TA3D Scripting guide

Roland Brochard

September 18, 2010

Abstract

This document is a small scripting guide. Its purpose it to introduce you to the Lua interface of TA3D which can be used to script game rules, AI, units, the developer shell, ...

There are several kinds of Lua scripts:

- unit scripts (in the *scripts* folder)
- game scripts (in the *scripts/game* folder)
- AI scripts (in the *scripts/ai* folder)
- developer shell scripts (in the *scripts/console* folder)

1 Introduction to Lua

The purpose of this guide is not to present Lua in details but only how to use it in TA3D. For general information about Lua please refer to the reference manual (http://www.lua.org/manual/5.1/).

Basically all you need to know in order to write (basic) Lua scripts for TA3D is the basics of Lua : variables, expressions and function calls.

There are 2 kinds of variables : global and local ones. Global variables don't have to be defined, the first time Lua encounters a variable it looks for it in the global environment and if it doesn't exist it creates it. Local variables have to be explicitly declared local.

```
i = 0              -- global variable
local e = 1        -- local variable
```

The scope of local variables is limited to the chunk of code where it is defined (inside a function, a loop, ...) whereas global variables can be accessed from everywhere (even other scripts as long as they share the same virtual machine like unit scripts). Accessing local variables is also faster so consider using them when you write performance critical code.

Functions are objects like numbers or strings and can be stored into variables which can then be used to call the function. You can define a function with the *function* keyword:

```
-- global declaration
function my_function(parameter1, parameter2)
  -- do something with parameter1 and parameter2
  return the_value_I_want_to_return
end

-- anonymous declaration
local variable = function (parameter1, parameter2)
  -- do something with parameter1 and parameter2
  return the_value_I_want_to_return
end
```

Calling a function can be done in 2 ways. The C-like function call is done that way:

```
-- C-like function call
my_function(parameter1, parameter2)
```

This example calls the function *my_function* with 2 arguments. You can also use the object oriented way:

```
-- here obj is an 'object' with a method
-- called my_method
obj:my_method(parameter1, parameter2)
```

In this example the *my_method* function is a normal function but it is called with 3 parameters the first one being the obj object. So you would have to define the function that way:

```
-- defining the my_method function of object obj
obj.my_method = function (this, param1, param2)
  -- do whatever you want with param1 and param2
  -- here this = obj if you call my_method with
  -- obj:my_method(...)
end
```

Well now you only need expressions to do some maths:

```
-- doing math is as simple as:
local result = (var0 + var1) / 2
```

You can use the standard operators +, -, *, /, % and through the *math* table you can access some more advanced math operations (see the Lua reference manual for this).

2 Unit scripts

Unit scripts are Lua scripts which follow a few conventions. Basically the script creates a model of object for a unit type which contains all the functions and information required to animate the unit.

So the first thing you need to do when creating a unit script is tell TA3D which unit will use this script and then get required information from the unit 3D model. You do this with 2 lines of code:

```
-- here we tell TA3D we want to animate a unit
-- whose internal name is armbrtha
createUnitScript("armbrtha")

-- just call __this:piece with the list of
-- piece names of the unit 3D model in order
-- to get those pieces IDs
__this:piece( "base", "flare", "turret", "barrel",
              "sleeve" )
```

Note the *__this* object which is the unit model object we're building when the script code is loaded by the engine. It is the Lua table we must fill with the unit functions and variables. Each unit of the game that will run our script will get a copy of this table to have its own data at runtime.

So what's next ? In its current state our script can be loaded by TA3D but it won't do anything visible that's the reason why we must add functions. All those functions are called with at least one parameter which is the unit object because it identifies the unit and this is required when you want to animate a piece of the model of this unit. Here is the list of functions the engine will call and when:

- **Create(this)**
  this function is called when the unit is created as soon as it appears in the game. When this function starts it's very unlikely (unless it's built instantly) the unit is built so don't animate it too early.

