Game Maker Language

By Trollsplatterer

0 Abstract

Hello there. This tutorial is meant to be an introduction to GML programming in Game Maker. Game Maker is a program to create games. It is event driven, which means that each step of the game, a sequence of events is triggered (i.e. player presses keyboard or mouse...).
Each event can execute commands (i.e. move objects). These commands are programmable. GM has two different ways of programming these commands:

- D&D actions: simple Drag and Drop items, providing fill-ins to simplify programming
- GML code: pure programming language

This text explains the basics of GML programming. Do not expect to become a novice programmer, using only this text, but after reading it, you should be able to expand your knowledge, using the GM help-file and other peoples examples.

1 GML vs D&D

“What's the advantage of GML over the drag and drop interface?”, one may ask. You can do everything you want with D&D, can't you?

I'll show you some advantages of GML.

1.1 Copy-paste

You can copy D&D actions within the same game-file, but you can't copy them to a different file. You have to merge the games to get the actions of an object from a different file. GML is pure text, so you can easily copy it from one file to another.

A second reason why you might want to copy code, is when you want to show your code to someone else, i.e. when you want some help. You can easily copy-paste the GML code and other people can easily modify it to tell you what you're doing wrong. If you want to show your D&D-code, you'll have a lot more trouble trying to show your code and people will have a hard time trying to answer clearly. Uploading your source-file (.gmk) is not a safe way either...

1.2 Clarity

Some people may object here, stating that GML is a lot more complex than D&D, so it can never result in clearer source code. I'm not talking about a sequence of two or three actions, here though.

If you ever had to add more than 20 actions to a single event, you'll know that the D&D structure tends to loose its clarity. Imagine what it'll look like if you require one hundred actions. The only way to add some structure is to use Start Block and Stop Block actions. You can also add some comments, but only single lines in separate actions.

In GML, you can use indentation, spacing and multiline comments to clarify the code. You can also add comments in the same line as the code. This results in a more readable code, so it's easier for yourself to find certain parts and for other people to understand what you're trying to do. Remember that you may not be able to remeber everything about your code yourself, so commenting is quite useful.

1.3 Speed

As you get to know GML, you'll be able to proceed faster than you could ever be, using the D&D interface. D&D requires a constant interaction of the mouse and the keyboard: drag an action, fill the fill-ins, click OK... With GML, you only have to use one action for every event: the code action. After selecting this action, you only need your keyboard.

1.4 More options

D&D is limited. Some GM functionalities require GML, such as 3D and multiplayer/online games. Some D&D actions are unavailable in GM lite, but can be achieved using code (i.e. setting the cursor).

GML contains a great number of functions to calculate mathematical matters or to check for collision at points or within regions, for example. You can force these in the D&D actions, using the Check Variable action, or the Test Expression action, but if you're capable of that, you should be

capable of using GML too, it's only a small step.

Many functions have the same result as a D&D action, but most provide more options (arguments) and return information on whether the action succeeded.

1.5 Insight

If you use a D&D action, you never really know what's happening behind the scenes. If you use the Move Fixed action, you set the movement of the object. But what does that mean? Which variables are influenced?

GML encourages you to use the variables directly. If you want to stop the movement, you set the variable speed to 0. You can also choose to use functions with the same results as the D&D actions, if you don't know how to make it happen yourself, using variables.

1.6 Reusability

If you know GML, you can write scripts. Scripts are functions of your own. Suppose you regularly need to use the same code, you can write it in a script and execute this script in every event where you need this code.

This requires less work and it's easier to modify it later on. The code is located at a single place, so you don't have to change it in every event.

1.7 Debugging

Large chunks of code have a high chance of containing flaws and it's hard to find these without any help. You can use the debug mode to check the state of the variables in every instance, but this only shows the states in between every step. It is impossible to check the value of a variable at a certain location withing a certain event.

In GML, you can use messages to show the value of a variable at a precise location in the code. You can also comment certain parts of code to check where the errors occur.

1.8 Other programming languages

Last but not least, knowing GML is useful if you're planning on learning some other programming language later on.

GML is a mixture of many different programming languages. Once you know your way in GML, the gate is open for you to try and learn some other language, such as C, C++, C#, Java, Visual Basic, PHP...

GML is a fun way of learning something you may well be capable of using later on. Computers will keep getting a stronger grip on our everyday life, so there's no harm in learning how to handle them.

2 Introduction to Game Maker

To understand this help-file, you'll have to know the basics of the way GM works. I'll use keywords you have to understand in order to know what I'm talking about. The file contains a small index at the bottom, so if you don't know a certain word, it may be explained in the index. A more thourough explanation may be found in the GM help file or using google.

If you already know everything about objects, instances and variables you can skip this chapter.

2.1 Steps

A game is executed in steps. In between each step, GM recalculates everything and at the end of the step, the screen is drawn.

The period of one step depends on the speed of the room. The room_speed is expressed in “steps per second”, so if you set the room speed to 30 (default value), there are 30 steps in every second.

All other speeds (object speed, animation speed) are relative to the room speed, so if you give an object a speed of “2”, it'll move at 60 pixels per second (2pixels x 30steps per second).

