LUNADLL - LUNALUA

Modifique SMBX a gusto de la manera más sencilla

Fundamentos
Ejemplos prácticos
Explicaciones claras
Paso por paso

Por Paredes Fernando alias hacheipe399

LunaDLL y LunaLua

Modifique SMBX a gusto de la manera más sencilla

Por Paredes Fernando Iván (hacheipe399)

Índice de contenido

LunaDLL

Super Mario Brothers X fue un gran juego y está aún en demanda. Tiene una comunidad enorme que crea episodios todo el tiempo y no se detiene. Esta demanda fue lo que motivó a Redigit (creador de SMBX) a añadir soporte para gráficos personalizados, para modificar uno mismo la apariencia de objetos como enemigos, decoraciones, bloques, etc. Con ello, las posibilidades de emular o recrear otros juegos de Nintendo dentro de SMBX fueron enormes. Los usuarios se conformaron con eso mientras el fangame seguía agregando cosas conforme se iba actualizando.

Sin embargo su desarrollo fue **detenido** en 2011 y ahora varios piensan que con la versión 1.3 no se puede hacer suficiente aun con la cantidad de cosas que posee por defecto como una gran cantidad de objetos o la posibilidad de jugar con la apariencia de los mismos.

Lo que más desean los usuarios del SMBX estándar es la adición de más objetos de la saga Mario Bros y un mejor sistema de eventos, porque con mostrar u ocultar capas, hablar o matar NPCs no se logra suficiente. Lo que más desean estos usuarios es más capacidad de personalización y de modificar el juego a gusto. Es por esto que el usuario kil3 en el año 2013 sacó a la luz un ejecutable de SMBX modificado para que cada cuadro se llame a un código C arbitrario que estaba almacenado en una librería llamada **lunadll.dll**, la cual rompió con las barreras del obsoleto SMBX estándar.

¿Qué es LunaDLL?

LunaDLL es una extensión para Super Mario Bros. X que consiste en una librería externa (archivo .dll) y una versión modificada del ejecutable para que la librería sea utilizada. En cada cuadro que corre en el juego se llama a la librería para poder ejecutar comandos que a su vez cambian cosas en el juego. LunaDLL dota al juego la capacidad de poseer niveles con funciones adicionales y más avanzadas, fruto de una programación previa utilizando archivos externos como lunadll.txt o lunadll.lua en el caso de LunaLUA (una variante de LunaDLL).

Los comportamientos personalizados son ejecutados a través de **comandos** preescritos en la librería cuyos parámetros modifican distintas cosas o valores del juego dependiendo del comando que se use. Para trabajar con LunaDLL se requiere un mínimo de comprensión de los temas que se van a detallar en este libro.

¿Cómo utilizo LunaDLL?

Para hacer lo que quieras en tus niveles, LunaDLL se vale del **auto código**, que es el lenguaje de programación del mismo. El auto código no es más que un conjunto de comandos seguidos de cinco parámetros y basado en “eventos” que pueden ser definidos por el usuario o eventos que ocurren por sección. El auto código debe escribirse en un archivo llamado **lunadll.txt** que va ubicado dentro de la carpeta del nivel. Por ej.: si su nivel se llama *Level 1-1.lvl*, su carpeta se tiene que llamar *Level 1-1*. Dentro de esa carpeta debe ir el archivo de texto. Si quiere que sus comandos se apliquen en todos los niveles, el archivo debe llamarse **lunaglobal.txt** y tiene que colocarse en la carpeta de los episodios.

Estructura del auto código

En lunadll.txt habrán sólo 4 cosas que va a escribir:

1. Símbolo de tiempo (#): Designa el momento en el que se va a realizar la acción. Corresponde generalmente al número de secciones:
    1. #-1: durante la carga del nivel.
    2. #0: Siempre (en todas las secciones)
    3. #1: sección 1. #2: Sección 2... #21: Sección 21.
    4. #1000+: eventos/bloques de comandos propios.
2. Comentario (//texto): Cualquier línea que contenga barras dobles será considerada como comentario. No hacen absolutamente nada en el código, sólo se utilizan con el fin de documentar el código, para que humanos sepan qué hace el mismo.
3. Comandos (Kill,0,0,0,0,0,0): Son lo más importante del auto código. Los comandos dicen al juego qué hay que hacer. No hay manera de recordarlos a todos, por ello es indispensable consultar la referencia. La mayoría tiene las mismas partes, separadas por comas. "Kill" mata algo, usualmente al jugador. Son la parte más fácil de recordar ya que son siempre palabras o abreviaciones en inglés. "InfiniteFlying" habilita el vuelo infinito y así suelen ser los nombres. Todos los comandos están acompañados de seis parámetros separados por comas, difíciles de recordar por lo que es esencial leer la referencia:
    - 1: El parámetro "objetivo": El primer número es un "objetivo" para el comando. Si se necesita apuntar al jugador, usualmente 0 es el jugador. Si se necesita apuntar a algún NPC (por ejemplo, la flor de fuego) se necesita apuntar al ID del NPC (14 en el caso de la flor).
    - 2, 3 y 4: Opciones: Son parámetros extra que pueden variar entre comandos. Si no se usa alguno, se coloca usualmente un 0. Véase la referencia para saber los comandos.
    - 5: Tiempo activo: En muchos casos, el parámetro 5 es el "tiempo activo". Básicamente es qué tan largo será el tiempo en que un comando se ejecute. Al finalizar éste tiempo, el comando se elimina. 0 significa siempre y 1 es 1 cuadro. 65 cuadros son 1 segundo. Acuérdese de que ese tiempo activo correrá cuando el jugador se encuentre en la sección correspondiente. Si el comando está en #21, éste aguardará hasta que el jugador entre en la sección 21, entonces allí el comando se ejecuta durante el tiempo establecido.
    - 6: Texto: Diferente al resto, el sexto parámetro puede escribirse como texto, pero cuando no se usa sólo se deja en 0. Usualmente se usa para escribir mensajes, números decimales u opciones de todo tipo.

4. Pie de Script (#END): Es una etiqueta especial que siempre se coloca al final del código, simple.

Ejemplos prácticos

Vamos a hacer un filtro básico. Un filtro fija el estado de un personaje:

```
#-1
FilterToBig,0,0,0,0,0,0
#END
```

Eso es todo. Primero tenemos un designador de sección, en este caso durante la carga del nivel que es donde normalmente queremos un filtro. Esto hace que Mario baje al nivel hongo si tiene algún powerup mejor y no hace nada si es pequeño.

El siguiente ejemplo pone comandos durante la carga, siempre y sección 2:

```
//Nivel completo
#-1
//Filtra de Mario a Link
FilterPlayer,0,1,5,0,0,0
#0
//Muestra la palabra "Hola" a x:300 y:300 con fuente 3
ShowText,0,300,300,3,0,Hola
#2
// Filtra varias cosas y reproduce el efecto 10 sin razón
SFX,10,0,0,0,0,0                    //Reproduce el sonido
FilterToSmall,0,0,0,0,1,0           //Hace pequeño al jugador
FilterReservePowerup,0,0,0,0,1,0    //Filtra el powerup de reserva
FilterMount,0,0,0,0,1,0             //Filtra la montura a Yoshi
#END
```

El siguiente ejemplo va a determinar cuántos golpes restantes tiene el jefe Mother Brain (ID 209, 10 golpes restantes por defecto):

```
#1
SetHits,209,1,9,0,1,0 //9 golpes asignados, 1 golpe restante.
#END
```

En SMBX los enemigos no tienen vida, tienen un contador de golpes. Lo que hace el comando es sumar una cantidad de golpes al contador. Si a Mother Brain (10 golpes) se le suma 9 al contador, le va a quedar 1 golpe restante, lo cual si se la daña, se muere. Un jefe Mother Brain con -10 golpes asignados, va a tener 20 golpes restantes para matarlo. Si hay más de uno en la sección, la cantidad se asigna a los contadores de todos los Mother Brains presentes.

El siguiente ejemplo muestra el texto de depuración en todo el nivel y activa un evento SMBX en la sección 3:

```
#0
DebugPrint,0,0,0,0,0,0
#3
TriggerSMBXEvent,0,0,0,0,1,TogglePlatforms
#END
```

El texto de depuración muestra algo de información sobre los comandos que se están ejecutando, es muy útil para saber si algo no funciona de la manera adecuada. El otro comando va a iniciar el evento "TogglePlatforms" si en el nivel hay un evento con ese nombre.

Memoria del juego

Como sabrá los valores de variables y cosas del juego se guardan en la memoria. Como no tenemos el código fuente de SMBX, no podemos agregar cosas nuevas por lo que necesitamos

manipular la memoria del juego para hacer cosas más avanzadas. Esto es lo más complicado que va a encontrar en LunaDLL: el manejo de memoria del juego.

El siguiente ejemplo va a multiplicar por 2 las vidas del jugador y si éste posee una flor de fuego, se activa el evento 1000:

```
#1
//Multiplica por 2 la cantidad decimal localizada en 0x00B2C5AC
MemAssign,0x00B2C5AC,2,3,0,1,f
//Activa evento 1000 si el número localizado en 0x112 es 3
OnPlayerMem,0x112,3,0,1000,0,w
#END
```

Algunos comandos verifican o realizan operaciones con valores alojados en la memoria del juego. Distintos comandos escanean y utilizan ciertas partes de la memoria dedicada a la información del jugador, de los NPCs o del juego completo.

