Luis Hernández López

Manual básico para aprendizaje en la creación de videojuegos con Game Maker: Studio (Actualizado 12-ago-2019)

Game Maker: Studio

Manual básico

ÍNDICE DE CONTENIDOS

[1 Introducción a Game Maker: Studio 1](#_Toc16535352)

[2 El lenguaje GML 3](#_Toc16535353)

[2.1 Literales y variables 3](#_Toc16535354)

[2.2 Macros 4](#_Toc16535355)

[2.3 Operadores 6](#_Toc16535356)

[2.4 Expresiones 7](#_Toc16535357)

[2.5 Sentencias condicionales: if y switch 8](#_Toc16535358)

[2.6 Sentencias iterativas: do, while, repeat y for 9](#_Toc16535359)

[2.7 Scripts 11](#_Toc16535360)

[2.8 Funciones útiles 13](#_Toc16535361)

[2.9 Objetos e instancias 14](#_Toc16535362)

[2.10 Sprites y sus propiedades 16](#_Toc16535363)

[2.11 Evento Step 17](#_Toc16535364)

[2.12 Evento Draw 18](#_Toc16535365)

[2.13 Evento Draw GUI 18](#_Toc16535366)

[2.14 Inputs por teclado y ratón 21](#_Toc16535367)

[2.15 Rooms 22](#_Toc16535368)

[2.16 Alarmas 24](#_Toc16535369)

[2.17 Sonidos 25](#_Toc16535370)

[2.18 Arrays 1D 25](#_Toc16535371)

[2.19 Array 2D 26](#_Toc16535372)

[3 Consejos para crear un juego de plataformas 28](#_Toc16535373)

[3.1 Preparar recursos básicos 28](#_Toc16535374)

[3.2 El escenario 28](#_Toc16535375)

[3.3 Gravedad y colisiones verticales 28](#_Toc16535376)

[3.4 Movimiento horizontal 29](#_Toc16535377)

[3.5 Reinicio de room 29](#_Toc16535378)

[3.6 Colisiones horizontales 30](#_Toc16535379)

[3.7 Saltos 30](#_Toc16535380)

[3.8 Obstáculos 30](#_Toc16535381)

[3.9 Enemigos que se mueven y chocan 31](#_Toc16535382)

[3.10 Enemigos que te matan al tocarlos 32](#_Toc16535383)

[3.11 Objeto meta 32](#_Toc16535384)

[3.12 Utilidades 32](#_Toc16535385)

# Introducción a Game Maker: Studio

En esta sección se tratará fundamentalmente cómo funciona un videojuego.

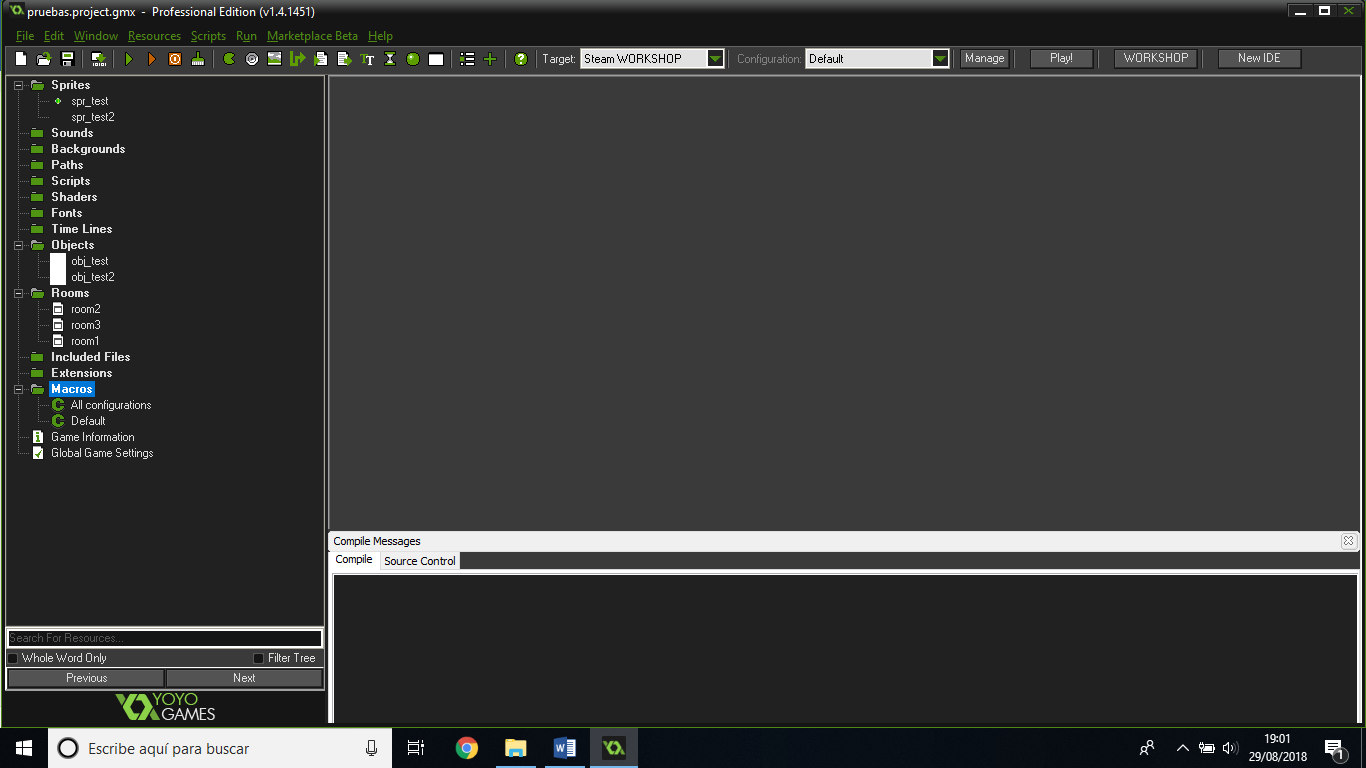


Figura – Interfaz de Game Maker Studio

Todo comienza con un **lienzo** o **“canvas”** vacío.

Este lienzo tiene unas dimensiones determinadas.

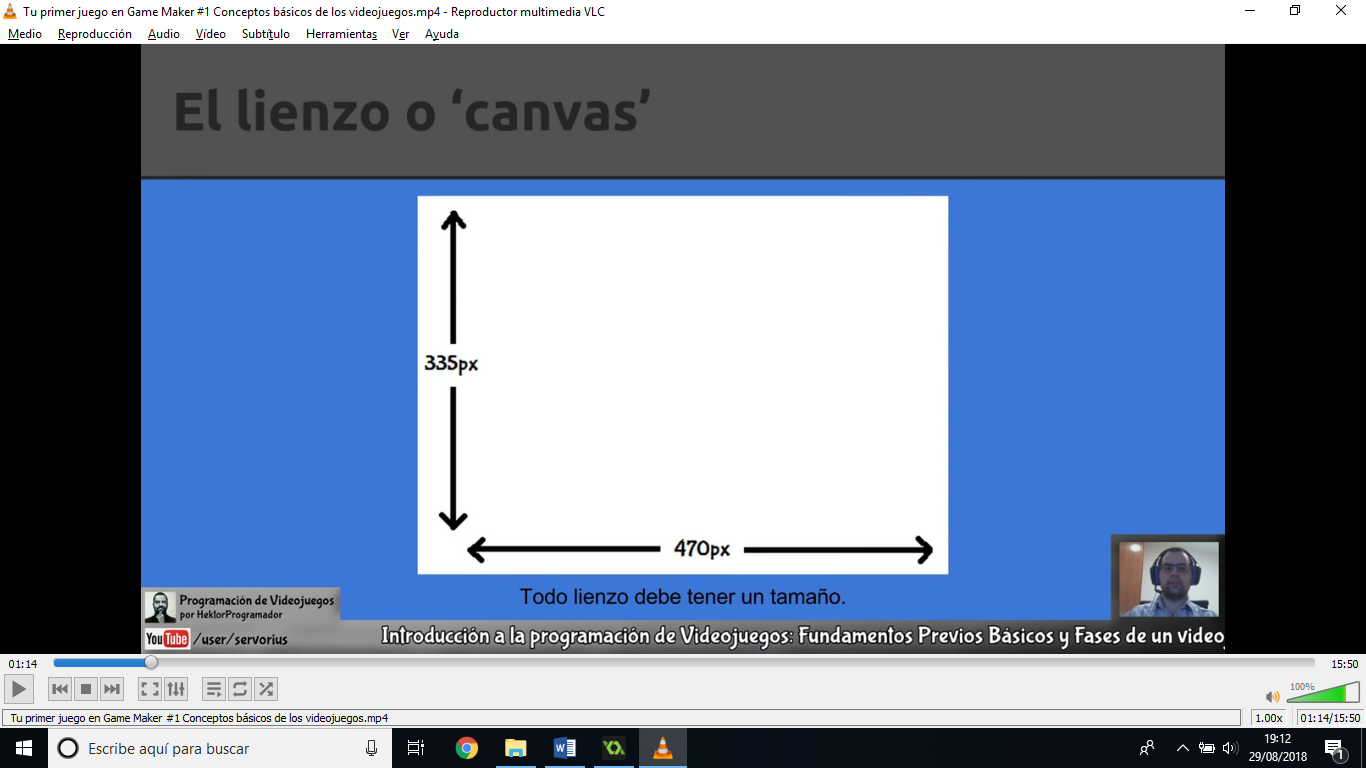


Figura – Dimensiones del canvas

Para trabajar digitalmente el canvas se utilizan las **coordenadas cartesianas**.

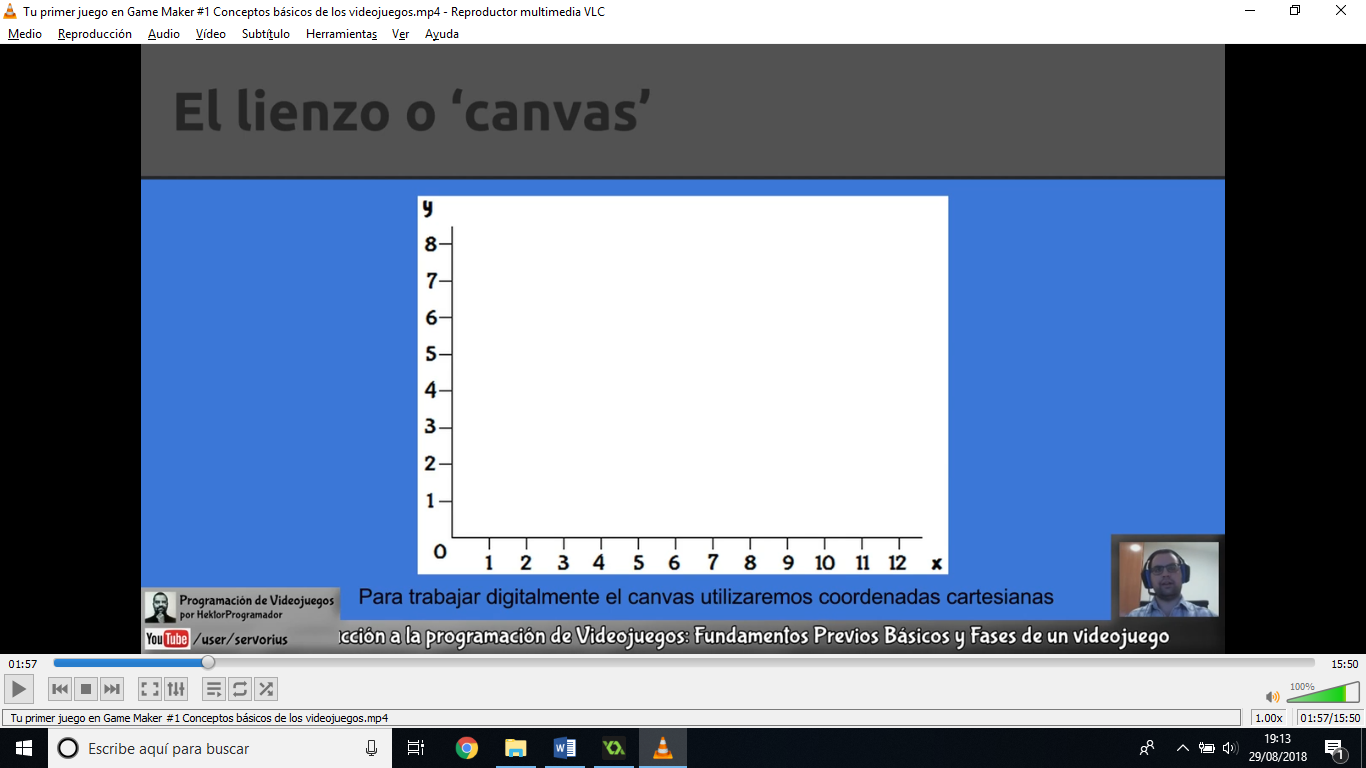


Figura – Sistema de coordenadas cartesianas

Sobre él se puede posicionar y dibujar las gráficas.

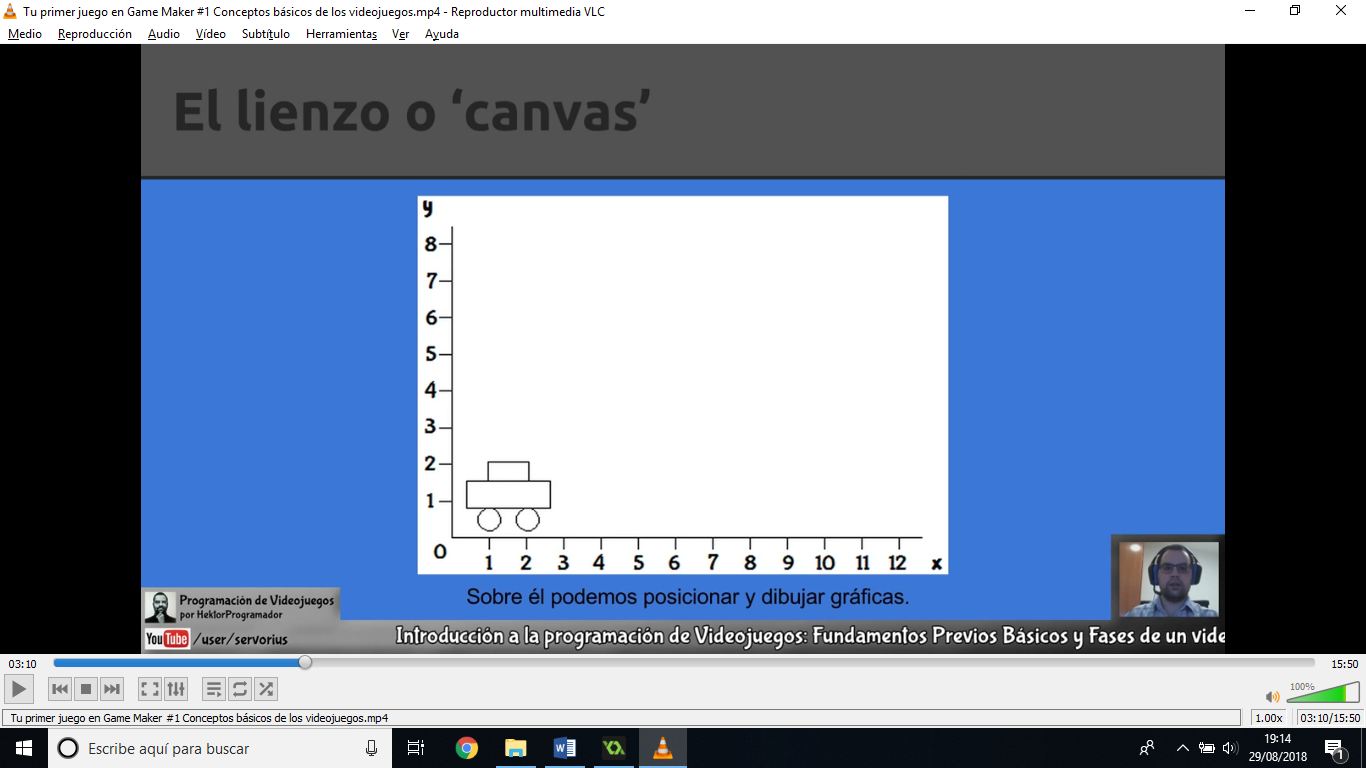


Figura - Sistema de coordenadas cartesianas (II)

Para generar el movimiento se recrean las gráficas en lugares distintos para momentos distintos de forma consecutiva. Estos momentos se denominan **fotogramas**.

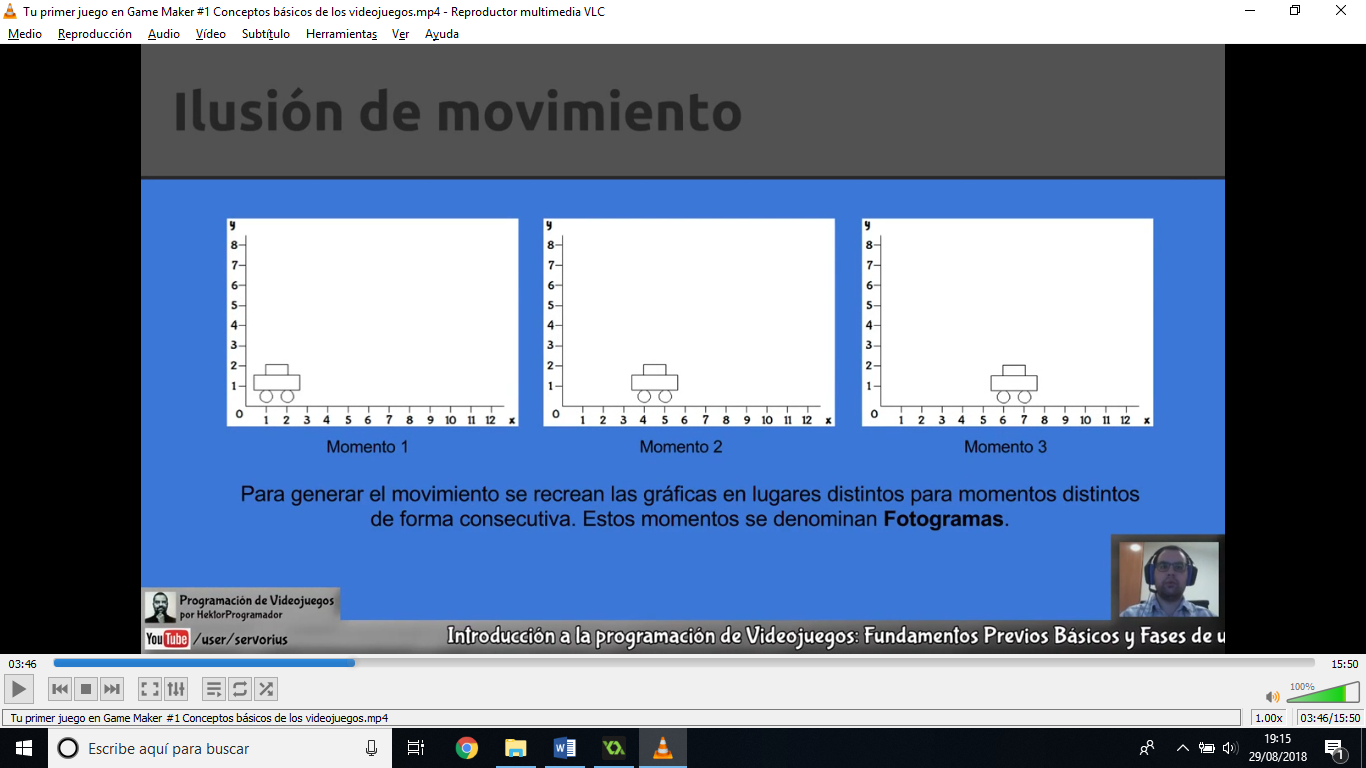


Figura -Fotogramas que forman movimiento

# El lenguaje GML

Aprendiendo el lenguaje de Game Maker se podrá hacer prácticamente cualquier cosa en 2D que se pueda ocurrir con este software de creación de videojuegos. En esta sección se explican los fundamentos de programación en GML junto con métodos y funciones muy útiles para la mayoría de los videojuegos que se piense programar.

