SWEndor Scripting Guide – Writing Scripts

Version: v0.1
Created: 29 December 2019

CONTENTS

INTRODUCTION

Script files are used by the game to define programmable logic.
They have a custom syntax that is interpreted by the game engine to generate behaviour. This way you can define your own scenario without needing to modify the game program.

SCRIPT ORDER

The game, when loading a custom scenario, will load script files in the order defined in its Scenario file.

It will then execute the following functions in order:

- The script whose name is defined under **`Fn_loadfaction`**. If a script by this name is not found, this step is skipped.
- The script whose name is defined under **`Fn_loadscene`**. If a script by this name is not found, this step is skipped.
- The script whose name is defined under **`Fn_load`**. If a script by this name is not found, this step is skipped.

When attempting to generate a player (at initialization or after player death), it will execute the following:

- The script whose name is defined under **`Fn_makeplayer`**. If a script by this name is not found, this step is skipped.

Every game tick, the game will execute the following:

- The scripts whose names are defined under **`Fns_gametick`**.

SYNTAX

Script files take a pseudo-C style language syntax.

A script may look a bit like this:

Example

```
float3 faction_empire_color = { 0, 0.8, 0 };
int tiea1;

loadfaction:
	Faction.Add("Empire", faction_empire_color);

load:
	// spawns the player
	string tiea1_type = GetPlayerActorType();
	float3 tiea1_pos = {100, 0, 10000};
	float3 tiea1_rot = {0, -180, 0};
	tiea1 = Actor.Spawn(tiea1_type, "Alpha-1", "", "", 0, "Empire", tiea1_pos, tiea1_rot);
	Actor.SetProperty(tiea1, " Health.MaxShd", 25);
	Actor.QueueLast(tiea1, "wait", 2.5);
	Actor.AddToSquad(Player.GetActor(), tiea1);
```

The script file begins with global statements, until a script header is encountered.

```
// global statements
float3 faction_empire_color = { 0, 0.8, 0 };
int tiea1;
```

The script header contains the name of the script, followed by the colon (':') sign.

```
loadfaction:
```

The script header is followed by script body which consists of a number of statements, until either another header or end of file is encountered.

```
	Faction.Add("Empire", faction_empire_color);
```

LITERALS

A literal is a single unit. It usually represents the smallest semantic unit of expression.

Examples of literals		
Boolean	Represented by (true|false)	`true`
Decimal integer	Represented by a series of digits. May be prefixed with a negative sign.	**256** **-12**
Hexadecimal integer	Represented by the prefix '0x' followed by a series of digits	**0x12** **0x0100**
Floating-point	Represented by a decimal format with a decimal point	**0.025** **10.0**
String	Represented by quoted strings	`"example"` `"this is a string"`
Variable	Represented by any unquoted string beginning with a character beginning with alphabet (A-Z, a-Z) or an underscore (_).	`actor_hp` `_x`

KEYWORDS

Some words are reserved as keywords and cannot be used as variable names.

Reserved keywords				
if	**else**	**then**	**for**	**foreach**
in				

EXPRESSIONS

An expression usually comprises one or more literal instances, joined by some operator or function block.

Examples of expressions
`{ 0, 1, 2 }`
`2 + 1`
`"Alpha-" + i`
`actor_hp > 10`

TYPES

Some words are reserved as types and are used for variable declarations. They cannot be used as variable names.

Supported Types		
`bool`	Boolean type. Represents a Boolean value	`true` `false`

`int`	Integer type. Represents an integral number. Supports both decimal and hexadecimal format	`256`
		`-12`
		`0x12`
		`0x0100`

`float`	Floating-point type. Represents a floating-point.	`0.025` `10.0`
`string`	String type. Represents a string.	`"example"` `"this is a string"`
`bool[]`	Array type for bool.	`{true, false, true}`
`int[]`	Array type for int.	`{0, -1, 1}`
`float[]`	Array type for float.	`{9.26, 1.35, 4.78}`
`string[]`	Array type for string.	`{"br", "ok", "en"}`
`float2`	Vector type for two floats. Equivalent to a float[] or int[] with two members	`{1, 2.5}` `{5, -4}`
`float3`	Vector type for three floats. Equivalent to a float[] or int[] with three members	`{0.1, -0.2, 0.3}` `{5, 7, 13}`
`float4`	Vector type for four floats. Equivalent to a float[] or int[] with four members	`{1, 2, 3, 4}` `{0.01, 0.25, 6.25, 31.25}`

STATEMENTS

Each statement is terminated by the semicolon (';') literal.

```
	string tiea1_type = GetPlayerActorType();
	float3 tiea1_pos = {100, 0, 10000};
	float3 tiea1_rot = {0, -180, 0};
```

A statement may span multiple times, but must be ended by the semicolon (';') literal.

```
	tiea1 = Actor.Spawn(tiea1_type,
			"Alpha-1",
			"",
			"",
			0,
			"Empire",
			tiea1_pos,
			tiea1_rot);
```

A statement usually comprises one or more expression phrases, which is a basic block of operations.

