Game Maker

Version 3.3 (April, 2001)

Mark Overmars

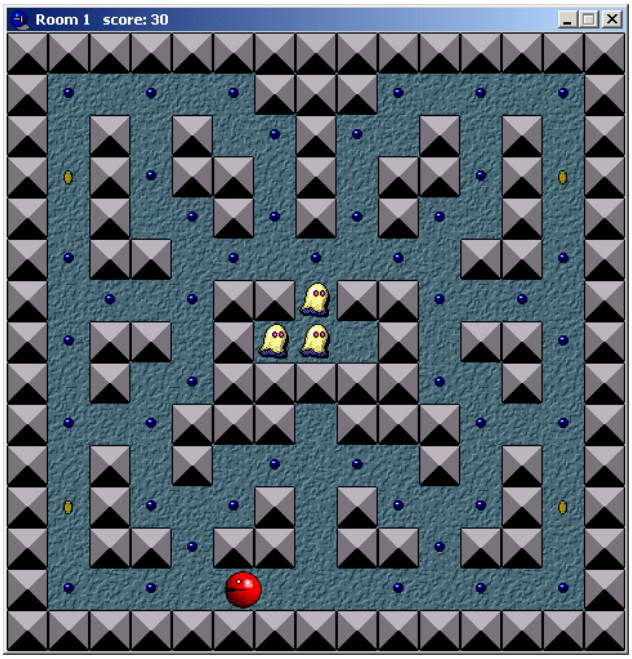


Table of Contents

Part I

Using Game Maker

Introduction

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

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

This document describes all features of *Game Maker* in detail. It consists of two parts. Part I describes the basic aspects of *Game Maker*. You are strongly recommended to completely read this part. Part II describes the built-in programming language and shows how you can use it to built more advanced games.

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

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

a number of full games created with *Game Maker* are provided. Please send you own creations to gamemaker@wanadoo.nl and I might add them to the site. (See below on how to do this.)

System requirements

Game Maker requires a reasonable powerful computer (Pentium 166 with 32 Mb of memory minimum; the requirements for the runner are a bit less) running Windows 95 or later (NT 4 is not supported). It prefers at least 65000 colors (high color, 16-bits) and a screen resolution of 800x600 or more. DirectX must be installed (version 5.0 or higher).

The global idea

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

So the first thing to do is to make some objects. Below you find more information on how to do this but let me give a global description here. Objects first of all have an image so that you can see them. Objects also have a name for easy reference. You can place multiple instances of the same object in a room. So if you have e.g. three monsters you need only to define one monster object (unless you want them to have a different image or different behavior). Objects can be solid, which means that they cannot occupy the same place, or not. Also, they can be active, that is, walk around and react with each other, or be passive. For example, background objects will neither be solid, nor active. Other objects can walk over them and nothing happens. Walls are solid but not

active, so other objects cannot run into them. Figures that move around in your game are active and might be solid or not, depending on their use.

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

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

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

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

A simple example

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

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

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

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

I hope this convinces you how simple things are. You might want to play a bit with this game and change some aspects. E.g. start with two dogs in the room (select the dog from the list at the bottom of the Rooms window and click anywhere in the field). Or change the alarm clock setting in the Objects form (click with the right mouse button on the action to change the settings). You can also make the object move rather than stand still (drag the action with the 8 arrows to the alarm event and select all arrow buttons). Finally, you might want to change the sound that the dog makes.

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

The main interface

When you start *Game Maker,* you are asked to select the game that you want to play or edit. Game files have an extension .gmf. If you installed *Game Maker* correctly, you can also start the program by double clicking on such a file. A game file XXX.gmf is accompanied by a folder XXX_data that contains further game data. If you have used a previous version of *Game Maker* the program will first convert all old-style files to new .gmf files. If you want to start creating a new game, click on the button labeled **Cancel**.

Now a toolbar appears at the top of the screen that might look disappointingly simple. It contains just a menu and a few buttons. But don't be fooled; a lot more awaits you. From left to right you find the following buttons:

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

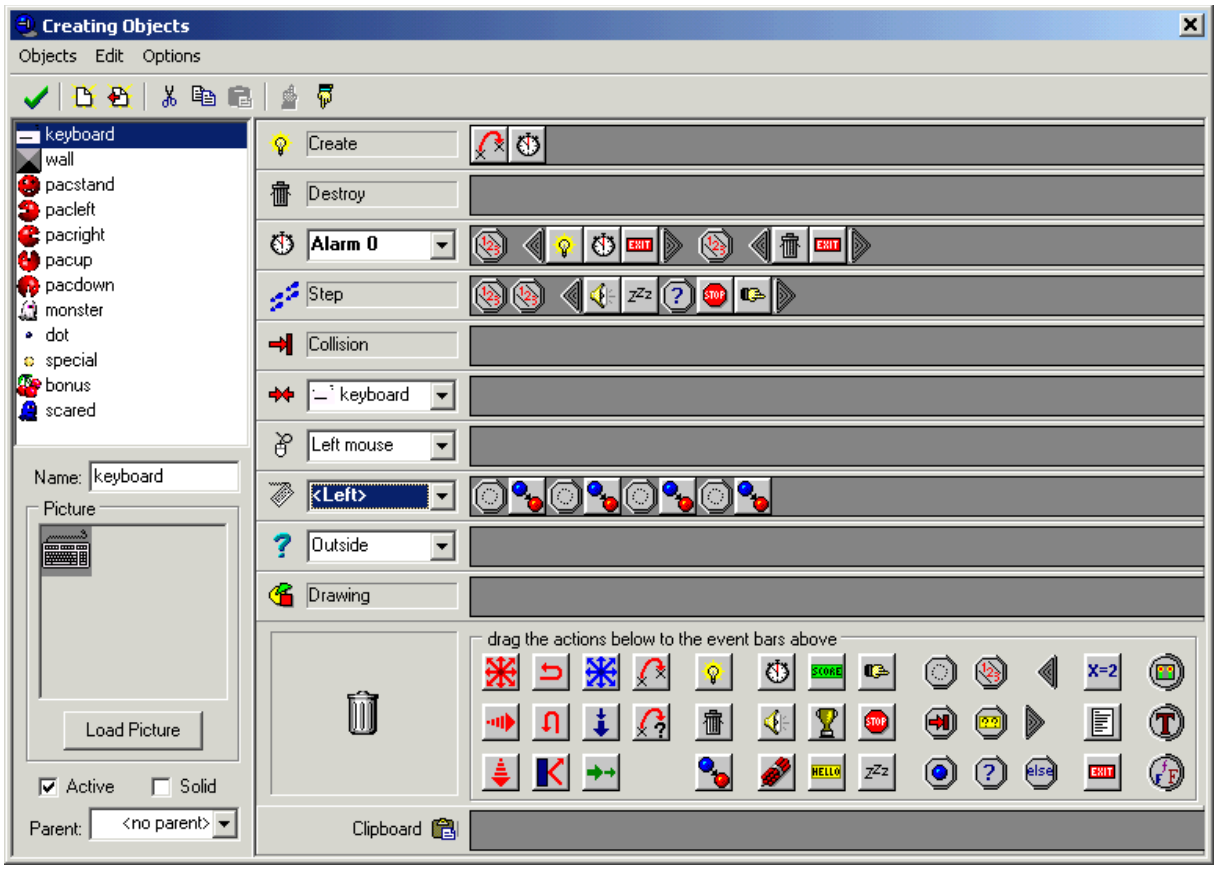
Some more commands are available in the menu. There are for example commands to save the game under a different name. For some of the other commands, see below.

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

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

Creating objects

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



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

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

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

To close the object form press the button with the green checkmark. If you don't want to save your changes, choose the appropriate command from the **Objects** menu.

Object image

Objects have an image associated with them such that you can see them. An image can have any size. An image can be an icon file (*.ico), a bitmap file (*.bmp), a gif file (*.gif), and even jpeg files or metafiles. To pick the

image for an object, click on the button **Load Picture** at the left bottom of the object form. Use the file dialog to pick the correct file. Images are considered partially transparent such that they can move over a background. The color of the left bottom pixel of the image is the transparency color. So always make sure that this pixel is part of the background. Let me describe the possible image types in a bit more detail.

