

Beginner's Guide to Game Maker 4.3 Programming



This is a tutorial in how to start programming using Game Maker 4.0. It is meant for beginners with little or no knowledge about computer programming languages. I used Game Maker 4.0 when writing this, but today (2002-08-15) the latest version of Game Maker is 4.2a. It ought to work just as well with that version.

When version 4.3 of Game Maker was released, there were some changes to the program that required a new revision of this guide. This is that revision.

Document written by Carl Gustafsson (carl.gustafsson@home.se)
Game Maker by Mark Overmars

This document is also available in online HTML format at:
<http://www.gamecreators.nl>, thanks to Martijn.

Document date: 2002-03-05
Revision 1 date: 2002-08-15
Revision 2 date: 2002-12-17

1 Contents

1	CONTENTS	2	5	ENEMIES SIGHTED!	20
2.1	ACKNOWLEDGEMENTS	3	5.1	ENEMY AIRCRAFT	20
2.2	GAME MAKER INFORMATION	3	5.2	RANDOMIZING	22
2.3	PURPOSE OF THIS DOCUMENT	3	5.3	OUCH! THAT HURT!	23
2.4	OTHER REFERENCES	3	5.4	SHOOTOUT	25
2.5	ABOUT CUT-AND-PASTE	3	5.5	PYROTECHNICS	26
2.6	FINISHED FILE	4	6	ENHANCING THE GAME	29
2.7	WELL?	4	6.1	CENTERED SPRITES	29
3	CREATING A GAME	5	6.2	SMOOTHER ENEMY APPEARANCE	30
3.1	SOME SPRITES	5	6.3	GOING GLOBAL	31
3.2	CREATE OBJECTSS	6	6.4	WHERE'S MY ENERGY?	33
3.3	ROOM FOR IMPROVEMENT	6	7	LIFE, THE UNIVERSE AND	
3.4	SAVE, SAVE, SAVE!	7	EVERYTHING.....	36	
3.5	ACTION	7	7.1	A GALAXY FAR, FAR AWAY	36
3.6	REFINING THE ACTIONS	8	7.2	COOL WORD: PARALLAX	37
4	THE CODING BEGINS	10	7.3	ENEMY FIRE	38
4.1	FIRST VARIABLE	10	7.4	MEANING OF LIFE	40
4.2	FIRST FUNCTION	12	7.5	SCORING.....	43
4.3	MORE VARIABLES	13	8	FINAL WORDS	44
4.4	FIRST SCRIPT	13	8.1	END OF THIS GUIDE.....	44
4.5	GETTING RID OF THE <NO KEY> EVENT	14	8.2	COMMUNICATION.....	44
4.6	BULLET LOADING TIME	15	8.3	USELESS STATISTICS	44
			8.4	BYE	44

2 Introduction

2.1 Acknowledgements

I would like to use this space to say a heartily Thank You to Mark Overmars for creating such a wonderful game building tool. Thank You Mark!

Thank you Martijn for converting this guide to HTML format and publishing it at <http://www.gamecreators.nl> .

Thank You also to all members of the Game Maker community who are great at helping me and each other with game building advice, help and hints.

Finally, I am grateful for all comments and suggestions that I have received regarding this document from a number of Game Maker users.

2.2 Game Maker information

Game Maker is written by Mark Overmars. It is a complete game building tool that can be used to create 2-dimensional computer games that can run on Microsoft Windows systems. The program can be downloaded from the Game Maker web site, <http://www.cs.uu.nl/people/markov/gmaker/index.html>.

The application includes an integrated graphics creation tool that can be used to create sprites, a drag-and-drop interface, making it a built-in programming language that is similar to well-known programming languages like C/C++, Pascal and BASIC.

With Game Maker it is possible to create computer games without using a single line of code, thanks to its intuitive drag-and-drop icons representing game events and actions. However, to be able to create more advanced games and really release the full potential of Game Maker the use of Game Maker Language (hereafter referred to as GML) is an absolute requirement.

2.3 Purpose of this document

This guide was written as an attempt to introduce users that are used to create games using the drag-and-drop method, to the concept of GML.

2.4 Other references

I am going to make a lot of references to the Game Maker PDF manual (Game_Maker.pdf), so I suggest you have it ready. It can be downloaded from the Game Maker web site (see 2.2 Game Maker information)

2.5 About Cut-and-paste

You might feel like cutting-and-pasting code from this document into Game Maker, instead of writing it. That might work, but it seems like Word is doing something bad to the minus signs (-) so I really encourage you to write it down manually in Game Maker instead.

2.6 *Finished file*

The Game Maker file that will be the result if you follow this guide can of course be obtained by sending an email to me. But, again, I encourage you to really carry out the actions described in here yourself. At least I learn a lot more when doing than when only reading about something.

2.7 *Well?*

Well? What are you waiting for? Read on! ;)

3 Creating a game

We need something to work with in order to be able to understand the concepts of GML. This means that I, in the beginning, am going to refer to the graphical drag-and-drop icons and compare them to the GML code.

So, start up Game Maker and create a blank game (File -> New, but I am sure you know that).

3.1 Some sprites

In order to see anything in our game we are going to need some sprites. If you do not know what a sprite is, I suggest you scan through Chapter 3 of the Game Maker Manual. There are some sprites included with the Game Maker installation, and to make things easier (this is not an image creation guide) I am going to use them in the game.


For the player sprite, we are going to use the image called "SR71.bmp". It can be found in the "Sprites \ Transport" folder of the Game Maker installation directory. The image looks like this:



Ah, yes! The SR-71 Blackbird is my absolute favorite plane! Add a new sprite. In the name box, write "sprPlayer". I always use the prefix "spr" in the names of my sprites, since if a sprite has the same name as an object, errors may occur. So, I consider it a good habit to have a naming convention for sprites and such. Then, when the object is created, you do not have to worry about the name coinciding with a sprite name. Another good thing about this is that later, when you look at your code, for example when debugging, you immediately know if you are referring to a sprite or not with a variable name. For objects, I suggest the use of "obj" as prefix.

OK, so you have named the sprite? Good. Now, click the "Load Sprite" button. In the file selection dialog that appears, browse through the Game Maker install directory until you find the "SR71.bmp" image file. Select it.

Make sure that the checkbox marked "Transparent" is checked (that is, there should be a tick mark in it). Otherwise, check it. This will make parts of the sprite transparent. Which parts? All pixels that have the same color as the pixel in the lower left corner of the sprite will be transparent when the sprite is later drawn on the screen.

Most games I know involve some kind of shooting. For shooting we need bullets. Create a new sprite and call it "sprBullet". For a bullet image, let us use a red ball. Red balls are common in games. Load the image "ball2.gif" into the "sprBullet" sprite (). The image file is located in "Sprites \ Breakout" in the Game Maker installation directory. Make sure the sprite is transparent (see above).

That is all sprites we will need for now.

3.2 Create Objectss

The sprites we have created are just dumb images. OK I agree, they have some intelligence, like transparency and bounding box information, but they really do not do anything. The things in a Game Maker game that actually performs some actions are the Objects. Did you read Chapter 3 of the Game Maker Manual? If not, please read it now, since it explains the meaning of objects.

Create a new object and call it "objPlayer". Now you see it was a good idea to call the Player sprite "sprPlayer", and not just "Player". In the "objPlayer" object's sprite selection box, select "sprPlayer". Now our "objPlayer" object will look like the "sprPlayer" sprite. Great!

Time to create the bullet object. Create a new object. Name it "objBullet" and select the sprite "sprBullet" as the sprite for the "objBullet" object.

3.3 Room for improvement

Now we need a place for the objects to act. This is done in a room.

Create a new room and call it "Room1" (OBS! no space characters). You may be tempted to call it "Room 1" (with a space before the "1"), but then you would have a hard time referencing it from the GML, so never use spaces in the name for your objects, sprites, rooms, etc. If you need to separate two words in the name, use the underscore character instead "_".

Click on the "Background" tab in "Room1" to view the background settings. Make sure "Draw background color" is enabled and click on "background color" (upper right corner of background settings). Select a nice green (like grass) color. Now your room should be all green. The room size should be width: 640, height: 480. If not so, change it to these values. The default Speed setting is 30, which is pretty normal for games. This means that a new game frame will be created 30 times each second. Hence the expression "FPS", Frames Per Second. Not to be confused with "FPS", First Person Shooter... Sorry, just being stupid.

We are now going to place an instance of our "objPlayer" object in the room. Click on the "Objects" tab in the room properties window. In the "Object to add with left mouse" selections box of the room, select "objPlayer". Now click ONCE in middle of the room. This should place an "objPlayer" instance where you clicked. If you happened to click more than once, or move the mouse too much when clicking there might have been created more than one "objPlayer" instance in the room. To remove them, right-click on them. Make sure there is only one "objPlayer" instance in the room.

Here we pause a moment to contemplate on the terms object and instance. To explain this, I am going to use a metaphor. Hope it works. Your object is like a cookie-form, you know the ones you use when making ginger-bread cookies. When placing your objects in your room, you are actually placing "instances" of the objects, which is like stamping out the cookies using your cookie-form. Each instance will act just as described in the object, but each instance will have its own local variables, like x and y position and speed (more on variables later). Just like each ginger-bread cookie you stamp out using your form is shaped like the form, but you can give them all different looks with some icing. Hey! I am getting hungry! Back to the game.

Click OK to close the room window.

3.4 Save, save, save!

Now we are almost ready to start the game, just to check that the "objPlayer" instance is displayed properly in the game room. But before we run it, **SAVE IT!** Remember to save your game often and ALWAYS save it before you run it. It MAY happen, under certain circumstances, that the computer freezes completely and all you can do is to restart it. NOT FUN if your game is not saved.

Right. Save it with an imaginative name (I called mine "GMLTutorial").

Now it is time to start it. Hit F5, or click on the green arrow to start the game.

Okaay! Now we have created the foundation for a Windows game. If you do not see the green background with an instance of "objPlayer" in the middle, you have missed something earlier in the tutorial, or something else is wrong. Check back to see if you missed anything.

Close the game window through pressing [ESC] or clicking on the window's "Close" icon (the cross-mark, you know).

3.5 Action

In order to be able to call our creation a game, we need to be able to interact with it in some way, and preferably something should be moving too. We will start out by making it possible to move the "objPlayer" with the cursor keys of the keyboard.

Back in Game Maker, double-click on the "objPlayer" object to open it. Now we are going to create some actions. When something happens to an object, it is called an "event". The object's response to this event is called an "action". What we want is that when we press any of the cursor keys, the "objPlayer" should start moving in that direction.

There is a button in the "objPlayer" window that says "Add Event". Click it. It brings up a new window called "event selector". Here it is possible to choose which events that the object should respond to. Click on the event button called "Keyboard". It contains a list with some sub-lists containing events that are triggered when different keyboard keys are pressed. Select the "<Left>" key in this list. Now the event list of the object should have an event called "<Left>" in it. It should be selected.

Now we can define the actions that should take place when the left cursor key is pressed on the keyboard. The list of actions is to the right of the object window. Find an action that is called "Start moving in a direction". It is the top left action on the "Move" tab, represented by 8 red arrows pointing away from the middle. Drag this icon onto the white space between the event list and the action list. We can call this the "Action sequence".

