Informática Gráfica II

Práctica 1

Curso 24/25

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. Las presentaciones relativas a la práctica estarán disponibles en el Campus Virtual para su consulta.

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 Pac-Man, donde controlaremos a un personaje (*héroe)* cuyo objetivo es comerse todas las *perlas* que hay en un laberinto sin que los enemigos (*villanos)* lo toquen. 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 con constantes estáticas, como por ejemplo las teclas del juego, los caracteres utilizados para codificar laberintos, los puntos al comer una *perla*, 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.
- Es posible – y debéis – utilizar el esqueleto del proyecto visto en clase para desarrollar la práctica.
- Considerad el método `seputpScene()` de la clase `IG2App` como el punto de partida para el juego. Podéis cambiarle el nombre si lo consideráis necesario.

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`** con una entidad **`sphere.mesh`**, la cual representará una *perla*.

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* y *perlas*, pero en el siguiente apartado, también leeremos del fichero la posición inicial del *héroe*.

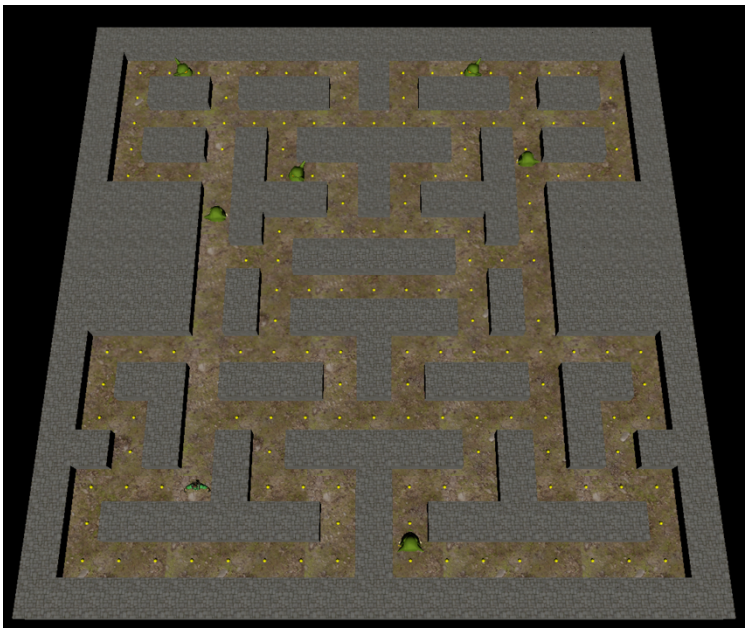
El formato del fichero de texto se describe a continuación:

```
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 el valor 'x' y 'o' que definen, respectivamente, un *muro* y una *perla*.

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