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

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

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

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

Events

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

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

-  **Creation Event**. A creation event occurs whenever an object is created. All instances of active objects will get a creation event when the game starts. You typically use them to give the object a direction and a speed. Also when you create an instance of an object during the game or when you change an instance into a different object, a creation event happens.
-  **Destroy Event**. This event happens when an instance of an object is destroyed. Often you don't need to do anything in this case, but you might use it to do something with the score, to end the game, or to create a new object somewhere else.
-  **Alarm Events**. Each active object has four different alarm clocks that you can set (see the actions below). An alarm clock counts down and when it reaches 0 an alarm event occurs. You can use this to let certain things happen from time to time. For example, an object can change its direction of motion every 10 steps (in such a case the alarm event, as one of its actions, sets the alarm again). Or you can open a door for a short period of time after which you close it again. You select the alarm from the drop-down list and indicate the actions. (Most of the time you will only use alarm0.)
-  **Step Event**. This event occurs every step in the game. For simple games you don't need to do anything here but for more complicated games it is one of the most crucial events. You can for example continuously change the speed or direction of motion.
-  **Collision Event**. A collision event happens when an active object bumps into a solid object. This is not allowed so the actions in this event should take care that the collision does not occur. (If the actions do not avoid the collision, the object will stop moving. If there are no actions defined, the object will not react to the collision and might go straight through the solid object.) A typical action here is to stop the motion, reverse the direction of motion, or choose a random new direction of motion. But also more complicated actions can occur, e.g., you can push the other object out of the way. Collision events only occur when you hit a solid object. If you want to have an action happening when you hit a non-solid object, use the meeting event below. When you specify a meeting event for a solid object, a collision event won't occur. In this way you can specify special behaviour for certain objects.

- **Meeting Events**. This happens when the object meets (that is, intersects) another object. For each object type you can specify a different set of actions. Indicate the object in the drop-down list and then specify the actions. You can specify meeting events for both solid and non-solid objects. Meeting events play a crucial rule. When your man meets a monster he might die. When he hits a piece of gold the piece of gold is destroyed and the score is raised. And when he meets a button a door can be opened (changed from a solid closed door into a non-solid open door).
- **Mouse Events.** A mouse event occurs when the user clicks with the mouse on the object. This can be used for user interaction. For example, you can generate some objects that look like buttons, and when the user clicks on them, something can happen in the game. In the drop-down list you can indicate on which mouse event to react: the left mouse, the right mouse, the middle mouse or no mouse. (This last event happens when the mouse is above the object but no button is pressed.)
- **Keyboard Events.** For further interaction you can specify keyboard events. In the dropdown list specify the key on the keyboard and next indicate the actions that should take place when the key is pressed (when the user keeps the key pressed, the event occurs repeatedly). You can specify actions for the arrow keys, for the numeric keypad (when <NumLock> is pressed) for the normal character keys, and for the function keys. One particular event is the <no key> event. This one is repeatedly generated whenever no key is pressed on the keyboard. This is for example useful to stop the current motion. Another special event is the <any key> event. It is generated when any key is pressed. (User interaction is also possible with the joystick. See part II of this document.)
- **Other Events.** Some other events exist to which the object might want to respond. At the moment there are just two but more might be added in the future. Pick the event you want from the drop-down list.
  - **Outside.** This event happens when the instance lies completely outside the room. This is typically a moment to destroy it.
  - **Boundary.** This event happens when the instance intersects the boundary of the room. Here you might want to stop its motion or reverse it.
- **Drawing Event.** When this event is left empty, every instance of the object is represented by its image. In some cases you want to draw something else. In this case you can put actions in this event. There are a number of special drawing actions that you can use for this.

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

Actions

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

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

Movement

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

-  **Set the direction of motion**. Click on the arrow indicating the direction you want, or click on the square in the middle to make the movement stop. You can press multiple directions. In this case a random choice is made.
-  **Reverse horizontal direction**. Reverses the horizontal direction of motion. Can e.g. be used when hitting a vertical wall.
-  **Reverse vertical direction**. Reverses the vertical direction of motion. Can e.g. be used when hitting a horizontal wall.
-  **Bounce the motion**. Changes the motion as if bounced against the other object. This only works in collision and meeting events. It also only works with solid other objects. It tries to also take other object in the neighborhood into account, leading to a rather realistic behavior.
-  **Set horizontal speed**. Sets the horizontal speed. The number you give is the number of pixels the object moves in each step. Default, the speed is 8. Note that you can set the speed relative to the current speed, if you mark the box labeled **Relative**. If you, e.g., want to make an object move faster all the time you can add a small value (e.g. 0.1) to the speed in every step.
-  **Set vertical speed**. Sets the vertical speed of the object.
-  **Set direction and speed**. This action provides a different way of defining the motion of an object. You indicate the direction of motion (in degrees, 0-359, counter-clockwise; 0 is moving to the right, 90 is moving up, etc.), and the speed (in pixels per step). This makes it easy to move the object in an arbitrary direction. (Internally it is converted into horizontal and vertical motions. The horizontal and vertical directions are set to 1. Preferably don't use the two different forms of defining motions on the same object.) When you set the relative checkmark, the action behaves a bit different. An extra motion is added in the given direction, with the given speed. So for example, when the object is moving to the right and you relatively give a direction of 90 degrees and some speed, the object starts moving diagonally right upwards.
-  **Set gravity**. In some games you want gravity. You specify the direction of gravity in degrees (270 is downwards) and the amount of gravity. In each step the direction and speed adapted accordingly.
-  **Set friction**. In a number of situations, you want the speed of an object to slowly reduce towards 0. For this you can use friction. In each step the speed is decreased by the given amount until it becomes 0.
-  **Move to position (x,y)**. Moves the object to the specified position. (0,0) is the top left position in the room. If you mark the box labeled **Relative** you can specify a relative position. In that case e.g. (-32,0) means that the object is moved 32 pixels to the left. (0,32) means that the object moves 32 pixels down.
-  **Move to a random empty cell**. Moves the object to a random empty cell.

Creating and destroying objects

-  **Create a new object at (x,y).** Create a new object at position (x,y) (absolute or relative to the current object). You indicate both the position and the type of object to create.
-  **Destroy object.** Use this action to destroy yourself or other objects. The destroy event will be generated.
-  **Change into another object.** Use this action to change the instance (or all instances of a particular type) into a different object. This is largely the same as destroying yourself and creating a new object of the new type at the current spot except that no new instance is created and things like speed, timers, etc. stay the same. But for the current object the destroy event is generated and for the new one the create event. For example, in the provided game Pacman this event is constantly used to change the pacman object and to change the monsters into scared monsters.

-  **Destroy all objects at a position.** Destroys all the objects that exist at the indicated position. For example, if you have a bomb in the game and it explodes, you can kill all the object left, right, above and below it.

Other actions

-  **Set an alarm clock.** Here you can set an alarm clock such that after the indicated number of steps the corresponding alarm event happens. You indicate the value of the alarm and the number of the alarm clock (0-3). Most of the time you will just use alarm0. This can be used to let things happen after certain intervals. Often the alarm event sets the alarm clock again to let the thing happen again some steps later. You can also set the alarm clock of another object.
-  **Set the score.** Here you can set the score or add to the score (relative). The score will be displayed in the form caption unless you disable this in the game options.
-  **Show the highscore.** This action shows the highscore table. If the current score is higher, the player can fill in his or her name. Best use when the player dies in the game, that is, just before stopping the game. The highscore table is saved.
-  **Play a sound.** You are asked for the name of the sound. See below how to add sounds to your games. You can also select <stop sound> to stop the sound that is currently playing.
-  **Show a message.** Displays a message box containing the message string. Game play will be interrupted until the player presses **OK**. Can be used to give certain instructions during the game. Use # in the string to indicate a new line.
-  **Go to another room.** Indicate the number of the room (absolute or relative to the current room number). This can be used to create games with multiple rooms or levels. For example, associate such an action with the meeting event of the man with a door. To restart the current room, move relative to room 0.
-  **Sleep a while.** Specify the time in milliseconds (i.e. 1000 is one second). Game play is interrupted for this period.
-  **End the game.** Ends the game.
-  **Set a variable.** Set a variable to a value. Give the name of the variable and the new value. Choose relative to add the value to the variable. By setting variables you can control many aspects of your game that cannot be controlled otherwise. For example, by setting the variable visible to 0 the object is invisible. By setting the variable image_scale to 2 the object appears twice as large. There are lots and lots of built-in variables. Also you can define your own variables that you can then later use e.g. when setting a speed or position for an object. See below for more information on variables.
-  **Provide some code.** Here you can provide a piece of code that is executed. Using code is the most advanced mechanism in *Game Maker*. In code you can do anything actions can do, and a lot more. See part II of this document for more information on using code.