A window will pop up when you drop the action in the action sequence. In this window you can specify the parameters that are needed to define the action. Click on the left arrow to select it in "Directions", and set the speed to "5". Then click "OK".

What we now have done is to define that when the left key is pressed on the keyboard, the "objPlayer" will start moving left.

Now, add the event for the "<Right>" key in event list. Look above to see how to add an event if you have forgotten how it is done. Add the "Start moving in a direction" action to that event, set the Right direction, and set the speed to 5.

Repeat this for the keys "<Up>" and "<Down>", setting their corresponding direction in the "Directions" section of the "Start moving in a direction" action.

Save the game and start it again.

You should now be able to move the plane using the cursor keys. Note however, that it is not possible to stop the plane. Neither can you move diagonally.

3.6 Refining the actions

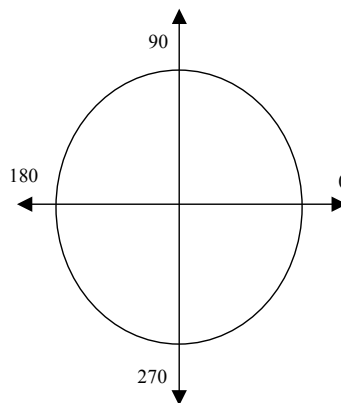
We will now refine the actions a bit, as well as add a "Shoot" action.

Open the object window for "objPlayer". Add the event "<No key>", which is also found among the other "Keyboard" events. This event will happen when all keys on the keyboard are released. Add a "Start moving in a direction" action and select the square in the middle of "Directions". Click OK. This should make the "objPlayer" stop when no key is pressed.

To add a shooting action, we need to decide which key should be used to fire the bullet. I usually choose the [SPACE] key for this.

In the object window for "objPlayer", add the "<Space>" key event. In the action list, select the "Objects" tab to display the actions that have something to do with objects. Select the "Create an instance of an object" (looks like a light bulb) action. Drag it into the Action Sequence list. In the window that pops up, choose the object "objBullet" and tick the checkbox marked "Relative". This means that an instance of the "objBullet" object will be created at the same coordinates as the instance of the "objPlayer" object. Click OK.

Now a Bullet will be created. But it needs to be moving too, to be of any use. To do this, open the object window for the "objBullet". Add the "Create" event. Add the action: "Set direction and speed of motion". This action can be found in the "Move" tab, and looks like 8 blue arrows. In the popup window for the action, enter "90" as the direction and "15" as the speed. This will make the bullet start moving in the direction "90" with speed "15". Directions are measured in degrees and work like this:



So, 90 would be straight up.

One more thing needs to be done before starting the game. What happens to the bullet that reaches the top of the screen? Nothing. It just continues on "forever". So, when enough bullets are fired, the computer's memory will be filled up with data about bullets that moves up, up, up, and we never see them. The thing to do is to make sure the bullet is destroyed once it reaches the top of the screen.

Add the event "Outside room", which is found in the "Other" event list. This event will happen when an instance moves outside the room. Add the action "Destroy the instance" from the "objects" tab of the actions. The default values are OK. Now the bullet will be destroyed once it reaches outside the screen.

Save and start the game and try moving, stopping and shooting.

4 The coding begins

All right. If you have followed the directions above, you should have a small game with a plane that moves and shoots. The reason I created this is just that now that you have done some dragging-and-dropping, we can relate the code statements to these actions to improve the understanding. (At least, that is a theory I have ☺).

Coding. That may sound spooky to some people, while others will relate it to very "cool stuff". Really, coding is like doing the thing we did before, with the icons and such, but with text instead. Actually it is possible to create a game builder where a game creator is able to do everything with icons and drag-and-drop that is possible with coding in Game Maker, but that would mean hundreds of different icons, and the list of icons in an action sequence would be longer than the screen, and it would be impossible to gain a good overview. (Wow, Word complained about a long sentence... ☺)

So, this is actually a situation where a picture does NOT say more than a thousand words.

4.1 First variable

The first thing we did with the drag-and-drop actions was making the plane move when the cursor keys are pressed. We are now going to exchange the action icons for code.

Open the "objPlayer" object and select the "<Left>" key event. Remove the "Start moving in a direction" action from the action sequence list by selecting it and pressing [DEL]. Now, view the "Code" tab in the actions list. The two actions I use the most here are "Execute a script" and "Execute a piece of code". Drag the action "Execute a piece of code" to the action sequence.

What now pops up is something that looks like an empty window – a CODE window <Tension-building music score here>. This is where we enter the code that should be executed as a response to the "<Left>" key event.

Enter the following code:

```
direction = 180;
speed = 5;
```

What this means is that the variable "direction" of the current instance of the "objPlayer" object is set to 180, and the variable "speed" is set to 5. This is called variable assignment. To read more about variables and assignments, see section 24.2 and 24.3 of the Game Maker Manual.

Essentially a variable is a place in the computer memory that can hold a value. There are different kinds of values that a variable can hold. Game Maker differs between two different types of variable information: numbers and strings. A number is just a plain number, like

```
1
5
2.21
63.132
24
```

Strings are lists of characters that are enclosed in single-quotes or double-quotes, like:

```
"Hello"
"This is a string"
'This is another string'
```

The use of both single-quotes and double-quotes for defining a string is one of the "nice" aspects of Game Maker. It makes it possible to easily include a single-quote or a double-quote in a string. Think about it. If you only could define a string with double-quotes, how would you tell the computer that you wanted a string that CONTAINED a double-quote? This code:

```
aLittleString = "And she said "gasp" and fainted";
```

will be very confusing for the computer. It would be treated as TWO strings, with the word "gasp" between them. The same goes for single-quotes. Instead, this code could be used (note the difference):

```
aLittleString = 'And she said "gasp" and fainted';
```

Back to the game.

The variables "direction" and "speed" are built-in variables, and every instance has them. When we are writing variables like this, we are referring to the so-called local variables that belong to an instance of an object. This means that if we would check the value of "direction" in, for example an instance of the "objBullet" object, we would not see the value 5, but instead another value, that is local to the "objBullet" instance.

By setting the variable "direction" to 180, we tell Game Maker that the direction in which this instance should move is 180 (left). Setting the "speed" variable to 5 instructs Game Maker to move the instance 5 pixels each frame, in the direction of the "direction" variable. Fair enough?

So, why is there a semicolon (;) at the end of each string? This tells the program interpreter that this is the end of the statement. Each statement should end with a semicolon. Works about the same as "." (dot) for people. Dots mark the end of a sentence. A statement in a computer program is about the same as a sentence to people. Actually, the GML does not need the semicolon. It understands the code anyway, if each statement is placed on a separate line, but it is considered "good programming" to use semicolons.

OK, now we are done with the <Left> event. Click the green check mark to store the code changes and close the window. Otherwise, this window blocks all other windows. It must be closed when you are done editing.

4.2 First function

Let us go to the <Right> event.

Remove the "Start moving in a direction" action from the <Right> key event. Add an "Execute a piece of code" action instead.

Now, we could do this in the same way as the <Left> event, by setting the "direction" and "speed" variables, but just to learn other ways to do the same thing, we do something different. We are going to use a function.

A function is like a collection of program statements that are bundled together and can be executed by calling the function name.

When your math teacher speaks of functions, he means something like this:

$$y(x) = 4 + 3x$$

This is a definition of a function. The name of the function is "y". The "x" in the parentheses is called an argument to the function "y".

The result of, for example, $y(5)$ would be 19. ($4 + 3 * 5$). The "*" (asterisk) character is commonly used as multiplication operator in computer languages.

What we have done here, if it had been a computer program, would be to call the function "y" with the argument "5". A function is called, takes some arguments, does some computing, and returns a value. Not all functions return values, and not all functions take any arguments at all, but this is the general idea of a function.

Note: in other languages the concept "function" could be called other things, like procedure, method, subroutine etc, but practically speaking they work very much the same.

So, if you look in the Game Maker Manual, on page 80, there is a definition of a function called "motion_set". The definition looks like this:

```
motion_set(dir, speed)
```

The text after the definition in the manual explains that this function will set the speed of an object to the "speed" argument, and the direction to the "dir" argument. OK, so now, let us use this function in the <Right> key event.

Do you still have the code window (empty) for the action in the <Right> key event? Good. Otherwise, double-click it to open it up again.

In the empty code window, write:

```
motion_set(0, 5);
```

Now we have called the function "motion_set" with the arguments "0" and "5" in the places where "dir" and "speed" should be defined. And voilà! When the <Right> cursor key is pressed in the game, your instance should now start moving in the direction 0 (right) with speed 5.

We have now done the same thing, but in two different ways. Which way is the best depends on the situation. In this case, I think the second way, with the function call is the best, but suppose that you just wanted to change the speed of the instance, and not the direction. If so, it would be easier to just assign a new speed value to the "speed" variable.

Later in this guide, we will define and use our own scripts, which works in a similar manner as the built-in functions.

4.3 More variables

There is a third way to accomplish what we have done in the <Right> and <Left> key events.

Select the "<Up>" key event for the "objPlayer" object. Add an "Execute a piece of code" action to the action sequence. In the code window, write:

```
hspeed = 0;  
vspeed = -5;
```

Now, what was that? "hspeed" and "vspeed" are two other examples of variables that are built-in in an object. These variables define the horizontal and vertical speed of the object. If you are a mathematician, you could say that "hspeed" and "vspeed" defines the speed vector in rectangular coordinates, while "direction" and "speed" defines the speed vector in polar coordinates.

Anyway, setting the vertical speed "vspeed" to -5 means that the instance should start moving upwards. That is because the y axis on a computer screen is pointing downwards, so the further down the screen you go, the higher the value of the y coordinate. So, in order to make an instance move upwards, we must have a negative vertical speed.

The horizontal speed is set to 0, meaning that the instance should not move left or right at all.

The only movement left now is the <Down> key. We are going to create it in a fourth way. Close the code window for the <Up> key.

4.4 First script

This time you are going to learn how a freestanding script works. Create a new script (Menu: Add -> Script).

This will bring forth a code window, much like the ones we have used before. The difference is that this window has a name box on its top. Enter the name "MovePlayerDown" there. The name could be anything, but should describe the script's functionality in a good way. Remember; do not use spaces in the name.

Enter the following code in the script:

```
hspeed = 0;  
vspeed = 5;
```

Then you can, if you want to, close the script, but you might as well leave it open.

Go back to the "objPlayer" object window. Select the <Down> key event. Remove the existing action from the action sequence. Add an "Execute a script" action.

A new window will pop up, where you can select which script should be run, and enter some arguments to it. To the right of the "Script:" textbox there is a selection icon. Click it. A list of scripts should appear. Only one script exists in the list so far, the "MovePlayerDown" script. Select it.

Our script does not use any arguments, so leave the rest of the window as it is and click OK.

Now, when the player presses the <Down> key, the "objPlayer" instance will call the "MovePlayerDown" script, which in turn will start the "objPlayer" moving downwards.