Hay tres tipos de regiones en la memoria de SMBX:

- Memoria global: almacena vidas, monedas, estado de interruptores, puntaje, temporizadores de todo, punteros a nombres de niveles, directorios y muchas cosas más.
- Memoria del jugador: espacios donde se almacenan valores relacionados al comportamiento del jugador.
- Memoria NPC: espacios donde se almacenan valores relacionados al comportamiento de los NPCs.

Cada parte de la memoria tiene un **comando condicional** (que realiza comparaciones y establece condiciones con una dirección específica) y un **comando operacional** (que realiza operaciones con una dirección específica). La única excepción es la memoria NPC que no tiene un comando condicional.

	Memoria global	Memoria del jugador	Memoria NPC
Com. condicional	OnGlobalMem	OnPlayerMem	**No tiene.**
Com. operacional	MemAssign	PlayerMemSet	NPCMemSet

Desafortunadamente no hay manera de estar seguro de qué valor está en qué parte de la memoria, es algo que lo averiguan los usuarios manualmente; por lo tanto se sabe que en la región 0x00B2C5AC se guardan las vidas del jugador y se pueden realizar operaciones con ese valor utilizando MemAssign:

0: Asignar

1: Sumar

2: Restar

3: Multiplicar

4: Dividir

5: XOR (O exclusivo)

Además los valores no están almacenados de manera uniforme, entonces necesita saber el

tipo de valor con el que va a trabajar. En **todos** los comandos operacionales y condicionales el último parámetro es el tamaño del dato:

- b: byte (1 byte de memoria)
- w: word (2 bytes de memoria)
- dw: double word (4 bytes de memoria)
- f: float / decimal (4 bytes de memoria)
- df: double float / decimal de doble precisión (8 bytes de memoria)

1 byte de memoria no es muy utilizado en este juego. Valores de 2 bytes son bastante usados. Valores dword no son utilizados. Valores decimales son ampliamente usados para muchas cosas.

Eventos personalizados

La razón por la que LunaDLL fue creado es la limitación del editor de niveles y sus eventos. Pero por ahora podemos ejecutar comandos sólo al entrar en secciones y no cuando queremos. Para ejecutar eventos en cualquier momento, el autocódigo nos provee de **eventos personalizados**.

Los eventos personalizados se pueden activar usando comandos condicionales o temporizadores. Estos eventos se designan con un número de sección **a partir del 1000**. Si el número designado es 999 o menor, no será reconocido como evento:

```
#1000
SFX,10,0,0,0,0,0 //Reproduce efecto de sonido de ID 10
#END
```

Eso fue un ejemplo de bloque de código personalizado. Todavía falta algo que lo active. Hay varios comandos que los activan como:

- Trigger: sólo activa un evento personalizado.
- Timer: activa un evento luego del tiempo especificado.
- BlockTrigger: activa un evento cuando el objetivo toca cierto bloque.
- OnInput: activa un evento al presionar un control.
- OnCustomCheat: activa un evento cuando escribe algo.

El siguiente ejemplo mueve una capa hacia la derecha despues de tipear “mover”:

```
#0
//Cuando se escriba “mover”, se activa el evento 1000
OnCustomCheat,0,0,0,1000,0,mover
#1000
//Mueve la capa 3 a una velocidad de 2.5 (asumiendo que agregaste bloques ahí)
LayerXSpeed,3,0,0,0,0,2.5
#END
```

La función se explica por sí misma. Cuando escribe “mover” con el teclado, se empiezan a mover los bloques que pertenezcan a la capa 3.

Hay un gran problema aquí: si se escribe “mover” 500 veces, el comando se ejecuta 500 veces ya que el evento personalizado se copia dentro de la sección 0. Para evitar esto, es recomendable reducir el tiempo activo del comando a 1 para que se elimine pasado el cuadro.

El siguiente ejemplo reproduce un efecto de sonido al tomar el hacha de SMB1:

```
#0
// Activa #1000 luego de obtener el hacha (ID 178) sólo una vez
IfNPC,178,2,0,1000,0,once
#1000
// Reproduce efecto “item.wav” que se encuentra en la carpeta del nivel
SFX,0,0,0,0,1,item.wav
#END
```

IfNPC activa un evento si se cumple la condición en relación a un NPC específico. El 2 del segundo parámetro verifica que el NPC no exista más, por lo que ese comando activa el evento 1000 si la hacha ya no existe más.

El siguiente ejemplo muestra un bucle infinito entre dos eventos personalizados:

```
#0
// Sólo cambia al evento 1000
Trigger,0,1000,0,0,0,0
// Los enemigos se vuelven amigables y cambia al evento 1001
#1000
Timer,0,1001,1,0,200,0
NPCMemSet,-1,0x46,0xFFFF,0,200,short
ShowText,0,400,550,3,200,YAY!
// Todos los enemigos se vuelven agresivos y cambia al evento 1000
#1001
Timer,0,1000,1,0,200,0
NPCMemSet,-1,0x46,0x0000,0,200,short
ShowText,0,400,550,3,200,GRR!
#END
```

Empieza el evento amigable y luego de 200 cuadros cambia al evento agresivo, y éste cambia al evento amigable a los 200 cuadros otra vez. Es un ciclo infinito. Además usa el comando operacional de memoria *NPCMemSet* que manipula a los NPCs para que sean amigables o agresivos.

El temporizador es el comando *Timer*. Ese requiere el evento que va a activar, la indicación si mostrar el contador en pantalla o no, la indicación si repetir o no y el tiempo que tarda para activar el evento.

Modificar parámetros

Puede surgir la situación en la que necesitemos reutilizar un comando viejo pero con opciones diferentes por cualquier motivo. Un comando muy poderoso que nos otorga el autocódigo es *ModParam*. Éste modifica parámetros de comandos en todo el documento. Para saber exactamente a qué comando modificar, necesitamos que ese tenga en el parámetro Texto un nombre.

```
#0
```

```
//El comando que vamos a modificar
ShowText,0,300,700,3,0,SCROLL

//Modificar tercer parámetro restando 1, sección 0, siempre, SCROLL
ModParam,3,1,2,0,0,SCROLL
#END
```

Los parámetros 4 y 6 se usan para apuntar al comando deseado. El cuarto parámetro indica la sección donde se encuentra el comando y el sexto es el nombre del mismo que debe ser precisado en el sexto parámetro del comando a modificar ya que casi ningún comando utiliza texto. En el caso de *ShowText* que muestra en pantalla "SCROLL", ModParam va a modificarlo porque en el sexto parámetro hay texto que coincide con el "SCROLL" de este último.

Variables

Para que las funciones sean más eficaces necesitamos que el juego maneje valores que se puedan usar y modificar, para ello existen las **variables** que son espacios en la memoria donde se guardan valores. LunaDLL incluye varios comandos que manejan variables que permiten al autocódigo funcionar como un programa.

Existen comandos que asignan variables que las manipulan con las siguientes operaciones:

Parámetro	Operación
0	Asignar.
1	Sumar.
2	Restar.
3	Multiplicar.
4	Dividir.
5	XOR (O exclusivo)

Hay comandos que comparan valores de variables para poder realizar eventos condicionales, que siguen las siguientes comparaciones:

Parámetro	Comparación
0	Igual que. Funciona si ambos valores son iguales.
1	Mayor que. Funciona si el valor es mayor que otro.
2	Menor que. Funciona si el valor es menor que otro.
3	No igual. Funciona si los valores son distintos.

A continuación se van a describir cada uno de los comandos que manipulan variables...

Crear variables

SetVar manipula variables que el usuario defina. Las mismas se definen e inicializan con este comando si es que en el código no existe alguna con el nombre especificado.

```
#1
```

```
//SINTAXIS: SetVar,0,OPERACIÓN,VALOR,0,TIEMPO ACTIVO,NOMBRE DE LA VARIABLE
//Va a crear una variable "Lifes"
SetVar,0,0,0,0,1,Lifes

#1001
//Va a multiplicar "Lifes" por 7
SetVar,0,3,7,0,1,Lifes
#END
```

Hay muchos comandos que no tienen un parámetro para especificar el nombre de la variable a manipular, por lo que el autocódigo nos provee de **referencias**. Son cadenas de caracteres que comienzan con el signo dólar ($) que funcionan como nombres de variables. Se escriben antes del comando. Para crear la variable por este método se usa SetVar dejando en blanco el parámetro de texto, obviamente con la referencia al principio:

```
#-1
//Crea la variable "CurrentLifes" usando una referencia
$CurrentLifes,SetVar,0,0,0,0,2,0

#1003
//Multiplica "CurrentLifes" por 7
$CurrentLifes,SetVar,0,3,7,0,2,0

//Muestra "CurrentLifes" en la pantalla
$CurrentLifes,ShowVar,0,65,20,3,0,
#END
```

