Designing Games with Game Maker

Version 7.0

Written by Mark Overmars

The documentation for *Game Maker* is divided into four parts:

Using Game Maker

This section describes the basic use of *Game Maker*. It explains the global idea behind the program and describes how to add sprites, background and sounds and how to define objects with events and actions and how to add them to rooms.

The following topics exist in this section:

Introduction
Installation
Upgrading to the Pro Edition
The Global Idea
A Simple Example
The Global User Interface
Defining Sprites

Sounds and Music
Backgrounds
Defining Objects
Events
Actions
Creating Rooms
Distributing your Game

Advanced use

This section describes the more advanced aspects of *Game Maker*. It deals with paths, fonts, time lines, scripts, and techniques for creating tiled rooms and using views in rooms.

The following topics exist in this section:

Advanced User Interface
More about Sprites
More about Sounds and Music
More about Backgrounds
More about Objects
More Actions
More about Rooms
Fonts
Paths
Time Lines
Scripts
Extension Packages

Finishing your game

This section deals with how to turn your project into a finished game. It describes how to add help information to

your game, how to set the various options for your game and how to create stand-alone games that you can distribute to others and can be run without the need for *Game Maker*.

The following topics exist in this section:

Game Information
Global Game Settings
Speed Considerations

The Game Maker Language

Game Maker contains a built-in programming language. This programming language gives you much more flexibility and control than the standard actions. This language we will refer to as GML (the *Game Maker* Language). In this section we describe the language GML and we give an overview of all the (close to 1000) functions and variables available to control all aspects of your game.

The following topics exist in this section:

Language overview
Computing things
Game play
User interaction
Game graphics
Sound and music
Splash screens, highscores and other pop-ups
Resources
Changing resources
Files, registry, and executing programs
Data structures
Creating particles

Multiplayer games
Using DLL's
3D Graphics

What is New

Version 7.0 of *Game Maker* has a number of important changes over version 6.1. In particular there is a new extension mechanism. Below the most important changes are described.

Incompatibilities

Version 7.0 uses an adapted file format. As a result file names now have the extension `.gmk`. The new version can though still read `.gm6` files created with the previous version and it is fully compatible with such files.

There is a minor incompatibility in the use of room transitions. In previous version transitions could also be used between frames in the same room. This is not longer possible. Transitions can only be used between rooms. But the number of transitions is considerably extended.

Extension packages

The major improvement of version 7.0 of *Game Maker* is the mechanism of extension packages that has been introduced. Extension packages can either add new collections of actions to *Game Maker* or they add new functions using DLL or GML files. A new item has been added to the resource list in which the user can indicate which extension packages to use. These are then automatically integrated in the system. Actions are shown in the object form and functions are colorcoded in the script

editor, show in the list of functions. Extension packages can have help files that are automatically included in the help menu. Also all required files are automatically added to the game executables.

Three extension package are default provided.

- GM Printing. Adds a large number of functions for printing shapes, text, sprites, screenshots, etc. and contains dialogs to select the printer and set printer preferences.
- GM Transitions. Adds over 60 room transitions to the program.
- GM Windows Dialogs. Adds functions to create most of the standard Windows dialogs for messages, input boxes, file selection, etc.

It is rather easy to create your own extension packages. There is a special program available for this. See the page http://www.yoyogames.com/extensions for more information on how to do this. The extension mechanism sort of replaces and enhances the current DLL mechanism (which remains available).

Publishing games

A publish button has been added to the toolbar. This will take you to our site were you can easily make your finished game available to the public.

Splash screens

The splash screen mechanism has been considerably enhanced. Splash screens, like videos and images are now

default shown in the main game window. There are many new options, e.g. to change the scaling. Also other image formats can be used. And rich text files can now contain images.

Game information

The game information is now default shown in the game window and can be closed with the close button.

Separate close button event

It can now be indicated whether or not the Close button must behave like the escape key or not. If not, there is a new event in the Other events that happens when the user clicks on the Close button. So you can now assign different behavior to the Close button and to the escape key.

Room transitions

The room transition mechanism is changed and extended, giving you more control over the transitions and adding many new transitions, like fade-ins, pushing the images, blending room images, rotations, etc. It is now even possible to define your own scripts to do the transitions (but this is rather advanced stuff). An extension package with additional room transitions is provided.

Data structures

A number of additional functions have been provided to deal with data structures. For example, data structures can now be copied, there are functions to write data structures to a string and read them back. This string can then used to e.g. save the datastructure to a file. Also there are more functions for grid data structures to e.g. copy and add parts of grids.

Including files

The mechanism to include files in the executables has been considerably extended. For each file it can now be indicated under what filename it must be stored and where it must be stored. There is a choice now whether or not to include the files in the editable version of the game. Also there are functions to export the files at a different moment than the start of the game.

Adding sprites and backgrounds with alpha channel

There are now functions sprite_add_alpha() and sprite_replace_alpha() to add or replace a sprite from a file that has an alpha channel (such as png files) to get nicer transparancy effects. Similar functions exist for backgrounds.

New registration mechanism

Version 7 uses a new registration mechanism. The free and registered version are now called the Lite and Pro Edition. There is an improved online purchase process that

immediately upgrades the program after the payment is made. Old version 5 and 6 registration keys can be exchanged for version 7 activation codes.

Other changes

There are a number of other changes and additions. Here are some of the important ones.

- The selected direction(s) in the Move Fixed action are now colored red.
- When running a game the loading bar is shown considerably earlier such that it is clear the game is running.
- Renamed all actions for more easy reference (compatible with the book).
- When adding or replacing sprites and backgrounds or when using splash images, many different file formats can now be used, including jpg, tif, bmp, gif, png, etc.
- In the image editor there now is a command (Ctrl-A) to select the whole image.
- A toolbar button was added on the main form to save the game as stand-alone executable.
- In the constants list in game settings, buttons were added to move them up or down in the list and to insert a constant above the current one.
- Function message_position(-1,-1) now sets the message box to the screen center.
- Print buttons were added in the script and code editor.
- Save and print buttons were added to the debug info forms.
- Events were added that happen when an instance lies outside a view or intersects the view boundary.
- The form showing errors in the game is enhanced and allows for copying them to the clipboard.

- Functions random_set_seed(seed), random_get_seed(), and randomize() were added.
- In the image editor you can now jump to next/prev subimage of a sprite.
- Global variables can now be declared using the keyword globalvar. After this declaration it is no longer necessary to add the word global and a dot in front of them.
- During game play F9 now takes a screenshot unless this is switched off in the global game settings.
- Added functions draw_line_width(x1,y1,x2,y2,w) and draw_line_width_color(x1,y1,x2,y2,w,col1,col2) to draw lines with a width.
- d3d_start() and d3d_end() now return whether successful.
- You can now set the variable cursor_sprite to automatically draw a sprite at the cursor location.
- Increased the maximum number of arguments to DLL functions to 16.
- In the options under Other you can now indicate version information for the game that will be embedded in the executable.
- Added functions sprite_save(ind,subimg,fname) and background_save(ind,fname) to save the resources as bitmaps.
- Added a variable program_directory that stores the location of the game executable.
- Added a constant c_orange.
- ...

Corrected bugs

The following bugs were corrected.

- Solved the problem that sometimes led to corrupted files. Version 7 will read files that were previously

marked as being corrupt.
- A problem with limited real precision was solved. This hould also solve problems with functions dealing with the date and time.
- A bug in timelines was corrected when duplicating to an earlier moment.
- Bitwise assignments now work correctly and give no syntax errors.
- show_message() actions and function now keep the box in the screen center.
- Using snow effects and explosion effects together now works correctly.
- A vulnerability to obtain information from a running game was removed.
- Debug info forms no longer keep jumping to the top position.
- Game Maker and the created games now work correctly under Windows Vista. As a result though the file size of the games has been considerably increased.
- When copying something in the code editor the font is now correct and color coding is applied.
- Backgrounds no longer loose their settings when a new image is loaded.
- An off-by-one error in drawing filled rectangles was solved.
- The game window is now made visible before executing create events.
- Solved a bug in the function median().
- Solved a crash when resizing a grid.
- Comparing and finding values in grids now works correctly for string values.
- Editing a non-transparent sprite will no longer sometimes turn it transparent in the preview.
- Solved a bug in collision checking with scaled instances.
- Solved a bug in collision checking with e.g. lines and rectangles that were not on integer coordinates.

- Function file_bin_open now create the file if it does not yet exist.
- Added checks in the object_set_parent function to avoid cycles.
- Solved an error in the bounce action with diagonal bounces.
- Corrected a bug in the rain effect when the room was higher than 1000 pixels.
- Corrected a bug that instances without sprites would contstantly get Outside Room events.
- Corrected a bug when dragging actions between Object and Timeline form (it is no longer possible).
- Increase the minimal height of the background form to make it impossible to hide the OK button.
- Corrected a bug with addressing variables in an instance after it changed object during the same step.
- Corrected a bug that (de-)activating instances without a sprite could work wrong.
- Corrected an error in drawing a path with 0 length.
- Solved a bug that the Exit Event action inside a Repeat block did not exit the event.
- Corrected the position of the loading bar for the game.
- Corrected an error in removing instance outside a room in the room editor.
- ...

Using Game Maker

Game Maker is an easy to use program for creating your own computer games. This section of the help file gives you all the information you need for creating your first games. Later sections will discuss more advanced topics, how to finish and distribute your game, and the built-in programming language GML.

Information on the basic use of *Game Maker* can be found in the following pages:

Introduction Installation
Upgrading to the Pro Edition
The Global Idea
A Simple Example
The Global User Interface
Defining Sprites
Sounds and Music
Backgrounds
Defining Objects
Events
Actions
Creating Rooms
Distributing your Game

So you want to create your own computer games

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

So does this mean that it is impossible to create your own computer games? Fortunately no. Of course you should not expect to create your own *Quake* or *Age of Empires* within a few weeks. But that is also not necessary. Simpler games, like *Tetris*, *Pacman*, *Space Invaders*, etc. are also fun to play and a lot easier to create. Unfortunately they still require good programming skills to handle the graphics, sounds, user interaction, etc.

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

into *Game Maker* that gives you full control over what is happening in your game.

Game Maker focuses on two-dimensional games. So it is not meant to create 3D worlds like *Quake*, even though there is some limited functionality for 3D graphics. But don't let this put you down. Many great games, like *Age of Empires*, the *Command & Conquer* series, and *Diablo* use two-dimensional sprite technology, even though they look very 3-dimensional. And designing two-dimensional games is a lot easier and faster.

Game Maker comes in two editions, the Lite Edition and the Pro Edition. The Lite Edition can be used free of charge but it is limited in its functionality and will display popup messages. You can though freely distribute the games you create with it, you can even sell them if you like. See the enclosed license agreement for more details. You are strongly encouraged to upgrade your copy of *Game Maker* to the Pro Edition. It will considerably extend the functionality of *Game Maker* and it will remove the logo when running games. This will also support the further development of *Game Maker*.

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

http://www.yoyogames.com/

for lots of examples, tutorials, ideas, and links to other site and forums. And soon you will become a master game maker yourself. Enjoy.

Installation

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

The first time you run *Game Maker* you are asked whether you want to run the program in **Simple** or **Advanced** mode. If you have not used a game creation program before and you are not an experienced programmer, you had better use simple mode (so select **No**). In simple mode fewer options are shown. You can easily switch to advanced mode later using the appropriate item in the **File** menu.

Requirements

Game Maker requires a modern Pentium PC running Windows 2000, Me, XP, Vista, or later. A DirectX 8 (or later) compatible graphics card with at least 32MB of memory is required for most created games. It requires a screen resolution of at least 800x600 and 65000 (16-bit) colors. Also a DirectX 8 compatible sound card is required. Make sure you have the most recent drivers installed. *Game Maker* requires DirectX version 8.0 or later to be installed on your computer. (You can download the newest version of DirectX from the Microsoft website at: http://www.microsoft.com/windows/directx/.) When designing and testing games, the memory requirements are pretty high (at least 128 MB and preferably more, also

depending on the operating system). When just running games, the memory requirements are less severe and depend a lot on the type of game.

Upgrading to the Pro Edition

Game Maker comes in two editions, the Lite Edition and the Pro Edition.

The **Lite Edition** is meant for those that take their first steps on the path of developing games. It can be used for free but is limited in its functionality. Also it shows a popup logo when running games and will regularly remind you of upgrading the program. When you are using *Game Maker* regularly you are strongly recommended to upgrade it to the Pro Edition.

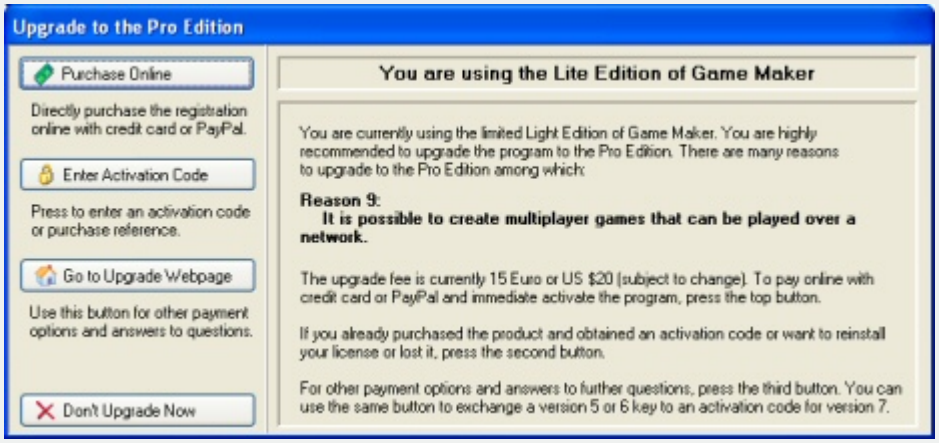
The **Pro Edition** contains considerably more functionality and does not display any logos or popup messages. More precisely, the Pro Edition has the following additional functionality:

- No *Game Maker* logo is shown when running a game.
- No regular popups remind you of upgrading.
- You can use rotated, color blended and translucent sprites.
- There are additional actions for e.g. CD music, rotated text, and colorized shapes.
- You can use special sound effects and positional sound.
- You can create splash screens with movies, images, texts, etc.
- There is a particle system to create explosions, fireworks, flames, rain, and other effects.
- A number of advanced drawing functions are available, for example colorized text and textured polygons.
- It is possible to create 3D games using functions for 3D graphics.

- It is possible to create multiplayer games that can be played over a network.
- You can define your own room transitions.
- You can use functions to create, load, and modify resources (sprites, backgrounds, etc.) while the game is running.
- There is a collection of functions to create and use data structures.
- There are functions for motion planning.
- You get the possibility to include additional files in the game executables that can be used when the game is run.
- The Pro Edition can easily be extended using extension package. These can be made by everybody and will in general be provided free of charge.
- Three such extension packages are included adding many room transitions, windows dialogs, and printing facilities.

Upgrading the Lite Edition to the Pro Edition costs only 15 Euro or US $20 (subject to change). This is a one-time fee that will at least be valid for all versions 7.x of *Game Maker*.

When you are running the Lite Edition, whenever you start *Game Maker* the following form will be shown:



You can also get to this form by choosing **Upgrade** from the **Help** menu. You can use this form to upgrade to the Pro Edition. There are a number of ways to do this.

The easiest way is to purchase the upgrade online. To this end press the button **Purchase Online**. You will be brought to a webpage were you can make you payment either by credit card or through PayPal. The payment will be handled by the company SoftWrap that is our authorized payment processor. Once you made the payment the software will immediately be upgraded to the Pro Edition without any further action from your side. Carefully save (and print) the confirmation you receive as it contains your purchase reference that you might need later if you want to reinstall the software.

If you purchased *Game Maker* before (and hence, have an activation code or a previous purchase reference) press the button **Enter Activation Code**. You will be brought to a webpage where you can either enter your activation code or your purchase reference from your previous payment. Here you can also retrieve your license if you lost it. After you filled in the correct information *Game Maker* will be upgraded to the Pro Edition. Note that you must have an Internet connection for activation.

If you want to pay for your license in a different way, or if you want to order an activation code to give as a present or use on a different machine, press the button **Go to Upgrade Webpage** or go yourself to the following webpage:

http://www.yoyogames.com/upgrade

Here you find instructions. Also you find information about ordering site license and about discounts. Moreover you find

the answers to many questions related to upgrading.

If you have a registration key for version 5 or 6 of *Game Maker* you can exchange this for a version 7 activation code. Instructions for this are given on the same webpage.

The global idea

Before delving into the possibilities of *Game Maker* it is good to get a feeling for the global idea behind the program. Games created with *Game Maker* take place in one or more *rooms*. (Rooms are flat, not 3D, but they can contain 3D-looking graphics.) In these rooms you place *objects*, which you can define in the program. Typical objects are the walls, moving balls, the main character, monsters, etc. Some objects, like walls, just sit there and donÆt do anything. Other objects, like the main character, will move around and react to input from the player (keyboard, mouse, and joystick) and to each other. For example, when the main character meets a monster he might die. Objects are the most important ingredients of games made with *Game Maker*, so let us talk a bit more about them.

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

Certain things will happen to objects. Such happenings are called *events*. Objects can take certain *actions* when events happen. There are a large number of different events that can take place and a large number of different actions that you can let your objects take. For example, there is a

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

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

Now you are ready to run the game. The first room will be shown and objects will come to life because of the actions in their creation events. They will start reacting to each other due to actions in collision events and they can react to the player using the actions in keyboard or mouse events.

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

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

There are actually a number of other types of resources: paths, scripts, fonts, and time lines. These are important for more complicated games. You will only see them when you run *Game Maker* in advanced mode. They will be treated in the advanced chapters later in this document.

Let us look at an example

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

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

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

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

The next step is to create the two objects. Let us first make the wall object. Again from the **Resources** menu choose **Create Object**. A form will open that looks quite a bit more complex than the ones we have seen so far. At the left there is some global information about the object. Give the object an appropriate name, and from the drop down menu pick the correct wall sprite. Because a wall is solid, you should check the box labeled **Solid**. That is all for the moment. Again create a new object, name it ball, and give it the ball sprite. We don't make the ball solid. For the ball, we need to define some behavior. In the middle you see an empty list of events. Below it there is a button labeled **Add Event**. Press it and you will see all possible events. Select the **Create** event. This is now added to the list of events. At the far right you see all the possible actions in a number of groups. From the **move** group choose the action with the 8 red arrows and drag it to the action list in the middle. This action will make the object move in a particular direction. Once you drop it in the action list, a dialog pops up in which you can indicate the direction of motion. Select all 8 arrows to choose a random direction. You can leave the speed as 8. Now close the dialog. So now the ball will start moving at the moment it is created. Secondly, we have to define what should happen in the case of a collision event with the wall. Again, press **Add Event**. Click on the button for collision events and in the drop down menu select the wall object. For this event we need the bounce action. (You can see what each action does by holding the mouse cursor still above it.) Finally, we need to define what to do when the user presses the left mouse button on the ball. Add the corresponding event and select the left mouse button from the pop-up menu. For this event we need a few actions: one to play a sound (can be found in the group of **main1** actions) and one to change the score (in the group **score**) and two more to let the ball jump to a new random position and moving in a new direction (in the same way as in the

creation event). For the sound action, select the correct sound. For the score action, type in a value of 1 and check the **Relative** box. This means that 1 is added to the current score. (If you make a mistake you can double click the action to change its settings.)

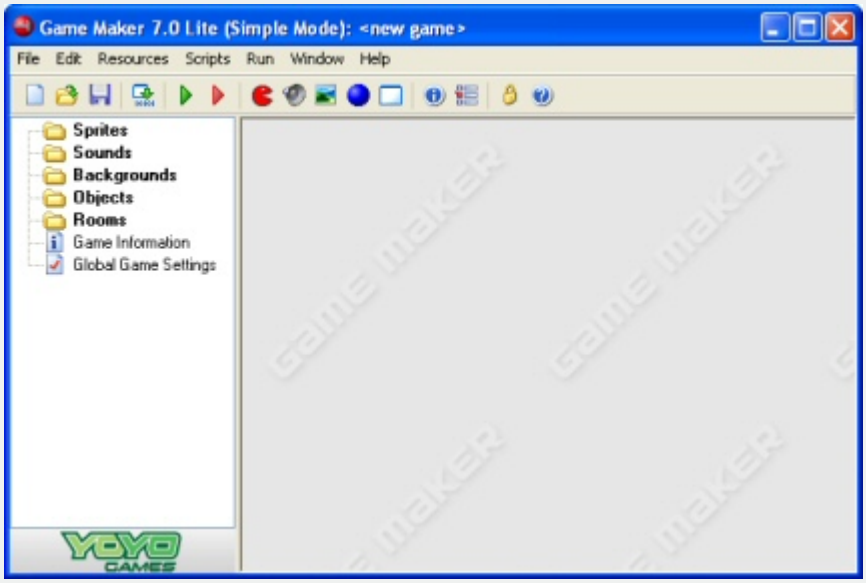
Our objects are now ready. What remains is to define the room. Create a new room in the game, again from the **Resources** menu. At the right you see the empty room. At the left you find some tabs, one for setting the background, one for setting some global properties like the width and height of the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

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

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

The global user interface

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



(Actually, this is what you see when you run *Game Maker* in simple mode. In advanced mode a number of additional items are shown.) At the left, you see the different resources mentioned above: Sprites, Sounds, Backgrounds, Objects, Rooms and two more: Game Information and Global Game Settings. At the top there is the familiar menu and toolbar. In this chapter we will describe briefly the various menu items, buttons, etc. In the later chapters we discuss a number of them in detail. Note that many things can be achieved in different ways: by choosing a command from the menu, by clicking a button, or by right clicking on a resource.

File menu

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

- **New**. Choose this command to start creating a new game. If the current game was changed you are asked whether you want to save it. There is also a toolbar button for this.
- **Open**. Opens a game file. *Game Maker* files have the extension .gmk. You can also open old .gm6 files. If you want to open .gmd files created with version 5 of *Game Maker* you must select the appropriate file type at the bottom of the dialog. These might though not work correctly in the new version.) There is a toolbar button for this command. You can also open a game by dragging the file into the *Game Maker* window.
- **Recent Files**. Use this submenu to reopen game files you recently opened.
- **Save**. Saves the game design file under its current name. If no name was specified before, you are asked for a new name. You can only use this command when the file was changed. Again, there is a toolbar button for this.
- **Save As**. Saves the game design file under a different name. You are asked for a new name.
- **Create Executable**. Once your game is ready you will probably want to give it to others to play. Using this command you can create a stand- alone version of your game. This is simply an executable that you can give to other people to run.
- **Advanced Mode**. When clicking on this command *Game Maker* will switch between simple and advanced mode. In advanced mode additional commands and resources are available.
- **Exit**. Probably obvious. Press this to exit *Game Maker* . If you changed the current game you will be asked whether you want to save it.

Edit menu

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

- **Insert resource**. Inserts a new instance of the currently selected type of resource before the current one. A form will open in which you can change the properties of the resource. This will be treated in detail in the following chapters.
- **Duplicate**. Makes a copy of the current resource and adds it. A form is opened in which you can change the resource.
- **Delete**. Deletes the currently selected resource (or group of resources). Be careful. This cannot be undone. You will, though, be warned.
- **Rename**. Gives the resource a new name. This can also be done in the property form for the resource. Also, you can select the resource and then click on the name.
- **Properties**. Use this command to bring up the form to edit the properties. Note that all the property forms appear within the main form. You can edit many of them at the same time. You can also edit the properties by double clicking on the resource.

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

Resources menu

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

Run menu

This menu is used to run the game. There are two ways to run a game.

- **Run normally**. Runs the game as it would normally run. The game is run in the most efficient way and will look and act as in an executable game.
- **Run in Debug mode**. Runs the game in debug mode. In this mode you can check certain aspects of the game and you can pause and step through it. This is useful when something goes wrong but is a bit advanced.

Once your game is finished, you can create a stand-alone executable of the game using the command in the file menu.

Window menu

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

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

Help menu

Here you find some commands to help you:

- **Contents**. Use this command to show this help file.
- **Online Help**. This command will bring you to a location on the website where you can obtain all sorts of additional help and where you can also download some tutorials.
- **Upgrade**. You can use this command to upgrade the Lite Edition of *Game Maker* to the Pro Edition. The Pro Edition has many aditional features. Here you can find information on how to upgrade the program. If you did upgrade it you can use this command to enter the registration key you received.
- **Web site**. Connects you to the *Game Maker* website where you can find information about the most recent version of *Game Maker* and collections of games and resources for *Game Maker*. We recommend that you check out the site at least once a month.
- **Forums**. This command will bring you to the forums where users help each other with many aspects of *Game Maker*.
- **About *Game Maker***. Gives some short information about this version of *Game Maker*.

The resource explorer

At the left of the main form you find the resource explorer. Here you will see a tree-like view of all resources in your game. It works in the same way as the Windows Explorer, and you are most likely familiar with it. If an item has a + sign in front of it you can click on the sign to see the resources inside it. By clicking on the - sign these disappear

again. You can change the name of a resource (except the top level ones) by selecting it (with a single click) and then clicking on the name. Double click on a resource to edit its properties. Use the right mouse button to access the same commands as in the Edit menu.

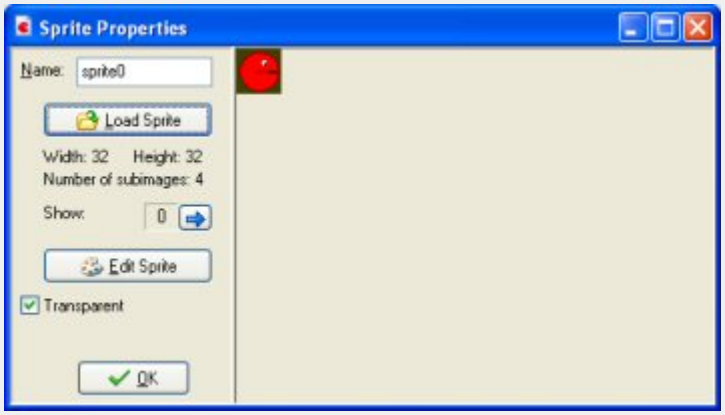
You can change the order of the resources by clicking on them with the mouse and holding the mouse button pressed. Now you can drag the resource to the appropriate place. (Of course the place must be correct. You cannot drag a sound into the list of sprites.)

Defining sprites

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

When you make a game you normally start by collecting a set of nice sprites for the objects in your game. Many collections of interesting sprites can be found on the *Game Maker* website. Other sprites can be found on the web, normally in the form of animated gif files.

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



At the top you can indicate the name of the sprite. All sprites (and all other resources) have a name. it's best to

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

To load a sprite, click on the button **Load Sprite**. A standard file dialog opens in which you can choose the sprite. *Game Maker* can load many different graphics files. When you load an animated gif, the different subimages form the sprite images. Once the sprite is loaded the first subimage is shown on the right. When there are multiple sub- images, you can cycle through them using the arrow buttons.

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

With the button **Edit Sprite** you can edit the sprite, or even create a completely new sprite.

Sounds and music

Most games have certain sound effects and some background music. Many useful sound effects can be found on the *Game Maker* website. Many more can be found on other places on the web.

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

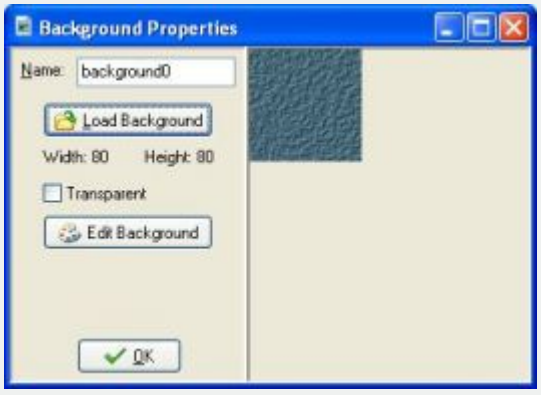
To load a sound, press the button labeled **Load Sound**. A file selector dialog pops up in which you can select the sound file. There are two types of sound files, wave files and midi files. Wave files are used for short sound effects. They use a lot of memory but play instantaneously. Use these for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

Once you load a music file you can listen to the sound using the play button. There is also a button **Save Sound** to save the current sound to a file. This button is not really required but you might need it if you lost the original sound.

Backgrounds

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

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



Press the button **Load Background** to load a background image. *Game Maker* supports many image formats. Background images cannot be animated! The checkbox **Transparent** indicates whether or not the background is partially transparent. Most backgrounds are not transparent so the default is not. As transparency color the color of the leftmost bottommost pixel is used.

You can change the background or create a new one using the button **Edit Background**.

Be careful with large backgrounds. A number of graphics cards cannot handle images that are larger than the screen. So preferably keep your background images smaller than 1024x1024.

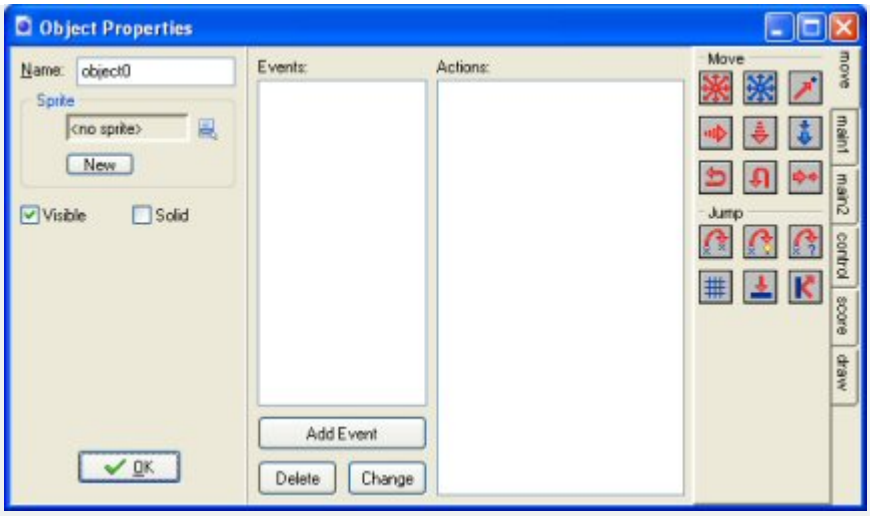
Defining objects

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

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

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

To create an object in your game, choose **Create Object** from the **Resources** menu. The following form will appear:



This is rather complex. At the left there is some general information about the object. In the middle there is the list of events that can happen to the object. At the right there are the different actions the object can perform. Events and actions will be discussed in the coming chapters.

As always, you can (and should) give your object a name. Next you can choose the sprite for the object. To this end, click with the left mouse button on the sprite box or the menu button next to it. A menu will pop-up with all the available sprites. Select the one you want to use for the object. If you do not have a sprite yet, you can click the button **New** to create a new sprite resource and change it. Also, when you select a resource there will be a button **Edit** here that you can use to change the sprite. This is faster than first finding the resource in the list of resources and then indicating you want to edit it.

Below this there are two check boxes. **Visible** indicates whether instances of this object are visible. Clearly, most objects are visible, but sometimes it is useful to have invisible objects. For example, you can use them for waypoints for a moving monster. Invisible objects will react to events and other instances do collide with them. The box

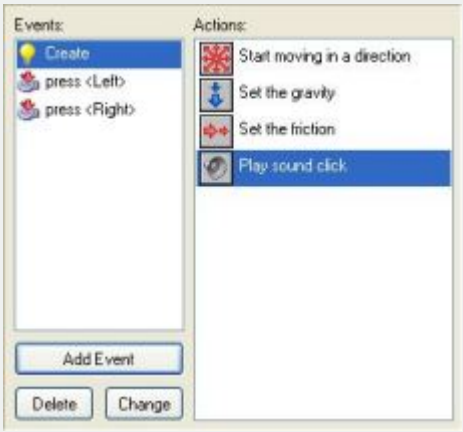
labeled **Solid** indicates whether this is a solid object (like a wall). Collisions with solid objects are treated differently from collisions with non-solid objects. You are strongly advised to use **Solid** only for object that are not moving.

Events

Game Maker uses what is called an event driven approach. This works as follows. Whenever something happens in the game the instances of the objects get events (kind of messages telling that something has happened). The instances can then react to these messages by executing certain actions. For each object you must indicate to which events it responds and what actions it must perform when the event occurs. This may sound complicated but is actually very easy. First of all, for most events the object does not have to do anything. For the events where something must be done you can use a very simple drag-and-drop approach to indicate the actions.

In the middle of the object property form there is a list of events to which the object must react. Initially it is empty. You can add events to it by pressing the button labeled **Add Event**. A form will appear with all different types of events. Here you select the event you want to add. Sometimes a menu pops up with extra choices. For example, for the keyboard event you must select the key. Below you find a complete list of the different events plus descriptions. One event in the list will be selected. This is the event we are currently changing. You can change the selected event by clicking on it. At the right there are all the actions represented by little icons. They are grouped in a number of tabbed pages. In the next chapter you will find descriptions of all the actions and what they do. Between the events and the actions there is the action list. This list contains the actions that must be performed for the current event. To add actions to the list, drag them with your mouse from the right to the list. They will be placed below each other, with a

short description. For each action you will be asked to provide a few parameters. These will also be described in the next chapter. So after adding a few actions the situation might look as follows:



Now you can start adding actions to another event. Click on the correct event with the left mouse button to select it and drag actions in the list.

You can change the order of the actions in the list again using drag-and-drop. If you hold the <Alt> key while dragging, you make a copy of the action. You can even use drag-and-drop between action lists for different objects. When you click with the right mouse button on an action, a menu appears in which you can delete the selected action (can also be done by using the <Del> key) or copy and paste actions. (You can select multiple actions for cutting, copying, or deleting by holding the <Shift> key of <Ctrl> key. Press <Ctrl><A> to select all actions.) When you hold your mouse at rest above an action, a longer description is given of the action. See the next chapter for more information on actions.

To delete the currently selected event together with all its actions press the button labeled **Delete**. (Events without any actions will automatically be deleted when you close the

form so there is no need to delete them manually.) If you want to assign the actions to a different event (for example, because you decided to use a different key for them) press the button labeled **Change** and pick the new event you want. (The event should not be defined already!) Using the menu that pops up when right-clicking on the event list, you can also duplicate an event, that is, add a new event with the same actions.

As indicated above, to add an event, press the button **Add Event**. The following form pops up:

Here you select the event you want to add. Sometimes a menu pops up with extra choices. Here is a description of the various events. (Again remember that you normally use only a few of them.)

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

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

Alarm events
Each instance has 12 alarm clocks. You can set these alarm

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

 **Step events**

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

 **Collision events**

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

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

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

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

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

 **Keyboard events**
When the player presses a key, a keyboard event happens for all instances of all objects. There is a different event for each key. In the menu you can pick the key for which you want to define the keyboard event and next drag actions there. Clearly, only a few objects need events for only a few keys. You get an event in every step as long as the player keeps the key depressed. There are two special keyboard events. One is called <No key>. This event happens in each

step when no key is pressed. The second one is called <Any key> and happens whatever key is pressed. When the player presses multiple keys, the events for all the keys pressed happen. Note that the keys on the numeric keypad only produce the corresponding events when <NumLock> is pressed.

Mouse events

A mouse event happens for an instance whenever the mouse cursor lies inside the sprite representing the instance. Depending on which mouse buttons are pressed you get the no button, left button, right button, or middle button event. The mouse button events are generated in each step as long as the player keeps the mouse button pressed. The press events are only generated once when the button is pressed. The release events are only generated when the button is released. Note that these events only occur when the mouse is above the instance. If you want to react to mouse press or release events at an arbitrary place, use the global mouse events that can be found in a submenu. There are two special mouse events. The mouse enter event happens when the mouse enters the instance. The mouse leave event happens when the mouse leaves the instance. These events are typically used to change the image or play some sound. Mouse wheel up and mouse wheel down events happen when the user moves the mouse wheel. Finally there are a number of events related to the joystick. You can indicate actions for the four main directions of the joystick (in a diagonal direction both events happen). Also you can define actions for up to 8 joystick buttons. You can do this both for the primary joystick and the secondary joystick.

Other events

There are a number of other events that can be useful in

certain games. They are found in this menu. The following events can be found here:

- **Outside**: This event happens when the instance lies completely outside the room. This is typically a good moment to destroy it.
- **Boundary**: This event happens when the instance intersects the boundary of the room, that is, it lies (at least) partially outside the room.
- **Views**: Here you find a number of events that are useful when you use views in your rooms. These events test whether the instance lies completely outside a particular view or interesects the view boundary.
- **Game start**: This event happens for all instances in the first room when the game starts. It happens before the room start event (see below) but after the creation events for the instances in the room. This event is typically defined in only one "controller" object and is used to start some background music and to initialize some variables, or load some data.
- **Game end**: The event happens to all instances when the game ends. Again typically just one object defines this event. It is for example used to store certain data in a file.
- **Room start**: This event happens for all instances initially in a room when the room starts. It happens after the creation events.
- **Room end**: This event happens to all existing instances when the room ends.
- **No more lives**: *Game Maker* has a built-in lives system. There is an action to set and change the number of lives. Whenever the number of lives becomes less than or equal to 0, this event happens. It is typically used to end or restart the game.
- **No more health**: *Game Maker* has a built-in health system. There is an action to set and change the health.

Whenever the health becomes less than or equal to 0, this event happens. It is typically used to reduce the number of lives or to restart the game.

- **End of animation**: As indicated above, an animation consists of a number of images that are shown one after the other. After the last one is shown we start again with the first one. The event happens at precisely that moment. As an example, this can be used to change the animation, or destroy the instance.
- **End of path**: This event happens when the instance follows a path and the end of the path is reached.
- **Close button**: This event happens when the user clicks on the close button of the window. This only happens when disabling in the game settings the option to have the close button behave as the Escape key.
- **User defined**: There are 16 of these events. They normally never happen unless you yourself call them from a piece of code.

 **Drawing event**

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

 **Key press events**

This event is similar to the keyboard event but it happens only once when the key is pressed, rather than continuously. This is useful when you want an action to happen only once.

Key release events

This event is similar to the keyboard event but it happens only once when the key is released, rather than continuously.

In some situation it is important to understand the order in which *Game Maker* processes the events. This is as follows:

- Begin step events
- Alarm events
- Keyboard, Key press, and Key release events
- Mouse events
- Normal step events
- (now all instances are set to their new positions)
- Collision events
- End step events
- Drawing events

The creation, destroy, and other events are performed when the corresponding things happen.

Actions

Actions indicate the things that happen in a game created with *Game Maker*. Actions are placed in events of objects. Whenever the event takes place these actions are performed, resulting in certain behavior for the instances of the object. There are a large number of different actions available and it is important that you understand what they do. In this chapter I will describe the actions available in simple mode. Note that a number of these actions are only available in the Pro Edition of *Game Maker*. This will be indicated.

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

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

When you drop an action in the action list, a window will pop-up most of the time, in which you can fill in certain parameters for the action. The parameters will be described below when describing the actions. Two types of parameters appear in many actions so we will describe these here. At the top you can indicate to which instance the action

applies. The default is `self`, which is the instance for which the action is performed. Most of the time, this is what you want. In the case of a collision event, you can also specify to apply the action to the other instance involved in the collision. For instance, in this way you can destroy the other instance. Finally, you can choose to apply the action to all instances of a particular object. In this way you could change all red balls into blue balls. The second type of parameter is the box labeled **Relative**. By checking this box, the values you type in are relative to the current values. For example, in this way you can add something to the current score, rather than changing the current score to the new value. The other parameters will be described below. You can later change the parameters by double clicking on the action.

Information on the different actions can be found in the following pages:

Move Actions Main Actions, Set 1
Main Actions, Set 2
Control Actions
Score Actions
Draw Actions
Using Variables and Expressions

Move actions

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

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

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

 **Move Towards**
This action gives a third way to specify a motion. You indicate a position and a speed and the instance starts moving with the speed towards the position. (It won't stop

at the position!) For example, if you want a bullet to fly towards the position of the spaceship you can use as position `spaceship.x, spaceship.y`. (You will learn more about the use of variables like these below.) If you check the **Relative** box, you specify the position relative to the current position of the instance. (The speed is not taken relatively!)

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

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

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

 **Reverse Horizontal**
With this action you reverse the horizontal motion of the

instance. This can for example be used when the object collides with a vertical wall.

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

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

 **Jump to Position**
Using this action you can place the instance in a particular position. You simply specify the x- and y-coordinate, and the instance is placed with its reference point on that position. If you check the **Relative** box, the position is relative to the current position of the instance. This action is often used to continuously move an instance. In each step we increment the position a bit.

 **Jump to Start**
This action places the instance back at the position where it was created.

 **Jump to Random**
This action moves the instance to a random position in the room. Only positions are chosen where the instance does not intersect any solid instance. You can specify the snapping used. If you specify positive values, the coordinates chosen with be integer multiples of the indicated values. This could for example be used to keep the instance aligned with the cells in your game (if any).

You can specify a separate horizontal snapping and vertical snapping.

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

 **Wrap Screen**
With this action you can let an instance wrap around, that is, when it leaves on one side of the room it reappears at the other side. This action is normally used in the **Outside** event. Note that the instance must have a speed for wrapping to work, cause the direction of wrapping is based on the direction of the motion. You can indicate whether to wrap only horizontal, only vertical, or in both directions.

 **Move to Contact**
With this action you can move the instance in a given direction until a contact position with an object is reached. If there already is a collision at the current position the instance is not moved. Otherwise, the instance is placed just before a collision occurs. You can specify the direction but also a maximal distance to move. For example, when the instance is falling you can move a maximal distance down until an object is encountered. You can also indicate whether to consider solid object only or all objects. You typically put this action in the collision event to make sure that the instance stops in contact with the other instance involved in the collision.

 **Bounce**
When you put this action in the collision event with some object, the instance bounces back from this object in a

natural way. If you set the parameter precise to false, only horizontal and vertical walls are treated correctly. When you set precise to true also slanted (and even curved) walls are treated correctly. This is though slower. Also you can indicate whether to bounce only against solid objects or against all objects. Please realize that the bounce is not completely accurate because this depends on many properties. But in many situations the effect is good enough.

Main actions, set 1

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

 **Create Instance** With this action you can create an instance of an object. You specify which object to create and the position for the new instance. If you check the **Relative** box, the position is relative to the position of the current instance. Creating instances during the game is extremely useful. A space ship can create bullets; a bomb can create an explosion, etc. In many games you will have some controller object that from time to time creates monsters or other objects. For the newly created instance the creation event is executed.

 **Create Moving**
This action works the same as the action above but with two additional fields. You can now also specify the speed and direction of the newly created instance. Note that if you check the **Relative** box, only the position is relative, not the speed and direction. For example, to make a bullet move in the direction of the person shooting you have to use a little trick. As position use 0,0 and check **Relative**. As direction we need the current direction of the instance. This can be obtained by typing in the word `direction`. (This actually is a variable that always indicates the current direction in which the instance is moving.)

 **Create Random**
This action lets you create an instance of one out of four objects. You specify the four objects and the position. An

instance of one of these four objects is created at the given position. If you check the **Relative** box, the position is relative to the position of the current instance. If you need a choice out of less than four objects you can use No Object for some of them. This is for example useful to generate a random enemy at a location.

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

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

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

 **Change Sprite**
Use this action to change the sprite for the instance. You indicate which new sprite. You can also indicate with subimage must be shown. Normally you would use 0 for this (the first subimage) unless you want to see a particular subimage. Use -1 if you do not want to change the current subimage shown. Finally you change the speed of the animation of the subimages. If you only want to see a particular subimage, set the speed to 0. If the speed is

larger than one subimages will be skipped. If it is smaller than 1 subimages will be shown multiple times. Don't use a negative speed. Changing sprites is an important feature. For example, often you want to change the sprite of a character depending on the direction in which it walks. This can be achieved by making different sprites for each of the (four) directions. Within the keyboard events for the arrow keys you set the direction of motion and the sprite.

 **Transform Sprite**
Use this action to change the size and orientation of the sprite for the instance. Use the scale factors to make it larger or smaller. The angle gives the counter-clockwise orientation of the sprite. For example, to make the sprite oriented in the direction of motion use as a value `direction`. For example, this is useful for a car. You can also indicate whether the sprite should be mirrored horizontally and/or flipped vertically. ***This action is only available in the Pro Edition.***

 **SColor Sprite**
Normally the sprite is drawn as it is defined. Using this action you can change the color of the sprite. This color is blended with the sprite, that is, it is combined with the colors of the sprite. If you want to draw a sprite in different colors you better define the sprite in black and white and use the blend color to set the actual color. You can also indicate an alpha transparency. With a value of 1 the sprite is opaque. With a value of 0 it is completely transparent. With a value in between you will partially see the background shine through it. This is great for making explosions. ***This action is only available in the Pro Edition.***

 **Play Sound**
With this action you play one of the sound resources you added to your game. You can select the sound you want to play and choose whether it should play only once (the default) or loop continuously. Multiple wave sounds can play at once but only one midi sound can play at a time. So if you start a midi sound, the current midi sound is stopped.

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

 **Check Sound**
If the indicated sound is playing the next action is performed. Otherwise it is skipped. You can select **Not** to indicate that the next action should be performed if the indicated sound is not playing. For example, you can check whether some background music is playing and, if not, start some new background music. Note that this action returns true when the sound actually plays through the speakers. After you call the action to play a sound it does not immediately reach the speakers so the action might still return false for a while. Similar, when the sound is stopped you still hear it for a while (e.g. because of echo) and the action will still return true.

 **Previous Room**
Move to the previous room. You can indicate the type of transition effect between the rooms. You should experiment to see what works best for you. If you are in the first room you get an error.

 **Next Room**
Move to the next room. You can indicate the transition.

 **Restart Room**
The current room is restarted. You indicate the transition effect.

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

 **Check Previous**
This action tests whether the previous room exists. If so, the next action is executed. You normally need this test before moving to the previous room.

