1. Introducción a Bullet 3D

Bullet 3D (http://bulletphysics.org) es un motor 3D utilizado en videojuegos de última generación (Madagascar Kart para X-Box 360) y efectos especiales de películas recientes (2012). Nos ofrece un conjunto de funciones que podemos utilizar para programar prácticamente cualquier juego en 3D que se nos ocurra.

En este tutorial aprenderemos a manejar las funciones básicas de manera que estemos preparados para programar un videojuego sencillo, y si lo deseamos, continuar por nuestra cuenta aprendiendo a utilizar otras funcionalidades que, por su utilidad limitada o similaridad con otras, no hemos visto necesario introducir en este tutorial.

1.1. Videojuegos de ejemplo

Todos los temas del tutorial llevan asociado un sencillo videojuego de ejemplo en el que se ilustran todos los conceptos que se han explicado. En el caso de este primer tema tenemos un videojuego de ejemplo muy simple que únicamente nos muestra un escenario con un cubo en 3D que podemos ver desde una cámara estática, se trata de una escena sin interacción.

Lo que haremos en este tema será ver la organización del videojuego de ejemplo y explicar con detalle las instrucciones básicas de Bullet 3D que allí se usan. El lenguaje que utilizaremos para trabajar con el motor Bullet 3D será BennuGD (http://bennugd.org), un lenguaje 100% orientado a la programación de videojuegos con el que nos resultará sencillísimo aprender a utilizar Bullet 3D.

Para poder seguir correctamente este tutorial y poder visualizar y editar algunos archivos es necesario que tengas instalado Bennupack 1.8 (http://bennupack.blogspot.com) o superior.

1.2. Compilar y ejecutar

Para poner en marcha el videojuego de ejemplo o las modificaciones que hagamos a partir de él en primer lugar es necesario verificar que las instrucciones que hemos escrito son correctas. Las instrucciones se escriben en los ficheros de extensión .prg, y podemos encontrar el fichero principal llamado videojuego.prg en el directorio del videojuego de ejemplo y otros ficheros secundarios en el subdirectorio /prg.

Es importante que recuerdes guardar el fichero cada vez que introduzcas una modificación, ya que si no lo haces se verificará la versión anterior no guardada y esto puede darte algún quebradero de cabeza al principio.

El fichero RunWin.bat es el encargado de realizar la verificación del código (Compilación) y de poner en marcha (Ejecución) el videojuego si no ha habido errores. Una compilación correcta nos mostrará un resumen de datos utilizados por nuestro videojuego, mientras que una compilación incorrecta nos mostrará un resumen del primer error localizado, indicando el fichero .prg en el que se encuentra y el número de línea de código implicada.

Recuerda que en caso de error de compilación se ejecutará la última versión correcta del videojuego, así que es importante que prestes atención al resultado de al compilación para comprobar si hay algún error que debas solucionar.

La compilación correcta genera un archivo de extensión .dcb con el mismo nombre que el fichero de código principal, en nuestro caso se llamará videojuego.dcb. El fichero .dcb no podemos abrirlo con ningún programa, pero es el que se ejecutará gracias a todos los ficheros .dll y .exe localizados dentro del subdirectorio /bgd_win. Podemos decir que /bgd_win es la maquinaria de Bullet 3D y BennuGD, de momento no tocaremos nada en ese subdirectorio.

Otro fichero interesante, aunque por ahora no será necesario modificarlo, es el fichero bgdc.import, en el cual debemos enumerar los nombres de los ficheros .dll que hay en el directorio bgd_win y que utilizaremos en nuestro videojuego, pues habrá funcionalidades (Ratón, audio, etc.) que requerirán utilizar ciertas .dll.

Finalmente, dentro del subdirectorio /doc tenemos un .pdf que nos servirá como manual de referencia para consultar las funciones de Bullet 3D, es recomendable tener siempre a mano ese manual, ya que es como un diccionario del lenguaje. También observa el subdirectorio /map. Como veremos a continuación allí tenemos un fichero comprimido .zip en el que se guarda el mapa de Quake 3 que se carga en el videojuego de ejemplo.