Para añadir código dentro de los objetos, se debe añadir dentro del evento correspondiente con la acción de nombre “Execute Code”.

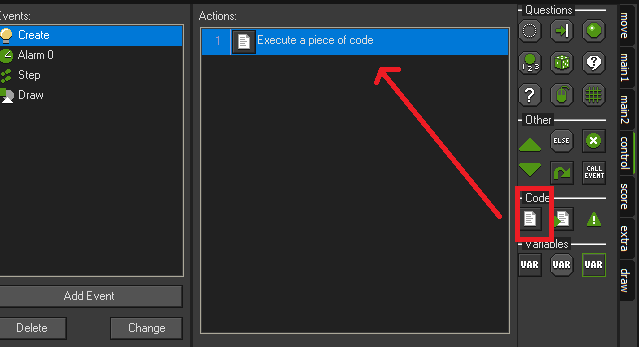


Figura – Acción necesaria para ejecutar un código GML

## Literales y variables

Los **literales** representan información y son el conjunto de tipos de datos presentes en GML.

Los distintos tipos de datos de GML son:

* Reales: 3; 5,42; -199; 5/4
* Cadenas: “Hola mundo”; ‘Esto es un texto’
* Booleanos: true (1); false (0)

Se puede visualizar variables y literales en la consola con las funciones:

* show\_message(valor)
* show\_debug\_message(valor)

Comentarios en linea: //

Comentarios en bloque: /\* \*/

Se puede poner “;” al final de las sentencias, pero no es necesario en GML.

Las **variables** son la parte fundamental de cualquier lenguaje. Son espacios de memoria en los que se almacenan valores. Requieren un identificador único o nombre. Este nombre debe cumplir:

* 64 caracteres de longitud máxima.
* Sin caracteres especiales ni espacios (no poner tildes, ñ,-).
* No empezar por número.

Las variables se pueden modificar en cuanto al valor almacenado.

Como consejo, emplear una de estas notaciones para nombrar variables:

* **Snake:** vida\_inicial, marcador\_de\_puntos
* **Camel case:** vidaActual, puntosPorRonda

Según el ámbito:

* **Global:** accesible desde cualquier lugar 🡪 global.nombre
* **De instancia:** accesible desde sus acciones 🡪 nombre
* **Local:** accesible solo desde el bloque/acción 🡪 var nombre

En cada room, en la pestaña settings, la opción “Creation Code” permite introducir un código que se ejecute en la room nada más inicializarla.

Si por ejemplo se quiere acceder desde el “Creation Code” a una variable de instancia se emplea la nomenclatura del punto:

El objeto es el accesor.

Si hubiera varias instancias del mismo objeto se tomaría el de la primera.

## Macros

En otros lenguajes, esto son las constantes.

Son también espacios de memoria, pero inalterables después de su declaración.

Tienen ámbito global implícito sin accesor.

Se definen en la sección de las “Macros”.

|  |  |
| --- | --- |
| Figura – Macros (I) | Figura – Macros (II) |

Generalmente se escriben en mayúsculas. Algunos ejemplos se pueden ver en Figura 8.

Las macros también pueden almacenar una función, como se observa en la Figura 9. Este es el motivo de que no se conozcan como constantes.

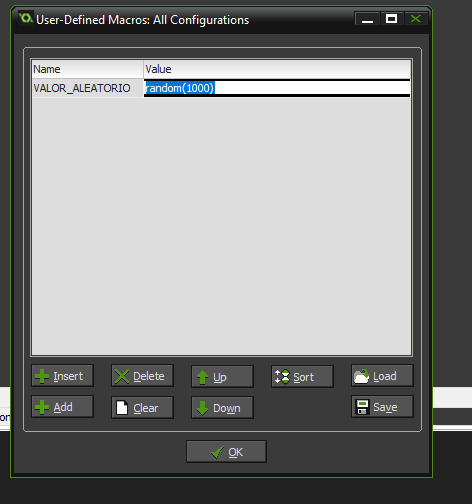


Figura – Macros (III)

En la sección Macros hay dos grupos:

* **All configurations:** aquí se recomienda almacenar las macros comunes a todos proyectos que se creen. Tener en cuenta que las Macros se pueden guardar.
* **Default:** para sustituir/redefinir o añadir nuevas macros en el proyecto actual.

NOTA:

* Función para obtener un número aleatorio flotante: random(nº límite)
* Función para obtener un número aleatorio entero: irandom(nº límite)

Estas dos funciones van desde cero hasta el número límite.

¿Por qué cada vez que se ejecuta en el juego las macros con número aleatorio se generan los mismos números? Esto es porque los números realmente no se generan de forma aleatoria. Lo que Game Maker hace es tomar una captura de la CPU y a partir de ella genera una “semilla”, que es a partir de la cual se genera el número aleatorio. Esta semilla ya la genera Game Maker desde el principio al iniciar el IDE (entorno de desarrollo). Para asegurar que no aparezcan los mismos números aleatorios se debe regenerar la semilla. Esto se consigue empleando la función randomize().

## Operadores

Son signos que indican una acción sobre una o más variables y/o literales.

Ya se ha hablado del operador asignación “=”.

Se debe estudiar las siguientes clases de operadores:

* Aritméticos
* Unitarios y de asignación
* Relacionales
* Lógicos

### Operadores aritméticos

Tabla – Operadores aritméticos

|  |  |
| --- | --- |
| **+** | Suma (especial en cadenas 🡪concatenación) |
| **-** | Resta |
| **\*** | Producto |
| **/** | División |
| **div** | División entera |
| **mod** | Resto de la división entera |

Estos dos últimos se explican con el ejemplo de la Figura 10.

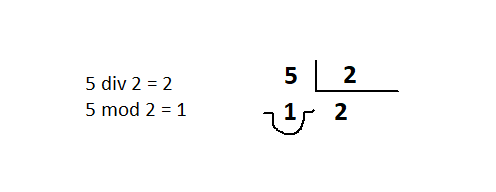


Figura – División y módulo (resto)

### Operadores unitarios y de asignación

Tabla – Operadores unitarios y de asignación

|  |  |
| --- | --- |
| **-** | Cambio de signo |
| **++** | Incremento en una unidad |
| **--** | Decremento en una unidad |
| **+=** | Suma en asignación |
| **-=** | Resta en asignación |
| **\*=** | Producto en asignación |
| **/=** | División en asignación |

### Operadores relacionales

Tabla – Operadores relacionales

|  |  |
| --- | --- |
| **==** | Igual que |
| **!=** | Distinto que |
| **>** | Mayor que |
| **<** | Menor que |
| **>=** | Mayor o igual que |
| **<=** | Menor o igual que |

Los dos primeros son globales, esto es, que se puede operar con números, cadenas de texto o booleanos. El resto operan en general con números únicamente.

De estos operadores siempre se obtiene un booleano como resultado.

### Operadores lógicos

Tabla – Operadores lógicos

|  |  |
| --- | --- |
| **!** | Negación |
| **and** | Conjunción |
| **or** | Disyunción |

De estos también se obtendrán booleanos. Para su mejor comprensión revisar las tablas de la verdad.

## Expresiones

Son combinaciones evaluables de variables, literales y operadores.

Representan un valor conjunto.

El valor puede cambiar según el valor que tengan las variables en cada momento.

Para resolver una expresión, existe un orden:

## Sentencias condicionales: if y switch

Forman parte de las sentencias de control. Permiten modificar el flujo de ejecución de instrucciones de un programa a partir de condicionar el resultado de una expresión lógica o sus posibles valores.

### Condicional If

“if” evalúa una expresión lógica. Si es verdadera, ejecuta unas instrucciones.

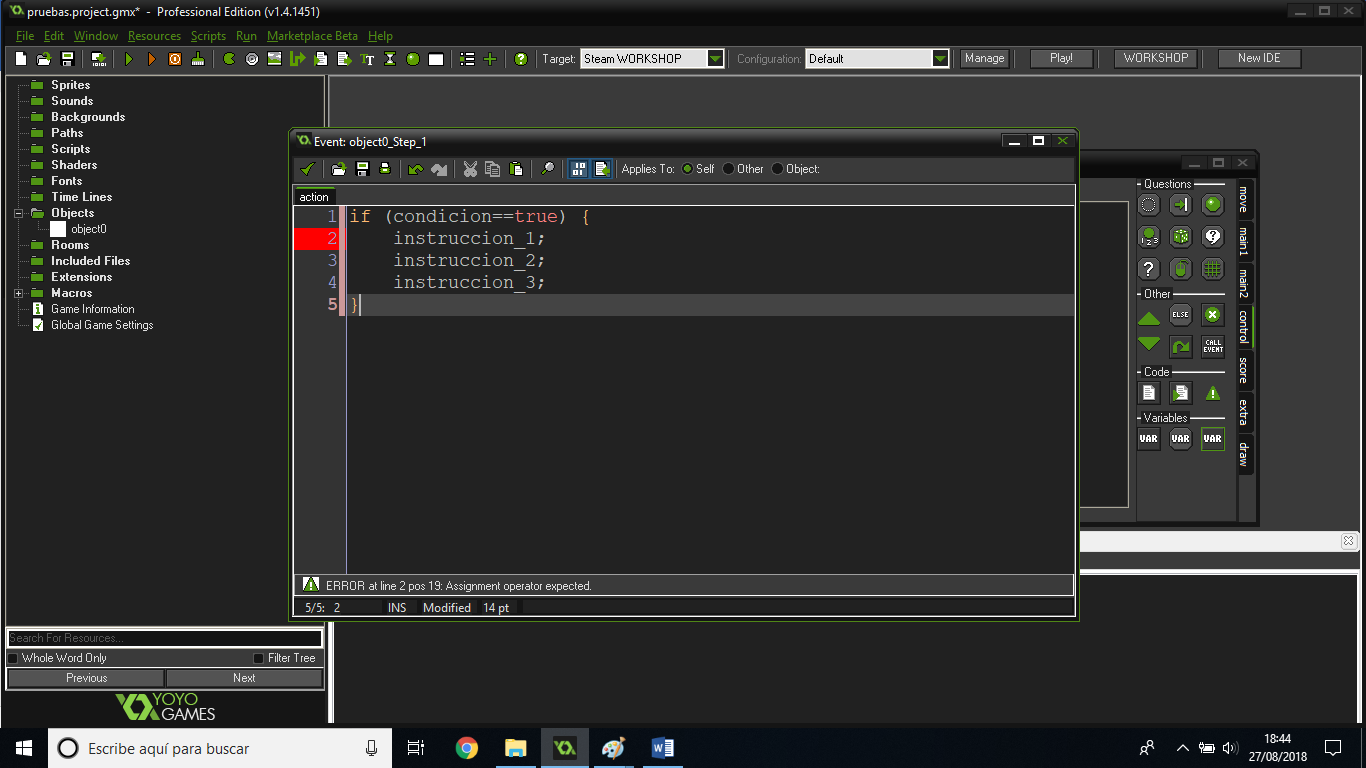


Figura – Sintaxis if

Con la variante “if/else” también se pueden ejecutar otras instrucciones en el caso de que sea falso.

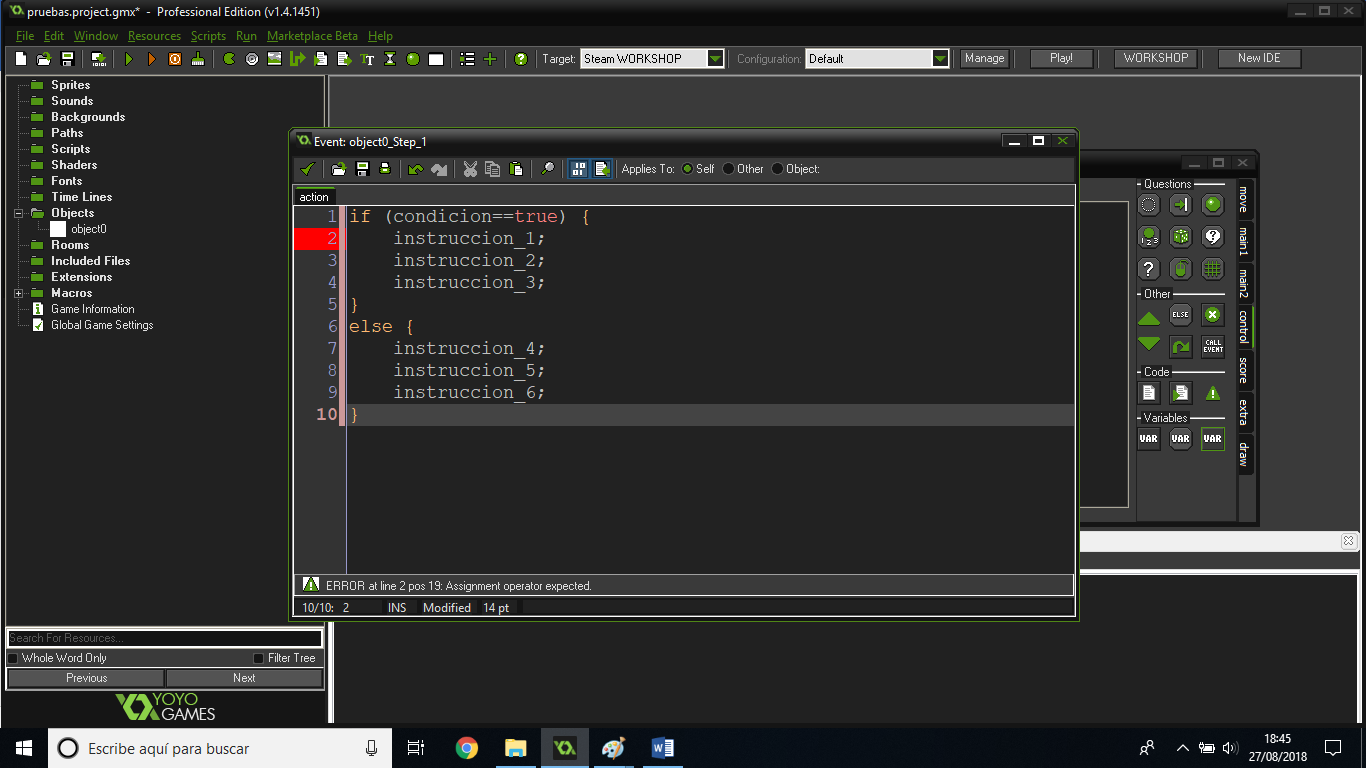


Figura – Sintaxis if/else

Con la variante “if/else if/else” se pueden encadenar múltiples expresiones condicionales.

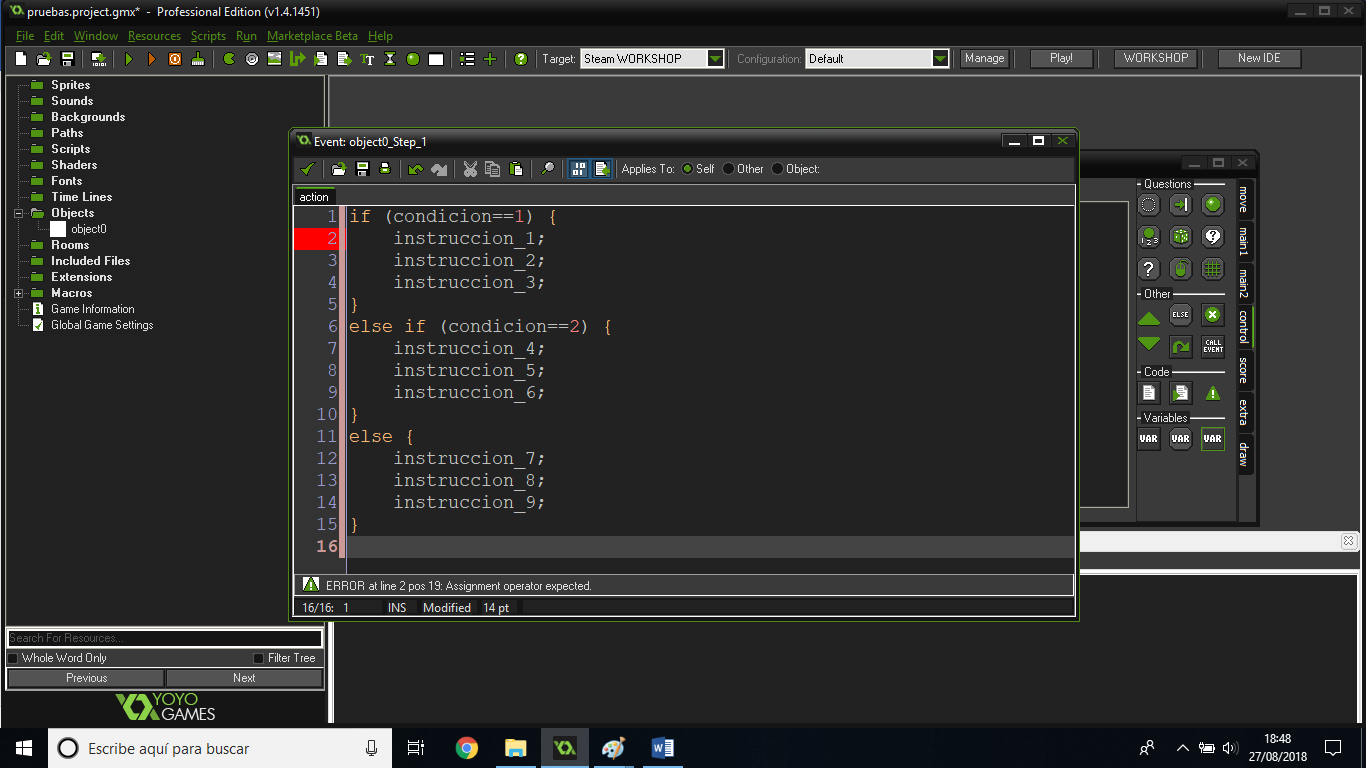


Figura – Sintaxis if/else if/else

### Condicional switch-case

Se podría traducir como “según el caso”.

Sirve para actuar en función de los distintos valores de una expresión. Se puede crear un comportamiento por defecto. La cláusula “break” permite romper la ejecución; olvidarla puede resultar en desastre.