```
19
19
xxxxxxxxxxxxxxxxxxx
xooooooooxoooooooox
xoxxoxxxoxoxxxoxxox
xooooooooooooooooox
xoxxoxoxxxxxoxoxxox
xooooxoooxoooxoooox
xxxxoxxxoxoxxxoxxxx
xxxxoxooooooooxxxxx
xxxxooxxxxxxxooxxxx
xxxxoxooooooooxxxxx
xxxxoxoxxxxxoxoxxxx
xooooooooxoooooooox
xoxxoxxxoxoxxxoxxox
xoooxooooooooooxoox
xxoxoxoxxxxxoxoxoxx
xooooxoooxoooxoooox
xoxxxxxxoxoxxxxxxox
xooooooooxoooooooox
xxxxxxxxxxxxxxxxxxx
```



En las posiciones donde se ubique una *perla* no habrá *muro*, y es por donde podrá moverse el *héroe*. La idea es que, en apartados posteriores, cada *perla* que toque el *héroe* desaparezca.

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 *perla* podrán ser clases que heredarán de `IG2Object`, aunque cada una tiene una particularidad. Por ejemplo, un *muro* no puede atravesarse, pero la posición donde se ubica una *perla* 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.

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 la dirección indicada. Si puede trasladarse, lo hará en una unidad en la dirección correspondiente.

Por ello, parece razonable pensar que el *héroe* deberá contar con dos datos en su clase: su dirección actual, y la dirección que indica la última tecla pulsada.

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.

Cuando el *héroe* colisione con una *perla*, ésta desaparecerá y nuestro *héroe* sumará 10 puntos.

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 los mismos, con la única diferencia de cómo se indica el mismo, bien mediante la pulsación de una tecla, bien mediante la decisión de un algoritmo. Por ello, puede resultar apropiado implementar en una clase toda 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 calcular el movimiento de los *villanos* utilizaremos el siguiente esquema:

1. Un *villano* nunca cambia de sentido (girar 180º) a menos que sea su única opción para no quedarse bloqueado.
2. Un *villano* sólo calcula una nueva dirección en los siguientes casos:
    a. Está bloqueado.
    b. Está en una posición donde es posible realizar un giro de 90º y avanzar en esa dirección. Así, la dirección tomada será la que **minimice** la distancia euclídea entre el *héroe* y el centro del primer bloque visitado por el *villano*.
3. En cada paso que dé el *villano* (si no está bloqueado) avanzará una unidad en la posición correspondiente.

Adicionalmente, y de forma **totalmente opcional**, podéis implementar distintos comportamientos para los *villanos*. Por ejemplo, un *villano* que se mueva de forma aleatoria, un *villano* que lo controle el usuario, y 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.

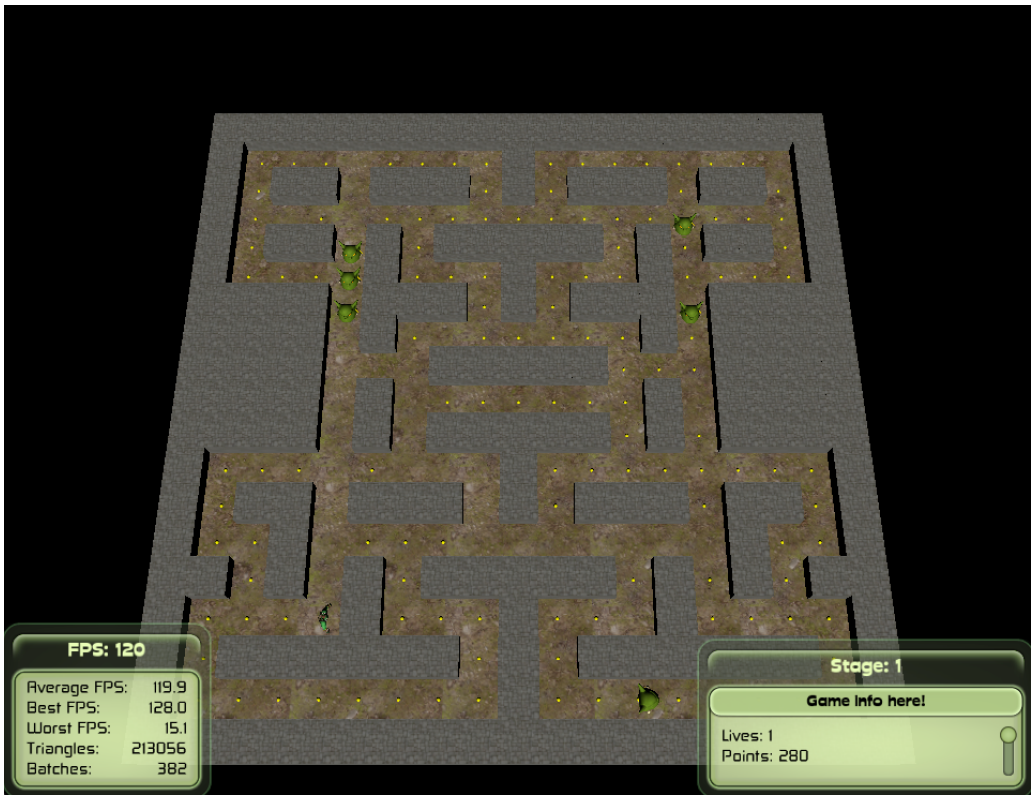
Apartado 4. El suelo del laberinto

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`. Entre otros datos, en el cuadro de texto, deberemos indicar las vidas restantes del *héroe* y los puntos obtenidos. La siguiente imagen muestra un ejemplo:



Esta información deberá actualizarse en tiempo de ejecución. De esta forma, 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 elemento del juego, el color o la textura que se aplicará sobre el mismo. Este fichero estará ubicado en `Media/IG2App`. Además, 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 tres parámetros adicionales, los cuales harán referencia a los materiales aplicados a las *perlas*, los *muros*, y el *suelo*.

Así, el nuevo formato del tablero será el siguiente:

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

donde `materialPerla` será el nombre del material aplicado a las *perlas*, `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 una luz de tipo spotLight. La idea es colocar la luz sobre el tablero, en dirección `-Y` de forma que, sobre el tablero, 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, sólo iluminaremos una parte del tablero, la cual se moverá con el *héroe* según se vaya desplazando.

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
materialPerla
materialMuro
materialSuelo
tipoLuz
caracteresFila_1
caracteresFila_2
…
caracteresFila_NumFilas
```

donde `tipoLuz` define la luz utilizada y puede tomar dos valores: `directional` para la luz direccional, o spot para los focos `spotLight`.

Fecha de entrega.

La práctica deberá entregarse, a través del C.V., antes del día **6 de noviembre de 2024 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 6 de noviembre**. 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.