Y eso es todo lo que encontramos en nuestro videojuego de ejemplo, pasemos a ver cómo está organizado el código (Ficheros .prg) ya que es lo que debemos editar para determinar el resultado de la ejecución.

1.3 Constantes

Observarás que las primeras instrucciones (Líneas de código) del fichero videojuego.prg son sentencias INCLUDE. Estas sentencias incluyen otros ficheros secundarios de código que ofrecen alguna utilidad especial y merece la pena mantener separados, así evitaremos que los ficheros de código se hagan demasiado largos.

El primer INCLUDE es especial y lo veremos más adelante, ya que es imprescindible para poder utilizar el motor 3D. En cambio el siguiente INCLUDE hace referencia al fichero de código const.prg que se encuentra dentro del subdirectorio /prg y es interesante que veamos a fondo su utilidad.

El módulo const.prg crea un bloque de constantes (CONST). Las constantes no son más que palabaras definidas por el programador que representan realmente un valor, generalmente numérico. Si observas, tenemos constantes para representar la resolución horizontal y vertical (800x600 en la plantilla), la profundidad de color (32 bits en la plantilla) e incluso el nombre del juego.

Para entender mejor la utilidad de las constantes: Imagina que durante el proceso de creación del videojuego te interesa tener un elemento centrado en la pantalla. Cuando la resolución es de 800x600 es suficiente con que el elemento se posicione en la posición 400,300 de la pantalla, ¿Pero qué ocurre si modificamos la resolución?, también deberíamos modificar la posición de ese elemento…

Para evitarlo se usan las constantes, y el elemento centrado se posicionará en _RESOLUCION_X / 2, _RESOLUCION_Y / 2. Lo hará en función de esas constantes, así si las modificamos por cualquier motivo el elemento de pantalla se ajustará automáticamente evitándonos muchos quebraderos de cabeza.

Por supuesto, en const.prg simplemente damos valores, las constantes declaradas se utilizan realmente en el módulo video.prg, que veremos a continuación.

1.4 Video

En el módulo video.prg tenemos declarado un proceso que realiza 4 instrucciones de configuración del modo de vídeo. Esas 4 instrucciones podrían perfectamente usarse en videojuego.prg directamente sin necesidad de usar un módulo secundario, pero durante el tutorial veremos que a largo plazo conviene tener un módulo específico para gestionar el modo de vídeo, ya que más adelante podremos querer cambiarlo durante la ejecución o nos interesará liberar recursos de vídeo al finalizar nuestro videojuego.

Las 3 primeras instrucciones hacen uso de las constantes definidas en const.prg y realizan, respectivamente, las tareas de creación de la ventana de ejecución con una cierta resolución, profundidad de color y modo de visualización.

En set_mode ( ), el cuarto parámetro MODE_WINDOW puede cambairse por MODE_FULLSCREEN para tener nuestro videojuego a pantalla completa. Si quieres conocer más modos aplicables puedes consultar más a fondo la documentación de la instrucción set_mode ( ) en el manual de Bennu.

En set_fps ( ) establecemos el número de frames por segundo a los que se ejecutará el videojuego. Por defecto la constante que los determina es 60, ya que a esa tasa de frames se consigue una fluidez perfecta. Una cantidad mayor implicaría un consumo de CPU excesivo, y una cantidad menor provoca que se note el cambio entre fotogramas. El segundo parámetro de set_fps ( ) indica el número de frames que el juego evitará dibujar en caso de que la CPU no satisfaga la tasa de frames requerida.

En set_title ( ) simplemente mostramos el nombre de nuestro videojuego en la marquesina de la ventana de ejecución.

Y finalmente tenemos la instrucción M8E_INIT ( ), la primera instrucción de Bullet 3D. Todas las funciones específicas de Bullet 3D comienzan con el prefijo "M8E_" y se escriben en mayúsculas para diferenciarlas de otras funciones que son propias de BennuGD y que por tanto también se utilizan para el diseño de videojuegos en 2D.

El parámetro de M8E_INIT ( ) es una constante que indica el controlador de video que se utilizará para la ejecución. Por defecto la plantilla usa EDT_DIRECT3D9 que indica Direct3D 9 porque es compatible con prácticamente cualquier equipo, pero si quieres lograr mayor calidad puedes utilizar EDT_OPENGL. También pueden usarse otros drivers de video de menor calidad. Puedes consultar la lista de constantes para los diferentes drivers en la lista de funciones de Bullet 3D.

