Carga y Visualización de Modelos 3D mediante JNI y OpenSceneGraph

Autor: Rafael Gaitán <rgaitan@ai2.upv.es>,

Fecha: 2008-04-10, 00:49

Descripción: Con la realización del taller se pretende dotar de las nociones básicas de JNI a los asistentes, desarrollando algunas envolturas JNI para la visualización de escenas 3D desde Java, haciendo uso de la librería OpenSceneGraph. A los alumnos se les dará unas nociones básicas iniciales y una infraestructura sencilla, tales como proyectos base, ejemplos y entorno para la programación multiplataforma con JNI.

Tabla de Contenidos

1. Introducción **5**

2. Preparación del Entorno de Trabajo **7**

2.1. Estructura base 7

2.2. Importar Proyectos en Eclipse 8

2.3. Generación de las Librerías Nativas 11

2.4. Estructura de los Proyectos 12

2.4.1. Estructura de los Proyectos JNI. 13

2.5. Generación del Producto Final 14

3. Nociones Básicas de *Java Native Interface* (JNI) **15**

3.1. Ejemplo JNI 15

3.1.1. Declaración del método nativo 16

3.1.2. Carga de la librería 16

3.2. Generar Archivo de Cabecera 17

3.3. Implementar Método Nátivo 18

3.3.1. Compilación librería nativa 18

3.4. Ejecutar Ejemplo 18

4. Trabajo a realizar 21

4.1. Ejercicio 1: Ampliación de Ejemplo-JNI 21

4.2. Ejercicio 2: JNI de la función para carga de escenas 3D 22

4.3. Ejercicio 3: JNI de un Visualizador de escenas 3D 24

5. Introducción a OpenSceneGraph 27

5.1. Beneficios del Uso de un Grafo de Escena 28

5.2. Grafo de Escena OpenSceneGraph 29

6. Nociones Avanzadas de JNI 31

6.1. Tipos de Datos 31

6.1.1. Tipos Básicos 32

6.1.2. Arrays de Tipos Básicos 33

6.2. JNI de clases C++ 34

1

Introducción

El mundo del desarrollo de aplicaciones se ha vuelto, a día de hoy, complejo, y con múltiples tecnologías involucradas. En la dificil tarea de obtener un software de calidad, Java es una de los lenguajes más extendidos.

En el mundo de las aplicaciones web, Java es un ganador. En las aplicaciones de escritorio se va implantando poco a poco gracias a *frameworks* como eclipse que pueden eriquecerse de forma fácil mediante una arquitectura de plugins y con una interfaz de usuario más amigable.

Java Native Interface es un framework de programación que permite que un programa escrito en Java, pueda interactuar con programas escritos en otros lenguajes como C y C++.

JNI se usa habitualmente para escribir métodos nativos cuando la librería de clases de Java no proporciona el soporte necesario a algún tipo de funcionalidad. Muchas de las clases de la librería estándar de Java dependen del JNI para proporcionar funcionalidad al desarrollador y al usuario, por ejemplo las funcionalidades de sonido o lectura/escritura de ficheros. Antes de comenzar un desarrollo es necesario asegurarse que la librería estándar de Java no proporciona una determinada funcionalidad antes de recurrir al JNI, ya que la primera ofrece una implementación segura e independiente de la plataforma.

Con la realización del taller se pretende dotar de las nociones básicas de JNI a los asistentes, desarrollando algunas envolturas JNI para la visualización de escenas 3D desde Java, haciendo uso de la librería OpenSceneGraph. A los alumnos se les dará unas nociones básicas iniciales y una infraestructura sencilla, tales como proyectos base, ejemplos y entorno para la programación multiplataforma con JNI.

El presente documento primero aborda la preparación del entorno de trabajo para el taller, posteriormente y ya teniendo los proyectos listos para funcionar se explica un ejemplo básico para una primera toma de contacto con JNI. A continuación se explica el trabajo a realizar. Los siguientes apartados tratan de forma breve OpenSceneGraph y algunas nociones avanzadas de JNI para los curiososos. Finalmente se listaran una serie de referencias de utilidad para profundizar más en JNI.

2

Preparación del Entorno de Trabajo

Tras la lectura de este apartado, se tendrá configurado el entorno de programación para el desarrollo de librerías JNI, haciendo uso de Maven como sistema de compilación y gestión de las dependencias en Java, de CMake para la compilación y generación de librerías nativas y finalmente algunos scripts en ANT que ayudarán a la gestión y automatización de algunas tareas.

2.1. Estructura base

Con el taller se proporciona un *.tar.gz* con la estructura del proyecto y con los binarios de **Maven** y **CMake**. Antes de pasar a trabajar es necesario descomprimir *encuentrosjava-taller-jni.tar.gz*.

Para la realización de algunos ejercicios en el taller es necesario hacer uso de dependencias externas. En este caso se requiere OpenSceneGraph. En el directorio *osg* de los fuentes descomprimidos, se encuentra un fichero .zip con los binarios precompilados de OSG. Descomprimirlo en el mismo directorio.

