

Informática Gráfica II

Práctica (Parte II)

Curso 25/26

Desarrollo de un videojuego utilizando Ogre3D

En esta práctica vamos a mejorar el videojuego realizado en la práctica anterior con los efectos vistos en clase, tales como animaciones, sistema de partículas, *multitexturas*, *skyplanes* y *shaders*.

Apartado 1. Animación – Creando una *intro* para el juego.

En este apartado vamos a crear una *animación inicial* para el juego. Además, animaremos a Sinbad cuando se mueva por el laberinto. Para la primera parte la idea es que, al iniciar el programa, se muestre la animación creada – en bucle – hasta que pulsemos una tecla, por ejemplo, la ‘s’, tras lo cual dará comienzo el juego. Esta animación contará con los siguientes elementos (aunque en apartados posteriores introduciremos algún cambio):

- Un suelo (distinto del suelo empleado en el laberinto).
- Sinbad (con y sin espadas)
- Ogrehead

Sinbad hará movimientos laterales, realizando rotaciones cuando llegue a los extremos, y un baile cuando esté en el centro. Tened en cuenta que, hasta el primer giro, Sinbad no tiene espadas, aparecen al girar la primera vez. Ogrehead hará un movimiento similar, y en su último giro se aplicará una escala de forma que cada vez sea más pequeño, hasta desaparecer cuando llega a la posición de Sinbad. Seguidamente se muestran algunas capturas de esta animación. Tenéis disponible un video en el C.V. para ver con más detalle cada una de las acciones.





El diseño de los *keyframes* y los valores para la transformación es libre, ya que el objetivo es que la animación se parezca a la mostrada en el vídeo. Sin embargo, algunos datos que se pueden utilizar para definir la animación son los siguientes:

- El tamaño del plano es de 150x300
- La animación dura 21 segundos, luego se repite en bucle
 - Finaliza cuando empieza el juego, al pulsar una tecla (por ejemplo 's')
- La animación de Sinbad puede realizarse con 9 keyframes (estado inicial incluido)
- La animación de ogrehead puede realizarse con 6 keyframes (estado inicial incluido)

Tened en cuenta que, para crear la animación, puede resultar útil crear un método que añada *keyframes* a un **NodeAnimationTrack**, como por ejemplo el método:

```
addKeyFrame (nodeAnimationTrack, giro, posición, más parámetros...);
```

de forma que en el personaje tendremos algo así:

```
// Keyframe 0
this->addKeyframe (track, Quaternion::IDENTITY, Vector3::ZERO, . . .);

// Keyframe 1
this->addKeyframe (track, Quaternion::IDENTITY, Vector3::ZERO, . . .);
```

Una vez definido el método para añadir *keyframes*, debemos controlar cuándo Sinbad tiene que correr o bailar. Podéis consultar el nombre para estas animaciones en el Tema 6. Para esto, podemos utilizar los instantes de tiempo de los *keyframes*, por ejemplo, mediante atributos en la clase, de forma que cada uno indica el instante en el que acaba el *keyframe*.

Para llevar la cuenta de este tiempo, tenemos dos opciones. La primera consiste en utilizar un *Timer*. Para ello basta con incluir en el código la directiva

```
#include <OgreTimer.h>
```

Y crearlo como se muestra a continuación:

```
Ogre::Timer* timer = new Ogre::Timer();
```

Seguidamente, podremos reiniciarlo:

```
timer->reset();
```

o calcular el tiempo transcurrido (en ms):

```
timer->getMilliseconds();
```

La segunda opción consiste en utilizar un contador e ir sumando el tiempo transcurrido en **frameRendered**, ya que sabemos que el tiempo transcurrido desde que se renderizó el último *frame* y lo obtenemos de **evt.timeSinceLastFrame**.

Una vez realizada la animación inicial, resultará sencillo incluir la animación del héroe durante el juego. Para ello, bastará con actualizar el tiempo transcurrido en el método **frameRendered**.

Apartado 2. Incluyendo las bombas en el juego.

