

23. Animaciones II

En algunas ocasiones conviene utilizar un fichero de código realizado por otra persona para solucionar una tarea compleja. En el caso de la animación, este tema tiene un fichero llamado animacion.prg que incluye todos los procesos, tipos de dato, constantes, etc. Necesarias para gestionar animaciones complejas.

Si tienes conocimientos suficientes sobre tipos de dato es probable que comprendas su funcionamiento viendo su código, pero dado que introduciremos algún concepto nuevo sobre tipos de dato es recomendable que atiendas a lo que se explica en este tema sobre **matrices**.

23.1 Una estructura de datos apropiada

Para gestionar todas las posibles animaciones de todos los posibles personajes de nuestro videojuego necesitaremos almacenar información sobre varias animaciones, que pueden llegar a ser realmente muchas.

Además, almacenando únicamente el gráfico inicial y final de la animación no tenemos suficiente, ya que habrás observado que cambiar de gráfico a cada FRAME hace que la animación parezca demasiado acelerada, sería recomendable mantener información sobre la latencia en FRAMES que debemos esperar antes de cambiar de gráfico.

El tipo de dato que utilizaremos para almacenar la animación será el siguiente:

```
TYPE tp_animacion
    int graph_ini;
    int graph_fin;
    int latencia;
END
```

Y para almacenar todas las animaciones que necesitaremos conviene que distingamos entre los diferentes personajes o elementos que animaremos y sus diferentes estados de animación. Si suponemos que cada personaje viene numerado como 0, 1, 2, etc. y cada animación también viene numerada como 0, 1, 2, etc. Nos viene a la mente que podemos usar un **vector**, pero en realidad usaremos una **matriz**, que no es más que un vector de vectores y que nos será realmente útil.

Para definir una matriz de animaciones podemos hacerlo así:

```
GLOBAL
    tp_animacion animaciones [ 16 ] [ 16 ];
```

Observa que sólo con añadir un nuevo [] podemos aumentar las dimensiones de la estructura de datos. En este caso es una matriz, pero con un [] adicional podríamos hacer un cubo, y con otro []... Tendríamos una estructura de datos difícil de representar mentalmente.

Hemos definido una matriz donde almacenar hasta 16 personajes con hasta 16 animaciones distintas. ¿Serán suficientes para nuestro videojuego?

Es recomendable que definamos 2 valores constantes donde guardar el número de personajes y de estados, y que usemos las constantes para definir el tamaño de la matriz:

```
CONST
    _MAX_PERSONAJES = 16;
    _MAX_ESTADOS = 16;

GLOBAL
    tp_animacion animaciones [ _MAX_PERSONAJES ] [ _MAX_ESTADOS ];
```

¿Cómo utilizamos una estructura de datos tan compleja?

A partir de ahora cada personaje tendrá una variable PRIVATE entera donde almacenar el número que indica su nombre y otra donde almacenar el número que indica su estado. El personaje hará uso de esas variables para acceder a la posición de la matriz donde se encuentra la información de la animación.

Por ejemplo, si tenemos un personaje numerado con 3 y que se encuentra en su animación 8 bastará con que asignemos a las variables nombre y estado lo siguiente:

```
nombre = 3;
estado = 8;
```

Y accedamos a la siguiente posición de la matriz:

```
animaciones [ nombre ] [ estado ]
```

Para encontrar los datos referentes a la animación que debe realizar.

Puede que te parezca demasiada complicación para realizar simples animaciones, pero debes pensar que podemos llegar a tener una gran cantidad de personajes y animaciones distintas y que sin duda esta estructura de datos nos facilitará enormemente trabajar con ellos.

23.2 Constantes para indexar

Hasta ahora, gracias a las variables PRIVATE de cada personaje que nos indican su nombre y su estado de animación podíamos acceder a la posición de la matriz donde se encuentra su animación. Pero tenemos un pequeño problema, y es que resultará complicado estar pensando en cuál número indica el nombre del personaje que estamos programando y cuál número indica el estado de animación que queremos que tome.

