Scripting tutorial

by Shining
(last update : 21/08/2007)

Summary

I/ Have fun with LUA !

LUA is a scripting language, designed to be much easier to use than C++ : learning it will be your first step to be able to customize Far Cry.
So, what do we have in a scripting language ? Variable, functions and some tools to play with them... Actually, the aim of a script is to manage variables : get them, test them, modify them according to what you need.

A/ The variables

There are different types of variable in LUA : the type you will probably use are number, string and arrays. To create a variable that doesn't exist, you will use the word “local”, then the name you want to give to your variable. And you can initialize it.

code:

```
local toto=1;
local zaza=”hello world !”;
```

This is what we called an instruction : as you can see, it's ended by a semicolon.
So here, my variable toto is a number and my variable zaza is a string. As you can see, when you want to create a string, you need to write your text between quotes.
An array can gather several values in just one variable : look how we can create an array.

code:

```
local myArray={x=1,test=”hello”,name=”bob”};
```

So now, you have one variable that contains three values :
myArray.x (equals 1)
myArray.test (equals “hello”)
myArray.name (equals “bob”)
You can too call these values this way : myArray[1], myArray[2], myArray[3]
Of course, once created, you can then change the value of the variable and its type :

code:

```
local aux=1;
aux=”test”;
```

B/ The functions

A function is often used to perform some processes that are based on arguments and that returns a result. Look at this example :

code:

```
function makeTheSum(firstValue,secondValue)
   local res=firstValue+secondValue;
   return res;
end
```

As you can see, to create a function, you need to declare it this way, with the word “function”, then the name of your function (here “plusOne”), then the arguments between parenthesis and separated by comma. Then we perform what we want with our argument (here an addition) and we return the result thanks to the keyword “return” : the end of the declaration of the function is done thanks to the keyword “end”. Functions don't necessarily return something.
So, how can we use it ?

code:

```
local myAddition=makeTheSum(3,4);
```

Now, myAddition equals 7.

C/ The Tools

"if" statement

Use to test if a condition is true. This is how we use it in LUA :

code:

```
local a=1;
local b=1;
local aux=0;
if (a==b)then
  aux=a+b;
end
```

The global structure is
if(condition)then
some code
end
Here, == means "equals". You can use too
~= which means different
> upper
< lower
>= upper or equal
<= lower or equal
Then, you can use logical terms "and" and "or", like that

code:

```
if(a==b and a~=aux)then
  --some code
end
```

"while" statement

Use to perform something as long as a condition is true
while (a<10) do
a=a+1;
end
Here, as long as a is lower than 10, I add one to a. So, when a equals 10, the loop stops.

“**for” statement**

- Used to perform something X times
local a=0;
for nb=1,10 do
a=a+1;
end
That will perform “a=a+1” 10 times.

- Used to skim through an array

code:

```
local myArray={w=0,x=1,y=2,z=3};
local res=0;
for key,value in myArray do
res=res+value;
end
```

key is the name of the variable in my array, and value is the value of the key variable.
So here, it will perform
res=res+0; (value of w)
res=res+1; (value of x)
res=res+2; (value of y)
res=res+3; (value of z)

II/ What does an entity script looks like ?

Now, you should be able to understand what is written in FarCry LUA files !
So let's open a file, like BasicEntity.lua

A/ Properties of an entity

As you can see, entity are arrays

code:

```
BasicEntity = {
```

In this array, there are variables (numbers, strings) and also other arrays. One of them is the array called “Properties” which is really important : every variables in it can be modified in the editor.

code:

```
Properties = {
        object_Model = "",

        aianchorAIAction = "",
        fAnchorRadius = 0,
```

But the name aren't exactly the same as in the editor, right ?
Actually, in FarCry, the first letters of a variable define his type in the editor.

