TUTORIALES BÁSICOS

Alumno: Velázquez Muñoz, Mario

Profesor: Juan Peralta Donate

Universidad Carlos III de Madrid

TABLA DE CONTENIDO

TABLA DE CÓDIGO

TABLA DE ILUSTRACIONES

INTRODUCCIÓN

Este documento es una traducción de los tutoriales básicos disponibles en la Wiki de la página oficial de Ogre (www.ogre3d.org). En la web se puede encontrar el código completo de cada uno de los tutoriales, mientras que en este documento, sólo se incluye el código completo de algunos de los tutoriales más básicos.

Debido a que Ogre es una herramienta en continuo cambio, es posible que los tutoriales no funcionen siempre al 100% según se han escrito en este manual. Por ello se recomienda adaptar las partes iniciales de cada tutorial (el código de inicio que se aporta en cada tutorial), para que compilen y funcionen con la versión actual de Ogre y los plugins adicionales que se requieran antes de continuar.

En cuanto a los tutoriales sobre CEGUI, a partir de la versión 1.7.0 de Ogre, este complemento dejó de estar adjunto al proyecto de Ogre, por lo que si se quieren seguir estos tutoriales se deberían instalar las librerías de CEGUI. Para instalarlas sólo hay que buscar información sobre la librería en la Wiki de Ogre y realizar una serie de pasos similares a los que se usan en el documento **Entorno de Trabajo OGRE 3D** para el caso de la librería de físicos.

En la wiki de Ogre, también se pueden encontrar tutoriales de nivel medio y nivel avanzado, así como gran cantidad de artículos sobre proyectos Ogre que pueden ayudar en la realización de aplicaciones.

Esta recopilación de tutoriales contiene los tutoriales básicos de Ogre para la versión 1.6.5, por lo que en versiones más avanzadas de Ogre es posible que deban sufrir algunas pequeñas modificaciones para su correcto funcionamiento.

TUTORIAL BÁSICO 1: SCENENODE, ENTITY Y SCENEMANAGER

1. PRERREQUISITOS

Se asume que se tienen conocimientos del lenguaje C++ y se sabe como configurar y compilar una aplicación en Ogre. (Si se tienen problemas para configurar una aplicación para Ogre, hay una pequeña guía en el documento OGRE, entorno de trabajo). No es necesario ningún conocimiento específico de Ogre para realizar este tutorial.

2. INTRODUCCIÓN

En este tutorial se introducirán las estructuras más elementales de Ogre: El SceneManager, el SceneNode y la Entity. No se cubrirá una gran cantidad de código, sino que se centrará en los conceptos generales para que se pueda empezar a aprender a programar en Ogre. A medida que se avance por el tutorial, se deberá ir añadiendo código lentamente al proyecto y observando los resultados que se obtienen.

3. PRIMEROS PASOS

3.1. CÓDIGO INICIAL

Se va a empezar usando un código pre-construido para este tutorial. Se debe hacer caso omiso de la mayoría del código, salvo por lo que se añada al método createScene. En próximos tutoriales se profundizará en cómo las aplicaciones de Ogre trabajan, pero por ahora se comienza con el nivel más básico. Hay que crear un proyecto en el compilador que se elija y añadir un archivo fuente que contenga este código:

```
#include "ExampleApplication.h"

class TutorialApplication : public ExampleApplication
{
protected:
public:
    TutorialApplication()
    {
    }

    ~TutorialApplication()
    {
    }
protected:
    void createScene(void)
    {
    }
};

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT
)
#else
int main(int argc, char **argv)
```

```
#endif
{
    // Create application object
    TutorialApplication app;

    try {
        app.go();
    } catch( Exception& e ) {
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
        MessageBox( NULL, e.what(), "An exception has occurred!",
MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
        fprintf(stderr, "An exception has occurred: %s\n",
e.what());
#endif
    }

    return 0;
}
```

Código 1: Código de inicio práctica 1.

Si se está usando el OgreSDK bajo Windows, hay que asegurarse de añadir (incluir) el directorio "[Directorio_OgreSDK]\sample\include" en este proyecto (el archivoExampleApplication.h se encuentra allí). Si se utiliza la distribución de código fuente de Ogre, dicho directorio debe estar situado en el directorio"[Directorio_OgreSource]\samples\common\include". Hay que asegurarse de poder compilar y ejecutar el código antes de continuar a la siguiente sección, aunque no se verá nada más que una pantalla negra con un cuadro de imágenes hasta que no se añadan cosas más adelante en el tutorial. (*Véase Figura 1)*

Ilustración 1: Primera panatalla de la práctica 1.

Una vez que el programa está funcionando, se pueden utilizar las teclas W,A,S,D para moverse, el ratón para mirar alrededor, y la tecla ESC para salir del programa.

3.2. SOLUCIÓN DE PROBLEMAS

Si se tienen problemas, hay que consultar el documento **"Entorno de trabajo OGRE 3D"** para configurar correctamente el compilador, o se puede buscar en el archivo Ogre.log para obtener información más detallada. Si se necesita más ayuda, se puede buscar por los foros (http://www.ogre3d.org/forums/). Es probable que el problema sea muy común y en ellos se encuentre una solución. Si no es así, se puede crear un tema nuevo, asegurándose de proporcionar los detalles pertinentes del archivo Ogre.log, mensajes de error, etc.

4. CÓMO TRABAJA OGRE

Éste es un tema amplio. Se empezará con el *SceneManager* y la forma de trabajar con*Entitys* y *SceneNodes*. Estas tres clases son los bloques fundamentales de lasaplicaciones de Ogre.

4.1. CONCEPTOS BÁSICOS DEL *SCENEMANAGER*

Todo lo que aparece en la pantalla es administrado por *el SceneManager*. Cuando se coloquen objetos en la escena, la clase *SceneManager* es la que sigue la pista de sus ubicaciones. Al crear cámaras para ver la escena el *SceneManager* realiza un seguimiento de ellas. Al crear planos, carteles, luces, etc ..., el *SceneManager* realiza un seguimiento de ellos. Existen múltiples tipos de *SceneManager*. Hay *SceneManager* para el terreno, hay *SceneManager* para mapas BPS, y así sucesivamente. Se profundizará más sobre los tipos de *SceneManager* a lo largo de los tutoriales.

4.2. CONCEPTOS BÁSICOS DE *ENTITY*

Una *Entity* es un tipo de objeto que se puede añadir a la escena. Se puede pensar en una *Entity* como cualquier cosa representada por una malla 3D. Un robot sería una *Entity*, un pez sería una *Entity*, el terreno por el cual los personajes anden sería una *Entity* muy grande. Cosas como luces, carteles, partículas, cámaras, etc. No serían *Entitys*. Algo a tener en cuenta sobre Ogre es que separa los objetos de su posición y su orientación. Es decir, no se puede colocar una *Entity* directamente en la escena. En vez de eso se debe adjuntar la *Entity* a un *SceneNode*, el cual contiene la información sobre la localización y la orientación de la *Entity*.

4.3. CONCEPTOS BÁSICOS DE *SCENENODE*

Como ya se ha mencionado, el *SceneNode* realiza un seguimiento de la ubicación y la orientación de todos los objetos que se le atribuyen. Cuando se crea una *Entity*, no se introduce en la escena hasta que se le asocia a un *SceneNode*. Además, el *SceneNode* no es un objeto que se muestre en la pantalla. Solo cuando se crea un *SceneNode* y se le asocia una *Entity* se muestra algo realmente en la pantalla. Los *SceneNode* pueden tener asociado cualquier número de objetos. Digamos que si se tiene un personaje que camina a través de la pantalla y se desea generar una luz a su alrededor. Lo que se debería hacer es crear primero un *SceneNode* y a continuación crear una *Entity* para el personaje y asociarla al *SceneNode*. Entonces se debería crear el objeto luz y asociarlo también al *SceneNode*. Los *SceneNode* también pueden contener otros *SceneNode* de manera que se le permite crear toda una jerarquía de nodos. Se cubrirán aspectos más avanzados del *SceneNode* en posteriores tutoriales. Un concepto importante a destacar es que la posición de un *SceneNode* siempre es relativa al *SceneNode* padre, y cada *SceneManager* contiene un nodo raíz que tiene asociados todos los *SceneNode*.

5. PRIMERA APLICACIÓN CON OGRE

Hay que buscar la función TutorialApplication::createScene. En este tutorial solo se manipulará el contenido de esta función. Lo primero que se hace es establecer la luz ambiente de la escena para que se pueda ver lo que se está haciendo. Esto se hace llamando a la función setAmbientLight y especificando el color deseado. Hay que tener en cuenta que la estructura ColourValue espera valores de rojo, verde y azul en el rango de 0 y 1. Hay que añadir esta sentencia a createScene:

```
mSceneMgr->setAmbientLight( ColourValue( 1, 1, 1 ) ); mSceneMgr->
setAmbientLight(ColourValue (1, 1, 1));
```

Código 2: Añadiendo luz ambiente a la escena.

Lo siguiente que se debe hacer es crear una *Entity*. Se hace esto llamando a la función createEntity del *SceneManager*.

```
Entity *ent1 = mSceneMgr->createEntity( "Robot", "robot.mesh" );
```

Código 3: Creación de una entidad.

Llegados a este punto surgen varias preguntas. En primer lugar, ¿de dónde proviene mSceneMgr, y cuáles son los parámetros que se usan para llamar a la función? La variable mSceneMgr contiene el actual objeto *SceneManager* (esto viene hecho por defecto por la clase ExampleApplication). El primer parámetro para crear una *Entity* es el nombre de la *Entity* que se está creando. **Todas las *Entitys* deben tener un nombre único.** Se obtendrá un error si se intentan crear dos *Entitys* con el mismo nombre. El parámetro "robot.mesh" especifica la malla que se quiere usar para la entidad. De nuevo, la malla que se está usando ha sido pre-creada por la clase ExampleApplication. Se ha creado una *Entity*, ahora hay que crear un *SceneNode* para asociarla a él. Como cada *SceneManager* tiene un *SceneNode* raíz, primero se crea un hijo de este nodo.

```
SceneNode *node1 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode" );
```

Código 4: Creación de un SceneNode.

Esta sentencia primero llama a getRootSceneNode del *SceneManager*. Entonces llama al método createChildSceneNode para crear el hijo del nodo raíz del *SceneManager*. El parámetro de createChildSceneNode es el nombre del *SceneNode* que se está creando. Como pasa con las *Entity*, dos *SceneNode* no pueden tener el mismo nombre. Finalmente, se necesita asociar la *Entity* al *SceneNode* para darle al robot una ubicación.

```
node1->attachObject( ent1 );
```

Código 5: Asociación de una Entity y un SceneNode.

¡Y eso es todo! Ahora se compila y ejecútala aplicación. Debería verse un robot de pie en la pantalla. (*Véase Ilustración 2)*

Ilustración 2: Primera práctica con un robot creado.

6. COORDENADAS Y VECTORES

Antes de ir más lejos, hay que hablar sobre las coordenadas de la pantalla y los vectores de objetos Ogre. Ogre (como otros motores gráficos) usa los ejes X y Z como el eje horizontal, y el eje Y como el eje vertical. Si se está mirando el monitor, el eje X va desde la parte izquierda hasta la parte derecha de la pantalla, siendo la dirección derecha el eje positivo. El eje Y va desde la parte superior hasta la inferior de la pantalla, siendo la dirección positiva hacia arriba. El eje Z va desde el interior de la pantalla hasta el exterior, siendo la dirección positiva hacia fuera. Se observa que el robot está mirando hacia el eje X.

Esta es una propiedad de la misma malla y de la forma en que fue diseñada. Ogre no hace suposiciones sobre cómo orientar sus modelos. Cada malla que se cargue puede tener diferente orientación. Ogre usa la clase Vector para representar tanto posición como dirección (No existe la clase Punto). Hay vectores definidos para 2 dimensiones (Vector2), 3 dimensiones (Vector3) y 4 dimensiones (Vector4), siendo Vector3 el más utilizado. Si no se está familiarizado con los conceptos de vectores es recomendable adquirir conocimientos básicos antes de hacer nada serio con Ogre. Las matemáticas relacionadas con vectores serán útiles a la hora de empezar a trabajar en proyectos más complejos.

7. AÑADIENDO OTROS OBJETOS

Ahora que se conoce el sistema de coordenadas, se puede volver al código. En las tres líneas que se han escrito, no se especifica en ningún sitio la localización exacta que se desea para el robot. La gran mayoría de funciones de Ogre tiene parámetros por defecto. Por ejemplo, la función SceneNode::createChildSceneNode de Ogre tiene 3 parámetros: El nombre del *SceneNode*, la posición del *SceneNode* y la orientación del *SceneNode*. La posición, como se ha podido comprobar, está definida, por defecto, en las coordenadas (0, 0, 0). Se va a crear otro *SceneNode*, pero esta vez especificando la localización inicial, y esta será diferente al origen de coordenadas.

```
Entity *ent2 = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode2", Vector3( 50, 0, 0 ) );
node2->attachObject( ent2 );
```

Código 6: Creación de un objeto en una posición determinada.

Esto resulta familiar. Se ha hecho exactamente lo mismo que antes, con dos excepciones. En primer lugar, se ha dado un nombre un poco diferente al *SceneNode* y a la *Entity*. La segunda cosa que se ha hecho es indicar que la posición de inicio sea 50 unidades en la dirección x respecto al nodo raíz (hay que recordar que la posición de los *SceneNode* es relativa al *SceneNode* padre). Se compila y ejecuta la aplicación. Ahora en la escena hay dos robots uno enfrente del otro. (*Véase Ilustración 3)*

Ilustración 3: Pantalla con dos robots.

8. *ENTITYS* MÁS A FONDO

La clase *Entity* es muy extensa, y no se va a cubrir el uso de cada porción del objeto aquí... sólo lo justo para empezar. Hay algunas funciones muy útiles para la *Entity* a destacar.

La primera es Entity::setVisible y Entity::isVisible. Puede configurar una *Entity* para ser visible o no simplemente llamando a esta función. Si se quiere esconder una *Entity*, pero más tarde mostrarla, entonces se usa esta función en vez de destruir la *Entity* y después volverla a crear.

La segunda es la función getName, la cual devuelve el nombre de la *Entity*.

La última función a destacar es getParentSceneNode, que devuelve el *SceneNode* al que se asocia la *Entity*.

9. *SCENENODE* MÁS A FONDO

La clase *SceneNode* es muy compleja. Hay muchísimas cosas que pueden hacerse con el*SceneNode*. Aquí tan solo se muestran las más útiles.

Se puede obtener y fijar la posición de un *SceneNode* con las funciones getPosition y setPosition (Posición relativa a la del *SceneNode* padre). También se puede mover el objeto en relación a su posición actual mediante el método translate.

El *SceneNode* no solo establece la posición, sino que también la escala y la rotación del objeto. Se puede fijar la escala de un objeto con la función scale. Se pueden utilizar las funciones pitch, yaw, y roll para rotar los objetos. Se puede usar la función resetOrientation para resetear todas las rotaciones hechas en el objeto. También se pueden usar las funciones setOrientation, getOrientation y rotate para opciones más avanzadas de rotación.

Ya se ha visto la función attachObject. Estas funciones también son útiles si se quieren manipular los objetos que están asociados al *SceneNode*: numAttachedObjects, getAttachedObject, detachObject, detachAllObject. Hay también una serie de funciones para tratar con *SceneNode* padre e hijo.

Como el posicionamiento y la translación se hace en relación al *SceneNode* padre, se pueden mover fácilmente 2 *SceneNode* a la vez. Actualmente se tiene este código en la aplicación:

```
Entity *ent1 = mSceneMgr->createEntity( "Robot", "robot.mesh" );
SceneNode *node1 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode" );
node1->attachObject( ent1 );
Entity *ent2 = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode2", Vector3( 50, 0, 0 ) );
node2->attachObject( ent2 );
```

Código 7: Código con dos objetos del tutorial 1.

Si se cambia la quinta línea de esta sentencia:

```
SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode2", Vector3( 50, 0, 0 ) );
```

Código 8: Línea a cambiar.

Por esta:

```
SceneNode *node2 = node1->createChildSceneNode( "RobotNode2", Vector3(
50, 0, 0 ) );
```

Código 9: Línea de cambio, se asocia un nodo hijo al anterior objeto.

Entonces se hace que RobotNode2 sea un hijo de RobotNode. Moviendo el nodo1se moverá también el nodo2, pero moviendo el nodo2 no se moverá el nodo1. Por ejemplo, este código solo moverá el nodo2:

```
node2->translate( Vector3( 10, 0, 10 ) );
```

Código 10: Mueve sólo el nodo hijo.

El siguiente código moverá el nodo1 y, como nodo2 es hijo de nodo1, también se moverá:

```
node1->translate( Vector3( 25, 0, 0 ) );
```

Código 11: Mueve el nodo padre y su hijo asociado.

Si se están teniendo problemas con esto, lo mejor es empezar en el nodo raíz e ir bajando.

Se supone (como es el caso) que se sitúa el nodo1 en (0, 0, 0) y se desplaza (25,0, 0), por tanto al final tenemos al nodo1 en la posición (25, 0, 0) en relación con su padre (el nodo raíz). El nodo2 empezó en la posición (50, 0, 0) y si lo desplazamos (10,0, 10) su nueva posición será (60, 0, 10) respecto a su padre (el nodo 1).

Se comprueba si esto realmente ocurre. Comenzando en el nodo raíz. Su posición es siempre (0, 0, 0). La posición de nodo1 será (raíz + nodo1) : (0, 0, 0) + (25, 0, 0) = (25, 0, 0). No sorprende. Ahora el nodo2. Este es un hijo de nodo1, por lo que su posición será (raíz + nodo1 + nodo2) : (0, 0, 0) + (25, 0, 0) + (60, 0, 10) = (85, 0, 10). Este ejemplo muestra la forma de herencia de nodo.

Por último, hay que tener en cuenta que se pueden obtener tanto *SceneNode* como *Entitys* por su nombre llamando a getSceneNode y getEntity de la *SceneManager*, así que no hay que mantener un puntero a cada *SceneNode* que cree.

10. COSAS A PROBAR

Por ahora se debe tener una comprensión básica sobre los conceptos *Entity, SceneManager*y *SceneNode*. Por lo que se sugiere que se comiencen a añadir y quitar robots a la escena. Una vez hecho, borrar todo el contenido de createScene, y practicar con cada uno de los siguientes códigos:

10.1. ESCALA

Se puede escalar la malla llamando al método scale en el *SceneNode*. Hay que cambiar los valores en scale y observar el resultado.

```
Entity *ent1 = mSceneMgr->createEntity( "Robot", "robot.mesh" );
SceneNode *node1 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode" );
node1->attachObject( ent1 );
node1->scale( .5, 1, 2 );
Entity *ent2 = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode2", Vector3( 50, 0, 0 ) );
node2->attachObject( ent2 );
node2->scale( 1, 2, 1 );
```

Código 12: Escala los objetos.

10.2. ROTACIONES

Se puede rotar el objeto usando los métodos yaw, pitch, y roll usando tanto grados como radianes. El método pitch es para rotar alrededor del eje x, el método yaw alrededor del eje y el método roll alrededor del eje z. (*Véase Ilustración 4)* Usando la mano derecha como guía, se pone el dedo pulgar señalando al eje Y, los demás dedos indicaran hacia donde está el sentido positivo de los ángulos.

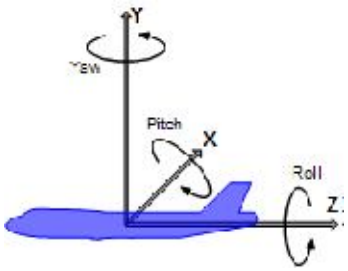

Ilustración 4: Ejemplo de rotaciones sobre los distintos ejes.

Se puede probar cambiando el valor de Degree y combinando varias transformaciones:

```
Entity *ent1 = mSceneMgr->createEntity( "Robot", "robot.mesh" );
SceneNode *node1 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode", Vector3( -100, 0, 0 ) );
node1->attachObject( ent1 );
node1->yaw( Degree( -90 ) );
Entity *ent2 = mSceneMgr->createEntity( "Robot2", "robot.mesh" );
SceneNode *node2 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode2");
node2->attachObject( ent2 );
node2->pitch( Degree( -90 ) );
Entity *ent3 = mSceneMgr->createEntity( "Robot3", "robot.mesh" );
SceneNode *node3 = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("RobotNode3", Vector3( 100, 0, 0 ) );
node3->attachObject( ent3 );
node3->roll( Degree( -90 ) );
```

Código 13: Ejemplo de combinación de varias rotaciones.

11. LO QUE ENVUELVE A OGRE

La mayoría de los archivos (.DLL y .CFG) nombrados en esta sección (y en los demás tutoriales) se encuentran en la carpeta bin del directorio de OgreSDK, tanto en la carpeta debug como en la release.

Hay que tener en cuenta que gran parte de la información en esta sección está enfocada para Windows. Bajo Linux se aplica la misma información, aunque las librerías acaban en.so y residen en otras localizaciones, y algunas cosas pueden ser un poco diferentes. Sise están teniendo problemas, se puede colgar la duda en los foros de ayuda de Ogre (http://www.ogre3d.org/forums/).

11.1. DLLS Y PLUGINS

Ahora que se ha visto el ambiente de Ogre, se va a explicar cómo trabajan las librerías de Ogre para hacer la vida más fácil cuando se trabaje con ellas.

Ogre se divide en 3 grandes grupos de librerías compartidas: la librería principal, los plugins y las librerías de terceros y librerías de ayuda.

Librería principal.

El primer grupo consta de la propia librería. La librería de Ogre se encuentra, en su totalidad en OgreMain.dll. Este DLL requiere algunas otras librerías como cg.dll. Estos archivos DLL se deben incluir con cada aplicación Ogre sin excepción.

Plugins.

El segundo grupo de librerías compartidas son los plugins. Ogre lleva una buena parte de la funcionalidad en las librerías compartidas de manera que puede ser activada o desactivada en

función de las necesidades de la aplicación. Los plugins básicos incluidos con Ogre tienen un nombre de archivo que empieza con el prefijo "Plugin_".

Se pueden crear nuevos plugins propios si la aplicación los necesita, pero no se cubrirá este tema en ningún tutorial. Ogre también usa plugins para el sistema de render (como OpenGL, DirectX, etc.). Estos plugins empiezan con el prefijo "RenderSystem_". Se pueden añadir o remover sistemas de render para la aplicación.

Esto es bastante útil si se está programando algo específico para, por ejemplo, OpenGL y se necesita desactivar la posibilidad de ejecutar el programa en DirectX. Además, si la aplicación no está orientada a una plataforma estándar, se puede crear un plugin propio de render, aunque esto no será cubierto en el tutorial. Se aprenderá cómo eliminar los plugins en la siguiente sección.

Librerías de terceros y librerías de ayuda.

