Lecture 6 - More GML

Written by Carl Gustafsson, based on my own Beginner's Guide

Goal of the lecture

The reader should, after this lecture, know about the more advanced parts of the Game Maker Language (GML).

This lecture was revised for round 2 of the Game Maker programming course at www.gameuniv.net. Changes to the original document are shown with slightly greenish background. If you read this document for the first time, just ignore those markings and read it as if nothing was marked.

This lecture was also revised for round 3 of the Game Maker programming course at www.gameuniv.net. Changes to the previous revision of the document are shown with slightly blueish background. People reading this document for the first time could ignore the different background colors. Most screenshots are revised, but the change is very little, so they are not marked.

Introduction

This lecture builds upon the previous lecture. The shoot'em up game is further developed with enemies and such. So, load up the gmd file that was created during lecture 5.

Again, I would like to repeat that since this build upon a Beginner's Guide I wrote some time ago, we have already done parts of it. You can look upon those parts as a repetition on previous lectures.

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.

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** that can be found in the resources folder. It looks like this:

Create a new sprite, call it **sprEnemy1** and load this image into it. 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 any 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 *properties* 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 particular 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 (but not necessarily) 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 it is good to have it 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 resources 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 instance that calls the script move downward across the screen with a speed of 10 pixels per step.

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 the section **The Game Maker Language (GML) -> Game play -> Instanecs** of the Game Maker help file 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** steps. That is about 1 second in "real time" provided that the room speed is 30 steps per second. 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 the section mentioned above in the Game Maker help file for more information about this function.

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

You might have noticed by now that some of the words that you write in the code editor changes color to blue or violet. That is because those are either reserved words (built-in functions and constants) or entity names (sprite names, object names, etc). Very good for verifying that you have spelled that object name correctly.

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 Room** (under **Other** in the event buttons). 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**.

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 help file, section **The Game Maker Language -> Computing things -> Real-valued functions**. The random function could be used to make things happen randomly, for example simulating a die. It returns a random value that is always less than the specified value. Like this:

```
random(3);
```

The above line will return values ranging from 0 to 2.9999999999999999999. 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 die, we would use:

```
myDie = random(6);
```

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

```
myDie = floor(myDie);
```

As the argument to the floor() function, we use the variable "myDie" 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.

```
myDie = myDie + 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:

```
myDie = 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 room size? No problem. In Game Maker there is a built-in variable that can be used to determine the size of the room. It is called **room_width**. So, instead of using **640**, we will use **room_width**. (see help file, section **The Game Maker Language -> Game play -> Rooms**.) So, now we can change the first line of the script to read:

```
instance_create(floor(random(room_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. (Actually, positions and times are OK with decimals too, but I wanted to teach the **floor** function ;))
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!

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.

I know there is a built-in **health** feature in Game Maker, but just for learning it, we make our own health feature. 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();
}
```

or, said with more human-readable words:

*Destroy the **other** instance.*
*Decrease the variable **myEnergy** by **30**.*
*Check if the variable **myEnergy** is less than or equal to **0**.*
(Start of block)
*If it is, **destroy myself***
(End of block)

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, that is what this tutorial is about 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 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.

