Designing Games with Game Maker

version 5.0 (April 14, 2003)

Written by Mark Overmars

Table of Contents

Chapter 1 So you want to create your own computer games

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

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

But here come *Game Maker*. *Game Maker* has been written to make it a lot easier to create such games. There is no need to program. An intuitive and easy to use drag-and-drop interface allows you to create your own games very quickly. You can import and create images, sprites (animated images) and sounds and use them. You easily define the objects in your game and indicate their behavior. And you can define appealing rooms with scrolling backgrounds in which the game take place. And if you want full control there is actually 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 no 3-D worlds like *Quake*. But don't let this put you down. Many great games, like *Age of Empires*, the *Command & Conquer* series, and *Diablo* use two-dimensional sprite technology, even though they look very 3-dimensional. And designing two-dimensional games is a lot easier and faster.

Probably the best part is that *Game Maker* can be used free of charge. And there are no restrictions on the games you create with it. In the games no nag screen will be shown, and you can even sell them if you like. See the enclosed license agreement for more details. You are though strongly encouraged to register your copy of *Game Maker*. This will 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 trivial. There are too many aspects that are important: game play, graphics, sounds, user interaction, etc. Start with easy examples and you will realize that creating games is great fun. Also check the web site

http://www.gamemaker.nl/

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

Chapter 2 Installation

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

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 better use simple mode (so select **No**). In simple mode fewer options are shown. You can easily switch to advanced mode letter using the appropriate item in the **File** menu.

Within the installation folder (default `C:/Program Files/Game_Maker5/`) there will be a number of other folders:

- `examples`: contains a number of example games, for you to check and/or change.
- `lib`: contains a number of libraries of actions. If you want to install additional action libraries you must put them in this folder.
- `sprites`: this folder is meant to contain sprites you can use. The default installation does install just a few sprites but from the *Game Maker* website (http://www.gamemaker.nl/) you can load a number of resource packs that contain additional sprites, sounds, backgrounds, etc.
- `backgrounds`, `sounds`: similar folders that are meant to contain the background images and sounds.

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

Chapter 3 Registration

As indicated above, *Game Maker* can be used free of charge. There are no restrictions on the games you create with it. In the games no nag screens are shown and you can even sell the games if you like. See the enclosed license agreement for more details.

You are though highly recommended to register your copy of *Game Maker*. Through registration you help the further development of the program. It will also remove the nag screen in the creator. In the future we plan further benefits for registered users, for example a game competition.

The registration fee for *Game Maker* is US $15 or 15 Euro. There are a number of ways in which you can register your copy of the program. The easiest way is to use online registration use a secure credit card payment system or a PayPal account. Alternatively you can transfer money to our bank account, send us a money order or send cash. Details can be found on the *Game Maker* registration web site:

www.gamemaker.nl/registration.html

To register your copy of *Game Maker* use the web site above or choose **Registration** from the **Help** menu. At the bottom of the form that appears click the button **Registration**. You will be brought to our web page were the different registration options are indicated, including the online registration.

Once your registration has been received an email will be send to you with the name and key and information how to enter the key in the program. To enter the key, again choose **Registration** from the **Help** menu. At the bottom of the form press the button **Enter Key**. Type the name and key and press **OK**. If you made no mistakes the program is registered.

Chapter 4 The global idea

Before delving into the possibilities of *Game Maker* it is good to first get a feeling for the global idea behind the program. Games created with *Game Maker* take place in one or more *rooms.* (Rooms are flat, not 3D, but they can contain 3D-looking graphics.) In these rooms you place *objects*, which you can define in the program. Typical objects are the walls, moving balls, the main character, monsters, etc. Some objects, like walls, just sit there and don't do anything. Other objects, like the main character, will move around and react to input from the player (keyboard, mouse, 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 call *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 sprite in *Game Maker* or load them from files (e.g. animated GIF's).

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

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

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

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

- *objects*: which are the true entities in the game
- *rooms*: the places (levels) in which the objects live
- *sprites*: (animated) images that are used to represent the objects
- *sounds*: these an 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, data files, 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.

Chapter 5 Let us look at an example

It is good to first 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 ball that is bouncing around between some walls. The player should try to click on the ball with the mouse. Each time he succeeds he gets a point.

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

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

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

The next step is to create the two objects. Let us first make the wall object. Again from the **Add** menu choose **Add Object**. A form will open that looks quite a bit more complex than the ones we saw so far. At the left there is some global information about the object. Give the object an appropriate name and from the drop down menu pick the correct wall sprite. Because a wall is solid you should check the box labeled **Solid**. That is all for the moment. Again create a new object, name it ball, and give it the ball sprite. We don't make the ball solid. For the ball we need to define some behavior. In the middle you see 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 **creation** 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 the 8 arrows to choose a random direction. You can leave the speed as 8. Now close the dialog. So now the ball will start moving at the moment it is created. Secondly we have to define what should happen in the case of a collision event with the wall. 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 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 move the ball to a new random position and moving in a new direction (in the same way as in the creation event). For the sound action, select the correct sound. For the score action, type in a value of 1 and check the **Relative** box. This means that 1 is added to the current score. (If you make a mistake you can double click the action to change its settings.)

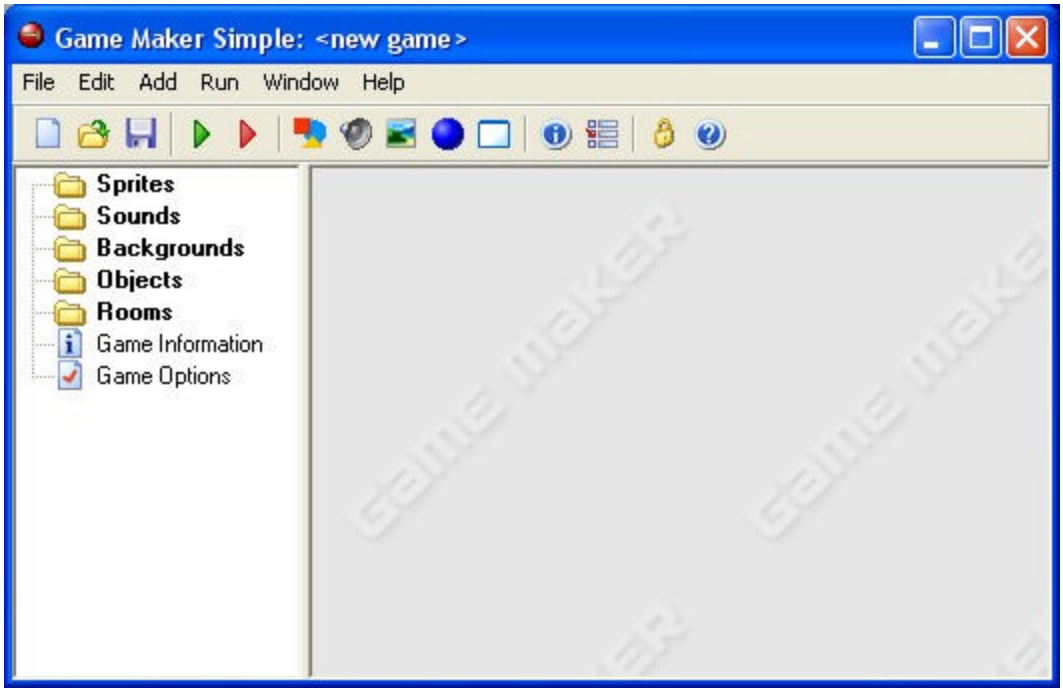
Our objects are now ready. What remains is to define the room. Add a new room to the game, again from the **Add** menu. At the right you see the empty room. At the left you find some tabs, one for setting the background, one for setting some global properties like the width and height of the room, and one were 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 and see what happens. If you made no mistakes the ball starts moving around. Try clicking on it with the mouse and see what happens. You can stop the game by pressing the <Esc> key. You can now make further changes.

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

Chapter 6 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. See Chapter 14 for details.) At the left you see the different *resources*, mention above: Sprites, Sounds, Backgrounds, Objects, Rooms and two more: Game Information and Game Options. At the top there is the well-known menu and toolbar. In this chapter I will describe briefly the various menu items, buttons, etc. In the later chapters we discuss a number of them in detail. Note that many things can be achieved in different ways: by choosing a command from the menu, by clicking a button, or by right clicking on a resource.

6.1 File menu

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

- **New**. Choose this command to start creating a new game. If the current game was changed you are asked whether you want to save it. There is also a toolbar button for this.
- **Open**. Opens a game file. *Game Maker* files have the extension .gmd. There is a toolbar button for this command. You can also open a game by dragging the file to the *Game Maker* window.
- **Recent Files**. Use this submenu to reopen game files your 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 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. You will find more information on distributing games in Chapter 14.
- **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.

6.2 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 might not be available.

- **Insert resource**. Insert 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**. Give the resource a new name. This can also be done in the property form for the resource. Also you can select the resource and then click on the name.
- **Properties**. Use this command to bring up the form to edit the properties. Note that all the property forms appear within the main form. You can edit many of them at the same time. You can also edit the properties by double clicking on the resource.

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.

6.3 Add menu

In this menu you can add 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.

6.4 Window menu

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

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

6.5 Help menu

Here you find some commands to help you:

- **Contents**. Here you can access the on-line version of this document.
- **How to use help**. In case you do not know, some help on using help.
- **Registration**. Even though *Game Maker* can be used for free, you are encouraged to register the program. It will remove the nag screen that sometimes occur and will help the further development of the program. Here you can find information on how to register the program. If you did register it you can use this to enter the registration key you receive.
- **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*. I recommend that you check out the site at least once a month for new information.
- **Forum**. Connects you to the *Game Maker* forum where many people will help you with your questions.
- **About Game Maker**. Give some short information about this version of *Game Maker*.

6.6 The resource explorer

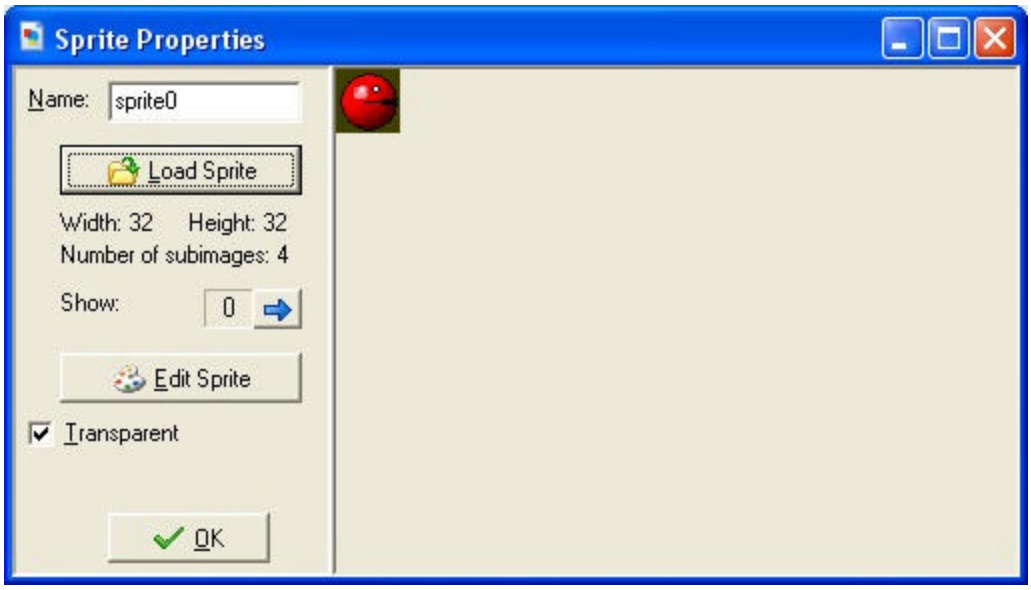
At the left of the main form you find the resource explorer. Here you will see a tree-like view of all resources in your game. It works in the same way as the windows explorer and you are most likely familiar with it. If an item has a + sign in front of it you can click on the sign to see the resources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

Chapter 7 Defining sprites

Sprites are the visual representations of all the objects in the game. A sprite is either a single image, drawn with any drawing program you like, or a set of images that, when played one of the other, 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 add a sprite, choose the item **Add Sprite** from the **Add** menu, or use the corresponding button on the toolbar. The following form will pop up.



At the top you can indicate the name of the sprite. All sprites (and all other resources) have a name. You best give each sprite a descriptive name. Make sure all resources get different names. Even though this is not strictly required, you are strongly advised to only use letters and digits and the underscore symbol (_) in a name of a sprite (and any other resource) and to let is 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 indicate the sprite. *Game Maker* can load many different graphics files. When you load an animated gif, the different sub-images form the sprite images. Once the sprite

is loaded the first sub-image is shown on the right. When there are multiple sub-images, you can cycle through them using the arrow buttons.

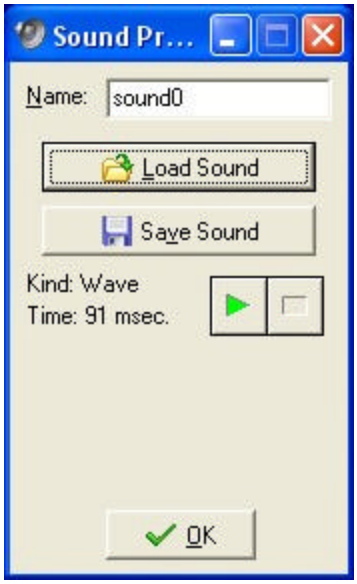
The checkbox labeled **Transparent** indicates whether the background should be considered as being transparent. Most sprites are transparent. The background is determined by the color of the leftmost bottommost pixel of the image. So make sure that no pixel of the actual image has this color. (Note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

With the button **Edit Sprite** you can edit the sprite, or even create a completely new sprite. For more information on creating and changing sprites, see Chapter 14.

Chapter 8 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 add a sound resource to your game, use the item **Add Sound** in the **Add** menu or use the corresponding button on the toolbar. The following form will pop up.



