Informática Gráfica II

Práctica (Parte I)

Curso 25/26

`Desarrollo de un videojuego utilizando Ogre3D (Parte I)`

En esta práctica vamos a desarrollar un videojuego – sencillo – utilizando el motor de renderizado Ogre3D. La práctica se desarrollará de forma incremental, es decir, cada apartado aportará una nueva funcionalidad al juego. Por ello, es recomendable realizar los apartados en el orden en el que están descritos en el enunciado. Además, en clase de teoría, se explicará cada apartado en detalle.

Para facilitar el desarrollo de la práctica, se proporciona la clase – ya implementada – `IG2Object`. Esta clase puede utilizarse como base para desarrollar nuevos elementos del juego como, por ejemplo, un personaje o un bloque del escenario.

En esencia, el juego está basado en el clásico Bomberman, donde controlaremos a un personaje (*héroe)* cuyo objetivo es eliminar – colocando bombas – a todos los enemigos (*villanos)* en un laberinto. Los *villanos* pueden estar controlados por el ordenador, o por otro usuario si se desea, siendo esto último opcional para la práctica. Cuando un *villano* toca al jugador, éste perderá una vida. El jugador gana si finaliza todas las fases del juego, y pierde en caso contrario.

Aspectos generales para el desarrollo:

- Utilizad el objeto `IG2Object` y modificadlo si lo creéis necesario.
- Tened en cuenta la estructura de clases. Un buen diseño puede ahorrar tiempo y esfuerzo.
- Considerad el uso de una clase para definir constantes estáticas, como por ejemplo las teclas del juego, los caracteres utilizados para codificar laberintos, etc.
- Seguramente, la estructura de las clases desarrolladas cambie – al realizar apartados posteriores – para adaptarse a los requisitos del enunciado, aunque no deberían ser cambios significativos. Por ejemplo, la lógica encargada de realizar la carga de un laberinto.
- Considerad el método `seputpScene()` de la clase `IG2App` como el punto de partida para el juego. Podéis cambiarle el nombre si lo consideráis necesario.
- Todas las mejoras que se añadan a la práctica serán tenidas en cuenta para la nota final. Por ejemplo, implementar un gestor de laberintos que controle cada fase del juego, desarrollar varias personalidades para los enemigos, añadir efectos adicionales a las explosiones de las bombas, etc.

Apartado 1. Los laberintos

La primera parte del juego consistirá en crear un laberinto en un entorno 3D. Para ello usaremos la malla `cube.mesh`. Combinando distintas entidades con esta malla, formaremos un laberinto. La idea es modelar cada laberinto como una cuadrícula, donde cada posición puede tener, o bien un *muro*, o un *hueco*. Cuando en una posición exista un *muro*, pondremos un `sceneNode` con la entidad `cube.mesh`. En caso contrario, pondremos el `SceneNode` sin entidad. En las posiciones donde no exista un muro, podremos colocar las bombas.

Los laberintos se leerán de ficheros de texto, y se crearán en tiempo de ejecución. De esta forma, podremos crear y probar distintos laberintos de forma sencilla. Tened en cuenta que la definición de un laberinto evolucionará según avancemos con la práctica, de forma que será necesario modificar el método encargado de crear el laberinto. Por ejemplo, en este primer apartado, crearemos únicamente *muros*, pero en el siguiente apartado, también leeremos del fichero la posición inicial del *héroe*.