So, if you put before the name of your variable

object_ : you will have an icon to select a cgf/cga from the library
aianchor : select an AI anchor from the existing ones
f : float number
snd : you will have an icon to select a sound from the library
b : boolean (checkbox, will be equal to 0 or 1)
vector_ : you will have 3 fields (x, y and z)

B/ Functions in an entity

And how link functions to this entity ?
Very easy : as you can see, just write
function [name of the entity]:[name of the function]([parameters)]
--some code
end
like that

code:

```
function BasicEntity:IsARigidBody()
    local qual=tonumber(getglobal("physics_quality"));

    if qual>0 or self.Properties.Physics.LowSpec.bKeepRigidBody==1 then
         return self.Properties.Physics.bRigidBody;
    end

    return self.Properties.Physics.LowSpec.bRigidBody;
end
```

So, as we are now in a function, let's try to understand it !
If you read the LUA part of this tutorial, you should be able to understand a lot of things : there's a variable declaration (local qual), some tests (if) and a value returned. But what is this “self” thing ?

Actually, this is what we call a reference : self is the entity we are dealing with, and, eventually, the array we've seen (BasicEntity = {).

As you know, we can call variables from an array thanks to a point after the name of the array. Therefore self.Properties.Physics.bRigidBody means that there is an array called Properties : in Properties, there is an array called Physics and in this array there is a number called bRigidBody.

Look an entity as a tree : you have a root (self, or any reference to the object), then the first branches, then the second branches and so on. To access to a leave, you have to write the whole path, from the root to the last branch.

Self only works in the entity script itself : if you want to access to another entity (like a dynamic light) in BasicEntity.lua, you will have to know its reference. How can you get it ?
The easier way is to use the function System:GetEntityByName("[name of the entity]"); this way for instance :

code:

```
local myLight=System:GetEntityByName("light1");
```

So now, you can use its functions and access to its properties. If you open DynamicLight.lua, you can find the variable bOnlyForHighSpec so I can test it in BasicEntity.lua, writing

code:

```
if (myLight.Properties.Optimization.bOnlyForHighSpec==1)then
some code
end
```

You can too call the function OnReset of this dynamic light this way :

code:

```
myLight:OnReset();
```

As you can see, to call an entity function, use a colon between the reference of the entity and the name of the function.

Hey, this is an example of a function that takes no argument and returns nothing : what is its purpose ?

Well, if you look at it, you will see that it manages several entity variables, call other functions and so on : this is a function that manages the entity.

C/ Callbacks and events

Let's continue to talk about this function : it is what we call a callback. What is that ? It is a function that is called by the system (you don't need normally to call it yourself) at a precise moment. Actually, almost every functions that begins with “On” are callbacks.

Let's see this first list :
OnInit() : called when you start the game.
OnReset() : called when you start and restart the game (in editor mode for instance)
OnPropertyChange() : called when you change a value in the editor.
OnDamage(hit) : called when an entity receives a hit (like a bullet). The parameter “hit” is an array that contains several pieces of information about that hit (position, direction, force and so on)
OnSave(stm) : called when you reach a savepoint
OnLoad(stm) : called when you start the game from a save
OnTimer() : called after you set a timer
OnUpdate(DeltaTime) : called every delta time seconds. Need to be enable thanks to the function EnableUpdate and SetUpdateType
OnShutDown() : called when you quit the game
OnCollide(hit) : called after a collision with another entity. The parameter “hit” is an array that contains several pieces of information about that hit (position, direction, force and so on)
OnDamage(hit) : called after losing some life

Now, let's see some other special functions... The “event” function.
If you create a function and put before its name Event_ then you will be able to call it from the editor (with a proximity trigger for instance). Actually, it will be in the Input/Output event list below the Properties of your entity. So you will be able to send a custom event to your entity... but how can we launch an output event from the script ? Use the function BroadcastEvent([reference to your entity],”[name of the event without Event_]”); like this

code:
```
BroadcastEvent(self, "Hide");
```

D/ Inheritance

Finally, let's talk about “inheritance”
Entities can be built according to a “root” model : for instance, player.lua is based on BasicPlayer.lua. This link can be really strong in this case

code:

```
Player.Client =
{
--      OnTimer = BasicPlayer.Client_OnTimer,
        OnInit = Player.Client_OnInit,
        OnShutDown = BasicPlayer.Client_OnShutDown,
```

Here, we told the system that if we call inside player.lua the function OnShutDown, it will use the OnShutDown function from BasicPlayer.lua, using parameters from player.lua (in fact, player.lua and BasicPlayer.lua must have some common parameters).

