In this tutorial we will explore how to use scripts to perform advanced sequences of actions in Game Maker. This tutorial is uses the scripting language built into Game Maker but introduces concepts and terms that are applicable for a wide range of programming languages.

We will use Temple of Locks V 4.0 to demonstrate the concepts in this tutorial.

CREATING A SCRIPT

Scripts are chunks of code used in Game Maker to perform complex actions not possible with the standard actions provided. Scripts work just like other resources; they are created and lie dormant until executed (just like actions) by the game.

To create a new script, click the button at the top. This will put in a new script under the Scripts folder. Opening the script will bring up a simple text editor that will allow you to enter in code. Open the new script and change its name to “script_controller”:

To actually use the script we need to use an Execute Script (in the control tab) inside of an action. Here we will remove all the code from the Step event in “object_controller” and replace it with a single Execute Script action with Script = “script_controller”. The list of arguments can be left blank for now; they are used to pass in values for functions and will be covered later. Now we should have the controller executing our script at every step.
The scripting editor has several useful tools built in. It can let you search for a word, search and replace words and jump to a given line of code ( , respectively). The most helpful tool will be the Syntax Checker ( ). The scripts have to be written in a VERY exact way (called the Syntax) with things coming in proper order. If there is a problem with the order or separation of the commands, there will be an error when executing. The Syntax Checker will catch these errors before the game starts and tell you exactly where they are in a message on the bottom of the window. Get in a habit of always hitting the Syntax Checker when you are done editing the code.

If you just need to execute a small piece of code one time you can also use Execute Code ( , in control tab) to insert some script directly into an event. This is good if you just need to check some variables or call a function.

VARIABLES

As we covered in earlier chapters, variables are at the heart of getting things done in game maker. In scripting we deal with them directly and they become even more important.

In order to access variables while scripting we just have to type their name, just like in any other action (being very about which object we are “in”). But in scripting we will want to assign values to variables as well. Look at the following statement in “script_controller”:

```
object_controller.timer = object_controller.timer - 1
```

This line contains 3 elements, first the `object_controller.timer =` which tells Game Maker to assign variable “timer” in “object_controller” whatever value comes after the `=`. Second, `object_controller.timer - 1` is a calculation that will find the value to be assigned to the variable on the other side of the `=`. Finally, and very importantly is the “;”. All scripting commands in Game Maker must be followed by a “;”.

This how the computer knows the last command is done and that it should get ready for a new one. This is an essential part of the Syntax and the Syntax Checker in the script editor will catch it if you forget it.

This command has the exact same effect as the Set Variable ( ) function set to assign “timer” value = -1 with Relative turned on. Everything to the right of the equals sign gets calculated first so there is no problem with the value of “timer” changing before the calculation is complete.
You will also notice that “object_controler” is a different color than the rest of the statement. This is because the script editor will color certain things to make it easier to follow the code and spot errors. If all the objects are colored purple and you type in an object that doesn’t turn purple, there is a good chance you typed in the name wrong.

We may also want to make some temporary new variables in our script; we define them at the top of our script. Here we will define two variables, “msg” and “time_left”:

```
var msg, time_left;
```

The command `var` is followed by the names of the variables separated by a “,” and ending with a “;”. These variables are temporary! They will cease to exist inside of Game Maker as soon as the script is finished. These variables are good for holding information that is needed only for a little while and need to be filled with values.

Variables can also hold text, or as they are more commonly called Strings (like a string of characters). Often we want to hold text in variables and then add to it over time. Here we assign our new variables with values:

```
msg = "Hello World!";
time_left = 1000 - object_controler.timer;
```

Strings are defined using the quotation marks in the code, anything within the quotation marks is considered to be a string. Strings cannot be used like numbers for mathematical operations and even though they may have numbers in them, they have no value. The string “32” is not the same as a variable holding the number 32.
While any variable can hold either a number or a string, but it is up to the coder to remember which is which. Take the situation where we try “msg = msg + 1;” we will get the error:

This is because Game Maker doesn’t know what adding means when it involves a string and a number.

CONDITIONALS

A big advantage of scripting is that it can make conditionals a lot simpler to do. While there are a lot of conditionals actions in the standard interface, all the conditionals in scripting are built with the `if` statement.

Here is an example `if` statement that restarts the game if the timer reaches 0:

```plaintext
if (object_controller.timer == 0) {
  game_restart();
}
```

Every `if` statement starts with a `if` (all lowercase), followed by the question that controls the conditional encased in “()”. The question is the core of the `if` statement, if it is true then everything inside the “{}” brackets will be executed. The “{}” brackets work just like the start and end block actions ( ) and should be used on all `if` statements. You do not need to put a “;” after the closing bracket (“}”). “game_restart()” is a function that does the same thing as the `Restart Game` ( ) action.