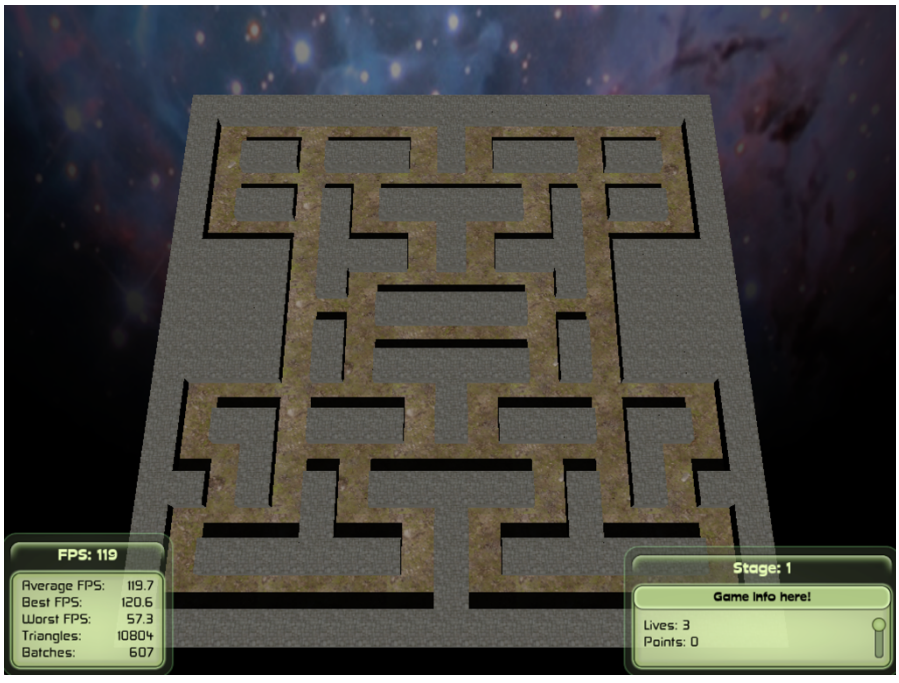
El formato del fichero de texto se describe a continuación:

```
NumFilas
NumColumnas
caracteresFila_1
caracteresFila_2
…
caracteresFila_NumFilas
```

donde `NumFilas` representa el número de filas del laberinto, `NumColumnas` representa el número de columnas del laberinto, y `caracteresFila_i` representa la secuencia de caracteres de la fila i-ésima, donde cada carácter puede tener los valores 'x' y 'o' que definen, respectivamente, un *muro* y una posición vacía.

El siguiente ejemplo muestra el contenido de un fichero que representa el mapa mostrado:

```
19
19
xxxxxxxxxxxxxxxxxxx
xooooooooxoooooooox
xoxxoxxxoxoxxxoxxox
xooooooooooooooooox
xoxxoxoxxxxxoxoxxox
xooooxoooxoooxoooox
xxxxoxxxoxoxxxoxxxx
xxxxoxooooooooxxxxx
xxxxooooxxxxxoooxxx
xxxxoxooooooooxxxxx
xxxxoxoxxxxxoxoxxxx
xooooooooxoooooooox
xoxxoxxxoxoxxxoxxox
xooxoooooooooooxoox
xxoxoxoxxxxxoxoxoxx
xoooooooxoooxooooox
xoxxxxxxoxoxxxxxxox
xooooooooxoooooooox
xxxxxxxxxxxxxxxxxxx
```



En la imagen anterior se muestra el laberinto con suelo y con texturas para que sea más fácil distinguir los bloques de los huecos. Sin embargo, todavía no pondremos texturas en el juego, únicamente los bloques.

En las posiciones vacías (sin *muro*) es por donde podrá moverse el *héroe*. Antes de codificar esta funcionalidad, pensad en la estructura de clases. A priori, parece sensato definir una clase para representar el laberinto, y otra para representar al *héroe*. Además, tanto el *muro* como la la posición vacía podrán ser clases que heredarán de `IG2Object`, aunque cada una tiene una particularidad. Por ejemplo, un *muro* no puede atravesarse, pero la posición vacía (sin *muro*) sí.

Apartado 2. Movimiento del *héroe* por el laberinto

En este apartado moveremos a nuestro *héroe* por el laberinto. La idea es que consigamos que se mueva por los huecos donde no haya *muros* y no pase a través de ellos. Con este objetivo, podemos utilizar, por ejemplo, las flechas. El tipo de datos para representar una tecla es `OgreBites::Keycode` y los valores son, para este caso, `SDLK_UP`, `SDLK_DOWN`, `SDLK_LEFT` y `SDLK_RIGHT`.