2.2. Importar Proyectos en Eclipse

Los pasos a seguir para poder importar los proyectos de eclipse y comenzar el trabajo es el siguiente:

1. Abrir Eclipse y seleccionar como workspace el directorio donde se ha descomprimido el fichero anterior.

2. Creación de una *external tool* para crear los proyectos de eclipse con **Maven**.

    1. Menú **Run**, **External Tools**, **Open Externals Tools Dialog**.
    2. **Program**, **New**
    3. Utilizar la siguiente configuración:

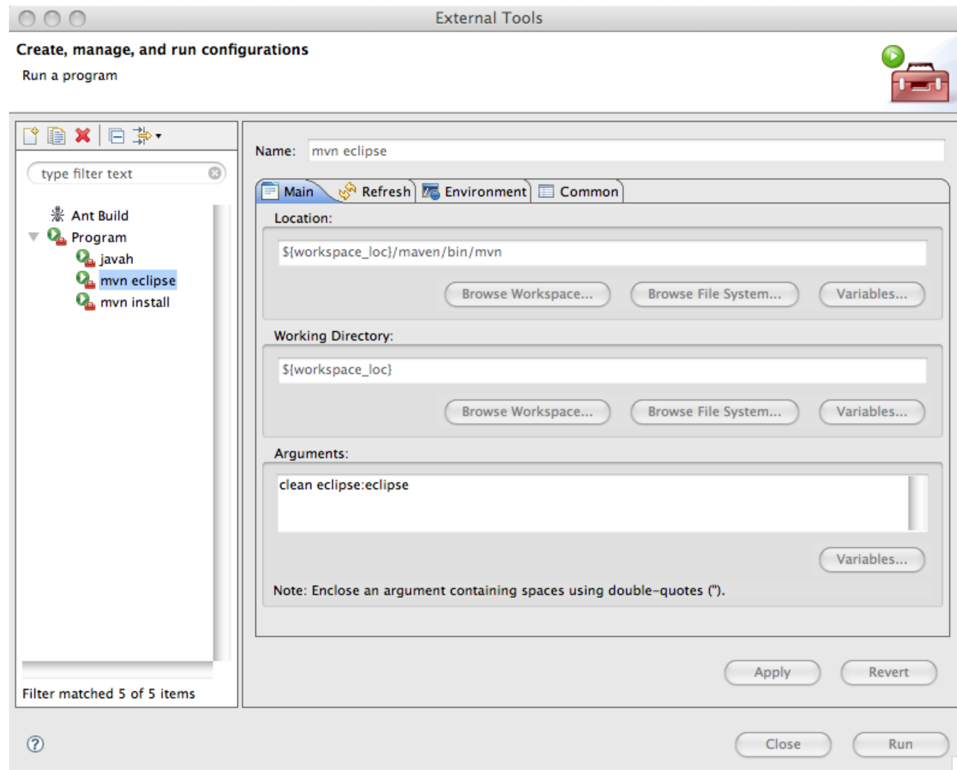
       Name: mvn eclipse

       Location:

       ${workspace_loc}/maven/bin/mvn (Unix)
       ${workspace_loc}\maven\bin\mvn.bat (Windows)

       Working Directory: ${workspace_loc}

       Arguments: clean eclipse:eclipse



3. Creación de una *external tool* para configurar el repositorio local de dependencias de **Maven** en Eclipse. *Para agilizar la creación de la external tool se puede duplicar el anterior y cambiar el nombre y los argumentos*.

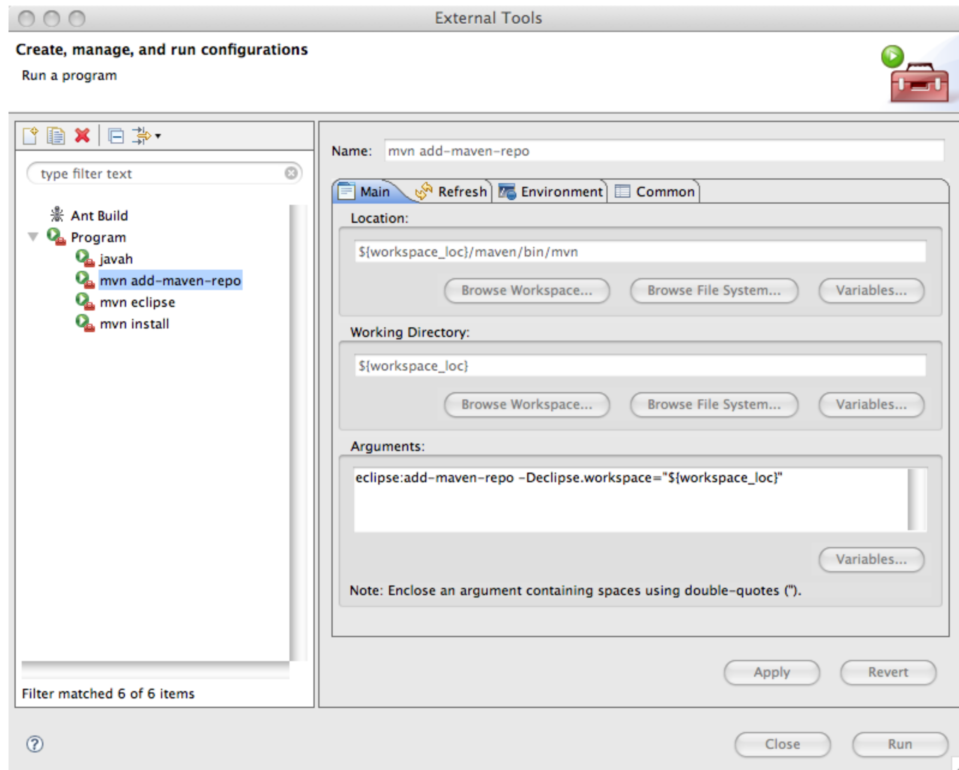
   Name: mvn eclipse

   Location:

   > ${workspace_loc}/maven/bin/mvn (Unix)
   > ${workspace_loc}\maven\bin\mvn.bat (Windows)