1.5 Globales

En este módulo declaramos una serie de identificadores (Variables) de uso global. Las variables globales tienen sentido cuando almacenan valores que requerirán ser consultados/modificados por distintos procesos del juego. En nuestro caso hemos declarado una variable global para almacenar el identificador del mapa y otra para el identificador de la cámara.

Observa que la declaración de ambas variables viene prefijada con la palabra "int". Esto indica que se trata de identificadores enteros (integer), que almacenarán números positivos o negativos, incluido el 0, sin decimales.

El hecho de que sean enteros tiene que ver con la naturaleza de Bullet 3D, que por sencillez hace que la práctica totalidad de los identificadores devueltos, ya sea por modelos, texturas, luces, etc. Sean simples enteros, por lo que la mayoría de los identificadores que declaremos serán enteros.

Por supuesto podemos dar el nombre que queramos conveniente a una variable, en nuestro caso hemos utilizado los nombres "mapa" y "cámara" por ser los nombres más ilustrativos para almacenar los identificadores del mapa y de la cámara, respectivamente.

1.6 Videojuego

Como habrás comprobado, los módulos secundarios de los que hace uso el videojuego de ejemplo son muy sencillos, y sin duda nos resultarán útiles a largo plazo, ya que pronto el juego comenzará a crecer en número de módulos y líneas de código, nos conviene tenerlo todo correctamente organizado y separadas en módulos las distintas ideas que programemos. Esta técnica además nos facilitará la reutilización de algunos módulos en futuros proyectos.

Ahora observa el resto del fichero principal de código videojuego.prg. A continuación de los INCLUDES viene un bloque BEGIN END dentro del cual se realizan una serie de instrucciones. Las instrucciones en BennuGD, a diferencia de las declaraciones de constantes o variables, siempre están dentro de un bloque BEGIN END. Este bloque se llama un proceso, hemos visto un sencillo proceso dentro del módulo video.prg, pero aquí tenemos un caso de proceso especial, ya que es el proceso principal del juego, el que se ejecuta al principio cuando arrancamos nuestro juego.

La primera instrucción es precisamente una invocación al proceso inicializar_video ( ) que tenemos declarado en video.prg y que se encarga de configurar correctamente el modo de video.

A continuación vienen 2 instrucciones interesantes de Bullet 3D:

M8E_ADDZIPFILE ( ) es una instrucción que recibe como parámetro la ruta de un fichero, este tipo de dato se denomina string y siempre viene entrecomillado. M8E_ADDZIPFILE ( ) descomprime el fichero dado como parámetro (Tiene que estar en formato .zip), y en nuestro caso se trata de un mapa de Quake III que está comprimido porque de otra forma ocuparía mucho espacio en disco.

Una vez aplicada la descompresión del mapa utilizamos M8E_LOADMODELEX ( ), que también recibe como parámetro la ruta de un fichero de modelo 3D, en este caso se trata del fichero que hemos descomprimido que es un mapa de Quake III en formato .bsp.

M8E_LOADMODELEX ( ) puede recibir como parámetro cualquier fichero de modelo 3D compatible con Bullet 3D (.3ds, .obj, .md3, etc.). Se utiliza especialmente para modelos grandes ya que está optimizada para cargarlos de manera eficiente. Su segundo parámetro es un entero que puede tomar valores entre 0 y 255 indicando la calidad con la que se cargará el modelo, a mayor valor, mayor calidad. Puedes consultar la lista de modelos 3D soportados por Bullet 3D en la lista de funciones.

Fíjate que M8E_LOADMODELEX ( ) nos genera un identificador de modelo cargado que guardamos en la variable global mapa. Como verás en este tutorial, todas las funciones que cargan un modelo nos generan un identificador entero que podemos guardar en una variable.