 **Check Next**
This action tests whether the next room exists. If so, the next action is executed. You normally need this test before moving to the next room.

Main actions, set 2

Here are some more main actions, dealing with timing, giving messages to the user, and dealing with the game as a whole.

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

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

 **Display Message**
With this action you can display a message in a dialog box. You simply type in the message. If you use a # symbol in the message text it will be interpreted as a new line character. (Use \# to get the # symbol itself.) If the message text starts with a quote or double quote symbol, it is interpreted as an expression. See below for more information about expressions.

 **Show Info**
With this action you pop up the game information window.

 **Restart Game**
With this action you restart the game from the beginning.

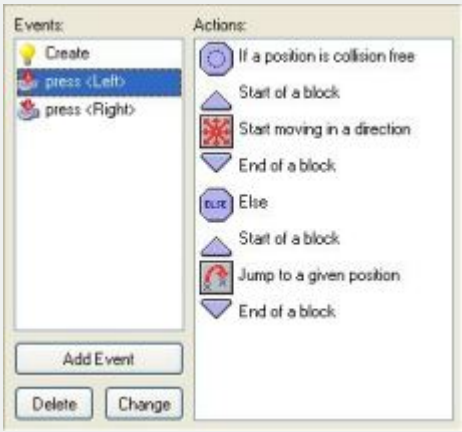
 **End Game**
With this action you end the game.

 **Save Game**
With this action you can save the current game status. You specify the filename for saving (the file is created in the working directory for the game). Later the game can be loaded with the next action. (Please realize that only the basic game status is save. Things that are for example not saved are the current sound that is playing, and advanced aspects like the contents of data structures, particles, etc.)

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

Control actions

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



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

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

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

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

 **Check Empty** This question returns true if the current instance, placed at the indicated position does not generate a collision with an object. You can specify the position as either absolute or relative. You can also indicate whether only solid, or all objects, should be taken into account. This action is typically used to check whether the instance can move to a particular position.

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

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

 **Test Instance Count**
You specify an object and a number. If the current number of instances of the object is equal to the number the question returns true. Otherwise it returns false. You can also indicate that the check should be whether the number of instances is smaller than the given value or larger than the given value. This is typically used to check whether all

instances of a particular type are gone. This is often the moment to end a level or a game.

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

 **Check Question**
You specify a question. A dialog is shown to the player with a yes and a no button. The result is true is the player answers yes.

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

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

 **Check Grid**
Returns true if the position of the instance lies on a grid.

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

 **Start Block**
Indicates the start of a block of actions.

 **End Block**
Indicates the end of a block of actions.

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

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

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

If you want more control over what is happening in the game you can use the built-in programming language that is described on part 4 of the documentation. If gives you much more flexibility than using the action. Simpler use but also important involves the use of your own variables. The following actions deal with this.

 **Execute Code**
When you add this action, a form shows in which you can type in a piece of code which must be execute. This can contain simple function calls or more complex code. Use the

code action peferably only for small pieces of code. For longer pieces you are strongly advised to use scripts which are described in part 2 of the documentation.

Comment

Use this action to add a line of comment to the action list. The line is shown in italic font. Adding comments helps you remember what your events are doing. The action does not do anything. But realize that it still is an action. So when you place it after a conditional action it is the action that is executed if the condition is true (even though it does not do anything).

Set Variable

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

Test Variable

With this action you can check what the value of a particular variable is. If the value of the variable is equal to the number provided, the question returns true. Otherwise it returns false. You can also indicate that the check should be whether the value is smaller than the given value or larger than the given value. See below for more information about variables. Actually, you can use this action also to compare two expressions.

Draw Variable

With this action you can draw the value of a variable at a

particular position on the screen. Note that this can only be used in the draw event of an object.

Score actions

In most games the player will have a certain score. Also many games give the player a number of lives. Finally, often the player has a certain health. The following actions make it easy to deal with the score, lives, and health of the player.

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

 **Test Score**
With this question action you can check whether the score has reached a particular value. You indicate the value and whether the score should be equal to that value, be smaller than the value or be larger than the value.

 **Draw Score**
With this action you can draw the value of the score at a particular position on the screen. You provide the positions and the caption that must be placed in front of the score. The score is drawn in the current font. This action can only be used in the drawing event of an object.

 **Show Highscore**
For each game the top ten scores are maintained. This action displays the highscore list. If the current score is among the top ten, the new score is inserted and the player can type his or her name. You can indicate what

background image to use, whether the window should have a border, what the color for the new entry and the other entries must be, and which font to use.

 **Clear Highscore**
This action clears the highscore table.

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

 **Test Lives**
With this question action you can check whether the number of lives has reached a particular value. You indicate the value and whether the number of lives should be equal to that value, be smaller than the value or be larger than the value.

 **Draw Lives**
With this action you can draw the number of lives at a particular position on the screen. You provide the positions and the caption that must be placed in front of the number of lives. The number of lives is drawn in the current font. This action can only be used in the drawing event of an object.

 **Draw Life Images**
Rather than drawing the number of lives left as a number, it is often nicer to use a number of small images for this. This action does precisely that. You specify the position and the

image and at the indicated position the number of lives is drawn as images. This action can only be used in the drawing event of an object.

 **Set Health**
Game Maker has a built-in health mechanism. You can use this action to change the health. A value of 100 is considered full health and 0 is no health at all. You simply provide the new value for the health. Often you want to subtract something from the health. In this case don't forget the check the **Relative** box. When the health becomes smaller or equal to 0 an out of health event is generated.

 **Test Health**
With this question action you can check whether the health has reached a particular value. You indicate the value and whether the health should be equal to that value, be smaller than the value or be larger than the value.

 **Draw Health**
With this action you can draw the health in the form of a health bar. When the health is 100 the full bar is drawn. When it is 0 the bar is empty. You indicate the position and size of the health bar and the color of the bar and the background.

 **Score Caption**
Normally in the window caption the name of the room and the score is displayed. With this action you can change this. You can indicate whether or not to show the score, lives, and/or health and what the caption for each of these must be.

Drawing actions

Normally in each step of the game, for each instance, its sprite is drawn in the room. You can change this by putting actions in the draw event. (Note that these are only executed when the instance is visible!) The following drawing actions are available. These actions only make sense in the drawing event. At other places they are basically ignored.

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

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

 **Draw Text**
You specify the text and the position. A # symbol in the text is interpreted as going to a new line. (Use \# to get the # symbol itself.) So you can create multi-line texts. If the text starts with a quote or a double quote, it is interpreted as an expression. For example, you can use

```
'X: ' + string(x)
```

to display the value of the x-coordinate of the instance. (The variable x stores the current x-coordinate. The

function `string()` turns this number into a string. + combines the two strings.)

 **Draw Scaled Text**
This action is similar to the previous action but this time you can also specify a horizontal and vertical scaling factor to change the size of the text and you can specify an angle to rotate it. ***This action is only available in the Pro Edition.***

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

 **Horizontal Gradient**
This action also draws a rectangle but this time using a gradient color that changes from left to right. You specify the rectangle and the two colors to use. ***This action is only available in the Pro Edition.***

 **Vertical Gradient**
This action also draws a rectangle but this time using a gradient color that changes from top to bottom. You specify the rectangle and the two colors to use. ***This action is only available in the Pro Edition.***

 **Draw Ellipse**
This action draws an ellipse. You specify the coordinates of the two opposite corners of the surrounding rectangle; either absolute or relative to the current instance position.

 **Gradient Ellipse**
Again an ellipse is drawn but this time you specify a color

for the center and the boundary. ***This action is only available in the Pro Edition.***

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

 **Draw Arrow**
Draws an arrow. You specify the coordinates of the two endpoints of the line and the size of the arrow tip.

 **Set Color**
Lets you set the color used for drawing shapes, lines, and text. (It does not influence the way sprites and backgrounds are drawn.)

 **Set Full Screen**
With this action you can change the screen mode from windowed to fullscreen and back. You can indicate whether to toggle the mode or whether to go to windowed or fullscreen mode.

 **Take Snapshot**
With this action you can take a snapshot image of the game and store it in a .bmp file. You specify the filename to store the image in. ***This action is only available in the Pro Edition.***

 **Create Effect**
With this action you can create all sorts of effects in a very simple way. You specify the type of effect, e.g. an explosion or smoke, its position, its size and its color and whether it should be shown below the objects or on top of them. For the rest it works automatic. (For rain and snow the position is irrelevant as it always falls down from the top of the

screen. To get continuous rain you should create it in each step.) ***This action is only available in the Pro Edition.***

Using expressions and variables

In many actions you need to provide values for parameters. Rather than just typing a number, you can also type a formula, e.g. 32*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 of the instance. There are a large number of other variables that you can use. Some of the most important ones are:

x the x-coordinate of the instance
y the y-coordinate of the instance
hspeed the horizontal speed (in pixels per step)
vspeed the vertical speed (in pixels per step)
direction the current direction of motion in degrees (0-360)
speed the current speed in this direction
visible whether the object is visible (1) or invisible (0)
image_index this variable indicate which subimage in the current sprite is currently shown. If you change it and set the speed to 0 (see below) you can display a fixed subimage.
image_speed this variable indicates the speed with which the subimages are shown. The default value is 1. If you make this value larger than 1 some subimages are skipped to make the animation faster. If you make it smaller than 1 the animation becomes slower by repeating subimages.
score the current value of the score
lives the current number of lives
health the current health (0-100)

mouse_x x-position of the mouse
mouse_y y-position of the mouse

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

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

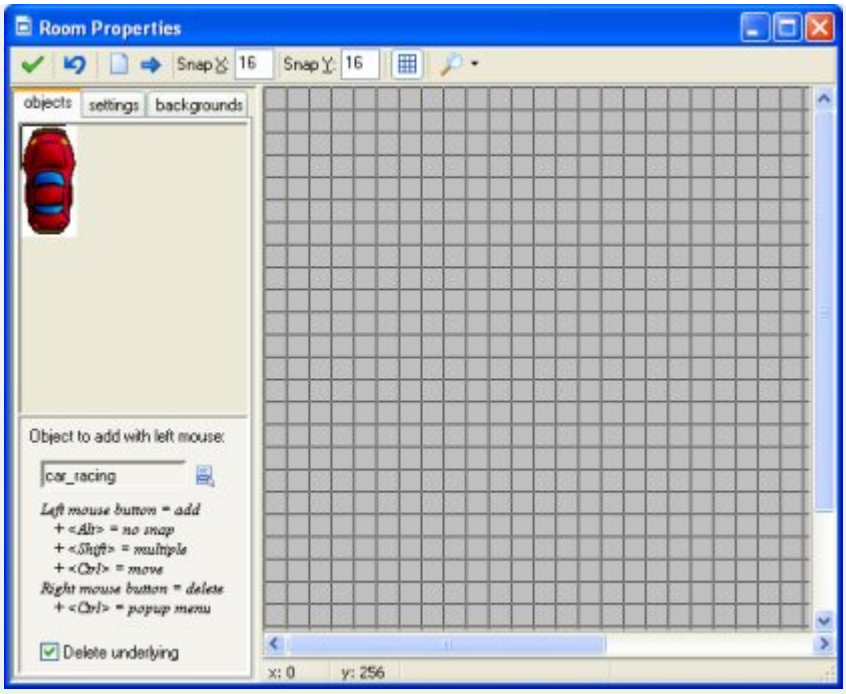
In your expressions you can also use functions. For example, the function `random(10)` gives a random real number below 10. So you can set for example the speed or direction of motion to a random value. Many more functions exist. They are described in part 4 of the documentation.

Creating rooms

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

There are a large number of possibilities when creating rooms. Besides setting a number of properties and adding the instances of the objects you can add backgrounds, define views, and add tiles. Most of these options are discussed later. In this chapter we will only discuss some basic settings, the addition of instances of objects, and the setting of background images.

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



At the top of the form there is a tool bar. On this you can indicate the size of the grid cells used for aligning objects. Also you can indicate whether or not to show the grid lines and whether or not to show the backgrounds, etc. It is sometimes useful to temporarily hide certain aspects of the room. Realize though that when you are adding instances of objects, these will always be shown, independent of the view setting.) There are also buttons to clear all instances from the room and to shift all instances over a number of pixels. Use negative numbers to shift them left or up. This is useful when for instance you decided to enlarge the room. (You can also use this to place instances outside the room, which is sometimes useful.). Finally there is the **Undo** button to undo the last change to the room and the **OK** button to save the changes. (Click on the cross at the top right to close the form without saving the changes.)

At the left you will see three tab pages (five in advanced mode). The **objects** tab is where you add instances of objects to the room. In the **settings** tab you can indicate a

number of settings for the room. In the **backgrounds** tab you can set background images for the room.

Adding instances

At the right in the room design form you see the room. At the start it is empty, with a gray background.

To add instances to the room, first select the **objects** tab if this one is not already visible. Next select the object you want to add by clicking on the button with the menu icon (or by clicking in the image area at the left). The image of the object appears at the left. (Note that when you changed the origin of the sprite there is a cross in the image. This indicates how the instances will be aligned with the grid.) Now click with your left mouse button in the room area at the right. An instance of the object appears. It will snap to the indicated grid. If you hold the <Alt> key while placing the instance it is not aligned to the grid. If you hold down the mouse button while dragging it over the room, you move the instance to the correct place. If you hold the

<Shift> key while pressing and moving the mouse multiple instances are added. With the right mouse button you can remove instances. In this way you define the contents of the room.

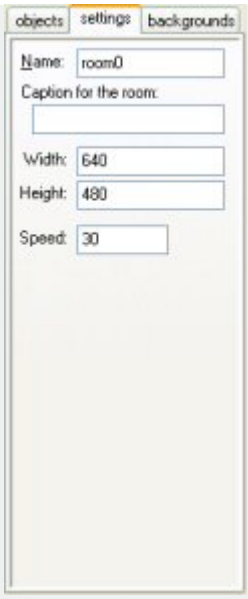
As you will notice, if you place an instance on top of another one, the original instance disappears. Normally this is what you want, but not always. This can be avoided by unchecking the box labeled **Delete underlying** at the left.

If you want to change the position of an instance, hold the <Ctrl> key and click with the left mouse button on the instance and hold down the button. You can now drag it to a new position. (Use <Alt> for precise positioning.)

If you hold the <Ctrl> key while clicking with the right mouse button on an instance, a menu appears. Here you can delete the object, type in a precise position for the instance, or move the bottommost instance at the position to the top or send the topmost instance to the bottom.

Room setting

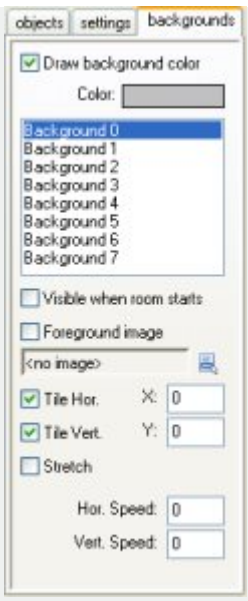
Each room has a number of settings that you can change by clicking on the **settings** tab.



Each room has a name. Best give it a meaningful name. There also is a caption. This caption is displayed in the window caption when the game is running. You can set the width and height of the room (in pixels). Also you can set the speed of the game. This is the number of steps per second. The higher the speed, the smoother the motion is. But you will need a faster computer to run it.

Setting the background

With the tab **backgrounds** you can set the background image for the room. Actually, you can specify multiple backgrounds. The tab page looks as follows:



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

At the top you see a list of 8 backgrounds. You can define each of them but most of the time you will need just one or two. To define a background, first select it in the list. Next check the box labeled **Visible when room starts** otherwise you won't see it. The name of the background will become bold when it is defined. Now indicate a background image in the menu. There are a number of settings you can change. First of all you can indicate whether the background image should tile the room horizontally and/or vertically. You can also indicate the position of the background in the room (this will also influence the tiling). A different option is to stretch the background. The background will then be scaled so that it fills the entire room. The aspect ration of the image will not be maintained. Finally you can make the background scrolling by giving it a horizontal or vertical

speed (pixels per step). Better not use scrolling with a stretched background. The result will be a bit jaggy.

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

Distributing your game

With the information in the preceding chapters you can create your games. When your game is finished you obviously want other people to play it. You can of course give them the .gmk file that you created and let them use *Game Maker* to play it but this is normally not what you want. First of all, you don't want others to be able to change the game, and secondly you also want people to play the game even if they do not have *Game Maker*. So you would like to create a stand-alone executable of your game.

Creating stand-alone executables is very easy in *Game Maker*. In the **File** menu you select the item **Create Executable**. You will be asked for the name of the executable that should contain the game. Indicate a name, press **OK** and you have your stand-alone game that you can give to anyone you like. You can change the icon for the stand-alone game in the **Global Game Settings**.

Once you have created a stand-alone executable in the way described above you can give this file to other people or place it on your website to download. You are free to distribute the games you create with *Game Maker* in any way you like. You can even sell them. This of course assumes that the sprites, images, and sounds you use can be distributed or sold as well. See the enclosed license agreement for more information.

It is normally useful to zip your executable, together with some readme information. In Windows XP this can be done directly through the right mouse button menu, and there are many free zip utilities available on the web.

Alternatively you can create an installer for your game. Again, a large number of free installation creation programs are available on the web.

Advanced use

This section of the help file gives you information about the more advanced aspects of *Game Maker*.

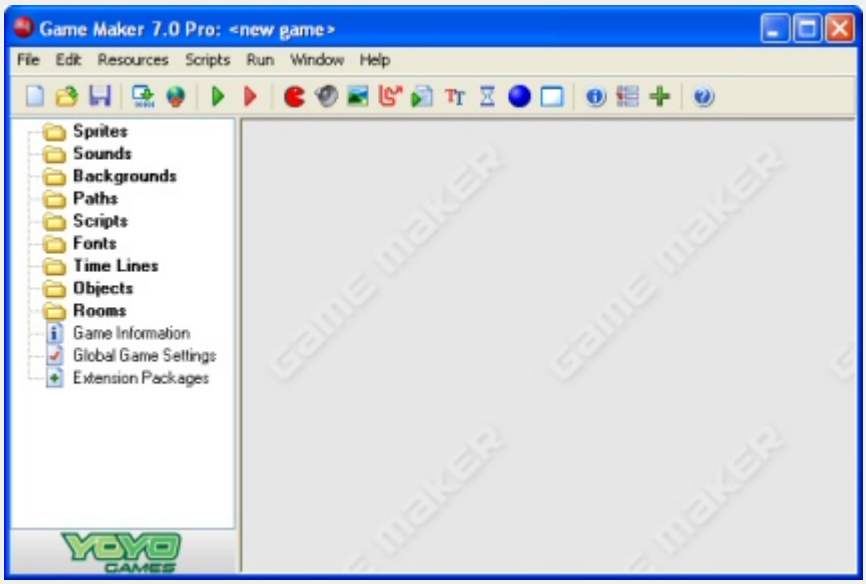
Information on the advanced use of *Game Maker* can be found in the following pages:

Advanced User Interface More about Sprites
More about Sounds and Music
More about Backgrounds
More about Objects
More Actions
More about Rooms
Fonts
Paths
Time Lines
Scripts
Extension Packages

Advanced mode

Up to now we have considered the simple features of *Game Maker*. But there are a lot more possibilities. To be able to use these you must run *Game Maker* in advanced mode. This is easy to change. In the **File** menu, click on the menu item **Advanced mode**. (To fully see the effects you should restart *Game Maker* or at least save your game and load it anew.)

When you start *Game Maker* in advanced mode, the following form is shown:



It contains all that was there in simple mode, but there are a number of additional resources, buttons, and menu items. Also, as we will see in the chapters that follow, the different resources have additional options. Here we will discuss the additional menu items.

File menu

In the file menu you can find the following additional commands:

- **Publish your Game.** This command will take you to our website were you can easily upload and publish your finished game such that everybody can play it. Carefully follow the instructions to make your game available. Please only use this for finished games, not for preliminary versions. ***This possibility is only available in the Pro Edition.***
- **Merge Game.** With this command you can merge all the resources (sprites, sounds, objects, rooms, etc.) from another game into the current game. This is very useful if you want to make parts you want to reuse (e.g. menu systems). (Note that all resources and instances and tiles will get a new id, which might cause problems if you use these in scripts.) It is your responsibility to make sure that the resources in the two files have different names, otherwise problems might occur.
- **Preferences.** Here you can set a number of preferences about *Game Maker*. They will be remembered between different calls of *Game Maker*. See below for a list of all possibilities.

Preferences

Under the **Preferences** menu item in the **File** menu you can set a number of preferences that will be maintained between runs of *Game Maker*. The following preferences can be set:

- **Show recently edited games in the file menu.** If checked the eight most recently edited games are shown under the recent files in the file menu.

- **Load last opened file on startup.** If checked when you start *Game Maker* the most recently opened file is opened automatically.
- **Keep backup copies of files.** If checked the program saves a backup copy of your game with the extension gb0-gb9. You can open these games in *Game Maker*. You are strongly advised to use at least one backup copy for your work!
- **Maximal number of backups.** Here you can indicate how many (1-9) different backup copies should be remembered by the program.
- **Show progress while loading and saving files.** If checked, when load or save a file a progress indicator is shown.
- **At startup check for, and remove old temporary files.** *Game Maker* and games created with it, create temporary files. Normally these are automatically removed but sometimes, for example when games crash, they are left behind. If this option is checked, *Game Maker* checks whether such files exist and removes them at startup.
- **Don't show the website in the main window.** When checked the image and link to the website on the main window are not shown.
- **Hide the designer and wait while the game is running.** When checked the game making program is hidden while you are testing a game.
- **Run games in secure mode.** If checked, any game created with *Game Maker* that runs on your machine will not be allowed to execute external programs or change or delete files at a place different from the game location. (This is a safeguard against Trojan horses although success is not guaranteed.) Checking this means that games that utilizes external files etc. won't be running correctly. The setting only works while *Game Maker* is running. So if you run the game independently

of *Game Maker*, for example as an executable stand-alone, it is NOT run in secure mode.

- **Show the origin and bounding box in the sprite image.** If checked, in the sprite properties form, in the sprite image, the origin and bounding box for the sprite are indicated.
- **In object properties, show hints for actions.** If checked, in the object properties form, when you hold your mouse over one of the actions, a description is shown.
- **When closing, remove instances outside the room.** If checked, the program warns you when there are instances or tiles outside a room and lets you remove them.
- **Remember room settings when closing the form.** If checked, a number of room settings, like whether to show the grid, whether to delete underlying objects, etc. are remembered when you edit the same room later.
- **Scripts and code and colors.** See the chapter on scripts for more information about these preferences.
- **Image editor.** Default *Game Maker* uses a built-in editor for images. If you favor a different image editing program you can specify here to use that different program for editing the images.
- **External sound editors.** You can indicate here which external editors to use for the different sound types. (Note that *Game Maker* does not have a built-in sound editor so if you don't specify editors here you cannot edit the sounds.)

Edit menu

In the edit menu you can find the following additional commands:

- **Add group.** Resources can be grouped together. This is very useful when you make large games. For example, you can put all sounds related to a certain object in a group, or you can group all objects that are used in a particular level. This command creates a new group in the currently selected resource type. You will be asked for a name. Groups can again contain groups, etc. As indicated below you can drag resources into the groups.
- **Find Resource.** With this command you type in the name of a resource and open the corresponding property form.
- **Expand Resource Tree.** Fully expands the resource tree, showing all resources.
- **Collapse Resource Tree.** Fully collapses the resource tree, hiding all resources.
- **Show Object Information.** Using this command you can get an overview of all objects in the game.

Resources menu

In this menu you can now also create the additional resources. Note that for each of them there is also a button on the toolbar and a keyboard shortcut.

Scripts menu

In the scripts menu you can find the following additional commands:

- **Import Scripts.** Can be used to import useful scripts from files.

- **Export Scripts.** Can be used to save your scripts in a file, to be used by others. When you select a script resource only this script is saved. When you select a group all scripts in the group are saved. When you select the root resource (or a different type of resource) all scripts are saved. This menu item is also available when right-clicking on a script or group of scripts.
- **Show Built-in Variables.** Shows a sorted list of all built-in variables, both local and global.
- **Show Built-in Functions.** Shows a sorted list of all built-in functions.
- **Show Extension Functions.** Shows a sorted list of all functions available in the extension packages you included in your game.
- **Show Constants.** Shows a sorted list of all built-in constants and constants defined in the game options.
- **Show Resource Names.** Shows a sorted list of all resource names. You can click on a name to open the particular resource for editing.
- **Search in Scripts.** You can search for a string in all scripts. You can click on one of the reported places to move there for editing.
- **Check Resource Names.** Does a check of all resource names. Names will be reported if they are not correct, if there are duplicate resource names, or when a resource name is the name of a variable, function, or constant. You can click on a name to open the particular resource for editing.
- **Check All Scripts.** Checks all scripts for errors. You can click on one of the reported places to move there for editing.

More about sprites

A number of advanced possibilities exist to create your own sprites.

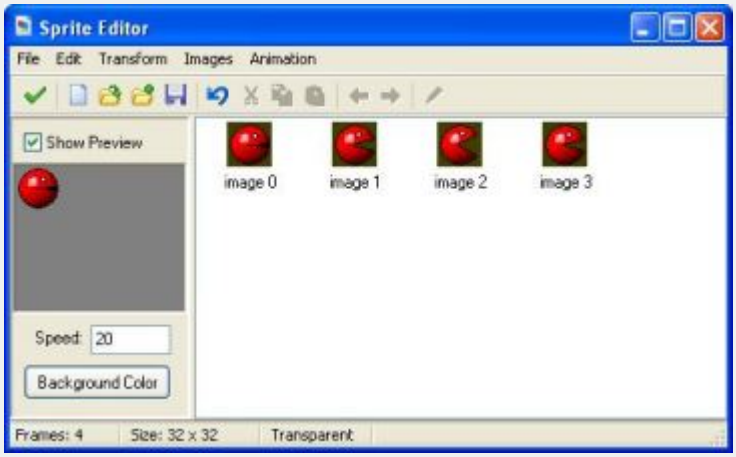
Information on the different advanced sprite options can be found in the following pages:

Editing your sprites Strips
Editing individual subimages
Advanced sprite settings

Editing your sprites

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

The sprite edit form will look as follows:



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

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

File menu

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

- **New.** Creates a new, empty sprite. You must indicate the size of the sprite. (Remember, all images in a sprite must have the same size.)
- **Create from file.** Creates the sprite from a file. Many file types can be used. They all create a sprite consisting of a single image, except for animated GIF files that are split into the subimages. Please note that the transparency color is the bottommost leftmost pixel, not the transparency color in the GIF file. You can select multiple images which will then all be loaded. They must though have the same size.
- **Add from file.** Adda an image (or images) from a file to the current sprite. If the images do not have the same size you can choose where to place them or to stretch them. You can select multiple images which will then all be loaded. They must though have the same size.
- **Save as GIF.** Saves the sprite as an animated gif.
- **Save as strip.** Saves the sprite as a bitmap, with all images next to each other.
- **Create from strip.** Allows you to create a sprite from a strip. See below for more information.
- **Add from strip.** Use this to add images from a strip. See below.

- **Close saving changes.** Closes the form, saving the changes made to the sprite. If you don't want to save the changes, click on the close button of the window.

Edit menu

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

Transform menu

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

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

Images menu

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

- **Cycle left.** Cycles all images one place to the left. This effectively starts the animation at a different point.
- **Cycle right.** Cycles all images one place to the right.
- **Black and white.** Makes the sprite black and white (does not affect the transparency color!).
- **Colorize.** Here you can change the color (hue) of the images. Use the slider to pick the different colors.
- **Colorize Partial.** Here you can change the color (hue) of part of the images. You can select an old color and a range around it and then indicate the new color with which to replace this range of colors. This can be used for instance to change only the color of the shirts of players.
- **Shift Hue.** Another way of changing the color of the images. But this time the colors are shifted over the amount indicated giving rather interesting effects.
- **Intensity.** Here you can change the intensity by providing values for the color saturation and the lightness of the images.
- **Invert.** Inverts the colors in the images.
- **Fade.** Here you specify a color and an amount. The colors in the images are now faded towards this color.
- **Transparency.** Here you can indicate a level of screen-door transparency. This is achieved by making a number of pixels transparent.
- **Blur.** By blurring the images the colors are mixed a bit, making it more vague. The higher the value, the more vague it becomes.
- **Outline.** Creates an outline around the image. You are asked for the color and whether the current images

must be removed (keeping only the outline) or whether the outline must be drawn on the image.
- **Boundary.** Similar to the outline but this time it is not drawn outside the image but on the boundary pixels of the image.
- **Crop.** This makes the images as small as possible. This is very useful because the larger the images, the more video memory *Game Maker* will use. You might want to leave a little border around the images though to avoid transparency problems.

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

Animation menu

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

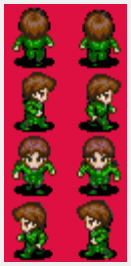
- **Set Length.** Here you can change the length of your animation. The animation is repeated enough times to create the number of frames you indicate. (Normally you want this to be a multiple of the current number of frames.)
- **Stretch.** This command also changes the length of the animation. But this time, frames are duplicated or removed to get the right number. So if you increase the number of frames the animation goes slower and if you decrease the number it goes faster.
- **Reverse.** Well, as you could guess this reverses the animation. So it is played backwards.

- **Add Reverse.** This time the reverse sequence is added, doubling the number of frames. This is very useful for making an object go left and right, change color and return, etc. You sometimes might want to remove the double first and middle frame that occur.
- **Translation sequence.** You can create an animation in which the image slightly translates in each step. You must provide the number of frames and the total amount to move horizontally and vertically.
- **Rotation sequence.** Creates an animation in which the image rotates. You can either choose clockwise or counterclockwise rotation. Specify the number of frames and the total angle in degrees (360 is a complete turn). (You might need to resize the canvas first to make sure the total image remains visible during the rotation.)
- **Colorize.** Creates an animation that turns the image into a particular color.
- **Fade to color.** Creates an animation that fades the image to a particular color.
- **Disappear.** Makes the image disappear using screen-door transparency.
- **Shrink.** Shrinks the image to nothing. You can indicate the direction.
- **Grow.** Grows the image from nothing.
- **Flatten.** Flattens the image to nothing in a given direction.
- **Raise.** Raises the image from a given direction
- **Overlay.** Overlays the animation with another animation or image in a file.
- **Morph.** Morphs the animation to an animation or image from a file. Note that morphing works best if the two animations cover the same area of the image. Otherwise, halfway certain pixels disappear and others suddenly appear.

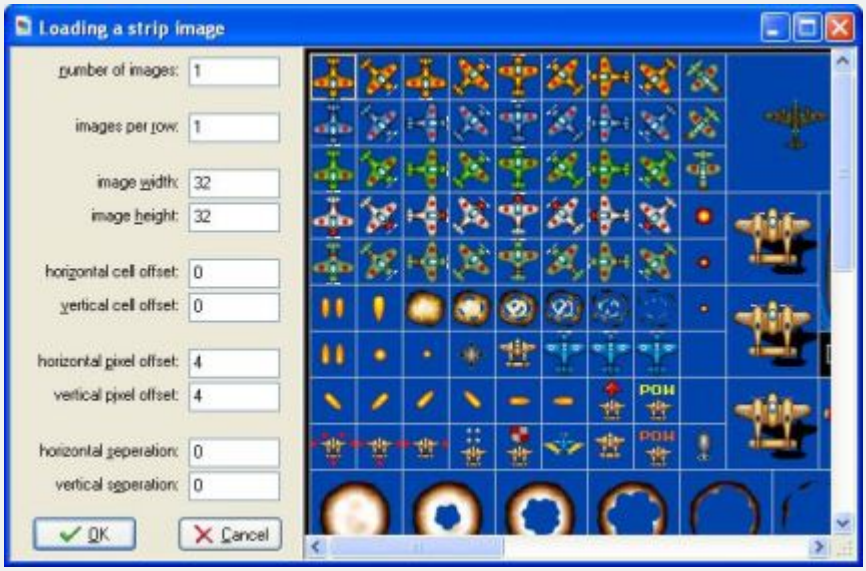
In particular the last two commands are very powerful. For example, to blow up an object, add a number of copies and then a number of empty frames. Then overlay it with an explosion animation. (Make sure the numbers of images match.) Alternatively, morph it to the explosion. With some practice you can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

Strips

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



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



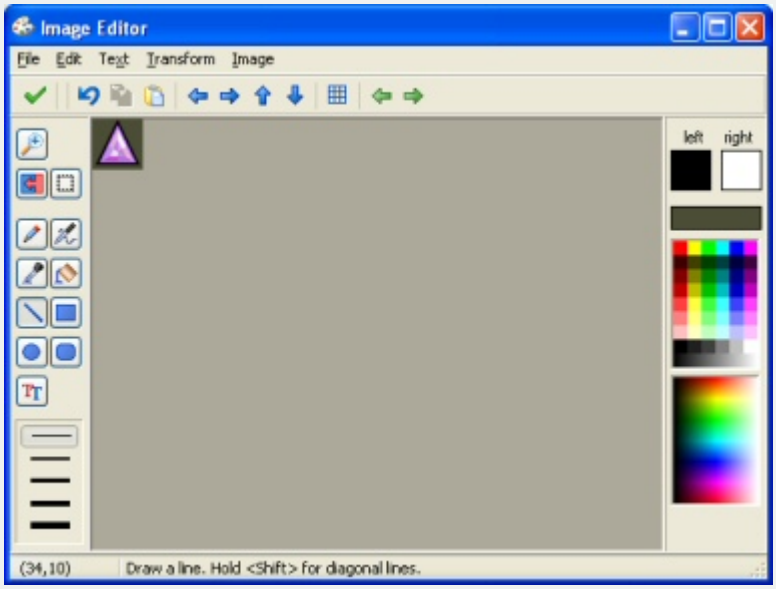
At the right you see (part of) the strip image you selected. At the left you can specify a number of parameters that specify which subimages you are interested in. Note that one or more rectangles in the image indicate the images you are selecting. The following parameters can be specified:

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

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

Editing individual subimages

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



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

special buttons to move all pixels in the image in a particular direction. Also you can indicate whether to show a grid when the image is zoomed (works only with a zoom factor of at least 4).

You can select areas in the usual way by pressed the select button and then drawing a rectangle. Next you place the mouse inside the selected area to move it somewhere else. Normally the original area is filled with the left mouse color. If you move the selection with the <Shift> key pressed, the original area will remain unaffected. With the <Shift> key you can also make multiple copies of the selected area. If you use the right mouse button for moving rather than the left mouse button, the selection is considered to be transparent.

The text tool might required some additional explanation. To add a text, press the text button and then click on the image. A pop-up window appears in which you can enter the text. Use the # symbol to insert a newline. Once you press **OK** the text is put in the image, with a box around it. You can now move the text by pressing with the mouse in the box and dragging the text. You can change the text by clicking with the right mouse button in the box. Using the **Text** menu you can also change the alignment and the font to be used.

At the right of the form you can select the colors to be used (one by the left mouse button and one by the right button). There are four ways to change the color. First of all you can click with the mouse button (left or right) in one of the 16 basic colors. Note that there is a special color box that contains the color of the bottom-left pixel of the image that is used as transparency color if the sprite is transparent. You can use this color to make part of your image transparent. The second way is to click in the image with

the changing color. Here you choose many more colors. You can hold down the mouse to see the color you are selecting. Thirdly, you can click with the left mouse in the boxes indicating the left and right color. A color dialog pops up in which you can select the color. Finally, you can select the color dropper tool at the left and click on a position in the image to copy the color there.

There are two special features. When you hold the <Ctrl> key you can pick a drawing color from the current image. When you hold the <Shift> key while drawing lines you will only get horizontal, vertical, or diagonal lines. Similar, when you hold the <Shift> key while drawing ellipses or rectangles you will get circles and squares.

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

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

Note that more fancy drawing routines are missing. For this you should use a more advanced drawing program (or simply the paint program that comes with Windows). The easiest way to do this is to use the copy button to put the image on the clipboard. Now in your painting program, use

paste to get it. Change it and copy it to the clipboard. Now, in *Game Maker* you can paste the updated image back in.

Advanced sprite settings

In advanced mode, in the sprite properties form there are a number of advanced options that we will treat here.

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

The edges of sprites can look a bit blocky. To avoid this you can check the box labelled **Smooth edges**. In this case pixel at the edges of the sprite (that is, pixels that are neighbors to transparent pixels) are drawn partially transparent. This can make them look a lot nicer. (Don't use this when the sprites need to match up to form larger shapes because in that case a partially transparent line will appear between the parts.) The effect of this setting is only visible in the game, not in the editor!

During gameplay, sprites are turned into textures. Textures must be moved to video memory (on the graphics card)

before they can be used. When the box **Preload texture** is checked this happens immediately when the game loads such that there will be no delay during the game. If you have lots of large sprites though that are not used in the beginning you might want to uncheck this option. *Game Maker* will then swap textures to video memory and back when required.

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

More about sounds and music

In advanced mode you have a lot more control over the sounds and pieces of music you add to your game. When you add a sound resource the following form will show:

Besides the buttons to load, save, and play sounds there are a lot of settings now that will be discussed here.

First of all you can indicate the kind of sound. Four kinds are possible. Normal sounds are in general used for sound effects in wave files (although they can be used for midi files as well). Multiple normal sounds can play at the same time. You can even play multiple copies of the same sound simultaneously. Background music is similar to normal sounds but only one can play at any moment. So once you start a new background sound, the currently playing one will be stopped. Midi files are default background music. 3D sound is sound for which you can apply 3D settings through

special functions. You will only need these for advanced sound effects.

Sound files are normally played through DirectX. This gives many possibilities but is limited to wave and midi files. If you want to play other files, like mp3 files, you should select the option to use the media player. This is much more limited though. No volume changes or effects can be used and only one piece can play at once. Note that midi files, when played through the media player may sound different from playing them as background or normal sounds. The reason is that the media player uses the hardware synthesizer (which is different on each machine) while otherwise a software version is used (which sounds the same on all machines). Preferably don't use mp3 files in you games. They need to be decompressed which takes processing time and might slow down the game. The fact that the file size is smaller does not mean that they use less memory. Also, not all machines support them. So your game might not run on all machines.

Secondly, you can indicate some sound effects, like chorus or echo (only in the Pro Edition of *Game Maker*!) You can select any combination. You can immediately listen to the results. (When using GML code you can even change the parameters of these effects.)

Also you can indicate the default volume of the sound and whether to pan it to the left or the right speaker.

For all sounds you can indicate whether they should be preloaded or not. When a sound is played it must be loaded into audio memory. If you preload the sound this is done at the start of the game, making it immediately available for playback. When not, it is loaded the first time it is used.

This will save memory but might give a small delay the first time the sound is used.

Game Maker does not have a built-in sound editor. But in the preferences you can indicate external editors that you want to use for editing sounds. If you selected these you can press the button labeled **Edit Sound** to edit the current sound. (The *Game Maker* window will be hidden while you edit the sound and returns when you close the sound editor.)

More about backgrounds

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

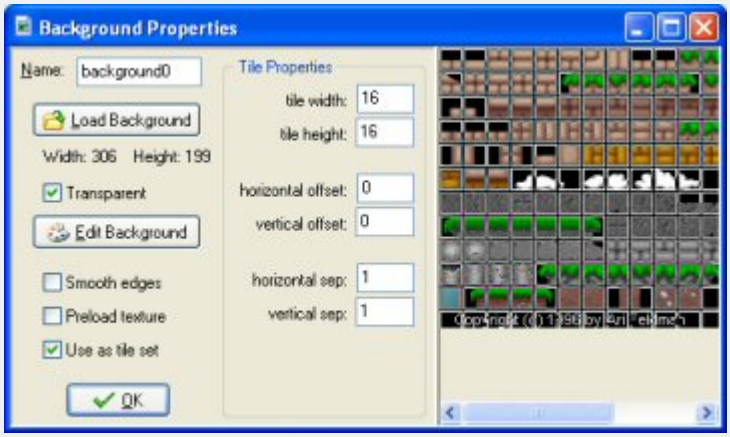
In advanced mode, the background property form has a number of advanced options.

The edges of backgrounds, in particular when made transparent, can look a bit blocky. To avoid this you can check the box labelled **Smooth edges**. In this case pixel at the edges of the background (that is, pixels that are neighbors to transparent pixels) are drawn partially transparent. This can make them look a lot nicer. (Don't use this when the backgrounds need to match up to form larger shapes because in that case a partially transparent line will appear between the parts.) The effect of this setting is only visible in the game, not in the editor!

During gameplay, backgrounds are turned into textures. Textures must be moved to video memory (on the graphics card) before they can be used. When the box **Preload texture** is checked this happens immediately when the game loads such that there will be no delay during the game. If you have lots of large backgrounds though that are not used in the beginning you might want to uncheck this

option. *Game Maker* will then swap textures to video memory and back when required.

Sometimes you want to use a background as a set of tiles, that is, a collection of smaller images in one big image. When create the rooms you can then add these subimages at different places in the room. This is very useful for creating nice looking levels. To use a background as a tile set, check the box labelled **Use as tile set**. The form now changes to look as follows:



You can indicate a number of settings for the tile set. In particular you can indicate the width and height of each tile. (Only one size can be given, so better make sure that all tiles in the set have the same size. If you have different sizes, create two or more tile sets.) You can also indicate an offset where the top leftmost tile starts. Finally, a separation between the tiles (this is normally 0 or 1) can be indicated. For more information on using tiles, see the chapter on creating rooms.

A word of warning is required here. When you put separating borders between the sprites and use interpolation between pixels (see the global game settings) this can result in cracks between the tiles. Better make sure

the pixels around the tiles actually match with the pixels just inside the tiles to avoid this.

More about objects

When you create an object in advanced mode, you can change some more advanced settings.

Depth

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

Persistent objects

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

Parents

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

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

It is actually good practice in such cases to create one base object. This base object contains all the default behavior but is never used in the game. All actual objects have this base object as parent. Parent objects can again have parents, and so on. (Obviously you are not allowed to create cycles.)

In this way you can create an object hierarchy. This is extremely useful to keep your game structured and you are strongly advised to learn to use this mechanism.

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

As indicated, wherever you use an object, this also implies the descendants. This happens when, in an action, you indicate that the action must be applied to instances of a certain object. It also happens when you use the `with()` statement in code (see below). And it works when you call functions like `instance_position`, `instance_number`, etc. Finally, it works when you refer to variables in other objects. In the example above when you set `ball1`.speed to 10 this also applies to ball2, ball3 and ball4.

Masks

When two instances collide a collision event occurs. To decide whether two instances intersect, the sprites are used. This is fine in most cases, but sometimes you want to base collisions on a different shape. For example, if you make an isometric game, objects typically have a height (to give them a 3D view). But for collisions you only want to

use the ground part of the sprite. This can be achieved by creating a separate sprite that is used as collision mask for the object.

Information

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

More actions

In advanced mode there are a number of additional actions available which will be described here.

Information on the different additional actions can be found in the following pages:

More Move Actions More Main Actions
More Control Actions
More Draw Actions
Particle Actions
Extra Actions

More move actions

Some additional move actions are available in advanced mode. The following actions are added:

 **Set Path** With this action you can specify that the instance should follow a particular path. You indicate the path that must be followed and the speed in pixels per step. When the speed is positive the instance starts at the beginning of the path. If it is negative it starts at the end. Next you specify the end behavior, that is, what should happen when the end of the path is reached. You can choose to stop the motion, restart from the beginning, restart from the current position (which is the same when the path is closed), or reverse the motion. Finally you can indicate that the path must be seen as absolute, that is, the position will be as indicated in the path (this is useful when you have designed the path at a particular place in the room) or relative, in which case the start point of the path is placed at the current location of the instance (end point when speed is negative). See the chapter on paths for more information.

 **End Path**
Use this action to stop the path for the instance.

 **Path Position**
With this action you can change the current position of the instance in the path. This must be a value between 0 and 1 (0=beginning, 1=end).

 **Path Speed**
With this action you can change the speed of the instance

on the path. A negative speed moves the instance backwards along the path. Set it to 0 to temporarily stop the motion along the path.

 **Step Towards**

This action should be placed in the step event to let the instance take a step towards a particular position. When the instance is already at the position it will not move any further. You specify the position to move to, the speed with which to move, that is, the size of the step, and whether the motion should stop when hitting a solid instance or when hitting any instance.

 **Step Avoiding**

This is a very powerful motion action. It should be placed in the step event. Like the previous action it lets the instance take a step towards a particular position. But in this case it tries to avoid obstacles. When the instance would run into a solid instance (or any instance) it will change the direction of motion to try to avoid the instance and move around it. The approach is not guaranteed to work but in most easy cases it will effectively move the instance towards the goal. For more complicated cases, there are motion planning functions. You specify the position to move to, the speed with which to move, that is, the size of the step, and whether the motion should avoid solid instances or any instance.

More main actions

Some additional main actions are available in advanced mode. The following actions are added:

 **Set Time Line** (Only available in advanced mode.) With this action you set the particular time line for an instance of an object. You indicate the time line and the starting position within the time line (0 is the beginning). You can also use this action to end a time line by choosing No Time Line as value.

 **Time Line Position**
(Only available in advanced mode.) With this action you can change the position in the current time line (either absolute or relative). This can be used to skip certain parts of the time line or to repeat certain parts. For example, if you want to make a looping time line, at the last moment, add this action to set the position back to 0. You can also use it to wait for something to happen. Just add the test action and, if not true, set the time line position relative to -1.