Para ello podemos hacer un buen uso de las CONST, ya que no son más que sustituciones textuales y nos servirán para poder hablar de `_SONIC` o de `_SHADOW` en lugar de usar un 1 o un 2 para determinar el nombre del personaje. De la misma manera podemos proceder con los estados de animación, por ejemplo:

```
CONST
    _SONIC = 0;
    _SHADOW = 1;

    _QUIETO = 0;
    _ANDAR = 1;
    _GOLPEAR = 2;
```

Gracias a lo anterior, cuando queramos que nuestro personaje Sonic realice la animación de andar bastará con que accedamos a:

```
animaciones [ _SONIC ] [ _ANDAR ]
```

Para obtener los datos de la animación.

Para mayor aclaración sobre lo expuesto en este punto, la tarea de acceder a una determinada posición de un vector o de una matriz se denomina **indexar** y es algo que resulta cómo hacer utilizando una variable o una constante.

23.3 El proceso animar

Como ya comentamos en el tema anterior, animar será una acción que todo personaje estará realizando todo el tiempo, y ya no sólo dependerá de que pulsemos alguna tecla, sino que siempre tendremos a nuestro personaje animándose, bien sea cuando esté quieto, cuando esté andando o cuando esté realizando cualquier otra acción.

Por eso vamos a implementar un PROCESS que accederá a father.graph y a la matriz de animaciones y que será invocado por todo personaje, en todo FRAME, para que se ocupe de su animación.

Hemos dicho que todo personaje tendrá una variable PRIVATE con su nombre y otra con su estado, por ejemplo:

```
PRIVATE
    int nombre = _SONIC;
    int estado = _QUIETO;
```

Y serán esas variables las que pasaremos como parámetro a la función animar para que indexe la matriz de animaciones y nos imponga en gráfico que nos corresponde, la función quedaría así:

```
PROCESS animar ( int nombre , int estado )
BEGIN
    father . graph++;
    IF ( father . graph > animaciones [ nombre ] [ estado ] . graph_fin OR
        father . graph < animaciones [ nombre ] [ estado ] . graph_ini )
        father . graph = animaciones [ nombre ] [ estado ] . graph_ini;
    END
END
```

La idea es la siguiente: Siempre incrementamos en 1 nuestro gráfico, y si para el nombre y el estado que tenemos nuestro gráfico NO se encuentra dentro del intervalo de animación correcto, entonces restauramos el gráfico inicial.

¿Cómo hacemos uso correcto de la función animar ()?

La invocaremos incondicionalmente en cada FRAME y le pasaremos nuestro nombre y nuestro estado. Si procedemos así bastará con cambiar el valor de nuestra variable estado para que la función animar haga que nos animemos dentro del nuevo intervalo de animación, simplificando la tarea infinitamente, un ejemplo de cambio de animación entre los estados _QUIETO y _ANDAR sería:

```
LOOP
...
animar ( nombre , estado );
...
IF ( key ( _LEFT ) )
...
    estado = _ANDAR;
ELSIF ( key ( _RIGHT ) )
...
    estado = _ANDAR;
ELSE
    estado = _QUIETO;
END
...
END
```

Observa que ya no tenemos que volver a preocuparnos por el valor de nuestra variable **graph**, sino que sólo tenemos que preocuparnos por el valor de nuestra variable **estado**, ya que el proceso animar () hará el resto.

23.4 La latencia de animación

Al principio de este tema hemos dicho que lo deseable para nuestra animación sería que no se realizase en cada FRAME sino que tuviese en cuenta una latencia de animación de forma que sólo se cambiase el valor de graph cada varios FRAMES.

Para que esto funcione tendremos que modificar ligeramente el proceso animar () de forma que haga uso del campo latencia de la matriz de animaciones.

¿Cómo podemos hacer que algo suceda “cada N FRAMES”?

Podemos tener una variable GLOBAL llamada tiempo que comience valiendo 0 y el proceso arbitro () la incremente en 1 a cada FRAME. Si hacemos esto, además de tener controlados los FRAMES que han transcurrido desde el inicio de la ejecución, también podemos hacer otras cosas útiles.

Observa que si dividimos el tiempo entre la latencia de animación (División de números enteros) siempre tenemos un resto que vale 0 exactamente cada vez que transcurren tantos FRAMES como indica la latencia.

Por ejemplo para latencia 4, si vamos dividiendo el tiempo entre 4 a cada FRAME sólo obtenemos resto 0 cada 4 FRAMES, esta idea podría servirnos, ¿Pero cómo obtenemos el resto de una división?

