of a horizontal part and a vertical part. With this action you can change the horizontal speed. A positive horizontal speed means a motion to the right. A negative one a motion to the left. The vertical speed will remain the same. Use relative to increase the horizontal speed (or decrease it by providing a negative number).



### Set the vertical speed

In a similar way, with this action you can change the vertical speed of the instance.



### Move towards a point

This action gives another way to specify a motion. You indicate a position and a speed and the instance starts moving with the speed towards the position. (It won't stop at the position!) For example, if you want a bullet to fly towards the position of the spaceship you can use as position `spaceship.x`, `spaceship.y`. (You will learn more about the use of variables like these below.) If you check the **Relative** box, you specify the position relative to the current position of the instance. (The speed is not taken relative!)



### Set the gravity

With this action you can create gravity for this particular object. You specify a direction (angle between 0 and 360 degrees) and a speed, and in each step this amount of speed in the given direction is added to the current motion of the object instance. Normally you need a very small speed increment (like 0.01). Typically you want a downward direction (270 degrees). If you check the **Relative** box you increase the gravity speed and direction. Note that, contrary to real life, different object can have different gravity directions.



### Set the friction

Friction slows down the instances when they move. You specify the about of friction. In each step this amount is subtracted from the speed until the speed becomes 0. Normally you want a very small number here (like 0.01).



### Jump to a given position

Using this action you can place the instance in a particular position. You simply specify the x- and y-coordinate, and the instance is placed with its reference point on that

position. If you check the **Relative** box, the position is relative to the current position of the instance.



#### **Jump to the start position**

This action places the instance back at the position where it was created.



#### **Jump to a random position**

This action move the instance to a random position in the room, Only position are chosen where the instance does not intersect any solid instance. You can specify the snapping used. If you specify positive values, the coordinates chosen with be integer multiples of the indicated values. This can be used to e.g. keep the instance aligned with the cells in your game (if any). You can specify a separate horizontal snapping and vertical snapping.



#### **Reverse horizontal direction**

With this action you reverse the horizontal motion of the instance. This can for example be used when the object collides with a vertical wall.



#### **Reverse vertical direction**

With this action you reverse the vertical motion of the instance. This can for example be used when the object collides with a horizontal wall.



#### **Bounce against objects**

When you put this action in the collision event with some object, the instance bounces back from this object in a natural way. If you set the parameter precise to false, only horizontal and vertical walls are treated correctly. When you set precise to true also slanted (and even curved) walls are treated satisfactory. This is though slower. Also you can indicate whether to bounce only from solid objects or from all objects. Please realize that the bounce is not completely correct because this depends on many properties. But in many situations the effect is good enough.



#### **Snap to grid**

With this action you can round the position of the instance to a grid. You can indicate both the horizontal and vertical snapping value (that is, the sized of the cells of the grid). This can be very useful to make sure that instances stay on a grid.

## **11.2 Object related actions**

The following set of actions deals with creating, changing, and destroying instances of objects.



#### **Create an instance of an object**

With this action you can create an instance of an object. You specify which object to create and the position for the new instance. If you check the **Relative** box, the position is relative to the position of the current instance. Creating instances during the game is

extremely useful. A space ship can create bullets; a bomb can create an explosion, etc. In many games you will have some controller object that from time to time creates monsters or other objects. For the newly created instance the creation event is executed.



#### **Change the instance**

With this action you can change the current instance into another object. So for example, you can change an instance of a bomb into an explosion. All settings, like the motion or the value of variables, will stay the same. You can indicate whether or not to perform the destroy event for the current object and the creation event for the new object.



#### **Destroy the instance**

With this action you destroy the current instance. The destroy event for the instance is executed.



#### **Destroy instances at a position**

With this action you destroy all instances whose bounding box contains a given position. This is for example useful when you use an exploding bomb. When you check the **Relative** box the position is taken relative to the position of the current instance.



#### **Change the sprite**

Use this action to change the sprite for the instance. You indicate the new sprite. You can also indicate a scaling factor. A factor of 1 means that the sprite is not scaled. The scaling factor must be larger than 0. Please realize that scaling the sprite will slow down the drawing. Changing sprites is an important feature. For example, often you want to change the sprite of a character depending on the direction in which it walks. This can be achieved by making different sprites for each of the (four) directions. Within the keyboard events for the arrow keys you set the direction of motion and the sprite.

### **11.3 Miscellaneous actions**

Here is a large collection of actions that deal with all sorts of aspects, like sound, rooms, score, etc.



#### **Play a sound**

With this action you play one of the sound resources you added to your game. You can indicate the sound you want to play and whether it should play once (the default) or loop continuously. Multiple wave sounds can play at once but only one midi sound can play. So if you start a midi sound, the current midi sound is stopped. Unless indicated in the advanced tab (see Chapter 14) only one instance of each sound can play. So if the same sound is already playing it is stopped and restarted.



#### **Stop a sound**

This action stops the indicated sound. If multiple instances of this sound are playing all are stopped.



### Set an alarm clock

With this action you can set one of the eight alarm clocks for the instance. You indicate the number of steps and the alarm clock. After the indicated number of steps, the instance will receive the corresponding alarm event. You can also increase or decrease the value by checking the **Relative** box. If you set the alarm clock to a value less than or equal to 0 you switch it off, so the event is not generated.



### Display a message

With this action you can display a message in a dialog box. You simply type in the message. If you use a # symbol in the message text it will be interpreted as a new line character. If the message text starts with a quote or double quote symbol, it is interpreted as an expression. See below for more information about expressions. (Note that this action does not work when your game must run in exclusive mode, see Chapter 20.)



### Set the score

*Game Maker* has a built-in score mechanism. The score is normally displayed in the window caption. You can use this action to change the score. You simply provide the new value for the score. Often you want to add something to the score. In this case don't forget to check the **Relative** box.



### Display the highscore list

This action displays the highscore list. For each game the top ten scores are maintained. These are shown in the list. If the current score is among the top ten, the new score is inserted and the player can type his or her name. There are many ways in which you can change the visual appearance of the highscore list. See Chapter 20 for detail. (This action does not work in exclusive mode!)



### Set the number of lives

*Game Maker* also has a built-in lives system. The number of lives is normally displayed in the caption but you can switch this off. With this action you can change the number of lives left. Normally you set it to some value like 3 at the beginning of the game and then decrease or increase the number depending on what happens. Don't forget to check the **Relative** box if you want to add or subtract from the number of lives. At the moment the number of lives becomes 0 (or smaller than 0) a "no more lives" event is generated.



### Sleep for a while

With this action you can freeze the scene for a particular time. This is typically used at the beginning or end of a level or when you give the player some message. You specify the number of milliseconds to sleep. Also you can indicate whether the screen should first be redrawn to reflect the most recent situation.



### Go to the previous room

Move to the previous room. You can indicate the type of transition effect between the rooms. Experiment to see what works nice for you. If you are in the first room you get an error.



#### **Go to the next room**

Move to the next room. You can indicate the transition.



#### **Restart the current room**

The current room is restarted. You indicate the transition effect.



#### **Go to a different room**

With this action can go to a particular room. You indicate the room and the transition effect.



#### **Restart the game**

With this action you restart the game from the beginning.



#### **End the game**

With this action you end the game.



#### **Save the game**

With this action you can save the current game status. You specify the filename for saving (the file is created in the working directory for the game). Later the game can be loaded with the next action.



#### **Load the game**

Load the game status from a file. You specify the file name. Make sure the saved game is for the same game and created with the same version of *Game Maker*. Otherwise an error will occur. (To be precise, the game is loaded at the end of the current step. So some actions after this one are still executed in the current game, not the loaded one!)



#### **Show the game information**

With this action you pop up the game information window. See Chapter 19 for more information on how to create the game information.

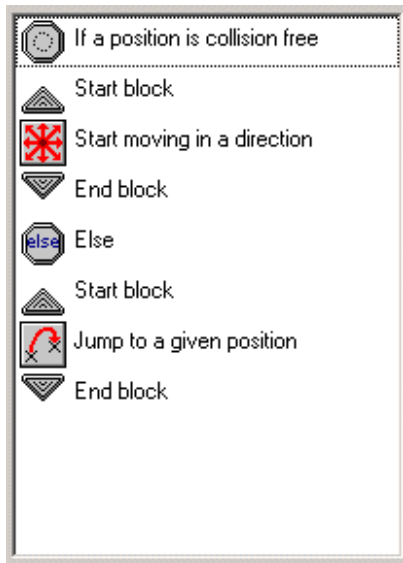


#### **Switch fullscreen mode**

With this action you can change the screen mode from windowed to fullscreen and back. You can indicate whether to toggle the mode or whether to go to windowed or fullscreen mode. (This does not work in exclusive mode.)

## 11.4 Questions

There are a number of actions that ask a question, for example whether a position is empty. When the answer is yes (true) the next action is executed, otherwise it is skipped. If you want multiple actions to be executed or skipped based on the outcome you can put them in a block by putting start block and end block actions around them. There can also be an else part which is executed when the answer is no. So a question typically looks as follows:



Here the question is asked whether a position for the current instance is collision free. If so, the instance starts moving in a given direction. If not, the instance jumps to a given position.

For all questions there is a field labeled **NOT**. If you check this field, the result of the question is reversed. That is, if the result was true it becomes false and if it was false, it becomes true. This allows you to perform certain actions when a question is not true.

For many questions you can indicate that they should apply to all instances of a particular object. In this case the result is true only if it is true for all instances of the object. For example, you can check whether for all balls the position slightly to the right is collision free.

The following questions and related actions are available. (Note that they all have a differently shaped icon such that they can more easily be distinguished from other actions.)



### **If a position is collision free**

This question returns true if the current instance, placed at the indicated position does not generate a collision with an object. You can specify the position either absolute or relative. You can also indicate whether only solid objects should be taken into account or

all objects should be taken into account. This action is typically used to check whether the instance can move to a particular position.



**If there is a collision at a position**

This is the reverse of the previous action. It returns true if there is a collision when the current instance is placed at the given position (again, either only with solid objects or with all objects).



**If there is an object at a position**

This question returns true if the instance placed at the indicate position meets an instance of the indicated object.



**If the number of instances is a value**

You specify an object and a number. If the current number of instances of the object is equal to the number the question returns true. Otherwise it returns false. You can also indicate that the check should be whether the number of instances is smaller than the given value or larger than the given value. This is typically used to check whether all instances of a particular type are gone. This is often the moment to end a level or a game.



**If a dice lands on one**

You specify the number of sides of the dice. Then if the dice lands on one, the result is true and the next action is performed. This can be used to put an element of randomness in your game. For example, in each step you can generate with a particular chance a bomb or change direction. The larger the number of sides of the dice is, the smaller the chance. You can actually use real numbers. For example if you set the number of sides to 1.5 the next action is performed two out of three times. Using a number smaller than 1 makes no sense.



**If a sound is playing**

The result of this question is true if the indicated sound is playing.



**If the user answers yes to a question**

You specify a question. A dialog is shown to the player with a yes and a no button. The result is true is the player answers yes. This action cannot be used in exclusive mode; the answer will then always be yes.



**If an expression is true**

You can enter an expression. If the expression evaluates to true (that is, a number larger or equal to 0.5) the next action is performed. See below for more information on expressions.



**If a mouse button is pressed**



Returns true if the indicated mouse button is pressed. A standard use is in the step event. You can check whether a mouse button is pressed and, if so, for example move to that position (use the jump to a point action with values `mouse_x` and `mouse_y`).



#### **If instance is aligned with grid**

Returns true if the position of the instance lies on a grid. You specify the horizontal and vertical spacing of the grid. This is very useful when certain actions, like making a turn, are only allowed when the instance is on a grid position.



#### **Else**

Behind this action the else part follows, that is executed when the result of the question is false.



#### **Start of block**

Indicates the start of a block of actions.



#### **End of block**

Indicates the end of a block of actions.



#### **Repeat next action**

This action is used to repeat the next action (or block of actions) a number of times. You simply indicate the number.



#### **Exit the current event**

When this action is encountered no further action in this event are executed. This is typically used after a question. For example, when a position is free nothing needs to be done so we exit the event. In this example, the following actions are only executed when there is a collision.

## **11.5 Drawing actions**

Drawing actions only make sense in the drawing event. At other places they are basically ignored. Please remember that drawing things other than sprites and background images is relatively slow. So use this only when strictly necessary.



#### **Draw a sprite image**

You specify the sprite, the position (either absolute or relative to the current instance position) and the subimage of the sprite. (The subimages are number from 0 upwards.) If you want to draw the current subimage, use number `-1`.



#### **Draw a background image**

You indicate the background image, the position (absolute or relative) and whether the image should be tiled all over the room or not.



### **Draw a text**

You specify the text and the position. If the text starts with a quote or a double quote, it is interpreted as an expression. For example, you can use

```
'Score: ' + string(score)
```

to display the value of the score. (The variable `score` stores the current score. The function `string()` turns this number into a string. `+` combines the two strings.) In a similar way you can display the number of lives left or any other variable.



### **Set the font for drawing text**

You can set the font that is from this moment on used for drawing text.



### **Draw a rectangle**

You specify the coordinates of the two opposite corners of the rectangle; either absolute or relative to the current instance position.



### **Draw an ellipse**

You specify the coordinates of the two opposite corners of the surrounding rectangle; either absolute or relative to the current instance position.



### **Draw a line**

You specify the coordinates of the two endpoints of the line; either absolute or relative to the current instance position.



### **Set the fill color**

Lets you set the color used to fill the rectangles and ellipses.



### **Set the line color**

Lets you set the color used for the lines around the rectangle and ellipse and when drawing a line.

## **11.6 Code related actions**

Finally there are a number of actions that primarily deal with code.



### **Execute a script**

With this action you can execute a script that you added to the game. You specify the script and the maximal 3 arguments for the script. See Chapter 18 for more information about scripts.



### **Set the value of a variable**

There are many built-in variables in the game. With this action you can change these. Also you can create your own variables and assign values to them. You specify the name of the variable and the new value. When you check the **Relative** box the value is added to the current value of the variable. Please note that this can only be done if the variable already has a value assigned to it! See below for more information about variables.



### Execute a piece of code

When you add this action, a form shows in which you can type in a piece of code. This works in exactly the same way as when defining scripts (see Chapter 18). The only difference is that you can indicate for what instances the piece of code must be executed. Use the code action for small pieces of code. For longer pieces you are strongly advised to use scripts.



### Call the inherited event

This action is only useful when the object has a parent object (see Chapter 16). It calls the corresponding event in the parent object.



### Comment

Use this action to add a line of comment to the action list. The line is shown in italic font. It does not do anything when executing the event. Adding comments helps you remember what your events are doing.

## 11.7 Using expressions and variables

In many actions you need to provide values for parameters. Rather than just typing a number, you can also type a formula, e.g.  $32 \times 12$ . But you can actually type much more complicated expressions. For example, if you want to double the horizontal speed, you could set it to  $2 \times \text{hspeed}$ . Here `hspeed` is a variable indicating the current horizontal speed of the instance. There are a large number of other variables that you can use. Some of the most important ones are:

- x** the x-coordinate of the instance
- y** the y-coordinate of the instance
- hspeed** the horizontal speed (in pixels per step)
- vspeed** the vertical speed (in pixels per step)
- direction** the current direction of motion in degrees (0-360)
- speed** the current speed in this direction
- visible** whether the object is visible (1) or invisible (0)
- image\_scale** the amount the image is scaled (1 = not scaled)
- image\_single** this variable indicate which subimage in the current sprite must be shown; if you set it to -1 (default) you loop through the images, otherwise only the indicated subimage (starting with number 0) is shown all the time
- score** the current value of the score
- lives** the current number of lives
- mouse\_x** x-position of the mouse

**mouse\_y** y-position of the mouse

You can change most of these variables using the set variable action. You can also define your own variables by setting them to a value. (Don't use relative, because they don't exist yet.) Then you can use these variables in expressions. Variables you create are local to the current instance. That is, each object has its own copy of them. To create a global variable, put the word `global` and a dot in front of it.

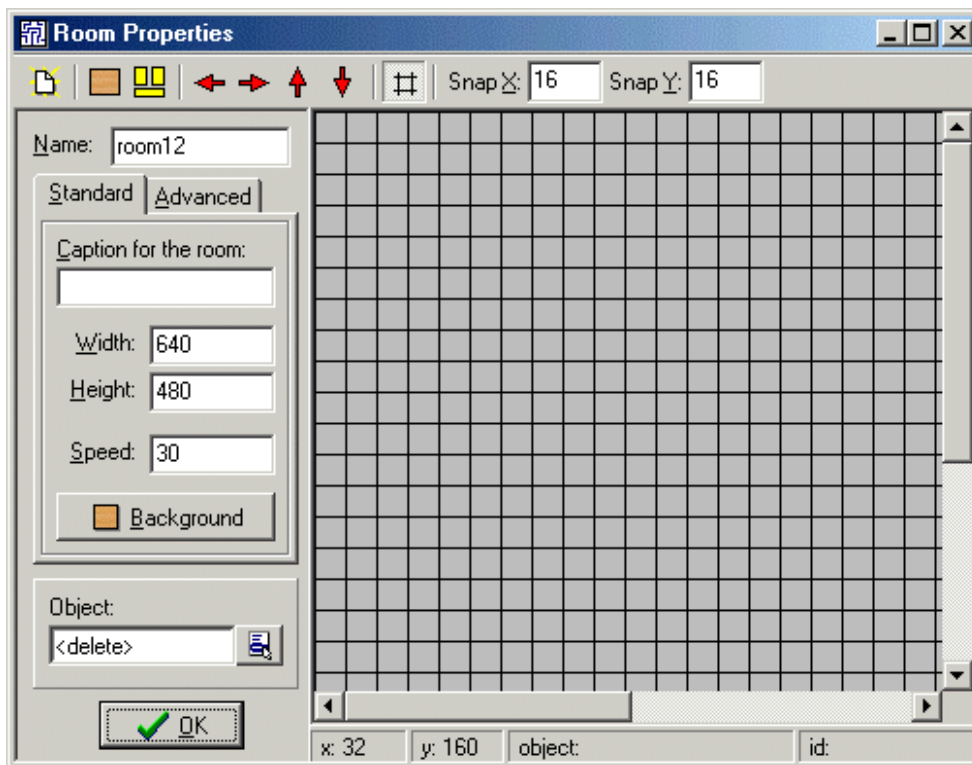
You can also refer to the values of variables for other objects by putting the object name and a dot in front of them. So for example, if you want a ball to move to the place where the coin is you can set the position to `(coin.x , coin.y)`. In the case of a collision event you can refer to the x-coordinate of the other object as `other.x`. In conditional expressions you can use comparisons like `<` (smaller than), `>`, etc.