Para representar al *héroe* en el juego utilizaremos a Sinbad, a quien ya conocemos de las clases anteriores. Sin embargo, de forma opcional, se permite utilizar, además de Sinbad, a otro personaje jugable. Inicialmente el *héroe* empieza con tres vidas y cero puntos. Esta información la mostraremos por pantalla en el apartado 5 de la práctica.

La posición del *héroe* en el tablero se indicará en el fichero que lo define. De esta forma, añadimos el carácter 'h' para representar su posición inicial.

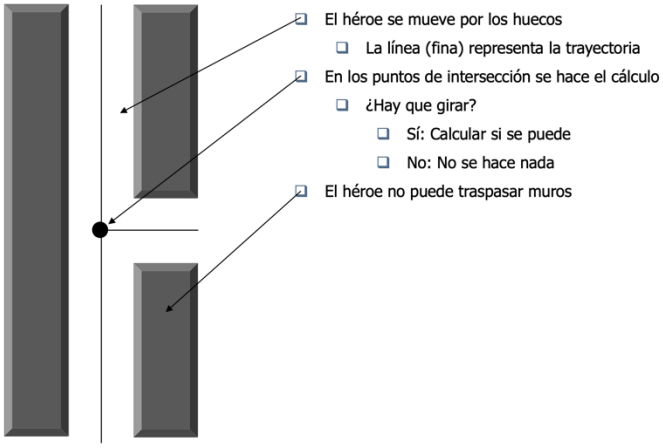
Para que el *héroe* reaccione a la pulsación de las teclas, debemos implementar el método `keypressed` (en la clase que represente al *héroe*) y actualizar en este método el nuevo movimiento a realizar. Además, en algún punto del programa – donde creamos la instancia del *héroe* – debemos añadir como observador (*listener*) al *héroe* para que reciba los eventos del teclado. Esto se consigue con el método `addInputListener`, como ya hemos visto en ejercicios anteriores en la clase `IG2App`.

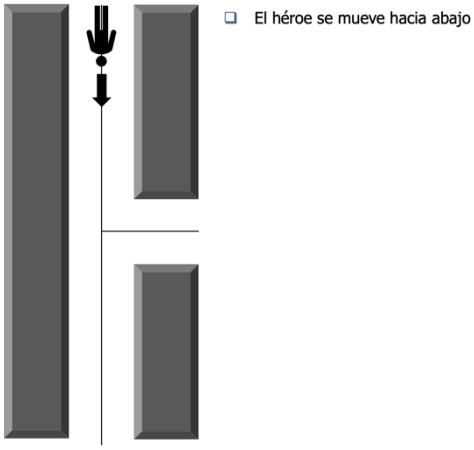
Para realizar el movimiento del *héroe* utilizaremos la traslación, para avanzar, y la rotación (sobre su eje Y), para girar. Además, debemos guardar en alguna variable el nuevo movimiento introducido por el usuario, debiendo calcular en el juego cuándo será posible realizarlo. Por ejemplo, si le indicamos que queremos que se mueva hacia la derecha, no podrá realizar ese movimiento hasta que exista un pasillo en esa dirección, de forma que el *héroe* seguirá moviéndose sin modificar su dirección actual. Es decir, el *héroe* solo se detiene cuando encuentra un *muro* enfrente, y solo gira si se ha indicado previamente el movimiento y existe un camino libre en esa dirección. El movimiento del *héroe* se realizará en función del tiempo transcurrido entre el proceso de *render* entre dos *frames*. De esta forma, garantizamos que la velocidad del *héroe* será la misma, independientemente del equipo donde ejecutemos el juego. Recordad que el método `frameRendered` recibe como parámetro de entrada un evento `evt` de tipo `Ogre::FrameEvent&`, de forma que podemos calcular este tiempo a través de `evt.timeSinceLastFrame`.