- **Killed(this, severity)**
  this function is called when the unit dies. The second parameter ranges from 0 to 100 and tells how badly it has been damaged before dying.
- **Activate(this)**
  this function is called when the unit 'activates' (it is turned on like solar panels for example)
- **Deactivate(this)**
  this function is called when the unit 'deactivates' (it is turned off like solar panels for example)
- **SweetSpot(this)**
  this function should only return the piece of the model enemies will target when shooting.
- **HitByWeapon(this, anglex, anglez)**
  guess what ? this function is called when the unit is hit by a weapon ! The *anglex* and *anglez* parameters are the direction of that weapon expressed as two rotation angles around the x and y axis.
- **TargetCleared(this)**
  this function is called when the target has been destroyed.
- **SetSpeed(this, wind_speed)**
  this function used to tell the script that wind speed has changed. This is used only by wind generators.
- **QueryNanoPiece(this)** for builders, this function should return the pieces that will emit nanolathe particles. Since all pieces should not be returned at the same time you must cycle through them, like this:

```
--this.QueryNanoPiece = function(this)
  this.spray = (this.spray + 1) % 2
  if this.spray == 0 then
    return this.beam1
  else
    return this.beam2
  end
end
```

- **StartBuilding(this)**
  this function is called when a builder (a factory or a mobile builder) starts building something.
- **StopBuilding(this)**
  this function is called when a builder (a factory or a mobile builder) stops building something (because it has finished or been interrupted).
- **QueryBuildInfo(this)**
  Like the *SweepSpot* function this function should return a piece of the model. This model piece will be used as the ground plate where the unit which is being built will be attached. This is used by factories only.
- **FirePrimary(this)**
  FireSecondary(this)
  FireTertiary(this)

FireWeaponN(this)
These functions are called when the first, second, third or N^{th} weapon is fired.

- **AimPrimary(this, heading, pitch)**
  AimSecondary(this, heading, pitch)
  AimTertiary(this, heading, pitch)
  AimWeaponN(this, heading, pitch)
  These functions are called when the first, second, third or N^{th} weapon aims in some direction specified by the *heading* and *pitch* parameters which are in TA angle units (if you use them you'll have to convert them to degrees or radians by multiplying by the TA2DEG or TA2RAD constants). When aiming is finished the script should tell the engine the weapon is ready to fire using the *this:set_script_value(#scriptname, true)* command like this:

```
--this.AimPrimary = function(this, heading, pitch)
  -- tell the engine we're not ready yet
  this:set_script_value("AimPrimary", false)

  -- convert angles from TA units to degrees
  heading = heading * TA2DEG
  pitch = pitch * TA2DEG

  -- update animation data
  this:turn( this.turret, y_axis, heading, 5 )
  this:turn( this.sleeve, x_axis, -pitch, 2 )
  -- check if another instance of the script
  -- is monitoring the process
  if this.aiming then
    return
  end

  -- tell the world we're doing the job
  this.aiming = true
  while this:is_turning( this.turret, y_axis )
        or this:is_turning( this.sleeve, x_axis )
  do
    this.yield()
  end

  -- tell the engine we're ready
  this:set_script_value("AimPrimary", true)
  this.aiming = false
end
```

  Note the *this.aiming* variable which is used to prevent multiple instances of the script to be running at the same time. You can use the signal stuffs used in OTA's BOS scripts, it'll work but since it requires creating/destroying lots of Lua threads it's very slow compared to this simple test (scripts don't run in parallel, this behavior is simulated so it's safe to use variables to synchronize several instances of a function).

- **AimFromPrimary(this)**
  AimFromSecondary(this)

AimFromTertiary(this)
AimFromWeaponN(this)
These functions are called before the first, second, third or N^{th} weapon start aiming, they should return the model piece from which aiming should be done.

- **QueryPrimary(this)**
  QuerySecondary(this)
  QueryTertiary(this)
  QueryWeaponN(this)
  These functions are called before the first, second, third or N^{th} weapon is fired, they should return the model piece from which the weapon object (bullet, missile, laser beam, ...) is created.

3 Game scripts

4 AI scripts

5 Developer shell scripts