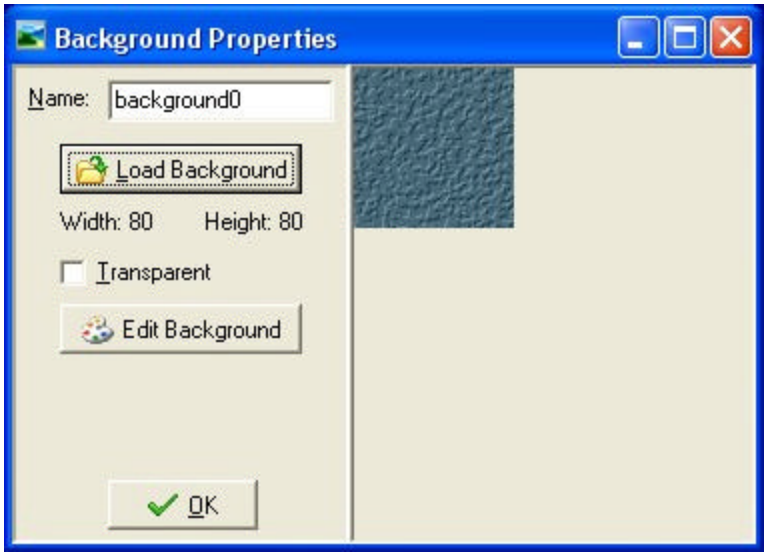
To load a sound, press the button labeled **Load Sound**. A file selector dialog pops up in which you can select the sound file. There are two types of sound files, wave files and midi files. (For information on mp3 files see Chapter 17.) Wave files are used for short sound effects. They use a lot of memory but play instantaneously. Use these for all the sound effects in your game. Midi files describe music in a different way. As a result they use a lot less memory, but they are limited to instrumental background music. Also, only one midi sound can play at any time.

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

Chapter 9 Backgrounds

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

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



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

You can change the background or create a new one using the button **Edit Background**. For more information, see Chapter 18.

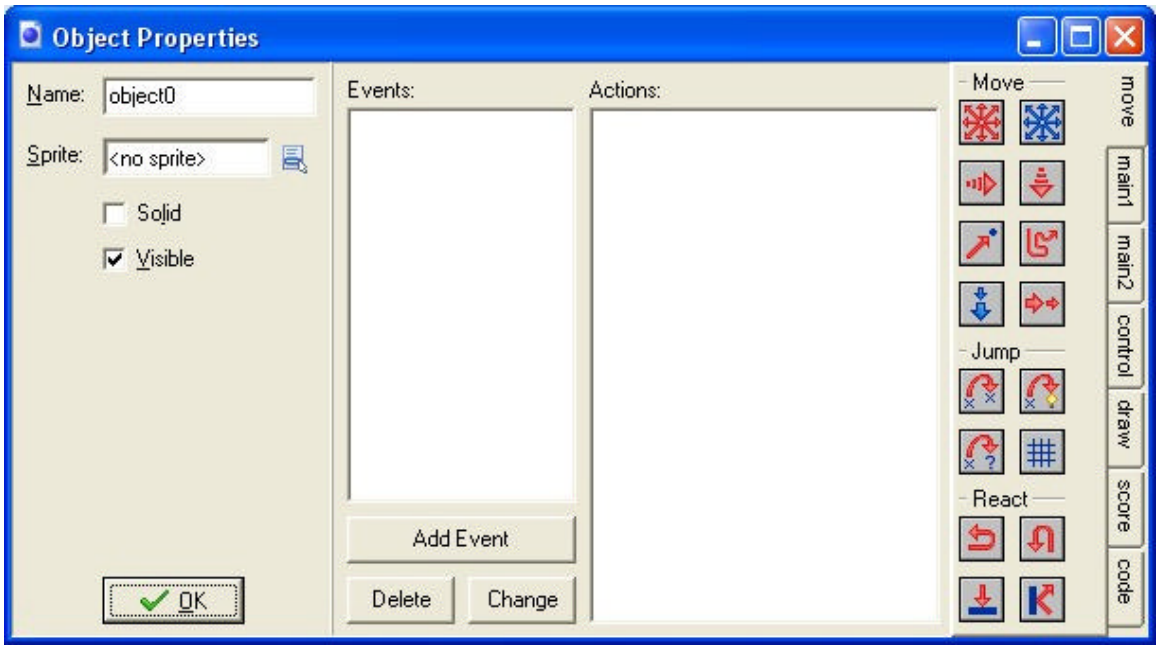
Chapter 10 Defining objects

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. They most of the time have a sprite as a graphical representation such that you see them. They have behavior because they can react to certain events. All things you see in the game (except for the background) are objects. (Or to be more precise, they are instances of objects.) The characters, the monsters, the balls, the walls, etc. are all objects. There might also be certain objects that you don't see but that control certain aspects of the game play.

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

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

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



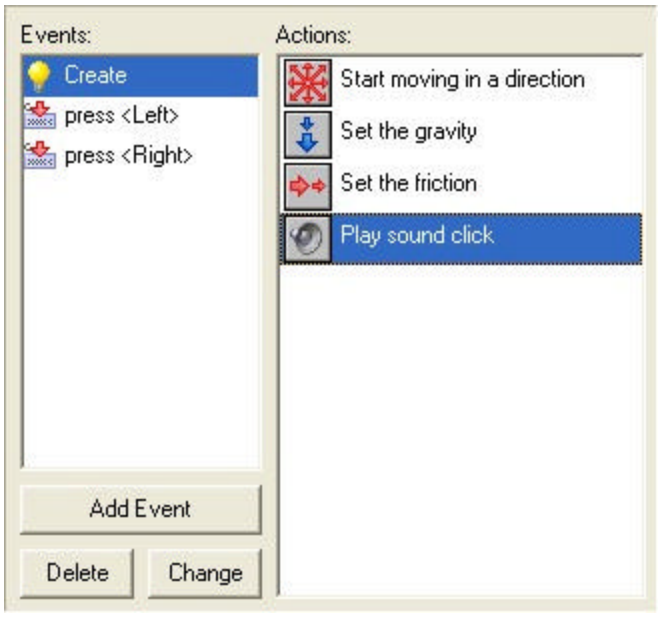
This is rather complex. At the left there is some general information about the object. In the middle there is the list of events that can happen to the object. See the next chapter for details. At the right there are the different actions the object can perform. These will be treated in Chapter 12.

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

Chapter 11 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 will 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 will 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 <Ctrl> key while dragging, you make a copy of the action. You can even use drag-and-drop between action lists for different objects. When you click with the right mouse button on an action, a menu appears in which you can delete the action (can also be done by using the <Del> key) or copy and paste actions. When you hold your mouse 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!)

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 e.g. use it to change the score or to create some other object.

Alarm events
Each instance has 8 alarm clocks. You can set these alarm clocks using certain actions (see next chapter). The alarm clock then ticks down until it reaches 0 at which moment

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

Step 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 for e.g. changing the sprite depending on the current direction.

Collision events

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

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

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

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

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

Keyboard events

When the player presses a key, a keyboard event happens for all instances of all objects. There is a different event for each key. In the menu you can pick the key for which you want to define the keyboard event and next drag actions there. Clearly, only a few objects need events for only a few keys. You get an event in every step as long as the player keeps down the key. There are two special keyboard events. One is called <No key>. This event happens in each step when no key is pressed. The second one is called <Any key> and happens whenever some key is pressed. 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 press and global release events instead. 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. 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.
- **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. This can be used to e.g. 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. See Chapter 21 for more information on paths.
- **User defined**: There are eight of these events. They normally never happen unless you yourself call them from a piece of code.

 **Drawing event**

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

Chapter 12 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 default actions. Additional actions might become available in the form of action libraries. These extend the possibilities of *Game Maker* further. Check the website for possible additional action libraries.

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

When you drop an action 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 I 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. In this way you can e.g. destroy the other instance. Finally, you can choose to apply the action to all instances of a particular object. In this way you can e.g. change all red balls into blue balls. The second type of parameter is the box labeled **Relative**. By checking this box, the values you type in are relative to the current values. For example, in this way you can add something to the current score, rather than changing the current score to the new value. The other parameters will be described below. You can later change the parameters by double clicking on the action.

12.1 Move actions

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

 **Start moving in a direction**

Use this action to start the instance moving in a particular direction. You can indicate the direction using the arrow keys. Use the middle button to stop the motion. Also you need to specify the speed of the motion. This speed is given in pixels per step. The default value is 8. Preferably don't use negative speeds. You can specify multiple directions. In

this case a random choice is made. In this way you can e.g. let a monster start moving either left or right.

 **Set direction and speed of motion**

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

 **Set the horizontal speed**

The speed of an instance consists of a horizontal part and a vertical part. With this action you can change the horizontal speed. A positive horizontal speed means a motion to the right. A negative one 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).

 **Set the vertical speed**

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

 **Move towards a point**

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

 **Set a path for the instance**

(Only available in advanced mode.) With this action you can indicate that the instance should follow a particular path. You indicate the path, the speed and the position in the path where to start (0=beginning, 1=end). See Chapter 21 for more information on paths.

 **Set the gravity**

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

direction. Note that, contrary to real life, different object can have different gravity directions.

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

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

 **Jump to a random position**
This action 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 can be used to e.g. keep the instance aligned with the cells in your game (if any). You can specify a separate horizontal snapping and vertical snapping.

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

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

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

 **Move to contact position**
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 against objects**

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

12.2 Main actions, set 1

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

 **Create an instance of an object**

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

 **Change the instance**

With this action you can change the current instance into an instance of another object. So for example, you can change an instance of a bomb into an explosion. All settings, like 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 the instance**

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

 **Destroy instances at a position**

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

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

 **Play a sound**
With this action you play one of the sound resources you added to your game. You can indicate the sound you want to play and whether it should play once (the default) or loop continuously. Multiple wave sounds can play at once but only one midi sound can play. So if you start a midi sound, the current midi sound is stopped. Unless the sound has multiple buffers (see Chapter 17) only one instance of each sound can play. So if the same sound is already playing it is stopped and restarted.

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

 **If a sound is playing**
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. For more information on actions that test certain questions, see Section **Error! Reference source not found.**.

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

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

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

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

 **If previous room exists**
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.

 **If next room exists**
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.

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

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

 **Set a 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.

 **Set the 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.

 **Display a message**
With this action you can display a message in a dialog box. You simply type in the message. If you use a # symbol in the message text it will be interpreted as a new line

character. (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. (Note that this action does not work when your game runs in exclusive mode, see Chapter 26.)

 **Show the game information**
With this action you pop up the game information window. See Chapter 25 for more information on how to create the game information. (This action does not work when your game runs in exclusive mode.)

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

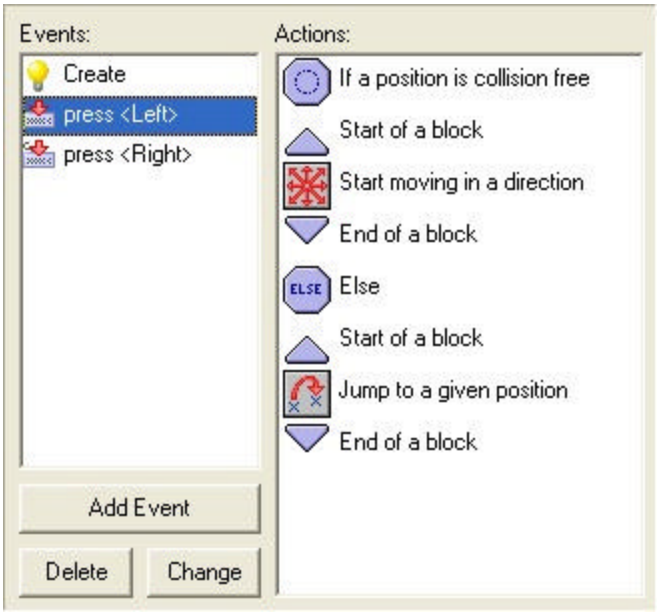
 **End the game**
With this action you end the game.

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

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

12.4 Control

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 such that they can more easily be distinguished from other actions.)

If a position is collision free

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

If there is a collision at a position

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

 **If there is an object at a position**
This question returns true if the instance placed at the indicate position meets an instance of the indicated object.

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

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

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

 **If an expression is true**
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.

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

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

 **Else**

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

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

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

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

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

12.5 Drawing actions

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

 **Draw a sprite image**
You specify the sprite, the position (either absolute or relative to the current instance position) and the sub-image of the sprite. (The sub-images are number from 0 upwards.) If you want to draw the current sub-image, use number –1.

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

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

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

 **Draw a line**

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

 **Draw a 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.)

 **Set the colors**
Lets you set the color used to fill the rectangles and ellipses and the color used for the lines around the rectangle and ellipse and when drawing a line.

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

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

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

 **If score has a value**
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 the value of 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.

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

 **Display the highscore table**
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. So you should not first add the score with the previous action. 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. (This action does not work in exclusive mode!)

 **Set the number of 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.

 **If lives is a value**
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 the number of 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 the lives as image**
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 the 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.

 **If health is a value**
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 the health bar**
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.

 **Set the window caption information**
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.

12.7 Code related actions

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

 **Execute a script**
(Only available in advanced mode.) 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. See Chapter 21 for more information about scripts.

 **Execute a piece of code**
When you add this action, a form shows in which you can type in a piece of code. This works in exactly the same way as when defining scripts (see Chapter 21). The only difference is that you can indicate for what instances the piece of code must be executed. Use the code action for small pieces of code. For longer pieces you are strongly advised to use scripts.

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

 **If a variable has a value**
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 the value of a variable**
With this action you can draw the value of a variable at a particular position on the screen.

 **Call the inherited event**
(Only available in advanced mode.) This action is only useful when the object has a parent object (see Chapter 19). It calls the corresponding event in the parent object.

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

12.8 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_scale` the amount the image is scaled (1 = not scaled)
`image_single` this variable indicate which subimage in the current sprite must be shown; if you set it to –1 (default) you loop through the images, otherwise only the indicated subimage (starting with number 0) is shown all the time
`image_speed` this variable indicates the speed with which the sub-images are shown. The default value is 1. If you make this value larger than 1 some sub-images are skipped to make the animation faster. If you make it smaller than 1 the animation becomes slower.
`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.