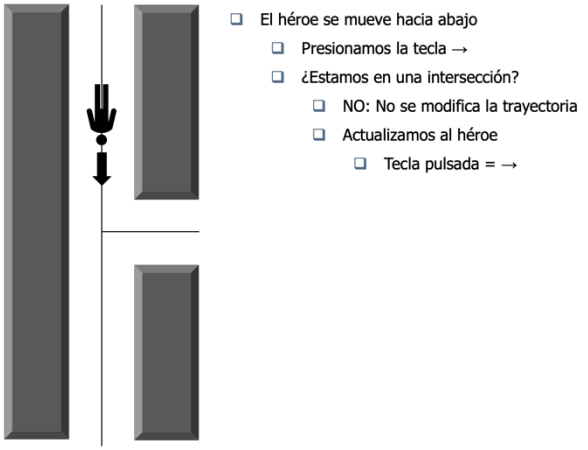
Por ello, parece razonable pensar que el *héroe* deberá contar con varios datos en su clase, su dirección actual, la dirección que indica la última tecla pulsada, y una constante para definir la velocidad a la que se mueve.

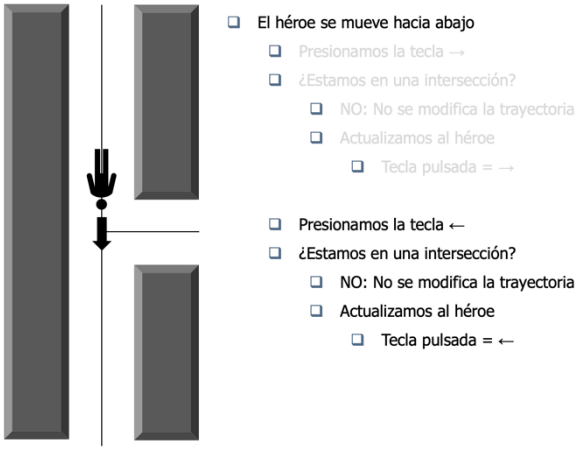
Existen varias formas de detectar colisiones para impedir, por ejemplo, que el *héroe* cambie de dirección o simplemente que se detenga cuando colisione contra un *muro*. Una posibilidad es utilizar las AABBs vistas en clase. Otra consiste en detectar cuándo el `SceneNode` del *héroe* está en el centro de una posición (que no contiene un *muro*) y entonces comprobar si éste puede moverse en dirección a la tecla pulsada. Seguidamente se muestran varios ejemplos.

En el siguiente ejemplo vemos cómo avanza el *héroe* en dirección ↓, pero cuando se presionan las teclas ← o → no puede girar porque no hay espacio.

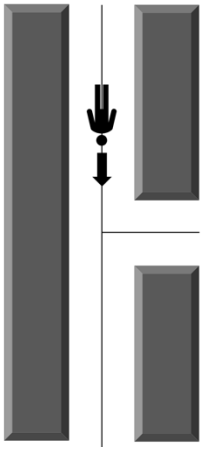




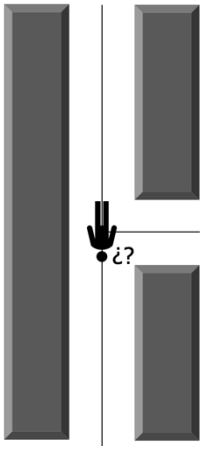




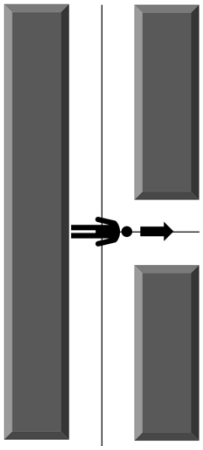
En el siguiente ejemplo vemos cómo el héroe gira hacia su izquierda (teniendo en cuenta su orientación) al presionar → antes de llegar al cruce.