2.2 Events

GM is event driven. That means that everything that happens, triggers an event. Objects can react to these events.
Some examples of events:
- Room Start: the first step in the current room
- Collision: the current instance hits the sprite of a different object
- Keyboard Press F: the player pressed the “F” button just now
- Keyboard F: the player is pressing the “F” button

The sequence of these events is this:
- Begin step events
- Alarm events
- Keyboard events
- Mouse events
- Step events
- (now all instances are set to their new positions)
- Collision events
- End step events
- Drawing events
- (now the screen is drawn)

(source: GM help file)

2.3 Resources

Resources are the different elements of the game:
- sprites (images and animations)
- sounds
- scripts
- objects
- ...

2.4 Actions

If an instance senses an event, it can execute a set of actions. Actions influence the behaviour of the current instance, or a different instance. Actions can cause movement, changement of sprite, creation of new instances...

2.5 Objects

Objects are the definition of an element of the game. They can define walls, characters, water, invisible control elements...

2.6 Instances

When you create a character-object, we call the character an *instance* of this character-object. So an object can have multiple instances in the game (i.e. walls: only one object, but you use many walls in your rooms). In fact, when your game is running, there are no objects in the game, only instances.

2.7 Variables

These are little memory-cells you can use to store information. They have a name and a value. An example is the *health*-variable. It contains your current health.
The yoyogames site contains a WIKI about variables:
http://wiki.yoyogames.com/index.php/Variables

3 Using GML code

3.1 Executing code

If you want to run code at a certain event, you have to add an action that executes GML code. GM has two different ways of adding code.

3.1.1 Code Action

The code action is located in the control tab of the actions. Select it and drag it into the actions-field. This will open a window.

In this window, you can enter code. The code will be executed when the code-action is reached during the game. You can mix D&D actions with code actions.

3.1.2 Scripts

Scripts are resources that contain reusable code. With scripts, you can create your own functions, using GML code. Examples of existing functions are “point_get_distance()” and “game_save()”. You could make a function “destroy_all_enemies()”.

The advantages over the code action are;

- You can execute the same code in different events and objects, yet the code itself is only located in the script, so you only have to change the script if you want to modify the code, you no longer have to modify every object.
- You can give arguments to the script, to manipulate its behaviour. For example, the function “`game_save("savegame.ts")`” saves the game with the name “savegame.ts”. The argument you use tells the function what name to use for the file. Scripts work exactly the same way.

If you want to run a script, you can use the Execute Script action in the control tab, or you can run the script with code. The syntax for running it using code is this:

```
my_script_name(arg0,arg1,arg2,arg3)
```

or

```
script_execute(my_script_name,arg0,arg1,arg2,arg3)
```

We'll get to see more about scripts once I explained the basics of code.

3.2 Basic programming in GML

What does GML look like? You've certainly seen complicated examples with many lines and strange symbols. I'll keep it simple here, in time you may be able to write complicated code yourself. In this chapter I'll give you a short summary of the possible commands in GML.

Note that the GML compiler (interpretes your code) is very loose about syntax. There are many different ways of writing the same command. Here's an example (don't worry about understanding it yet):

```
if health <= 0 then game_end() else show_message("Argl")
```

is equal to the code

```
if(health <= 0)
{
	game_end();
}
else
{
	show_message("Argl");
}
```

I use my own way of writing code, which is the second example. You may sometimes find code samples other people wrote that have an entirely different structure. That doesn't mean that piece of code is wrong. GML allows this variety in styles and it's up to you which style you choose.

My style has these rules:

- A statement ends with a ";" character.
- Every block of code within another block has an additional indentation (tab).
- If a block of code only contains one statement, the parentheses *may* be omitted.
- Strings (characters) are bounded by double-quotes, unless a double-quote is part of the letters.
- A condition in an if (or likewise) statement is bounded by opening and closing parentheses.
- Functions are run by their name, followed by the arguments between parentheses.

3.2.1 Variables

To store a value in a variable, you use the "="-character. The syntax of this command is:

```
variablename = value;
```

After this command, the variable with the name variablename will contain the value "value".
A second way of storing something in a variable, is using an expression as a value:

```
variablename = expression;
```

An expression can be anything, really. Some examples are:

```
15 * 60
123 * 23 / (13 mod 7) + 12
variablename + 43
-23 * random(variablename)
```

Expressions can contain variables and functions. First the expression is interpreted and the functions are executed, so their results can be used. Then the resulting value is stored in the variable variablename.

A common problem with variables is the creation. GM creates the variable when it is set for the first time. You can't use the value of a variable before it has one.
The mistake usually occurs when the programmer tries this statement:

```
my_variable += 1;  //increase the value of the variable my_variable
```

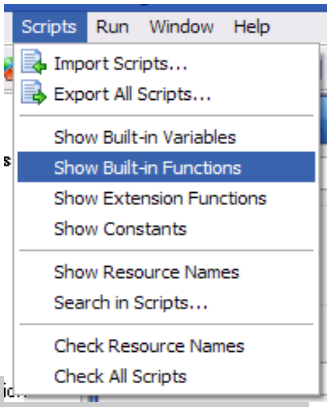
To increase the value of a variable, it has to have a value to begin with. This code is in fact the abbreviation of:

```
my_variable = my_variable + 1;
```

This statement clearly shows that the variable has to have a value before it is executed.