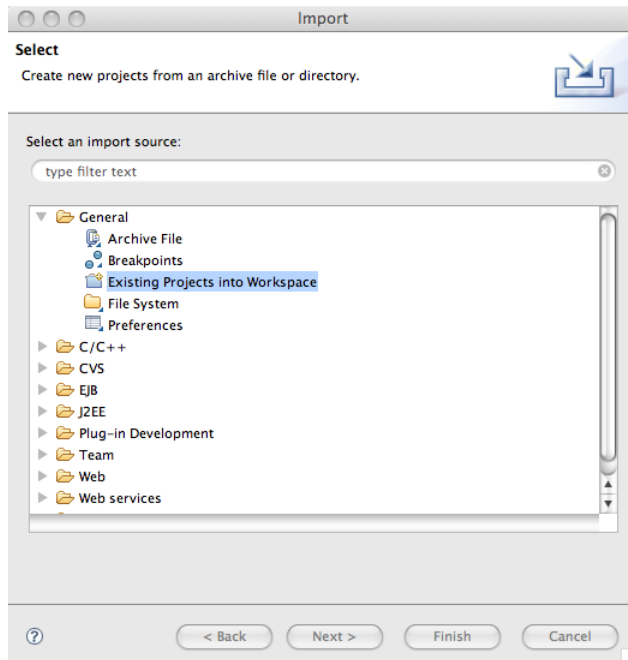
   Working Directory: ${workspace_loc}

   Arguments: eclipse:add-maven-repo -Declipse.workspace=${workspace_loc}



4. Reiniciar eclipse para que tenga en cuenta la variable *M2_REPO* recien configurada mediante la *external tool*.

5. Importar los proyectos de Eclipse. **File**, **Import**, **General**, **Existing Projects into Workspace** y a continuación seleccionar el directorio del workspace de eclipse (donde se encuentran los fuentes del taller)



2.3. Generación de las Librerías Nativas

Para la generación de las librerías nativas, se hace uso de **CMake**, que es una herramienta que permite la creación de los proyectos de construcción de las librerías en múltiples plataformas y para múltiples sistemas.

Para facilitar el trabajo y la construcción integrada dentro del entorno de trabajo se ha hecho uso de un script de **ANT** que es lanzado desde **Maven**. El script de **ANT** está preparado para hacer uso de la copia que se ha distribuido de CMake, por lo que creando una *external tool* en eclipse con el *target* adecuado será suficiente.

Configuración del *external tool* para la compilación de las librerías nativas:

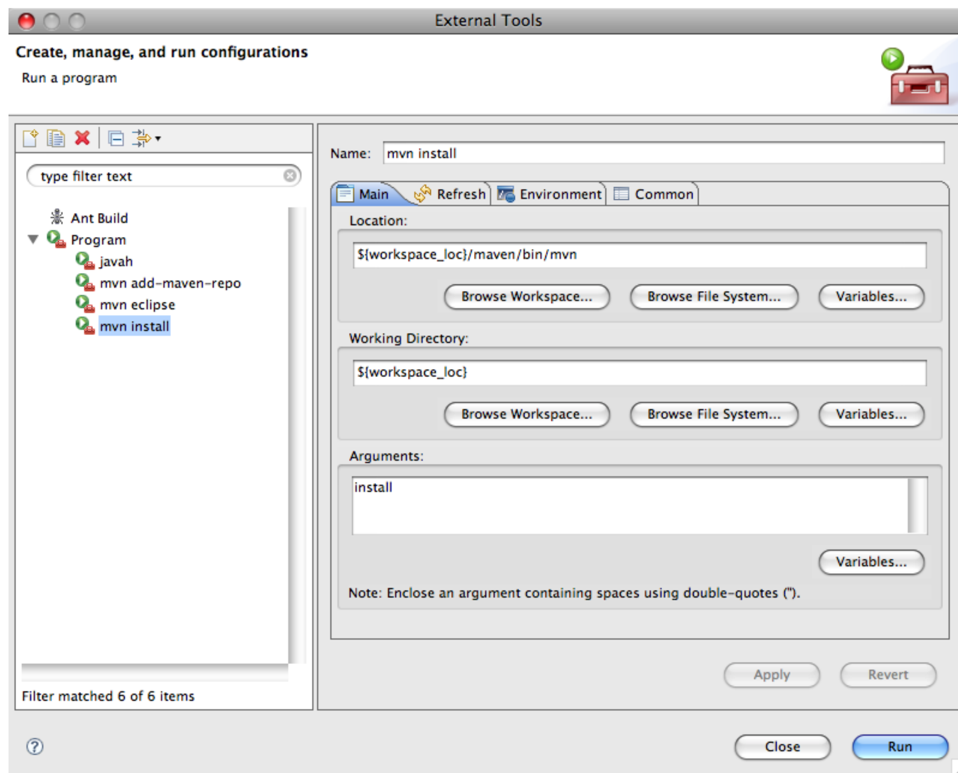
Name: mvn install

Location:

${workspace_loc}/maven/bin/mvn (Unix)
${workspace_loc}\maven\bin\mvn.bat (Windows)

Working Directory: ${workspace_loc}

Arguments: install



El script de **ANT**, determina en que plataforma se está y genera una configuración de **CMake** diferente para cada una. Por ejemplo para el entorno Windows, hace uso de NMAKE, de manera que genera los makefiles que NMAKE sabe interpretar. Para sistemas Unix (Linux y MacOsX), hace uso de GNU Make.

Se ha obtado por esta configuración para facilitar la generación de las librerías en el taller sin centrarse demasiado en los aspectos de la compilación. También se requiere tener instalado Visual Studio 2005 para la generación de los binarios en el entorno Windows cuando se hace uso de las *externals tools* desde eclipse.

2.4. Estructura de los Proyectos

Al estar haciendo uso de **Maven** como sistema de construcción y gestión de dependencias, la estructura que se ha optado es la de un multiproyecto de **Maven**, donde en el directorio raíz se ha creado un *pom.xml* que contiene la configuración general de

los subproyectos, además de propagar los *targets* con los que se ejecuta en el directorio raíz.