What is the point of making a freestanding script? Well, in this case, there is not much of a point, but if the script would be very long, it is easier to maintain if it is freestanding. You do not need to open the object and search for the correct event to find and edit the script. It is also possible to have many freestanding scripts open at the same time. That is not possible with scripts that belong to an object event.

The most important reason to make a freestanding script is that any object may use it. If an instance of the object "objBullet" needed to move downwards with speed 5, it could also call on this new script, and the script would act on the "objBullet" instance instead of on an "objPlayer" instance.

It is also possible to call a freestanding script from another script.

Almost forgot about the <No key> event. That is where the plane is stopped.

Close the code window, and select the <No key> event. Remove the "Start moving in a direction" action and add an "Execute a piece of code" action instead. Write the following in the new code window:

```
speed = 0;
```

The direction does not matter when the speed is 0, right?

Dat's it!

We have now exchanged the graphical drag-and-drop icons for the code text. It did not require much coding, did it?

But there are lots of things to improve with coding, so we are going to continue on this game a bit more.

Save your game and try it. You should not notice any difference. The code works in the same way as the icons did.

4.5 Getting rid of the <No key> event

Actually, the <No key> event is no good for stopping a player-controlled instance. You might have noticed that if you are shooting while moving, and release the move key, but keep holding the shoot key, you are still moving. This is because since you are holding down the shoot key ([SPACE]) you are not getting any <No key> event. And you can imagine how many <No key> events we would receive in a two-player game. No, not many. How can we solve this?

This is the way I would solve it.

Open the "objPlayer" object and select the <No key> event. Then click on the button "Delete" below the events list. This should remove the <No key> event and all its actions. (You may have to answer "Yes" to an "are you sure?" question first though).

Select the <Left> key event, and double-click on the "Execute a piece of code" action in the action sequence. This should bring up your code window.

Instead of setting a speed and a direction for the "objPlayer" here, we could just change the coordinates for it ourselves. In that case it would ONLY move while the key is pressed. Another thing we will gain with this is that we will be able to move diagonally as well. Great!

So, delete all code in the code window, and write this instead:

```
x = x - 5;
```

This means that we set the variable `x` to itself minus 5, that is, we decrease its value by 5.

The `x` variable of an instance contains its x-coordinate in the room. So, by decreasing the `x` variable, the instance will move 5 pixels to the left. This is repeated for each game frame as long as the player holds down the <Left> key. That is exactly what we want.

We want the same thing with the other direction keys, so let us change the code for the other keys too.

Open up the event for the <Right> key in the "objPlayer" object. Double-click on the action in the action sequence to open the code window. Delete the "motion set" function call and write the following instead:

```
x += 5;
```

This is the same as writing

```
x = x + 5;
```

only this is shorter.

Now I think you know what to do with the <Up> and <Down> keys. That is right. The code for the <Up> key event should contain:

```
y -= 5;
```

and the code for the <Down> key event should be:

```
y += 5;
```

For the <Down> key we made a freestanding script. We could as well keep it and edit that instead of changing the action into an "Execute a piece of code" action. So, just open the script "MovePlayerDown" and change the code as stated above.

Now it is time to save the game and run it again. There are two things I want you to notice about the game now.

1. The plane does not continue moving when the fire key is held down but the move keys are released. Good.
2. It is possible to move the plane diagonally. That is good too.

4.6 Bullet loading time

When shooting bullets from the plane you may have noticed that they come in a never-ending, uninterrupted flow. We **may** want this, but I do not think it looks very good. So, let us add some loading time to the gun. To do this, we will use the alarm feature of the "objPlayer" object, and a local variable.

The theory behind the loading time goes like this:

When the [Space] key is pressed, before firing a bullet, a local variable is checked to see if the gun is ready to fire. We can call this variable "gunReady". The variable "gunReady" will take on one of two values. Either it is "true", that means that the gun can fire, or it is "false", which means that the gun cannot fire right now. If "gunReady" is true when [Space] is pressed, a bullet is fired, and the gunReady is set to "false". This means that when [Space] is pressed again, the gun will not fire. However, at the same time as we set "gunReady" to false, we set an alarm timer. When the timer runs out, it is going to set the variable "gunReady" back to "true" and we can shoot again. This will repeat itself during the whole game.

OK. The first thing to do is to make sure that the variable "gunReady" has a value the first time it is checked when pressing the Fire button. This is called

to initialize a variable. If it is not set to a value before it is checked, the game will be halted with an error.

The "CREATE" event of an object is a good place to initialize variables. Open the object window for the "objPlayer" object and add the "CREATE" event. Add an "Execute a piece of code" action to the event. In the code window that appears, write this:

```
gunReady = true;
```

That will set the variable "gunReady" to "true". The word "true" is one of Game Maker's built-in constants. Practically speaking, using the word "true" is the same as using the number "1". The word "false" is another of Game Maker's built-in constants. It represents the number "0". That means that we could as well write

```
gunReady = 1;
```

and achieve the same result. But using the words "true" and "false" kind of makes more sense.

I will take this time to introduce a new concept in coding; the concept of comments. Comments are very important in code. The comments are meant for the human reader of the code, and not for the computer, which will simply ignore it. Use comments a lot to describe what you mean with your code. This will make it a lot easier later when you want to change something. Or debug it. To add a comment, write the two characters "//" before the comment. Like this:

```
// Make the gun ready to fire.  
gunReady = true;
```

The computer will completely ignore the comment and execute the other code. But when someone sees this code segment, they will understand more about what is happening than if they only saw the computer code.

A comment can also be added after a program statement, like this:

```
gunReady = true; // Make the gun ready to fire.
```

but I prefer the style with the comment on its own row. You do whatever you like.

Now, remember how we shoot the bullet? In the <Space> key event of the "objPlayer" we create an instance of the bullet object. Open the <Space> key event for the "objPlayer". Delete the "Create an instance of an object" action from the action sequence. Add an "Execute a piece of code" action to the event. This will, as usual, open up a code window. The first thing we will do here is to check if the gun is ready. That is, we will check if the variable "gunReady" is "true". How do we "check" the value of a variable? That is what the "if" statement is for.

The definition of the "if" statement is like this (excerpt from the manual):

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>
}
```

Hmmm. That might be a bit hard to understand if you are not a programmer. To clear things out, I will simply write out the "if" statement as it should look in our code window. Write down this:

```
if (gunReady = true) then
{
}
```

That was not so hard, was it? This means that if the variable "gunReady" has the value "true", the code that we (later) put inside the "curly braces" will be executed. Otherwise, it will just be ignored. I have added the word "then" after the parenthesis, but that is not required. It just adds to the readability of the code. You may use the "then" word or not, as you like. The so-called "curly braces" will be used a lot in the code. They come from the C/C++ language. The thing is that if we just want to perform one single code statement inside the "if" statement, the braces are not needed, but it is considered good programming style to always include them. The braces kind of create a "block" of code that, to the language interpreter, looks like a single statement. To understand why this is useful, consider the following example.

If the variable "speed" is higher than 5, we want to move the object to the position where x = 40 and y = 80. Simple enough. So, we write:

```
if (speed > 5) then
x = 40;
y = 80;
```

But THAT will NOT work as we would expect it to do. The "if" statement only affects the FIRST statement, just below it. That means that if "speed" is NOT higher than 5, the "x = 40" will be skipped, but the "y = 80" will be executed anyway. To solve this, we need either another "if" statement for the "y = 80" statement, that is the same as the first "if" statement, or, much easier, we could use "curly braces". Like this:

```
if (speed > 5) then
{
x = 40;
y = 80;
}
```

This means that both the "x" thing and the "y" thing are included in the "if" statement. Now, if "speed" is NOT higher than 5, neither "x = 40" nor "y = 80" will be executed. If "speed" however IS higher than 5, both the following

statements will be executed. Now, do you understand the point in using "curly braces"? They are used for more statements, but we will take them as we go.

Now, back to the game code. In your code window you should now have a complete "if" statement with "curly braces". But there is still nothing inside the braces. Here we will create an instance of the "objBullet" object. Earlier this was done using a drag-and-drop action. Now we will do it in code. It is quite simple. We will use the function "instance_create". It can be found in the Game Maker manual, in section 26.2, page 83.

Inside the curly braces, enter this:

```
instance_create(x, y, objBullet);
```

There! Now you have created a bullet. But what about the "x" and the "y"? Should we not enter some numerical value there? Actually that is what we have done. The variables "x" and "y" holds the position of the current "objPlayer" instance (remember, we are in the "objPlayer" object while coding), and these coordinates will do as the starting coordinates of the "objBullet" instance. This is the same as creating an instance with the drag-and-drop icon and checking the "Relative" checkbox.

Now, that was not all we should do, was it? No. We also should set the "gunReady" variable to "false" to make sure that the gun can not be fired immediately again. Enter this, just after the "instance_create" line:

```
gunReady = false;
```

That was easy!

Now we will have to set an alarm to make sure that the "gunReady" variable becomes "true" again, after some time. Setting an alarm is done like this:

```
alarm[0] = 10;
```

There are 8 alarm clocks (alarm[0] – alarm[7]). We have used alarm[0] here. These brackets ("[]") are used to define an array. An array is like a list of variables that all have the same name, but are numbered to make them differ from one another. We will look closer at arrays in another tutorial. Now the alarm "alarm[0]" will trigger in 10 frames.

We are now done with the code in the <Space> key event. If you want, you could shorten the "if" statment row like this:

```
if (gunReady) then
```

Because that is the same as checking if "gunReady" contains ANY positive number, which it does if it is "true".

So, the code should now look like this:

```
if (gunReady) then
{
    instance_create(x, y, objBullet);
    gunReady = false;
    alarm[0] = 10;
}
```

The three lines between the curly braces are "indented", that is, you should add a "TAB" character before them. This is just to make the code easier to read. The language interpreter does not care about "indentations" at all, but it is good for human readability. It is easier to see which part of the code is collected inside a "curly braces block".

Alright!

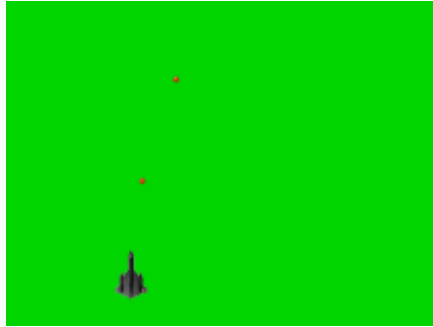
One thing remains; the alarm[0] event. Close this code window and add the event for alarm[0]. Add an "Execute a piece of code" action and write the following code in the window:

```
gunReady = true;
```

That will make sure that when the alarm "goes off", 10 frames after the bullet has been fired, the variable "gunReady" is set to "true" again.

Now, save and test your game!

If you have followed the directions correctly, the "objPlayer" should now fire with longer intervals (about 3 bullets per second). Much better.



5 Enemies sighted!

There are very few games that do not include computer-controlled enemies to the player. After all, we must have someone to fight. In this chapter we will add some enemy craft and learn to use the "random" function and check collisions.

5.1 Enemy aircraft

I have decided that the enemy should consist of enemy aircraft that appear at the top of the screen and fly across to the bottom of the screen.

We need to do the following:

- Have the craft appear at random time intervals.
- The craft should appear at random places along the top screen.
- When the craft disappear at the bottom of the screen, they should be destroyed.
- Later we will add a collision event between the enemy craft and the "objPlayer".

First, we must create an enemy craft sprite and an enemy craft object.

I have decided to use the image "Mig41.gif" from the folder "Sprites \ Transport" in the Game Maker directory. It looks like this:



Create a new sprite, call it "sprEnemy1" and load this image into it. Did you forget how? Check *chapter 3.1 Some sprites*. The craft is however facing upwards, and we need our enemy craft to face downwards, unless of course we want it to attack in reverse, but I do not think that would look good.

In order to rotate the craft, open the sprite and click the "Edit sprite" button. Click on "image 0" to select it, and choose "Transform -> Rotate 180" from the menu. That should rotate the craft to face downwards. Click OK to close the sprite edit window. Click OK again to close the sprite window.

Now we need to make an enemy object. Create a new object and call it "objEnemy1". Choose "sprEnemy1" as the sprite for the object.

We want the enemies to come down at certain intervals and fly across the screen. In order to control the creation of the enemies, we need a control object. We **could** use the "objPlayer" as this object, but I prefer not to. I would rather have a dedicated enemy control object.

Time to explain the term "control object". The concept of a control object is to have an instance of an object that is always present in the game. It should not be possible to shoot it down or otherwise destroy it. This control object is used to create instances of other objects, such as swarms of enemies that come flying down. Instances of control objects are also good at keeping some data in their local variables that cannot be kept in, for example the instance of the player object, since it is destroyed at times (we will come to that). A control object is usually invisible, so that the user does not know it is there. It does its dirty job "undercover".

So, the instance of this control object will not be visible when playing the game, but we need it to be represented by some sprite anyway, in order to place it and see it in the room. So, we do like this.

Create a new sprite, called "sprEnemyController" and load into it the image "trigger.gif" from the "Sprites \ Maze" folder in the Game Maker directory. That looks good for a controller, doesn't it ☺ ? Create a new object, call it "objEnemyController" and assign the sprite "sprEnemyController" to it. Make sure the checkbox "Visible" in the object window of "objEnemyController" is **unchecked**. That will make the controller object invisible to the player, but we will be able to see it when designing the game.

Open Room1. Add an instance of the "objEnemyController" object somewhere in the room. It does not matter where you place it. Just somewhere you can see it. Also, the player instance (the "objPlayer") should be placed at the bottom of the room in order to give the player time to react before the enemies come. If you placed the "objPlayer" somewhere in the middle of the room, like I did, delete it by right-clicking on it, select the "objPlayer" instance from the "Object" selection box and place a new instance of the "objPlayer" object in the middle, near the bottom of the screen. There!

Now we should create the script for initializing a new enemy. Create a new script and call it "Enemy1Init". For a starter we only want the enemy craft to travel down across the screen. So, in the new script we write:

```
vspeed = 10;
```

That will make whatever object calls the script move downward across the screen with a speed of 10.

We also need the enemies to disappear once they move below the screen. Create a new script and call it "EnemyDisappear". Enter this code:

```
instance_destroy();
```

That code is a call to the function "instance_destroy", which destroys the instance that calls it. See chapter 26.2 (page 83) of the Game Maker manual for more information about this function.

That is all code needed for the enemy so far. We will make it a little more advanced later on.

Now we will add two scripts for the enemy controller object. Create two new scripts and call them "EnemyControllerInit" and "CreateEnemy1". In the script "EnemyControllerInit", write the following:

```
alarm[0] = 30;
```

That will set the alarm[0] function of the "objEnemyController" to trigger after 30 frames. That is about 1 second in "real time" provided that the room speed is 30 FPS. When the alarm triggers it will call the other script, called "CreateEnemy1". In that script, write:

```
instance_create(50, 0, objEnemy1);
alarm[0] = 30;
```

The first line will create a new instance of the object "objEnemy1" at the x-coordinate 50 and the y-coordinate 0. Check chapter 26.2 (page 83) of the Game Maker manual for more information about this function.

The second line sets the alarm[0] once again to 30 frames (1 second) to make sure another enemy appears after 1 second.

Now we should make the objects call these scripts. Open the object "objEnemy1". Add the event "CREATE". This event will happen when an instance of the object is created. Add the action "Execute a script". In the

window that pops up, select the script "Enemy1Init". Click OK to close the action window.

Add the event "Outside". Add the action "Execute a script" to the event and select the script "EnemyDisappear". Click OK to close the action window.

Open the object window for "objEnemyController". Add the event "CREATE". Add the action "Execute a script" and select the script "EnemyControllerInit". Close the action window. And, finally, to the event "Alarm 0", add the action "Execute a script" and select the script "CreateEnemy1".

Save the game and run it.

You should see enemy aircraft appearing and flying across the screen, disappearing at the bottom. Note that the instance of the object "objEnemyController" is not visible when running the game. If it is, you forgot to uncheck the checkbox "Visible" in the object window of "objEnemyController".

5.2 Randomizing

The appearance of the enemies is quite boring though. We would want them to appear at random positions along the top of the screen, and at random intervals and random speeds. That would make the game a lot more interesting, don't you think? That is what the "random" function is for.

Check out the definition for the function "random" in the Game Maker manual, chapter 25.2 (page 78). The random function could be used to make things happen randomly, for example simulating a dice. It returns a random value that is always less than the specified value. Like this:

```
random(3);
```

The above line will return values from 0 to 2.99999999999999999999. If I am correctly informed, Game Maker works with 20 decimals.

To get rid of the decimals, we could use another Game Maker function, called "floor". This function simply takes away all decimals, so that, for example:

```
floor(1.23423) = 1
floor(2.999999999) = 2
```

If we wanted to simulate a 6-sided dice, we would use:

```
myDice = random(6);
```

The above line would give values from 0 to 5.9999... Then we use the floor function:

```
myDice = floor(myDice);
```

As the argument to the floor() function, we use the variable "myDice" itself. This is perfectly OK. The result will now be an integer between 0 and 5. Almost there. We could now add 1 to the result.

```
myDice = myDice + 1;
```

And finally the result would be values from 1 to 6, all with the same probability.

To save some place in the coding window, we could do all there calculations in one line:

```
myDice = floor(random(6)) + 1;
```

The above line might be a bit hard to read, but it carries out all the previous calculations in one statement.

Now we will put the random function to work in three places of our code. Open up the script "CreateEnemy1".

First we want the enemies to appear at random positions along the top of the screen. That means we want to use random values ranging from 0 to 639, which is the width of our game screen (640 pixels wide). The statement

```
random(640);
```

will produce values from 0 to 639.999999. Adding the "floor" function,

```
floor(random(640));
```

will produce values from 0 to 639 with no decimals. That looks good. But what if we later want to change the screen size? No problem. In Game Maker there is a variable that can be used to determine the size of the screen. It is called "screen_width". So, instead of using "640", we will use "screen_width". (see manual chapter 28.6, page 103). So, now we can change the first line of the script to read:

```
instance_create(floor(random(screen_width)), 0, objEnemy1);
```

Be VERY careful with the parentheses, since Game Maker might crash if they are not all there.

If you want, you can save the game and test it.

We also want the enemies to appear at different time intervals. Say between 0.3 and 2 seconds. 0.3 seconds mean 10 frames, and 2 seconds mean 60 frames. So we want a random number between 10 and 60. Change the second line in the script to:

```
alarm[0] = floor(random(51)) + 10;
```

That will result in alarm times from 10 to 60 frames.

Save and run the game again.

Finally, we want the enemies to fly at random speeds. Open the script "Enemy1Init" and change the code line to this:

```
vspeed = random(8) + 2;
```

This will give speeds from 2 to 9.999999. We do not use "floor" here, since speeds with decimals are OK.

Test the game. Now you will see that the enemies appear at different places, at different times and at different speeds. Just what we wanted!

5.3 Ouch! That hurt!

So far, the enemies are not dangerous. There is no point in avoiding them. Well, that we will have to remedy!

If you have created any game before, you have probably used the Collision Detection feature of Game Maker. It is really great. We will use it now to detect collisions between the player and the enemies.

Create a new script and call it "PlayerEnemy1Collision". Open the object window for the "objPlayer" object and add the event for collision with the "objEnemy1" object. That is the event button with the two red arrows pointing at each other. Click on the selection box next to it and select "objEnemy1". Good.

Here you add an "Execute a script" action and select the script "PlayerEnemy1Collision" for the action. Close the action window and the "objPlayer" object window.

Now we can concentrate on the script. We want to make it possible to run into a number of enemies before the player dies. Otherwise the game may be too difficult to play. So, we are going to use something we call "Energy" to measure how much beating the "objPlayer" can take before it blows up. Let's say the "objPlayer" starts out having 100 energy units. Then, every time it runs into an enemy craft it will lose 30 energy units. Fair enough? Then, when it reaches 0 energy units, it will be destroyed. The enemy craft will be destroyed immediately when hitting the "objPlayer".

In the collision event script we will therefore need to decrease the energy of "objPlayer". It could be done like this:

```
myEnergy -= 30;
```

We also need to check if the energy is 0. If it is, the instance of "objPlayer" should be destroyed. Remember the "if" statement?

```
if (myEnergy = 0) then
{
    instance_destroy();
}
```

Here I have deliberately included a bad thing about the "if" statement. Can you see what is wrong? It is the test, "=". This statement will only check if the energy is **exactly** 0. But what about if the energy is first 10, and then you run into an enemy and lose 30 energy units, and the energy becomes -20? Then this "if" statement will not trigger and destroy the instance of "objPlayer". So, it is better to use the "<=" test operator. It means "Less than, or equal to". So, write this instead:

```
if (myEnergy <= 0) then
{
    instance_destroy();
}
```

That will destroy the "objPlayer" if the energy is 0 or less than 0.

We also want the enemy to be destroyed, so add the line

```
with (other) instance_destroy();
```

to the BEGINNING of the script.

This introduces the "with" statement. The "with" statement is extremely useful. It allows us to do something to another instance. In this case we want to destroy the instance that we collide with. This is called the "other" instance in a collision event. So **with** the **other** instance we want it to **destroy** itself. The code line above is the same as writing "instance_destroy()" in the code of the other instance's object.

So, the entire script "PlayerEnemy1Collision" should now look like:

```
with (other) instance_destroy();
myEnergy -= 30;
if (myEnergy <= 0) then
{
    instance_destroy();
}
```

Now we only need to set a starting energy level. We decided to start at 100 energy units. This should be set in the "CREATE" event of the "objPlayer". We had better create a script for it, this is what this tutorial is for after all. So, create a new script and call it "PlayerInit".

Enter the following line in the script:

```
myEnergy = 100;
```

If we do not initialize the variable "myEnergy" to anything, an error will appear when the "objPlayer" hits an instance of the "objEnemy1" object. You could try to run the game (save first!) before setting the CREATE event of the "objPlayer" to see what it looks like when you forget to initialize a variable. It could be good to know, because it is easy to forget it. When you hit another craft it says:

```
"Unknown variable or function: myEnergy"
```

So, click "Abort" and let us set the CREATE event of the "objPlayer" object. Open the "objPlayer" object, select the CREATE event. Here you will see the "Execute a piece of code" action that we added earlier. Double-click on it. It should only contain one line of code; the initialization of the "gunReady" variable. This is a good thing to move to our new "PlayerInit" script. So, select the code in the code window and copy it. Close the code window and delete the action "Execute a piece of code" from the action sequence. Add an "Execute a script" action and select the script "PlayerInit". Now, open the script "PlayerInit" and paste the code from the action we just deleted. If you have done it right, the script "PlayerInit" should now contain:

```
gunReady = true;
myEnergy = 100;
```

Now, save the game and run it again.

You should notice that the enemy aircraft disappear when you run into them. When you run into the fourth craft, the instance of "objPlayer" too disappears. Right! Now there is a point in avoiding the enemies.

5.4 *Shootout*

So far your bullets have done nothing to reduce the oncoming swarm of enemy craft. The point of having a gun in a game is to be able to do some damage. So we need the bullets fired from the "objPlayer" instance to damage the enemy craft.

We will make a script that takes care of the collision between a bullet and an enemy craft. Create a new script and call it "BulletEnemyCollision".

This script should work in almost the same way as the collision script for the "objPlayer" and "objEnemy1". The energy level of the enemy should be decreased, and a check should be made to see if the energy level is 0 or less. Let's say that the enemy craft start out with an energy level of 100, they too. Then when a bullet hits them, their energy level should be decreased with 50 units, which means that it takes two bullets to kill an enemy.

Here is the script we will use:

```
with (other)
{
    // Lower enemy energy
    myEnergy -= 50;
    // Check if enemy energy is 0 or less
    if (myEnergy <= 0)
    {
        // If so, destroy enemy.
        instance_destroy();
    }
}
// Destroy this bullet
instance_destroy();
```

The difference between this script and the previous is that here most of the work is done on the enemy instance, and not on the instance of the object that contains the code (the "objBullet" object).

To have more than one statement inside a "with" statement, you could use the curly braces like this. As you can see, it is also possible to "nest" the curly braces. That means, in this example, that you can have an "if" statement *inside* a "with" statement. When you do like this, you usually use two TAB characters in the beginning of the deepest nested lines to show that they belong inside the "if" statement. Once again, the TAB characters, or, as it is also called, "indentation" is only important to the human eye. The computer does not care.

So, this script lowers the enemy's energy with 50 units and checks if the energy level is 0 or less. If so, the enemy is destroyed.

Finally the script also destroys the bullet. Otherwise it would have continued to move across the screen. I have also put some comments in this script to show how I usually use comments. They are supposed to increase the readability of the code and make it easier to understand.

Open the object window for the "objBullet" object. Find and select the collision event with "objEnemy1". Add an "Execute a script" action and select the script we just created ("BulletEnemyCollision").

Now we only need to give the enemies a starting energy level. We already have an initializing script for the enemies, so let's use it. Open up the script "Enemy1Init" and add the line:

```
myEnergy = 100;
```

That should set the starting energy for the enemies to 100.

Save the game and try it out. Great! Now it is possible to shoot the enemy craft, but only the slow ones. The ones moving fast are hard to hit.

5.5 Pyrotechnics

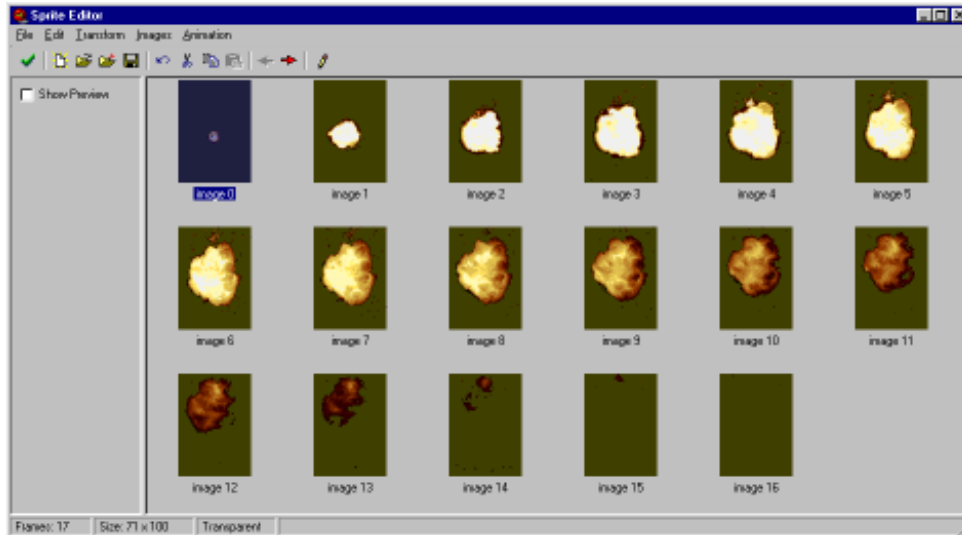
Explosions! They are what are missing from our game. Fortunately Game Maker comes bundled with a nice explosion animation. We will create an explosion object from it.

First, we need to create the sprite. This time it will be an animating sprite, meaning it has more than one image. Create a new sprite and call it "sprExplosion". Load into it the image file that is called "explode2.gif". It should be located in the Game Maker directory, in the "Sprites \ Various" folder. When you have loaded it, you will notice that the sprite window says the number of subimages is 17. Click on the "Edit Sprite" button.

You will now see all 17 subimages of the sprite. They are named "image 0" to "image 16". To see them animated, check the little checkbox in the upper left corner of the Sprite Edit window, the one that is called "Show Preview".

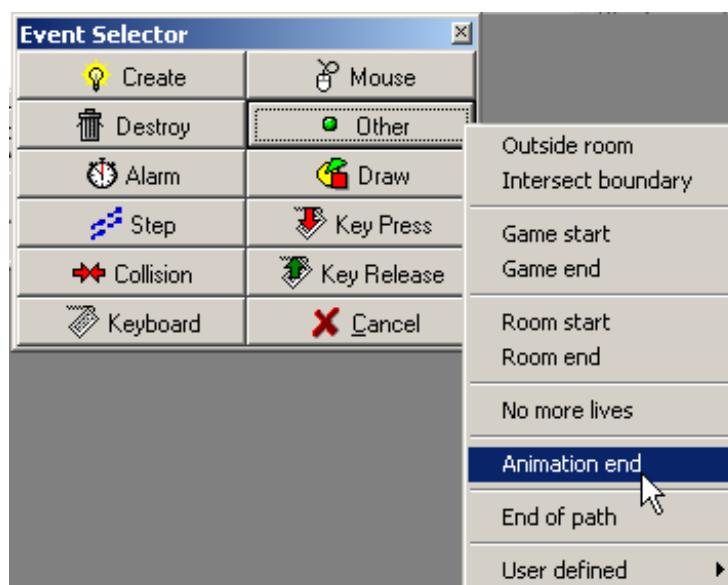
If you look at the images, you can see that image 0 displays the explosion already a bit exploded. It seems that the animation starts somewhere in the middle of the explosion. This is strange, and I have no idea why it is like this. We will have to change it in order for the explosion to look correct when used in the game.

Select image 0 and click the red arrow pointing to the right in the toolbar. This moves the image one step to the right. Keep clicking on the right arrow until the image is moved all the way to the end of the animation. The name of the selected image should now be image 16. Now, select the new "image 0" and move it to the end of the image sequence. You can use the key combination [CTRL] + [R] instead of clicking on the right arrow all the time. Keep moving images like this until image 0 is the image with the smallest bright spot and image 16 is an empty image with just the green background. It should look like this when you are ready:



If you check the "Show Preview" box again, you should notice no difference. But that is just because the animation is looped endlessly when looking at the preview. When using it in the game, we will only see the sequence once, and then it is more important where it starts.

Close the sprite edit window and the sprite window. Create a new object, called "objExplosion". Select the sprite "sprExplosion" for the new object. In the "ANIMATION END" event of the "objExplosion" object, add an "Execute a piece of code" action. The "ANIMATION END" event can be found in the selection box of the "Other" events key:



In the code window that pops up, destroy the instance:

```
instance_destroy();
```

That will destroy the explosion instance once it has played through its animation frames.

Now, open the object "objEnemy1". In the "DESTROY" event of "objEnemy1", add an "Execute a piece of code" event. We use this action instead of the script action because the code we will execute here is so small that it is completely unnecessary to have a freestanding script for it. When the enemy is destroyed we want an explosion to show up, so we want to create an explosion instance when the enemy is destroyed. In the new script window that pops up, write:

```
instance_create(x, y, objExplosion);
```

That will create an explosion at the same coordinates as the "objEnemy1" instance. Remember, for an instance, the variables x and y contains the coordinates for that instance.

Save the game and start it. Now there is a beautiful explosion showing up whenever an enemy is shot down.

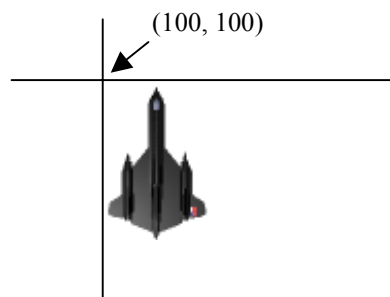
Better add the explosion to the "DESTROY" event of the "objPlayer" too. You should be able to do that on your own now. I will not tell you how to do ;). Use the same "objExplosion" object as for the enemy.

6 Enhancing the game

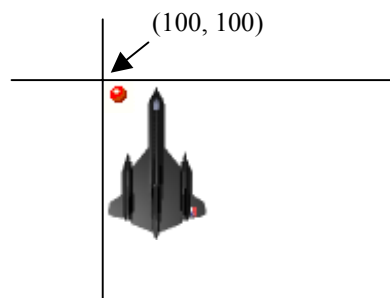
There are some parts of the game that need enhancement to look good. For example the bullet does not come out at the middle of the "objPlayer", and the enemies sort of "pops up" on the screen instead of flying into it. Time to fix that.

6.1 Centered sprites

We will begin to look at why the bullets do not come out from the middle of the player plane. Open the sprite "sprPlayer" and click on the "Advanced" tab. Here you will find the "Origin" settings. It consists of an x value and a y value. These values determine the so-called origin of the sprite. Their default values are (0, 0). That means that if we place the sprite at location (100, 100), in the room, it will look like this:



Then, when a bullet is created at the same location as the plane, it will look like this:

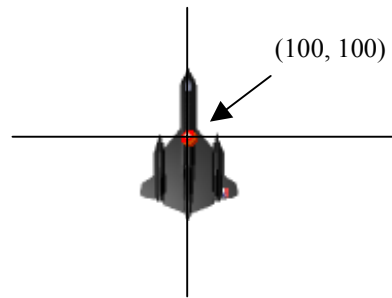


That is because the bullet sprite is much smaller than the plane sprite.

We would rather want the ball to start in the middle front of the plane. What we do is that we "center" the origins of both sprites. Start with the "sprPlayer" sprite.

Click on the "Standard" tab of the "sprPlayer" sprite again. Here you can see the width and height of the sprite. Note that the width of the sprite is 52, and the height is 78. Now, go back to the "Advanced" tab again. Enter "26" as the X origin, and "39" as the Y origin ($26 = 52/2$ and $39 = 78/2$). Do the same thing with the "sprBullet" sprite, where you should enter "8" as the origin for X, and "8" as the origin for Y (this is the middle of the sprite).

Now, if the player sprite and the bullet sprite are placed at the same location, for example (100, 100), it will look like this:



The center of the bullet sprite will coincide with the center of the player sprite. Good. But we would like it to start more like just in front of the plane. This will be fixed through fiddling a bit with the "instance_create" function, but first, save and run the game to see the difference.

Notice that the bullets now appear at the center of the player plane?

Now, open up the windows for the "sprExplosion" sprite and center it. The origin coordinates should be (35, 50), since the sprite is 71 x 100 pixels large. Then, open the "sprEnemy1" sprite and enter (32, 32) as the origin of that sprite. Good. Now all sprites are what I called "centered".

Open the "objPlayer" object and select the <Space> key event. Double-click on the "Execute a piece of code" action in the action sequence to open up its code window. This is where an instance of the "objBullet" object is created. We want the bullet to be created a bit higher up on the screen. Higher up means lower y-coordinates. So, change the "instance_create" line so that it reads:

```
instance_create(x, y - 15, objBullet);
```

Close the code window, save the game and try it out. Hmm. A bit better, but the bullet should appear even higher up. OK. Let us change the code again. This time, change it to:

```
instance_create(x, y - 25, objBullet);
```

Then try out the game again. This looks good to me. Sometimes you just have to try out different values until things look good.

6.2 Smoother enemy appearance

The enemies still pop up out of nowhere. It would be nicer if they kind of flew into the screen. That could be fixed through making sure they are created *outside* the screen and then fly into it. This will bring up a small problem with the "Outside" event, but we will look at that as it comes up.

Open up the script called "CreateEnemy1". This is where the enemy instances are created. The "instance_create" function is used to create an enemy instance at a specified x and y coordinate. We want to change this so that the enemy is created higher up, which means we will have to lower the y coordinate of the creation point. Change the first line in the script to read:

```
instance_create(floor(random(screen_width)), -64, objEnemy1)
```

Save and run the game. You will notice that *no enemies appear at all*. Why is this? This is because of the "Outside" event of the enemy object. Open the object "objEnemy1" and select the "Outside" event. You will see that this event executes the script called "EnemyDisappear". This script is called every time the enemy instance is located outside the screen. But we do not want

the enemy to disappear when the instance is *above* the screen, only when it is *below* the screen. OK, so let us do a test in the "EnemyDisappear" script. Open the script and change it so that it looks like this:

```
if (y > screen_height + sprite_yoffset) then
{
    instance_destroy();
}
```

That will check if the enemy is located *below* the screen, that is, if the y coordinate of the enemy is larger than the screen height *plus* the *enemy sprite origin*. The y coordinate of the sprite origin is automatically stored in the variable "sprite_yoffset", which we use here. This is needed, because otherwise the enemy would be destroyed as soon as the origin was outside the screen, which would mean that half the enemy sprite would still be inside the screen and visible. Not good. This script will make sure that the entire enemy sprite is outside the screen before destroying the instance.

If you try out the game now, you should notice that the enemies appear smoothly flying into the screen, and flying out from the screen. But there is a part of an explosion showing up as the enemies are destroyed. This must be corrected. The explosion instance is created in the "DESTROY" event of the "objEnemy1" object. Open it up and double-click on the action "Execute a piece of code" to edit it.

All this code does is to create an instance of the "objExplosion" object. We could make it test the y coordinate of the enemy before creating the explosion so that the explosion is only created if the enemy is still on the screen.

Do to that, we test that the y coordinate is less than the screen height plus the sprite y origin. Change the script so that it reads:

```
if (y <= screen_height + sprite_yoffset) then
{
    instance_create(x, y, objExplosion);
}
```

Close the code window and try the game. Now the enemies should disappear silently, without and explosion. But the explosion should still be created if the enemies are shot down.

6.3 Going global

One thing that I think is very important to know of is global variables. They are variables that do not belong to any particular instance, but are accessible from all instances, all the time. You could think of it as if there was an instance called "global" that contained all global variables.

Global variables are good for containing values like score, health, max and min values and such. The first thing that we are going to use a global variable for is the speed of the player. I think the speed is a bit too slow. To change the speed now requires the change of a number at four different places in the code. It would be better if the speed could be changed by just changing the code in a single place. This could be done in other ways, but we will use a global variable.

Create a script that is called "GameStart". Add the following line to the script:

```
global.playerMaxSpeed = 8;
```

Note the word "global" and the dot before the variable name. This is how global variables are used. To read a bit more about global variables, read section 24.5 of the Game Maker Manual.

Now we need to execute this script from somewhere. The "CREATE" event of the "objPlayer" object seems like a good place. Select that event, add an "Execute a script" action, and select the script we just created. Then move the action up one step so that it is on the top of the list (before the "PlayerInit" script). This is done by just dragging the action to the top of the list.

Now, in the <Left> key event of "objPlayer", change the code that is executed there to:

```
x -= global.playerMaxSpeed;
```

The code in the <Right> key event should read:

```
x += global.playerMaxSpeed;
```

And the code in the <Up> key event should be:

```
y -= global.playerMaxSpeed;
```

Finally, open the script "MovePlayerDown" and change the code to:

```
y += global.playerMaxSpeed;
```

Now, save and run the game.

If we later decide to change the player speed, we only need to change the code in *one* place, in the script "GameStart".

It is a good programming practice to use numbers as little as possible in your games. The numbers should instead be defined in an initialization script, like "GameStart", to global variables or something like that, and then those variables should be used instead. This greatly simplifies any changes that need to be done to the game later.

We will add some other things to the "GameStart" script. First, we add the maximum energy level of the player. Like this:

```
global.playerMaxEnergy = 100;
```

Then we change the "myEnergy" line in the script "PlayerInit" to read:

```
myEnergy = global.playerMaxEnergy;
```

It is a good thing to have all those value definitions in a single place.

We continue adding values to the "GameStart" script until it looks like this:

```
global.playerMaxSpeed = 8;  
global.playerMaxEnergy = 100;  
global.enemy1MaxEnergy = 100;  
global.enemy1Damage = 30;  
global.bulletToEnemy1Damage = 50;
```

There may be more values that can be changed like this, but I will stop here. To use these global variables, we need to change some scripts a bit. Open the script "Enemy1Init" and change the setting of the "myEnergy" variable so that it reads:

```
myEnergy = global.enemy1MaxEnergy;
```

Then open the script "PlayerEnemy1Collision" and change the decrease of the "myEnergy" variable to:

```
myEnergy -= global.enemy1Damage;
```

Finally, open the "BulletEnemyCollision" and change the energy decrease statement to:

```
myEnergy -= global.bulletToEnemy1Damage;
```


Now it is much easier to change those parameters of the game. Save the game. There should be no need to run it, other than to see that no errors appear. Nothing should have changed regarding the gameplay.

6.4 Where's my energy?

We have been talking a bit about the energy of the player and that if the energy goes down to 0 the player is destroyed. However, so far we have not seen any indication of that energy. Time to make an energy meter.

The energy meter will be created without using any kind of sprite. Instead we will have a first look at some of the other drawing possibilities of Game Maker.

In order to draw anything on the screen, an object is needed. We will create another controller object to do this. There will not be any sprite drawn, but we need a sprite anyway, just to represent the instance of the controller object in the room. So, create a new sprite, call it "sprPlayerController" and load the image "Dish.bmp" from the "Sprites \ Space" folder of the Game Maker directory.

Then create a new object, "objPlayerController" and select the "sprPlayerController" sprite for it. Do **NOT** make the new object invisible. I mean this! I have received a huge amount of emails from people who claim that the energy bar does not show up even though they have followed this tutorial by the letter. Then, when I look at their file, I see that they have made the object "objPlayerController" invisible. I agree that most controllers should be invisible, but this one is used to draw an energy bar and therefore **needs** to be visible, otherwise the "DRAW" event of the object will not be executed. You will understand what I mean later on. I wish I had been clearer on this when I first released this Guide, but I was not, and, as you understand, this text you are currently reading was added in the revision (2002-08-15).

Add an instance of the new object to the room. This object will be used to create the "objPlayer" instance. Therefore this instance should be removed from the room. Remove the instance of "objPlayer" (the SR71 plane) from the room through right-clicking on it.

Now the player will not exist when first starting the game, we will have to create the player from the controller. The reason for doing this is that it is not always clear in which order Game Maker creates the instances, at least I do not know in which order they are created. So we must make sure that first the controller instance is created, and then the player instance. Another reason is that the controller needs to know the instance ID of the player instance, and that is easiest to retrieve through creating the player instance from the controller object.

The script "GameStart" should be run by the "objPlayerController" instead of the "objPlayer". Drag the "Execute a script" action that executes the "GameStart" script from the "CREATE" event of "objPlayer" to the "CREATE" event of the "objPlayerController" object.

Now we need a script to create the player. Create a new script and call it "CreatePlayer". Add this line:

```
myPlayer = instance_create(screen_width / 2, 400, objPlayer);
```

That will create an instance of the "objPlayer" object. The x coordinate will be the center of the screen (screen_width / 2), and the y coordinate is somewhere along the bottom of the screen (80 pixels from the bottom).

Notice that I have written "myPlayer = " in front of the "instance_create" function. That is because when the "instance_create" function has created the instance of the player, it returns a value. That is how a function works, remember? Usually we do not care about that value, and that is OK. But this time we want to store the returned value in a local variable, "myPlayer". The value that is returned is the "instance ID" of the newly created player instance. We will use it later to retrieve some local variables from the player instance.

In the object window for "objPlayerController", select the "CREATE" event, add an "Execute a script" action and select the "CreatePlayer" for that action.

Now it is time to draw the energy bar. Add a new script (don't you love them already? ☺ ☺) and call it "DrawEnergyBar".

In the "DRAWING" event of the "objPlayerController" object, add an "Execute a script" action. Select the "DrawEnergyBar" script.

Then, go back to the script. I intend to draw an energy bar that looks something like this:



First we draw the gray background rectangle, and then we draw the blue energy rectangle on top of it. The green frame around the rectangle above is just the background from the room showing. Don't bother with that.

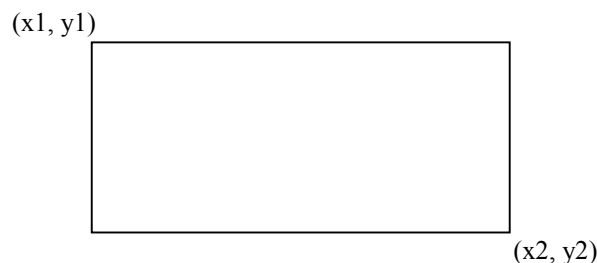
To draw shapes in Game Maker we first need to set a pen and brush color. Enter these lines into the script:

```
pen_color = make_color(150, 150, 150);
brush_color = make_color(150, 150, 150);
```

The "make_color" function creates a color out of three color values. The values inside the parentheses are the red, green and blue levels of the color. The numbers range from 0 to 255, which means that "make_color(255, 0, 0)" would create a very red color, and "make_color(255, 0, 255)" would create a very violet color (red and blue). The values used in the script (150, 150, 150) will create a medium gray color.

The pen color is used to "outline" the rectangle, and the brush color is used to fill it with a color.

When the color setting is done, we could draw the rectangle. To do that we need two pairs of coordinates, (x1, y1) and (x2, y2). See the image below:



We want to put the energy bar in the lower left corner of the screen, so we could do like this:

```
x1 = 5;
y1 = screen_height - 15;
x2 = 110;
y2 = screen_height - 5;
draw_rectangle(x1, y1, x2, y2);
```

Now we have created four variables and used them as coordinates in the "draw_rectangle" function. We could just have entered the coordinate calculations directly into the "draw_rectangle" function, but the line would have been so long, so I chose this way to represent them.

Finally we want to draw the blue bar that represents the energy itself. Change the color to some green color and draw the new rectangle:

```
pen_color = make_color(0, 0, 255);
brush_color = make_color(0, 0, 255);
x1 = 7;
y1 = screen_height - 13;
x2 = 7 + myPlayer.myEnergy;
y2 = screen_height - 7;
draw_rectangle(x1, y1, x2, y2);
```

Here we see what the "myPlayer" variable should be used for. It is used for getting the "myEnergy" variable from the "objPlayer" instance that was created in an earlier script. To get a local variable from another instance, you use the instance ID of that instance, followed by a dot and the name of the local variable that you want. Voilà! (Means "there you go" in French. Honest!)

There is one thing about this that is dangerous though! If the player instance, with instance ID "myPlayer" is destroyed, it is no longer possible to access the local variable "myEnergy" through "myPlayer.myEnergy". This results in a fatal error and Windows may freeze completely. The best thing to do is to test if the "myPlayer" instance exists before using its local variables. This is done using the function "instance_exists" like this:

```
if (instance_exists(myPlayer))
```

Now, this is the complete code of the "DrawEnergyBar" script:

```
pen_color = make_color(150, 150, 150);
brush_color = make_color(150, 150, 150);
x1 = 5;
y1 = screen_height - 15;
x2 = 110;
y2 = screen_height - 5;
draw_rectangle(x1, y1, x2, y2);

if (instance_exists(myPlayer))
{
    pen_color = make_color(0, 0, 255);
    brush_color = make_color(0, 0, 255);
    x1 = 7;
    y1 = screen_height - 13;
    x2 = 7 + myPlayer.myEnergy;
    y2 = screen_height - 7;
    draw_rectangle(x1, y1, x2, y2);
}
```

Make sure that the script contains the code above, save the game and test it.

A blue energy bar with a gray background will now be visible in the lower left corner of the screen.

If the energy bar does not show up on the screen, double-check to see that the object "objPlayerController" really is set to be **visible** (see the beginning of this section).

7 Life, the Universe and Everything

I don't know about you, but I am getting really tired of the green background. Time to do something about that.

This will be the last chapter, since if I do not stop now, the guide may never be finished. Hopefully, if I receive a good enough response, I will continue to make a follow-up with some more advanced programming, but that is for the future to see.

7.1 *A galaxy far, far away*

The most common background for space games would be some stars whooshing by. I know that the SR71 and the MIG41 do not fly in space, but I, being the supreme ruler of this document, have decided to let them into the great void.

I did not find any good enough space background in the Game Maker directory, at least not a scrollable one, so we will have to create our own. But I will keep it simple.

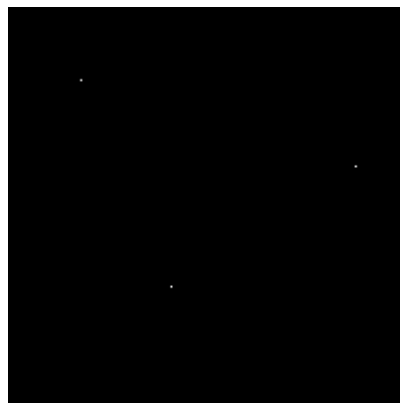
First we will create a background image that looks like simple stars. Create a new background. Call it "bgrStars1". Click the "Edit Background" button. This will open up the Game Maker background editor. It is not the best 2D graphics application in the world, but it knows a few tricks and is quite good for a starter. The first thing we will do is to set the size of the background image. The background image will be "copied" over the entire room if it is not as big as the room. This is called "tiling". To not make the background look too regular we will have to increase the size of this background image a bit.

Select the menu choice "Transform -> Resize Canvas". In the window that appears, look for the two textboxes just before the two "pixels" words. Enter the number "200" in both of these. If the checkbox "Keep aspect ratio" is checked, you will only need to change one of them, and the other will follow.

Click OK. Now our background image should be 200 x 200 pixels large.

Select the black color in the color selection box on the right hand of the window. Now select the bucket icon that is called "Fill an area". Then click in the image. The entire image should now be black.

Next, select the white color and click on the pencil icon that is called "Draw on the image". Use it to click a few stars scattered randomly around the image. Do not draw too many. I drew only 3, and the result looked OK in the game. This is how my image looked:



Click OK to close the image editor. Now, open the room window through double-clicking on "Room1". Then click the "Backgrounds" tab in the room window. Here it is possible to select which background images should be visible in the room. Select "Background 0". Then, in the selection box a bit further down, select the background image we just created, "bgrStars1". Also make sure that the checkbox "Visible when room starts" is checked. The checkboxes "Tile Hor." and "Tile Vert." should also be checked.

Another thing to do here is to *uncheck* the checkbox marked "Draw background color". This will turn off the green color that has been used as background so far. It is no longer needed since the new background image will cover the entire room.

OK. We are done here for now. Click OK, save the game and test it.

If everything is as it should, there should now be some stars as a background to the game. It looks rather boring though. Lets add some movement to the stars.

Open the "Room1" window and click on the "Background" tab again. Far down in the tab you will find a textbox marked "Vert. speed". It is currently set to 0. Set it to 3. Click OK, save and test the game again.

But we are not done with the background yet.

7.2 Cool word: *Parallax*

Time to add some depth to the background. We will use an illusion called "Parallax". It is based on the fact that when you are moving and looking sideways, like out of the side window of your car, objects that are far away, like trees at the horizon, looks like they move past slowly, while objects that are close to the car, like rails and road markings, swish by really fast. This can be used to create an illusion too. The human eye is tricked and the brain thinks that what the eyes see has 3-dimensional depth.

Create a new background image, call it "bgrStars2", and resize it to 200 x 200 pixels, just like above. Paint it black and add some stars to it. But this time the color of the stars should be light gray, and not completely white. You can add a few more stars to this background. Five, maybe.

Then open the room window again and click the Background tab. Select "Background 1" and select the background image "bgStars2" for that background. Make sure the "Visible when room starts" checkbox is checked. Give the new background a "Vert. speed" of "2".

Close the window and try the game. It still looks like a flat starfield. That is because we have forgotten to make the new starfield transparent. OK. Open the background "bgtStars2", and check the little box called "Transparent". Then save and try the game again. Alright! Now there are two starfields moving at different speeds, creating a small illusion of depth.

To make it even better, we will add a third starfield layer. Do just like the second background, but call this one "bgrStars3", and use a darker gray color for the stars in it. You can add a bit more stars too, maybe 10 or so. Then check the "Transparent" checkbox and open up the room. Add the new starfield background as "Background 2" and set the "Vert. speed" to 1.

For the third time, test the game. That looks quite OK, does it not? With just a little bit imagination you will see a "deep" space moving past beneath the plane.

7.3 Enemy fire

It is now time to make the enemies shoot back at you. That will not be so difficult, now that you know a bit about GML. The plan is that with random intervals, the enemy planes should fire a bullet at you. Whenever a bullet hits you, your energy would decrease by 10 units.

First we need a new bullet object. We should not use the one that is fired from the player plane, even if we are going to reuse its sprite.

Create a new object, call it "objEnemy1Bullet" and select the sprite "sprBullet" for it. Now we need the enemy plane to shoot this bullet at the player. Create a new script for this and call it "Enemy1Fire". The script should create a bullet instance and set the alarm timer of the enemy so that another bullet can be created.

Write this in the script:

```
instance_create(x, y, objEnemy1Bullet);
```

That will create an instance of the "objEnemy1Bullet" object in the same coordinates as the enemy plane is. But it is not going anywhere. We need to give it a speed and aim it towards the player. Fortunately there is a Game Maker function that makes this very easy. It is called "move_towards_point", and then you tell the function to which coordinates the instance should move, and how fast.

Now you think that it is as easy as to just write the function. But no, if we just wrote this function, the enemy plane would move towards the player, and not the bullet. To do that we need to get the Instance ID of the created bullet instance. So, change the line you just wrote to this:

```
myBullet = instance_create(x, y, objEnemy1Bullet);
```

Now we have got the instance ID of the new bullet and can use it in a "with" statement. Like this:

```
with (myBullet) move_towards_point(objPlayer.x, objPlayer.y, 5);
```

That will start the bullet in the direction where the player is and with the speed 5. Good. Time to set the alarm to go off after some time again, to create another bullet. Add this line to the code:

```
alarm[0] = 60 + random(60);
```

That may seem a bit strange, but what we have done here is to add the value 60 and a random value between 0 and 59.99999. Thus the time for the next bullet to be created will be somewhere between 2 and 4 seconds (60 and 119.99999 frames). Here I do not care so much about the decimals and such, because we will not be able to notice them in the game.

The code in the script "Enemy1Fire" should now look like this:

```
myBullet = instance_create(x, y, objEnemy1Bullet);  
with (myBullet) move_towards_point(objPlayer.x, objPlayer.y, 5);  
alarm[0] = 60 + random(60);
```

What is left to do now is to add the script to the "alarm 0" event of the object "objEnemy1". Do that now. You should know how by now. We also need to make sure that the script is run the first time. This is done by setting the alarm in the script that is run at the creation of the enemy instance. That script is called "Enemy1Init". Open up the script and add the following line to it:

```
alarm[0] = random(60);
```

Here we do not care to add the "60" in front of the random function. I just did not feel like it. Do it if you want, but then no enemy plane will fire until they have existed for 2 seconds. Now they fire the first time between 0 and 2 seconds after their creation.

The bullet should, just like the player's bullets, disappear once it has moved outside the screen. In the "objEnemy1Bullet" object, open the "Outside" event and add a "Destroy the instance" from the "Object" tab of the action panel. Accept the default settings and click OK.

Finally, the bullet should do some damage to the player. Create a new script and call it "PlayerBullet1Collision". Here we should destroy the bullet instance as well as lower the energy of the player. We also need to check if the player energy is 0 or less, and then destroy the player. This means that the code should look about the same as in the "PlayerEnemy1Collision" script. Write down this code in the new script:

```
// Destroy the bullet
with (other) instance_destroy();
// Lower the player's energy
myEnergy -= global.bullet1Damage;
// Check if the energy is zero or less.
if (myEnergy <= 0)
{
    instance_destroy();
}
```

Notice that I have use the global variable "global.bullet1Damage" to lower the energy of the player. This variable must be defined. That is done through adding it to the "GameStart" script. Add this line to that script:

```
global.bullet1Damage = 10;
```

This means that when the enemy bullet hits the player, the energy will be lowered by 10 units.

Add the "PlayerBullet1Collision" script to the "objPlayer" object, in the collision event with the "objEnemy1Bullet" object.

Compare the two scripts "PlayerBullet1Collision" and "PlayerEnemy1Collision". The last "if" statement and the statement it contains look the same on the two scripts. The first line, too, look the same, but we will ignore it for now.

Since we use the same block of code in more than one place, it is a good idea to put it in its own script. So, create a new script and call it "CheckPlayerEnergy". Copy the entire "if" statement from the "PlayerBullet1Collision" script to the new script, so it looks like this:

```
// Check if the energy is zero or less.
if (myEnergy <= 0)
{
    instance_destroy();
}
```

Then we delete the "if" statements from the other two scripts, and instead call on this new script. Just to show how a script can be called from another script. Instead of the "if" statements in those two scripts, it should look like this:

```
CheckPlayerEnergy();
```

There. That is how easy it is to use a script from another script.

To be on the safe side, here is a list of how the entire "PlayerBullet1Collision" script should look like:

```
// Destroy the bullet
with (other) instance_destroy();
// Lower the player's energy
myEnergy -= global.bullet1Damage;
// Check the player energy.
CheckPlayerEnergy();
```

And the "PlayerEnemy1Collision" script:

```
with (other) instance_destroy();
myEnergy -= global.enemy1Damage;
CheckPlayerEnergy();
```

Save the game and try it. When the enemy planes shoot at you, and you are hit, the energy will decrease until the plane is destroyed. Good. But wait! Once the player plane is destroyed you get an error message saying "Unknown variable or function" and referring to the "move_towards_point" function when the enemy bullet is created.

The reason for this is that we made the bullets go for the player plane, and once the plane is destroyed, the bullets no longer know where to go. Or, more programmatically speaking, we are referring to an instance that no longer exists. So, we will have to check if it exists before firing the bullet.

Open up the script "Enemy1Fire". This is where the enemy bullet is created. Now we need to test if an instance of the "objPlayer" is available. That can be done with the "instance_exists" function. Here is its definition:

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

So, we need to use the "if" statement and the "instance_exists" function in the script. Make the script look like this:

```
if (instance_exists(objPlayer)) then
{
    myBullet = instance_create(x, y, objEnemy1Bullet);
    with (myBullet) move_towards_point(objPlayer.x, objPlayer.y, 5);
}
alarm[0] = 60 + random(60);
```

Note that the "alarm[0]" setting is not included inside the "if" statement. That is because we want the script to activate a new alarm event even if the player plane does not exist at the moment.

Nice. This will only create the bullet and move it if the player plane exists in the game.

7.4 Meaning of life

To round up this programming guide, I think it would be a good idea to add some life to the game, as well as some scoring.

Open up the script "CreatePlayer". This is where the player instance is created by the "objPlayerController". Add the line:

```
playerLives = 3;
```

This means that the number of lives that the player has is stored in the instance of the "objPlayerController" object.

Now we need to decrease the number of lives every time the player explodes. That is done in the "CheckPlayerEnergy" script. Open it up.

Inside the "if" statement here, we want to lower the number of lives of the "objPlayerController". We also want to make sure that the player is created again after some time. For that, we will use the alarm function of the "objPlayerController". So, inside the "if" statement, before the "instance_destroy" function, add these lines:

```
objPlayerController.playerLives -= 1;
objPlayerController.alarm[0] = 60;
```

That will decrease the number of lives with 1, and set the alarm 0 event of the "objPlayerController" to trigger after 60 frames (2 seconds).

So, now the entire "CheckPlayerEnergy" script should look like this:

```
if (myEnergy <= 0)
{
    objPlayerController.playerLives -= 1;
    objPlayerController.alarm[0] = 60;
    instance_destroy();
}
```

Time to create a new script that takes care of the creation of the new player, if there are any lives left. Call the script "LivesCheck". Here we should check if the player has any lives left, and if so, create a new instance of the player object. Enter this code:

```
if (playerLives > 0)
{
    myPlayer = instance_create(screen_width / 2, 400, objPlayer);
}
else
{
    game_end();
}
```

Here I have introduced the "else" statement. It can be used after an "if" statement. It works so that if the expression in the "if" statement (playerLives > 0 in our case) is *not* true, the statements inside the "else" statement are executed. In this example this would mean that if playerLives is *not* greater than 0, the game will end. That is what the "game_end()" function is for.

Now we need to add this new script to the "Alarm 0" event of the "objPlayerController" object. Do that.

Save and test the game. You should now have three lives to use before the game ends.

It would be nice if the lives could be displayed for the user in some way.

To do that, open up the script "DrawEnergyBar". This is where the energy bar is drawn. How about drawing a small version of the player's plane sprite next to the energy bar for each life?

Select the sprite called "sprPlayer" and right-click on its name in the tree structure on the left of Game Maker's main window. Select "Duplicate" from the pop-up menu. This will create a duplicate of the player sprite. Open up the new sprite and call it "sprLife". Click the "Edit Sprite" button. Select the menu "Transform -> Stretch". Enter "50%" in the "Width" and "Height" boxes. Select "Excellent" in the Quality selector. Click OK. Click OK again. Now we have a small version of the player sprite that can be good to use as a life sprite. We need to change the "origin" of the sprite though, so that it is centered. Go to the Advanced tab of the sprite window and set the "Origin" to X: 13 and Y: 19. Then click OK to close this window.

Go back to the "DrawEnergyBar" script. To draw a sprite in the screen, we will use the function "draw_sprite". Here is its definition:

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

We will use the sprite "sprLife" and its first subimage, which has number "0". So the function will be used as (the coordinates are not decided yet):

```
draw_sprite(sprLife, 0, someXCoordinate, someYCoordinate);
```

We will also use a new language statement, the "for" statement in order to draw the correct number of life sprites. Please have a look at the definition of the "for" statement in section 24.11, page 74 of the Game Maker Manual.

"For" loops are used to repeat a bunch of statements. In programming lingo this is called "iterating". An example of a "for" statement could be this:

```
number = 8;
for (i = 0; i < 5; i += 1)
{
    number += 1;
}
```

What this program will do is first to set the variable "number" to 8. Then it will enter the "for" loop, which adds the value "1" to the variable "number" a couple of times. But how many? The result is that the variable number will be "13" when this is done. That means that the "for" loop has been run 5 times.

The first time the "for" loop is run, the variable "i" is set to 0. Then the expression in the middle of the "for" statement parenthesis is checked ($i < 5$). If this is true, the statement inside the "for" loop is executed ($number += 1$). Then, finally the last statement of the "for" loop parenthesis ($i += 1$) is executed, which will make the variable "i" now hold the value 1. Once again the middle expression is checked ($i < 5$). It is still true, and the two following statements are executed again. This happens again until the variable "i" becomes "5". Then the expression " $i < 5$ " is not true anymore, and the "for" loop exits. Good.

What we want to do is to draw the same number of sprites as the value of the "playerLives" variable. This could be done through adding the following code to the end of the "DrawEnergyBar" script:

```
for (i = 0; i < playerLives; i += 1)
{
    draw_sprite(sprLife, 0, 140 + 30 * i, screen_height - 25);
}
```

Wow, that was a lot at the same time. First, the "for" loop will be executed as many times as the value of the "playerLives" variable, right? The first time the "for" loop is executed, the variable "i" will be 0. That means that the x coordinate of the first sprite that is drawn will be $140 + 30 * 0$, which is 140. The next time the "for" loop is executed, "i" will be 1. Thus, the x coordinate for the second sprite will be $140 + 30 * 1$, which is 170. Finally, the last time the "for" loop is executed, "i" will be 2, which makes the x coordinate of the last sprite $140 + 30 * 2 = 200$. After that, "i" will be 3, and the "for" loop is exited. The y coordinate is the same all the time, 25 pixels from the bottom of the screen.

This means that there will be three sprites drawn next to each other down the bottom left of the screen, next to the energy bar. When the player loses one life, the variable "playerLives" becomes 2, and the "for" loop will only be run through 2 times, thus only drawing 2 sprites. Great, huh?

7.5 Scoring

I first intended to include the scoring part in the last section, but decided to put it here, in its own section. Of course we need to add some scores to the game. Otherwise it would be uninteresting to shoot down the enemies.

There is a built-in variable called "score" that we will use for this. That makes the score automatically show up in the Window's caption.

First, we need to set the score to 0 when the game starts. Open the script called "GameStart". Add the line

```
score = 0;
```

to that script.

We also need to decide how many scores the player should get for destroying one enemy. I think we should give the player 100 scores for it. Add the following line to the "GameStart" script:

```
global.enemy1Score = 100;
```

Good. Now we need to add that score to the "score" variable whenever the player has destroyed an enemy. This happens in two scripts. First, open up the script called "PlayerEnemy1Collision". Add the following line to the end of that script:

```
score += global.enemy1Score;
```

Then, open the script called "BulletEnemyCollision". Here it is a little bit trickier, since we have to add the score addition inside the "if" statement here. Just to make sure no mistakes are made, I will print the entire script here as it will look after the addition of the "score += global.enemy1Score":

```
with (other)
{
    // Lower enemy energy
    myEnergy -= global.bulletToEnemy1Damage;
    // Check if enemy energy is 0 or less
    if (myEnergy <= 0)
    {
        // If so, destroy enemy.
        instance_destroy();
        // NEW!!! Add score to the player
        score += global.enemy1Score;
    }
}
// Destroy this bullet
instance_destroy();
```

That would be it. Save the game and try it out.

8 Final words

8.1 *End of this guide*

This represents the end of this guide on programming Game Maker 4.3. I hope you have enjoyed reading it and following the instructions, but mostly I hope that you have learned something from it. It is possible to make really great games with Game Maker, and I am looking forward to see the games you create.

If I find the time and inspiration I will make a follow-up to this guide where I will delve into a little bit more advanced programming. Hopefully I will also be able to express a few topics that were requested for this guide, for example a good explanation of variable arrays. This, I hope, will at least serve as a good entrance to the wonderful world of programming.

8.2 *Communication*

If you have any questions at all, or just feel the need to read what other Game Maker users write you are always welcome to the official Game Maker Community. The web address is:

<http://gmcommunity.com/forums/>

My current homepage, which does not look too beautiful but contains a few goodies, can be found at:

<http://hem.passagen.se/birchdale/carl>

Please send me an email if you find any error or such in this document and I will try to correct it. My mail address can be found below.

8.3 *Useless statistics*

WOW! According to Word, I have spent 897 minutes on this document, writing 96859 characters and 17523 words. That is 107.98 characters per minute. And I have saved it 189 times. ☺

8.4 *Bye*

Finally I want to, again, thank all of you who have helped me with this document and other Game Maker questions.

Remember this supposedly Norwegian proverb when making games:

NOTHING IS IMPOSSIBLE EXCEPT SKIING THROUGH A REVOLVING DOOR

Regards

Carl Gustafsson

Karlskrona, Sweden

carl.gustafsson@home.se