TorqueScript for Beginners

By: McTwist
February 21, 2018

Contents

1 Introduction

Starting now, every lessons programming language will be taught in TorqueScript. At first this might be daunting, due to the nature that the language is in itself quite verbose. But one should not be afraid of its shenanigans, as that is the true challenge of a programmer: To know its limitations and try to exploit them. All lessons from here-on-out will use the infamous *Appendix A Reference* as a reference [1].

There is always different ways of how code may be written. Both for how many ways a problem could be solved, but also for how a piece of code could visually look to make the reading of it more appealing. One should take note when reading others code that there may be both bad or good ways of doing this. This document will contain the one that is preferred for the writer, but the reader may apply their own standard as they see fit.

2 Old Frontiers

As the name TorqueScript implies, it is a scripting language. The performance might not be so good, but the power is in its versatility to add new code as the program go. In Blockland, for example, the scripts are not only loaded when a program starts up. One could easily load a script while the program is running, directly changing how the program flow. The method that makes it possible to do this is called *parsing*.

2.1 Heads up

There is some caveats that adjusts the previous lesson a bit due to how this language works. Keep in mind that some of these are for most languages.

First, is that a statement should always end with a **semicolon** (;). There is of course exceptions when this is not needed, and that is code blocks from *functions*, *conditions* and *loops*.

The biggest rule when it comes to naming conventions for variables, functions and objects, is that all of those are *case-insensitive*. But worth noting is that a name can only consist of the first letter as an ASCII character, a to z and A to Z, and all the rest characters may also include numbers. There is exceptions, however.

Last thing, whitespaces, as long not within strings, are only used to separate words from one another, but other than that, they are ignored completely.

2.2 Comments

All programs have some way to document itself, and that is handled with *comments*. These are used, and should be used, to comment code to make the flow easier to understand. It also explains how certain algorithms works for future reference and for other that will read it.

```
1 // This is a single line comment
2 // The next line
```

Everything to the right of that comment is ignored by the parser. Everything on the left is parsed normally.

Please note that even if the reference brings up multiline comments, those does not work in Blockland. That certain functionality is lacking or removed, is a common case in Blockland.

2.3 Variables

For the variables, there is two types: Global and local. The global can be accessed anywhere. This means that there can only be one global variable with the same name for a program. The local is bound to their respective functions. Therefore, a local variable cannot be accessed outside of its destined function, but its name may be reused in any other function.

A global variable is defined with its prefix of a **dollar sign** (`$`).

```
1 $global = 4;
```

A local on the other hand is starting with a **percentage** (`%`).

```
1 %local = 4;
```

Despite their name, they are completely separated and therefore does not affect one another. As mentioned previously, the program is case-insensitive, so that means that a variable with same name, but different cases, will be the same variable.

```
1 $GLOBAL = "James";
```

Basically, `$global` is exactly the same as `$GLOBAL`, meaning that when one assign to the second one, the first one get the same value.

Please note that the prefix and the name cannot be separated with a whitespace.

A variable's default value is empty string. So if the variable that is being used have not been set, it will automatically contain an empty string.

2.4 Operators

Here is the first changes of how TorqueScript handles syntax apart from the previous algorithm specific lesson. There is of course the default operators previously mentioned works the same. But there is some additions that needs to be mentioned. Some is even used in other languages. These are either used to ease it for the programmer for often used tasks, or due to some internal jinx in the engine.

First is the incrementer. This effectively replaces the `+1` that were used frequently in previous lesson.

```
1 $var = 1;
2 $var++; // 2
```

Same goes for the decrementer.

```
1 $var--; // 1
```

Keep in mind that this also gives out the new value. In this example `$var2` will have what `$var` is after it is incremented.

```
1 $var2 = $var++; // 2, 2
```

Moving on, there is the **not** (!) operator. This will reverse the boolean value. True becomes false and vice-versa.

```
1 $check = true;
2 $check = !$check;
```

Then comes the modulo, as mentioned in previous lesson. Due to that it is the same character as the prefix of a local variable, one should take great care to not mix them up and make sure that a space is separated if using the modulo.