Por ejemplo, en el *external tool* que se ha configurado para generar los proyectos de eclipse (*mvn eclipse*), se ha puesto como directorio de trabajo ${workspace_loc}, es decir, que se va a ejecutar *mvn eclipse:eclipse* en el directorio raíz del de los fuentes, donde se encuentra el *pom.xml* raíz.

Los proyectos de los que se compone el taller son: *jni-example*, *libjni-osg* y finalmente *osg-examples*.

jni-example: Ejemplo básico a modo de tutorial.

libjni-osg: Envoltura jni de unas pocas clases y métodos de OpenSceneGraph que hay que mejorar y ampliar.

osg-examples: Proyecto únicamente Java, que contiene un ejemplo básico para validar la librería *libjni-osg*.

2.4.1. Estructura de los Proyectos JNI.

La estructura de los proyectos JNI que se ha seguido es la siguiente:

- Ficheros de configuración de **Maven** (pom.xml) en el directorio raíz del proyecto.
- Ficheros de configuración de *CMake* (CMakeLists.txt) en el directorio raíz del proyecto y en el/los directorio/s de archivos nativos.
  - Otros ficheros de configuración se encuentran en el directorio CMakeModules del directorio raíz del proyecto
- Archivos y paquetes Java (src/main/java)
- Archivos Nativos (src/main/native)
  - Si hay múltiples librerías existirá un directorio por cada una de ellas, con su correspondiente fichero CMakeLists.txt
- Tests Unitarios (src/test/java)

2.5. Generación del Producto Final

Para facilitar la portabilidad del producto y los ejercicios del taller, al ejecutar *mvn install*, se generan y copian en el directorio *product* todos los archivos necesarios para poder ejecutar los ejemplos sin necesidad de hacer uso de Eclipse.

En el directorio *product* se encuentran los siguientes ficheros:

- **run-examples.(bat/sh):** Permiten lanzar los ejemplos tanto de libjni-osg, como de jni-example para ver los resultados.

Archivos generados al finalizar la instrucción *mvn install*:

- **lib:** Directorio donde se encuentran los jars generados:
  - lib/jni-example-1.0-SNAPSHOT.jar
  - lib/libjni-osg-1.0-SNAPSHOT.jar
  - lib/osg-examples-1.0-SNAPSHOT.jar
- **binaries:** Directorio donde se han generado los binarios nativos JNI. Según la plataforma se encontrarán en un directorio diferente (mac, linux32 o win32).

3

Nociones Básicas de *Java Native Interface* (JNI)

JNI define la manera en que se han de realizar las llamadas y cómo debe ser el nombre de las funciones natives, de manera que la máquina virtual sea capaz de localizar e invocar los métodos nativos.

Este capitulo explica como usar JNI en programas escrito en el lenguaje Java para llamar cualquier librería en la máquina local. Para mostrar como utilizar JNI se va a usar un pequeño ejemplo de integración de una librería C/C++.

3.1. Ejemplo JNI

Para llamar a una función de una librería nativa desde Java, hay que tener en cuenta tres cosas:

- Definir un metodo en una clase Java usando la palabra reservada **native**.
- Cargar desde Java la librería nativa, haciendo uso de **System.loadLibrary(libname)**
- Llamar al método definido como nativo.

A continuación se muestra un ejemplo muy sencillo para cargar y ejecutar una llamada a una librería nativa ya creada

```
package es.uji.jornadas;

public class App
{
   static {
       System.loadLibrary("jni-example");
   }
   public native void helloWord();
   public static void main( String[] args )
   {
       App app=new App();
       app.helloWord();
   }
}
```

3.1.1. Declaración del método nativo

La declaración nativa provee un puente para ejecutar funciones nativas en la maquina virtual de Java. En este ejemplo el método *helloWorld* enlaza con una función en C llamada *Java_es_uji_jornadas_App_helloWorld*.

3.1.2. Carga de la librería

La librería que contiene el código nativo se carga llamando el método *System.loadLibrary()*. Colocar esta llamada en la parte estática de la clase asegura que la librería se cargará antes de llamar al constructor y sólo se realizará una única vez.

```
static {
    System.loadLibrary("jni-example");
}
```

3.2. Generar Archivo de Cabecera

El JDK ofrece la utilidad *javah* para generar el código C en una fichero cabecera *.h* con los nombres de las funciones que la máquina virtual buscará al llamar a los métodos declarados como nativos en Java

Al ejectuar la siguiente instrucción sobre el *.class* generado por eclipse se obtendría el fichero de cabecera *.h* con el nombre de la función y los parametros necesarios.

```
javah es.uji.jornadas.App
```

Dado que se está usando el entorno de programación eclipse se va a crear una *external tool* que haga el trabajo sin tener que recurrir a la linea de comandos. Los datos que habría que poner son los siguientes:

- **Cualquier SO:**
  - Name: javah
- **Windows XP:**
  - Location: C:\Archivos de programa\Java\jdk1.6.0_05\bin\javah
  - **Working Directory: ${project_loc}\target\classes (en el caso de estar** usando maven para generar el proyecto) o ${project_loc}\bin (en el caso de usar eclipse sin maven).
  - Arguments: -d ${project_loc}\src\main\native es.uji.jornadas.App
- **Ubuntu Linux:**
  - Location: /usr/lib/jvm/java-6-sun/javah
  - **Working Directory: ${project_loc}/target/classes (en el caso de estar** usando maven para generar el proyecto) o ${project_loc}/bin (en el caso de usar eclipse sin maven).
  - Arguments: -d ${project_loc}/src/main/native es.uji.jornadas.App