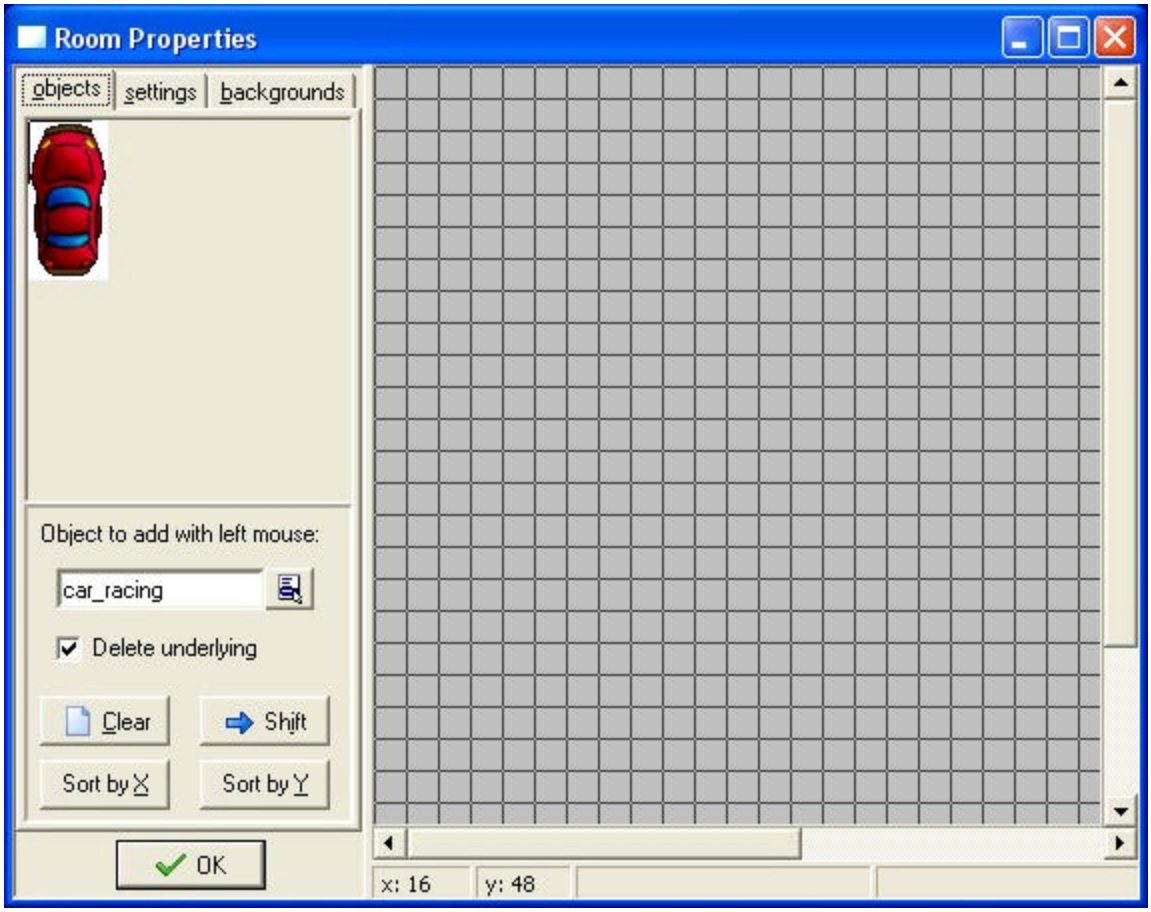
In your expressions you can also use functions. For example, the function `random(10)` gives a random real number below 10. So you can set for example the speed or direction of motion to a random value. Many more functions exist. For more precise information on expressions and functions see Chapter 28 and further.

Chapter 13 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 discusses later in Chapter 20. 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 **Add Room** from the **Add** menu. The following form will appear:



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.

13.1 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 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. (You can change the grid in the settings; see below. If you hold the <Alt> key while placing the instance it is not aligned to the grid.) With the right mouse button you can remove instances. In this way you define the contents of the room. If you hold down the mouse button while dragging it over the room, multiple instances are added or removed.

As you will notice, if you place an instance on top of another one, the original instance disappears. Normally this is what you want, but not always. This can be avoided by unchecking the box labeled **Delete underlying** at the left. There are three other actions you can perform using the right mouse button: When you hold the <Ctrl> key while clicking on an instance with the right mouse button, the bottommost instance at the position is brought to the top. Holding the <Alt> key will send the topmost instance to the bottom. This can be used to change the order of overlapping instances. Finally, holding the <Shift> key while clicking with the right mouse button will remove all instances at the position, not just the top one.

There are four useful buttons in the tab at the left. When you press the **Clear** button all instances are removed from the room. When you press the **Shift** button you can shift all instances over a number of pixels. Use negative numbers to shift them left or up. This is useful when you decided to e.g. enlarge the room. (You can also use this to place instances outside the room, which is sometimes useful.) Finally there are two buttons to sort the instances by X or by Y coordinate. This is useful when instances partially overlap.

13.2 Room setting

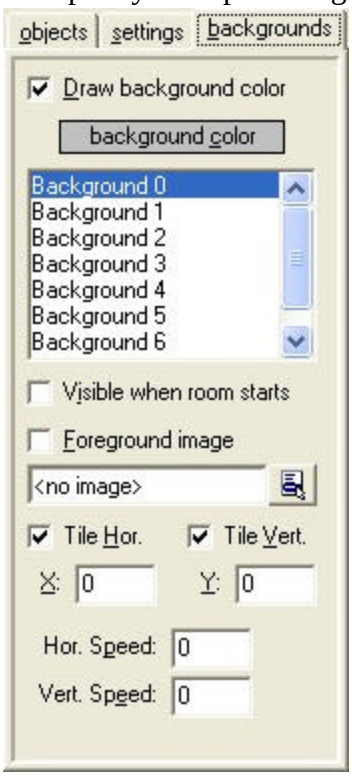
Each room has a number of settings that you can change by clicking on the **settings** tab. We will only consider the most important ones here.

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 rather fast computer to run it.

At the bottom of the **settings** tab you can indicate the size of the grid cells used for aligning objects. By clicking on the button labeled **Show** you can indicate whether to e.g. show the grid lines. (You can also indicate here whether to show the backgrounds, etc. It is sometimes useful to temporarily hide certain aspects of the room.)

13.3 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, you better 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). Finally you can make the background scrolling by giving it a horizontal or vertical speed (pixels per step).

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.

Chapter 14 Distributing your game

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

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

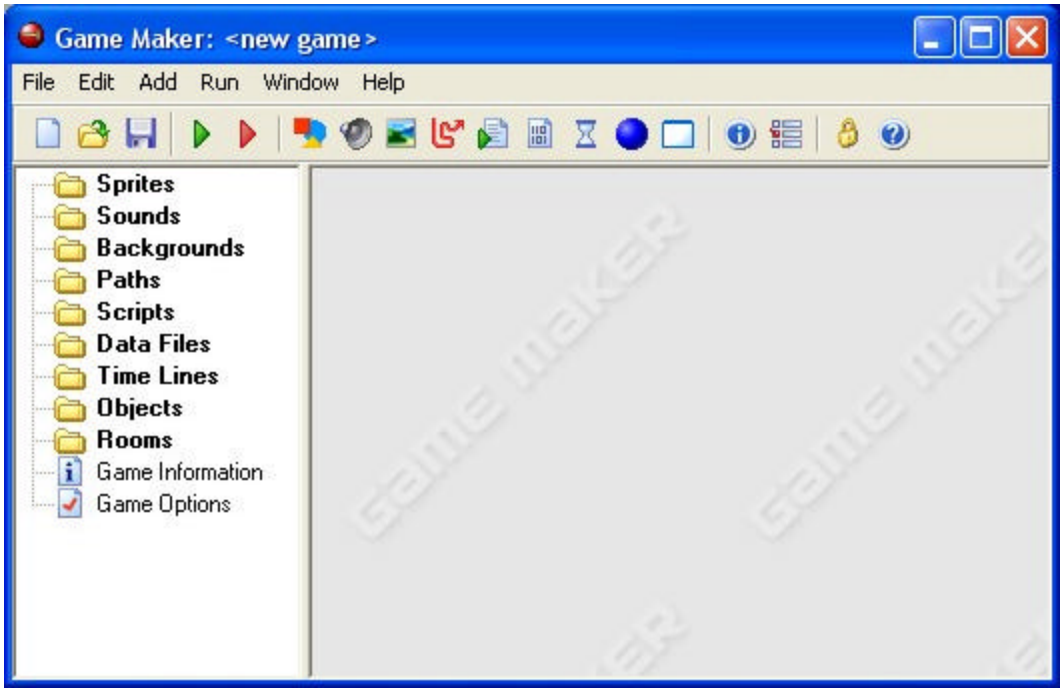
The second way is to create a stand-alone executable of the game. This can be achieved by choosing the item **Create Executable** in the **File** menu. You will be asked for the name of the executable that should contain the game. Indicate a name, press **OK** and you have your stand-alone game that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

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

Chapter 15 Advanced mode

Up to now we 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.

15.1 File menu

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

- **Import scripts**. Can be used to import useful scripts from files. This will be discussed in more detail in Chapter 21.
- **Export scripts**. Can be used to save your scripts in a file, to be used by others. Again see Chapter 21.
- **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*. 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 ba0-ba9. You can open these games in *Game Maker*.
  - **Maximal number of backups**. Here you can indicate how many (1-9) different backup copies should be remembered by the program.
  - **Hide the designer and wait while the game is running**. If checked, when you run the game, the designer window will disappear and come back when the game is finished.
  - **Run games in secure mode**. If checked, any game created with *Game Maker* that runs on your machine will not be allowed to execute external programs or change or delete files at a place different from the game location. (This is a safeguard against Trojan horses.) Checking this might make that certain games don't work correctly.
  - **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.
  - **Show popup at right mouse click in room image**. If checked, a right mouse click in the room will not delete the instance but shows a menu in which you can, among others, set initialization code.
  - **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.
  - **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.)
  - **Scripts and code**. See Chapter 23 for more information about these preferences.
  - **Colors**. See Chapter 23 for more information about these preferences.

- **Image editor**. Default *Game Maker* uses a built-in editor for images. If you have a better other image editing program you can indicate here to use a different program for editing the images.

15.2 Edit menu

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

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

15.3 Add menu

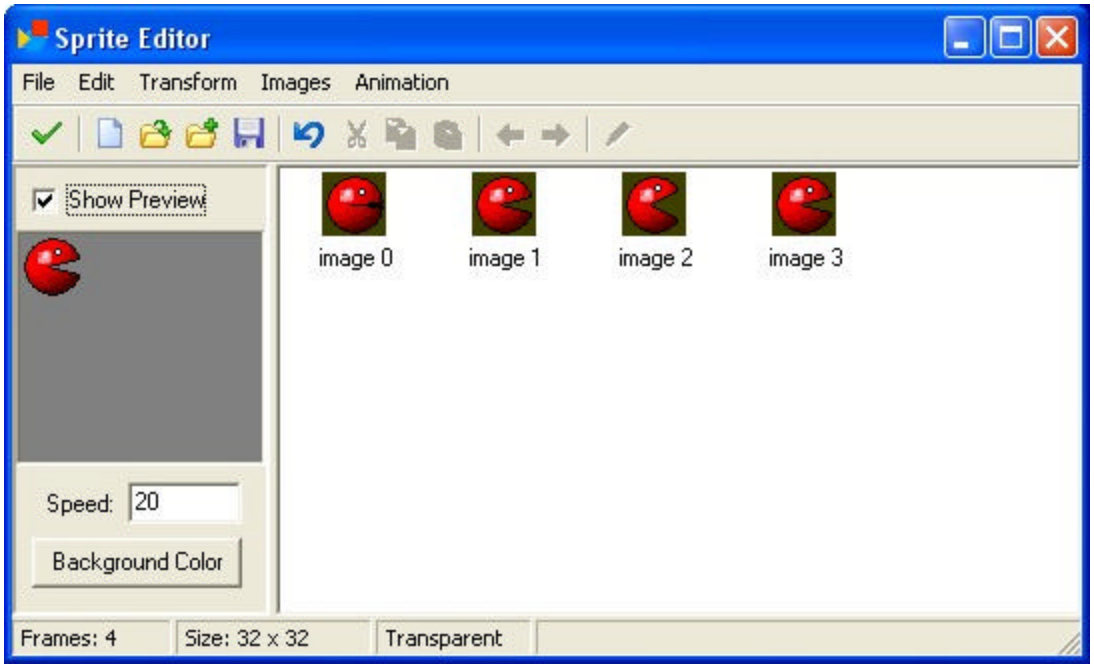
In this menu you can now also add the additional resources. Note that for each of them there is also a button on the toolbar and a keyboard shortcut.

Chapter 16 More about sprites

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

16.1 Editing your sprites

The sprite edit form will look as follows:



At the right you see the different images that make up the sprite. Note that in *Game Maker* all sub-images 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 sub-image with the mouse.

16.1.1 File menu

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

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

16.1.2 Edit menu

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

16.1.3 Transform menu

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

- **Mirror horizontal**. Mirrors the images horizontally.
- **Flip vertical**. Flips the images vertically.
- **Shift**. Here you can shift the images an indicated amount horizontally and vertically.
- **Rotate**. You can rotate the images 90 degrees, 180 degrees, or an arbitrary amount. In the latter case you can also specify the quality. Experiment to get the best effects.
- **Resize Canvas**. Here you can change the size of the canvas. You can also indicate where the old images are placed on the new canvas.
- **Stretch**. Here you can stretch the images into a new size. You can indicate the scale factor and the quality.
- **Scale**. This 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.

16.1.4 Images menu

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

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

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

16.1.5 Animation menu

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

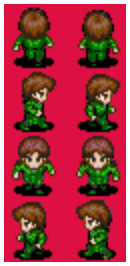
- **Set Length**. Here you can change the length of your animation. The animation is repeated enough times to create the number of frames you indicate. (Normally you want this to be a multiple of the current number of frames.)
- **Stretch**. This command also changes the length of the animation. But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

- **Rotation sequence**. Creates an animation in which the image rotates. You can either choose clockwise or counterclockwise rotation. Specify the number of frames and the total angle in degrees (360 is a complete turn). (You might need to resize the canvas first to make sure the total image remains visible during the rotation.)
- **Colorize**. Creates an animation that turns the image into a particular color.
- **Fade to color**. Creates an animation that fades the image to a particular color.
- **Disappear**. Makes the image disappear using screen-door transparency.
- **Shrink**. Shrinks the image to nothing. You can indicate the direction.
- **Grow**. Grows the image from nothing.
- **Flatten**. Flattens the image to nothing in a given direction.
- **Raise**. Raises the image from a given direction
- **Overlay**. Overlays the animation with another animation or image in a file.
- **Morph**. Morphs the animation to an animation or image from a file. Note that morphing works best if the two animations cover the same area of the image. Otherwise, halfway certain pixels disappear and others suddenly appear.