How does it work ?
We don't see that but when you call class function like function Player:ChangeEnergy(Units) this way self:ChangeEnergy(Units), you do Player.ChangeEnergy(self,Units) : each time, the class array is used as an argument and the variables in this array are used (self.iPlayerEffect, self.keycards and so on...). That is why we can write BasicPlayer. Client_OnShutDown(self) in player.lua as long as the variables used in this BasicPlayer function exist too in player.lua.

E/ Different kind of variables

To end this, I'd like talk about different kind of variables...
- local variable : they exist only in the function where they have been created, thanks to the keyword “local”. Once the function is finished, you can't access them anymore.
- class variable : they are located in the class array of an entity script. You will use self to get them and you can modify them in a function : they will be still modified once the function ends. Every entities created in-game have their own variables, they are not shared.
- global variables : they are not linked to entity and you can get them everywhere in your script. To create one global variable, use the function Game:CreateVariable(“name of the variable”,”initial value”); To get the value of a global variable, use Game:GetVariable(“name of the variable”); and to set the value of a global variable, use Game:SetVariable(“name of the variable”,”new value”);

F/ Modifying properties of an entity

As we have seen, you can modify variables of an entity inside its script using "self" and outside its script by getting its reference (using System:GetEntityByName for instance).
Now, there are three cases :

1) you just want to change a value for an internal use so that is sufficient

2) you want to change something visible in the entity (physics, health, graphics...) and it has an immediate effect, eg

code:
```
_localplayer.cnt.health=_localplayer.cnt.health/2;
```

is enough to halve player's health

3) you want to change something visible in the entity (physics, health, graphics...) but it seems it has changed nothing (or it took effect only when you switch in-game), eg

code:
```
_localplayer.DynProp.gravity=-10;
```

The player should fly but nothing happens... In this case, you must call a specific function to tell the system that you have changed one variable. The easier way is to call the OnReset function (or the OnInit function) of this entity but it may not work (or give you other results you don't want) and it's not very clean. So look for where the variable you modify is used in the entity scripts. If you don't find it, look for the array that contains this variable (eg. DynProp). If you still don't find it, look for in the "root" entity (eg BasicPlayer.lua for player.lua).
In our case, to make the player fly, we will find what you want in BasicPlayer.lua, looking for the DynProp array : got the function self.cnt.SetDynamicsProperties(self.DynProp) !
So, to make the player fly dynamically in-game, we will write

code:
```
_localplayer.DynProp.gravity=-10;
_localplayer.cnt:SetDynamicsProperties( _localplayer.DynProp);
```

III/ Special scripts

There are other scripts than entity scripts that are special tools or that perform special tasks. Let's see some of them.

A/ Mission script

Very well-known I think ! Used to call some Mission functions from a trigger, like that

code:

```
function Mission:Event_StartCutScene4()
        Movie:PlaySequence( "cin4" );
end
```

B/ HudCommon.lua

Script that manages the Hud (displaying icons...). Its function Hud:OnUpdateCommonHudElements() is called every frame so it can be pretty useful.

C/ Common.lua

Gather global functions that can be called everywhere in your scripts. So you can add your own global functions.

D/ ClassRegistry.lua

Gather every entity scripts. If you create one, don't forget to add it in this file.

E/ ModExe.lua

Called when your mod is loaded so you can use it to perform some initializations, like creating global variables...

IV/ AI

A/ Introduction

What is AI in FarCry ? Well, the main terms we are going to use are behaviors, characters, signal, events and goal pipes. Every piece of information I'm going to give you here are fully explained and detailed in the AI manual.pdf, found in the SDK folder (Docs/AI).
An AI agent has a character that defines which behavior he will have according to some situations. A behavior is based on several events that are triggered when the AI agent got some signals. Then the AI agent successively performs actions listed in a goal pipe.

B/ Creating a goal pipe

Most of goal pipes are created in the file PipeManager.lua, but you can create a goal pipe everywhere in your scripts, even during the game. A goal pipe contains “atomic goals” which are basically simple goals (that cannot be divided into sub-goals) such as look, run, cover and so on.