Su limitación es que únicamente se pueden comprobar valores explícitos y no condiciones.

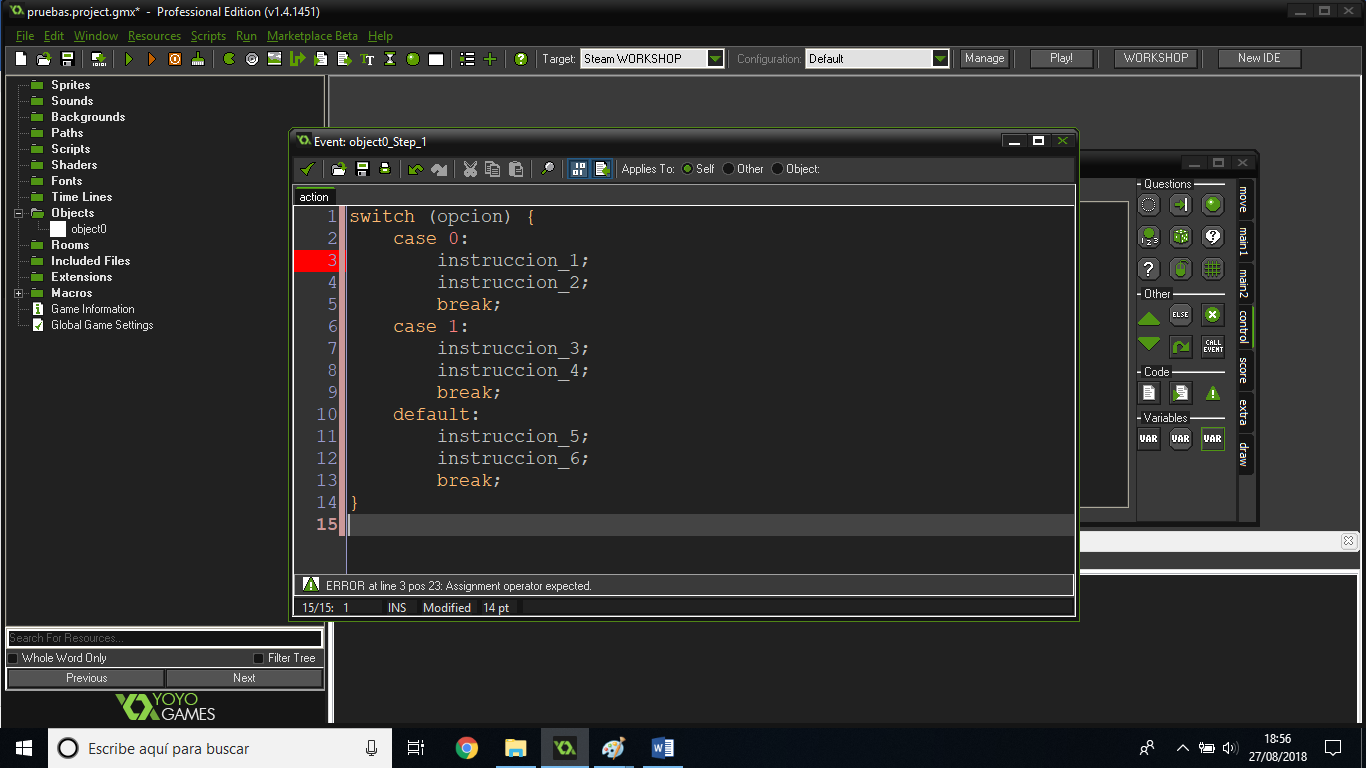


Figura – Sintaxis switch

Para que con distintas opciones se ejecuten las mismas instrucciones de colocan los casos uno detrás de otro. Pensar que, al fin y al cabo, lo que detiene la ejecución del case es el “break”. Si no se pone se ejecuta hacia abajo.

Si no se coloca el “break”, se ejecutarán todas las instrucciones por debajo del primer case que se ejecute.

El bloque “default” permite ejecutar una serie de instrucciones en caso de que la opción introducida no esté entre las anteriores.

## Sentencias iterativas: do, while, repeat y for

Las **sentencias iterativas** permiten modificar el flujo de ejecución de las instrucciones de un programa repitiendo bloques de código en función de una expresión lógica. La gracia está en modificar alguna/s variable/s en cada iteración.

Al igual que en el switch-case, se puede romper la iteración con la cláusula “break”.

Es en este punto en el que se comienza a apreciar realmente el potencial de los ordenadores.

### Sentencia do-until

* Se basa en repetir un bloque HASTA que se cumpla una determinada condición.
* La condición está al final del bloque.
* Se repite mientras condición==false.
* Cuando condición==true, finaliza el proceso iterativo.
* Como mínimo siempre se ejecuta una vez.

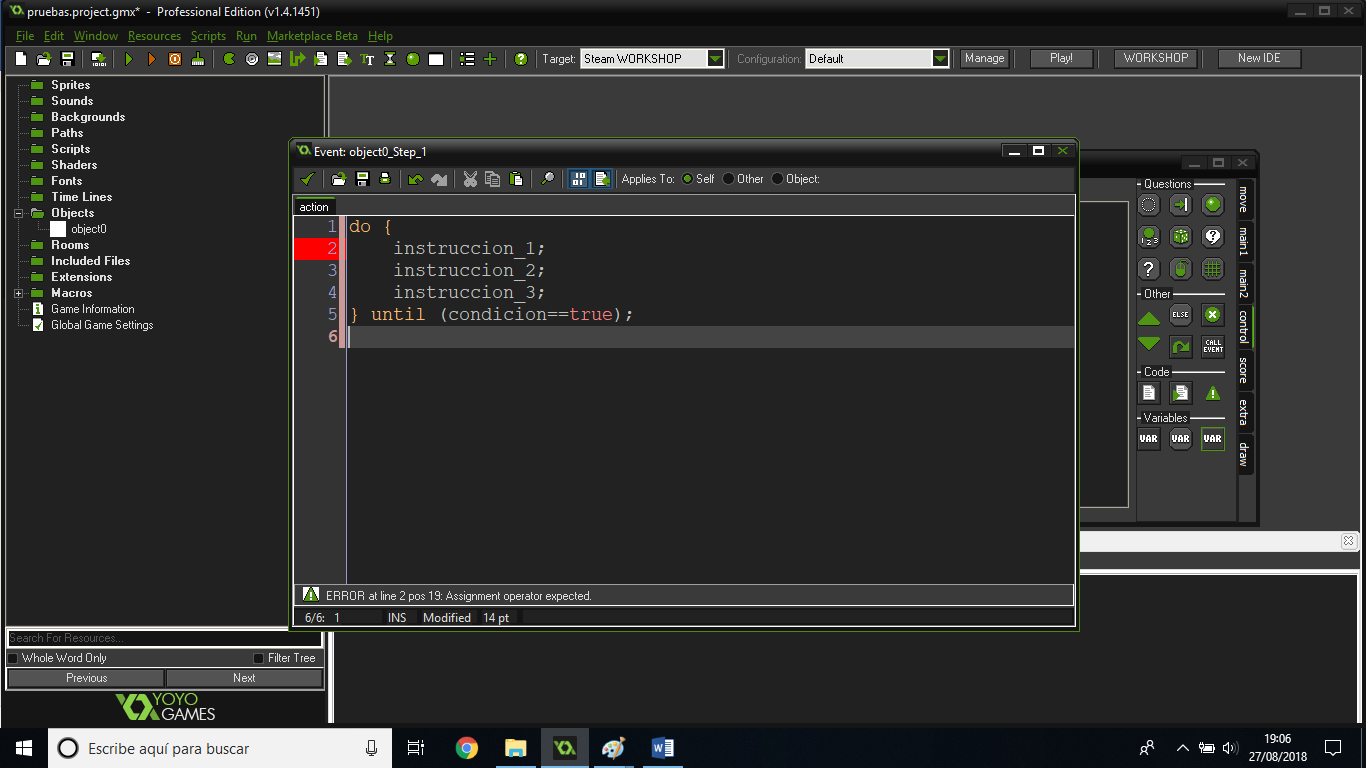


Figura – Sintaxis bucle do-until

### Sentencia while

* Se basa en repetir un bloque mientras se cumpla una determinada condición.
* La condición está al principio.
* Se repite mientras condición==true.
* Cuando la condición==false, finaliza el proceso iterativo.
* No tiene por qué cumplirse ninguna vez.

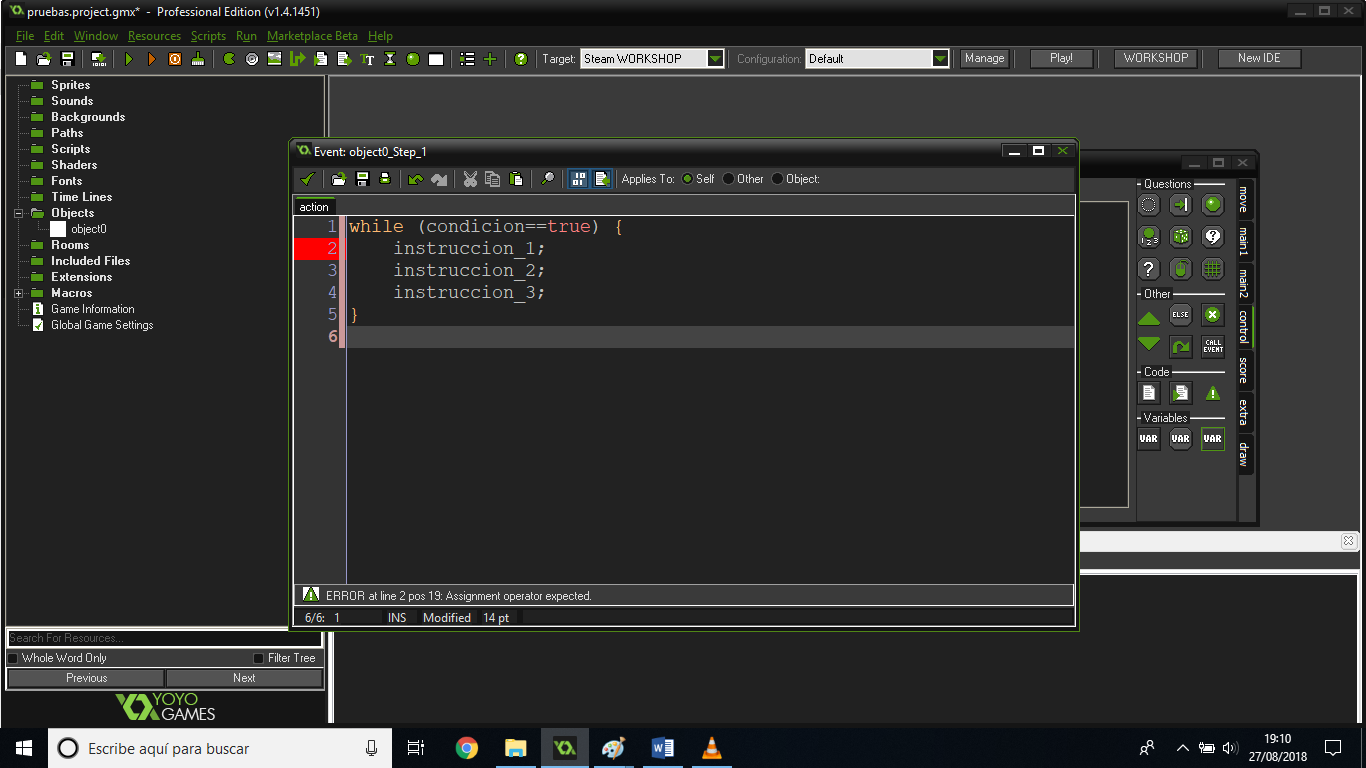


Figura – Sintaxis while

### Sentencia repeat

Se basa en repetir un bloque un número determinado de veces.

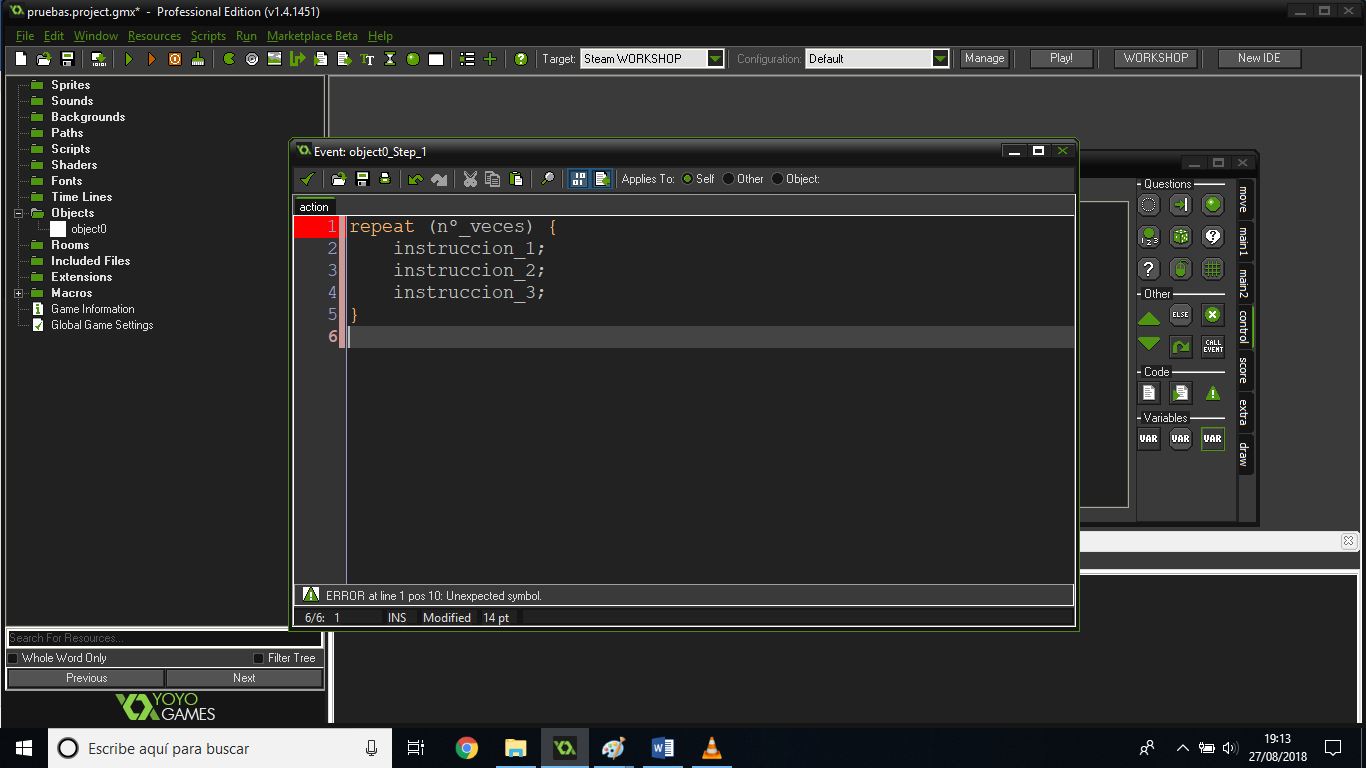


Figura – Sintaxis repeat

### Sentencia for

Se basa en repetir un bloque un número determinado de veces utilizando un contador.

Consta de tres partes:

* Inicialización del contador
* Expresión condicional
* Acción de paso: suele ser incrementar el contador

Seguiría el cuerpo del bucle con el código a ejecutar.

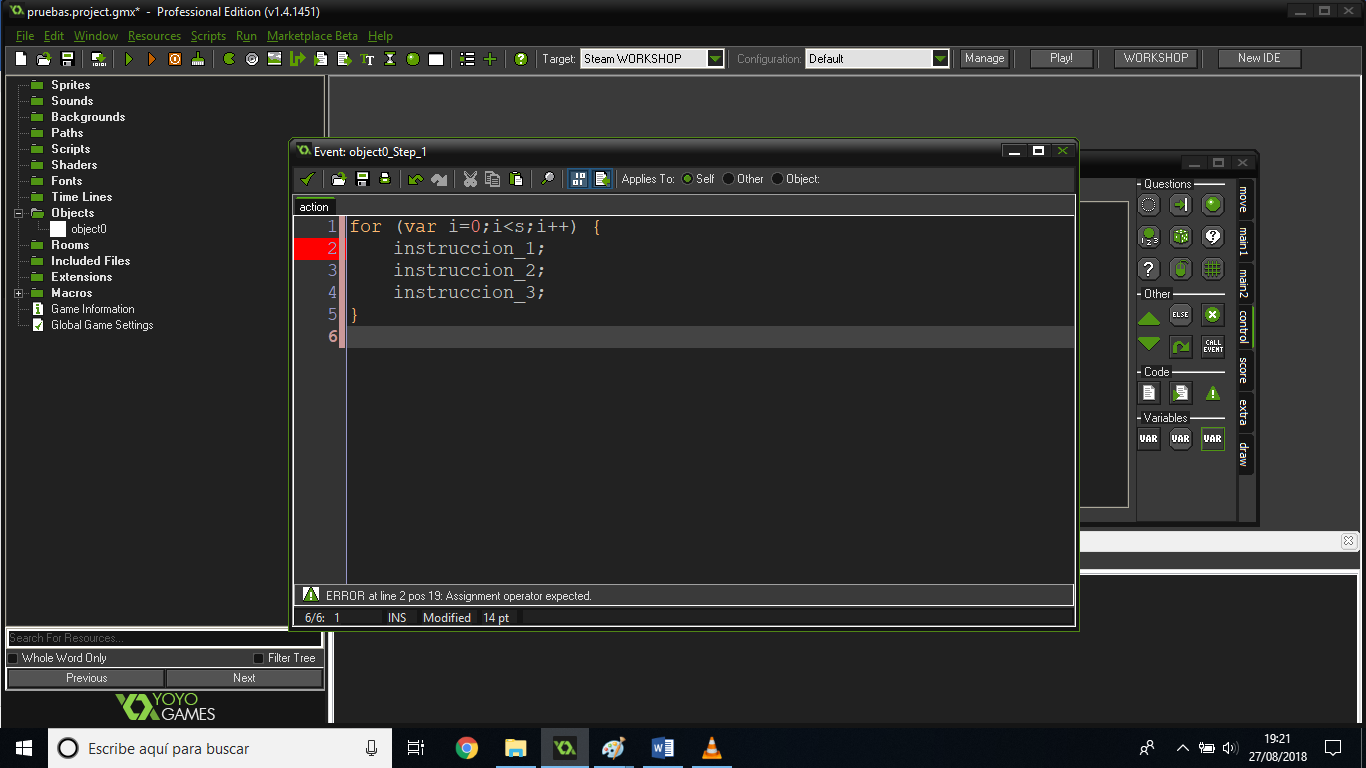


Figura – Sintaxis bucle for

## Scripts

### ¿Qué son los scripts?

Los **scripts** son algo así como lo que en otros lenguajes se conoce como funciones.

Son bloques de código identificados con un nombre, que concuerda con el nombre del fichero que lo contiene.

### Creación de un script básico

Un script muy básico se puede ver en la Figura 19.