Moreover, there is the **compound assignment**. This takes the assignment operator and adds a prefix of any of the other operators. That makes it possible to apply an operator to a value on the left side with the value on the right side but still store it in the variable on the left side.

```
1 $var = 1;
2 $var *= 2; // Same as $var = $var * 2;
```

2.4.1 Strings

As we are handling string, there got to be an easy way to concatenate them. In this case, there's the **at-sign** (`@`).

```
1 $name = "James" @ " Doe";
```

Strings are used quite frequently, so there is plenty of ways to concatenate them. There is **space** (`SPC`), **tab** (`TAB`) and **newline** (`NL`), all of which is doing what their names imply.

```
1 $name = "James" SPC "Doe";
```

For the relational operators, they work exactly the same, but there is one difference that for strings, one need to use the **string-equal** (`$=`).

```
1 $name $= "James" SPC "Doe";
```

The reason for this is that the variable's value type is converted. The normal operators converts the value on each side to a number, while the string version converts them to strings.

```
1 $var1 = "James";
2 $var2 = "Jane";
3 $var1 == $var2; // This is true (1 == 1)
4 $var1 $= $var2; // This is false (James == Jane)
```

There will be a further explanation later on about type conversions.

String operators does not work with compound assignment.

2.5 Conditions

These are quite straightforward. Each syntax required a pair of parentheses next to it to work.

```
1 $num = 4;
2 if ($num == 3)
3     $num = 4;
4 else if ($num == 4)
5     $num = 5;
6 else
7     $num = 3;
```

But in the middle of this, what happens if one would like to put more more lines in an `if`? In comes **code-blocks** (`{}`). Most languages uses this to make sure that a certain part of a code is linked to the code preceding it. Like in the previous lesson, `then` and `end` was used to mark this.

```
1 $num = 4;
2 $name = "James";
3 if ($num == 3)
4 {
5     $num = 4;
6     $name = "James";
7 }
8 else if ($num == 4)
9 {
10    $num = 5;
11    $name = "Janet";
12 }
13 else
14 {
15    $num = 3;
16    $name = "Eric";
17 }
```

Code blocks can also be used on their own to mark a block of code as its own. However, TorqueScript does not scope their variables, so code blocks is generally only used for a preceding programming syntax, like `if`.

There is one last syntax that is really valuable to use whenever one find out that many static values need to be checked. This is **switch-case-default**. It redirects to the correct value. If one gives it a 3, it will check for the `case` that contains that value and execute that code block. If none is found, it will run the `default` code block.

```
1 $num = 4;
2 $name = "James";
3 switch ($num)
4 {
5 case 3:
6     $num = 4;
7     $name = "James";
8 case 4:
9     $num = 5;
10    $name = "Janet";
11 default:
12    $num = 3;
13    $name = "Eric";
14 }
```

As can tell, the code is more compact and one understands that `$num` is used throughout the code. Even better, now one may easily add more cases without having to think about duplicate code or mistyping.

There is also the ternary operator, but that is left as an exercise for the reader to understand.

2.5.1 Strings

Once again, there is some special handling for strings when it comes to switch-case-default. Instead of `switch`, they make use of `switch$`.

```
1 $num = -1;
2 $name = "James";
3 switch$ ($name)
4 {
5 case "James":
6     $num = 4;
7 case "Janet":
8     $num = 5;
9 default:
10    $num = 0;
11 }
```

2.6 Loops

For the loops, this is almost the same. `while` works like so.

```
1 $i = 1;
2 while ($i <= 5)
3 {
4     echo($i);
5     $i++;
6 }
```

For the `for`, however, there is a slight difference which is applied to most programming languages.

```
1 for ($i = 1; $i <= 5; $i++)
2 {
3     echo($i);
4 }
```

Do not panic. It could be read like so: For `$i` assigned 1, while `$i` less equal 5, increase `$i` with 1. Basically, this loop will print 1 to 5. It works exactly like the previous loop, but the main difference is that it is more compact and, for most people, easier to read.

In addition, there is two keywords that could be used with loops: `break` and `continue`. Those are left to the reader as an exercise.

2.7 Functions

This is exactly like in the previous lesson, but with the addition of local variables.

```
1 function factorial(%n)
2 {
3     if (%n <= 1)
4         return %n;
5     return factorial(%n - 1) * %n;
6 }
```

As one can see, this looks really easy to do, if one understood the previous lesson.

