Librería TriEspacio

Manual técnico y de usuario

(c) 2002 Baltasar el arquero, baltasarq@yahoo.es

1 Introducción

La presente librería, TriEspacio, permite el movimiento de un objeto en un espacio de tres dimensiones.
Las aplicaciones son diversas. Un desierto por cruzar, un mar en el que encontrar una isla de un tesoro, el espacio, el fondo del mar ... son lugares donde el distribuir partes por localidades no es completamente satisfactorio.
La aproximación ideal es guardar las coordenadas del personaje en el espacio, x, y, z, hasta llegar a una coordenada concreta, una zona o un obstáculo.
Los ficheros de la librería son *triespacio.h,* y *triespacio.pdf*, la presente documentación
El fichero *triespacio.h* debe incluirse antes de ser utilizado en el código del juego. Puede encontrarse una demo acerca de cómo utilizar la librería en *demoespacio.inf*.

2 Cómo funciona

2.1 Introducción

La librería se basa en dos clases claramente diferenciadas: espacio y ente. El primero sirve para definir objetos que se mueven en un espacio, mientras el segundo es el espacio en sí mismo. Así, como en el ejemplo en *demoespacio.inf*, hay un espacio (el mar), por el que se mueven dos entes (la barca y el tiburón). Un juego que utilice esta librería deberá declarar al menos un objeto espacio en su programa. Por ese espacio, pueden moverse varios entes, de hecho, tanto como se desee. No existen límites de memoria por el uso de más o menos objetos, o por el uso de unas coordenadas máximas en el espacio de 10x10x10 o de 100x100x100. Eso sí, un juego con un objeto espacio y un objeto ente ocupará menos memoria que un juego con un objeto espacio y dos objetos ente, tal y como intuitivamente se puede imaginar.

2.2 La clase espacio

La clase espacio se utiliza creando un objeto de la misma:

```
espacio mar
private
      maxx 10,
      maxy 10,
      maxz 1
;
```

Así, se se ha creado un objeto mar, de 10 posiciones de ancho por 10 de largo, y con sólo 1 posición de altura. Es decir, hemos creado un plano.

La clase espacio soporta pocas posibilidades más. Una de ellas es la de reaccionar ante la llegada del jugador a los límites. Para saber que objeto ha sido el que ha roto los límites, se utiliza *dev_UltMovil()*.

```
espacio mar
private
      maxx 10,
      maxy 10,
      maxz 1
with
      siYesMaxY [;
            if (self.dev_UltMovil() == barca)
                  "Por fín … has llegado a la isla.";
      ]
;
```

Es posible también definir obstáculos y zonas, de forma que sea posible detectar cuando el jugador llega a ellas, a través de la llamada a *fueZona()* y *fueObst()*.

```
object obs
class array
private
      elementos
      ! obstáculo número 0
            3 7    ! Longitud en el eje x: De 3 a 7 en X
                   !(el centro)
            5 6    ! Longitud en el eje y: De 5 a 6 en Y (1 de largo)
            0 1    ! El total de abajo arriba.
      ,
```

```
      dimx 1,      ! 1 obstáculo
      dimy 6       ! siempre son 6 coordenadas por obstáculo
;

espacio mar
private
      obstaculos obs,
      maxx 10,
      maxy 10,
      maxz 1
;
```

Ahora, existe un obstáculo en el centro del mapa, lo cuál impedirá que la barca llegue a su objetivo. Los entes que se topen con él retrocederán a la posición anterior. La llamada *fueObst()* podrá ser utilizada. Sólo existe un obstáculo, que es, por tanto, el número 0.

De la misma forma, pueden declararse zonas, que no impiden que el ente las atraviese, pero que son detectables mediante *fueZona()*.