Figura – Script básico

Para llamar al script, introducir:

### Uso de variables externas

Como lo que hace el script es sustituir la llamada por las líneas de código que contiene el script, lo representado en la Figura 20 funciona, aunque no es aconsejable. Y es que se nos limita el nombre de las variables, y no tiene interés eso. Se plantea entonces el paso de argumentos.

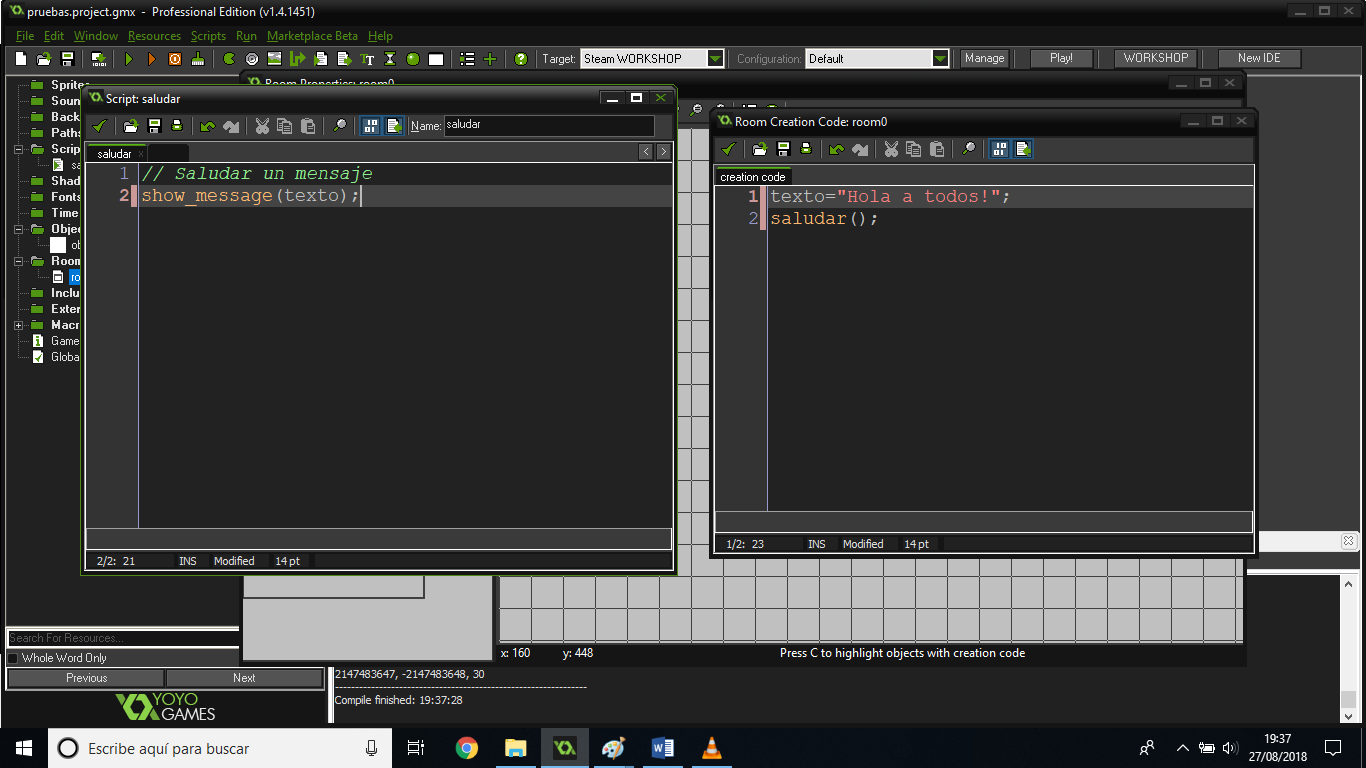


Figura – Uso de variables externas en scripts

### Paso de argumentos

Al crear un script se crea una colección implícita de argumentos. A ellos se puede acceder con los nombres argument0, argument1, argument2, etc. o en forma de array: argument[0], argument[1], argument[2], etc.

Un ejemplo sencillo se puede ver en la Figura 21.

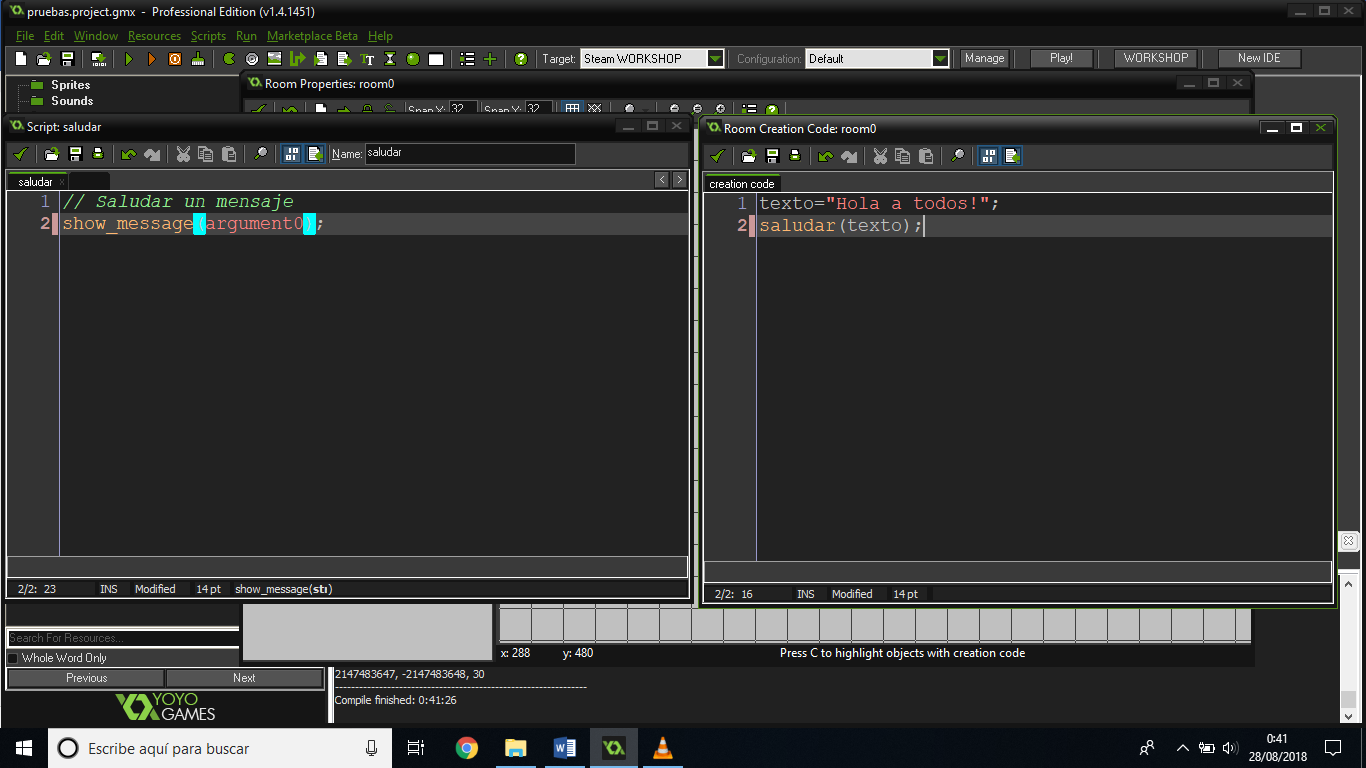


Figura – Paso de argumentos

Otro ejemplo sencillo en el que se repite el saludo tantas veces como se le indique en un segundo parámetro se puede ver en la Figura 22. En ese caso, el nombre de Héctor se repite tres veces.

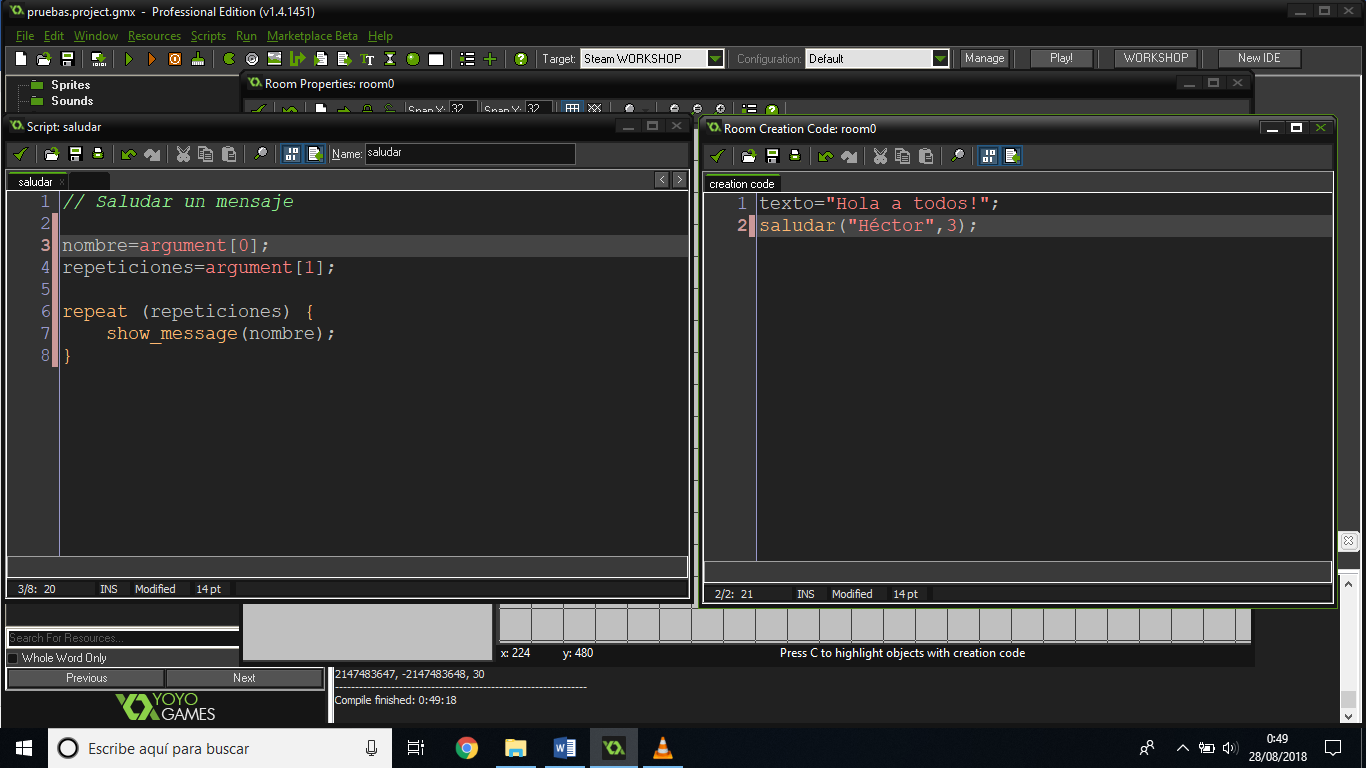


Figura – Paso de argumentos (II)

### Documentación del script

Para ello añadir una línea al principio del script que empiece por “///” y seguido de una ayuda para que se muestre en la barra de estado inferior en las ventanas de código cuando se vaya a llamar al script en el IDE. Mostrará ahí esa línea y permitirá, por ejemplo, recordar fácilmente cual es el orden en el que hay que introducir los argumentos.

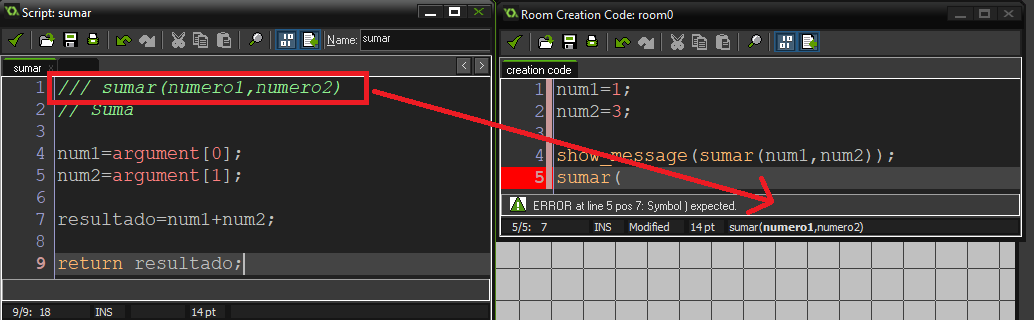


Figura – Documentación del script

### Retorno de valores

Esto es la devolución de información por parte de un script; se captura. Esto se hace con la instrucción “return”. Se puede observar esto en un ejemplo sencillo de un script que devuelve la suma de dos números en la Figura 24. Notar en ese ejemplo que si se elimina la línea 4 del código de la room el programa funcionaría igual. Esto es por lo que ya se ha explicado: el script superpondría la llamada al mismo script, y, tal y como se han definido corresponderían a la room o al objeto que fuese.

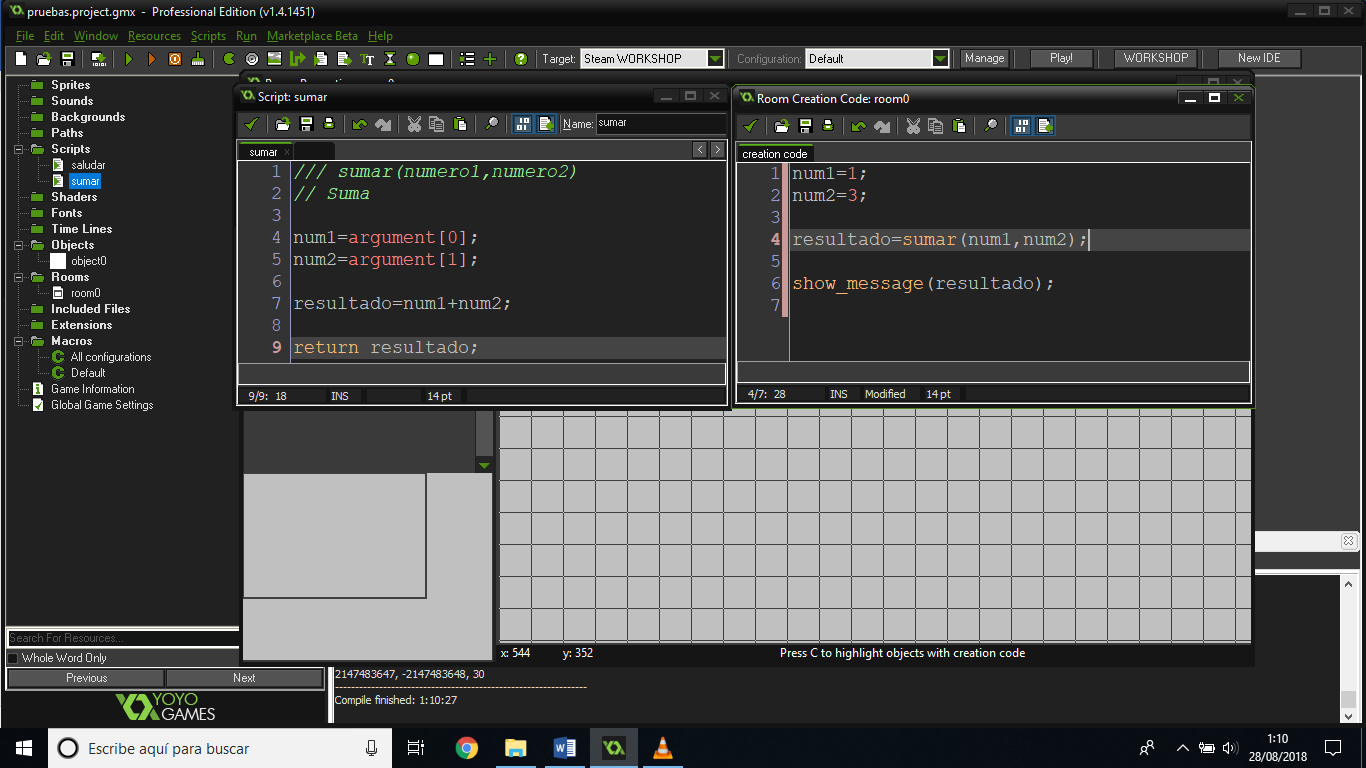


Figura – Devolución de valores

Para evitar este tipo de problemas conviene definir las variables que solo se vayan a emplear dentro de un script como locales, de forma que dichas variables pertenezcan únicamente al script, por lo que lo más correcto sería lo mostrado en la Figura 25.

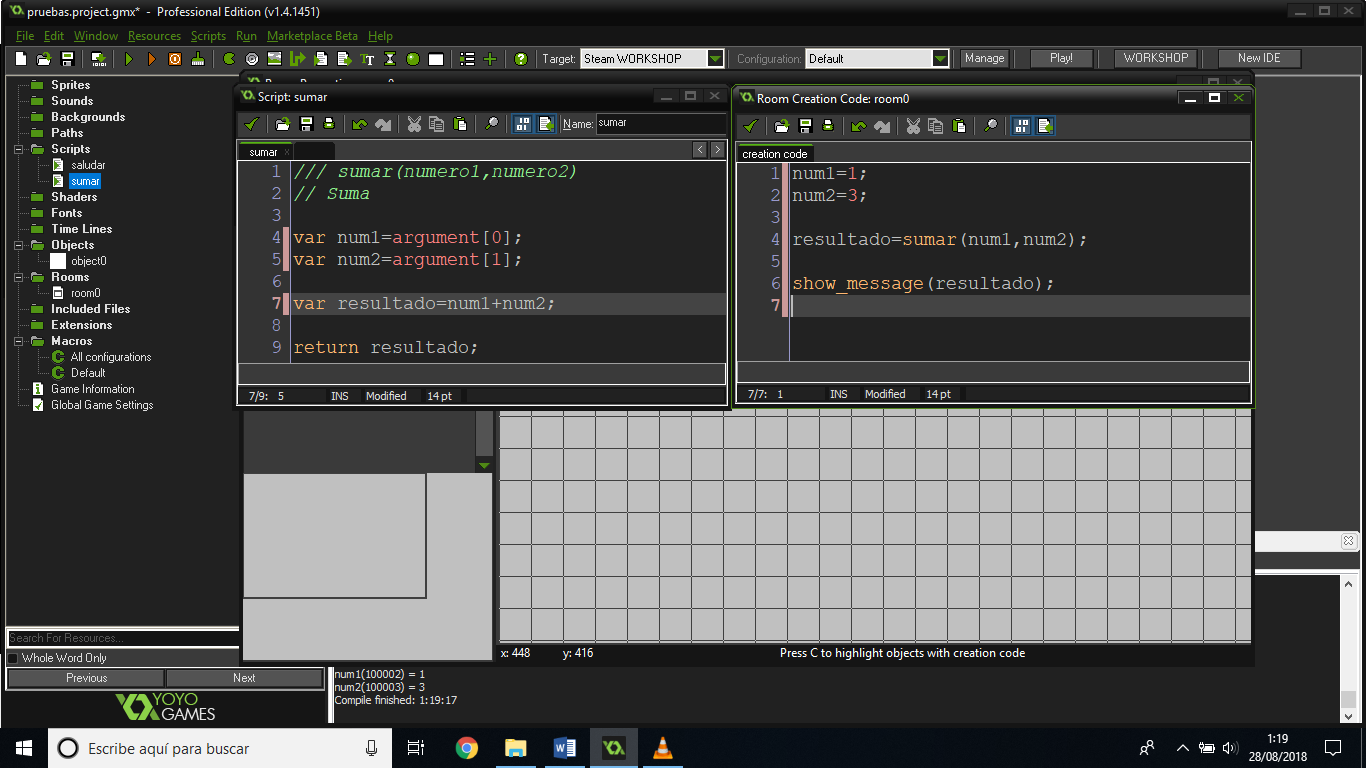


Figura – Devolución de valores (II)

### Buenas prácticas con scripts

Los scripts deben ser pequeñas porciones de código; esto es, que realicen una única tarea. Es mejor a la larga hacer varios scripts que enganchar en uno solo varios if o switch.