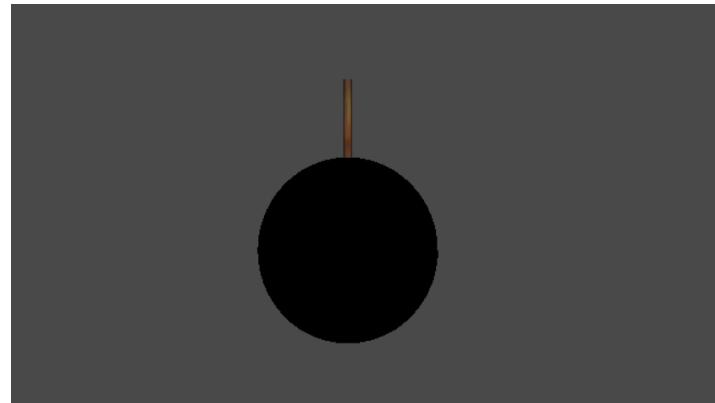
En el juego, el *héroe* puede colocar bombas en el laberinto. Las bombas, pasado un tiempo determinado – una vez colocadas – explotan, haciendo que su explosión afecte a los bloques adyacentes donde no hay muro. Es decir, la explosión se expande en las direcciones norte, sur, este y oeste, a partir del bloque donde está situada la bomba, de forma que este bloque no se tiene en cuenta para calcular la longitud de la explosión. Si en una determinada dirección, la explosión encuentra un muro sólido, deja expandirse en esa dirección. Este comportamiento debe ser configurable, por lo cual se recomienda definir constantes para controlar los siguientes aspectos:

- El número máximo de bombas que se pueden colocar en el tablero de forma simultánea.
- La longitud (en número de bloques) del alcance de las bombas.
- Tiempo que tarda la bomba en explotar, una vez colocada.

La explosión de la bomba sólo tendrá efecto en el momento en el cual explote, es decir, cuando haya vencido el tiempo establecido en la constante correspondiente. Esto puede controlarse teniendo en cuenta el tiempo que ha transcurrido desde el render del último *frame*, tal y como hicimos para calcular el movimiento de los personajes. En ese momento, todo personaje – héroe o villano – situado en los bloques donde afecte la explosión, serán eliminados: se decrementará una vida del *héroe* y/o se eliminará del tablero al/los villano/s alcanzado/s.

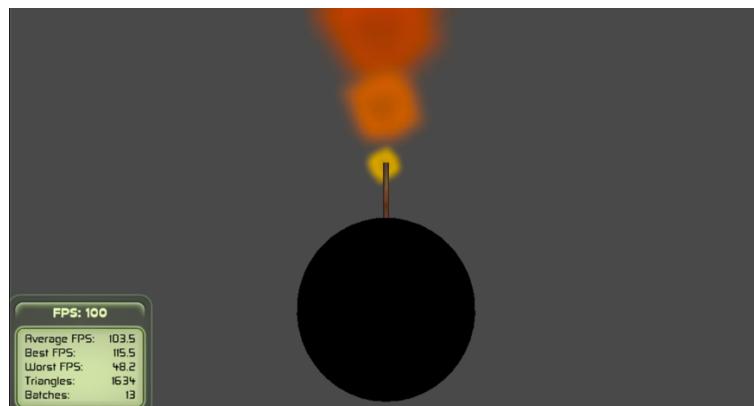
2.1 Diseño y animación de la bomba

La bomba estará formada por dos entidades, una esfera (para representar el cuerpo de la bomba) y un cilindro (para representar la mecha). La siguiente imagen muestra una posible representación de la misma.



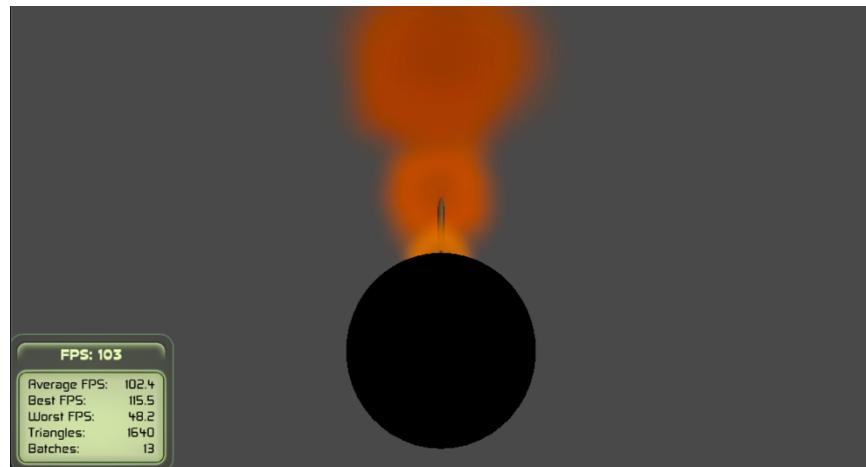
Adicionalmente, incluiremos un efecto de escalado para animar este objeto. Para ello, vamos a escalar la bomba en función el tiempo transcurrido, fijando un tamaño mínimo, un tamaño máximo y el tiempo entre el cual la animación transcurre entre los mismos. Tened en cuenta que cuando se modifica la escala del objeto (o la bomba completa, incluyendo la mecha) se aplica sobre el tamaño original del objeto, no sobre el tamaño que refleja su estado actual.