In your expressions you can also use functions. For example, the function `random(10)` gives a random real number below 10. So you can set for example the speed or direction of motion to a random value. Many more functions exist. For more precise information on expressions and functions see Chapter 23 and further.

## Chapter 12 Creating rooms

Now that you have defined object with their behavior in the form of events and actions, it is time to create the rooms or levels in which the game takes place. Any game will need at least one room. In these rooms we place instances of the objects. Once the game starts the first room is shown and the instances in it come to life because of the actions in their creation events.

To create a room, choose **Add Room** from the **Add** menu. The following form will appear:



At the left you find some settings for the room. As before there is a **Standard** tab and an **Advanced** tab. We will only discuss the standard commands here. For the advanced options see Chapter 17).

Each room has a name. Best give it a meaningful name. There also is a caption. This caption is displayed in the window caption when the game is running. You can set the width and height of the room (in pixels). Don't make these too large. It will slow down the game and cost many resources. Also you can set the speed of the game. This is the number of steps per second. The higher the speed, the smoother the motion is. But you will need a rather fast computer to run it.

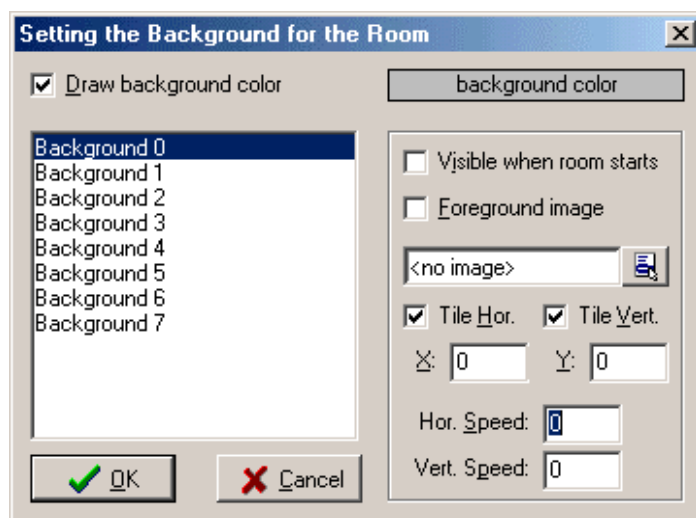
To add instances to the room, first select the object at the bottom left. Now click with your mouse in the room area at the right. An instance of the object appears. It will snap to

the indicated grid. You can change the grid in the toolbar. Also you can indicate there whether or not to draw the grid lines. With the right mouse button you can remove instances. In this way you define the contents of the room. If you hold down the mouse button , multiple instances are added or removed.

As you will notice, if you place an instance on top of another one, the original instance disappears. Normally this is what you want, but not always. This can be avoided by holding the <Ctrl> key when placing the instance. There are three more useful combinations when dealing with overlapping instances. When you hold the <Ctrl> key while clicking on an instance with the right mouse button, the bottommost instance at the position is brought to the top. Holding the <Alt> key will send the topmost instance to the bottom. This can be used to change the order of overlapping instances. Finally, holding the <Shift> key while clicking with the right mouse button will remove all instances at the position, not just the top one.

On the toolbar you will find some buttons to shift all instances in one of the four directions. This is useful when you decided to e.g. enlarge the room. (You can also use this to place instances outside the room, which is sometimes useful.) Also there is a button to clear the room.

With the button **Background** you can set the background image for the room. Actually, you can specify multiple backgrounds. The following form will show:



At the top you will see the background color. You can click on it to change it. The background color is only useful if you don't use a background image that covers the whole room. Otherwise, you better uncheck the box labeled **Draw background color** because this will be a waste of time.

At the left you see a list of 8 backgrounds. You can define each of them but most of the time you will need just one or two. To define a background, first select it in the list. Next check the box labeled **Visible when room starts** otherwise you won't see it. The name of

the background will become bold when it is defined. Now indicate a background image in the menu. There are a number of settings you can change. First of all you can indicate whether the background image should tile the room horizontally and/or vertically. You can also indicate the position of the background in the room (this will also influence the tiling). Finally you can make the background scrolling by giving it a horizontal or vertical speed.

There is one more checkbox labeled **Foreground image**. When you check this box, the background is actually a foreground, which is drawn on top of everything else rather than behind it. Clearly such an image should be partially transparent to be of any use.

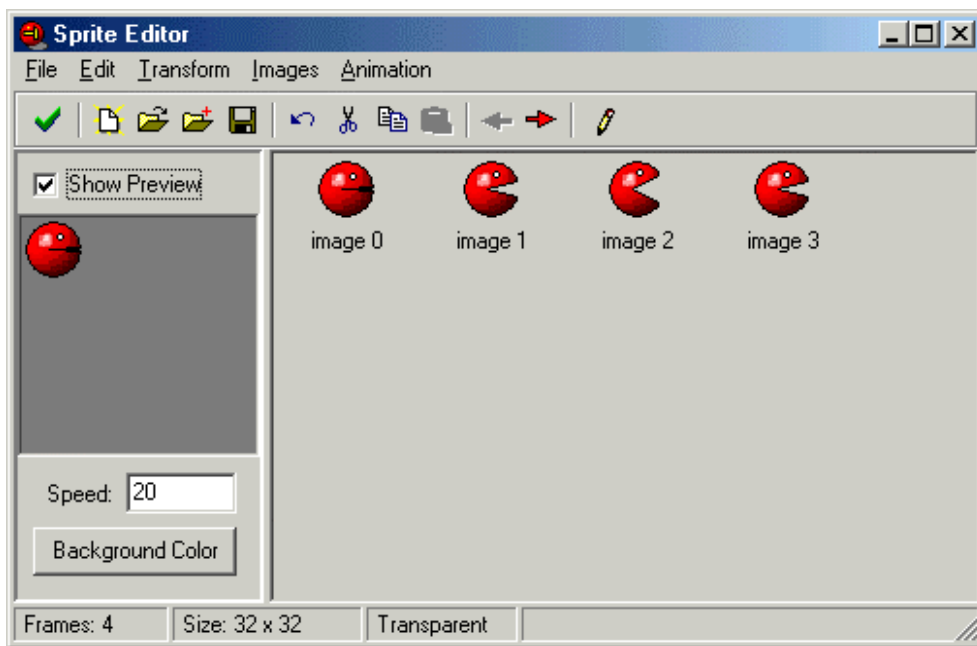
A good use of background and foreground images can greatly enhance your game. As you will see in Chapter 15, backgrounds can also be used to store static objects.

## Chapter 13 More about sprites

Up to now we loaded our sprites from files. It is though also possible to create and in particular modify them within *Game Maker*. To do this, open the sprite property window by double clicking on one of your sprite (or by creating a new one). Now press the button labeled **Edit Sprite**. A new form will appear showing all the sub-images that make up the sprite.

### 13.1 Editing your sprites

The sprite edit form will look as follows:



At the right you see the different images that make up the sprite. Note that in *Game Maker* all subimages of a sprite must have the same size. At the left an animation of the sprite plays. (If you don't see the animation, check the box labeled **Show Preview**. Below the preview you can change the speed of the animation and the background color. In this way you can get an idea of what the animation will look like in the game.

The sprite editor contains many commands to create and change the sprite. These are all given through the menus. (For some there are buttons on the toolbar.) Some commands work on individual images. They require that you first select a subimage with the mouse.

#### 13.1.1 File menu

The file menu contains a number of commands related to loading and saving sprites.

- **New.** Create a new, empty sprite. You must indicate the size of the sprite. (Remember, all images in a sprite must have the same size.)



- **Create from file.** Create the sprite from a file. Many file types can be used. They all create a sprite consisting of a single image, except for animated GIF files that are split into the subimages. Please note that *Game Maker* cannot handle optimized animated GIF files! Also, note that the transparency color is the bottommost leftmost pixel, not the transparency color in the GIF file.
- **Add from file.** Add an image (or images) from a file to the current sprite. If the images do not have the same size you can choose were to place them or to stretch them.
- **Save as GIF.** Saves the sprite as an animated gif.
- **Save as strip.** Saves the sprite as a bitmap, with all images next to each other.
- **Create from strip.** Allows you to create a sprite from a strip. See below for more information.
- **Add from strip.** Use this to add images from a strip. See below.
- **Close saving changes.** Closes the form, saving the changes made to the sprite. If you don't want to save the changes, click on the close button of the window.

### 13.1.2 Edit menu

The edit menu contains a number of commands that deal with the currently selected sprite. You can cut it to the clipboard, paste an image from the clipboard, clear the current sprite, delete it, and move sprites left and right in the sequence. Finally, there is a command to edit an individual image using the built-in painting program (see below).

### 13.1.3 Transform menu

In the transform menu you can perform a number of transformations on the images.

- **Mirror horizontal.** Mirrors the images horizontally.
- **Flip vertical.** Flips the images vertically.
- **Shift.** Here you can shift the images an indicated amount horizontally and vertically.
- **Rotate.** You can rotate the images 90 degrees, 180 degrees, or an arbitrary amount. In the latter case you can also specify the quality. Experiment to get the best effects.
- **Resize Canvas.** Here you can change the size of the canvas. You can also indicate where the old images are placed on the new canvas.
- **Stretch.** Here you can stretch the images into a new size. You can indicate the scale factor and the quality.
- **Scale.** This commands scales the images (but not the image size!). You can indicate the scale factor, the quality, and the position of the current images in the scaled ones.

### 13.1.4 Images menu

In the images menu you can perform a number of operation on the images.

- **Cycle left.** Cycles all images one place to the left. This effectively starts the animation at a different point.

- **Cycle right.** Cycles all images one place to the right.
- **Black and white.** Makes the sprite black and white (does not affect the transparency color!).
- **Colorize.** Here you can change the color (hue) of the images. Use the slider to pick the different colors.
- **Intensity.** Here you can change the intensity by providing values for the color saturation and the lightness of the images.
- **Fade.** Here you specify a color and an amount. The colors in the images are now faded towards this color.
- **Transparency.** Here you can indicate a level of screen-door transparency. This is achieved by making a number of pixels transparent.
- **Blur.** By blurring the images the colors are mixed a bit, making it more vague. The higher the value, the more vague it becomes.
- **Crop.** This makes the images as small as possible. This is very useful because the larger the images, the more video memory *Game Maker* will use. You might want to leave a little border around the images though to avoid transparency problems.

You will have to experiment with these commands to get the sprites you want.

### 13.1.5 Animation menu

Under the animation menu you can create new animations out of the current animation. There are many options and you should experiment a bit with them to create the effects you want. Also don't forget that you can always save an animation and later add it to the current one. Also you can always add some empty images and delete unwanted ones. I will briefly go through the different possibilities.

- **Set Length.** Here you can change the length of your animation. The animation is repeated enough times to create the number of frames you indicate. (Normally you want this to be a multiple of the current number of frames.)
- **Stretch.** This command also changes the length of the animation. But this time, frames are duplicated or removed to get the right number. So if you increase the number of frames the animation goes slower and if you decrease the number it goes faster.
- **Reverse.** Well, as you could guess this reverses the animation. So it is played backwards.
- **Add Reverse.** This time the reverse sequence is added, doubling the number of frames. This is very useful for making an object go left and right, change color and return, etc. You sometimes might want to remove the double first and middle frame that occur.
- **Translation sequence.** You can create an animation in which the image slightly translates in each step. You must provide the number of frames and the total amount to move horizontally and vertically.
- **Rotation sequence.** Creates an animation in which the image rotates. You can either choose clockwise or counterclockwise rotation. Specify the number of frames and the total angle in degrees (360 is a complete turn). (You might need to

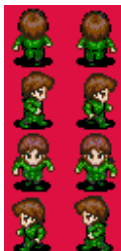
resize the canvas first to make sure the total image remains visible during the rotation.)

- **Colorize**. Creates an animation that turns the image into a particular color.
- **Fade to color**. Creates an animation that fades the image to a particular color.
- **Disappear**. Makes the image disappear using screen-door transparency.
- **Shrink**. Shrinks the image to nothing. You can indicate the direction.
- **Grow**. Grows the image from nothing.
- **Flatten**. Flattens the image to nothing in a given direction.
- **Raise**. Raises the image from a given direction
- **Overlay**. Overlays the animation with another animation or image in a file.
- **Morph**. Morphs the animation to an animation or image from a file. Note that morphing works best if the two animations cover the same area of the image. Otherwise, halfway certain pixels disappear and others suddenly appear.

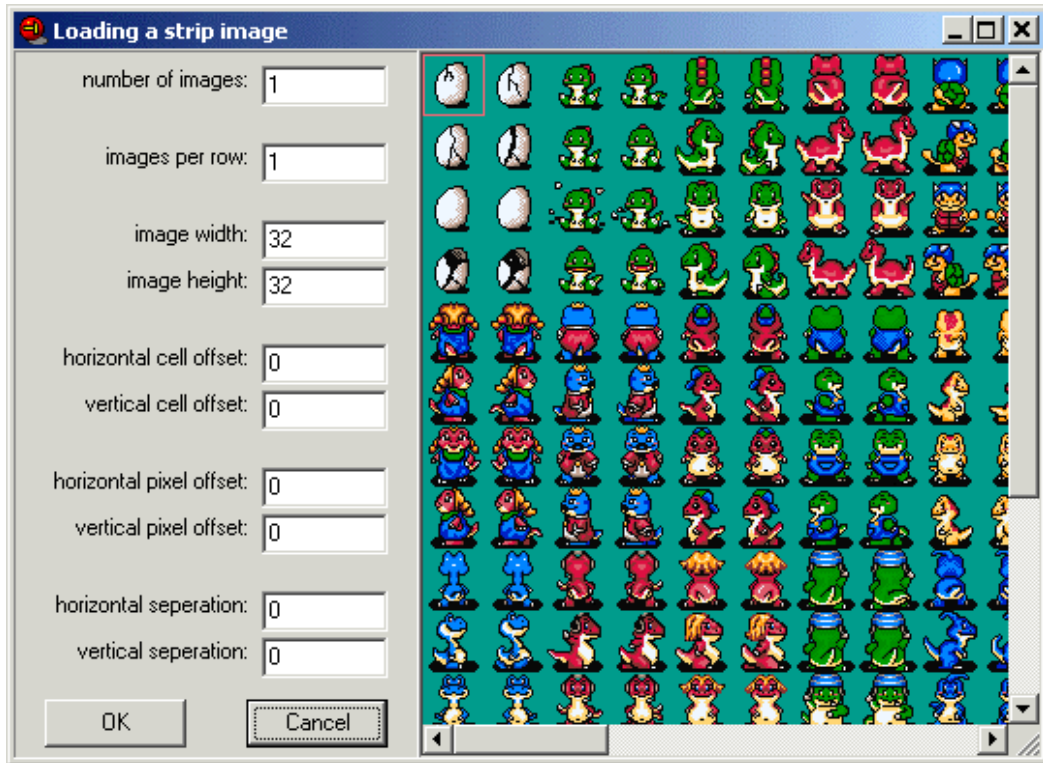
In particular the last two commands are very powerful. For example, to blow up an object, add a number of copies and then a number of empty frames. Then overlay it with an explosion animation. (Make sure the number of images match.) Alternatively, morph it to the explosion. With some practice you can make great sprites.

### 13.1.6 Strips

As indicated above, sprites are normally either stored as animated gif files or as strips. A strip is one big bitmap that stores all the images next to each other. The only problem is that the size of the individual subimages is not stored in the image. Also, many strip files available on the web store multiple sprites in one file. For example, in the following piece of a strip file contains four different animations.