2.3 La clase ente

Ahora sólo es necesario crear objetos ente que puedan moverse por el objeto espacio que hemos creado. Para ello, sólo debemos asegurarnos de utilizar el mar para desplazarse:

```
      ente barca
      private
            mapa mar
      with
            x 5,
            y 0,
            z 0
      ;
```

Con esta programación tan sencilla ya están creados los objetos que se necesitan. Un mar de 10x10 y una barca situada en el límite inferior, en la mitad del ancho. Sólo es necesario hacer que el jugador entre en la barca en algún momento (véase el código *demoespacio.inf* para saber varios detalles de cómo hacerlo), y que la barca reaccione a ciertas posibilidades. Por ejemplo, es posible que deseemos visualizar un mensaje cada vez que la barca toque tierra:

```
      ente barca
      private
            mapa mar
      with
            x 5,
            y 0,
            z 0,
            siYesMaxY [;
                  "Notas como el fondo del bote roza la arena.";
            ],
            siYesMinY [;
                  "La quilla roza el fondo de la costa que abandonaste
en un principio.";
            ]
      ;
```

A continuación, un código que, desde una localidad normal, permite que el jugador se "suba" al bote.

```
    Object playa "Playa"
    with descripcion "Es una playa paradisíaca. Al norte, puedes ver el
mar y un bote.",
```

```
      al_n jugadorA(barca, 2),
      al_s banderafin = 1;
    has luz;
    Object isla "Isla"
    with descripcion "Desde aquí puedes ver la playa de la que
partiste.",
      al_s jugadorA(barca, 2),
      al_n banderafin = 2;
    has luz;
```

También debe ser posible "bajar" del bote ...

```
    ente barca
      private
            mapa mar
      with
            nombre 'barca',
            x 5,
            y 0,
            z 0,
            siYesMaxY [;
                  "Notas como el fondo del bote roza la arena.";
            ],
            siYesMinY [;
                  "La quilla roza el fondo de la costa que abandonaste
en un principio.";
            ],
            antes [;
                  bajar: if ((self.y > (mar.devMaxY() - 2))
                              { jugadorA(isla, 2); rtrue; }
                         if (self.y < 2)
                              { jugadorA(playa, 2) rtrue; }
                         print "Te ahogas sin remedio ...^";
                        banderafin = 1;
            ]
      has femenino luz;
```

Este ejemplo, mucho más ampliado, puede verse en el código demoespacio.inf. Se aconseja al lector interesado una lectura concienzuda del mismo. La explicación hasta ahora puede servir como introducción al código.

3 Documentación técnica

A continuación, se incluyen las partes más relevantes del código con amplios comentarios.

```
   Clase que definirá los objetos que simulan un mar, un desierto, ...
etc.

class espacio
with
! ---------------------- Límites ---------------------------------
! Máximas coordenadas del espacio
      devMaxx [; return self.maxx; ],
      devMaxy [; return self.maxy; ],
      devMaxz [; return self.maxz; ],

! ---------------------- Acciones cuando se rompe algún límite ---
Las siguientes acciones pueden redefinirse en las clases hijas u
objetos , de forma que se pueda reaccionar en el caso de que se llegue
al máximo o mínimo X, Y y Z. Para saber qué objeto móvil se ha movido,
llámese a la función dev_UltMovil().
      siXesMaxX [; rtrue; ],  ! En principio, no hacen nada
      siYesMaxY [; rtrue; ],
      siZesMaxZ [; rtrue; ],

      siXesMinX [; rtrue; ],
      siYesMinY [; rtrue; ],
      siZesMinZ [; rtrue; ],

Las siguientes acciones pueden ser redefinidas en las clases hijas
para detectar cuando se llega a un obstáculo o a una zona.
      siZonaEnc [; rtrue; ],
      siObstEnc [; rtrue; ],

Cuando se llega a una zona o un obstáculo, estas funciones devuelven
el número de obstáculo o zona encontrada, o -1 en caso de que no se
encontrara ninguno. No deben ser redefinidas.
      dev_fueZona [; return self.fueZona; ],
      dev_fueObst [; return self.fueObst; ],

La siguiente función devuelve el último móvil que se desplazó en el
espacio.
      dev_ultMovil [; return self.ult_movil; ],

;

! ---------------------------------------------------------------
! Clase ente, la que se mueve por el mapa.
! ---------------------------------------------------------------

class ente

with
! -------------- Acciones a cumplir si se cumple alguna condición ----
Las siguientes acciones pueden ser redefinidas por los objetos que se
creen de esta clase, y son ejecutadas cuando se rompe algún límite
máximo o mínimo, cuando se llega a un obstáculo o cuando se alcanza
una zona.
      siZesMaxz [; return false; ],
```

```
      siXesMaxx [; return false; ],
      siYesMaxy [; return false; ],
      siZesMinz [; return false; ],
      siXesMinx [; return false; ],
      siYesMiny [; return false; ],
      siObstEnc [; return false; ],
      siZonaEnc [; return false; ],

! -------------- Movimiento del ente ---------------------------------
```
Las siguientes funciones devuelven la última posición del ente antes de moverse. Esto es especialmente útil después de un movimiento absoluto.

```
      dev_ultX [; return self.ultx; ],
      dev_ultY [; return self.ulty; ],
      dev_ultZ [; return self.ultz; ],
```

