

20. Un RPG

El matamarcianos del tema 10 fue un logro considerable. Ahora nos enfrentamos a un reto todavía mayor, y es que gracias a todos los conceptos que hemos estado viendo (Textos, variables, módulos, etc.) podemos ponernos manos a la obra para implementar un videojuego con las características básicas de un RPG.

Antes de empezar con la tarea de programación del RPG es conveniente que introduzcamos algunos conceptos previos que sin duda nos facilitarán mucho la tarea.

20.1 Trabajar con varios FPG

Como ya comentamos en el apartado 18.4 de este tutorial, es posible trabajar con varios FPG si almacenamos en distintas variables de tipo int los identificadores que nos devuelve la función `load_fpg()` cada vez que cargamos un FPG.

Trabajar con varios FPG tiene la misma utilidad que trabajar con varios módulos de código, ya que gracias a ello podremos dividir las tareas gráficas entre varias personas y si mantenemos cada elemento gráfico del videojuego en un FPG distinto nos será mucho más cómodo reutilizarlos más adelante.

A partir de ahora puedes olvidarte de los intervalos numéricos que asignamos a cada uno de los elementos gráficos del videojuego (Ver Tabla del apartado 2.3 de este tutorial). Ahora al tener cada elemento en un FPG separado podemos asignar a cada gráfico el identificador numérico que queramos, aunque generalmente asignaremos los números del 1 en adelante.

Para realizar la carga de nuestros nuevos FPG crearemos un módulo de código muy sencillo que se ocupará de la tarea, pues debemos recordar que cuanto más separemos las distintas partes de nuestro videojuego más fácil se nos hará continuar trabajando con él.

Como novedad, el fichero .prg encargado de la carga de los FPG no lo escribiremos nosotros íntegramente, sino que partiremos del que viene incluido en el videojuego de ejemplo asociado a este tema. Obviamente tendremos que modificarlo parcialmente, así que vamos a explicar un poco su contenido:

<p>GLOBAL</p>	<p>Tendremos un vector GLOBAL de tipo int al que llamaremos fpgs [].</p> <p>En cada una de las posiciones de este vector almacenaremos cada uno de los identificadores de los FPGs que cargaremos, por tanto debemos hacer un vector con posiciones suficientes para todos ellos.</p>
<p>CONST</p>	<p>La bloque CONST es una zona especial que todavía no habíamos visto, pero que introduciremos poco a poco a partir de ahora porque nos será útil para facilitar ciertas cosas.</p> <p>Una CONST no es más que un determinado texto que el compilador sustituye por el valor que queramos (Un número entero, un real, una string...).</p> <p>En nuestro caso tenemos un vector de identificadores y en cada una de sus posiciones tenemos el identificador de un FPG determinado. Observa que si usamos CONST nos será mucho más sencillo referirnos a fpgs [_SONIC] ó fpgs [_SHADOW] que referirnos a fpgs [0] ó fpgs [1].</p> <p>Es por ello que hemos definido una serie de CONST que asocian el nombre del personaje a cada posición del vector. Observa que añadimos el carácter '_' al inicio del nombre de cada CONST para evitar que sus nombres puedan coincidir con los nombres de algún proceso. También observa que las escribimos en mayúsculas, esto nos servirá para NO confundirlas con variables, ya que las CONST no pueden cambiar de valor, son constantes.</p> <p><i>Nota: El compilador exige que el bloque CONST se escriba antes del bloque GLOBAL, en esta tabla se explica después porque el vector GLOBAL es el que nos crea la necesidad de usar CONST y así es más fácil comprender lo que se explica.</i></p>
<p>PROCESS</p>	<p>Finalmente y con todo lo anterior, tenemos un proceso llamado inicializar_fpgs () que se encargará de ejecutar tantos load_fpg () como sea necesario para cargar todos nuestros FPGs y almacenar el identificador de cada uno en su posición correspondiente del vector fpgs [].</p>

Añade los FPGs que creas convenientes al directorio /images de tu videojuego, copia el fichero de código fpg.prg en el directorio /prg de tu videojuego y procede a modificar su código para que se adapte a tu videojuego y los nuevos FPGs que has incluido en él.

Es importante que no olvides añadir la correspondiente sentencia INCLUDE para que el fichero de código fpg.prg también sea compilado, y ten en cuenta que el proceso inicializar_fpgs () requerirá ser invocado para que se carguen los FPGs.

Finalmente, para que los procesos ya existentes en tu videojuego sepan cuál es el FPG que deben usar para localizar su gráfico será necesario que a partir de ahora **siempre asignes a su aspecto file el identificador que les corresponde**, por ejemplo.

```
file = fpgs [ _SONIC ];
```

20.2 Modificando un poco el enemigo

Nuestro proceso enemigo actual ha quedado obsoleto, apenas hace poco más que moverse de manera lineal por la pantalla y no atiende a la existencia de nuestro protagonista.

Todavía no vamos a entrar en temas de inteligencia artificial ni mucho menos, pero sí vamos a conseguir que al menos el enemigo haga lo posible por perseguir a nuestro protagonista, al menos cuando éste esté cerca.

