

Prácticas de Informática Gráfica

Grado en Informática y Matemáticas.

Grado en Informática y Administración y Dirección de Empresas
Curso 2024-25.



**UNIVERSIDAD
DE GRANADA**

ETSI Informática y de Telecomunicación.
Departamento de Lenguajes y Sistemas Informáticos.

Índice general.

Índice.	3
0. Prerrequisitos, materiales y compilación	9
0.1. Prerrequisitos software	9
0.1.1. Linux	9
0.1.2. macOS	9
0.1.3. Windows	10
0.2. Materiales y compilación	11
0.2.1. Materiales para las prácticas. Entregas.	11
0.2.2. Compilación en la línea de órdenes	12
0.2.3. Edición y compilación con <i>VS Code</i>	14
0.3. La librería auxiliar GLM	14
0.3.1. Tuplas de valores enteros o reales con 2, 3 o 4 valores	14
0.3.2. Matrices de reales	16
1. Visualización de modelos simples	19
1.1. Objetivos	19
1.2. Desarrollo	19
1.3. Teclas a usar. Interacción.	19
1.4. Estructura del código: clases, métodos y funciones.	20
1.4.1. Cauce, modos de visualización y de envío	20
1.4.2. Clases para descriptores de VBOs y VAOs (DescrVBOAtribs , DescrVBOInds y DescrVAO).	21
1.4.3. Clase abstracta para objetos gráficos 3D (Objeto3D)	24
1.4.4. Clase para mallas indexadas (MallaInd)	24
1.4.5. La clase Escena	26
1.4.6. Uso de la instancia de la clase Cauce	27
1.4.7. Clases para los objetos de la práctica 1	27

1.5.	Tareas	27
1.5.1.	En el archivo vaos-vbos.cpp	28
1.5.2.	En el archivo malla-ind.cpp	29
1.5.3.	En el archivo escena.cpp	29
1.5.4.	Clases nuevas para otros tipos de mallas indexadas	30
1.6.	Ejercicios adicionales	30
1.6.1.	Ejercicio 1	30
1.6.2.	Ejercicio 2	32
1.6.3.	Ejercicio 3	32
2.	Modelos PLY y Poligonales	33
2.1.	Objetivos	33
2.2.	Desarrollo	33
2.3.	Tareas	35
2.4.	Algoritmo para la creación de malla por revolución	36
2.5.	Estructura del código. Clases.	37
2.5.1.	La clase Escena2	38
2.5.2.	Clase MallaPLY : mallas creadas a partir de un archivo PLY.	38
2.5.3.	Clase MallaRevol : mallas obtenidas por revolución de un perfil.	38
2.5.4.	Clase MallaRevolPLY : revolución de un perfil leído en un PLY	39
2.5.5.	Clase: Cilindro, Cono, Esfera	39
2.5.6.	Archivos PLY disponibles.	40
2.6.	Ejercicios adicionales	40
2.6.1.	Ejercicio 1	40
2.6.2.	Ejercicio 2	41
2.6.3.	Ejercicio 3	41
3.	Modelos jerárquicos	43
3.1.	Objetivos	43
3.2.	Desarrollo.	43
3.3.	Tareas.	44