Conditionals

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

-  **If there is no collision at (x,y).** You specify a position (relative to the current position or absolute). If moving there will cause no collision, the next action or block of actions is performed. This can e.g. be use to check whether a position is collision-free before going there.
-  **If there is a collision at (x,y).** You specify a position (relative to the current position or absolute). If moving there will cause a collision, the next action or block of actions is performed. So this is the reverse of the previous action.

-  **If there is a particular object at (x,y).** Again you specify a position, but also an object. If that object occurs at that position (that is, the current object will meet it when moved there) the next action is performed.
-  **If the number of instances is a value.** You provide the number and the object. If the number of instances of that object is equal to the number the next action is performed. For example, if there are no coins left (number is 0) you can do something.
-  **If question to player.** You specify a question for the player. If the player answers yes, the next action is performed.
-  **If expression.** Here you can type in an arbitrary expression (see below). If it is true, the next action is performed. You can use this to e.g. check the value of a variable that you set.
-  **Begin block.** Use this symbol immediately after a conditional to create a block of actions.
-  **End block.** Place this at the end of the block.
-  **Else.** After this action you can specify an else part for a condition. If the condition is false the action (or block of actions) after it will be skipped and the action (or block of actions) after the else action are performed. If the condition is true, the action after the else action is skipped.
-  **Exit event.** This action exits the event, that is, no further actions in the event are performed. This can be useful after a conditional.

Drawing actions

Normally you don't need to specify your own drawing actions. Objects are drawn automatically. But if you do, using the drawing event, you can use the following special drawing actions. (These have no effect in other events.)

-  **Draw an image.** You specify a position (relative to the current position or absolute). Also you need to indicate the object whose image you want to draw. So it is possible to draw somebody else's image. This is useful for having objects have different appearances.
-  **Draw a text.** You specify a position (relative to the current position or absolute) and the text to draw. If the text starts with a quote or double quote, it is seen as a string expression. For example, if you specify as text `'Score: ' + string(score)` it will display the current value of the score. See below for more information on expressions.
-  **Set the font.** You specify a font in the usual way. This font is further used for drawing texts.

Using expressions and variables

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

- **x** the x-coordinate of the instance
- **y** the y-coordinate of the instance
- **hdir** the horizontal direction (-1 = left, 0 = no motion, 1 = right)
- **vdir** the vertical direction (-1 = upwards, 0 = no motion, 1 = downwards)
- **hspeed** the horizontal speed (in pixels per step)
- **vspeed** the vertical speed (in pixels per step)
- **direction** the current direction of motion in angles (see the set motion action)
- **speed** the current speed in this direction
- **alarm0 – alarm3** the values of the alarm clocks (in steps)
- **score** the current value of the score
- **visible** whether the object is visible (1) or invisible (0)
- **image_scale** the amount the image is scaled (1 = not scaled)

You can change such 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 are local to the current instance. That is, each object has its own copy of it. 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 if it. So e.g. if you want a ball to move to the place where the coin is you can set the position to (coin.x,coin.y). In the case of a collision or meeting event you can refer to the x-coordinate of the other object as other.x. In conditional expressions you can use comparisons like < (smaller than), >, etc.

In your expressions you can also use function. For example, the function random(10) gives a random number below 10. So you can set e.g. the speed or direction of motion to a random value. Many more functions exist. For complete information on expressions and functions see part II of this document.

Using code

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

Parent object

You might have noticed that, when defining objects, there is another field called **Parent**. This is a very powerful, but somewhat complex concept. So you might want to skip this section on first reading.

Every object can have a parent. If a parent is defined the object inherits its behavior from the parent object. This means that all actions for events defined in the parent object also apply to this child object. This is useful in many cases. For example, assume you have three different types of monsters that look differently but all behave the same. Now you only have to define all actions for one of them and for the other two indicate that the first monster is the parent. They will inherit all the behavior of the first monsters.

This also works in meeting and collision events for other objects. Again in the above example, for my person object I only have to define a meeting event with the first monster. Meetings with the other monsters inherit the meeting actions.

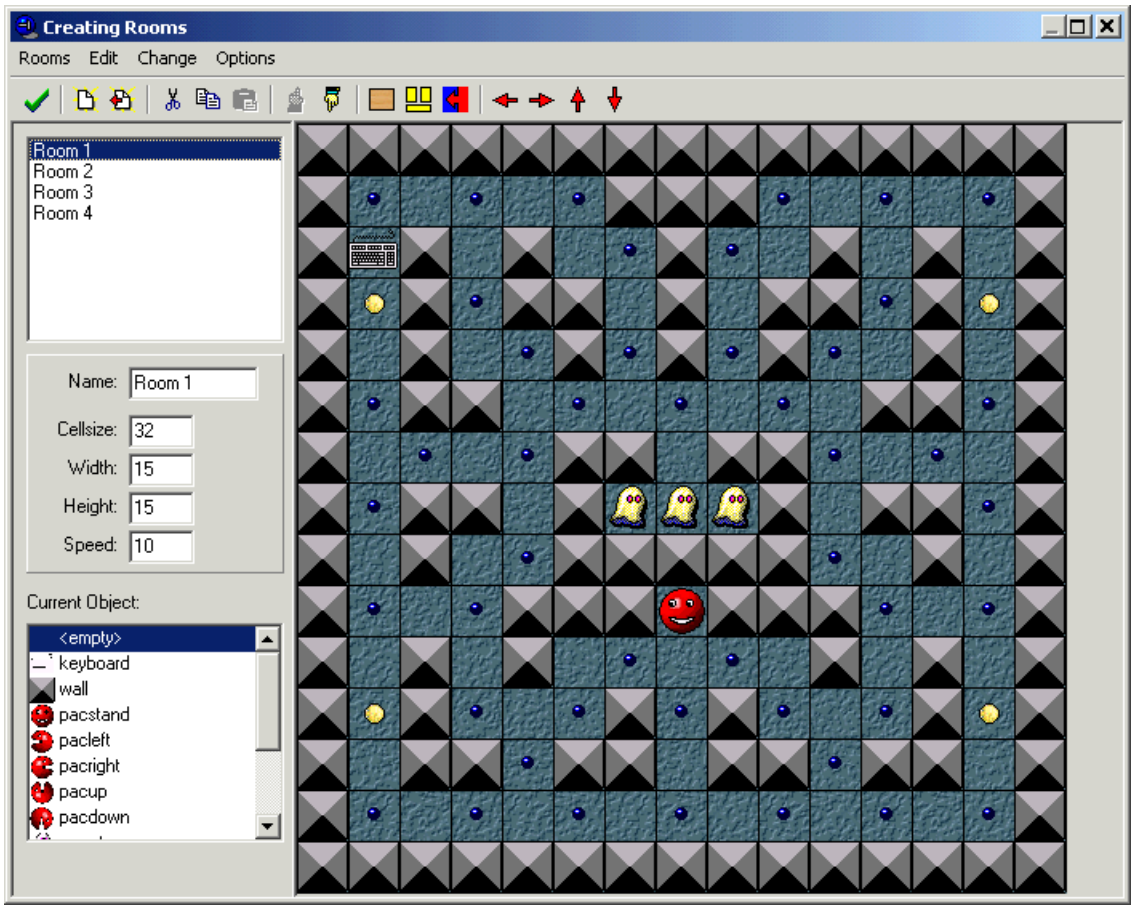
What is more, you can override certain events. That is, when in the child the object you put actions in some events these will override the actions in the parent object. So for this particular event the parent actions are not performed. For example, again with the monsters, you might want all the monsters to move in different directions but for the rest behave the same. This can be achieved by defining the motion in the creation event of the two child monsters. So they will execute their own create event but for the rest inherit the behavior.

Parent objects can again have parents, inheriting their behavior for as far as they don't override it. In this way you can build whole hierarchies. Be careful that no loops appear. (Actually, the program checks for this and warns you and corrects it.)

The use of parent objects and inheritance is very powerful and, when applied correctly will make your game more structured and saves a lot of duplication of actions (which easily leads to errors).