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();
```

Now, one could always argue that this script should be in the enemy object instead (since it mostly deals with the enemy), and you can do what you like in your games. For now, let us do it in this way. :)

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, just as with the **if** statement. 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, the "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.

Note that the same variable name, **myEnergy**, is used for both the enemy and the player energy. That is because normal variables like these are so called "local" variables. Local variables are unique for each instance in the game. So, each enemy and the player have their own **myEnergy** variable, totally independent of all the others' variables.

Pyrotechnics

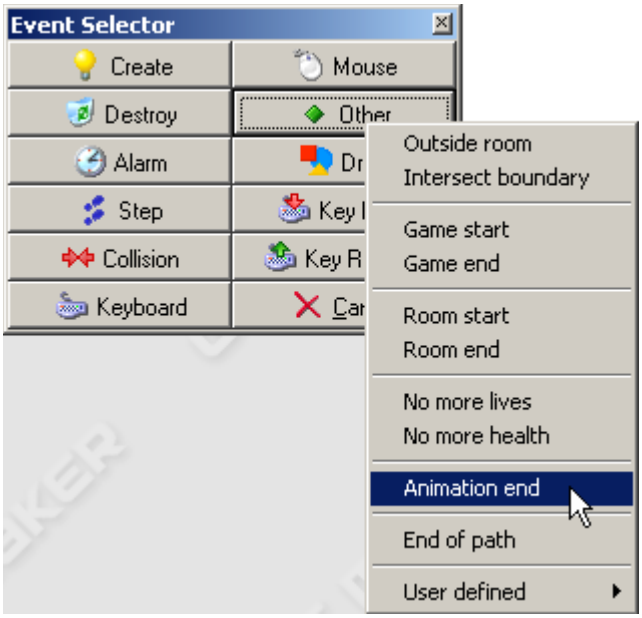
Explosions! They are what are missing from our game. Fortunately Game Maker comes bundled with a nice explosion animation. We will create an explosion object from it.

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 **Explosion.gif**. It should be located in the resources 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. You should now see an animating explosion that looks pretty nice.

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** contain 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.

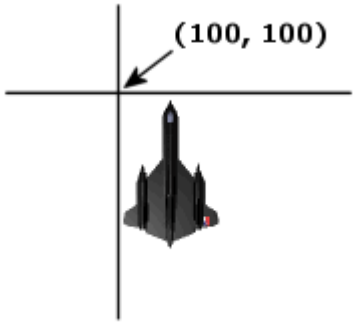
Better add the explosion to the **DESTROY** event of the **objPlayer** too. You should be able to do that on your own now. I will not tell you how to do ;). Use the same **objExplosion** object as for the enemy.

Enhancing the game

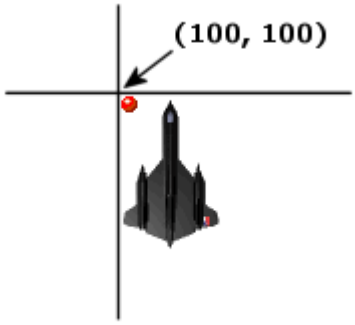
There are some parts of the game that need enhancement to look good. For example the bullet does not come out at the middle of the **objPlayer**, and the enemies sort of "pop up" on the screen instead of flying into it. Time to fix that.

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**.
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:

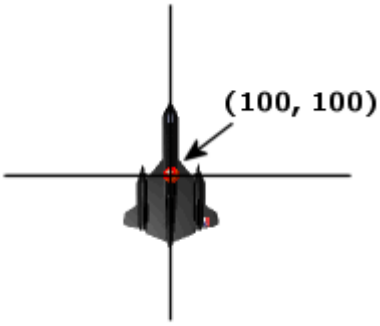


That is because the bullet sprite is much smaller than the plane sprite.

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.

Look at the width and height of the sprite. Note that the width of the sprite is 52, and the height is 78. Now look at the "origin" values 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). In the latest version of Game Maker there is even a button that does this for you. Just click the **Center** button. If you look at the image of the sprite in the right part of the sprite window, you will see two lines that are crossed through the center of the image. They represent the "origin" 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-coordinate. 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.

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(room_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 > room_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.

To do 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 <= room_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.

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 ever-present 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 **The Game Maker Language (GML) -> GML Language Overview -> Addressing variables in other instances** of the Game Maker Help File.

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.

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 use 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 **SatelliteDish.gif** from the resources.

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.

Add an instance of the new object to the room. This object will be used to create the **objPlayer** instance. Therefore the **objPlayer** 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(room_width / 2, 400, objPlayer);
```

That will create an instance of the **objPlayer** object. The x coordinate will be the center of the room (room_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**.

Actually, Game Maker includes a feature to draw an energy bar quite easily, but I want to show you how to do it manually, just for the sake of learning new things.

In the **DRAW** 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:

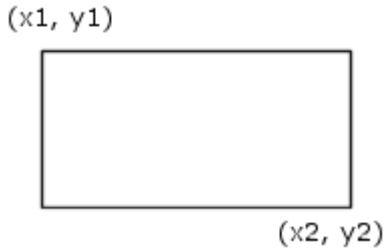
First we draw the gray background rectangle, and then we draw the blue energy rectangle on top of it. The blue frame around the rectangle above is just the background from the room showing. Don't bother with that.