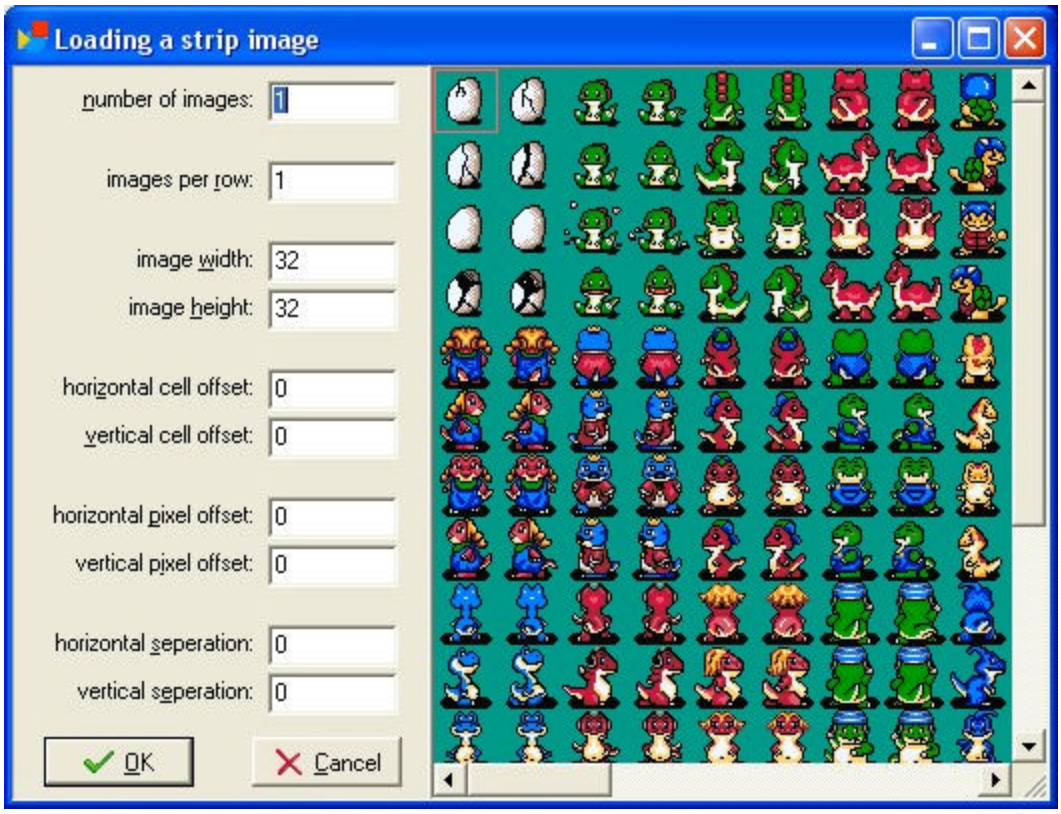
In particular the last two commands are very powerful. For example, to blow up an object, add a number of copies and then a number of empty frames. Then overlay it with an explosion animation. (Make sure the numbers of images match.) Alternatively, morph it to the explosion. With some practice you can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

16.1.6 Strips

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



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



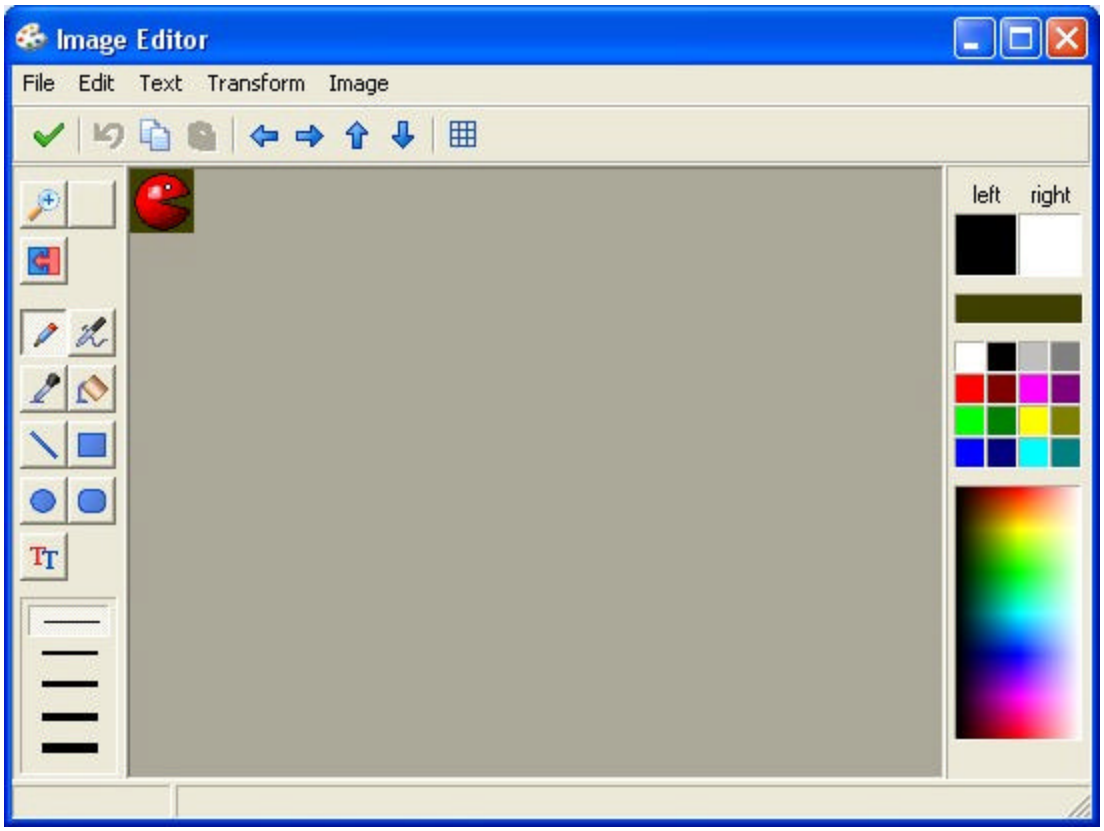
At the right you see (part of) the strip image you selected. At the left you can specify a number of parameters that specify which subimages you are interested in. Note that one or more rectangles in the image indicate the images you are selecting. The following parameters can be specified:

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

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

16.2 Editing individual sub-images

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



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

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

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

16.3 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 sub-images. When the bounding boxes do overlap, it is checked whether two pixels in the current sub-images of the two sprites overlap. This second operation is expensive and requires extra memory and preprocessing. So if you are not interested in precise collision checking for a certain sprite, you should uncheck the box labeled **Precise collision checking**. In this case only bounding box checking is performed. You can also change the bounding box. This is hardly ever required but sometimes you might want to make the bounding box smaller, such that collisions with some extending parts of the sprite are not taken into account.

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

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

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

Chapter 17 More about sounds and music

When you add sound resources to your game there are a number of other aspects that you can indicate. These are only visible in advanced mode.

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

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

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

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 filled these in 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.)

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

Chapter 18 More about backgrounds

Besides loading them from files, you can also create your own backgrounds. To this end, press the button labeled **Edit Background**. A little built-in painting program opens in which you can create or change your background. Please realize that this is not a full-blown program. For more advanced editing tools use some paint program. The built-in paint program is described in Section 16.2. There is one option that is particularly useful. In the **Image** menu you find a command **Gradient Fill**. This can be used to create some nice gradient backgrounds.

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

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

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

Chapter 19 More about objects

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

19.1 Depth

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

19.2 Persistent objects

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

19.3 Parents

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

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

It is actually good practice in such cases to create one base object, for example a ball0 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 you game structured and you are strongly advised to learn to use this mechanism.

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

19.4 Masks

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

19.5 Information

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

Chapter 20 More about rooms

Rooms in *Game Maker* have many options. In Chapter 13 we only treated the most important ones. In this chapter we will discuss the other options.

20.1 Advanced settings

There where two aspects in the **settings** tab that we did not yet discuss. First of all, there is a checkbox labeled **Persistent**. Normally, when you leave a room and return to the same room later, the room is reset to its initial settings. This is fine if you have a number of levels in your game but it is normally not what you want in for example an RPG. Here the room should be the way you left it the last time. Checking the box labeled **Persistent** will do exactly that. The room status will be remembered and when you return to it later, it will be exactly the same as you left it. Only when you restart the game will the room be reset. Actually, there is one exception to this. If you marked certain objects as being persistent (see Chapter 19), 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 e.g. fill in certain variables for the room, create certain instances, etc. It is useful to understand what exactly happens when you move to a particular room in the game.

- First, 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. If you set the correct preference, right clicking on an instance in the room will show a pop-up menu. Here you can give the usual comments to delete it or move it in the depth order, but you can also indicate some creation code. 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.

20.2 Adding tiles

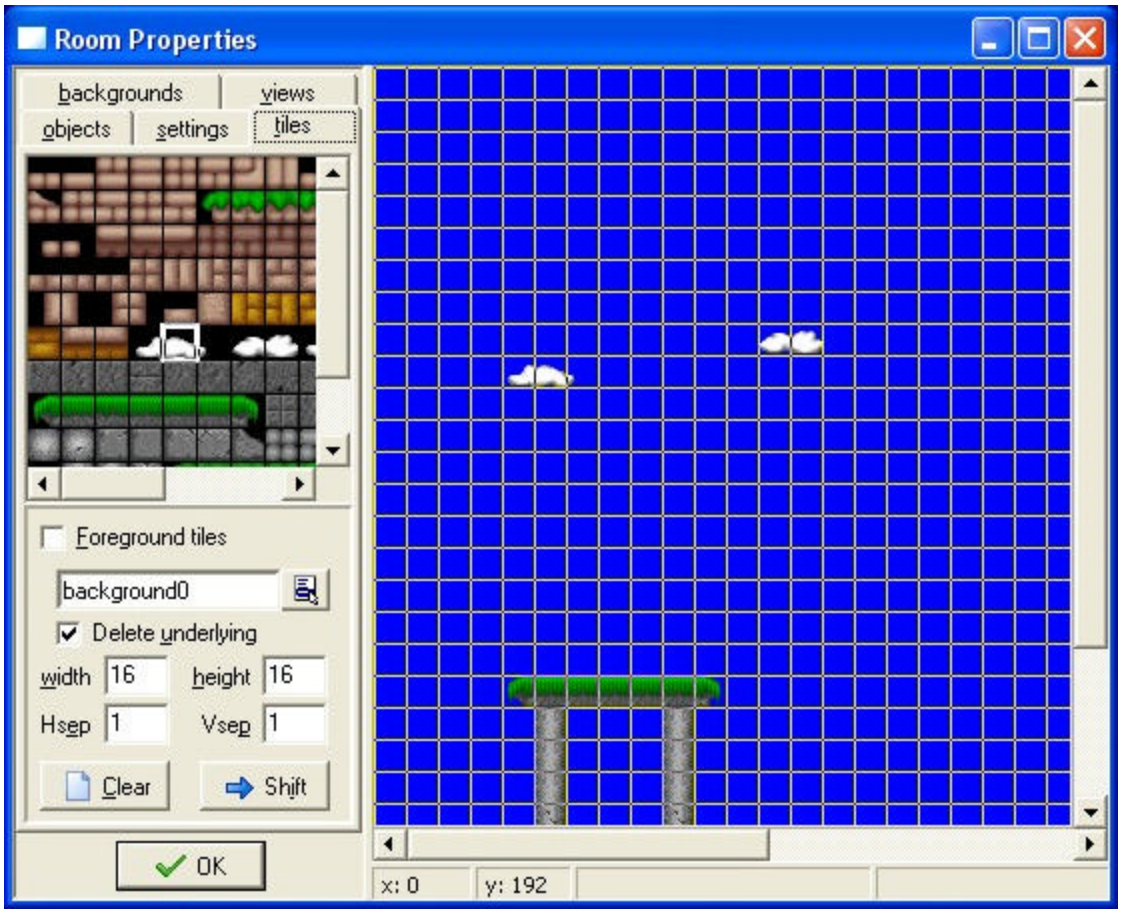
You can also create so-called tiled background. The reason for this is as follows: In many games you like to have nice looking backgrounds. For example, in a maze game, the walls of the maze should nicely match up, and in platform game you like to see beautifully drawn platforms, trees, etc. You can do this in *Game Maker* by defining many different objects and composing your rooms from these objects. The problem though is

that this takes a lot of work, uses large amounts of resources, and makes the games run slow because of the many different objects. For example, to create nice walls in maze games you already need 15 differently shaped wall objects.

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

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

To add tiles to your room you first need to add a background image to your game that contains the tiles. A few of these are provided with *Game Maker*. If you want to have your tiles partially transparent, make sure you make the background image transparent. 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. Below the tile set you can change a number of settings. You can set the width and height of an individual tile, and a separation between the tiles (this is normally 0 or 1).

Now you can add tile 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. There are also buttons to clear all tiles and to shift all tiles.

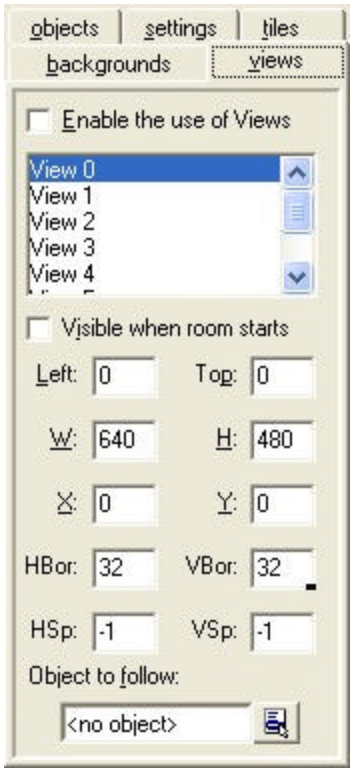
Note that there is a box labeled **Foreground tiles**. If you check this, the tiles will be drawn in front of the objects rather than behind them. This can be used in many ways. Note that when you check the box also only foreground tiles are removed.