To select individual sprites out of such files, you can choose **Create from Strip** or **Add from Strip** from the **File** menu. After indicating the appropriate strip image file, the following form will show:



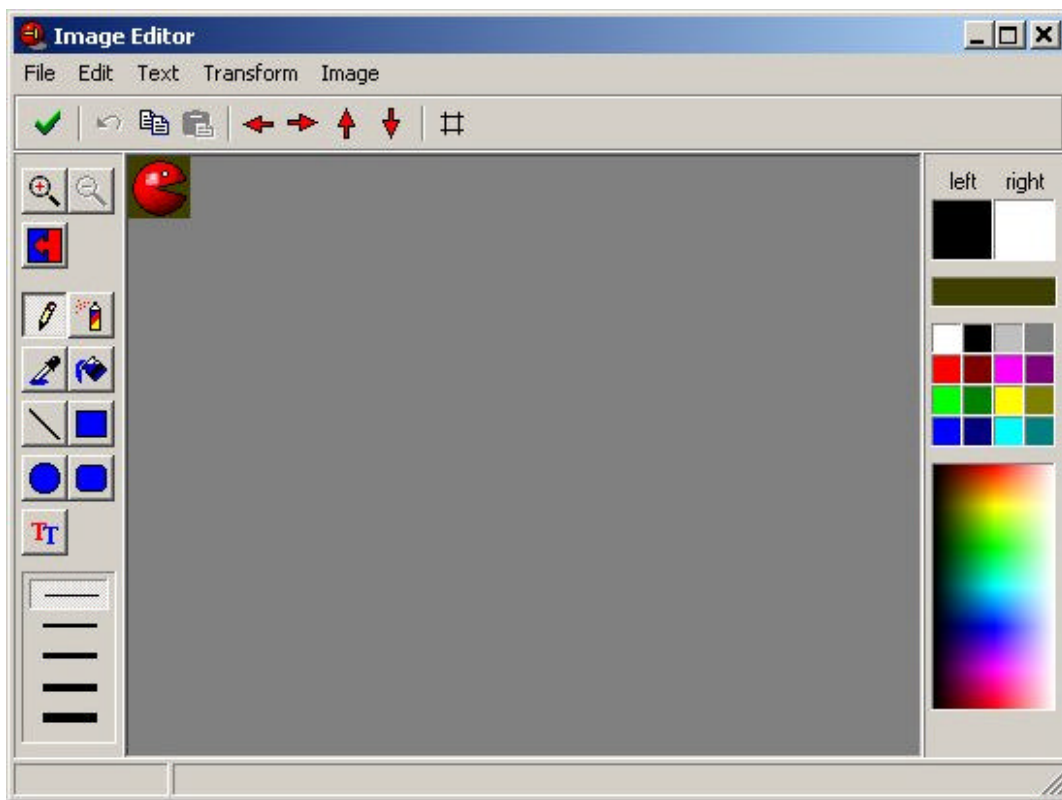
At the right you see (part of) the strip image you selected. At the left you can specify a number of parameters that specify which subimages you are interested in. Note that one or more rectangles in the image indicate the images you are selecting. The following parameters can be specified:

- **Number of images.** This is the number of images you want to take from the strip.
- **Images per row.** How many images of the ones you want are there per row. For example, by setting this to 1 you will select a vertical sequence of images.
- **Image width.** Width of the individual images.
- **Image height.** Height of the individual images.
- **Horizontal cell offset.** If you don't want to select the top-left images, you can set here how many images should be skipped horizontally.
- **Vertical cell offset.** Here you indicate how many images to skip vertically.
- **Horizontal pixel offset.** Sometimes there is some additional space at the left top. Here you indicate this amount (in pixels).
- **Vertical pixel offset.** Vertical amount of extra space.
- **Horizontal separation.** In some strips there are lines or empty space between the images. Here you can indicate the horizontal amount to skip between the images (in pixels).
- **Vertical separation.** Vertical amount to skip between the images.

Once you selected the correct set of images, press **OK** to create your sprite. Please remember that you are only allowed to use images created by others when you have their permission or when they are freeware.

## 13.2 Editing individual sub-images

You can also edit the individual sub-images. To this end select a sub-image and choose **Edit Image** from the **Image** menu. This will open a little built-in painting and imaging program. Please realize that this is a limited program that is mainly meant to make small changes in existing images and not to draw new ones. For that, you better use a full-blown drawing program and use files (or copy and paste) to put the image into *Game Maker*.



The form shows the image in the middle and a number of basic drawing buttons at the left. Here you can zoom in and out, draw pixels, lines, rectangles, text, etc. Note that the color depends on whether you use the left or right mouse button. For some drawing tools you can set properties (like line width or border visibility). There is a special button to change all pixels that have one color into another color. This is in particular useful to change the background color that is used for transparency. On the toolbar there are some special buttons to move all pixels in the image in a particular direction. Also you can indicate whether to show a grid when the image is zoomed (works only with a zoom factor of at least 4).

At the right of the form you can select the colors to be used (one by the left mouse button and one by the right button). There are four ways to change the color. First of all you can click with the mouse button (left or right) in one of the 16 basic colors. Note that there is a special color box that contains the color of the bottom-left pixel of the image that is used as transparency color if the sprite is transparent. You can use this color to make part of your image transparent. The second way is to click in the image with the changing color. Here you choose many more colors. You can hold down the mouse to see the color you are selecting. Thirdly, you can click with the left mouse in the boxes indicating the left and right color. A color dialog pops up in which you can select the color. Finally, you can select the color dropper tool at the left and click on a position in the image to copy the color there.

In the menus you can find the same transformation and image changing commands that are also available in the sprite editor. This time though they only apply to the current image. (When the sprite has multiple images, commands that change the size, like stretch, are not available.) You can also save the image as a bitmap file. There are two additional commands in the **Image** menu:

- **Clear.** Clear the image to the left color (which then automatically becomes the transparency color).
- **Gradient fill.** With this command you can fill the image with a gradually changing color (not very useful for making sprites, but it looks nice, and can be used for backgrounds, which use the same paint program).

Note that there is no mechanism to select parts of the image. Also some fancy drawing routines are missing. For this you should use a more advanced drawing program (or simply the paint program that comes with Windows). The easiest way to do this is to use the copy button to put the image on the clipboard. Now in your painting program, use paste to get it. Change it and copy it to the clipboard. Now, in *Game Maker* you can paste the updated image back in.

### 13.3 Advanced sprite settings

In the sprite properties form there is an **Advanced** tab. Here you find a number of advanced options that we will treat here.

First of all there are options related to collision checking. Whenever two instances meet a collision event is generated. Collisions are checked in the following way. Each sprite has a bounding box. This box is such that it contains the non-transparent part of all the subimages. When the bounding boxes do overlap, it is checked whether two pixels in the current subimages of the two sprites overlap. This second operation is expensive and requires extra memory and preprocessing. So if you are not interested in precise collision checking for a certain sprite, you should uncheck the box labeled **Precise collision checking**. In this case only bounding box checking is performed. You can also change the bounding box. This is hardly ever required but sometimes you might want to make the bounding box smaller, such that collisions with some extending parts of the sprite are not taken into account.

Sprites can be stored in two places: video memory and standard memory. Video memory is normally located on the graphics card and is faster. So if you have many instances of the sprite you prefer to store it there. But the amount of video memory is limited, depending on the graphics card the player has. So you are recommended to store large sprites not in video memory.

Some sprites you might use only in one or two levels of your game. It is a bit wasteful to keep these sprites in memory all the time. In this case you can check the box labeled **Load only on use**. The sprite is now loaded at the first moment it is required. At the end of the room it is discarded again to free the memory. For large games with many sprites it is important to carefully manage which sprites are loaded and which ones are in video memory. (You can also load and discard sprites from pieces of code.)

Finally, you can indicate the origin of the sprite. This is the point in the sprite that corresponds with its position. When you set an instance at a particular position, the origin of the sprite is placed there. Default it is the top left corner of the sprite but it is sometimes more convenient to use the center or some other important point. You can even choose an origin outside the sprite.

## Chapter 14 More about sounds and music

When you add sound resources to your game there are a number of other aspects that you can indicate. These are found under the advanced tab in the sound property form.

For all sounds you can indicate whether they should be loaded only on use. This is the default for midi files but not for wave files. If you check this box, the sound is not loaded into memory when the game starts. Only at the moment it is needed it is loaded. This might give a slight hick-up. But it saves a lot of memory and it means that loading the game is faster. Also, at the end of the room, the sound is discarded and the memory is freed. Only if it is required again is it loaded again. Don't use this for short sound effects but only for longer background music or fragments that are played only occasionally.

For wave files you can indicate the number of buffers. This number indicates the number of times the sound can play simultaneously. For example, when you have some exploding sound and a number of explosions can happen at almost the same time, you might want to increase this number such that all explosions can be heard simultaneously. Be careful though. Multiple buffers cost (depending on the sound card) more memory.

Also you can indicate whether the sound should be prepared for sound effects. These effects, like panning the sound and changing the volume, can only be used from code. Sounds that allow for sound effects take up more resources.

There is also a button here to save the sound in a file. This is useful when you lost the original sound file and want to use the sound for something else.

Besides wave files and midi files, there is actually a third kind of sound files: mp3 files. These are compressed wave files. Although you don't see them when selecting a sound file you can actually use them in *Game Maker*. First select to show all files at the bottom of the file open dialog, and you can load them. Be careful though. There are a number of disadvantages. First of all, they need to be decompressed which takes processing time and might slow down the game. The fact that the file size is smaller does not mean that they use less memory. Secondly, not all machines support them. So your game might not run on all machines. Preferably don't use them but convert your mp3 files into wave files. If you still want to use them, only use them as background music.



## Chapter 15 More about backgrounds

Besides loading them from files, you can also create your own backgrounds. To this end, press the button labeled **Edit Background**. A little built-in painting program opens in which you can create or change your background. Please realize that this is not a full-blown program. For more advanced editing tools use some paint program. The built-in paint program is described in Section 13.2 There is one option that is particularly useful. In the **Image** menu you find a command **Gradient Fill**. This can be used to create some nice gradient backgrounds.

Also the background property from has an **Advanced** tab. Here a number of advanced aspects of backgrounds are available.

Normally backgrounds are stored in video memory. This is fine when they are small but when you use large backgrounds you might want to use normal memory instead. This will be slightly slower, but video memory is limited. To this end uncheck the box labeled **Use video memory**.

Also, default backgrounds are loaded when they are needed and discarded again at the end of the room. This saves a lot of memory but will make the starting of room slightly slower and can give a little hick-up when changing the background halfway a room. To avoid this, uncheck the box labeled **Load only on use**.

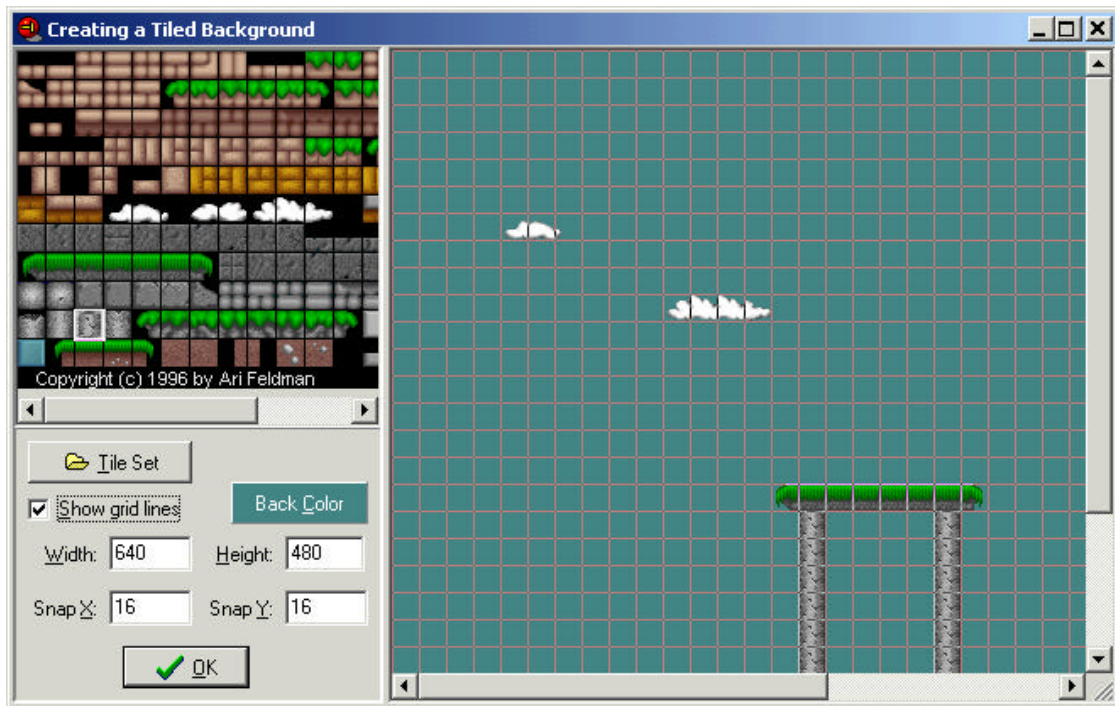
### 15.1 Creating tiled backgrounds

You can also create so-called tiled background. The reason for this is as follows: In many games you like to have nice looking backgrounds. For example, in a maze game, the walls of the maze should nicely match up, and in platform game you like to see beautifully drawn platforms, trees, etc. You can do this in *Game Maker* by defining many different objects and composing your rooms from these objects. The problem though is that this takes a lot of work, uses large amounts of resources, and makes the games slow because of the many different objects. For example, to create nice walls in maze games you already need 15 differently shaped wall objects.

The standard way out, used in many games, is that the walls and other static objects are actually drawn on the background. But, you might ask, how does the game know that an object hits a wall if it is drawn on the background only? The trick is as follows: You create just one wall object in your game. It must have the right size but it does not need to look nice. When creating the room, place this object at all places where there is a wall. And, here comes the trick, we make this object invisible. So when playing the game you don't see the wall objects. You see the beautiful background instead. But the solid wall objects are still there and the object in the game will react to them.

You can use this technique for any object that is not changing its shape or position. (You can also not use it when the object must be animated.) For platform games, you probably need just one floor and one wall object, but you can make beautifully looking backgrounds where it looks as if you walk on grass, on tree branches, etc.

To create such a tiled background, press the button labeled **Create with tiles**. The following form will show (actually, we already added some tiles in this form).



At the left top there is the current set of tiles used. You can load different tile sets by clicking on the button **Tile Set**. A number of useful sets are provided with *Game Maker*. Below the tile set you can change a number of settings. You can set the width and height of the background, the color used as background behind the tiles (which is normally used as transparency color), and how the tiles must snap to a grid.

To place a tile in the background image, select it in the tile set. Next click with your left mouse on the background, and the tile appears. Erase it (in the background color) using the right mouse button. Some special key combinations can be used:

- Holding <Ctrl> will mirror the tile horizontally
- Holding <Alt> will flip the tile vertically
- Holding <Shift> will double the size of the tile

You can use any combination of these three.

Once your background is ready, click **OK** to save it.

Often you want your tiled background to have a background again. As you should know, this can be achieved by using two backgrounds in your rooms. Make the tiled background transparent in this case. (The program will always make sure that the left bottom pixel of the tiled background has the background color, so this works correctly.)

You can add your own tile sets to *Game Maker*. A tile set is a simple bitmap file that stores all the tiles. Between any two tiles there best be a one pixel wide line. All these separator lines must have the color that is used for transparency in the tile set. (As actual transparency color the color of the lower leftmost pixel in the image is used.) The name of the tile set files is crucial. They must end in `_wwwxhhh.bmp`, where `www` is the width and `hhh` the height of the tiles in pixels. So for example, a file called `examples_32x16.bmp` contains tiles of width 32 and height 16. In some cases you might have tile images that do not have the 1-pixel separator lines. These can be used as well. In that case the file name must have the letter `n` at the end. So for example `examples_32x16n.bmp`.

## Chapter 16 More about objects

When you create an object, you can change some more advanced setting in the **Advanced** tab.

### 16.1 Depth

First of all, you can set the **Depth** of the instances of the object. When the instances are drawn on the screen they are drawn in order of depth. Instances with the largest depth are drawn first. Instances with the smallest depth are drawn last. When instances have the same depth, they are drawn in the order in which they were created. If you want to guarantee that an object lies in front of the others give it a negative depth. If you want to make sure it lies below other instances, give it a large positive depth. You can also change the depth of an instance during the game using the variable called depth.

### 16.2 Persistent objects

Secondly, you can make an object persistent. A persistent object will continue existing when you move from one room to the next. It only disappears when you explicitly destroy it. So you only need to put an instance of the object in the first room and then it will remain available in all rooms. This is great when you have e.g. a main character that moves from room to room. Using persistent objects is a powerful mechanism but also one that easily leads to errors.

### 16.3 Parents

Every object can have a parent object. When an object has a parent, it inherits the behavior of the parent. Stated differently, the object is sort of a special case of the parent object. For example, if you have 4 different balls, named ball1, ball2, ball3 and ball4, that all behave the same but have a different sprite, you can make ball1 the parent of the other three. Now you only need to specify events for ball1. The others will inherit the events and behave exactly the same way. This saves a lot of work.

Often, objects should behave almost completely the same but there will be some small differences. For example, one monster might move up and down and the other left and right. For the rest they have exactly the same behavior. In this case almost all events should have the same actions but one or two might be different. Again we can make one object the parent of the other. But in this case we also define certain events for the child object. These events "override" the parent events. So whenever an event for the child object contains actions, these are executed instead of the event of the parent. If you also want to execute the parent event you can call the so-called "inherited" event using the appropriate action.

Parent objects can again have parents, and so on. (Obviously you are not allowed to create cycles.) In this way you can create an object hierarchy. This is extremely useful to keep you game structured and you are strongly advised to learn to use this mechanism.