## Funciones útiles

### Funciones de redondeo

Tabla – Funciones de redondeo

|  |  |
| --- | --- |
| **round (número)** | Redondea al alza si la parte decimal es mayor de 0,5 |
| **floor (número)** | Redondea siempre a la baja |
| **ceil (número)** | Redondea siempre al alza |
| **sign (número)** | Devuelve el signo del número (-1 o 1) |
| **abs (número)** | Devuelve el valor absoluto de un número (distancia) |
| **clamp (número,mín,máx)** | Devuelve un límite si el número se sale del rango |

### Funciones de aleatoriedad

Tabla – Funciones de aleatoriedad

|  |  |
| --- | --- |
| **choose (valor1, valor2, …, valorN)** | Escoge un valor aleatorio de una lista de valores. |
| **random (número)** | Genera un real aleatorio comprendido entre cero y el número. |
| **irandom (número)** | Genera un entero aleatorio comprendido entre cero y el número. |
| **random\_range (número1, número2)** | Genera un real aleatorio entre dos números. |
| **irandom\_range (número1, número2)** | Genera un entero aleatorio ente dos números. |

### Funciones de conversión

Tabla – Funciones de conversión

|  |  |
| --- | --- |
| **string(real)** | Convierte un real a una cadena |
| **real(“cadena”)** | Convierte una cadena a un real |

## Objetos e instancias

### ¿Cuáles son las diferencias?

Los **objetos** son modelos o plantillas; algo así como un molde de galletas.

Las **instancias** son el resultado de crear los objetos en la room; algo así como las galletas.

Por tanto, lo que existe en la room son las instancias.

Las instancias comparten eventos y acciones del objeto, pero son independientes porque cuentan con propiedades individuales.

### Eventos y acciones

Los eventos de objeto capturan momentos específicos durante su ciclo de vida:

* Create y Destroy
* Inputs, de room, de animación, etc.
* Step, Draw y Draw GUI

Las acciones son comportamientos predefinidos que se pueden sustituir por bloques de código GML.

### Propiedades

Aportan información o comportamientos.

Son como “variables” internas.

Tabla – Propiedades de los objetos

|  |  |
| --- | --- |
| **id** | **xstart/ystart** |
| **object\_index** | **xprevious/yprevious** |
| **x/y** | **visible** |
| **hspeed/vspeed** | **depth** |
| **direction** | **persistent** |
| **speed** | **…** |

Para saber el nombre de un objeto según su índice:

Sobre la velocidad observar sus componentes horizontal y vertical en la Figura 26.

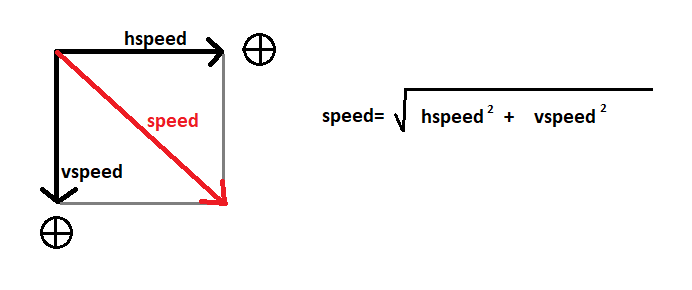


Figura – Componentes de la velocidad de un objeto

Y es que la velocidad puede establecerse de dos formas:

* Dando las dos componentes (horizontal y vertical), esto es, hspeed y vspeed.
* Dando el módulo de la velocidad y su dirección y sentido (speed + direction).

“direction” se mide en grados (ver Figura 27). Por defecto, su valor es cero.

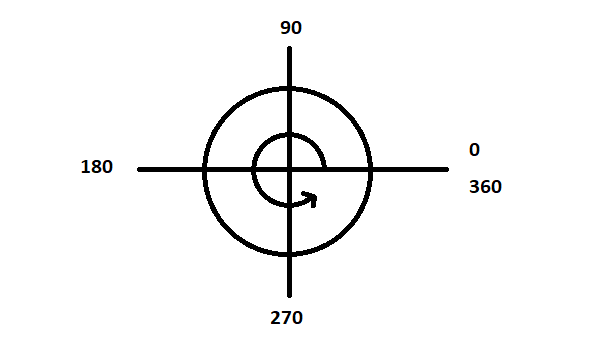


Figura – Ángulos en GML

El sistema de coordenadas no es el que se suele estudiar en matemáticas. Aquí hay una pequeña diferencia: el eje vertical tiene el signo cambiado. Lo que se muestre en la room es aquello que aparece en la x e y positivas. Ver Figura 28.

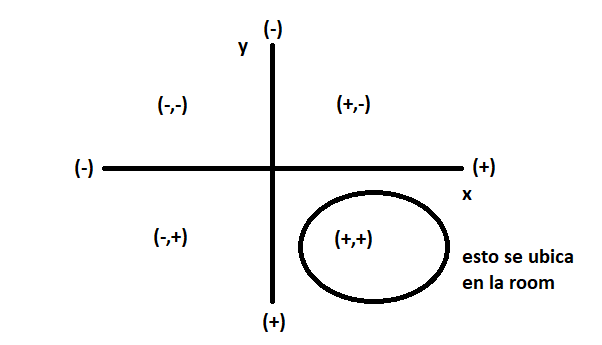


Figura - Sistema de coordenadas en Game Maker

### Herencia

Un objeto puede heredar los eventos de otro.

Si el objeto hijo redefine los eventos del padre entonces los del padre no se ejecutarán en el hijo. Esto es así a no ser que se utilice un método especial llamado event\_inherited() antes de escribir el código del hijo. Así se ejecutarán ambos códigos.

Para heredar: Parent: obj\_padre (Figura 29).

Tener en cuenta que también se heredarán las variables si no se sobreescriben.

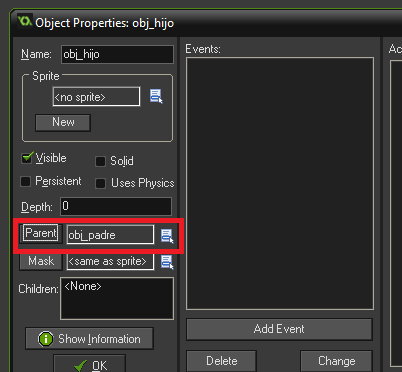


Figura - Herencia

## Sprites y sus propiedades

Los **sprites** son imágenes que se emplean para representar un objeto.

### Propiedades básicas

Tabla – Propiedades básicas de los sprites de los objetos

|  |  |
| --- | --- |
| **sprite\_index(índice)** | Permite definir el sprite asignado (con el valor -1 se quita el sprite) |
| **sprite\_width** | Devuelve el ancho en píxeles del sprite actual asignado al objeto (lectura) |
| **sprite\_height** | Devuelve el alto en píxeles del sprite actual asignado al objeto (lectura) |

Para obtener el nombre del sprite a partir de su índice:

### Propiedades de transformación

Tabla – Propiedades de transformación

|  |  |
| --- | --- |
| **image\_xscale** | Escalado horizontal (1=100%) |
| **image\_yscale** | Escalado vertical (1=100%) |
| **image\_angle** | Ángulo de la imagen del sprite |
| **image\_blend** | Color de mezclado del sprite |
| **image\_alpha** | Opacidad del sprite (1=100%) |

Notar que los image\_scale permiten hacer “mirroring” al poner valores negativos.

NOTA: Son diferentes image\_angle y direction.

Para los colores se puede acceder a algunos predefinidos usando c\_blue, c\_red, etc.

### Propiedades de la animación

Tabla – Propiedades de la animación

|  |  |
| --- | --- |
| **image\_index** | Imagen actual de la animación |
| **image\_speed** | Velocidad de la animación |
| **image\_number** | Nº de imágenes que forman la animación (lectura) |

La fórmula para conseguir el image\_speed correcto sin depender de que room\_speed se fije es:

Suponiendo que nuestra animación consta de 4 imágenes, dura 0.5 segundos y la Room está a 30 FPS sería: (4/30) / 0.5 = 0.26.

## Evento Step

Es muy importante entender que los eventos fundamentales que componen un videojuego serán dos: el Step y el Draw. Estos son los que componen el loop.

En el evento Step se programa la lógica del juego. Esto incluye:

* Inputs por teclado, ratón, etc.
* Recálculo de posiciones
* Colisiones

Posee diferentes momentos importantes:

* Begin Step
* Step
* End Step

En realidad, se puede usar solamente el evento Step, pero se debe ser ordenado.

## Evento Draw

Por defecto se dibuja el sprite asignado al objeto.

Manualmente, haciendo uso de funciones de dibujo se pueden trazar muchos otros elementos: fondos, sprites, textos, formas geométricas, etc.

Hay que considerar que estas funciones que aquí se explican funcionan solo en el evento Draw.

El proceso de dibujo es de atrás hacia adelante; entre objetos, el orden se controla con la profundidad (depth). Tener en cuenta que, a mayor profundidad, mayor es la facilidad de que otro objeto le quede por delante y se vea menos.

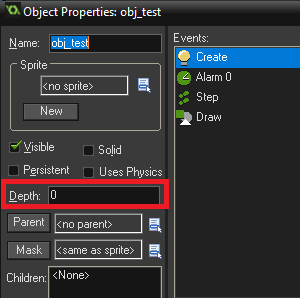


Figura - Profundidad

Momentos importantes:

* Begin Draw
* Draw
* End Draw

Nuevamente si se es ordenado, es suficiente con usar Draw.

Algunas funciones relevantes se recogen en Tabla 12.

Tabla – Funciones del evento Draw

|  |  |
| --- | --- |
| **draw\_self()** | **draw\_set\_halign(fa\_center)** |
| **draw\_rectangle(x1,y1,x2,y2,outline)** | **draw\_set\_valign(fa\_middle)** |
| **draw\_set\_colour(c\_color)** | **draw\_text(x,y,string)** |

## Evento Draw GUI

La **GUI (Graphical User Interface)** recoge todos aquellos elementos sobre los que el jugador interactúa, menús, botones, seleccionables y todo aquello que se dibuje sobre el fondo. En la Figura 31 se puede ver un ejemplo.



Figura – Ejemplo de GUI

Otro concepto interesante es el de **HUD (Head Up Display)**. Se refiere a la pantalla transparente sobre el juego en la que se proyecta información en forma de indicadores, barras, medidores, avatares u otros que ayudan a conocer el estado en el que se encuentra el jugador durante el juego. Barras de vida, radares de enemigos o velocímetros forman parte del HUD. La Figura 32 muestra un ejemplo de HUD.



Figura – Ejemplo de HUD

El evento Draw GUI:

* Es como un Draw normal con una depth muy baja
* Se utiliza para juegos multiresolución
* O también al ser utilizados sobre vistas en rooms
* Los elementos se posicionan y escalan en función de la pantalla

### Caso de estudio

Se tiene una room 1500x1500. Se visualiza una porción de 500x500 centrada sobre un personaje en medio. Se quiere dibujar un HUD con un avatar y una barra de vida sobre la parte que visualizamos.

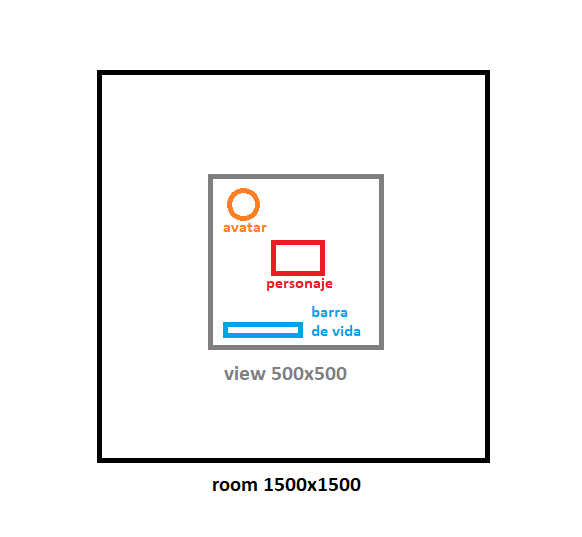


Figura – Caso de estudio

Para su resolución, tener en cuenta una serie de consideraciones:

* Se debe saber cómo funcionan las views (vistas). Un resumen de los parámetros que se deben conocer se expone en la Figura 34.

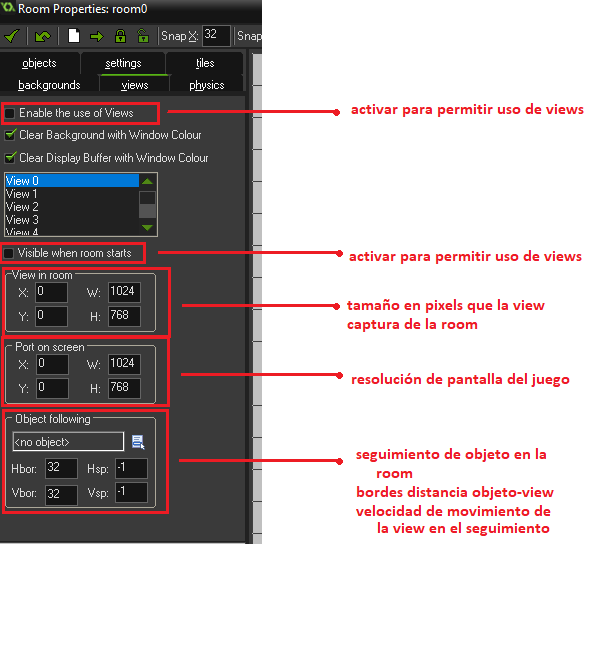


Figura – Funcionamiento de las views

* Para colocar los elementos del HUD sobre la vista se debe emplear el evento Draw GUI. Mientras que el evento Draw emplea la room como referencia en el sistema de coordenadas, Draw GUI emplea la vista. Para que Game Maker use Draw GUI en lugar del Draw, se debe escribir un código en blanco en este último para sobrescribirlo.
* Asignar en el Create la posición que se quiera que tenga el objeto en la view.

## Inputs por teclado y ratón

### Teclado

Detección de presión sobre teclas.

Tabla – Inputs por teclado

|  |  |
| --- | --- |
| **keyboard\_check (key)** | Comprueba que se pulse todo el rato |
| **keyboard\_check\_pressed (key)** | Comprueba si se ha pulsado una vez |
| **keyboard\_check\_released (key)** | Comprueba si se ha soltado una vez |

¿Qué sustituir por key? Hay diferentes identificadores de keys:

* **Teclas generales**: vk\_nokey, vk\_anykey, vk\_left, etc.
* **Teclas de caracteres**: ord(“W”) // ordinal Unicode

### Ratón

Detectar presión de los botones del ratón.

Tabla – Inputs de ratón

|  |  |
| --- | --- |
| **mouse\_check\_button (key)** | Comprueba que se pulse todo el rato |
| **mouse\_check\_pressed (key)** | Comprueba si se ha pulsado una vez |
| **mouse\_check\_released (key)** | Comprueba si se ha soltado una vez |

Los identificadores de keys generales se puede encontrar en Tabla 15.

Tabla – Identificadores de keys del ratón

|  |
| --- |
| **mb\_left // mb\_middle // mb\_right** |
| **mb\_none // mb\_any** |
| **mb\_wheel\_up // mb\_wheel\_down** |

Variables intrínsecas de posición del ratón en la room:

Tabla – Variables intrínsecas de la posición del ratón en la room

|  |  |
| --- | --- |
| **mouse\_x** | **mouse\_y** |

Función que da la dirección entre dos puntos con sentido desde el punto (x1,y1) hasta el (x2,y2):

## Rooms

### Propiedades y métodos básicos

Tabla – Propiedades y métodos básicos para control de rooms

|  |  |
| --- | --- |
| **room\_width** | Ancho de la room |
| **room\_height** | Alto de la room |
| **room\_speed** | Velocidad de la room (steps por segundo) |
| **room** | Identificador de la room actual |
| **room\_first** | Room inicial |
| **room\_last** | Room final |
| **room\_previous(room)** | Room anterior a la room especificada |
| **room\_next(room)** | Room siguiente a la room especificada |
| **room\_restart()** | Reiniciar room |
| **room\_exists(room)** | Comprobar si existe la room especificada |
| **room\_goto(room)** | Cambiar a la room especificada |
| **room\_goto\_next()** | Cambiar a la room siguiente |
| **room\_goto\_previous()** | Cambiar a la room anterior |

Para averiguar el nombre de una room en concreto:

El orden de ejecución de las rooms es de arriba hacia abajo (Figura 35), siendo la primera de todas la que sí o sí se ejecutará inicialmente. Con los métodos anteriores se puede manipular según interese a partir de ahí.

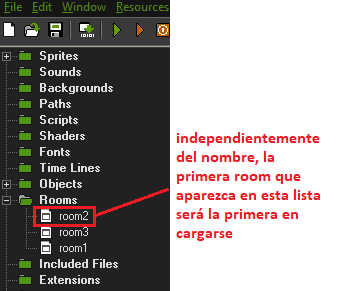


Figura – Orden de ejecución de rooms

Al poner una room en marcha o reiniciarla, todas las instancias se crean de nuevo.

### Conservar objetos entre rooms

¿Es posible?

Sí, utilizando la persistencia.

Utilizando la persistencia el objeto conserva sus propiedades y su posición. El evento Create solo se ejecuta la primera vez.

¿Cómo detectar cambios de room?

Con los eventos room\_start y room\_end. En ellos se puede hacer determinadas acciones.

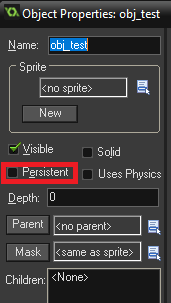


Figura - Persistencia

## Alarmas

Las alarmas permiten ejecutar eventos en el tiempo.

Permite la ejecución de código en el futuro.

Se programan en steps, no en segundos. Si se quiere hacer en segundos se debe considerar la variable intrínseca room\_speed.

Game Maker: Studio permite la creación de hasta 12 alarmas por objeto.

Los momentos importantes en el evento Alarm:

* Creación: se establece el código a ejecutar.
* Llamada: se establece el temporizador en steps.
* Cuenta atrás: se resta 1 al temporizador por step.
* Ejecución: cuando se llega a cero se ejecuta.
* Finalizada: una vez se ha ejecutado, temporizador a -1.