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 tiles rooms with very little memory consumption.

20.3 Views

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

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



At the top there is a box labeled **Enable the use of Views**. You much 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. Next you indicate the area of the room that should be shown in the view. You specify the left and top position, and the width and the height of the view. Below that you indicate the position of the view on the screen.

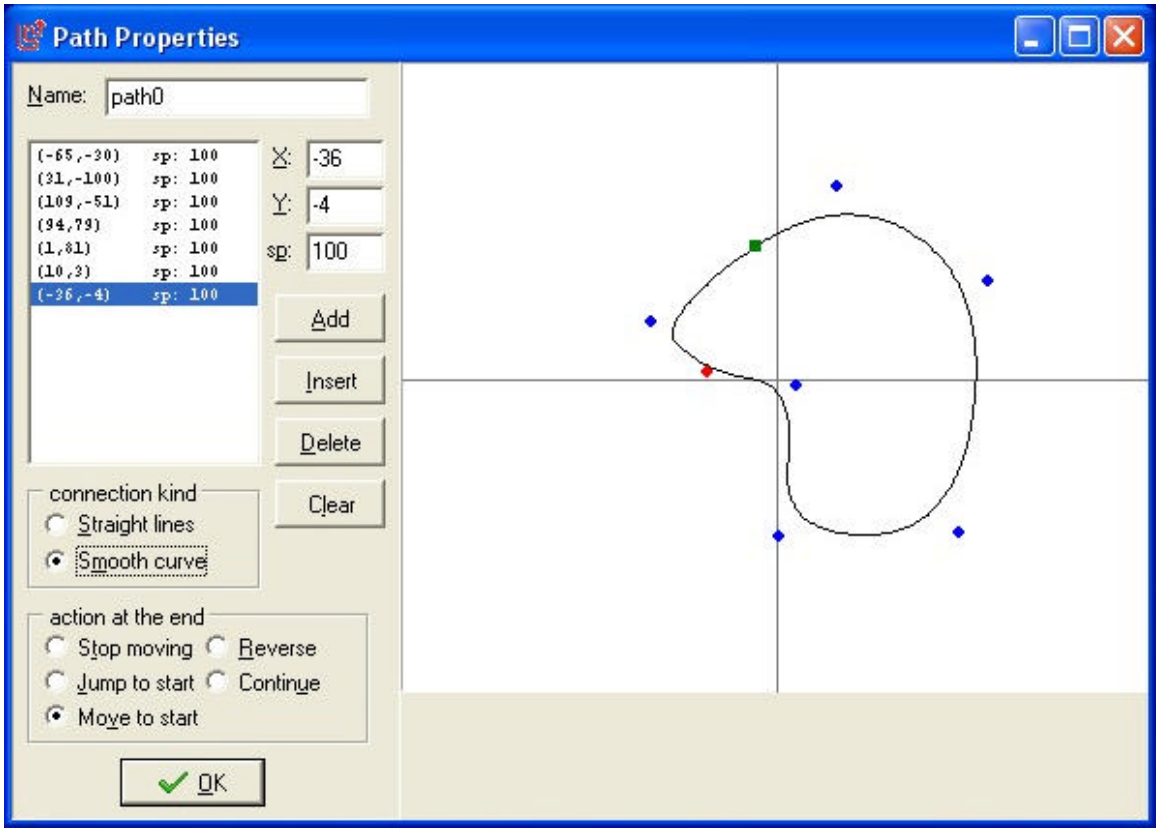
As indicated above, you often want the view to follow a certain object. This object you can indicate at the bottom. If there are multiple instances of this object, only the first one is followed by the view. (In code you can also indicate that a particular instance must be followed.) Normally the character 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 of the screen, but it gives a much smoother game play. Use –1 if you want the view to change instantaneously.

Chapter 21 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 e.g. the creation event of the object to tell the object to follow the particular path. This chapter will explain this in detail. The current implementation is rather limited. Expect more possibilities in future versions (compatible with the current version).

21.1 Defining paths

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



At the left top 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). The position is not absolute. As is indicated below, 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 of 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**. Now you can indicate the actual position and speed. 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. Finally, you can use **Clear** to completely clear the path.

At the right of the form you will see the actual path. 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.)

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 what should happen when the last point is reached. There are a number of options. The most common is to keep on moving to the first point, closing the path. Alternatively you can stop moving, jump to the first point, or traverse the same path backwards. The final option restarts the path from its current position. In this way the path will normally "walk away". Only the first five iterations are shown but the path will continue after that.

21.2 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 two further values you can provide. First of all there is the speed with which the path must be executed. (This is the same as the normal speed setting.) Remember that when defining the path you specify the actual speed relative to this indicated speed. Secondly you can indicate where the path should be started. A value of 0 indicates the start (which is most common). A value of 1 indicates the end of the path, that is, the moment the path is executed a second time. E.g. when the path is reversing, a value of 0.5 is the moment where it reverses.

When using scripts or pieces of code you have more control over the way the path is executed. There are four variables that influence this. The variable path_index indicates the index of the path. The variable path_position indicates the current position on the path (between 0 and 1 as indicated above). It changes while the instance follows the path. The speed is controlled by the standard speed variable. Note that the direction variable is in each step automatically set to the correct direction along the path. So you can use this variable to e.g. choose the correct subimage. 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).

You might wonder what happens when the instance collides with another instance while it follows a path. First of the collision event is executed. If the other instance is solid the

instance will stop, as it should (assuming there is a collision event defined). The path_position variable will though continue to follow the path. So at some moment the instance might start moving again in a different direction, if such a position is reached on the path.

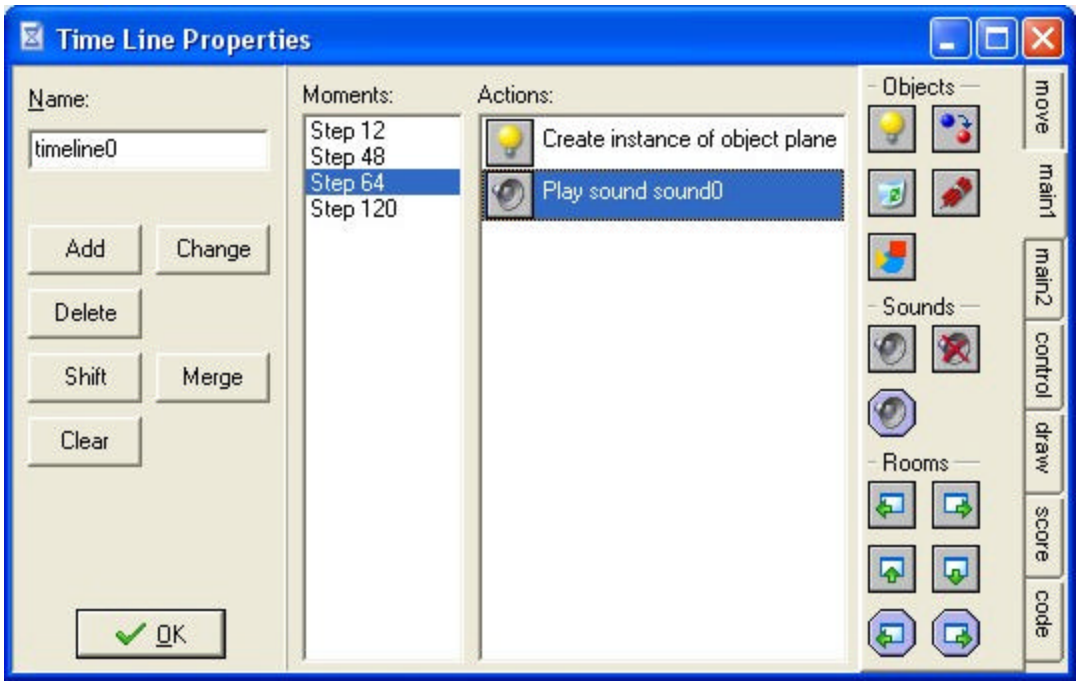
21.3 The path event

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.

Chapter 22 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 created 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 me 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 **Add Time Line** from the **Add** 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 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 moment 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**

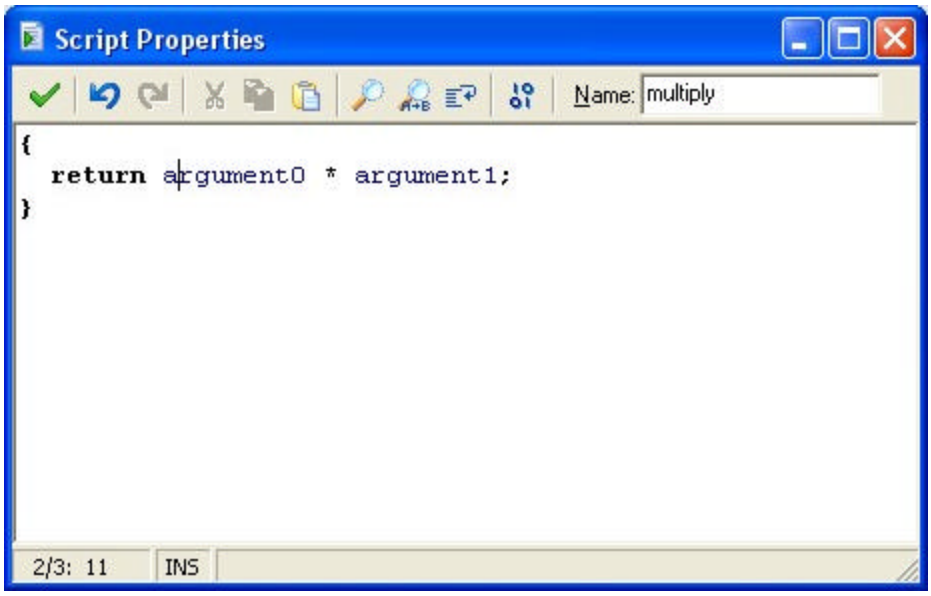
With this action you can change the position in the current time line (either absolute or relative). This can be used to skip certain parts of the time line or to repeat certain parts. For example, if you want to make a looping time line, at the last moment, add this action to set the position back to 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.

Chapter 23 Scripts

Game Maker has a built-in programming language. Once you get more familiar with *Game Maker* and want to use it to its fullest extend, it is advisable to start learning to use this language. For a complete description see Chapter 28. There are two ways to use the language. First of all you can create scripts. These are pieces of code that you give a name. They are shown in the resource tree and can be 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 a piece of code in the built-in programming language that performs a particular task. A script can take a number of arguments. To execute a script from within some event, you can use the script action. In this action you specify the script you want to execute, together with the up to five arguments. (You can also execute scripts from within a piece of code in the same way you call a function. In that case you can use up to 16 arguments.) When the script returns a value, you can also use it as a function when providing values in other actions.

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



(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. The editor has a number of useful properties many available through buttons (press the right mouse button for some additional commands):

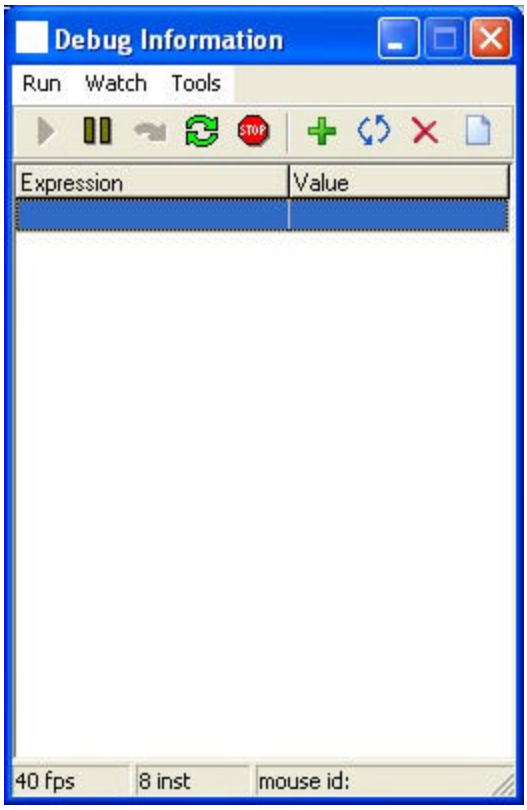
- Multiple undo and redo either per key press or in groups (can be changed in the preferences)
- Intelligent auto indent that aligns with the previous line (can be set in the preferences)
- Intelligent tabbing that tabs till the first non space in the previous lines (can be set in the preferences)
- 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 careful. Scripts can be stored in libraries that can be added to your game. To import a library, use the item **Import scripts** from the file menu. To save your scripts in the form of a library use **Export scripts**. Script libraries are simple text files (although they have the extension .gml). Preferably don't edit them directly because they have a special structure. Some libraries with useful scripts are included. (To avoid unnecessary work when loading the game, after importing a library, best delete those scripts that you don't use.)

When creating scripts you easily make mistakes. Always test the scripts by using the appropriate button. When an error occurs during the execution of a script this is reported, with an indication of the type of error and the place. If you need to check things more carefully, you can run the game in debug mode. Now a form appears in which you can monitor lots of information in your game.

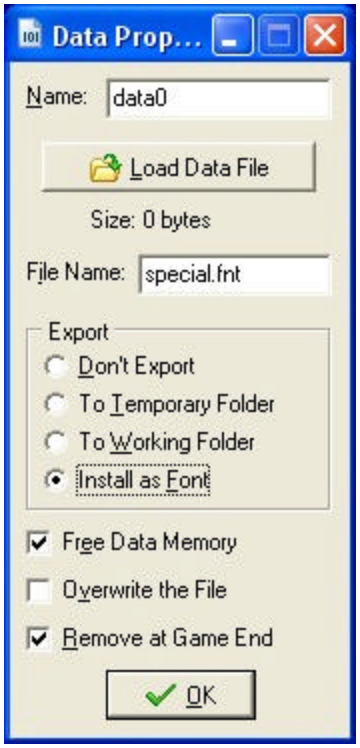


Under the **Run** menu you can pause the game, run it step by step and even restart it. Under the **Watch** menu you can watch the value of certain expressions. Use **Add** to type in some expression whose value is shown in each step of the game. In this way you can see whether your game is doing things 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.

