SCRIPTING

Durante los próximos días aprenderemos a integrar un lenguaje de scripting dentro de nuestro videojuego.

Lenguajes de Scripting

Dentro de la industria de los videojuegos nos encontramos diferentes de lenguajes de scripting utilizados, los más utilizados son:

- LUA
- Python

El lenguaje de scripting que utilizaremos será el lenguaje más utilizado dentro de los videojuegos y este es el lenguaje LUA. De su página web http://www.lua.org podemos extraer la siguiente información:

- Lua fue creado el año 1.993 y fue creado en la Universidad PUC-Rio (the Pontifical Catholic University of Rio de Janeiro en Brasil). Lua significa luna en portugués
- Es el lenguaje de scripting líder dentro de la industria de los videojuegos
- Lua es gratuito tanto para uso comercial como académico
- Y remarcar que Lua es robusto, rápido, portable a otras plataformas, fácilmente integrable, potente y requiere de poca memoria

Instalar LUA

La instalación de LUA empezará en descargar la librería con el código fuente desde su página web. Una vez descargada crearemos un proyecto para Visual Studio de tipo librería estática e introduciremos todo el código fuente para crear la .lib correspondiente.

Uso de LUA

Para poder utilizar el lenguaje de scripting LUA dentro de nuestro videojuego crearemos un mánager de scripting como el siguiente:

```
#ifndef SCRIPTMANAGER_H
#define SCRIPTMANAGER_H

#include <string>
#include <vector>

extern "C"
{
	#include "lua.h"
	#include "lualib.h"
	#include "lauxlib.h"
}

class CScriptManager
{
private:
	lua_State 	*m_LS;
public:
	CScriptManager();
	void Initialize();
	void Destroy();
	void RunCode(const std::string &Code) const;
	void RunFile(const std::string &FileName) const;
	void Load(const std::string &XMLFile);
	lua_State * GetLuaState() const {return m_LS;}
	void RegisterLUAFunctions();
};

#endif
```

Una vez declarada la clase deberemos implementar el código correspondiente a cada uno de sus métodos. Para ello utilizaremos código similar al siguiente.

```
//Para inicializar el motor de LUA
void CScriptManager::Initialize()
{
	m_LS=luaL_newState();
	luaL_openLibs(m_LS);
	//Sobreescribimos la función _ALERT de LUA cuando se genere algún error al ejecutar
	código LUA
	lua_register(m_LS,"_ALERT",Alert);
}

//Código de la función Alert que se llamará al generarse algún error de LUA
int Alert(IN lua_State * State)
{
	std::string l_Text;
	int n = lua_gettop(State);
	int i;
	lua_getglobal(State, "tostring");
	for (i=1; i<=n; i++) {
		const char *s;
		lua_pushvalue(State, -1);
		lua_pushvalue(State, i);
		lua_call(State, 1, 1);
```

```
                s = lua_tostring(State, -1);
                if (s == NULL)
                        return luaL_error(State, "`tostring' must return a string to `print'");
                if (i>1) l_Text += '\t';
                l_Text += s;
                lua_pop(State, 1);
        }
        l_Text += '\n';
        Info( l_Text.c_str() );
        return true;
}

//Para desinicializar el motor de LUA
void CScriptManager::Destroy()
{
        lua_close(m_LS);
}

//Para ejecutar un fragmento de código LUA
void CScriptManager::RunCode(const std::string &Code) const
{
        if(luaL_dostring(m_LS,Code.c_str()))
        {
                const char *l_Str=lua_tostring(m_LS, -1);
                Info("%s",l_Str);
        }
}

//Para ejecutar un fichero de código LUA
void CScriptManager::RunFile(const std::string &FileName) const
{
        if(luaL_dofile(m_LS, FileName.c_str()))
        {
                const char *l_Str=lua_tostring(m_LS, -1);
                Info("%s",l_Str);
        }
}
```

Registro de funciones

Una vez sabemos inicializar el motor de LUA debemos aprender a registrar funciones LUA, para ello utilizaremos siempre el formato de función siguiente:
int NombreFuncion(lua_State *L);

Todas las funciones de LUA se registran de la manera que hemos comprobado reciben como parámetro el lua_State y devuelven un entero.

El entero que devuelve será el número de valores que retornará la función a LUA y del lua_State extaeremos los parámetros que esperamos de la función.