También

Load player/NPC/global var

Asignan a una variable un valor que se encuentre en alguna parte de la memoria del juego. Tienen casi los mismos parámetros que los comandos de memoria. Funciona sólo con referencias:

```
#1001
//LOADPLAYERVAR: CARGA UN VALOR DE LA MEMORIA DEL JUGADOR.
//SINTAXIS: $REFERENCIA,LoadPlayerVar,0,DIRECCION,OPERACION,0,TIEMPO,TAMAÑO
//EJEMPLO: Asigna la posición X del jugador (0xC0) a la variable Xpos
$Xpos,LoadPlayerVar,0,0xC0,0,0,0,df

#1002
//LOADNPCVAR: CARGA UN VALOR DE LA MEMORIA NPC.
//SINTAXIS: $REFERENCIA,LoadNPCVar,ID,DIRECCION,OPERACION,SECCION,TIEMPO,TAMAÑO
//EJEMPLO: Asigna la posición X (0x78) de los NPC-88 a la variable NpcX
$NpcX,LoadNPCVar,88,0x78,0,3,0,df

#1003
//LOADGLOBALVAR: CARGA UN VALOR DE LA MEMORIA GLOBAL.
//SINTAXIS: $REFERENCIA,LoadGlobalVar,DIRECCION,OPERACION,0,0,TIEMPO,TAMAÑO
//EJEMPLO: Asigna a CurrentLifes las vidas del jugador
$CurrentLifes,LoadGlobalVar,0x00B2C5AC,0,0,0,1,f

#END
```

Condicionales

Es posible realizar condiciones y acciones con comandos usando *IfVar* y *CompareVar*. El primero determina y activa un evento personalizado si se cumple una condición y el segundo compara dos variables a la vez y activa un evento si la condición es verdadera. Con estos comandos el autocódigo puede actuar como un programa dentro del juego. He aquí un ejemplo con la sintaxis de ambos comandos y eventos que se activan:

```
#0
//SINTAXIS DE IFVAR:
//IfVar,0,COMPARACION,VALOR,EVENTO,TIEMPO ACTIVO,NOMBRE DE VARIABLE
//Ejemplo: Si variable "BossesBeaten" es mayor que 7, activar #1000
IfVar,0,1,7,1000,0,BossesBeaten

//SINTAXIS DE COMPAREVAR:
//$VARIABLE1,CompareVar,0,COMPARACION,0,EVENTO,TIEMPO ACTIVO,VARIABLE 2
//EJEMPLO: Activar #1001 si StartCount es mayor que DeathCount
$StartCount,CompareVar,0,1,0,1001,0,DeathCount

#1000
//Multiplica por 2 la variable "CurrentLifes"
$CurrentLifes,SetVar,0,3,2,0,1,0

#1001
//Muestra "CurrentLifes" en pantalla como "LIFES: #"
$CurrentLifes,ShowVar,0,65,20,3,65,LIFES:
#END
```

Hay que aclarar que no hay una forma directa de escribir una segunda acción en caso de que las condiciones sean falsas. Para ello hay que utilizar más comandos con comparaciones distintas.

Guardar variables

Las variables en un nivel son permanentes mientras el juego se ejecute. LunaDLL nos permite **guardar** variables en archivos externos para ser utilizados en cualquier instancia de SMBX. El guardado se maneja a través de **bancos** que son los lugares donde se almacenan las variables de LunaDLL. Hay dos comados para ésto: *BankVar* y *WriteBank*. El primero guarda la variable especificada en el banco, pero no en el archivo; para ello está el segundo comando.

Guardar variables es muy útil si se quieren hacer condicionales entre cosas que ocurren entre dos niveles distintos. Las variables se guardan en el archivo *LunaSavedVars#.txt*. Los únicos parámetros que estos comandos tienen son el tiempo activo (5°) y el nombre de la variable (6° parámetro). Imagine que ésta pieza de código está escrita en el txt del nivel "Lower Norfair.lvl":

```
#2015
//Guarda en el banco la variable "RidleyBeaten".
BankVar,0,0,0,0,1,RidleyBeaten

#2016
//Guarda el banco de RidleyBeaten en el archivo externo.
WriteBank,0,0,0,0,1,RidleyBeaten
#END
```

En el mismo episodio hay otro nivel llamado "Crateria.lvl" que va a activar un evento si la

variable RidleyBeaten es igual a 1:

```
#1999
//Activa #2000 si RidleyBeaten es igual a 1
IfVar,0,0,1,2000,0,RidleyBeaten

#2000
//Activa evento "Elevador"
TriggerSMBXEvent,0,0,0,0,1,Elevador
#END
```

LunaLua

LunaDLL es bastante útil ya que con un simple comando podemos cambiar el comportamiento de un objeto o una parte del juego. Además hay comandos condicionales y hay soporte para variables.

Pero el autocódigo también es limitado ya que el comportamiento de los objetos depende de un comando y la combinación de eventos personalizados y condiciones varias no siempre es eficiente, no se pueden crear funciones o ciclos avanzados que actúen con absoluta precisión, sumado a que la manipulación de variables no es la mejor. Lo peor de todo es que los comandos son ilegibles por más documentado que esté el fichero.

Para trabajar con más comodidad y complejidad necesitamos programar al juego, no sólo ordenarle. El autocódigo no es un lenguaje de programación, por lo que a Kevsoft se le ocurrió y creó una brillante herramienta para SMBX: LunaLua. Con esto, LunaDll dejaría de ser un conjunto de líneas casi abstractas e incomprensibles, sino que se convertiría en una auténtica forma de programar cosas propias para SMBX utilizando el fácil lenguaje de programación Lua, cuya sintaxis es como una combinación entre BASIC y javascript lo que lo convierte en uno de los lenguajes más sencillos que existen.

¿Qué es LunaLUA?

Es una variante de LunaDLL que en lugar de auto código, utiliza el lenguaje de programación Lua. Con Lua se pueden hacer funciones propias utilizando variables, librerías externas y APIs, cosas que no se pueden realizar con simples comandos. A diferencia de LunaDLL, éste módulo utiliza archivos de LUA, como **lunadll.lua**, **lunaworld.lua**, etc. Actualmente LunaLUA también soporta auto código en archivos txt, por lo que es más conveniente tener la librería LunaLUA para poder trabajar con ambos lenguajes.

LunaLUA incluye SDL2_Mixer que es un motor de audio compatible con una enorme variedad de formatos y códecs de audio. Como sabrás, SMBX originalmente soportaba sólo MP3, WAV y MID con errores. SDL2_Mixer soporta MP3, WAV, MID, OGG, FLAC, MOD, IT, XX, S3M, etc. El SDL2_Mixer prefiere que el audio de SMBX esté en formato ogg, pero la descarga del módulo incluye todos los sonidos y música ya convertidos a ogg. En todo caso se puede usar algún convertidor de audio para convertirlos y resamplearlos a 44100 hz como Audacity o el SoX Audio Converter que viene incluído con PGE (editor alternativo al Level Editor de SMBX).

Fundamentos de Lua

Lua es un lenguaje bastante simple. Trabaja definiendo funciones que pueden ser llamadas en otras partes del código. También puede definir variables que permiten manejar valores.

Para comenzar hay que descargarse LunaLua y pegar todo el contenido en el directorio raíz de SMBX. Luego, en la carpeta del nivel hay que crear un archivo llamado lunadll.lua, no archivos de texto .txt ya que estamos trabajando con el lenguaje Lua. Una vez hecho esto, podemos comenzar a trabajar.

Es recomendable utilizar Notepad++, Programmer's Notepad o cualquier otro editor

destinado a la escritura de código que incluya o pueda incluir el lenguaje Lua para trabajar con una sintaxis resaltada para hacer el trabajo más cómodo.

Funciones y variables

Una pieza de código simple de Lua se ve como esto:

```
miVar = 0;

function miFuncion()
    miVar = miVar + 1; --Suma 1 a miVar.
end
```

Esto define una variable llamada “miVar” y la inicializa en 0. También define una función llamada “miFuncion” que suma 1 a *miVar*.

Ese texto que va luego de los dos guiones es un **comentario**. A diferencia del autocódigo, en Lua se pueden hacer comentarios **multilínea**, es decir, donde no haya necesidad de agregar los símbolos al principio de cada línea. En Lua los comentarios multilínea se abren y cierran con corchetes dobles. Los corchetes que abren necesitan de los guiones dobles:

```
--[[Ésta es nuestra primera función.
Primero define una variable llamada miVar cuyo valor al principio será de 0.
Luego definimos una función llamada miFuncion que va a sumar 1 a miVar cada
cuadro del juego.]]

miVar = 0;

function miFuncion()
    miVar = miVar + 1;
end
```

Ahora necesitamos **llamar** a miFuncion en alguna parte del código. LunaLua ya viene con un sistema de eventos que son funciones preescritas llamadas automáticamente por LunaLua si usted las define. Las más importantes son *onLoad* y *onLoop*. La función onLoad es llamada al cargarse el nivel y onLoop es llamada en cada cuadro que corre en el juego.