Tras ejecutar la *external tool javah*, el fichero de cabecera se generará en el directorio *src/main/native* con el nombre *es_uji_jornadas_App.h*

3.3. Implementar Método Nátivo

Una vez generada correctamente la cabecera hay que implementar la parte nativa. Para eso, creamos un archivo llamado *es_uji_jornadas_App.cpp* en el directorio *src/main/native*. El código se muestra a continuación:

```
void JNICALL Java_es_uji_jornadas_App_helloWord
                (JNIEnv *env, jobject obj) {
    std::cout << "Hello World" << std::endl;
}
```

3.3.1. Compilación librería nativa

Para compilar se va a hacer uso de maven, ant y cmake. Para ello se debe crear una *external tool* para que el sistema de construcción del ejemplo haga el trabajo. La configuración es la siguiente:

Name: mvn project install

Location: ${workspace_loc}/maven/bin/mvn

Working Directory: ${project_loc}

Arguments: mvn install

Atención: Es necesario tener seleccionado el proyecto *jni-example* para poder ejectuar el *external tool* configurado.

3.4. Ejecutar Ejemplo

Para ejecutar el ejemplo, tan solo hay que crear un lanzador de eclipse, para ello pulsar el boton derecho del ratón sobre el fichero *App.java* del proyecto *jni-example*. Seleccionar **Run As** y a continuación **Java Application**.

Tras la ejecución obtendréis el siguiente error:

Exception in thread 'main' java.lang.UnsatisfiedLinkError: no jni-example in java.library.path

La excepción lanzada por la máquina virtual indica que no ha sido posible encontrar la librería nativa jni-example. El problema con JNI es que **para que la máquina virtual encuentre la librería es necesario configurar la ruta de búsqueda de librerías del sistema operativo**.

Para ello ir al menú *Run* de eclipse, y seleccionar *Open Run Dialog*. A continuación dentro de la opción *Java Application* elegir el lanzador de la aplicación *App* y añadir en la pestaña *Environment* la siguiente variable (según el sistema operativo):

Windows: PATH=${workspace_loc}/product/binaries/win32

Linux: LD_LIBRARY_PATH=${workspace_loc}/product/binaries/linux32

MacOsX: DYLD_LIBRARY_PATH=${workspace_loc}/product/binaries/mac

Al finalizar, volver a ejecutar. El resultado debería ser el siguiente:

```
Hello World
```

4

Trabajo a realizar

El taller consta de tres ejercicios, cada uno con dificultades añadidas. El primero es la continuación del ejercicio ejemplo donde se pide ampliar con un método nátivo muy sencillo. El segundo pasa a trabajar directamente con OpenSceneGraph mediante el uso de JNI, donde se pide la posibilidad de cargar escenas 3D en Java. Finalmente se pide la realización de un visualizador de escenas 3D, creando la envoltura de las funciones mínimas necesarias.

4.1. Ejercicio 1: Ampliación de Ejemplo-JNI

Para coger un poco de soltura en la creación de métodos nativos mediante JNI, se propone ampliar el proyecto *jni-example*.

Se propone ampliar la clase *App*, con un nuevo método nativo:

```
public native boolean testValue(int value);
```

De manera que desde C se compruebe si el valor es igual a un *valor* cualquiera elegido. Si el valor es igual, la función debe devolver *true*, en caso contrario debe devolver *false*.

Los pasos a seguir:

1. Escribir método con el modificador nativo en la clase App.
2. Modificar *main* en *App.java*, para comprobar y llamar al método *testValue*
3. Ejecutar *javah* sobre el proyecto.
4. Copiar el nombre de la nueva función generada al archivo *.cpp*.
5. Ejecutar *mvn install* y comprobar que no existan errores de compilación.
6. Ejectuar el ejemplo de nuevo.

4.2. Ejercicio 2: JNI de la función para carga de escenas 3D

En este ejercicio se propone ampliar la librería JNI *libjni-osg* para permitir cargar escenas en 3D. Si se han seguido las instrucciones para preparar correctamente el entorno, la librería *libjni-osg* se habrá compilado correctamente.

Tal como se explica en la *Introducción a OpenSceneGraph*, OSG se compone de una librería principal *osg core*, y de otras que añaden otro tipo de funcionalidad. Para la realización del ejercicio, es necesario exportar el método:

```
osg::Node *osgDB::readNodeFile(const std::string &fileName);
```

El procedimiento a seguir es similar al ejercicio anterior, pero en este caso el fichero java a modificar es *osgDB.java* que se encuentra dentro del paquete *org.openscenegraph.osgDB* en el proyecto *libjni-osg*.

Atención: Para no tener que estar ejecutando *javah* cada vez y modificando los argumentos para que se genere para la clase adecuada, se puede copiar el nombre de otro método y cambiar sólo lo necesario para que la máquina virtual encuentre la función.

Ejemplo:

En *JNIosgDB.cpp* se encuentra la función para escribir a disco escenas 3D:

```
JNIEXPORT void JNICALL
Java_org_openscenegraph_osgDB_osgDB_native_1writeNodeFile
(JNIEnv *env, jclass, jlong nodeCptr, jstring filename)
{
   osg::Node *node = reinterpret_cast<osg::Node*>(nodeCptr);
   if(node == NULL) return;
   osgDB::writeNodeFile(*node, jstring2string(env,filename));
}
```

Si os fijais en el nombre se puede observar lo siguiente:

1. Todos comienzan con el nombre *Java*
2. A continuación y separado por _ está el nombre del paquete al que pertenece la clase.
3. Después el nombre de la clase.
4. Finalmente el nombre del método nátivo. **Nótese que detrás de native_, se escribe un 1 que únicamente sirve para determinar que el _ es parte del nombre del método**

Por tanto para la nueva función requerida

```
public Node native_readNodeFile(String fileName);
```

La función que debe ir en el archivo *JNIosgDB.cpp* sería:

```
EXPORT jlong JNICALL
Java_org_openscenegraph_osgDB_osgDB_native_1readNodeFile
(JNIEnv *env, jclass, jstring filename)
{
    osg::Node *node =
       osgDB::readNodeFile(jstring2string(env,filename));
    if(node == NULL) return 0;
    node->ref();
    return reinterpret_cast<jlong>(node);
}
```

Para validar el ejercicio podemos hacer uso del proyecto *osg-examples*, que define un par de archivos.