La instrucción M8E_LOADCUBEMODEL ( ) es otra función que carga un modelo 3D, en este caso carga un simple cubo, y en el parámetro ponemos introducir un número con la longitud de su arista. Es interesante saber que ese número no tiene por qué ser un entero, se trata de un número real (float) que puede tener decimales. Si modificamos el valor de ese número cambiaremos el tamaño del cubo, observarás que para valores excesivamente altos el cubo se hará tan grande que nos encontraremos en su interior, y no lograremos verlo. Si buscas información sobre esta instrucción en la lista de funciones de Bullet 3D observarás que nos genera un entero con el identificador del cubo cargado, aunque en la plantilla no lo estamos almacenando podríamos tener una variable global donde guardarlo (p.ej. cubo = M8E_LOADCUBEMODEL (10);).

A continuación ponemos una cámara en la escena que nos servirá para visualizar los elementos que hemos cargado (Un mapa y un cubo hasta ahora). Bullet 3D tiene varias funciones para cargar cámaras, aunque para esta primera toma de contacto optamos por la más simple de todas, una cámara estática M8E_ADDCAM ( ) que nos genera un identificador que guardamos en la variable camara.

La función M8E_ADDCAM ( ) requiere 6 parámetros para su invocación, si es la primera vez que trabajas en 3D te resultará curioso, observa que para posicionar una cámara en el espacio necesitamos 3 valores para sus coordenadas (x,y,z) y otros 3 valores más para indicar el punto al que mira la cámara (x,y,z). Pues en ese mismo orden damos esos 6 parámetros a nuestra cámara. Puedes intentar hacer que la cámara mire hacia arriba, para ello bastaría con hacer que la coordenada y del objetivo (El 5º parámetro) tenga un valor más alto que la coordenada y de la posición de la cámara (El 2º parámetro).

Observa que tal y como aparece la cámara en el videojuego de ejemplo está mirando al punto (0,0,0) y es exactamente en ese punto donde se ha cargado el cubo. Esto se debe a que la carga de modelos en Bullet 3D los posiciona automáticamente en el punto (0,0,0) a no ser que apliquemos una posición distinta, como veremos en el próximo tema.

Con las instrucciones anteriores ya tenemos creada nuestra escena, pues hemos cargado un escenario, un modelo con forma de cubo y hemos puesto una cámara para verlo. Pero verás que a continuación en videojuego.prg tenemos un bloque LOOP END, veamos cuál es su función.

1.7 El bucle de control

Un bloque LOOP END en BennuGD es un bloque de instrucciones que se ejecutarán repetidamente durante toda la ejecución, ésto en programación se denomina formalmente bucle.

BennuGD se ejecuta en frames, esto quiere decir que varias veces por segundo (Según lo que hayamos indicado en set_fps ( )) se dibuja en pantalla el resultado de la ejecución. Para poder indicar qué es lo que queremos que se ejecute en cada frame los distintos procesos incorporarán casi siempre un bucle dentro del cual se ejecutará una sentencia FRAME;

Aunque todavía es pronto para entenderlo totalmente, anticipamos que FRAME; es una sentencia que, además de indicar el momento en el que se dibujará el resultado de la ejecución de un proceso, también sirve para que se sincronicen todos los procesos activos, de forma que todos ellos ejecutan FRAME; al mismo tiempo garantizando que funcionen todos al mismo ritmo.

Ahora mismo estamos estudiando el proceso principal del videojuego, y aunque es el único proceso activo también es necesario que incorpore una sentencia FRAME; dentro de un bucle, en otro caso, como comprobarás más adelante, el juego se quedaría bloqueado.

Dentro del bloque LOOP END también tenemos una instrucción de Bullet 3D. M8E_RENDER ( ) es una instrucción que realiza el dibujado 3D de la escena completa. No debemos confundirla con FRAME; ya que FRAME; es una sentencia que debe ejecutar todo proceso mientras que M8E_RENDER ( ) es una instrucción que realiza el bucle principal y que sólo debe ejecutarse una vez por FRAME;

M8E_RENDER ( ) recibe 4 parámetros enteros, cuyos valores pueden valer entre 0 y 255 e indican las componentes de color alpha, red, green y blue (argb) que se mostrarán en el fondo de la escena. Para que lo entiendas más claramente puedes poner la cámara mirando al cielo y jugar con los valores, observarás que es el color del cielo lo que realmente cambia. Cabe destacar que al tratarse del fondo de la escena el parámetro alpha no introduce ningún cambio, ya que el alpha es la transparencia del color y no tenemos nada detrás del cielo para poder apreciarla.