Por ejemplo :

```
//Si quisieramos implementar una función que en C fuese
//void SetSpeedPlayer(int Speed);
//Su implementación sería
int SetSpeedPlayer(lua_State *L)
{
	m_PlayerSpeed=(int)lua_tointeger(L,1);
	return 0;
}

//Si quisieramos implementar una función que en C fuese
//int GetSpeedPlayer();
//Su implementación sería
int GetSpeedPlayer(lua_State *L)
{
	lua_pushinteger(L, (int) m_PlayerSpeed);
	return 1;
}
```

Por último para hacer visibles estas funciones desde LUA deberemos registrarlas en el lua_State de la siguiente manera.

```
lua_register(m_LS, "set_speed_player", SetSpeedPlayer);
lua_register(m_LS, "get_speed_player", GetSpeedPlayer);
```

Lenguaje LUA

Una vez sabemos registrar funciones y inicializar el motor de LUA debemos ser capaces de implementar código en LUA.

Lo primero que debemos conocer es la sintaxis del lenguaje.

```
-- Esto es un comentario en una linea

--[[Esto es un comentario en diferentes lineas
de LUA]]

-- Declaración de variables

local var_str="string value"
local var_value=3
local var_value_float=3.5

-- Declaración de tablas
```

```
local tabla_strings={“valor_a”, “valor_b”, “valor_c”}
local tabla_valors={3,2,4}

-- Sentencia if then else
if value == true or value==”string value” and value~=”5” and value==”nil” then
        return true
else
        return false
end

-- Sentencia while do end
while value<20 do
        value=value+1
end

-- Sentencia repeat until
repeat
        value=value+1
until value==20

-- Sentencia for
for i=1,10 do
        value=value+1
end

-- Declaración de funciones
function f_name(parametre_1, parametre_2)
        return parametre_1+parametre_2
end

-- Y la llamada de la función
local sum = f_name(3,2)
```

Para cualquier aclaración del lenguaje o funcionamiento del lenguaje siempre podemos revisar la documentación de ayuda de la propia página web de LUA.

Registro de clases en LUA

Como hemos visto al inicializar el motor de LUA, nos encontramos con una librería que funciona en standard C, esto significa que no utiliza clases, por lo tanto no permite el registro de clases ni su uso.

Sin embargo mediante una nueva librería que utilizaremos a la vez que LUA nos permitirá registrar clases y poder utilizarlas dentro de nuestro código LUA, esta librería será LUABind.

¿Qué es LUABind?

Según su página web:

"Luabind is a library that helps you create bindings between C++ and Lua. It has the ability to expose functions and classes, written in C++, to Lua. It will also supply the functionality to define classes in lua and let them derive from other lua classes or C++ classes. Lua classes can override virtual functions from their C++ baseclasses. It is written towards Lua 5.x, and does not work with Lua 4.

It is implemented utilizing template meta programming. That means that you don't need an extra preprocess pass to compile your project (it is done by the compiler). It also means you don't (usually) have to know the exact signature of each function you register, since the library will generate code depending on the compile-time type of the function (which includes the signature). The main drawback of this approach is that the compilation time will increase for the file that does the registration, it is therefore recommended that you register everything in the same cpp-file."

A destacar los siguientes puntos:

- LUABind es una librería que ayuada a crear binding entre C++ y Lua
- Permite registrar funciones en escritas en C++
- LUABind sólo funciona con la versión 5.x de LUA
- Por su propia implementación está implementada a través de templates,por lo tanto no necesita ningún tipo de programación extra a la usual
- Su principal inconveniente es que es pesado de compilar debido a este punto a favor

Instalar LUABind

Para instalar LUABind descargaremos de la página web la versión de LUABind que nos interese y con esta versión crearemos un proyecto y lo compilaremos creándonos una librería .lib con la que compilaremos nuestro proyecto.

http://www.rasterbar.com/products/luabind.html

Si buscamos documentación de la librería podemos acceder a la siguiente web.

http://www.rasterbar.com/products/luabind/docs.html

Instalar Boost

Para la correcta instalación de LUABind deberemos instalar también la librería boost de la página web http://www.boost.org. Esta librería introduce una colección de funcionalidades que son estándar para diferentes sistemas operativos.