Creating rooms

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



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

At the right there is the room. You place objects in the room by choosing the object in the list at the bottom left, and then clicking with your mouse in the correct cell. With the right mouse button you can clear cells. Normally, each cell can contain only one object. So when you add an object, the existing object is removed. You can avoid this by holding the <Ctrl> key when clicking with the mouse. But be careful. Because the objects are drawn on top of each other you might not see them. You can also change the order of the instances added. If you hold the <Ctrl> key while clicking the right mouse button on a cell, the bottommost instance in the cell is moved to the top. If you hold the <Alt> key the topmost instance is removed to the bottom.

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

Room settings

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

- **Name.** The name of the room that will be shown in the caption, when running the game.
- **Cellsize.** The size of a cell in the room in pixels. Normally this is 32 but if you want to use smaller images or want more precise control over the position, you might want to change this (some of the provided examples use 24).
- **Width.** The horizontal number of cells.
- **Height.** The vertical number of cells.

- **Speed.** The game speed, that is, the number of steps per second. (If your machine is slow this number might not really be achieved.)

Defining the Background

If you click on the button or menu item labeled **Background** a form is shown in which you can set a number of aspects of the background. You have the following options:

- **No Background.** This does what you expect: no background is shown. The background will be black. This is useful when your instances cover the whole background. In this case it save some drawing time.
- **Single Color.** Click here to have a colored background. Click on the box to change the color.
- **Two colors.** Use this to have a gradient fill of the background between two colors. You can specify both colors and whether the fill should be horizontal or vertical. (Realize that drawing gradient fills is relatively slow. It is normally faster to create a background image with the required fill.)
- **Image.** Click here to have a background image. Click on the box to indicate the filename for the image. You can use many different types of images (icons, bitmaps, jpeg files, gif files, and metafiles). There are four ways in which the image can be placed:
    - **Left-Top.** The image is placed in the left top. If it does not fill the entire room the rest has no background.
    - **Tiled.** Copies of the image are placed next to each other to fill the entire background.
    - **Stretched.** The image is stretched over the background.
    - **Scrolling.** In this case you can make a scrolling background. You can set the horizontal and vertical scrolling speed. The image is always tiled in this case.

  Note that background images are stored in the data folder for the game in a format that can not be read by any other program than *Game Maker*.

Defining Views

Normally, *Game Maker* shows the whole room when you are playing the game. In many games though you do not see the whole playing field but only part of it. You explore the area by moving some object through the room. The part of the room you see scrolls automatically with the moving object. You can achieve this in *Game Maker* in a very simple way. First click on the button or menu item **Views**. A new form will pop up that looks rather empty. Click on the checkbox **Use Views** and suddenly a lot of information appears. Let us first only look at the tab **Primary View**. When you want to use a restricted view this is all you need. Only the following fields are important (leave the others 0):

- **Width.** The horizontal number of cells in the view. Make sure it is not larger than the width of the room.
- **Height.** The vertical number of cells in the view.
- **Hor Border.** The number of cells that must remain visible to the left and the right of the object. If this is 0 the room will start scrolling only when the object reaches the left or right boundary. Otherwise, it starts scrolling earlier. Normally, a setting of 1 or 2 is the best.
- **Vert Border.** The number of cells that must remain visible above and below the object.
- **Object.** The object that must always remain visible. The view scrolls when this object moves around. You can set it to noone, but this is only useful if you control the view from within a piece of code (see part II).

You can also display multiple views of the same room. There are a number of uses for this. One use is multi-player games. For example, you might have two persons racing against each other. For both racers you want to show the part of the room they are in. Another use is when you have a scrolling part and a fixed part (e.g. with status information). You can place both parts in the room (preferably a bit away from each other) and then use two views to show them at the correct place on the screen. *Game Maker* allows you to have four different views. For each view there is a tab. You can again specify the values indicated above but now you also need to specify where the view appears on the screen. For this you use the following fields:

- **Left.** This specifies the position of the view on the screen. A value of 0 is at the left. A higher value indicates the number of cells left of the view. So, for example, if your first view has a width of 8 and you what the second view to be to the right of it (with a 1 cell margin between them), use a value of 9 for Left.
- **Top.** The number of cells above the view. Can be used to place a view below another view.

Make sure that views don't overlap. You can though leave a margin between them.

If a view must show a fixed part of the room you can use the fields **Left Cell** and **Top Cell**. This specifies the number of cells in the room that lie to the left and above the view. This only makes sense if you did not specify any object to remain visible.

Normally you want the views you use to be visible at the start of the room. To this end check the checkbox at the top. You can also make views visible and invisible from a piece of code. The fist view is default visible but this can be switched off. The others are default not visible.

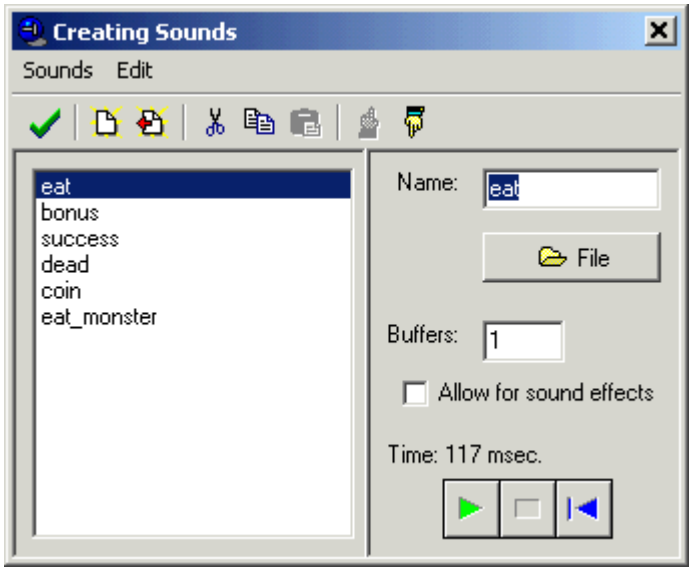
Realize that when you have multiple views the room is drawn multiple times each step. This can slow things down a bit. When you use the mouse the position is correctly calculated, dependent on the view the mouse is in. (If the mouse is in no view, the position is calculated relative to the primary view, even if that view is not visible.)

Transitions

Sometimes you don't want the new room to appear just like that, but you like it to sort of slowly appear. For this you can use the room transitions. You find them under the **Transition effects** item in the **Change** menu. There are a number of different possibilities that create the new image from one of the sides or slide it in from a side. Experiment with the best effects. You can indicate the number of steps. More steps make it smoother but might take too long on slow computers. You can also indicate how long the transition should take (in milisseconds). (This is a lower bound because it also depends on the drawing speed.)

Creating sounds

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



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

There are two types of sound files that can be used: midi files (extension .mid) wave files (extension .wav). Midi files are typically used for background music. They will loop forever. (Please note that there is an easier mechanism for background music. See the section on game options below.) Wave files are for short sound effects. To add a sound effect to a particular event, use the sound action.

You will also see a line reading **Buffers**. This value can normally be kept to 1. It is useful to set **Buffers** to a higher value if the same sound must be played multiple times simultaneously. For example, a shooting sound might overlap with itself when the player fires the gun in rapid succession. In this case you need a value higher

than 1. (Only use this when strictly necessary because it uses resources.) There is also the check box labeled **Allow for sound effects**. When you check this box, from within code you can use sound effects like changing the volume or the frequency. For most games this is not necessary. See part II of this document for more information about sound effects.

Creating game information

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

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

Game options

There are a large number of options you can set for your game. These option provide a simple mechanism to change the visual appearance of the game, to generate background music, and to display start an ending screens. To change the game options press the button **Game Options**. A form appears with a number of tabbed pages. We will discuss them here:

General options

Here you can set the following options:

- **Start in full screen mode.** If checked the game will appear in full screen mode, scaled, and without a window caption. This makes the game look very professional.
- **Scale the image in full screen mode.** If checked the graphics is scaled when using full screen mode. Also the user can scale the window. If you uncheck this that no scaling is done. This can speed up the game and give a better visual appearance in some cases.
- **Use DirectX exclusive mode.** When checked the games are run in exclusive mode. In this mode you can indicated the required screen resolution. No other windows can interfere with the program. This mode is normally fast and you can guarantee that your game always runs on the other machine. Be very careful though! In exclusive mode no other windows can be shown. This means that you should not use actions or code that displays dialogs (like showing a message, showing the highscore table, or showing some splash screen). Also some warnings are not displayed and debugging is not possible in this mode.
- **Display the cursor.** Indicates whether the cursor should be displayed. Especially for full screen games you often don't want to show the cursor. It distracts attention and makes the graphics slower.
- **Show the score.** Indicates whether to automatically display the score in the caption or the window. Switch this off if your game does not use scores or if it has its own mechanism of displaying scores.
- **Let <F4> switch between screen modes.** Indicates whether you allow the player to switch between full screen and windowed mode using the F4 key.
- **Let <F1> show the help file.** Indicates whether the player can display the help file using the F1 key. If there is no standard help file or if you want to provide different behavior for F1 you can remove the check mark.
- **Let <Esc> end the game.** Normally the player can end the game by pressing the Esc key. If you don't want this (for example because you want to save some information when the user quits) or if you want to use the Esc key for other behavior, remove the check mark.
- **Let <F5> and <F6> save and load the game.** Normally the player can save his current position in the game using the <F5> key and load the previously saved position using the <F6> key. Here you can switch this off in case you don't want the player to be able to save his position.
- **Generate continuous keyboard events.** When checked (default) keyboard event are continuously generated as long as the key is kept pressed by the player. If you unckeck this box, only one event occurs (until the keyboard start repeating the key).
- **Generate continuous mouse events.** When checked (default) mouse events are continuously generated as long as the mouse button is kept pressed by the player. If you uncheck it, only one event is generated.