The question you ask right after `if` is where all the work of the conditional is. We have “object_controller.timer == 0”, this checks if the value of the variable “timer” in the object “object_controller” is 0. Notice that here we use the double equal sign (“==”), this is because the single equal sign (“=”) is used for assigning variables while the cole is used for comparing values. If “timer” holds the value 0 then the statement is true and the code inside of the “{}” executes.
There are a variety of comparison operators like “==” that we can use to build questions:

- “<” and “<=”: “less than” and “less than or equal to”.
  - *Ex.* (object_controller.timer <= 0) would fire if it is 0 or below
- “>” and “>=”: “greater than” and “less
  - *Ex.* (0 >= object_controller.timer) would fire if it is 0 or below
- “!”: Not equal to
  - *Ex.* (object_controller.timer != 0) fires whenever timer is NOT 0

With these, we can perform any of the comparisons that we could make with Test Variable (VAR). When using either of the “… or equal to” comparisons make sure you put them in the right order - it has to be “<=”, NOT “=>”.

Just like the Else (ELSE) action in the standard interface, we can perform else on if statements as well. For example, if we want to count down the timer if the timer is NOT equal to 0:

```java
if (object_controller.timer == 0) {
    game_restart();
}
else {
    object_controller.timer = object_controller.timer - 1;
}
```

You put the else right after the closing “)” of if and give it another set of “{“ for its commands.

We can also chain together questions into a single question by using the following modifiers:

- “& &” : “And” - fires if both the comparisons are true
  - *Ex.* (object_controller.timer <1000 & & object_controller.timer > 0) - fires if “timer” is less than 1000 AND greater than 0.
- “| |”: “Or” - fires if EITHER comparison is true
  - *Ex.* (object_controller.timer == 1 | | object_controller.timer == 0) - fires if “timer” is equal to 1 OR equal to 0.

Chaining together expressions can make things a lot easier than doing them with actions because you can just use one conditional. Let’s say we want to restart the game if the player object was falls off the bottom of the screen. We would need to ask 3 questions:

- Is object_player.x > 0?
- Is object_player.x < room_width?
- Is object_player.y > room_height?
With the standard interface, it would look like this:

```
if (object_player.x > 0 && object_player.x < room_width && object_player.y > room_height) {
  game_restart();
}
```

**FUNCTIONS**

*Functions* are collections of code with assigned names. When code is actually executed variables are replaced with the values that they are holding; similarly, when the program is run functions are replaced by the code they represent. We have already run into several functions in our previous games, like “random()” and “choose()”. We have just seen another one, “game_restart()” which allows us to restart the game.

Functions get colored a dark blue in the script editor and are always followed by “()”. The purpose of those “()” brackets is so that you can pass in *parameters* to the function. Take the “choose” function:

```
object_player.weapon = choose(1,2,3,5,7,11);
```

This has the numbers 1,2,3,5,7,11 as *parameters* and are used by the function to make a choice. The “choose” function also returns a value, so that at the end of its code it is replaced by the choice it made. Some functions, like “game_restart()”, don’t return values and simply perform some function in the game.

Almost all the actions that we have been dealing with in the standard interface have equivalent functions that can be used in the scripting language, for example:

- **Next Room** = “room_goto_next()”
- **Play Sound** = “play_sound(index)” - index is the name of the sound to play
- **Destroy at Position** = “position_destroy(x,y)” - X and Y coordinates to destroy
Game Maker has a large collection of functions for lots of different uses, all described in the Help file. Finding the right function you want can be a bit tricky so look around.

Combining variables, conditionals and functions we can recreate all the functionality for the “object_controller” step event for Temple of Locks (plus a half-way message):

```plaintext
var msg, time_left;
| msg = “Half Way Done!”;
| time_left = 1000 - object_controller.timer;

//Only do timer stuff if the player exists
if (instance_exists(object_player)) {

    //If the room is halfway done, display a message
    if (object_controller.timer == time_left) {
        show_message(msg);
    }

    //Decrease the timer
    object_controller.timer = object_controller.timer - 1;

    //If there is no time left, change the player into a dead player
    if (object_controller.timer == 0) {
        with (object_player) {
            instance_change(object_player_dead, true);
        }
    }
}
```

The segments in green that start with “//” are comments - these are notes to let anyone looking at the code follow what is going on. They have no effect on the code, but are always a good idea.

You will also see two new concepts. First is the `with` command near the bottom, this is the same as the “Applies to” box in the action details. “with (object_player)” is the same as having the action apply to “object_player”. The “{}” brackets mark off the limit of the “Applies to”.