Hay una operación llamada módulo que aunque no se explica en los estudios básicos de matemáticas tiene una gran utilidad en programación, ya que obtiene el resto de una división, y no el cociente que es lo habitual. El operador que realiza esta operación en Bennu es '%' y podemos usarlo dentro de nuestro proceso animar así:

```
PROCESS animar ( int nombre , int estado )
BEGIN

    IF ( tiempo % animaciones [ nombre ] [ estado ] . latencia == 0 )
        father . graph++;
    END

    IF ( father . graph > animaciones [ nombre ] [ estado ] . graph_fin OR
        father . graph < animaciones [ nombre ] [ estado ] . graph_ini )
        father . graph = animaciones [ nombre ] [ estado ] . graph_ini;
    END

END
```

El operador % también tiene otras aplicaciones como conseguir que nuestro personaje no pueda disparar una vez por FRAME, si no que tenga una latencia de disparo que sólo le permita disparar cada N FRAMES. Es algo que puedes intentar implementar en tu videojuego.

23.5 Inicializar las animaciones

Ya hemos explicado qué es lo que almacenará nuestra matriz de animaciones y cómo haremos uso de ella gracias al proceso animar (), pero nos falta un detalle importante, y es que la matriz de animaciones no ha sido inicializada y no almacena ningún valor.

Tendremos que completar todos los campos de la matriz con los valores de gráfico inicial y final y la latencia que queramos para cada animación de cada personaje. Es una tarea aburrida, pero no te tomará mucho tiempo.

Para evitar "ensuciar" nuestro código haremos un nuevo PROCESS en animacion.prg que se encargará de inicializar las animaciones con los valores que deseemos. El proceso será así:

```
PROCESS inicializar_animaciones ( )  
BEGIN  
    animaciones [ _SONIC ] [ _QUIETO ] . graph_ini = 1;  
    animaciones [ _SONIC ] [ _QUIETO ] . graph_fin = 9;  
    animaciones [ _SONIC ] [ _QUIETO ] . latencia = 3;  
  
    animaciones [ _SONIC ] [ _ANDAR ] . graph_ini = 10;  
    animaciones [ _SONIC ] [ _ANDAR ] . graph_fin = 21;  
    animaciones [ _SONIC ] [ _ANDAR ] . latencia = 3;  
  
    ...  
END
```

Ante todo no olvides invocar este proceso al inicio de tu videojuego, ya que en otro caso la matriz de animaciones estará vacía y todos los personajes que la utilicen permanecerán con `graph = 0`;

23.6 Latencia por defecto

Observarás que generalmente el valor de la latencia para todas las animaciones suele ser el mismo, especialmente si todos los personajes proceden de sprites del mismo videojuego.

Para evitar tener que especificar la misma latencia para todas las animaciones de todos los personajes, podemos dar una latencia por defecto de forma que, si no especificamos ningún valor para la latencia de una animación, ésta tome el valor de latencia por defecto.

Para ello podemos asignar el valor por defecto en la declaración del TYPE `tp_animacion`, así:

```
TYPE tp_animacion  
    int graph_ini;  
    int graph_fin;  
    int latencia = 3;  
  
END
```

Con lo anterior, no será necesario que especifiquemos la latencia para cada animación, sino que bastará con especificar el valor de las latencias que no vayan a ser 3 exactamente.

23.7 Una máquina de estados

Hemos simplificado bastante el trabajo con animaciones, ya que ahora sólo tenemos que preocuparnos por el valor de la variable estado de cada personaje.

¿Pero cuándo debemos modificar el valor de la variable estado? ¿Cuáles son las condiciones que deben cumplirse para que el funcionamiento sea totalmente correcto?

Es complicado determinar los cambios de estado, y de hecho lo que necesitaremos programar se denomina formalmente **máquina de estados**.

Una máquina de estados conviene ser dibujada para comprender su funcionamiento, y por eso en este tema tenemos asociado un diagrama en el que se muestran 3 distintos estados y todas las condiciones que deberíamos programar para conseguir que nuestro personaje cambiase entre los estados básicos `_QUIETO`, `_ANDAR` y `_GOLPEAR`.

Observa cómo hemos diseñado la máquina de estados y especialmente presta atención a que cada nuevo estado de animación implica tantas nuevas condiciones de cambio de estado como estados anteriores teníamos... Esto es un indicador de que cada nuevo estado que incluyamos, en el peor de los casos podrá llegar a duplicar el número de condicionales necesarios.