Para ello vamos a eliminar todas las instrucciones del enemigo referentes a su movimiento y dirección, y en su lugar vamos a implementar su movimiento en base a 4 nuevas funciones de Bennu que nos ayudarán bastante, especialmente a la hora de implementar comportamientos más complejos:

Función	Parámetro	Devuelve
exists ()	De la misma forma que la función collision (), esta función recibe como parámetro la palabra type seguida de un espacio y el nombre del proceso sobre el cuál queremos comprobar si existe al menos uno de ellos activo actualmente en el videojuego.	Devuelve verdad (TRUE ó 1) activando por tanto una condición dentro de un IF si existe al menos un proceso de ese tipo activo en el videojuego. En otro caso devuelve falso (FALSE ó 0).
get_id ()	De la misma forma que exists () recibe como parámetro la palabra type seguida de un espacio y el nombre del proceso del cuál queremos obtener un identificador.	Devuelve un identificador de proceso de cualquiera de los procesos activos del tipo dado como parámetro, si no hay ninguno nos devolverá 0. Generalmente primero haremos la comprobación de si hay algún proceso activo mediante exists (), y en caso afirmativo usaremos get_id (). Si hay varios procesos activos del mismo tipo nos devolverá el identificador de cualquiera de ellos, y si la invocamos de manera consecutiva durante el mismo FRAME nos retornará los identificadores de los distintos procesos uno por uno, hasta terminar y devolver 0.
get_angle ()	Recibe como parámetro un identificador de proceso activo, obtenido mediante get_id (), por ejemplo.	Devuelve el ángulo en milésimas de grado que forma el proceso que ha invocado get_angle () con el proceso cuyo identificador ha sido pasado como parámetro. Si asignamos ese ángulo a angle podemos mantener al proceso mirando al proceso dado como parámetro.

Función	Parámetro	Devuelve
advance ()	<p>Recibe como parámetro un valor de tipo int cuya utilidad es indicar el avance en píxels que queremos que realice en pantalla el proceso que invoca esta función.</p> <p>El avance se realizará en la dirección del ángulo que indica su aspecto angle. Es muy útil para realizar desplazamientos que no vayan exclusivamente en el eje x o en el y.</p>	No devuelve ningún valor útil, simplemente tiene el efecto mencionado en la columna Parámetro sobre el proceso que lo invoca.

El uso adecuado de las 4 funciones nos permitirá conseguir que el enemigo siempre persiga al protagonista, dotándolo de un mínimo de inteligencia.

Si el proceso enemigo tiene una variable PRIVATE de tipo entero a la que llamaremos objetivo (int objetivo;) podemos hacer lo siguiente:

```

IF ( exists ( type protagonista )
    objetivo = get_id ( type protagonista );
    angle = get_angle ( objetivo );
    advance ( 2 );
END
// Condición: Si hay un proceso protagonista activo
// Obtenemos su identificador de proceso
// Obtenemos el ángulo que formamos con él
// Avanzmos 2 píxeles en ese ángulo
// Fin de la condición
  
```

Recuerda que el identificador de un proceso es como una estructura o TYPE. Mediante el operador '.' nos permitirá acceder a cualquiera de sus aspectos básicos como pueden ser graph, x, y, etc.

Si comprobamos adecuadamente la diferencia entre la x del enemigo y la x del protagonista y entre la y del enemigo y la y del protagonista podremos conseguir que los enemigos sólo avancen en la dirección del protagonista cuando éste se encuentre suficientemente cerca.

Intenta implementar también este comportamiento en tu videojuego. Sería aconsejable que añadas x e y como parámetros al proceso enemigo para que a partir de ahora sea invocado en posiciones aleatorias de todo el scroll, no únicamente en las esquinas del mismo.

20.3 Práctica: Programar personajes amigos y diálogos

La característica principal de un RPG es la presencia de personajes no jugadores con los que podemos interactuar.

En nuestro caso vamos a crear un nuevo módulo para gestionar este tipo de personajes y en él incluiremos un par de personajes muy sencillos con los que podremos dialogar cuando nos acerquemos a ellos y pulsemos la tecla Enter. Ésto podrás programarlo fácilmente si haces uso de las funciones `collision()` y `key()`.

El primero de los personajes se encargará de curar a nuestro protagonista cuando su vida no se encuentre al máximo. También nos dirá una frase de entre 3 distintas que almacenaremos en un vector de strings.

El segundo de los personajes se encargará de aumentar nuestra salud máxima cada vez que nuestra puntuación supere los 100 puntos.

Intenta programar este comportamiento. Recuerda que la vida de nuestro protagonista es una de sus variables `PRIVATE` y que por tanto ningún otro proceso es capaz de modificar su valor. Será nuestro protagonista quien por su parte detectará la colisión con el personaje amigo y se ocupará de modificar los valores.

Por supuesto Benu ofrece soluciones mucho más elegantes que lo anterior para modificar las variables de un proceso, pero al nivel en el que nos encontramos tenemos suficiente con saber programarlo de esa manera.