Además, crearemos un sistema de partículas que simule el efecto de prender la mecha. Para ello, situaremos el sistema de partículas en el extremo superior de la misma, y éste se irá desplazando hasta el centro de la bomba según vaya avanzando el tiempo. Como ya se comentó en el apartado anterior, el tiempo desde que se deposita la bomba, hasta que ésta explota, es configurable. De forma similar, controlaremos los tamaños que alcanza la bomba, y lo rápido que pasa de uno a otro. La siguiente imagen muestra el inicio de la animación, donde el sistema de partículas se sitúa en la parte superior.



Para este sistema de partículas utilizaremos un emisor de tipo **Point** y varios modificadores (*affector*) de tipo **ColourFader**, **ColourImage**, **Rotator**, **Scaler** y **DirectionRandomiser**. Tened en cuenta que para el modificador **ColourImage** se utilizará la imagen `smokeColors.png`. Además, como las bombas podrán activarse varias veces durante el juego, será recomendable que el sistema de partículas asociado tenga una duración indefinida para poder reiniciarlo cuando sea necesario.

En la siguiente imagen, vemos que el sistema de partículas ha avanzado hasta llegar al cuerpo de la bomba.



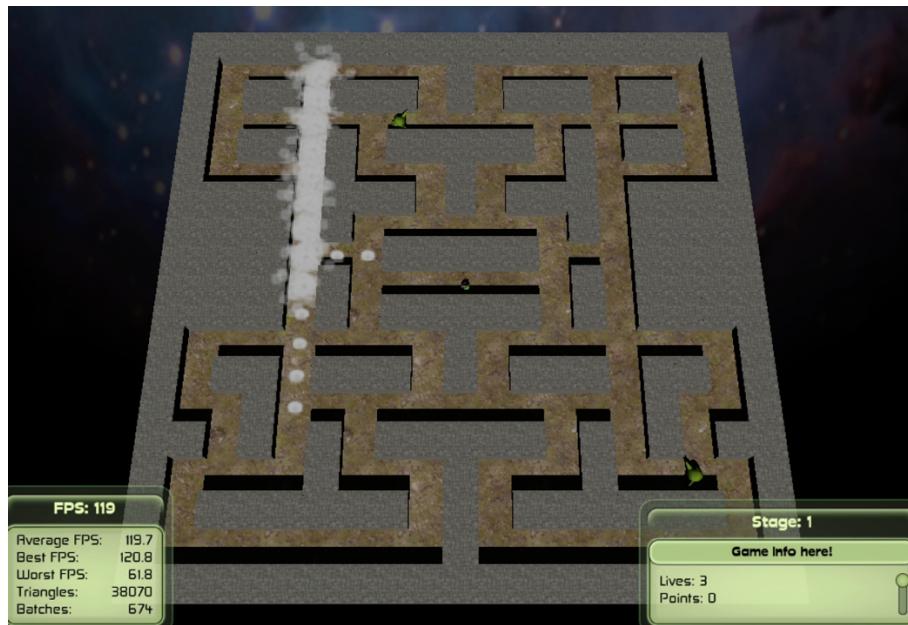
En el C.V. tenéis disponible un vídeo donde podréis apreciar con mayor detalle el efecto descrito.

2.2 Explosión de la bomba

Para reflejar en el juego la explosión de la bomba, utilizaremos sistemas de partículas que simulen humo. Tened en cuenta que sólo se colocarán los sistemas de partículas en los bloques donde no haya muro sólido, es decir, por donde pueden pasar el héroe y los villanos. La siguiente imagen muestra un ejemplo en el que el héroe coloca dos bombas, y éstas están configuradas para expandir la explosión en 5 bloques.