- El héroe se mueve hacia abajo
  - Presionamos la tecla →
  - ¿Estamos en una intersección?
    - NO: No se modifica la trayectoria
    - Actualizamos al héroe
      - Tecla pulsada = →

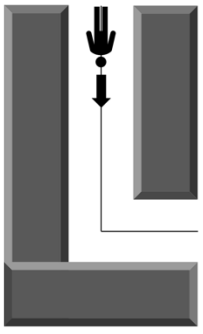


- El héroe se mueve hacia abajo
  - Presionamos la tecla →
  - ¿Estamos en una intersección?
    - NO: No se modifica la trayectoria
    - Actualizamos al héroe
      - Tecla pulsada = →
  - ¿Estamos en una intersección?
    - SÍ: Calculamos…
    - ¿Podemos movernos a →?

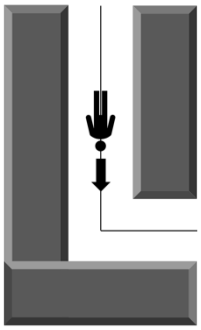


- El héroe se mueve hacia abajo
  - Presionamos la tecla →
  - ¿Estamos en una intersección?
    - NO: No se modifica la trayectoria
    - Actualizamos al héroe
      - Tecla pulsada = →
  - ¿Estamos en una intersección?
    - SÍ: Calculamos…
    - ¿Podemos movernos a →?
      - Sí: Rotamos

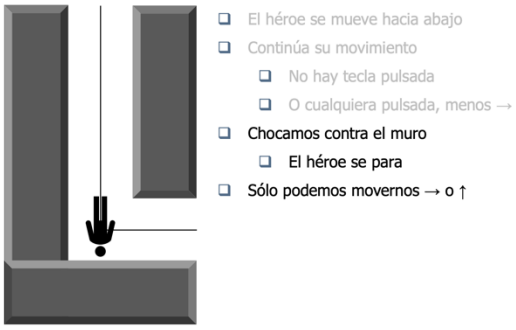
En el siguiente ejemplo vemos cómo el héroe se queda parado al chocar contra un muro. Los únicos posibles movimientos son ↑ y →.



- El héroe se mueve hacia abajo



- El héroe se mueve hacia abajo
- Continúa su movimiento
  - No hay tecla pulsada
  - O cualquiera pulsada, menos →



Para facilitar la implementación de los giros, se puede hacer uso del método `getOrientation()` de la clase `IG2Object`, el cual devuelve un `Vector3` con la orientación del personaje (héroe o villano). Así, podemos saber la nueva "posible" dirección con la tecla pulsada. Para ello, usad las constantes de `Vector3`: `Vector3::UNIT_X`, `Vector3::NEGATIVE_UNIT_X`, `Vector3::UNIT_Z`, etc.

La posición del personaje podemos obtenerla con `IG2Object::GetPosition()`. Una posible implementación puede consistir en pedirle al laberinto si el siguiente bloque en la nueva dirección es traspasable. Así, el código para implementar este apartado se simplifica considerablemente. De hecho, la idea es intentar abstraer lo máximo posible la parte del movimiento, por ejemplo, traduciendo la posición del héroe a coordenadas en bloques.

Finalmente, para calcular el ángulo de giro podemos utilizar Quaternions, como veremos en los ejemplos de clase. La clase `SceneNode` ya proporciona el siguiente método

```
void rotate(const Quaternion& q, TransformSpace relativeTo = TS_LOCAL);
```

La idea es proporcionar el Quaternion que contenga el giro que queremos, y para ello podemos usar:

```
Vector3 newDirVector = this->getNexDirVector();
Quaternion q = this->getOrientation().getRotationTo(newDirVector);
```

siendo `newDirVector` un `Vector3` con la nueva dirección después de girar, y `q` el Quaternion que contiene el giro. De esta forma, podemos pasarlo fácilmente como parámetro al método `rotate()`.

Apartado 3. Los *villanos*

Además del *héroe*, será necesario crear a los *villanos* del juego. En nuestro caso usaremos, como tipo común de *villano*, a *ogrehead*. La malla para este *villano*, como ya vimos en la práctica 0, es `ogrehead.mesh`.

Para completar el elenco de *villanos*, vamos a crear uno nuevo mediante la combinación de las mallas proporcionadas por Ogre, de forma similar a como creamos en clase el avión con su piloto.