To create a goal pipe, use the function AI:CreateGoalPipe(“[name of the goal pipe]”);
Once it's created, you can add goals in it with the function AI:PushGoal(“[name of the goal pipe]”,”[name of the atomic goal]”,[0 or 1 : 1 if the goal has to be finished, 0 if it's not necessary],[then arguments that are specific to the atomic goal])
So, let's have an example :

code:

```
AI:CreateGoalPipe("run_to_player");
AI:PushGoal("run_player","run",0,1);
--the fourth parameter of the “run” goal is 0 (walk
--when move) or 1 (run when move)
AI:PushGoal("run_player","locate",1,"player");
--the fourth parameter of the “locate” goal is an AI --
object name, like a tag point name
AI:PushGoal("run_player","approach",1,0);--the fourth
--parameter indicates the distance to approach
```

So if an AI agent select this pipe, he will be in “run” mode then locate the player and he will run to the player.
But how can he select a goal pipe ? We are going to see that with the behavior creation.

C/Behavior creation

To add your new behavior, you have to create a file YourBehavior.lua in Scripts/AI/Behaviors/Personalities. If it's the first time, use the file called Prototype.lua.
Then you will have to add this behavior in the file AIBehavior.lua (in Scripts/AI/Behavior) : to do that, write below

code:
```
AIBehaviour = {

        AVAILABLE = {
yourBehavior =  Scripts/AI/Behaviors/Personalities/YourBehavior.lua
```

So, what does a behavior script look like ? It will begin like that

code:
```
    AIBehaviour.yourBehaviorName = {
            Name = "yourBehaviorName",
```

So, as you can see, it's an array that contains the name of the behavior and this kind of function :

code:
```
    OnPlayerSeen = function( self, entity, fDistance )

    end,
```

These functions are callbacks : if you use them, you won't need to call them, the system will. In the case of OnPlayerSeen, if you have selected in the editor one behavior that has a OnPlayerSeen function, it will be called each time your AI will see the player. In this function, self is the behavior script itself, entity is the AI that has seen the player and fDistance is the distance between the AI and the player. You can see in Prototype.lua every callbacks and their parameters.

Now, you can make your AI select a goal pipe you created in these functions with the function SelectPipe(0,"[name of the goal pipe]"); this way :

code:
```
    OnPlayerSeen = function( self, entity, fDistance )
            entity:SelectPipe(0,"run_to_player");
    end,
```

Here, when our AI will see the player, he will run to the player.
Actually, OnPlayerSeen is what we call a signal (sent to the AI). But how can we create our own signals ?Very easy.

D/ Creating signals and sending them

First, create in your behavior the function that will react to your signal, like "OnHearingTheProgrammer" :

code:
```
OnHearingTheProgrammer = function( self, entity )
        Hud:AddMessage("what can I do for you master ?");
        entity:SelectPipe(0,"run_to_player");
end,
```

But this time, it's not the system which will send this signal : it's you ! ;)
How ? There are two kinds of signal you can send : anonymous signal (sent at a defined point in the map) or not (sent by an AI). There are the two functions :
AI:FreeSignal(signal_type,signal_name,position,radius);
AI:Signal(signal_filter,signal_type,signal_name,entity_id);
where
- signal_type is -1 (AI that receives the signal will always perform it),0 (AI that receives the signal will perform it if he's not "ignorant")or 1 (AI that receives the signal will perform it if he's not "ignorant" and if he's enabled).
- signal_filter defines which kind of agents will receive the signal (0 for everyone)
- signal_name is the name of the signal
- position is the vector x,y,z that defines the position from where the signal will be sent
- radius is a number (circle around the position where the signal can be received)
- entity_id is usually self.id (if its called from the AI entity script) or entity.id (if it's called inside the behavior)

So, in our example, we could write somewhere in our script (like a mission event...)

code:
```
AI:FreeSignal(0,"OnHearingTheProgrammer",{x=0,y=100,z=16},30);
```

E/ Character creation

Finally, let's talk a little bit about Character.
A character allow an AI to change his current behavior to a new behavior if he received a defined signal. The script is written like this :

code:

```
AICharacter.YourCharacter= {

        CurrentBehavior1 = {
                Signal1Received  = "name of the new behavior",
                Signal2Received= "name of another behavior",
        },

        CurrentBehavior2 = {
                Signal3Received  = "name of the new behavior",
                Signal4Received= "name of another behavior",
        },
}
```

So if the AI uses the behavior1 and if he received the signal1, he will select another behavior...