Ahora vamos a llamar a miFuncion todo el tiempo:

```
miVar = 0;

function onLoop()
    miFuncion();
end

function miFuncion()
    miVar = miVar + 1;
end
```

Llamar a una función es como tomar el cuerpo de esa función y ejecutarlo. Este ejemplo hace exactamente lo mismo que miFuncion:

```
miVar = 0;
```

```
function onLoop()
    miVar = miVar + 1;
end
```

Interactuando con SMBX

Si ejecuta SMBX va a ver que nada pasa, eso es porque no le dijimos a Lua que haga algo con esa variable. Está haciendo lo que le dijimos: sumar 1 todo el tiempo, pero no lo aplicamos en el juego. Ahora vamos a agregar una acción simple usando la función *Text.print* de LunaLua que va a mostrar texto en la pantalla:

```
miVar = 0;

function onLoop()
    miFuncion();
    Text.print(tostring(miVar), 0, 0);
end

function miFuncion()
    miVar = miVar + 1;
end
```

Si ejecuta el juego va a ver un contador incrementándose rápidamente en la esquina superior izquierda. Ahora vamos a hablar sobre la función Text.print:

Así como LunaLua proviene con eventos preescritos, también tiene **funciones** preprogramadas que permiten aplicar el código al juego. La función *Text.print()* muestra texto en la pantalla. Ésta necesita saber lo que va a mostrar, la fuente utilizada (3 es la fuente blanca de SMBX, también la usa cuando se deja ese parámetro en blanco) y las coordenadas bidimensionales para ubicar el texto (X, Y), en este caso lo ubica en la esquina superior izquierda. Dentro hay otra función llamada *tostring* preescrita en el lenguaje Lua, no en el módulo Luna que permite mostrar la variable en formato texto que Text.print pueda comprender.

Argumentos

Las funciones pueden contener cosas llamadas **argumentos** que no son más que piezas de datos que permiten a las funciones operar de manera más distinta y compleja. Por ahora miFuncion sólo suma 1 a miVar, pero vamos a hacer que se le pueda añadir cualquier número:

```
function miFuncion(sumando)
    miVar = miVar + sumando;
end
```

Ya añadimos un argumento que permite utilizar esta función en más situaciones. Sólo tenemos que hacer un cambio más para que funcione. Cuando llamemos a la función, necesitamos **incluir** el argumento. Ya definimos una variable temporal llamada *sumando* que sólo existe dentro de miFuncion, por lo que sólo se puede usar dentro de ella. Cuando llamemos a la función, necesitamos darle un valor a *sumando*. Eso lo podemos hacer colocando el valor entre los paréntesis. Los paréntesis quieren decir que se desea llamar a la función, con los argumentos dentro. Si la función no tiene argumentos (como en el primer ejemplo), solamente se colocan los paréntesis sin nada dentro. Ahora tenemos un programa con argumentos:

```
miVar = 0;

function onLoop()
    miFuncion(1);
    Text.print(tostring(miVar), 0, 0);
end

function miFuncion(sumando)
    miVar = miVar + sumando;
end
```

Esto debería funcionar como el ejemplo anterior. La ventaja de esto es que podemos cambiar el incremento sólo cambiando el argumento. Esto es muy útil si tenemos un código global o API con la función y queramos utilizar un valor específico para un nivel en particular. Veamos otro ejemplo:

```
miVar = 0;

function onLoop()
    miFuncion(10);
    Text.print(tostring(miVar), 0, 0);
end

function miFuncion(sumando)
    miVar = miVar + sumando;
end
```

Si ejecuta el juego verá que el contador se incrementa en múltiplos de 10 y lo único que cambiamos fue el argumento de miFuncion.

Podemos añadir los argumentos que queramos separados por comas. Vamos a hacer miFuncion un poco más compleja. Primero multiplica un número y luego le suma otro:

```
miVar = 0;

function onLoop()
    miFuncion(1, 2);
    Text.print(tostring(miVar), 0, 0);
end

function miFuncion(sumando, multiplicador)
    miVar = (miVar*multiplicador) + sumando;
end
```

Esto simplemente va a multiplicar a miVar por 2 y luego le va a sumar 1. Podemos cambiar esos valores incluyéndolos en la llamada a la función.

Condiciones

La clave para un buen código es algo que se llama **condición**. Ésta nos permite controlar el juego dependiendo de lo que pase. Ésta es su estructura:

```
if (condición) then
    acción 1;
else
    acción 2;
```

```
end
```

Las condiciones se basan en lógica booleana, que controla que algo sea **verdadero** (`true`) o **falso** (`false`). Por ejemplo: 1 es igual a 1, por lo tanto es verdadero. Pero 1 no es igual a 2, entonces es falso. El ejemplo de arriba quiere decir "Si la condición es verdadera, se hace la acción 1, de lo contrario se hace la acción 2".

Los bloques condicionales se abren con la palabra **if** y se cierran con **end**. Si se quiere especificar una segunda acción por si la condición es falsa, se escribe **else** luego de la primera acción. Pero si no queremos hacer una segunda acción, simplemente no se escribe:

```
if (condición) then
    acción;
end
```

En una condición se pueden usar comparadores matemáticos que se van a especificar a continuación:

`a < b`	Verdadero si **a** es menor que **b**. Falso si **a** es mayor que **b**.
`a > b`	Verdadero si **a** es mayor que **b**. Falso si **a** es menor que **b**.
`a == b`	Verdadero si **a** es igual a **b**. Falso si **a** es distinto que **b**. **DOBLE SIGNO IGUAL NO ES LO MISMO QUE UN SOLO SIGNO**. El doble signo igual se usa para **comparar** y el simple se usa para **asignar**.
`a ~= b`	Verdadero si **a** es distinto que **b**. Falso si **a** es igual a **b**.
`a <= b`	Verdadero si **a** es menor o igual que **b**. Falso si a es mayor que **b**.
`a >= b`	Verdadero si **a** es mayor o igual que **b**. Falso si a es menor que **b**.
`not a`	Verdadero si **a** es falso. Falso si **a** es verdadero.
`a and b`	Verdadero si **a** y **b** son verdaderos. Falso si uno o ambos son falsos.
`a or b`	Verdadero si uno o ambos son verdaderos. Falso si **ambos** son falsos.

Éstas expresiones se pueden combinar de múltiples maneras. Por ejemplo:

```
if ((a > b) and (b > c)) then
```

está perfecto. Bueno, ahora que ya sabemos hacer comparaciones, vamos a expandir nuestra función *miFuncion*.

Se habrá dado cuenta que la función incrementa un número realmente rápido, entonces vamos a decirle que lo haga hasta cierto punto. Para ello vamos a usar una condición:

```
miVar = 0;

function onLoop()
    miFuncion(1, 1, 100000);
    Text.print(tostring(miVar), 0, 0);
end

function miFuncion(sumando, multiplicador, limite)
    if (miVar < limite) then --Hace la acción si miVar no llega al límite
        miVar = (miVar*multiplicador) + sumando;
    end
```

```
end
```

Añadimos un argumento llamado *limite* y se modificó el multiplicador para que tenga el mismo efecto del principio, que sólo sume 1 cada cuadro. Cuando ejecute el juego verá que el contador incrementa hasta llegar a 100.000, eso es exactamente lo que le dijimos que haga. Ya que la condición determina si *miVar* es menor que *limite*, entonces la operación se hace cada vez que se cumpla la condición. Cuando *miVar* llega al número límite, la condición pasa a ser **falsa** (ya que deja de ser menor que *limite*) y se deja de realizar la operación.

Podemos usar condiciones para hacer la función aún mejor. Vamos a hacer que una vez llegado el límite, se realice una cuenta regresiva y así de manera infinita. Hasta ahora utilizamos valores numéricos pero podemos usar booleanos también y guardarlos en variables:

```
miVar = 0;
incrementar = true

function onLoop()
    miFuncion(1, 1, 100000);
    Text.print(tostring(miVar), 0, 0);
end

function miFuncion(sumando, multiplicador, limite)
    if(incrementar) then
        if (miVar < limite) then
            miVar = (miVar*multiplicador) + sumando;
        else
            incrementar = false
        end
    else
        if (miVar > 0) then
            miVar = (miVar - sumando) / multiplicador;
        else
            incrementar = true
        end
    end
end
```

Parece que es mucho código, pero en realidad no es muy complejo y además este tipo de estructuras es muy común. Creamos una variable booleana llamada *incrementar* que es verdadera cuando queremos que el número suba y falsa cuando queremos que baje. Por el hecho de ser una variable booleana no necesita operadores de comparación en la condición, ya que puede ser verdadera o falsa por sí misma, por lo tanto sólo colocamos el nombre de la variable.

Primero que nada dividimos la función en dos partes: una para incrementar y otra para decrementar:

```
if (incrementar) then
    --Acciones para incrementar número.
else
    --Acciones para decrementar número.
end
```

Ahora en cada una de esas secciones necesitamos hacer dos cosas: primero aumentar o disminuir el número de la variable y luego cambiar *incrementar* cuando estemos listos. Para eso dividimos los condicionales en dos partes una vez más usando **else**.

```
if (miVar < limit) then
    miVar = (miVar*multiplicador) + sumando;
else
    incrementar = false
end
```

Mientras nuestra variable esté bajo el límite, el número seguirá incrementándose. Una vez que deje de estar bajo el límite (cuando la cuenta se detenga), se fija *incrementar* a **falso**. Esto quiere decir que en el siguiente cuadro vamos a mirar hacia la otra sección del condicional, ya que al llegar al límite *incrementar* se fija en falso lo cual se tienen que hacer las acciones luego del **else** del condicional principal.