Ogre en sí es solo una librería de renderizado de gráficos. Es por ello que también existen muchas librerías para ampliar su funcionalidad. Las demos de Ogre SDK incluyen unas cuantas de estas librerías. La librería CEGUI es un sistema GUI que es muy fácil de integrar a Ogre. Las DLLs que empiezan con "CEGUI" y"OgreGUIRenderer.dll" son parte de ella. Se aprenderá a usar CEGUI en tutoriales posteriores. La comunicación con el teclado y ratón se realiza con OIS. Está contenido en OIS.dll. También hay otras librerías (que no se incluyen con SDK) que dan aún más funcionalidad (como sonido) y de las cuales puede encontrar más información en los foros (http://www.ogre3d.org/forums/).

La moraleja de esta historia es que cuando se está probando una aplicación a nivel local, puede dejarse todo activado. Sin embargo, cuando se esté listo para distribuir la aplicación, se tendrá que construir el modo de lanzamiento, e incluir todos los DLLs que se usen, y eliminar los DLLs que no se necesiten.

11.2. ARCHIVOS DE CONFIGURACIÓN

Ogre ejecuta varios archivos de configuración. Estos controlan qué plugins se cargan, dónde se localizan los recursos de la aplicación, y así sucesivamente. Se va a examinar brevemente cada uno de los archivos de configuración y lo que hacen. Si hay más preguntas, se puede ir a los foros de ayuda de Ogre.

Plugins.cfg

Este archivo contiene los plugins que utiliza la aplicación. Si se desea añadir o eliminar un complemento en la aplicación, se tendrá que modificar este archivo. Para quitar un plugin, simplemente hay que quitar la línea adecuada, o comentarla poniendo un # al comienzo de la sentencia. Para añadir un plugin, hay que añadir una línea como "plug-in = [PluginName]". Hay que tener en cuenta que no se pone .dll al final del nombre del plugin. Tampoco hay que añadir "RenderSystem_" o "Plugin_" al inicio.

También se puede definir la localización dónde Ogre busca los plugins cambiando la variable "PluginFolder". Se pueden utilizar rutas relativas o absolutas, pero no se pueden utilizarlas variables de entorno como $(AlgunaVariable).

Resources.cfg

Este archivo contiene una lista de directorios que Ogre explora en busca de recursos. Los recursos incluyen guiones, mallas, texturas, etc. Se pueden utilizar ambas rutas, relativas y absolutas, pero no se pueden utilizar las variables de entorno como $(AlgunaVariable). Hay que tener en cuenta que Ogre no analiza subcarpetas, así que si hay múltiples niveles, debe de

indicarse manualmente. Por ejemplo, si hay un directorio “res\meshes” y “res\meshes\small”, hay que añadir las dos entradas al archivo de recursos.

Media.cfg

Este archivo le indica a Ogre información más detallada sobre los recursos.

Es muy poco probable que se tenga que modificar el contenido de este archivo, así que se pasará por alto.

Ogre.cfg

Este archivo está generado por la configuración de la pantalla de Ogre. Este archivo indica información sobre el ordenador y su configuración gráfica. No se debe distribuir este archivo a otras personas cuando se comparta la aplicación, ya que pueden tener diferentes configuraciones. Hay que tener en cuenta que no se debe modificar este archivo.

Quake3settings.cfg

Este archivo se utiliza con el BSPSceneManager. No será necesario este archivo a menos que se esté utilizando este SceneManager. No se debe distribuir este archivo junto con la aplicación, excepto, de nuevo, si se está usando el BSPSceneManager.

Estos son todos los archivos de configuración que manipula directamente Ogre. Ogre debe ser capaz de encontrar “plugins.cfg”, “resources.cfg” y “media.cfg” para funcionar correctamente. En tutoriales más avanzados se explicará más acerca de estos archivos y como cambiar su ubicación.

12. CONCLUSIÓN

Llegados a este punto se debe tener un conocimiento muy básico del *SceneManager*, el *SceneNode* y la clase *Entity*. No hay que estar familiarizado con todas las funciones que se han introducido. Dado que son los objetos más básicos, se utilizarán muy a menudo.

También se debe estar familiarizado con la creación de un entorno de trabajo de Ogre para los proyectos.

TUTORIAL BÁSICO 2: CÁMARAS, LUCES Y SOMBRAS

1. PRERREQUISITOS

Se asume que se tienen conocimientos del lenguaje C++ y se sabe como configurar y compilar una aplicación en Ogre. (Si se tienen problemas para configurar una aplicación para Ogre, hay una pequeña guía en el documento OGRE, entorno de trabajo de OGRE 3D). No es necesario ningún conocimiento específico de Ogre para realizar este tutorial.

2. INTRODUCCIÓN

En este tutorial se introducen algunas estructuras más de Ogre, como ampliación a lo ya aprendido. Se centrará en las luces y en cómo se usan para crear sombras en Ogre. También se cubrirán aspectos básicos sobre las cámaras.

A medida que se avanza por el tutorial, se deberá ir añadiendo código lentamente al proyecto y observando los resultados que se obtienen. En la sección 8 se puede encontrar el código completo, para que, en caso de tener problemas, pueda compararse con el propio.

3. PRIMEROS PASOS

Al igual que en el anterior tutorial, se usa un código pre-construido como base para el proyecto. Se añaden dos nuevos métodos a la clase TutorialApplication: createViewport y createCamera. Estas dos funciones ya están definidas en ExampleApplication, y este tutorial se centrará en ellas para ve rcómo se crean y se usan las cámaras y los Viewports.

Se crea un proyecto y se añade un archivo que contenga el siguiente código:

```
#include "ExampleApplication.h"
class TutorialApplication : public ExampleApplication
{
protected:
public:
TutorialApplication()
{
}
~TutorialApplication()
{
}
protected:
virtual void createCamera(void)
{
}
virtual void createViewports(void)
{
}
void createScene(void)
{
Entity *ent;
Light *light;
}
};
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
```

```
#include "windows.h"
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT
)
#else
int main(int argc, char **argv)
#endif
{
// Create application object
TutorialApplication app;
try {
app.go();
} catch( Exception& e ) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
MessageBox( NULL, e.getFullDescription().c_str(), "An exception has
occurred!", MB_OK |
MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
e.getFullDescription().c_str());
#endif
}
return 0;
}
```

Código 14: Código de inicio del Tutorial 2.

Si se está usando el OgreSDK bajo Windows, hay que asegurarse de añadir el directorio "[Directorio_OgreSDK]\sample\include" en este proyecto (el archivo ExampleApplication.h se encuentra allí). Si se utiliza la distribución de código fuente de Ogre, dicho directorio debe estar situado en el directorio "[Directorio_OgreSource]\samples\common\include". Hay que asegurarse de poder compilar el código antes de continuar a la siguiente sección **(aunque no se debe ejecutar, ya que se bloqueará hasta que se añada más código)**. Como siempre, si ocurren problemas, se pueden consultar en los foros de ayuda (http://www.ogre3d.org/forums/).

4. CÁMARAS

4.1. CÁMARAS EN OGRE

Una cámara es lo que se utiliza para ver la escena que se ha creado. Es un objeto especial que trabaja de una forma similar a los SceneNode. El objeto Cámara tiene las funciones setPosition, yaw, roll y pitch, y se puede asociar a cualquier SceneNode.

Como en los SceneNodes, la posición de las cámaras es relativa a la de los SceneNodepadre. Para realizar todos los movimientos y rotaciones, se puede pensar en las cámaras como SceneNodes.

Una cosa a destacar es que sólo se utilizará una cámara a la vez (por ahora). Es decir, no se creará una cámara para ver una porción de la escena y una segunda cámara para ver otra parte de la escena y se irán activando o desactivando en función de lo que se desee ver. En vez de eso, se crearán SceneNodes que actuaran como puntos donde se quiere que la cámara se sitúe. Cuando se quiera enfocar una parte en concreto, simplemente se asocia la cámara a un SceneNode o a otro. Se revisará esta técnica en el tutorial FrameListener.

4.2. CREANDO UNA CÁMARA

Se va a usar el método que ExampleApplication utiliza para crear la cámara por defecto.

Hay que buscar la función TutorialApplication::createCamera. La primera cosa a hacer es crear una cámara. Como las cámaras están vinculadas al SceneManager en el que residen, se usará el objeto SceneManager para crearlas. Hay que añadir esta línea de código para crear la cámara:

```
// create the camera
mCamera = mSceneMgr->createCamera("PlayerCam");
```

Código 15: Creación de una cámara.

Esto creará una cámara con el nombre "PlayerCam". Se puede usar la función getCamera del SceneManager para obtener cámaras basadas en su nombre y así no tener que mantener un puntero a ellas.

La próxima cosa a realizar es determinar la posición de la cámara y la dirección a laque está orientada. Se situarán objetos alrededor del origen, por ello se pondrá la cámara a una buena distancia en el eje z positivo y orientada hacia el origen. Hay que añadir este código a continuación del anterior:

```
// set its position, direction
mCamera->setPosition(Vector3(0,10,500));
mCamera->lookAt(Vector3(0,0,0));
```

Código 16: Posicionamiento y orientación de una cámara.

La función lookAt es muy útil. Se puede enfocar la cámara en la posición que se desee en lugar de tener que usar las funciones yaw, rotate y pitch para rotarla. El SceneNode también tiene esta función, haciendo en algunos casos, más fácil configurar la orientación de las Entitys.

Para finalizar, se configura el NearClipDistance a unas 5 unidades. El NearClipDistance de la cámara especifica cuan cerca o lejos puede estar algo antes de dejar de verlo. Configurar el NearClipDistance hace que sea más fácil ver a través de las Entitys cuando se está muy cerca de ellas. La alternativa a esto es que al estar muy cerca de un objeto no se vea nada más en la pantalla que una pequeña porción de este. También se puede configurar el FarClipDistance. Eso detendrá el motor de renderizado de cualquier objeto más lejos del valor dado. Esto se usa, principalmente, para incrementar el framerate si se están renderizando muchas cosas en la pantalla. Para fijar el NearClipDistance hay que añadir esta línea:

```
mCamera->setNearClipDistance(5);
```

Código 17: Distancia mínima a la cámara.

Para configurar el FarClipDistance simplemente hay que hacer una llamada similar a la función setFarClipDistance. (Aunque no se debería usar el FarClipDistance con el tipo de sombras llamado Stencil Shadow, el cual se usará más adelante en este tutorial).

5. VIEWPORTS

5.1. VIEWPORTS EN OGRE

Al empezar a tratar con múltiples cámaras, el concepto de la clase Viewport empieza a ser mucho más útil. Se tratará este tema porque es importante que se entienda cómo Ogre decide qué cámara usar cuando renderiza una escena. En Ogre es posible tener varios SceneManagers funcionando al mismo tiempo. También es posible dividirla pantalla en varias áreas, y tener diferentes cámaras renderizando

diferentes áreas en la pantalla. (La vista dividida en un juego para 2 jugadores, por ejemplo). (Véase Ilustración5 y 6) Aunque es posible hacer estas cosas, no se cubrirán hasta llegar a tutoriales de un nivel más avanzado.

Ilustración 5: Pantalla con un Viewport.

Ilustración 6: Pantalla con dos Viewports.

Para entender cómo Ogre renderiza una escena, hay que considerar el árbol de estructuras de Ogre: La cámara, el SceneManager, y la RenderWindow. La RenderWindow es, básicamente, la pantalla donde se muestra todo. El SceneManager crea cámaras para verla escena. Se debe indicar a la RenderWindow que cámaras mostrar en la pantalla, y la porción de ventana donde se desea renderizarlas. El área en la que se le indica a RenderWindow dónde mostrar las cámaras es el Viewport. En la mayoría de los usos de Ogre, generalmente se crea una única cámara y esta se muestra en toda la RenderWindow, necesitando tan sólo un Viewport.

En este tutorial se registra la cámara para crear un Viewport. Se usará este Viewport para fijar el color del fondo de la escena que se quiere renderizar.

5.2. CREANDO UN VIEWPORT

Hay que buscar la función TutorialApplication::createViewports. Para crear el Viewport simplemente se llama a la función addViewport de RenderWindow indicando la cámara que se está usando. La clase ExampleApplication ya ha incluido el RenderWindow a la clase mWindow, así que simplemente hay que añadir la siguiente instrucción:

```
// Create one viewport, entire window
Viewport* vp = mWindow->addViewport(mCamera);
```

Código 18: Añadiendo un Viewport.

Ahora con el Viewport, ¿qué se puede hacer con él? La respuesta es: no mucho. La cosa más importante que se puede hacer es llamar a la función setBackgroundColour para fijar el color que se desea en el fondo de la escena. Como en este tutorial se trata con luces, se pondrá un fondo negro:

```
vp->setBackgroundColour(ColourValue(0,0,0));
```

Código 19: Color de fondo de la pantalla.

Los parámetros de ColourValue son la cantidad de color rojo, verde y azul (respectivamente) que se desean, con valores entre 0 y 1. Para acabar, la cosa más importante que se necesita hacer es establecer la relación de aspecto de la cámara. Si se está usando algo diferente al Viewport a pantalla completa, entonces configurar mal dicha relación podría resultar en un aspecto realmente raro de la escena.

Se continúa y se omite que se está usando la relación de aspecto por defecto:

```
// Alter the camera aspect ratio to match the viewport
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp-
>getActualHeight()));
```

Código 20: Establece la relación de aspecto entre cámara y pantalla.

Llegados a este punto se debería ser capaz de compilar y ejecutar la aplicación, aunque no se verá nada más que la pantalla negra (pulsar la tecla ESC para salir). Hay que asegurarse de que se pueda ejecutar la aplicación sin fallos antes de continuar.

6. LUCES Y SOMBRAS

6.1. TIPOS DE SOMBRAS

Ogre soporta tres tipos de sombras:

1. Modulative Texture Shadows (SHADOWTYPE_TEXTURE_MODULATIVE)

– Es el tipo de sombras que necesita menos recursos de los tres y, por consiguiente, el menos vistoso. Crea una textura negra y blanca de la sombra y entonces la aplica a la escena. (Véase Ilustración7)

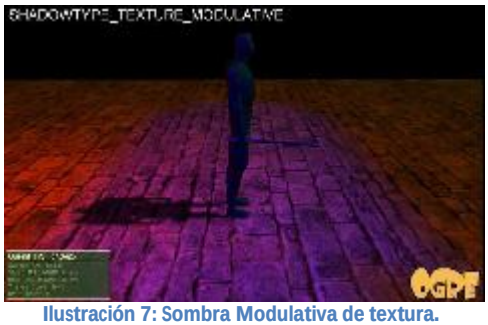

Ilustración 7: Sombra Modulativa de textura.

2. Modulative Stencil Shadows (SHADOWTYPE_STENCIL_MODULATIVE)

Esta técnica renderiza todas las sombras volúmicas como una modulación después de que todos los objetos no transparentes se hayan renderizado en la pantalla. Necesita algunos recursos más que el método anterior pero es un poco más vistosa. (Véase Ilustración8)

Ilustración 8: Sombra mmodulativa de plantilla.

3. Additive Stencil Shadows (SHADOWTYPE_STENCIL_ADDITIVE)

Esta técnica renderiza para cada luz las sombras que luego se añaden a la escena. Resulta muy costoso para la tarjeta gráfica porque, para cada luz, deberá realizar un paso adicional de renderizado (Véase Ilustración 9).

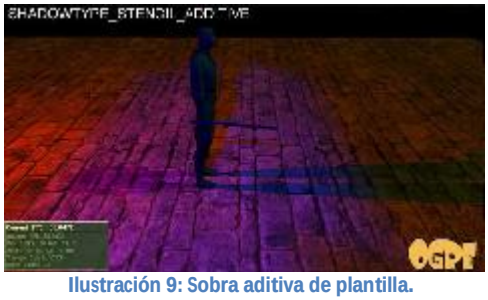

Ilustración 9: Sobra aditiva de plantilla.

Ogre no soporta sombras por shaders de forma nativa. Si se requieren este tipo de sombras habrá que incluir los programas propios de vértices y fragmentos.

6.2. USANDO SOMBRAS EN OGRE

Usar sombras en Ogre es relativamente fácil. La clase SceneManager tiene la función setShadowTechnique que se puede usar para configurar el tipo de sombra que se quiere. Entonces cuando se crea una Entity, se llama a la función setCastShadows para fijar si emite sombras o no. Ahora se fija la luz ambiente y el tipo de sombra. Hay que buscar la función TutorialApplication::createScene y añadir este código:

```
mSceneMgr->setAmbientLight(ColourValue(0, 0, 0));
mSceneMgr->setShadowTechnique(SHADOWTYPE_STENCIL_ADDITIVE);
```

Código 21: Añade técnica de sombra.

Ahora el SceneManager usa sombras tipo additive stencil shadows. Se crea un objeto en la escena y su respectiva sombra:

```
ent = mSceneMgr->createEntity("Ninja", "ninja.mesh");
ent->setCastShadows(true);
mSceneMgr->getRootSceneNode()->createChildSceneNode()-
>attachObject(ent);
```

Código 22: Creación de una entidad que emite sombras.

De nuevo, ninja.mesh ha sido pre-creado por ExampleApplication. También se necesita crear algo en lo que el ninja se apoye (para poder mostrar la sombra allí). Se crea un simple plano bajo él. Esto no pretende ser un tutorial sobre el uso del MeshManager, pero se explicarán los puntos básicos, ya que se

necesita para crear el plano. En primer lugar se necesita definir el objeto Plano, cosa que se hace determinando la distancia hasta el origen y la orientación de su eje normal.

Se podría, por ejemplo, usar planos para crear partes de un cuerpo geométrico, en ese caso habría que especificar algo diferente a 0 para la distancia al origen. Por ahora solo se quiere un plano con su eje normal paralelo al eje Y positivo (es decir, boca arriba), y sin distancia al origen:

```
Plane plane(Vector3::UNIT_Y, 0);
```

Código 23: Creación de un plano.

Ahora hay que registrar el plano para que se pueda usar en la aplicación. La clase MeshManager sigue la pista de todas las mallas que se cargan en la aplicación (por ejemplo, sigue la pista de robot.mesh y de ninja.mesh). La función createPlane coge un plano y hace una malla con sus parámetros. Esto registra el plano para usarlo:

```
MeshManager::getSingleton().createPlane("ground",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
1500,1500,20,20,true,1,5,5,Vector3::UNIT_Z);
```

Código 24: Carga del plano en la aplicación.

No se desea entrar en los usos específicos del MeshManager aún. Básicamente se ha registrado un plano de tamaño 1500x1500 y se ha creado una nueva malla llamada “ground”. Ahora se puede crear una Entity con esta malla y colocarla en la escena:

```
ent = mSceneMgr->createEntity("GroundEntity", "ground");
mSceneMgr->getRootSceneNode()->createChildSceneNode()-
>attachObject(ent);
```

Código 25: Creación de una entidad basada en un plano.

Atractivo, ¿eh? Hay dos cosas más que se necesitan hacer con “ground” antes de acabar. La primera cosa es decirle al SceneManager que no se quiere que este produzca sombras ya que es este el que se usa para mostrarlas. La segunda cosa es que se debe poner una textura en él. Las mallas robot y ninja ya tienen un script material definido para ellas. Cuando se ha creado manualmente la malla “ground”, no se ha especificado qué textura usar en ella. Se usará el script material “examples/Rockwall” que Ogre incluye en sus ejemplos:

```
ent->setMaterialName("Examples/Rockwall");
ent->setCastShadows(false);
```

Código 26: Establecimiento de material sin proyección de sombras.

Ahora que se tienen un Ninja y un suelo en la escena, hay que compilar y ejecutar el programa. Lo que se ve es... ¡nada! ¿Qué está pasando? En el tutorial anterior se añadieron robots y se mostraron bien. La razón de que el ninja no se muestre ahora es porque la luz ambiente de la escena ha sido configurada para una total oscuridad. Así que hay que añadir una luz para poder ver lo que está pasando.

6.3. TIPOS DE LUCES

Hay 3 tipos de luces en Ogre:

1. Punto de luz (LT_POINT) – Se emite luz en todas las direcciones desde un punto.

2. Spotlight (LT_SPOTLIGHT) – Funciona exactamente igual que una linterna. Se proporciona la posición del foco de la luz y la dirección en la que se quiere iluminar. También se puede modificar el ángulo de la luz y la intensidad del círculo de luz.

3. Direccional (LT_DIRECTIONAL) – Simula una luz direccional en la lejanía que se proyecta en toda la escena hacia una misma dirección. Imaginando que se tiene una escena nocturna y se quiere simular la luz de la luna. Se podría hacer configurando la luz ambiente de la escena, pero esto no sería exactamente realista ya que la luz de la luna no ilumina todos los sitios de la misma manera(ni el sol). Una forma para hacerlo más real seria con una luz direccional apuntando hacia donde la luz de la luna ilumina.

6.4. CREANDO LAS LUCES

Para crear una luz en Ogre hay que llamar a la función createLight del SceneManager,e indicar el nombre de la luz. Esto es muy parecido a cuando se crean Entitys o cámaras. Después de crear la luz, hay que configurar la posición manualmente o asociarla a un SceneNode. A diferencia del objeto cámara, la luz solo tiene las funciones setPosition y setDirection (y no todo el conjunto de funciones como translate, pitch, yaw, roll, etc.). Entonces si se necesita crear una luz estacionaria, se tendría que llamara la función setPosition. Si necesita una luz en movimiento (como una luz que sigue a un personaje), entonces hay que asociarla a un SceneNode.

Se empieza con un punto de luz básico. La primera cosa que se hace es crear la luz y configurar el tipo y la posición:

```
light = mSceneMgr->createLight("Light1");
light->setType(Light::LT_POINT);
light->setPosition(Vector3(0, 150, 250));
```

Código 27: Creación de una luz de punto.

Ahora que está creada la luz, se puede configurar el color difuso y especular.

Se va a hacer rojo:

```
light->setDiffuseColour(1.0, 0.0, 0.0);
light->setSpecularColour(1.0, 0.0, 0.0);
```

Código 28: Establecimiento de parámetros de la luz.

Se compila y ejecuta la aplicación. ¡Éxito! Ahora se puede ver al ninja y su sombra (Véase Ilustraciones 10 y 11). Una cosa a destacar es que no se ve el foco de luz. Si se desea mostrar el origen de la luz se colocaría un objeto en el lugar donde esté situada la luz para mostrarla.

Ilustración 10: Escenario iluminado del tutorial 2.

Ahora se probará con luces direccionales. Se ve como la parte frontal del ninja es negra. Se va a añadir una pequeña luz direccional amarilla iluminando la parte delantera del ninja. Para ello se creará la luz y se configurará el color de forma similar al anterior punto de luz:

Ilustración 11: Pantalla del tutorial 2 con el ninja.

```
light = mSceneMgr->createLight("Light3");
light->setType(Light::LT_DIRECTIONAL);
light->setDiffuseColour(ColourValue(.25, .25, 0));
light->setSpecularColour(ColourValue(.25, .25, 0));
```

Código 29: Creación de una luz direccional.

Como se supone que las luces direccionales provienen de distancias grandes, no hace falta configurar su posición, solo la dirección. Se fija la dirección de la luz en el eje Z positivo y en el eje Y negativo (como si viniese de una dirección con inclinación de 45 grados respecto al suelo, situada delante y encima del ninja).

```
light->setDirection(Vector3( 0, -1, 1 ));
```

Código 30: Establecimiento de la dirección de la luz.