F/ AI objects

To end the AI section, I would like to talk about AI objects like tag points. You will use these to build paths for instance, to show him where he has to go... You will note that you can't move tag points in-game. If you need to change dynamically the path of your agent, you can create several goal pipes (with a lot of tag points), or you can move entities that will “emulate” tag points. To do that, use the function

code:

```
AI:RegisterWithAI(self.id,AIOBJECT_WAYPOINT);
```

in the entity script you will use to emulate tag points.

V/ FarCry & LUA functions

A/ LUA functions

- **type(variable)** : To know what kind of type your variable is, use the type function

code:

```
local a=1;
local aux=type(a);
--aux equals "number"
```

- **getn(array)** : return the number of fields of an array

code:

```
local myArray={x=1,toto="hello"};
local numFields=getn(myArray);
--numFields equals 2
```

- **randomseed(seed); random()** : get a random number

code:

```
randomseed(System:GetCurrTime());
local randNum=random(10);
-- randNum is between 0 and 10
```

B/ Entity functions

- **System:GetEntityByName("name of the entity")** : return the reference of an entity to modify it

code:

```
local myAI=System:GetEntityByName("merc1");
myAI:Event_Die();
```

- **entity:GetPos()** : return an array with three position fields, x, y and z

code:

```
local playerPos=_localplayer:GetPos();
local playerPosX=playerPos.x;
```

- **entity:GetAngles()** : return an array with three angles fields, x, y and z

code:

```
local playerAng=_localplayer:GetAngles();
local playerAngX=playerAng.x;
```

-**entity:SetPos(xyz vector)** : change the position of the entity

code:

```
local myAI=System:GetEntityByName("merc1");
local newPos={x=100,y=200,z=16};
myAI:SetPos(newPos);
```

-**entity:SetAngles(xyz vector)** : change the angles of the entity

code:

```
local myAI=System:GetEntityByName("merc1");
local newAng={x=100,y=200,z=180};
myAI:SetAngles(newAng);
```

- **entity:SetMaterial("path to your material in the database")** : change the material of your entity

code:

```
local myBuggy=System:GetEntityByName("buggy1");
myBuggy:SetMaterial("level.mat1.wood");
```

- **entity:StartAnimation(0,"name of the animation",layer,blend time,speed,loop)** : start the animation of an entity. For cga, the name of the animation is "Default". For cgf, see your cal file or your caf files.

code:

```
_localplayer:StartAnimation(0,"swim",0,0,1,0);
```

- **entityTree:Bind(entityBranch)** : bind an entity to another

code:

```
local fleur=System:GetEntityByName("flower1");
_localplayer:Bind(fleur);
```

- **entityTree:AttachToBone(entityBranch,"name of the bone")** : attach an entity to an entity's bone

code:

```
local fleur=System:GetEntityByName("flower1");
_localplayer:AttachToBone(fleur,"biped1 LHand");
```

C/ Graphic functions

- System:SetScreenFx("ScreenFade",1)
System:SetScreenFxParamFloat("ScreenFade","ScreenFadeTime",time fade) : fade in (time fade>0) or fade out (time fade<0) the screen during “time fade” seconds

- System:SetScreenFxParamFloat("FlashBang", "FlashBangForce", number);
System:SetScreenFxParamFloat("FlashBang", "FlashBangFlashPosX",number);
System:SetScreenFxParamFloat("FlashBang", "FlashBangFlashPosY", number);
System:SetScreenFxParamFloat("FlashBang", "FlashBangTimeScale", number);
System:SetScreenFxParamFloat("FlashBang", "FlashBangTimeOut", number);
System:SetScreenFx("FlashBang", 1) : make a flash