Music

Many games have some background music that is being played. Such background music gives the game some atmosphere. Choosing appropriate background music enhances the pleasure of playing the game. Background music typically uses midi files. In the music tab of the game options form you can easily specify the background music. First indicate that you want the game to have automatically played background music. Next you can specify a list of pieces. Add pieces by pressing the **Add** button, delete them using the **Delete** button, and move them up and down in the list using the other two buttons. Finally you can indicate whether the pieces must be played in the given order (starting again with the first one once the last piece is played) or in some random order. The pieces will automatically be copied to the games data folder once you save the game.

Start and end screen

Many games start with a screen giving some information about the game, before the actual game is played. Such screens can contain an image, some text, or even a little video. It is trivial to do this in *Game Maker*. First select the tab Start screen in the Options form. Now select the type of start screen. For each type you can indicate the file name (rtf for text), and whether it should be shown in full screen mode or windowed. For images or text you can specify the delay before the game should start automatically. (The game always starts if the user clicks with the mouse in the screen or presses a key.) For video you can indicate whether the game must be started after the video is played once or whether the video should loop till the player presses a key. For text you can also specify the background color.

Some videos come with their own sound. In this case you probably don't want the background music to interfere with that. This can be solved by indicating that the background music must be started after the screen is closed.

In a similar way you can define an ending screen, which will typically contain some acknowledgements (e.g. that the game was created with *Game Maker*).

Other options

Here you can set some other options that influence the behavior of the game.

- **Number of collision tests.** When there are collisions, the corresponding actions are performed. This can again result in collisions (or not resolve the collisions). By putting a number higher than 1 here, *Game Maker* will try multiple times to resolve the collisions. After the number of tries is done, objects that still cause collisions are placed back at their previous location. In previous releases of *Game Maker* 16 tries were used. For most games though this is completely useless and it takes a lot of time. So the new default is to try it just once. You can though change this value.
- **Stop moving when collisions are not resolved.** Normally, if the collisions are not resolved after the number of tries indicated, the objects are placed back at the previous place. By unchecking this box that does not happen. The result is that objects will go through the solid objects. This is sometimes the behavior you want. But, if you are sure that your collision and meeting events resolve all collisions you should also uncheck this box because it save a time consuming extra test, making your games faster.
- **Use precise, pixel-wise, collision checking.** By checking this box, collision tests are done exactly, on pixel basis rather than using the bounding box. This will be a bit slower but it is a lot more accurate. (Please note that the original images are used for the testing with scaling taken in to account. So if you use the drawing event to draw something else, the collision checker does not see this!)
- **Preprocess for precise collision checking.** When you want precise collision checking for objects normally the structure for doing this is only created once it is needed. This might cause a little hiccup in the beginning of the game. To avoid this, check this box. It will though increase the time before the game starts and it will increase storage requirements.
- **Highscore background.** Here you can indicate whether the highscore form should have a background image and, is so, indicate the file name of the.
- **Highscore form border.** Here you can indicate whether the highscore form must have a border.
- **Freeze when form is not active.** When checked the game freezes when the game form is not active. So e.g. when the user brings another window to the foreground.
- **Discard video memory after each room.** When playing the game all images of the objects are place in video memory such that they can be drawn fast on the screen. When you have many objects with large images and you use only a few of them in each room this wastes a lot of space. In this case you can check this option. Now only the images required in each room are used. Starting a room though takes a bit longer and when a new object appears in a room there can be a slight hiccup.

Distributing your game

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

Please mail your creations to gamemaker@wanadoo.nl such that I can place them on the web site.

If you want to create a version of the game that does not require *Game Maker* to be installed on your computer, open the game and choose the menu item **Create stand-alone**. You are asked for the place where to create the stand-alone game. Indicate the directory and you are done. If your game was named XXX, at the indicated place a new directory XXX has been created in which you will find a program called XXX.exe (plus a game file game.gmf and a directory called game_data that stores the game data). Executing XXX.exe will play your game. Note that in the stand-alone version the menu and toolbar are not shown. The player can view the help file by using <F1>. If the game ends, the program ends as well. If you put an image named loadimage.bmp in the game data folder, this picture will be shown while the game is loading, rather than the boring standard text.

Stand-alone versions of games are protected in the sense that the game can no longer be loaded into *Game Maker*. So in this way you can distribute your work without running the risk of other people using it for their own purposes.

You are free to distribute the stand-alone version in any way you like. You can also sell it.

Part II

Advanced Game Design

Introduction

Using the standard commands in *Game Maker* you can make rather interesting games. But to make more complicated games you will need to do a little programming. *Game Maker* has a built-in programming language in which you can control all aspects of the game. Using the language, you have much more precise control over the game graphics, over the way objects interact and over the way your game interacts with the player. In this way, with relatively little effort, you can make games that can hardly be distinguished from professional games.

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

This document describes the language used. It is a subset of C, with a few additions that are specific to *Game Maker*. The language contains variables (real numbers or strings) and control structures (conditionals, loops). There are a large number of built-in variables and functions that relate to the game. First the basic syntax of the language is described. Next we delve into the details of how to control the game and the graphics.

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

Creating code

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

The code editor

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

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

To make life easier, it is possible to let the code editor color the code with different colors. In this way you see a clear distinction between normal text, comments, and keywords. Also built-in functions and variable names, constants, objects, and sounds are colored. This helps you in seeing whether the names you types are correct. You can switch color-coding on and off in the **Options** menu. Here you can also change the colors used and the font. Settings are saved.

Debugging the code

It is easy to make errors when writing code. *Game Maker* helps you in a number of ways to find errors. First of all, inside the code editor you can check your code. To this end press the button **Check the code**. It will detect a

number of errors in your code, like wrongly typed keywords, missing brackets, or non-existing functions (or functions with the wrong number of arguments). Always check your code when you are ready with it. It helps a lot. Using color coding will also help you because the colors will immediately tell you when you e.g. reuse a built-in variable name or use an non-existing function.

Some errors can only be detected when the program is running. A form will show up indicate the error and where it occurred. You can either abort execution of the game, or continue to maybe gather some more information.

If your game is not working the way you expect, you often like to check certain variables or expressions. For this *Game Maker* contains a little debugger. To start it, choose **Debug** from the **Play** menu. A form opens in which you can indicate the information you want to see. At the top you always see the number of frames per second (the real speed of the game) and the total number of instances. To see other information, click on the button **Add**. Now you can type in some expression, e.g. ball.x to see the x-coordinate of the ball. Press OK and the expression will be shown in the list, together with the current value. You can add as many expressions as you like. You can also enter more complicated expression or use function. For example, as you will see below number(ball) will show you the number of the balls. Selecting an expression and using the button **Change** you can change the expression. With **Delete** you can delete it, and with **Delete All** you can clear the list of expressions.

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

The language

When you want to use code, there are a couple of things you have to be careful about. First of all, for all your objects and sounds you must use names that start with a letter and only consist of letters, digits and the underscore '_' symbol. Otherwise you cannot refer to them from within the code. Also be careful not to name objects self, other, global, or sound because these have special meaning in the language.

Program

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

```
{
  <statement>;
  <statement>;
  …
}
```

A statement can again be a block of statements. There are a number of different types of statements, which will be discussed below.