El decremento es similar al incremento:

```
if (miVar > 0) then
    miVar = (miVar - sumando) / multiplicador;
else
    incrementar = true
end
```

Hay pocos cambios. Primero necesitamos cambiar la condición. Para el decremento queremos que el número baje hasta cero, entonces el número debe disminuir mientras el valor sea mayor a cero. Luego utilizamos una operación matemática para achicar el valor: restamos el *sumando* a *miVar* y lo dividimos por *multiplicador*, básicamente operaciones inversas. Cuando el valor deja de ser mayor a cero, se fija *incrementar* en **verdadero** para que comience el aumento otra vez.

Si ejecuta el juego con este código, va a ver los números subir y bajar entre 0 y 100.000.

Alcance de las variables

En Lua las variables pueden tener distinto alcance. Cuando hace algo como esto:

```
miVar = 0;
```

está haciendo algo que se llama **variable global**. Esto quiere decir que en cualquier parte del código (¡incluso en otros archivos!) se puede modificar o acceder a ella. Es terrible ya que cualquier otra cosa puede modificar la variable sin su consentimiento, además puede sobreescribir la variable de otro archivo sin saberlo.

Es por ello que para evitarlo Lua incluye una palabra llamada `local` que restringe el alcance de una variable si se coloca detrás de una o de una definición de función. Esa palabra se usa para que podamos controlar de dónde podemos acceder a una variable. Si tomamos el ejemplo anterior y convertimos a todas las variables en locales, vamos a poder accederlas desde el mismo fichero y no de ningún otro lo cual es lo preferible:

```
miVar = 0;
incrementar = true

function onLoop()
    miFuncion(1, 1, 100000);
    Text.print(tostring(miVar), 0, 0);
end
```

```
function miFuncion(sumando, multiplicador, limite)
    if(incrementar) then
        if (miVar < limite) then
            miVar = (miVar*multiplicador) + sumando;
        else
            incrementar = false
        end
    else
        if (miVar > 0) then
            miVar = (miVar - sumando) / multiplicador;
        else
            incrementar = true
        end
    end
end
```

La palabra local no sólo restringe las variables al fichero, sino que las restringe donde sea que las declaremos. Por ejemplo: vamos a crear una variable para que sólo pueda ser utilizada dentro de la función *onLoop*:

```
function onLoop()
    miFuncion(1, 1, 100000);
    local varText = tostring(miVar);
    Text.print(varText, 0, 0);
end
```

Acá *varText* puede ser usada sólo **dentro** de *onLoop* en las líneas **luego** de haber sido declarada. Los siguientes ejemplos son erróneos ya que pretenden usar a *varText* **fuera** de su alcance:

```
function onLoad()
    local varText = tostring(miVar);
end

function onLoop()
    miFuncion(1, 1, 100000);
    Text.print(varText, 0, 0); --¡Debería estar dentro de onLoad()!
end
```

```
function onLoop()
    miFuncion(1, 1, 100000);
    Text.print(varText, 0, 0); --¡Debería estar luego de declarar varText!
    local varText = tostring(miVar);
end
```

Lo mejor es tener todas las variables locales para evitar saturar el código. Pero en el caso que necesitemos modificar un argumento de una función o una variable específica no hay que hacerlos locales.

Tablas

Lua tiene lo que se llaman **tablas** que son conjuntos de cosas, por decirlo de una manera simple. Se pueden usar para hacer listas de variables o incluso de funciones. Puede definir una tabla de ésta manera:

```
local miTabla = {}
```

Eso va a crear una tabla vacía, aunque también puede crear una con elementos dentro:

```
local miTabla = { 10, 15, 20, 30 }
```

Eso va a crear una tabla con cuatro elementos que pueden ser obtenidos en otra parte del código de la siguiente manera:

```
miTabla[1]
```

Eso va a obtener el primer elemento de la tabla que es 10 (en Lua las tablas comienzan en 1). Los números con los que se acceden a elementos son los **números de índice**.Puede tratar al elemento como una **variable**, puede leerlo así como asignarle valores:

```
miTabla[4] = 55;
```

Con esa línea se va a modificar la tabla, quedando:

```
{ 10, 15, 20, 55 }
```

Lua tiene un buen concepto de **orden**, ya que uno puede asignar a un elemento de la tabla un **nombre** para no recurrir a contar el número de índice. Si no se define un nombre, Lua identifica a los objetos con el número de índice:

```
local miTabla = { a = 10, b = 15, c = 20, d = 30 }
```

Entonces en lugar de acceder al elemento a través del número de índice, lo hacemos a través del nombre:

```
miTabla[c]
```

Eso le va a devolver el valor 20. También podemos añadir elementos nuevos a la tabla usando una identificación:

```
miTabla[e] = 55;
```

Eso va a añadir un elemento de valor 55 a la tabla con el nombre *e* que se puede obtener mediante:

```
miTabla[e]
```

Si el elemento tiene nombre, también se puede obtener mediante:

```
miTabla.e
```

Que es mucho más fácil de leer y manejar.

Los valores no pueden ser sólo numeros, sino cadenas de caracteres, verdadero o falso, otras tablas o incluso funciones.

LunaLua y SMBX

La clave para que algo hecho con LunaLua funcione es que interactúe con el juego, de lo contrario el programa sería inútil; por ello vamos a hablar de eso ahora. Ya aprendimos la función

Text.print() que muestra texto en pantalla, pero eso es útil en ciertos casos. Lua puede hacer un montón de cosas más.

Estructuras

LunaLua viene con algunas **estructuras** que permiten al programador acceder a datos de SMBX. Hay varios paquetes de datos que se pueden acceder con LunaLua que afectan al funcionamiento del juego. Algunos de ellos son las clases Player, NPC y Layer, que como sus nombres lo indican, permiten acceder a datos del jugador, de los NPCs y de las capas.

Primero veamos la estructura Player. Lo primero que tenemos que hacer es obtener una **referencia** al jugador. Esto nos da la estructura Player y nos deja aplicarla al jugador (también podemos hacerlo para el jugador 2, separado). Ahora usar la estructura es fácil ya que LunaLua viene con un valor preconstruído llamado *player* que nos deja acceder al jugador directamente.

Una vez que tengamos la referencia podemos acceder a los datos. Los datos que podemos acceder se encuentran en la lista de datos de la clase Player. Vamos a usar el campo speedX que nos deja obtener o fijar la velocidad horizontal del jugador. Este es un simple código que no deja al jugador moverse a la derecha o izquierda:

```
function onLoop()
    player.speedX = 0;
end
```

Y puede hacer cosas similares con otros datos de la clase. Se tratan como variables y las funciones son como las funciones que usted puede definir. Con eso en mente, vamos a mirar a una función más avanzada: *mem*. Ésta nos deja hacer lo que queremos ya que accede a la memoria de un objeto directamente. Particularmente no es fácil de usar pero ahora se va a explicar:

La función *mem* requiere un **espacio de memoria** y un **tipo** o **tamaño**. Estos nos permiten manipular el espacio correcto de la manera correcta. También podemos añadir un tercer parámetro que **escribe** un valor en ese espacio. Un ejemplo es este simple filtro:

```
function onLoad()
    player:mem(0xF0, FIELD_WORD, 1);
end
```

Eso forza al jugador a ser Mario cuando el nivel comienza. Es recomendable mirar el mapa de memoria de SMBX que enlista la cantidad de direcciones conocidas, el valor recomendable para trabajar y su función.

Ahora que hablamos sobre el jugador, vamos a movernos a los personajes no jugadores (**NPCs**). Una ventaja es que son exactamente lo mismo. Para hacer algo con los NPCs, primero necesitamos tener una referencia para acceder a los campos y funciones de la clase. También hay una función *mem* para los NPCs que trabaja con direcciones que hacen distintas cosas para ellos.

Hay una sola diferencia. Para el jugador sólo usamos la palabra *player* para tener una referencia ya que hay un número específico de jugadores, usualmente uno. El número de NPCs en un nivel puede cambiar dependiendo del nivel, entonces no podemos usar una palabra para obtener referencias.

Podemos usar una de estas dos funciones: *NPC.get()* y *NPC.get(id, sección)*. Éstos nos dan

tablas de NPCs, tanto del nivel completo o filtrado por un ID y una sección (puede escribir -1 como argumento para remover alguno de los filtros). Podemos usar un **ciclo** para ejecutar código en todos los NPCs de la tabla.

Vamos a hacer que todos los goombas del nivel (ID 1) sigan al jugador. Vamos a empezar a crear el ciclo:

```
function onLoop()
    for k,v in pairs(NPC.get(1, -1)) do
    --Algo de código aquí
    end
end
```

Ahora tenemos un ciclo. Dentro de este ciclo, tenemos dos variables temporales k y v. Éstas no pueden ser usadas fuera del ciclo. La variable k sostiene el nombre del elemento de la tabla que estamos ciclando y la variable v sostiene el valor en esa tabla. Usualmente estamos más interesados en v porque ahí es donde está el dato. Este va a ejecutar el código dentro del cicle una vez para cada elemento en la tabla (usando *NPC.get(1, -1)* quiere decir que estamos ejecutándolo para cada goomba del nivel). Ahora podemos escribir algo de código como si estuviésemos haciéndolo para un sólo NPC, ¡pero será aplicado para todos!

