Lecture 5 - Introducing GML

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

Goal of the lecture

The lecture is an introduction to GML - Game Maker Language. After the lecture, the reader should understand what a script is, and how to make use of scripts and code in Game Maker.

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 will introduce you to the Game Maker Language. It is based upon the document "Beginner's Guide to Game Maker 4.3 Programming" that I wrote some time ago. Since I did not want to adapt it too much when making it into this lecture, the first parts of it will serve as a repetition on previous lectures in this course.

GML will be introduced while we build ourselves a little shoot'em-up game. I will perhaps use code even in places where it would have been easier with drag-and-drop, just to show you how it can be done. When using Game Maker later on, you should think about if things are easier solved with drag-and-drop than with code. Usually, if an event has more than, say, 10 actions, it would probably be better to use a script, since that gives a better overview (at least it does for me). Another place where scripts are very handy is where lots of objects or events have the same action sequence. Then you could reuse a single script in all those places. And, if you discover that you need to change something in all those events, you only need to change it in one place - the script. Neat, huh? :)

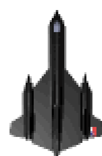
A little notice: If you feel like using something else instead of the images and sounds that are in the resources for this lecture, feel very free to do so! :) The only thing I want you to consider is that large images and animations creates large file sizes, so for the assignments I would prefer if you do not add TOO much heavy graphics. Though, of course, it is always nice to see what you can come up with! :)

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).

Some sprites

In order to see anything in our game we are going to need some sprites. 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.gif**. It can be found in the images folder of resources file that follows this lesson. The image looks like this:

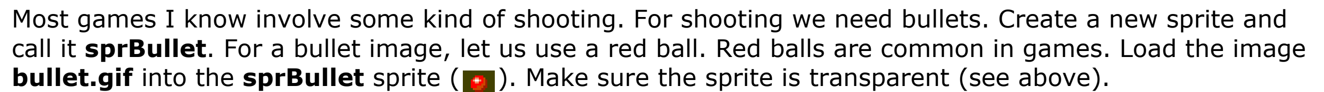


Ah, yes! The SR-71 Blackbird is my absolute favorite plane! Add a new sprite. In the name box, write **sprPlayer**. I always use the prefix **spr** in the names of my sprites, since if a sprite has the same name as an object, errors may occur. So, I consider it a good habit to have a naming convention 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 a variable name is referring to a sprite or not. 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 resources for this lecture until you find the **SR71.gif** 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 **bullet.gif** into the **sprBullet** sprite (). Make sure the sprite is transparent (see above).

That is all the sprites we will need for now.

Create Objects

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 perform some actions are the 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.

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** (NB! no space characters). The rooms can be named on the **settings** tab. 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 **Backgrounds** 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 blue (like sky) color. Now your room should be all blue. The room size (**settings** tab) 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 silly.

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 the middle of the room. This should place an **objPlayer** instance where you clicked. If you happened to click more than once, more than one **objPlayer** instance might have been added to 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 the green "OK" checkmark to close the room window.

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 blue 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).

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.

Refining the actions

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

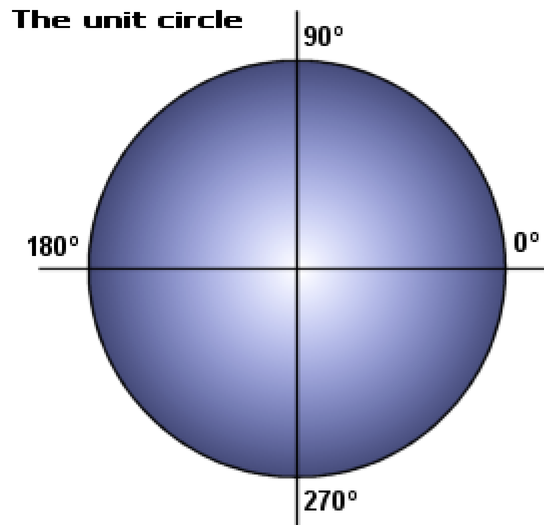
Open the object window for **objPlayer**. Add the event **<No key>**, which is also found among the other **Keyboard** events. This event will happen when all keys on the keyboard are released. Add a **Start moving in a direction** action and select the square in the middle of **Directions**. Click **OK**. 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 **main1** tab to display the actions that have something to do with objects, sounds and rooms. 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 **main1** 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.

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.

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 **Control** tab in the actions list. The two actions I use the most here are **Execute a script** and **Execute a piece of code** from

the **Code** section of this tab. 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 sections **Variables** and **Assignments** in the **The Game Maker Language (GML) / GML Language overview** chapter 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 **speed** 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 (or "step"), 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 "." (period) for people. A period marks 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.

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 100, 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.

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** define the speed vector in rectangular coordinates, while **direction** and **speed** define 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.

First script

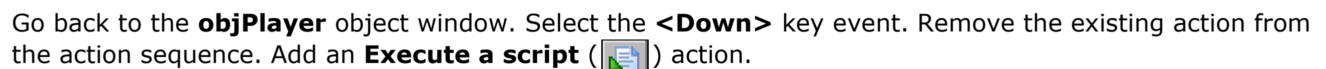
This time you are going to learn how a freestanding script works. Create a new script (**Menu: Add -> Script**).