La siguiente imagen muestra el efecto al explotar. Observad que la bomba en la parte superior explota antes, por eso los sistemas de partículas han liberado un mayor número de partículas, formando el efecto "humo" de forma más consistente, mientras la bomba colocada en la parte inferior empieza su explosión poco tiempo antes de haber realizado la captura:



En el C.V. tenéis disponible un vídeo donde podéis observar el comportamiento descrito en este apartado.

NOTA: NO es necesario implementar el *timer* cuando muere el héroe.

Cada sistema de partículas debe tener un nodo (**SceneNode**) asociado. De esa forma, nos permitirá colocarlo fácilmente en el laberinto. Cada sistema de partículas podremos iniciar o pararlo con **setEmitting (bool enable)**. De forma similar, con el método **clear()**, eliminaremos inmediatamente todas las partículas visibles del emisor.

Para el sistema de partículas de humo del laberinto se utilizará:

- Un emisor de tipo **Point**
- Una duración de 0 sg para poder reiniciarlo cuando sea necesario.
- Una **Quota** de 300 (con valores altos, y muchos sistemas, baja mucho los fps)
 - Ángulo con valor alto (simula mayor dispersión)
- El color (atributo **colour** del emisor)
 - Recuerda que blanco es 1, negro es 0
 - Translúcido es 0.5, transparente es 0, opaco es 1
 - La idea es que se simule el efecto humo y se dificulte la visibilidad

Apartado 3. Efectos con texturas y añadiendo un cielo

En este apartado vamos a aplicar efectos sobre las texturas del juego. Al menos, es necesario aplicarlo sobre un elemento con textura como, por ejemplo, el suelo de la intro. También podéis aplicar otros de los efectos que hemos visto en clase sobre los bloques, o el suelo del laberinto.

El efecto aplicado deberá utilizar dos ficheros de imagen. En la siguiente captura se muestra un efecto utilizando la textura del suelo y `lightMap.jpg` para dar un efecto de sombra.



Seguidamente, crearemos un cielo como se ve en la siguiente imagen. Tened en cuenta que el cielo NO se ve durante la intro, debe aparecer sólo durante el juego. Dependiendo de cómo sea la orientación de vuestro laberinto, tendréis que crear el plano – con la normal correspondiente – y probar con los distintos parámetros. Además, el cielo debe moverse. Utilizad el parámetro `lighting off` para que no le afecte la luz.



Apartado 4. Shaders

En este apartado vamos a definir dos shaders.

4.1 Efecto de onda (wave) para el suelo

Para este shader vamos a simular el efecto del movimiento del mar. Este efecto lo crearemos definiendo un nuevo material – llamado **practica/wavesShader** – que aplicaremos al suelo de la presentación, tal y como se ve en la siguiente imagen.



En este caso, el shader de fragmentos (**wavesShaderFS.gls1**) es sencillo, basta con obtener el color a partir de las coordenadas de textura pasadas por el shader de vértices y la unidad de textura – que utilizará el fichero **Water02.jpg** – pasada a través de la aplicación. No se le aplica ninguna modificación al color obtenido.