 **Show Video**
With this action you can show a video/movie file. You specify the file name and whether it should be shown full screen or in the game window. Make sure the video file exists. You should either distribute it with the game or include it in the game through the **Global Game Settings**. ***This action is only available in the Pro Edition.***

 **Replace Sprite**
This action can be used to replace a sprite from the contents of a file. You indicate the sprite you want to

replace, the filename and the number of subimages in the sprite. Many different images file formats are supported, e.g. .bmp, .jpg, .tif, and .gif. For a gif file the number of subimages is automatically decided based on the number of subimages in the gif file. Other settings for the sprite, e.g. whether it is transparent or not, are not changed. You can use this action to avoid storing all sprites in the program itself. For example, at the beginning of a level you can replace sprites by the actual character sprites you want to use. DON'T change a sprite that is at that moment being used in an instance in the room. This might give unwanted effects with collisions. ***This action is only available in the Pro Edition.***

 **Replace Sound**

With this action you can replace a sound by the contents of a file (.wav, .mid, or .mp3). You specify the sound and the filename. This avoids having to store all the sounds in the game itself. For example, you can use different pieces of background music and pick the one you want to play. DON'T change a sound while it is playing. ***This action is only available in the Pro Edition.***

 **Replace Background**

With this action you can replace a background by the contents of a file. Many different images file formats are supported, e.g. .bmp, .jpg, .tif, and .gif. You specify the background and the filename. This avoids having to store all the backgrounds in the game itself. DON'T change a background that is visible. ***This action is only available in the Pro Edition.***

More control actions

Some additional control actions are available in advanced mode. The following actions are added:

 **Execute Script** With this action you can execute a script that you added to the game. You specify the script and the maximal 5 arguments for the script.

Call Parent Event
This action is only useful when the object has a parent object. It calls the corresponding event in the parent object.

More draw actions

The following additional draw action is available in advanced mode:

 **Set Font** You can set the font that is from this moment on used for drawing text. This must be one of the font resources you have define. If you choose No Font a default 12 point Arial font is used.

Particle actions

A set of action dealing with particles is available on the **Extra** tab. ***These actions are only available in the Pro Edition of Game Maker***.

Particle systems are meant to create special effects. Particles are small elements (represented by a pixel or a little shape). Such particles move around according to predefined rules and can change color while they move. Many such particles together can create e.g. fireworks, flames, rain, snow, star fields, flying debris, etc.

Game Maker contains an extensive particle system that can be accessed through functions. A more limited particle system can be accessed through the actions described below.

A particle system can deal with particles of different types. After creating the particle system the first thing to do is specify the particle types. Using the actions below you can specify up to 16 types of particles. Each type has a shape, a size, a start color and an end color. The color slowly changes from the start color to the end color. Particles have a limited life time. In the type you specify the minimal and maximal life time of the particles. Particles also have a

speed and a direction. Finally, gravity and friction can work on particles.

After you specify the particle types you must create them at places in the room. You can either burst a number of particles of a particular type from a place or you can create a constant stream of particles appearing. Particles appear at emitters. The particle system can have up to 8 emitters working at the same time. So after you create the particle types you must create the emitters and tell them to burst or stream particles.

Here is the complete set of actions. Best experiment with them to get the required effect.

 **Create Part System** This action creates the particle system. It must be called before any other actions can be used. You only need to call it once. You can specify the depth at which the particles are drawn. If you use a large positive depth the particles appear behind the instances. If you use a negative depth they appear in front of the instances.

 **Destroy Part System**
This action destroys the particle system, freeing all its memory. Don't forget to call this (e.g. when you move to a different room) because particle systems use a lot of storage.

 **Clear Part System**
This action removes all the particles currently visible. It does not stop the emitters so new particles might be created again if you have streaming emitters (see below).

 **Create Particle**
With this action you create a particle type. You can choose

one of the 16 types available. For the particle type you can specify its shape or the sprite to be used for it. If you specify a sprite the sprite will be used. If you set the sprite to no sprite, the shape will be used. There are a number of interesting built-in shapes. You also indicated its minimal and maximal size (when the particle appears a random value between these bounds is used). Finally you specify the increase in size in each step. For a decrease, use a negative value. Note that only a particle type is created, not an actual particle. For this you need emitters (see below).

 **Particle Color**

A particle can have a color (default the color is white). With this action you can set the color to be used for a particular type. You must indicate the particle type the color is defined for. Next you specify how a color is applied. Either a random color is choosen between two given colors, or the color starts with the first color and than gradually over the lifetime of the particle, it changes to the second color. Both colors must be given. Finally you can indicate the alpha transparency. You specify the transparency at the moment the particle is created and when it dies. The transparency slowly changes between these values. It is normally nice to decrease the alpha value over the lifetime of a particle.

 **Particle Life**

A particle lives for a limited number of steps. After this is disappears. With this action you set the life time for a particle type. You give two limiting values and the actual life time is chosen randomly between them.

 **Particle Speed**

With this action you can set the speed and direction of motion for a particle type. Again you give two limits and the actual value is chosen randomly between them. For example, to make the particle move in a random direction,

give 0 and 360 for the limits for the direction. You can also specify a friction. This amount is subtracted from the speed in each step until it becomes 0. (You can make a particle speed up by using a negative friction.)

 **Particle Gravity**
With this action you set the amount of gravity and direction of gravity for a particular particle type. 270 is downwards.

 **Particle Secondary**
This is a bit more complicated. Particles can create other particles during their life time and when they die. With this action you can specify this. You can define the type and number of particles that must be created at each step during the life time and you can specify the type and number of particles that must be created when the particle dies. Be very careful here. You can easily create huge numbers of particles in this way, slowing down the system considerably. For the numbers you can also use a negative value. A negative value x means that in each step a particle is created with chance -1/x. So for example, if you want to generate a secondary particle about every 4 steps, use a value of -4. Secondary particles are great for creating effects like tails of particles or exploding particles.

 **Create Emitter**
This action creates a particle emitter. Particles are generated by emitters. You can have up to eight emitters. Choose the emitter and specify the shape of it and its size and position (in the form of a bounding box).

 **Destroy Emitter**
This action destroys the indicated emitter. Note that existing particles that came from this emitter are not removed.

 **Burst from Emitter**

Even if you defined a particle type and an emitter there are still no particles. You still have to tell the emitter to generate the particles. With this action you tell a particular emitter to generate a given number of particles of a given type. All these particles are generated at once. For the number you can also use a negative value. A negative value x means that a particle is created with chance -1/x. So for example, if you want to generate a particle with a chance of 25 percent, use a value of -4.

 **Stream from Emitter**

With this action you tell a particular emitter to stream a given number of particles of a given type. In each step this number of particles is generated, leading to a continuous stream of particles. The emitter will continue streaming particles until you destroy it or tell it to stream 0 particles. For the number you can also use a negative value. A negative value x means that in each step a particle is created with chance -1/x. So for example, if you want to generate a particle about every 4 steps, use a value of -4.

Extra actions

On the **Extra** tab there also are a number of actions related to the playing of CDs. ***These actions are only available in the Pro Edition of Game Maker***.

 **Play CD** With this action you can play some tracks from a CD in the default CD-drive. You specify the starting track and the final track.

 **Stop CD**
Stops the currently playing CD.

 **Pause CD**
Pauses the currently playing CD.

 **Resume CD**
Resumes a paused CD.

 **Check CD**
If there is a CD in the default drive then the next action is executed.

 **Check CD Playing**
If a CD is playing in the default drive then the next action is executed.

Finally, there are two additional actions that can be useful in certain games.

 **Set Cursor**
You can use this action to replace the windows mouse cursor by a sprite. You specify the sprite and whether the windows mouse cursor should still be shown. The sprite can

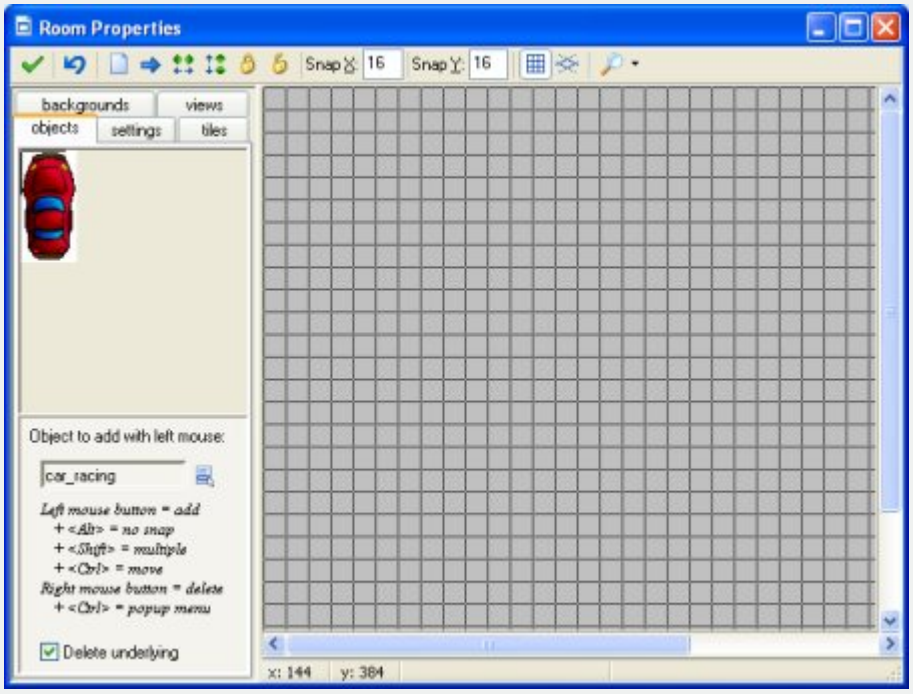
be animated. Note that the sprite will only be shown in the room, not in the area outside it.

 **Open Webpage**

You can indicate a web address in this action. This webpage is then opened in the default browser on the machine. (The action can actually also be used to open other documents.) The action does not work in secure mode.

More about rooms

Rooms in *Game Maker* have many options. Before we only treated the most important ones. In this chapter we will discuss the other options. When you open the room form in advanced mode it looks as follows:



As you will see, some new buttons have been added to the toolbar. There are buttons to sort the instances horizontally or vertically. This is useful when instance partially overlap. (When adding tiles these buttons and the others work on the tiles rather than the instances.) Also there are buttons to lock all instances or unlock all instances. Locked instances cannot be moved or deleted. This saves you from incidentally removing instances. Using the right mouse button menu (hold <Ctrl> and right click on an instance) you can also lock or unlock individual instances.

Finally, you can indicate that you want to use an isometric grid. This is very useful when creating isometric games. First of all, the grid lines now run diagonally. Also the snapping of instances is different. (It works best when the origin of the instance is at the top left corner as is default.)

Also there are two new tabs which we will discuss below.

Information on the different advanced room options can be found in the following pages:

Advanced settings Adding tiles
Views

Advanced settings

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

Secondly, there is a button labeled **Creation code**. Here you can type in a piece of code in GML (see later) that is executed when the room is created. This is useful to fill in certain variables for the room, create certain instances, etc. It is important to understand what exactly happens when you move to a particular room in the game.

- First, in the current room (if any) all instances get a room-end event. Next the non-persistent instances are removed (no destroy event is generated!).
- Next, for the new room the persistent instances from the previous room are added.
- All new instances are created and their creation events are executed (if the room is not persistent or has not been visited before).
- When this is the first room, for all instances the game-start event is generated.

- Now the room creation code is executed.
- Finally, all instances get a room-start event.

So, for example, the room-start events can use variables set by the creation code for the room and in the creation code you can refer to the instances (both new ones and persistent ones) in the room.

There is one further option. In the pop-up menu when you right click on an instance with the <Ctrl> key you can now indicate some creation code for the specific instance. This code is executed when the room is started, just before the creation event of the instance is executed. This is very useful to e.g. set certain parameters that are specific to the instance.

Adding tiles

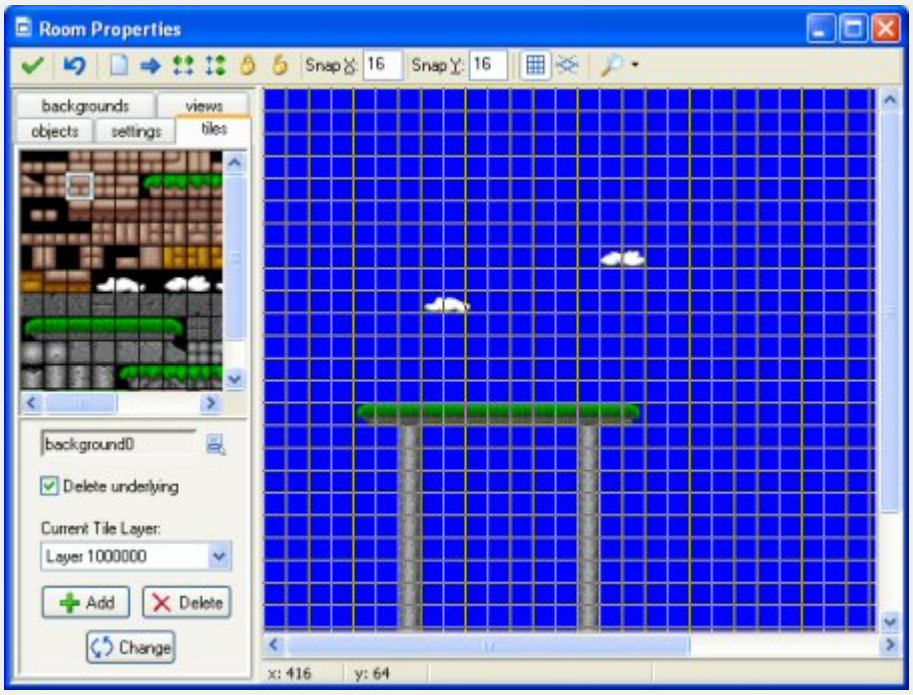
You can also create so-called tiled background. The reason for this is as follows: In many games you will want to have nice looking backgrounds. For example, in a maze game, the walls of the maze should nicely match up, and in platform games you like to see beautifully drawn platforms, trees, etc. You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

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

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

To add tiles to your room you first need to add a background resource to your game that contains the tiles. If you want to have your tiles partially transparent, make sure you make the background image transparent. When adding the background resource indicate that it must be used as a tile set. Next indicate the size of each tile and whether there is room between the tiles, as was indicated in the chapter on background resources.

Now, when defining your room, click on the tab **tiles**. The following form is shown (actually, we already added some tiles in this room).



At the left top there is the current set of tiles used. To select the set, click on the menu button below it and select the appropriate background image.

Now you can add tiles by selecting the tile you want at the top left, and next clicking at the appropriate place in the room at the right. This works in exactly the same way as for adding instances. Underlying tiles are removed, unless you

uncheck the box **Delete underlying**. You can use the right button to delete tiles. Hold the <Shift> key to add multiple tiles. And hold the <Ctrl> key to move tiles to a new place. The <Alt> key will avoid snapping to the gird. Also there is again a pop- up menu when you hold the <Ctrl> key and click on a tile with the right mouse button. The buttons in the toolbar will now clear all tiles, shift all tiles, sort the tiles or lock/unlock them. (Actually they only operate on the current layer; see below.)

In some situations you might want to put a part of the background in the room that is not exactly the size of a tile or consists of multiple tiles. This can be done as follows. In the top-left image press the left mouse button while holding the <Alt> key. Now you can drag an area which you can then place in the room in the same way as tiles. To select multiple tiles, hold the <Shift> key. Note that this only works correctly when there is no separation between the tiles. If you want to select an area that is multiple of the room grid size, hold the <Ctrl> key rather than the <Shift> key. (Note that you can actually change the key you hold during the dragging. This can sometimes be useful.)

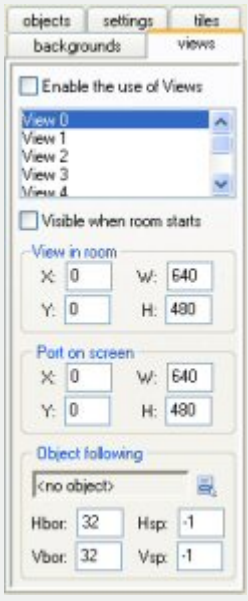
Tiles can be placed in layers at different depths. At the bottom you see the current depth. Default this is 1000000 which is normally behind all instances. So the instances will move in front of the tiles. You can use the **Add** button to add new tile layers, each with a different depth. Negative depths can be used to put tiles in front of instances. If you also give objects different depths you can put them between different tile layers. If you press **Delete** you delete a tile layer together with all its tiles. (There must always be at least one layer.) If you press **Change** you can change the depth of a tile layer. If you give it the same depth as another layer, the layers are merged.

Using tiles is a powerful feature that should be used as much as possible. It is much faster than using objects and the tile images are stored only once. So you can use large tiled rooms with very little memory consumption.

Views

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

When you click the tab labeled **views** the following information will show:



At the top there is a box labeled **Enable the use of Views**. You must check this box to use views. Below this you see

the list of at most eight views you can define. Below the list you can give information for the views. First of all you must indicate whether the view should be visible when the room starts. Make sure at least one view is visible. Visible views are shown in bold.

A view is defined by a rectangular area in the room. This is the area that must be shown in the view. You specify the position of the top-left corner and the width and height of this area. Secondly, you must specify where this area is shown in the window on the screen. This is called the (view)port. Again you specify the position of the top-left corner and the size. If you have a single view the position is typically (0,0). Note that the size of the port can be different from the size of the view. In this case the view will be scaled to fit in the port. (In code it is also possible to rotate a view.) The ports can overlap. In this case the are drawn in the indicated order on top of each other.

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

Fonts

When you want to draw text in your game this text is drawn in an Arial 12 points font. To make more fancy looking texts you probably want to use different fonts. To use different fonts you must create font resources. In each font resource you specify a particular type of font which can then be used in your game using the action to set a font.

To create a font resource in your game, use the item **Create Font** in the **Resources** menu or use the corresponding button on the toolbar. The following form will pop up.

As always you should give your font resource a name. Next you can pick the name of the font. Also you can indicate its size and whether it should be bold and/or italic. Realize that large fonts take a lot of memory to store. So you are strongly recommended not to use fonts with a size larger than say 32. (It is possible to scale the fonts while running

the game.) An example of the indicated font is shown at the bottom.

A font typically consist of 256 characters, numbered from 0 to 255. But in general you use only a small portion of these. So default in a font only the characters from 32 till 127 are stored in the font. The more characters you store in the font the more memory it will use. You can change the character range used. To see the index of each character you can use the Character Map that can be found in the Windows Start menu under Accessories/System Tools. Some standard ranges can be indicated using the buttons: The **Normal** range from 32 till 127, the **All** range from 0 till 255, the **Digits** range that only contains the 10 digits, and the **Letters** range that contains all uppercase and lowercase letters. Other ranges can be used by typing in the first and last character index. If a character does not lie in the range it is replaced by a space.

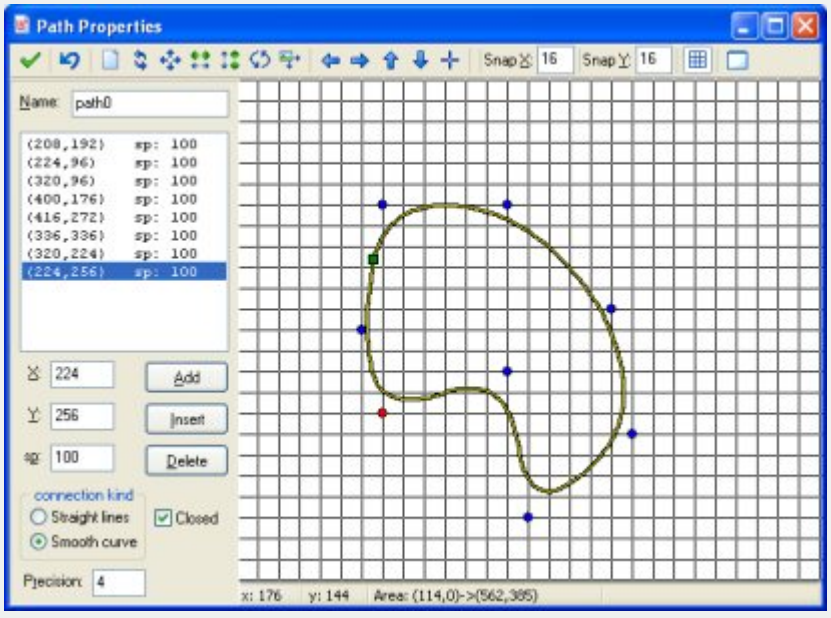
Normally you have a large number of different fonts installed on your computer and there are websites where you can download hundreds more. The problem is that if you use them in your game and then give your game to somebody else to play there is a big chance that the other person does not have the same fonts installed on his or her computer. To avoid this, *Game Maker* embeds all the fonts you want to use in the game file, but only in the stand-alone games. So if you give a stand-alone game to someone else you don't need to provide the font files. But if you give somebody the editable version of your game and you use fancy fonts, you better also provide the person with the correct font files.

Paths

In more advanced games you often want to let instances follow certain paths. Even though you can indicate this by e.g. using timer events or code, this is rather complicated. Path resources are an easier mechanism for this. The idea is rather simple. You define a path by drawing it. Next you can place an action in the creation event of the object to tell the object to follow the particular path. This chapter will explain this in detail.

Defining paths

To create a path in your game, choose **Create Path** from the **Resources** menu. The following form will pop up (in the example we already added a little path).



At the top left of the form you can set the name of the path, as usual. Below it you find the points that define the path.

Each point has both a position and a speed (indicated with sp). Depending on how you use the path, the position is either absolute, that is, the instance for which you will later use the path will follow it at that particular place, or it is relative, that is, the instance will always start at the first position on the path and follow the path from there. The speed should be interpreted as follows. A value of 100 means the original speed given for the path when assigning it to the instance. A lower value reduces the speed, a higher value increases it (so it indicates the percentage of the actual speed). Speed will be interpolated between points, so the speed changes gradually.

To add a point press the button **Add**. A copy is made of the currently selected point. Now you can change the actual position and speed by changing the values in the edit boxes. Whenever you select a point in the list, you can also change its values. Press **Insert** to insert a new point before the current one, and **Delete** to delete the current point.

At the right of the form you will see the actual path. The red dot indicates the currently selected control point. The blue dots are the other control points. The green square indicates the position where the path starts. You can also change the path using the mouse. Click anywhere on the image to add a point. Click on an existing point and drag it to change its position. When you hold <Shift> while clicking on a point, you insert a point. Finally, you can use the right mouse button to remove points. (Note that you cannot change the speed this way.) Normally the points will be aligned with a grid. You can change the grid settings at the top tool bar. Here you can also indicate whether the grid should be visible or not. If you want to precisely position a point, hold the <Alt> key while adding or moving it.

You can influence the shape of the path in two ways. Fist of all you can use the type of connection. You can either choose straight line connections or a smooth path. Secondly, you can indicate whether the path must be closed or not.

On the toolbar there are a number of important controls. From left to right they have the following meaning. The first button indicates that you are ready and want to close the form, keeping the changes. (If you want to discard the changes, press the cross to close the window and indicate that you do not want to save the changes.) Next there is the button to undo the last change.

The following set of toolbar buttons allows you the clear the path, reverse the order in which the path is traversed, shift the path, mirror it horizontally, flip it vertically, rotate it, and scale it. Next there are buttons to shift the view (not the path itself; the actual view area is indicated in the status bar at the bottom) and to center the view.

As already indicated above you can next set the snap values and whether to show the grid. Finally there is a button to indicate that you want to see a particular room as background for the path. Using this you can easily put the path at a particular place in the room, for example on a race track, so that later the instances will follow the correct route. (This only makes sense when you use absolute paths; see below.)

Assigning paths to objects

To assign a path to an instance of an object, you can place the path action in some event, for example in the creation event. In this action you must specify the path from the

drop down menu. There are some further values you can provide.

You must indicate the path that must be followed and the speed in pixels per step. When the speed is positive the instance starts at the beginning of the path. If it is negative it starts at the end. Remember that when you defined the path you specify the actual speed relative to this indicated speed. There is also an action to change the speed with which the path is executed. You could, for example, use this to let an instance slow down or speed up along its path. Note that the normal speed of the instance is ignored (actually set to 0) when executing a path. Also things like gravity and friction do not influence the motion along a path.

Next you specify the end behavior, that is, what should happen when the end of the path is reached. You can choose to stop the motion and end the path. You can also restart the path from the beginning, that is, the instance jumps back to the position where the path was started and executes the path again. A third option is to restart from the current position, that is, the instance follows the path again but now with this new starting position (this is the same when the path is closed). Finally you can choose to reverse the motion, making the instance go back and forth along the path. Note that also at the end of the path an event happens; see below.

Finally you can indicate whether the path must be absolute or relative. An absolute path is executed at the place where it is defined. The instance is placed at the start position and moved from there (end position when speed is negative). This is, for example, useful when you have a race track on which you have defined the path. When you choose relative the instances starts executing the path from its current

position. This is useful when an instance should make a local motion. For example, space ships in a space invader game can make a particular turn from their current position.

When you want to place the instance at a different point along its path you can use the action to set the path position. A path position always lies between 0 and 1, 0 indicating the start position and 1 the end position on the path. Note that in each step the direction variable is automatically set to the correct direction along the path. You can use this variable to choose the correct orientation for the sprite.

When using scripts or pieces of code you have more control over the way the path is executed. There is a function to start a path for an instance. The variable `path_position` indicates the current position on the path (between 0 and 1 as indicated above). The variable `path_speed` indicates the speed along the path. A variable `path_scale` can be used to scale the path. A value of 1 is the original size. A larger value indicates that the path is made larger; a smaller value makes it smaller. The variable `path_orientation` indicates the orientation in which the path is executed (in degrees counter-clockwise). This enables you to execute the path in a different orientation (e.g. moving up and down rather than left and right). There is also a variable to control the end behavior. Finally there are lots of functions to ask for properties of paths (e.g. the x and y coordinate at a certain positions) and there are functions to create paths. There are even functions that create collision free paths for an instance to reach a certain goal. See the later sections on GML for details on this.

You might wonder what happens when the instance collides with another instance while it follows a path. Basically the same happens as when the instance moves with a speed.

When there is a solid instance, the instance is placed back at its previous location. When both instances are not solid they are placed at their new positions Next the collision event(s) are executed and it is checked whether the collision has been resolved. If not and the other instance is solid the instance will stop, as it should (assuming there is a collision event defined). Also, the `path_position` variable is not increased. When the blocking instance disappears the instance will continue to follow its path. To handle collisions yourself the variable `path_positionprevious` can be useful. It holds the previous position for the path and you can set the path position to this variable to avoid advancing along the path.

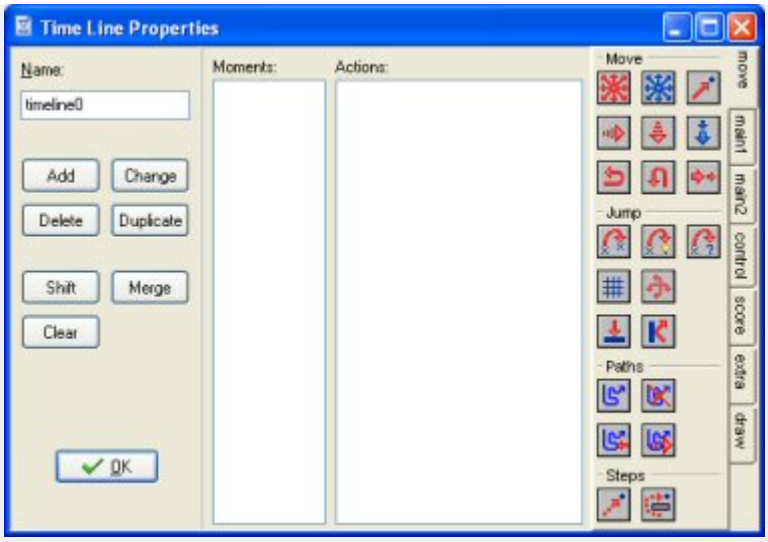
The path event

As described above, you can indicate what must happen when the instance reaches the end of the path. At this moment also an **End of Path** event occurs. You can find it under the **Other** events. Here you can place actions. For example, you might want to destroy the instance, or let it start a new (different) path.

Time Lines

In many games certain things must happen at certain moments in time. You can try to achieve this by using alarm events but when things get too complicated this won't work any more. The time line resource is meant for this. In a time line you specify which actions must happen at certain moments in time. You can use all the actions that are also available for the different events. Once you create a time line you can assign it to an instance of an object. This instance will then execute the actions at the indicated moments of time. Let us explain this with an example. Assume you want to make a guard. This guard should move 20 time steps to the left, then 10 upwards, 20 to the right, 10 downwards and then stop. To achieve this you make a time line where you start with setting a motion to the left. At moment 20 you set a motion upward, at moment 30 a motion to the right, at moment 50 a motion downwards and at moment 60 you stop the motion. Now you can assign this time line to the guard and the guard will do exactly what you planned. You can also use a time line to control your game more globally. Create an invisible controller object, create a time line that at certain moments creates enemies, and assign it to the controller object. If you start to work with it you will find out it is a very powerful concept.

To create a time line, choose **Create Time Line** from the **Resources** menu. The following form will pop up.



It looks a bit like the object properties form. At the left you can set the name and there are buttons to add and modify moments in the time line. Next there is the list of moments. This list specifies the moments in time steps at which assigned action(s) will happen. Then there is the familiar list of actions for the selected moment and finally there is the total set of actions available.

To add a moment press the button **Add**. Indicate the moment of time (this is the number of steps since the time line was started). Now you can drag actions to the list as for object events. There are also buttons to delete the selected moment, to change the time for the selected moment, to duplicate a moment and to clear the time line.

Finally there are two special buttons. With the **Merge** button you can merge all moments in a time interval into one. With the **Shift** button you can shift all moments in a time interval forwards or backwards by a given amount of time. Make sure you do not create negative time moments. They will never be executed.

There are two actions related to time lines.

 **Set a time line** With this action you set the particular time line for an instance of an object. You indicate the time line and the starting position within the time line (0 is the beginning). You can also use this action to end a time line by choosing No Time Line as value.

 **Set the time line position**
With this action you can change the position in the current time line (either absolute or relative). This can be used to skip certain parts of the time line or to repeat certain parts. For example, if you want to make a looping time line, at the last moment, add this action to set the position back to 0. You can also use it to wait for something to happen. Just add the test action and, if not true, set the time line position relative to -1.

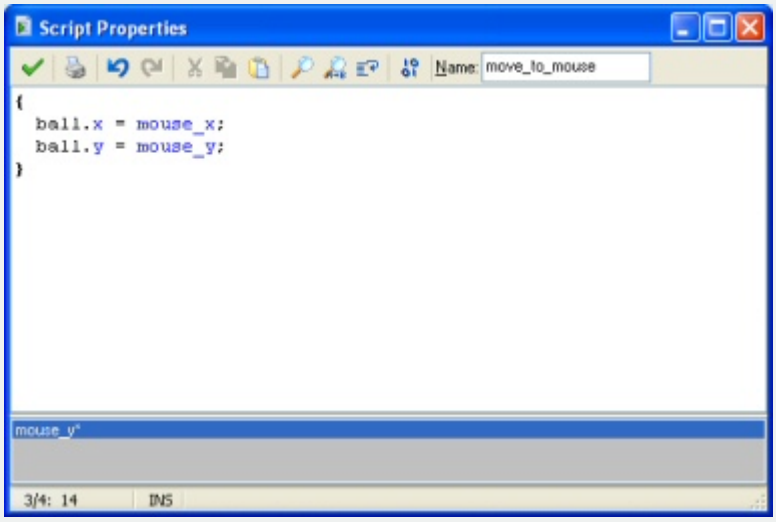
Scripts

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

As stated before, a script is written with code in GML (the built-in programming language) and is meant to perform a particular task. Scripts can takes input-variables called arguments (sometimes called parameters). To execute a script from any event, you can use either the script action or you can use code. In the Script-action you specify the script you want to execute, together with the up to five arguments. If you use a script from within a piece of code it is done the same way you call a GM-function. In that case you can use up to 16 arguments. Scripts can return one value. This is often used to build calculating methods (mathematical methods). The **return** keyword is used for this. No code after the return keyword is executed! When a

script returns a value, you can also use it as a function when providing values in other actions.

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



(Actually, this is the built-in script editor. In the preferences you can also indicate that you want to use an external editor.) At the top right you can indicate the name of the script. You have a little editor in which you can type the script. Note that at the bottom a list is shown of all functions, built-in variables, and constants. This helps you find the one you need. You can double click one to add it (or use <Ctrl>P for the same purpose.) Showing this list can be switched off in the preferences. The editor has a number of useful properties, many available through buttons (press the right mouse button for some additional commands):

- Multiple undo and redo either per key press or in groups (can be changed in the preferences)
- Intelligent auto indent that aligns with the previous line (can be set in the preferences)

- Intelligent tabbing that tabs till the first non space in the previous lines (can be set in the preferences)
- Use <Ctrl>I to indent selected lines and <Shift> <Ctrl>I to unindent selected lines
- Cut and paste
- Search and replace
- Use <Ctrl> + up, down, page-up, or page-down to scroll without changing the cursor position
- Use F4 to open the script or resource whose name is at the cursor position (does not work in the code action; only in scripts)
- Saving and loading the script as a text file

Also there is a button with which you can test whether the script is correct. Not all aspects can be tested at this stage but the syntax of your script will be tested, together with the existence of functions used.

As you might have noticed, parts of the script text are colored. The editor knows about existing objects, built-in variables and functions, etc. Color- coding helps a lot in avoiding mistakes. In particular, you see immediately if you misspelled some name or use a keyword as a variable. Color-coding is though a bit slow. In the preferences in the file menu you can switch color- coding on and off. Here you can also change the color for the different components of the programs. (If something goes wrong with color coding, press F12 twice, to switch it off and back on.) Also you can change the font used in scripts and code.

Scripts are extremely useful to extend the possibilities of *Game Maker*. This does though require that you design your scripts carefully. Scripts can be stored in libraries that can be added to your game. To import a library, use the item **Import scripts** from the file menu. To save your scripts in the form of a library use **Export scripts**. Script libraries are

simple text files (although they have the extension .gml). Preferably don't edit them directly because they have a special structure. Some libraries with useful scripts are included. (To avoid unnecessary work when loading the game, after importing a library, best delete those scripts that you don't use.)

When creating scripts you can easily make mistakes. Always test the scripts by using the appropriate button. When an error occurs during the execution of a script this is reported, with an indication of the type of error and the place. Rarely you will see a popup with the text "Unexpected error occurred during the game" This error message indicate that some problem occurred in windows or in the hardware. Often the reason for this is infinite recursion, lack of memory or insufficient hardware, drivers or firmware. Generally speaking, these errors have to do with problems outside the gm-environment. If you need to check things more carefully, you can run the game in debug mode. Now a form appears in which you can monitor lots of information in your game.

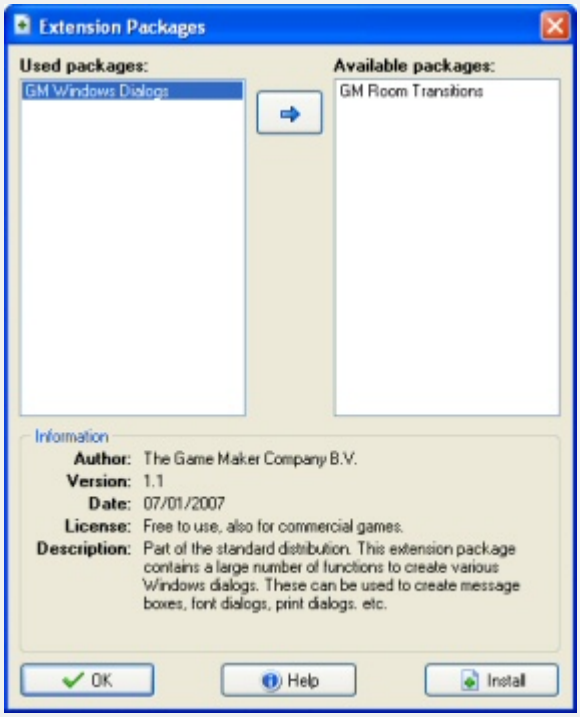
Under the **Run** menu you can pause the game, run it step by step and even restart it. Under the **Watch** menu you can watch the value of certain expressions. Use **Add** to type in some expression whose value is shown in each step of the game. In this way you can see whether your game is doing things the right way. You can watch many expressions. You can save them for later use (e.g. after you made a correction to the game). Under the **Tools** menu you find items to see even more information. You can see a list of all instances in the game, you can watch all global variables (well, the most important ones) and the local variables of an instance (either use the object name or the id of the instance). You can also view messages which you can send from your code using the function `show_debug_message(str)`. Finally you can give the game commands and change the speed of the game. If you make complicated games you should really learn how to use the debug options.

Extension Packages

This functionality is only available in the Pro Edition of Game Maker.

Extension packages extend the possibilities of *Game Maker*. An extension package can add a set of actions to *Game Maker* or it adds a number of additional functions and constants to the GML language built into *Game Maker*. When extension packages are available to you, help about these is placed in the **Help** menu.

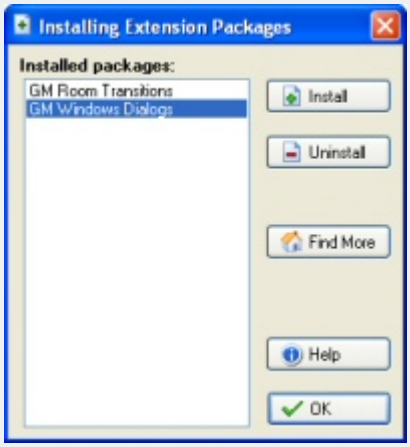
When double clicking on the **Extension Packages** resource item the following form is shown:



In this form you can indicate which extension packages must be used in the game. On the left there is the list of **Used** packages and on the right is the list of **Available**

packages. To move a package from one list to the other, simply select it and press the button between the list. When selecting a package a short description is given at the bottom. Further help is available by pressing the **Help** button.

Extension packages are an extremely powerful part of *Game Maker*. A few extension packages are provided with *Game Maker* but many more will become available through the website. To install new packages or uninstall existing ones, press the **Install** button. The following form is shown



You will see a list of all extension packages installed. By selecting a package and clicking the **Uninstall** button the package is removed from the system.

Install packages are distributed in the form of a .gex file. You can find a number of such packages on our website which you can go to by pressing the button **Find More**. Once you donwloaded such a package file on your computer, press the **Install** button and select the package file. It will then be installed in the system.

If you want to create your own extension packages, please check out the information that is available on http://www.yoyogames.com/extensions.

Finishing your game

When you want to distribute your game you better make sure that it has all the ingredients that make it a great game. Besides the game itself this means that you should provide game information, set the correct global game settings, and take care of speed. This section gives you information about these aspects.

Information on finishing your game can be found in the following pages:

Game Information Global Game Settings
Speed Considerations

Game information

A good game provides the player with some information on how to play the game. This information is displayed when the player presses the <F1> key during game play. To create the game information, double click **Game Information** in the resource tree at the left of the screen. A little built-in editor is opened where you can edit the game information. You can use different fonts, different colors, and styles. Also you can set the background color.

In the **File** menu you can also set a number of **Options**. Default the help is shown in the game window and game play is temporarily halted. Here you can indicate to show the help file in a separate window. In this case you can indicate the caption of the game information during the game. Also you can indicate the position (use -1 for centered) and size of the game information window and whether it should have a border and be sizeable by the player. You can force the information window to stay on top and you can indicate whether the game should continue playing while the information is shown.

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. You might want to indicate at the bottom of the help file that the user must press Escape to continue playing.)

If you want to make a bit more fancy help, use a program like Word. Then select the part you want and use copy and paste to move it from Word to the game information editor. For more advanced games though you probably will not use

this mechanism at all but use some dedicated rooms to show help about the game.

Global Game Settings

There are a number of settings you can change for your game. These change the shape of the main window, set some graphics options, deal with interaction settings, to loading image, constants and information about the creator of the game. Also you can indicate here which files should be included in stand-alone games and how errors should be handled.

The settings can be changed by double clicking on **Global Game Settings** in the resource tree at the left of the screen. They are subdivided in a number of tabbed pages. (Some options are only available in advanced mode.)

Information about the different settings can be found in the following pages:

Graphics and Window Options The Screen Resolution
Various Other Options
Loading Options
Constants
Including Files in Stand-alone Games
Error Options
Information about the Game

Graphics options

In this tab you can set a number of options that are related to the graphical appearance of your game. It is normally useful to check out the effects of these options because they can have a significant effect on the way the game looks. Remember though that different users have different machines. So better make sure that the settings also work on other peoples machines.

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

Scaling
Here you can indicate what happens when the window is larger than the room or when the game is run in full-screen mode. There are three choices. You can indicate a fixed scaling. The room is drawn scaled with the given amount in the center of the window or the center of the screen. 100 indicates no scaling. You typically use fixed scaling when your sprites and rooms are very small. The second option is scale the room such that it fills the window or screen but keep the aspect ratio (ratio between width and height) the same. The third option is to scale such that the window or screen is completely filled. This can lead to distortions in the image (in particular in windowed mode when the user can resize the window).

Interpolate colors between pixels
When checked, colors of pixels in sprites, backgrounds, and tiles that are not aligned with pixels on the screen will be interpolated. This in particular is the case when they are scaled, rotated, or placed at non-integer positions. Interpolation makes movement smoother but can also give

a blurred effect. (Also for tiles it can lead to cracks between them, if not carefully designed.)

Color outside the room region
When the room does not completely fill the window or screen there is some area unused around it. Here you can specify the color of the area.

Allow the player to resize the game window
When checked in windowed mode the user can change the size of the game window by dragging with the mouse at its corners.

Let the game window always stay on top
When checked in windowed mode the game window always stays on top of other windows.

Don't draw a border in windowed mode
When checked in windowed mode the game window will not have a border or a caption bar.

Don't show the buttons in the window caption
When checked in windowed mode the window caption will not show the buttons to close the window or to minimize or maximize it.

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

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

Resolution

In this tab you can set the screen resolution in which your game must run. By default the resolution is not changed. But sometimes you want to run the game in a lower resolutions or you want to set the frequency of the monitor to make sure the timing in the game works correctly. If you want to change the resolution you must first check the box labeled **Set the resolution of the screen**.

There are three things you can change. First of all there is the color depth. This indicates the number of bits used to represent the color for a pixel. Most machines now only allow for 16-bit (High Color) or 32-bit (Full Color) but older machines also allowed for 8-bit and sometimes 24-bit color. *Game Maker* only works correctly in 16-bit and 32-bit color. 32-bit color gives nicer looking images but will take more memory and processing time. If you want your game to run will on most older machines set the color depth to 16-bit. Otherwise use 32-bit or don't change it.

Secondly there is the screen resolution, the number of pixels (horizontal and vertical) on the screen. Changing the resolution is useful when e.g. your rooms are very small. In this case it might help to reduce the resolution of the screen. Realize though that this will also effect other applications running. This can in particular give problems with low resolutions. So in general it is better to only do this when running the game in full screen mode. *Game Maker* will automatically change the resolution back to the starting situation once the game finishes.

Finally you can change the refresh frequency. This indicates how many times per second the image on the screen is

refreshed. If your room speed is larger than the frequency not all steps are actually visible. It works best if the frequency is a multiple of the room speed. (If you specify a frequency that is too high or not available the frequency is not changed.)

There also is a setting here to **Use synchronization to avoid tearing**. This requires some explanation. A display is redrawn a number of times per second, depending on the refresh frequency. If a room is drawn halfway such a refresh, the top of the display will still show the old image while the bottom part shows the new image. This is called tearing. To avoid this you can check this option. In this case the new room image is only copied to the screen when the refresh is not inside the window avoiding the tearing most of the time. The disadvantage is that we normally have to wait till the next refresh. This means that the maximal number of frames is bounded by the frequency of the monitor and, when the processing is not fast enough, the framerate immediately drops to half that number. Also there can be a conflict between the internal timing of the game and the synchronization. If you want to do this best set the monitor frequency to e.g. 60 and also make the room speed either 30 or 60.

Various other options

Here you can set a number of additional options. First of all you can set some default keys:

Let <Esc> end the game When checked, pressing the escape key will end the game. More advanced games normally don't want this to happen because they might want to do some processing (like saving) before ending the game. In this case, uncheck this box and provide your own actions for the escape key.

Treat the close button as <Esc> key
When checked the close button of the window will generate an escape key event. When not checked the close button events are called instead.

Let <F1> show the game information
When checked pressing the F1 key will display the game information.

Let <F4> switch between screen modes
When checked the F4 key will switch between fullscreen and windowed mode.

Let <F5> save the game and <F6> load a game
When checked the player can use <F5> to store the current game situation and <F6> to load the last saved game. (Note that only the basic game data is stored. Once you use advanced features like particles or data structures those settings are not saved and you might have to create a save mechanism yourself.

Let <F9> take a screenshot of a game
When checked the player can use <F9> to create a screenshot of the game. This screenshot will be place in the file screenshotXXX.bmp in the folder where the game is running, where XXX is the number of the screenshot. When creating multiple screenshot the number is automatically increased.

Also you can set the priority of the game process. This priority indicates how much processor time is allotted to the game. In normal mode the operating system tries to give processor time to each process that needs it in some reasonable way. The higher you put the priority the more time is allotted to the game, making it run more smoothly and faster. But other processes get less time (also Windows processes so even the mouse might not move anymore). Use this with care.

Finally, you can set the version information of the game. This information is shown in the helptip that pops up when the user rests his mouse on the game executable. It is also shown when the users clicks with the right mouse button on the program and chooses properties and then version. If you distribute your game you best fill in this information.

You should give your game a version number that consists of four parts: the major version, the minor version, the release number and the build number. Also you should provide the name of the company (you), the name of the product, the copyright information, and a very brief description. All these fields should be at most 64 characters long.

Loading options

Here you can indicate what should happen when loading a game. First of all you can specify your own loading image. Secondly, you can indicate whether to display a loading progress bar at the bottom of the image. You have three options here. Either no loading bar is displayed, or the default bar is displayed or you can specify two images: the background of the loading bar and the foreground. You can indicate whether the front loading bar must be scaled (default) or clipped while it becomes longer. In the second case, make sure enough the image is large enough to fill the bar. (Note that both images must be specified in this case, not just one.)