There is also a second use of the parent object. It also inherits the collision behavior for other objects. Let me explain this with an example. Assume you have four different floor objects. When a ball hits the floor it must change direction. This has to be specified in the collision event of the ball with the floor. Because there are four different floors we need to put the code on four different collision events of the ball. But when you make one floor the parent of the other three, you only need to specify the collision event with this one floor. The other collisions will perform the same event. Again, this saves a lot of copying.

One warning. Parents only apply when looking for events. When you specify an object at another place only instances of that object are affected. For example, in our first example, if you apply a certain action to all ball1 instances, it will not change ball2 instances, even though ball1 is the parent of ball2. The same happens in code.

## **16.4 Masks**

When two instances collide a collision event occurs. To decide whether two instances intersect, the sprites are used. This is fine in most cases, but sometimes you want to base collisions on a different shape. For example, if you make an isometric game, objects typically have a height (to give them a 3D view). But for collisions you only want to use the ground part of the sprite. This can be achieved by creating a separate sprite that is used as collision mask for the object.

## Chapter 17 More about rooms

Also the room property form has an **Advanced** tab. Here a number of advanced aspects of rooms are available.

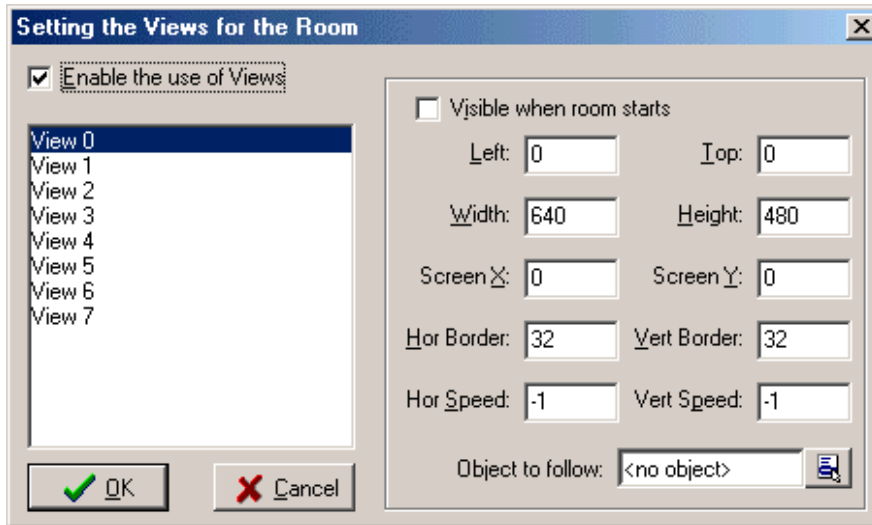
First of all, there is a checkbox labeled **Persistent**. Normally, when you leave a room and return to the same room later, the room is reset to its initial settings. This is fine if you have a number of levels in your game but it is normally not what you want in for example an RPG. Here the room should be the way you left it the last time. Checking the box labeled **Persistent** will do exactly that. The room status will be remembered and when you return to it later, it will be exactly the same as you left it. Only when you restart the game will the room be reset. Actually, there is one exception to this. If you marked certain objects as being persistent (see Chapter 16), instances of this object will not stay in the room but move to the next room.

Secondly, there are buttons in which you can sort all instances in the room horizontally or vertically. This can be important when the instances partially overlap. The order determines the drawing order (unless you changed the depth of the objects).

Also there is a button labeled **Creation code**. Here you can type in a piece of code in GML (see later) that is executed when the room is created. This is useful to e.g. fill in certain variables for the room, create certain instances, etc. It is useful to understand what exactly happens when you move to a particular room in the game. First the room and all its instances are generated (when the room is not persistent or has not been visited before). Next, when this is the first room, the start game events are executed for all instances. Next (when the room is not persistent or not visited before) the room creation code is executed. Next the start room events are executed for all instances. And finally, for the created instances the creation event is executed. So, for example, the creation events can use variables set by the creation code for the room and in the creation code you can refer to the instances in the room.

Finally, there is a button labeled **Views**. This gives a mechanism of drawing different parts of your room at different places on the screen. There are many uses for views. First of all, in a number of games you want to show only part of the room at any time. For example, in most platform games, the view follows the main character. In two-player games you often want a split-screen mode in which in one part of the screen you see one player and in another part you see the other player. A third use is in games in which part of the room should scroll with e.g. the main character while another part is fixed (for example some status panel). This can all be easily achieved in *Game Maker*.

When you click the button labeled **Views** a rather empty dialog pops up. Enable the checkbox labeled **Enable the use of views** and the following information will show:



At the left you see the list of at most eight views you can define. At the right you can give information for the views. First of all you must indicate whether the view should be visible when the room starts. Make sure at least one view is visible. Visible views are shown in bold. Next you indicate the area of the room that should be shown in the view. You specify the left and top position, and the width and the height of the view. Below that you indicate the position of the view on the screen.

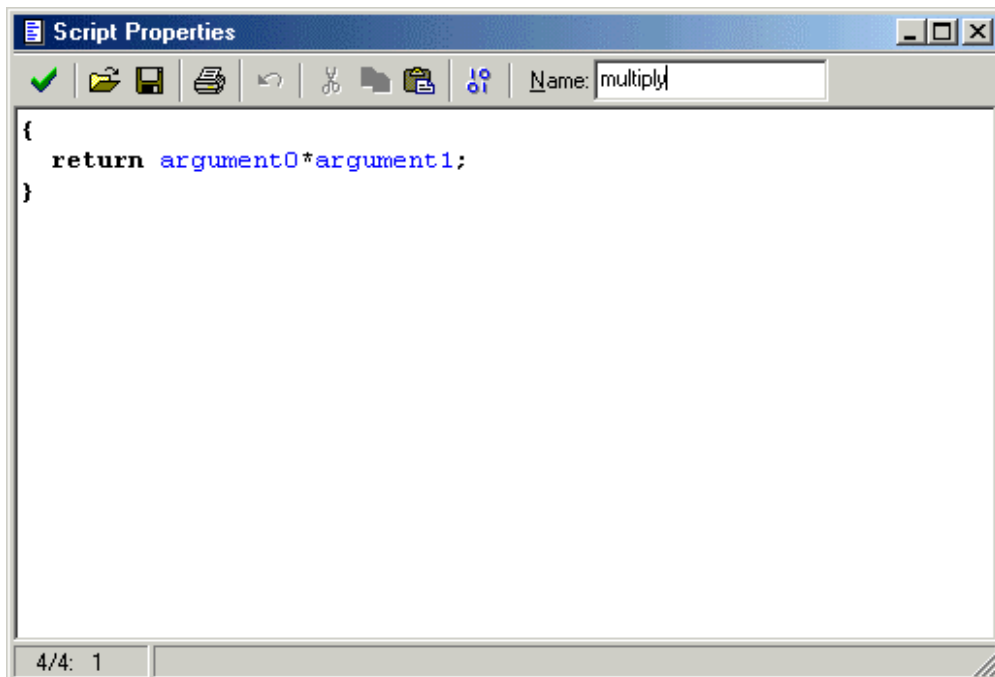
As indicated above, you often want the view to follow a certain object. This object you can indicate at the bottom. If there are multiple instances of this object, only the first one is followed by the view. Normally the character should be able to walk around a bit without the view changing. Only when the character gets close to the boundary of the view, should the view change. You can specify the size of the border that must remain visible around the object. Finally, you can restrict the speed with which the view changes. This might mean that the character can walk off the screen, but it gives a much smoother game play. Use  $-1$  if you want the view to change instantaneously.

## Chapter 18 Scripts

*Game Maker* has a built-in programming language. Once you get more familiar with *Game Maker* and want to use it to its fullest extend, it is advisable to start learning to use this language. For a complete description see Chapter 23. There are two ways to use the language. First of all you can create scripts. These are pieces of code that you give a name. They are shown in the resource tree and can be save to a file and loaded from a file. They can be used to form a library that extends the possibilities of *Game Maker*. Alternatively, you can add a code action to some event and type a piece of code there. Adding code actions works in exactly the same way as adding scripts except for two differences. Code actions don't have a name and cannot use arguments. Also they have the well-known field to indicate to what objects the action should apply. For the rest you enter code in exactly the same way as in scripts. So we further concentrate on scripts in this chapter.

As stated before, a script is a piece of code in the built-in programming language that performs a particular task. A script can take a number of arguments. To execute a script from within some event, you can use the script action. In this action you specify the script you want to execute, together with the up to three arguments. (You can also execute scripts from within a piece of code in the same way you call a function. In that case you can use up to 10 arguments.) When the script returns a value, you can also use it as a function when providing values in other actions.

To add a script to your game, choose **Add Script** from the **Add menu**. The following form will pop up (in the example we already added a little script that computed the product of the two arguments).





At the top right you can indicate the name of the script. You have a little editor in which you can type the script. There are a few buttons for cut and paste and for loading or saving the script as a text file. Finally there is a button with which you can test whether the script is correct. Not all aspects can be tested at this stage but the syntax of your script will be tested, together with the existence of functions used.

As you might have noticed, parts of the script text are colored. The editor knows about existing objects, built-in variables and functions, etc. Color-coding helps a lot in avoiding mistakes. In particular, you see immediately if you misspelled some name or use a keyword as a variable. Color-coding is though a bit slow. In the preferences in the file menu you can switch color-coding on and off. Here you can also change the color for the different components of the programs. Also you can change the font used in scripts and code.

Scripts are extremely useful to extend the possibilities of *Game Maker*. This does though require that you design your scripts careful. Scripts can be stored in libraries that can be added to your game. To import a library, use the item **Import scripts** from the file menu. To save your scripts in the form of a library use **Export scripts**. Script libraries are simple text files (although they have the extension .gml). Preferably don't edit them directly because they have a special structure. Some libraries with useful scripts are included. (To avoid unnecessary work when loading the game, after importing a library, best delete those scripts that you don't use.)

When creating scripts you easily make mistakes. Always test the scripts by using the appropriate button. When an error occurs during the execution of a script this is reported, with an indication of the type of error and the place. If you need to check things more carefully, you can run the game in debug mode. Now a form appears in which you can monitor certain information in your game. Using the **Add** button you can type in some expression whose value is shown in each step of the game. In this way you can see whether your game is doing things the right way. Also you can pause your game, perform it step by step, and restart it. Furthermore, an indication of the speed in frames per second is given.

## Chapter 19 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 the game information, double click **Game Information** in the resource tree at the left of the screen. 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.

## Chapter 20 Game options

There are a number of options you can change for your game. They can be found by double clicking on **Game Options** in the resource tree at the left of the screen. They are subdivided in a number of tabbed pages.

### 20.1 Graphics options

#### Start in fullscreen mode

When checked the game runs in the full screen; otherwise it runs in a window.

#### Scale the image in fullscreen mode

When checked, the image is scaled to fill the whole screen as much as possible.

Otherwise, it is centered on the screen. Scaling is slower but most modern graphics cards can do this with little overhead. But make sure the screen is larger than the window because scaling down is normally very slow.

#### Use exclusive graphics mode

In exclusive mode, the game has the full control over the screen. No other applications can use it anymore. As a result you can set the resolution of the screen and then color depth. It also makes the graphics often a bit faster. If you want to make sure the computer of the player has the right screen resolution, you best use exclusive mode. A warning is in place though. In exclusive mode no other windows can show. This e.g. means that you cannot use actions that display a message, ask a question, show the highscore list, or show the game information. Also no errors can be reported. In general, when something goes wrong in exclusive mode, the player has no other option than to restart the computer. So make sure your game works absolutely correct. You cannot run your game in debug mode when using exclusive mode.

#### Wait for vertical blank before drawing

The screen of your computer is refreshed a number of times per second (normally between 50 and 100). After refreshing the screen there is a so-called vertical blank in which nothing happens on the screen. If you draw the screen continuously, part of one image and part of the next might show on the screen, which can give a poor visual effect. If you wait for the vertical blank before drawing the next frame, this problem disappears. The disadvantage is that the program must wait for the vertical blank which will slow it down slightly.

#### Display the cursor

Indicates whether you want the mouse pointer to be visible. Turning it off is normally faster and nicer. (You can easily make you own cursor object in *Game Maker*.)

#### Display the caption in fullscreen mode

When checked, in fullscreen mode a little white box is drawn at the left top, displaying the room caption, the score and the number of lives. You can switch this off here. It is normally nicer if you draw these things yourself at an appropriate place in your rooms.

## 20.2 Key options

### **Let <Esc> end the game**

When checked, pressing the escape key will end the game. More advanced games normally don't want this to happen because they might want to do some processing (like saving) before ending the game. In this case, uncheck this box and provide your own actions for the escape key. (Clicking on the cross of the window will also generate an escape key event.)

### **Let <F1> show the game information**

When checked pressing the F1 key will display the game information (not in exclusive mode).

### **Let <F4> switch between screen modes**

When checked the F4 key will switch between fullscreen and windowed mode (not in exclusive mode).

### **Let <F5> and <F6> load and save the game**

When checked the player can use <F5> to store the current game situation and <F6> to load the last saved game.

## 20.3 Interaction options

### **Generate continuous keyboard events**

When checked, keyboard events are created in each step as long as a key is pressed. When unchecked the event is only generated the moment the key is pressed (and once it starts repeating). For different types of games you need a different setting here.

### **Generate continuous mouse events**

When checked mouse events are created in each step as long as a mouse button is pressed. When unchecked the event is only generated once when the button is pressed.

### **Freeze the game when the form loses focus**

When checked, whenever the player brings some other form to the top (e.g. another application) the game freezes until the game window again gets the focus.

## 20.4 Score options

### **Show the score**

When checked, the score is shown in the caption. Also the number of lives is shown in the caption.

The other options deal with the appearance of the highscore list. First of all you can give the highscore list a background image (preferably of size 360x400 but it will be stretched otherwise). Alternatively, you can specify the background color. Next you can indicate the color used for the text and color used for the new item when the player has a new highest score. You can also indicate the font for the text and finally you can indicate whether the form should have a border and a caption or not. Using the correct settings will lead to appealing highscore lists.

## 20.5 Loading options

Here you can indicate what should happen when loading a game. First of all you can specify your own loading image. Secondly, you can indicate whether to display a loading progress bar at the bottom of the image. You have three options here. Either no loading bar is displayed, or the default bar is displayed or you can specify two images: the background of the loading bar and the foreground. They will be scaled to obtain the correct size. (Note that both images must be specified in this case, not just one.)

Secondly, you can indicate here the icon that should be used for stand-alone games. You can only use 16-color 32x32 icons. If you try to select another type of icon you will get a warning.

Finally you can change the unique game id. This id is used for storing the highscore list and save game files. If you release a new version of your game and don't want to take over the old highscore list, you should change this number.

## Chapter 21 Speed considerations

If you are making complicated games you probably want to make them run as fast as possible. Even though *Game Maker* does its best to make games run fast, a lot depends on how you design your game. Also, it is rather easy to make games that use large amounts of memory. In this chapter I will give some hints on how to make your games faster and smaller.

First of all, look carefully at the sprites and backgrounds you use. Animated sprites take a lot of memory and drawing lots of sprites takes a lot of time. So make your sprites as small as possible. Remove any invisible area around it (there is a command for this in the sprite editor). Carefully decide which sprites to store in video memory and which ones to only load on use. The same applies to background images. In general you want to load them on use and, in particular when they are large, you don't want to store them in video memory. If you have a covering background, make sure you switch off the use of a background color.

If you use full screen mode or exclusive mode, make sure the size of the room (or window) is never larger than the screen size. Most graphics card can efficiently scale images up but they are very slow in scaling images down! Also, preferably draw as little other things than sprites. This is slow. If you do need them, preferably draw them immediately after each other. Finally, whenever possible, switch off the cursor. It slows down the graphics.

Also be careful with the use of many views. For each view the room is redrawn.

Besides the graphics, there are also other aspects that influence the speed. Make sure you have as few instances as possible. In particular, destroy instances once they are no longer required (e.g. when they leave the room). Avoid lots of work in the step event or drawing event of instances. Often things do not need to be checked in each step. Interpretation of code is reasonably fast, but it is interpreted. Also, some functions and actions take a lot of time, in particular those that have to check all instances (like for example the bounce action).

Think about where to treat the collision events. You normally have two options. Objects that have no collision events at all are treated much faster, so preferably treat them in those objects for which there are just a few instances.

Be careful with using large sound files. They take a lot of memory and also compress badly. You might want to check your sounds and see whether you can sample them down.

Finally, if you want to make a game that many people can play, make sure you test it on small machines.

## Chapter 22 Distributing your game

Once you have created a nice game you probably want to give it to other people to play. You are free to distribute your games in any way you like. You can even sell them. See the enclosed license agreement for more information.

There are basically three different ways in which you can distribute you games. The easiest way is to simply give people the \*.gmd file that holds the game. This does though mean that the other person must have *Game Maker*. (You are not allowed to distribute *Game Maker* with your game but they can freely download it from the site.) Also the other person can change the game.

The second way is to create a stand-alone version of the game. A stand-alone version is simply one executable. This can be achieved by choosing the item **Create stand-alone** in the **File** menu. You will be asked for the name of the executable that should contain the game. Indicate a name, press **OK** and you have your stand-alone game that you can give to anyone you like. You can set the icon for the stand-alone game in the Options form. (If your game uses any other files you should copy them to the folder containing the stand-alone game.) Now you should give this file to the other people (You might want to zip it first.)

The third way is to create an installer. A number of freeware installer programs are available on the web. Again you first make a stand-alone version and then you use the installer to create an installation. How to do this depends on the installer you use.