Chapter 24 Data files

In more advanced games you often need to use additional files, for example files that describe certain properties, backgrounds and sprites that you want to load during the game, movies, DLL files (see later) or your own fonts. You can distribute these files with your game but it is nicer to embed them in the game itself. For this you can use data file resources. A data file resource simply stores the contents of a file. When the game starts this file is written to the disk and can then be used in the game.

To add a data file choose **Add Data File** from the **Add** menu. The following form will show:



You can give the resource a name as always. Press the button **Load Data File** to load the file into the resource. You can choose the **File Name** that must be used for storing the resource (no path is allowed in the filename). Next you can indicate what to do with the data file when the game starts:

- **Don't Export**. Don't export it at all. (There is a GML function to export it later.)
- **To Temporary Folder**. The file is written to the temporary folder for the game. This is in particular useful for e.g. loading backgrounds, etc.
- **To Working Folder**. The file is written to the folder in which the game is running. Many routines search for files there but you must be careful as the player might run the game from a read-only device.
- **Install a Font**. If your file is a font file you can choose this option. The file is saved in the temporary folder and next installed as a font such that it can be used in the game.

Finally there are a few options

- **Free Data Memory**. When checked the memory used for the data file is freed after exporting. This saves memory but means that the resource can no longer be used in functions.
- **Overwrite the File**. When checked the file is overwritten if it already exists.
- **Remove at Game End**. When check the file is removed again when the game ends. (Note that files in the temporary folder are always removed when the game ends.)

If an error occurs during exporting the game will still run. But the file or font might not be available.

Data files can take a lot of memory but loading and exporting them is fast.

Chapter 25 Game information

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

One interesting option in the **Format** menu is to **Mimic Main Form**. When you check this option the help form is displayed exactly at the position and the size of the game form. As a result it looks like the text appears in the game window. Choosing the correct background color now provides a nice visual effect. (You might want to indicate at the bottom of the help file that the user must press Escape to continue playing.)

A good advice is to make the information short but precise. Of course you should add your name because you created the game. All example games provided have an information file about the game and how it was created.

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

Chapter 26 Game options

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

26.1 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.

Scale percentage in windowed mode
Here you can indicate that the image in windowed mode should be scaled. 100 is no scaling. You typically use when your sprites and rooms are very small. Scaling is slower but most modern graphics cards can do this with little overhead. Better don't use values below 100 because scaling down is normally very slow.

Scale percentage in fullscreen mode
Here you can indicate that the image in fullscreen mode should be scaled. 100 is no scaling. A value of 0 indicates maximal possible scaling. Scaling is slower but most modern graphics cards can do this with little overhead. Better don't use values below 100 because scaling down is normally very slow.

Only scale when there is hardware support
If you check this scaling is only done when there is hardware support for this. Unfortunately though some graphics cards indicate that there is hardware support even if there is not.

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 it.

Wait for vertical blank before drawing
The screen of your computer is refreshed a number of times per second (normally between 50 and 100). After refreshing the screen there is a so-called vertical blank in which nothing happens on the screen. If you draw the screen continuously, part of one image and part of the next might show on the screen, which can give a poor visual effect.

If you wait for the vertical blank before drawing the next frame, this problem disappears. The disadvantage is that the program must wait for the vertical blank which will slow it down slightly.

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

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

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.

26.2 Resolution

In this tab you can set the screen resolution in which your game must run.

Set the resolution of the screen
The screen has a particular resolution. Three aspects play a role here: the number of pixels (horizontal and vertical) on the screen, the number of bits used for representing the colors, and the frequency with which the screen is refreshed. Normally *Game Maker* does not change these settings, that is, it uses the settings for the machine on which the game is running. This can lead to poor graphics. For example, if your rooms are small and a user uses a large screen resolution, the game will play in a very small window. You can solve this by using full screen mode and scaling the image, but that might slow down the game. The best way to solve this is to tell *Game Maker* to change the screen resolution when running the game. It will change it back afterwards. To do so, check this option. A number of additional options will occur. First of all you can indicate the color depth setting (16 or 32 bits; normally 16 bits is best; on Windows'98 games will always run in 16 bit color depth, even if you specify 32 bits). Secondly you can indicate the screen size (320x240, 640x480, 800x600, 1024x768, 1280x1024, or 1600x1200). Some warnings are in place here though. If you choose a small resolution (e.g. 320x240) Windows will resize all your forms when you run the game. This might cause problems with other applications you are running (and also in *Game Maker* itself). (Use exclusive mode to avoid this.) When you use the largest two you should also be careful because not all computers support it. You can also indicate not to change the screen size. Finally you can indicate the frequency (60, 70, 85, 100; if the one you specify is too high, the default frequency is used; you can also specify to use the default frequency). Also the following option occurs:

Use exclusive graphics mode
In exclusive mode, the game has the full control over the screen. No other applications can use it anymore. It makes the graphics often a bit faster and allows for some special

effects (like gamma settings). If you want to make sure the computer of the player is and stays in the right screen resolution, you best use exclusive mode. A warning is in place though. In exclusive mode no other windows can show. This e.g. means that you cannot use actions that display a message, ask a question, show the highscore list, or show the game information. Also no errors can be reported. In general, when something goes wrong in exclusive mode the game ends, and sometimes this does not help and the player has no other option than to restart the computer. So make sure your game works absolutely correct. You cannot run your game in debug mode when using exclusive mode.

26.3 Key options

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

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

Let <F4> switch between screen modes
When checked the F4 key will switch between fullscreen and windowed mode (not in exclusive mode).

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

26.4 Loading options

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

Secondly, you can indicate here the icon that should be used for stand-alone games. You can only use 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.

26.5 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. (Except in exclusive mode.) 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.

26.6 Info options

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 multiple people on a game or make new, updated version. The information is not accessible when the game is running.

Chapter 27 Speed considerations

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

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

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

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

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

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

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

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

Chapter 28 The Game Maker Language (GML)

As you have read before, *Game Maker* contains a built-in programming language. This programming language gives you much more flexibility and control than the standard actions. This language we will refer to as GML (the Game Maker Language). There are 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 resulting in a value.

In this chapter I will describe the basic structure of programs in GML. When you want to use programs in GML, there are a couple of things you have to be careful about. First of all, for all your resources (sprites, objects, sounds, etc.) you must use names that start with a letter and only consist of letters, digits and the underscore '_' symbol. Otherwise you cannot refer to them from within the program. 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.

28.1 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.

28.2 Variables

Like any programming language GML contains variables. Variables are memory locations that store information. They have a name such 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.

28.3 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 ^=.

28.4 Expressions

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

- `&& || ^^`: combine Boolean values (&& = 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

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 useless assignments.

```
{
  x = 23;
  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);
}
```

28.5 Extra variables

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

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

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

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);
}
```

28.6 Addressing variables in other instances

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

```
x = 3;
```

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

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

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

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

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

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

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

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

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

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

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

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

```
{
```

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

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

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

28.7 Arrays

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

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

28.8 If statement

An if statement has the form

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

or

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

The statement can also be a block. The expression will be evaluated. If the (rounded) value is <=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};
}
```

28.9 Repeat statement

A repeat statement has the form

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

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

Example
The following program creates five balls at random positions.

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

28.10 While statement

A while statement has the form

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

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

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

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

28.11 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))
}
```

28.12 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;
}
```

28.13 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. (No default statement is required. 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;
}
```

28.14 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).

28.15 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.

28.16 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.)

28.17 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`.

28.18 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, such that they can be used in expressions. For this end you use the return statement:

```
return <expression>
```

Execution of the script ends at the return statement!

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

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

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

28.19 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();
}
```

28.20 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.)

28.21 Functions and variables in GML

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

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

Chapter 29 Computing things

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

29.1 Constants

The following constants exist:

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

29.2 Real-values functions

The following functions exist that deal with real numbers.

`random(x)` Returns a random real number between 0 and x. The number is always smaller than x.
`abs(x)` Returns the absolute value of x.
`sign(x)` Returns the sign of x (-1 or 1).
`round(x)` Returns x rounded to the nearest integer.
`floor(x)` Returns the floor of x, that is, x rounded down to an integer.
`ceil(x)` Returns the ceiling of x, that is, x rounded up to an integer.
`frac(x)` Returns the fractional part of x, that is, the part behind the decimal dot.
`sqrt(x)` Returns the square root of x. x must be non-negative.
`sqr(x)` Returns x*x.
`power(x,n)` Returns x to the power n.
`exp(x)` Returns e to the power x.
`ln(x)` Returns the natural logarithm of x.
`log2(x)` Returns the log base 2 of x.
`log10(x)` Returns the log base 10 of x.
`logn(n,x)` Returns the log base n of x.
`sin(x)` Returns the sine of x (x in radians).
`cos(x)` Returns the cosine of x (x in radians).
`tan(x)` Returns the tangent of x (x in radians).
`arcsin(x)` Returns the inverse sine of x.
`arccos(x)` Returns the inverse cosine of x.
`arctan(x)` Returns the inverse tangent of x.
`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(x,y)` Returns the minimum of x and y.
`max(x,y)` Returns the maximum of x and y.
`min3(x,y,z)` Returns the minimum of x, y and z.

max3(x,y,z) Returns the maximum of x, y and z.
mean(x,y) Returns the average of x and y.
point_distance(x1,y1,x2,y2) Returns the distance between point (x1,y1) and point (x2,y2).
point_direction(x1,y1,x2,y2) Returns the direction from point (x1,y1) toward point (x2,y2) in degrees.
is_real(x) Returns whether x is a real value (as opposed to a string).
is_string(x) Returns whether x is a string (as opposed to a real value).

29.3 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 indicated the number of decimal places.
string_length(str) Returns the number of characters in the string.
string_pos(substr,str) Returns the position of substr in str (0=no occurrence).
string_copy(str,index,count) Returns a substring of str, starting at position index, and of length count.
string_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.

Chapter 30 GML: Game play

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

30.1 Moving around

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

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

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

To 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).
path_index Index of the current path the instance follows. Set to –1 to have no path.
path_position Position in the current path. 0 is the beginning of the path. 1 is the end of the path.
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.

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

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

move_random(hsnap,vsnap) Move the instance to a free random, snapped position, like the corresponding action.
move_snap(hsnap,vsnap) Snap the instance, like the corresponding action.
move_towards_point(x,y,sp) Moves the instances with speed sp toward position (x,y).
move_bounce_solid(adv) Bounce against solid instances, like the corresponding action. adv indicates whether to use advance bounce, that also takes slanted walls into account.
move_bounce_all(adv) Bounce against all instances, instead of just the solid ones.
move_contact_solid(dir,maxdist) Move 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) Move 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`.

30.2 Instances

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

object_index* Index of the object this is an instance of. 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 (using the dot construction described in section 28.6). Fortunately there are a number of variables and routines that help you locate instance id's.

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

Let me give an example. Assume each unit in your game has a particular 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.
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) Destroy all instances whose sprite contains position (x,y).

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

30.3 Timing

Good games required careful timing of things happening. Fortunately *Game Maker* does most of the timing for you. It makes sure things happen at a constant rate. This rate is defined when defining the rooms. But you can change it using the global variable `room_speed`. So for example, you can slowly increase the speed of the game, making it more difficult, by adding a very small amount (like 0.001) to `room_speed` in every step. If your machine is slow the game speed might not be achieved. This can be checked using the variable `fps` that constantly monitors the actual number of frames per second. Finally, for some advance timing you can use the variable `current_time` that gives the number of milliseconds since the computer was started. Here is the total collection of variables available (only the first one can be changed):

room_speed Speed of the game in the current room (in steps per second).
fps* Number of frames that are actually drawn per second.
current_time* Number of milliseconds that have passed since the system was started.
current_year* The current year.
current_month* The current month.
current_day* The current day.
current_weekday* The current day of the week (1=sunday, …, 7=saturday).
current_hour* The current hour.
current_minute* The current minute.
current_second* The current second.

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

sleep(numb) Sleeps numb milliseconds.

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

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

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 with 1. You can change this amount by setting this variable to a different value. You can use reals, e.g. 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.

30.4 Rooms and score

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

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

The following variables and functions exist that deal with rooms.

room Index of the current room; can be changed to go to a different room, but you better use the routines below.
room_first* Index of the first room in the game.
room_last* Index of the last room in the game.
room_goto(numb) Goto the room with index numb.
room_goto_previous() Go to the previous room.
room_goto_next() Go to the next room.
room_restart() Restart the current room.
room_previous(numb) Return the index of the room before numb (-1 = none) but don't go there.
room_next(numb) Return the index of the room after numb (-1 = none).
game_end() End the game.
game_restart() Restart the game.

Rooms have a number of additional properties:

room_width* Width of the room in pixels.
room_height* Height of the room in pixels.
room_caption Caption string for the room that is displayed in the caption of the window.
room_persistent Whether the current room is persistent.

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

game_save(string) Save the game to the file with name string.
game_load(string) Load the game from the file with name string.

Another important aspect of many games is the score, 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 you better 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.

There is also a built-in mechanism to keep track of a highscore list. It can contain up to ten names. For more information, see Chapter 34.

30.5 Generating events

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

draw event is called multiple times in each step). You can also apply an event to the current instance from within a piece of code. The following functions exist:

event_perform(type,numb) Performs event numb of 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 7. 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_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_user0
ev_user1
ev_user2
ev_user3
ev_user4
ev_user5
ev_user6
ev_user7

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

ev_step_normal
ev_step_begin
ev_step_end

event_perform_object(obj,type,numb) This functions works the same as the function above except that this time you can specify events in another object. Note that the actions in these events are applied to the current instance, not to instances of the given object.

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

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

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

event_type* Type of the current event begin executed.

event_number* Number of the current event begin executed.

event_object* The object index for which the current event is being executed.

event_action* The index of the action that is currently being executed (0 is the first in the event, etc.).