That's why it is a good idea to set every variable to an initial value in the creation event of the object.

3.2.2 Functions



GM contains a huge amount of predefined functions. Some only return a value, some execute commands to influence your game.
It would take me far too long to give a description of every function in GM, but all the functions are described in the Game Maker Help File. A list of all the functions can be retrieved by pressing Scripts => Show Built-in Functions.

Functions have arguments, which decide the result of the functions.
As a first example, we'll use the function `random(arg)`. This function returns a random real value between 0 and the value of *arg*. You can choose the maximum value by entering a value as argument.

```
variablename = random(1);
```

would store the result of the function (a number with the value [0,1[ ) in the variable named variablename.

A second example is the function `motion_set(dir,speed)`. The arguments in this function decide the motion of the instance in which this code is executed.

```
motion_set(direction,0);
```

stops the motion of the instance, but keeps the current direction set.

```
motion_set(0,speed);
```

sets the direction to 0 degrees and keeps the current speed intact.

Arguments can also contain expressions, but these decrease the readibility of the function:

```
motion_set(floor(direction + random(20) - 10),speed + random(3) - 1.5);
```

Modifies the speed and direction relatively with a random value, but it may become unclear which value is the argument of which function, as functions can have up to 16 arguments...

An alternative is using self-defined variables to store the result of the expressions:

```
//define temperary script variables
var new_direction, new_speed;
//remember the result of the expressions in these variables
new_direction = floor(direction + random(20) - 10);
new_speed = speed + random(3) - 1.5;
//use the variables as arguments for the function
motion_set(new_direction,new_speed);
```

The result is a longer code (even if you remove the comments), but it's easier to read and it's easier to see mistakes and make improvements, and as you see, it's easier to add comments.

3.2.3 Comments

Comments are text messages you write to clarify your code. This can be useful to explain your code to others, but it's also helpful if you need to modify your code later on.
You can also disactivate a section of code by commenting it, so you don't have to delete the code, but it isn't executed either.
You can add an initial comment, at the start of the code, in which you explain the purpose of the code and the required arguments.

There are two ways of adding comment:

a. Multiline comments
With multiline comments, everything that's located between the start delimitor `(/*)` and the end delimitor `(*/)` is seen as comment and is not executed during the game.

```
my_variable = 86400; /*add the comments here*/
```
or
```
/*some text
some more text
even more text*/
```
or even
```
my_variable = /*some comments*/ 86400;
```

b. Single line comments
Single line comments start with two slashes (//). The part of the line that's behind these characters is seen as comment.

```
//add comments here
```
or
```
my_variable = 86400; //this is a comment
```

3.2.4 Conditions

Many commands only need to be executed if a certain condition is true. For example, the game has to stop if the health of the main character reaches 0.

In code, you can handle this using the IF expression:

```
if(condition)
{
      code block
}
else
{
      code block
}
```
The condition is evaluated. The resulting value of this expression should be *true* or *false*.
Some examples of conditions are:
`150 > 140` (150 is larger than 140)
`x < 140` (horizontal location is smaller than 140)
`distance_to_object(obj_troll) <= 100` (... is smaller than or equal to 100)
If the expression is true, the code block between the first two brackets is chosen. Otherwise (false) the second block is executed.

The second part is not obligatory, so if you don't want anything to be executed if the condition is false, you can use this structure:

```
if(expression)
{
      code block
}
```

If a code block only contains one single command, the parethesis can be removed:
```
if(expression)
      Single command;
```
Some people find this confusing, but if you use proper indentation it doesn't result in unclear code.

I'll give a short example of a conditional code:
Suppose you would want a character (obj_char) to turn around (turn 180 degrees) and speed up if it gets too close (150 pixels) to a troll (obj_troll):
Code action in the Step event of object obj_char:

```
if(distance_to_object(obj_troll) < 150) //if the troll gets too close
{
      direction = direction + 180;  //rotate 180°
      speed = speed + 2;            //increase speed by 2
}
```

You can use these comparisons in expressions:

<	smaller than
<=	smaller than or equal to
== or =	equal to
!= or <>	not equal to
>	larger than
>=	larger than or equal to

You can also combine two or more comparisons, using the keywords **and** (alternative: &&) and **or** (alternative: ||). Suppose you only want the character to run away from the troll if it is too weak to fight him (local variable HP is smaller than or equal to 15), then the previous code would become:

```
if(distance_to_object(obj_troll) < 150 and HP <= 15)
{
      direction = direction + 180;  //rotate 180°
      speed = speed + 2;            //increase speed by 2
}
```

3.2.5 Loops

Sometimes you may want certain blocks of code to be repeated a number of times. With loops, you can repeat a block of code until a given condition is false.