If you created a great game, please mail it to [gamemaker@wanadoo.nl](mailto:gamemaker@wanadoo.nl) such that I might place it on the web site. Please do not send install programs, only .gmd files or stand-alone versions.

## Chapter 23 The Game Maker Language (GML)

As you have read before, *Game Maker* contains a built-in programming language. This programming language gives you much more flexibility and control than the standard actions. This language we will refer to as GML (the Game maker Language). There are three different places where you can type programs in this language. First of all, when you define scripts. A script is a program in GML. Secondly, when you add a code action to an event. In a code action you again have to provide a program in GML. Finally, wherever you need to specify a value in an action, you can also use an expression in GML. An expression, as we will see below is not a complete program, but a piece resulting in a value.

In this chapter I will describe the basic structure of programs in GML. When you want to use programs in GML, there are a couple of things you have to be careful about. First of all, for all your resources (sprites, objects, sounds, etc.) 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 program. Also be careful not to name resources self, other, global, or all because these have special meaning in the language. Also you should not use any of the keywords, indicated below.

### 23.1 A program

A program consists of a block. A block consists of one or more statements, enclosed by the symbols ‘{’ 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.

### 23.2 Variables

Like any programming language GML contains 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 `mouse_x` and `mouse_y` that indicate the current mouse position, while all others are local to the object instance for which we execute the program, 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 ‘\_’. (The maximal length is 64 symbols.) When you use a new variable it is local to the current instance and is not known in programs for other instances (even of the same object). You can though refer to variables in other instances; see below.



## 23.3 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.)

## 23.4 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 binary operators exist (in order of priority):

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

Also, the following unary operators exist:

- `!`: not, turns true into false and false into true
- `-`: negates the next value

As values you can use numbers, 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);
}
```

## 23.5 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;
  }
}
```

## 23.6 Addressing variables in other instances

As described above, you can set variables in the current instance using statements like

```
x = 3;
```

But in a number of cases you want to address variables in another instance. For example, you might want to stop the motion of all balls, or you might want to move the main character to a particular position, or, in the case of a collision, you might want to set the sprite for the other instance involved. This can be achieved by preceding the variable name with the name of an object and a dot. So for example, you can write

```
ball.speed = 0;
```

This will change the speed of all instances of object `ball`. There are a number of special "objects".

- `self`: The current instance for which we are executing the action
- `other`: The other instance involved in a collision event
- `all`: All instances
- `noone`: No instance at all (sounds weird probably but it does come in handy as we will see later on)
- `global`: Not an instance at all, but a container that stores global variables

So, for example, you can use the following kind of statements:

```
other.sprite_index = sprite5;
all.speed = 0;
global.message = 'A good result';
global.x = ball.x;
```

Now you might wonder what the last assignment does when there are multiple balls. Well, the first one is taken and its `x` value is assigned to the global value.

But what if you want to set the speed of one particular ball, rather than all balls. This is slightly more difficult. Each instance has a unique id. When you put instances in a room in the designer, this instance id is shown when you rest the mouse on the instance. These

are numbers larger than or equal to 100000. Such a number you can also use as the left-hand side of the dot. But be careful. The dot will get interpreted as the decimal dot in the number. To avoid this, put brackets around it. So for example, assuming the id of the ball is 100032, you can write:

```
(100032).speed = 0;
```

When you create an instance in the program, the call returns the id. So a valid piece of program is

```
{
    nnn = instance_create(100,100,ball);
    nnn.speed = 8;
}
```

This creates a ball and sets its speed. Note that we assigned the instance id to a variable and used this variable as indication in front of the dot. This is completely valid. Let me try to make this more precise. A dot is actually an operator. It takes a value as left operand and a variable (address) as right operand, and returns the address of this particular variable in the indicated object or instance. All the object names, and the special objects indicated above simply represent values and these can be dealt with like any value. For example, the following program is valid:

```
{
    obj[0] = ball;
    obj[1] = flag;
    obj[0].alarm[4] = 12;
    obj[1].id.x = 12;
}
```

The last statement should be read as follows. We take the id of the first flag. For the instance with that id we set the x coordinate to 12.

Object names, the special objects, and the instance id's can also be used in a number of functions.

## 23.7 Arrays

You can use 1- and 2-dimensional arrays in GML. Simply put the index between square brackets for a 1-dimensional array, and the two indices with a comma between them for 2-dimensional arrays. At the moment you use an index the array is generated. Each array runs from index 0. So be careful with using large indices because memory for a large array will be reserved. Never use negative indices. The system puts a limit of 32000 on each index and 1000000 on the total size. So for example you can write the following:

```
{
    a[0] = 1;
    i = 1;
    while (i < 10) { a[i] = 2*a[i-1]; i += 1;}
    b[4,6] = 32;
}
```

```
}
```

## 23.8 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. It is a good habit to always put curly brackets around the statements in the if statement. So best use

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

### Example

The following program moves the object toward the middle of the screen.

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

## 23.9 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 program creates five balls at random positions.

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

## 23.10 While statement

A while statement has the form

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

As long as the expression is true, the statement (which can also be a block) is executed. Be careful with your while loops. You can easily make them loop forever, in which case your game will hang and not react to any user input anymore.

### Example

The following program 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 (!place_free(x,y))
  {
    x = random(room_width);
    y = random(room_height);
  }
}
```

## 23.11 For statement

A for statement has the form

```
for (<statement1> ; <expression> ; <statement2>) <statement3>
```

This works as follows. First statement1 is executed. Then the expression is evaluated. If it is true, statement 3 is executed; then statement 2 and then the expression is evaluated again. This continues until the expression is false.

This may sound complicated. You should interpret this as follows. The first statement initializes the for-loop. The expression tests whether the loop should be ended. Statement2 is the step statement that goes to the next loop evaluation.

The most common use is to have a counter run through some range.

### Example

The following program initializes an array of length 10 with the values 1-10.

```
{
  for (i=0; i<9; i+=1) list[i] = i+1;
}
```

## 23.12 Exit statement

The exit statement has the form

```
exit
```

It simply ends the execution of this program. (It does not end the execution of the game! For this you need the function `game_end()`; see below.)

## 23.13 Functions

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

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

There are two types of functions. First of all, there is a huge collection of built-in functions, to control all aspects of your game. Secondly, any script you define in your game can be used as a function.

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.

## 23.14 Scripts

When you create a script, you want to access the arguments passed to it (either when using the script action, or when calling the script as a function from a program (or from another, or even the same script)). These arguments are stored in the variables `argument0`, `argument1`, ..., `argument9`. So there can be at most 10 arguments. (Note that when calling the script from an action, only the first 3 arguments can be specified.)

Scripts can also return a value, such that they can be used in expressions. For this end you use the return statement:

```
return <expression>
```

Execution of the script ends at the return statement!

### Example

Here is the definition for a little script that computes the square of the argument:

```
{  
    return (argument0*argument0);  
}
```

To call a script from within a piece of code, just act the same way as when calling functions. That is, write the script name with the argument values in parentheses.

## 23.15 With constructions

As indicated before, it is possible to read and change the value of variables in other instances. But in a number of cases you want to do a lot more with other instances. For example, imagine that you want to move all balls 8 pixels down. You might think that this is achieved by the following piece of code

```
ball.y = ball.y + 8;
```

But this is not correct. The right side of the assignment gets the value of the y-coordinate of the first ball and adds 8 to it. Next this new value is set as y-coordinate of all balls. So the result is that all balls get the same y-coordinate. The statement

```
ball.y += 8;
```

will have exactly the same effect because it is simply an abbreviation of the first statement. So how do we achieve this? For this purpose there is the **with** statement. Its global form is

```
with (<expression>) <statement>
```

<expression> indicates one or more instances. For this you can use an instance id, the name of an object (to indicate all instances of this object) or one of the special objects (all, self, other, noone). <statement> is now executed for each of the indicated instances, as if that instance is the current (self) instance. So, to move all balls 8 pixels down, you can type.

```
with (ball) y += 8;
```

If you want to execute multiple statements, put curly brackets around them. So for example, to move all balls to a random position, you can use

```
with (ball)
{
    x = random(room_width);
    y = random(room_height);
}
```

Note that, within the statement(s), the indicated instance has become the self instance. Within the statements the original self instance has become the other instance. So for example, to move all balls to the position of the current instance, you can type

```
with (ball)
{
    x = other.x;
    y = other.y;
}
```

Use of the with statement is extremely powerful. Let me give a few more examples. To destroy all balls you type

```
with (ball) instance_destroy();
```

If a bomb explodes and you want to destroy all instances close by you can use

```
with (all)
{
    if (distance_to_object(other) < 50) instance_destroy();
}
```

## 23.16 Comment

You can add comment to your programs. Everything on a line after `//` is not read. It is used internally to indicate an end of line.

## 23.17 Pascal style

The interpreter is actually pretty relaxed. You can also use programs that look 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 program is also valid:

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

## 23.18 Functions and variables in GML

GML contains a large number of built-in functions and variables. With these you can control any part of the game. For all actions there are corresponding functions so you actually don't need to use any actions if you prefer using code. But there are many more functions and variables that control aspects of the game that cannot be used with actions only. So if you want to make advanced games you are strongly advised to read through the following chapters to get an overview of all that is possible. Please note that these variables and functions can also be used when providing values for actions. So even if you don't plan on using code or writing scripts, you will still benefit from this information.

The following convention is used below. Variable names marked with a `*` are read-only, that is, their value cannot be changed. Variable names with `[0..n]` after them are arrays. The range of possible indices is given.



## Chapter 24 Computing things

*Game Maker* contains a large number of functions to compute certain things. Here is a complete list.

### 24.1 Constants

The following constants exist:

**true** Equal to 1.  
**false** Equal to 0.  
**pi** Equal to 3.1415...

### 24.2 Real-values functions

The following functions exist that deal with real numbers.

**random(x)** Returns a random real number between 0 and x. The number is always smaller than 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, that is, x rounded down to an integer.  
**ceil(x)** Returns the ceiling of x, that is, x rounded up to an integer.  
**frac(x)** Returns the fractional part of x, that is, the part behind the decimal dot.  
**sqrt(x)** Returns the square root of x. x must be non-negative.  
**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).  
**arcsin(x)** Returns the inverse sine of x.  
**arccos(x)** Returns the inverse cosine of x.  
**arctan(x)** Returns the inverse tangent of x.  
**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.  
**min3(x,y,z)** Returns the minimum of x, y and z.  
**max3(x,y,z)** Returns the maximum of x, y and z.  
**mean(x,y)** Returns the average of x and y.

**point\_distance(x1,y1,x2,y2)** Returns the distance between point (x1,y1) and point (x2,y2).  
**point\_direction(x1,y1,x2,y2)** Returns the direction from point (x1,y1) toward point (x2,y2) in degrees.  
**is\_real(x)** Returns whether x is a real value (as opposed to a string).  
**is\_string(x)** Returns whether x is a string (as opposed to a real value).

## 24.3 String handling functions

The following functions deal with characters and string.

**chr(val)** Returns a string containing the character with ascii code val.  
**ord(str)** Returns the ascii code of the first character in str.  
**real(str)** Turns str into a real number. str can contain a minus sign, a decimal dot and even an exponential part.  
**string(val)** Turns the real value into a string using a standard format (no decimal places when it is an integer, and two decimal places otherwise).  
**string\_format(val,tot,dec)** Turns val into a string using your own format: tot indicates the total number of places and dec indicated the number of decimal places.  
**string\_length(str)** Returns the number of characters in the string.  
**string\_pos(substr,str)** Returns the position of substr in str (0=no occurrence).  
**string\_copy(str,index,count)** Returns a substring of str, starting at position index, and of length count.  
**string\_delete(str,index,count)** Returns a copy of str with the part removed that starts at position index and has length count.  
**string\_insert(substr,str,index)** Returns a copy of str with substr added at position index.  
**string\_lower(str)** Returns a lowercase copy of str.  
**string\_upper(str)** Returns an uppercase copy of str.  
**string\_repeat(str,count)** Returns a string consisting of count copies of str.  
**string\_letters(str)** Returns a string that only contains the letters in str.  
**string\_digits(str)** Returns a string that only contains the digits in str.  
**string\_lettersdigits(str)** Returns a string that contains the letters and digits in str.

## Chapter 25 GML: Game play

There are a large number of variables and functions that you can use to define the game play. These in particular influence the movement and creation of instances, the timing, and the handling of events.

### 25.1 Moving 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 instance. (To be precise, they indicate the place where the origin of the sprite is placed. Position (0,0) is the top-left corner of the room. You can change the position of the 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. A positive horizontal speed means a motion to the right, a negative horizontal speed mean a motion to the left. Positive vertical speed is downwards and negative vertical speed is upwards. So you have to set these variables only once (for example in the creating event) to give the object instance a constant motion.

There is quit a different way for specifying motion, using a direction (in degrees 0-359), and a speed (should be non-negative). You can set and read these variables to specify an arbitrary motion. (Internally this is changed into values for `hspeed` and `vspeed`.) Also there is the friction and the gravity and gravity direction. Finally, there is the function `motion_add(dir, speed)` to add a motion to the current one.

To summarize, each instance has the following variables, and functions:

- x** Its x-position.
- y** Its y-position.
- xprevious** Its previous x-position.
- yprevious** Its previous y-position.
- xstart** Its starting x-position in the room.
- ystart** Its starting y-position in the room.
- hspeed** Horizontal component of the speed.
- vspeed** Vertical component of the speed.
- direction** Its current direction (0-360, counter-clockwise, 0 = to the right).
- speed** Its current speed (pixels per step).
- friction** Current friction (pixels per step).
- gravity** Current amount of gravity (pixels per step).
- gravity\_direction** Direction of gravity (270 is downwards).
- motion\_set(dir, speed)** Sets the motion with the given speed in direction `dir`.

**motion\_add(dir,speed)** Adds the motion to the current motion (as a vector addition).

There are a large number of functions available that help you in defining your motions:

**place\_free(x,y)** Returns whether the instance placed at position(x,y) is collision-free. This is typically used as a check before actually moving to the new position.

**place\_empty(x,y)** Returns whether the instance placed at position (x,y) meets nobody. So this function takes also non-solid instances into account.

**place\_meeting(x,y,obj)** Returns whether the instance placed at position (x,y) meets obj. obj can be an object in which case the function returns true is some instance of that object is met. It can also be an instance id, the special word `all` meaning an instance of any object, or the special word `other`.

**place\_snapped(hsnap,vsnap)** Returns whether the instance is aligned with the snapping values.

**move\_random(hsnap,vsnap)** Move the instance to a free random, snapped position, like the corresponding action.

**move\_snap(hsnap,vsnap)** Snap the instance, like the corresponding action.

**move\_towards\_point(x,y,sp)** Moves the instances with speed sp toward position (x,y).

**move\_bounce\_solid(adv)** Bounce against solid instances, like the corresponding action. adv indicates whether to use advance bounce, that also takes slanted walls into account.

**move\_bounce\_all(adv)** Bounce against all instances, instead of just the solid ones.

**move\_contact(dir)** Move the instance in the direction until a contact position is reached. If there is no collision at the current position, the instance is placed just before a collision occurs. If there already is a collision, the instance is moved to the first position where there is no collision anymore.

**distance\_to\_point(x,y)** Returns the distance of the bounding box of the current instance to (x,y).

**distance\_to\_object(obj)** Returns the distance of the instance to the nearest instance of object obj.

**position\_empty(x,y)** Returns whether there is nothing at position (x,y).

**position\_meeting(x,y,obj)** Returns whether at position (x,y) there is an instance obj. obj can be an object, an instance id, or the keywords `self`, `other`, or `all`.

## 25.2 Instances

In the game, the basic units are the instances of the different objects. During game play you can change a number of aspects of these instances. Also you can create new instances

and destroy instances. Besides the movement related variables discussed above and the drawing related variables discussed below, each instance has the following variables:

**object\_index\*** Index of the object this is an instance of. Cannot be changed.  
**id\*** The unique identifier for the instance ( $\geq 100000$ ). (Note that when defining rooms the id of the instance under the mouse is always indicated.)  
**mask\_index** Index of the sprite used as mask for collisions. Give this a value of -1 to make it the same as the `sprite_index`.  
**solid** Whether the instance is solid. This can be changed during the game.  
**persistent** Whether the instance is persistent and will reappear when moving to another room. You often want to switch persistence off at certain moments. (For example if you go back to the first room.)

There is one problem when dealing with instances. It is not so easy to identify individual instances. They don't have a name. When there is only one instance of a particular object you can use the object name but otherwise you need to get the id of the instance. This is a unique identifier for the instance. you can use it in **with** statements and as object identifier (using the dot construction described in section 23.6). Fortunately there are a number of variables and routines that help you locate instance id's.

**instance\_count\*** Number of instances that currently exist in the room.  
**instance\_id[0..n-1]\*** The id of the particular instance. Here n is the number of instance.

Let me give an example. Assume each unit in your game has a particular health and you want to locate the strongest one, you could use the following code:

```
{
  maxid = -1;
  maxhealth = 0;
  for (i=0; i<instance_count; i+=1)
  {
    iii = instance_id[i];
    if (iii.object_index == unit)
    {
      if (iii.health > maxhealth)
      {maxid = iii; maxhealth = iii.health;}
    }
  }
}
```

After the loop `maxid` will contain the id of the unit with largest health. (Don't destroy instances during such a loop because they will automatically be removed from the array and as a result you will start skipping instances.)

**instance\_find(obj,n)** Returns the id of the (n+1)'th instance of type `obj`. `obj` can be an object or the keyword `all`. If it does not exist, the special object `noone` is returned.

**instance\_exists(obj)** Returns whether an instance of type obj exists. obj can be an object, an instance id, or the keyword all.

**instance\_number(obj)** Returns the number of instances of type obj. obj can be an object or the keyword all.

**instance\_position(x,y,obj)** Returns the id of the instance of type obj at position (x,y). When multiple instances are at that position the first is returned. obj can be an object or the keyword all. If it does not exist, the special object `noone` is returned.

**instance\_nearest(x,y,obj)** Returns the id of the instance of type obj nearest to (x,y). obj can be an object or the keyword all.

**instance\_furthest(x,y,obj)** Returns the id of the instance of type obj furthest away from (x,y). obj can be an object or the keyword all.

**instance\_place(x,y,obj)** Returns the id of the instance of type obj met when the current instance is placed at position (x,y). obj can be an object or the keyword all. If it does not exist, the special object `noone` is returned.

The following functions can be used for creating and destroying instances.

**instance\_create(x,y,obj)** Creates an instance of obj at position (x,y). The function returns the id of the new instance.

**instance\_destroy()** Destroys the current instance.

**instance\_change(obj,perf)** Changes the instance into obj. perf indicates whether to perform the destroy and creation events.

**position\_destroy(x,y)** Destroy all instances whose sprite contains position (x,y).

**position\_change(x,y,obj,perf)** Change all instances at (x,y) into obj. perf indicates whether to perform the destroy and creation events.

## 25.3 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 `room_speed`. 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 `room_speed` in every step. If your machine is slow the game speed might not be achieved. This can be checked using the variable `fps` that constantly monitors the actual number of frames per second. Finally, for some advance timing you can use the variable `current_time` that gives the number of milliseconds since the computer was started. Here is the total collection of variables available (only the first one can be changed):

**room\_speed** Speed of the game in the current room (in steps per second).

**fps\*** Number of frames that are actually drawn per second.

**current\_time\*** Number of milliseconds that have passed since the system was started.

**current\_year\*** The current year.

**current\_month\*** The current month.  
**current\_day\*** The current day.  
**current\_weekday\*** The current day of the week (1=sunday, ..., 7=saturday).  
**current\_hour\*** The current hour.  
**current\_minute\*** The current minute.  
**current\_second\*** The current second.

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

**sleep(numb)** Sleeps numb milliseconds.

Finally, as you should know, every instance has 8 different alarm clocks that you can set. To change the values (or get the values) of the different alarm clocks use the following variable:

**alarm[0..7]** Value of the indicated alarm clock. (Note that alarm clocks only get updated when the alarm event for the object contains actions!)

## 25.4 Rooms and score

Games work in rooms. Each room has an index that is indicated by the name of the room. The current room is stored in variable `room`. You cannot assume that rooms are numbered in a consecutive order. So never add or subtract a number from the room variable. Instead use the functions and variables indicated below. So a typical piece of code you will use is:

```

{
  if (room != room_last)
  {
    room_goto_next();
  }
  else
  {
    game_end();
  }
}

```

The following variables and functions exist that deal with rooms.

**room** Index of the current room; can be changed to go to a different room, but you better use the routines below.  
**room\_first\*** Index of the first room in the game.  
**room\_last\*** Index of the last room in the game.  
**room\_goto(numb)** Goto the room with index numb.  
**room\_goto\_previous()** Go to the previous room.  
**room\_goto\_next()** Go to the next room.

**room\_restart()** Restart the current room.  
**room\_previous(numb)** Return the index of the room before numb (-1 = none) but don't go there.  
**room\_next(numb)** Return the index of the room after numb (-1 = none).  
**game\_end()** End the game.  
**game\_restart()** Restart the game.

Rooms have a number of additional properties:

**room\_width\*** Width of the room in pixels.  
**room\_height\*** Height of the room in pixels.  
**room\_caption** Caption string for the room that is displayed in the caption of the window.  
**room\_persistent** Whether the current room is persistent.

Many games offer the player the possibility to save the game and load a saved game. In *Game Maker* this happens automatically when the player press <F5> for saving and <F6> for loading. You can also save and load games from within a piece of code (note that loading only takes place at the end of the current step).

**game\_save(string)** Save the game to the file with name string.  
**game\_load(string)** Load the game from the file with name string.

Another important aspect of many games is the score and the number of lives. *Game Maker* keeps track of the score in a global variable `score` and the number of lives in a global variable `lives`. Both are normally shown in the window caption. You can change the score by simply changing the value of this variable. The same for lives. If `lives` is larger than 0 and becomes smaller than or equal to 0 the no-more-lives event is performed for all instances. If you don't want to show the score and lives in the caption, set the variable `show_score` to false. (You can also change this in the options form.) For more complicated games you better display the score yourself.

**score** The current score.  
**lives** Number of lives.  
**show\_score** Whether to show the score (and lives).

There is also a built-in mechanism to keep track of a highscore list. It can contain up to ten names. For more information, see Chapter 29.

## 25.5 Generating events

As you know, *Game Maker* is completely event driven. All actions happen as the result of events. There are a number of different events. Creation and destroy events happen when an instance is created or destroyed. In each step, the system first handles the alarm events. Next it handles keyboard and mouse events and next the step event. After this the instances are set to their new positions after which the collision event is handled. Finally



the draw event is used to draw the instances (note that when there are multiple views the draw event is called multiple times in each step). You can also apply an event to the current instance from within a piece of code. The following functions exist:

**event\_perform(type,numb)** Performs event numb of type type to the current instance. The following event types can be indicated:

- ev\_create**
- ev\_destroy**
- ev\_step**
- ev\_alarm**
- ev\_keyboard**
- ev\_mouse**
- ev\_collision**
- ev\_other**
- ev\_draw**

When there are multiple events of the given type, numb can be used to specify the precise event. For the alarm event numb can range from 0 to 7. For the keyboard event you have to use the keycode for the key (see 0). For mouse events you can use the following constants:

- ev\_left\_button**
- ev\_right\_button**
- ev\_middle\_button**
- ev\_no\_button**

For the collision event you give the index of the other object. Finally, for the other event you can use the following constants:

- ev\_outside**
- ev\_boundary**
- ev\_game\_start**
- ev\_game\_end**
- ev\_room\_start**
- ev\_room\_end**
- ev\_no\_more\_lives**
- ev\_animation\_end**

For the step event you give the index can use the following constants:

- ev\_step\_normal**
- ev\_step\_begin**
- ev\_step\_end**

**event\_perform\_object(obj,type,numb)** This functions works the same as the function above except that this time you can specify events in another object.

Note that the actions in these events are applied to the current instance, not to instances of the given object..

**event\_user(numb)** In the other events you can also define 8 user events. These are only performed if you call this function numb must lie in the range 0 to 7.

**event\_inherited()** Performs the inherited event. This only works if the instance has a parent object.

You can get information about the current event being executed using the following read-only variables:

**event\_type\*** Type of the current event begin executed.

**event\_number\*** Number of the current event begin executed.  
**event\_object\*** The object index for which the current event is being executed.  
**event\_action\*** The index of the action that is currently being executed (0 is the first in the event, etc.).

## 25.6 Miscellaneous functions

Here are a few other functions that are useful in certain circumstances.

**object\_name(ind)** Returns the name of the object with the given index  
**room\_name(ind)** Returns the name of the room with the given index  
**sprite\_name(ind)** Returns the name of the sprite with the given index  
**background\_name(ind)** Returns the name of the background with the given index  
**sound\_name(ind)** Returns the name of the sound with the given index  
**script\_name(ind)** Returns the name of the script with the given index

## Chapter 26 GML: 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 variables and functions exist:

**mouse\_x**\* X-coordinate of the mouse. Cannot be changed.  
**mouse\_y**\* Y-coordinate of the mouse. Cannot be changed.  
**mouse\_button** Currently pressed mouse button. As value use mb\_none, mb\_left, mb\_middle, or mb\_right.  
**keyboard\_lastkey** Keycode of last key pressed. See below for keycode constants. You can change it, e.g. set it to 0 if you handled it.  
**keyboard\_key** Keycode of current key pressed (see below; 0 if none).  
**keyboard\_lastchar** Last character pressed (as string).  
**keyboard\_string** String containing the last at most 80 characters typed. This string will only contain the printable characters typed. It also correctly responds to pressing the backspace key by erasing the last character.

To check whether a particular key or mouse button is pressed you can use the following functions. This is in particular useful when multiple keys are pressed simultaneously.

**keyboard\_check(key)** Returns whether the key with the particular keycode is pressed.  
**keyboard\_check\_direct(key)** Returns whether the key with the particular keycode is pressed by checking the hardware directly. The result is independent of which application has focus. It allows for a few more checks. In particular you can use keycodes vk\_lshift, vk\_lcontrol, vk\_lalt, vk\_rshift, vk\_rcontrol and vk\_ralt to check whether the left or right shift, control or alt key is pressed. (This does not work under windows 95!).  
**mouse\_check\_button(numb)** Returns whether the mouse button is pressed (use as values mb\_none, mb\_left, mb\_middle, or mb\_right).

The following constants for virtual keycodes exist:

**vk\_nokey** keycode representing that no key is pressed  
**vk\_anykey** keycode representing that any key is pressed  
**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  
**vk\_multiply** multiply key on the numeric keypad  
**vk\_divide** divide key on the numeric keypad  
**vk\_add** add key on the numeric keypad  
**vk\_subtract** subtract key on the numeric keypad  
**vk\_decimal** decimal dot keys on the numeric keypad

For the letter keys use for example `ord( 'A' )`. (The capital letters.) The following constants can only be used in `keyboard_check_direct`:

**vk\_lshift** left shift key  
**vk\_lcontrol** left control key  
**vk\_lalt** left alt key  
**vk\_rshift** right shift key  
**vk\_rcontrol** right control key  
**vk\_ralt** right alt key

For example, assume you have an object that the user can control with the arrow keys you can put the following piece of code in the step event of the object:

```

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

```

Of course it is a lot easier to simply put this in the keyboard events.

As you probably know, in the game options you can indicate whether keyboard and mouse event should be continuous or only once when the user presses the button or key. You can set and change this within code using the following variables:

**keyboard\_continuous** Indicated whether keyboard events should be continuous (1) or not (0).  
**mouse\_continuous** Indicated whether mouse events should be continuous (1) or not (0).

There are three additional functions related to interaction.

**keyboard\_clear(key)** Clears the state of the key. This means that it will no longer generate keyboard events until it starts repeating.  
**mouse\_clear(button)** Clears the state of the mouse button. This means that it will no longer generate mouse events until the player releases it and presses it again.  
**io\_clear()** Clears all keyboard and mouse states.  
**io\_handle()** Handle user io, updating keyboard and mouse status.  
**keyboard\_wait()** Waits till the user presses a key on the keyboard.

## 26.1 Joystick support

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 `keyboard_check()` function because that function checks the keyboard. Instead there is a whole set of functions to deal with joysticks. *Game Maker* supports up to two joysticks. So all of these functions take a joystick id as argument.

**joystick\_exists(id)** Returns whether joystick id (1 or 2) exists.  
**joystick\_direction(id)** Returns the keycode (vk\_numpad1 to vk\_numpad9) corresponding to the direction of joystick id (1 or 2).  
**joystick\_check\_button(id,numb)** Returns whether the joystick button is pressed (numb in the range 1-4).  
**joystick\_xpos(id)** Returns the position (-1 to 1) of the x-axis of joystick id.  
**joystick\_ypos(id)** Returns the joystick's y-position.  
**joystick\_zpos(id)** Returns the joystick's z-position (if it has a z-axis).

## Chapter 27 GML: Game graphics

An important part of a game is the graphics. *Game Maker* normally takes care of most of this and for simple games there is need to worry about it. But sometimes you want to take more control. For some aspects there are actions but from code you can control many more aspects. This chapter describes all variables and functions available for this and gives some more information about what is really happening.

### 27.1 Window and cursor

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

**full\_screen** This variable is true when in full-screen mode. You can change the mode by setting this variable to true or false.

Note that in full screen mode the caption and the score are shown on the screen. (This can be avoided using the game options.) In full screen mode the image is either centered or scaled. You can control this using the following variable:

**scale\_screen** This variable is true when in scaled mode. You can change the mode by setting this variable to true or false.

Scaled mode can be slow on machines with a slow processor or graphics card. Default each game runs with a visible cursor. For lots of games you don't want this. To remove the cursor, use the variable:

**show\_cursor** If set to 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. Just create an object with a negative depth that, in its step event, follows the mouse position.

To find out the resolution of the monitor you can use the following two read-only variables:

**monitor\_width** The width of the monitor, in pixels.

**monitor\_height** The height of the monitor, in pixels.

### 27.2 Sprites and images

Each object has a sprite associated with it. This is either a single image or it consists of multiple image. For each instance of the object the program draws the corresponding image on the screen, with its origin (as defined in the sprite properties) at the position (x,y) of the instance. When there are multiple images, it cycles through the images to get an animation effect. There are a number of variables that affect the way the image is

drawn. These can be used to change the effects. Each instance has the following variables:

**visible** If visible is true (1) the image is drawn, otherwise it is not drawn.

Invisible instances are still active and create collision 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.

**sprite\_index** This is the index of the current sprite for the instance. You can change it to give the instance a different sprite. As value you can use the names of the different sprites you defined. Changing the sprite does not change the index of the currently visible subimage.

**sprite\_width\*** Indicates the width of the sprite. This value cannot be changed but you might want to use it.

**sprite\_height\*** Indicates the height of the sprite. This value cannot be changed but you might want to use it.

**sprite\_xoffset\*** Indicates the horizontal offset of the sprite as defined in the sprite properties. This value cannot be changed but you might want to use it.

**sprite\_yoffset\*** Indicates the vertical offset of the sprite as defined in the sprite properties. This value cannot be changed but you might want to use it.

**image\_number\*** The number of subimages for the current sprite for the instance (cannot be changed).

**image\_index** When the image has multiple subimages the program cycles through them. This variable indicates the currently drawn subimage (they are numbered starting from 0). You can change the current image by changing this variable. The program will continue cycling, starting at this new index.

**image\_single** Sometimes you want a particular subimage to be visible and don't want the program to cycle through all of them. This can be achieved 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. For example, assume you have an object that can rotate and you create a sprite that has subimages for a number of orientations (counter-clockwise). Then, in the step event of the object you can set