Variables

Like any programming language there are variables. Variables can store either real values or strings. Variables do not need to be declared. There are a large number of built-in variables. Some are general, like mousex and mousey that indicate the current mouse position, while all others are local to the object instance for which we execute the code, like x and y that indicate the current position of the instance. A variable has a name that must start with a letter and can contain only letters, numbers, and the underscore symbol '_'. (The maximal length is 32 symbols.) When you use a new variable it is local to the current instance and is not known in code for other instances (even of the same object). You can though refer to variables in other instances; see below.

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

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

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

Expressions
Expressions can be real numbers (e.g. 3.4), strings between single or double quotes (e.g. 'hello' or "hello") or more complicated expressions. For expressions, the following operators exist (in order of priority):

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

As values you can use number, variables, or functions that return a value. Sub-expressions can be placed between brackets. All operators work for real values. Comparisons also work for strings and + concatenates strings.

Example
Here is an example with some useless assignments.

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

Note that ! in the last expression means not.

Variables in other instances
As stated above, variables you create or refer to are local to the current instance (except for some predefined global variables). You might want to use and set variables in other instances. This can be achieved by preceding the variable name by the name of the object and a dot. So, for example, to address the x-coordinate of the ball use

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

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

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

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

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

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

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

See below for more information on this construction.

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

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

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

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

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

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

Arrays
You can use 1-dimensional arrays. Simply put the index between square brackets. 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. So for example you can write the following:

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

If statement

An if statement has the form

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

or

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

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

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

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

Repeat statement

A repeat statement has the form

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

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

Example
The following code creates five balls at random positions.

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

While statement

A while statement has the form

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

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

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

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

Exit statement

The exit statement simply ends the execution of this piece of code. (It does not end the execution of the game! For this you need the function `end_game();` see below.)

Functions

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

```
<function>(<arg1>,<arg2>,…)
```

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

Forall constructions

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

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

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

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

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

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

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

If you want to perform some code for all instances of all objects you can use the construction:

```
forallinstances <statement>
```

So for example, to move the whole scene a bit to the right you can write

```
forallinstances x += 8;
```

Addressing individual instances

As was indicated before. You can in general only address all instances of a particular object. But sometimes it is important to be able to address one particular instance. This can be achieved as follows. Every instance has a unique identifier (a number). This number is created when the instance is created and it never changes. New instances get other numbers. The identifier of an instance is stored in the variable `id`. When you create an object from within code using the function `create()` the identifier is returned. You address the variables of instance number n by preceding them with `instance[n].` For example, the following piece of code creates an instance of object wall and makes it invisible:

```
{
  iii = create(0,0,wall);
  instance[iii].visible = false;
}
```

After executing this code only the newly created wall is invisible, not all the others. In this way you can read and set all variables in the instance. If you want to do other things, like calling functions for the one instance, you can use the `with` construction.

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

The expression should result in the identifier of an instance. For example, in the above example, we could move the wall to a random position by adding

```
  with (iii) move_random(self);
```

(Note that it is not allowed to use `move_random(iii)` or `move_random(instance[iii])`. The function `move_random()` requires an object as an argument, not an instance or an identifier.)

Comment

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

Pascal style

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

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

Game actions

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

Moving objects around

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

If the object moves with constant speed and direction, there is an easier way to do this. Each object instance has a horizontal speed (hspeed) and a vertical speed (vspeed). Both are indicated in pixels per step. Secondly, there is a horizontal direction (hdir) and a vertical direction (vdir). After each step the program sets x = x + hspeed*hdir and y = y + vspeed*vdir. So you have 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. You can set and read these variables to specify an arbitrary motion. (Internally this is changed into values for hdir, etc. To be more precise, it sets both hdir and vdir to 1 and sets hspeed, and vspeed to the appropriate values.) Also there is the friction and the gravity and gravity direction. Finally, there is the function add_motion(dir,speed) to add a motion to the current one.
To summarize, each instance has the following variables, and functions:

- **x**: The x-coordinate.
- **y**: The y-coordinate.
- **xprevious**: The x-coordinate in the previous step (useful to go back).
- **yprevious**: The y-coordinate in the previous step.
- **hdir**: The horizontal direction (-1 = left, 0 = no motion, 1 = right).
- **vdir**: The vertical direction (-1 = upwards, 0 = no motion, 1 = downwards).
- **hspeed**: The horizontal speed (in pixels per step).
- **vspeed**: The vertical speed (in pixels per step).
- **direction**: The current direction (in angles, counter clockwise; 0 is to the right).
- **speed**: The current speed.
- **friction**: The current friction.
- **gravity**: The current gravity.
- **gravitydir**: The direction of the gravity.
- **set_motion(dir,speed)**: Sets the motion (is the same as assigning the values to the variables direction and speed.
- **add_motion(dir,speed)**: Adds an amount of motion in direction dir to the current motion.
- **bounce()**: Bounce against the other object. Only works in collision and meeting events when the other instance is solid.
- **move_towards(x,y,speed)**: Let the instance move with its reference point towards position x,y with the indicated speed. This can e.g. be used to let one instance (say a bullet) move towards another object (say a monster).
- **move_contact()**: If the instance is in collision with another instance, it is moved back until it just touches the other instance. This is very useful to let the instance e.g. land on the ground. Only works in collision and meeting events where the other instance is solid.

To help you keep your objects at the right places, the following variables and functions exist:

- **roomwidth**: The width of the room in pixels (cannot be changed).
- **roomheight**: The height of the room in pixels (cannot be changed).
- **cellsize**: The size in pixels of a cell (cannot be changed).
- **is_aligned()**: Returns whether the instance is aligned with the grid.
- **align()**: Aligns the current instance with the grid.

Two functions that can help you to determine a direction or speed are:

- **point_distance(x1,y1,x2,y2)**: Returns the distance between the points (x1,y1) and (x2,y2).
- **point_direction(x1,y1,x2,y2)**: Returns the direction (0-360) from point (x1,y1) to (x2,y2).

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

- **is_free(x,y)**: Returns whether the position (x,y) is collision-free for the instance, that is, when we place the instance at position (x,y) no collision with a solid object occurs.
- **is_empty(x,y)**: Returns whether the position (x,y) is empty for the instance. The only difference with the previous function is that in this case non-solid objects are taken into account.
- **is_meeting(x,y,obj)**: Returns whether the instance meets object obj when placed at position (x,y).
- **nothing_at(x,y)**: Returns whether there is no object at position (x,y). The difference with is_empty(x,y) is that now the size of the current instance is not taken into account.
- **object_at(x,y,obj)**: Returns whether there is an instance of object obj at position (x,y).
- **distance_to_point(x,y)**: Returns the distance of the bounding box of the object towards the point (x,y). The result is 0 if the point (x,y) lies in the bounding box.
- **distance_to_object(obj)**: Returns the distance between the bounding box and the nearest bounding box of an instance of object. The result is 0 if the bounding boxes intersect. If no instance of obj exists 1000000 is returned.

Determining whether object instances intersect each other or whether a point (like the mouse) lies in an object is normally done using the bounding box of the image. (If there are multiple sub-images a bounding box is used that surrounds all of them.) Sometimes this is not what you want. *Game Maker* gives you the possibility to change the bounding box. But even this might not be enough. In that case it is also possible to indicate that for a particular instance an exact intersection test is performed, comparing the pixels themselves.

- **bb_l**: Left side of the bounding box (with respect to the top left corner of the object, indicated by position (x,y)).
- **bb_r**: Right side of the bounding box.
- **bb_t**: Top side of the bounding box.
- **bb_b**: Bottom side of the bounding box.
- **set_collision_mode(val)**: Determines the way of doing collision. 0 is fast, using bounding boxes. 1 is slightly slower, using the pixels themselves (still first a bounding box test is done). This call only has effect if you did not enable the precise collision option in the game options form.

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

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

Creating, changing, and destroying instances

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

- **create(x,y,obj)**: Creates an instance of object obj at position (x,y).
- **destroy(obj)**: Destroys all instances of object obj (if obj=self or obj=other then only the self instance or the other instance is destroyed).
- **change(obj1,obj2)**: Changes all instances of obj1 into object obj2.
- **change_at(x,y,obj)**: Changes all instances at position (x,y) in object obj, that is, all object instances whose bounding box contains position (x,y).
- **destroy_at(x,y)**: Destroys all instances at position (x,y).
- **number(obj)**: Returns the number of instances of object obj.