Se compila y ejecuta la aplicación. Ahora se tienen dos sombras en la pantalla (Véase Ilustraciones 12 y 13), aunque como la luz direccional es tenue, la sombra también lo es.

Ilustración 12: Otra captura del tutorial 2.

Ilustración 13: Captura del tutorial 2, con el ninja.

El último tipo de luces que se va a probar es el spotlight. Se creará un spotlight azul:

```
light = mSceneMgr->createLight("Light2");
light->setType(Light::LT_SPOTLIGHT);
light->setDiffuseColour(0, 0, 1.0);
light->setSpecularColour(0, 0, 1.0);
```

Código 31: Creación de luz tipo lámpara.

También se necesita configurar la posición y la dirección donde el spotlight enfoca. Se creará un foco encima del ninja.

```
light->setDirection(-1, -1, 0);
light->setPosition(Vector3(300, 300, 0));
```

Código 32: Posicionamiento y enfoque de la luz.

Los spotlight también dejan especificar la anchura de la luz. Hay que imaginar por un momento una linterna. El haz de luz en el centro es más brillante que lo que lo envuelve. También se puede configurar la anchura de ambos llamando a la función setSpotlightRange:

```
light->setSpotlightRange(Degree(35), Degree(50));
```

Código 33: Establecimiento del rango de la luz de foco.

Se compila y ejecuta la aplicación. Se obtiene un ninja lila. (Véase Ilustraciones 14 y 15).

Ilustración 14: Pantalla del tutorial 2 con tres luces.

Ilustración 15: Pantalla del turotial 2 con 3 luces y ninja.

7. COSAS A PROBAR

7.1. DIFERENTES TIPOS DE SOMBRAS

En este tutorial solo se han usado el tipo de sombras additive stencil shadow. Se puede probar usando los otros dos tipos de sombras y observar qué pasa. Existen más funciones relacionadas con las sombras en la clase SceneManager.

7.2. ATENUACIÓN DE LUZ

Las luces definen la función setAttenuation que permite controlar cómo las luces se disipan cuando se alejan. Hay que añadir un llamamiento a la función en el punto de luz y probar diferentes valores de atenuación.

7.3. SCENEMANAGER::SETAMBIENTLIGHT

Se recomienda experimentar con la función setAmbientLight del mSceneMgr.

7.4. COLOR DE FONDO DE VIEWPORT

Se puede cambiar el valor por defecto de ColourValue en la función createViewports. Aunque no es apropiado poner un color diferente al negro en esta situación, es bueno saber cómo poder cambiarlo.

7.5. CAMERA::SETFARCLIPDISTANCE

En createCamera se ha configurado el setNearClipDistance. Hay que añadir un llamamiento a la función setFarClipDistance y fijarlo a 500, se observa qué pasa cuando se mueve desde que se ve el ninja hasta que no se ve con las stencil shadows.

7.6. PLANOS

No se ha explicado gran cosa sobre los planos en este tutorial (ya que no era el foco de este artículo). En posteriores tutoriales se ampliará la información. Si se desea, se puede empezar a jugar con la función createPlane.

8. CÓDIGO COMPLETO

```
#include "ExampleApplication.h"
class TutorialApplication : public ExampleApplication
{
protected:
public:
TutorialApplication()
{
}
~TutorialApplication()
{
}
protected:
virtual void createCamera(void)
{
mCamera = mSceneMgr->createCamera("PlayerCam");
mCamera->setPosition(Vector3(0,10,500));
mCamera->lookAt(Vector3(0,0,0));
mCamera->setNearClipDistance(5);
}
virtual void createViewports(void)
{
Viewport* vp = mWindow->addViewport(mCamera);
vp->setBackgroundColour(ColourValue(0,0,0));
mCamera->setAspectRatio(Real(vp->getActualWidth()) / Real(vp-
>getActualHeight()));
}
```

```
void createScene(void)
{
Entity *ent;
Light *light;
mSceneMgr->setAmbientLight( ColourValue( 0, 0, 0 ) );
mSceneMgr->setShadowTechnique( SHADOWTYPE_STENCIL_ADDITIVE );
ent = mSceneMgr->createEntity("Ninja", "ninja.mesh");
ent->setCastShadows(true);
mSceneMgr->getRootSceneNode()->createChildSceneNode()-
>attachObject(ent);
Plane plane(Vector3::UNIT_Y, 0);
MeshManager::getSingleton().createPlane("ground",
ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME, plane,
1500,1500,20,20,true,1,5,5,Vector3::UNIT_Z);
ent = mSceneMgr->createEntity("GroundEntity", "ground");
mSceneMgr->getRootSceneNode()->createChildSceneNode()-
>attachObject(ent);
ent->setMaterialName("Examples/Rockwall");
ent->setCastShadows(false);
light = mSceneMgr->createLight("Light1");
light->setType(Light::LT_POINT);
light->setPosition(Vector3(0, 150, 250));
light->setDiffuseColour(1.0, 0.0, 0.0);
light->setSpecularColour(1.0, 0.0, 0.0);
light = mSceneMgr->createLight("Light3");
light->setType(Light::LT_DIRECTIONAL);
light->setDiffuseColour(ColourValue(.25, .25, 0));
light->setSpecularColour(ColourValue(.25, .25, 0));
light->setDirection(Vector3( 0, -1, 1 ));
light = mSceneMgr->createLight("Light2");
light->setType(Light::LT_SPOTLIGHT);
light->setDiffuseColour(0, 0, 1.0);
light->setSpecularColour(0, 0, 1.0);
light->setDirection(-1, -1, 0);
light->setPosition(Vector3(300, 300, 0));
light->setSpotlightRange(Degree(35), Degree(50));
}
};
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
INT WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT
)
#else
int main(int argc, char **argv)
#endif
{
// Create application object
TutorialApplication app;
try {
app.go();
} catch(Exception& e) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
MessageBox(NULL, e.getFullDescription().c_str(), "An exception has
occurred!", MB_OK |
MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
e.getFullDescription().c_str());
```

```
#endif
}
return 0;
}
```

Código 34: Código completo del tutorial 2.

TUTORIAL BÁSICO 3: TERRENO, CIELO, NIEBLA Y OBJETO RAIZ

1. PRERREQUISITOS

Se asume que se tienen conocimientos del lenguaje C++ y se sabe como configurar y compilar una aplicación en Ogre. (Si se tienen problemas para configurar una aplicación para Ogre, hay una pequeña guía en el documento del entorno de trabajo OGRE 3D). No es necesario ningún conocimiento específico de Ogre para realizar este tutorial.

2. INTRODUCCIÓN

En este tutorial se explorará cómo manipular el terreno, el cielo y la niebla en las aplicaciones de Ogre. Después de este tutorial se entenderán las diferencias entre Skyboxes, Skyplanes y Skydomes y cómo usar cada uno de ellos. También se conocerán los diferentes tipos de niebla, y cómo usarlos.

A medida que se avance por el tutorial se deberán ir añadiendo instrucciones al propio proyecto y observar los resultados que se obtienen. En la sección 8 de este artículo se podrá encontrar el código completo, para poder compararlo con el propio en caso de tener problemas.

3. PRIMEROS PASOS

Como en los tutoriales anteriores, se parte de un código básico. Hay que crear un proyecto en el compilador y añadir este código:

```
#include "ExampleApplication.h"
class TutorialApplication : public ExampleApplication
{
protected:
public:
TutorialApplication()
{
}
~TutorialApplication()
{
}
protected:
void chooseSceneManager(void)
{
}
void createScene(void)
{
}
};
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
// Create application object
TutorialApplication app;
try {
app.go();
```

```
} catch(Exception& e) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
MessageBox(NULL, e.getFullDescription().c_str(), "An exception has
occurred!", MB_OK |
MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
e.getFullDescription().c_str());
#endif
}
return 0;
}
```

Código 35: Código de partida del tutorial 3.

Si se está usando el OgreSDK bajo Windows, hay que asegurarse de añadir el directorio "[Directorio_OgreSDK]\sample\include" en el proyecto (el archivo ExampleApplication.h se encuentra allí). Si se utiliza la distribución de código fuente de Ogre, dicho directorio debe estar situado en el directorio "[Directorio_OgreSource]\samples\common\include". Hay que asegurarse de poder compilar el código antes de continuar a la siguiente sección **(aunque no hay que intentar ejecutarlo, ya que se bloqueará hasta que se añada más código)**. Como siempre, si ocurren problemas, se pueden consultar los foros de ayuda (http://www.ogre3d.org/forums/).

4. EL OBJETO RAÍZ Y LA CREACIÓN DEL SCENEMANAGER

4.1. RAÍZ

En esta demo se renderizará un terreno en Ogre. Para hacer esto se necesita configurar el TerrainSceneManager en lugar del SceneManager que ExampleApplication configura por defecto. Hay que buscar la función chooseSceneManager y añadir este código:

```
mSceneMgr = mRoot->createSceneManager(ST_EXTERIOR_CLOSE);
```

Código 36: Activación de un TerrainSceneManager.

El objeto raíz (mRoot es un ejemplo de raíz) es el núcleo de los objetos de Ogre. En este fragmento de código se está diciendo al nodo raíz que se quiere un SceneManager del tipo ST_EXTERIOR_CLOSE. El objeto raíz entonces busca en el SceneManagerEnumerator hasta encontrar el SceneManager del tipo que se ha indicado.

Una vez la aplicación esté configurada, rara vez se tendrá que tratar con el objeto raíz de Ogre, y no siempre se interactuará con el SceneManagerEnumerator directamente.

4.2. CREACIÓN DEL SCENEMANAGER

Hay que hablar ahora sobre la creación y el almacenamiento del SceneManager para ahorrar futuras confusiones. El SceneManager no es único. Se pueden crear tantos como se desee y, a diferencia con los SceneNodes/Luces/etc se puede crear directamente con la instrucción "new SceneManager()" (no hay que usar el método createSceneManager, pero debería hacerse). Se pueden tener múltiples SceneManager conteniendo geometrías separadas y diferentes Entitys al mismo tiempo. Se pueden intercambiar entre cada uno de ellos al mismo tiempo recreando el Viewport (esto se cubre en el tutorial número 4 del nivel intermedio) o mostrar diferentes SceneManager al mismo tiempo usando varios Viewports. ¿Por qué se debería usar la función createSceneManager en lugar de crear el objeto SceneManager manualmente? Bueno, el sistema de plugins en Ogre da una gran flexibilidad al trabajar

con un sólo SceneManager. Hay unos cuantos tipos de escena definidos en la enumeración SceneType. Hasta ahora ExampleApplication había estado escogiendo ST_GENERIC como SceneManager. Ahora se puede pensar que es el SceneManager básico, pero esto es cierto sólo si no se ha manipulado el archivo plugins.cfg. Sino este no es el que se ha estado usando! El plugin OctreeSceneManager se registra a sí mismo como ST_GENERIC y sobrescribe la clase SceneManager básica. OcreeSceneManager usa un sistema de elección de objetos que no es visible y generalmente es más rápido que el SceneManager normal. Si se ha borrado el plugin OctreeSceneManager del archivo plugins.cfg entonces probablemente se estará usando el SceneManager básico al pedir el ST_GENERIC, u otro mejor dependiendo de los plugins. Esto es lo bueno del sistema.

Hasta ahora sólo se ha cubierto la mayoría de los usos básicos del objeto raíz cuando se crea el SceneManager. De hecho, se puede solicitar un SceneManager de una string en lugar de utilizar la enumeración SceneType. Ogre usa una fábrica flexible de SceneManager, que permite definir cualquier tipo de SceneManager y crearlo y destruirlo como se desee. Por ejemplo, si se ha instalado un plugin que permite crear un SceneManager llamado "FooSceneManager". Se crearía uno de estos con la siguiente instrucción:

```
// do not add this to the project
mSceneMgr = mRoot->createSceneManager("FooSceneManager");
```

Código 37: Creación de un SceneManager alternativo.

Esto crearía un FooSceneManager con el nombre por defecto. Es recomendable llamar a los SceneManagers por el segundo parámetro de createSceneManager. Por ejemplo, si se quieren crear dos SceneManager con dos nombres, se haría lo siguiente.

```
// do not add this to the project
mSceneMgr1 = mRoot->createSceneManager("FooSceneManager", "foo");
mSceneMgr2 = mRoot->createSceneManager("FooSceneManager", "bar");
```

Código 38: Creación de varios SceneManagers.

Dándole nombre a los SceneManager, no hay que mantener un puntero en ellos. El objeto raíz lo hará automáticamente. Si más tarde se desea usar el "foo" o el "bar" SceneManager, entonces se puede hacer lo siguiente:

```
// do not add this to the project
SceneManager *foo = mRoot->getSceneManager("foo");
SceneManager *bar = mRoot->getSceneManager("bar");
```

Código 39: Recuperación de SceneManagers.

Cuando se acaba con un SceneManager, hay que usar la función destroySceneManager para destruirlo y aprovechar la memoria.

Aunque no se cubre en este tutorial, también se pueden definir fábricas de SceneManagers propias por subclases de la clase SceneManagerFactory. Esto es útil cuando se ha creado un SceneManager propio o si se quiere tomar un SceneManager estándar y realizar algunos cambios en él antes de pasar por la aplicación (como creando cámaras, creando luces, cargando geometrías, y así sucesivamente).

5. TERRENO

5.1. AÑADIENDO EL TERRENO A LA ESCENA

Ahora que está todo esto claro, es el momento de crear el Terreno. El SceneManager base define el método setWorldGeometry, que las subclases usan para la creación de la mayoría de efectos de la escena. Con la clase TerrainSceneManager, este espera un archivo desde el cual cargar la configuración del terreno. Hay que buscar la función TutorialApplication::createScene y añadir este código:

```
mSceneMgr->setWorldGeometry("terrain.cfg");
```

Código 40: Establecimiento de un terreno mediante un fichero de configuración.

Se compila y ejecuta el programa. Esto es fácil. Es posible que se quiera configurar la cámara para que empiece en un lugar sobre el terreno, en lugar del lugar actual, que es bajo el terreno.

5.2. EL ARCHIVO TERRAIN.CFG

Existen varias opciones en el archivo terrain.cfg, y sólo se cubrirán las más básicas para cambiar las imágenes usadas para generar el terreno. La cosa más importante a destacar sobre el TerrainSceneManager es que ha sido diseñado con la funcionalidad de búsqueda en mente, aunque aún no ha sido implementada. La búsqueda de terreno es un sistema donde el terreno se separa en trozos, y solo se muestra cuando el usuario puede verlo. Esto permite definir un mundo muy grande y ser capaz de usarlo sin necesidad de dejar caer el framerate en una cifra significante. En Ogre hay plugins que hacen esto: Paging Scene Manager.

El TerrainSceneManager usa Heigtmaps para generar el terreno. Se puede especificar el heightmap que se quiere usar configurando la propiedad “Heightmap.image”. Se puede configurar la textura que se desea usar para el terreno gracias a la propiedad “WorldTexture”. El SceneManager del terreno también permite especificar la propiedad “DetailTexture”, que está entrelazado con el WorldTexture para darle un toque más realista al terreno. Se debería buscar cada imagen especificada en el archivo terrain.cfg y echarle un ojo (deberían encontrarse en el directorio Media/materials/textures).

Los detalles sobre cómo crear heightmaps son discutidos en los foros.

5.3. ILUMINANDO EL TERRENO

Todo el anterior tutorial se ha hablando sobre las luces y las sombras, pero la mala noticia es que no es fácil trabajar con esto en el TerrainSceneManager. Por ahora, sólo se sabe que es mucho más fácil tomar los detalles de las texturas y añadirles los efectos de luz. También se cubre la manera de hacer “oscuridad falsa” en la sección de la Niebla.

6. CIELO

Ogre proporciona tres diferentes tipos de cielo: SkyBoxes, SkyDomes y SkyPlanes.

Se explica cada uno de ellos con detalle. Hay que añadir la siguiente instrucción al método chooseSceneManager para usar las texturas de ejemplo:

```
ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
```

Código 41: Inicialización de todos los grupos de recursos.

6.1. SKYBOXES

Un SkyBox es básicamente un cubo gigante que envuelve a todos los objetos de la escena. Hay que añadir esta línea de código a createScene:

```
mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox");
```

Código 42: Establecimiento de un SkyBox.

Se compila y ejecuta el programa. Bonito ¿eh? (Véase Ilustración 16) (Hay que remarcar que el Skybox contiene ahora una textura de baja resolución, si la textura tuviese una alta resolución quedaría mucho mejor). Existen bastantes parámetros muy útiles para los SkyBoxes que se pueden fijar cuando se llama a setSkyBox. La primera opción es si activar o no el SkyBox. Si se quiere desactivar simplemente hay que escribir "mSceneMgr->setSkyBox(false, "");". El segundo parámetro es la textura a usar en el SkyBox.

Es importante entender la función del tercer y el cuarto parámetro de setSkyBox.

El tercer parámetro fija la distancia a la que el SkyBox se encuentra de la cámara, y el cuarto fija si el SkyBox se dibuja antes que el resto de la escena o después. Entonces, se puede ver qué sucede cuando se cambia el parámetro de la distancia desde el valor por defecto de 5000 unidades a algo muy pequeño:

```
mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox", 10);
```

Código 43: Establecimiento de un SkyBox de forma alternativa.

Ilustración 16: Terreno y SkyBox.

No cambia nada! Esto es porque el cuarto parámetro que controla si dibujar el SkyBox primero o no está fijado a true por defecto. Si el SkyBox se dibuja primero, entonces todo lo renderizado antes (como el terreno) se dibujara encima de él, y el SkyBox siempre aparecerá en el fondo. (Obsérvese que no debería fijar la distancia a un valor próximo al dado al fijar el NearClipDistance en la cámara o este no se mostrará!) No es recomendable dibujar el SkyBox primero, ya que entonces se renderizará todo. Cuando se dibuja último, solo las porciones visibles se renderizan, ahorrando así recursos.

Se configura el SkyBox para dibujarse lo último:

```
mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox", 5000, false);
```

Código 44: Establecimiento de un SkyBox lejano.

De nuevo, parece lo mismo que antes, pero ahora las partes del SkyBox que no son visibles no se renderizan. Hay una cosa con la que se debe ir con cuidado cuando se usan estas técnicas. Si se configura el SkyBox para estar muy cerca, podría estar cortando partes de la geometría de la escena. Por ejemplo, probando esto:

```
mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox", 100, false);
```

Código 45: Establecimiento de un SkyBox cercano.

Ilustración 17: Terreno con SkyBox cercano.

Como se puede observar (Véase Ilustración17), el terreno atraviesa el SkyBox perdiendo así la vista de parte de él. Definitivamente no es lo que se quiere. Si se usa el SkyBox en la aplicación hay que decidir cómo se quiere usar. La velocidad que se obtiene renderizando el SkyBox después del terreno es muy modesta, y se debe tener en cuenta la geometría de la escena. Generalmente hablando, es muy recomendable dejar los parámetros 3 y 4 en los valores por defecto.

6.2. SKYDOMES

El SkyDomes es muy similar al SkyBox, y se usa llamando a setSkyDome. Un cubo gigante es creado alrededor de la cámara, pero la diferencia más grande es que la textura se proyecta en el SkyBox de una manera esférica. En realidad se está mirando a un cubo, pero parece como si la textura se representase alrededor de una esfera. El principal inconveniente de este método es que el fondo del cubo no tiene textura, por lo que siempre se necesita tener algún tipo de terreno que esconda la base.

La textura de ejemplo que Ogre proporciona para el SkyDome lo mostrará. Hay que borrar la instrucción de llamada a setSkyBox en createScene y añadir este código en su lugar:

```
mSceneMgr->setSkyDome(true, "Examples/CloudySky", 5, 8);
```

Código 46: Establecimiento de un SkyDome.

Cuando se ejecute (Véase Ilustración18), hay que mover la cámara hasta el final del terreno y enfocarla hacia debajo del terreno.

Ilustración 18: Terreno con SkyDome.

Después de ver esto, se pulsa el botón R para fijar la visión de malla. Como se puede observar, se sigue viendo un cubo (sin base), pero parece como si las nubes estuvieran distribuidas sobre una esfera.

(Destacar también que el movimiento de las nubes es una propiedad del material “Examples/CloudSky”, no del SkyDome en sí.)

Los dos primeros parámetros de setSkyDome son los mismos que los de setSkyBox, y se puede desactivar el SkyDome llamando a “mSceneMgr->setSkyDome(false,””);”. El tercer parámetro es la curvatura usada en el SkyDome. La referencia API sugiere usar valores entre 2 y 65; los más bajos para mejores efectos de distancia y los valores altos para menos distorsión. Se puede probar configurando el tercer parámetro a 2 (Véase Ilustración 19) y 65 (Véase Ilustración 20) y observar la diferencia.

Ilustración 19: Terreno con SkyDome y poca curvatura.

El cuarto parámetro es el número de veces que la textura es hecha mosaico, valor que se deberá modificar en función del tamaño de la textura. Hay que asegurarse de tener en cuenta que este parámetro es un valor real y no un entero. Puede fijarse a 1.234, si es el valor que mejor queda en la aplicación. El quinto y sexto parámetro son la distancia y el hecho de dibujar primero o no, respectivamente, cosa que ya se ha cubierto en la sección de SkyBox.

Ilustración 20: Terreno con SkyDome y mucha curvatura.

6.3. SKYPLANES

Los SkyPlanes son muy diferentes a los SkyBoxes y a los SkyDomes. En vez de un cubo donde renderizar la textura del cielo, se usa simplemente un plano. (Hay que destacar que para poder usar bien los SkyPlanes deben encontrarse cerca del medio del terreno y cerca del cielo).Se borra el código sobre el SkyDome en createScene. La primera cosa que se hace es crear el plano, y orientarlo hacia abajo. El método setSkyPlane que se usará no tiene el parámetro de distancia como el SkyBox o el SkyDome. En vez de esto, se puede configurar gracias a la variable ‘d’ del plano:

```
Plane plane;
plane.d = 1000;
plane.normal = Vector3::NEGATIVE_UNIT_Y;
```

Código 47: Creación de un plano, para un SkyPlane.

Ahora que está definido el plano, se crea el Skyplane. Hay que observar que el cuarto parámetro es la medida del SkyPlane (en este caso 1500x1500 unidades) y el quinto parámetro es el número de veces que se quiere hacer la textura mosaico:

```
mSceneMgr->setSkyPlane(true, plane, "Examples/SpaceSkyPlane", 1500,
75);
```

Código 48: Establecimiento de un SkyPlane.

Se compila y ejecuta el programa. (Véase Ilustración21) Hay dos problemas al crear el SkyPlane. La primera de todas, la textura que se ha usado tiene una resolución demasiado baja, y no funciona bien. Esto se puede arreglar cambiando la textura por una con una mejor resolución. De todas formas, el principal problema con esta técnica es que al mirar hacia el horizonte, se puede ver donde acaba el SkyPlane. Aunque tenga una buena textura, si puede verse el horizonte no quedará bien. El uso de los SkyPlane es útil cuando existen muros altos (o colinas) alrededor del Viewport en la escena. Usar unSkyPlane en esta situación es mucho más rentable (en cuanto a recursos gráficos) que usar un SkyBox o un SkyDome.