You can create loops with the while-statement.
Syntax:

```
while(condition)
{
      code block
}
```

Here's a simple example of how to create 20 instances of object obj_troll:

```
//initiate a counter-variable
counter = 1;
//repeat until the counter reaches a value that's higher than 20
while(counter <= 20)
{
      //create an instance of object obj_troll at location (100,100)
      instance_create(100,100,obj_troll);
      //increase the counter by 1
      counter = counter + 1;
}
```

Notice what would happen if you would forget the last statement (`counter = counter + 1;`). The result would be an eternal loop, which would create so many trolls your game would crash.

That's why you have to watch out when you're using loops. Make sure the condition becomes false in time.

There is a special kind of loop, which can be used to repeat a block of code a certain amount of

times, without using a variable to count up or down.
syntax:

```
repeat(number)
{
      code block
}
```

example:

```
//repeat 20 times
repeat(20)
{
      //create an instance of object obj_troll at location (100,100)
      instance_create(100,100,obj_troll);
}
```

3.2.6 Arrays

Variables are the most basic elements of programming, so they are used a lot. The problem is that a variable can only contain one value at a time. If you want to store a list of related values, i.e. the amount of bullets for the different guns, you could make a separate variable for every type of bullet.

The alternative is using an array. An array is a list of variables with the same name, but a different location in memory. Every element of this list has its number.
Syntax:

```
array_name[array_location]
```

This element of an array can be treated like a variable; you can store values in it and you can retrieve values from it.

Example 1: Ammo inventory

Here's how you could create the bullets-array:

```
bullet_list[1] = 100;   //revolver (9mm)
bullet_list[2] = 0;     //shotgun
bullet_list[3] = 0;     //machinegun
bullet_list[4] = 0;     //uzi
bullet_list[5] = 0;     //flame thrower
```

Here's how you could use the shotgun-element of the list to check if you can fire the shotgun:

```
//test if there are still shotgun shells left
if(bullet_list[2] > 0)
{
      //fire the shotgun
      instance_create(x,y,obj_shotgunshell);
      //decrease the amount of shotgun shells
      bullet_list[2] = bullet_list[2] - 1;
}
```

Example 2: Weapon arrays

You could take this one step further and make a list for several values of the weapons; sprites, bullet-object-id's, bullet-amounts, bullet-speeds... Then you could use one variable to choose the current weapon. Changing weapons will now only require changing this single variable. Here's how you could do this for 3 different weapons:

We'll use:
- 1 global variable which contains the currently selected weapon: `global.current_weapon`
- arrays for the character-sprite (with the gun), the bullet-object, the ammo and the fireing-rate

Here's how this is done:

First we define the arrays at the creation of the character-object.
Code action in the Creation event of the main character object:

```
weapon_sprite[1] = spr_9mm;         //sprite for the 9mm-equipped character
weapon_bullet[1] = obj_bullet_9mm;  //object that's fired when you press the LMB
weapon_ammo[1]   = 100;             //current amount of bullets left
weapon_speed[1]  = 2/room_speed;    //2 bullets per second if you keep fireing

weapon_sprite[2] = spr_shotgun;
weapon_bullet[2] = obj_bullet_shotgun;
weapon_ammo[2]   = 0;
weapon_speed[2]  = 1/room_speed;

weapon_sprite[3] = spr_machinegun;
weapon_bullet[3] = obj_bullet_machinegun;
weapon_ammo[3]   = 0;
weapon_speed[3]  = 10/room_speed;
```

You also have to enable the player to choose a different weapon. This should result in a different value of the variable "`current_weapon`" and the selection of the right sprite.
Code action in the event which changes the weapon to machinegun:

```
current_weapon = 3; //select a new weapon
sprite_index = weapon_sprite[current_weapon]; //select the right sprite
```

When the player pulls the trigger, we'll have to create a bullet, unless the player tries to shoot before the next bullet has been loaded (fireing rate...). To do this, we'll use an alarm. Alarms decrease with 1 every step, so if you set an alarm to 30 steps, it'll call its alarm event after 30 steps.
We will use this alarm to re-enable fireing.
Code action in the LMB event:

```
if(fireing_enabled && weapon_ammo[current_weapon] > 0) //enabled & ammo left
{
      //create the bullet-object
      instance_create(x,y,weapon_bullet[current_weapon]);
      //decrease the ammo
      weapon_ammo[current_weapon] = weapon_ammo[current_weapon] - 1;
      //disable fireing until Alarm event 0
      fireing_enabled = false;
      alarm[0] = 1/weapon_speed;
}
```

Code action in the Alarm0 event:

```
fireing_enabled = true;
```

Now we only have to initiate some of the values at the start of the game.
Additional code in the Code action at the creation of the object:

```
...//previous code

current_weapon = 1;     //set the initial weapon to the 9mm
fireing_enabled = true; //enable fireing
sprite_index = weapon_sprite[current_weapon];
```

3.3 Tips and tricks

Now that you have an idea of the basics of GML, you may be interested in some useful tips on how to make your life as a GML programmer easier.
The GML language contains many functions and extra features. I'll discuss some here, but there are certainly more around. The best way of learning them is by reading the GM help-file and checking out other peoples GML code.