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

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

Timing

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

- **gamespeed**: The speed of the game in steps per second (change this to speed up or slow down the game).
- **fps**: Actual number of frames per second (cannot be changed).
- **current_time**: Number of milliseconds since the system was started (cannot be changed).
- **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 4 different alarm clocks that you can set. To change the values (or get the values) of the different alarm clocks use the following variables:

- **alarm[0]**: The value of the alarm clock 0 (in steps).
- **alarm[1]**: The value of the alarm clock 1 (in steps).
- **alarm[2]**: The value of the alarm clock 2 (in steps).
- **alarm[3]**: The value of the alarm clock 3 (in steps).

So alarm is actually an array with 4 elements.

Rooms and score

Games work in rooms. Rooms are numbered from 1 to lastroom. The current room is stored in variable room. To restart the same room just write room = room or goto_room(room). Make sure rooms you go to exist. So a typical piece of code you will use is:

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

Summarizing, the following variables and functions exist.

- **room**: The number of the current room. You can also change this variable to change the room.
- **lastroom**: The number of the last room. You cannot change this variable.
- **goto_room(numb)**: This function makes the game go to the room number numb.
- **find_room(str)**: With this function you can find the room number for a room with the indicated string as name. If such a room does not exist, the value 0 is returned.
- **end_game()**: Ends the game.

- **gamename**: The name of the game. You cannot change this variable.
- **roomname**: The name of the current room. This name is displayed in the window caption. You can change this to change the window caption!

Another important aspect of many games is the score. *Game Maker* keeps track of a score in a global variable score, which is shown in the window caption. You can change the score by simply changing the value of this variable. If you don't want to show the score in the caption, set the variable showscore to false. (You can also change this in the options form.) For more complicated games you better display the score yourself.

There is also a built-in mechanism to keep track of a highscore list. It can contain up to 10 names. There are routines to add players to the list, to clear the list and to show the list. Also you can change font and color.

- **score**: The current score (increase the score with e.g. score += 10;).
- **showscore**: Whether to show the score in the caption.
- **highscore_show(numb)**: Shows the highscore table. Numb is the new score. If it is higher than one of the scores in the table, the player can enter his or her name. Normally you would use this as highscore_show(score), but to e.g. just show the highscore list, use highscore_show(-1). The highscore list is automatically saved with the game.
- **highscore_clear()**: Clears the highscore list.
- **highscore_add(str,numb)**: Adds a player with name str and score numb to the highscore list (without showing the list).
- **highscore_setcolor(col1,col2)**: Use col1 for the color of the first player and col2 for the other players in the list (see below for the way to indicate colors).
- **highscore_setfont(str)**: Use str as fontname in the highscore list. (Make sure it exists.)
- **highscore_value(place)**: Returns the score of the person on the given place (1-10).
- **highscore_name(place)**: Returns the name of the person on the given place (1-10).

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 change these keys and, what is better, you can save and load games from within a piece of code.

- **loadkey**: Key used for loading the game (set to 0 to disable this).
- **savekey**: Key used for saving the game (set to 0 to disable this).
- **load_game(str)**: Loads the game from file with name str. (Must be in the folder of the game.)
- **save_game(str)**: Saves the game situation to file with name str. (You cannot specify a path, just the file name.)

User interaction

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

- **mousex:** x-coordinate of the mouse.
- **mousey:** y-coordinate of the mouse.
- **mousebutton:** Currently pressed mouse button. As value use mb_none, mb_left, mb_middle, or mb_right.
- **lastkeypressed**: The keycode of the last key pressed on the keyboard. See below for keycode constants. You can change it, e.g. set it to 0 if you handled it.
- **keypressed**: The keycode of the currently pressed key (0=no key pressed).
- **lastcharpressed**: This string variable gives you the character that was last pressed.
- **inputstring**: This string contains the last 80 visual characters pressed (backspace is interpreted as removing the last character).

To check whether a particular key or mouse button is pressed you can use the following functions:

- **check_key(keycode)**: Returns whether the key with the particular keycode is pressed.

- **check_mouse_button(numb)**: Returns whether the mouse button is pressed (use as values mb_left, mb_middle, or mb_right).

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

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

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

The following constants for virtual keycodes exist:

- **vk_nokey** check_key returns true when no key is pressed
- **vk_anykey** check_key returns true when 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_divide, vk_multiply, vk_add, vk_subtract, vk_decimal** the other keys on the numeric keypad
- for the letter key use the asci value of the capital, e.g. ord('D')
- for the number keys, just use the asci value of the number, e.g. ord('5')

For low-level checking of keys there is another routine that directly checks the hardware state of the key. The result is independent from which form or application has focus.

- **check_key_direct(keycode)**: 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!).

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

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

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:

- **continuous_keyboard**: Indicated whether keyboard events should be continuous (1) or not (0).
- **continuous_mouse**: Indicated whether mouse events should be continuous (1) or not (0).

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

- **show_message(str)**: Displays a dialog box with the indicated string as a message.
- **show_question(str)**: Displays a dialog box with the string as a question; returns true if the user selects yes and false otherwise.
- **get_integer(str,def)**: Asks the user in a dialog box for a number; str is the question, def the default value. It returns the value given by the user.
- **get_string(str,def)**: Asks the user in a dialog box for a string; str is the question, def the default string. It returns the string typed in.
- **show_info()**: Displays the information form for the game.

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

- **check_joystick_direction()**: Returns the keycode corresponding to the current joystick direction (vk_numpad1 to vk_numpad9).
- **check_joystick_button(numb)**: Returns whether the joystick button is pressed (numb in the range 1-4).
- **get_joystick_xpos()**: Returns the precise position of the joysticks x-axis. The result lies between –1 (completely to the left) and 1 (completely to the right). 0 is the center. Most joysticks though are not completely centered so you might want to create some "dead zone" around 0 where you don't do anything.
- **get_joystick_ypos()**: Returns the joysticks y-position.
- **get_joystick(zpos()**: Returns the joysticks z-position (if it has a z-axis).

Splash screens and other pop-ups

As you probably read, in the options form you can specify a start or an end screen to show. Such 'splash' screens with text, images or video are useful to give information to the player. You can actually show such splash screens at any moment during the game. The game is temporarily paused while the splash screen is shown. There are three different routines:

- **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 data directory of the game yourself. 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.
- **show_image (fname,full ,delay)** : Shows an image splash screen. fname is the name of the image file (.bmp,.gif,.jpg,.wmf). You must put this file in the data directory 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 data directory of the game yourself. full indicates whether to show it in full screen mode. loop indicates whether to loop the.

If the splash screens do not provide the functionality you want you can resort to executing a different program. There are two functions available for this: execute and shellexecute. The function execute starts a program (that must be located in the data directory for the game), possibly with some arguments. It can wait for the program to finish (pausing the game) or continue the game. The function shellexecute 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. The file must be located in the data directory for the game. Shellexecute cannot wait for completion so the game will continue.

- **execute(program,args,wait)**: Executes the program with the given arguments. If wait = true the program pauses the game till the program returns.
- **shellexecute(file,args)**: Opens the file with the standard associated program. The game will continue.

Computing things

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

Constants

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

Mathematical functions

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

String functions

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

- **chr(val)** returns a string containing the character with asci code val
- **ord(str)** returns the asci code of the first character in str
- **string(val)** turns the real value into a string
- **string_length(str)** returns the number of characters in the string
- **string_pos(substr,str)** returns the position of the first occurrence 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 starting at index of length count removed
- **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 upper and lowercase 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

Sounds

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

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

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

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

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

- **sound_volume(numb,value)**: Changes the volume for the indicates sound number (0 = low, 1 = high).
- **sound_pan(numb,value)**: Changes the pan for the indicates sound number (0 = left, 1 = right).
- **sound_frequency(numb,value)**: Changes the frequency for the indicates sound number (0 = low, 1 = high).

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* tries to use DirectSound. In this case all wave files are stored in memory and can have effects. If DirectSound is not available (like on NT4 machines), the standard system is used which means that only one wave file can be played at a time. *Game* Maker actually also tries to play other music files when you specify them, in particular mp3 files. It uses DirectShow for this. This enables sound effects and does streaming sound, that is, sounds do not need to be stored fully in internal memory, so you can use large mp3 files. If DirectShow is not available it uses the standard multimedia player. 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. You can use the variable `directsound` to check what is happening. A value of 2 means that DirectShow is working, 1 means that DirectSound is used and 0 means that system sound is used.