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.

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.

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**, which 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 then **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 (unless we turn off this error checking, which I do not recommend).

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.

Please take care when writing the name of the variable. Game Maker is "Case-sensitive", so if you write e.g. "GunReady", it is **not** the same as "gunReady".

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 before the code line. You do whatever you like the best.

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 in the empty code window:

```
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 help file, under the section **The Game Maker Language -> Game Play -> Instances**.

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 12 alarm clocks (**alarm[0]** – **alarm[11]**). 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 lecture. 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 to, 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 entire 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!

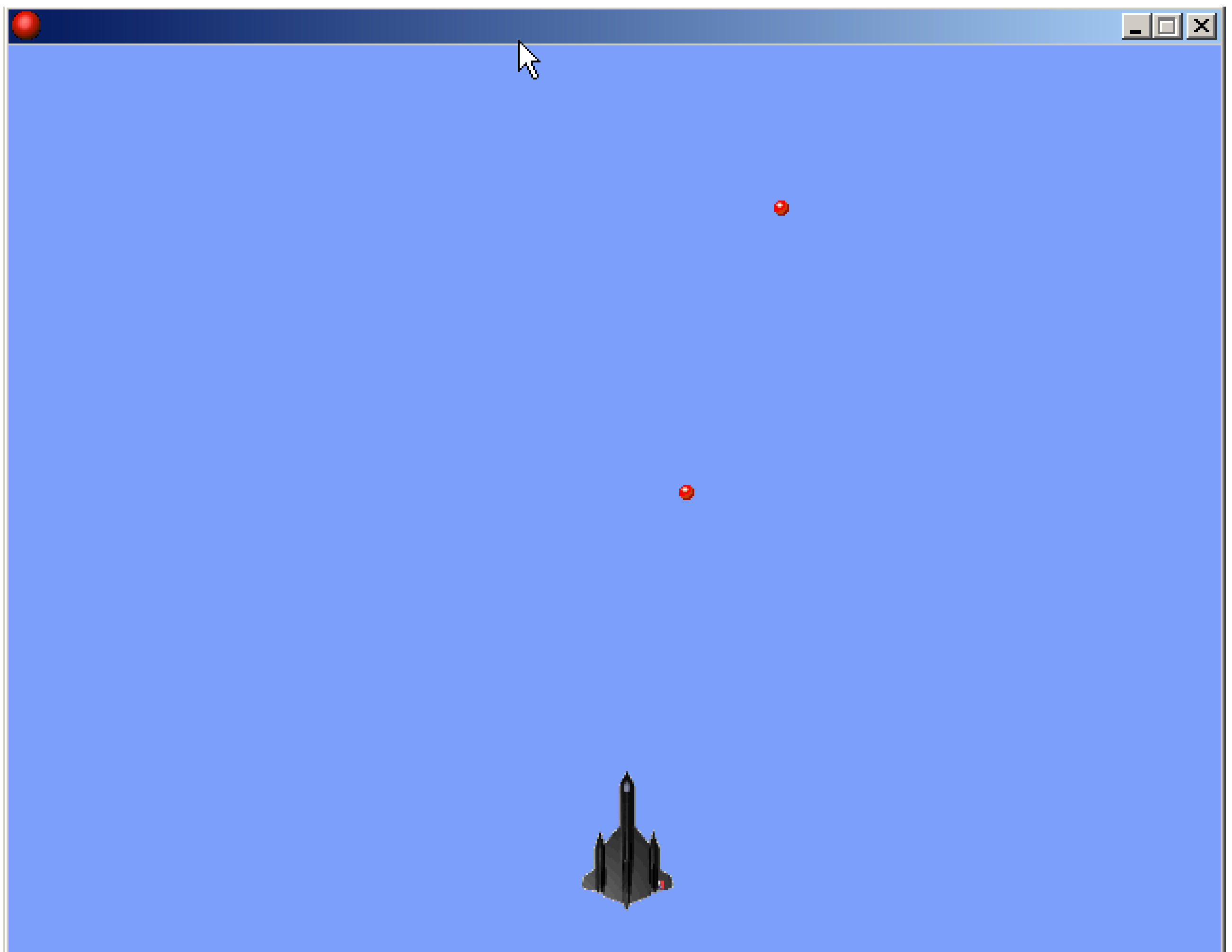
One thing remains; the **alarm[0]** event. Close this code window and add the event for **alarm[0]**. Add an **Execute a piece of code** action and write the following code in the window:

```
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.



Hello there, coder!

Well, now you can code GML! :) Though, the examples in this lecture might be a bit strange, since it is easier to use drag-and-drop for most of them. I wanted, however, to introduce GML and compare it to drag-and-drop in the beginning.

In the next lecture we will dive deeper into GML. Good luck!

Carl

Assignments

Read the GM Manual

Due date: Thursday, 11 August 2005, 08:00 AM (30 days 20 hours early)
Maximum grade: 0

Hi!

Since there was never any programming assignment for this lesson, I want you to read the following parts of th GM Manual (or help file):

- Scripts (Under "Advanced use")
- The entire section "Finishing your game"
- GML Language Overview
and
- Computing things

Good luck!

Carl