Sólo nos falta ver él último bloque que tenemos dentro del bucle de control, se trata de un bloque IF END, que explicaremos con detenimiento.

1.8 Condicionales

Un bloque IF END en BennuGD se denomina condicional. Se trata de un conjunto de instrucciones que se ejecutarán si y sólo si se cumple una determinada condición.

En el caso de nuestro videojuego de ejemplo tenemos una condición muy sencilla pero necesaria, la condición de finalización del juego.

Observa que a continuación de IF y entre paréntesis tenemos la instrucción key ( ). Es una instrucción de BennuGD que genera el Verdadero (TRUE) o False (FALSE) según se haya pulsado o no la tecla que le damos como parámetro. En este caso tenemos key (_ESC) y por tanto la condición se cumple cuando pulsamos la tecla Esc.

Dentro del bloque IF END se especifican las instrucciones a ejecutar en caso de que se cumpla la condición. En este caso tenemos una única instrucción exit ( ). La instrucción exit ( ) de BennuGD finaliza la ejecución y cierra el videojuego, por lo que queda claro que la utilidad de este condicional es cerrar el juego cuando se pulsa la tecla Escape.

Como curiosidad, habrás observado que exit ( ) recibe como parámetro un número. Se trata del código de finalización del programa y en él podemos especificar distintos valores para indicar si la salida del programa se ha debido a una finalización normal del mismo (En esos casos conviene usar el valor 0) o bien ha sido resultado de una situación de error, en este caso podemos indicar con distintos números si hemos finalizado el juego porque el driver de vídeo no es compatible, porque no se ha podido cargar un determinado modelo 3D, etc.

1.9 Ejercicios propuestos

De forma optativa y para que te resulte más fácil comprender lo que se ha explicado en este tema, se enumeran una serie de ejercicios interesantes con los que puedes practicar, sólo te tomarán unos minutos:

a) Eliminar el escenario

Puedes omitir las instrucciones M8E_ADDZIPFILE ( ) y M8E_LOADMODELEX ( ) añadiendo "//" delante de ellas. Con eso las conviertes en comentarios y no serán ejecutadas. La escena debería dibujarse de manera idéntica, sólo que no habrá escenario y verás solamente el cubo sobre un fondo del color indicado en M8E_RENDER ( ).

b) Ver el subterráneo

Si utilizas valores negativos para el posicionamiento de la cámara en el eje y podrás ver lo que hay bajo el suelo del escenario. Si te alejas lo suficiente podrás ver el escenario desde fuera y observarás que puedes ver a través de las paredes. Esto es debido a que los polígonos en 3D sólo tienen una superficie visible, y si la miras desde el otro lado ver a través de ella.

c) Cambiar la tecla de finalización del juego

Actualmente el condicional que fija la condición de salida del juego utiliza Escape. Puedes utilizar cualquier tecla del teclado, todas ellas son constantes que comienzan con el caracer "_", puedes consultar las constantes asociadas a cada tecla si buscas información sobre la instrucción key ( ) en el manual de BennuGD.

d) Cambiar el cubo por una esfera

Si cambias la instrucción M8E_LOADCUBEMODEL ( ) por M8E_LOADSPHEREMODEL ( ) en lugar de cargar un cubo cargarás una esfera, el parámetro indicará su radio. También puedes utilizar una instrucción seguida de la otra, con distintos parámetros para conseguir tener las esquinas del cubo asomando a través de la esfera.

e) Cargar un modelo de Quake III

Aunque lo veremos más adelante, puedes cambiar M8E_LOADCUBEMODEL ( ) por M8E_LOADANIMODEL ( ). Esta última instrucción no requiere un parámetro numérico indicando el tamaño, si no que requiere una ruta de fichero indicando la ubicación de un modelo 3D soportado por Bullet 3D de la misma forma que M8E_LOADMODELEX ( ), sólo que se utiliza para modelos pequeños. Puedes probar con un fichero de formato .md2, que como veremos, son modelos animados de Quake.