```
{
    image_single= direction * image_number/360;
}
```

**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, drawing each subimage multiple times. Larger values will skip subimages to make the motion faster.

**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, unless you set it to a different value in the object properties. 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.

**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. Realize that scaled images (in particular when you make them smaller) take more time to draw. Changing the scale can be used to get a 3-D effect.

**bbox\_left\*** Left side of the bounding box used of the image of the instance (taking scaling into account).

**bbox\_right\*** Right side of the bounding box of the instance image

**bbox\_top\*** Top side of the bounding box of the instance image.

**bbox\_bottom\*** Bottom side of the bounding box of the instance image.

Sprites take lots of memory. To draw them fast enough it is important to store them in video memory. As indicated in Chapter 13 you can indicate which sprites should be stored in video memory. Also you can indicate that certain sprites should only be loaded when needed. These sprites will be discarded again at the end of the level. You can partially control this process from code. The following functions exist:

**sprite\_discard(numb)** Frees the (video) memory used for the sprite. If the sprite has the load-on-use property set it will be completely removed. Otherwise, a copy is maintained in normal memory (of which there is normally enough) such that the sprite can be restored when needed.

**sprite\_restore(numb)** Restores the sprite in (video) memory. Normally this happens automatically when the sprite is needed. But this might cause a small hick-up, in particular when load-on-use is set and the sprite is large. So you might want to force this for example at the beginning of the room in which the sprite is needed.

**discard\_all()** Discard all sprites, backgrounds and sounds that have load-on-use set.

When a game uses a lot of different large sprite images, this makes the game file large and, hence, the loading slow. Also, if you want to keep them in memory while you need them, it increases the amount of memory required considerably. Alternatively, you can distribute the sprite images with the game (as .bmp, .jpg, or .gif files; no other formats allowed) and load them during the game. There are three routines for this:

**sprite\_add(fname, imgnumb, precise, transparent, videomem, loadonuse, xorig, yorig)** Add the image stored in the file fname to the set of sprite resources. Only bmp, jpg and gif images can be dealt with. When the image is a bmp or jpg image it can be a strip containing a number of subimages for the sprite next to each other. Use imgnumb to indicate their number (1 for a single image). For (animated) gif images, this argument is not used; the number of images in the gif file is used. precise indicates whether precise collision checking should be used. transparent indicates whether the image is partially transparent, videomem



indicates whether the sprite must be stored in videomemory, and `loadonuse` indicates whether the sprite should only be loaded when used. `xorig` and `yorig` indicate the position of the origin in the sprite. The function returns the index of the new sprite that you can then use to draw it or to assign it to the variable `sprite_index` of an instance. When an error occurs `-1` is returned.

**`sprite_replace(ind, fname, imgnumb, precise, transparent, videomem, loadonuse, xorig, yorig)`** Same as above but in this case the sprite with index `ind` is replaced. The function returns whether it is successful.

**`sprite_delete(ind)`** Deletes the sprite from memory, freeing the memory used. (It can no longer be restored.)

WARNING: When you save the game during playing, added or replaced sprites are NOT stored with the save game. So if you load the saved game later, these might not be there anymore. Also there are some copyright issues with distributing gif files with your (commercial) application. So better don't use these.

## 27.3 Backgrounds

Each room can have up to 8 backgrounds. Also it has a background color. All aspects of these backgrounds you can change in a piece of code using the following variables (note that some are arrays that range from 0 to 7, indicating the different backgrounds):

**`background_color`** Background color for the room.

**`background_showcolor`** Whether to clear the window in the background color.

**`background_visible[0..7]`** Whether the particular background image is visible.

**`background_foreground[0..7]`** Whether the background is actually a foreground.

**`background_index[0..7]`** Background image index for the background.

**`background_x[0..7]`** X position of the background image.

**`background_y[0..7]`** Y position of the background image.

**`background_width[0..7]`** Width of the background image.

**`background_height[0..7]`** Height of the background image.

**`background_h tiled[0..7]`** Whether horizontally tiled.

**`background_v tiled[0..7]`** Whether vertically tiled.

**`background_hspeed[0..7]`** Horizontal scrolling speed of the background (pixels per step).

**`background_vspeed[0..7]`** Vertical scrolling speed of the background (pixels per step).

Background images take lots of memory. To draw them fast enough it can be useful to store them in video memory. As indicated in Chapter 15 you can indicate which backgrounds should be stored in video memory. Also you can indicate that certain backgrounds should only be loaded when needed. These backgrounds will be discarded again at the end of the level. You can partially control this process from code. The following functions exist:

**background\_discard(numb)** Frees the (video) memory used for the background image. If the background has the load-on-use property set it will be completely removed. Otherwise, a copy is maintained in normal memory (of which there is normally enough) such that the background can be restored when needed.

**background\_restore(numb)** Restores the background image in (video) memory. Normally this happens automatically when the background is needed. But this might cause a small hiccup, in particular when load-on-use is set and the background is large. So you might want to force this for example at the beginning of the room in which the background is needed.

**discard\_all()** Discard all sprites, backgrounds and sounds that have load-on-use set.

When a game uses a lot of different background images, this makes the game file large and, hence, the loading slow. Also, if you want to keep them in memory while you need them, it increases the amount of memory required considerably. Alternatively, you can distribute the background images with the game (as .bmp, .jpg, or .gif files; no other formats allowed) and load them during the game. There are three routines for this. Another use is when you want to let the player choose a background. Also, you might want to save the image from within the game and use that later as a background (e.g. for a painting program). Finally, complicated backgrounds, stored as jpg files use a lot less memory. Here are the functions:

**background\_add(fname,transparent,videomem,loadonuse)** Add the image stored in the file fname to the set of background resources. Only bmp and jpg images can be dealt with. transparent indicates whether the image is partially transparent, videomem indicates whether the background must be stored in videomemory, and loadonuse indicates whether the background should only be loaded when used. The function returns the index of the new background that you can then use to draw it or to assign it to the variable background\_index[0] to make it visible in the current room. When an error occurs -1 is returned.

**background\_replace(ind,fname,transparent,videomem,loadonuse)** Same as above but in this case the background with index ind is replaced. The function returns whether it is successful. When the background is currently visible in the room it will be replaced also.

**background\_delete(ind)** Deletes the background from memory, freeing the memory used. (It can no longer be restored.)

WARNING: When you save the game during playing, added or replaced backgrounds are NOT stored with the save game. So if you load the saved game later, these might not be there anymore. Also there are some copyright issues with distributing gif files with your (commercial) application. So better don't use these.

## 27.4 Drawing functions

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. You can only use these in the drawing event of an object; these functions don't make any sense anywhere else in code (although, see Section 27.7). Please realize that the graphics hardware in computers only makes the drawing of images fast. So any other drawing routine will be relatively slow. Also *Game Maker* is optimized towards drawing images. So avoid other drawing routines as much as possible. (Whenever possible, create a bitmap instead.) Also realize that collisions between instances are determined by their sprites (or masks) and not by what you actually draw. The following image related drawing functions exist:

**draw\_sprite(n,img,x,y)** Draws subimage img (-1 = current) of the sprite with index n with its origin at position (x,y).

**draw\_sprite\_scaled(n,img,x,y,s)** Draws the sprite scaled with a factor s.

**draw\_sprite\_stretched(n,img,x,y,w,h)** Draws the sprite stretched such that it fills the region with top-left corner (x,y) and width w and height h.

**draw\_sprite\_transparent(n,img,x,y,s,alpha)** Draws the sprite scaled with factor s merged with its background. alpha indicates the transparency factor.

A value of 0 makes the sprite completely transparent. A value of 1 makes it completely solid. This function can create great effect (for example partially transparent explosions). It is though very slow because it is done in software and, hence, should be used with care.

**draw\_background(n,x,y)** Draws the background with index n at position (x,y).

**draw\_background\_scaled(n,x,y,s)** Draws the background scaled.

**draw\_background\_stretched(n,x,y,w,h)** Draws the background stretched to the indicated region.

**draw\_background\_transparent(n,x,y,s,alpha)** Draws the background scales with factor s and transparency alpha (0-1) (slow!).

**draw\_background\_tiled(n,x,y)** Draws the background tiled such that it fills the entire room.

The following drawing functions draw basic shapes. They use a number of properties, in particular the brush and pen color that can be set using certain variables.

**draw\_pixel(x,y)** Draws a pixel at (x,y) in the brush color.

**draw\_getpixel(x,y)** Returns the color of the pixel at (x,y).

**draw\_fill(x,y)** Flood fill from position (x,y) in the brush color.

**draw\_line(x1,y1,x2,y2)** Draws a line from (x1,y1) to (x2,y2).

**draw\_circle(x,y,r)** Draws a circle at (x,y) with radius r.

**draw\_ellipse(x1,y1,x2,y2)** Draws an ellipse.

**draw\_rectangle(x1,y1,x2,y2)** Draws a rectangle.

**draw\_roundrect(x1,y1,x2,y2)** Draws a rounded rectangle.

**draw\_triangle(x1,y1,x2,y2,x3,y3)** Draws a triangle.  
**draw\_arc(x1,y1,x2,y2,x3,y3,x4,y4)** Draws an arc of an ellipse.  
**draw\_chord(x1,y1,x2,y2,x3,y3,x4,y4)** Draws a chord of an ellipse.  
**draw\_pie(x1,y1,x2,y2,x3,y3,x4,y4)** Draws a pie of an ellipse.  
**draw\_button(x1,y1,x2,y2,up)** Draws a button, up indicates whether up (1) or down (0).  
**draw\_text(x,y,string)** Draws the string at position (x,y).

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 variables 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 variables 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).

**brush\_color** Color used to fill shapes. A whole range of predefined colors is available:

**c\_aqua**  
**c\_black**  
**c\_blue**  
**c\_dkgray**  
**c\_fuchsia**  
**c\_gray**  
**c\_green**  
**c\_lime**  
**c\_ltgray**  
**c\_maroon**  
**c\_navy**  
**c\_olive**  
**c\_purple**  
**c\_red**  
**c\_silver**  
**c\_teal**  
**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.

**brush\_style** Current brush style used for filling. The following styles are available:

**bs\_hollow**  
**bs\_solid**  
**bs\_bdiagonal**  
**bs\_fdiagonal**  
**bs\_cross**  
**bs\_diagcross**  
**bs\_horizontal**  
**bs\_vertical**

**pen\_color** Color of the pen to draw boundaries.

**pen\_size** Size of the pen in pixels.

**font\_color** Color of the font to use.  
**font\_size** Size of the font to use (in points).  
**font\_name** Name of the font (a string).  
**font\_style** Style for the font. The following styles are available:  
     **fs\_normal**  
     **fs\_bold**  
     **fs\_italic**  
     **fs\_bolditalic**  
**font\_angle** Angle with which the font is rotated (0-360 degrees). For example, for vertical text use value 90.  
**font\_align** Alignment of the text w.r.t. the position given. The following values can be used  
     **fa\_left**  
     **fa\_center**  
     **fa\_right**

A few miscellaneous functions exist:

**string\_width(string)** Width of the string in the current font. Can be used for precisely positioning graphics.  
**string\_height(string)** Height of the string in the current font.  
**screen\_gamma(r,g,b)** Sets the gamma correction values. r,g,b must be in the range from -1 to 1. The default is 0. When you use a value smaller than 0 that particular color becomes darker. If you use a value larger than 0 that color becomes lighter. Most of the time you will keep the three values the same. For example, to get the effect of lightning you can temporarily make the three values close to 1. This function works only in exclusive mode!  
**screen\_save(fname)** Saves a bmp image of the screen in the given filename. Useful for making screenshots.  
**screen\_save\_part(fname,left,top,right,bottom)** Saves part of the screen in the given filename.

## 27.5 Views

As you should know you can define up to eight different views when designing rooms. In this way you can show different parts of the room at different places on the screen. Also, you can make sure that a particular object always stays visible. You can control the views from within code. You can make views visible and invisible and change the place or size of the views on the screen or the position of the view in the room (which is in particular useful when you indicated no object to be visible), you can change the size of the horizontal and vertical border around the visible object, and you can indicate which object must remain visible in the views. The latter is very important when the important 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 (assuming this must happen in the first view):

```
{
```

```

    view_object[0] = object_index;
}

```

The following variables exist that influence the view. All, except the first two are arrays ranging from 0 (the first view) to 7 (the last view).

**view\_enabled** Whether views are enabled or not.  
**view\_current** \* The currently drawn view (0-7). Use this only in the drawing event. You can for example check this variable to draw certain things in only one view. Variable cannot be changed.  
**view\_visible[0..7]** Whether the particular view is visible on the screen.  
**view\_left[0..7]** Left position of the view in the room.  
**view\_top[0..7]** Top position of the view in the room.  
**view\_width[0..7]** Width of the view (in pixels).  
**view\_height[0..7]** Height of the view (in pixels).  
**view\_x[0..7]** X-position of the view on the screen.  
**view\_y[0..7]** Y-position of the view on the screen.  
**view\_hborder[0..7]** Size of horizontal border around the visible object (in pixels).  
**view\_vborder[0..7]** Size of vertical border around visible object (in pixels).  
**view\_hspeed[0..7]** Maximal horizontal speed of the view.  
**view\_vspeed[0..7]** Maximal vertical speed of the view.  
**view\_object[0..7]** Object whose instance must remain visible in the view.

Note that the size of the image on the screen is decided based on the visible views at the beginning of the room. If you change views during the game, they might no longer fit on the screen. The screen size though is not adapted automatically. So if you need this you have to do it yourself, using the following variables:

**screen\_width** Width of the image on the screen, that is, the area in which we draw. When there are no views, this is the same as `room_width`.  
**screen\_height** Height of the image on the screen.

## 27.6 Transitions

As you know, when you move from one room to another you can indicate a transition. You can also set the transition for the next frame without moving to another room using the variable called `transition_kind`. If you assign a value between 1 and 13 to it the corresponding transition is used (these are the same transitions you can indicate for the rooms). A value of 0 indicates no transition. It only affects the next time a frame is drawn. You can also set these variables before going to the next room using code.

**transition\_kind** Indicates the next frame transition (0-13).  
**transition\_time** Total time used for the transition (in milliseconds).  
**transition\_steps** Number of steps for the transition.

## 27.7 Repainting the screen

Normally at the end of each step the room is repainted on the screen. But in rare circumstances you need to repaint the room at other moments. This happens when your program takes over the control. For example, before sleeping a long time a repaint might be wanted. Also, when your code displays a message and wants to wait for the player to press a key, you need a repaint in between. There are two different routines to do this.

**screen\_redraw()** Redraws the room by calling all draw events.  
**screen\_refresh()** Refreshes the screen using the current room image (not performing drawing events).

To understand the second function, you will need to understand a bit better how drawing works internally. There is internally an image on which all drawing happens. This image is not visible on the screen. Only at the end of a step, after all drawing has taken place, the screen image is replaced by this internal image. (This is called double buffering.) The first function redraws the internal image and then refreshes the screen image. The second function only refreshes the image on the screen.

Now you should also realize why you couldn't use drawing actions or functions in other events than drawing events. They will draw things on the internal image but these won't be visible on the screen. And when the drawing events are performed, first the room background is drawn, erasing all you did draw on the internal image. But when you use `screen_refresh()` after your drawing, the updated image will become visible on the screen. So, for example, a script can draw some text on the screen, call the refresh function and then wait for the player to press a key, like in the following piece of code.

```
{
  draw_text(screen_width/2,100,'Press any key to continue. ');
  screen_refresh();
  keyboard_wait();
}
```

Please realize that, when you draw in another event than the drawing event, you draw simply on the image, not in a view! So the coordinates you use are the same as if there are no views.

Be careful when using this technique. Make sure you understand it first and realize that refreshing the screen takes some time.

## Chapter 28 GML: Sound and music

Sound plays a crucial role in computer games. 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.

Sounds are added to your game in the form of sound resources. 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 four basic functions related to sounds, two to play a sound, one to check whether a sound is playing, and another to stop a sound. All take the index of the sound as argument. The name of the sound represents its index. But you can also store the index in a variable, and use that one.

**sound\_play(index)** Plays the indicates sound once.

**sound\_loop(index)** Plays the indicates sound, looping continuously.

**sound\_stop(index)** Stops the indicates sound. If there are multiple sounds with this index playing simultaneously, all will be stopped.

**sound\_isplaying(index)** Returns whether the indicates sound is playing.

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 advanced tab of the sound properties by checking the appropriate 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 effects. 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(index,value)** Changes the volume for the indicates sound (0 = low, 1 = high).



**sound\_pan(index,value)** Changes the pan for the indicates sound (0 = left, 1 = right).

**sound\_frequency(index,value)** Changes the frequency for the indicates sound (0 = low, 1 = high).

Sounds use many resources and most systems can store and play only a limited number of sounds. If you make a large game you would like to have more control over which sounds are loaded in memory at what times. You can use the load-on-use option for sounds to make sure sounds are only loaded when used. This though has the problem that you might get a small hiccup when the sound is used first. Also, it does not help much when you have just one large room. For more control you can use the following functions.

**sound\_discard(index)** Frees the memory used for the indicated sound.

**sound\_restore(index)** Restores the indicated sound in memory.

**discard\_all()** Discard all sprites, backgrounds and sounds that have load-on-use set.

When your game uses many different complicated sounds, for example, as background music, you better not store them all in the game. This makes the game file very large. Instead, it is better to provide them as separate files with the game and load them when they are needed. This will also reduce the loading time of the game. The following three routines exist for this:

**sound\_add(fname,buffers,effects,loadonuse)** Adds a sound resource to the game. fname is the name of the sound file. buffers indicates the number of buffers to be used, and effects and loadonuse indicate whether sound effects are allowed and whether the sound should be stored in internal memory (true or false). The function returns the index of the new sound, which can be used to play the sound. (-1 if an error occurred, e.g. the file does not exist).

**sound\_replace(index,fname,buffers,effects,loadonuse)** Same as the previous function but this time not a new sound is created but the existing sound index is replaced, freeing the old sound. Returns whether correct.

**sound\_delete(index)** Deletes the indicated sound, freeing all memory associated with it. It can no longer be restored.

**WARNING:** When you save the game during playing, added or replaced sounds are NOT stored with the save game. So if you load the saved game later, these might not be there anymore.

Sound is a complicated matter. Midi files are played using the standard multimedia player. Only one midi file can be played at once and there is no support for sound effects. For wave files *Game Maker* uses DirectSound. In this case all wave files are stored in memory and can have effects. *Game Maker* actually also tries to play other music files when you specify them, in particular mp3 files. It uses the standard multimedia player for this. Be careful though. Whether this works depends on the system and sometimes on

other software installed or running. So you are recommended not to use mp3 files when you want to distribute your games.

## Chapter 29 GML: Splash screens, highscores, and other pop-ups

Many games have so-called splash screen. These screens show a video, an image, or some text. Often they are used at the beginning of the game (as an intro), the beginning of a level, or at the end of the game (for example the credits). In *Game Maker* such splash screens with text, images or video can be shown at any moment during the game. The game is temporarily paused while the splash screen is shown. These are the functions to use:

**show\_text(fname,full,backcol,delay)** Shows a text splash screen. fname is the name of the text file (.txt or .rtf). You must put this file in the folder of the game yourself. Also when you create a stand-alone version of your game, you must not forget to add the file there. full indicates whether to show it in full screen mode. backcol is the background color, and delay is the delay in seconds before returning to the game. (The player can always click with the mouse in the screen to return to the game.)

**show\_image(fname,full,delay)** Shows an image splash screen. fname is the name of the image file (only .bmp, .jpg and .wmf files). You must put this file in the folder of the game yourself. full indicates whether to show it in full screen mode. delay is the delay in seconds before returning to the game.

**show\_video(fname,full,loop)** Shows a video splash screen. fname is the name of the video file (.avi,.mpg). You must put this file in the folder of the game yourself. full indicates whether to show it in full screen mode. loop indicates whether to loop the video.

**show\_info()** Displays the game information form.

A number of other functions exist to pop up messages, questions, a menu with choices, or a dialog in which the player can enter a number, a string, or indicate a color or file name:

**show\_message(str)** Displays a dialog box with the string as a message.

**show\_question(str)** Displays a question; returns true when the user selects yes and false otherwise.

**show\_menu(str,def)** Shows a popup menu. str indicates the menu text. This consists of the different menu items with a vertical bar between them. For example, str = 'menu0|menu1|menu2'. When the first item is selected a 0 is returned, etc. When the player selects no item, the default value def is returned.

**get\_integer(str,def)** Asks the player in a dialog box for a number. str is the message. def is the default number used.

**get\_string(str,def)** Asks the player in a dialog box for a string. str is the message. def is the default value.

**get\_color(defcol)** Asks the player for a color. defcol is the default color.

**get\_open\_filename(filter,fname)** Asks the player for a filename to open with the given filter. The filter has the form 'name1|mask1|name2|mask2|...'. A

mask contains the different options with a semicolon between them. \* means any string. For example: 'bitmaps|\*.bmp;\*.wmf'.

**get\_save\_filename(filter,fname)** Asks for a filename to save with the given filter.

**get\_directory(dname)** Asks for directory. dname is the default name.

One special pop-up is the highscore list that is maintained for each game. The following functions exist:

**highscore\_show(numb)** Shows the highscore table. numb is the new score. If this score is good enough to be added to the list, the player can input a name. Use -1 to simply display the current list.

**highscore\_clear()** Clears the highscore list.

**highscore\_add(str,numb)** Adds a player with name str and score numb to the list.

**highscore\_value(place)** Returns the score of the person on the given place (1-10). This can be used to draw your own highscore list.

**highscore\_name(place)** Returns the name of the person on the given place (1-10).

Please realize that none of these pop-ups can be shown when the game runs in exclusive graphics mode!

## Chapter 30 GML: Files, registry, and executing programs

In more advanced games you probably want to read data from a file that you provide with the game. For example, you could make a file that describes at what moments certain things should happen. Also you probably want to save information for the next time the game is run (for example, the current room). The following functions exist for this:

**file\_exists(fname)** Returns whether the file with the given name exists (true) or not (false).  
**file\_delete(fname)** Deletes the file with the given name.  
**file\_rename(oldname,newname)** Renames the file with name oldname into newname.  
**file\_copy(fname,newname)** Copies the file fname to the newname.  
**file\_open\_read(fname)** Opens the indicated file for reading.  
**file\_open\_write(fname)** Opens the indicated file for writing, creating it if it does not exist.  
**file\_close()** Closes the current file (don't forget to call this!).  
**file\_write\_string(str)** Writes the string to the currently open file.  
**file\_write\_real(x)** Write the real value to the currently open file.  
**file\_writeln()** Write a newline character to the file.  
**file\_read\_string()** Reads a string from the file and returns this string. A string ends at the end of line.  
**file\_read\_real()** Reads a real value from the file and returns this value.  
**file\_readln()** Skips the rest of the line in the file and starts at the start of the next line.  
**file\_eof()** Returns whether we reached the end of the file.  
**directory\_exists(dname)** Returns whether the indicated directory does exist.  
**directory\_create(dname)** Created a directory with the given name (including the path towards it) if it does not exist.

If the player has checked secure mode in his preferences, for a number of these routines, you are not allowed to specify a path, and only files in the application folder can e.g. be written.

The following three read-only variables can be useful:

**game\_id\*** Unique identifier for the game. You can use this if you need a unique file name.  
**working\_directory\*** Working directory for the game. (Not including the final backslash.)  
**temp\_directory\*** Temporary directory created for the game. You can store temporary files here. They will be removed at the end of the game.

If you want to store a small amount of information between runs of the game there is a simpler mechanism than using a file. You can use the registry. The registry is a large database that Windows maintains to keep track of all sorts of settings for programs. An entry has a name, and a value. You can use both string and real values. The following functions exist:

**registry\_write\_string(name, str)** Creates an entry in the registry with the given name and string value.  
**registry\_write\_real(name, x)** Creates an entry in the registry with the given name and real value.  
**registry\_read\_string(name)** Returns the string that the given name holds. (The name must exist. otherwise an empty string is returned.)  
**registry\_read\_real(name)** Returns the real that the given name holds. (The name must exist. Otherwise the number 0 is returned.)  
**registry\_exists(name)** Returns whether the given name exists.

Actually, values in the registry are grouped into keys. The above routines all work on values within the key that is especially created for your game. Your program can use this to obtain certain information about the system the game is running on. You can also read values in other keys, but you are not allowed to write them because this easily destroys the system. Note that keys are again placed in groups. The following routines only work on the group HKEY\_CURRENT\_USER. So, for example, if you want to find out the current temp dir, use