3.3.1 Abbreviated assignments

A normal assignment looks like this:

```
left_hand_value = expression;
```

Assignments frequently use the value of the variable within the expression, for example to increase the value of a counter:

```
counter = counter + 1;
```

Especially for these types of assignments, most programming languages allow these special assignment operators:

```
variable += expression; //increase
variable -= expression; //decrease
variable /= expression; //divide
variable *= expression; //multiply
```

To increase the counter you might as well use this code:

```
counter += 1;
```

3.3.2 For loop

The for-loop is a special version of the while-loop. A common use of a loop is this one:

```
//repeat 10 times
counter = 1;
while(counter <= 10) //counter => 1 to 10
{
      ... (statements, which may use the value of counter)
      counter += 1;  //increase the counter
}
```

This structure consists of 3 elements:

- set the initial value to the counter
- set an end-condition
- increase the counter

The for-loop joins these elements in a single command:

```
for(initial value ; end-condition ; increase the counter)
{
      code block
}
```

The example would look like this:

```
for(counter = 1;counter <= 10;counter += 1)
{
      code block
}
```

3.3.3 Break or Continue

Sometimes you may want to change the behaviour of a loop; stop it or skip a part. There are two special keywords for that purpose:

a. break
The break statement stops the execution of a loop and proceeds behind the loop.
It also stops the execution of a with-statement or a switch statement (see later).
If it's used outside one of these statements, it stops the current code-action or script and proceeds with the next action in line.
syntax:

```
break;
```

example:

```
var connection = false, connect_attempts = 0;
while(!connection)
{
      connection = connect_with_my_server(); //try to connect to the server
      if(connect_attempts > 20 && !connection)
      {
            //stop after 20 failures
            show_message("Something's wrong! Couldn't connect to the server!");
            break; //stop the loop
      }
}
```

b. continue
Continue skips the rest of the code-block in a loop and returns to the beginning of the loop. In a for-loop, the counter is also incremented.
syntax:

```
continue;
```

example:

```
var connection = false, connect_attempts = 0, answ;
while(!connection)
{
      connection = connect_with_my_server();
      if(connect_attempts > 10 && !connection)
      {
            answ = show_message_ext("Couldn't connect to server! Try again?",
                                    "Yes, keep trying!", "No, stop it!");
            if(answ == 1)
                  continue;
            else
            {
                  show_message("Connection failed");
                  break; //stop the loop
            }
      }
}
```

3.3.4 Switch

Sometimes you have to test the value of a variable for number of specific keywords or numbers, for example after asking the player a question. One way of doing this would be using several if-statements:

```
//ask the player a question with 3 possible answers
answer = show_message_ext("How are you?","Great!","Sad","Nearly dead!");
//if the answer was "Great!", the value of the variable answer is 1
if(answer == 1)
      show_message("You will no longer be after I said what I have to say!");
//if the answer was "Sad", the value of the variable answer is 2
if(answer == 2)
      show_message("Oh, that'll only get worse after this conversation...");
//if the answer was "Nearly dead", the value of the variable answer is 3
if(answer == 3)
      show_message("Then you are about to die!");
//if the player presses escape, the variable will contain 0
if(answer == 0)
      show_message("OK, ignore me if you won't hear what I have to say!");
```

There is a special statement for this type of checking. The switch-statement allows you to test a variable for several fixed values. Our example would look like this:

```
//ask the player a question with 3 possible answers
answer = show_message_ext("How are you?","Great!","Sad","Nearly dead!");
//check which button was pressed and answer
switch(answer)
{
      //if the value of "answer" is 1...
      case 1:  show_message("I'm happy to hear so. Now give me you money!");
               break; //stop here
      //if the value of "answer" is 2...
      case 2:  show_message("Give me some money, or it'll only get worse!");
               break;
      //if the value of "answer" is 3...
      case 3:  show_message("Die then, so I can take your money!");
               break;
      //if none of the above were true...
      default: show_message("You ignore me? You are about to die!");
}
```

If the value of the variable is equal to the value of the case, the statements starting after that case are executed until the first break-statment or until the end of the switch-structure. If you don't put break-statements, the code in the next cases will also be executed.

3.3.5 Strings

A string is just a line of text, a combination of several characters. Sometimes you may like to show text to the player. This is easy if the line of text is fixed, if the contents are always the same. Sometimes you'll have to change the text, depending on certain conditions, though.
Suppose you want the player to enter his own name and you want this name to be used in messages throughout the game. How will you do this if those texts are fixed?
Here is a list of useful code for strings. You'll also find some string manipulation functions in the GM help file.

a. Combining strings:
Pasting strings together is called concatenation.
syntax:

```
string1 + string2;
```

example:

```
message = "My name is " + global.player_name + ", do NOT call me babe!";
```

b. Converting numbers to text:

syntax:

```
string(number)
```

example:

```
message = "I made it, I only died " + string(global.deathcount) + " times!";
```

c. Using special characters:

These characters have a special meaning within strings:

-> " " and ' ' define the start and the end of a string

-> # causes a line break when the string is displayed

If you want these characters to be displayed you have to use some special tricks:

-> display double quotes: use single quotes to define the start and the end of the string.

m = 'He told us; "There was no way out and the troll entered the room!".'