Usemos nuestras lecciones de condicionales para comprobar si el jugador está a la izquierda o derecha del NPC y luego fijar su velocidad apropiadamente:

```
function onLoop()
    for k,v in pairs(NPC.get(1, -1)) do
        if (v.x > player.x) then
            v.speedX = -1;
        else
            v.speedX = 1;
        end
    end
end
```

Ahora verá que cada goomba en el nivel va a caminar alrededor del jugador, más que en una simple línea. Puede ejecutar el código que quiera en estos ciclos, y son muy útiles a la hora de crear un comportamiento. Cabe aclarar que puede usar la **librería de IDs** en lugar de escribir los ID numéricos directamente, eso va a hacer su código más legible.

La clase Layer es la más simple de las tres. Ésta sólo permite acceder a las capas de SMBX y fijar algunas propiedades usando funciones de la clase. La única cosa que necesita es obtener una referencia. Para ésta, puede usar la función *Layer.get()*.

```
local miCapa = Layer.get("Capa 1");
```

Simplemente cree una capa en SMBX, luego utilice esa función para obtener una referencia a la capa basada en su nombre. Luego puede usar las funciones relevantes

Más eventos

Hablamos ya sobre los eventos *onLoad* y *onLoop* que son las más importantes, pero hay otras bastantes útiles que puede usar. Primero mire a *onLoadSection#*. Ésta permite ejecutar código cuando entre a una sección específica, que puede ser útil.

```
function onLoadSection0()
    player:mem(0xF0, FIELD_WORD, 1);
end
```

Esto forza al jugador a ser Mario, pero sólo cuando entra a la sección 1. Si comienza en la sección 2 y luego entra a la sección 1, el jugador cambia a Mario pero no volverá a ser el personaje anterior a menos que se lo diga.

En LunaLua los números de sección están entre 0 y 20 en lugar de 1 y 21. Esto significa que *onLoadSection0* se refiere a la sección 1, *onLoadSection1* se refiere a la sección 2 y *onLoadSection20* se refiere a la sección 21, etc.

Otro evento muy útil es *onKeyDown(keycode)*. Este permite ejecutar código cuando se presiona una tecla específica. Vamos a cambiar el personaje cuando presionemos abajo. Podemos hacerlo usando este evento de esta manera:

```
function onKeyDown(keycode)
    if (keycode == KEY_DOWN) then
        player:mem = (0xF0, FIELD_WORD, 1);
    end
end
```

Esto va a forzar al jugador a ser Mario cuando éste presione el control abajo. Una cosa interesante es que este evento requiere un argumento. Ya dijimos que son como variables temporales. En este caso, el argumento *keycode* contendrá el control presionado cuando LunaLua llame a esta función.

También podemos hacer uso de algunas **constantes** que nos permiten referirnos a los controles a través de nombres en lugar de números.

Ejemplos prácticos

Para empezar, mire hacia la función *onLoad*. Es un evento que puede poner en su archivo Lua que se ejecuta cuando un nivel se está cargando.

```
function onLoad()
   --Algo de código acá
end
```

Ahora vamos a hacer que el jugador sea forzado a ser “grande”. Para ello vamos a usar el campo *player.powerup* para esto. Una de las ventajas de LunaLua es que muchos de los valores que vamos a necesitar están guardados como constantes. Esto significa en que no hay que preocuparse demasiado por cuál es el número, podemos obtener el powerup por su nombre:

```
function onLoad()
     player.powerup = PLAYER_BIG;
end
```

Eso va a forzar al jugador a ser grande cuando el nivel comience. Funciona, pero si el jugador entra con una flor de fuego, lo vamos a forzar a ser sólo “grande”, entonces éste va a perder su poder. Para que eso no pase, tenemos que fijarnos si el jugador no tiene nada “mejor” al iniciar antes de forzar el estado:

```
function onLoad()
    if (player.powerup == PLAYER_SMALL) then
        player.powerup = PLAYER_BIG;
    end
end
```

Ahora el jugador va a ser grande si al iniciar el nivel era pequeño, así no se va a perder ningún powerup ganado en niveles anteriores. También puede hacer lo mismo para otros powerups. Este código le va a otorgar una flor de fuego si al comenzar el nivel es pequeño o sólo grande:

```
function onLoad()
    if (player.powerup == PLAYER_SMALL or player.powerup == PLAYER_BIG) then
        player.powerup = PLAYER_FIREFLOWER;
    end
end
```

Filtros de personaje

Esto es lo mismo que los filtros de poderes, solo que tenemos que usar la función *mem* para esto. Esta nos permite cambiar muchas cosas del jugador pero debe ser usada con cuidado, sino obtendremos resultados indeseados. Ahora vamos a usarla de una manera bastante segura:

```
function onLoad()
    player:mem(0xF0, FIELD_WORD, 1)
end
```

Este código va a hacer que el jugador sea Mario cuando el nivel comience. *0xF0* es la dirección de la memoria que vamos a cambiar, mirando hacia un valor *Word* (2 bytes). El último argumento es el valor que queremos asignarle, que es el número de índice de Mario. Acá están los números para todos los personajes:

1	Mario
2	Luigi
3	Peach
4	Toad
5	Link

Otros valores no deben ser usados, ya que son inseguros.

Ahora podemos hacer otras pruebas cambiando los personajes dependiendo de algunos factores. Ahora el personaje va a ser Mario si en el juego elegimos Mario, Peach o Toad; y va a ser Luigi si elegimos Luigi o Link.

```
function onLoad()
    local character = player:mem(0xF0, FIELD_WORD);
    if (character == 1 or character == 3 or character == 4) then
        player:mem(0xF0, FIELD_WORD, 1);
    else
        player:mem(0xF0, FIELD_WORD, 2);
    end
end
```

Acá se usa la otra forma de la función *mem*. Al tener sólo dos argumentos, *mem* asigna el

valor de ese espacio de memoria a la variable que queramos.

Sobreescribir o bloquear controles

Vamos a bloquear el salto secundario del jugador. Primero necesitamos una instancia de la clase Player. Desde acá, anulamos la función *onInputUpdate()* y controlar el valor *Player.altJumpKeyPressing.* Internamente, este campo es actualizado **siempre que** el control Correr sea presionado. No importa la tecla asignada o si se juega desde teclado o control, podemos asegurarnos de que este valor es constante y no necesitará controlar múltiples teclas. Como es un campo, podemos forzarlo a **falso** y como la función es en ciclo, se va a fijar continuamente en falso, por lo que el jugador no tendrá salto secundario.

```
function onInputUpdate()
   if(player.altJumpKeyPressing) then
       player.altJumpKeyPressing = false;
   end
end
```

Si queremos hacerlo para el segundo jugador, podemos añadir a la función el mismo código pero con la referencia ***player2***. Para realizarlo primero vamos a comprobar que éste exista en el juego:

```
if (player2) then
  if(player2.altJumpKeyPressing) then
     player2.altJumpKeyPressing = false;
  end
end
```

Control mejorado

Este método va a hacer que el jugador no pueda saltar usando el salto secundario y desmontará botas o yoshis. Tal vez esto sea bueno para un jugador, pero no para otros. Afortunadamente, con un poco de comprobación de memoria y algo más de código podemos solucionarlo.

La dirección *0x108* del jugador contiene información de montura. 0 significa que el jugador no monta nada, por lo que cualquier cosa mayor a 0 indica que el jugador está montando algo. En adición, podemos fijar *Player.jumpKeyPressing* a verdadero para que el jugador salte normalmente. Nuestro código mejorado va a lucir como esto:

```
function onInputUpdate()
    if(player.altJumpKeyPressing) and (tonumber(player:mem(0x108, FIELD_WORD) <
1) then
        player.altJumpKeyPressing = false;
        player.jumpKeyPressing = false;
    end
    if(player2)then
        if(player2.altJumpKeyPressing) and (tonumber(player2:mem(0x108,
FIELD_WORD) < 1) then
            player2.altJumpKeyPressing = false;
            player2.jumpKeyPressing = false;
        end
    end
end
```

Música y sonidos

Reproduciendo sonidos predeterminados

Puede reproducir los sonidos a través de sus IDs usando la siguiente función:

```
playSFX(23);
```

Este ejemplo va a reproducir un sonido con el ID 23. La lista completa se encuentra acá.

Reproduciendo sonidos propios

Puede reproducir sonidos propios que se encuentren en la carpeta de su nivel:

```
Audio.playSFX("mi ultra-mega cool sfx.ogg");
```

Cuando llame a la función, el archivo "mi ultra-mega cool sfx.ogg" va a sonar. Puede usar cualquiera de los formatos soportados.