Ilustración 21: Terreno con SkyPlane.

Afortunadamente, esto no es todo lo que se puede hacer con el SkyPlane (como se verá en el apartado sobre la niebla). El sexto parámetro es parecido al "renderFirst" (decide si dibujar el cielo primero o último) comentado en las secciones anteriores. El séptimo parámetro permite especificar la curvatura del SkyPlane, para así poder usar una superficie curvada en vez de un plano. También se puede fijar el número de segmentos en X e Y que se usan para crear el SkyPlane. El octavo parámetro y el noveno son estos números de segmentos, respectivamente:

```
mSceneMgr->setSkyPlane(true, plane, "Examples/SpaceSkyPlane", 1500,
50, true, 1.5f,
150, 150);
```

Código 49: Establecimiento de un SkyPlane con todos los parámetros.

Se compila y ejecuta la aplicación. (Véase Figura 22) Ahora el SkyPlane luce mucho mejor. También se puede usar otra textura:

```
mSceneMgr->setSkyPlane(true, plane, "Examples/CloudySky", 1500, 40,
true, 1.5f, 150,150);
```

Se compila y ejecuta la aplicación. (Véase Ilustración23). El movimiento de las nubes y la forma de mosaico parece hacer que se vea un poco peor que un SkyDome, especialmente cuando se encuentra cerca del borde del terreno y se asoma al horizonte.

Otra nota: se puede borrar el SkyPlane llamando a 'mSceneMgr->setSkyPlane (false,Plane (), "");'

Ilustración 22: Terreno con SkyPlane.

Ilustración 23: Terreno con SkyPlane.

6.4. ¿CUAL DE ELLOS USAR?

Qué tipo de cielo usar depende completamente de la aplicación. Si se tiene que ver todo a su alrededor, incluso en la dirección negativa del eje Y, entonces sólo se tiene la opción de usar el SkyBox. Si se tiene terreno, o algo parecido a suelo que bloquea la vista hacia el eje negativo Y, entonces el SkyDome dará un resultado más realista. Para áreas en las que no se puede ver el horizonte (como un valle alrededor de montañas en todas direcciones, o el interior de un castillo), el SkyPlane dará resultados muy buenos a cambio de pocos recursos. La principal razón para usar el SkyPlane, como se verá en la siguiente sección, es que funciona bien con los efectos de niebla.

7. NIEBLA

7.1. INTRODUCCIÓN A LA NIEBLA

La niebla en Ogre es muy fácil de usar. Hay una advertencia que se necesita conocer antes de empezar a usar niebla en programas. Cuando se usa el TerrainSceneManager, hay que ser cuidadoso para llamar a la función setFog antes de la función setWorldGeometry (sise usa OpenGL para renderizar). (En otros SceneManagers generalmente no importa).

Antes de empezar, hay que borrar todo el contenido de la función createScene excepto la instrucción de llamamiento a setWorldGeometry.

La cosa más importante a conocer sobre la configuración de la niebla es que no se creará una entity en el espacio vacío. En vez de eso, la niebla no es más que un filtro que se aplica a todo lo que se está viendo. Esto tiene interesantes implicaciones. La más relevante de ellas es que cuando se está mirando a la nada, no se verá la niebla. De hecho, sólo se verá el color de fondo del viewport. Entonces, para configurar correctamente la niebla, hay que fijar el color de fondo al mismo color que el de la niebla.

Existen dos tipos de nieblas: lineal y exponencial. La niebla lineal se distribuye de una forma lineal, mientras que la exponencial lo hace de manera exponencial (el espesor de la niebla crece al crecer la distancia de una forma superior a la que lo hizo en la unidad anterior). Es más fácil ver la diferencia a partir de ejemplos.

7.2. TIPOS DE NIEBLA

El primer tipo de niebla con el que se practicará es el lineal, ya que es el más fácil de entender. La primera cosa que se hará después de llamar a setWorldGeometry es configurar el color de fondo del Viewport. Se puede hacer esto sobrescribiendo la función createViewport (como se hizo en el último tutorial), pero a veces se necesitará configurarlo sin tener que recrear el viewport cada vez. Así es como se hará:

```
ColourValue fadeColour(0.9, 0.9, 0.9);
mWindow->getViewport(0)->setBackgroundColour(fadeColour);
```

Código 50: Cambio de color a un Viewport.

Se puede utilizar la función getNumViewports para obtener el número de viewports e iterara través de ellos si se tiene más de uno, pero como este caso no es raro (y como se dijo, sólo se trabajará con uno), se puede obtener el viewport directamente. Una vez fijado el color de fondo, se puede proceder a crear la niebla. Hay que recordar que se debe añadir este código después de la instrucción de setWorldGeometry si se usa DirectX (o antes de setWorldGeometry si se usa OpenGL)

```
mSceneMgr->setFog(FOG_LINEAR, fadeColour, 0.0, 50, 500);
```

Código 51: Creación de niebla lineal.

El primer parámetro de la función setFog es el tipo de niebla (en este caso, lineal). El segundo parámetro a configurar es el color de la niebla que se quiere usar (en este caso un verde muy flojo). El tercer parámetro no se usa en la niebla lineal. El cuarto y quinto parámetro especifican el rango donde la niebla se distribuye. En este caso se fija quela niebla empiece en 50 y acabe en 500. Esto significa que desde 0 a 50 unidades enfrente de la cámara no habrá niebla. Entre 50 y 500 unidades enfrente de la cámara es donde se distribuirá la niebla. A 500 unidades desde la cámara no se verá nada más que niebla. Se compila y ejecuta la aplicación. (Véase Ilustración24).

Ilustración 24: Terreno con niebla lineal.

El otro tipo de niebla que se puede usar es la niebla exponencial. En lugar de fijar sitios donde empieza y termina la niebla, se fija la densidad de la niebla (los parámetros cuatro y cinco no tienen uso). Hay que reemplazar la anterior instrucción de llamada a setFog por esta:

```
mSceneMgr->setFog(FOG_EXP, fadeColour, 0.005);
```

Código 52: Creación de niebla exponencial.

Se compila y ejecuta la aplicación. (Véase Ilustración25). Se obtiene un efecto diferente.

Existe otra función de niebla exponencial que es más grave que la primera. Se reemplaza la anterior instrucción setFog por esta:

```
mSceneMgr->setFog(FOG_EXP2, fadeColour, 0.003);
```

Código 53: Creación de niebla exponencial con otros parámetros.

Se compila y ejecuta la aplicación de nuevo. (Véase Ilustración26) Hay que experimentar con los tres tipos de niebla y escoger la que más se adecue a la aplicación.

Ilustración 25: Terreno con niebla exponencial 1.

Ilustración 26: Terreno con niebla exponencial 2.

7.3. LA NIEBLA Y EL CIELO

Se pueden encontrar algunos problemas interesantes cuando se prueba a usar niebla con un SkyBox o SkyDome. Como los SkyDomes y SkyBoxes son sólo cubos, usarlos con niebla es problemático ya que la niebla trabaja de una manera esférica. Hay que borrar el contenido del método createScene. Si se escogen inteligentemente los parámetros del SkyDome y de la niebla, se puede ver el problema directamente:

```
ColourValue fadeColour(0.9, 0.9, 0.9);
mSceneMgr->setFog(FOG_LINEAR, fadeColour, 0.0, 50, 515);
mWindow->getViewport(0)->setBackgroundColour(fadeColour);
mSceneMgr->setWorldGeometry("terrain.cfg");
mSceneMgr->setSkyDome(true, "Examples/CloudySky", 5, 8, 500);
```

Código 54: Recreación de problema usando SkyDome y niebla.

Se compila y ejecuta la aplicación. (Véase Ilustración 27) Si se mueve la cámara por los alrededores, se verán diferentes porciones del SkyDome en función de qué parte del SkyDome se esté mirando (observando el color azul a los lados, pero no en el medio).

Ilustración 27: Terreno con niebla y SkyDome.

Esto no es lo que realmente se quiere. Otra opción es usar un SkyPlane en su lugar.

Se reemplaza el código en createScene por este otro:

```
ColourValue fadeColour(0.9, 0.9, 0.9);
mSceneMgr->setFog(FOG_LINEAR, fadeColour, 0.0, 0, 130);
mWindow->getViewport(0)->setBackgroundColour(fadeColour);
Plane plane;
plane.d = 100;
plane.normal = Vector3::NEGATIVE_UNIT_Y;
mSceneMgr->setWorldGeometry("terrain.cfg");
mSceneMgr->setSkyPlane(true, plane, "Examples/CloudySky", 500, 20,
true, 0.5, 150,150);
```

Código 55: Código con niebla y SkyPlane.

Se compila y ejecuta la aplicación de nuevo. (Véase Ilustración 28) Luce mejor. Si se mira hacia arriba se puede ver el cielo (como en la vida real si la niebla es baja). No importa si se usa o no una curvatura, esto solventa el problema del horizonte al usar SkyPlanes.

Hay una forma de hacer que la niebla no afecte al cielo enteramente, pero esto requiere modificar la textura del cielo, cosa que escapa a este tutorial.

Ilustración 28: Terreno con niebla y SkyDome.

7.3. NIEBLA COMO OSCURIDAD

Puede que no se desee utilizar cielo cuando se configura la niebla, ya que si la niebla es suficientemente espesa no se puede ver el cielo de ninguna manera. El truco que se ha descrito anteriormente permite realizar un efecto gráfico ingenioso que en algunos casos puede ser útil. En lugar de establecer la niebla de un color brillante, se configura a uno muy oscuro y ver qué pasa. (Se observa que se ha fijado la distancia del SkyPlane a la cámara a sólo 10 unidades):

```
ColourValue fadeColour(0.1, 0.1, 0.1);
```

```
mWindow->getViewport(0)->setBackgroundColour(fadeColour);
mSceneMgr->setFog(FOG_LINEAR, fadeColour, 0.0, 10, 150);
mSceneMgr->setWorldGeometry("terrain.cfg");
Plane plane;
plane.d = 10;
plane.normal = Vector3::NEGATIVE_UNIT_Y;
mSceneMgr->setSkyPlane(true, plane, "Examples/SpaceSkyPlane", 100, 45,
true, 0.5,150, 150);
```

Código 56: Código con niebla oscura y SkyPlane.

Se compila y ejecuta la aplicación. (Véase Ilustración29)

Ilustración 29: Terreno con niebla oscura y SkyPlane.

No es demasiado terrible. Por supuesto, cuando se esté capacitado, se debería usar una iluminación adecuada en lugar de este procedimiento, pero esto muestra la flexibilidad de la niebla, y algunas de las cosas interesantes que se pueden hacer con el motor. El uso de la niebla negra podría ser una interesante manera de simular ceguera o un efecto de oscuridad en un juego con vista en primera persona.

8. CÓDIGO COMPLETO

```
#include "ExampleApplication.h"
class TutorialApplication : public ExampleApplication
{
protected:
public:
TutorialApplication()
{
}
~TutorialApplication()
{
}
protected:
void chooseSceneManager(void)
{
mSceneMgr = mRoot->createSceneManager(ST_EXTERIOR_CLOSE);
}
void createScene(void)
{
mSceneMgr->setWorldGeometry("terrain.cfg");
//mSceneMgr->setSkyBox(true, "Examples/SpaceSkyBox");
ColourValue fadeColour(0.9, 0.9, 0.9);
mWindow->getViewport(0)->setBackgroundColour(fadeColour);
mSceneMgr->setFog(FOG_LINEAR, fadeColour, 0.0, 50, 500);
}
};
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
```

```
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
// Create application object
TutorialApplication app;
try {
app.go();
} catch(Exception& e) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
MessageBoxA(NULL, e.getFullDescription().c_str(), "An exception has
occurred!", MB_OK
| MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
e.getFullDescription().c_str());
#endif
}
return 0;
}
```

Código 57: Código completo del tutorial 3.

TUTORIAL BÁSICO 4: FRAME LISTENERS Y UNBUFFERED INPUT

1. PRIMEROS PASOS

Como en los tutoriales anteriores, se parte de un código básico. Se crea un proyecto en el compilador y se añade este código:

```
#include "ExampleApplication.h"

class TutorialFrameListener : public ExampleFrameListener
{
public:
    TutorialFrameListener(RenderWindow* win, Camera* cam, SceneManager
*sceneMgr)
        : ExampleFrameListener(win, cam, false, false)
{
    }

// Overriding the default processUnbufferedKeyInput so the key updates
we define
    // later on work as intended.
    bool processUnbufferedKeyInput(const FrameEvent& evt)
{
         return true;
    }

// Overriding the default processUnbufferedMouseInput so the Mouse
updates we define
    // later on work as intended.
    bool processUnbufferedMouseInput(const FrameEvent& evt)
{
         return true;
    }

bool frameStarted(const FrameEvent &evt)
{
        return ExampleFrameListener::frameStarted(evt);
    }
protected:
    bool mMouseDown;       // Whether or not the left mouse button was
down last frame
    Real mToggle;          // The time left until next toggle
    Real mRotate;          // The rotate constant
    Real mMove;            // The movement constant
    SceneManager *mSceneMgr;   // The current SceneManager
    SceneNode *mCamNode;   // The SceneNode the camera is currently
attached to
};

class TutorialApplication : public ExampleApplication
{
public:
    TutorialApplication()
    {
    }

    ~TutorialApplication()
    {
    }
```

```
protected:
    void createCamera(void)
    {
    }

    void createScene(void)
    {
    }

    void createFrameListener(void)
    {
    }
};

#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    TutorialApplication app;

    try {
        app.go();
    } catch(Exception& e) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
        MessageBox(NULL, e.getFullDescription().c_str(), "An exception
has occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
                e.getFullDescription().c_str());
#endif
    }

return 0;
}
```

Código 58: Código de partida del tutorial 4.

Si se está usando el OgreSDK bajo Windows, hay que asegurarse de añadir el directorio "[Directorio_OgreSDK]\sample\include" en el proyecto (el archivoExampleApplication.h se encuentra allí). Si se utiliza la distribución de código fuente de Ogre, dicho directorio debe estar situado en el directorio "[Directorio_OgreSource]\samples\common\include". Hay que asegurarse de poder compilar el código antes de continuar a la siguiente sección **(aunque no hay que intentar ejecutarlo, ya que se bloqueará hasta que se añada más código)**. Como siempre, si ocurren problemas, se pueden consultar los foros de ayuda (http://www.ogre3d.org/forums/).

2. FRAMELISTENERS

2.1 INTRODUCCIÓN

En tutoriales previos sólo se vio lo que se puede hacer cuando se añade código al método createScene. En Ogre, se puede registrar una clase para recibir notificaciones antes y después de que se renderice un frame en la pantalla. El interfaz FrameListener define dos funciones:

Bool frameStarted(const FrameEvent& evt).

Bool frameEnded(const FrameEvent& evt).

El bucle principal de Ogre (Root::startRendering) puede parecerse a esto:

1. El objeto Root llama al método frameStarted en todos los FramesListeners registrados.
2. El objeto Root renderiza un frame.
3. El objeto Root llama al método frameEnded en todos los FrameListeners registrados.

Este bucle continúa hasta que uno de los FrameListeners retorna false desde una frameStarted o frameEnded. El valor de retorno de estas funciones simplemente indica "seguir renderizando". Si se retorna false desde cualquiera, finaliza el programa. El objeto FrameEvent contiene dos variables, pero sólo timeSinceLastFrame es útil en un FrameListener. Esta variable contiene cuánto tiempo ha pasado desde que el último frameStarted o frameEnded se lanzó. Hay que tener en cuenta que en el método frameStarted, FrameEvent::timeSinceLastFrame contendrá el tiempo pasado desde el último frameStarted.

Un concepto importante para los FrameListeners en Ogre es que no se puede determinar el orden en que se llama a los FrameListeners. Si se necesita asegurar las llamadas en orden, sólo se puede registrar un único FrameListener y que él llame a los objetos en orden.

Puede darse cuenta de que el bucle principal realmente sólo hace las tres cosas, y que tampoco pasa nada entre frameEnded y frameStarted. La decisión de donde poner el código es de cada uno. Se puede poner todo en un gran método frameStarted o frameEnded, o dividirlo entre los dos.

2.2 REGISTRANDO UN FRAMELISTENER

El código anterior compila, pero tenemos que sobrescribir el método createFrameListener de ExampleApplication y de createCamera, si se ejecuta la aplicación no habrá forma de matarla. Primero hay que solucionar 6 problemas antes de continuar con el resto del tutorial.

Hay que buscar el método TutorialApplication::createCamera y añadir el siguiente código:

```
// create camera, but leave at default position
mCamera = mSceneMgr->createCamera("PlayerCam");
        mCamera->setNearClipDistance(5);
```

Código 59: Establecimiento de una cámara.

No se ha realizado nada fuera de lo común. La única razón para sobrescribir el método es porque el método mueve la cámara y cambia su orientación, cosa que no se quiere para este tutorial.

Como la clase Root es la encargada de renderizar los frames, también tiene que encargarse de mantener el registro de los FrameListeners. La primera cosa que hay que hacer es crear una instancia de la clase TutorialFrameListener y registrarla con el objeto Root. En el método TutorialApplication::createFrameListener, hay que añadir este código:

```
        // Create the FrameListener
         mFrameListener = new TutorialFrameListener(mWindow, mCamera,
mSceneMgr);
         mRoot->addFrameListener(mFrameListener);
```

Código 60: Creación de un FrameListener.

Las variables mRoot y mFrameListener se definen en la clase ExampleApplication. El método addFrameListener añade el FrameListener, y el método removeFrameListener elimina un FrameListener (el FrameListener no recibirá actualizaciones). Hay que tener en cuenta que los métodos add/remove sólo usan un puntero a un FrameListener. Esto significa que se necesitará guardar un puntero por cada FrameListener que se cree para poder borrarlo luego.

El ExampleFrameListener (de donde deriva TutorialFrameListener), proporciona una función showDebugOverlay(bool), que dice a ExampleApplication si mostrar o no la caja de tasas de frame en la esquina inferior. Se habilitará:

```
//Show the frame stats overlay
mFrameListener->showDebugOverlay(true);
```

Código 61: Habilitación de los fps.

Compilar y continuar.

3. CONFIGURANDO LA ESCENA

3.1 INTRODUCCIÓN

Antes de entrar directamente en el código, se comenta lo que se hará y la función que se desea obtener en la escena.

Se mostrará un objeto (un ninja) en la escena, y un punto de luz. Si se pulsa el botón izquierdo del ratón, la luz se encenderá o apagará. Manteniendo pulsado el botón derecho se cambia al modo “mouse look” (que se puede mirar alrededor con la cámara). También se pondrán SceneNodes alrededor de la escena que se acoplarán a la cámara desde diferentes puntos de vista. Pulsando el botón 1 y 2, se elige el punto de vista desde donde ver la escena.

3.2 EL CÓDIGO

En el método TutorialApplication::createScene, la primera cosa a hacer es establecer la luz ambiente muy floja. Los objetos deben permanecer visibles cuando la luz no esté encendida, pero tiene que notarse la diferencia cuando se enciende.

```
mSceneMgr->setAmbientLight(ColourValue(0.25, 0.25, 0.25));
```

Código 62: Creación de una luz ambiente.

Ahora, se añade una entidad Ninja a la escena:

```
Entity *ent = mSceneMgr->createEntity(“Ninja”, “ninja.mesh”);
SceneNodo *node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode(“NinjaNode”);
```

```
Node->attachObject(ent);
```

Código 63: Creación de un ninja en la escena.

Ahora se crea un punto blanco de luz, y se coloca en la escena, a poca distancia (relativamente) del ninja:

```
Light *light = mSceneMgr->createLight("Light1");
        light->setType(Light::LT_POINT);
        light->setPosition(Vector3(250, 150, 250));
        light->setDiffuseColour(ColourValue::White);
light->setSpecularColour(ColourValue::White);
```

Código 64: Creación de un punto de luz.

Ahora hay que crear los ScenesNodes que se adjuntarán con la cámara

```
// Create the scene node
        node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("CamNode1", Vector3(-400, 200, 400));
        node->yaw(Degree(-45));
        node->attachObject(mCamera);

// create the second camera node
        node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("CamNode2", Vector3(0, 200, 400));
```

Código 65: Creación de SceneNodes donde acoplar la cámara.

Con esto queda finalizada la clase TutorialApplication.

4.TUTORIALFRAMELISTENER

4.1 VARIABLES

Se ha definido algunas variables en la clase TutorialFrameListener:

```
bool mMouseDown;       // Whether or not the left mouse button was
down last frame
    Real mToggle;          // The time left until next toggle
    Real mRotate;          // The rotate constant
    Real mMove;            // The movement constant
    SceneManager *mSceneMgr;   // The current SceneManager
    SceneNode *mCamNode;   // The SceneNode the camera is currently
attached to
```

Código 66: Algunas variables definidas en la clase TutorialFrameListener.

El mSceneMgr tiene un puntero al SceneManager actual y mCamNode tiene el SceneNode actual al que está adjunta la Camara. El mRotate y mMove son constantes de rotación y movimiento. Si se quieren cambiar estos parámetros, basta con cambiar su valor.

Las otras dos variables (mToggle y mMouseDown) controlan las entradas. Para este tutorial se utiliza ratón “sin búfer” y teclado “sin búfer” (con búfer se verá en el próximo tutorial). Esto significa que se estará llamando a los métodos del frame listener para preguntar el estado del teclado y del ratón. Se encontrará un problema interesante cuando se intente usar el teclado para cambiar el estado de un objeto en la pantalla. Si la tecla está pulsada, se puede actuar con la información, pero ¿qué pasa en el siguiente frame? Se comprueba que la tecla está pulsada y se hace lo mismo? En algunos casos (como el movimiento con teclas) que es lo que se quiere hacer, si. Pero por ejemplo si se dice que al pulsar una tecla cambiamos el modo de luz, si en un frame salta y lo cambia, y en el siguiente continúa pulsada, se

vuelve a cambiar de nuevo. Hay que guardar el conjunto de estados de la tecla entre frames para evitar este problema. Hay dos métodos para resolver esto.

El mMouseDown almacena si el botón del ratón estaba pulsado o no en el frame anterior (si está abajo, no se realiza la acción, hasta que se suelte el botón). mToggle especifica el tiempo hasta que se puede hacer de nuevo la acción. Esto es, cuando se pulsa un botón, mToggle se establece a una cantidad de tiempo donde no pueden ocurrir acciones.

4.2 CONSTRUCTOR

La primera cosa a tener en cuenta sobre el constructor es que hay que llamar al constructor de ExampleFrameListener:

```
: ExampleFrameListener(win, cam, false, false)
```

Código 67: Constructor con las herencias de ExampleFrameListener.

La cosa importante a tener en cuenta es que la tercera y la cuarta variable deben de establecerse a false. La tercera variable especifica si se quiere usar una entrada de teclado con búfer, la cuarta indica si se quiere usar entrada de ratón con búfer.