El nuevo *villano* creado deberá cumplir las siguientes condiciones:

- Estar formado por, al menos, **tres mallas distintas**.
- Estar formado por, al menos, **diez entidades**.
- Contener, al menos, **dos partes móviles** que tengan, al menos, **tres entidades cada una** que realicen rotaciones.
- Contener un *timer* que controle el tiempo que las partes móviles realizarán **movimientos de rotación en cada sentido**.

En este punto del desarrollo, deberíamos diseñar una estructura de clases para representar tanto al *héroe* como a los *villanos*. Tened en cuenta que los movimientos serán muy similares, con la única diferencia de cómo se indican, bien mediante la pulsación de una tecla, bien mediante la decisión de un algoritmo. Por ello, puede resultar apropiado implementar en una clase la lógica común al movimiento del *héroe* y los *villanos*, aunque cada uno se especialice en su propio comportamiento. Las posiciones iniciales de los *villanos* en el laberinto se indicarán en el fichero con el carácter 'v'.

Para definir el comportamiento de los villanos, se puede implementar un movimiento aleatorio de la siguiente forma.

El villano avanzará en su dirección hasta que:

a) Encuentre un cruce con varias opciones de movimiento.
   a. En este caso, se calcula aleatoriamente la nueva dirección, salvo el giro de 180º.
b) Se bloquee contra un muro enfrente.
   a. Si es posible tomar una dirección que no sea un giro de 180º, se calcula de forma aleatoria.
   b. Si no queda otra alternativa, el villano gira 180º y continúa avanzando.

Adicionalmente, y de forma **totalmente opcional**, podéis implementar distintos comportamientos para los *villanos*. Por ejemplo, un *villano* que lo controle el usuario (modo versus), otro que tenga varios estados en los que, o bien persiga al *héroe*, o bien se aleje de él, tal y como ocurre en el juego Pac-Man original, etc.

La colisión con el *héroe* se calcula fácilmente. Cada vez que un *villano* toque al *héroe*, éste perderá una vida. Cuando esto ocurra, si el *héroe* tiene más vidas, se moverá a su posición inicial (indicada en el tablero) y continuará el juego. En caso contrario, el juego finalizará.

En la clase `IG2Object` tenemos el método que nos devuelve su AABB:

```
const AxisAlignedBox& IG2Object::getAABB();
```

La clase `AxisAlignedBox` contiene el método:

```
inline bool intersects(const AxisAlignedBox& b2);
```

el cual calcula cuándo se cruzan dos AABBs. Así, si tenemos las dos AABBs (la del *héroe* y la del *villano*) podemos saber si éstas intersecan para detectar la colisión.

Apartado 4. El suelo del laberinto

Este apartado es sencillo. En esencia, consiste en crear un suelo para la base del laberinto. Para ello, crearemos un plano que situaremos en la base del mismo, de forma que todos los bloques de *muros*, y los personajes del suelo, reposen sobre él.

Para ello, se utilizará el método `Ogre::MeshManager::getSingleton().createPlane` que veremos en clase. Posteriormente, en el apartado 6, pondremos textura al suelo para darle un aspecto más realista.