Sobre los steps, se podría decir que una room con un room\_speed de 30 funciona a 30 FPS.

### Evento Alarma

“alarm[0]” devuelve el temporizador, su estado (el momento en el que se encuentra de la cuenta atrás).

Para configurar el temporizador para que ejecute el código de la alarma en 3 segundos: alarm[0]=room\_speed\*3.

Tener en cuenta que la alarma puede no llegar a ejecutarse nunca si no se le deja al temporizador llegar a cero.

### Caso de estudio

Crear ejemplo sencillo de disparar bolas. Destruir las bolas si salen de la room. Limitar el máximo de disparos por segundo.

* Disparos ilimitados
* 1 disparo cada 2 segundos
* 1 disparo cada segundo
* 5 disparos por segundo
* 10 disparos por segundo

Truco para crear las bolas disparadas:

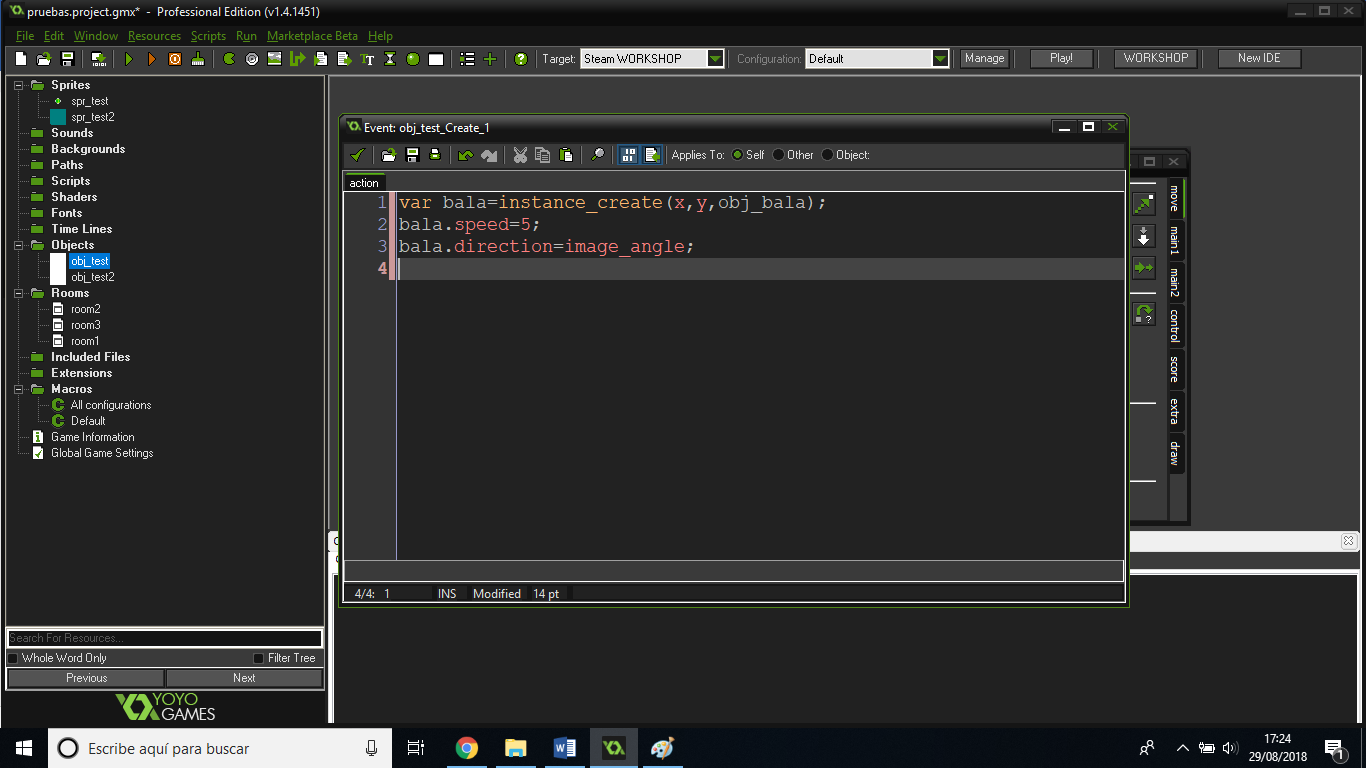


Figura – Caso de estudio (II)

Para destruir la bala, en el evento Outside Room añadir un instance\_destroy().

## Sonidos

### Métodos básicos

Control de sonido:

Tabla – Métodos para control de sonido

|  |
| --- |
| **audio\_play\_sound (sound\_id, prioridad, loop)** |
| **audio\_stop\_sound (sound\_id)** |
| **audio\_pause\_sound (sound\_id)** |
| **audio\_resume\_sound (sound\_id)** |

La prioridad es un valor que está entre 0 y 100.

Comprobaciones de sonido:

Tabla – Métodos para comprobar sonido

|  |
| --- |
| **audio\_is\_playing (sound\_id)** |
| **audio\_is\_paused (sound\_id)** |

Operaciones globales con sonidos:

Tabla – Métodos para operar globalmente el sonido

|  |
| --- |
| **audio\_stop\_all ()** |
| **audio\_pause\_all ()** |
| **audio\_resume\_all ()** |

Estos controles serán imprescindibles para evitar el acoplamiento y superposición indeseados de sonidos y canciones.

## Arrays 1D

Son variables especiales que almacenan múltiples valores. Son útiles en menús, RPGs, juegos de cartas, récords y estadísticas, etc.

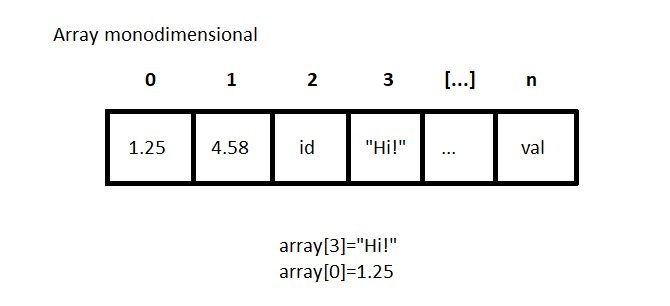


Figura – Array 1D

Para hacer un recorrido dinámico suele ser interesante emplear un bucle tipo for o repeat.

Para saber el número de elementos que contiene un array:

Para asignar un valor a un arreglo al final:

Para borrar un elemento del array:

Realmente no se pueden borrar los elementos, aunque sí que contengan nada (noone). Las listas sí permiten borrar elementos.

## Array 2D

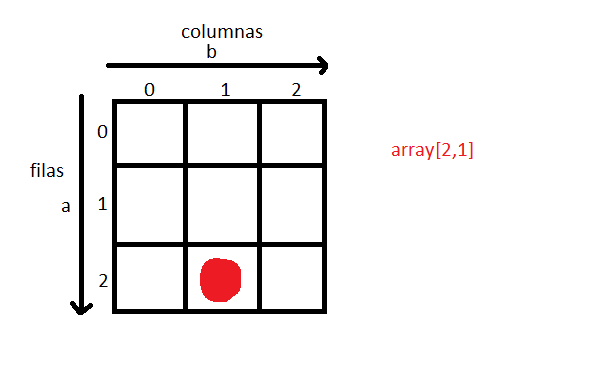


Figura -Array 2D

Puede ser necesario hacer global el array.

Crear macros para ayudar a poner los índices del array.

@ es el accesor para poner los índices del array.

Enganchar un for dentro de un for o repeat dentro de repeat para recorrerlos. Se requiere conocer dos métodos para hacerlo de forma eficaz:

Nº de filas del arreglo:

Nº de columnas del arreglo:

# Consejos para crear un juego de plataformas

## Preparar recursos básicos

Esto incluye:

* Sprites
* Sonidos
* Backgrounds
* Fuentes necesarias

## El escenario

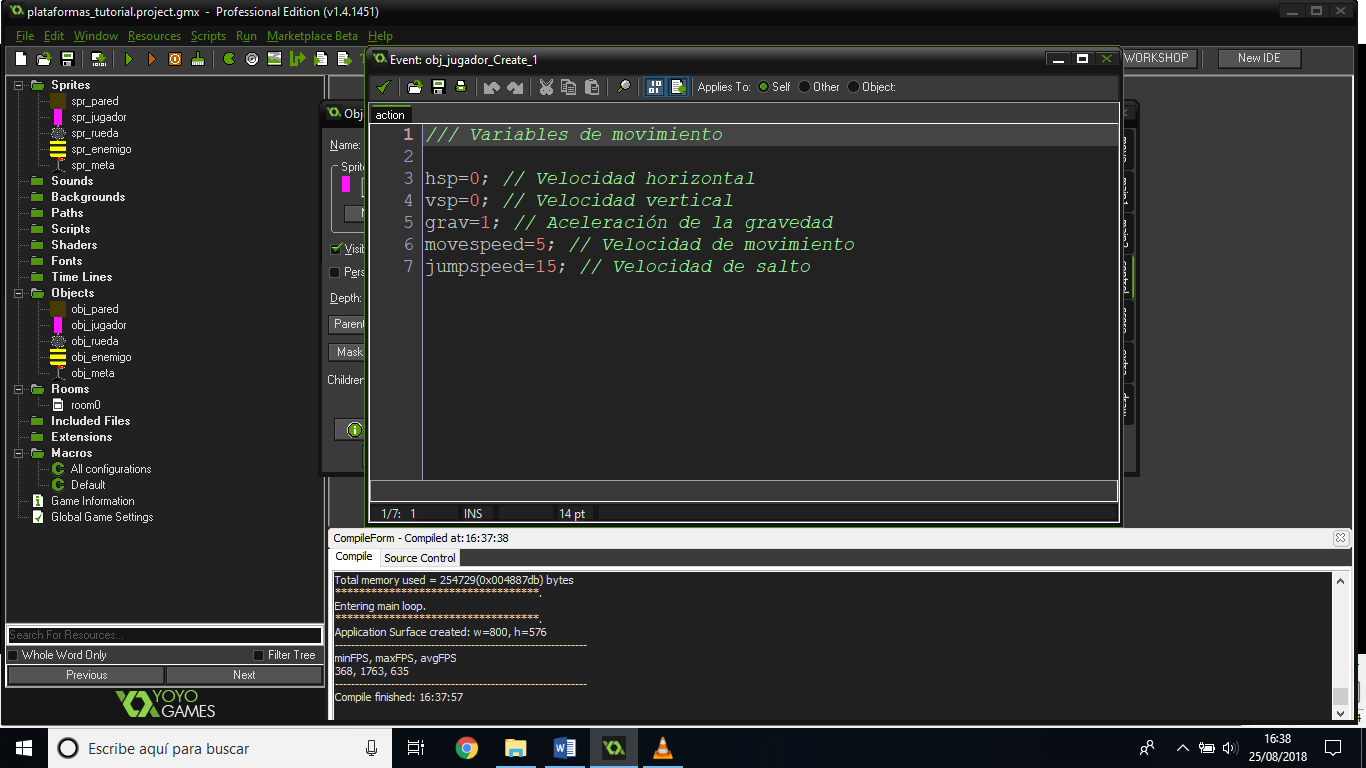
Crear room.

* Establecer dimensiones (ancho y alto).
* Establecer su velocidad (steps por segundo).
* Establecer background o color de fondo.

Dibujarlo, arrastrando los bloques.

## Gravedad y colisiones verticales

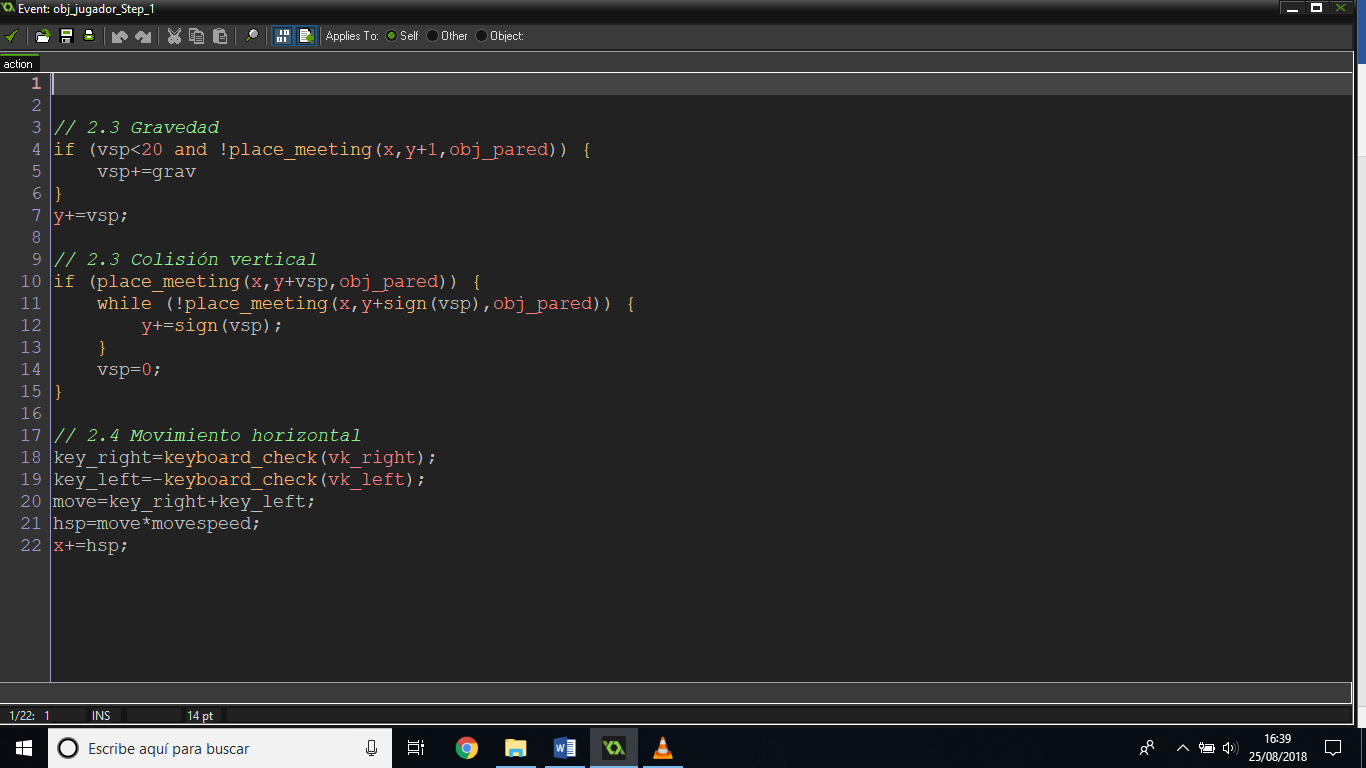
En objeto jugador asignar una serie de variables en el evento Create relacionadas con el movimiento:



Figura

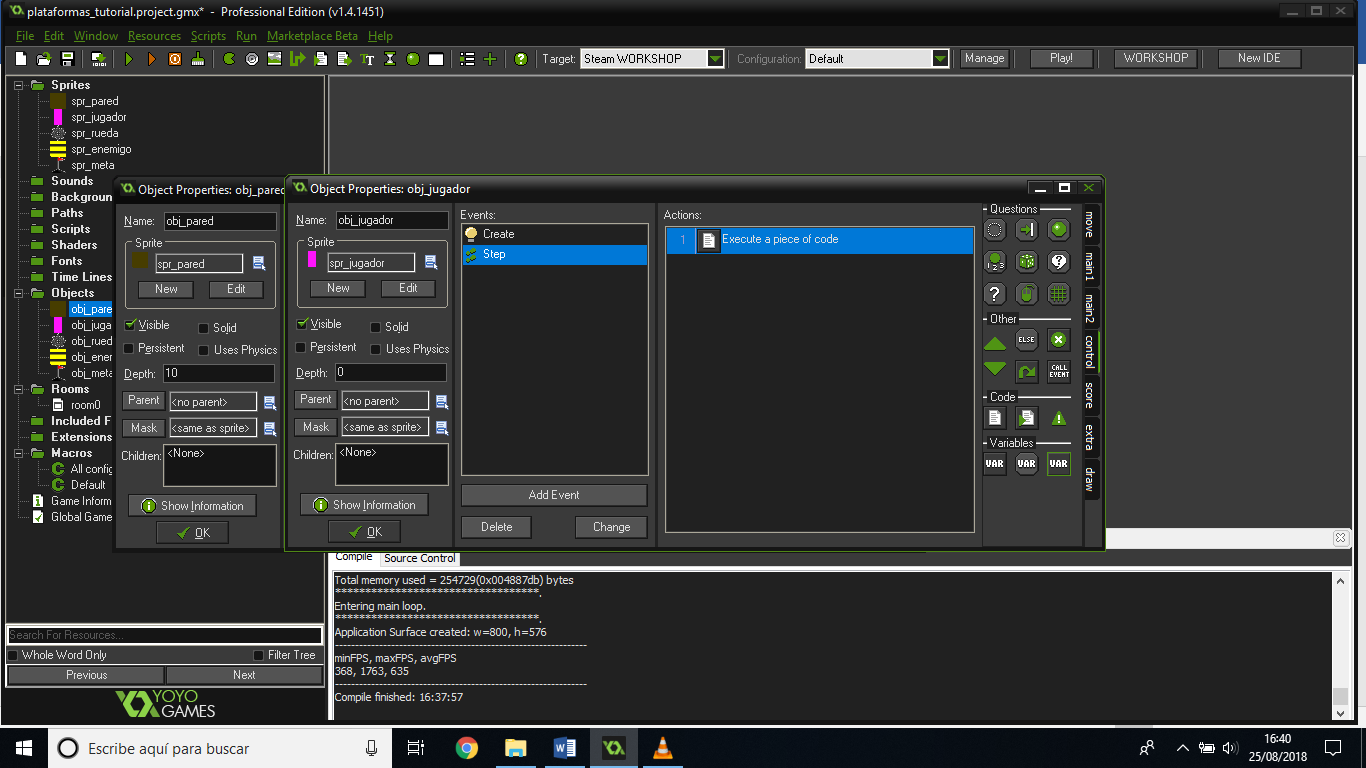
Para añadir la gravedad:

* Limitar la velocidad vertical de caída debida a la gravedad.
* Comprobar que se está en el aire (no hay objetos bloque debajo del jugador). Uso del método place\_meeting(x,y,objeto), que permite detectar colisiones entre la máscara del objeto sobre el que se aplica y el otro que se le pasa al método.



Figura

Recordar que si el jugador cae por detrás de las plataformas y se quiere que quede por delante, lo que se debe hacer es cambiar la profundidad (depth) de los objetos. A mayor profundidad, más detrás queda el objeto en el ordenamiento de capas (éstas puede que no se vean).