The other new concept is `true`. This is a special word and is used for answering yes or no questions, like the second parameter of the “instance_change” function (Perform actions when changing?). For example, the command “if (true) { ... }” would always execute.

---

**LOOPS**

Up to now we have seen how we can use scripting to replace the actions we used in the standard interface, but the real power of scripting (and programming languages in general) is in performing a large amount of actions using just a few commands. This is done through **looping**.

**Looping** is the process of performing a single event repeatedly for a collection of values or until some condition is met. For example: a loop could look through all the objects on the
screen and pick out all the enemies that are within 100 pixels of the player, or it could delete all the enemies on the screen till there are only 3 left. Looping allows us to deal with large amounts of objects and data quickly and simply.

There are two basic types of looping that we will use in Game Maker. The first is the **while** loop which is designed to loop while some condition is met. These loops work a lot like if conditionals that repeat when the condition is true and terminate when it is not.

For example: If we wanted the <ctrl> button in Temple of Locks to delete not just the box ahead of us at a certain distance in front of us, but the first box in front of us at any distance we could modify the Key press <ctrl> event inside the player with a **Execute Code** replacing the destroy event:

Inside the **Execute Code** we use the following **while** loop:

```plaintext
// x position to search
var new_x;
new_x = x;

// Loop while there is NOT collision with brick at (new_x,y)
// and new_x is inside the room
while (!collision_point(new_x,y,object_brick,false,true) && new_x > 0 && new_x < room_width) {
    // Advance new_x by 5 pix
    new_x = new_x + (5 * LeftRight);
}

// If the collision happened, remove the brick
if (collision_point(new_x,y,object_brick,false,true)) {
    position_destroy(new_x,y);
}
```

This code uses a temporary variable (“new_x”) that it increments by 5 as long as the new X position DOES NOT have a collision with a brick (the “!” at the beginning of a conditional is equivalent to the NOT box in standard conditionals) and is not outside the room. (The parameters for the “collision_point” function are described in the Help file.)

It is important to make sure we test to see if “new_x” is still in the room because if there were no brick to collide with the loop could run forever and freeze the game. Finally we check to see if the loop terminated because there was a collision, and if so we destroy the object at the new position.
All this code is functionally equivalent to:

```plaintext
// A x position to search
var new_x;
new_x = x;

// If the collision happened, remove the brick
if (collision_point(new_x, y, object_brick, false, true)) {
    position_destroy(new_x, y);
}
// advance position of new_x
new_x = new_x + (5 * LeftRight);

if (collision_point(new_x, y, object_brick, false, true)) {
    position_destroy(new_x, y);
} else {
    new_x = new_x + (5 * LeftRight);
}

if (collision_point(new_x, y, object_brick, false, true)) {
    position_destroy(new_x, y);
} else {
    new_x = new_x + (5 * LeftRight);
    ...
}
```

Obviously the first code is preferable to typing out all those conditionals. It also allows us to stop the code right at the moment we need to.

The other type of loop in Game Maker is called a for loop, which is a type of while loop designed for the type of looping we just did. A lot of looping involves counting up to a certain point and performing some operation at each step (like we just did). The following is the equivalent of or previous loop:

```plaintext
// Loop till
for (new_x = x; new_x > 0 && new_x < room_width; new_x = new_x + (5 * LeftRight)) {
    // If the collision happened, remove the brick
    if (collision_point(new_x, y, object_brick, false, true)) {
        position_destroy(new_x, y);
        break;
    }
}
```

The structure of a for loop is “for (exp1; exp2; exp3) []” where:

- exp1: is executed BEFORE the loop starts, this is executed only once and is usually where you define temporary variables (new_x = x).
- exp2: this is the conditional that controls the loop, the loop will continue until this expression is false (new_x > 0 && new_x < room_width).
- exp3: this is executed AFTER every cycle of the loop, this is usually where you increment values (new_x = new_x + (5 * LeftRight)).
All the stuff in the “{}” is the code that gets executed with each loop. In our example it checks to see if there is a collision at the “new_x” point. You will also notice the expression `break`, this command instantly exits out of whatever loop we are currently in and is a quick way of making sure your loop doesn’t go on for too long. In our case, we use `break` because we want to only destroy a single brick.

Loops can make your code simple and efficient, but only if the loop is properly designed. Make sure that when you create the loop that you keep in mind:

1. The condition that keeps the loop going should eventually be false. Endless loops are the quickest way to break your program. It’s often to put in a sort of failsafe condition that will always terminate the loop.
2. The loop should only run as long as necessary. If you are searching through a list looking for one thing, `break` out of the loop when you find it. Don’t make the loop do unnecessary work.
3. Make sure that whatever you want the loop to do will actually be accomplished when the loop finishes. The loop should produce some result that is essential to the execution of the program as a whole.