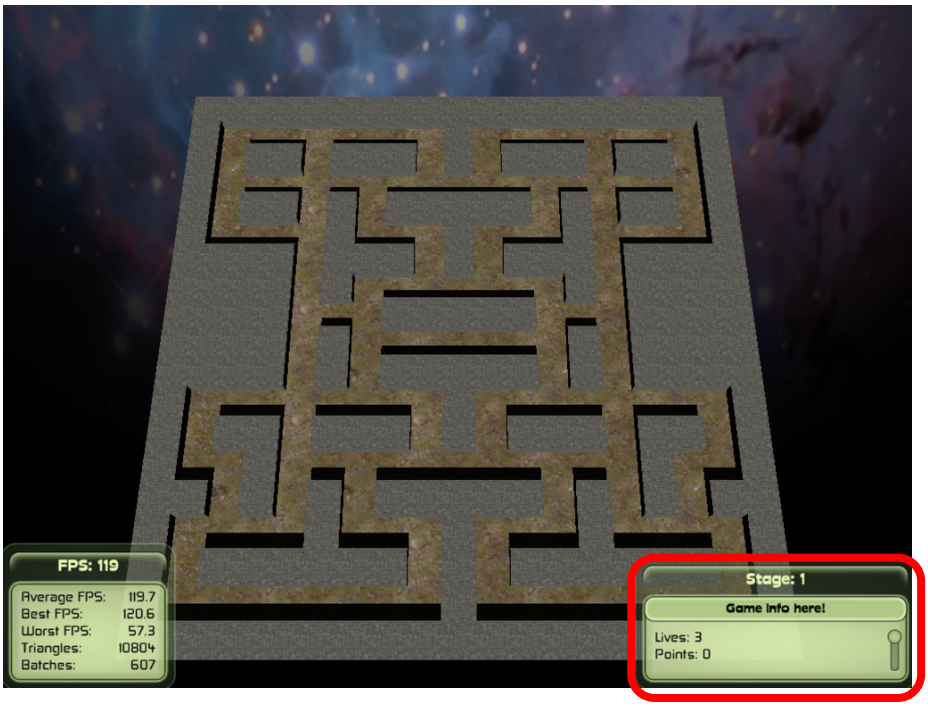
Apartado 5. Información del juego por pantalla (overlay system)

La información del juego se mostrará en un cuadro de texto situado la esquina inferior derecha. Para ello, utilizaremos un `OgreBites::TextBox` y una `OgreBites::Label`. Tened en cuenta que primero se crea la etiqueta, y luego el cuadro de texto. En el cuadro de texto, deberemos mostrar las vidas restantes del *héroe* y los puntos obtenidos (aunque por ahora no usemos los puntos). Para crearlos utilizamos los métodos:

```
mTrayMgr->createLabel(OgreBites::TL_BOTTOMRIGHT, nombre, texto, ancho);
mTrayMgr->createTextBox(OgreBites::TL_BOTTOMRIGHT, nombre, texto, ancho, alto);
```

donde `mTrayMgr` es de tipo `OgreBites::TrayManager` y se crea al inicio, antes de generar los laberintos.

La siguiente imagen muestra un ejemplo con la información del juego en la esquina inferior derecha:



Esta información deberá actualizarse en tiempo de ejecución. Por ello, hay que tener en cuenta que, aunque estos objetos se creen antes de iniciar el juego, deberán llegar a la clase encargada de gestionar la lógica del mismo.

Apartado 6. Pongamos color al juego

En este apartado vamos a asignar texturas a cada elemento del juego. Tal y como se ha explicado en clase, las texturas de los elementos se definen en un fichero de texto con extensión `.material`, donde se debe indicar, para cada uno, el color o la textura que se aplicará sobre el mismo. Este fichero estará ubicado en el directorio donde tengáis los recursos relativos a la práctica (por ejemplo, `ig2Media`) dentro de vuestro proyecto. Todas las texturas utilizadas, se copiarán en este directorio.

Es **obligatorio** que todos los elementos del juego tengan aplicada una textura o un color. Por ejemplo, en las imágenes anteriores, el suelo tiene la textura `grass.PNG` (con la extensión en mayúscula).

Con el fin de facilitar la aplicación de los materiales, y evitar la compilación del programa cada vez que modifiquemos el material de un objeto, vamos a indicar en el fichero del tablero dos parámetros adicionales, los cuales harán referencia a los materiales aplicados a los *muros*, y el *suelo*.

Así, el nuevo formato del tablero será el siguiente:

```
NumFilas
NumColumnas
materialMuro
materialSuelo
caracteresFila_1
caracteresFila_2
…
caracteresFila_NumFilas
```

Donde `materialMuro` será el material aplicado a cada *muro* del tablero, y `materialSuelo` será el material aplicado al suelo.