Para instalar esta librería sólo deberemos extraer los ficheros a una carpeta y nos creará el contenido de la librería, para el uso de esta librería no deberemos generar ningún proyecto desde el Visual Studio, sólo deberemos decirle la situación de los header files.

http://www.boost.org/

Incompatibilidad LuaBind 0.9 con LUA 5.2

Debido a unos cambios internos en la librería LUA en la versión 5.2, la versión 0.9 de luabind no es del todo compatible con LUA, para ello deberemos realizar pequeños cambios en las llamadas a funciones, nos basaremos en la siguiente web para realizar los cambios y conseguir hacer funcionar la librería sobre la nueva versión de LUA.

https://git.colberg.org/luabind.git

Uso de LUABind

Una vez tenemos generada la librería de LUABind podemos utilizarla dentro de nuestro proyecto, para su uso sólo deberemos implementar la siguiente línea de código en el Init de la clase CScriptManager.

```
luabind::open(m_LS);
```

Deberemos también incluir los archivos de cabecera del luabind que necesitaremos para poder implementar código con luabind. Así como incluir el namespace de luabind.

```
#include <luabind/luabind.hpp>
#include <luabind/function.hpp>
#include <luabind/class.hpp>
#include <luabind/operator.hpp>

using namespace luabind;
```

Para registrar funciones realizadas en C++ o C utilizaremos la siguiente macro y el código utilizado para la función será el estándar en cuanto a parámetros y funciones. Por ejemplo:

```
#define REGISTER_LUA_FUNCTION(FunctionName,AddrFunction)
{luabind::module(LUA_STATE) [ luabind::def(FunctionName,AddrFunction) ];}

void SetSpeedPlayer(int Speed)
{
	m_PlayerSpeed=Speed;
}
```

```
//Para registrar la función que sea visible en LUA
REGISTER_LUA_FUNCTION(“set_speed_player”, SetSpeedPlayer);
```

Como apreciamos no hemos modificado nada en la forma de escribir la función que ya es un avance respecto a la forma de trabajar de LUA, pero el gran avance lo vamos a apreciar en la posibilidad de registrar clases.

```
//Para registrar la clase CscriptManager que sea visible en LUA
module(LUA_STATE) [
	class_<CScriptManager>("CScriptManager")
		.def("load_file", & CScriptManager::LoadFile)
		.def("run_code", & CScriptManager::RunCode)
		.def("run_file", & CScriptManager::RunFile)
		.def("load", & CScriptManager::Load)
];
```

```
//Si queremos registrar una clase CPlayer y queremos registrar sus métodos y algún atributo lo haríamos de la siguiente forma
module(LUA_STATE) [
	class_<CPlayer>("CPlayer")
		.def("set_speed”, &CPlayer::SetSpeed)
		.def("get_speed", &CPlayer::GetSpeed)
		.def_readwrite("speed", &CPlayer::m_Speed)
];
```

```
//Si queremos registrar una clase CPoint y queremos registrar su constructor  y operadores de suma, resta, multiplicación, división u operador de comparación
module(LUA_STATE) [
	class_< Vect3f >("Vect3f")
		.def(constructor<float, float, float>())
		.def(const_self + const_self)
		.def(const_self - const_self)
		.def(const_self * const_self)
		.def(const_self / const_self)
		.def(const_self == const_self)
		.def_readwrite("x", &Vect3f::x)

];
```

```
//Si queremos registrar una clase templatizada como la clase CTextureManager debemos primero registrar su clase base con la clases templatizada y después la clase
module(LUA_STATE) [
	class_<CTemplatedMapManager<CTexture>>("CTemplatedMapManager")
		.def("get_resource", &CTemplatedMapManager< CTexture >::GetResource)
];
module(LUA_STATE) [
	class_< CTextureManager<CTexture>,
			CTemplatedMapManager<CTexture>>("CTextureManager")
		.def(“load”, CTextureManger::Load)
];
```

Por último el código LUA para poder trabajar con objetos y atributos de un objeto sería el siguiente.

```
local player=get_player()
-- Para llamar a un método del objeto player
player:set_speed(25)
-- Para acceder a la propiedad speed del objeto player
local player_speed=player.speed
```

```
-- Para crear una variable de tipo Vect3f
local point3=Vect3f(0, 0, 0)

-- Para establecer un valor en la propiedad x de la variable de tipo Vect3f
point3.x=5.0
```