30.6 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

Chapter 31 GML: User interaction

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

mouse_x* X-coordinate of the mouse. Cannot be changed.
mouse_y* Y-coordinate of the mouse. Cannot be changed.
mouse_button Currently pressed mouse button. As value use mb_none, mb_any, mb_left, mb_middle, or mb_right.
keyboard_lastkey Keycode of last key pressed. See below for keycode constants. You can change it, e.g. set it to 0 if you handled it.
keyboard_key Keycode of current key pressed (see below; 0 if none).
keyboard_lastchar Last character pressed (as string).
keyboard_string String containing the last at most 80 characters typed. This string will only contain the printable characters typed. It also correctly responds to pressing the backspace key by erasing the last character.

Sometime 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 pressed.
keyboard_check_direct(key) Returns whether the key with the particular keycode is pressed by checking the hardware directly. The result is independent of which application has focus. It allows for a few more checks. In particular you can use keycodes vk_lshift, vk_lcontrol, vk_lalt, vk_rshift, vk_rcontrol and vk_ralt to check whether the left or right shift, control or alt key is pressed. (This does not work under windows 95!).
mouse_check_button(numb) Returns whether the mouse button is pressed (use as values mb_none, mb_left, mb_middle, or mb_right).

The following 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. (Does not work under Windows 95.)
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.) 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

They do not work on older version of Windows!

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 three additional functions related to interaction.

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

31.1 Joystick support

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.

Chapter 32 GML: Game graphics

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

32.1 Window and cursor

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

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

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

> **`scale_window`** This variable indicates the percentage of scaling in windowed mode. 100 indicates no scaling.
> **`scale_full`** This variable indicates the percentage of scaling in fullscreen mode. 100 indicates no scaling. 0 indicates the maximum scaling possible.

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

> **`show_cursor`** If set to false the cursor is made invisible inside the playing area, otherwise it is made visible.

You can also set the cursor to one of the many predefined cursors in windows using the following function:

> **`set_cursor(cur)`** Set the cursor to the given value. You can use the following constant: cr_default, cr_none, cr_arrrow, 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.

By the way, note that it is very easy to make your own cursor object. Just create an object with a negative depth that, in its step event, follows the mouse position.

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

monitor_width The width of the monitor, in pixels.
monitor_height The height of the monitor, in pixels.

32.2 Sprites and images

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

visible If visible is true (1) the image is drawn, otherwise it is not drawn. Invisible instances are still active and create collision events; you only don't see them. Setting the visibility to false is useful for e.g. controller objects (make them non-solid to avoid collision events) or hidden switches.
sprite_index This is the index of the current sprite for the instance. You can change it to give the instance a different sprite. As value you can use the names of the different sprites you defined. Changing the sprite does not change the index of the currently visible sub-image.
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 sub-images for the current sprite for the instance (cannot be changed).
image_index When the image has multiple sub-images the program cycles through them. This variable indicates the currently drawn sub-image (they are numbered starting from 0). You can change the current image by changing this variable. The program will continue cycling, starting at this new index.
image_single Sometimes you want a particular sub-image to be visible and don't want the program to cycle through all of them. This can be achieved by setting this variable to the index of the sub-image you want to see (first sub-image has index 0). Give it a value –1 to cycle through the sub-images. This is useful when an object has multiple appearances. For example, assume you have an object that can rotate and you create a sprite that has sub-images for a number of orientations (counter-clockwise). Then, in the step event of the object you can set

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

image_speed The speed with which we cycle through the sub-images. A value of 1 indicates that each step we get the next image. Smaller values will switch sub-images slower, drawing each sub-image multiple times. Larger values will skip sub-images to make the motion faster.
depth Normally images are drawn in the order in which the instances are created. You can change this by setting the image depth. The default value is 0, unless you set it to a different value in the object properties. The higher the value the further the instance is away. (You can also use negative values.) Instances with higher depth will lie behind instances with a lower depth. Setting the depth will guarantee that the instances are drawn in the order you want (e.g. the plane in front of the cloud). Background instances should have a high (positive) depth, and foreground instances should have a low (negative) depth.
image_scale A scale factor to make larger or smaller images. A value of 1 indicates the normal size. Changing the scale also changes the values for the image width and height and influences collision events as you might expect. Realize that scaled images (in particular when you make them smaller) take more time to draw. Changing the scale can be used to get a 3-D effect.
image_alpha Transparency (alpha) value to use when drawing the image. A value of 1 is the normal setting; a value of 0 is completely transparent. Use with care. Drawing partially transparent images takes a lot of time and will slow down the game.
bbox_left* Left side of the bounding box used of the image of the instance (taking scaling into account).
bbox_right* Right side of the bounding box of the instance image
bbox_top* Top side of the bounding box of the instance image.
bbox_bottom* Bottom side of the bounding box of the instance image.

32.3 Backgrounds

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

background_color Background color for the room.
background_showcolor Whether to clear the window in the background color.
background_visible[0..7] Whether the particular background image is visible.
background_foreground[0..7] Whether the background is actually a foreground.
background_index[0..7] Background image index for the background.
background_x[0..7] X position of the background image.
background_y[0…7] Y position of the background image.
background_width[0…7]* Width of the background image.
background_height[0…7]* Height of the background image.

background_htiled[0..7] Whether horizontally tiled.
background_vtiled[0..7] Whether vertically tiled.
background_hspeed[0..7] Horizontal scrolling speed of the background (pixels per step).
background_vspeed[0..7] Vertical scrolling speed of the background (pixels per step).
background_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. Use with care. Drawing partially transparent backgrounds takes a lot of time and will slow down the game.

32.4 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 e.g. 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. When designing a room you can only indicate whether to use background tiles (with depth 1000000) or foreground tiles (with depth –1000000) but you can actually 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).
- **alpha.** An alpha value indicating tile transparency. 1 = not transparent, 0 = fully transparent. You should use this with care because partially transparent tiles are very slow to draw and can lead to problems on certain systems.

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_find(x,y,foreground) Returns the id of the tile at position (x,y). When no tile exists at the position –1 is returned. When foreground is true, only tiles with depth < 0 are returned. Otherwise only tiles with depth >= 0 are returned. When multiple foreground or background tiles exist at the position the first one is returned.
tile_delete_at(x,y,foreground) Deletes the tiles at position (x,y). When foreground is true, only tiles with depth < 0 are deleted. Otherwise only tiles with depth >= 0 are deleted. When multiple (foreground or background) tiles exist at the position all of them are deleted.
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_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_alpha(id,alpha) Sets the alpha value of the tile with the given id.

32.5 Drawing functions

It is possible to let objects look rather different from their image. There is a whole collection of functions available to draw different shapes. Also there are functions to draw text. You can only use these in the drawing event of an object; these functions don't make any sense anywhere else in code (although, see Section 32.8). Please realize that the graphics hardware in computers only makes the drawing of images fast. So any other drawing routine will be relatively slow. Also *Game Maker* is optimized towards drawing

images. So avoid other drawing routines as much as possible. (Whenever possible, create a bitmap instead.) Also realize that collisions between instances are determined by their sprites (or masks) and not by what you actually draw. The following image related drawing functions exist:

draw_sprite(n,img,x,y) Draws subimage img (-1 = current) of the sprite with index n with its origin at position (x,y).
draw_sprite_scaled(n,img,x,y,s) Draws the sprite scaled with a factor s.
draw_sprite_stretched(n,img,x,y,w,h) Draws the sprite stretched such that it fills the region with top-left corner (x,y) and width w and height h.
draw_sprite_transparent(n,img,x,y,s,alpha) Draws the sprite scaled with factor s merged with its background. alpha indicates the transparency factor. A value of 0 makes the sprite completely transparent. A value of 1 makes it completely solid. This function can create great effect (for example partially transparent explosions). It is though very slow because it is done in software and, hence, should be used with care.
draw_sprite_tiled(n,img,x,y) Draws the sprite tiled such that it fills the entire room. (x,y) is the place where one of the sprites is drawn.

draw_background(n,x,y) Draws the background with index n at position (x,y).
draw_background_scaled(n,x,y,s) Draws the background scaled.
draw_background_stretched(n,x,y,w,h) Draws the background stretched to the indicated region.
draw_background_transparent(n,x,y,s,alpha) Draws the background scales with factor s and transparency alpha (0-1) (slow!).
draw_background_tiled(n,x,y) Draws the background tiled such that it fills the entire room.

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

draw_pixel(x,y) Draws a pixel at (x,y) in the brush color.
draw_getpixel(x,y) Returns the color of the pixel at (x,y).
draw_fill(x,y) Flood fill from position (x,y) in the brush color.
draw_line(x1,y1,x2,y2) Draws a line from (x1,y1) to (x2,y2).
draw_circle(x,y,r) Draws a circle at (x,y) with radius r.
draw_ellipse(x1,y1,x2,y2) Draws an ellipse.
draw_rectangle(x1,y1,x2,y2) Draws a rectangle.
draw_roundrect(x1,y1,x2,y2) Draws a rounded rectangle.
draw_triangle(x1,y1,x2,y2,x3,y3) Draws a triangle.
draw_arc(x1,y1,x2,y2,x3,y3,x4,y4) Draws an arc of an ellipse.
draw_chord(x1,y1,x2,y2,x3,y3,x4,y4) Draws a chord of an ellipse.
draw_pie(x1,y1,x2,y2,x3,y3,x4,y4) Draws a pie of an ellipse.

draw_button(x1,y1,x2,y2,up) Draws a button, up indicates whether up (1) or down (0).

draw_text(x,y,string) Draws the string at position (x,y). 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.

draw_text_sprite(x,y,string,sep,w,sprite,firstchar,scale) Drawing text using the functions above is relatively costly. This function works exactly the same as the previous one but takes its character images from a sprite. This sprite must have a subimage for each character. The first character is indicate with the argument firstchar. From this character on the characters should follow the ASCI order. You can check the character map tool of windows to see the correct order of the characters. If you only need the first few (e.g. up to the numbers or the uppercase characters) you don't need to provide the other characters. scale indicates the scale factor used (1 is the normal size). Be careful will scaling though, it can slow down the game considerably. Please realize that these sprites tend to be large. Also, you obviously don't need precise collision checking for them.

draw_polygon_begin() Start describing a polygon for drawing.

draw_polygon_vertex(x,y) Add vertex (x,y) to the polygon.

draw_polygon_end() End the description of the polygon. This function actually draws it.

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

brush_color Color used to fill shapes. A whole range of predefined colors is available:

c_aqua
c_black
c_blue
c_dkgray
c_fuchsia
c_gray
c_green
c_lime
c_ltgray
c_maroon
c_navy
c_olive

c_purple
c_red
c_silver
c_teal
c_white
c_yellow

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

brush_style Current brush style used for filling. The following styles are available:

bs_hollow
bs_solid
bs_bdiagonal
bs_fdiagonal
bs_cross
bs_diagcross
bs_horizontal
bs_vertical

pen_color Color of the pen to draw boundaries.

pen_size Size of the pen in pixels.

font_color Color of the font to use.

font_size Size of the font to use (in points).

font_name Name of the font (a string).

font_style Style for the font. The following styles are available (you can add them if you want the combination of the styles):

fs_normal
fs_bold
fs_italic
fs_underline
fs_strikeout

font_angle Angle with which the font is rotated (0-360 degrees). For example, for vertical text use value 90.

font_align Alignment of the text w.r.t. the position given. The following values can be used

fa_left
fa_center
fa_right

A few miscellaneous functions exist:

string_width(string) Width of the string in the current font as it would 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 drawn using the draw_text() function.

string_width_ext(string,sep,w) Width of the string in the current font as it would 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 drawn using the draw_text_ext() function.
screen_gamma(r,g,b) Sets the gamma correction values. r,g,b must be in the range from –1 to 1. The default is 0. When you use a value smaller than 0 that particular color becomes darker. If you use a value larger than 0 that color becomes lighter. Most of the time you will keep the three values the same. For example, to get the effect of lightning you can temporarily make the three values close to 1. This function works only in exclusive mode!
screen_save(fname) Saves a bmp image of the screen in the given filename. Useful for making screenshots.
screen_save_part(fname,left,top,right,bottom) Saves part of the screen in the given filename.

32.6 Views

As you should know you can define up to eight different views when designing rooms. In this way you can show different parts of the room at different places on the screen. Also, you can make sure that a particular object always stays visible. You can control the views from within code. You can make views visible and invisible and change the place or size of the views on the screen or the position of the view in the room (which is in particular useful when you indicated no object to be visible), you can change the size of the horizontal and vertical border around the visible object, and you can indicate which object must remain visible in the views. The latter is very important when the important object changes during the game. For example, you might change the main character object based on its current status. Unfortunately, this does mean that it is no longer the object that must remain visible. This can be remedied by one line of code in the creation event of all the possible main objects (assuming this must happen in the first view):

```
{
  view_object[0] = object_index;
}
```

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

view_enabled Whether views are enabled or not.
view_current* The currently drawn view (0-7). Use this only in the drawing event. You can for example check this variable to draw certain things in only one view. Variable cannot be changed.
view_visible[0..7] Whether the particular view is visible on the screen.
view_left[0..7] Left position of the view in the room.
view_top[0..7] Top position of the view in the room.
view_width[0..7] Width of the view (in pixels).
view_height[0..7] Height of the view (in pixels).
view_x[0..7] X-position of the view on the screen.
view_y[0..7] Y-position of the view on the screen.

view_hborder[0..7] Size of horizontal border around the visible object (in pixels).
view_vborder[0..7] Size of vertical border around visible object (in pixels).
view_hspeed[0..7] Maximal horizontal speed of the view.
view_vspeed[0..7] Maximal vertical speed of the view.
view_object[0..7] Object whose instance must remain visible in the view. 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 variables:

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

32.7 Transitions

As you know, when you move from one room to another you can indicate a transition. You can also set the transition for the next frame without moving to another room using the variable called `transition_kind`. If you assign a value between 1 and 17 to it the corresponding transition is used (these are the same transitions you can indicate for the rooms). A value of 0 indicates no transition. It only affects the next time a frame is drawn. You can also set these variables before going to the next room using code.

transition_kind Indicates the next frame transition (0-17).
transition_time Total time used for the transition (in milliseconds).
transition_steps Number of steps for the transition.

32.8 Repainting the screen

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

screen_redraw() Redraws the room by calling all draw events.
screen_refresh() Refreshes the screen using the current room image (not performing drawing events).