-> display single quotes: use double quotes to define the start and the end of the string.

m = "There's no way he'd have gotten out if the troll hadn't been stuck in the gate."

-> display both double and single quotes: combine strings using concatenation:

m = 'He said: "' + "There's no way I'll ever try that again!" + '"';

-> display the number sign: use the escape character (backslash) before the character.

m = "The HTML notation for blue is \#0000FF or rgb(0,0,255)."

3.3.6 Alarms

In code, there is no function to start an alarm. If you want an alarm to start counting down, you have to set the right variable in the alarm-array:

syntax:

```
alarm[num] = amount_of_steps;
```

example:

```
alarm[0] = 180;
```

You can also test the current value of an alarm, by treating this element as a variable:

```
//if alarm0 is nearly at its end, speed up
if(alarm[0] == 30)
      speed += 2;
```

If you use this, you have to remember that an alarm only counts down if these conditions are met:

1. The value of the element in the alarm-array is larger than 0.
2. The event of that particular alarm contains at least one action.

If you want to use the alarm without executing an action when the alarm-event is triggered, you can always fill the event with a comment as a dummy action.

An alternative is making your own alarm. This enables you to use more than 12 alarms in one object and it no longer requires the alarm event. This is the exact way an alarm works:

1. uninitialised: `value = -1;` (in the objects create event)
2. initialization:
```
value = entered value;
value = round(value); //take the rounded value
```
3. count:
```
if(value >= 0) //count until the value is negative
   value -= 1; //decrease the value
```
4. trigger alarm:
```
if(value == 0)
{
      ... //code to be executed at the alarm event
}
```

If you want to make your own alarm, you only have to mimic this behaviour, using any variable.

3.3.7 Matrices

I already told you about arrays. A matrix is an array of arrays.
syntax:

```
matrixname[rownumber,columnnumber];
```

You can imagine an array being a table:

mat[0,0]	mat[0,1]	mat[0,2]	mat[0,3]	mat[0,4]	mat[0,5]	mat[0,6]	mat[0,7]	mat[0,8]	mat[0,9]
mat[1,0]	mat[1,1]	mat[1,2]	mat[1,3]	mat[1,4]	mat[1,5]	mat[1,6]	mat[1,7]	mat[1,8]	mat[1,9]
mat[2,0]	mat[2,1]	mat[2,2]	mat[2,3]	mat[2,4]	mat[2,5]	mat[2,6]	mat[2,7]	mat[2,8]	mat[2,9]
mat[3,0]	mat[3,1]	mat[3,2]	mat[3,3]	mat[3,4]	mat[3,5]	mat[3,6]	mat[3,7]	mat[3,8]	mat[3,9]
mat[4,0]	mat[4,1]	mat[4,2]	mat[4,3]	mat[4,4]	mat[4,5]	mat[4,6]	mat[4,7]	mat[4,8]	mat[4,9]

example:
In the second example of the arrays-chapter we saw that you could use several arrays to store the behaviour of the weapons. In this example we used 4 arrays:

```
weapon_sprite[n]
weapon_bullet[n]
weapon_ammo[n]
weapon_speed[n]
```

For this purpose we could use a matrix:

```
weapon_sprite[n]  => weapon[0,n]
weapon_bullet[n]  => weapon[1,n]
weapon_ammo[n]    => weapon[2,n]
weapon_speed[n]   => weapon[3,n]
```

Now we have a single element which contains all the information we need to make the weaponery work. All you have to do is remember well which row contains which type of data.

3.3.8 Manually trigger events

GML contains a function to trigger any event manually. This is useful if you use the same code in different events:
example:

- press ctrl event => fire bullet code
- press LMB event => fire bullet code

A better way of doing this is:

- press ctrl event => fire bullet code
- press LMB event => trigger press ctrl event

The result during the game is the same, but if anything has to be modified to the “fire bullet code”, it only has to be changed in one event. You could also do this using a script.
syntax:

```
event_perform(event_type,event_number);
```

The required keywords for event_type and event_number can be found in the GM help file.
examples:

```
event_perform(ev_mouse,ev_left_press);    //Left Mouse Pressed
event_perform(ev_alarm,3);                //Alarm3
event_perform(ev_keypress,ord("F"));      //Keyboard Pressed F
event_perform(ev_collision,obj_enemy1);   //Collision with object obj_enemy1
```

3.3.9 With statement

The with statement executes code in an instance from within another instance. It enables you to not only set variables, but also execute functions.
syntax:

```
with(object_or_instance_id)
{
      code block
}
```

Here's an example on how to destroy a bullet-object from within your character:

```
var bul;
//check if there's a bullet at the current location
bul = instance_place(x,y,obj_bullet);
if(bul != noone)
{
      //decrease the health with the value of the variable "power" of the bullet
      health -= bul.power;
      //destroy the bullet
      with(bul)
      {
            instance_destroy();
      }
}
```

If the with statement refers to an object, the code is executed in every instance of this object.
example:

```
//speed up all the smarter enemy objects when the alarm goes off
with(obj_enemy_smart1)
{
      speed += 3;
}
```

Note that the with statement doesn't run the code in the children of a parent object. It only executes the code in the instances of that specific object.