En el constructor TutorialFrameListener, se establece el valor por defecto de las variables:

```
//key and mouse state tracking
mMouseDown = false;
mToggle = 0.0;

//Populate the camera and scene manager containers
mCamNode = cam->getParentSceneNode();
mSceneMgr = sceneMgr;

//set the rotation and move speed
mRotate = 0.13;
mMove = 250;
```

Código 68: Inicialización de las variables.

La variable mCamNode se inicializa para ser el padre actual de la cámara.

4.3 EL MÉTODO FRAMESTARTED

Ahora se comienza con la parte real del tutorial: realizar acciones en cada fotograma. Ahora mismo el método frameStarted tiene el siguiente código:

```
Return ExampleFrameListener::frameStarted(evt);
```

Código 69: Código inicial del frameStarted.

Este trozo de código permite que funcione el tutorial hasta llegar a este punto. El método ExampleFrameListner::frameStarted define un montón de comportamientos (como los enlaces de teclas, movimientos de cámaras,...). *Hay que vaciar el contenido del método TutorialFrameListener::frameStarted.*

El sistema de entrada abierta (OIS) aporta tres clases primarias para recoger entradas: teclado (Keyboard), ratón (Mouse) y joystick (Joystick). En estos tutoriales sólo se usarán el teclado y ratón.

La primera cosa que hay que hacer cuando se usan entradas sin búfer, es capturar el estado actual del teclado y ratón. Esto se hace llamando al método capture. El framework de ejemplo ya crea esos objetos en mMouse y mKeyboard. Hay que añadir el siguiente código al método ahora vacío:

```
mMouse->capture();
mKeyboard->capture();
```

Código 70: Captura de los estados iniciales de ratón y teclado.

A continuación, hay que estar seguros que se salga del programa si se pulsa la tecla de escape. Para chequear el botón que se ha pulsado se utiliza el método isKeyDown del InputReader y un KeyCode (ver API con códigos). Si se pulsa escape, se retorna false para salir del programa:

```
if (mKeyboard->isKeyDown(OIS::KC_ESCAPE))
return false;
```

Código 71: Forma de salir de la aplicación

Para continuar renderizando tiene que retornar true. Por lo que al final del método se pone la siguiente línea.

```
return true;
```

Código 72: Se continua con el renderizado.

El resto de código comentado va arriba de la línea final “return true”.

La primera cosa que hay que hacer con el FrameListener es hacer que el botón izquierdo del ratón encienda y apague la luz. Se puede encontrar si un botón del ratón está pulsado llamando a getMouseButton del InputReader con el botón por el que queremos preguntar. Normalmente 0 es el botón izquierdo, 1 es el botón derecho, y 2 es el botón central. En algunos sistemas el botón 1 es el central y el 2 el derecho.

```
Bool currMouse = mMouse->getMouseState().buttonDown(OIS::MB_Left);
```

Código 73: Comprobación del estado de un botón del ratón.

La variable currMouse será true si el botón está pulsado. A continuación se cambia la luz dependiendo de esta variable y de si no estaba pulsado en el fotograma anterior (sólo se cambia la luz cada vez que se pulsa de nuevo). Hay que tener en cuenta también que el método setVisible de la clase Light determina si el objeto realmente emite luz o no:

```
If (currMouse && ! mMouseDown)
{
        Ligth *ligth = mSceneMgr->getLigth(“Light1”);
        Light->setVisible(! Light->isVisible());
} //if
```

Código 74: Cambio del estado de la luz.

Ahora se establece la variable mMouseDown igual al contenido de currMouse. En el siguiente fotograma se preguntará si anteriormente estaba pulsado o no.

```
mMouseDown = currMouse;
```

Código 75: Establecimiento del valor de una variable.

Compilar y ejecutar la aplicación. Ahora al pulsar el botón izquierdo cambiará la luz. Hay que tener en cuenta que ya no se llama al método frameStarted de ExampleFrameListener, ya que ya no se moverá la cámara.

Este método de almacenar el estado anterior del botón funciona bien, ya que se sabe que se actúa en el estado del botón. La desventaja es tener que utilizar esto para cada tecla que se quiere recoger una acción, no se debería de tener que usar una varible boolean para ello. Una forma de evitar tener que hacer esto es no perder la vista la última vez que se presionó la tecla, y sólo permitir acciones que pasen después de cierto tiempo. Se guarda este estado en mToggle. Si mToggle es mayor que 0, no se realizan acciones, si es menor que 0, se realizan acciones. Se usará este método para las otras dos teclas.

La primera cosa que se tiene que hacer es decrementar la variable mToggle por el tiempo que ha pasado desde el último frame:

```
mToggle -= evt.timeSinceLastFrame;
```

Código 76: Estableciendo variable de control de tiempo.

Ahora se puede actuar sobre la variable. El siguiente fijado de letra a realizar es hacer que la tecla 1 enlace la Camera al primer SceneNode. Antes de esto, se comprueba si la variable mToggle es menor que 0:

```
If ((mToggle < 0.0f) && mKeyboard->isKeyDown(OIS::KC_1))
{
```

Código 77: Condición de control de tecla.

Ahora se establece la variable mToggle de forma que se pase un Segundo hasta que se pueda realizar la siguiente acción:

```
mToggle = 0.5f;
```

Código 78: Establecimiento del valor de la variable de control.

A continuación, hay que borrar la cámara de donde esté enlazada, establecer mCamNode a "CamNode1", y enlazar la cámara.

```
mCamera->getParentSceneNode()->detachObject(mCamera);
mCamNode = mSceneMgr->getSceneNode("CamNode1");
mCamNode->attachObject(mCamera);
}
```

Código 79: Cambio de posición de la cámara.

Hay que hacer esto para la CamNode2 cuando se pulse el botón 2. El código es idéntico excepto por cambiar 1 por 2, y usar un else if en lugar de if (porque no pueden realizar ambos al mismo tiempo)

```
Else if ((mToggle < 0.0f) && mKeyboard->isKeyDown(OIS::KC_2))
{
        mToggle = 0.5f;
        mCamera->getParentSceneNode()->detachObject(mCamera);
        mCamNode = mSceneMgr->getSceneNode("CamNode2");
        mCamNode->attachObject(mCamera);
}
```

Código 80: Cambio de posición de cámara con el segundo botón.

Compilar y ejecutar el tutorial. Ahora se puede intercambiar el punto de vista pulsando 1 y 2.

La siguiente cosa que se necesita es trasladar mCamNode siempre que el usuario pulse las teclas de flechas o las teclas WASD. Al contrario que en los casos anteriores, no se necesita comprobar el estado de las mismas, ya que en cada frame que se mantenga pulsada la tecla, se tiene que trasladar la cámara. Esto hace el código bastante simple. Primero se crea un Vector3 para guardar donde se quiere trasladar:

```
Vector3 transVector = Vector3::ZERO;
```

Código 81: Declaración de vector de translación de cámara.

A continuación, cuando se pulsa la tecla la tecla W o la flecha arriba, hay que mover hacia delante (es el eje Z negativo):

```
If (mKeyboard->isKeyDown(OIS::KC_UP) || mKeyboard-
>isKeyDown(OIS::KC_W)
        trasnVector.z -= mMove;
```

Código 82: Establecimiento de pulsación W o fleha arriba.

Hay que hacer lo mismo para el resto de teclas. Además para mover en el eje y, se usan las teclas E,Q o PageUp, pageDown:

```
if (mKeyboard->isKeyDown(OIS::KC_DOWN) || mKeyboard-
>isKeyDown(OIS::KC_S))
transVector.z += mMove;
if (mKeyboard->isKeyDown(OIS::KC_LEFT) || mKeyboard-
>isKeyDown(OIS::KC_A))
transVector.x -= mMove;
if (mKeyboard->isKeyDown(OIS::KC_RIGHT) || mKeyboard-
>isKeyDown(OIS::KC_D))
transVector.x += mMove;
if (mKeyboard->isKeyDown(OIS::KC_PGUP) || mKeyboard-
>isKeyDown(OIS::KC_Q))
transVector.y += mMove;
if (mKeyboard->isKeyDown(OIS::KC_PGDOWN) || mKeyboar-
>isKeyDown(OIS::KC_E))
transVector.y -= mMove;
```

Código 83: Establecimiento de translación según teclas pulsadas.

Ahora la variable transVector tiene la translación que hay que aplicar a la cámara del SceneNode.

El primer obstáculo que se encuentra cuando se hace esto es que si se rota el SceneNode, las coordenadas x, y, z serán erróneas cuando se hace la translación. Para solucionar esto, hay que aplicar todas las rotaciones realizadas al SceneNode al nodo de translación. Esto es realmente más simple de lo que parece.

Para representar rotaciones, Ogre no utiliza matrices de transformación como algunos motores gráficos. En lugar de ello utiliza Cuaterniones para todas las operaciones de rotación. La matemática detrás de los Cuaterniones implica álgebra linear en cuatro dimensiones, que es bastante difícil de comprender. Pero no es necesario conocerlo. Es muy simple utilizar un cuaternión par rotar un vector. Se puede obtener un cuaternión que representa esas rotaciones llamando a SceneNode::getOrientation(), con ello se puede aplicar a la translación utilizando multiplicación.

El segundo obstáculo es que hay que ver si hay que escalar la cantidad a trasladar por la cantidad de tiempo pasado desde el último frame. Por otro lado, lo rápido que se mueve dependerá del framerate de la aplicación. Esta es la función que se necesita realizar para trasladar el nodo de cámara sin encontrar estos problemas:

```
mCamNode->translate(transVector * evt.timeSinceLastFrame,
Node::TS_LOCAL);
```

Código 84: Traslación de la cámara.

Se ha introducido algo nuevo. Siempre que se traslada un nodo, o se rota sobre un eje, se puede especificar el Espacio de Transformación que se quiere utilizar para mover el objeto. Normalmente cuando se traslada un objeto, no hay que establecer este parámetro. Por defecto es TS_PARENT, que indica que el objeto se mueve en cualquier espacio de transformación en el que esté el nodo padre. En este caso, el nodo padre es el nodo principal de la escena (root). Cuando se pulsa W (o flecha para

arriba), se resta de la dirección Z, indicando que se mueve a través del eje Z negativo. Si no se especifica TS_LOCAL en el código anterior, se tendría que mover la cámara a través del eje Z global. Sin embargo como se intenta que la cámara vaya hacia delante cuando se pulsa esta tecla en la dirección que esté orientada, hay que usar el espacio de transformación "local".

Hay otra forma de hacer esto (aunque es menos directo). Se podría tener la orientación del nodo, un cuaternión, y multiplicarlo por el vector de dirección para obtener el mismo resultado. Esto sería perfectamente válido:

```
//Do not add this to the program
mCamNode->translate(mCamNode->getOtientation() * transVector *
evt.timeSinceLastFrame, Node::TS_WORLD);
```

Código 85: Traslación de la cámara con referencia global.

Esto también traslada el nodo cámara en el espacio local. En este caso, no hay razón real para hacer esto. Ogre define tres espacios de trasformación: TS_LOCAL, TS_PARENT, y TS_WORLD. Puede haber alguna situación en la que se necesite rotar en otro espacio que estos tres. En ese caso, se podría hacer similar que la forma del código anterior. Tomar un cuaternión representando el espacio de vector (o la orientación de cualquier objeto), multiplicarlo por el vector de traslación para obtener el vector de traslación correcto, y entonces moverlo en el espacio TS_WORD.

Ahora que ya se tiene el movimiento por teclas, hay que hacer que el ratón afecta en la dirección en la que se está mirando, pero sólo si el usuario tiene pulsado el botón derecho del ratón. Para hacer esto, primero hay que comprobar si el botón derecho está pulsado:

```
If (mMouse->getMouseState().buttonDown(OIS::MB_Right))
{
```

Código 86: Botón derecho del ratón pulsado.

Si es así, se vira y cabecea la cámara basada en lo que se ha movido el ratón desde el último frame. Para hacer esto, se toman los cambios relativos en X e Y, y se cambia en llamadas a funciones de viraje (yaw) y cabeceo (pitch):

```
        mCamNode->yaw(Degree(-mRotate * mMouse->getMouseState().X.rel),
Node::TS_WORLD);
        mCamNode->pitch(Degree(-mRotate * mMouse-
>getMouseState().Y.rel), Node::TS_LOCAL);
}
```

Código 87: Rotación de la cámara.

Hay que tener en cuenta que se utiliza TS_WORLD para los virajes (por defecto se usa TS_LOCAL en las funciones de rotación si no se especifica). Hay que intentar asegurarse de que el cabeceo del objeto no afecta al viraje. Siempre se quiere que el viraje rote sobre el mismo eje. Este es el tercer obstáculo: olvidar las iteraciones entre rotaciones.

Este tutorial no pretende ser una guía completa sobre las rotaciones y cuaterniones (que es material suficiente para llenar un tutorial completo por sí mismo). En el siguiente tutorial, se utiliza un búfer de entrada de ratón en vez de la comprobación de las claves que se establece en cada fotograma.

TUTORIAL BÁSICO 5: ENTRADA CON BÚFER

1. INTRODUCCIÓN

Este es un pequeño tutorial para aprender a usar la entrada con búfer de OIS (Object Oriented Input System), opuesto al tutorial anterior. Este tutorial difiere del anterior en que los eventos de teclado y ratón se manejarán de forma inmediata, como ocurren, en lugar de una vez por fotograma. Hay que tener en cuenta que esto es sólo una introducción a la entrada con búfer, y no un tutorial completo de cómo utilizar OIS. Para más información se puede consultar un artículo sobre el uso de OIS (http://www.ogre3d.org/wiki/index.php/Using_OIS).

2. INICIO

Este tutorial se construirá encima del tutorial anterior, pero se cambiará la forma de realizar la entrada. Como la funcionalidad será básicamente la misma, se utiliza la misma clase TutorialApplication del último tutorial, pero se empezará con el TutorialFrameListener. Se crea un proyecto y se añade un archivo de recurso con el siguiente código:

```
#include "ExampleApplication.h"

class TutorialFrameListener : public ExampleFrameListener, public
OIS::MouseListener, public OIS::KeyListener
{
public:
    TutorialFrameListener(RenderWindow* win, Camera* cam, SceneManager
*sceneMgr)
        : ExampleFrameListener(win, cam, true, true)
{
    }

bool frameStarted(const FrameEvent &evt)
{
if(mMouse)
            mMouse->capture();
        if(mKeyboard)
mKeyboard->capture();
        return mContinue;
    }

    // MouseListener
bool mouseMoved(const OIS::MouseEvent &e) { return true; }
    bool mousePressed(const OIS::MouseEvent &e, OIS::MouseButtonID id)
{ return true; }
    bool mouseReleased(const OIS::MouseEvent &e, OIS::MouseButtonID
id) { return true; }

    // KeyListener
    bool keyPressed(const OIS::KeyEvent &e) { return true; }
    bool keyReleased(const OIS::KeyEvent &e) { return true; }
protected:
    Real mRotate;          // The rotate constant
    Real mMove;            // The movement constant

    SceneManager *mSceneMgr;   // The current SceneManager
    SceneNode *mCamNode;   // The SceneNode the camera is currently
attached to
```

```
    bool mContinue;        // Whether to continue rendering or not
    Vector3 mDirection;     // Value to move in the correct direction
};

class TutorialApplication : public ExampleApplication
{
public:
    void createCamera(void)
    {
        // create camera, but leave at default position
        mCamera = mSceneMgr->createCamera("PlayerCam");
        mCamera->setNearClipDistance(5);
    }

    void createScene(void)
    {
        mSceneMgr->setAmbientLight(ColourValue(0.25, 0.25, 0.25));

        // add the ninja
        Entity *ent = mSceneMgr->createEntity("Ninja", "ninja.mesh");
        SceneNode *node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("NinjaNode");
        node->attachObject(ent);

        // create the light
        Light *light = mSceneMgr->createLight("Light1");
        light->setType(Light::LT_POINT);
        light->setPosition(Vector3(250, 150, 250));
        light->setDiffuseColour(ColourValue::White);
        light->setSpecularColour(ColourValue::White);

        // Create the scene node
        node = mSceneMgr->getRootSceneNode()-
>createChildSceneNode("CamNode1", Vector3(-400, 200, 400));

        // Make it look towards the ninja
        node->yaw(Degree(-45));

        // Create the pitch node
        node = node->createChildSceneNode("PitchNode1");
        node->attachObject(mCamera);

        // create the second camera node/pitch node
node = mSceneMgr->getRootSceneNode()->createChildSceneNode("CamNode2",
Vector3(0, 200, 400));
        node = node->createChildSceneNode("PitchNode2");
}

    void createFrameListener(void)
    {
        // Create the FrameListener
        mFrameListener = new TutorialFrameListener(mWindow, mCamera,
mSceneMgr);
        mRoot->addFrameListener(mFrameListener);

        // Show the frame stats overlay
        mFrameListener->showDebugOverlay(true);
    }
};
```

```
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    TutorialApplication app;

    try {
        app.go();
    } catch(Exception& e) {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
        MessageBox(NULL, e.getFullDescription().c_str(), "An exception
has occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
            e.getFullDescription().c_str());
#endif
    }

return 0;
}
```

Código 88: Código de partica del tutorial 5.

Si se está usando el OgreSDK bajo Windows, hay que asegurarse de añadir el directorio "[Directorio_OgreSDK]\sample\include" en el proyecto (el archivoExampleApplication.h se encuentra allí). Si se utiliza la distribución de código fuente deOgre, dicho directorio debe estar situado en el directorio "[Directorio_OgreSource]\samples\common\include". Hay que asegurarse de poder compilar el código antes de continuar a la siguiente sección **(aunque no hay que intentar ejecutarlo, ya que se bloqueará hasta que se añada más código)**. Como siempre, si ocurren problemas, se pueden consultar los foros de ayuda (http://www.ogre3d.org/forums/).

3. ENTRADA DE BÚFER EN DOS PALABRAS

3.1 INTRODUCCIÓN

En el anterior tutorial se usó entrada sin búfer, esto es que, en cada frame, se pregunta por el estado de OIS::Keyboard y OIS::Mouse para ver qué teclas o botones están pulsados. La entrada por búfer, utiliza interfaces escuchadores para informar al programa de qué eventos han ocurrido. Por ejemplo, cuando una tecla está pulsada, se lanza el evento KeyListener::keyPressed y cuando se suelta el botón, se lanza el evento KeyListener::keyReleased en todas las clases KeyListener registradas. Esto evita tener que comprobar y guardar los estados de las teclas.

OIS también admite eventos sin búfer de Joystick a través del interfaz OIS::JoystickListener, aunque no se cubre en este tutorial.

Una cosa importante sobre el sistema de escuchadores de OIS es que sólo se puede tener un escuchador por cada objeto Keyboard, Mouse o Joystick. Esto es así por simplicidad (y por velocidad). Llamando a la función setEventCallback (que se verá más adelante) múltiples veces, sólo obtendrá eventos el último escuchador registrado. También hay que estar seguro de llamar a Keyboard::capture y Mouse::capture

en el método frameStarted. OIS no utiliza thread para determinar los estados del teclado y ratón, por lo que hay que especificar cuándo deberían de capturar la entrada.

3.2 EL INTERFAZ KEYLISTENER

El interfaz KeyListener proporciona dos funciones virtuales puras. La primera es la función keyPressed (se llama cada vez que se pulsa una tecla) y la segunda es keyReleases (llamada cada vez que se suelta una tecla). El parámetro que se pasa es un KeyEvent, que contiene el código de tecla que se está pulsando/soltando.

3.3 EL INTERFAZ MOUSELISTENER

El interfaz MouseListener es un poco más complicado que el KeyListener. Contiene funciones para ver cuándo se ha pulsado o soltado un botón del ratón. También contiene una función mouseMoved, que se llama cuando se mueve el ratón. Cada una de estas funciones recibe un objeto MouseEvent, que contiene el estado actual del ratón en la variable "state".

La cosa más importante sobre el objeto MouseState es que no sólo contiene las coordenadas relativas X e Y del movimiento del ratón (que es, cuánto se ha movido desde la última vez que se llamó a MouseListener::mouseMoved), sino que también contiene las coordenadas absolutas X e Y (que es, dónde están exactamente en la pantalla).

4. EL CÓDIGO

4.1 EL CONSTRUCTOR TUTORIALFRAMELISTENER

Antes de comenzar a modificar el TutorialFrameListener, hay que hacer dos cambios importantes en la clase TutorialFrameListener. El primero es que se implementarán más interfaces en su interior:

```
Class TutorialFrameListener : public ExampleFrameListener, puclic
OIS::MouseListener, public OIS::KeyListener
```

Código 89: Interfaces implementadas en TutorialFrameListener.

Se hace subclase de OIS MouseListener y KeyListener, de forma que se puedan recibir eventos de ellos. Hay que tener en cuenta que OIS MouseListener maneja tanto eventos de botones como de movimiento del ratón.

También hay que cambiar el constructor del ExampleFrameListener:

```
        : ExampleFrameListener (win, cam, true, true)
```

Código 90: Cambio en el constructor TutorialFrameListener.

Los parámetros "true, true" especifican que se utilizará entrada con búfer para el teclado y el ratón.

4.2 VARIABLES

Algunas variables han cambiado con respecto al otro tutorial. Se borran mToggle y mMouseDown y se añaden las siguientes:

```
Real mRotate;          // The rotate constant
Real mMove;            // The movement constant
```

```
    SceneManager *mSceneMgr;   // The current SceneManager
    SceneNode *mCamNode;   // The SceneNode the camera is currently
attached to

    bool mContinue;        // Whether to continue rendering or not
    Vector3 mDirection;     // Value to move in the correct direction
```

Código 91: Variables declaradas dentro de la clase TutorialFrameListener.

mRotate, mMove, mSceneMgr y mCamNode son los mismos que en tutorial anterior (aunque se cambiará el valor de mRotate ya que se usa de forma distinta ahora). La variable mContinue se obtiene del método frameStarted. Cuando se establece mContinue a false, se saldrá del programa. La variable mDirection contiene información de cómo hay que trasladar el nodo de cámara en cada fotograma.

4.3 CONSTRUCTOR TUTORIALFRAMELISTENER

En el constructor se inicializarán algunas variables como se hizo en el tutorial anterior, y se establecerá el mContinue a true. Hay que añadir el siguiente código al constructor de TutorialFrameListener

```
        // Populate the camera and scene manager containers
mCamNode = cam->getParentSceneNode();
        mSceneMgr = sceneMgr;

        // set the rotation and move speed
mRotate = 0.13;
        mMove = 250;

        // continue rendering
        mContinue = true;
```

Código 92: Inicialización de variables.

Los objetos OIS mMouse y mKeyboard se han obtenido ya en el constructor ExampleFrameListener. Se puede registrar el TutorialFrameListener como el escuchador llamando a setEventCallback en esos objetos de entrada como sigue:

```
        mMouse->setEventCallback(this);
        mKeyboard->setEventCallback(this);
```

Código 93: Captura de eventos de ratón y teclado.

Finalmente, se necesita inicializar mDirection a cero:

```
        mDirection = Vector3::ZERO;
```

Código 94: Inicialización de la variable de dirección.