It is possible to indicate that the loading image must be transparent. In this case the left bottom pixel of the background image is used as transparent color. Also the alpha translucency can be indicated. A value of 0 means fully translucent. A value of 255 means fully opaque. (Both only work under Windows 2000, XP, or later.)

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

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

Constants

Under this tab you can define global constants that can be used in all scripts and pieces of code, or as values for actions. Each constant has a name and a value. Names should follow the same rules as variables, that is, they must start with a letter or underscore symbol and further consist of letters, digits or underscore symbols. You are though strongly recommended to make all your constants easily distinguishable. A usual convention is to use only capital letters and underscores.

A value of a constant should be a constant expression. That is, it is either a constant number or a string (with quotes around it) or it is an expression. The expression is evaluated before anything else happens in the game. So it for example cannot reference the current room, instances, or scripts. But it can contain the built-in constants and the names of resources.

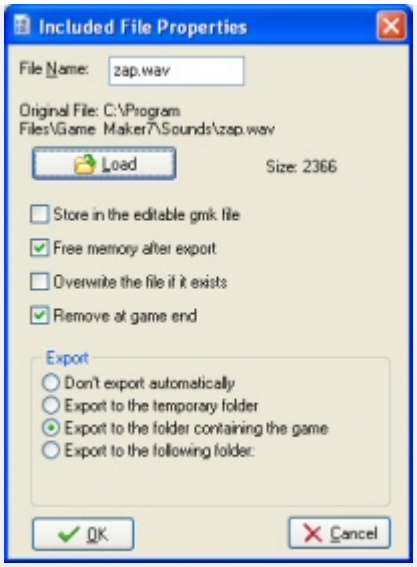
You can append a constant at the end of the list using the button **Append** and delete it using **Delete**. You can also insert a constant above the current selected constant. You can change a constant name or value by clicking on it. There are also buttons to clear all constants, to move them up or down in the list, and to sort them by name.

Including Files in Stand-alone Games

This functionality is only available in the Pro Edition of Game Maker.

As has been indicated before, you can create stand-alone versions of your game. Sometimes your game will use additional files. For example you might want to include video files or text files that are used in the game. In some cases you want to add DLLs or images and sounds that are loaded during the running of the game. You can provide these together with the stand-alone executable but it is sometimes easier to include them in the file. In this way only one file needs to be distributed. Included files will default be exported when the game starts.

You can indicate the files you want to include in the executable in this tab of the game settings. At the top of the form is a list of files to include. Use **Add** to add a new file to the list, **Change** to change a selected file, **Delete** to delete the selected file, and **Clear** to clear the complete list. When adding or changing an entry, the following form will pop up:



Here you can indicate a number of properties. You can press the **Load** button to indicate which file must be included. You can change the **File Name** if you want to save the file under a different name than the original file name.

There are a number of options you can set. When you check **Store in the editable gm6 file** the included file in not only added to the executable but also to the editable version of the game. The advantage is that you can e.g. send the editable version to somebody else and can be sure the file is still there. Also, you can remove the included file if you want or move it elsewhere without problems. The disadvantage is that the editable version of the game becomes larger and takes longer to load.

Checking the option **Free memory after export** means that after the file has been exported (when running the game) it will be removed from memory. If you want to be able to export it again later you should uncheck this option. If a file that is exported already exists it is normally not written. You can change this behavior by checking **Overwrite existing files**. Also, when the game is finished the files are normally not removed (unless they are in the

temporary folder which is completely removed). You can change this by checking **Remove at game end**.

A word of warning is in place here. If you are testing your game, the working directory of the game is the directory where the .gmk file is stored. If your include files are also stored here and you choose to remove them at the end of the game you might loose them alltogether! So better not store these files with the .gmk file but e.g. in a subfolder!

Finally you can indicate to what folder the files are exported. There are four options here. In the default situation the files are exported in the folder where the stand-alone game is stored. This is also the working directory for the game. So the game can just use the file names to access them (no paths are required). This works well if the stand-alone is stored e.g. on the hard disk but will fail if it is stored on a read-only device, e.g. on a CD.

The second possibility is to indicate that the files must be exported into the temporary folder that is created during the running of the game. If you select this option you will need to provide the path to the temporary folder when you use the file name during the game. This path can be obtained using the built-in variable `temp_directory`. Don't forget to add the backslash in this case. So to e.g. play a video file you might type in the following piece of code:

```
{
  splash_show_video(temp_directory+'\movie.avi',true);
}
```

Realize that this temporary folder is removed once the game is finished. So you e.g. cannot store save games or special information in this case. Only choose this option

when you want the game to be run from a CD or when you do not write any files from within your game.

The third option is to indicate the folder to which the file must be exported yourself. If it does not exist it will be created. E.g. you could specify `C:\MyGame`. (Note that you need to provide a full patgh and that you should not include a dach at the end.) This is only allowed when the game does not run in secure mode.

Finally, you can indicate not the export the file automatically. In this case you should use e.g. the functions `export_include_file(fname)` to export it yourself when required.

Error options

Here you can set a number of options that relate to the way errors are reported.

Display error messages When checked, error messages are shown to the player. In the final version of the game you might want to uncheck this option.

Write error messages to file `game_errors.log`
When checked all error messages are written to a file called `game_errors.log` in the game folder.

Abort on all error messages
Normally, certain errors are fatal while others can be ignored. When checking this option all errors are considered fatal and lead to aborting the game. In the final version of the game you distribute you might want to check this option.

Treat uninitialized variables as 0
One common error is to use a variable before a value is assigned to it. Sometimes this is difficult to avoid. When checking this option such uninitialized variables no longer report an error but are treated as value 0. Be careful though. It might mean that you don't spot typing mistakes anymore.

Information about the game

Here you can indicate the author of the game, the version of the game, and some information about the game. Also the last changed date is maintained. This is useful if you are working with many people on a game or make new, updated version. The information is not accessible when the game is running.

Speed considerations

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

First of all, look carefully at the sprites and backgrounds you use. Animated sprites take a lot of memory and drawing lots of sprites takes a lot of time. So make your sprites as small as possible. Remove any invisible area around it (the command **crop** in the sprite editor does that automatically). The same applies to background images. If you have a covering background, make sure you switch off the use of a background color.

If you use full screen mode, make sure the size of the room (or window) is never larger than the screen size. Most graphics cards can efficiently scale images up but they are more slow in scaling images down! Whenever possible, switch off the cursor. It slows down the graphics.

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

Besides the graphics, there are also other aspects that influence the speed. Make sure you have as few instances as possible. In particular, destroy instances once they are no longer required (e.g. when they leave the room). Avoid lots of work in the step event or drawing event of instances. Often things do not need to be checked in each step.

Interpretation of code is reasonably fast, but it is interpreted. Also, some functions and actions take a lot of time; in particular those that have to check all instances (like for example the bounce action).

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

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

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

The Game Maker Language (GML)

Game Maker contains a built-in programming language. This programming language gives you much more flexibility and control than the standard actions. This language we will refer to as GML (the *Game Maker* Language). In this section we describe the language GML and we give an overview of all the (close to 1000) functions and variables available to control all aspects of your game.

Information on GML can be found in the following pages:

Language overview Computing things
Game play
User interaction
Game graphics
Sound and music
Splash screens, highscores and other pop-ups
Resources
Changing resources
Files, registry, and executing programs
Data structures
Creating particles
Multiplayer games
Using DLL's
3D Graphics

GML Language overview

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

In this chapter we will describe the basic structure of programs in GML. When you want to use programs in GML, there are a couple of things you have to be careful about. First of all, for all your resources (sprites, objects, sounds, etc.) you must use names that start with a letter and only consist of letters, digits and the underscore '_' symbol. Otherwise you cannot refer to them from within the program. Make sure all resources have different names. Also be careful not to name resources self, other, global, or all because these have special meaning in the language. Also you should not use any of the keywords, indicated below.

Information on the GML language can be found in the following pages:

A Program Variables
Assignments

Expressions
Extra Variables
Addressing Variables in Other Instances
Arrays
If Statement
Repeat Statement
While Statement
Do Statement
For Statement
Switch Statement
Break Statement
Continue Statement
Exit Statement
Functions
Scripts
With Construction
Comment
Functions and Variables in GML

A program

A program consists of a set of instructions, called statements. A program must start with the symbol '{' and end with the symbol '}'. Between these symbols there are the statements. Statements must be separated with a ';' symbol. So the global structure of every program is:

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

There are a number of different types of statements, which will be discussed below.

Variables

Like any programming language GML contains variables. Variables are memory locations that store information. They have a name so that you can refer to them. A variable in GML can store either a real number or a string. Variables do not need to be declared like in many other languages. There are a large number of built-in variables. Some are general, like `mouse_x` and `mouse_y` that indicate the current mouse position, while all others are local to the object instance for which we execute the program, like x and y that indicate the current position of the instance. A variable has a name that must start with a letter and can contain only letters, numbers, and the underscore symbol '_'. (The maximal length is 64 symbols.) When you use a new variable it is local to the current instance and is not known in programs for other instances (even of the same object). You can though refer to variables in other instances; see below.

Assignments

An assignment stores a value in a variable. An assignment has the form:

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

An expression can be a simple value but can also be more complicated. Rather than assigning a value to a variable, one can also add the value to the current value of the variable using +=. Similar, you can subtract it using -=, multiply it using *=, divide it using /= ,or use bitwise operators using |=, &\, or ^=.

Expressions

Expressions can be real numbers (e.g. 3.4), hexadecimal numbers, starting with a $ sign (e.g. $00FFAA), strings between single or double quotes (e.g. 'hello' or “hello”) or more complicated expressions. For expressions, the following binary operators exist (in order of priority):

- && || ^^: combine Boolean values (&& = and, || = or, ^^ = xor)
- < <= == != > >=: comparisons, result in true (1) or false (0)
- | & ^: bitwise operators (| = bitwise or, & = bitwise and, ^ = bitwise xor)
- << >>: bitwise operators (<< = shift left, > > = shift right)
- + -: addition, subtraction
- * / div mod: multiplication, division, integer division, and modulo

Note that value of `x div y` is the value of `x/y` rounded in the direction of zero to the nearest integer. The `mod` operator returns the remainder obtained by dividing its operands. In other words, `x mod y = x - (x div y) * y`. Also, the following unary operators exist:

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

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

note that, contrary to certain languages, both arguments to a Boolean operation are always computed, even when the first argument already determines the outcome.)

Example

Here is an example with some assignments.

```
{
  x = 23;
  color = $FFAA00;
  str = 'hello world';
  y += 5;
  x *= y;
  x = y << 2;
  x = 23*((2+4) / sin(y));
  str = 'hello' + " world";
  b = (x < 5) && !(x==2 || x==4);
}
```

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;
  }
}
```

Alternatively you can declare the variables as being global. This declaration looks as follows.

```
globalvar <varname1>,<varname2>,<varname3>, ... ;
```

Once this declaration has been executed, the variable is always treated as global without the need to put the word `global` and a dot in front of it. It only needs to be declared once in a piece of code that is executed. After that in all other places the variable is considered as being global.

Sometimes you want variables only within the current piece of code or script. In this way you avoid wasting memory and you are sure there is no naming conflict. It is also faster than using global variables. To achieve this you must declare the variables at the beginning of the piece of code using the keyword var. This declaration looks as follows.

```
var <varname1>,<varname2>,<varname3>, ... ;
```

For example, you can write:

```
{
  var xx,yy;
  xx = x+10;
  yy = y+10;
  instance_create(xx,yy,ball);
}
```

Addressing variables in other instances

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

```
x = 3;
```

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

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

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

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

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

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

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

But what if you want to set the speed of one particular ball, rather than all balls? This is slightly more difficult. Each instance has a unique id. When you put instances in a room in the designer, this instance id is shown when you rest the mouse on the instance. These are numbers larger than or equal to 100000. Such a number you can also use as the left-hand side of the dot. But be careful. The dot will get interpreted as the decimal dot in the number. To avoid this, put brackets around it. So for example, assuming the id of the ball is 100032, you can write:

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

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

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

This creates a ball and sets its speed. Note that we assigned the instance id to a variable and used this variable as indication in front of the dot. This is completely valid. Let us try to make this more precise. A dot is actually an operator. It takes a value as left operand and a variable (address) as right operand, and returns the address of this particular variable in the indicated object or instance. All the object

names, and the special objects indicated above simply represent values and these can be dealt with like any value. For example, the following program is valid:

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

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

Object names, the special objects, and the instance id's can also be used in a number of functions. They are actually treated as constants in the programs.

Arrays

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

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

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

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

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

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

Repeat statement

A repeat statement has the form

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

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

Example The following program creates five balls at random positions.

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

While statement

A while statement has the form

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

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

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

```
{
  while (!place_free(x,y))
  {
    x = random(room_width);
    y = random(room_height);
  }
}
```

Do statement

A do statement has the form

```
do <statement> until(<expression>)
```

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

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

```
{
  do
  {
    x = random(room_width);
    y = random(room_height);
  }
  until (place_free(x,y))
}
```

For statement

A for statement has the form

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

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

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

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

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

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

Switch statement

In a number of situations you want to let your action depend on a particular value. You can do this using a number of if statements but it is easier to use the switch statement. A switch statement has the following form:

```
switch (<expression>)
{
  case <expression1>: <statement1>; ... ; break;
  case <expression2>: <statement2>; ... ; break;
  ...
  default: <statement>; ...
}
```

This works as follows. First the expression is executed. Next it is compared with the results of the different expressions after the case statements. The execution continues after the first case statement with the correct value, until a break statement is encountered. If no case statement has the right value, execution is continued after the default statement. (It is not required to have a default statement.) Note that multiple case statements can be placed for the same statement. Also, the break is not required. If there is no break statement the execution simply continues with the code for the next case statement.

Example The following program takes action based on a key that is pressed.

```
switch (keyboard_key)
{
  case vk_left:
  case vk_numpad4:
    x -= 4; break;
  case vk_right:
```

```
    case vk_numpad6:
      x += 4; break;
  }
```

Break statement

The break statement has the form

```
break
```

If used within a for-loop, a while-loop, a repeat-loop, a switch statement, or a with statement, it end this loop or statement. If used outside such a statement it ends the program (not the game).

Continue statement

The continue statement has the form

```
continue
```

If used within a for-loop, a while-loop, a repeat-loop, or a with statement, it continues with the next value for the loop or with statement.

Exit statement

The exit statement has the form

```
exit
```

It simply ends the execution of this script or piece of code. (It does not end the execution of the game! For this you need the function game_end(); 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>,...)
```

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

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

Note that it is impossible to use a function as the lefthand side of an assignment. For example, you cannot write `instance_nearest(x,y,obj).speed = 0`. Instead you must write `(instance_nearest(x,y,obj)).speed = 0`.

Scripts

When you create a script, you want to access the arguments passed to it (either when using the script action, or when calling the script as a function from a program (or from another, or even the same script). These arguments are stored in the variables argument0, `argument1`, ..., `argument15`. So there can be at most 16 arguments. (Note that when calling the script from an action, only the first 5 arguments can be specified.) You can also use `argument[0]` etc.

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

```
return <expression>
```

Execution of the script ends at the return statement!

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

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

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

With constructions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Comment

You can add comment to your programs. Everything on a line after // is not read. You can also make a multi-line comment by placing the text between / * and */. (Colorcoding might not work correctly here! Press F12 to re-colorcode the text if an error occurs.)

Functions and variables in GML

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

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

Computing things

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

Information on the GML language can be found in the following pages:

Constants Real-valued functions
String handling functions
Dealing with dates and time

Constants

The following mathematical constants exist:

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

Real-valued functions

The following functions exist that deal with real numbers.

random(x) Returns a random real number between 0 and x. The number is always smaller than x.
random_set_seed(seed) Sets the seed (an integer) that is used for the random number generation. Can be used to repeat the same random sequence. (Note though that also some actions and the system itself uses random numbers.
random_get_seed() Returns the current seed.
randomize() Sets the seed to a random number.
choose(val1,val2,val3,...) Returns one of the arguments choosen randomly. The function can have up to 16 arguments.
abs(x) Returns the absolute value of x.
sign(x) Returns the sign of x (-1, 0 or 1).
round(x) Returns x rounded to the nearest integer.
floor(x) Returns the floor of x, that is, x rounded down to an integer.
ceil(x) Returns the ceiling of x, that is, x rounded up to an integer.
frac(x) Returns the fractional part of x, that is, the part behind the decimal dot.
sqrt(x) Returns the square root of x. x must be non-negative.
sqr(x) Returns x*x.
power(x,n) Returns x to the power n.
exp(x) Returns e to the power x.
ln(x) Returns the natural logarithm of x.
log2(x) Returns the log base 2 of x.

log10(x) Returns the log base 10 of x.
logn(n,x) Returns the log base n of x.
sin(x) Returns the sine of x (x in radians).
cos(x) Returns the cosine of x (x in radians).
tan(x) Returns the tangent of x (x in radians).
arcsin(x) Returns the inverse sine of x.
arccos(x) Returns the inverse cosine of x.
arctan(x) Returns the inverse tangent of x.
arctan2(y,x) Calculates arctan(Y/X), and returns an angle in the correct quadrant.
degtorad(x) Converts degrees to radians.
radtodeg(x) Converts radians to degrees.
min(val1,val2,val3,...) Returns the minimum of the values. The function can have up to 16 arguments. They must either be all real or all strings.
max(val1,val2,val3,...) Returns the maximum of the values. The function can have up to 16 arguments. They must either be all real or all strings.
mean(val1,val2,val3,...) Returns the average of the values. The function can have up to 16 arguments. They must all be real values.
median(val1,val2,val3,...) Returns the median of the values, that is, the middle value. (When the number of arguments is even, the smaller of the two middle values is returned.) The function can have up to 16 arguments. They must all be real values.
point_distance(x1,y1,x2,y2) Returns the distance between point (x1,y1) and point (x2,y2).
point_direction(x1,y1,x2,y2) Returns the direction from point (x1,y1) toward point (x2,y2) in degrees.
lengthdir_x(len,dir) Returns the horizontal x-component of the vector determined by the indicated length and direction.
lengthdir_y(len,dir) Returns the vertical y-component of

the vector determined by the indicated length and direction.

`is_real(x)` Returns whether x is a real value (as opposed to a string).

`is_string(x)` Returns whether x is a string (as opposed to a real value).

String handling functions

The following functions deal with characters and string.

`chr(val)` Returns a string containing the character with asci code val.
`ord(str)` Returns the asci code of the first character in str.
`real(str)` Turns str into a real number. str can contain a minus sign, a decimal dot and even an exponential part.
`string(val)` Turns the real value into a string using a standard format (no decimal places when it is an integer, and two decimal places otherwise).
`string_format(val,tot,dec)` Turns val into a string using your own format: tot indicates the total number of places and dec indicates the number of decimal places.
`string_length(str)` Returns the number of characters in the string.
`string_pos(substr,str)` Returns the position of substr in str (0=no occurrence).
`string_copy(str,index,count)` Returns a substring of str, starting at position index, and of length count.
`string_char_at(str,index)` Returns the character in str at position index.
`string_delete(str,index,count)` Returns a copy of str with the part removed that starts at position index and has length count.
`string_insert(substr,str,index)` Returns a copy of str with substr added at position index.
`string_replace(str,substr,newstr)` Returns a copy of str with the first occurrence of substr replaced by newstr.
`string_replace_all(str,substr,newstr)` Returns a copy of

str with all occurrences of substr replaced by newstr.
`string_count(substr,str)` Returns the number of occurrences of substr in str.
`string_lower(str)` Returns a lowercase copy of str.
`string_upper(str)` Returns an uppercase copy of str.
`string_repeat(str,count)` Returns a string consisting of count copies of str.
`string_letters(str)` Returns a string that only contains the letters in str.
`string_digits(str)` Returns a string that only contains the digits in str.
`string_lettersdigits(str)` Returns a string that contains the letters and digits in str.

The following functions deal with the clipboard for storing text.

`clipboard_has_text()` Returns whether there is any text on the clipboard.
`clipboard_get_text()` Returns the current text on the clipboard.
`clipboard_set_text(str)` Sets the string str on the clipboard.

Dealing with dates and time

In *Game Maker* there are a number of functions to deal with dates and time. A date-time combination is stored in a real number. The integral part of a date-time value is the number of days that have passed since 12/30/1899. The fractional part of the date-time value is fraction of a 24 hour day that has elapsed. The following functions exist:

`date_current_datetime()` Returns the date-time value that corresponds to the current moment.

`date_current_date()` Returns the date-time value that corresponds to the current date only (ignoring the time).

`date_current_time()` Returns the date-time value that corresponds to the current time only (ignoring the date).

`date_create_datetime(year,month,day,hour,minute,second)` Creates a date-time value corresponding to the indicated date and time.

`date_create_date(year,month,day)` Creates a date-time value corresponding to the indicated date.

`date_create_time(hour,minute,second)` Creates a date-time value corresponding to the indicated time.

`date_valid_datetime(year,month,day,hour,minute,second)` Returns whether the indicated date and time are valid.

`date_valid_date(year,month,day)` Returns whether the indicated date is valid.

`date_valid_time(hour,minute,second)` Returns whether the indicated time is valid.

`date_inc_year(date,amount)` Returns a new date that is amount years after the indicated date. amount must be an integer number.

`date_inc_month(date,amount)` Returns a new date that is amount months after the indicated date. amount must be an integer number.
`date_inc_week(date,amount)` Returns a new date that is amount weeks after the indicated date. amount must be an integer number.
`date_inc_day(date,amount)` Returns a new date that is amount days after the indicated date. amount must be an integer number.
`date_inc_hour(date,amount)` Returns a new date that is amount hours after the indicated date. amount must be an integer number.
`date_inc_minute(date,amount)` Returns a new date that is amount minutes after the indicated date. amount must be an integer number.
`date_inc_second(date,amount)` Returns a new date that is amount seconds after the indicated date. amount must be an integer number.
`date_get_year(date)` Returns the year corresponding to the date.
`date_get_month(date)` Returns the month corresponding to the date.
`date_get_week(date)` Returns the week of the year corresponding to the date.
`date_get_day(date)` Returns the day of the month corresponding to the date.
`date_get_hour(date)` Returns the hour corresponding to the date.
`date_get_minute(date)` Returns the minute corresponding to the date.
`date_get_second(date)` Returns the second corresponding to the date.
`date_get_weekday(date)` Returns the day of the week corresponding to the date.

`date_get_day_of_year(date)` Returns the day of the year corresponding to the date.
`date_get_hour_of_year(date)` Returns the hour of the year corresponding to the date.
`date_get_minute_of_year(date)` Returns the minute of the year corresponding to the date.
`date_get_second_of_year(date)` Returns the second of the year corresponding to the date.
`date_year_span(date1,date2)` Returns the number of years between the two dates. It reports incomplete years as a fraction.
`date_month_span(date1,date2)` Returns the number of months between the two dates. It reports incomplete months as a fraction.
`date_week_span(date1,date2)` Returns the number of weeks between the two dates. It reports incomplete weeks as a fraction.
`date_day_span(date1,date2)` Returns the number of days between the two dates. It reports incomplete days as a fraction.
`date_hour_span(date1,date2)` Returns the number of hours between the two dates. It reports incomplete hours as a fraction.
`date_minute_span(date1,date2)` Returns the number of minutes between the two dates. It reports incomplete minutes as a fraction.
`date_second_span(date1,date2)` Returns the number of seconds between the two dates. It reports incomplete seconds as a fraction.
`date_compare_datetime(date1,date2)` Compares the two date-time values. Returns -1, 0, or 1 depending on whether the first is smaller, equal, or larger than the second value.
`date_compare_date(date1,date2)` Compares the two date-

time values only taking the date part into account. Returns -1, 0, or 1 depending on whether the first is smaller, equal, or larger than the second value.
date_compare_time(date1,date2) Compares the two date-time values only taking the time part into account. Returns -1, 0, or 1 depending on whether the first is smaller, equal, or larger than the second value.
date_date_of(date) Returns the date part of the indicated date-time value, setting the time part to 0.
date_time_of(date) Returns the time part of the indicated date-time value, setting the date part to 0.
date_datetime_string(date) Returns a string indicating the given date and time in the default format for the system.
date_date_string(date) Returns a string indicating the given date in the default format for the system.
date_time_string(date) Returns a string indicating the given time in the default format for the system.
date_days_in_month(date) Returns the number of days in the month indicated by the date-time value.
date_days_in_year(date) Returns the number of days in the year indicated by the date-time value.
date_leap_year(date) Returns whether the year indicated by the date-time value is a leap year.
date_is_today(date) Returns whether the indicated date-time value is on today.

Game play

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

Information on game play can be found in the following pages:

Moving Around Paths
Motion Planning
Collision Detection
Instances
Deactivating Instances
Timing
Rooms
Score
Generating Events
Miscellaneous Variables and Functions

Moving around

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

If the object moves with constant speed and direction, there is an easier way to do this. Each object instance has a horizontal speed (`hspeed`) and a vertical speed (`vspeed`). Both are indicated in pixels per step. A positive horizontal speed means a motion to the right, a negative horizontal speed mean a motion to the left. Positive vertical speed is downwards and negative vertical speed is upwards. So you have to set these variables only once (for example in the creating event) to give the object instance a constant motion.

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

To be complete, each instance has the following variables and functions dealing with its position and motion:

x Its x-position.
y Its y-position.
xprevious Its previous x-position.
yprevious Its previous y-position.
xstart Its starting x-position in the room.
ystart Its starting y-position in the room.
hspeed Horizontal component of the speed.
vspeed Vertical component of the speed.
direction Its current direction (0-360, counter-clockwise, 0 = to the right).
speed Its current speed (pixels per step).
friction Current friction (pixels per step).
gravity Current amount of gravity (pixels per step).
gravity_direction Direction of gravity (270 is downwards).
motion_set(dir,speed) Sets the motion with the given speed in direction dir.
motion_add(dir,speed) Adds the motion to the current motion (as a vector addition).

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

place_free(x,y) Returns whether the instance placed at position(x,y) is collision-free. This is typically used as a check before actually moving to the new position.
place_empty(x,y) Returns whether the instance placed at position (x,y) meets nobody. So this function takes also non-solid instances into account.
place_meeting(x,y,obj) Returns whether the instance placed at position (x,y) meets obj. obj can be an object in which case the function returns true is some instance of that object is met. It can also be an instance id, the special word all meaning an instance of any object, or

the special word other.
place_snapped(hsnap,vsnap) Returns whether the instance is aligned with the snapping values.
move_random(hsnap,vsnap) Moves the instance to a free random, snapped position, like the corresponding action.
move_snap(hsnap,vsnap) Snaps the instance, like the corresponding action.
move_wrap(hor,vert,margin) Wraps the instance when it has left the room to the other side. `hor` indicates whether to wrap horizontaly and `vert` indicates whether to wrap vertically. `margin` indicates how far the origin of the instance must be outside the room before the wrap happens. So it is a margin around the room. You typically use this function in the Outside event.
move_towards_point(x,y,sp) Moves the instances with speed sp toward position (x,y).
move_bounce_solid(adv) Bounces against solid instances, like the corresponding action. adv indicates whether to use advance bounce, that also takes slanted walls into account.
move_bounce_all(adv) Bounces against all instances, instead of just the solid ones.
move_contact_solid(dir,maxdist) Moves the instance in the direction until a contact position with a solid object is reached. If there is no collision at the current position, the instance is placed just before a collision occurs. If there already is a collision the instance is not moved. You can specify the maximal distance to move (use a negative number for an arbitrary distance).
move_contact_all(dir,maxdist) Same as the previous function but this time you stop at a contact with any object, not just solid objects.
move_outside_solid(dir,maxdist) Moves the instance in the

direction until it no longer lies within a solid object. If there is no collision at the current position the instance is not moved. You can specify the maximal distance to move (use a negative number for an arbitrary distance).
move_outside_all(dir,maxdist) Same as the previous function but this time you move until outside any object, not just solid objects.
distance_to_point(x,y) Returns the distance of the bounding box of the current instance to (x,y).
distance_to_object(obj) Returns the distance of the instance to the nearest instance of object obj.
position_empty(x,y) Returns whether there is nothing at position (x,y).
position_meeting(x,y,obj) Returns whether at position (x,y) there is an instance obj. obj can be an object, an instance id, or the keywords `self`, `other`, or `all`.

Paths

In *Game Maker* you can define paths and order instances to follow such paths. Although you can use actions for this, there are functions and variables that give you more flexibility:

path_start(path,speed,endaction,absolute) Starts a path for the current instance. The `path` is the name of the path you want to start. The `speed` is the speed with which the path must be followed. A negative speed means that the instance moves backwards along the path. The `endaction` indicates what should happen when the end of the path is reached. The following values can be used:

0 : stop the path
1: continue from the start position (if the path is not closed we jump to the start position
2: continue from the current position
3: reverse the path, that is change the sign of the speed

The argument `absolute` should be true or false. When true the absolute coordinates of the path are used. When false the path is relative to the current position of the instance. To be more precise, if the speed is positive, the start point of the path will be placed on the current position and the path is followed from there. When the speed is negative the end point of the path will be placed on the current position and the path is followed backwards from there.
path_end() Ends the following of a path for the current instance.

path_index* Index of the current path the instance follows. You cannot change this directly but must use the function above.
path_position Position in the current path. 0 is the beginning of the path. 1 is the end of the path. The value must lie between 0 and 1.
path_positionprevious Previous position in the current path. This can be used e.g. in collision events to set the position on the path back to the previous position.
path_speed Speed (in pixels per step) with which the path must be followed. Use a negative speed to move backwards.
path_orientation Orientation (counter-clockwise) into which the path is performed. 0 is the normal orientation of the path.
path_scale Scale of the path. Increase to make the path larger. 1 is the default value.
path_endaction The action that must be performed at the end of the path. You can use the values indicated above.

Motion planning

Motion planning helps you to move certain instances from a given location to a different location while avoiding collisions with certain other instances (e.g. walls). Motion planning is a difficult problem. It is impossible to give general functions that will work properly in all situations. Also, computing collision free motions is a time-consuming operation. So you have be careful how and when you apply it. Please keep these remarks in mind when you use any of the following functions.

Different forms of motion planning are provided by *Game Maker*. The simplest form lets an instance take a step towards a particular goal position, trying to go straight if possible but taking a different direction if required. These functions should be used in the step event of an instance. They correspond to the motion planning actions that are also available:

`mp_linear_step(x,y,stepsize,checkall)` This function lets the instance take a step straight towards the indicated position (x,y). The size of the step is indicated by the `stepsize`. If the instance is already at the position it will not move any further. If `checkall` is true the instance will stop when it hits an instance of any object. If it is false it only stops when hitting a solid instance. Note that this function does not try to make detours if it meets an obstacle. It simply fails in that case. The function returns whether or not the goal position was reached.
`mp_linear_step_object(x,y,stepsize,obj)` Same as the function above but this time only instances of `obj` are considered as obstacles. `obj` can be an object or an

instance id.

mp_potential_step(x,y,stepsize,checkall) Like the previous function, this function lets the instance take a step towards a particular position. But in this case it tries to avoid obstacles. When the instance would run into a solid instance (or any instance when checkall is true) it will change the direction of motion to try to avoid the instance and move around it. The approach is not guaranteed to work but in most easy cases it will effectively move the instance towards the goal. The function returns whether or not the goal was reached.

mp_potential_step_object(x,y,stepsize,obj) Same as the function above but this time only instances of `obj` are considered as obstacles. `obj` can be an object or an instance id.

mp_potential_settings(maxrot,rotstep,ahead,onspot) The previous function does its work using a number of parameters that can be changed using this function. Globally the method works as follows. It first tries to move straight towards the goal. It looks a number of steps ahead which can be set with the parameter `ahead` (default 3). Reducing this value means that the instance will start changing direction later. Increasing it means it will start changing direction earlier. If this check leads to a collision it starts looking at directions more to the left and to the right of the best direction. It does this in steps of size `rotstep` (default 10). Reducing this gives the instance more movement possibilities but will be slower. The parameter `maxrot` is a bit more difficult to explain. The instance has a current direction. `maxrot` (default 30) indicates how much it is allowed to change its current direction in a step. So even if it can move e.g. straight to the goal it will only do so if it does not violate this maximal change of direction. If you make `maxrot` large the instance can change a lot in each step.

This will make it easier to find a short path but the path will be uglier. If you make the value smaller the path will be smoother but it might take longer detours (and sometimes even fail to find the goal). When no step can be made the behavior depends on the value of the parameter `onspot`. If `onspot` is true (the default value), the instance will rotate on its spot by the amount indicated with `maxrot`. If it is false it will not move at all. Setting it to false is useful for e.g. cars but reduces the chance of finding a path.

Please note that the potential approach uses only local information. So it will only find a path if this local information is enough to determine the right direction of motion. For example, it will fail to find a path out of a maze (most of the time).

The second kind of functions computes a collision-free path for the instance. Once this path has been computed you can assign it to the instance to move towards the goal. The computation of the path will take some time but after that the execution of the path will be fast. Of course this is only valid if the situation has not changed in the meantime. For example, if obstacles change you possibly will need to recompute the path. Again notice that these functions might fail. ***These functions are only available in the Pro Edition of Game Maker.***

The first two functions use the linear motion and potential field approach that were also used for the step functions.

`mp_linear_path(path,xg,yg,stepsize,checkall)` This function computes a straight-line path for the instance from its current position to the position (xg,yg) using the indicated step size. It uses steps as in the function `mp_linear_step()`. The indicated path must already exist

and will be overwritten by the new path. (See a later chapter on how to create and destroy paths.) The function will return whether a path was found. The function will stop and report failure if no straight path exists between start and goal. If it fails a path is still created that runs till the position where the instance was blocked.

mp_linear_path_object(path,xg,yg,stepsize,obj) Same as the function above but this time only instances of `obj` are considered as obstacles. `obj` can be an object or an instance id.

mp_potential_path(path,xg,yg,stepsize,factor,checkall) This function computes a path for the instance from its current position and orientation to the position (xg,yg) using the indicated step size trying to avoid collision with obstacles. It uses potential field steps, like in the function `mp_potential_step()` and also the parameters that can be set with `mp_potential_settings()`. The indicated path must already exist and will be overwritten by the new path. (See a later chapter on how to create and destroy paths.) The function will return whether a path was found. To avoid the function continuing to compute forever you need to provide a length factor larger than 1. The function will stop and report failure if it cannot find a path shorter than this factor times the distance between start and goal. A factor of 4 is normally good enough but if you expect long detours you might make it longer. If it fails a path is still created that runs in the direction of the goal but it will not reach it.

mp_potential_path_object(path,xg,yg,stepsize,factor,obj) Same as the function above but this time only instances of `obj` are considered as obstacles. `obj` can be an object or an instance id.

The other functions use a much more complex mechanism using a grid-based approach (sometimes called an A* algorithm). It will be more successful in finding paths (although it still might fail) and will find shorter paths but it required more work on your side. The global idea is as follows. First of all we put a grid on (the relevant part of) the room. You can choose to use a fine grid (which will be slower) or a coarse grid. Next, for all relevant objects we determine the grid cells they overlap (either using bounding boxes or precise checking) and mark these cells as being forbidden. So a cell will be marked totally forbidden, even if it only partially overlaps with an obstacle. Finally we specify a start and a goal position (which must lie in free cells) and the function computes the shortest path (actually close to the shortest) between these. The path will run between centers of free cells. So if the cells are large enough so that the instance placed at its center will lie completely inside it this will be successful. This path you can now give to an instance to follow.

The grid-based approach is very powerful (and is used in many professional games) but it requires that you do some careful thinking. You must determine which area and cell size are good enough for solving the game. Also you must determine which objects must be avoided and whether precise checking is important. All these parameters strongly influence the efficiency of the approach.

In particular the size of the cells is crucial. Remember that the cells must be large enough so that the moving object placed with its origin on the center of a cell must lie completely inside the cell. (Be careful about the position of the origin of the object. Also realize that you can shift the path if the origin of the object is not in its center!) On the other hand, the smaller the cells the more possible paths exist. If you make cells too large openings between

obstacles may get closed because all cells intersect an obstacle.

The actual functions for the grid-based approach are as follows:

mp_grid_create(left,top,hcells,vcells,cellwidth,cellheight) This function creates the grid. It returns an index that must be used in all other calls. You can create and maintain multiple grid structures at the same moment. left and top indicate the position of the top-left corner of the grid. hcells and vcells indicate the number of horizontal and vertical cells. Finally cellwidth and cellheight indicate the size of the cells.

mp_grid_destroy(id) Destroys the indicated grid structure and frees its memory. Don't forget to call this if you don't need the structure anymore.

mp_grid_clear_all(id) Mark all cells in the grid to be free.

mp_grid_clear_cell(id,h,v) Clears the indicated cell. Cell 0,0 is the top left cell.

mp_grid_clear_rectangle(id,left,top,right,bottom) Clears all cells that intersect the indicated rectangle (in room coordinates).

mp_grid_add_cell(id,h,v) Marks the indicated cell as being forbidden. Cell 0,0 is the top left cell.

mp_grid_add_rectangle(id,left,top,right,bottom) Marks all cells that intersect the indicated rectangle as being forbidden.

mp_grid_add_instances(id,obj,prec) Marks all cells that intersect an instance of the indicated object as being forbidden. You can also use an individual instance by making obj the id of the instance. Also you can use the keyword **all** to indicate all instances of all objects. prec indicates whether precise collision checking must be used (will only work if precise checking is enabled for

the sprite used by the instance).

mp_grid_path(id,path,xstart,ystart,xgoal,ygoal,allowdiag) Computes a path through the grid. path must indicate an existing path that will be replaced by the computer path. xstart and ystart indicate the start of the path and xgoal and ygoal the goal. allowdiag indicates whether diagonal moves are allowed instead of just horizontal or vertical. The function returns whether it succeeded in finding a path. (Note that the path is independent of the current instance; It is a path through the g rid, not a path for a specific instance.)

mp_grid_draw(id) This function draws the grid with green cells being free and red cells being forbidden. This function is slow and only provided as a debug tool.

Collision checking

When planning motions or deciding on certain actions, it is often important to see whether there are collisions with other objects at certain places. The following routines can be used for this. All these have three arguments in common: The argument `obj` can be an object, the keyword **`all`**, or the id of an instance. The argument `prec` indicates whether the check should be precise or only based on the bounding box of the instance. Precise checking is only done when the sprite for the instance has the precise collision checking set. The argument `notme` can be set to true to indicate that the calling instance should not be checked. All these functions return either the id of one of the instances that collide, or they return a negative value when there is no collision.

`collision_point(x,y,obj,prec,notme)` This function tests whether at point (x,y) there is a collision with entities of object obj.

`collision_rectangle(x1,y1,x2,y2,obj,prec,notme)` This function tests whether there is a collision between the (filled) rectangle with the indicated opposite corners and entities of object obj. For example, you can use this to test whether an area is free of obstacles.

`collision_circle(xc,yc,radius,obj,prec,notme)` This function tests whether there is a collision between the (filled) circle centered at position (xc,yc) with the given radius and entities of object obj. For example, you can use this to test whether there is an object close to a particular location.

`collision_ellipse(x1,y1,x2,y2,obj,prec,notme)` This function tests whether there is a collision between the (filled)

ellipse with the indicated opposite corners and entities of object obj.

`collision_line(x1,y1,x2,y2,obj,prec,notme)` This function tests whether there is a collision between the line segment from (x1,y1) to (x2,y2) and entities of object obj. This is a powerful function. You can e.g. use it to test whether an instance can see another instance by checking whether the line segment between them intersects a wall.

Instances

In the game, the basic units are the instances of the different objects. During game play you can change a number of aspects of these instances. Also you can create new instances and destroy instances. Besides the movement related variables discussed above and the drawing related variables discussed below, each instance has the following variables:

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

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

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

Note that the assignment of the instances to the instance id's changes every step so you cannot use values from previous steps. Let me give an example. Assume each unit in your game has a particular power and you want to locate the strongest one, you could use the following code:

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

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

instance_find(obj,n) Returns the id of the (n+1)'th instance of type obj. obj can be an object or the keyword all. If it does not exist, the special object noone is returned. Note that the assignment of the instances to the instance id's changes every step so you cannot use values from previous steps.
instance_exists(obj) Returns whether an instance of type obj exists. obj can be an object, an instance id, or the

keyword all.

instance_number(obj) Returns the number of instances of type obj. obj can be an object or the keyword all.

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

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

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

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

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

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

instance_copy(performevent) Creates a copy of the current instance. The argument indicates whether the creation event must be executed for the copy. The function returns the id of the new copy.

instance_destroy() Destroys the current instance.

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

position_destroy(x,y) Destroys all instances whose sprite contains position (x,y).

`position_change(x,y,obj,perf)` Changes all instances at (x,y) into obj. perf indicates whether to perform the destroy and creation events.

Deactivating instances

When you create large room, for example in platform games, with a small view, many instances lie outside the view. Such instances though are still active and will execute their events. Also when performing collision checks these instances are taken into account. This can cost a lot of time, which is often not necessary. (For example, often it is not important whether instances outside the view move.) To solve this problem *Game Maker* contains some functions to deactivate and activate instances. Before using them you must though clearly understand how they work.

When you deactivate instances they are in some sense removed from the game. They are not visible anymore nor are any events executed for them. So for all actions and functions they don't exist anymore. This saves a lot of time but you have to be careful. For example, when you delete all instances of a particular type, deactivated instances are not deleted (because they don't exist). So don't think that a key a player picks up can unlock a deactivated door.

The most crucial mistake you can make is to deactivate the instance that is responsible for the activation. To avoid this some of the routines below allow you to insist that the calling instance should not be deactivated itself.

Here are the available routines:

`instance_deactivate_all(notme)` Deactivates all instances in the room. If notme is true the calling instance is not deactivated (which is normally what you want).
`instance_deactivate_object(obj)` Deactivates all instances in the room of the given object. You can also use all to

indicate that all instances must be deactivated or the id of an instance to deactivate an individual instance.

instance_deactivate_region(left,top,width,height,inside,notme) Deactivates all instances in the indicated region (that is, those whose bounding box lies partially inside the region). If inside is false the instances completely outside the region are deactivated. If notme is true the calling instance is not deactivated (which is normally what you want).

instance_activate_all() Activates all instances in the room.

instance_activate_object(obj) Activates all instances in the room of the given object. You can also use all to indicate that all instances must be activated or the id of an instance to activate an individual instance.

instance_activate_region(left,top,width,height,inside) Activates all instances in the indicated region. If inside is false the instances completely outside the region are activated.

For example, to deactivate all instances outside the view and activate the ones inside the view, you could place the following code in the step event of the moving character:

```
{
  instance_activate_all();

instance_deactivate_region(view_xview[0],view_yview[0]
,

view_wview[0],view_hview[0],false,true);
}
```

In practice you might want to use a region slightly larger than the view.

Timing

Good games require careful timing of things happening. Fortunately *Game Maker* does most of the timing for you. It makes sure things happen at a constant rate. This rate is defined when defining the rooms. But you can change it using the global variable room_speed. So for example, you can slowly increase the speed of the game, making it more difficult, by adding a very small amount (like 0.001) to room_speed in every step. If your machine is slow the game speed might not be achieved. This can be checked using the variable fps that constantly monitors the actual number of frames per second. Finally, for some advanced timing you can use the variable current_time that gives the number of milliseconds since the computer was started. Here is the total collection of variables available (only the first one can be changed):

room_speed Speed of the game in the current room (in steps per second).
fps* Number of frames that are actually drawn per second.
current_time* Number of milliseconds that have passed since the system was started.
current_year* The current year.
current_month* The current month.
current_day* The current day.
current_weekday* The current day of the week (1=sunday, ..., 7=saturday).
current_hour* The current hour.
current_minute* The current minute.
current_second* The current second.

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

`sleep(numb)` Sleeps numb milliseconds.

As you should know, every instance has 12 different alarm clocks that you can set. To change the values (or get the values) of the different alarm clocks use the following variable:

`alarm[0..11]` Value of the indicated alarm clock. (Note that alarm clocks only get updated when the alarm event for the object contains actions!)

We have seen that for complex timing issues you can use the time line resource. Each instance can have a time line resource associated with it. The following variables deal with this:

`timeline_index` Index of the time line associated with the instance. You can set this to a particular time line to use that one. Set it to -1 to stop using a time line for the instance.

`timeline_position` Current position within the time line. You can change this to skip certain parts or to repeat parts.

`timeline_speed` Normally, in each step the position in the time line is increased by 1. You can change this amount by setting this variable to a different value. You can use real numbers like 0.5. If the value is larger than one, several moments can happen within the same time step. They will all be performed in the correct order, so no actions will be skipped.

Rooms

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

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

The following variables and functions exist that deal with rooms.

room Index of the current room; can be changed to go to a different room, but you had better use the routines below.
room_first* Index of the first room in the game.
room_last* Index of the last room in the game.
room_goto(numb) Goto the room with index numb.
room_goto_previous() Go to the previous room.
room_goto_next() Go to the next room.
room_restart() Restart the current room.
room_previous(numb) Return the index of the room before numb (-1 = none) but don't go there.

room_next(numb) Return the index of the room after numb (-1 = none).
game_end() End the game.
game_restart() Restart the game.

When calling one of the above functions to change the room or end or restart the game, please realize that this change does actually not occur at that precise moment. It only happens after the current action is fully executed. So the rest of the script will still be executed, and the same applies to possible calling scripts.

Rooms have a number of additional properties:

room_width* Width of the room in pixels.
room_height* Height of the room in pixels.
room_caption Caption string for the room that is displayed in the caption of the window.
room_persistent Whether the current room is persistent.

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

game_save(string) Saves the game to the file with name string.
game_load(string) Loads the game from the file with name string.