También puede tener un uso más avanzado de sonidos si va a usar punteros Mix_Chunk:

```
--Crea un puntero global para el trozo de audio
local locoBell = Audio.newMix_Chunk()

function locoBellSound()
        --Load the sound chunk into buffer and get pointer to them
        locoBell = Audio.SfxOpen("loco bell.ogg")
        --Set the volume of 1-st channel
        Audio.SfxVolume(1, 75)
        --Play the loco bell sound trice
        Audio.SfxPlayCh(1, locoBell, 2)
        --1er ARGUMENTO: es un número de canal de mezcla que se puede utilizar
para reproducir sonidos en paralelo
        --Si Este argumento será igual a -1, se utilizará cualquier canal libre
que será devuelto por esta función.
        --2do ARGUMENTO: es un puntero que abre un trozo de música con la
función SfxOpen()
        --3er ARGUMENTO: es una serie de bucles (0 - reproducir una vez, 1 -
dos veces, -1 - bucle infinito).
        --Para poder romper el bucle tendrá que guardar el número de canales en
cualquier variable
end
```

Este ejemplo va a reproducir "loco bell.ogg" tres veces cuando esta funcion se llame.

Reproduciendo loops propios

Este es un ejemplo de uso de loops en un sonido:

```
local planeSnd = Audio.newMix_Chunk()

-- Reproduce un el sonido de un avión con una entrada de 1500 ms
function HeyHeyHereIsAPlane()
        --Fija el volumen máximo para el primer canal
        Audio.SfxVolume(1, 128)
        --Comienza el bucle infinito del avión en el primer canal con una
entrada de 1.5 segundos
```

```
        Audio.SfxFadeInCh(1, planeSnd, -1, 1500)
end

--Decrementa el bucle en 2000 ms
function PlaneFlyOut()
        --Detiene el bucle y decrementa en 2 segundos.
        Audio.SfxFadeOut(1, 2000);
end

--Temporizador del sonido
planeTime = 0

function onLoad()
        --Abre el sonido y almacena el puntero en una variable especial
        planeSnd = Audio.SfxOpen("plane.ogg")
        --Fija el temporizador
        planeTime = 10*65;
        --Ejecuta el sonido
        HeyHeyHereIsAPlane()
end

function onLoop()
        --Si el temporizador está en uso
        if(planeTime~=0) then
                --Decrementar el tiempo
                planeTime = planeTime-1
                --Si el tiempo termina, finalizar el loop
                if(planeTime==0) then
                        PlaneFlyOut()
                end
        end
end
```

Este es un pequeño ejemplo de un sonido de avión que vuela y se va luego de 10 segundos.

Música: manera simple

El método más simple es usando música de otras secciones:

```
playMusic(4)
```

Esta función va a reproducir la música perteneciente a la quinta sección.

Manera avanzada

Tiene la habilidad de reproducir cualquier música del directorio de su nivel si usa funciones especiales:

```
--Carga un archivo de música en el código
Audio.MusicOpen("music file.ogg")
--Reproduce el archivo previamente cargado
Audio.MusicPlay()
```

Nota: para hacer funcionar músicas propias en Lua, necesita que la sección donde va a reproducirla tenga por defecto “Nada” en las **opciones de nivel** o en su lugar fije cualquier música e inserte en el evento onLoad() una llamada a la siguiente función:

```
Audio.SeizeStream(0)
```

Este ejemplo agarra la música de la primera sección. Cuando su personaje esté en la sección, va a poder usar correctamente musicOpen y otras funciones más.

Nota: No olvide poner en la sección cualquier música predeterminada para confundir al motor y hacerle pensar que la música está sonando. De esta manera se previenen bugs cuando se cambien entre otras aplicaciones.

Ejemplo: Lista de reproducción

Este es un ejemplo de uso de múltiples archivos de música en una sección:

```
-- Temporizador
TickTack = 0
-- Canción
CurTrack = 1

-- Lista
Tracks = {
            "i_bach_joke.it",
            "schwing.mod",
            "Double_Cherry_Pass.it",
            "ice_ow.mid",
            "AliBaba_4Cumbia.mp3",
            "the_reincarnation_of_yammah.s3m",
            "backtoth.mod",
            "8bitenized.xm",
            "monkey_island_v1.xm",
            "D_MESSAG.it"
        }

function switchMusic()
        -- Si el tiempo se va
        if(TickTack<=0)then
                CurTrack = math.random(table.getn(Tracks))
                -- Abrir canción y añadirla en cola
                Audio.MusicOpen(Tracks[CurTrack])
                -- Fijar el tiempo en 20 segundos
                TickTack = 20*65
                -- Volumen máximo
                Audio.MusicVolume(128)
                -- Comienza la reproducción en un incremento de 1.5 segundos
                Audio.MusicPlayFadeIn(1500)
        else
                -- Decrementa tiempo
                TickTack=TickTack-1
        end

        if(TickTack==130)then
                -- Disminuye la música en 2 segundos
                Audio.MusicStopFadeOut(2000)
        end
        -- Muestra el nombre de archivo del tema actual
        Text.print(Tracks[CurTrack], 10, 10);
end

function resumeMusic()
```

```
        Audio.MusicOpen(Tracks[CurTrack])
        Audio.MusicVolume(127)
        Audio.MusicPlay()
end

function onLoad()
        --Agarra la cola de la segunda sección
        Audio.SeizeStream(1)
end

-- Reproduce todas las músicas propias en la segunda sección
function onLoadSection1(playerIndex)
        if(playerIndex==1) then
                -- Open current track to overwrite previous music file
                resumeMusic()
        end
end

-- Hace el ciclo en la segunda sección
function onLoopSection1(playerIndex)
        if(playerIndex==1) then
                switchMusic()
        end
end
```

Ese fue un ejemplo de un reproductor de música para la segunda sección que cada 20 segundos reproduce un nuevo tema de la lista, una tabla que contiene la lista de archivos de la carpeta del nivel.

NPCs y ciclos

Clase NPC

La clase NPC contiene varios métodos para modificar valores NPCs durante la ejecución del juego. No hay una manera real de obtener un sólo NPC, entonces necesita obtener la tabla del tipo de NPCs en el nivel. Una tabla es lo mismo que un array en otros lenguajes como C++, que es un contenedor de múltiples objetos accesibles a través de un índice.

Para obtener esto, usamos el método **NPC.get()** que tiene dos parámetros: el ID del NPC y la sección. Con el argumento -1 se puede anular uno de los filtros (todos los NPCs si en el primero o todo el nivel en el segundo).

Por ejemplo:

```
function onLoop()
    tableOfGoombas = NPC.get(1, -1);
end
```

Este código asigna la variable *tableOfGoombas* a la tabla de todos los goombas del nivel. Esto sabrá si está familiarizado con las tablas en Lua. Para más información, lea el capítulo Fundamentos de Lua para saber cómo funcionan las tablas. Si ya está familiarizado con arrays de otros lenguajes de programación, hay una diferencia que necesita saber: las tablas en lua están **indizadas desde 1, no desde 0**. Entonces, en lugar de tipear *tableOfGoombas[0]*, para obtener la primer instancia de Goomba necesita usar 1 en su lugar.

Utilizando la clase apropiadamente

Es importante usar la clase **sólo desde un ciclo**. No guarde cualquier valor NPC entre código ya que puede causar errores, siempre acceda a una tabla fresca de NPCs desde sus funciones *onLoop()* u *onLoopSection#()*.

Los NPCs están cambiando constantemente en SMBX. En un cuadro un NPC puede existir pero en otro no. No podemos predecir eso, entonces necesitamos actualizar la tabla utilizando ciclos únicamente.

Una cosa inteligente es controlar que hay objetos en la tabla, de no haber objetos, habrán errores en el código. La manera más simple de hacerlo es con una sentencia `if` y la función `table.getn()`. Esa función toma un argumento de una tabla y retorna la cantidad de objetos en ella. Entonces, si nuestra tabla *tableOfGoombas* tiene 3 goombas, la función va a retornar 3. Podemos usar ese valor para controlar que hayan objetos:

```
if (table.getn(tableOfGoombas) > 0) then
--código acá
end
```

Ciclos For

Los ciclos `for` son la manera más fácil de acceder a valores de una tabla. Cuando cambiamos algo de los NPCs, sólo asignándolo a 1 va a causar inconsistencias. Usando un ciclo nos aseguramos de que la tabla entera es modificada.

Los ciclos ejecutan series de sentencias basadas en pasos. Recuerde cómo las tablas están indizadas y puede acceder a ellas via *tabla[int]*. Esto hace a los ciclos la manera lógica de modificar múltiples NPCs. Para un ejemplo simple, vamos a matar a todos los goombas de un nivel:

```
function onLoop()
    tableOfGoombas = NPC.get(1, -1);
    if(table.getn(tableOfGoombas) > 0) then
        for i=1,table.getn(tableOfGoombas) do
            tableOfGoombas[i]:kill();
        end
    end
end
```

Esto va a iterar a través de 1 a *table.getn(tableOfGoombas)*, fijando *i* al paso actual. Entonces, ¿por qué podemos acceder al NPC usando tableOfGoombas[i]? Usando eso nos aseguramos no sólo de que todos los goombas sean matados, sino de que no estemos accediendo a cualquier valor fuera de rango. Entonces, como añadimos seguridad, sólo ejecutamos el ciclo si hay más de 0 objetos en la tabla. Haciendo esto, prevenimos errores. También, porque la tabla está constantemente actualizada estando en *onLoop()*, podemos asegurar de que cualquier Goomba que esté apareciendo también será matado instantáneamente.