Game graphics

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

Window and cursor

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

- **fullscreen(full):** If full = true the game runs in full screen mode, otherwise it runs in windowed mode.
- **fullscreenmode:** This variable is true when in fullscreen mode. You can also change the mode by settting this variable to true or false.

Note that in full screen mode captions are not shown. The score is shown on the screen. (This can be avoided using the game options. See part I of the documentation.) Running in full screen mode is only reasonable when the computer supports DirectDraw. (Most do, but not all.) You can use the variable `directdraw` to check this.

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

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

By the way, note that it is very easy to make your own cursor object.

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

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

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

There is one rather special variable called transition. It determines how the next frame is drawn. 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). It only affects the next time a frame is drawn.

- **transition**: Indicates the next frame transition (0-13).
- **transition_time**: Time for the transition (in milliseconds).
- **transition_steps**: Number of steps for the transition.

The background

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

- **back_color1**: First background color (if there is no background image).
- **back_color2**: Second background color (used when there is a two color background).
- **back_x**: Horizontal offset of the background image.
- **back_y**: Vertical offset of the background image.
- **back_hspeed**: Horizontal speed of a scrolling background image.

- **back_vspeed**: Vertical speed of a scrolling background image.

The views

As you probably know you can define up to four 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 cannot create new views from within code but you can make views visible and invisible and change the place or size of the views on the screen (but realize that the size of the window is not changed; it is based on your initial views) or 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 visible object changes during the game. For example, you might change the main character object based on its current status. Unfortunately, this does mean that it is no longer the object that must remain visible. This can be remedied by one line of code in the creation event of all the possible main objects (assuming this must happen in the first view):

```
{
  view_object[1] = self.object;
}
```

The following variables exist that influence the view. All, except the first are arrays ranging from 1 (the first view) to 4 (the last view).

- **view_used**: Whether views are used (cannot be changed).
- **view_visible**: Whether the particular view is visible.
- **view_x**: Horizontal offset of the view in the room (in pixels).
- **view_y**: Vertical offset of the view.
- **view_l**: Horizontal position of the view in the window (in pixels).
- **view_t**: Vertical position of the view in the window.
- **view_w**: Width of the view in pixels.
- **view_h**: Height of the view.
- **view_hborder**: Size of the minimum horizontal border around the visible object.
- **view_vborder**: Size of the minimum vertical border around the visible object.
- **view_object**: Object that must remain visible.

The object image

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

- **visible**: If visible is true (1) the image is drawn, otherwise it is not drawn. Invisible object still are active and create collision and meeting events; you only don't see them. Setting the visibility to false is useful for e.g. controller objects (make them non-solid to avoid collision events) or hidden switches.
- **image_width**: Indicates the width of the image. This value cannot be changed but you might want to use it.
- **image_height**: Indicates the height of the image. This value cannot be changed but you might want to use it.
- **image_scale**: A scale factor to make larger or smaller images. A value of 1 indicates the normal size. Changing the scale also changes the values for the image width and height and influences collision events as you might expect. (It does not change the values of the bounding box variables!) Realize that scaled images (both smaller and larger) take more time to draw. Changing the scale can be used to get a 3-D effect.
- **image_number**: The number of subimages for the object (cannot be changed).
- **image_index**: When the image has multiple subimages the program cycles through them. This can be avoided by setting this variable to the index of the subimage you want to see (first subimage has index 0). Give it a value –1 to cycle through the subimages. This is useful when an object has multiple appearances.

- **image_speed**: The speed with which we cycle through the subimages. A value of 1 indicates that each step we get the next image. Smaller values will switch subimages slower, larger values will actually skip subimages to make the motion faster.
- **image_depth**: Normally images are drawn in the order in which the instances are created. You can change this by setting the image depth. The default value is 0. The higher the value the further the instance is away. (You can also use negative values.) Instances with higher depth will lie behind instances with a lower depth. Setting the depth will guarantee that the instances are drawn in the order you want (e.g. the plane in front of the cloud). Background instances should have a high (positive) depth, and foreground instances should have a low (negative) depth.

Advanced drawing routines

It is possible to let objects look rather different from their image. There is a whole collection of functions available to draw different shapes. Also there are functions to draw text. You can only use these in the drawing event of an object; these functions don't make any sense anywhere else in code. 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.)

- **draw_image(x,y,obj)**: Draws the image of object obj with top left at position (x,y). To draw your own image use draw_image(x,y,self). But you can also draw the image of another object. This is very useful when you want objects to have different appearances.
- **draw_subimage(x,y,obj,ind)**: Draws the subimage ind of the object. This only makes sense when the object is an animated image with multiple subimages. It can also be used to put different images for the same object in one animated image.
- **draw_tiled_image(x,y,obj)**: Draws the image of object obj tiled over the whole room. Position (x,y) gives the position with respect to which the image is tiled. This can be used to e.g. make a star field that moves over the room.
- **draw_line(x1,y1,x2,y2)**: Draws a line from (x1,y1) to (x2,y2).
- **draw_circle(xc,yc,r)**: Draws an circle with (xc,yc) as center and r as radius.
- **draw_ellipse(x1,y1,x2,y2)**: Draws an ellipse with (x1,y1) left top and (x2,y2) right bottom.
- **draw_rectange(x1,y1,x2,y2)**: Draws a rectangle.
- **draw_roundrect(x1,y1,x2,y2)**: Draws a rounded box.
- **draw_button(x1,y1,x2,y2,down)**: Draws a button-like shape; down indicates whether the button is down (true) or up (false).
- **draw_triangle(x1,y1,x2,y2,x3,y3)**: Draws a triangle.
- **draw_text(x,y,str)** Draws the string at the indicated place.

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

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

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

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

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

Please realize that drawing some shape does not change the bounding box of the object used for determining collisions and mouse events. The bounding box is determined by the initial image. As indicated before you can adapt the bounding box using the variables bb_l, bb_r, bb_t and bb_b.

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

- **set_brush_color(col)**: Sets the brush color, that is, the color used for filling shapes. A whole range of predefined colors is available:
  - **c_aqua**
  - **c_black** (default)
  - **c_blue**
  - **c_dkgray**
  - **c_fuchsia**
  - **c_gray**
  - **c_green**
  - **c_lime**
  - **c_ltgray**
  - **c_maroon**
  - **c_navy**
  - **c_olive**
  - **c_purple**
  - **c_red**
  - **c_silver**
  - **c_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.
- **set_pen_color(col)**: Sets the color of the pen (used for outlines, etc.).
- **set_brush_style(style)**: Sets the fill style. The following styles are available:
  - **bs_hollow**
  - **bs_solid** (default)
  - **bs_bdiagonal**
  - **bs_fdiagonal**
  - **bs_cross**
  - **bs_diagcross**
  - **bs_horizontal**
  - **bs_vertical**
- **set_pen_size(size)**: Set the size (width) of the pen.

- **set_font_color(col)**: Sets the color of the font used for drawing text.
- **set_font_size(size)**: Sets the size of the font.
- **set_font_name(name)**: Sets the name of the font, e.g. `set_font_name('Times New Roman')`.
- **set_font_angle(angle)**: Sets the angle with which the font is drawn in radians; for example, for vertical text use set_font_angle(pi/2).
- **set_font_style(style)**: Sets the style of the font. The following styles are available:
  - **fs_normal** (default)
  - **fs_bold**
  - **fs_italic**
  - **fs_bolditalic**
- **set_font_align(align)**: Sets the alignment of the font with respect to the indicated position. The following alignments exist:
  - **fa_left** (default)
  - **fa_center**
  - **fa_right**

File IO

Your game might want to write some data in a file and read it back some other time. The following routines exist:

- **file_exists(fname)**: Returns whether the file exists (true) or not (false).
- **file_delete(fname)**: Deletes the file with the given name.
- **file_rename(oldname,newname)**: Renames the file.
- **file_copy(fname,newname)**: Copies the file to the new name.
- **file_open_read(fname)**: Opens the indicated file for reading.
- **file_open_write(fname)**: Opens the indicated file for writing.
- **file_close()**: Closes the current file (don't forget to call this!).
- **file_write_string(str)**: Writes the string to the file
- **file_write_real(x)**: Write the real value to the file
- **file_writeln()**: Write a newline character to the file
- **file_read_string()**: Reads a string from the file and returns this string
- **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
- **file_eof()**: Returns whether we reached the end of the file

Note that you can only write to files that are stored in the directory in which the game is stored. So you are not allowed to include a path in the file name.