Please realize that only the basic game data is being saved. If for example you play a particular piece of music, the precise position in the music is not saved. Also changed resources are not saved. Other things that are not saved

are the contents of data structures, particles, and multiplayer settings.

Transitions

When you move from one room to another you can select a transition. To set the transition to the next frame you must set the variable called `transition_kind`. If you assign a value larger than 0 to it the corresponding transition is used for the next room transition. It only affect the next transition. After this the value is reset to 0, which indicates no transition.

`transition_kind` Indicates the next room transition. You can use the following builtin values

0 = no effect
1 = Create from left
2 = Create from right
3 = Create from top
4 = Create from bottom
5 = Create from center
6 = Shift from left
7 = Shift from right
8 = Shift from top
9 = Shift from bottom
10 = Interlaced from left
11 = Interlaced from right
12 = Interlaced from top
13 = Interlaced from bottom
14 = Push from left
15 = Push from right
16 = Push from top
17 = Push from bottom
18 = Rotate to the left
19 = Rotate to the right

20 = Blend the rooms
21 = Fade out and in

transition_steps Indicates the number of steps in the transition. The more steps, the longer the transition takes. Default is 80.

transition_define(kind,name) You can actually create your own transitions. To this end you must define a script (possibly in an extension package) to do the transition. With this function you can then add the transition to the system. `kind` is the index of the transition (either a new one or an existing transitions). `name` is the name of the script. Note that the name of the script is a string! So there must be quotes around it. Note that this is really advanced stuff. The script must take five arguments: a surface with the image of the previous room, a surface with the image of the next room, the width of the surfaces, the height of the surfaces, and the fraction of the transition (between 0 and 1). It must then draw the image using the two surfaces.

transition_exists(kind) This function returns whether a transition of the indicated kind exists.

Please note that transitions do not work when using using 3d graphics. Also, room transitions in general do not work correctly when the sizes of the rooms (or to be more precise of the region on the screen) are not the same.

Score

Other important aspects of many games are the score, the health, and the number of lives. *Game Maker* keeps track of the `score` in a global variable `score` and the number of `lives` in a global variable lives. You can change the score by simply changing the value of this variable. The same applies to health and lives. If lives is larger than 0 and becomes smaller than or equal to 0 the no-more-lives event is performed for all instances. If you don't want to show the score and lives in the caption, set the variable `show_score`, `etc.`, to false. Also you can change the caption. For more complicated games best display the score yourself.

`score` The current score.
`lives` Number of lives.
`health` The current health (0-100).
`show_score` Whether to show the score in the window caption.
`show_lives` Whether to show the number of lives in the window caption.
`show_health` Whether to show the health in the window caption.
`caption_score` The caption used for the score.
`caption_lives` The caption used for the number of lives.
`caption_health` The caption used for the health.

Generating events

As you know, *Game Maker* is completely event driven. All actions happen as the result of events. There are a number of different events. Creation and destroy events happen when an instance is created or destroyed. In each step, the system first handles the alarm events. Next it handles keyboard and mouse events and next the step event. After this the instances are set to their new positions after which the collision event is handled. Finally the draw event is used to draw the instances (note that when there are multiple views the draw event is called multiple times in each step). You can also apply an event to the current instance from within a piece of code. The following functions exist:

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

```
ev_create ev_destroy
ev_step
ev_alarm
ev_keyboard
ev_mouse
ev_collision
ev_other
ev_draw
ev_keypress
ev_keyrelease
```

When there are multiple events of the given type, numb can be used to specify the precise event. For the alarm event numb can range from 0 to 11. For the keyboard

event you have to use the keycode for the key. For mouse events you can use the following constants:

```
ev_left_button
ev_right_button
ev_middle_button
ev_no_button
ev_left_press
ev_right_press
ev_middle_press
ev_left_release
ev_right_release
ev_middle_release
ev_mouse_enter
ev_mouse_leave
ev_mouse_wheel_up
ev_mouse_wheel_down
ev_global_left_button
ev_global_right_button
ev_global_middle_button
ev_global_left_press
ev_global_right_press
ev_global_middle_press
ev_global_left_release
ev_global_right_release
ev_global_middle_release
ev_joystick1_left
ev_joystick1_right
ev_joystick1_up
ev_joystick1_down
ev_joystick1_button1
ev_joystick1_button2
ev_joystick1_button3
```

```
ev_joystick1_button4
ev_joystick1_button5
ev_joystick1_button6
ev_joystick1_button7
ev_joystick1_button8
ev_joystick2_left
ev_joystick2_right
ev_joystick2_up
ev_joystick2_down
ev_joystick2_button1
ev_joystick2_button2
ev_joystick2_button3
ev_joystick2_button4
ev_joystick2_button5
ev_joystick2_button6
ev_joystick2_button7
ev_joystick2_button8
```

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

```
ev_outside
ev_boundary
ev_game_start
ev_game_end
ev_room_start
ev_room_end
ev_no_more_lives
ev_no_more_health
ev_animation_end
ev_end_of_path
ev_close_button
```

```
ev_user0
ev_user1
ev_user2
ev_user3
ev_user4
ev_user5
ev_user6
ev_user7
ev_user8
ev_user9
ev_user10
ev_user11
ev_user12
ev_user13
ev_user14
ev_user15
```

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

```
ev_step_normal
ev_step_begin
ev_step_end
```

event_perform_object(obj,type,numb) This functions works the same as the function above except that this time you can specify events in another object. Note that the actions in these events are applied to the current instance, not to instances of the given object.. **event_user(numb)** In the other events you can also define 16 user events. These are only performed if you call this function. numb must lie in the range 0 to 15. **event_inherited()** Performs the inherited event. This only works if the instance has a parent object.

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

`event_type*` Type of the current event begin executed.
`event_number*` Number of the current event begin executed.
`event_object*` The object index for which the current event is being executed.
`event_action*` The index of the action that is currently being executed (0 is the first in the event, etc.).

Miscellaneous variables and functions

Here are some variables and functions that deal with errors.

error_occurred Indicates whether an error has occurred
error_last String indicating the last error message
show_debug_message(str) Shows the string in debug mode

The following functions exist that allow you to check whether certain variables exist and with which you can set variables and get their values. In all these functions the variable name is passed as a string!

variable_global_exists(name) Returns whether a global variable with the given name (a string) exists.
variable_local_exists(name) Returns whether a local variable with the given name (a string) exists for the current instance.
variable_global_get(name) Returns the value of the global variable with the given name (a string).
variable_global_array_get(name,ind) Returns the value of index ind of the global array variable with the given name (a string).
variable_global_array2_get(name,ind1,ind2) Returns the value of index ind1,ind2 of the global 2-dimensional array variable with the given name (a string).
variable_local_get(name) Returns the value of the local variable with the given name (a string).
variable_local_array_get(name,ind) Returns the value of index ind of the local array variable with the given name (a string).
variable_local_array2_get(name,ind1,ind2) Returns the

value of index ind1,ind2 of the local 2-dimensional array variable with the given name (a string).

variable_global_set(name,value) Sets the global variable with the given name (a string) to the given value.

variable_global_array_set(name,ind,value) Sets the index ind in the global array variable with the given name (a string) to the given value.

variable_global_array2_set(name,ind1,ind2,value) Sets the index ind1,ind2 in the global 2-dimensional array variable with the given name (a string) to the given value.

variable_local_set(name,value) Sets the local variable with the given name (a string) to the given value.

variable_local_array_set(name,ind,value) Sets the index ind in the local array variable with the given name (a string) to the given value.

variable_local_array2_set(name,ind1,ind2,value) Sets the index ind1,ind2 in the local 2-dimensional array variable with the given name (a string) to the given value.

For example, you can write:

```
{
  if variable_global_exists('ammunition')
    global.ammunition += 1
  else
    global.ammunition = 0
}
```

You can also use these functions to pass variables to a script in a sort of by-reference way, by passing their names as strings and using the functions to change them.

You can change the program priority using the following function:

`set_program_priority(priority)` Sets the priority for the program. You can indicate a value between -3 and +3. A value of -3 means that the program will only run if no other process on the computer requires processing time, or stated differently, when all other processes are idle. Values of -2 and -1 are below normal, so other processes will get priority. 0 is the normal value. +1 and +2 give a higher priority, resulting possibly in higher speed and smoother game flow. But other processes will get much less processing time. +3 indicates real-time mode. In real-time mode basically all time is allotted to the game. This can lead to serious problems with any other applications running on the computer. Also keyboard events and e.g. the pressing of the close box might no longer be recorded by Windows. So only use this if you want all the processor time. Also better check carefully before using it and save the game before running.

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.

Information on user interaction can be found in the following pages:

The Keyboard The Mouse
The Joystick

The keyboard

For keyboard interaction, the following variables and functions exist:

keyboard_lastkey Keycode of last key pressed. See below for keycode constants. You can change it, e.g. set it to 0 if you handled it.
keyboard_key Keycode of current key pressed (see below; 0 if none).
keyboard_lastchar Last character pressed (as string).
keyboard_string String containing the last at most 1024 characters typed. This string will only contain the printable characters typed. It also correctly responds to pressing the backspace key by erasing the last character.

Sometimes it is useful to map one key to another. For example you might want to allow the player to use both the arrow keys and the numpad keys. Rather than duplicating the actions you can map the numpad keys to the arrow keys. Also you might want to implement a mechanism in which the player can set the keys to use. For this the following functions are available:

keyboard_set_map(key1,key2) Maps the key with keycode key1 to key2.
keyboard_get_map(key) Returns the current mapping for key.
keyboard_unset_map() Resets all keys to map to themselves.

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

particular useful when multiple keys are pressed simultaneously.

keyboard_check(key) Returns whether the key with the particular keycode is currently down.
keyboard_check_pressed(key) Returns whether the key with the particular keycode was pressed since the last step.
keyboard_check_released(key) Returns whether the key with the particular keycode was released since the last step.
keyboard_check_direct(key) Returns whether the key with the particular keycode is pressed by checking the hardware directly. The result is independent of which application has focus. It allows for a few more checks. In particular you can use keycodes vk_lshift, vk_lcontrol, vk_lalt, vk_rshift, vk_rcontrol and vk_ralt to check whether the left or right shift, control or alt key is pressed.

The following routines can be used to manipulate the keyboard state:

keyboard_get_numlock() Returns whether the numlock is set.
keyboard_set_numlock(on) Sets (true) or unsets (false) the numlock.
keyboard_key_press(key) Simulates a press of the key with the indicated keycode.
keyboard_key_release(key) Simulates a release of the key with the indicated keycode.

The following constants for virtual keycodes exist:

vk_nokey keycode representing that no key is pressed
vk_anykey keycode representing that any key is pressed

vk_left keycode for left arrow key
vk_right keycode for right arrow key
vk_up keycode for up arrow key
vk_down keycode for down arrow key
vk_enter enter key
vk_escape escape key
vk_space space key
vk_shift shift key
vk_control control key
vk_alt alt key
vk_backspace backspace key
vk_tab tab key
vk_home home key
vk_end end key
vk_delete delete key
vk_insert insert key
vk_pageup pageup key
vk_pagedown pagedown key
vk_pause pause/break key
vk_printscreen printscreen/sysrq key
vk_f1 ... vk_f12 keycodes for the function keys F1 to F12
vk_numpad0 ... vk_numpad9 number keys on the numeric keypad
vk_multiply multiply key on the numeric keypad
vk_divide divide key on the numeric keypad
vk_add add key on the numeric keypad
vk_subtract subtract key on the numeric keypad
vk_decimal decimal dot keys on the numeric keypad

For the letter keys use for example `ord('A')`. (The capital letters.) For the digit keys use for example `ord('5')` to get the <5> key. The following constants can only be used in `keyboard_check_direct`:

vk_lshift left shift key
vk_lcontrol left control key
vk_lalt left alt key
vk_rshift right shift key
vk_rcontrol right control key
vk_ralt right alt key

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

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

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

There are some additional functions related to keyboard interaction.

keyboard_clear(key) Clears the state of the key. This means that it will no longer generate keyboard events until it starts repeating.
io_clear() Clears all keyboard and mouse states.
io_handle() Handle user io, updating keyboard and mouse status.
keyboard_wait() Waits till the user presses a key on the keyboard.

The mouse

For mouse interaction, the following variables and functions exist:

mouse_x* X-coordinate of the mouse in the room. Cannot be changed.
mouse_y* Y-coordinate of the mouse in the room. Cannot be changed.
mouse_button Currently pressed mouse button. As value use `mb_none`, `mb_any`, `mb_left`, `mb_middle`, or `mb_right`.
mouse_lastbutton Last pressed mouse button.

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

mouse_check_button(numb) Returns whether the mouse button is currently down (use as values `mb_none`, `mb_left`, `mb_middle`, or `mb_right`).
mouse_check_button_pressed(numb) Returns whether the mouse button was pressed since the last step.
mouse_check_button_released(numb) Returns whether the mouse button was released since the last step.

There are some additional functions related to mouse interaction.

mouse_clear(button) Clears the state of the mouse button. This means that it will no longer generate mouse events until the player releases it and presses it again.
io_clear() Clears all keyboard and mouse states.
io_handle() Handle user io, updating keyboard and

mouse status.

`mouse_wait()` Waits till the user presses a mouse button .

You can change the way the mouse cursor looks like. You can choose any sprite for this. To this end you can use the following variable:

`cursor_sprite` Indicates the sprite that is used to represent the cursor (default no sprite is used, represented by a value of -1). You can assign one of the sprites to this variable to have it been drawn automatically at the position of the mouse cursor. (You can also switch off the windows mouse cursor in the **Global Game Settings**.)

The joystick

There are some events associated with joysticks. But to have full control over the joysticks there is a whole set of functions to deal with joysticks. *Game Maker* supports up to two joysticks. So all of these functions take a joystick id as argument.

`joystick_exists(id)` Returns whether joystick id (1 or 2) exists.
`joystick_name(id)` Returns the name of the joystick
`joystick_axes(id)` Returns the number of axes of the joystick.
`joystick_buttons(id)` Returns the number of buttons of the joystick.
`joystick_has_pov(id)` Returns whether the joystick has point-of-view capabilities.
`joystick_direction(id)` Returns the keycode (vk_numpad1 to vk_numpad9) corresponding to the direction of joystick id (1 or 2).
`joystick_check_button(id,numb)` Returns whether the joystick button is pressed (numb in the range 1-32).
`joystick_xpos(id)` Returns the position (-1 to 1) of the x-axis of joystick id.
`joystick_ypos(id)` Returns the joysticks y-position.
`joystick_zpos(id)` Returns the joysticks z-position (if it has a z-axis).
`joystick_rpos(id)` Returns the joysticks rudder position (or fourth axis).
`joystick_upos(id)` Returns the joysticks u-position (or fifth axis).
`joystick_vpos(id)` Returns the joysticks v-position (or

sixth axis).

`joystick_pov(id)` Returns the joysticks point-of view position. This is an angle between 0 and 360 degrees. 0 is forwards, 90 to the right, 180 backwards and 270 to the left. When no point-of-view direction is pressed by the user -1 is returned.

Game graphics

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

Information on game graphics can be found in the following pages:

Sprites and Images Backgrounds
Drawing Sprites and Backgrounds
Drawing Shapes
Fonts and Text
Advanced Drawing Functions
Drawing Surfaces
Tiles
The Display
The Window
Views
Repainting the Screen

Sprites and images

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

visible If visible is true (1) the image is drawn, otherwise it is not drawn. Invisible instances are still active and create collision events; only you don't see them. Setting the visibility to false is useful for e.g. controller objects (make them non-solid to avoid collision events) or hidden switches.
sprite_index This is the index of the current sprite for the instance. You can change it to give the instance a different sprite. As value you can use the names of the different sprites you defined. Changing the sprite does not change the index of the currently visible subimage.
sprite_width* Indicates the width of the sprite. This value cannot be changed but you might want to use it.
sprite_height* Indicates the height of the sprite. This value cannot be changed but you might want to use it.
sprite_xoffset* Indicates the horizontal offset of the sprite as defined in the sprite properties. This value cannot be changed but you might want to use it.
sprite_yoffset* Indicates the vertical offset of the sprite as defined in the sprite properties. This value cannot be

changed but you might want to use it.
image_number* The number of subimages for the current sprite for the instance (cannot be changed).
image_index When the image has multiple subimages the program cycles through them. This variable indicates the currently drawn subimage (they are numbered starting from 0). You can change the current image by changing this variable. The program will continue cycling, starting at this new index. (The value can have a fractional part. In this case it is always rounded down to obtain the subimage that is drawn.)
image_speed The speed with which we cycle through the subimages. A value of 1 indicates that each step we get the next image. Smaller values will switch subimages slower, drawing each subimage multiple times. Larger values will skip subimages to make the motion faster. Sometimes you want a particular subimage to be visible and don't want the program to cycle through all of them. This can be achieved by setting the speed to 0 and choosing the correct subimage. For example, assume you have an object that can rotate and you create a sprite that has subimages for a number of orientations (counter-clockwise). Then, in the step event of the object you can set

```
{
  image_index = direction * image_number/360;
  image_speed = 0;
}
```

depth Normally images are drawn in the order in which the instances are created. You can change this by setting the image depth. The default value is 0, unless you set it to a different value in the object properties. The higher the value the further the instance is away. (You can also use negative values.) Instances with

higher depth will lie behind instances with a lower depth. Setting the depth will guarantee that the instances are drawn in the order you want (e.g. the plane in front of the cloud). Background instances should have a high (positive) depth, and foreground instances should have a low (negative) depth.
image_xscale A scale factor to make larger or smaller images. A value of 1 indicates the normal size. You must separately set the horizontal xscale and vertical yscale. Changing the scale also changes the values for the image width and height and influences collision events as you might expect. Changing the scale can be used to get a 3-D effect. You can use a value of -1 to mirror the sprite horizontally.
image_yscale The vertical yscale. 1 is no scaling. You can use a value of -1 to flip the sprite vertically.
image_angle The angle with which the sprite is rotated. You specify this in degrees, counterclockwise. A value of 0 indicates no rotation. ***This variable can only be set in the Pro Edition!***
image_alpha Transparency (alpha) value to use when drawing the image. A value of 1 is the normal opaque setting; a value of 0 is completely transparent.
image_blend Blending color used when drawing the sprite. A value of c_white is the default. When you specify a different value the image is blended with this color. This can be used to colorize the sprite on the fly. ***This variable can only be set in the Pro Edition!***
bbox_left* Left side of the bounding box of the instance in the room, as defined by its image (taking scaling into account).
bbox_right* Right side of the bounding box of the instance in the room.
bbox_top* Top side of the bounding box of the instance in the room.

bbox_bottom* Bottom side of the bounding box of the instance in the room.

Backgrounds

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

background_color Background color for the room.
background_showcolor Whether to clear the window in the background color.
background_visible[0..7] Whether the particular background image is visible.
background_foreground[0..7] Whether the background is actually a foreground.
background_index[0..7] Background image index for the background.
background_x[0..7] X position of the background image.
background_y[0...7] Y position of the background image.
background_width[0...7]* Width of the background image.
background_height[0...7]* Height of the background image.
background_htiled[0..7] Whether horizontally tiled.
background_vtiled[0..7] Whether vertically tiled.
background_xscale[0..7] Horizontal scaling factor for the background. (This must be positive; you cannot use a negative value to mirror the background.)
background_yscale[0..7] Vertical scaling factor for the background. (This must be positive; you cannot use a negative value to flip the background.)
background_hspeed[0..7] Horizontal scrolling speed of the background (pixels per step).

background_vspeed[0..7] Vertical scrolling speed of the background (pixels per step).
background_blend[0..7] Blending color to use when drawing the background. A value of c_white is the default. ***Only available in the Pro Edition!***
background_alpha[0..7] Transparency (alpha) value to use when drawing the background. A value of 1 is the normal setting; a value of 0 is completely transparent.

Drawing sprites and backgrounds

Objects normally have a sprite associated to it that is drawn. But you can use the draw event to draw other things. This section and the next couple give you information on what is possible. First of all, there are functions to draw sprites and backgrounds in different ways. These give you more control over the appearance of the sprite. Also you can draw (parts of) backgrounds.

draw_sprite(sprite,subimg,x,y) Draws subimage subimg (-1 = current) of the sprite with its origin at position (x,y). (Without color blending and no alpha transparency.)
draw_sprite_stretched(sprite,subimg,x,y,w,h) Draws the sprite stretched so that it fills the region with top-left corner (x,y) and width w and height h.
draw_sprite_tiled(sprite,subimg,x,y) Draws the sprite tiled so that it fills the entire room. (x,y) is the place where one of the sprites is drawn.
draw_sprite_part(sprite,subimg,left,top,width,height,x,y) Draws the indicated part of subimage subimg (-1 = current) of the sprite with the top-left corner of the part at position (x,y).

draw_background(back,x,y) Draws the background at position (x,y). (Without color blending and no alpha transparency.)
draw_background_stretched(back,x,y,w,h) Draws the background stretched to the indicated region.
draw_background_tiled(back,x,y) Draws the background tiled so that it fills the entire room.
draw_background_part(back,left,top,width,height,x,y) Draws

the indicated part of the background with the top-left corner of the part at position (x,y).

The following functions are extended functions of the ones indicated above. ***These extended versions can only be used in the Pro Edition!***

draw_sprite_ext(sprite,subimg,x,y,xscale,yscale,rot,color,alpha) Draws the sprite scaled with factors xscale and yscale and rotated counterclockwise over rot degrees. color is the blending color (use c_white for no blending) and alpha indicates the transparency factor with which the images is merged with its background. A value of 0 makes the sprite completely transparent. A value of 1 makes it completely solid. This function can create great effect (for example partially transparent explosions).

draw_sprite_stretched_ext(sprite,subimg,x,y,w,h,color,alpha) Draws the sprite stretched so that it fills the region with top-left corner (x,y) and width w and height h. color is the blending color and alpha indicates the transparency setting.

draw_sprite_tiled_ext(sprite,subimg,x,y,xscale,yscale,color,alpha) Draws the sprite tiled so that it fills the entire room but now with scale factors and a color and transparency setting.

draw_sprite_part_ext(sprite,subimg,left,top,width,height,x,y,xscale,yscale,color,alpha) Draws the indicated part of subimage subimg (-1 = current) of the sprite with the top-left corner of the part at position (x,y) but now with scale factors and a color and transparency setting.

draw_sprite_general(sprite,subimg,left,top,width,height,x,y,xscale,yscale,rot,c1,c2,c3,c4,alpha) The most general drawing function. It draws the indicated part of subimage subimg (-1 = current) of the sprite with the top-left corner of the part at position (x,y) but now with

scale factors, a rotation angle, a color for each of the four vertices (top-left, top-right, bottom-right, and bottom-left), and an alpha transparency value. Note that rotation takes place around the top-left corner of the part.

draw_background_ext(back,x,y,xscale,yscale,rot,color,alpha) Draws the background scaled and rotated with blending color (use c_white for no blending) and transparency alpha (0-1).

draw_background_stretched_ext(back,x,y,w,h,color,alpha) Draws the background stretched to the indicated region. color is the blending color and alpha indicates the transparency setting.

draw_background_tiled_ext(back,x,y,xscale,yscale,color,alpha) Draws the background tiled so that it fills the entire room but now with scale factors and a color and transparency setting.

draw_background_part_ext(back,left,top,width,height,x,y,xscale,yscale,color,alpha) Draws the indicated part of the background with the top-left corner of the part at position (x,y) but now with scale factors and a color and transparency setting.

draw_background_general(back,left,top,width,height,x,y,xscale,yscale,rot,c1,c2,c3,c4,alpha) The most general drawing function. It draws the indicated part of the background with the top-left corner of the part at position (x,y) but now with scale factors, a rotation angle, a color for each of the four vertices (top-left, top-right, bottom-right, and bottom-left), and an alpha transparency value. Note that rotation takes place around the top-left corner of the part.

Drawing shapes

There is a whole collection of functions available to draw different shapes. Also there are functions to draw text (see next section). You can only use these in the drawing event of an object; these functions in general don't make any sense anywhere else in code. Realize that collisions between instances are determined by their sprites (or masks) and not by what you actually draw. The following drawing functions exist that can be used to draw basic shapes.

draw_clear(col) Clears the entire room in the given color (no alpha blending).
draw_clear_alpha(col,alpha) Clears the entire room in the given color and alpha value (in particular useful for surfaces).
draw_point(x,y) Draws a point at (x,y) in the current color.
draw_line(x1,y1,x2,y2) Draws a line from (x1,y1) to (x2,y2).
draw_line_width(x1,y1,x2,y2,w) Draws a line from (x1,y1) to (x2,y2) with width w.
draw_rectangle(x1,y1,x2,y2,outline) Draws a rectangle. `outline` indicates whether only the outline must be drawn (true) or it should be filled (false).
draw_roundrect(x1,y1,x2,y2,outline) Draws a rounded rectangle. `outline` indicates whether only the outline must be drawn (true) or it should be filled (false).
draw_triangle(x1,y1,x2,y2,x3,y3,outline) Draws a triangle. `outline` indicates whether only the outline must be drawn (true) or it should be filled (false).
draw_circle(x,y,r,outline) Draws a circle at (x,y) with

radius r. `outline` indicates whether only the outline must be drawn (true) or it should be filled (false).
`draw_ellipse(x1,y1,x2,y2,outline)` Draws an ellipse. `outline` indicates whether only the outline must be drawn (true) or it should be filled (false).
`draw_set_circle_precision(precision)` Sets the precision with which circles are drawn, that is, the number of segments they consist of. The precision must lie between 4 and 64 and must be dividable by 4. This is also used for drawing ellipses and rounded rectangles.
`draw_arrow(x1,y1,x2,y2,size)` Draws an arrow from (x1,y1) to (x2,y2). size indicates the size of the arrow in pixels.
`draw_button(x1,y1,x2,y2,up)` Draws a button, `up` indicates whether up (1) or down (0).
`draw_path(path,x,y,absolute)` With this function you can draw the indicated path in the room with its start at position (x,y). If absolute is true the path is drawn at the position where it was defined and the values of x and y are ignored.
`draw_healthbar(x1,y1,x2,y2,amount,backcol,mincol,maxcol,direction,showback,showborder)` With this function you can draw a healthbar (or any other bar that indicates some value, like e.g. the damage). The arguments x1, y1, x2 and y2 indicate the total area for the bar. amount indicates the percentage of the bar that must be filled (must lie between 0 and 100). backcol is the color of the background for the bar. mincol and maxcol indicate the color when the amount is 0 and 100 respectively. For an amount in between the color is interpolated. So you can easily make a bar that goes e.g. from green to red. The direction is the direction in which the bar is drawn. 0 indicates that the bar is anchored at the left, 1 at the right, 2 at the top and 3 at the bottom. Finally showback indicates whether a background box must be

shown and showborder indicated whether the box and bar should have a black border line.

Most of the above functions use the color and alpha setting that can be changed with following functions.

draw_set_color(col) Sets the drawing color to be used from now on for drawing primitives.
draw_set_alpha(alpha) Sets the alpha transparency value to be used from now on for drawing primitives. Should lie in the range 0-1. 0 is fully transparent, 1 is fully opaque.
draw_get_color() Returns the drawing color used for drawing primitives.
draw_get_alpha() Returns the alpha value used for drawing primitives.

A whole range of predefined colors is available:

c_aqua
c_black
c_blue
c_dkgray
c_fuchsia
c_gray
c_green
c_lime
c_ltgray
c_maroon
c_navy
c_olive
c_orange
c_purple
c_red

c_silver

c_teal

c_white

c_yellow

The following functions can help you to create the colors you want.

make_color_rgb(red,green,blue) Returns a color with the indicated red, green, and blue components, where red, green and blue must be values between 0 and 255.
make_color_hsv(hue,saturation,value) Returns a color with the indicated hue, saturation and value components (each between 0 and 255).
color_get_red(col) Returns the red component of the color.
color_get_green(col) Returns the green component of the color.
color_get_blue(col) Returns the blue component of the color.
color_get_hue(col) Returns the hue component of the color.
color_get_saturation(col) Returns the saturation component of the color.
color_get_value(col) Returns the value component of the color.
merge_color(col1,col2,amount) Returns a merged color of col1 and col2. The merging is determined by amount. A value of 0 corresponds to col1, a value of 1 to col2, and values in between to merged values.

The following miscellaneous functions exist.

draw_getpixel(x,y) Returns the color of the pixel corresponding to position (x,y) in the room. This is not

very fast, so use with care.
`screen_save(fname)` Saves a bmp image of the screen in the given filename. Useful for making screenshots.
`screen_save_part(fname,x,y,w,h)` Saves part of the screen in the given filename.

Fonts and text

In games you sometimes need to draw texts. To draw a text you have to specify the font to use. Fonts can be defined by creating font resources (either in the *Game Maker* program or using the functions to create resources). There are different functions to draw texts in different way. In each function you specify the position of the text on the screen. There are two functions to set the horizontal and vertical alignment of the text with respect to that position.

For text drawing the following functions exist:

draw_set_font(font) Sets the font that will be used when drawing text. Use -1 to set the default font (Arial 12).
draw_set_halign(halign) Sets the horizontal alignment used when drawing text. Choose one of the following constants as values:

fa_left
fa_center
fa_right

draw_set_valign(valign) Sets the vertical alignment used when drawing text. Choose one of the following constants as values:

fa_top
fa_middle
fa_bottom

draw_text(x,y,string) Draws the string at position (x,y), using the drawing color and alpha. A # symbol or carriage return chr(13) or linefeed chr(10) are

interpreted as newline characters. In this way you can draw multi-line texts. (Use \# to get the # symbol itself.)

draw_text_ext(x,y,string,sep,w) Similar to the previous routine but you can specify two more things. First of all, `sep` indicates the separation distance between the lines of text in a multiline text. Use -1 to get the default distance. Use `w` to indicate the width of the text in pixels. Lines that are longer than this width are split- up at spaces or - signs. Use -1 to not split up lines.

string_width(string) Width of the string in the current font as it would be drawn using the `draw_text()` function. Can be used for precisely positioning graphics.

string_height(string) Height of the string in the current font as it would be drawn using the `draw_text()` function.

string_width_ext(string,sep,w) Width of the string in the current font as it would be drawn using the `draw_text_ext()` function. Can be used for precisely positioning graphics.

string_height_ext(string,sep,w) Height of the string in the current font as it would be drawn using the `draw_text_ext()` function.

The following routines allow you to draw scaled and rotated text and also to use gradient colors on texts. ***These functions are only available in the Pro Edition!***

draw_text_transformed(x,y,string,xscale,yscale,angle) Draws the string at position (x,y) in the same way as above, but scale it horizontally and vertically with the indicated factors and rotate it counter-clockwise over `angle` degrees.

draw_text_ext_transformed(x,y,string,sep,w,xscale,yscale,angle) Combines the function `draw_text_ext` and

draw_text_transformed. It makes it possible to draw a multi-line text rotated and scaled.

draw_text_color(x,y,string,c1,c2,c3,c4,alpha) Draws the string at position (x,y) like above. The four colors specify the colors of the top-left, top-right, bottom-right, and bottom-left corner of the text. alpha is the alpha transparency to be used (0-1).

draw_text_ext_color(x,y,string,sep,w,c1,c2,c3,c4,alpha) Similar to draw_text_ext() but with colored vertices.

draw_text_transformed_color(x,y,string,xscale,yscale,angle,c1,c2,c3,c4,alpha) Similar to draw_text_transformed() but with colored vertices.

draw_text_ext_transformed_color(x,y,string,sep,w,xscale,yscale,angle,c1,c2,c3,c4,alpha) Similar to draw_text_ext_transformed() but with colored vertices.

Advanced drawing functions

This functionality is only available in the Pro Edition of Game Maker.

Above, a number of basic drawing functions have been described. Here you find a number of additional functions that offer you a lot more possibilities. First of all there are functions to draw shapes with gradient colors. Secondly there are functions to draw more general polygons, and finally there is the possibility to draw texture mapped polygons.

The following extended versions of the basic drawing functions exist. each of them gets extra color parameters that are used the determine the color at different vertices. The standard drawing color is not used in these functions.

draw_point_color(x,y,col1) Draws a point at (x,y) in the given color.

draw_line_color(x1,y1,x2,y2,col1,col2) Draws a line from (x1,y1) to (x2,y2), interpolating the color between col1 and col2.

draw_line_width_color(x1,y1,x2,y2,w,col1,col2) Draws a line from (x1,y1) to (x2,y2) with width w interpolating the color between col1 and col2.

draw_rectangle_color(x1,y1,x2,y2,col1,col2,col3,col4,outline) Draws a rectangle. The four colors indicated the colors at the top-left, top-right, bottom-right, and bottom-left vertex. outline indicates whether only the outline must be drawn (true) or it should be filled (false).

draw_roundrect_color(x1,y1,x2,y2,col1,col2,outline) Draws

a rounded rectangle. col1 is the color in the middle and col2 the color at the boundary. outline indicates whether only the outline must be drawn (true) or it should be filled (false).

draw_triangle_color(x1,y1,x2,y2,x3,y3,col1,col2,col3,outline) Draws a triangle. The three colors are the colors of the three vertices which is interpolated over the triangle. outline indicates whether only the outline must be drawn (true) or it should be filled (false).

draw_circle_color(x,y,r,col1,col2,outline) Draws a circle at (x,y) with radius r. col1 is the color in the middle and col2 the color at the boundary. outline indicates whether only the outline must be drawn (true) or it should be filled (false).

draw_ellipse_color(x1,y1,x2,y2,col1,col2,outline) Draws an ellipse. col1 is the color in the middle and col2 the color at the boundary. outline indicates whether only the outline must be drawn (true) or it should be filled (false).

You can also draw larger primitives. This works slightly different. You begin by specifying the primitive you want to draw. Next you specify the vertices for it, and finally you end the primitive, at which moment it is drawn. There are six types of primitives:

pr_pointlist The vertices are a set of points.

pr_linelist The vertices are a set of line segments. Each pair of vertices forms a line segment. So there must be an even set of vertices.

pr_linestrip The vertices form a polyline with the first connected to the second, the second to the third, etc. The last one is not connected to the first one. You have to specify an extra copy of the first vertex for this.

pr_trianglelist The vertices are a set of triangles. Each

triple of vertices forms a triangle. So the number of vertices must be a multiple of 3.

pr_trianglestrip The vertices again form triangles but this time it works slightly different. The first three form the first triangle. The last two of these vertices, together with the next vertex, form the second triangle, etc. So each new vertex specifies a new triangle, connected to the previous one.

pr_trianglefan Similar to a triangle list but this time the first vertex is part of all the triangles. Again, each new vertex specifies a new triangle, connected to the previous vertex and the first vertex.

The following functions exist for drawing primitives

draw_primitive_begin(kind) Start a primitive of the indicated kind.

draw_vertex(x,y) Add vertex (x,y) to the primitive, using the color and alpha value set before.

draw_vertex_color(x,y,col,alpha) Add vertex (x,y) to the primitive, with its own color and alpha value. This allows you to create primitives with smoothly changing color and alpha values.

draw_primitive_end() End the description of the primitive. This function actually draws it.

Finally, it is possible to draw primitives using sprites or backgrounds as textures. When using a texture the image is placed on the primitive, reshaping it to fit the primitive. Textures are used to add detail to primitives, e.g. a brick wall. To use textures you first must obtain the id of the texture you want to use. For this the following functions exist:

sprite_get_texture(spr,subimg) Returns the id of the texture corresponding to subimage subimg of the

indicated sprite.

background_get_texture(back) Returns the id of the texture corresponding to the indicated background.

A selected texture might not yet be in video memory. The system will put it there once you need it but sometimes you want to decide this yourself. For this the following two functions exist:

texture_preload(texid) Puts the texture immediately into video memory.

texture_set_priority(texid,prio) When there is too little video memory some will be removed temporarily to make room for others that are needed. The ones with lowest priority are removed first. Default, all have priority 0 but you can change the priority here. (Use positive values!)

To add textures to primitives you must specify which parts of the textures must be put where on the primitive. Positions in the texture are indicated with values between 0 and 1 but there is a problem here. Sizes of textures must be powers of 2 (so e.g. 32x32 or 64x64). If you want to use sprites or background as textures you better make sure they have such a size. If not, the test will be blank. To find out which part of the texture is actually used you can use the following two functions. They return a value between 0 and 1 that indicates the width or height of the actual part of the texture being used. Specifying this value as texture coordinate will indicate the right or bottom side of the texture.

texture_get_width(texid) Returns the width of the texture with the given id. The width lies in the range 0-1.

texture_get_height(texid) Returns the height of the

texture with the given id. The height lies in the range 0-1.

To draw textured primitives you use the following functions:

draw_primitive_begin_texture(kind,texid) Start a primitive of the indicated kind with the given texture.
draw_vertex_texture(x,y,xtex,ytex) Add vertex (x,y) to the primitive with position (xtex,ytex) in the texture, blending with the color and alpha value set before. xtex and ytex should normally lie between 0 and 1 but also larger values can be used, leading to a repetition of the texture (see below).
draw_vertex_texture_color(x,y,xtex,ytex,col,alpha) Add vertex (x,y) to the primitive with position (xtex,ytex) in the texture, blending with its own color and alpha value.
draw_primitive_end() End the description of the primitive. This function actually draws it.

There are three functions that influence how textures are drawn:

texture_set_interpolation(linear) Indicates whether to use linear interpolation (true) or pick the nearest pixel (false). Linear interpolation gives smoother textures but can also be a bit blurry and sometimes costs extra time. This setting also influence the drawing of sprites and background. Default is false. (This can also be changed in the global game settings.)
texture_set_blending(blend) Indicates whether to use blending with colors and alpha values. Switching this off might be faster on old hardware. This setting also influence the drawing of sprites and background. Default is true.
texture_set_repeat(repeat) Indicates whether to use repeat the texture. This works as follows. As indicated

above texture coordinates lie in the range 0-1. If you specify a value larger than 1, default the rest is not drawn. By setting repeat to true the texture is repeated. Note that sprites and backgrounds are always drawn without repeating. So once you draw a sprite of background this value is reset to false. Default is false.

There are two more function that are not only useful for drawing textures. Normally primitives are blended with the background using the alpha value. You can actually indicate how this must happen. Besides the normal mode it is possible to indicate that the new color must be added to the existing color or subtracted from the existing color. This can be used to create e.g. spot lights or shadows. Also it is possible to sort of take the maximum of the new and existing color. This can avoid certain saturation effects that you can get with adding. Note that both subtracting and maximum do not take the alpha value fully into account. (DirectX does not allow this.) So you better make sure the outside area is black. There are two functions. The first one only gives you the four options described above. The second function gives you a lot more possibilities. You should experiment a bit with the settings. If used effectively they can be used to create e.g. interesting explosion or halo effects.

`draw_set_blend_mode(mode)` Indicates what blend mode to use. The following values are possible: `bm_normal`, `bm_add`, `bm_subtract`, and `bm_max`. Don't forget to reset the mode to normal after use because otherwise also other sprites and even the backgrounds are drawn with the new blend mode.

`draw_set_blend_mode_ext(src,dest)` Indicates what blend mode to use for both the source and destination color. The new color is some factor times the source and

another factor times the destination. These factors are set with this function. To understand this, the source and destination both have as red, green, blue, and alpha component. So the source is (Rs, Gs, Bs, As) and the destination is (Rd, Gd, Bd, Ad). All are considered to lie between 0 and 1. The blend factors you can choose for source and destination are:

- bm_zero: Blend factor is (0, 0, 0, 0).
- bm_one: Blend factor is (1, 1, 1, 1).
- bm_src_color: Blend factor is (Rs, Gs, Bs, As).
- bm_inv_src_color: Blend factor is (1–Rs, 1–Gs, 1–Bs, 1–As).
- bm_src_alpha: Blend factor is (As, As, As, As).
- bm_inv_src_alpha: Blend factor is (1–As, 1–As, 1–As, 1–As).
- bm_dest_alpha: Blend factor is (Ad, Ad, Ad, Ad).
- bm_inv_dest_alpha: Blend factor is (1–Ad, 1–Ad, 1–Ad, 1–Ad).
- bm_dest_color: Blend factor is (Rd, Gd, Bd, Ad).
- bm_inv_dest_color: Blend factor is (1–Rd, 1–Gd, 1–Bd, 1–Ad).
- bm_src_alpha_sat: Blend factor is (f, f, f, 1); f = min(As, 1–Ad).

For example, the normal blending mode sets the source blending to bm_src_alpha and the destination blending to bm_inv_src_alpha. Don't forget to reset the mode to normal after use because otherwise also other sprites and even the backgrounds are drawn with the new blend mode.

Drawing texture primitives is a bit of work but can lead to great results. You can even use it to make fake 3D games.

Drawing Surfaces

This functionality is only available in the Pro Edition of Game Maker.

In certain situations you might want to paint not directly on the screen but on a canvas that can then later be used to paint things on the screen. Such a canvas is called a surface. For example, you want to let the user draw on the screen. The paint should not be drawn on the screen (because it will be removed each next step), but instead you want to draw it on a separate surface that is copied onto the screen in each step. Or you want to use a texture that changes over time.

Surfaces make this all possible. They are actually rather simple to use. You first create a surface. Next you indicate that further drawing should happen on this surface. From that moment on all drawing functions operate on the surface. Once you are done you reset the drawing target and further drawing happens on the screen again. You can draw the surface on the screen in many different ways or use it as a texture. There are though a few things you must be aware of. See the remarks at the end.

The following functions exist to deal with surfaces

`surface_create(w,h)` Creates a surface of the indicated width and height. Returns the id of the surface, which must be used in all further calls. Note that the surface will not be cleared. This is the responsibility of the user. (Set it as a target and call the appropriate clear function.)

`surface_free(id)` Frees the memory used by the surface.

surface_exists(id) Returns whether the surface with the indicated id exists.

surface_get_width(id) Returns the width of the surface.
surface_get_height(id) Returns the height of the surface.
surface_get_texture(id) Returns the texture corresponding to the surface. This can be used to draw textured objects with the image of the surface.

surface_set_target(id) Sets the indicated surface as the drawing target. All subsequent drawing happens on this surface. It resets the projection to simply cover the surface.
surface_reset_target() Resets the drawing target to the normal screen.

surface_getpixel(id,x,y) Returns the color of the pixel corresponding to position (x,y) in the surface. This is not very fast, so use with care.
surface_save(id,fname) Saves a bmp image of the surface in the given filename. Useful for making screenshots.
surface_save_part(id,fname,x,y,w,h) Saves part of the surface in the given filename.

draw_surface(id,x,y) Draws the surface at position (x,y). (Without color blending and no alpha transparency.)
draw_surface_stretched(id,x,y,w,h) Draws the surface stretched to the indicated region.
draw_surface_tiled(id,x,y) Draws the surface tiled so that it fills the entire room.
draw_surface_part(id,left,top,width,height,x,y) Draws the indicated part of the surface with its origin at position (x,y).
draw_surface_ext(id,x,y,xscale,yscale,rot,color,alpha)

Draws the surface scaled and rotated with blending color (use c_white for no blending) and transparency alpha (0-1).

draw_surface_stretched_ext(id,x,y,w,h,color,alpha) Draws the surface stretched to the indicated region. color is the blending color and alpha indicates the transparency setting.

draw_surface_tiled_ext(id,x,y,xscale,yscale,color,alpha) Draws the surface tiled so that it fills the entire room but now with scale factors and a color and transparency setting.

draw_surface_part_ext(id,left,top,width,height,x,y,xscale,yscale,color,alpha) Draws the indicated part of the surface with its origin at position (x,y) but now with scale factors and a color and transparency setting.

draw_surface_general(id,left,top,width,height,x,y,xscale,yscale,rot,c1,c2,c3,c4,alpha) The most general drawing function. It draws the indicated part of the surface with its origin at position (x,y) but now with scale factors, a rotation angle, a color for each of the four vertices (top-left, top-right, bottom-right, and bottom-left), and an alpha transparency value.

surface_copy(destination,x,y,source) Copies the source surface at position (x,y) in the destination surface. (Without any form of blending.)

surface_copy_part(destination,x,y,source,xs,ys,ws,hs) Copies the indicated part of the source surface at position (x,y) in the destination surface. (Without any form of blending.)

Note that there are no functions to copy part of the screen to a surface. (This is impossible due to possible format differences between the screen and the surfaces.) If this is required you must set a surface as render target and next

draw the room. You can then use the surface copying routines to get parts of it.

Note that you can also create sprites and backgrounds from surfaces. See the section on changing resources for more information.

Some care must be taken when using these functions. In particular please notice the following:

- You should never change the drawing target while you are actually drawing on the screen, that is, never use it in drawing events. This will cause serious problems with the projection and viewport.
- Surfaces do not work correctly with 3D mode. You can use them while not in 3D mode (by calling d3d_end() before using them, but once you start 3D mode again the surfaces will be destroyed.
- For reasons of speed, the surface is maintained in videomemory only. As a result, you might loose the surface when e.g. the screen resolution changes or the screensaver pops up.
- Surfaces will not be saved when saving a game.

Tiles

As you should know you can add tiles to rooms. A tile is a part of a background resource. Tiles are just visible images. They do not react to events and they do not generate collisions. As a result, tiles are handled a lot faster than objects. Anything that does not need events or collisions can best be done through tiles. Also, often one better uses a tile for the nice graphics while a simple object is used to generate the collision events.

You actually have more control over tiles than you might think. You can add them when designing the room but you can also add them during the running of the game. You can change their position, and even scale them or make them partially transparent. A tile has the following properties:

- **`background`**. The background resource from which the tile is taken.
- **`left, top, width, height`**. The part of the background that is used.
- **`x,y`**. The position of the top left corner of the tile in the room.
- **`depth`**. The depth of the tile. You can choose any depth you like, making tiles appear between object instances.
- **`visible`**. Whether the tile is visible.
- **`xscale, yscale`**. Each tile can be drawn scaled (default is 1).
- **`blend`**. A blending color used when drawing the tile.
- **`alpha`**. An alpha value indicating tile transparency. 1 = not transparent, 0 = fully transparent.