To draw shapes in Game Maker we first need to set a **color**. Enter this lines into the script:

```
draw_set_color(make_color_rgb(150, 150, 150));
```

The **make_color_rgb** 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_rgb(255, 0, 0)** would create a very red color, and **make_color_rgb(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.

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

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. The last value in the **draw_rectangle** function determines if the rectangle should be an outline (true) or a filled rectangle (false).

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:

```
draw_set_color(make_color_rgb(0, 0, 255));
x1 = 7;
y1 = room_height - 13;
x2 = 7 + myPlayer.myEnergy;
y2 = room_height - 7;
draw_rectangle(x1, y1, x2, y2, false);
```

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:

```
draw_set_color(make_color_rgb(150, 150, 150));
x1 = 5;
y1 = room_height - 15;
x2 = 110;
y2 = room_height - 5;
draw_rectangle(x1, y1, x2, y2, false);

if (instance_exists(myPlayer))
{
    draw_set_color(make_color_rgb(0, 0, 255));
    x1 = 7;
    y1 = room_height - 13;
    x2 = 7 + myPlayer.myEnergy;
    y2 = room_height - 7;
    draw_rectangle(x1, y1, x2, y2, false);
}
```

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) and that it is added to the room.

Life, the Universe and Everything

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

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 of the blue 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 **Backgrounds** 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.

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, look like they move past slowly, while objects that are close to the car, like rails, road markings and moose, 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 **bgrStars2**, 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.

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, it would be the enemy plane that would move towards the player, and not the bullet that would. To fix 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 b 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, add the **Outside room** event and add a **Destroy the instance** from the **Main1** 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, looks 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 (including the code inside the curly braces) 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 from the manual:

> ***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.

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.

There **are** built-in lives and score functionalities that are useful especially for displaying score and lives, but, again, for the sake of learning, we will look on how to make our own score and lives display. After all, we used the built-in functions in lessons 3 and 4. :)

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). Since we know that there always will be exactly one instance of the **objPlayerController** object in the game, it is OK to reference it using its **object name**, like above. This will be like referencing its single instance. Take great care when referencing other objects like this if they have multiple instances.

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(room_width / 2, 400, objPlayer);
}
else
{
    game_end();
}
```

Here I have introduced the **else** statement. It can be used after an **if** statement, just like the **ELSE** action that we have used earlier. 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 it 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 Resource Explorer 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. 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(sprite, subimg, x, y)*** *Draws subimage subimg (-1 = current) of the sprite with index sprite 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 the Game Maker Help File.

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 through **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 then 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, room_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 room.

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 twice, thus only drawing 2 sprites. Great, huh?

Scoring

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.

Conclusion

This represents the end of the sixth lecture. I hope you have found your way through it. :) Some of the parts in these GML lectures are a bit strange, since the problem might be easier solved with drag-and-drop actions, but I wanted you show you ways to solve it with the Game Maker Language too, just so that you know how it is done. Then you can decide yourself which is the best way.

Actually, we will continue on this space game during the next lecture too, if it goes as I have planned. We should then look a bit at the other entities in Game Maker - Time Lines and Paths.

Good luck!

Carl

Assignments

Add sounds to the shoot'em up game

Due date: Wednesday, 31 August 2005, 08:00 AM (50 days 17 hours early)
Maximum grade: 100

In this assignment, you should add sounds to the game that was made in lecture 6.

The sounds should be added using the **sound_play** command in a piece of code.

Try to find some nice sounds on the internet (or in your own sounds collection ;)). One good place to start looking is: http://www.a1freesoundeffects.com/ .

The following events should have sounds:

- Player shooting a bullet

- Enemy shooting a bullet

- Bullet hitting enemy (without killing)

- Bullet hitting player (without killing)

- Player exploding

- Enemy exploding

As always, zip the -gmd file and upload it here for grading.

Good luck!

Regards
Carl

Read the GM Manual

Due date: Wednesday, 31 August 2005, 08:00 AM (50 days 17 hours early)
Maximum grade: 0

Hi!

I want you now to read the following parts of the GM manual (or help file):

All these sections are under the main section "The Game Maker Language (GML)"

- Game Play
- User Interaction
- Game Graphics
- Sound and music

Regards
Carl