3.3.10 Scripts

I mentioned scripts before, but I never told you how to use them. A script is not just a block of code you can reuse. It can be used to make a separate function, with its own arguments and return value. Here's how you make one:

- In the menu, press Resources, Create Script.
- Enter a name for the script.
- Add some comments at the top of the script, in which you discribe the use of the script, its arguments and its return value. This step is not required, though advised.
- Start writing the script.

A script can have up to 16 arguments. Inside the script, you can use them by the name `argumentX`, where the *X* is the number of the argument.
An argument contains a value which is given to the script at the function call.

example:
script name: **speed_up_enemies**
code:

```
/*-------------------------------------
Speed up all the enemies with argument0.
Use negative values to slow them down.
-------------------------------------*/
with(obj_enemy1)
      speed += argument0;
with(obj_enemy2)
      speed += argument0;
with(obj_enemy3)
      speed += argument0;
```

A script can also return a value. You can use this to make your own mathematical functions, to manipulate strings or to tell the calling procedure that something went wrong (by returning *false* or *noone*, for example).
syntax:

```
return value;
```

The value may as well be an expression which returns a value.
Example:
function name: **discr**
code:

```
/*-----------------------------------------
Returns the discriminant of the 3 arguments:
Ax² + Bx + C => return B² - 4AC
-----------------------------------------*/
return (sqr(argument1) - 4 * argument0 * argument2);
```

Once the script has been created, it can be called on, just like an predefined function:
syntax:

```
return_value = function_name(arg0,arg1,...);
```

example

```
/*Solve the problem Ax² + Bx + C = 0, using the discriminant.*/
A = get_integer("Ax² + Bx + C = 0#Enter A:",0);
B = get_integer("Ax² + Bx + C = 0#Enter B:",0);
C = get_integer("Ax² + Bx + C = 0#Enter C:",0);
//call the self-made function
d = discr(1,B,C);
//make use of its return value
if(d < 0)
      show_message("There is no solution (no imaginary numbers).");
else if(d == 0)
      show_message("x = " + string(B / (2 * A)));
else
{
      show_message("x1 = " + string((B - sqrt(d))/(2 * A)));
      show_message("x2 = " + string((B + sqrt(d))/(2 * A)));
}
```

3.3.11 Setting the cursor

The D&D action for setting the cursor-pointer to a different sprite is a pro-feature. You can change the cursor in lite too, though, by simply setting the variable cursor_sprite.

syntax:
```
cursor_sprite = my_sprite_name;
```
Now the sprite *my_sprite_name* will be drawn with its origin at the mouse position.

You can also change the cursor into some windows-like cursors; an hourglass, a text selection bar, an image selection cross...
This can be achieved with this function:
```
window_set_cursor(keyword);
```
The keywords can be retrieved from the GM help file: GML=>Game Graphics=>The Window.
examples:
```
window_set_cursor(cr_hourglass);    //hourglass
window_set_cursor(cr_beam);         //text selection
window_set_cursor(cr_cross);        //image selection
```
You can also check the current cursor with the function window_get_cursor(). This function returns the keyword-code for the current cursor.
example:
```
//set the self-made variable mouse_is_enabled to true if the cursor is visible
if(window_get_cursor() == cr_none)
      mouse_is_enabled = false;     //invisible
else
      mouse_is_enabled = true;      //visible
```
alternative code:
```
//set the variable mouse_is_enabled to the result of the condition
mouse_is_enabled = window_get_cursor() == cr_none;
```

3.3.12 Keyboard and mouse

You already know how to use keyboard and mouse events, but GML enables you to check the state of these user interface elements within other events. For example, you can check if the player is pressing the shift-key in the up-key event, to decide the speed of the instance, without using any of the shift-key events.

a. keyboard
These functions check for the state of the keyboard:
```
keyboard_check(keycode);            //cfr. key event
keyboard_check_pressed(keycode);    //cfr. key pressed event
keyboard_check_released(keycode);   //cfr. key released event
keyboard_check_direct(keycode);     //some additional checks
```

These functions return “true” when the key with that keycode is pressed.
To use these you have to know which keycodes belong to which keys. Keycodes from character keys can be calculated using the ord() function:
```
keyboard_check(ord("A"))
keyboard_check_pressed(ord("4"))
```
Remember that not all keyboards have a querty-layout. The commonly used WASD-buttons are very hard to master on an azerty keyboard. Imagine using the buttons ZQSD on a querty keyboard for walking. To avoid this, you can enable the player to choose his interface (qwerty or azerty), you can let him choose his own buttons or you can avoid the letters A,Z,Q and W.