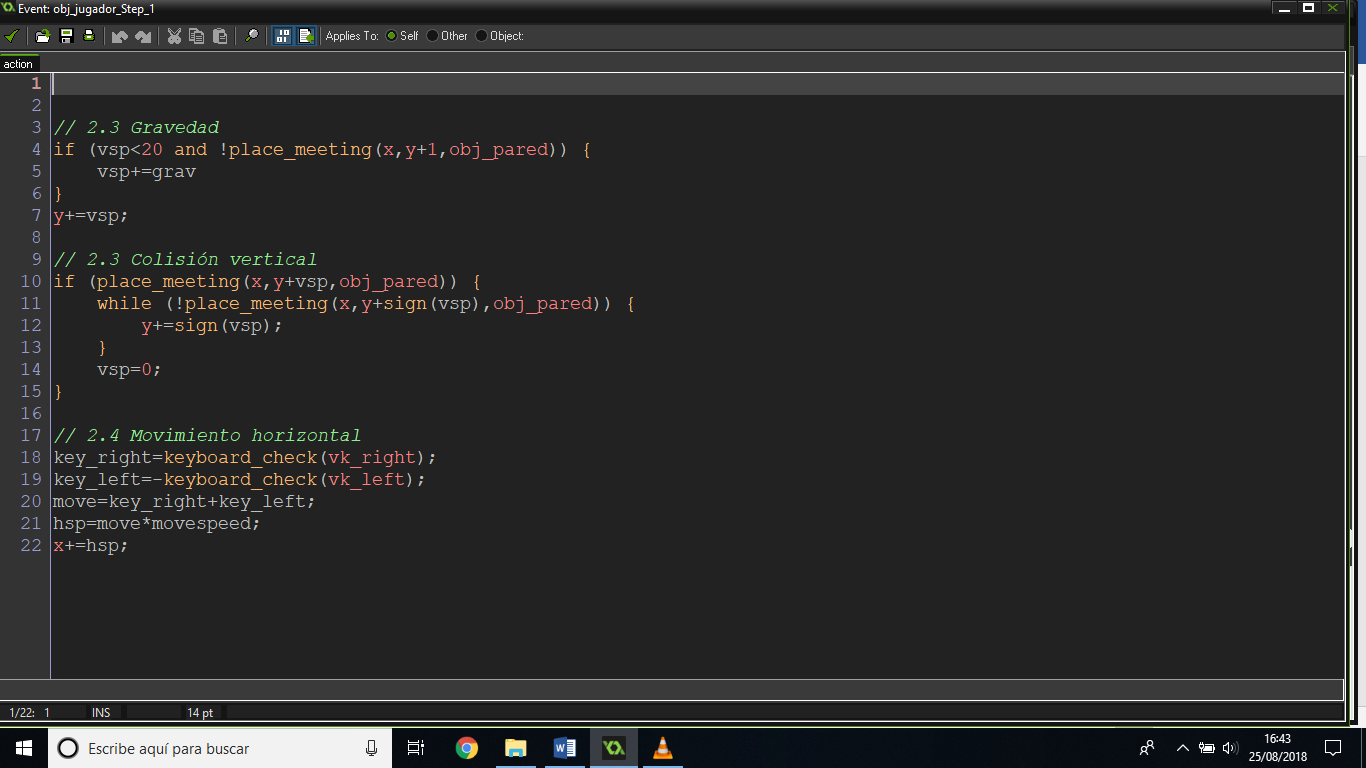


Figura

## Movimiento horizontal

Uso de keyboard\_check().

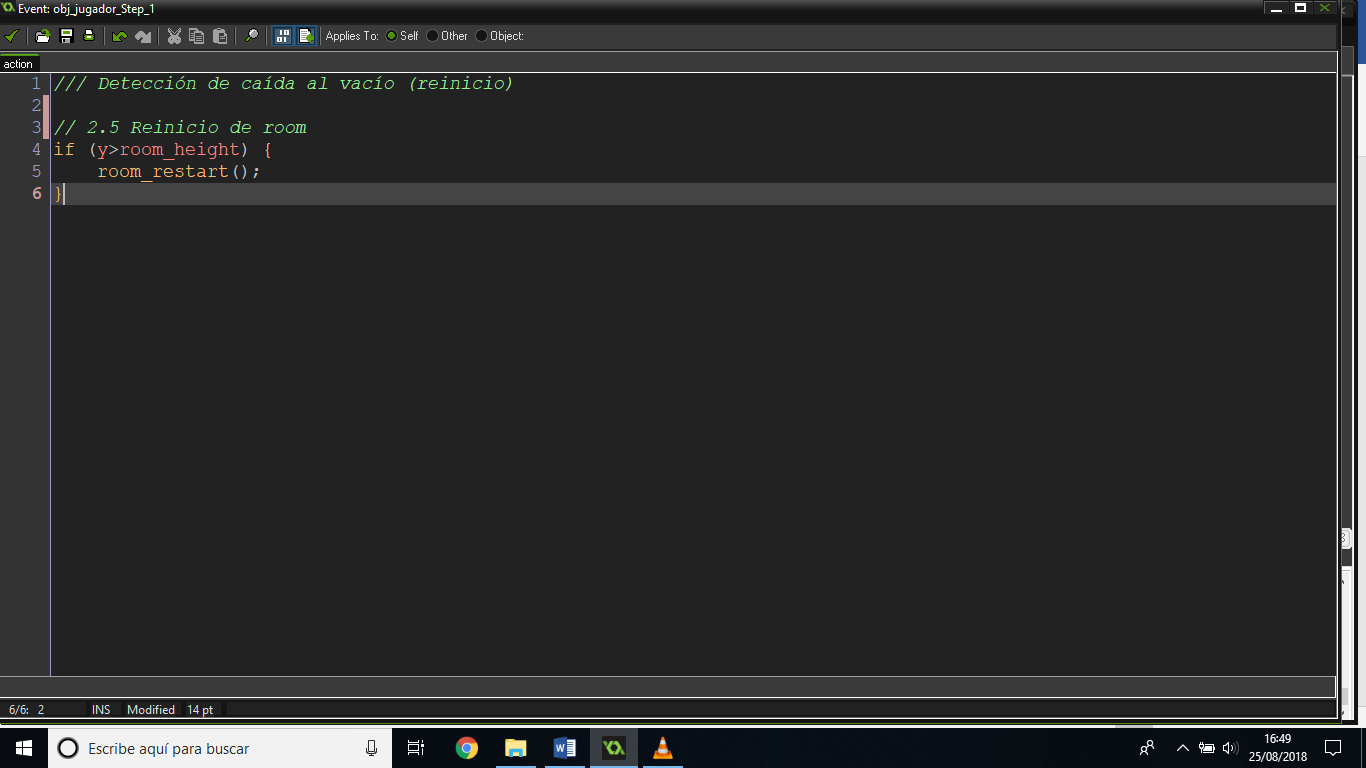
Recordar que true es igual a 1 y false igual a 0.



Figura

## Reinicio de room

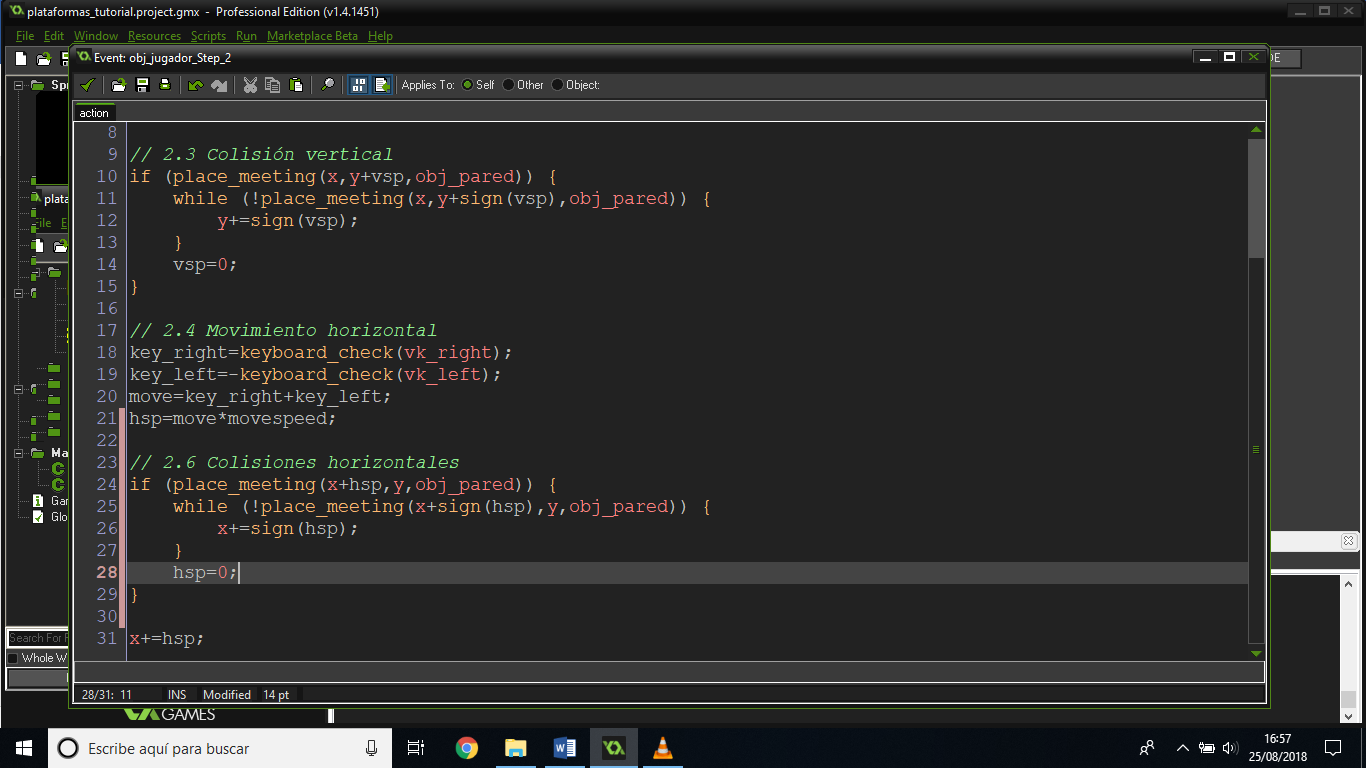
Se plantea reiniciar el juego al caer al vacío ya que no se puede seguir jugando después. Incluir en evento Step del jugador:



Figura

## Colisiones horizontales

Análogo a colisiones verticales. Es importante considerar el orden adecuado en el que se pondrá la instrucción “x+=hsp”.



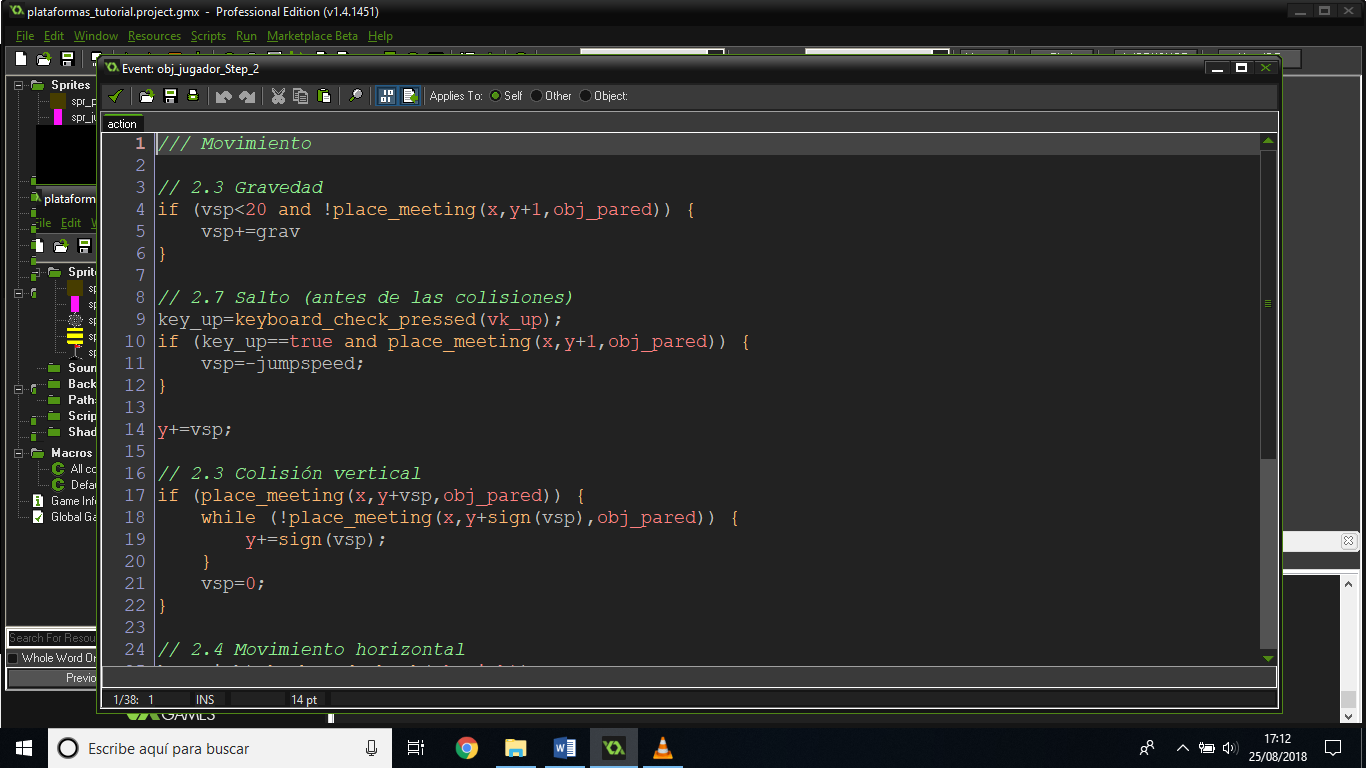
Figura

## Saltos

Se debe considerar:

* Que se esté tocando el suelo.
* Hacerlo antes de las colisiones pues si después se choca con una pared debería de dejar de saltar. Así se cancela la velocidad antes de terminar el step.

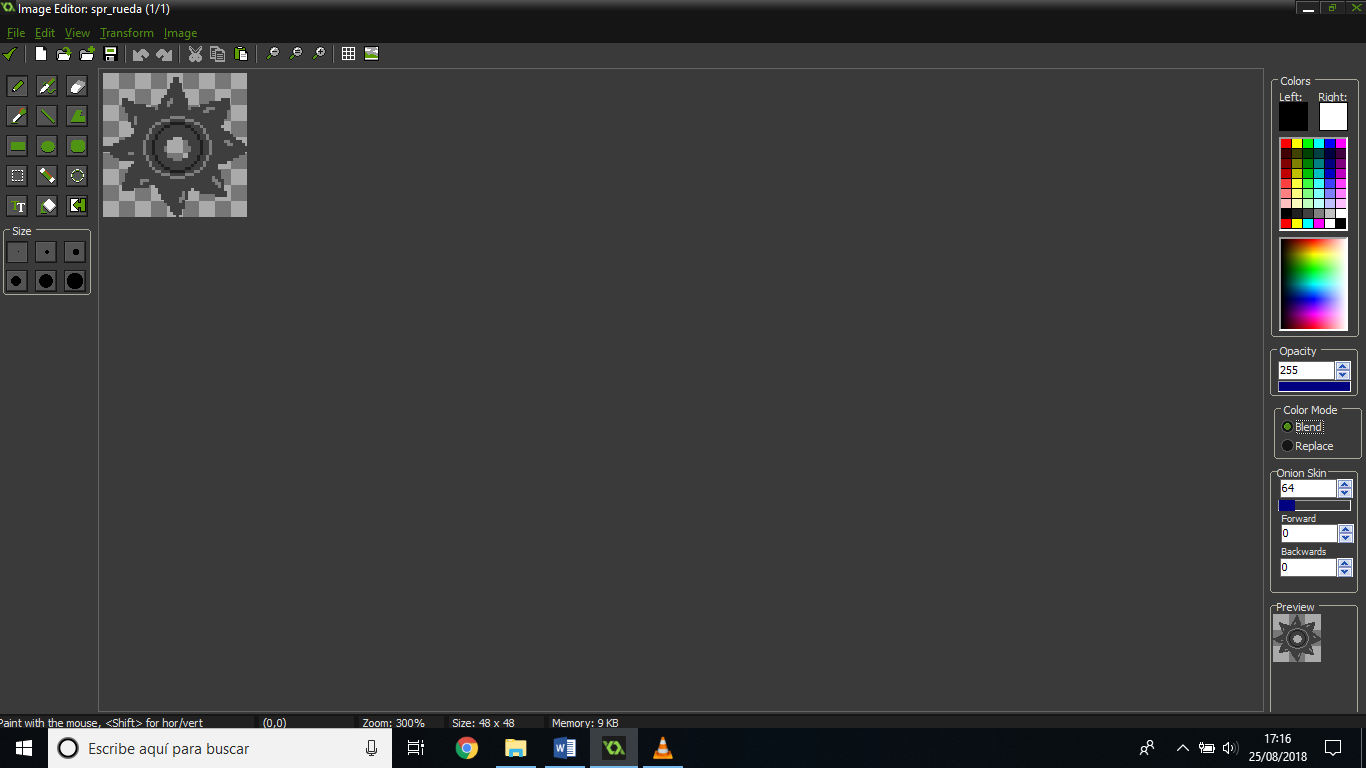
Controlar el lugar donde se coloca la instrucción “y+=vsp”.



Figura

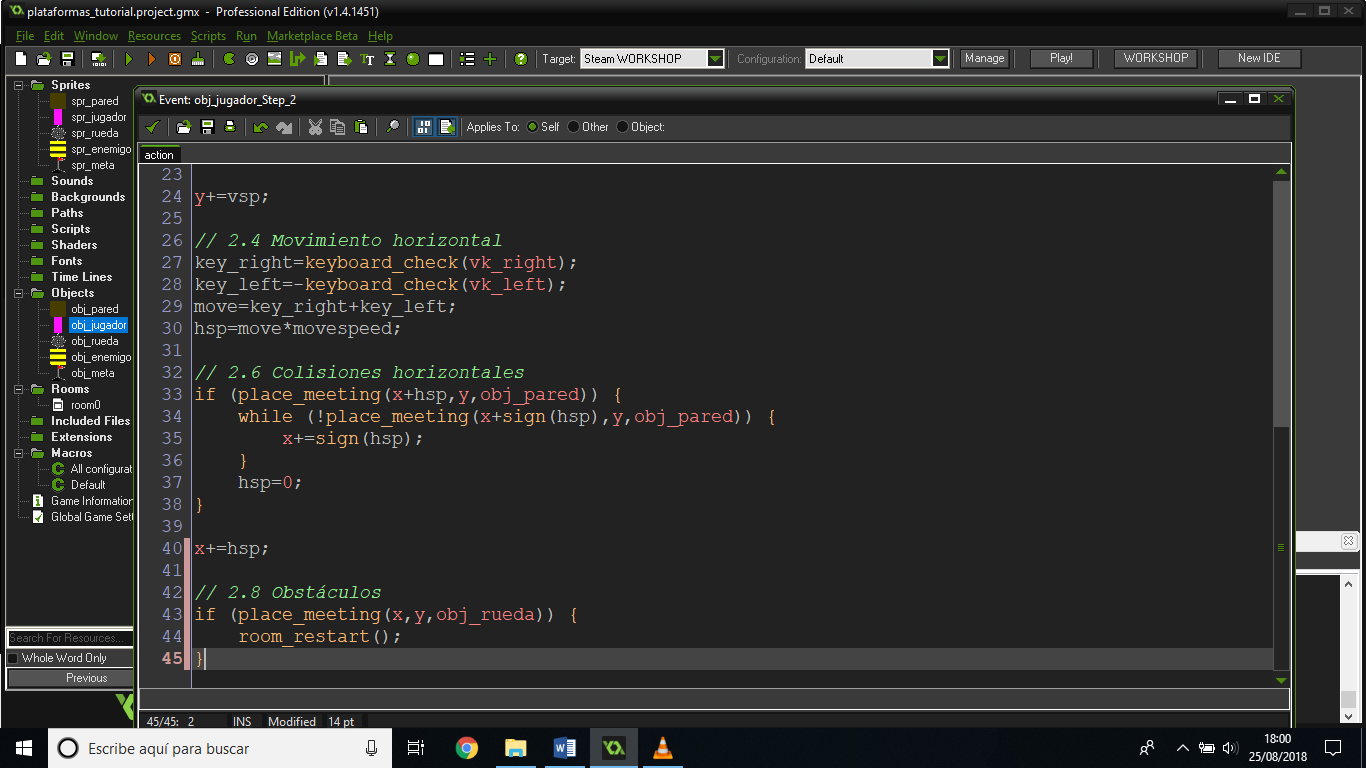
## Obstáculos

Obstáculo es, por ejemplo, el típico pincho que quita vidas al tocarlo.