Examples of statements

```
float3 faction_empire_color = { 0, 0.8, 0 };

loadfaction:

load:
	// spawns the player
	string tiea1_type = GetPlayerActorType();
	float3 tiea1_pos = {100, 0, 10000};
	float3 tiea1_rot = {0, -180, 0};
	tiea1 = Actor.Spawn(tiea1_type, "Alpha-1", "", "", 0, "Empire", tiea1_pos, tiea1_rot);
	Actor.SetProperty(tiea1, "Health.MaxShd", 25);
	Actor.QueueLast(tiea1, "wait", 2.5);
	Actor.AddToSquad(Player.GetActor(), tiea1);
```

```
bool enabled;
```

```
Faction.Add("Empire", faction_empire_color);
```

```
float[] actor_ids = {12, 24, 360};
```

```
j += 1;
```

OPERATORS

Unary operators

Unary operators are applied to a single operand. This operand can be a literal or an expression

Unary operators		
`+x`	Identity	`+25`
`-x`	Numeric negation. Use on numeric operands only.	`-12`
`!x`	Logical Negation. Use on `bool` operands only.	`!true` `!(x > 5)`
`~x`	Alias of `!x`	`~(y == 0)`

Binary operators

Binary operators are applied to two operands. Each operand can be a literal or an expression.

Binary operators			

`x + y`	Returns the sum of x and y if both are numeric, or the concatenation of x and y if at least one of them is a string. Does not work on `bool` or other formats.	`1.05 + 2`
		`"part " + "one"`
		`"part " + 1`
			`3.05`
			`"part one"`
			`"part 1"`

`x - y`	Returns the difference of x and y if both are numeric, the result is negative if y is larger than x. Does not work on `bool`, `string` or other formats.	`90 - 45.1`	`44.9`
`x * y`	Returns the multiplicative result of x and y if both are numeric. Does not work on `bool`, `string` or other formats.	`0.1 * 0.8`	`0.08`
`x / y`	Returns the division result of x and y if both are numeric. Does not work on `bool`, `string` or other formats.	`0.2 / 45`	`0.0044444`
`x % y`	Returns the modulus result of x and y if both are numeric. Does not work on `bool`, `string` or other formats.	`10.1 % 3`	`1.1`

`x || y`	Returns the logical OR of x and y. This operation performs lazy evaluation; y need not be evaluated if x is true. Both operands must be `bool`.	`(i == 3) || (j == 2)`	`[i = 3, j = 3]`
			`true`
			`[i = 2, j = 3]`
			`false`

`x && y`	Returns the logical AND of x and y. This operation performs lazy evaluation; y need not be evaluated if x is false. Both operands must be `bool`.	`(i == 3) && (j == 2)`	`[i = 3, j = 3]`
			`false`
			`[i = 2, j = 3]`
			`false`

`x == y`	Returns true if x and y are equal. Incompatible types return false (float and int are compatible for equality checks, but int and bool are not).	`t == 5`	`[t = 3] false` `[t = 5] true`

`x != y`	Returns false if x and y are equal. Incompatible types return true.	`action != "wait"`	`[action = "run"]`
			`true`
			`[action = "drive"]`
			`true`

`x <> y`	Alias of `x != y`	`action <> "run"`	`[action = "run"]`
			`false`
			`[action = "drive"]`
			`true`

`x > y`	Returns true if x is more than y, otherwise returns false.	`positive > 0`	`true`
`x < y`	Returns true if x is less than y, otherwise returns false.	`negative < 0`	`true`
`x >= y`	Returns true if x is more than or equal to y, otherwise returns false.	`zero >= 0`	`true`
`x <= y`	Returns true if x is less than or equal to y, otherwise returns false.	`zero <= 0`	`true`

Ternary operators

Ternary operators are applied to three operands. Each operand can be a literal or an expression.