- System:LoadImage(“path of the image from FarCry folder") : load an image

- System:DrawImage(reference of the image,X position, Y position, width, height, blend type, red, green, blue, transparency) : draw an image, must be called each frame

code:

```
local myImage=System:LoadImage(“Mods/YourMod/Images/image1")
System:DrawImage(myImage,0, 10,512, 512, 4,0, 0, 0, 0.5)
```

D/ Input functions

- **Input:GetXKeyDownName()** : return the name of the key currently held down

- **Input:GetXKeyPressedName()** : return the name of the key currently pressed

- **Input:BindCommandToKey("function name","key",0 -will be triggered each frame or 1 -only one time)** : bind a key to a global function

- **Input:BindAction("function name","key","action map : default, binozoom, vehicle...",0 -will be triggered each frame or 1 -only one time)** : Bind a key (or other input) to a certain action (eg. JUMP)

E/ Game functions

- **Movie:PlaySequence("name of your sequence")** : play your sequence

- **Game:SendMessage("StartLevel nameofYourLevel")** : launch a level

code:

```
Game:SendMessage( "StartLevel carrier" )
```

- **Game:SetThirdPerson(0 or 1)** : activate (1) or deactivate (0) the third person mode

F/ Sound functions

- **Sound:Load3DSound(path of the sound from FarCry folder)** : load a 3D sound (must be mono)

code:
```
local mySound= Sound:Load3DSound("Mods/myMod/Sounds/sound1.wav");
```

- **Sound:LoadSound(path of the sound from your mod folder)** : load a sound, return its reference

- **Sound:PlaySound(reference of the sound)**: play a sound

code:
```
local mySound= Sound:LoadSound("Mods/myMod/Sounds/sound1.wav");
Sound:PlaySound(mySound);
```

- **Sound:SetSoundVolume(reference of the sound, volume from 0 to 255)** : set the volume of a sound

- **Sound:SetSoundPosition(reference of the 3D sound,xyz vector)** : place a 3D sound in the space

- **Sound:SetSoundLoop(reference of the sound, 0 or 1)** : set if the sound must loop (1) or not (0)

- **Sound:IsPlaying(reference of the sound)** : return 1 if the sound is playing, nil if not

- **Sound:GetSoundLength(reference of the sound)** : return how last the sound.

G/ System functions

- **System:GetCurrTime()** : return time elapsed since you launch a level (or the editor)

- **System:GetFrameTime()** : return time elapsed between two frames

H/ File functions

- openfile(path to the file from the FarCry folder,"r+" or "w+" or "a+") : open a file in the specified mode, r+ for reading the file only, w+ for writing in it (will erase the file if it exists, will create it if it doesn't), a+ for adding content in the file (will create the file if it doesn't exist)

code:

```
local hand=openfile("Mods/YourMod/MyText/text1.txt");
```

- read(reference to the file,"*l") : read a line in your file

code:

```
local s="";
local hand=openfile("Mods/YourMod/MyText/text1.txt");
s=read(hand,"*l");
Hud:AddMessage(s);--first line of your text
s=read(hand,"*l");
Hud:AddMessage(s); --second line of your text
--and so on, until s==nil
```

- write(reference to the file, one string, another string, another string and so on...)

code:

```
local hand=openfile("Mods/YourMod/MyText/text1.txt");
write(hand,"hello","how are you ?","\n");
-- \n for a new line
```

- closefile(reference of the file) : close a file

I/ Debugging your code

Now that you can do what you want with scripts, which ways can you use to test what you wrote ?

1) Print messages on screen ! This is the best way to know how things happen : functions called, variable values and so on. You can use Hud:AddMessage but sometimes, it's not convenient so you can use too the System:LogAlways (will print message on the console) or the Game:WriteHudString (must be called each frame).

2) You can use the included debugger for LUA scripts. Start Far Cry in devmode and set sys_script_debugger to 1 inside the system.cfg.

3) Begin with big effects then reduce them, so you will be sure that what you wrote works.

4) Don't hesitate to change variable values in scripts to see if "something" happens and what.

Make headway step by step or you won't be able to see what's wrong and you could erase code that works... Too bare ! ;)