```
path = registry_read_string_ext('/Environment', 'TEMP');
```

The following functions exist.

**registry\_read\_string\_ext(key, name)** Returns the string that the given name in the indicated key holds. (The name must exist. otherwise an empty string is returned.)  
**registry\_read\_real\_ext(key, name)** Returns the real that the given name in the indicated key holds. (The name must exist. Otherwise the number 0 is returned.)  
**registry\_exists\_ext(key, name)** Returns whether the given name exists in the given key.

*Game Maker* also has the possibility to start external programs. There are two functions available for this: `execute_program` and `execute_shell`. The function `execute_program` starts a program, possibly with some arguments. It can wait for the program to finish (pausing the game) or continue the game. The function `execute_shell` opens a file. This can be any file for which some association is defined, e.g. an html-file, a word file, etc. Or it can be a program. It cannot wait for completion so the game will continue.

**execute\_program(prog, arg, wait)** Execute program prog with arguments arg. wait indicates whether to wait for finishing.  
**execute\_shell(prog, arg)** Executes the program (or file) in the shell.

Both functions will not work if the player set the secure mode in the preferences. You can check this using the read-only variable:

**secure\_mode\*** Whether the game is running in secure mode.

## Chapter 31 GML: Multiplayer games

Playing games against the computer is fun. But playing games against other human players can be even more fun. It is also relatively easy to make such games because you don't have to implement complicated computer opponent AI. You can of course sit with two players behind the same monitor and use different keys or other input devices, but it is a lot more interesting when each player can sit behind his own computer. Or even better, one player sits on the other side of the ocean. *Game Maker* has multiplayer support. Please realize that creating effective multiplayer games that synchronize well and have no latency is a difficult task. This chapter gives a brief description of the possibilities. On the website a tutorial is available with more information.

### 31.1 Setting up a connection

For two computer to communicate they will need some connection protocol. Like most games, *Game Maker* offers four different types of connections: IPX, TCP/IP, Modem, and Serial. The IPX connection (to be more precise, it is a protocol) works almost completely transparent. It can be used to play games with other people on the same local area network. It needs to be installed on your computer to be used. (If it does not work, consult the documentation of Windows. Or go to the Network item in the control panel of Windows and add the IPX protocol.) TCP/IP is the internet protocol. It can be used to play with other players anywhere on the internet, assuming you know their IP address. On a local network you can use it without providing addresses. A modem connection is made through the modem. You have to provide some modem setting (an initialization string and a phone number) to use it. Finally, when using a serial line (a direct connection between the computers) you need to provide a number of port settings. There are four GML functions that can be used for initializing these connections:

**`mplay_init_ipx()`** initializes an IPX connection.

**`mplay_init_tcpip(addr)`** initializes a TCP/IP connection. `addr` is a string containing the web address or IP address, e.g. 'www.gameplay.com' or '123.123.123.12', possibly followed by a port number (e.g. ':12'). Only when joining a session (see below) you need to provide an address. On a local area network no addresses are necessary.

**`mplay_init_modem(initstr,phonenr)`** initializes a modem connection. `initstr` is the initialization string for the modem (can be empty). `phonenr` is a string that contains the phone number to ring (e.g. '0201234567'). Only when joining a session (see below) you need to provide a phone number.

**`mplay_init_serial(portno,baudrate,stopbits,parity,flow)`** initializes a serial connection. `portno` is the port number (1-4). `baudrate` is the baudrate to be used (100-256K). `stopbits` indicates the number of stopbits (0 = 1 bit, 1 = 1.5 bit, 2 = 2 bits). `parity` indicates the parity (0=none, 1=odd, 2=even, 3=mark). And `flow` indicates the type of flow control (0=none, 1=xon/xoff, 2=rts, 3=dtr, 4=rts and dtr). Returns whether successful. A typical call is `mplay_init_serial(1,57600,0,0,4)`. Give 0 as a first argument to open a dialog for the user to change the settings.



Your game should call one of these functions exactly once. All functions report whether they were successful. They are not successful if the particular protocol is not installed or supported by your machine. To check whether there is a successful connection available you can use the following function

**mplay\_connect\_status()** returns the status of the current connection. 0 = no connection, 1 = IPX connection, 2 = TCP/IP connection, 3 = modem connection, and 4 = serial connection.

To end the connection call

**mplay\_end()** ends the current connection.

## 31.2 Creating and joining sessions

When you connect to a network, there can be multiple games happening on the same network. We call these sessions. These different sessions can correspond to different games or to the same game. A game must uniquely identify itself on the network. Fortunately, *Game Maker* does this for you. The only thing you have to know is that when you change the game id in the options form this identification changes. In this way you can avoid that people with old versions of your game will play against people with new versions.

If you want to start a new multiplayer game you need to create a new session. For this you can use the following routine:

**mplay\_session\_create(sesname,playnumb,playername)** creates a new session on the current connection. *sesname* is a string indicating the name of the session. *playnumb* is a number that indicates the maximal number of players allowed in this game (use 0 for an arbitrary number). *playname* is the name of you as player. Returns whether successful.

One instance of the game must create the session. The other instance(s) of the game should join this session. This is slightly more complicated. You first need to look what sessions are available and then choose the one to join. There are three routines important for this:

**mplay\_session\_find()** searches for all sessions that still accept players and returns the number of sessions found.

**mplay\_session\_name(numb)** returns the name of session number *numb* (0 is the first session). This routine can only be called after calling the previous routine.

**mplay\_session\_join(numb,playername)** makes you join session number *numb* (0 is the first session). *playername* is the name of you as a player. Returns whether successful.

There is one more routine that can change the session mode. Should be called before creating a session:

**`mplay_session_mode(move)`** sets whether or not to move the session host to another computer when the host ends. `move` should either be true or false (the default).

To check the status of the current session you can use the following function

**`mplay_session_status()`** returns the status of the current session. 0 = no session, 1 = created session, 2 = joined session.

A player can stop a session using the following routine:

**`mplay_session_end()`** ends the session for this player.

### 31.3 Players

Each instance of the game that joins a session is a player. As indicated above, players have names. There are three routines that deal with players.

**`mplay_player_find()`** searches for all players in the current session and returns the number of players found.

**`mplay_player_name(numb)`** returns the name of player number `numb` (0 is the first player, which is always yourself). This routine can only be called after calling the previous routine.

**`mplay_player_id(numb)`** returns the unique id of player number `numb` (0 is the first player, which is always yourself). This routine can only be called after calling the first routine. This id is used in sending and receiving messages to and from individual players.

### 31.4 Shared data

Shared data communication is probably the easiest way to synchronize the game. All communication is shielded from you. There is a set of 10000 values that are common to all entities of the game. Each entity can set values and read values. *Game Maker* makes sure that each entity sees the same values. A value can either be a real or a string. There are just two routines:

**`mplay_data_write(ind, val)`** write value `val` (string or real) into location `ind` (`ind` between 0 and 10000).

**`mplay_data_read(ind)`** returns the value in location `ind` (`ind` between 0 and 10000). Initially all values are 0.

To synchronize the data on the different machines you can either use a guaranteed mode that makes sure that the change arrives on the other machine (but which is slow) or non-guaranteed. To change this use the following routine:

**`mplay_data_mode(guar)`** sets whether or not to use guaranteed transmission for shared data. `guar` should either be true (the default) or false.

## 31.5 Messages

The second communication mechanism that *Game Maker* supports is the sending and receiving of messages. A player can send messages to one or all other players. Players can see whether messages have arrived and take action accordingly. Messages can be sent in a guaranteed mode in which you are sure they arrive (but this can be slow) or in a non-guaranteed mode, which is faster.

The following messaging routines exist:

**`mplay_message_send(player,id,val)`** sends a message to the indicated player (either an identifier or a name; use 0 to send the message to all players). `id` is an integer message identifier and `val` is the value (either a real or a string). The message is sent in non-guaranteed mode.

**`mplay_message_send_guaranteed(player,id,val)`** sends a message to the indicated player (either an identifier or a name; use 0 to send the message to all players). `id` is an integer message identifier and `val` is the value (either a real or a string). This is a guaranteed send.

**`mplay_message_receive(player)`** receives the next message from the message queue that came from the indicated player (either an identifier or a name). Use 0 for messages from any player. The routine returns whether there was indeed a new message. If so you can use the following routines to get its contents:

**`mplay_message_id()`** Returns the identifier of the last received message.

**`mplay_message_value()`** Returns the value of the last received message.

**`mplay_message_player()`** Returns the player that sent the last received message.

**`mplay_message_name()`** Returns the name of the player that sent the last received message.

**`mplay_message_count(player)`** Returns the number of messages left in the queue from the player (use 0 to count all message).

**`mplay_message_clear(player)`** Removes all messages left in the queue from the player (use 0 to remove all message).

A few remarks are in place here. First of all, if you want to send a message to a particular player only, you will need to know the players unique id. As indicated earlier you can obtain this with the function `mplay_player_id()`. This player identifier is also used when receiving messages from a particular player. Alternatively, you can give the name of the player as a string. If multiple players have the same name, only the first will get the message.

Secondly, you might wonder why each message has an integer identifier. The reason is that this helps your application to send different types of messages. The receiver can check the type of message using the id and take appropriate actions. (Because messages are not guaranteed to arrive, sending id and value in different messages would cause serious problems.)

## Chapter 32 GML: Using DLL's

In those cases where the functionality of GML is not enough for your wishes, you can actually extend the possibilities by using plug-ins. A plug-in comes in the form of a DLL file (a Dynamic Link Library). In such a DLL file you can define functions. Such functions can be programmed in any programming language that supports the creation of DLL's (e.g. Delphi, Visual C++, Visual Basic, etc.) You will though need to have some programming skill to do this. Plug-in functions must have a specific format. They can have zero, one or two arguments, each of which can either be a real number (float) or a null-terminated string. They must return either a real or a null-terminated string.

In Delphi you create a DLL by first choosing **New** from the **File** menu and then choosing DLL. Here is an example of a DLL you can use with *Game Maker* written in Delphi. (Note that this is Delphi code, not GML code!)

```
library MyDLL;

uses SysUtils, Classes;

function MyMin(x,y:real):real; cdecl;
begin
    if x<y then Result := x else Result := y;
end;

var res : array[0..1024] of char;

function DoubleString(str:PChar):PChar; cdecl;
begin
    StrCopy(res,str);
    StrCat(res,str);
    Result := res;
end;

exports MyMin, DoubleString;

begin
end.
```

This DLL defines two functions: `MyMin` that takes two real arguments and returns the minimum of the two, and `DoubleString` that doubles the string. Note that you have to be careful with memory management. That is why I declared the resulting string global. Also notice the use of the `cdecl` calling convention. This is required. Once you build the DLL in Delphi you will get a file `MyDLL.DLL`. This file must be placed in the running directory of your game. (Or any other place where windows can find it.)

To use this DLL in *Game Maker* you first need to specify the external functions you want to use and what type of arguments they take. For this there are the following functions in GML:

**external\_define0(dll,name,restype)** Defines an external function without arguments. `dll` is the name of the dll file. `name` is the name of the functions. And `restype` is the type of the result. For this use either `ty_real` or `ty_string`.

**external\_define1(dll,name,arg1type,restype)** Defines an external function with one arguments. Same as above but now `arg1type` is the type of the first argument. For this again use either `ty_real` or `ty_string`.

**external\_define2(dll,name,arg1type,arg2type,restype)** Defines an external function with two arguments.

**external\_define3(dll,name,arg1type,arg2type,arg3type,restype)** Defines an external function with three arguments.

**external\_define4(dll,name,arg1type,arg2type,arg3type,arg4type,restype)** Defines an external function with four arguments.

Each of these functions returns the id of the external function that must be used for calling it. So in the above example, at the start of the game you would use the following GML code:

```
{
    global.mmm =
        external_define2('MYOWN.DLL','MyMin',ty_real,ty_real,ty_real);
    global.ddd =
        external_define1('MYOWN.DLL',DoubleString,ty_string,ty_string);
}
```

Now whenever you need to call the functions, use e.g.

```
{
    aaa = external_call2(global.mmm,x,y);
    sss = external_call1(global.ddd,'Hello');
}
```

So there are the following calling functions:

**external\_call0(id)** Calls the external function with the given id, that has no arguments.

**external\_call1(id,arg1)** Calls the external function with the given id, that has one argument.

**external\_call2(id,arg1,arg2)** Calls the external function with the given id, that has two arguments.

**external\_call3(id,arg1,arg2,arg3)** Calls the external function with the given id, that has three arguments.

**external\_call4(id,arg1,arg2,arg3,arg4)** Calls the external function with the given id, that has four arguments.

You might wonder how to make a function in a DLL that does something in the game. For example, you might want to create a DLL that adds instances of objects to your game. The easiest way is to let your DLL function return a string that contains a piece of

GML code. This string that contains the piece of GML can be executed using the GML function

**execute\_string(str)** Execute the piece of code in the string str.

Now you can call an external function and then execute the resulting string, e.g. as follows:

```
{
  ccc = external_call2(global.mmm,x,y);
  execute_string(ccc);
}
```

Note that DLLs cannot be used in secure mode.

Using external DLLs is an extremely powerful function. But please only use it if you know what you are doing.