Special keys have special keywords. These are described in the GM help file, under GML => User Interaction => The Keyboard.
Some examples:
```
keyboard_check(vk_shift);
keyboard_check(vk_up);
keyboard_check(vk_numpad3);
```

Another example:

```
//create a bullet when the control-button is pressed
if(keyboard_check_pressed(vk_control))
      instance_create(x,y,obj_bullet);
```

The function keyboard_check_direct() adds the option to use different actions for the two control, alt or shift buttons. These only have one keyword with the other three keyboard functions, but the direct-function also accepts these keywords:

- vk_lcontrol and vk_rcontrol (left- and right contol button)
- vk_lshift and vk_rshift (left- and right shift button)
- vk_lalt and vk_ralt (left- and right alt button)

It also checks the state of the keyboard during times when the game-window no longer has focus, i.e. when it's minimized or when the player runs a different application.

Except for this, it has the same result as keyboard_check().

b. mouse

These functions check the state of the mouse:

```
mouse_check_button(keycode)
mouse_check_pressed(keycode)
mouse_check_released(keycode)
```

The keywords for them are:

- mb_none
- mb_left
- mb_middle
- mb_right

The position of the mouse within the room is kept in the variables mouse_x and mouse_y. These variables are read-only, so you can't change their values to relocate the mouse-position.
To calculate the movement of the mouse, you have to keep track of the previous position of the mouse:

```
//check the difference with the previous position
mouse_mov_hor = mouse_x - mouse_xprev;
mouse_mov_ver = mouse_y - mouse_yprev;
//remember the current position of the mouse
mouse_xprev = mouse_x;
mouse_yprev = mouse_y;
```

Now you can use the variables mouse_mov_hor and mouse_mov_ver to check how far the user moved the mouse cursor and in which direction he moved it. This has its limitations, though, because the mouse cursor can't move outside the screen. If you would use this to rotate the character in a 3D game, the character would stop rotating once the cursor reaches the border of the screen, even if it's set to invisible.

There is a solution to this problem, there is a way to relocate the cursor.
These functions return the location of the mouse-cursor on the game window:

```
window_mouse_get_x()
window_mouse_get_y()
```

These are the relative positions to the upper-left corner of the window.
There is also a function to change the position of the mouse cursor:

```
window_mouse_set(xpos,ypos);
```

Now we can relocate the mouse cursor after checking its movement:

```
//allow to disable this feature, in case you want to use the mouse
//during the game (window_set_cursor() changes the cursor or clears it)
if(window_get_cursor() == cr_none) //check if the cursor is visible
{
      //check the difference with the previous position (set fixed to (100,100))
      mouse_mov_hor = mouse_x - 100;
      mouse_mov_ver = mouse_y - 100;
      //reset to position (100,100)
      window_mouse_set(100,100);
}
```

3.3.13 Other GML requireing features

Some features of GM require GML code to work. For 3D games, you have to use code. The functions for 3D have no D&D equivalent. The same goes for multiplayer games.
Changing resources is also very limited in D&D; you can only change sprites, sounds and backgrounds, you can't create them or remove them. You can do all that in GML.
But these features would require a separate tutorial. The YoYoGames website contains various tutorials to help you with some of these features, here are the links:

- Multiplayer tutorial
- 3D tutorial

These also require pro, though.

4. What now?

Now you know the basics of programming GML. You should be able to understand other peoples code and make your own games using GML.

Remember that the GM help-file is your best friend while you're programming. Always check it out before you bother someone else, it is a lot faster, using the index and the search tabs at the upper-left corner.

If you have any questions about programming, you can always ask them at the GMC forum. This is a forum, full of experienced people who are willing to help you out with any GM related problem. Just make sure you ask your question at the right subforum and have some patience.

If you have any questions or suggestions about this tutorial, please send me an e-mail at info@trollsplatterer.be. I will try to answer as soon as possible, usually within 24 hours.

Good luck with your future games!

Peter Van Dyck (alias Trollsplatterer)
info@trollsplatterer.be
http://www.trollsplatterer.be
Belgium
March 31, 2008

Index

action	Instruction to GM.
argument	Values you can give to a function to manipulate its behaviour.
array	Sequence of variables which use a common name and an array-index.
compiler	Software that reads the code and turns it into computer language.
condition	An expression which results in a value that is either true or false.
expression	Sequence of commands that result in a value.
help file	Press F1 or Help => Contents in GM to get help about GM.
false	Negative result of an expression (failure), its real value is negative or 0.
increment	Increase a value with 1.
indentation	Tabs before the lines of code, used to give structure to the code.
instance	Element in the game.
LMB	Left Mouse Button.
object	Definition of an element in the game.
read-only	Can only be read from, cannot be written to.
room speed	Amount of steps per second that are executed in the current room.
script	Code that can be called using a script action or by running it with code.
statement	Sequence of commands, i.e. an expression, function or variable assignment.
step	One cycle of the game.
string	Sequence of characters, forming a word or phrase.
syntax	Structure of the characters and words in the code.
true	Positive result of an expression (success), its real value is positive (default:1).
variable	Memory, both predefined and for your own purpose.