4.4 FIJACIONES DE TECLA

Antes de ir más lejos, hay que fijar la tecla de escape para salir del programa. Se busca el método TutorialFrameListener::keyPressed. Este método se llama con un objeto KeyEvent cada vez que se pulsa una tecla. Se puede obtener el código de la tecla pulsada (KC_*) consultando la variable “key” del objeto. Hay que construir un switch para diferenciar cuando se pulsa la tecla escape o cualquier otra, de forma similar a esta:

```
bool keyPressed(const OIS::KeyEvent &e)
{
        switch (e.key)
        {
case OIS::KC_ESCAPE:
```

```
            mContinue = false;
            break;
        default:
            break;
}
        return mContinue;
    }
```

Código 95: Código inicial de keyPressed.

Hay que estar seguro de que ahora se puede compilar y ejecutar la aplicación antes de continuar.

Se necesitan añadir fijaciones para las otras teclas en el switch. Primero se añade el control para intercambiar el punto de vista pulsando 1 y 2. El código (incluido en el switch) es similar al de la práctica anterior pero sin la variable mToggle

```
        case OIS::KC_1:
            mCamera->getParentSceneNode()->detachObject(mCamera);
            mCamNode = mSceneMgr->getSceneNode("CamNode1");
mCamNode->attachObject(mCamera);
            break;

        case OIS::KC_2:
            mCamera->getParentSceneNode()->detachObject(mCamera);
            mCamNode = mSceneMgr->getSceneNode("CamNode2");
mCamNode->attachObject(mCamera);
break;
```

Código 96: Capturas de tecla y cambio de posición de cámara.

Como se puede ver, es mucho más claro que hacerlo con una variable temporal.

La siguiente cosa que hay que hacer es añadir el movimiento del teclado. Cada vez que el usuario pulsa una tecla que produce movimiento, se añade o resta a mMove (dependiendo de la dirección):

```
        case OIS::KC_UP:
        case OIS::KC_W:
mDirection.z = -mMove;
break;

        case OIS::KC_DOWN:
        case OIS::KC_S:
            mDirection.z = mMove;
            break;

        case OIS::KC_LEFT:
        case OIS::KC_A:
            mDirection.x = -mMove;
            break;
        case OIS::KC_RIGHT:
        case OIS::KC_D:
            mDirection.x = mMove;
            break;

        case OIS::KC_PGDOWN:
        case OIS::KC_E:
            mDirection.y = -mMove;
            break;

        case OIS::KC_PGUP:
        case OIS::KC_Q:
            mDirection.y = mMove;
```

```
break;
```

Código 97: Movimiento de la cámara.

Ahora se necesita "deshacer" el cambio al vector mDirection cuando la tecla se suelta para parar el movimiento. En el código de keyReleased se añade este código:

```
        switch (e.key)
{
case OIS::KC_UP:
        case OIS::KC_W:
        case OIS::KC_DOWN:
        case OIS::KC_S:
mDirection.z = 0;
break;

        case OIS::KC_LEFT:
        case OIS::KC_A:
        case OIS::KC_RIGHT:
        case OIS::KC_D:
            mDirection.x = 0;
            break;

        case OIS::KC_PGDOWN:
        case OIS::KC_E:
        case OIS::KC_PGUP:
        case OIS::KC_Q:
            mDirection.y = 0;
break;

        default:
            break;
        } // switch
        return true;
```

Código 98: Control del soltado de teclas.

Ahora que está mDirection actualizado, hay que hacer la translación realmente. Este código es el mismo que en el código anterior, hay que añadirlo a la función frameStarted:

```
mCamNode->translate(mDirection * evt.timeSinceLastFrame,
Node::TS_LOCAL);
```

Código 99: Traslado de la cámara.

Se compila y ejecuta la aplicación. Ahora se obtiene un movimiento basado en el teclado usando entrada por búfer.

4.5 FIJACIONES DE RATÓN

Ahora que se ha completado el teclado, se trabaja en el movimiento del ratón. Se empieza con el intercambio de luz apagada/encendida según las pulsaciones del botón izquierdo. En la función mousePressed y se observan los parámetros. Con OIS, se tiene acceso a MouseEvent y a MouseButtonID. Se puede comprobar MouseButtonID para ver qué botón está pulsado. Se reemplaza el código de mousePressed por este:

```
        Light *light = mSceneMgr->getLight("Light1");
        switch (id)
        {
case OIS::MB_Left:
            light->setVisible(! light->isVisible());
```

```
break;
default:
            break;
        }
        return true;
```

Código 100: Control de pulsación de ratón.

Con esto ya está funcionando, ahora la única cosa que queda por hacer es fijar el botón derecho para realizar el modo de vista. Cada vez que el ratón se mueve, se chequea para comprobar si el botón derecho está pulsado. Si lo está, se rota la cámara basándose en el movimiento relativo. Se puede acceder al movimiento relativo del ratón con el objeto MouseEvent pasado a esta función. Contiene una variable llamada "state" que contiene MouseState (que tiene básicamente información detallada del ratón). MouseState::buttonDown indica si un botón en particular está pulsado o no, y las variables "X" e "Y" dan el movimiento relativo del ratón. En la función mouseMoved se reemplaza el código de su interior por este:

```
        if (e.state.buttonDown(OIS::MB_Right))
{
mCamNode->yaw(Degree(-mRotate * e.state.X.rel), Node::TS_WORLD);
            mCamNode->pitch(Degree(-mRotate * e.state.Y.rel),
Node::TS_LOCAL);
}
        return true;
```

Código 101: Control de rotación de cámara.

Se compila y se ejecuta la aplicación y la cámara actuará en modo visión cuando se pulsa el botón derecho del ratón.

5. OTROS SISTEMAS DE ENTRADA

OIS es generalmente bueno, y sirve para la mayoría de los propósitos de una aplicación. Aunque hay algunas alternativas si se quiere usar algo diferente. Algunos sistemas de ventanas puede ser lo que se busca, como wxWidgets (http://www.wxwindows.org/), que alguna gente a integrado en Ogre. Se puede utilizar el sistema de mensajes estándar de Windows o uno de los muchos toolkits GUI de entrada para Linux.

Se puede probar SDL, que no sólo proporciona sistemas de entrada/ventanas, sino que también entrada de gamepad/joystick. Para comenzar con el sistema de joystick, hay que envolver la aplicación con las llamadas a SDL_Init y SDL_Quit:

```
SDL_Init(SDL_INIT_JOYSTICK | SDL_INIT_NOPARACHUTE);
       SDL_JoystickEventState(SDL_ENABLE);

       app.go();

SDL_Quit();
```

Código 102: Control de entrada de Joystick con SDL.

Para configurar el Joystick, se llama a SDL_JoystickOpen con el número de Joystick (se pueden especificar varios llamando con 0, 1, 2, ...):

```
SDL_Joystick* mJoystick;
mJoystick = SDL_JoystickOpen(0);

if ( mJoystick == NULL )
```

```
        ; // error handling
```

Código 103: Configuración de Joystick con SDL.

Si la llamada devuelve NULL, es que hay un problema abriendo el joystick. Esto casi siempre significa que el joystick no existe. Se puede usar SDL_NumJoystick para ver cuántos joysticks están añadidos al sistema. También se necesita cerrar el Joystick con:

```
SDL_JoystickClose(mJoystick);
```

Código 104: Cancelado de Joystick.

Para usar el Joystick, se llama a SDL_JoystckGetButton y SDL_JoystickGetAxis. Este sería el código de movimiento para un controlador de Playstation2, que tiene 4 ejes y 12 botones:

```
SDL_JoystickUpdate();

        mTrans.z += evt.timeSinceLastFrame * mMoveAmount *
SDL_JoystickGetAxis(mJoystick, 1) / 32767;
        mTrans.x += evt.timeSinceLastFrame * mMoveAmount *
SDL_JoystickGetAxis(mJoystick, 0) / 32767;

        xRot -= evt.timeSinceLastFrame * mRotAmount *
SDL_JoystickGetAxis(mJoystick, 3) / 32767;
        yRot -= evt.timeSinceLastFrame * mRotAmount *
SDL_JoystickGetAxis(mJoystick, 2) / 32767;
```

Código 105: Control de Joystick con SDL.

mTrans se rellena después en el método de la cámara SceneNode::translate, xRot en SceneNode::yaw, y yRot en SceneNode::pitch. Hay que tener en cuenta que SDL_JoystickGetAxis retorna un valor entre -32767 y 32767, pero se escala entre -1 y 1. Se debería de hacer esto cuando se empieza a usar SDL. Si se está buscando más información sobre esta área, la mejor documentación se puede encontrar aquí: http://www.jikos.cz/~crag/private/dox/SDL__joystick_8h-source.html

TUTORIAL BÁSICO 6: SECUENCIA DE INICIO DE OGRE

1. INTRODUCCIÓN

Este tutorial enseña cómo empezar con Ogre sin utilizar el framework de ejemplo. Cuando se realice este tutorial, se podrán crear aplicaciones sin tener que utilizar ExampleApplication o ExampleFrameListener.

2. COMENZANDO

2.1 CÓDIGO INICIAL

En este tutorial, se utiliza un código base predefinido como punto de partida. Si se han realizado los otros tutoriales, el código resultará familiar. Hay que crear un proyecto y añadir un archivo de fuente con estos datos:

```
#include <Ogre.h>
#include <OIS/OIS.h>
#include <CEGUI/CEGUI.h>
#include <CEGUI/RendererModules/Ogre/CEGUIOgreRenderer.h>

using namespace Ogre;

class ExitListener : public FrameListener
{
public:
    ExitListener(OIS::Keyboard *keyboard)
        : mKeyboard(keyboard)
    {
    }

    bool frameStarted(const FrameEvent& evt)
    {
        mKeyboard->capture();
        return !mKeyboard->isKeyDown(OIS::KC_ESCAPE);
    }

private:
    OIS::Keyboard *mKeyboard;
};

class Application
{
public:
    void go()
    {
        createRoot();
        defineResources();
        setupRenderSystem();
        createRenderWindow();
        initializeResourceGroups();
        setupScene();
        setupInputSystem();
        setupCEGUI();
        createFrameListener();
        startRenderLoop();
    }
```

```
    ~Application()
    {
    }

private:
    Root *mRoot;
    OIS::Keyboard *mKeyboard;
    OIS::InputManager *mInputManager;
    CEGUI::OgreRenderer *mRenderer;
    ExitListener *mListener;

    void createRoot()
    {
    }

    void defineResources()
    {
    }

    void setupRenderSystem()
    {
    }

    void createRenderWindow()
    {
    }

    void initializeResourceGroups()
    {
    }

    void setupScene()
    {
    }

    void setupInputSystem()
    {
    }

    void setupCEGUI()
    {
    }

    void createFrameListener()
    {
    }

    void startRenderLoop()
    {
    }
};

#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
```

```
#endif
{
    try
    {
        Application app;
        app.go();
    }
    catch(Exception& e)
    {
#if OGRE_PLATFORM == PLATFORM_WIN32 || OGRE_PLATFORM ==
OGRE_PLATFORM_WIN32
        MessageBoxA(NULL, e.getFullDescription().c_str(), "An
exception has occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
        fprintf(stderr, "An exception has occurred: %s\n",
            e.getFullDescription().c_str());
#endif
    }

return 0;
}
```

Código 106: Código de partida del tutorial 6.

Hay que estar seguro de que este código compila antes de continuar. Si se obtienen errores de dependencias perdidas con CEGUI como CEGUISingleton.h o problemas en linkers, hay que estar seguro de que las rutas están en el include: $(OGRE_HOME)\include, $(OGRE_HOME)\include\CEGUI, y $(OGRE_HOME)\samples\include. Se necesitan también añadir CEGUIBase_d.lib y OgreGUIRenderer_d.lib a las dependencias de enlaces.

2.2 PROCESO DE INICIO EN POCAS PALABRAS

Una vez que se comprenda lo que se va a hacer bajo la cubierta, conseguir hacer funcionar a Ogre es realmente muy sencillo de realizar. El ejemplo de framework parece desalentador al principio, ya que intenta muchas cosas que pueden ser o no necesarias para las aplicaciones. Después de realizar este tutorial, se podrán seleccionar las que se necesiten para una aplicación concreta. Antes de sumergirse en todo esto, se dará un vistazo rápido de cómo funciona el proceso de iniciado en alto nivel.

El ciclo básico de vida de Ogre se parece a esto:

1. Crear el objeto Root.
2. Definir los recursos que Ogre utilizará.
3. Elegir y establecer el RenderSystem (DirectX, OpenGL, etc).
4. Crear el RenderWindow (la ventana donde reside Ogre).
5. Inicializar los recursos que se van a utilizar.
6. Crear una escena usando esos recursos.
7. Establecer librerías y plugins de terceros.
8. Crear cualquier número de escuchadores de fotograma.
9. Comenzar con el bucle de renderizado.

A continuación se verá cada uno de estos apartados en profundidad.

Se recomienda hacer los pasos 1-4 en orden. Los pasos 5-6 se pueden realizar bastante más tarde en la aplicación. Se podrían hacer 7-8 antes que los pasos 5-6 si se desea, pero nunca hacerlos antes del paso 4.

3. LEVANTANDO OGRE

3.1 CREANDO EL OBJETO ROOT

La primera cosa a realizar es la más simple. El objeto Root es el núcleo de la librería Ogre, y se debe crear antes de realizar cualquier otra cosa en el motor. Hay que buscar en la aplicación la función Application::createRoot y añadir el siguiente código:

```
mRoot = new Root();
```

Código 107: Creación del objeto Root.

Esto es todo lo que se necesita hacer. El constructor de Root tiene 3 parámetros. El primero es el nombre y localización del fichero de configuración de plugins. El segundo es la localización del fichero de configuración de Ogre (que dice a Ogre cosas como la tarjeta de video, configuraciones visuales, etc.). El último parámetro es la localización y nombre del fichero de log en el que escribe Ogre. Como no se va a cambiar nada, se puede dejar con los valores por defecto.

3.2 RECURSOS

Nota: Esta sección tendría más sentido si se abriera “recources.cfg” y darle un vistazo antes de continuar.

La siguiente tarea a realizar es definir los recursos que la aplicación usa. Esto incluye texturas, modelos, scripts, y demás. Hay que definir todos los recursos que la aplicación puede que use.

Para hacer esto, hay que añadir cada carpeta donde residan recursos en ResourceGroupManager. Hay que buscar la función defineResources y añadir el siguiente código:

```
        String secName, typeName, archName;
        ConfigFile cg;
        Cf.load(“resources.cfg”);
```

Código 108: Cargado del fichero de recursos.

Esto utiliza la clase de Ogre ConfigFile para parsear todos los recursos desde “resources.cfg”, pero no los cargará en Ogre (hay que hacerlo manualmente). Hay que recordar que en una aplicación propia se pueden utilizar parseadores y archivos de configuración propios, sólo hay que reemplazar ConfigFile por el propio. El método de carga de recursos no importa cómo se haga, siempre que se añaden los recursos al ResourceGroupManager. Ahora que se ha parseado el fichero de configuración, se necesita añadir secciones al ResourceGroupManager. El siguiente código comienza un bucle a través del archivo de configuración:

```
        ConfigFile::SectionIterator seci = cf.getSectionIterator();
        while (seci.hasMoreElements())
        {
```

Código 109: Bucle a través de los elementos de configuración.

Por cada sección, se obtiene el contenido:

```
        secName = seci.peekNextKey();
        ConfigFile::SettingsMultiMap *settings = seci.getNext();
        ConfigFile::SettingsMultiMap::iterator I;
```

Código 110: Apertura por fichero de configuración.

Finalmente, se añade el nombre de sección (que está en el grupo de recursos), el tipo de recursos (zip, carpeta, etc), y nombre del fichero del recurso al ResourceGroupManager:

```
        For (i = settings->begin(); i ¡= settings->end(): ++i)
        {
                typeName = i->first;
                archName = i->second;

        ResourceGroupManager::getSingleton().addResourceLocation(archNa
me, typeName, secName);
        }
}
```

Código 111: Adición de recursos al sistema.

Esta función completa añade los recursos desde el archivo de configuración, pero sólo dice a Ogre donde están. Antes de que se pueda utilizar cada uno de ellos, hay que inicializar el grupo que se desee utilizar, o se debe de inicializar todos. Se verá esto más adelante en la función “inicializando Recursos”.

3.3 CREANDO EL RENDERSYSTEM

A continuación hay que elegir el sistema de renderizado (normalmente DirectX o OpenGL en una máquina Windows) y configurarlo. La mayor parte de las aplicaciones de demo de Ogre, utilizan un diálogo de configuración, que es una forma muy razonable de configurar la aplicación. Ogre también ofrece una forma para restaurar la configuración que un usuario establece, de forma que no se tiene que volver a configurar después de la primera vez. Hay que buscar la función setupRenderSystem y añadir el siguiente código:

```
if (!mRoot->restoreConfig() && !mRoot->showConfigDialog())
            throw Exception(52, "User canceled the config dialog!",
"Application::setupRenderSystem()");
```

Código 112: Muestra el diálogo de configuración de pantalla.

En la primera parte de este establecimiento, se intenta recuperar el archivo de configuración. Si la función retorna falso, indica que el fichero no existe, por lo que hay que mostrar el dialogo de configuración, que es la segunda porción del if. Si también retorna falso, indica que el usuario a cancelado el diálogo de configuración (indicando que quiere salir del programa). En este ejemplo, se lanza una excepción, pero en la práctica, lo más probable es simplemente retornar falso y cerrar la aplicación, la razón por la que se captura una excepción también es porque restoreConfig o showConfigDialog pueden lanzar una excepción y lo mejor es capturar todas las que ocurran. Este cambio (salir de la aplicación) sería un poco difícil, por lo que se captura una excepción en este tutorial.

Si se utiliza esta técnica en la práctica, se recomienda que si se recoge una excepción durante la inicialización de Ogre, se borre el fichero ogre.cfg en el catch de la excepción. Es posible que las opciones elegidas en el dialogo de configuración causen el problema y se necesite cambiarlas. Incluso si no se utiliza esto en una aplicación para distribuir, apagar el diálogo puede ayudar en tiempo de desarrollo, ya que no es necesario establecer las configuraciones gráficas cada vez que se inicializa la aplicación.

La aplicación puede configurar manualmente el RenderSystem, si se pretende utilizar algo distinto al dialogo de configuración de Ogre. Un ejemplo básico podría ser el siguiente (no añadir este código a la aplicación):

```
// Do not add this to the application
        RenderSystem *rs = mRoot->getRenderSystemByName("Direct3D9
Rendering  Subsystem");
        // or use "OpenGL Rendering Subsystem"
        mRoot->setRenderSystem(rs);
        rs->setConfigOption("Full Screen", "No");
        rs->setConfigOption("Video Mode", "800 x 600 @ 32-bit
colour");
```

Código 113: Configuración de Ogre sin diálogo.

Se puede utilizar Root::getAvailableRenderers para ver qué sistemas de renderizado están disponibles para utilizar con la aplicación. Una vez que se ha recuperado el RenderSystem, se puede utilizar RenderSystem::getConfigOptions para ver qué opciones están disponibles para el usuario. Combinando esas dos funciones, se puede crear un diálogo de configuración propio para la aplicación.

3.4 CREANDO EL RENDERWINDOW

Una vez elegido el sistema de renderizado, se necesita una ventana para renderizar Ogre en su interior. Realmente hay un montón de opciones de cómo hacer esto, pero sólo se verán algunas.

Si se desea que Ogre cree la ventana de renderizado en lugar del desarrollador, es muy fácil de hacer. Hay que buscar la función createRenderWindow y añadir el siguiente código:

```
        mRoot->initialise(true, “Tutorial Render Window”);
```

Código 114: Inicialización automática de la ventana de renderizado.

Esta llamada inicializa el RenderSystem que se configuró en la sección anterior. El primer parámetro indica si Ogre debe crear o no la RenderWindow en lugar del desarrollador. Alternativamente, se puede crear una ventana de renderizado usando el API win32, wxWidgets, o alguno de los que tienen los sistemas Linux o Windows. Un ejemplo rápido de cómo hacer esto en Windows puede ser como sigue (no incorporar al código):

```
// Do not add this to the application
        mRoot->initialise(false);
        HWND hWnd = 0;  // Get the hWnd of the application!
NameValuePairList misc;
misc["externalWindowHandle"] = StringConverter::toString((int)hWnd);
        RenderWindow *win = mRoot->createRenderWindow("Main
RenderWindow", 800, 600, false, &misc);
```

Código 115: Creación de la ventana de renderizado de forma manual.

Hay que tener en cuenta que hay que llamar a Root::initialise, pero el primer parámetro se establece a false. Luego se obtiene el HWND de la ventana en la que se quiere renderizar Ogre. Cómo obtener esto será determinado por el kit de herramientas GUI que se utiliza para crear la ventana (en Linux posiblemente será diferente). Después de realizar esto, se utiliza NameValuePairList para asignar el manejo a “externalWindowHandle”. La función Root::createRenderWindow pude utilizarse para crear la clase RenderWindow de la ventana que se ha creado. Se recomienda consultar el API de esta función para más información.

3.5 INICIALIZANDO RECURSOS

Ahora que se tiene el objeto Root, y los objetos RenderSystem y RenderWindow creados y listos para funcionar, se está listo para comenzar a crear la escena. La única cosa que queda por hacer es inicializar los recursos que se quieren utilizar. En un juego o aplicación muy grande, puede haber cientos o miles de recursos que el juego utilice – todo desde mallas y texturas hasta scripts. Pero en cualquier momento, sólo se utiliza un grupo pequeño de esos recursos. Para guardar requisitos de memoria, se pueden cargar sólo los recursos que la aplicación está utilizando. Esto se realiza dividiendo los recursos en secciones y sólo inicializando los que se van a utilizar. Esto no se cubre en este tutorial, pero se puede ver en el tutorial avanzado sobre recursos (http://www.ogre3d.org/wiki/index.php/Advanced_Tutorial_1).

Antes de inicializar los recursos, se debe de establecer el número por defecto de mipmaps que usan las texturas. Se debe de establecer antes de inicializar los recursos. Hay que buscar la función initializeResourceGroups y añadir el siguiente código:

```
TextureManager::getSingleton().setDefaultNumMipmaps(5);

ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
```

Código 116: Establecimiento del MipMaps de la textura e inicializado de recursos.

La aplicación tiene ahora todos los grupos de recursos inicializados y listos para utilizarse.

3.6 CREANDO UNA ESCENA

Lo siguiente es crear una escena. Esto se ha hecho varias veces en otros tutoriales. No se añade nada nuevo, pero hay que saber que hay tres cosas que deben de hacerse antes de empezar a añadir cosas a la escena: crear el SceneManager, crear la Camera y crear el Viewport. Hay que buscar la función setupScene y dentro del mismo, crear estos objetos.

Se pueden crear tantos SceneManagers y Cameras como se quiera, pero hay que tener cuidado cuando realmente se quiere renderizar algo en la pantalla utilizando una cámara, hay que añadir un Viewport para ello. Esto se hace utilizando la clase RenderWindow que fue creada en la sección anterior. Como no se tiene puntero a este objeto, se accede a través de Root::getAutoCreatedWindow.

Después de estas tres cosas, se puede añadir lo que sea a la escena como se quiera.