To understand the second function, you will need to understand a bit better how drawing works internally. There is internally an image on which all drawing happens. This image is not visible on the screen. Only at the end of a step, after all drawing has taken place,

the screen image is replaced by this internal image. (This is called double buffering.) The first function redraws the internal image and then refreshes the screen image. The second function only refreshes the image on the screen.

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

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

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

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

Chapter 33 GML: Sound and music

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

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

There are 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 one.

`sound_play(index)` Plays the indicates sound once.
`sound_loop(index)` Plays the indicates sound, looping continuously.
`sound_stop(index)` Stops the indicates sound. If there are multiple sounds with this index playing simultaneously, all will be stopped.
`sound_stop_all()` Stops all sounds.
`sound_isplaying(index)` Returns whether the indicates sound is playing.

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

`sound_volume(index,value)` Changes the volume for the indicates sound (0 = low, 1 = high).

sound_pan(index,value) Changes the pan for the indicates sound (0 = left, 1 = right).
sound_frequency(index,value) Changes the frequency for the indicates sound (0 = low, 1 = high).

Sound is a complicated matter. Midi files are played using the standard multimedia player. Only one midi file can be played at once and there is no support for sound effects. For wave files *Game Maker* uses DirectSound. In this case all wave files are stored in memory and can have effects. *Game Maker* actually also tries to play other music files when you specify them, in particular mp3 files. It uses the standard multimedia player for this. Be careful though. Whether this works depends on the system and sometimes on other software installed or running. So you are recommended not to use mp3 files when you want to distribute your games.

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!

Chapter 34 GML: Splash screens, highscores, and other pop-ups

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

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

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

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

show_info() Displays the game information form.

load_info(fname) Load the game information from the file named fname. This should be an rtf file. This makes it possible to show different help files at different moments. You can use the data file resource to put these file inside the game.

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.
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.
message_button_font(name,size,color,style) Sets the font for the buttons in the pop-up box.
message_input_font(name,size,color,style) Sets the font for the input field in the pop-up box.
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 be 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.
message_size(w,h) Fixes the size of the pop-up box on the screen. If you choose 0 for the width the width of the image is used. If you choose 0 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.

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_show_ext(numb,back,border,col1,col2,name,size) 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. 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.

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

Chapter 35 GML: Resources

In *Game Maker* you can define various types of resources, like sprites, sounds, data files, objects, etc. In this chapter you will find a number of functions that act on the resources. Before you start using these, maker sure you understand the following. Whenever you change a resource the original data is gone! This means that the resource is changed for all instances that use it. For example, if you change a sprite all instances that use this sprite will have it changed during the remainder of the game. Restarting a room or restarting the game will not bring the resource back to its original form! Also when saving the game situation the changed resources are NOT saved. If you load a saved game it is your responsibility to have the resource back in the appropriate state.

35.1 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_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_precise(ind)` Returns whether the sprite with the given index uses precise collision checking.
`sprite_get_videomem(ind)` Returns whether the sprite with the given index uses video memory.
`sprite_get_loadonuse(ind)` Returns whether the sprite with the given index is loaded only on use.

Sprites take lots of memory. To draw them fast enough it is important to store them in video memory. As indicated in Chapter 14 you can indicate which sprites should be

stored in video memory. Also you can indicate that certain sprites should only be loaded when needed. These sprites will be discarded again at the end of the level. You can partially control this process from code. The following functions exist:

sprite_discard(numb) Frees the (video) memory used for the sprite. If the sprite has the load-on-use property set it will be completely removed. Otherwise, a copy is maintained in normal memory (of which there is normally enough) such that the sprite can be restored when needed.
sprite_restore(numb) Restores the sprite in (video) memory. Normally this happens automatically when the sprite is needed. But this might cause a small hick-up, in particular when load-on-use is set and the sprite is large. So you might want to force this for example at the beginning of the room in which the sprite is needed.
discard_all() Discard all sprites, backgrounds and sounds that have load-on-use set.

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

sprite_add(fname,imgnumb,precise,transparent,videomem,loadonuse,xorig,yorig) Add the image stored in the file fname to the set of sprite resources. Only bmp, jpg and gif images can be dealt with. When the image is a bmp or jpg image it can be a strip containing a number of subimages for the sprite next to each other. Use imgnumb to indicate their number (1 for a single image). For (animated) gif images, this argument is not used; the number of images in the gif file is used. precise indicates whether precise collision checking should be used. transparent indicates whether the image is partially transparent, videomem indicates whether the sprite must be stored in videomemory, and loadonuse indicates whether the sprite should only be loaded when used. xorig and yorig indicate the position of the origin in the sprite. The function returns the index of the new sprite that you can then use to draw it or to assign it to the variable sprite_index of an instance. When an error occurs –1 is returned.
sprite_replace(ind,fname,imgnumb,precise,transparent,videomem,loadonuse,xorig,yorig) Same as above but in this case the sprite with index ind is replaced. The function returns whether it is successful.
sprite_delete(ind) Deletes the sprite from memory, freeing the memory used. (It can no longer be restored.)

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

35.2 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=wave, 1=midi, 2=mp3, 10=unknown).
sound_get_buffers(ind) Returns the number of buffers of the sound with the given index.
sound_get_effect(ind) Returns whether the sound with the given index allows for special effects.
sound_get_loadonuse(ind) Returns whether the sound with the given index is loaded only on use.

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

sound_discard(index) Frees the memory used for the indicated sound.
sound_restore(index) Restores the indicated sound in memory.
discard_all() Discard all sprites, backgrounds and sounds that have load-on-use set.

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

sound_add(fname,buffers,effects,loadonuse) Adds a sound resource to the game. fname is the name of the sound file. buffers indicates the number of buffers to be used, and effects and loadonuse indicate whether sound effects are allowed and whether the sound should be stored in internal memory (true or false). The function returns the index of the new sound, which can be used to play the sound. (-1 if an error occurred, e.g. the file does not exist).
sound_replace(index,fname,buffers,effects,loadonuse) Same as the previous function but this time not a new sound is created but the existing sound index is replaced, freeing the old sound. Returns whether correct.
sound_delete(index) Deletes the indicated sound, freeing all memory associated with it. It can no longer be restored.

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

35.3 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_videomem(ind)` Returns whether the background with the given index uses video memory.
`background_get_loadonuse(ind)` Returns whether the background with the given index is loaded only on use.

Background images take lots of memory. To draw them fast enough it can be useful to store them in video memory. As indicated in Chapter 18 you can indicate which backgrounds should be stored in video memory. Also you can indicate that certain backgrounds should only be loaded when needed. These backgrounds will be discarded again at the end of the level. You can partially control this process from code. The following functions exist:

`background_discard(numb)` Frees the (video) memory used for the background image. If the background has the load-on-use property set it will be completely removed. Otherwise, a copy is maintained in normal memory (of which there is normally enough) such that the background can be restored when needed.
`background_restore(numb)` Restores the background image in (video) memory. Normally this happens automatically when the background is needed. But this might cause a small hick-up, in particular when load-on-use is set and the background is large. So you might want to force this for example at the beginning of the room in which the background is needed.
`discard_all()` Discard all sprites, backgrounds and sounds that have load-on-use set.

When a game uses a lot of different background images, this makes the game file large and, hence, the loading slow. Also, if you want to keep them in memory while you need them, it increases the amount of memory required considerably. Alternatively, you can

distribute the background images with the game (as .bmp, .jpg, or .gif files; no other formats allowed) and load them during the game. There are three routines for this. Another use is when you want to let the player choose a background. Also, you might want to save the image from within the game and use that later as a background (e.g. for a painting program). Finally, complicated backgrounds, stored as jpg files use a lot less memory. Here are the functions:

background_add(fname,transparent,videomem,loadonuse) Add the image stored in the file fname to the set of background resources. Only bmp and jpg images can be dealt with. transparent indicates whether the image is partially transparent, videomem indicates whether the background must be stored in videomemory, and loadonuse indicates whether the background should only be loaded when used. The function returns the index of the new background that you can then use to draw it or to assign it to the variable background_index[0] to make it visible in the current room. When an error occurs –1 is returned.
background_replace(ind,fname,transparent,videomem,loadonuse) Same as above but in this case the background with index ind is replaced. The function returns whether it is successful. When the background is currently visible in the room it wil be replaced also.
background_delete(ind) Deletes the background from memory, freeing the memory used. (It can no longer be restored.)

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

35.4 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_end(ind) Returns what happens at the end of the path with the given index (0=stop, 1=jump to start, 2=connect to start, 3=reverse, 4=continue).

35.5 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.

35.6 Data Files

The following functions will give you information about a data file:

datafile_exists(ind) Returns whether a data file with the given index exists.
datafile_get_name(ind) Returns the name of the data file with the given index.
datafile_get_filename(ind) Returns the file name of the data file with the given index.

The following functions can be used if you did not automatically export the data file on game start.

datafile_export(ind,fname) Exports the data file to the given file name (if you did not already do that default on startup).
datafile_discard(ind) Frees the internally stored data for the data file.

35.7 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 (-1 if it has no parent).
object_is_ancestor(ind1,ind2) Returns whether object ind2 is an ancestor of object ind1.

35.8 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 change the contents of the current room.

Chapter 36 GML: Files, registry, and executing programs

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

`file_exists(fname)` Returns whether the file with the given name exists (true) or not (false).
`file_delete(fname)` Deletes the file with the given name.
`file_rename(oldname,newname)` Renames the file with name oldname into newname.
`file_copy(fname,newname)` Copies the file fname to the newname.
`file_open_read(fname)` Opens the indicated file for reading.
`file_open_write(fname)` Opens the indicated file for writing, creating it if it does not exist.
`file_open_append(fname)` Opens the indicated file for appending data at the end, creating it if it does not exist.
`file_close()` Closes the current file (don't forget to call this!).
`file_write_string(str)` Writes the string to the currently open file.
`file_write_real(x)` Write the real value to the currently open file.
`file_writeln()` Write a newline character to the file.
`file_read_string()` Reads a string from the file and returns this string. A string ends at the end of line.
`file_read_real()` Reads a real value from the file and returns this value.
`file_readln()` Skips the rest of the line in the file and starts at the start of the next line.
`file_eof()` Returns whether we reached the end of the file.
`directory_exists(dname)` Returns whether the indicated directory does exist.
`directory_create(dname)` Created a directory with the given name (including the path towards it) if it does not exist.
`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`** directoris
- **`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.

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

The following three read-only variables can be useful:

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

In certain situations you might want to give players the possibility to give 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 (note that the name of the program itself is one of them.
parameter_string(n) Returns command-line parameters n. The first parameter has index 0. This is the name of the program.

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

registry_write_string(name,str) Creates an entry in the registry with the given name and string value.
registry_write_real(name,x) Creates an entry in the registry with the given name and real value.
registry_read_string(name) Returns the string that the given name holds. (The name must exist. otherwise an empty string is returned.)
registry_read_real(name) Returns the real that the given name holds. (The name must exist. Otherwise the number 0 is returned.)
registry_exists(name) Returns whether the given name exists.

Actually, values in the registry are grouped into keys. The above routines all work on values within the key that is especially created for your game. Your program can use this to obtain certain information about the system the game is running on. You can also read values in other keys. You can write them also but be very careful. YOU 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 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

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

execute_program(prog,arg,wait) Execute program prog with arguments arg. wait indicates whether to wait for finishing.
execute_shell(prog,arg) Executes the program (or file) in the shell.

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

`secure_mode*` Whether the game is running in secure mode.

Chapter 37 GML: Multiplayer games

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

37.1 Setting up a connection

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

`mplay_init_ipx()` initializes an IPX connection.

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

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

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

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

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

To end the connection call

mplay_end() ends the current connection.

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.

37.2 Creating and joining sessions

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

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

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

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

mplay_session_find() searches for all sessions that still accept players and returns the number of sessions found.

mplay_session_name(numb) returns the name of session number `numb` (0 is the first session). This routine can only be called after calling the previous routine.
mplay_session_join(numb,playername) makes you join session number `numb` (0 is the first session). `playername` is the name of you as a player. Returns whether successful.

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

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

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

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

A player can stop a session using the following routine:

mplay_session_end() ends the session for this player.

37.3 Players

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

mplay_player_find() searches for all players in the current session and returns the number of players found.
mplay_player_name(numb) returns the name of player number `numb` (0 is the first player, which is always yourself). This routine can only be called after calling the previous routine.
mplay_player_id(numb) returns the unique id of player number `numb` (0 is the first player, which is always yourself). This routine can only be called after calling the first routine. This id is used in sending and receiving messages to and from individual players.

37.4 Shared data

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

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

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

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

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

37.5 Messages

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

The following messaging routines exist:

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

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

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

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

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

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

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

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

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

A few remarks are in place here. First of all, if you want to send a message to a particular player only, you will need to know the players unique id. As indicated earlier you can

obtain this with the function `mplay_player_id()`. This player identifier is also used when receiving messages from a particular player. Alternatively, you can give the name of the player as a string. If multiple players have the same name, only the first will get the message.

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

Chapter 38 GML: Using DLL's

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, Visual 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 12 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:real):real;  cdecl;
begin
  if x<y then Result := x else Result := y;
end;

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

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

exports MyMin, DoubleString;

begin
end.
```

This DLL defines two functions: `MyMin` that takes two real arguments and returns the minimum of the two, and `DoubleString` that doubles the string. Note that you have to be careful with memory management. That is why I declared the resulting string global. Also notice the use of the cdecl calling convention. 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.) You can also use a data file resource to store the DLL inside the game.

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 (0-12). 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('MYOWN.DLL','MyMin',dll_dcecl,
                                    ty_real,2,ty_real,ty_real);
  global.ddd = external_define('MYOWN.DLL','DoubleString',dll_dcecl,
                                    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');
}
```

You might wander how to make a function in a DLL that does something in the game. For example, you might want to create a DLL that adds instances of objects to your game. The easiest way is to let your DLL function return a string that contains a piece of GML code. This string that contains the piece of GML can be executed using the GML function

> **execute_string(str)** Execute the piece of code in the string str.

Alternatively you can let the DLL create a file with a script that can be execute (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 function. But please only use it if you know what you are doing.