Ternary operators			

`b ? x : y`	Returns x if b evaluates to true, otherwise returns y.	`(a > 1) ? 15 : 12.5;`	`[a = 0]`
			`12.5`
			`[a = 2]`
			`15`

FUNCTION CALLS

Functions is the primary means for a script to invoke behavior from the game engine.

Context functions are called with the following format

Parameterless null function

Context function contract:
```
void Scene.FadeOut()
```
Example call in script:
```
Scene.FadeOut();
```

Parameterized null function

Context function contract:
```
void Scene.SetMaxBounds(float3 value)
```
Example call in script:
```
Scene.SetMaxBounds({10000,200,30000});
```

Parameterless value function

Context function contract:
```
int Player.GetActor()
```
Example call in script:
```
int playerID = Player.GetActor();
```

Parameterized value function

Context function contract:
```
string Actor.GetActorType(int actorID)
```
Example call in script:
```
string type = Actor.GetActorType(playerID);
```

The full list of available functions are defined in the Scripting Guide for **Context Functions**.

USE OF VARIABLES

Variables are a powerful tool as they can be used to store values after the evaluation of a statement. This value will then be available to subsequent statements or scripts.

Variables must be declared before they can be used. Variables cannot be declared more than once in the same scope

Variable declaration statements		
`type x;`	Declares an unassigned variable with a value type and a variable name x.	`int first_var;`
`type x = y;`	Declares a variable with a value type and a variable name x. The variable is assigned the value of y.	`string word = "assigned";`

Assignment operator statements perform an assignment of a value to a variable. Some assignment statements double as a binary operator.

Assignment statements		
`x = y;`	Assigns the value of y to the variable x. y may be a literal or an expression. The value type of y must be compatible with the type of x.	`int count;` `count = 1; //count is now 1`
`x += y;`	Alias of `x = x + y`	`count += 1; //count is now 2`

`x -= y;`	Alias of `x = x - y`	`count -= -1; //count is now 3`
`x *= y;`	Alias of `x = x * y`	`count *= 6; //count is now 18`
`x /= y;`	Alias of `x = x / y`	`count /= 3; //count is now 6`
`x %= y;`	Alias of `x = x % y`	`count %= 3; //count is now 0`
`x |= y;`	Alias of `x = x || y`	`bool tick = false;` `tick |= (count >= 0); // tick is now true`
`x &= y;`	Alias of `x = x && y`	`tick &= (count > 1); // tick is now false`

STATEMENT BLOCKS

More complex behavior is implemented as statement blocks – a group of statements joined by a particular syntax structure.

If-then-else statement block

```
if (<expression_bool>)
{
        <statement>
        <statement>
        …
}
else
{
        <statement>
        <statement>
        …
}
```

First evaluates **`<expression_bool>`**. If the result is **`true`**, evaluate the statements in the first block only. Otherwise, evaluate the statements in the **`else`** block only (if present). The **`else`** block is optional. If there is only one statement in the block, the encapsulating scope braces (`{, }`) is optional.

Foreach statement block

```
foreach (<type> <variable_name> in <source_array>)
{
        <statement>
        <statement>
        …
}
```

First, declare a new variable of type **<type>** and name **<variable_name>.** For each value found in **<source_array>**, assign that value to the variable, and then evaluate each of the statements in the subsequent group. After all values have been iterated, the variable is discarded as script execution leaves its scope. If there is only one statement in the block, the encapsulating scope braces (`{, }`) is optional.

<source_array> should be an array type of **<type>**. For example, int[].

For statement block

```
for (<statement_first> <expression_bool>; <statement_next>)
{
        <statement>
        <statement>
        …
}
```

First, evaluates **<statement_first>.** Then evaluates the statements in the block. Evaluates **<expression_bool>**. If the result is **true**, evaluate **<statement_next>**, evaluates the statements in the block, and re-evaluate **<expression_bool>**. Repeat until **<expression_bool>** returns **false**, after which it will exit this scope.

If there is only one statement in the block, the encapsulating scope braces (`{, }`) is optional.

While statement block

```
while (<expression_bool>)
{
        <statement>
        <statement>
        …
}
```

Evaluates **<expression_bool>**. If the result is **true**, evaluates the statements in the block, and re-evaluate **<expression_bool>**. Repeat until **<expression_bool>** returns **false**, after which it will exit this scope.

If there is only one statement in the block, the encapsulating scope braces (`{, }`) is optional.