- **TestOsgDB.java:** Carga una escena de disco e imprime el nombre de la escena. (*getName()* del nodo raíz).
- **TestOsgViewer.java:** Carga una escena de disco, configura un *Viewer* y lo ejecuta mostrando la escena 3D.

Para la correcta ejecución se debe definir correctamente la variable de búsqueda de librerías nativas en la pestaña *Environment*:

- Windows XP:

  PATH=${workspace_loc}\product\binaries\win32;${workspace_loc}\osg\bin
- Linux:

  LD_LIBRARY_PATH=${workspace_loc}/product/binaries/linux32:${workspace_loc}/osg/lib

4.3. Ejercicio 3: JNI de un Visualizador de escenas 3D

Este ejercicio consiste en exportar las funciones mínimas necesarias para la creación de una ventana de OpenGL, haciendo uso de la librería *osgViewer* de *OpenSceneGraph*, y por supuesto establecer una escena previamente cargada.

Los métodos de viewer que se requieren para poder ejecutar el ejemplo son:

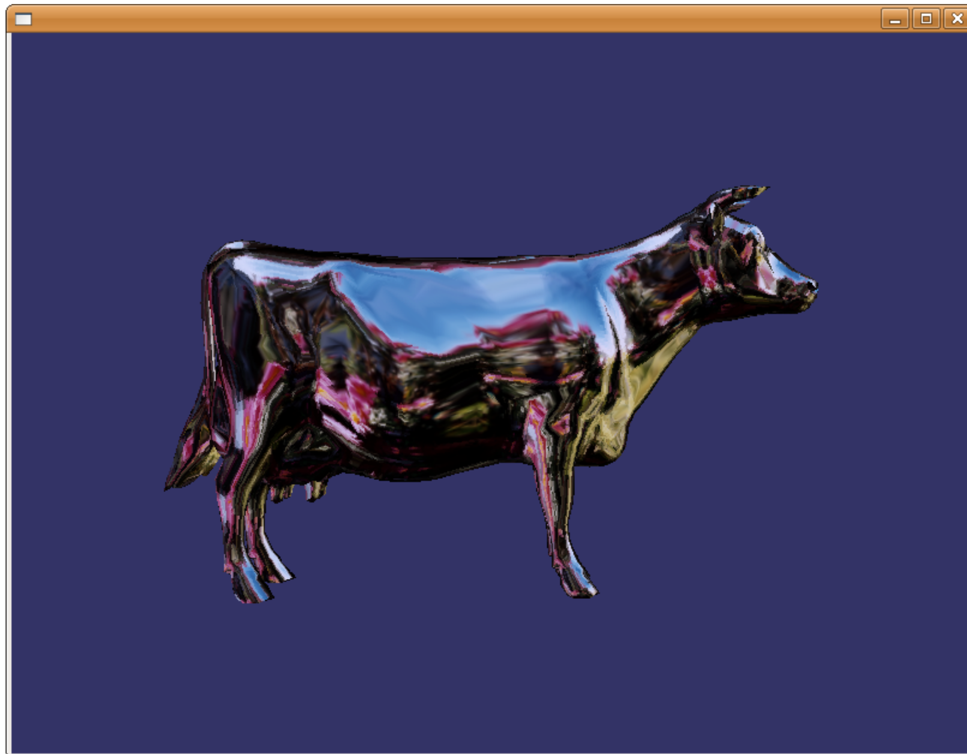
```
private native void
  native_setSceneData(long cptr, long cptrSceneData);

private native long
  native_getSceneData(long cptr);

private native void
  native_setUpViewInWindow(long cptr, int x, int y,
               int width, int height, int screenNum);
```

```
private native void
  native_run(long cptr);
```

Una vez finalizado ejecutar **TestOsgViewer.java** para probarlo. Si todo funciona correctamente el resultado debería ser el siguiente:

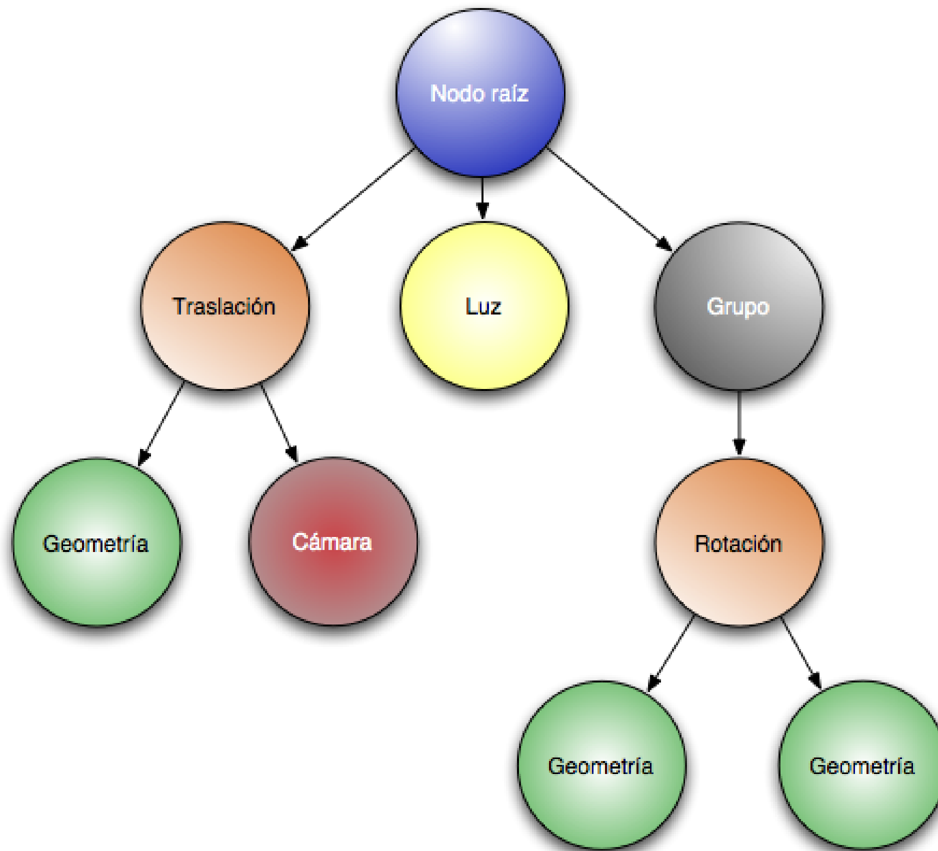


5

Introducción a OpenSceneGraph

Un grafo de escena es un concepto tan sencillo como un *grafo acíclico dirigido* (DAG). Comienza con un nodo raíz, que contiene todo el mundo virtual, ya sea 2D o 3D. El mundo se distribuye en una jerarquÌa de nodos que representan las agrupaciones espaciales, los ajustes de la posición, las animaciones de los objetos o las definiciones de las relaciones lógicas entre ellos. Las hojas del grafo representan los objetos fÌsicos, es decir las geometrÌas dibujables en caso de una escena tridimensional y sus caracterÌsticas materiales. Un nodo del grafo puede tener varios *padres*.

Un grafo de escena no es la completa definición de un motor de videojuegos o de simulación, tan solo representa la escena a visualizar del mismo y puede formar una parte importante de dichos sistemas. El hecho de que no esté todo integrado en el grafo, permite la interoperabilidad con otros componentes, siendo una potente herramienta para distintas tareas. Este factor lo hace un importante elemento de nuestro sistema servidor, ya que nos **permite representar cualquier escena e implementar la mayorÌa de los algoritmos conocidos de aceleración de la visibilidad**.



5.1. Beneficios del Uso de un Grafo de Escena

Las razones clave por las que en muchos desarrollos gráficos se hace uso de un grafo de escena son la **Eficiencia**, **Productividad**, **Portabilidad** y **Escalabilidad**:

Eficiencia: Los grafos de escena proveen un excelente entorno de trabajo para aumentar la eficiencia. Un buen grafo de escena emplea dos técnicas clave. *cálculo de la visibilidad para objetos que no son visibles*, y un *estado ordenado de propiedades* como son texturas, materiales de los objetos, luces, transformaciones y cámaras.

Productividad: Quitan mucho trabajo en el desarrollo de aplicaciones gráficas de alto rendimiento. Uno de los conceptos mas importantes en la programación orientada objetos es la composición de objetos (objetos que contienen otros objetos), que encaja perfectamente en la estructura de **DAG** de un grafo de escena lo que hace que sea un

diseño altamente flexible y reusable. En pocas palabras permite adaptarse fácilmente para resolver nuestro problemas.

Portabilidad: Encapsulan la mayoría de las tareas de dibujado gráfico de bajo nivel. Si éstas son portables, el uso del grafo de escena en otra plataforma es tan sencillo como recompilar el código fuente.

Escalabilidad: Junto con el hecho de que los grafos de escena pueden manejar escenas de gran complejidad, también facilitan la tarea de manejar complejas configuraciones hardware, como son clusters, o sistemas multiprocesador.

5.2. Grafo de Escena OpenSceneGraph

OpenSceneGraph es una herramienta de software libre para el desarrollo de aplicaciones gráficas de alto rendimiento como son simuladores de vuelo, juegos, realidad virtual y visualización científica. Basado en el concepto de grafo de escena, provee un entorno de trabajo orientado a objetos por encima de OpenGL liberando al desarrollador la implementación y optimización de llamadas gráficas de bajo nivel, además de añadir multitud de utilidades para el rápido desarrollo de aplicaciones gráficas.

El propósito de OpenSceneGraph es llevar libremente los beneficios del uso de la tecnología de un grafo de escena a todo el mundo, tanto para propositos comerciales como no comerciales. Se encuentra escrito completamente en C++ y OpenGL, haciendo uso intensivo de la *STL* y de *Patrones de Diseño*.

Los puntos clave de OpenSceneGraph son su rendimiento, escalabilidad, portabilidad y la productividad todas asociadas al hecho de usar un completo grafo de escena.

OpenSceneGraph está dividido en varios componentes, de los cuales el único básico es *osg core*, el resto añaden funcionalidad externa, las más relevantes se muestran a continuación:

- **osgDB:** Carga de escenas y manejo de datos.
- **osgUtil:** Funciones de utilidad para el procesado y modificación de la escena y cálculo de la visibilidad
- **osgViewer:** Librería para la inicialización y manejo de ventanas gráficas de manera independiente de la plataforma.

6

Nociones Avanzadas de JNI

Este capítulo trata de servir como referencia para aspectos más avanzados en el uso de JNI en el desarrollo de aplicaciónes Java.