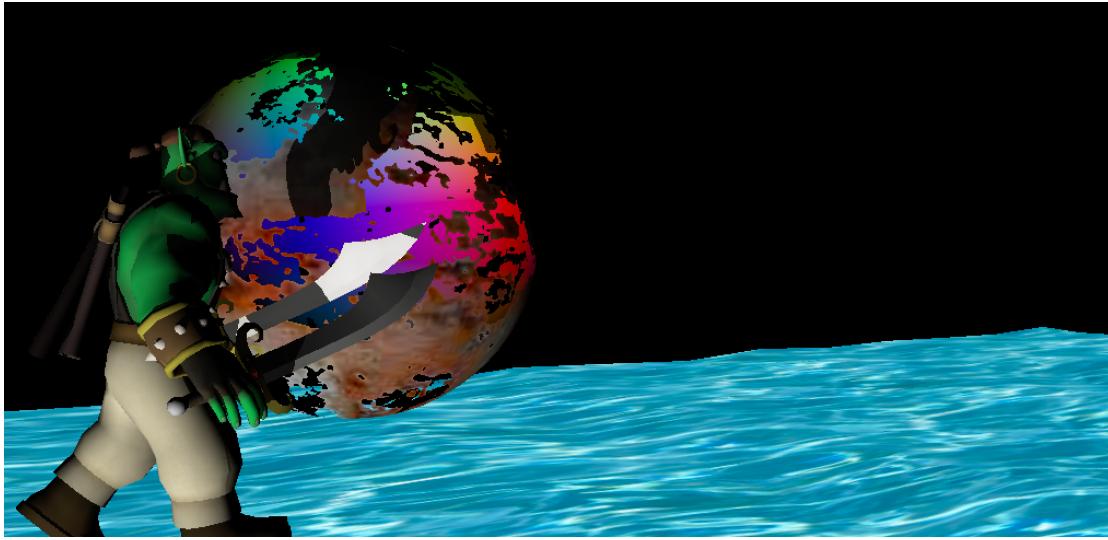
Al shader de vértices (**wavesShaderVS.gls1**) le pasamos desde la aplicación: la matriz de proyección (**worldviewproj_matrix**), un valor que varíe en el tiempo en el intervalo [0, 1], y el centro del plano donde se va a aplicar el shader. En esencia, el efecto se consigue variando la componente **y** del vértice. Existen múltiples soluciones para conseguir el efecto deseado. Una aproximación es la siguiente:

```
vertexCoord.y += sin(vertexCoord.x + (time*a)) * b + sin(vertexCoord.z + (distanceCenter) + (time*a)) * b;
```

donde **vertexCoord** es una variable auxiliar con las coordenadas de **vertex**, **time** es la variable de tipo **uniform** que le pasamos desde la aplicación y se modifica de forma automática, y **distanceCenter** es la distancia desde el vértice actual al centro de la malla donde aplicaremos el efecto. Los valores de **a** y **b** podéis ajustarlos para conseguir un resultado similar al mostrado en los vídeos que están disponibles en el C.V.

4.2 Shader para hacer hueca una esfera con un interior *colorido*

Para este shader vamos a crear un efecto aplicado a una esfera similar al efecto que provoca el óxido en los metales, de forma que las partes **con más corrosión (colores con un alto componente rojo)**, nos permitan ver el interior de la misma. Para ello, colocaremos una esfera en el centro del suelo de la intro, tal y como se aprecia en la siguiente imagen. La luz direccional utilizada en la captura tiene dirección (1, 1, -1).



Tenéis varios vídeos disponibles en el C.V. para que podáis analizar la posición y escala – aproximada – de la misma, así como el interior, al utilizar direcciones diferentes para la luz.

Para crear esta esfera podéis utilizar la malla “`uv_sphere.mesh`”, la cual utiliza coordenadas de textura. El material en cuestión – al que llamaremos `practica/sphere` – definirá una unidad de textura para la cara exterior, para lo cual utilizaremos el fichero de imagen `corrosion.jpg`. La idea es dar un efecto de corrosión, de forma que en la parte exterior de la esfera se aplique la textura, mientras que a la parte interior mostremos los colores como se aprecian en la imagen. Tened en cuenta que el color rojo se aprecia en las partes más exteriores del eje `x`, el color azul en las partes más exteriores del eje `z`, y el verde en las partes más exteriores del eje `y`.

Para implementar el shader pedido hay que tener en cuenta varios aspectos. Inicialmente, el `culling`, ya que nos vamos a encargar en los shaders de calcular qué color final tendrá cada cara (`front` y `back`). Recordad que para esto es necesario indicar `cull.hardware none` y `cull.software none` en el material. Además, debemos saber qué cara del fragmento estamos procesando. Para ello, pasaremos desde la aplicación el siguiente parámetro: `param_named_auto flipping render_target_flipping`.

En lo referente a la iluminación, la desactivaremos en el pase del material. En el shader, aplicaremos la iluminación utilizando la técnica de Lambert. Es importante remarcar que los cálculos se llevarán a cabo **en el shader de fragmentos**, NO en el de vértices. En este último, es posible que necesitemos obtener información de la escena, pero no relativos a la luz. En la escena se empleará una única luz direccional. El shader de vértices y fragmentos se codificarán en los ficheros `sphereVS.gls1` y `sphereFS.gls1`, respectivamente.

Fecha de entrega.

La práctica deberá entregarse, a través del C.V., antes del día **3 de diciembre 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 3 de diciembre**. 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.