To change the properties of a particular tile you need to know its id. When you add tiles when creating rooms the id is shown in the information bar at the bottom. There is also a function to find the id of a tile at a particular position.

The following functions exist that deal with tiles:

tile_add(background,left,top,width,height,x,y,depth) Adds a new tile to the room with the indicated values (see above for their meaning). The function returns the id of the tile that can be used later on.
tile_delete(id) Deletes the tile with the given id.
tile_exists(id) Returns whether a tile with the given id exists.

tile_get_x(id) Returns the x-position of the tile with the given id.
tile_get_y(id) Returns the y-position of the tile with the given id.
tile_get_left(id) Returns the left value of the tile with the given id.
tile_get_top(id) Returns the top value of the tile with the given id.
tile_get_width(id) Returns the width of the tile with the given id.
tile_get_height(id) Returns the height of the tile with the given id.
tile_get_depth(id) Returns the depth of the tile with the given id.
tile_get_visible(id) Returns whether the tile with the given id is visible.
tile_get_xscale(id) Returns the xscale of the tile with the given id.
tile_get_yscale(id) Returns the yscale of the tile with the given id.

tile_get_background(id) Returns the background of the tile with the given id.
tile_get_blend(id) Returns the blending color of the tile with the given id.
tile_get_alpha(id) Returns the alpha value of the tile with the given id.

tile_set_position(id,x,y) Sets the position of the tile with the given id.
tile_set_region(id,left,right,width,height) Sets the region of the tile with the given id in its background.
tile_set_background(id,background) Sets the background for the tile with the given id.
tile_set_visible(id,visible) Sets whether the tile with the given id is visible.
tile_set_depth(id,depth) Sets the depth of the tile with the given id.
tile_set_scale(id,xscale,yscale) Sets the scaling of the tile with the given id.
tile_set_blend(id,color) Sets the blending color of the tile with the given id. ***Only available in the Pro Edition!***
tile_set_alpha(id,alpha) Sets the alpha value of the tile with the given id.

The following functions deal with layers of tiles, that is, collections of tiles at the same depth.

tile_layer_hide(depth) Hides all tiles at the indicated depth layer.
tile_layer_show(depth) Shows all tiles at the indicated depth layer.
tile_layer_delete(depth) Deletes all tiles at the indicated depth layer.

tile_layer_shift(depth,x,y) Shifts all tiles at the indicated depth layer over the vector x,y. Can be used to create scrolling layers of tiles.

tile_layer_find(depth,x,y) Returns the id of the tile with the given depth at position (x,y). When no tile exists at the position -1 is returned. When multiple tiles with the given depth exist at the position the first one is returned.

tile_layer_delete_at(depth,x,y) Deletes the tile with the given depth at position (x,y). When multiple tiles with the given depth exist at the position they are all deleted.

tile_layer_depth(depth,newdepth) Changes the depth of all tiles at the indicated depth to the new depth. With this function you can move whole tile layers to a new depth.

The display

The display represents the whole area on the monitor. It has a size (typically 1024x768, or 1280x1024), a color depth, that is, the number of bits that are used to represent a single pixel (typically 16 = High Color or 32 = Full Color) and a refresh frequency, that is, the number of times per second the display is refreshed (typically between 60 and 120). These settings can normally be changed though the display properties. For games though, in particular when they run in full-screen mode, it is important to be able to change these settings. All these settings can be initialized for the game in the **Game Settings**. For use during game play the following functions exist. Note though that changing the settings during game play will result in a time delay as things has to be rebuilt. ***The functions to set the mode are only available in the Pro Edition.***

`display_get_width()` Returns the width of the display in pixels.

`display_get_height()` Returns the height of the display in pixels.

`display_get_colordepth()` Returns the color depth in bits.

`display_get_frequency()` Returns the refresh frequency of the display.

`display_set_size(w,h)` Sets the width and height of the display in pixels. Returns whether this was successful. (Realize that only certain combinations are allowed.)

`display_set_colordepth(coldepth)` Sets the color depth. In general only 16 and 32 are allowed values. Returns whether successful.

`display_set_frequency(frequency)` Sets the refresh frequency for the display. Only few frequencies are

allowed. Typically you could set this to 60 with a same room speed to get smooth 60 frames per second motion. Returns whether successful.

display_set_all(w,h,frequency,coldepth) Sets all at once. Use -1 for values you do not want to change. Returns whether successful.

display_test_all(w,h,frequency,coldepth) Tests whether the indicated settings are allowed. It does not change the settings. Use -1 for values you do not want to change. Returns whether the settings are allowed.

display_reset() Resets the display settings to the ones when the program was started.

Sometimes it is useful to get information about the position of the mouse on the display or to change this position. For this the following functions exist:

display_mouse_get_x() Returns the x-coordinate of the mouse on the display.

display_mouse_get_y() Returns the y-coordinate of the mouse on the display.

display_mouse_set(x,y) Sets the position of the mouse on the display to the indicated values.

The window

The actual game happens in a window. This window has a number of properties, like whether it has a border, whether it is full screen, etc. Normally these are set in the **Game Settings**. But you can change then during the game. The following functions exist for this:

`window_set_visible(visible)` Sets whether the game window is visible. Clearly you normally want the window to remain visible during the whole game. The program will not receive keyboard events when the window is invisible.

`window_get_visible()` Returns whether the game window is visible.

`window_set_fullscreen(full)` Sets whether the window is shown in full screen mode.

`window_get_fullscreen()` Returns whether the window is shown in full screen mode.

`window_set_showborder(show)` Sets whether the border around the window is shown. (In full screen mode it is never shown.)

`window_get_showborder()` Returns whether the border around the window is shown in windowed mode.

`window_set_showicons(show)` Sets whether the border icons (iconize, maximize, close) are shown. (In full screen mode these are never shown.)

`window_get_showicons()` Returns whether the border icons are shown in windowed mode.

`window_set_stayontop(stay)` Sets whether the window must always stay on top of other windows.

`window_get_stayontop()` Returns whether the window always stays on top of other windows.

window_set_sizeable(sizeable) Sets whether the window is sizeable by the player. (The player can only size it when the border is shown and the window is not in full screen mode.)

window_get_sizeable() Returns whether the window is sizeable by the player.

window_set_caption(caption) Sets the caption string for the window. Normally you specify this when defining the room and it can be changed using the variable room_caption. So this function is normally not useful, unless you draw the room yourself rather than letting *Game Maker* do it. The caption is only visible when the window has a border and when it is not in full screen mode.

window_get_caption() Returns the window caption.

window_set_cursor(curs) Sets the mouse cursor used in the window. You can use the following constant:

cr_default
cr_none
cr_arrow
cr_cross
cr_beam
cr_size_nesw
cr_size_ns
cr_size_nwse
cr_size_we
cr_uparrow
cr_hourglass
cr_drag
cr_nodrop
cr_hsplit
cr_vsplit
cr_multidrag

cr_sqlwait

cr_no

cr_appstart

cr_help

cr_handpoint

cr_size_all

In particular, to hide the mouse cursor, use `cr_none` as value.

window_get_cursor() Returns the cursor used in the window.

window_set_color(color) Sets the color of the part of the window that is not used for displaying the room.

window_get_color() Returns the window color.

window_set_region_scale(scale,adaptwindow) If the window is larger than the actual room normally the room is displayed in a region centered in the window. It is though possible to indicate that it must be scaled to fill the whole or part of the window. A value of 1 is no scaling. If you use a value of 0 the region will be scaled to fill the whole window. If you set it to a negative value it will be scaled to the maximal size inside the window while maintaining the aspect ratio (this is often what you want). `adaptwindow` indicates whether the window size must be adapted if the scaled room does not fit in. Adapting the window is only effective when the scale factor is positive.

window_get_region_scale() Returns the scale factor for the drawing region.

The window has a position on the screen and a size. (When we talk about position and size we always mean the client part of the window without the borders.) You can change these although you hardly ever do from within your game. Normally they are determined automatically or by the

player. The following functions can be used to change the window position and size. Note that these functions deal with the windowed mode. If the window is in full screen mode they can still be used but will only have effect when switching full screen mode off.

window_set_position(x,y) Sets the position of the (client part of the) window to the indicated position.
window_set_size(w,h) Sets the size of the (client part of the) window to the indicated size. Note that is the indicated size is too small to fit the drawing region it is kept large enough for the region to fit it.
window_set_rectangle(x,y,w,h) Sets the position and size of the window rectangle. (Does both previous routines in one step.)
window_center() Centers the window on the screen.
window_default() Gives the window the default size and position (centered) on the screen.
window_get_x() Returns the current x-coordinate of the window.
window_get_y() Returns the current y-coordinate of the window.
window_get_width() Returns the current width of the window.
window_get_height() Returns the current height of the window.

Again, you probably never want to use any of the window positioning functions as *Game Maker* takes care of these automatically.

In rare cases you might want to know the position of the mouse with respect to the window. (Normally you always use the mouse position with respect to the room or with respect to a view.) The following functions exist for this.

`window_mouse_get_x()` Returns the x-coordinate of the mouse in the window.
`window_mouse_get_y()` Returns the y-coordinate of the mouse in the window.
`window_mouse_set(x,y)` Sets the position of the mouse in the window to the indicated values.

Views

As you should know you can define up to eight different views when designing rooms. A view is defined by its view area in the room and its viewport on the screen (or to be precise in the drawing region within the window). Using views you can show different parts of the room at different places on the screen. Also, you can make sure that a particular object always stays visible.

You can control the views from within code. You can make views visible and invisible and change the place or size of the views in the drawing region or the position and size of the view in the room (which is in particular useful when you indicate no object to be visible). You can change the size of the horizontal and vertical border around the visible object, and you can indicate which object must remain visible in the views. The latter is very important when the important object changes during the game. For example, you might change the main character object based on its current status. Unfortunately, this does mean that it is no longer the object that must remain visible. This can be remedied by one line of code in the creation event of all the possible main objects (assuming this must happen in the first view):

```
{
  view_object[0] = object_index;
}
```

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

view_enabled Whether views are enabled or not.
view_current* The currently drawn view (0-7). Use this only in the drawing event. You can for example check this variable to draw certain things in only one view. Variable cannot be changed.
view_visible[0..7] Whether the particular view is visible on the screen.

view_xview[0..7] X position of the view in the room.
view_yview[0..7] Y position of the view in the room.
view_wview[0..7] Width of the view in the room.
view_hview[0..7] Height of the view in the room.
view_xport[0..7] X-position of the viewport in the drawing region.
view_yport[0..7] Y-position of the viewport in the drawing region.
view_wport[0..7] Width of the viewport in the drawing region.
view_hport[0..7] Height of the viewport in the drawing region.
view_angle[0..7] Rotation angle used for the view in the room (counter-clockwise in degrees.
view_hborder[0..7] Size of horizontal border around the visible object (in pixels).
view_vborder[0..7] Size of vertical border around visible object (in pixels).
view_hspeed[0..7] Maximal horizontal speed of the view.
view_vspeed[0..7] Maximal vertical speed of the view.
view_object[0..7] Object whose instance must remain visible in the view. If there are multiple instances of this object only the first one is followed. You can also assign an instance id to this variable. In that case the particular instance is followed.

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

window_set_region_size(w,h,adaptwindow) Set the width and height of the drawing region in the window. adaptwindow indicates whether the window size must be adapted if the region does not fit in. The window size will always be adapted if you use fixed scaling. (See the function window_set_region_scale().)
window_get_region_width() Returns the current width of the drawing region.
window_get_region_height() Returns the current height of the drawing region.

In a game you often need the position of the mouse cursor. Normally you use for this the variables mouse_x and mouse_y. When there are multiple views, these variables give the mouse position with respect to the view the mouse is in. Sometimes though, you might need the mouse position with respect to a particular view, also when it is outside that view. For this the following functions exist.

window_view_mouse_get_x(id) Returns the x-coordinate of the mouse with respect to the view with index id.
window_view_mouse_get_y(id) Returns the y-coordinate of the mouse with respect to the view with index id.
window_view_mouse_set(id,x,y) Sets the position of the mouse with respect to the view with index id.
window_views_mouse_get_x() Returns the x-coordinate of the mouse with respect to the view it is in (same as mouse_x).

window_views_mouse_get_y() Returns the y-coordinate of the mouse with respect to the view it is in (same as `mouse_y`).

window_views_mouse_set(x,y) Sets the position of the mouse with respect to the first view that is visible.

Repainting the screen

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

`screen_redraw()` Redraws the room by calling all draw events.

`screen_refresh()` Refreshes the screen using the current room image (not performing drawing events).

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

Now you should also realize why you cannot use drawing actions or functions in other events than drawing events. They will draw things on the internal image but these won't be visible on the screen. And when the drawing events are performed, first the room background is drawn, erasing all you drew on the internal image. But when you use `screen_refresh()` after your drawing, the updated image will become visible on the screen. So, for example, a script can

draw some text on the screen, call the refresh function and then wait for the player to press a key, like in the following piece of code.

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

Please realize that, when you draw in another event than the drawing event, you draw simply on the image, not in a view! So the coordinates you use are the same as if there are no views. Be careful when using this technique. Make sure you understand it first and realize that refreshing the screen takes some time.

When you are drawing the room yourself it can be useful to NOT let it be drawn automatic at all. For example, you might want to draw the room only every 5 steps. You can use the following functions for this:

`set_automatic_draw(value)` Indicates whether to automatically redraw the room (true, default) or not (false).

Finally there is a function with which you can set whether to synchronize the drawing with the refresh frequency of the monitor:

`set_synchronization(value)` Indicates whether to synchronize the drawing with the refresh frequency of the monitor.

You can also force a wait for the next vertical synchronization using the following function:

`screen_wait_vsync()` Waits for the next vertical synchronization of the monitor.

Sound and music

Sound plays a crucial role in computer games. Sounds are added to your game in the form of sound resources. Make sure that the names you use are valid variable names. As you will have seen you can indicate four different types of sound: normal sounds, background music, 3D sounds, and sounds that must be played through the media player.

Normal sounds are used for sound effects. In general wave files are used for this. Many of them can play at the same moment (even multiple instances of the same normal sound). You can apply all sorts of effects to them.

Background music typically consist of midi files but sometimes also wave files are used. Sound effects can be applied to it. The only difference with normal sounds is that only one background music can play at any moment. If you start one the current one is stopped.

3D sounds allow for 3D sound effects which is described below. They are mono sounds (wave or midi).

Finally, if you want to use another sound type, in particular mp3, these cannot be played through DirectX. As a result the normal media player must be used for this. This is much more limited. Only one sound can play at the same time. No effects can be applied (not even volume changes) and the timing for e.g. looping sounds is poor. There can also be delays in playing these sounds. You are strongly encouraged not to use them. (Some computers might also not support them)

Information on sound and music can be found in the following pages:

- Basic sound functions Special effects
- 3D music
- CD music

Basic sound functions

There are five basic functions related to sounds, two to play a sound, one to check whether a sound is playing, and two to stop sounds. Most take the index of the sound as argument. The name of the sound represents its index. But you can also store the index in a variable, and use that.

`sound_play(index)` Plays the indicates sound once. If the sound is background music the current background music is stopped.

`sound_loop(index)` Plays the indicates sound, looping continuously. If the sound is background music the current background music is stopped.

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

`sound_stop_all()` Stops all sounds.

`sound_isplaying(index)` Returns whether (a copy of) the indicated sound is playing. Note that this functions returns true when the sound actually plays through the speakers. After you call the function to play a sound it does not immediately reach the speakers so the function might still return false for a while. Similar, when the sound is stopped you still hear it for a while (e.g. because of echo) and the function will still return true.

It is possible to use further sound effects. In particular you can change the volume and the pan, that is, whether the sound comes from the left or right speaker. In all these cases the volume can only be reduced. These functions do

not work for files that play through the media player (like mp3 files).

sound_volume(index,value) Changes the volume for the indicated sound (0 = low, 1 = high).
sound_global_volume(value) Changes the global volume for all sounds (0 = low, 1 = high).
sound_fade(index,value,time) Changes the volume for the indicated sound to the new value(0 = low, 1 = high) during the indicated time (in milliseconds). This can be used to fade out or fade in music.
sound_pan(index,value) Changes the pan for the indicated sound (-1 = left, 0 = center, 1 = right).
sound_background_tempo(factor) Changes the tempo of the background music (if it is a midi file). `factor` indicates the factor with which to multiply the tempo. So a value of 1 corresponds to the normal tempo. Larger values correspond to a faster tempo, smaller values to a slower tempo. Must lie between 0.01 and 100.

Besides midi and wave files (and mp3 file) there is actually a fourth type of file that can be played: direct music files. These have the extension .sgt. Such files though often refer to other files describing e.g. band or style information. To find these files, the sound system must know where they are located. To this end you can use the following functions to set the search directory for files. Note though that you must add the files yourself. *Game Maker* does not automatically include such additional files.

sound_set_search_directory(dir) Sets the directory in which direct music files are to be found. The dir string should not incluce the final backslash.

Sound effects

This functionality is only available in the Pro Edition of Game Maker.

Sound effects can be used to change the way sounds and background music sounds. Realize that sound effects only apply to wave files and midi files, not to mp3 files. This section describes the functions that exist for using and changing sound effects. Realize that to use these functions you need to have a good understanding of how sound and synthesizers work. No explanation of the different parameters is given here. Search the web or books for further informations.

To apply a sound effect to a particular sound you can either indicate this when defining the sound resource or you can use the following function

`sound_effect_set(snd,effect)` Sets a (combination of) sound effect(s) for the indicated sound. `effect` can be any of the following values:

`se_none se_chorus`
`se_echo`
`se_flanger`
`se_gargle`
`se_reverb`
`se_compressor`
`se_equalizer`

You can set a combination of effects by adding up the values. So e.g. you can use

```
sound_effect_set(snd,se_echo+se_reverb);
```

to get a combination of echo and reverb effects.

All effects have some default settings. You can change these settings once an effect has been applied to a sound. The order here is crucial. You first apply the effect to the sound and next set the parameters for it. Once you reapply effects to the sound, the settings are gone and you have to set them again. Note that all parameters must lie in a particular range, which is indicated below. The following functions exist for changing effect parameters:

sound_effect_chorus(snd,wetdry,depth,feedback,frequency,wave,delay,phase) Sets the parameters for the chorus effect for the indicated sound. The following parameters can be set:

`wetdry` Ratio of wet (processed) signal to dry (unprocessed) signal. (range: 0 to 100, default 50)
`depth` Percentage by which the delay time is modulated by the low-frequency oscillator, in hundredths of a percentage point. (range: 0 to 100, default 25)
`feedback` Percentage of output signal to feed back into the effect's input. (range: -99 to 99, default 0)
`frequency` Frequency of the LFO. (range: 0 to 10, default 0)
`wave` Waveform of the LFO. (0 = triangle, 1 = wave, default 1)
`delay` Number of milliseconds the input is delayed before it is played back. (range: 0 to 20, default 0)
`phase` Phase differential between left and right LFOs. (range: 0 to 4, default 2)

sound_effect_echo(snd,wetdry,feedback,leftdelay,rightdelay,pandelay) Sets the parameters for the echo effect for the indicated sound. The following parameters can be set:

`wetdry` Ratio of wet (processed) signal to dry (unprocessed) signal. (range: 0 to 100, default 50)
`feedback` Percentage fed back into input (range: 0 to 100, default 0)
`leftdelay` Delay for left channel, in milliseconds. (range: 1 to 2000, default 333)
`rightdelay` Delay for right channel, in milliseconds. (range: 1 to 2000, default 333)
`pandelay` Whether to swap left and right delays with each successive echo. (0 = don't swap, 1 = swap, default 0)

sound_effect_flanger(snd,wetdry,depth,feedback,frequency,wave,delay,phase) Sets the parameters for the flanger effect for the indicated sound. The following parameters can be set:

`wetdry` Ratio of wet (processed) signal to dry (unprocessed) signal. (range: 0 to 100, default 50)
`depth` Percentage by which the delay time is modulated by the low-frequency oscillator, in hundredths of a percentage point. (range: 0 to 100, default 25)
`feedback` Percentage of output signal to feed back into the effect's input. (range: -99 to 99, default 0)
`frequency` Frequency of the LFO. (range: 0 to 10, default 0)
`wave` Waveform of the LFO. (0 = triangle, 1 = wave, default 1)
`delay` Number of milliseconds the input is delayed before it is played back. (range: 0 to 20, default 0)

`phase` Phase differential between left and right LFOs. (range: 0 to 4, default 2)

`sound_effect_gargle(snd,rate,wave)` Sets the parameters for the gargle effect for the indicated sound. The following parameters can be set:

`rate` Rate of modulation, in Hertz. (range: 1 to 1000, default 1)
`wave` Shape of the modulation wave. (0 = triangle, 1 = square, default 0)

`sound_effect_reverb(snd,gain,mix,time,ratio)` Sets the parameters for the reverb effect for the indicated sound. The following parameters can be set:

`gain` Input gain of signal, in decibels (dB). (range: -96 to 0, default 0)
`mix` Reverb mix, in dB. (range: -96 to 0, default 0)
`time` Reverb time, in milliseconds. (range: 0.001 to 3000, default 1000)
`ratio` Frequency ratio. (range: 0.001 to 0.999, default 0.001)

`sound_effect_compressor(snd,gain,attack,release,threshold,ratio,delay)` Sets the parameters for the compressor effect for the indicated sound. The following parameters can be set:

`gain` Output gain of signal after compression. (range: -60 to 60, default 0)
`attack` Time before compression reaches its full value. (range: 0.01 to 500, default 0.01)
`release` Speed at which compression is stopped after input drops below threshold. (range: 50 to 3000,

default 50)
`threshold` Point at which compression begins, in decibels. (range: -60 to 0, default -10)
`ratio` Compression ratio. (range: 1 to 100, default 10)
`delay` Time after lThreshold is reached before attack phase is started, in milliseconds. (range: 0 to 4, default 0)

`sound_effect_equalizer(snd,center,bandwidth,gain)` Sets the parameters for the equalizer effect for the indicated sound. The following parameters can be set:

`center` Center frequency, in hertz. (range: 80 to 16000)
`bandwidth` Bandwidth, in semitones.(range: 1 to 36)
`gain` Gain. (range: -15 to 15)

3D sound

This functionality is only available in the Pro Edition of Game Maker.

3D sounds refers to sound that has a position (and velocity) with respect to the listener. Although its most prominent use is in 3D games you can also effectively use it in 2D games. The idea is that a sound has a position in space. In all functions the listener is assumed to be at position (0,0,0). The system calculates how the listener would hear the sound and adapt it accordingly. The effect is especially good when you have a good speaker system but is already works on small speakers.

Besides a position, the sound can also has a velocity. This leads to well-known doppler effects which are correctly modelled. Finally the sound can have an orientation and, again, the sound is adapted accordingly.

Game Maker supports 3D sound options through the functions below. They only work for sound resources that were indicated to be 3D. (The disadvantage is that 3D sounds will be mono, not stereo.)

`sound_3d_set_sound_position(snd,x,y,z)` Sets the position of the indicated sound with respect to the listener to the indicated position in space. Values on the x-axis increase from left to right, on the y-axis from down to up, and on the z-axis from near to far. The values are measured in meters. The volume with which the sound is heard depends on this measure in the same way as in the real world.

`sound_3d_set_sound_velocity(snd,x,y,z)` Sets the velocity of

the indicated sound to the indicated vector in space. Please note that setting the velocity does not mean that the position changes. The velocity is only used for calculating doppler effects. So if you want to move the sound you must yourself change the position of the sound.

sound_3d_set_sound_distance(snd,mindist,maxdist) Sets the minimum distance at which the sound does no longer increase in loudness and the maximum distance at which the sound can no longer be heard. So when the distance lies between 0 and the minimum distance the sound is at maximal amplitude. When between the minimal distance and the maximal distance the amplitude slowly decreases until either the maximal distance is reached or the sound is anyway no longer audible. Default the minimum distance is 1 meter and the maximal distance is 1 billion meters.

sound_3d_set_sound_cone(snd,x,y,z,anglein,angleout,voloutside) Normally sound has the same amplitude at a given distance in all directions. You can set the sound cone to change this and make sound directional. x,y,z specify the direction of the sound cone. `anglein` specifies the inside angle. If the listener is inside this angle it hears the sound at its normal volume. `angleout` specifies the outside angle. When the listener is outside this the volume is indicated with `voloutside`. To be precise, `voloutside` is a negative number that indicates the number of hundreds of decibel that must be subtracted from the inside volume. Between the inside and outside angle the volume gradually decreases.

CD music

This functionality is only available in the Pro Edition of Game Maker.

There are also a number of functions dealing with playing music from a CD.

`cd_init()` Must be called before using the other functions. Should also be called when a CD is changed (or simply from time to time).
`cd_present()` Returns whether a CD is present in the default CD drive.
`cd_number()` Returns the number of tracks on the CD.
`cd_playing()` Returns whether the CD is playing.
`cd_paused()` Returns whether the CD is paused or stopped.
`cd_track()` Returns the number of the current track (1=the first).
`cd_length()` Returns the length of the total CD in milliseconds.
`cd_track_length(n)` Returns the length of track n of the CD in milliseconds.
`cd_position()` Returns the current position on the CD in milliseconds.
`cd_track_position()` Returns the current position in the track being played in milliseconds.
`cd_play(first,last)` Tells the CD to play tracks first until last. If you want to play the full CD give 1 and 1000 as arguments.
`cd_stop()` Stops playing.
`cd_pause()` Pauses the playing.
`cd_resume()` Resumes the playing.

`cd_set_position(pos)` Sets the position on the CD in milliseconds.

`cd_set_track_position(pos)` Sets the position in the current track in milliseconds.

`cd_open_door()` Opens the door of the CD player.

`cd_close_door()` Closes the door of the CD player.

There is one very general function to access the multimedia functionality of windows.

`MCI_command(str)` This functions sends the command string to the Windows multimedia system using the Media Control Interface (MCI). It returns the return string. You can use this to control all sorts of multimedia devices. See the Windows documentation for information in how to use this command. For example `MCI_command('play cdaudio from 1')` plays a cd (after you have correctly initialized it using other commands). This function is only for advanced use!

Splash screens, highscores, and other pop-ups

In this section we will describe a number of functions that can be used to display splash screens with videos, images, etc., to display messages and ask questions to the player, and to show the highscore list.

Information on splash screen and messages can be found in the following pages:

Splash Screens Popup Messages and Questions
Highscore List

Splash screens

This functionality is only available in the Pro Edition of Game Maker.

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

Default these splash screens are shown inside the game window, but it is also possible to show them in a separate window. The game is interrupted while the splash screen is shown. The player can return to the game by pressing the escape key or by clicking with the mouse in the window. (These settings can be changed; see below.)

The following functions can be used to display the splash screens:

`splash_show_video(fname,loop)` Shows a video splash screen. fname is the name of the video file. Whether a particular movie file is supported depends on the drivers on the machine. Typically you can use .avi, .mpg, and .wmv files but avoid special codecs. You best put this file in the folder of the game yourself or in a subfolder. loop indicates whether to loop the video.

`splash_show_text(fname,delay)` Shows a text splash screen. fname is the name of the text file. You can either display standard text files (.txt) or rich text files (.rtf). Rich text files can e.g. contain images. delay indicates the delay in milliseconds before returning to the game.

use 0 to wait until the player presses the escape key or clicks with the mouse in the window.

splash_show_image(fname,delay) Shows an image splash screen. fname is the name of the image file. Many image types are supported (for example .bmp, .jpg, .tif, and .wmf) but no animated images. delay is the delay in milli seconds before returning to the game.

You can change the way the splash screens are displayed using the functions below:

splash_set_main(main) Indicated whether the splash screen must be shown in the main game window (true, default) or in a separate window (false).

splash_set_scale(scale) Sets the scale factor to be used when displaying a splash video or image. When using a value of 0 the scale factor is chosen such that the window is filled (default).

splash_set_cursor(vis) Sets whether the cursor should be visible in the splash screen. Default it is visible. For movies the cursor cannot be switched off.

splash_set_color(col) Sets the color of the area surrounding the image or video.

splash_set_caption(cap) Sets the caption for the splash window. This only has effect when is a separate splash window is used. Default the empty string is used.

splash_set_fullscreen(full) Indicates whether to use a full screen window or not. This only has effect when is a separate splash window is used. Default a normal window is used.

splash_set_border(border) Indicates whether the window should have a border. This only has effect when is a separate normal splash window is used. Default a border is used.

splash_set_size(w,h) Sets the size of the splash window.

This only has effect when is a separate normal splash window is used. Default size is 640x480.

splash_set_adapt(adapt) Indicated whether the size of the window must be adapted to the scaled size of the video or image. This only has effect when is a separate splash window is used. Default adapt is true.

splash_set_top(top) Indicates whether the window should stay on top of other windows. This only has effect when is a separate splash window is used. Default the value is true.

splash_set_interrupt(interrupt) Indicates whether the game play should be interrupted while showing the splash window. This only has effect when is a separate splash window is used. Default the value is true.

splash_set_stop_key(stop) Indicates whether to stop the display of the splash screen when the player pressed the Escape key. Default the value is true.

splash_set_stop_mouse(stop) Indicates whether to stop the display of the splash screen when the player pressed the mouse inside the splash screen. Default the value is true.

There is one particular type of splash info, which is the game information that the user can enter in *Game Maker*. You can display it using the following function. You can also load a separate info file. This is closely related to the displaying of the text splash screen but the display is governed by the settings provided when defining the game information and not by the settings above. It is also displayed in a different window, so it is possible to display both the game information and a splash screen at the same moment. These functions also work in the Lite Edition.

show_info() Displays the game information window.

load_info(fname) Loads the game information from the

file named fname. This should be a rich text file (.rtf). This makes it possible to show different help files at different moments. Note that contrary to the spash screens, this rtf file cannot contain images.

Pop-up messages and questions

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

show_message(str) Displays a dialog box with the string as a message.

show_message_ext(str,but1,but2,but3) Displays a dialog box with the string as a message and up to three buttons. But1, but2 and but3 contain the button text. An empty string means that the button is not shown. In the texts you can use the & symbol to indicate that the next character should be used as the keyboard shortcut for this button. The function returns the number of the button pressed (0 if the user presses the Esc key).

show_question(str) Displays a question; returns true when the user selects yes and false otherwise.

get_integer(str,def) Asks the player in a dialog box for a number. str is the message. def is the default number shown.

get_string(str,def) Asks the player in a dialog box for a string. str is the message. def is the default value shown.

message_background(back) Sets the background image for the pop-up box for any of the functions above. back must be one of the backgrounds defined in the game. If back is partially transparent so is the message image (only for Windows 2000 or later).

message_alpha(alpha) Sets the alpha translucence for the pop-up box for any of the functions above. alpha must lie between 0 (completely translucent) and 1 (not

translucent) (only for Windows 2000 or later).
message_button(spr) Sets the sprite used for the buttons in the pop-up box. spr must be a sprite consisting of three images, the first indicates the button when it is not pressed and the mouse is far away, the second indicates the button when the mouse is above it but not pressed and the third is the button when it is pressed.
message_text_font(name,size,color,style) Sets the font for the text in the pop-up box. (This is a normal Windows font, not one of the font resources you can out in your game!) style indicates the font style (0=normal, 1=bold, 2=italic, and 3=bold-italic).
message_button_font(name,size,color,style) Sets the font for the buttons in the pop-up box. style indicates the font style (0=normal, 1=bold, 2=italic, and 3=bold-italic).
message_input_font(name,size,color,style) Sets the font for the input field in the pop-up box. style indicates the font style (0=normal, 1=bold, 2=italic, and 3=bold-italic).
message_mouse_color(col) Sets the color of the font for the buttons in the pop-up box when the mouse is above it.
message_input_color(col) Sets the color for the background of the input filed in the pop-up box.
message_caption(show,str) Sets the caption for the pop-up box. show indicates whether a border must b e shown (1) or not (0) and str indicates the caption when the border is shown.
message_position(x,y) Sets the position of the pop-up box on the screen. Use -1, -1 to center the box.
message_size(w,h) Fixes the size of the pop-up box on the screen. If you choose -1 for the width the width of the image is used. If you choose -1 for the height the height is calculated based on the number of lines in the message.
show_menu(str,def) Shows a popup menu. str indicates

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

`show_menu_pos(x,y,str,def)` Shows a popup menu as in the previous function but at position x,y on the screen.

`get_color(defcol)` Asks the player for a color. defcol is the default color. If the user presses Cancel the value -1 is returned.

`get_open_filename(filter,fname)` Asks the player for a filename to open with the given filter. The filter has the form 'name1|mask1|name2|mask2|...'. A mask contains the different options with a semicolon between them. * means any string. For example: 'bitmaps|*.bmp;*.wmf'. If the user presses Cancel an empy string is returned.

`get_save_filename(filter,fname)` Asks for a filename to save with the given filter. If the user presses Cancel an empy string is returned.

`get_directory(dname)` Asks for a directory. dname is the default name. If the user presses Cancel an empy string is returned.

`get_directory_alt(capt,root)` An alternative way to ask for a directory. capt is the caption to be show. root is the root of the directory tree to be shown. Use the empty string to show the whole tree. If the user presses Cancel an empy string is returned.

`show_error(str,abort)` Displays a standard error message (and/or writes it to the log file). abort indicates whether the game should abort.

Highscore list

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

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

highscore_set_background(back) Sets the background image to use. back must be the index of one of the background resources.

highscore_set_border(show) Sets whether the highscore form must have a border or not.

highscore_set_font(name,size,style) Sets the font used for the text in the table. (This is a normal Windows font, not one of the font resources.) You specify the name, size and style (0=normal, 1= bold, 2=italic, 3=bold-italic).

highscore_set_colors(back,new,other) Sets the colors used for the background, the new entry in the table, and the other entries.

highscore_set_strings(caption,nobody,escape) Changes the different default strings used when showing the highscore table. caption is the caption of the form. nobody is the string used when there is nobody at the particular rank. escape is the string at the bottom indicating to press the escape key. You can in particular use this when your game should use a different language.

highscore_show_ext(numb,back,border,col1,col2,name,size) Shows the highscore table with a number of options (can also be achieved by using a number of the previous functions). numb is the new score. If this score

is good enough to be added to the list, the player can input a name. Use -1 to simple display the current list. back is the background image to use, border indicates whether or not to show the border. col1 is the color for the new entry, col2 the color for the other entries. name is the name of the font to use, and size is the font size.

highscore_clear() Clears the highscore list.

highscore_add(str,numb) Adds a player with name str and score numb to the list.

highscore_add_current() Adds the current score to the highscore list. The player is asked to provide a name.

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

highscore_name(place) Returns the name of the person on the given place (1-10).

draw_highscore(x1,y1,x2,y2) Draws the highscore table in the room in the indicated box, using the current font.

Resources

In *Game Maker* you can define various types of resources, like sprites, sounds, fonts, objects, etc. In this chapter you will find a number of functions that give information about the resources. In the next chapter you find information on how to modify and create resources on the fly.

Information on resources can be found in the following pages:

Sprites Sounds
Backgrounds
Fonts
Paths
Scripts
Time lines
Objects
Rooms

Sprites

The following functions will give you information about a sprite:

sprite_exists(ind) Returns whether a sprite with the given index exists.
sprite_get_name(ind) Returns the name of the sprite with the given index.
sprite_get_number(ind) Returns the number of subimages of the sprite with the given index.
sprite_get_width(ind) Returns the width of the sprite with the given index.
sprite_get_height(ind) Returns the height of the sprite with the given index.
sprite_get_transparent(ind) Returns whether the sprite with the given index is transparent.
sprite_get_smooth(ind) Returns whether the sprite with the given index has smoothed edges.
sprite_get_preload(ind) Returns whether the sprite with the given index must be preloaded.
sprite_get_xoffset(ind) Returns the x-offset of the sprite with the given index.
sprite_get_yoffset(ind) Returns the y-offset of the sprite with the given index.
sprite_get_bbox_left(ind) Returns the left side of the bounding box of the sprite with the given index.
sprite_get_bbox_right(ind) Returns the right side of the bounding box of the sprite with the given index.
sprite_get_bbox_top(ind) Returns the top side of the bounding box of the sprite with the given index.
sprite_get_bbox_bottom(ind) Returns the bottom side of

the bounding box of the sprite with the given index.
sprite_get_bbox_mode(ind) Returns the bounding box mode (0=automatic, 1=full image, 2=manual) of the sprite with the given index.
sprite_get_precise(ind) Returns whether the sprite with the given index uses precise collision checking.

In certain situations you might want to save the bitmap corresponding to a particular subimage of the sprite to a file. For this the following function can be used:

sprite_save(ind,subimg,fname) Saves subimage subimg of sprite ind to the file with the name fname. This must be a .bmp file. ***Only available in the Pro Edition.***

Sounds

The following functions will give you information about a sound:

sound_exists(ind) Returns whether a sound with the given index exists.
sound_get_name(ind) Returns the name of the sound with the given index.
sound_get_kind(ind) Returns the kind of the sound with the given index (0=normal, 1=background, 2=3d, 3=mmplayer).
sound_get_preload(ind) Returns whether the sound with the given index has preload set.

Sounds use many resources and most systems can store and play only a limited number of sounds. If you make a large game you would like to have more control over which sounds are loaded in audio memory at what times. You can use the switch off the preload option for sounds to make sure sounds are only loaded when used. This though has the problem that you might get a small hiccup when the sound is used first. Also, sounds are not automatically unloaded when you don't need them anymore. For more control you can use the following functions.

sound_discard(index) Frees the audio memory used for the indicated sound.
sound_restore(index) Restores the indicated sound in audio memory for immediate playing.

Backgrounds

The following functions will give you information about a background:

background_exists(ind) Returns whether a background with the given index exists.
background_get_name(ind) Returns the name of the background with the given index.
background_get_width(ind) Returns the width of the background with the given index.
background_get_height(ind) Returns the height of the background with the given index.
background_get_transparent(ind) Returns whether the background with the given index is transparent.
background_get_smooth(ind) Returns whether the background with the given index has smoothed edges.
background_get_preload(ind) Returns whether the background with the given index must be preloaded.

In certain situations you might want to save the bitmap corresponding the background to a file. For this the following function can be used:

background_save(ind,fname) Saves the background ind to the file with the name fname. This must be a .bmp file. ***Only available in the Pro Edition.***

Fonts

The following functions will give you information about a font:

`font_exists(ind)` Returns whether a font with the given index exists.
`font_get_name(ind)` Returns the name of the font with the given index.
`font_get_fontname(ind)` Returns the fontname of the font with the given index.
`font_get_bold(ind)` Returns whether the font with the given index is bold.
`font_get_italic(ind)` Returns whether the font with the given index is italic.
`font_get_first(ind)` Returns the index of the first character in the font with the given index.
`font_get_last(ind)` Returns the index of the last character in the font with the given index.

Paths

The following functions will give you information about a path:

path_exists(ind) Returns whether a path with the given index exists.
path_get_name(ind) Returns the name of the path with the given index.
path_get_length(ind) Returns the length of the path with the given index.
path_get_kind(ind) Returns the kind of connections of the path with the given index (0=straight, 1=smooth).
path_get_closed(ind) Returns whether the path is closed or not.
path_get_precision(ind) Returns the precision used for creating smoothed paths.
path_get_number(ind) Returns the number of defining points for the path.
path_get_point_x(ind,n) Returns the x-coordinate of the n'th defining point for the path. 0 is the first point.
path_get_point_y(ind,n) Returns the y-coordinate of the n'th defining point for the path. 0 is the first point.
path_get_point_speed(ind,n) Returns the speed factor at the n'th defining point for the path. 0 is the first point.
path_get_x(ind,pos) Returns the x-coordinate at position pos for the path. pos must lie between 0 and 1.
path_get_y(ind,pos) Returns the y-coordinate at position pos for the path. pos must lie between 0 and 1.
path_get_speed(ind,pos) Returns the speed factor at position pos for the path. pos must lie between 0 and 1.

Scripts

The following functions will give you information about a script:

`script_exists(ind)` Returns whether a script with the given index exists.
`script_get_name(ind)` Returns the name of the script with the given index.
`script_get_text(ind)` Returns the text string of the script with the given index.

Time lines

The following functions will give you information about a timeline:

`timeline_exists(ind)` Returns whether a time line with the given index exists.

`timeline_get_name(ind)` Returns the name of the time line with the given index.

Objects

The following functions will give you information about an object:

`object_exists(ind)` Returns whether an object with the given index exists.
`object_get_name(ind)` Returns the name of the object with the given index.
`object_get_sprite(ind)` Returns the index of the default sprite of the object with the given index.
`object_get_solid(ind)` Returns whether the object with the given index is default solid.
`object_get_visible(ind)` Returns whether the object with the given index is default visible.
`object_get_depth(ind)` Returns the depth of the object with the given index.
`object_get_persistent(ind)` Returns whether the object with the given index is persistent.
`object_get_mask(ind)` Returns the index of the mask of the object with the given index (-1 if is has no special mask).
`object_get_parent(ind)` Returns index of the parent object of object ind (a negative value is returned if it has no parent).
`object_is_ancestor(ind1,ind2)` Returns whether object `ind2` is an ancestor of object `ind1`.

Rooms

The following functions will give you information about a room:

`room_exists(ind)` Returns whether a room with the given index exists.

`room_get_name(ind)` Returns the name of the room with the given index.

Note that, because rooms change during the playing of the room, there are other routines to get information about the contents of the current room.

Changing resources

This functionality is only available in the Pro Edition of Game Maker.

It is possible to create new resources during the game. Also you can change existing resources. This chapter describes the possibilities. Be warned though. **Changing resources easily leads to serious errors in your games!!!** You must follow the following rules when changing resources:

- Don't change resources that are being used. This will lead to errors! For example, don't change a sprite that is being used by an instance.
- When you save the game during playing, added and changed resources are NOT stored with the save game. So if you load the saved game later, these might not be there anymore. In general, when you manipulate resources you can no longer use the built-in system for loading and saving games.
- When you restart the game during playing, the changed resources are NOT restored to their original shape. In general, when you manipulate resources you can no longer use the action or function to restart the game.
- Resource manipulation can be slow. For example, changing sprites or backgrounds is relatively slow. So don't use it during the game play.
- Creating resources during game play (in particular sprites and background) easily uses huge amount of memory. So be extremely careful with this. For example, if you have a 32 frame 128x128 animated sprite and you decide to create 36 rotated copies of it you will use up 36x32x128x128x4 = 72 MB of memory!

- Make sure you delete resources you no longer need. Otherwise the system soon runs out of memory.

In general, you should not change any resources during game play. Better create and change the resources at the beginning of the game or maybe at the beginning of a room.

Information on changing resources can be found in the following pages:

Sprites Sounds
Backgrounds
Fonts
Paths
Scripts
Time lines
Objects
Rooms

Sprites

The following routines are available for changing sprite properties:

sprite_set_offset(ind,xoff,yoff) Sets the offset of the sprite with the given index.
sprite_set_bbox_mode(ind,mode) Sets the bounding box mode of the sprite (0=automatic, 1=full image, 2=manual).
sprite_set_bbox(ind,left,top,right,bottom) Sets the bounding box of the sprite with the given index. Works only when the bounding box mode is manual.
sprite_set_precise(ind,mode) Sets whether the sprite with the given index uses precise collision checking (true or false).

The following routines can be used to create new sprites and to remove them.

sprite_duplicate(ind) Creates a duplicate of the sprite with the given index. It returns the index of the new sprite. When an error occurs -1 is returned.
sprite_assign(ind,spr) Assigns the indicated sprite to sprite ind. So this makes a copy of the sprite. In this way you can easily set an existing sprite to a different, e.g. new sprite.
sprite_merge(ind1,ind2) Merges the images from sprite ind2 into sprite ind1, adding them at the end. If the sizes don't match the sprites are stretched to fit. Sprite ind2 is not deleted!
sprite_add(fname,imgnumb,precise,transparent,smooth,preload,xorig,yorig) Adds the image stored in the file fname to

the set of sprite resources. Many different image file types can be dealt with. When the image is not a gif image it can be a strip containing a number of subimages for the sprite next to each other. Use imgnumb to indicate their number (1 for a single image). For (animated) gif images, this argument is not used; the number of images in the gif file is used. precise indicates whether precise collision checking should be used. `transparent` indicates whether the image is partially transparent. `smooth` indicates whether to smooth the edges. `preload` indicates whether to preload the image into texture memory. `xorig` and `yorig` indicate the position of the origin in the sprite. The function returns the index of the new sprite that you can then use to draw it or to assign it to the variable sprite_index of an instance. When an error occurs -1 is returned.

`sprite_add_alpha(fname,imgnumb,precise,preload,xorig,yorig)` Adds the image stored in the file fname to the set of sprite resources, but this time the file has an alpha channel to indicate transparency (as for example in .png files). The arguments are the same as above (but two are missing as they are not relevant in this case). When an error occurs -1 is returned.

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

`sprite_replace_alpha(ind,fname,imgnumb,precise,preload,xorig,yorig)` Same as above but in this case the file has an alpha channel. The function returns whether it is successful.

`sprite_create_from_screen(x,y,w,h,precise,transparent,smooth,preload,xorig,yorig)` Creates a sprite by copying the given area from the screen. This makes it possible to

create any sprite you want. Draw the image on the screen using the drawing functions and next create a sprite from it. (If you don't do this in the drawing event you can even do it in such a way that it is not visible on the screen by not refreshing the screen.) The other parameters are as above. The function returns the index of the new sprite. A work of caution is required here. Even though we speak about the screen, it is actually the drawing region that matters. The fact that there is a window on the screen and that the image might be scaled in this window does not matter.

sprite_add_from_screen(ind,x,y,w,h) Adds an area of the screen as a next subimage to the sprite with index ind.

sprite_create_from_surface(id,x,y,w,h,precise,transparent,smooth,preload,xorig,yorig) Creates a sprite by copying the given area from the surface with the given id. This makes it possible to create any sprite you want. Draw the image on the surface using the drawing functions and next create a sprite from it. The function returns the index of the new sprite. Note that alpha values are maintained in the sprite.

sprite_add_from_surface(ind,id,x,y,w,h) Adds an area of the surface id as a next subimage to the sprite with index ind.

sprite_delete(ind) Deletes the sprite from memory, freeing the memory used.