Ciclos For each

Como Lua no tiene un ciclo for each, podemos emular uno cambiando algunos parámetros. Una tabla contiene dos valores: un índice y un objetos. Como en el ejemplo, una tabla de 3 goombas va a verse como esto:

Índice	Valor
1	Objeto Goomba
2	Objeto Goomba
3	Objeto Goomba

Los ciclos for toman dos argumentos: un mínimo y un máximo. Puede cambiar esos valores para representar índice-objeto dentro de la tabla, usando la palabra `in` en el ciclo. Por ejemplo:

```
for _,i in pairs(tableOfGoombas) do
    i:kill();
end
```

Este es exactamente el mismo código de arriba (menos la sentencia de seguridad), la diferencia es que *i* es la actual instancia del índice en la tabla. _ es en esta instancia del número de índice de la tabla. Le pusimos de nombre _ porque no nos interesa ahora el número. Podemos usar la función **pairs** para retornar la tabla representada como pares. El par en este caso es índice, objeto.

Comportamiento personalizado

Como sabrán, Birdo es el dinosaurio rosado de SMB2 que lanza huevos de la boca. Está incluído en SMBX, pero no sus variantes como lo es el Birdo Rojo que alterna entre huevos y bolas de fuego aleatoriamente. En varios episodios y niveles los diseñadores reemplazaban los gráficos de huevos y Birdo y ajustaban usando txts para simular un Birdo Rojo, pero no era exactamente igual al dinosaurio rojo genuino de SMB2. Con LunaLua hacer esto es más que sencillo. Vamos a ver qué tenemos que hacer:

1. Generar un número aleatorio.
2. Controlar que si da un número específico, lanzar o no bolas de fuego.

Podemos hacerlo utilizando el campo *id* de la clase NPC. Ese valor hace todo el trabajo ya que con él se puede cambiar de NPC cuando queramos. Los pasos a seguir van a ser éstos:

1. Obtener la tabla de Birdos en el nivel y de sus huevos (ya que éste NPC lo vamos a cambiar dependiendo del rango).
2. Controlar que ambos elementos existan (más de 0).
3. Generar un número aleatorio.
4. Si el número es igual a un valor de nuestro gusto (en este caso va a ser 2), cambiar el ID del huevo a una bola de fuego propia.

Nota: en las versiones de LunaLua anteriores a 0.7.1.0 el valor *id* es de **sólo lectura**. Esto quiere decir que sólo se puede obtener el valor, no modificarlo. Sin embargo, éste valor puede ser cambiado usando *NPC:mem()*. Entonces en vez de usar:
tableOfBirdoEggs[table.getn(tableOfBirdoEggs)].id = 282,

Podemos usar:

tableOfBirdoEggs[table.getn(tableOfBirdoEggs)]:mem(E2, FIELD_WORD, 282);

Funcionalmente son lo mismo, pero usar *id* en el código es conveniente, así que es recomendable

usar la última versión posible.

El código va a ser algo así:

```
local hasGenerated = false;
local ran;

function onLoop()
  tableOfBirdo = NPC.get(39, -1);
  tableOfBirdoEggs = NPC.get(40, -1);
  if(tableOfBirdo[1] ~= nil) then
    if(tonumber(tableOfBirdo[1]:mem(0xF0, FIELD_DFLOAT)) == 1) then
      if(hasGenerated ~= true) then
        ran = math.random(0, 2); --Nota: math.random(0, 2) genera un número
aleatorio entre 0 y 2.
        hasGenerated = true;
      end
      if(ran == 2) then
        if(table.getn(tableOfBirdoEggs) > 0) then
          tableOfBirdoEggs[table.getn(tableOfBirdoEggs)].id = 282;
          playSFX(42); --Efecto de sonido 42: gran bola de fuego
        end
      end
    end
    if(tonumber(tableOfBirdo[1]:mem(0xF8, FIELD_DFLOAT)) == 280) then
      hasGenerated = false; --Reinicia todo
    end
  end
end
```

Ahora se va a explicar qué hay en el código. Primero tenemos dos variables: *ran* es nuestro número aleatorio generado por la función *math.random()* y hasGenerated es un seguro para que no se generen valores aleatorios todo el tiempo. Esa es una de las desventajas de tener código en un ciclo constante.

Después obtenemos las tablas de Birdo y sus huevos. Comprobamos que el Birdo no es **nulo** y luego hacemos la magia. En este ejemplo usamos NPC.mem() para acceder a la memoria de SMBX. *0xF0* es la dirección del timer de Birdo. Cuando está en 1, Birdo está disparando. Ese es un buen momento para comprobar si el huevo es lanzado, lo cual hacemos junto al número generado aleatoriamente.

Si ese número es igual a 2, cambiamos el **último** (de ahí accedemos con *table.getn()*, esto obtendrá el valor máximo que es siempre la última adición a la tabla) huevo de Birdo a una bola de fuego cuyo ID es 282, reproduciendo al mismo tiempo el efecto de sonido 42 que es el sonido de esa bola de fuego. Finalmente, comprobamos *0xF8* que significa que Birdo se reinicia por lo tanto miramos por otro huevo.

Le doy un ejercicio: haga algunas mejoras para éste Birdo:

1. Un rango mayor de números aleatorios, o un sistema más aleatorio para lanzar bolas de fuego.
2. Use un ciclo para iterar a través de todos los Birdos rojos del nivel. Usando éste método, sólo uno de ellos va a tener este comportamiento.

Crear una API propia

Una **API** es un conjunto de funciones y métodos ofrecidos por bibliotecas para ser utilizados por otros archivos. En LunaLua, las bibliotecas son archivos .lua ubicados en el directorio *LuaScriptsLib*, para ser utilizados por archivos lunadll.lua o lunaworld.lua de los niveles y episodios. Esto es muy útil cuando necesitamos utilizar funciones complejas para un conjunto de niveles pero no deseamos o no podemos escribir tanto código para realizar ciertas tareas. Un claro ejemplo de esto es la API *CinematX* hecha por rockythechao que permite agregar secuencias cinematográficas a SMBX. El funcionamiento de éstas secuencias es muy compleja y además requiere de ciertas particularidades que varían según el nivel, por ello la API provee de funciones que pueden ser utilizadas para cada nivel o secuencia en particular.

Otro claro ejemplo es *smb3overhaul* que implementa la interfaz del SMB3 a SMBX. Sería muy tedioso escribir la interfaz en todos los niveles del episodio o en todos los episodios del juego, por lo que la API se puede llamar con algunas funciones desde cualquier archivo.

En LunaLua, una API no es más que una **tabla** cuyo contenido son **funciones**, por lo que para escribir una API requerimos de la definición de una tabla y una sentencia que la retorne. En este ejemplo, vamos a crear un contador de cuadros:

```
local contadorCuadros = { }

return contadorCuadros
```

Cuando loadAPI es llamado, el archivo va a retornar la tabla y la guarda, por lo que ahora tenemos la habilidad de enfocarnos en las funciones.

Ahora vamos a crear una función que cuente los cuadros y una variable que almacene esos cuadros:

```
local contadorCuadros = { }
local cuadros = 0;

function contadorCuadros.contarCuadros()
    cuadros = cuadros + 1;
end

return contadorCuadros
```

Para contar los cuadros necesitamos enfocarnos en el evento onLoop(). Para hacerlo necesitamos registrar el evento en nuestra API. El mejor momento para hacerlo es cuando la API se inicializa usando *onInitAPI*.

```
local contadorCuadros = { }
local cuadros = 0;

function contadorCuadros.contarCuadros()
    cuadros = cuadros + 1;
end

function contadorCuadros.onInitAPI()
    registerEvent(contadorCuadros, "onLoop", "contarCuadros"); --Llama a
contarCuadros() cuando onLoop() se ejecuta
end
```

```
return contadorCuadros
```

Lo último que se hace es crear funciones que sean utilizados por archivos externos. En este ejemplo vamos a crear una función que devuelva el cuadro actual:

```
local contadorCuadros = { }
local cuadros = 0;

function contadorCuadros.contarCuadros()
    cuadros = cuadros + 1;
end

function contadorCuadros.onInitAPI()
    registerEvent(contadorCuadros, "onLoop", "contarCuadros");
end

function contadorCuadros.cuadroActual()
    return cuadros; --Devuelve el cuadro actual
end

return contadorCuadros
```

La API ya está terminada. Ahora hay que usarlo en un archivo externo de la siguiente manera:

```
contadorAPI = loadAPI("contador")

function onLoop()
    Text.print("Cuadro actual: "..tostring(contadorAPI.cuadroActual()), 1, 1);
end
```

Para accederla mediante otros archivos, es importante que en el argumento de *loadAPI* se escriba el nombre de archivo de la librería. En este caso, el nombre de archivo de esta API es *contador.lua*.

Fuentes consultadas:

Kil3 (2013). Tutorial de LunaDLL. http://engine.wohlnet.ru/docs/Collected/LunaDLL/Tutorial.txt

Kil3 (2013). Referencia de comandos de LunaDLL: http://engine.wohlnet.ru/docs/Collected/LunaDLL/lunaref.txt

Kevsoft y Wohlstand (2014/2015). Wiki de Platformer Game Engine: LunaLua API page. http://engine.wohlnet.ru/pgewiki/Category:LunaLua_API