Figura

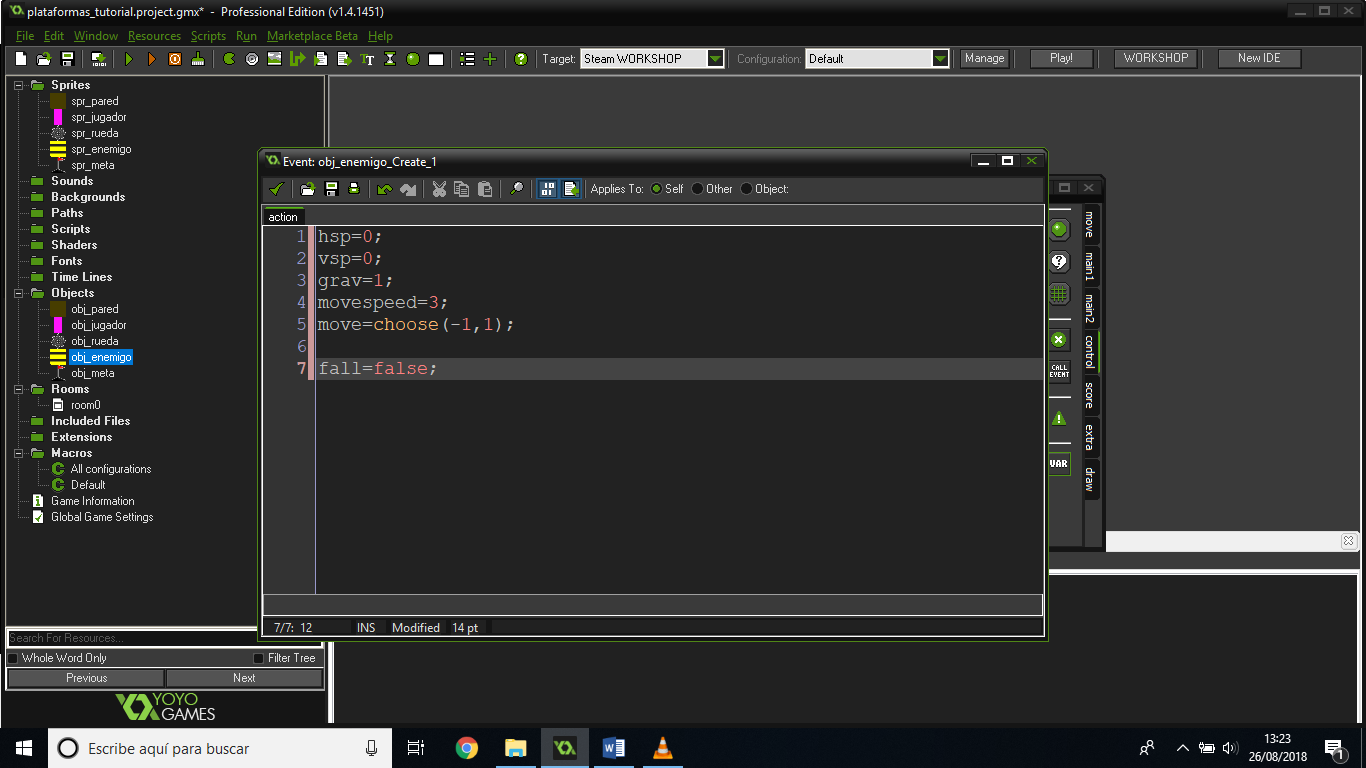
Para dar movimiento al disco, se puede crear la variable de velocidad de giro, centrar el sprite y en cada step variar su posición con image\_angle.



Figura

## Enemigos que se mueven y chocan

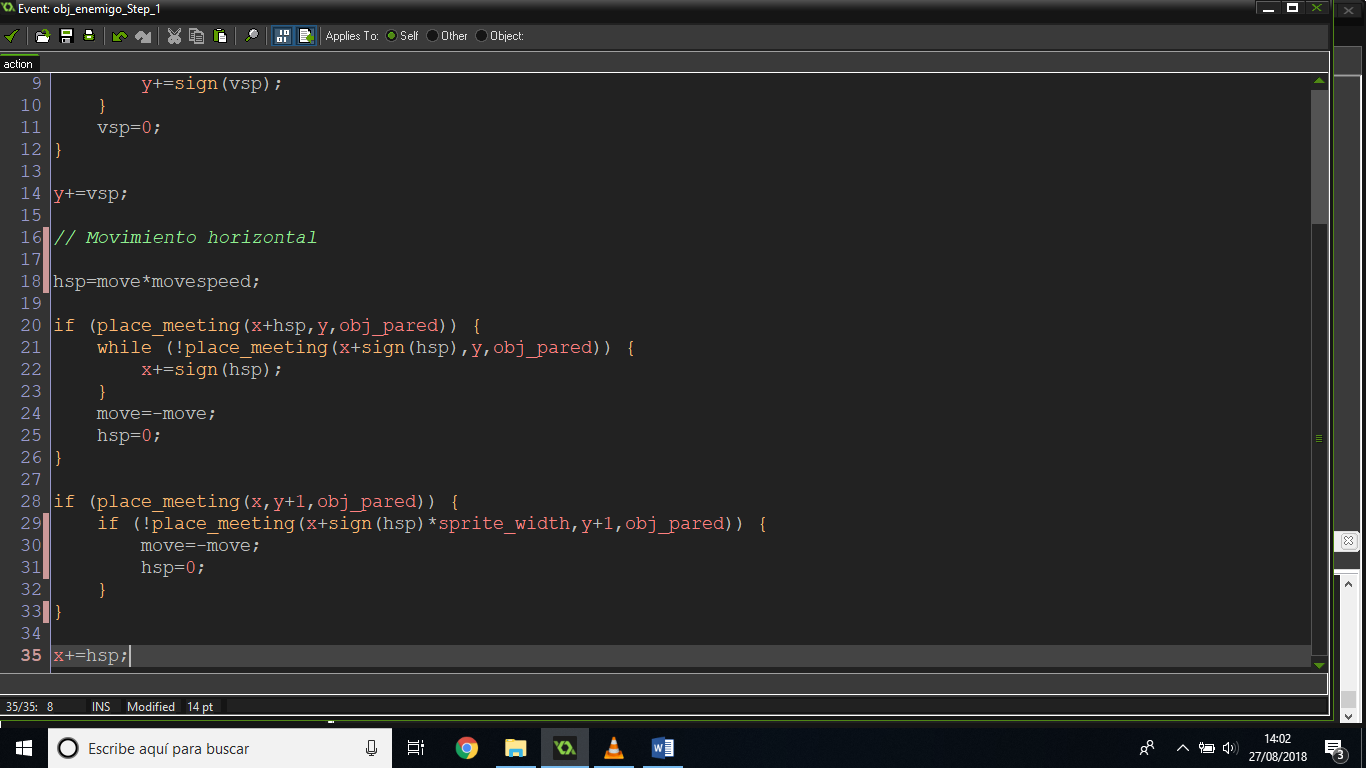
Se crea un objeto Enemigo. Se crea un evento Create en el que se definen una serie de variables.



Figura

Crear para el objeto Enemigo en el evento Step (análogo a lo hecho en el objeto Jugador):

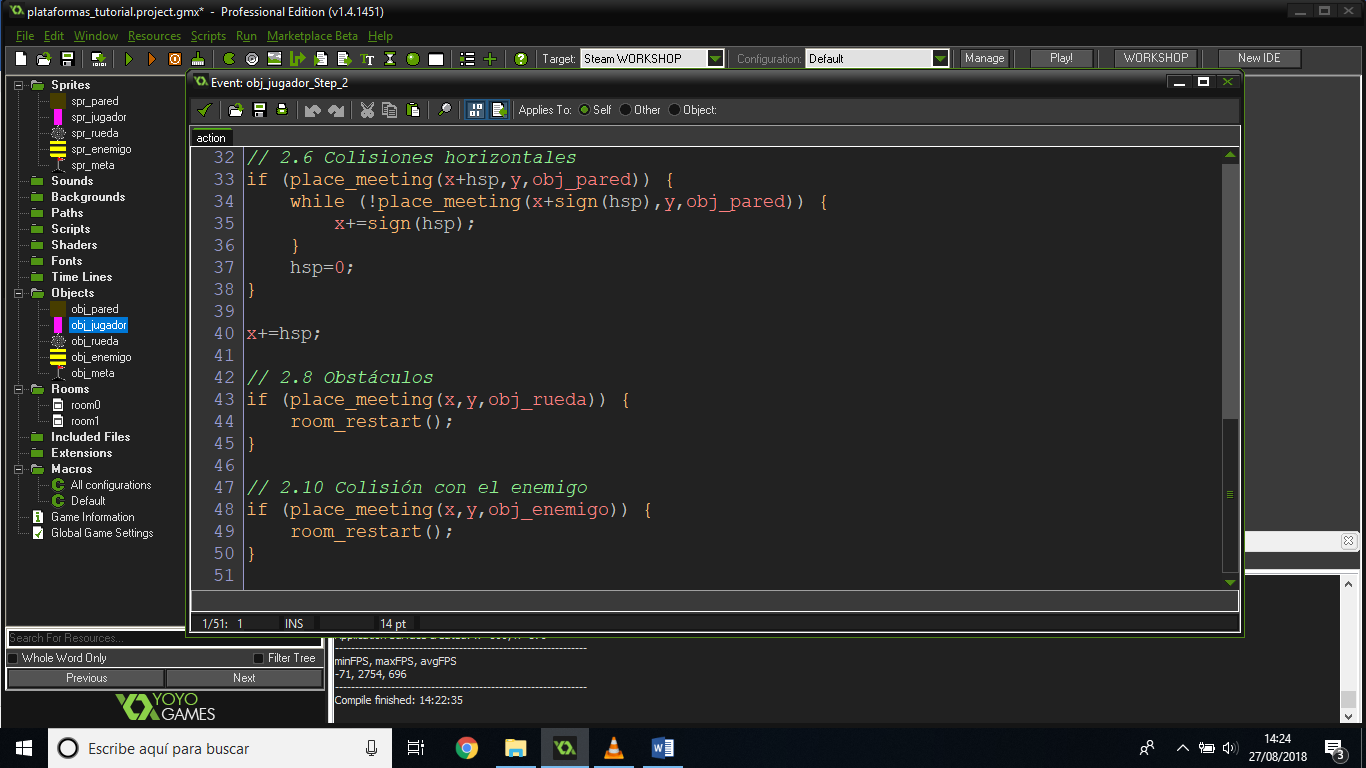
* Gravedad
* Colisión vertical contra el suelo
* Colisiones horizontales. Plantear cambio de dirección al chocar horizontalmente. Para hacer cambio de dirección al llegar al límite de la plataforma poner condición inicial de que se esté tocando el suelo.



Figura

## Enemigos que te matan al tocarlos

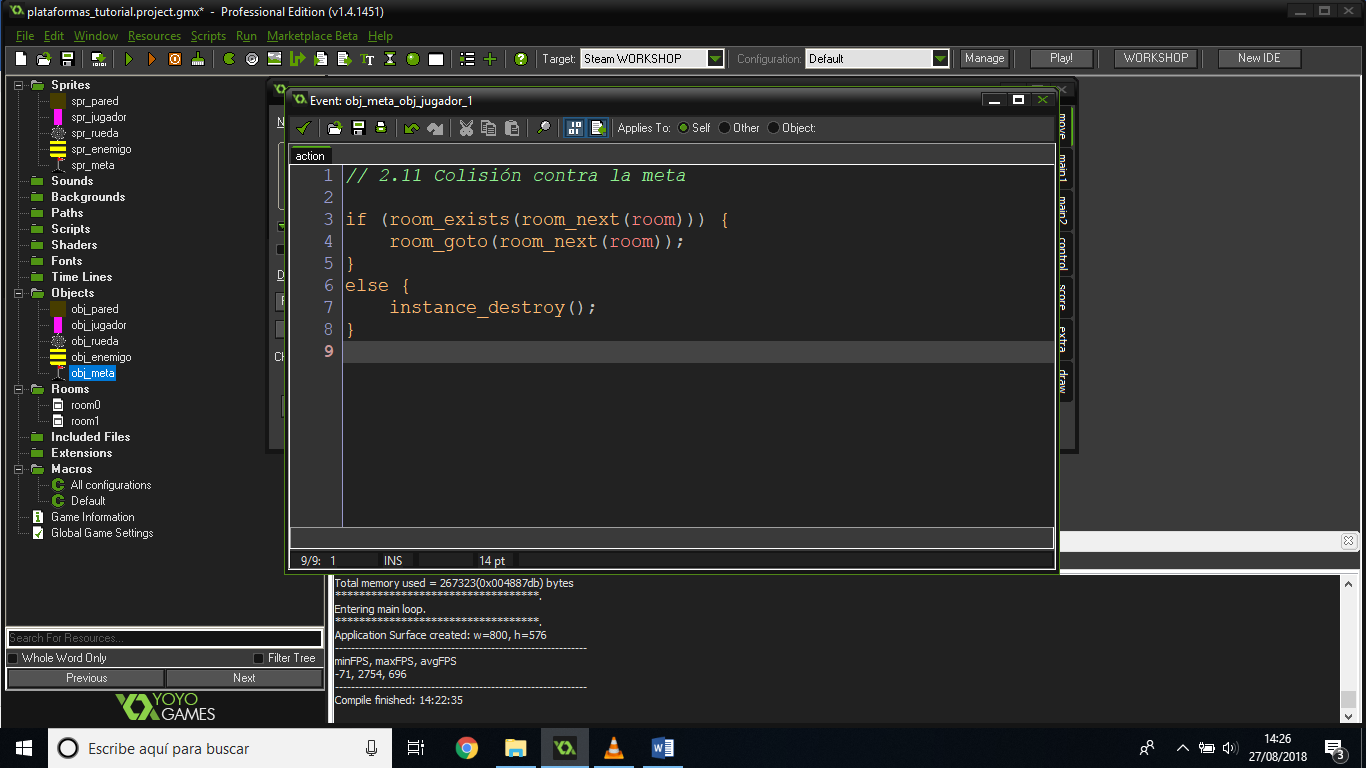
Muy sencillo.



Figura

## Objeto meta

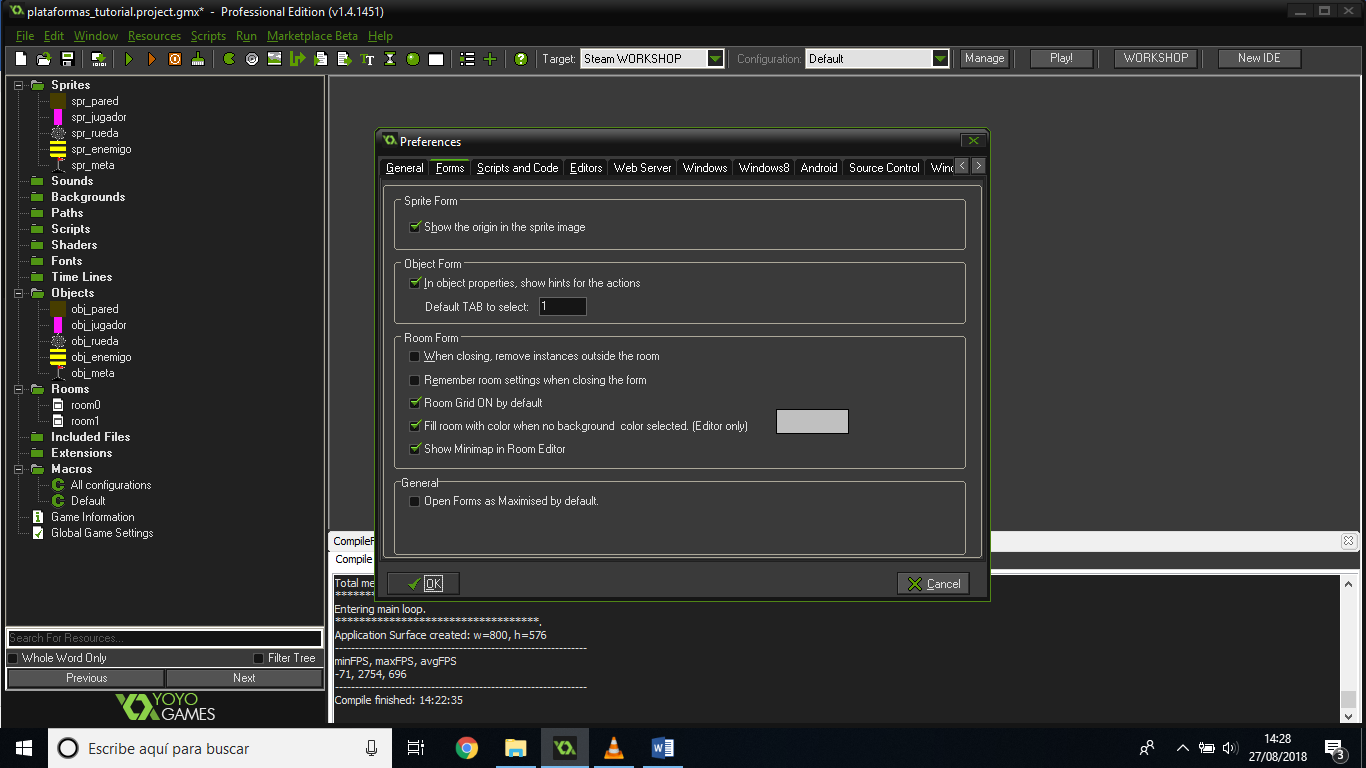
En el evento Colisión del objeto meta poner:



Figura

## Utilidades

Si se quiere impedir la salida de la room se puede poner una pared que no se muestre en pantalla en el exterior. Para evitar que ésta se borre al compilar desactivar la casilla “When closing, remove instance outside the room”.



Figura