The following routine exists to change the appearance of a sprite.

sprite_set_alpha_from_sprite(ind,spr) Changes the alpha (transparancy) values in the sprite with index `ind` using the intensity values in the sprite `spr`. This cannot be undone.

Sounds

The following routines can be used to create new sounds and to remove them.

sound_add(fname,kind,preload) Adds a sound resource to the game. fname is the name of the sound file. kind indicates the kind of sound (0=normal, 1=background, 2=3d, 3=mmplayer) preload indicates whether the sound should immediately be stored in audio memory (true or false). The function returns the index of the new sound, which can be used to play the sound. (-1 if an error occurred, e.g. the file does not exist).

sound_replace(index,fname,kind,preload) Same as the previous function but this time a new sound is not created but the existing sound index is replaced, freeing the old sound. Returns whether correct.

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

Backgrounds

The following routines can be used to create new backgrounds and to remove them.

background_duplicate(ind) Creates a duplicate of the background with the given index. It returns the index of the new background. When an error occurs -1 is returned.

background_assign(ind,back) Assigns the indicated background to background ind. So this makes a copy of the background.

background_add(fname,transparent,smooth,preload) Adds the image stored in the file fname to the set of background resources. Many different types of images can be dealt with. `transparent` indicates whether the image is partially transparent. `smooth` indicates whether to smooth the edges. `preload` indicates whether to preload the image into texture memory. The function returns the index of the new background that you can then use to draw it or to assign it to the variable background_index[0] to make it visible in the current room. When an error occurs -1 is returned.

background_add_alpha(fname,preload) Adds the image stored in the file fname to the set of background resources, but this time the file has an alpha channel to indicate transparency (as for example in .png files). The arguments are the same as above (but two are missing as they are not relevant in this case). When an error occurs -1 is returned.

background_replace(ind,fname,transparent,smooth,preload) Same as above but in this case the background with index ind is replaced. The function returns whether it is

successful. When the background is currently visible in the room it wil be replaced also.

background_replace_alpha(ind,fname,preload) Same as above but in this case the file has an alpha channel.

background_create_color(w,h,col,preload) Creates a background of the given size and with the given color. It returns the index of the new background. When an error occurs -1 is returned.

background_create_gradient(w,h,col1,col2,kind,preload) Creates a gradient filled background of the given size. col1 and col2 indicate the two colors. kind is a number between 0 and 5 indicating the kind of gradient: 0=horizontal 1=vertical, 2= rectangle, 3=ellipse, 4=double horizontal, 5=double vertical. It returns the index of the new background. When an error occurs -1 is returned.

background_create_from_screen(x,y,w,h,transparent,smooth,preload) Creates a background by copying the given area from the screen. This makes it possible to create any background you want. Draw the image on the screen using the drawing functions and next create a background from it. (If you don't do this in the drawing event you can even do it in such a way that it is not visible on the screen by not refreshing the screen.) The other parameters are as above. The function returns the index of the new background. A work of caution is required here. Even though we speak about the screen, it is actually the drawing region that matters. The fact that there is a window on the screen and that the image might be scaled in this window does not matter.

background_create_from_surface(id,x,y,w,h,transparent,smooth,preload) Creates a background by copying the given area from the surface with the given id. This makes it possible to create any background you want. Draw the image on the surface using the drawing functions and

next create a background from it. Note that alpha values are maintained the background.

`background_delete(ind)` Deletes the background from memory, freeing the memory used.

The following routine exists to change the appearance of a background.

`background_set_alpha_from_background(ind,back)` Changes the alpha (transparancy) values in the background with index `ind` using the intensity values in the background `back`. This cannot be undone.

Fonts

It is possible to create, replace, and delete fonts during the game using the following functions. (Don't replace a font that is set as the current font or at least set it again afterwards.)

font_add(name,size,bold,italic,first,last) Adds a new font and returns its index, indicating the name, size, whether it is bold or italic, and the first and last character that must be created.

font_add_sprite(spr,first,prop,sep) Adds a new font and returns its index. The font is created from a sprite. The sprite should contain a subimage for each character. `first` indicate the index of the first character in the sprite. For example, use `ord('0')` if your sprite only contains the digits. `prop` indicates whether the font is proportional. In a proportional font, for each character the width of the bounding box is used as the character width. Finally, `sep` indicates the amount of white space that must separate the characters horizontally. A typical value would lie between 2 and 8 depending on the font size.

font_replace(ind,name,size,bold,italic,first,last) Replaces the font `ind` with a new font, indicating the name, size, whether it is bold or italic, and the first and last character that must be created.

font_replace_sprite(ind,spr,first,prop,sep) Replaces the font `ind` with a new sprite-based font.

font_delete(ind) Deletes the font with the given index, freeing the memory it uses.

Paths

It is possible to create paths and to add points to paths. However, never change a path that is being used by an instance. This can lead to unexpected results. The following functions exist:

`path_set_kind(ind,val)` Sets the kind of connections of the path with the given index (0=straight, 1=smooth).
`path_set_closed(ind,closed)` Sets whether the path must be closed (true) or open (false).
`path_set_precision(ind,prec)` Sets the precision with which the smooth path is calculated (should lie between 1 and 8).
`path_add()` Adds a new empty paths. The index of the path is returned.
`path_delete(ind)` Deletes the path with the given index.
`path_duplicate(ind)` Creates a duplicate copy of the path with the given index. Returns the index of the copy.
`path_assign(ind,path)` Assigns the indicated path to path ind. So this makes a copy of the path. In this way you can easily set an existing path to a different, e.g. new path.
`path_append(ind,path)` Appends the indicated path to path ind.
`path_add_point(ind,x,y,speed)` Adds a point to the path with the given index, at position (x,y) and with the given speed factor. Remember that a factor of 100 corresponds to the actual speed. Lower values mean slowing down and higher mean speeding up.
`path_insert_point(ind,n,x,y,speed)` Inserts a point in the path with the given index before point n, at position (x,y) and with the given speed factor.

path_change_point(ind,n,x,y,speed) Changes the point n in the path with the given index to position (x,y) and the given speed factor.
path_delete_point(ind,n) Deletes the point n in the path with the given index.
path_clear_points(ind) Clears all the points in the path, turning it into an empty path.
path_reverse(ind) Reverses the path.
path_mirror(ind) Mirrors the path horizontally (with respect to its center).
path_flip(ind) Flips the path vertically (with respect to its center).
path_rotate(ind,angle) Rotates the path counter clockwise over angle degrees (around its center).
path_scale(ind,xscale,yscale) Scales the path with the given factors (from its center).
path_shift(ind,xshift,yshift) Shifts the path over the given amount.

Scripts

Scripts cannot be changed during the execution of the game. The scripts are part of the game logic. Changing scripts would lead to self-rewriting code which very easily leads to errors. Also there are other ways to achieve this. If you really need to execute a piece of code that is not known at design time (e.g. from a file) you can use the following functions:

`execute_string(str,arg0,arg1,...)` Execute the piece of code in the string str with the indicated arguments.
`execute_file(fname,arg0,arg1,...)` Execute the piece of code in the file with the indicated arguments.

Sometimes you want to store a script index in a variable and execute it. For this you can use the following function

`script_execute(scr,arg0,arg1,...)` Execute the script with index scr with the given arguments.

Time lines

The following routines are available for creating and changing time lines. Don't change time lines that are in use!

`timeline_add()` Adds a new time line. It returns the index of the time line.

`timeline_delete(ind)` Deletes the time line with the given index. Make sure no instances uses the time line in any room.

`timeline_moment_add(ind,step,codestr)` Adds a code action to the time line at moment step. codestr contains the code for the actions. If the step does not exist it is created. So you can add multiple code actions for the same moment.

`timeline_moment_clear(ind,step)` You can use this function to clear all the actions for a particular moment.

Objects

Also objects can be manipulated and created during the game play. NEVER change or delete an object for which there are instances. This can lead to unexpected effects as certain object properties are stored with the instance and, hence, changing them in the object will not have the desired effect.

`object_set_sprite(ind,spr)` Sets the sprite of the object with the given index. Use -1 to remove the current sprite from the object.

`object_set_solid(ind,solid)` Sets whether instances created of the object must default be solid (true or false).

`object_set_visible(ind,vis)` Sets whether instances created of the object must default be visible (true or false).

`object_set_depth(ind,depth)` Sets the default depth of instances created of the object.

`object_set_persistent(ind,pers)` Sets whether instances created of the object must default be persistent (true or false).

`object_set_mask(ind,spr)` Sets the sprite mask of the object with the given index. Use -1 to set the mask to be the sprite of the object.

`object_set_parent(ind,obj)` Sets the parent of the object. Use -1 to not have a parent. Changing the parent changes the behavior of instances of the object.

The following routines are useful for creating objects on the fly. As with all resource changing routines, be very careful that you don't create new objects all the time.

`object_add()` Adds a new object. It returns the index of the object. You can now use this index in the routines above to set certain properties of the object and then you can use the index to create instances of the object.

`object_delete(ind)` Deletes the object with the given index. Make sure no instances of the object exist in any of the rooms.

`object_event_add(ind,evtype,evnumb,codestr)` To give the object a behavior we must define events for the object. You can only add code actions to events. You need to specify the object, the event type, the event number (use the constants that have been specified before for the event_perform() function). Finally you provide the code string that must be executed. You can add multiple code actions to each event.

`object_event_clear(ind,evtype,evnumb)` You can use this function to clear all the actions for a particular event.

Creating objects is in particular useful when you are designing scripts or action libraries. For example, an initialization script can create an object to display a text and another script can add such an object with a particular text. In this way you have a simple mechanism to display texts without the need to create objects using the standard interface.

Rooms

Manipulating rooms on the fly is a dangerous thing to do. You have to realize that rooms change all the time due to what is happening in the game. This normally only involves the currently active room and there are many routines described in previous sections to manipulate the instances, backgrounds, and tiles in the active room. But changes in the active room will last if the room is persistent. Hence, you should never manipulate aspects of the currently active room or any room that is persistent and that has already been visited before. Such changes in general won't be noticed but might sometimes even lead to unexpected errors. Due to the fact that rooms are linked in a complicated way there is also no routine to delete a room.

The following routines are available

`room_set_width(ind,w)` Sets the width for the room with the indicated index.

`room_set_height(ind,h)` Sets the height for the room with the indicated index.

`room_set_caption(ind,str)` Sets the caption for the room with the indicated index.

`room_set_persistent(ind,val)` Sets whether the room with the indicated index is persistent or not.

`room_set_code(ind,str)` Sets the initialization code string for the room with the indicated index.

`room_set_background_color(ind,col,show)` Sets the color properties for the room with the indicated index if is does not have a background image. col indicates the color and show indicates whether the color must be shown or not.

room_set_background(ind,bind,vis,fore,back,x,y,htiled,vtiled,hspeed,vspeed,alpha) Sets background with index bind (0-7) for the room with the indicated index. vis indicates whether the background is visible and fore whether it is actually a foreground. back is the index of the background image. x,y indicate the position of the image and htiled and vtiled indicate whether the image must be tiled. hspeed and vspeed indicate the speed with which the background moves and alpha indicates an alpha translucency value (1 = solid and fastest).
room_set_view(ind,vind,vis,xview,yview,wview,hview,xport,yport,wport,hport,hborder,vborder,hspeed,vspeed,obj) Sets the view with index vind (0-7) for the room with the indicated index. vis indicates whether the view is visible. xview, yview, wview, and hview indicate the position of the view in the room. xport, yport, wport, and hport indicate the position on the screen. When the view must follow an object hborder and vborder indicate the minimal visible border that must be kept around the object. hspeed and vspeed indicate the maximal speed with which the view can move. obj is the index of the object or the index of the instance.
room_set_view_enabled(ind,val) Sets whether views must be enabled for the room with the indicated index.
room_add() Adds a new room. It returns the index of the room. Note that the room will not be part of the room order. So the new room does not have a previous or a next room. If you want to move to an added room you must provide the index of the room.
room_duplicate(ind) Adds a copy of the room with the given index. It returns the index of the room.
room_assign(ind,room) Assigns the indicated room to room ind. So this makes a copy of the room.
room_instance_add(ind,x,y,obj) Adds a new instance of object obj to the room, placing it at the indicate

position. It returns the index of the instance.
room_instance_clear(ind) Removes all instances from the indicated room.
room_tile_add(ind,back,left,top,width,height,x,y,depth) Adds a new tile to the room at the indicate position. It returns the index of the tile. back is the background from which the tile is taken. left, top, width and height indicate the part of the background that forms the tile. x,y is the position of the tile in the room and depth is the depth of the tile.
room_tile_add_ext(ind,back,left,top,width,height,x,y,depth, xscale,yscale,alpha) Same as the previous routine but this time you can also specify a scaling factor in x and y direction and an alpha transparency for the tile.
room_tile_clear(ind) Removes all tiles from the indicated room.

Files, registry, and executing programs

In more advanced games you probably want to read data from a file that you provide with the game. Or you might want to store information between runs of the game. And in some situations you might need to execute external programs.

Information can be found in the following pages:

Files Registry
INI Files
Executing Programs

Files

It is useful to use external files in games. For example, you could make a file that describes at what moments certain things should happen. Also you probably want to save information for the next time the game is run (for example, the current room). The following functions exist to read and write data in text files:

`file_text_open_read(fname)` Opens the file with the indicated name for reading. The function returns the id of the file that must be used in the other functions. You can open multiple files at the same time (32 max). Don't forget to close them once you are finished with them.

`file_text_open_write(fname)` Opens the indicated file for writing, creating it if it does not exist. The function returns the id of the file that must be used in the other functions.

`file_text_open_append(fname)` Opens the indicated file for appending data at the end, creating it if it does not exist. The function returns the id of the file that must be used in the other functions.

`file_text_close(fileid)` Closes the file with the given file id.

`file_text_write_string(fileid,str)` Writes the string to the file with the given file id.

`file_text_write_real(fileid,x)` Write the real value to the file with the given file id.

`file_text_writeln(fileid)` Write a newline character to the file.

`file_text_read_string(fileid)` Reads a string from the file with the given file id and returns this string. A string

ends at the end of line.

file_text_read_real(fileid) Reads a real value from the file and returns this value.

file_text_readln(fileid) Skips the rest of the line in the file and starts at the start of the next line.

file_text_eof(fileid) Returns whether we reached the end of the file.

To manipulate files in the file system you can use the following functions:

file_exists(fname) Returns whether the file with the given name exists (true) or not (false).

file_delete(fname) Deletes the file with the given name.

file_rename(oldname,newname) Renames the file with name oldname into newname.

file_copy(fname,newname) Copies the file fname to the newname.

directory_exists(dname) Returns whether the indicated directory does exist. The name must include the full path, not a relative path.

directory_create(dname) Creates a directory with the given name (including the path towards it) if it does not exist. The name must include the full path, not a relative path.

file_find_first(mask,attr) Returns the name of the first file that satisfies the mask and the attributes. If no such file exists, the empty string is returned. The mask can contain a path and can contain wildchars, for example 'C:\temp*.doc'. The attributes give the additional files you want to see. (So the normal files are always returned when they satisfy the mask.) You can add up the following constants to see the type of files you want:

fa_readonly read-only files
fa_hidden hidden files
fa_sysfile system files
fa_volumeid volume-id files
fa_directory directories
fa_archive archived files

file_find_next() Returns the name of the next file that satisfies the previously given mask and the attributes. If no such file exists, the empty string is returned.
file_find_close() Must be called after handling all files to free memory.
file_attributes(fname,attr) Returns whether the file has all the attributes given in attr. Use a combination of the constants indicated above.

The following functions can be used to change file names. Note that these functions do not work on the actual files they only deal with the strings.

filename_name(fname) Returns the name part of the indicated file name, with the extension but without the path.
filename_path(fname) Returns the path part of the indicated file name, including the final backslash.
filename_dir(fname) Returns the directory part of the indicated file name, which normally is the same as the path except for the final backslash.
filename_drive(fname) Returns the drive information of the filename.
filename_ext(fname) Returns the extension part of the indicated file name, including the leading dot.
filename_change_ext(fname,newext) Returns the indicated file name, with the extension (including the dot)

changed to the new extension. By using an empty string as the new extension you can remove the extension.

In rare situations you might need to read data from binary files. The following low-level routines exist for this:

file_bin_open(fname,mod) Opens the file with the indicated name. The mode indicates what can be done with the file: 0 = reading, 1 = writing, 2 = both reading and writing). When the file does not exist it is created. The function returns the id of the file that must be used in the other functions. You can open multiple files at the same time (32 max). Don't forget to close them once you are finished with them.
file_bin_rewrite(fileid) Rewrites the file with the given file id, that is, clears it and starts writing at the start.
file_bin_close(fileid) Closes the file with the given file id.
file_bin_size(fileid) Returns the size (in bytes) of the file with the given file id.
file_bin_position(fileid) Returns the current position (in bytes; 0 is the first position) of the file with the given file id.
file_bin_seek(fileid,pos) Moves the current position of the file to the indicated position. To append to a file move the position to the size of the file before writing.
file_bin_write_byte(fileid,byte) Writes a byte of data to the file with the given file id.
file_bin_read_byte(fileid) Reads a byte of data from the file and returns this.

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

If you included files in the game executable and did not automatically export them at the start of the game, you can use the following functions to do this.

export_include_file(fname) Exports the included file with the name fname. This must be a string variable, so don't forget the quotes.
export_include_file_location(fname,location) Exports the included file with the name fname to the given location. Location must contain the path and the filename.
discard_include_file(fname) Discard the included file with the name fname, freeing the memory used. This must be a string variable, so don't forget the quotes.

The following four read-only variables can be useful:

game_id* Unique identifier for the game. You can use this if you need a unique file name.
working_directory* Working directory for the game. (Not including the final backslash.)
program_directory* Directory in which the game executable is stored. (Not including the final backslash.) When you run a standalone game this is normally the same as the working directory unless the game e.g. opens a file using the file selector. Note that when testing a game you are creating the program and working directory will be different. In that case the working directory is the place where the editable version is stored while the program directory is a temporary directory for testing.
temp_directory* Temporary directory created for the game. (Not including the final backslash.) You can store temporary files here. They will be removed at the end of the game.

In certain situations you might want to give players the possibility of providing command line arguments to the game they are running (for example to create cheats or special modes). To get these arguments you can use the following two routines.

`parameter_count()` Returns the number of command-line parameters. The actual parameters can be retrieved with the following function.

`parameter_string(n)` Returns command-line parameters n. The first parameter has index 1. The last one has index parameter_count(). Index 0 is a special one. It is the filename of the game executable (including the path).

You can read the value of environment variables using the following function:

`environment_get_variable(name)` Returns the value (a string) of the environment variable with the given name.

Registry

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

registry_write_string(name,str) Creates an entry in the registry with the given name and string value.
registry_write_real(name,x) Creates an entry in the registry with the given name and real value.
registry_read_string(name) Returns the string that the given name holds. (The name must exist. Otherwise an empty string is returned.)
registry_read_real(name) Returns the real value that the given name holds. (The name must exist. Otherwise the number 0 is returned.)
registry_exists(name) Returns whether the given name exists.

Actually, values in the registry are grouped into keys. The above routines all work on values within the key that is especially created for your game. Your program can use this to obtain certain information about the system the game is running on. You can also read values in other keys. You can write them also but be very careful. YOU CAN EASILY DESTROY YOUR SYSTEM this way. (Write is not allowed in secure mode.) Note that keys are again placed in groups. The following routines default work on the group HKEY_CURRENT_USER. But you can change the root group.

So, for example, if you want to find out the current temp dir, use

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

The following functions exist.

registry_write_string_ext(key,name,str) Creates an entry in the key in the registry with the given name and string value.
registry_write_real_ext(key,name,x) Creates an entry in the key in the registry with the given name and real value.
registry_read_string_ext(key,name) Returns the string that the given name in the indicated key holds. (The name must exist. Otherwise an empty string is returned.)
registry_read_real_ext(key,name) Returns the real value that the given name in the indicated key holds. (The name must exist. Otherwise the number 0 is returned.)
registry_exists_ext(key,name) Returns whether the given name exists in the given key.
registry_set_root(root) Sets the root for the other routines. Use the following values:

0 = HKEY_CURRENT_USER
1 = HKEY_LOCAL_MACHINE
2 = HKEY_CLASSES_ROOT
3 = HKEY_USERS

INI files

To pass certain parameter settings to programs a standard mechanism is the use of INI files. INI files contain sections and each section contains a number of name-value pairs. For example, here is a typical INI file:

```
[Form]
Top=100
Left=100
Caption=The best game ever
[Game]
MaxScore=12324
```

This file contains two sections, one call Form and the other called Game. The first section contains three pairs. The first two have a real value while the third has a string value. Such INI files are easy to create and change. The following functions exist in *Game Maker* to read and change the data in them.

`ini_open(name)` Opens the INI file with the given name. The ini file must be stored in the same folder as the game!

`ini_close()` Closes the currently open INI file.

`ini_read_string(section,key,default)` Reads the string value of the indicated key in the indicated section. When the key or section does not exist the default value is returned.

`ini_read_real(section,key,default)` Reads the real value of the indicated key in the indicated section. When the key or section does not exist the default value is returned.

`ini_write_string(section,key,value)` Writes the string value for the indicated key in the indicated section.

ini_write_real(section,key,value) Writes the real value for the indicated key in the indicated section.
ini_key_exists(section,key) Returns whether the indicated key exists in the indicated section.
ini_section_exists(section) Returns whether the indicated section exists.
ini_key_delete(section,key) Deletes the indicated key from the indicated section.
ini_section_delete(section) Deletes the indicated section.

Executing programs

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

`execute_program(prog,arg,wait)` Executes program prog with arguments arg. wait indicates whether to wait for finishing.

`execute_shell(prog,arg)` Executes the program (or file) in the shell.

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

`secure_mode*` Whether the game is running in secure mode.

Data structures

This functionality is only available in the Pro Edition of Game Maker.

In games you often need to store information. For example you need to store lists of items that a person carries or you want to store places that still need to be visited. You can use the arrays for this. But if you want to do more complicated operations, like sorting the data or searching for a particular item, you need to write large pieces of GML code which can be slow to execute.

To remedy this, *Game Maker* has a number of built-in data structures that can be accessed through functions. At the moment there are six different types of data structure available: stacks, queues, lists, maps, priority queues, and grids. Each of these data structures is tuned for a particular type of use (see below).

All data structures work globally in the same way. You can create a data structure with a function that returns an id for the structures. You use this id to perform operations on the data structures. Once you are done you destroy the data structure again to save storage. You can use as many of the structures at the same moment as you want. All structure can store both strings and real values.

Please note that data structures and their content are not saved when you save the game using the actions or functions for that. If you use data structures and want to allow for saves you have to create your own mechanims for that.

When comparing values, for example when searching in a map or sorting a list, *Game Maker* must decide when two values are equal. For strings and integer values this is clear but for real numbers, due to round-off errors, equal number can easily become unequal. For example (5/3)*3 will not be equal to 5. To avoid this, a precision is used. When the difference between two numbers is smaller than this precision they are considered equal. Default a precision of 0.0000001 is used. You can change this precision using the following functions:

`ds_set_precision(prec)` Sets the precision used for comparisons.

This precision is used in all data structures but not in other comparisons in GML!

Information on data structures can be found in the following pages:

Stacks
Queues
Lists
Maps
Priority Queues
Grids

Stacks

A stack data structure is a so called LIFO (Last-In First-Out) structures. You can push values on a stack and the remove them again by popping them from the stack. The value that was pushed on the stack most recently is the first to be popped from it again. Stacks are often used when there are interrupts to handle, or when having recursive functions. The following functions exist for stacks:

`ds_stack_create()` Creates a new stack. The function returns an integer as an id that must be used in all other functions to access the particular stack. You can create multiple stacks.
`ds_stack_destroy(id)` Destroys the stack with the given id, freeing the memory used. Don't forget to call this function when you are ready with the structure.
`ds_stack_clear(id)` Clears the stack with the given id, removing all data from it but not destroying it.
`ds_stack_copy(id,source)` Copies the stack source into the stack with the given id.
`ds_stack_size(id)` Returns the number of values stored in the stack.
`ds_stack_empty(id)` Returns whether the stack is empty. This is the same as testing whether the size is 0.
`ds_stack_push(id,val)` Pushes the value on the stack.
`ds_stack_pop(id)` Returns the value on the top of the stack and removes it from the stack.
`ds_stack_top(id)` Returns the value on the top of the stack but does not remove it from the stack.
`ds_stack_write(id)` Turns the data structure into a string and returns this string. The string can then be used to e.g. save it to a file. This provides an easy mechanism

for saving data structures.
`ds_stack_read(id,str)` Reads the data structure from the given string (as created by the previous call).

Queues

A queue is somewhat similar to a stack but it works on a FIFO (First-In First-Out) basis. The value that is put in the queue first is also the first to be removed from it. It works like a queue in a shop. The person who is first in a queue is served first. Queues are typically used to store things that still need to be done but there are many other uses. The following functions exist (note that the first five are equivalent to the functions for stacks; all data structures have these five functions).

`ds_queue_create()` Creates a new queue. The function returns an integer as an id that must be used in all other functions to access the particular queue. You can create multiple queues.
`ds_queue_destroy(id)` Destroys the queue with the given id, freeing the memory used. Don't forget to call this function when you are ready with the structure.
`ds_queue_clear(id)` Clears the queue with the given id, removing all data from it but not destroying it.
`ds_queue_copy(id,source)` Copies the queue source into the queue with the given id.
`ds_queue_size(id)` Returns the number of values stored in the queue.
`ds_queue_empty(id)` Returns whether the queue is empty. This is the same as testing whether the size is 0.
`ds_queue_enqueue(id,val)` Enters the value in the queue.
`ds_queue_dequeue(id)` Returns the value that is longest in the queue and removes it from the queue.
`ds_queue_head(id)` Returns the value at the head of the queue, that is, the value that has been the longest in the queue. (It does not remove it from the queue.)

`ds_queue_tail(id)` Returns the value at the tail of the queue, that is, the value that has most recently been added to the queue. (It does not remove it from the queue.)

`ds_queue_write(id)` Turns the data structure into a string and returns this string. The string can then be used to e.g. save it to a file. This provides an easy mechanism for saving data structures.

`ds_queue_read(id,str)` Reads the data structure from the given string (as created by the previous call).

Lists

A list stores a collection of values in a particular order. You can add values at the end or insert them somewhere in the middle of the list. You can address the values using an index. Also you can sort the elements, either in ascending or descending order. Lists can be used in many ways, for example to store changing collections of values. They are implemented using simple arrays but, as this is done in compiled code it is a lot faster than using an array yourself. The following functions are available:

`ds_list_create()` Creates a new list. The function returns an integer as an id that must be used in all other functions to access the particular list.
`ds_list_destroy(id)` Destroys the list with the given id, freeing the memory used. Don't forget to call this function when you are ready with the structure.
`ds_list_clear(id)` Clears the list with the given id, removing all data from it but not destroying it.
`ds_list_copy(id,source)` Copies the list source into the list with the given id.
`ds_list_size(id)` Returns the number of values stored in the list.
`ds_list_empty(id)` Returns whether the list is empty. This is the same as testing whether the size is 0.
`ds_list_add(id,val)` Adds the value at the end of the list.
`ds_list_insert(id,pos,val)` Inserts the value at position pos in the list. The first position is 0, the last position is the size of the list minus 1.
`ds_list_replace(id,pos,val)` Replaces the value at position pos in the list with the new value.
`ds_list_delete(id,pos)` Deletes the value at position pos

in the list. (Position 0 is the first element.)
ds_list_find_index(id,val) Find the position storing the indicated value. If the value is not in the list -1 is returned.
ds_list_find_value(id,pos) Returns the value stored at the indicated position in the list.
ds_list_sort(id,ascend) Sorts the values in the list. When ascend is true the values are sorted in ascending order, otherwise in descending order.
ds_list_shuffle(id) Shuffles the values in the list such that they end up in a random order.
ds_list_write(id) Turns the data structure into a string and returns this string. The string can then be used to e.g. save it to a file. This provides an easy mechanism for saving data structures.
ds_list_read(id,str) Reads the data structure from the given string (as created by the previous call).

Maps

In quite a few situations you need to store pairs consisting of a key and a value. For example, a character can have a number of different items and for each item it has a particular number of those. In this case the item is the key and the number is the value. Maps maintain such pairs, sorted by key. You can add pairs to the map and search for the value corresponding to certain keys. Because the keys are sorted you can also find previous and next keys. Sometimes it is also useful to use a map to just store keys without a corresponding value. In that case you can simply use a value of 0. The following functions exist:

`ds_map_create()` Creates a new map. The function returns an integer as an id that must be used in all other functions to access the particular map.
`ds_map_destroy(id)` Destroys the map with the given id, freeing the memory used. Don't forget to call this function when you are ready with the structure.
`ds_map_clear(id)` Clears the map with the given id, removing all data from it but not destroying it.
`ds_map_copy(id,source)` Copies the map source into the map with the given id.
`ds_map_size(id)` Returns the number of key-value pairs stored in the map.
`ds_map_empty(id)` Returns whether the map is empty. This is the same as testing whether the size is 0.
`ds_map_add(id,key,val)` Adds the key-value pair to the map.
`ds_map_replace(id,key,val)` Replaces the value corresponding with the key with a new value.
`ds_map_delete(id,key)` Deletes the key and the

corresponding value from the map. (If there are multiple entries with the same key, only one is removed.)

`ds_map_exists(id,key)` Returns whether the key exists in the map.

`ds_map_find_value(id,key)` Returns the value corresponding to the key.

`ds_map_find_previous(id,key)` Returns the largest key in the map smaller than the indicated key. (Note that the key is returned, not the value. You can use the previous routine to find the value.)

`ds_map_find_next(id,key)` Returns the smallest key in the map larger than the indicated key.

`ds_map_find_first(id)` Returns the smallest key in the map.

`ds_map_find_last(id)` Returns the largest key in the map.

`ds_map_write(id)` Turns the data structure into a string and returns this string. The string can then be used to e.g. save it to a file. This provides an easy mechanism for saving data structures.

`ds_map_read(id,str)` Reads the data structure from the given string (as created by the previous call).

Priority queues

In a priority queue a number of values are stored, each with a priority. You can quickly find the values with minimum and maximum priority. Using this data structure you can handle certain things in the order of priority. The following functions exist:

`ds_priority_create()` Creates a new priority queue. The function returns an integer as an id that must be used in all other functions to access the particular priority queue.
`ds_priority_destroy(id)` Destroys the priority queue with the given id, freeing the memory used. Don't forget to call this function when you are ready with the structure.
`ds_priority_clear(id)` Clears the priority queue with the given id, removing all data from it but not destroying it.
`ds_priority_copy(id,source)` Copies the priority queue source into the priority queue with the given id.
`ds_priority_size(id)` Returns the number of values stored in the priority queue.
`ds_priority_empty(id)` Returns whether the priority queue is empty. This is the same as testing whether the size is 0.
`ds_priority_add(id,val,prio)` Adds the value with the given priority to the priority queue.
`ds_priority_change_priority(id,val,prio)` Changes the priority of the given value in the priority queue.
`ds_priority_find_priority(id,val)` Returns the priority of the given value in the priority queue.
`ds_priority_delete_value(id,val)` Deletes the given value (with its priority) from the priority queue.
`ds_priority_delete_min(id)` Returns the value with the

smallest priority and deletes it from the priority queue.
`ds_priority_find_min(id)` Returns the value with the smallest priority but does not delete it from the priority queue.
`ds_priority_delete_max(id)` Returns the value with the largest priority and deletes it from the priority queue.
`ds_priority_find_max(id)` Returns the value with the largest priority but does not delete it from the priority queue.
`ds_priority_write(id)` Turns the data structure into a string and returns this string. The string can then be used to e.g. save it to a file. This provides an easy mechanism for saving data structures.
`ds_priority_read(id,str)` Reads the data structure from the given string (as created by the previous call).

Grids

A grid is simply a two-dimensional array. A grid has an integer width and height. The structure allows you to set and retrieve the value of cells in the grid by giving the index of it (which starts with 0 in both the x- and the y-direction). But you can also set the value in regions, add values, and retrieve the sum, max, min, and mean value over a region. The structure is useful to represent e.g. a playing field. Even though all functionality can also be achieved using two-dimensional arrays, the operations on regions are a lot faster. The following functions exist:

`ds_grid_create(w,h)` Creates a new grid with the indicated width and height. The function returns an integer as an id that must be used in all other functions to access the particular grid.

`ds_grid_destroy(id)` Destroys the grid with the given id, freeing the memory used. Don't forget to call this function when you are ready with the structure.

`ds_grid_copy(id,source)` Copies the grid source into the grid with the given id.

`ds_grid_resize(id,w,h)` Resizes the grid to the new width and height. Existing cells keep their original value.

`ds_grid_width(id)` Returns the width of the grid with the indicated id.

`ds_grid_height(id)` Returns the height of the grid with the indicated id.

`ds_grid_clear(id,val)` Clears the grid with the given id, to the indicated value (can both be a number or a string).

`ds_grid_set(id,x,y,val)` Sets the indicated cell in the grid with the given id, to the indicated value (can both be a number or a string).

ds_grid_add(id,x,y,val) Add the value to the indicated cell in the grid with the given id. For strings this corresponds to concatenation.

ds_grid_multiply(id,x,y,val) Multiplies the value to the indicated cell in the grid with the given id. Is only valid for numbers.

ds_grid_set_region(id,x1,y1,x2,y2,val) Sets the all cells in the region in the grid with the given id, to the indicated value (can both be a number or a string).

ds_grid_add_region(id,x1,y1,x2,y2,val) Add the value to the cell in the region in the grid with the given id. For strings this corresponds to concatenation.

ds_grid_multiply_region(id,x1,y1,x2,y2,val) Multiplies the value to the cells in the region in the grid with the given id. Is only valid for numbers.

ds_grid_set_disk(id,xm,ym,r,val) Sets all cells in the disk with center (xm,ym) and radius r.

ds_grid_add_disk(id,xm,ym,r,val) Add the value to all cells in the disk with center (xm,ym) and radius r.

ds_grid_multiply_disk(id,xm,ym,r,val) Multiply the value to all cells in the disk with center (xm,ym) and radius r.

ds_grid_set_grid_region(id,source,x1,y1,x2,y2,xpos,ypos) Copies the contents of the cells in the region in grid source to grid id. xpos and ypos indicate the place where the region must be placed in the grid. (Can also be used to copy values from one place in a grid to another.)

ds_grid_add_grid_region(id,source,x1,y1,x2,y2,xpos,ypos) Adds the contents of the cells in the region in grid source to grid id. xpos and ypos indicate the place where the region must be added in the grid. (id and source can be the same.)

ds_grid_multiply_grid_region(id,source,x1,y1,x2,y2,xpos,ypos) Multiplies the contents of the cells in the region in

grid source to grid id. xpos and ypos indicate the place where the region must be multiplied in the grid. (id and source can be the same.) Only valid for numbers.

ds_grid_get(id,x,y) Returns the value of the indicated cell in the grid with the given id.

ds_grid_get_sum(id,x1,y1,x2,y2) Returns the sum of the values of the cells in the region in the grid with the given id. Does only work when the cells contain numbers.

ds_grid_get_max(id,x1,y1,x2,y2) Returns the maximum of the values of the cells in the region in the grid with the given id. Does only work when the cells contain numbers.

ds_grid_get_min(id,x1,y1,x2,y2) Returns the minimum of the values of the cells in the region in the grid with the given id. Does only work when the cells contain numbers.

ds_grid_get_mean(id,x1,y1,x2,y2) Returns the mean of the values of the cells in the region in the grid with the given id. Does only work when the cells contain numbers.

ds_grid_get_disk_sum(id,xm,ym,r) Returns the sum of the values of the cells in the disk.

ds_grid_get_disk_min(id,xm,ym,r) Returns the min of the values of the cells in the disk.

ds_grid_get_disk_max(id,xm,ym,r) Returns the max of the values of the cells in the disk.

ds_grid_get_disk_mean(id,xm,ym,r) Returns the mean of the values of the cells in the disk.

ds_grid_value_exists(id,x1,y1,x2,y2,val) Returns whether the value appears somewhere in the region.

ds_grid_value_x(id,x1,y1,x2,y2,val) Returns the x-coordinate of the cell in which the value appears in the region.

ds_grid_value_y(id,x1,y1,x2,y2,val) Returns the y-coordinate of the cell in which the value appears in the region.

ds_grid_value_disk_exists(id,xm,ym,r,val) Returns whether the value appears somewhere in the disk.

ds_grid_value_disk_x(id,xm,ym,r,val) Returns the x-coordinate of the cell in which the value appears in the disk.

ds_grid_value_disk_y(id,xm,ym,r,val) Returns the y-coordinate of the cell in which the value appears in the disk.

ds_grid_shuffle(id) Shuffles the values in the grid such that they end up in a random order.

ds_grid_write(id) Turns the data structure into a string and returns this string. The string can then be used to e.g. save it to a file. This provides an easy mechanism for saving data structures.

ds_grid_read(id,str) Reads the data structure from the given string (as created by the previous call).

Creating particles

This functionality is only available in the Pro Edition of Game Maker.

Particle systems are meant to create special effects. Particles are small elements, represented by a little sprite. Such particles move around according to predefined rules and can change size, orientation, color, etc. while they move. Many such particles together can create e.g. fireworks, flames, explosions, rain, snow, star fields, flying debris, etc.

Game Maker contains an extensive particle system that can be used to create great effects. Because of its generality it is not simple to use so better read this section carefully before trying.

If this is too complicated for you, there is also a very simple mechanism to create different types of explosions, smoke, rain, and even fireworks.

Particle systems have many parameters and it is not always easy to understand how to create the effects you want. First of all there are particle types. A particle type defines a particular kind of particles. Such types have many parameters that describe the shape, size, color, and motion of the particles. Particle types need to be defined only once and can then be used everywhere in the game.

Secondly there are particle systems. There can be different particle systems in the game. A particle system can have particles of the different types. A particle system has emitters that create the particles, either continuously or in

bursts. It can also have attractors that attract particles. Finally, it can have destroyers that destroy particles. Once particles are created in a particle system, they are automatically handled (updated and drawn) by the system.

Information on particles can be found in the following pages:

- Simple Effects Particle Types
- Particle Systems
- Emitters
- Attractors
- Destroyers
- Deflectors
- Changers
- Firework Example

Simple Effects

The easiest way of creating particles is to use the effects mechanism. Effects are created using the particle system but you do not have to worry about all the details. You simply specify the type of effect, the position where it must be created, its size, and its color. That is all.

There are a number of different kinds of effects:

- `ef_explosion`
- `ef_ring`
- `ef_ellipse`
- `ef_firework`
- `ef_smoke`
- `ef_smokeup`
- `ef_star`
- `ef_spark`
- `ef_flare`
- `ef_cloud`
- `ef_rain`
- `ef_snow`

Some you want to create just once (like the explosion) and some you want to create in every step (like the smoke or the rain). Note that rain and snow are always created at the top of the room so the position is irrelevant in this case.

Even though this might sound limited, they can actually be used to create great effects. For example by creating a smal puff of red smoke below a moving spaceship in each step, a tail of fire is created. The following two functions exist to create the effects:

effect_create_below(kind,x,y,size,color) Creates an effect of the given kind (see above) at the indicated position. size give the size as follows: 0 = small, 1 = medium, 2 = large. color indicates the color to be used. The effect is created below the instances, that is, at a depth of 100000.

effect_create_above(kind,x,y,size,color) Similar to the previous function but this time the effect is created on top of the instances, that is, at a depth of -100000.

If you want to remove all effects, call the following function:

effect_clear() Clears all effects.

Particle types

A particle type describes the shape, color, motion, etc. of a particular kind of particles. You need to define a particle type only once in the game. After this it can be used in any particle system in the game. Particle types have a large number of parameters that can be used to change all aspects of it. Setting these right you can create almost any effect you like. We will discuss the settings below.

A number of routines are available to create new particle types and destroy them again:

`part_type_create()` Creates a new particle type. It returns the index of the type. This index must be used in all calls below to set the properties of the particle type. So you will often store it in a global variable.
`part_type_destroy(ind)` Destroys particle type ind. Call this if you don't need it anymore to save space.
`part_type_exists(ind)` Returns whether the indicated particle type exists.
`part_type_clear(ind)` Clears the particle type ind to its default settings.

The shape of a particle

A particle has a shape. This shape is indicated by a sprite. You can use any sprite you like for your particles but there are 15 built-in sprites. These are all 64x64 in size and have alpha values set such that they nicely blend with the background. They are indicate by the following constants:

- pt_shape_pixel
- pt_shape_disk
- pt_shape_square
- pt_shape_line
- pt_shape_star
- pt_shape_circle
- pt_shape_ring
- pt_shape_sphere
- pt_shape_flare
- pt_shape_spark
- pt_shape_explosion
- pt_shape_cloud
- pt_shape_smoke
- pt_shape_snow

You set the shape using the following function:

part_type_shape(ind,shape) Sets the shape of the particle type to any of the constants above (default is pt_shape_pixel).

You can also use your own sprite for the particle. If the sprite has multiple subimages you can indicate what should be done with them. You can pick a random one, animate the sprite, start at the beginning of the animation or at a random place, etc. You use the following function for this.

part_type_sprite(ind,sprite,animat,stretch,random) Sets your own sprite for the particle type. With animate you indicate whether the sprite should be animated (1) or not (0). With stretch (1 or 0) you indicate whether the animation must be stretched over the lifetime of the particle. And with random (1 or 0) you can indicate whether a random subimage must be choosen as starting image.

Once you have choosen the sprite for the particle type (either a default shape or your own) you can indicate the size of it. A size of 1 indicates the normal size of the sprite. A particle type can be defined such that all particles have the same size or have different sizes. You can indicate a range of sizes. Also, you can indicate whether the size should change over the lifetime of the particle and whether some wiggling in the size will happen, giving a blinking effect.

part_type_size(ind,size_min,size_max,size_incr,size_wiggle) Sets the size parameters for the particle type. You specify the minimum starting size, the maximum starting size, the size increase in each step (use a negative number for a decrease in size) and the amount of wiggling. (The default size is 1 and default the size does not change.)

part_type_scale(ind,xscale,yscale) Sets the horizontal and vertical scale. This factor is multiplied with the size. It is in particular useful when you need to scale differently in x- and y-direction.

The particles also have an orientation. Again the orientation can be the same for all particles, can be different, and can change over the lifetime of the sprite. The angles specify counter-clockwise rotations, in degrees.

part_type_orientation(ind,ang_min,ang_max,ang_incr,ang_wiggle,ang_relative) Sets the orientation angle properties for the particle type. You specify the minimum angle, the maximum angle, the increase in each step and the amount of wiggling in angle. (Default all values are 0.) You can also indicate whether the given angle should be relative (1) to the current direction of motion or absolute (0). E.g. by setting all values to 0 but

ang_relative to 1, the particle orientation will precisely follow the path of the particle.

Color and blending

Particles will have a color. There are different ways in which you can specify colors for a particle. The simplest way is to indicate a single color. You can also specify two or three colors between which the color of the particle is interpolated during its life time. For example, the particle can start white and become more and more black over its lifetime. Another possibility is that you indicate that the color of each particle must be different, picked from a range of colors. You can either give a range in red, green and blue, or a range in hue, saturation, and value.

Default the color is white. When you use a sprite with its own colors, this is normally what you want and no color needs to be specified.

part_type_color1(ind,color1) Indicates a single color to be used for the particle.

part_type_color2(ind,color1,color2) Specifies two colors between which the color is interpolated.

part_type_color3(ind,color1,color2,color3) Similar but this time the color is interpolated between three colors that represent the color at the start, half-way, and at the end.

part_type_color_mix(ind,color1,color2) With this function you indicate that the particle should get a color that is a random mixture of the two indicated colors. This color will remain fixed over the lifetime of the particle.

part_type_color_rgb(ind,rmin,rmax,gmin,gmax,bmin,bmax) Can be used to indicate that each particle must have a fixed

color but choosen from a range. You specify a range in the red, green, and blue component of the color (each between 0 and 255).

`part_type_color_hsv(ind,hmin,hmax,smin,smax,vmin,vmax)` Can be used to indicate that each particle must have a fixed color but choosen from a range. You specify a range in the hue saturation and value component of the color (each between 0 and 255).

Besides the color you can also give an alpha transparency value. The built-in particle shapes already have some alpha transparency but you can use these settings to e.g. make the particle vanish over its life time.

`part_type_alpha1(ind,alpha1)` Sets a single alpha transparency parameter (0-1) for the particle type.

`part_type_alpha2(ind,alpha1,alpha2)` Similar but this time a start and end value are given and the alpha value is interpolated between them.

`part_type_alpha3(ind,alpha1,alpha2,alpha3)` This time three values are given between which the alpha transparency is interpolated.

Normally particles are blended with the background in the same way as sprites. But it is also possible to use additive blending. This gives in particular a great effect for explosion.

`part_type_blend(ind,additive)` Sets whether to use additive blending (1) or normal blending (0) for the particle type.

Life and death

Particles live for a limited amount of time, their lifetime. After this they disappear. Lifetime is measured in steps. You can indicate the lifetime (or a range of lifetimes) for each particle type. Particles can create new particles of different types. There are two ways for this. They can create new particles in each step or they can create particles when they die. Be careful that the total number of particles does not get too high.