Primero se explica como enlazan los tipos básicos de Java con los de C mediante JNI, a continuación se da una breve referencia de como se deben de crear envolturas JNI de funciones C y de C++. Finalmente se tratan temas como la instanciación de clases Java desde JNI y como se puede efectuar la llamada a métodos Java desde C mediante el uso de la API de JNI.

6.1. Tipos de Datos

Los métodos declarados como nátivos en Java, pueden contener tanto parametros de entrada como valores de retorno. Los tipos de los datos, pueden ser tan variables como lo puedan ser en Java. Por lo que en JNI se tiene un convenio de conversión y de tipos básicos que se muestra en la tabla a continuación:

6.1.1. Tipos Básicos

Tipo Java	Tipo Nativo	Requiere Conversión de Datos
boolean	jboolean	Sí
byte	jbyte	No
char	jchar	No
short	jshort	No
int	jint	No
long	jlong	No
float	jfloat	No
double	jdouble	No
void	void	No
String	jstring	Sí
Object	jobject	No

Aunque el String sea en realidad un object, dado que es un tipo bastante fundamental en Java se ha creado un tipo especial *jstring*, para poder procesar la información de la cadena.

Para poder obtener la información de un *jstring* es necesario hacer uso de la API de JNI para extraer la cadena de carácteres. A continuación se muestran un par de funciones para convertir los datos de un *jstring* a un std::string de C++ y viceversa.

```
# include <jni.h>
# include <string>
# include <iostream>

static std::string
jstring2string(JNIEnv *env, jstring jstr) {
   const char *str;
   str = env->GetStringUTFChars(jstr, NULL);
   if (str == NULL) {
       return ""; /* OutOfMemoryError already thrown */
```

```
    }
    std::string ret(str);
    env->ReleaseStringUTFChars(jstr, str);
    return ret;
}

static jstring
string2jstring(JNIEnv *env, std::string str) {
    jstring jstr = env->NewStringUTF(str.c_str());
    return jstr;
}
```

El caso del tipo de datos *jboolean*, es necesario comprobar con las definiciones *JNI_TRUE* o *JNI_FALSE*, para obtener el valor correcto. A continuación se muestra un ejemplo de como se podría convertir a *bool* de C++:

```
bool cValue = (javaBoolean==JNI_TRUE)?true:false;
jboolean javaBoolean = cValue?JNI_TRUE:false;
```

6.1.2. Arrays de Tipos Básicos

Para tipos más complejos como pueden ser los arrays se sigue el siguiente convenio:

Tipo Java	Tipo Nativo
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
short[]	jshortArray
int[]	jintArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Object[]	jobjectArray

Para poder trabajar con los valores del Array, JNI ofrece una extensa API, tanto para crearlos, obtener valores y establecer valores en los elementos de un array.

6.2. JNI de clases C++

Uno de los problemas de JNI es que no facilita la conexión con objetos en C++. En la mayoria de los casos al hacer uso de librerías en C, no hay que conservar el estado de la aplicación, pero con orientación a objetos es diferente. Hay que conservar la referencia al objeto y hay que modificar su estado mediante uso de la interfaz que proveen.

Supongamos que tenemos en C++ la siguiente clase:

```
class Test {
    private:
        int _a;
    public:
        Test();
        ~Test();
        void setA(int valor);
        int getA();
};
```

Para poder hacer uso de la clase *Test* desde Java, es necesario conservar en Java una referencia al objeto nativo. Para ello hacemos uso de un tipo *long* para guardar el valor del puntero del objeto nativo creado.

Para simular el mismo proceso de vida del objeto nativo en java, es fundamental crear llamadas nativas para crear el objeto (Constructor), para destruirlo (finalize), y para cada uno de los métodos se tiene como primer parámetro el puntero nativo para poder realizar el casting adecuado en la parte nativa, con excepción del constructor que es el que lo crea (native_createTest()).

A continuación se muestra un ejemplo de la envoltura en Java de la clase *Test*.

```
public class Test {
    private native long native_createTest();
```

```
    private native void native_disposeTest(long cptr);
    private native void native_setA(long cptr, int value);
    private native int native_getA(long cptr);

    static {
        System.loadLibrary("jni-test");
    }
    private long _cptr;
    public Test() {
        _cptr = native_createTest();
    }
    public void setA(int valor) {
         native_setA(_cptr,valor);
    }
    public int getA() {
        return native_getA(_cptr);
    }
    protected void finalize() throws Throwable {
        native_disposeTest(_cptr);
        super.finalize();
    }
}
```

la librería JNI debera compilar un archivo que exporte las funciones adecuadas para poder llamar a los métodos de la clase *Test*. A continuación se muestra el fichero *JNITest.cpp*:

```
#include <jni.h>
#ifndef _Included_Test
#define _Included_Test
#ifdef __cplusplus
extern "C"{
#endif
JNIEXPORT jlong JNICALL Java_Test_native_1createTest
  (JNIEnv *, jobject)
{
   Test *test=new Test();
```

```
   return reinterpret_cast<jlong>(test);
}

JNIEXPORT void JNICALL Java_Test_native_1disposeTest
  (JNIEnv *, jobject, jlong cptr)
{
   Test *test=reinterpret_cast<Test*>(cptr);
   delete test;
}

JNIEXPORT void JNICALL Java_Test_native_1setA
  (JNIEnv *, jobject, jlong cptr, jint value)
{
   Test *test=reinterpret_cast<Test*>(cptr);
   test->setA(value);
}

JNIEXPORT jint JNICALL Java_Test_native_1getA
  (JNIEnv *, jobject, jlong cptr)
{
   Test *test=reinterpret_cast<Test*>(cptr);
   return test->getA();
}

#ifdef __cplusplus
}
#endif
#endif
```