4. INICIANDO LIBRERÍAS DE TERCEROS

4.1 OIS

Aunque no es la única opción para la entrada, OIS es la mejor. Aquí se cubre cómo inicializar OIS en la aplicación.

4.2 CONFIGURACIÓN Y ENTRADA SIN BÚFER

OIS utiliza un InputManager general que es un poco complicado de configurar, pero fácil de usar una vez que se ha creado apropiadamente. OIS no está integrado en Ogre, es una librería independiente, lo que significa que se necesitará proporcionar alguna información al inicio para que funcione correctamente. En la práctica realmente sólo se necesita la ventana manejada donde se renderiza Ogre. Ogre realiza esto de forma sencilla. Hay que buscar la función setupInputSystem y añadir el siguiente código:

```
size_t windowHnd = 0;
        std::ostringstream windowHndStr;
        OIS::ParamList pl;
        RenderWindow *win = mRoot->getAutoCreatedWindow();

        win->getCustomAttribute("WINDOW", &windowHnd);
        windowHndStr << windowHnd;
        pl.insert(std::make_pair(std::string("WINDOW"),
windowHndStr.str()));
        mInputManager = OIS::InputManager::createInputSystem(pl);
```

Código 117: Establecimiento del manager de entrada OIS.

Esto establece el InputManager para usarlo, pero realmente para usar OIS para obtener la entrada del ratón, teclado o joystick, hay que crear estos objetos.

```
try
        {
mKeyboard = static_cast<OIS::Keyboard*>(mInputManager-
>createInputObject(OIS::OISKeyboard, false));
            //mMouse = static_cast<OIS::Mouse*>(mInputManager-
>createInputObject(OIS::OISMouse, false));
            //mJoy = static_cast<OIS::JoyStick*>(mInputManager-
>createInputObject(OIS::OISJoyStick, false));
}
catch (const OIS::Exception &e)
{
throw Exception(42, e.eText, "Application::setupInputSystem");
}
```

Código 118: Creación de objetos de entrada de teclado, ratón y Joystick.

En esta aplicación no se utilizarán ni ratón ni joystick, pero se muestra cómo se crean. El segundo parámetro de InputManager::createInputObject es si utilizar o no la entrada con búfer (como se ha mostrado en tutoriales anteriores). Estableciendo el segundo parámetro a false, se crea un objeto de entrada sin búfer, que es lo que se utiliza en este tutorial.

4.3 CONFIGURANDO EL FRAMELISTENER

No importa si se está utilizando entrada con búfer o no, cada fotograma debe de llamar al método de captura de todos los teclados, ratones y joystick en uso. En este tutorial, el código de inicio ya tiene hecho esto. Para entrada sin búfer, esto es todo lo que se necesita hacer. En cada frame se pregunta por el estado del teclado, ratón. Para entrada con búfer, hay que hacer un pequeño trabajo.

Para utilizar entrada con búfer, se necesita añadir una clase para manejar la entrada (o añadir el código del FrameListener que se tiene creado). Las dos cosas a realizar para establecer la entrada con búfer (aparte de pasar el parámetro true a la llamada de createInputObject) es implementar los interfaces de escucha apropiados y registrar la clase creada con el evento callback. Se pueden ver ejemplos en el tutorial anterior. Aquí hay una clase que implementa todo:

```
// Don't add this to the project
class BufferedInputHandler : public OIS::KeyListener, public
OIS::MouseListener, public OIS::JoyStickListener
{
public:
    BufferedInputHandler(OIS::Keyboard *keyboard = 0, OIS::Mouse
*mouse = 0, OIS::JoyStick *joystick = 0)
{
if (keyboard)
```

```
            keyboard->setEventCallback(this);

        if (mouse)
            mouse->setEventCallback(this);

        if (joystick)
            joystick->setEventCallback(this);
}

    // KeyListener
virtual bool keyPressed(const OIS::KeyEvent &arg) { return true; }
    virtual bool keyReleased(const OIS::KeyEvent &arg) { return true;
}

// MouseListener
virtual bool mouseMoved(const OIS::MouseEvent &arg) { return true; }
    virtual bool mousePressed(const OIS::MouseEvent &arg,
OIS::MouseButtonID id) { return true; }
    virtual bool mouseReleased(const OIS::MouseEvent &arg,
OIS::MouseButtonID id) { return true; }

// JoystickListener
virtual bool buttonPressed(const OIS::JoyStickEvent &arg, int button)
{ return true; }
    virtual bool buttonReleased(const OIS::JoyStickEvent &arg, int
button) { return true; }
    virtual bool axisMoved(const OIS::JoyStickEvent &arg, int axis) {
return true; }
};
```

Código 119: Clase escuchadora por búfer.

Se recomienda sólo implementar los listeners que se van a utilizar.

4.4 SOLUCIÓN DE PROBLEMAS EN LA ENTRADA CON BÚFER

Si no se recibe la entrada en búfer como se espera, hay varias cosas que se deben de comprobar antes de hacer cualquier otra cosa:

1. Hacer las llamadas a InputManager::createInputObject con el flag de búfer a true (segundo parámetro).

2. ¿Se llama a setEventCallback en cada objecto de entrada con búfer?

3. ¿Hay otras clases llamando a setEventCallback? (OIS sólo permite una rellamada de eventos).

4.5 CEGUI

CEGUI es una librería GUI muy flexible que viene directamente integrada con Ogre. En esta aplicación no se va a utilizar CEGUI, por lo que se explicará más adelante. CEGUI requiere el RenderWindow y el SceneManager donde renderizarlo. (Nota: Para usar correctamente CEGUI sin errores de compilación, hay que añadir CEGUIBase_d.lib y OgreGUIRenderer_d.lib en Vinculadores->Entrada->Dependencias Adicionales).

Se añade este código en la función setupCEGUI:

```
        mRenderer = &CEGUI::OgreRenderer::bootstrapSystem();
```

Código 120: Inicialización de CEGUI.

Ahora CEGUI está listo para utilizarse.

5 FINALIZANDO EL INICIO Y EL BUCLE DE RENDERIZADO

5.1 FRAME LISTENER

Antes de comenzar el bucle de renderizado y tener la aplicación funcionando, se necesita añadir los listeners de fotogramas que va a utilizar la aplicación. Hay que tener en cuenta que ya se tiene creado un FrameListener simple llamado ExitListener, que espera por la tecla de Escape para finalizar el programa. En un programa, se tendrán muchos más listeners haciendo cosas mucho más complejas. Se recomienda dar un vistazo a ExitListener y comprender lo que se hace en su interior, luego hay que añadir el siguiente código a la función createFrameListener:

```
        mListener = new ExitListener(mKeyboard);
        mRoot->addFrameListener(mListener);
```

Código 121: Creación y acoplado de un Listener al sistema.

5.2 EL BUCLE DE RENDERIZADO

Lo último que se necesita realizar para comenzar a usar Ogre es realizar el bucle de renderizado. Ogre hace esto muy sencillo. Hay que buscar la función startRenderLoop y añadir el siguiente código:

```
        mRoot->startRendering();
```

Código 122: Inicio de renderizado.

Esto renderizará la aplicación hasta que un FrameListener retorne falso. Se puede también sacar un único frame y trabajar entre cada fotograma si se desea. Root::renderOneFrame renderiza un frame y retorna falso si cualquier FrameListener retorna falso:

```
// Do not add this to the application
        while (mRoot->renderOneFrame())
        {
            // Do some things here, like sleep for x milliseconds or
perform other actions.
            // However, make sure you call the display update
function:
WindowEventUtilities::messagePump();
        }
```

Código 123: Renderizado de un frame.

Sin embargo, parece mucho mejor poner un FrameListener que realice algo en concreto en lugar de hacer esto. El único uso que parece bueno para este tipo de patrón es para dormirlo un cierto tiempo de milisegundos para bajar el framerate de forma artificial hasta un número establecido. Esto no se querrá realizar en un FrameListener porque esto se desordena con la variable FrameEvent::timeSinceLastFrame.

5.3 MANEJO OPCIONAL DE EVENTOS DE VENTANA

Si se desea interceptar ciertos Eventos de Ventana, se puede añadir un WindowEventListener en lugar de comprobarlo manualmente en un mensaje saliente como en el ejemplo anterior. Por ejemplo:

```
// Do not add this to the application
        class WinListener : public WindowEventListener
        {
```

```
        public:
            WinListener()
            {
            }

            bool windowClosing(RenderWindow* rw)
            {
                // Disables standard exit methods such as ALT-F4 and
the close button
return false;
            }
        };
```

Código 124: Clase de manejo de eventos de ventana.

Y entonces hay que hacer esto antes de llamar a mRoot->startRendering();

```
// Do not add this to the application
        SceneManager *mgr = mRoot->getSceneManager("Default
SceneManager");
        RenderWindow *win = mRoot->getAutoCreatedWindow();

        WindowEventUtilities::addWindowEventListener(win, new
WinListener());
```

Código 125: Código adicional para el manejo de eventos de ventana.

6 LIMPIEZA

La última tarea a realizar es limpiar todos los objetos que se han creado cuando se finaliza la aplicación. Para hacer esto, simplemente se borran o destruyen todos los objetos en orden inverso a su creación. Se empieza con el OIS, que tiene funciones específicas que se deben de llamar para destruir sus objetos. Hay que buscar la función ~Application y añadir el siguiente código:

```
mInputManager->destroyInputObject(mKeyboard);
        OIS::InputManager::destroyInputSystem(mInputManager);
```

Código 126: Destrucción de objetos de OIS.

CEGUI:

```
        CEGUI::OgreRenderer::destroySystem();
```

Código 127: Destrucción de objetos de CEGUI.

Finalmente hay que borrar los objetos Root y FrameListener. El resto de objetos que se han creado (el SceneManager, RenderWindow, etc) se limpiarán cuando se borre el objeto Root.

```
        delete mListener;
        delete mRoot;
```

Código 128: Destrucción de objetos de Ogre.

Ahora se puede compilar y ejecutar la aplicación, aunque sólo se vea una pantalla en negro.

TUTORIAL BÁSICO 7: CEGUI Y OGRE

1 INTRODUCCIÓN

En este tutorial se explica cómo utilizar CEGUI con Ogre. Al final de este tutorial se deberían conocer las funcionalidades básicas de CEGUI en una aplicación. NOTA: Este tutorial es un pequeño inicio sobre CEGUI, las cuestiones sobre esta herramienta se pueden consultar con detalle en su página oficial: http://www.cegui.org.uk/wiki/index.php/Main_Page

CEGUI no viene incluido en las últimas versiones de Ogre, por lo que para realizar este tutorial, primero hay que instalar CEGUI en el sistema.

2 COMENZANDO

2.1 EL CÓDIGO INICIAL

En este tutorial, se utilizará un código base predefinido como punto de partida. Hay que crear un proyecto y añadir un fichero de código origen con este código:

```
#include "ExampleApplication.h"
#include <OIS/OIS.h>
#include <CEGUI/CEGUI.h>
#include <CEGUI/RendererModules/Ogre/CEGUIOgreRenderer.h>

class TutorialListener : public ExampleFrameListener, public
OIS::MouseListener, public OIS::KeyListener
{
public:
    TutorialListener(RenderWindow* win, Camera* cam)
        : ExampleFrameListener(win, cam, true, true)
{
mContinue=true;
       mMouse->setEventCallback(this);
mKeyboard->setEventCallback(this);
    } // CEGUIDemoListener

bool frameStarted(const FrameEvent &evt)
{
mKeyboard->capture();
        mMouse->capture();

        return mContinue && !mKeyboard->isKeyDown(OIS::KC_ESCAPE);
}

bool quit(const CEGUI::EventArgs &e)
{
        mContinue = false;
        return true;
    }

    // MouseListener
bool mouseMoved(const OIS::MouseEvent &arg)
{
        return true;
    }
```

```
bool mousePressed(const OIS::MouseEvent &arg, OIS::MouseButtonID id)
{
        return true;
    }

bool mouseReleased(const OIS::MouseEvent &arg, OIS::MouseButtonID id)
{
        return true;
    }

    // KeyListener
bool keyPressed(const OIS::KeyEvent &arg)
{
        return true;
    }

bool keyReleased(const OIS::KeyEvent &arg)
    {
        return true;
    }

private:
    bool mContinue;
};


class CEGUIDemoApplication : public ExampleApplication
{
public:
    CEGUIDemoApplication()
    {
    }

    ~CEGUIDemoApplication()
    {
        CEGUI::OgreRenderer::destroySystem();
    }
protected:
    CEGUI::OgreRenderer *mRenderer;

    void createScene(void)
    {
    }

    void createFrameListener(void)
    {
        mFrameListener= new TutorialListener(mWindow, mCamera);
        mFrameListener->showDebugOverlay(true);
        mRoot->addFrameListener(mFrameListener);
    }

};

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"


INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
```

```
#endif
{
    // Create application object
    CEGUIDemoApplication app;

    try {
        app.go();
    } catch(Exception& e) {
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
        MessageBoxA(NULL, e.getFullDescription().c_str(), "An
exception has occurred!",
            MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
            e.getFullDescription().c_str());
#endif
    }


return 0;
}
```

Código 129: Código de partida del tutorial 7.

2.2 COMPILAR EL CÓDIGO

Hay que estar seguro de que este código compila antes de continuar con el tutorial. La aplicación no debe de realizar nada, salvo presentar una pantalla en negro (se sale presionando Escape). Si se obtienen errores de vinculación, hay que estar seguro de tener cargadas las librerías CEGUIBase_d.lib y CEGUIOgreRenderer_d.lib en la parte de vinculación, entrada.

2.3 PEQUEÑA INTRODUCCIÓN

CEGUI es una librería GUI totalmente caracterizada, que puede introducirse en aplicaciones 3D como Ogre (soporta también OpenGl, DirectX, e Irrlicht). Al igual que Ogre es sólo una librería gráfica, CEGUI es sólo una librería GUI, no hace su propio renderizado ni enlaza eventos de teclado o ratón. De hecho para renderizarlo hay que aportar un renderizador (la librería CEGUIOgreRenderer que trae CEGUI), y para controlar eventos de ratón y teclado, hay que controlarlos de forma manual en el sistema. Esto en principio puede parecer pesado, pero en realidad se necesita muy poco código para hacer que esto funcione.

3. INTEGRACIÓN CON OGRE

3.1 DEFINIENDO LOS GRUPOS DE RECURSOS DE CEGUI

CEGUI al igual que Ogre tiene varios tipos de recursos que necesita para funcionar. Tiene varios manejadores de recursos (como Ogre) que deben de localizar sus respectivos recursos. Así, se necesitan definir los grupos de recursos y su localización dentro de resources.cfg. La distribución CEGUI instala los recursos en diferentes localizaciones dependiendo de la plataforma. Por ejemplo en Linux, los recursos se instalan en /usr/local/share/CEGUI, por lo que se tendría que añadir lo siguiente al archivo resources.cfg de Ogre:

```
[Imagesets]
 FileSystem=/usr/local/share/CEGUI/imagesets
 [Fonts]
```

```
 FileSystem=/usr/local/share/CEGUI/fonts
 [Schemes]
 FileSystem=/usr/local/share/CEGUI/schemes
 [LookNFeel]
 FileSystem=/usr/local/share/CEGUI/looknfeel
 [Layouts]
 FileSystem=/usr/local/share/CEGUI/layouts
```

Código 130: Fichero de recursos CEGUI.

3.2 INICIALIZANDO CEGUI

Hay que buscar la función createScene y añadir el siguiente código:

```
        mRenderer = &CEGUI::OgreRenderer::bootstrapSystem();
```

Código 131: Inicialización de CEGUI.

Ahora que CEGUI ha sido inicializado, se necesitan agregar los grupos de recursos a cada manejador de recursos de CEGUI añadiendo el siguiente código:

```
CEGUI::Imageset::setDefaultResourceGroup("Imagesets");
 CEGUI::Font::setDefaultResourceGroup("Fonts");
 CEGUI::Scheme::setDefaultResourceGroup("Schemes");
 CEGUI::WidgetLookManager::setDefaultResourceGroup("LookNFeel");
CEGUI::WindowManager::setDefaultResourceGroup("Layouts");
```

Código 132: Inicialización de recursos CEGUI.

Como se puede ver, se utilizan los recursos definidos en resources.cfg. CEGUI es altamente personalizable, y permite definir la vista y forma de la aplicación cambiando su "piel" (esquema [scheme], en términos de CEGUI). Aquí no se cubre cómo usar la librería, pero si se desea aprender más, se puede ver en la web de CEGUI. El siguiente código selecciona el recubrimiento o "piel":

```
CEGUI::SchemeManager::getSingleton().create("TaharezLook.scheme");
```

Código 133: Selección de recubrimiento.

La siguiente línea se incluye para establecer el cursor del ratón por defecto:

```
CEGUI::System::getSingleton().setDefaultMouseCursor("TaharezLook",
"MouseArrow");
```

Código 134: Establecimiento del ratón por defecto.

El primer parámetro especifica el juego de imágenes y el segundo especifica el nombre de la imagen a utilizar de este juego de imágenes.

A lo largo de toda esta serie de tutoriales, se utilizará CEGUI para mostrar el cursor del ratón, incluso cuando no tiene uso para la librería GUI. Es posible utilizar otra librería GUI para renderizar el ratón, o se puede crear un cursor de ratón propio utilizando Ogre directamente.

Finalmente hay que tener en cuenta que en el último trozo de código en el que se ha establecido el cursor de ratón por defecto, no se ha establecido utilizando la función MouseCursor::setImage como se verá en próximos tutoriales. Esto es porque en este tutorial se tendrá siempre un tipo de ventana CEGUI (aunque puede estar invisible), de forma que estableciendo el cursor por defecto se hará, en efecto, el cursor del ratón con la imagen seleccionada. Si se establece el cursor directamente y no se establece el por defecto, el cursor se volverá invisible cada vez que se pase por una ventana CEGUI, como se verá en posteriores tutoriales. En esta situación se llamará a MouseCursor::setImage para poner el cursor en la aplicación como en este ejemplo:

```
CEGUI::MouseCursor::getSingleton().setImage(CEGUI::System::getSingleto
n().getDefaultMouseCursor());
```

Código 135: Establecimiento de imagen de ratón.

3.3 INYECTANDO EVENTOS DE TECLA.

CEGUI no maneja entrada. No lee movimientos de ratón o entrada de teclado. En lugar de ello, esto depende del usuario para inyectar eventos de tecla y ratón en el sistema. La siguiente cosa que se necesita hacer es manejar los eventos del teclado. Si se está trabajando con CEGUI, se necesitan tener el teclado y ratón en modo búfer de forma que se puedan recibir los eventos directamente e inyectarlos como ocurren. Hay que buscar la función keyPressed y añadir el siguiente código:

```
CEGUI::System &sys = CEGUI::System::getSingleton();
Sys.injectKeyDown(arg.key);
Sys.injectChar(arg.text);
```

Código 136: Inyección de eventos de tecla.

Después de obtener el objeto de sistema, se necesitan hacer dos cosas. La primera es inyectar el evento de pulsación de tecla en CEGUI. La segunda es inyectar el carácter actual que se ha pulsado. Es muy importante inyectar el carácter apropiadamente ya que inyectar la pulsación de tecla no siempre dará el resultado deseado cuando se utiliza un teclado no inglés. La función injectChar se diseñó con la compatibilidad Unicode en mente.

Ahora se necesita inyectar el evento de soltado de tecla en el sistema. Hay que buscar la función keyReleases y añadir el siguiente código:

```
CEGUI::System::getSingleton().injectKeyUp(arg.key);
```

Código 137: Inyección de soltado de tecla.

En este caso no se necesita el carácter.

3.4 CONVIRTIENDO E INYECTANDO EVENTOS DE RATÓN

Ahora que se ha finalizado la entrada de teclado, hay que tener en cuenta la entrada de ratón. Hay una pequeña cuestión que se necesita direccionar. Cuando se inyectan los eventos de pulsado, soltado de teclado en CEGUI, no se necesita convertir la tecla. Tanto OIS como CEGUI utilizan los mismos códigos de tecla para la entrada de teclado. Pero no son iguales para los botones del ratón. Antes de poder inyectar las pulsaciones de botón de ratón en CEGUI, se necesita escribir una función que convierte los IDs de botón de OIS a botón CEGUI. Hay que añadir la siguiente función junto al inicio del código fuente, justo antes de la clase TutorialListener:

```
CEGUI::MouseButton convertButton(OIS::MouseButtonID buttonID)
{
    switch (buttonID)
    {
    case OIS::MB_Left:
        return CEGUI::LeftButton;

    case OIS::MB_Right:
        return CEGUI::RightButton;

    case OIS::MB_Middle:
        return CEGUI::MiddleButton;

default:
```

```
        return CEGUI::LeftButton;
}
}
```

Código 138: Método de conversión de referencias al ratón entre OIS y CEGUI.

Ahora se está preparado para inyectar eventos de ratón. Hay que buscar la función mousePressed y añadir el siguiente código:

```
CEGUI::System::getSIngleton().injectMouseButtonDown(convertButton(id))
;
```

Código 139: Inyección de pulsación de tecla de ratón.

Se convierte el botón y el resultado se pasa a CEGUI. Ahora se busca la función mouseReleased y se añade el siguiente código:

```
CEGUI::System::getSingleton().injectMouseButtonUp(convertButton(id));
```

Código 140: Inyección de soltado de tecla de ratón.

Finalmente se necesita insertar el movimiento del ratón en CEGUI. El objeto CEGUI::System tiene una función injectMouseMove que espera movimientos relativos de ratón. El manejador OIS::mouseMoved da esos movimientos relativos en las variables state.X.rel y state.Y.rel. Hay que buscar la función mouseMoved y añadir el siguiente código:

```
CEGUI::System::getSingleton().injectMouseMove(arg.state.X.rel,
arg.state.Y.rel);
//Scroll wheel.
If (e.state.Z.rel)
   mSystem->injectMouseWheelChange(e.state.Z.rel / 120.0f);
```

Código 141: Inyección de movimiento de ratón.

120 es en resumen un 'número mágico' utilizado por Microsoft. OIS también lo utiliza.

Ahora CEGUI está completamente configurado para recibir eventos de teclado y ratón.

4. VENTANAS, HOJAS, Y COMPONENTES

4.1 INTRODUCCIÓN

CEGUI es muy diferente a la mayoría de los sistemas GUI. En CEGUI, todo lo que se muestra es una subclase de CEGUI::Window, y una ventana puede tener cualquier número de ventanas hijo. Esto significa que cuando se crea un fotograma para contener varios botones, ese fotograma es una ventana (Window). Esto lleva a varias cosas raras a ocurrir. Se puede poner un botón dentro de otro botón, aunque en realidad esto nunca se hará en la práctica. La razón por la que se menciona esto es porque cuando se busca un componente particular que se ha establecido en la aplicación, todos ellos se llaman Windows y se accede a ellos a través de funciones que los referencian como ventanas.

En los usos más prácticos de CEGUI, no se creará cada objeto individual a través del código. En lugar de ello se crea una capa GUI para la aplicación en un editor como CEGUI Layout Editor. Después de establecer todas las ventanas, botones, y otros complementos en la pantalla como se desea, el editor guarda la capa en un fichero de texto. Luego se puede cargar esta capa en CEGUI llamando a una hoja GUI (que también es una subclase de CEGUI::Window).