`part_type_life(ind,life_min,life_max)` Sets the lifetime bounds for the particle type. (Default both are 100.)
`part_type_step(ind,step_number,step_type)` Sets the number and type of particles that must be generated in each step for the indicated particle type. If you use a negative value, in each step a particle is generated with a chance -1/number. So for example with a value of -5 a particle is generated on average once every 5 steps.
`part_type_death(ind,death_number,death_type)` Sets the number and type of particles that must be generated when a particle of the indicated type dies. Again you can use negative numbers to create a particle with a particular chance. Note that these particles are only created when the particle dies at the end of its life, not when it dies because of a destroyer (see below).

Particle motion

Particles can move during their lifetime. They can get an initial speed (or range of speeds) and direction, and the speed and direction can change over time. Also gravity can be defined that pulls the particles in a particular direction. The following functions exist for this:

part_type_speed(ind,speed_min,speed_max,speed_incr,speed_wiggle) Sets the speed properties for the particle type. (Default all values are 0.) You specify a minimal and maximal speed. A random value between the given bounds is chosen when the particle is created. You can indicate a speed increase in each step Use a negative number to slow the particle down (the speed will never become smaller than 0). Finally you can indicate some amount of wiggling of the speed.

part_type_direction(ind,dir_min,dir_max,dir_incr,dir_wiggle) Sets the direction properties for the particle type. (Default all values are 0.) Again you specify a range of directions (in counterclockwise degrees; 0 indicated a motion to the right). For example, to let the particle move in a random direction choose 0 and 360 as values. You can specify an increase in direction for each step, and an amount of wiggling.

part_type_gravity(ind,grav_amount,grav_dir) Sets the gravity properties for the particle type. (Default there is no gravity.) You specify the amount of gravity to be added in each step and the direction. E.g. use 270 for a downwards direction.

Particle systems

Particles live in particle systems. So to have particles in your game you need to create one or more particle systems. There can be different particle systems (but preferably keep their number small). For example, if your game has a number of balls and each ball should have a tail of particles, most likely each ball has its own particle system. The easiest way to deal with particle systems is to create one and then create particles in it, using the particle types you specified before. But, as we will see below, particle systems can contain emitters that automatically produce particles, attractors that attract them, and destroyers that destroy them.

Once particles are added to a particle system they are automatically updated each step and drawn. No further action is required. To make it possible that particles are drawn, behind, in front of, or between object instances, each particle system has a depth, similar to instances and tiles.

Particle systems will live on forever after they are created. So even if you change room or restart the game, the systems and the particles remain. So you better make sure you destroy them once you no longer need them.

The following basic functions deal with particle systems:

`part_system_create()` Creates a new particle system. It returns the index of the system. This index must be used in all calls below to set the properties of the particle system.

`part_system_destroy(ind)` Destroys the particle system ind.

Call this if you don't need it anymore to save space.

part_system_exists(ind) Returns whether the indicated particle system exists.

part_system_clear(ind) Clears the particle system ind to its default settings, removing all particles and emitter and attractors in it.

part_system_draw_order(ind,oldtonew) Sets the order in which the particle system draws the particles. When oldtonew is true the oldest particles are drawn first and the newer one lie on top of them (default). Otherwise the newest particles are drawn first. This can give rather different effects.

part_system_depth(ind,depth) Sets the depth of the particle system. This can be used to let the particles appear behind, in front of, or in between instances.

part_system_position(ind,x,y) Sets the position where the particle system is drawn. This is normally not necessary but if you want to have particles at a position relative to a moving object, you can set the position e.g. to that object.

As indicated above, the particle system is automatically updated and drawn. But sometimes this is not what you want. To facilitate this, you can switch off automatic updating or drawing and then decide yourself when to update or draw the particle system. For this you can use the following functions:

part_system_automatic_update(ind,automatic) Indicates whether the particle system must be updated automatically (1) or not (0). Default is 1.

part_system_automatic_draw(ind,automatic) Indicates whether the particle system must be drawn automatically (1) or not (0). Default is 1.

part_system_update(ind) This functions updates the

position of all particles in the system and lets the emitters create particles. You only have to call this when updating is not automatic. (Although sometimes it is also useful to call this function a couple of time to get the system going.)

part_system_drawit(ind) This functions draws the particles in the system. You only have to call this when drawing is not automatic. It should be called in the draw event of some object.

The following functions deal with particles in a particle systems:

part_particles_create(ind,x,y,parttype,number) This functions creates `number` particles of the indicated type at postion (x,y) in the system.

part_particles_create_color(ind,x,y,parttype,color,number) This functions creates `number` particles of the indicated type at postion (x,y) in the system with the indicated color. This is only useful when the particle type defines a single color (or does not define a color at all).

part_particles_clear(ind) This functions removes all particles in the system.

part_particles_count(ind) This functions returns the number of particles in the system.

Emitters

Emitters create particles. They can either create a continuous stream of particles or can burst out a number of particles when using the appropriate function. A particle system can have an arbitrary number of emitters. An emitter has the following properties:

- **`xmin`**, **`xmax`**, **`ymin`**, **`ymax`** indicate the extend of the region in which the particles are generated.
- **`shape`** indicates the shape of the region. It can have the following values:
  - `ps_shape_rectangle`
  - `ps_shape_ellipse`
  - `ps_shape_diamond`
  - `ps_shape_line`
- **`distribution`** indicates the distribution used to generate the particles. It can have the following values:
  - `ps_distr_linear` indicates a linear distribution, that is everywhere in the region the chance is equal
  - `ps_distr_gaussian` indicates a Gaussian distribution in which more particles are generated in the center than at the sides of the region
  - `ps_distr_invgaussian` indicates an inverse Gaussian distribution in which more particles are generated at the sides of the region than in the center
- **`particle type`** indicates the type of particles being generated
- **`number`** indicates the number of particles generated in each step. If smaller than 0, in each step a particle is generated with a chance -1/number. So for example with a value of -5 a particle is generated on average once every 5 steps.

The following functions are available to set the emitters and to let them create particles. Note that each of them gets the index of the particle system to which it belongs as a first argument.

part_emitter_create(ps) Creates a new emitter in the given particle system. It returns the index of the emitter. This index must be used in all calls below to set the properties of the emitter.

part_emitter_destroy(ps,ind) Destroys emitter ind in the particle system. Call this if you don't need it anymore to save space.

part_emitter_destroy_all(ps) Destroys all emitters in the particle system that have been created.

part_emitter_exists(ps,ind) Returns whether the indicated emitter exists in the particle system.

part_emitter_clear(ps,ind) Clears the emitter ind to its default settings.

part_emitter_region(ps,ind,xmin,xmax,ymin,ymax,shape,distribution) Sets the region and distribution for the emitter.

part_emitter_burst(ps,ind,parttype,number) Bursts once `number` particles of the indicated type from the emitter.

part_emitter_stream(ps,ind,parttype,number) From this moment on create `number` particles of the indicated type from the emitter in every step. If you indicate a number smaller than 0 in each step a particle is generated with a chance of -1/number. So for example with a value of -5 a particle is generated on average once every 5 steps.

Attractors

Besides emitters a particle system can also contain attractors. An attractor attracts the particles (or pushes them away). A particle system can have multiple attractors. You are though recommended to use few of these because they will slow down the processing of the particles. An attractor has the following properties:

- **`x,y`** indicate the position of the attractor.
- **`force`** indicates the attracting force of the attractor. How the force acts on the particles depends on the following parameters.
- **`dist`** indicates the maximal distance at which the attractor has effect. Only particles closer that this distance to the attractor will be attracted.
- **`kind`** indicates the kind of attractor. The following values exist
  - `ps_force_constant` indicates that the force is constant independent of the distance.
  - `ps_force_linear` indicates a linearly growing force. At the maximal distance the force is 0 while at the position of the attractor it attains the given value.
  - `ps_force_quadratic` indicates that the force grows quadratic.
- **`additive`** indicates whether the force is added to the speed and direction in each step (true) or only applied to the position of the particle (false). When additive the particle will accelerate towards the attractor while with a non-additive force it will move there with constant speed.

The following functions exist to define attractors. Note that each of them gets the index of the particle system to which it belongs as a first argument.

part_attractor_create(ps) Creates a new attractor in the given particle system. It returns the index of the attractor. This index must be used in all calls below to set the properties of the attractor.
part_attractor_destroy(ps,ind) Destroys attractor ind in the particle system. Call this if you don't need it anymore to save space.
part_attractor_destroy_all(ps) Destroys all attractors in the particle system that have been created.
part_attractor_exists(ps,ind) Returns whether the indicated attractor exists in the particle system.
part_attractor_clear(ps,ind) Clears the attractor ind to its default settings.
part_attractor_position(ps,ind,x,y) Sets the position of attractor ind to (x,y).
part_attractor_force(ps,ind,force,dist,kind,aditive) Sets the force parameters of attractor ind.

Destroyers

Destroyers destroy particles when they appear in their region. A particle system can have an arbitrary number of destroyers. A destroyer has the following properties:

- **xmin, xmax, ymin, ymax** indicates the extent of the region in which the particles are destroyed.
- **shape** indicates the shape of the region. It can have the following values:
  - ps_shape_rectangle
  - ps_shape_ellipse
  - ps_shape_diamond

The following functions are available to set the properties of the destroyers. Note that each of them gets the index of the particle system to which it belongs as a first argument.

part_destroyer_create(ps) Creates a new destroyer in the given particle system. It returns the index of the destroyer. This index must be used in all calls below to set the properties of the destroyer.
part_destroyer_destroy(ps,ind) Destroys destroyer ind in the particle system. Call this if you don't need it anymore to save space.
part_destroyer_destroy_all(ps) Destroys all destroyers in the particle system that have been created.
part_destroyer_exists(ps,ind) Returns whether the indicated destroyer exists in the particle system.
part_destroyer_clear(ps,ind) Clears the destroyer ind to its default settings.
part_destroyer_region(ps,ind,xmin,xmax,ymin,ymax,shape) Sets the region for the destroyer.

Deflectors

Deflectors deflect particles when they appear in their region. Note that only the position of the particle is taken into account not its sprite or size. A particle system can have an arbitrary number of deflectors. A deflector has the following properties:

- **`xmin`**, **`xmax`**, **`ymin`**, **`ymax`** indicates the extent of the region in which the particles are deflected.
- **`kind`** indicates the kind of deflector. It can have the following values:
  - `ps_deflect_horizontal` deflects the particle horizontally; typically used for vertical walls
  - `ps_deflect_vertical` deflects the particle vertically; typically used for horizontal walls
- **`friction`** the amount of friction as a result of the impact with the deflector. The higher this amount the more the particle is slowed down on impact.

The following functions are available to set the properties of the deflector. Note that each of them gets the index of the particle system to which it belongs as a first argument.

`part_deflector_create(ps)` Creates a new deflector in the given particle system. It returns the index of the deflector. This index must be used in all calls below to set the properties of the deflector.
`part_deflector_destroy(ps,ind)` Destroys deflector ind in the particle system. Call this if you don't need it anymore to save space.
`part_deflector_destroy_all(ps)` Destroys all deflectors in the particle system that have been created.

part_deflector_exists(ps,ind) Returns whether the indicated deflector exists in the particle system.
part_deflector_clear(ps,ind) Clears the deflector ind to its default settings.
part_deflector_region(ps,ind,xmin,xmax,ymin,ymax) Sets the region for the deflector.
part_deflector_kind(ps,ind,kind) Sets the kind for the deflector.
part_deflector_friction(ps,ind,friction) Sets the friction for the deflector.

Changers

Changers change certain particles when they appear in their region. A particle system can have an arbitrary number of changers. A changer has the following properties:

- **xmin**, **xmax**, **ymin**, **ymax** indicates the extent of the region in which the particles are changed.
- **shape** indicates the shape of the region. It can have the following values:
  - ps_shape_rectangle
  - ps_shape_ellipse
  - ps_shape_diamond
- **parttype1** indicates the particle type that is changed.
- **parttype2** indicates the particle type into which it is changed.
- **kind** indicates the kind of changer. It can have the following values:
  - ps_change_motion only changes the motion parameters of the particle, not the color and shape or lifetime settings
  - ps_change_shape only changes the shape parameters like size and color and shape
  - ps_change_all changes all parameters, this basically means that the particle is destroyed and a new one of the new type is created.

The following functions are available to set the properties of the changer. Note that each of them gets the index of the particle system to which it belongs as a first argument.

part_changer_create(ps) Creates a new changer in the given particle system. It returns the index of the

changer. This index must be used in all calls below to set the properties of the changer.

part_changer_destroy(ps,ind) Destroys changer ind in the particle system. Call this if you don't need it anymore to save space.

part_changer_destroy_all(ps) Destroys all changers in the particle system that have been created.

part_changer_exists(ps,ind) Returns whether the indicated changer exists in the particle system.

part_changer_clear(ps,ind) Clears the changer ind to its default settings.

part_changer_region(ps,ind,xmin,xmax,ymin,ymax,shape) Sets the region for the changer.

part_changer_types(ps,ind,parttype1,parttype2) Sets which particle type the changer must changed into what other type.

part_changer_kind(ps,ind,kind) Sets the kind for the changer.

Firework Example

Here is an example of a particle system that creates fireworks. The firework uses two particle types: one that forms the rocket and one that forms the actual fireworks. The rocket generates the firework particles when it dies. We also generate one emitter in the particle system that regularly stream out rocket particles along the bottom of the screen. To make this work you need an object. In its creation event we place the following code that creates the particle types, particle system, and the emitter:

```
{
  // make the particle system
  ps = part_system_create();

  // the firework particles
  pt1 = part_type_create();
  part_type_shape(pt1,pt_shape_flare);
  part_type_size(pt1,0.1,0.2,0,0);
  part_type_speed(pt1,0.5,4,0,0);
  part_type_direction(pt1,0,360,0,0);
  part_type_color1(pt1,c_red);
  part_type_alpha2(pt1,1,0.4);
  part_type_life(pt1,20,30);
  part_type_gravity(pt1,0.2,270);

  // the rocket
  pt2 = part_type_create();
  part_type_shape(pt2,pt_shape_sphere);
  part_type_size(pt2,0.2,0.2,0,0);
  part_type_speed(pt2,10,14,0,0);
  part_type_direction(pt2,80,100,0,0);
  part_type_color2(pt2,c_white,c_gray);
  part_type_life(pt2,30,60);
  part_type_gravity(pt2,0.2,270);
  part_type_death(pt2,150,pt1);    // create the
firework on death
```

```
    // create the emitter
    em = part_emitter_create(ps);

  part_emitter_region(ps,em,100,540,480,490,ps_shape_rec
  tangle,ps_distr_linear);
    part_emitter_stream(ps,em,pt2,-4);   // create one
  every four steps
  }
```

That will do the trick. You might want to make sure the particle system (and maybe particle types) are destroyed when moving to another room, otherwise the firework will continue forever.

Multiplayer games

This functionality is only available in the Pro Edition of Game Maker.

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

Information on multiplayer games can be found in the following pages:

Setting up a Connection Creating and Joining Sessions
Players
Shared Data
Messages

Setting up a connection

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

`mplay_init_ipx()` initializes an IPX connection.

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

`mplay_init_modem(initstr,phonenr)` initializes a modem connection. initstr is the initialization string for the modem (can be empty). phonenr is a string that

contains the phone number to ring (e.g. '0201234567'). Only when joining a session (see below) do you need to provide a phone number.

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

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

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

To end the connection call

mplay_end() ends the current connection.

When using a TCP/IP connection you might want to tell the person you want to play the game with what the ip address of your computer is. The following function helps you here:

`mplay_ipaddress()` returns the IP address of your machine (e.g. '123.123.123.12') as a string. You can e.g. display this somewhere on the screen. Note that this routine is slow so don't call it all the time.

Creating and joining sessions

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

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

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

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

`mplay_session_find()` searches for all sessions that still accept players and returns the number of sessions found.

mplay_session_name(numb) returns the name of session number numb (0 is the first session). This routine can only be called after calling the previous routine.
mplay_session_join(numb,playername) makes you join session number numb (0 is the first session). playername is your name as a player. Returns whether successful.

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

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

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

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

A player can stop a session using the following routine:

mplay_session_end() ends the session for this player.

Players

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

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

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

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

Shared data

Shared data communication is probably the easiest way to synchronize the game. All communication is shielded from you. There is a set of 1000000 values that are common to all entities of the game (preferably only use the first few to save memory). Each entity can set values and read values. *Game Maker* makes sure that each entity sees the same values. A value can either be a real or a string. There are just two routines:

`mplay_data_write(ind,val)` write value val (string or real) into location ind (ind between 0 and 1000000).
`mplay_data_read(ind)` returns the value in location ind (ind between 0 and 1000000). Initially all values are 0.

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

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

Messages

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

The following messaging routines exist:

`mplay_message_send(player,id,val)` sends a message to the indicated player (either an identifier or a name; use 0 to send the message to all players). id is an integer message identifier and val is the value (either a real or a string). The message is sent in non-guaranteed mode. If val contains a string the maximal string length allowed is 30000 characters.

`mplay_message_send_guaranteed(player,id,val)` sends a message to the indicated player (either an identifier or a name; use 0 to send the message to all players). id is an integer message identifier and val is the value (either a real or a string). This is a guaranteed send. If val contains a string the maximal string length allowed is 30000 characters.

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

`mplay_message_id()` Returns the identifier of the last

received message.

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

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

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

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

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

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

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

Using DLL's

This functionality is only available in the Pro Edition of Game Maker.

Please not that since version 7 there is a new extension mechanism in Game Maker. You are strongly encouraged to use that extension mechanism, rather than the functions described in this section. See http://www.yoyogames.com/extensions for details. These functions are mainly left in for compatibility with the past.

In those cases were the functionality of GML is not enough for your wishes, you can actually extend the possibilities by using plug-ins. A plug-in comes in the form of a DLL file (a Dynamic Link Library). In such a DLL file you can define functions. Such functions can be programmed in any programming language that supports the creation of DLL's (e.g. Delphi, C, C++, etc.) You will though need to have some programming skill to do this. Plug-in functions must have a specific format. They can have between 0 and 16 arguments, each of which can either be a real number (double in C) or a null-terminated string. (For more than 4 arguments, only real arguments are supported at the moment.) They must return either a real or a null-terminated string.

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

```
library MyDLL;

uses SysUtils, Classes;

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

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

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

exports MyMin, DoubleString;

begin
end.
```

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

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

external_define(dll,name,calltype,restype,argnumb,arg1type, arg2type, ...) Defines an external function. dll is the name of the dll file. name is the name of the functions. calltype is the calling convention used. For this use either dll_cdecl or dll_stdcall. restype is the type of the result. For this use either ty_real or ty_string. argnumb is the number of arguments (0-16). Next, for each argument you must specify its type. For this again use either ty_real or ty_string. When there are more than 4 arguments all of them must be of type ty_real.

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

```
{
  global.mmm =
external_define('MyDLL.DLL','MyMin',dll_cdecl,

ty_real,2,ty_real,ty_real);
  global.ddd =
external_define('MyDLL.DLL','DoubleString',dll_cdecl,

ty_string,1,ty_string);
}
```

Now whenever you need to call the functions, you use the following function:

external_call(id,arg1,arg2,...) Calls the external function with the given id, and the given arguments. You need to provide the correct number of arguments of the correct type (real or string). The function returns the result of the external function.

So, for example, you would write:

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

If you don't need to use the DLL anymore you had better free it.

external_free(dll) Frees the DLL with the given name. This is in particular necessary if the game should remove the DLL. As long as the DLL is not freed it cannot be removed. Best do this e.g. in an end of game event.

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

execute_string(str,arg0,arg1,...) Execute the piece of code in the string str with the indicated arguments.

Alternatively you can let the DLL create a file with a script that can be executed (this function can also be used to later modify the behavior of a game).

execute_file(fname) Execute the piece of code in the file.

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

```
{
  ccc = external_call(global.ddd,x,y);
```

```
    execute_string(ccc);
  }
```

In some rare cases your DLL might need to know the handle of the main graphics window for the game. This can be obtained with the following function and can then be passed to the DLL:

`window_handle()` Returns the window handle for the main window.

Note that DLLs cannot be used in secure mode.

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

3D Graphics

This functionality is only available in the Pro Edition of Game Maker.

Game Maker is a program meant for making 2-dimensional and isometric games. Still there is some functionality to create 3-dimensional graphics. Before you start with this though there are a few things you must understand.

- The 3D functionality in *Game Maker* is limited to the graphics part. There is no support for other 3D functionality. Once you start using 3D graphics you might get problems with other aspects of *Game Maker*, like the views, depth ordering, etc. The functionality is limited and has low priority to be extended. So don't expect support for 3D object models, etc.
- When you use the 3D functionality, there are a number of other things that can no longer be used.
  - You cannot use background and foregrounds in your rooms anymore. (The reason is that they are tiled to fill the image but with perspective projections this no longer work correctly).
  - You cannot use the mouse position anymore. The mouse will not be transformed to the 3D coordinates. You can still get the position of the mouse on the screen (in the view) but you will have to do calculation with this yourself (or not use the mouse at all).
  - You cannot use tiles anymore. Tiles will most likely no longer match up correctly.
  - Collision checking still uses the 2-d positions of the instances in the room. So there is no collision detection in 3D. Sometimes you can still use this (if

you use the room as a representation of a flat world (e.g. for racing or FPS games) but in other situations you have to do things yourself.

- All 3D functionality is through code. You must be rather fluent with the GML language. Also you must really understand a lot about how *Game Maker* works otherwise you will run into trouble.
- You must have some basic knowledge about 3D graphics. In particular I will use terms like perspective projections, hidden surface removal, lighting, and fog, without much explanation.
- There is no 3D modelling in *Game Maker*. Also I do not plan on adding support for loading 3D models.
- You must work carefully to keep a reasonable speed. Also, things are not really optimized for speed.

If this did not discourage you, read on.

Information on 3d graphics can be found in the following pages:

Going to 3D mode Easy drawing
Drawing polygons in 3D
Drawing basic shapes
Viewing the world
Transformations
Fog
Lighting
Creating models
Final words

Going to 3D mode

If you want to use 3D mode you first need to set *Game Maker* in 3D mode. You can later switch back to 2D mode if you want. The following two functions exist for this.

`d3d_start()` Start using 3D mode. Returns whether successful.
`d3d_end()` Stop using 3D mode. Returns whether successful.

Note that all functions related to 3D mode start with `d3d_`.

Starting 3D mode will result in the following changes. First of all hidden surface removal is switched on (using a 16-bit z-buffer). This means that for each pixel on the screen only the drawing with the smallest z-value (= depth value) is drawn. If instances have the same depth it is unclear what will happen and you can get ugly effects. Make sure instances that might overlap do not have the same depth value!

Secondly, the normal orthographic projection is replaced by a perspective one. This means the following. Normally the size of instances on the screen is independent on its depth. With a perspective projection instances that have a greater depth will appear smaller. When the depth is 0 it is equal to the old size (unless you change the projection; see below). The viewpoint for the camera is placed at a distance above the room. (This distance is equal to the width of the room; that gives a reasonable default projection.) Only instances in front of the camera are drawn. So don't use instances with a depth smaller than 0 (or at least not smaller than -w where w is the width of the room or the view).

Thirdly, the vertical y-coordinate is reversed. While normally the (0,0) position is at the top-left of the view, in 3D modo the (0,0) position is at the bottom-left position, as is normal for 3-dimensional views.

You can actually switch hidden surface remove and perspective projection on or off using the following functions.

`d3d_set_hidden(enable)` Enables hidden surface removal (true) or disables it (false).
`d3d_set_perspective(enable)` Enables the use of a perspective projection (true) or disables it (false).

Easy drawing

Once 3D mode has been switched on you can use *Game Maker* as you are used to it (except for the remarks made at the beginning). Only objects will appear in different sizes based on their depth setting. You can even use views. One additional function can be useful. If you draw a number of things in a piece of code you might want to change the depth value between the primitives you draw. For this you use:

`d3d_set_depth(depth)` Sets the depth used for drawing.

Note that at the moment a new instance is drawn the depth is again set to the depth of that instance.

Drawing polygons in 3D

The problem with drawing in the old way is that a sprite or polygon always lies in the xy-plane, that is, all corners have the same depth. For true 3D you want to be able to have vertices at different depths. From this moment on we will talk about z-coordinate rather than depth. So we want to specify coordinates as (x,y,z) tuples. For this there are special version of the advanced drawing functions:

`d3d_primitive_begin(kind)` Start a 3D primitive of the indicated kind: `pr_pointlist, pr_linelist, pr_linestrip,pr_trianglelist,pr_trianglestrip` or `pr_trianglefan`.

`d3d_vertex(x,y,z)` Add vertex (x,y,z) to the primitive, using the color and alpha value set before.

`d3d_vertex_color(x,y,z,col,alpha)` Add vertex (x,y,z) to the primitive, with its own color and alpha value. This allows you to create primitives with smoothly changing color and alpha values.

`d3d_primitive_end()` End the description of the primitive. This function actually draws it.

For example, to draw a tetrahedron (three sided pyramid) standing on the z=0 plane with its top at z = 200, you can use the following code:

```
{
  d3d_primitive_begin(pr_trianglelist);
    d3d_vertex(100,100,0);
    d3d_vertex(100,200,0);
    d3d_vertex(150,150,200);
    d3d_vertex(100,200,0);
    d3d_vertex(200,200,0);
    d3d_vertex(150,150,200);
```

```
    d3d_vertex(200,200,0);
    d3d_vertex(100,100,0);
    d3d_vertex(150,150,200);
    d3d_vertex(100,100,0);
    d3d_vertex(100,200,0);
    d3d_vertex(200,200,0);
  d3d_primitive_end();
}
```

Now if you would use this, most likely you would just see a triangle on the screen because the top of the tetrahedron will be behind it from the viewpoint. Also, using just one color, it would be difficult to see the different faces. Below we will see ways to change the viewpoint. Assigning colors can be done as before by added `draw_set_color(col)` function calls between the vertices.

You can also use textured poygons in 3D. It works exactly the same as described in the advanced drawing functions in the documentation. But this time you need 3D variants of the basic functions. One thing you must realize. In a texture the position (0,0) is the top-left corner. But often, when using projections (as indicated below), the bottom-left corner is (0,0). In such a case you might need to flip the texture vertically.

`d3d_primitive_begin_texture(kind,texid)` Start a 3D primitive of the indicated kind with the given texture.
`d3d_vertex_texture(x,y,z,xtex,ytex)` Add vertex (x,y,z) to the primitive with position (xtex,ytex) in the texture, blending with the color and alpha value set before.
`d3d_vertex_texture_color(x,y,z,xtex,ytex,col,alpha)` Add vertex (x,y,z) to the primitive with position (xtex,ytex) in the texture, blending with its own color and alpha value.
`d3d_primitive_end()` End the description of the primitive. This function actually draws it.

So, for example you can use the following code to draw a background image that disappears into the distance

```
{
  var ttt;
  ttt = background_get_texture(back);
  d3d_primitive_begin_texture(pr_trianglefan,ttt);
    d3d_vertex_texture(0,480,0,0,0);
    d3d_vertex_texture(640,480,0,1,0);
    d3d_vertex_texture(640,480,1000,1,1);
    d3d_vertex_texture(0,480,1000,0,1);
  d3d_primitive_end();
}
```

A triangle has a front and a back side. The front side is defined to be the side where the vertices are defined in counter-clockwise order. Normally both sides are drawn. But if you make a closed shape this is a waste because the back side of the triangle can never be seen. In this case you can switch on backface culling. This saves about half the amount of drawing time but it leaves you with the task of defining your polygons in the right way. The following function exists:

`d3d_set_culling(cull)` Indicates to start backface culling (true) or stop backface culling (false).

Drawing basic shapes

A number of functions exist for drawing basic shapes, like blocks and walls. Note that these shapes also work correctly with backface culling on.

d3d_draw_block(x1,y1,z1,x2,y2,z2,texid,hrepeat,vrepeat)
Draws a block in the current color with the indicated opposite corners using the indicated texture. Use -1 to not use a texture. `hrepeat` indicates how often the texture must be repeated along the horizontal edge of each face. `vrepeat` does the same for the vertical edge.

d3d_draw_cylinder(x1,y1,z1,x2,y2,z2,texid,hrepeat,vrepeat,closed,steps) Draws a vertical cylinder in the current color in the indicated bounding box using the indicated texture. Use -1 to not use a texture. `hrepeat` indicates how often the texture must be repeated along the horizontal edge of each face. `vrepeat` does the same for the vertical edge. `closed` indicates whether to close the top and bottom of the cylinder. `steps` indicates how many rotational steps must be taken. A typical value is 24.

d3d_draw_cone(x1,y1,z1,x2,y2,z2,texid,hrepeat,vrepeat,closed,steps) Draws a vertical cone in the current color in the indicated bounding box using the indicated texture. Use -1 to not use a texture. `hrepeat` indicates how often the texture must be repeated along the horizontal edge of each face. `vrepeat` does the same for the vertical edge. `closed` indicates whether to close the top and bottom of the cylinder. `steps` indicates how many rotational steps must be taken. A typical value is 24.

d3d_draw_ellipsoid(x1,y1,z1,x2,y2,z2,texid,hrepeat,vrepeat,

steps) Draws an ellipsoid in the current color in the indicated bounding box using the indicated texture. Use -1 to not use a texture. `hrepeat` indicates how often the texture must be repeated along the horizontal edge of each face. `vrepeat` does the same for the vertical edge. `steps` indicates how many rotational steps must be taken. A typical value is 24.

d3d_draw_wall(x1,y1,z1,x2,y2,z2,texid,hrepeat,vrepeat) Draws a vertical wall in the current color with the given corners using the indicated texture. Use -1 to not use a texture. `hrepeat` indicates how often the texture must be repeated along the horizontal edge of each face. `vrepeat` does the same for the vertical edge.

d3d_draw_floor(x1,y1,z1,x2,y2,z2,texid,hrepeat,vrepeat) Draws a (slanted) floor in the current color with the given corners using the indicated texture. Use -1 to not use a texture. `hrepeat` indicates how often the texture must be repeated along the horizontal edge of each face. `vrepeat` does the same for the vertical edge.

The following piece of code draws two blocks:

```
{
  var ttt;
  ttt = background_get_texture(back);
  d3d_draw_block(20,20,20,80,40,200,ttt,1,1);
  d3d_draw_block(200,300,-10,240,340,100,ttt,1,1);
}
```

Viewing the world

Default you look along the negative z-axis toward the middle of the room. Often in 3D games you want to change how you look at the world. For example, in a first person shooter you probably want to have the camera look from a position a bit above the xy-plane along the xy-plane. In graphics terms you are setting the correct projection. To change the way you look the following two functions exist.

`d3d_set_projection(xfrom,yfrom,zfrom,xto,yto,zto,xup,yup,zup)` Defines how to look in the world. You specify the point to look from, the point to look to and the up vector.

This function requires some explanation. To set the projection you first need the position you look from. This is indicated by the parameters `(xfrom,yfrom,zfrom)`. Next you must specify the direction you look in. This is done by giving a second point to look towards. This is the point `(xto,yto,zto)`. Finally, you can still rotate the camera around the line from the viewpoint to the looking point. To specify this we must give an up vector, that is, the direction that is upwards in the camera. This is given by the last three parameters `(xup,yup,zup)`. Let me given an example. To look along the xy-plane as in a first person shooter you can use

```
{
  d3d_set_projection(100,100,10,200,100,10,0,0,1);
}
```

So you look from point (100,100) and 10 above the plane in the direction of (200,100). The up vector points is the z-direction as required. To make this slightly more

complicated, assume you have an instance in your room that specifies the position of the camera. It will have a current (x,y) position and a direction (and maybe even a speed). You can now specify this as your camera by using the following code:

```
{
  with (obj_camera)
    d3d_set_projection(x,y,10,
                x+cos(direction*pi/180),y-
sin(direction*pi/180),10,
                0,0,1);
}
```

This might look a bit complicated. We look from the camera position (x,y), 10 above the ground. To determine a point in the correct direction we need to do a little arithmetic. This point is indicated by the next three parameters. Finally we use the up vector as above.

One important remark! When *Game Maker* starts drawing a room it will set the view point back to the default position. So the first thing you must do when drawing the scene is set is to the projection you want. This must be done in a drawing event!

There is also an extended version of the function above:

d3d_set_projection_ext(xfrom,yfrom,zfrom,xto,yto,zto,xup,yup,zup,angle,aspect,znear,zfar) An extended version of this function in which you also specify the angle defining the field of view, the aspect ratio between horizontal and vertical size of the view, and the near and far clipping planes.

The additional parameters work as follows. If you specified the camera position, point to look at, and up vector, you can

still change how wide the lens of the camera is. This is called the field of view. A reasonable value is 45 degrees and this is what is default taken. But you can change this if you like. Next you can specify the aspect ratio between the horizontal and vertical projection. Normally you want to use the same as the aspect ration of the room or view, e.g. 640/480. Finally you can indicate the clipping planes. Objects that are closer than `znear` to the camera are not drawn. Similar for objects further than `zfar`. It can be important to set these parameters to reasonable values because they also influence the precision of the z-comparisons. If you make the range too large the precision gets worse. Default we use 1 and 32000. `znear` must be larger than 0!

Sometimes you temporarily need a normal orthographic projection as is used when there is no 3D. Or you want to return to the default perspective projection. For this you can use the following functions:

`d3d_set_projection_ortho(x,y,w,h,angle)` Sets a normal orthographic projection of the indicated area in the room, rotated over the indicated angle.
`d3d_set_projection_perspective(x,y,w,h,angle)` Sets a normal perspective projection of the indicated area in the room, rotated over the indicated angle.

A standard use for this is to draw an overlay to e.g. show the score or other aspects. To do so we set an orthographic projection. We also must temporarily switch off hidden surface removal cause we want the information to be drawn regardless of the current depth value. The following example shows how to create an overlay with the score.

```
{
  draw_set_color(c_black);
```

```
d3d_set_projection_ortho(0,0,room_width,room_height,0)
;
  d3d_set_hidden(false);
  draw_text(10,10,'Score: ' + string(score));
  d3d_set_hidden(true);
}
```

Transformations

Transformation allow you to change the place where things are drawn in the world. For example, the function to draw blocks can only draw axis-parallel blocks. By first setting a rotation transformation you can create rotated blocks. Also sprites are always drawn parallel to the xy-plane. By setting a transformation you can change this. There are two types of functions: functions that set the transformation and functions that add transformations.

`d3d_transform_set_identity()` Sets the transformation to the identity (no transformation).
`d3d_transform_set_translation(xt,yt,zt)` Sets the transformation to a translation over the indicated vector.
`d3d_transform_set_scaling(xs,ys,zs)` Sets the transformation to a scaling with the indicated amounts.
`d3d_transform_set_rotation_x(angle)` Sets the transformation to a rotation around the x-axis with the indicated amount.
`d3d_transform_set_rotation_y(angle)` Sets the transformation to a rotation around the y-axis with the indicated amount.
`d3d_transform_set_rotation_z(angle)` Sets the transformation to a rotation around the z-axis with the indicated amount.
`d3d_transform_set_rotation_axis(xa,ya,za,angle)` Sets the transformation to a rotation around the axis indicated by the vector with the indicated amount.
`d3d_transform_add_translation(xt,yt,zt)` Adds a translation over the indicated vector.
`d3d_transform_add_scaling(xs,ys,zs)` Adds a scaling with

the indicated amounts.
d3d_transform_add_rotation_x(angle) Adds a rotation around the x-axis with the indicated amount.
d3d_transform_add_rotation_y(angle) Adds a rotation around the y-axis with the indicated amount.
d3d_transform_add_rotation_z(angle) Adds a rotation around the z-axis with the indicated amount.
d3d_transform_add_rotation_axis(xa,ya,za,angle) Adds a rotation around the axis indicated by the vector with the indicated amount.

Realize that rotation and scaling are with respect to the origin of the world, not with respect to the object that is to be drawn. If the object is not at the origin it will also move to a different place, which is not what we want. So to e.g. rotate an object over its own x-axis, we must first translate it to the origin, next rotate it, and finally translate it back to its position. This is what the functions to add transformations are for.

The following examples might explain this better. Assume we have a sprite `spr` that we want to draw at position (100,100,10). We can use the following code to do this

```
{
  d3d_transform_set_translation(100,100,10);
  draw_sprite(spr,0,0,0);
  d3d_transform_set_identity();
}
```

Note that because we use a translation we should now draw the sprite at position (0,0). (This assumes the current instance has a depth of 0! If you are not sure, first set the depth.) If we would use this in our first person shooter we would not see the sprite. The reason is that it is still parallel to the xy-plane. We want to rotate it over 90 degrees along

the x-axis (or y-axis). So we need to add a rotation. Remember the order: we must first rotate the sprite and then translate it. So we can use the following code.

```
{
  d3d_transform_set_identity();
  d3d_transform_add_rotation_x(90);
  d3d_transform_add_translation(100,100,10);
  draw_sprite(spr,0,0,0);
  d3d_transform_set_identity();
}
```

Sometimes you temporarily want to save the current transformation, for example to add an additional transformation and then restore the old one (this often happens when drawing hierarchical models). To this end you can push the current transformation on a stack and later pop it from the stack to make it the current transformation again. The following functions exist for this:

d3d_transform_stack_clear() Clears the stack of transformations.

d3d_transform_stack_empty() Returns whether the transformation stack is empty.

d3d_transform_stack_push() Pushes the current transformation on the stack. Returns whether there was room on the stack to push it there (if you forget popping transformation you at some moment will run out of room on the stack).

d3d_transform_stack_pop() Pops the top transformation from the stack and makes it the current one. Returns whether there was a transformation on the stack.

d3d_transform_stack_top() Makes the top transformation the current one, but does not remove it from the stack. Returns whether there was a transformation on the stack.

`d3d_transform_stack_discard()` Removes the top transformation from the stack but does not make it the current one. Returns whether there was a transformation on the stack.

Using transformation is a powerful mechanism. But be careful and always set the transformation back to the identity once you are done.

Fog

Fog can be used in 3D games to make objects in the distance look blurred or even disappear. This helps in creating atmosphere and it makes it possible to not draw objects that are far away. To enable or disable fog use the following function:

`d3d_set_fog(enable,color,start,end)` Enables or disables the use of fog. `color` indicates the fog color. `start` indicates the distance at which fog must start. `end` indicates the distance at which fog is maximal and nothing can be seen anymore.

To better understand what is happening, there are actually two types of fog, table based fog and vertex based fog. The first type calculates fog values on a pixel basis. The second type calculates the fog value for each vertex and then interpolates these. The first type is better but not always supported. *Game Maker* tries to use table based fog when supported and otherwise uses vertex based fog (unless no fog is supported). Note that certain graphics card indicate that they can handle table based fog but offer the user the possibility to switch this off in the advanced display settings. In this case the result might be a black screen!

Lighting

Scenes you draw with the functions above look rather flat because there is no light. The color if the faces is equal, independent of their orientation. To create more realistically looking scenes you must enable lighting and place lights at the correct places. Creating correctly lit scenes is not easy but the effect is very good.

To enable lighting you can use the following function;

`d3d_set_lighting(enable)` Enables or disables the use of lighting.

When using lighting, for each vertex of a polygon the color is determined. Next, the color of internal pixels is based on the color of these vertices. There are two ways this can be done: Either the whole polygon gets the same color, or the color is smoothly interpolated over the polygon. Default smooth shading is used. This can be changed using the following function:

`d3d_set_shading(smooth)` Set whether to use smooth shading or not.

To use lighting you obviously need to define lights. Two different lights exist: directional lights (like the sun), and positional lights. Light have a color. (We only support diffuse light, not specular reflection.) The following functions exist to define and use lights:

`d3d_light_define_direction(ind,dx,dy,dz,col)` Defines a directed light. `ind` is the index of the light (use a small positive number). (dx,dy,dz) is the direction of the light.

`col` is the color of the light (often you want to use `c_white`. This function does not turn the light on.
`d3d_light_define_point(ind,x,y,z,range,col)` Defines a point light. `ind` is the index of the light use a small positive number). (x,y,z) is the position of the light. `range` indicates till how far the light shines. The intensity of the light will decrease over this range. `col` is the color of the light. This function does not turn the light on.
`d3d_light_enable(ind,enable)` Enables (true) or disables (false) light number `ind`.

The way an object reflects light depends on the angle between the light direction and the normal of the surface, that is, the vector pointing away from the surface. Hence, to create lighted objects you need not only provide the position of the vertices but also their normals. For this four additional functions are avaliable to define the vertices of primitives:

`d3d_vertex_normal(x,y,z,nx,ny,nz)` Add vertex (x,y,z) to the primitive, with normal vector (nx,ny,nz).
`d3d_vertex_normal_color(x,y,z,nx,ny,nz,col,alpha)` Add vertex (x,y,z) to the primitive, with normal vector (nx,ny,nz), and with its own color and alpha value.
`d3d_vertex_normal_texture(x,y,z,nx,ny,nz,xtex,ytex)` Add vertex (x,y,z) to the primitive, with normal vector (nx,ny,nz), and with position (xtex,ytex) in the texture, blending with the color and alpha value set before.
`d3d_vertex_normal_texture_color(x,y,z,nx,ny,nz,xtex,ytex,col,alpha)` Add vertex (x,y,z) to the primitive, with normal vector (nx,ny,nz), and with position (xtex,ytex) in the texture, blending with its own color and alpha value.

Note that for the basic shapes that you can draw the normals are automatically set correctly.

Creating models

When you need to draw large models it is rather expensive to call all the different drawing functions again and again in every step. To avoid this you can create models. A model consists of a number of drawing primitives and shapes. Once a model is created you can draw it at different places with just one function call. Models can also be loaded from a file or saved to a file.

Before giving the different functions available there is one important point: the handling of textures. As described earlier, texture are taken from sprites and backgrounds. The indices of the textures can be different at different moments. As a result models do not contain any texture information. Only when you draw a model you provide the texture. So you can only use one texture in a model. If you need more textures you must either combine them in one (and carefully deal with the texture coordinates) or you must use multiple models. The advantage of this is that you can draw the same model easily with different textures.

For creating, loading, saving, and drawing models, the following functions exist:

- **`d3d_model_create()`** Creates a new model and returns its index. This index is used in all other functions dealing with models.
- **`d3d_model_destroy(ind)`** Destroys the model with the given index, freeing its memory.
- **`d3d_model_clear(ind)`** Clears the model with the given index, removing all its primitives.
- **`d3d_model_save(ind,fname)`** Saves the model to the indicated file name.

d3d_model_load(ind,fname) Loads the model from the indicated file name.

d3d_model_draw(ind,x,y,z,texid) Draws the model at position (x,y,z). `texid` is the texture that must be used. Use -1 if you do not want to use a texture. If you want to rotate or scale the model you can use the transformation routines described before.

For each primitive function there is an equivalent to add it to a model. The functions have the same arguments as before except that each has a first argument the index of the model, and no texture information is provided.

d3d_model_primitive_begin(ind,kind) Adds a 3D primitive to the model of the indicated kind: `pr_pointlist`, `pr_linelist`, `pr_linestrip`,`pr_trianglelist`,`pr_trianglestrip` or `pr_trianglefan`.

d3d_model_vertex(ind,x,y,z) Add vertex (x,y,z) to the model.

d3d_model_vertex_color(ind,x,y,z,col,alpha) Add vertex (x,y,z) to the model, with its own color and alpha value.

d3d_model_vertex_texture(ind,x,y,z,xtex,ytex) Add vertex (x,y,z) to the model with position (xtex,ytex) in the texture.

d3d_model_vertex_texture_color(ind,x,y,z,xtex,ytex,col,alpha) Add vertex (x,y,z) to the model with texture and color values.

d3d_model_vertex_normal(ind,x,y,z,nx,ny,nz) Add vertex (x,y,z) to the model, with normal vector (nx,ny,nz).

d3d_model_vertex_normal_color(ind,x,y,z,nx,ny,nz,col,alpha) Add vertex (x,y,z) to the model, with normal vector (nx,ny,nz), and with its own color and alpha value.

d3d_model_vertex_normal_texture(ind,x,y,z,nx,ny,nz,xtex,ytex) Add vertex (x,y,z) to the model, with normal vector

(nx,ny,nz), with texture position.

d3d_model_vertex_normal_texture_color(ind,x,y,z,nx,ny,nz,xtex,ytex,col,alpha) Add vertex (x,y,z) to the model, with normal vector (nx,ny,nz), with texture and color values.

d3d_model_primitive_end(ind) End the description of the primitive in the model.

Besides primitives you can also add basic shapes to the models. Again the functions look almost the same but with a model index and without texture information:

d3d_model_block(ind,x1,y1,z1,x2,y2,z2,hrepeat,vrepeat) Adds a block shape to the model.

d3d_model_cylinder(ind,x1,y1,z1,x2,y2,z2,hrepeat,vrepeat,closed,steps) Adds a cylinder shape to the model.

d3d_model_cone(ind,x1,y1,z1,x2,y2,z2,hrepeat,vrepeat,closed,steps) Adds a cone shape to the model.

d3d_model_ellipsoid(ind,x1,y1,z1,x2,y2,z2,hrepeat,vrepeat,steps) Adds a ellipsoid shape to the model.

d3d_model_wall(ind,x1,y1,z1,x2,y2,z2,hrepeat,vrepeat) Adds a wall shape to the model.

d3d_model_floor(ind,x1,y1,z1,x2,y2,z2,hrepeat,vrepeat) Adds a floor shape to the model.

Using models can considerable speed up the graphics in your 3D games and you should use them whenever you can.

Final words

The 3D functions in *Game Maker* can be used to make some nice 3D games. However, they are limited in functionality and still leave quite a lot of work to you. Don't expect that you can make your own *Quake* with it. *Game Maker* is and remains primarily a package for making 2-dimensional games.