Calling them is the same way as before.

```
1 $var = 4;
2 $var = factorial($var);
```

2.8 Tables or arrays

For the variables, there is also this way to make arrays. That is, a way to make iterable variables. To make it clear, there is no real arrays in TorqueScript, but it has this delusion that there is. It is actually a way to dynamically modify the variable name with other variables. One assigns a value to an array like so.

```
1 $name[1] = "James";
```

This introduces the **array** (`[]`). This could be applied to any variable, as long as it is placed after the name. What it does is that it creates a new variable name with the parameter.

```
1 $name1 = "James";
```

Those are exactly the same. Moreover, one could add more parameters to the array.

```
1 $person[1, "name"] = "James";
```

This is essentially like so.

```
1 $person1_name = "James";
```

This might look really odd, but that is how it really works. One may apply as many parameters as they wish, and it will concatenate them with an underscore. This way, one could apply it for a loop.

```
1 for ($i = 0; $i <= 5; $i++)
2     $var[$i] = $i * $i;
```

As mentioned in a previous section, this is where the exception of naming convention comes in: Due to that one may now use a string, which could include almost any character, means that a variable could have other characters than the previous mentioned ones. However, one could still only access the variable through the means of arrays as the previous naming convention does only not apply within arrays.

Keep in mind that arrays could only be used as long as: there is a variable; there is at least one character at the beginning. The arrays can applied to any variable, as long as these requirements are followed.

2.9 Objects

For objects, this starts pretty easy. One create an object like before, but are now restricted to the already defined ones.

```
1 $obj = new ScriptObject();
```

This will create a new object called `ScriptObject` and assign it to the variable `$obj`. To access its variables, one does like previous mentioned.

```
1 $obj.name = "James";
```

Heads up for the missing `$` for the second variable. This is because the **dot** (`.`) will tell that it should access an object from the left side to get the variable name on the right side. This is then assigned a string.

2.9.1 Methods

There is of course more object types. To mention a few that will be used now: `SimObject`, `SimGroup`, `SimSet`, `ScriptGroup`. These are used differently depending on what one want to do. Blockland is using them extensively to make handling of other objects easier.

The previous used type `ScriptObject`, is just a normal type which is used as the name implies: An object handling script data. Be aware that it does not handle script itself, but the variables that one have defined within it.

Then there is `ScriptGroup`. This is where **methods** come into play. They are basically functions, but is called through an object. In this case, a `ScriptObject` will be created and stored within a `ScriptGroup` so it will keep track of it for us.

```
1 $obj = new ScriptObject();
2 $group = new ScriptObject();
3 $group.add($obj);
```

Internally, the value of `$obj` is put into a list that then can be accessed like so.

```
1 $group.getCount(); // Amount of objects in group
2 $group.getObject(0); // Get first object from group
```

Keep in mind that computers always counts from 0.

How the methods are created is taken up in next lesson.

As an object are created, it takes up room in the memory. To solve this, one should always remove objects that is not used any more by calling its `delete` method.

```
1 $group.delete();
```

All group objects(`SimGroup` and `ScriptGroup`) have the ability that whenever they are deleted, they will also delete all objects inside them. Also, whenever an object is added to a group, it is removed from its previous group.

Remember to always clear the variable with its default value of empty string.

```
1 $obj = "";
2 $group = "";
```

More about these kind of things and why they need to be done, will be weeded out extensively in a future lesson.

3 Assignment

Copy-paste this piece of code in the console of Blockland to get all functions that exists that moment. Read and try to understand some of them, especially those that are after the first block of functions.

```
1 dumpConsoleFunctions();
```

Then type in this to get all available objects and their methods. This is only optional and could be skipped, but left here to make it easier to follow on the next lesson.

```
1 dumpConsoleClasses();
```

The last thing to try out is to first create an object of any type, and then call the method of dump on it. This will give all variables and available methods for that specific object.

```
1 $obj = new ScriptObject();
2 $obj.name = "James";
3 $obj.dump();
```

References

[1] GarageGames. *TorqueScript, Appendix A - Quick References.* Hall Of Worlds, http://mirror.aposoc.net/Blockland/References/Appendix%20A%20-%20Quick%20References.pdf. Last updated 2006.