Finalmente, para ver cómo utilizar todos estos componentes, hay que consultar la página de CEGUI.

4.2 CARGANDO UNA HOJA

La carga de una hoja CEGUI es muy sencilla. La clase WindowManager proporciona una función "loadWindowLayout" que carga la hoja y la coloca en un objeto CEGUI::Window. Luego se llamará a CEGUI::System::setGUISheet para mostrarla. No se utilizará esto en este tutorial, aunque en caso de añadirlo, estas serían las líneas para añadir:

```
// Do not add this to the program
 CEGUI::Window *guiRoot =
CEGUI::WindowManager::getSingleton().loadWindowLayout("TextDemo.layout
");
 CEGUI::System::getSingleton().setGUISheet(guiRoot);
```

Código 142: Carga de una hoja.

Esto establece la hoja actual para mostrar. Se puede posteriormente recuperar esta hoja llamando a System::getGUISheet. Se puede también cambiar la hoja GUI sin error llamando a setGUISheet.

4.3 CREANDO UN OBJETO MANUALMENTE

Como se ha dicho anteriormente, la mayor parte del tiempo que se use CEGUI, se usarán hojas que se han creado utilizando un editor. Ocasionalmente, sin embargo, se necesitará crear manualmente un componente para ponerlo en la pantalla. En este ejemplo, se añadirá un botón de Salir al que se le añadirá funcionalidad. Como sólo se añadirá un botón, primero se necesita crear una ventana por defecto CEGUI::Window que contendrá todos los componentes que se vayan creando. Se añade lo siguiente al final de la función createScene:

```
CEGUI::WindowManager &wmgr = CEGUI::WindowManager::getSingleton();
CEGUI::Window *sheet = wmgr.createWindow("DefaultWindow",
"CEGUIDemo/Sheet");
```

Código 143: Creación de hoja en código.

Esto utiliza el WindowManager para crear una "DefaultWindow" llamada "CEGUIDemo/Sheet". Aunque se puede nombrar como se quiera, es muy común (y recomendado) nombrar el componente de una forma de herencia como "Aplicación/MenuPrincipal/Submenu3/BotonCancel". La siguiente cosa a realizar es crear el botón Quit y establecer su tamaño:

```
CEGUI::Window *quit = wmgr.createWindow("TaharezLook/Button",
"CEGUIDemo/QuitButton");
 quit->setText("Quit");
 quit->setSize(CEGUI::UVector2(CEGUI::UDim(0.15, 0), CEGUI::UDim(0.05,
0)));
```

Código 144: Creación de un botón Quit.

Esto es cercano a ser enigmático. CEGUI utiliza un sistema de "dimensión unificada" para las posiciones y tamaños. Cuando se establece el tamaño, se debe crear un objeto UDim para decir el tamaño que debería de ser. El primer parámetro es el tamaño relativo del objeto en relación con su padre. El segundo parámetro es el tamaño absoluto del objeto (en píxeles). La cosa importante a realizar es que sólo se debe de establecer uno de los dos parámetros de UDim. El otro parámetro debe de ser 0. De forma que en este caso se ha creado un botón con una anchura del 15% de la anchura de su padre y una altura de un 5% con respecto a la altura de su padre. Si se quisiera especificar que el tamaño fuera de 20 píxeles por 5 píxeles, se debería de hacer igual pero rellenando el segundo parámetro de UDim en lugar del primero a 20 y 5 respectivamente.

La última cosa que se tiene que hacer es agregar el botón de Quit a la hoja que se ha creado, y luego establecer la hoja GUI actual para el sistema:

```
sheet->addChildWindow(quit);
 CEGUI::System::getSingleton().setGUISheet(sheet);
```

Código 145: Colocación del botón en la hoja.

Ahora si se compila y ejecuta la aplicación, se verá un botón Quit en la parte superior izquierda de la pantalla, pero no pasa nada cuando se pulsa sobre él.

5. EVENTOS

Los eventos en CEGUI son muy flexibles. En lugar de utilizar un interfaz que se implementa para recibir eventos, utiliza un mecanismo de retrollamada que enlaza cualquier función pública para ser manejadora de eventos. Desafortunadamente esto significa que registrar eventos es un poco más complicado que lo es en Ogre. Ahora se registrará un evento para manejar los clics en el botón Quit para salir del programa cuando se pulsa el botón. Para hacer esto, primero se necesita un puntero al botón que se creó en la sección anterior. Hay que buscar el constructor de TutorialListener y añadir el siguiente código:

```
CEGUI::WindowManager &wmgr = CEGUI::WindowManager::getSingleton();
 CEGUI::Window *quit = wmgr.getWindow("CEGUIDemo/QuitButton");
```

Código 146: Obtención de un puntero a un botón.

Ahora que se tiene un puntero al botón, se subscribirá el evento de clic. Cada componente en CEGUI tiene un juego de eventos que soporta, y todos ellos empiezan con “Event”. Aquí está el código para registrar el evento:

```
quit->subscribeEvent(CEGUI::PushButton::EventClicked,
CEGUI::Event::Subscriber(&TutorialListener::quit, this));
```

Código 147: Registro de un evento de clic.

El primer parámetro es el evento en sí. El segundo parámetro es un objeto Event::Subscriber. Cuando se crea un objeto Subscriber, la primera cosa que se pasa es un puntero a la función que manejará el evento (nota, el símbolo & da un puntero de la función). La segunda cosa que se pasa es el objeto TutorialListener que manejará el evento (en este caso this). Ahora la función TutorialListener::quit (que ya ha sido definida) manejará el clic del ratón y finalizará el programa.

Hay que compilar y ejecutar el programa para comprobar la funcionalidad.

Una cosa a tener en cuenta es que se puede crear cualquier número de funciones de manejo de eventos de CEGUI. La única restricción es que deben retornar un boolean y deben tomar un solo parámetro de tipo “const CEGUI::EventArgs &”. Para más información, ver la web de CEGUI.

6. RENDERIZAR EN TEXTURA

Una de las cosas más interesantes que se pueden hacer con CEGUI es crear una ventana de renderizado en textura (RTT: http://www.ogre3d.org/wiki/index.php/RTT). Esto permite crear un segundo puerto de vista que puede ser renderizado directamente en un complemento CEGUI. Para hacer esto, primero hay que establecer una escena para ver. Hay que añadir el siguiente código al final de la función createScene:

```
mSceneMgr->setAmbientLight(ColourValue(1, 1, 1));
 mSceneMgr->setSkyDome(true, "Examples/CloudySky", 5, 8);
```

```
 Entity* ogreHead = mSceneMgr->createEntity("Head", "ogrehead.mesh");
 SceneNode* headNode = mSceneMgr->getRootSceneNode()-
>createChildSceneNode(Vector3(0, 0, -300));
headNode->attachObject(ogreHead);
```

Código 148: Establecimiento de un escenario básico.

Ahora hay que crear una RenderTexture. El objeto RenderSystem proporciona la funcionalidad para renderizar una textura. Para hacer esto, se crea una textura con la función TextureManager::createManual. Para este programa se crea una textura 512 x 512:

```
TexturePtr tex = mRoot->getTextureManager()->createManual(
"RTT",
   ResourceGroupManager::DEFAULT_RESOURCE_GROUP_NAME,
   TEX_TYPE_2D,
   512,
   512,
   0,
   PF_R8G8B8,
TU_RENDERTARGET);
RenderTexture *rtex = tex->getBuffer()->getRenderTarget();
```

Código 149: Creación de una textura de renderizado.

Ver la referencia API para obtener más información sobre esta función. A continuación se necesita crear una Camera y un Viewport para apuntar a la escena que se ha creado. Hay que tener en cuenta que se han cambiado algunas opciones de Viewport, incluyendo apagar los recubrimientos (Overlays)... que es muy importante hacer o se obtendrán recubrimientos de CEGUI y Ogre dentro de la mini-ventana.

```
Camera *cam = mSceneMgr->createCamera("RTTCam");
 cam->setPosition(100, -100, -400);
 cam->lookAt(0, 0, -300);
 Viewport *v = rtex->addViewport(cam);
 v->setOverlaysEnabled(false);
 v->setClearEveryFrame(true);
v->setBackgroundColour(ColourValue::Black);
```

Código 150: Creación de cámara y viewport.

Hay que tener en cuenta que se añade el Viewport a la textura (en lugar de al RenderWindow, como se hace normalmente). Ahora se ha creado la escena y la textura, se necesita incrustar en CEGUI. Se puede crear una CEGUI::Texture desde cualquier textura de Ogre llamando a CEGUI::OgreRenderer::createTexture:

```
CEGUI::Texture &guiTex = mRenderer->createTexture(tex);
```

Código 151: Enlazce de textura con CEGUI.

Desafortunadamente, aquí es donde las cosas se complican. En CEGUI nunca se dará una única textura o imagen. CEGUI trabaja con un set en lugar de con una única imagen. Esto es útil para trabajar con rejillas enteras de imágenes cuando se intenta definir una apariencia que se está creando (por ejemplo, ver TaharezLook.tga en datafiles/imagesets). Sin embargo, incluso cuando sólo se intenta definir una única imagen, se debe crear un juego de imágenes entero para ello. Así es cómo se hará:

```
CEGUI::Imageset &imageSet =
   CEGUI::ImagesetManager::getSingleton().create("RTTImageset",
guiTex);
 imageSet.defineImage("RTTImage",
                      CEGUI::Point(0.0f, 0.0f),
CEGUI::Size(guiTex.getSize().d_width,
                                  guiTex.getSize().d_height),
```

```
                                CEGUI::Point(0.0f, 0.0f));
```

Código 152: Creación de un juego de imágenes.

La primera línea crea el juego de imágenes (llamado “RTTImageset”) de la textura que se proporciona. La siguiente línea (que llama a defineImage), especifica que la primera y única imagen se llama “RTTImage” y es tan grande como la textura guiTex que se ha proporcionado. Finalmente se necesita crear el componente StaticImage que renderizará la textura. La primera parte no es diferente de crear cualquier otra ventana:

```
CEGUI::Window *si =
CEGUI::WindowManager::getSingleton().createWindow("TaharezLook/StaticI
mage", "RTTWindow");
si->setSize(CEGUI::UVector2(CEGUI::UDim(0.5f, 0),
                                CEGUI::UDim(0.4f, 0)));
 si->setPosition(CEGUI::UVector2(CEGUI::UDim(0.5f, 0),
                                    CEGUI::UDim(0.0f, 0)));
```

Código 153: Creación de ventana CEGUI.

Ahora se necesita especificar qué imagen de este componente StaticWidget se mostrará. Una vez de nuevo, como CEGUI siempre funciona con juegos de imágenes, se debe de obtener el nombre de imagen exacto del juego de imágenes, y mostrarlo:

```
si->setProperty("Image",
CEGUI::PropertyHelper::imageToString(&imageSet.getImage("RTTImage")));
```

Código 154: Establecimiento de imagen en ventana.

Lo que se ha hecho es empaquetar la imagen en un juego de imágnes para luego extraerlo. Manipular imágenes no es una parte sencilla de CEGUI. Lo último que se necesita es añadir el complemento StaticImage a la hoja GUI que se creó anteriormente:

```
sheet->addChildWindow(si);
```

Código 155: Establecimiento de ventana en la hoja.

Ahora se ha finalizado, sólo hay que compilar y ejecutar la aplicación.

7. CONCLUSIÓN

7.1 ALTERNATIVAS

QuickGUI (http://www.ogre3d.org/wiki/index.php/QuickGUI).

BetaGUI(http://www.ogre3d.org/wiki/index.php/BetaGUI).

MyGUI (http://www.ogre3d.org/wiki/index.php/MyGUI).

7.2 MÁS INFORMACIÓN

Tutoriales de CEGUI (http://www.cegui.org.uk/api_reference/).

Web oficial (http://www.cegui.org.uk/wiki/index.php/Main_Page).

TUTORIAL BÁSICO 8: USANDO VARIOS MANEJADORES DE ESCENA

1. INTRODUCCIÓN

En este pequeño tutorial, se cubre cómo intercambiar entre manejadores de escena múltiple.

2. PRERREQUISITOS

Hay que crear un archivo .cpp y añadir el siguiente código:

```
#include "ExampleApplication.h"
#define CAMERA_NAME "SceneCamera"

void setupViewport(RenderWindow *win, SceneManager *curr)
{
}

void dualViewport(RenderWindow *win, SceneManager *primary,
SceneManager *secondary)
{
}

class SMTutorialListener : public ExampleFrameListener, public
OIS::KeyListener
{
public:
    SMTutorialListener(RenderWindow* win, SceneManager *primary,
SceneManager *secondary)
        : ExampleFrameListener(win, primary->getCamera(CAMERA_NAME),
true, false),
          mPrimary(primary), mSecondary(secondary), mDual(false),
mContinue(true)
{
        mKeyboard->setEventCallback(this);
    }

bool frameStarted(const FrameEvent& evt)
{
        mKeyboard->capture();
        return mContinue;
    }

bool keyPressed(const OIS::KeyEvent &arg)
{
       switch (arg.key)
       {
case OIS::KC_ESCAPE:
           mContinue = false;
           break;

       default:
           break;
       }

       return true;
    }

    bool keyReleased(const OIS::KeyEvent &) {return true;}
```

```
private:
    SceneManager *mPrimary, *mSecondary;
bool mDual, mContinue;

    static void swap(SceneManager *&first, SceneManager *&second)
    {
        SceneManager *tmp = first;
        first = second;
        second = tmp;
    }
};

class SMTutorialApplication : public ExampleApplication
{
public:
    SMTutorialApplication()
    {
    }

    ~SMTutorialApplication()
    {
    }
protected:
    SceneManager *mPrimary, *mSecondary;

    void chooseSceneManager(void)
    {
    }

    void createCamera()
    {
    }

    void createViewports()
    {
    }

    void createScene(void)
    {
}

    void createFrameListener(void)
    {
mFrameListener = new SMTutorialListener(mWindow, mPrimary,
mSecondary);
        mFrameListener->showDebugOverlay(true);
        mRoot->addFrameListener(mFrameListener);
    }
};

#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
#define WIN32_LEAN_AND_MEAN
#include "windows.h"

INT WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT)
#else
int main(int argc, char **argv)
#endif
{
    // Create application object
    SMTutorialApplication app;
```

```
    try {
        app.go();
    } catch(Exception& e) {
#if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
        MessageBoxA(NULL, e.getFullDescription().c_str(), "An
exception has occurred!", MB_OK | MB_ICONERROR | MB_TASKMODAL);
#else
fprintf(stderr, "An exception has occurred: %s\n",
            e.getFullDescription().c_str());
#endif
    }

return 0;
}
```

Código 156: Código de partida del tutorial 8.

Hay que asegurarse de que se puede compilar este código antes de continuar, pero no hay que ejecutar el código aún.

3. CONFIGURANDO LA APLICACIÓN

3.1 CREANDO LOS SCENEMANAGERS

Se ha cubierto anteriormente cómo seleccionar un SceneManager, de forma que no se entrará en el detalle de esta función. La única cosa que se ha cambiado es que ahora se crean dos SceneManagers. Hay que buscar la función chooseSceneManager y añadir el siguiente código:

```
mPrimary = mRoot->createSceneManager(ST_GENERIC, “primary”);
mSecondary = mRoot->createSceneManager(ST_GENERIC, “secondary”);
```

Código 157: Creación de dos SceneManagers.

3.2 CREANDO LAS CÁMARAS

Lo siguiente a realizar es crear un objeto Camera para cada uno de los SceneManagers. La única diferencia con respecto a tutoriales anteriores es que en esta ocasión se están creando dos cámaras con el mismo nombre. Hay que localizar la función createCamera y añadir el siguiente código:

```
        mPrimary->createCamera(CAMERA_NAME);
        mSecondary->createCamera(CAMERA_NAME);
```

Código 158: Creación de dos cámaras.

3.3 CREANDO LOS PUERTOS DE VISTAS (VIEWPORTS)

Creando Viewports para esta aplicación, se tendrán que hacer algunas diferencias con respecto a tutoriales anteriores. Cuando se crea un viewport, hay que hacer dos cosas: configurar el Viewport y establecer la proporción de aspecto de la cámara que se está utilizando. Para comenzar, hay que añadir el siguiente código a la función createViewports:

```
        setupViewport(mWindow, mPrimary);
```

Código 159: Establecimiento de un Viewport.

El código actual para configurar un Viewport está dentro de la función setupViewport, ya que se utilizará más veces en el código. La primera cosa que se necesita realizar es borrar todos los Viewports creados previamente. Ahora mismo no hay ninguno creado, pero cuando se llame de nuevo a esta función

después, hay que estar seguro de que se borran todos los Viewports. Después de esto se configuran los Viewport igual que en otros tutoriales. Hay que añadir el siguiente código a la función setupViewport al inicio del fichero:

```
        win->removeAllViewports();

        Camera *cam = curr->getCamera(CAMERA_NAME);
        Viewport *vp = win->addViewport(cam);

        vp->setBackgroundColour(ColourValue(0, 0, 0));
        cam->setAspectRatio(Real(vp->getActualWidth()) / Real(vp-
>getActualHeight());
```

Código 160: Código del método setupViewPort.

3.4 CREANDO EL SCENE

Finalmente, se necesita crear un Scene por cada SceneManager. No se va ha hacer nada complejo, simplemente algo diferente de forma que se sepa cuándo se ha intercambiado entre los dos. Hay que buscar la función createScene y añadir el siguiente código:

```
        //Setup the TerrainSceneManager
        mPrimary->setSkyBox(true, "Examples/SpaceSkyBox");

        //Setup de Generic SceneManager
        mSecondary->setSkyDome(true, "Examples/CloudySky", 5, 8);
```

Código 161: Creación de dos escenarios.

Hay que estar seguro que el código compila antes de continuar. Se puede ejecutar el programa en este punto, pero no tendrá ninguna funcionalidad distinta a salir cuando se pulsa Escape.

4. AÑADIENDO FUNCIONALIDAD

4.1 SCENEMANAGERS DUALES

El primer trozo de funcionalidad que se quiere añadir al programa es permitir al usuario renderizar ambos SceneManagers cara a cara. Cuando se pulsa la tecla V, se cambia a modo de Viewport dual. El plan básico para esto es simple. Para apagar el modo Viewport dual, simplemente se puede llamar a setupViewport (que se ha creado en la sección anterior) con el SceneManager primario para recrear el Viewport en modo simple. Cuando se quiere encender, se llamará a una nueva función llamada dualViewport. Hay que guardar el estado del Viewport con la variable mDual. Hay que añadir el siguiente código al switch de la función keyPressed:

```
        Case OIS::KC_V:
                mDual = !mDual;

                if (mDual)
                        dualViewport(mWindow, mPrimary, mSecondary);
                else
                        setupViewport(mWindow, mPrimary);
                break;
```

Código 162: Código de keyPressed para control de vista.

Ahora se intercambia el valor de la variable mDual y se llama a la función apropiada basándose en el modo en el que se está. Ahora se define la función dualViewport que contiene el código para mostrar dos Viewports a la vez.

Para mostrar dos SceneManagers cara a cara, básicamente se hace lo mismo que se ha hecho en la función setupViewport. La única diferencia es que se crean dos Viewport, uno para cada cámara. Hay que añadir el siguiente código a la función dualViewport:

```
win->removeAllViewports();

   Viewport *vp = 0;
   Camera *cam = primary->getCamera(CAMERA_NAME);
   vp = win->addViewport(cam, 0, 0, 0, 0.5, 1);
   vp->setBackgroundColour(ColourValue(0,0,0));
   cam->setAspectRatio(Real(vp->getActualWidth()) / Real(vp-
>getActualHeight()));

   cam = secondary->getCamera(CAMERA_NAME);
   vp = win->addViewport(cam, 1, 0.5, 0, 0.5, 1);
   vp->setBackgroundColour(ColourValue(0,0,0));
   cam->setAspectRatio(Real(vp->getActualWidth()) / Real(vp-
>getActualHeight()));
```

Código 163: Código de dualViewport.

Todo esto debería resultar familiar excepto por los parámetros extra que se han añadido en la llamada a la función addViewport. El primer parámetro de esta función es la cámara que se está usando. El segundo parámetro es el orden z del Viewport. Un orden alto de z se establece por encima de órdenes z más pequeños. Hay que tener en cuenta que no se pueden tener dos Viewports con órdenes z iguales, incluso si no se superponen. Los siguientes dos parámetros son las posiciones izquierda y superior de los Viewport, que deben de estar entre 0 y 1. Finalmente, los últimos dos parámetros son la anchura y altura de los Viewport como un porcentaje de la pantalla (de nuevo con valores entre 0 y 1). Así que en este caso, el primer Viewport que se crea estará en la posición (0, 0) y ocupará media pantalla de ancho y toda la pantalla de alto (0.5, 1). El segundo Viewport estará en la posición (0.5, 0) y también ocupará media pantalla de alcho y toda la pantalla de alto.

Ahora se compila y ejecuta la aplicación. Pulsando V se pueden mostrar dos SceneManagers al mismo tiempo.

4.2 INTERCAMBIANDO SCENEMANAGERS

La última pieza de la funcionalidad que hay que añadir al programa, es intercambiar los SceneManagers cuando se pulsa la tecla C. Para hacer esto, primero se intercambian las variables mPrimary y mSecondary de forma que cuando se llama a las funciones setupViewport o dualViewport, no habrá que preocuparse de qué SceneManager está en cada variable. La SceneManager primaria siempre se mostrará en modo simple, y la primaria siempre se mostrará en la parte izquierda en el modo dual. Hay que añadir el siguiente código al switch de la función keyPressed:

```
        Case OIS::KC_C:
                Swap(mPrimary, mSecondary);
```

Código 164: Código para keyPressed con intercambio de vista.

Después de intercambiar las variables, se necesita realmente hacer el cambio. Hay que hacer esto llamando a la función se configuración del Viewport apropiado dependiendo de si se está en modo simple o dual:

```
        If (mDual)
                dualViewport(mWindow, mPrimary, mSecondary);
        else
```

```
setupViewport(mWindow, mPrimary);
```

Código 165: Código para intercambio de vista.

Ahora se compila y ejecuta el programa. Intercambiando los Viewport con C y cambiado de modo con V.

5. CONCLUSIÓN

5.1 RECUBRIMIENTOS (OVERLAYS)

Hay que notar que cuando se ejecuta en modo dual, El Overlay de debug de Ogre muestra dos lados. Se puede apagar el renderizado de Overlay, para basarse en Viewport. Hay que utilizar la función Viewport::setOverlaysEnabled para apagarlos.

5.2 ÚLTIMA NOTA

Hay que mantener siempre en mente que la clase Viewport, aunque no tiene mucha funcionalidad por si misma, es la llave para todo el renderizado Ogre. No importa cúantos SceneManagers se creen, o cuantas Cameras en cada SceneManager, ninguno se renderizará a no ser que se configure un Viewport por cada cámara se que está mostrando. No hay que olvidar el limpiar los Viewports que no se están utilizando antes de crear más.