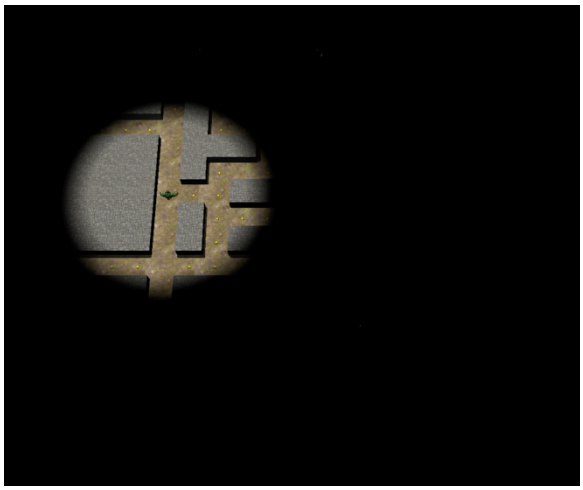
Apartado 7. Luces

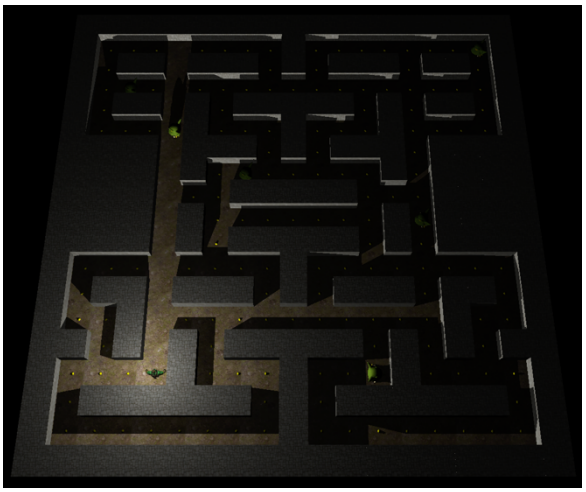
En el último apartado de la práctica utilizaremos distintos tipos de luces: *directional*, *spotlight* y *point*.

La luz *directional* iluminará todo el tablero. Podéis darle cierta inclinación para que se aprecien las sombras.

Para las luces de tipo *spotlight*, colocaremos la luz sobre el tablero en dirección `-Y`, de forma que la única parte iluminada sea el círculo generado por el foco. La idea es que el foco comparta las coordenadas `x` y `z` del *héroe*, y se mueva con él. Así, solo iluminaremos una parte del tablero, la cual se moverá con el *héroe* según se vaya desplazando. La imagen de la izquierda muestra un ejemplo con esta luz.

Para la luz de tipo *point*, seguiremos la misma línea. Colocaremos la luz sobre la cabeza del héroe, de forma que ésta se mueva con él. Un ejemplo con esta luz se muestra en la imagen derecha.





El tipo de luz utilizada en el juego se definirá también en el fichero, después de los materiales, tal y como se ve en el siguiente cuadro

```
NumFilas
NumColumnas
materialMuro
materialSuelo
tipoLuz
caracteresFila_1
caracteresFila_2
…
caracteresFila_NumFilas
```

donde `tipoLuz` define la luz utilizada y puede tomar tres valores: `directional` para la luz direccional, `spot` para los focos *spotLight* y `point` para luces que iluminan en un punto.

Fecha de entrega.

La práctica deberá entregarse, a través del C.V., antes del día **29 de octubre de 2025 a las 11:00 horas**. Únicamente será necesario realizar una entrega por grupo. La defensa de la práctica se realizará **en la clase de laboratorio del día 29 de octubre**. Para realizar la defensa, **los dos miembros del grupo** (si se ha realizado en pareja) deben asistir presencialmente a clase. Además, se utilizará el orden de entrega para realizar la defensa, de forma que el primer grupo en entregar la práctica será el primero en realizar la defensa.

Para entregar la práctica, es necesario enviar – **únicamente** – el código fuente y el material adicional relativo a scripts, materiales, texturas, etc. Para ello, generad un fichero .zip con el código fuente – eliminando previamente los directorios x64 – y otro con el material adicional.