La dirección del movimiento de un ente se codifica como un número, que puede ser:
0,1,2,3, para n,s,e,o.
4,5,6,7 para ne,no,se,so
8,9 para arriba/abajo.
10 significa que el último movimiento fue absoluto.
11 significa que se retrocedió.
La siguiente función devuelve el último movimiento efectuado siguiento la anterior codificación.

```
      dev_ultmov [; return self.ultmov; ],
```

La siguiente propiedad es utilizada para indicar cuantas posiciones avanza el ente en cada ocasión. Puede ser modificada en cualquier momento.

```
      accel 1,                      ! Lo rápido que va el ente
```

Las coordenadas actuales del ente. Jamás deberían modificarse directamente, a no ser que se sepa exactamente lo que se está haciendo. Puede ser útil redefinirlas, sin embargo, para darle al ente una posición de inicio diferente de la 0,0,0.

```
      x 0,
      y 0,
      z 0,
```

La siguiente propiedad puede modificarse para indicar cuando bloquear al ente.

```
      puede_moverse true,     ! puede hacer movimientos ?
```

Las siguientes acciones gestionan el movimiento relativo y absoluto. En cuanto al relativo, sólo se le pasa la dirección (codificada como ya se ha visto). La cantidad de posiciones a moverse viene definida por accel.
Es posible llamar directamente a estas funciones, sobre todo en el caso de PSI's, que obviamente no responden a las órdenes del jugador.

```
      movimientoRel [ dir;
            self.posRel(dir);
            return self.siMueve(self.mapa.mueve(self));
      ],

      movimientoAbs [ x y z;
            self.posAbs(x, y, z);
            return self.siMueve(self.mapa.mueve(self));
      ],
```

Éstas son las direcciones de movimiento. Si el jugador está en este objeto y teclea 'norte', lo que en realidad sucede es que el ente se mueve en el mapa. Pueden redefinirse para evitar el movimiento en alguna dirección, por ejemplo.

```
      al_s  [; return self.movimientoRel(1);],
      al_n  [; return self.movimientoRel(0);],
      al_e  [; return self.movimientoRel(2);],
      al_o  [; return self.movimientoRel(3);],

      al_no [; return self.movimientoRel(5);],
      al_ne [; return self.movimientoRel(4);],
      al_se [; return self.movimientoRel(6);],
      al_so [; return self.movimientoRel(7);],

      arriba [; return self.movimientoRel(8);],
      abajo  [; return self.movimientoRel(9);],
```

La siguiente acción es siempre ejecutada cada vez que ha habido un intento de mover al ente. 'movido' es true si el ente finalmente se movió, y 'false' en otro caso. Puede ser redefinida en los objetos de esta clase.

```
      siMueve [ movido; return movido; ]
;
```