3.4. Teclas a usar. Gestión de animaciones	44
3.5. Visualización y gestión de nodos del grafo (clase NodoGrafoEscena)	45
3.5.1. El método visualizarGL	45
3.5.2. El método visualizarGeomGL	46
3.5.3. El método agregar	46
3.5.4. El método leerPtrMatriz	47
3.6. Implementación de parámetros y animaciones	47
3.6.1. Definición de clases para objetos parametrizados	48
3.6.2. Definición de nodos del grafo de escena parametrizados	49
3.7. Diseño e implementación de un grafo de escena parametrizado original	49
3.7.1. Contenidos del archivo PDF con la documentación	50
3.7.2. La clase Escena3	51
3.8. Algunos ejemplos de modelos jerárquicos	51
3.9. Ejercicios adicionales	51
3.9.1. Ejercicio 1	51
3.9.2. Ejercicio 2	52
4. Materiales, fuentes de luz y texturas	55
4.1. Objetivos	55
4.2. Desarrollo	55
4.3. Tareas	56
4.4. Teclas a usar	56
4.5. Clases y métodos a añadir o completar.	57
4.5.1. Las clases FuentesLuz y ColFuentesLuz	57
4.5.2. Clase Textura	58
4.5.3. Clase Material	59
4.5.4. Añadidos a la clase NodoGrafoEscena	59
4.5.5. Añadidos a la clase MallaIndy y derivadas	59
4.5.6. Añadidos a la clase Escena	60
4.5.7. Visualización de normales.	61
4.5.8. Clase Escena4	62

4.5.9. Clase LataPeones	63
4.5.10. Materiales y textura en el grafo de la práctica 3	64
4.6. Cálculo de tablas de normales y coordenadas de textura	65
4.6.1. Clases Cubo24 y NodoCubo24	65
4.6.2. Clase MallaRevol	66
4.6.3. Clases Cubo , Tetraedro y MallaPLY	68
4.7. Ejercicios adicionales	69
4.7.1. Ejercicio 1	69
4.7.2. Ejercicio 2	71
4.7.3. Ejercicio 3	71
5. Interacción: cámaras y selección.	73
5.1. Objetivos	73
5.2. Desarrollo	73
5.2.1. Gestión de cámaras	73
5.2.2. Selección de objetos.	73
5.2.3. Documento del grafo es escena con identificadores de selección.	74
5.3. Tareas.	74
5.4. Teclas e interacción con el ratón.	75
5.5. Escena de la práctica 5. La clase VariasLatasPeones	75
5.6. Extensión del grafo original. Documentación.	76
5.7. Gestión interactiva de cámaras de 3 modos.	77
5.7.1. Uso del teclado	77
5.7.2. Uso del ratón	78
5.7.3. Código de actualización de la cámara actual	78
5.7.4. Apuntar la cámara al objeto al que se hace click.	79
5.8. Selección.	79
5.8.1. El método cuandoClick de la clase Objeto3D	80
5.8.2. Visualización de mallas y nodos en modo selección	80
5.8.3. Visualización de la escena en modo de selección	81
5.8.4. Gestión de clicks en el método seleccion de AplicacionIG	81

5.8.5. Búsqueda recursiva de un identificador y cálculo del centro.	82
5.9. Ejercicios adicionales	83
5.9.1. Ejercicio 1	83
5.9.2. Ejercicio 2	85

Prerequisitos, materiales y compilación.

En esta sección se detallan las herramientas software necesarias para realizar las prácticas en los sistemas operativos Linux (Ubuntu u otros), macOS (de Apple) y Windows (de Microsoft), así como los recursos o materiales que se proporcionan para las prácticas y la forma de compilar el código.

0.1. Prerequisitos software

En esta sección se indican las componentes software (compiladores y librerías) que deben usarse para compilar las prácticas en Linux, macOS o Windows. Se puede elegir cualquiera de los tres sistemas operativos para trabajar en función de las preferencias o disponibilidad del estudiante. El código fuente que se debe escribir, así como el funcionamiento de las prácticas, no dependen en absoluto de dicho sistema operativo.

Los compiladores que se deben usar son el compilador de C/C++ de GNU (o LLVM) en Linux, o bien los compiladores de C/C++ que se incorporan en XCode (en macOS) o en *Visual Studio* (en Windows). Las librerías que usaremos son GLEW, GLFW, GLM y JPEG (aunque GLEW no se usa en macOS).

Para la edición de los fuentes se puede usar *VSCode* de Microsoft. Se proporcionan archivos de tipo `.workspace` que facilitan el uso de *VSCode* para las tareas de edición, incluyendo el subrayado automático de errores.

0.1.1. Linux

Se puede usar el instalador de paquetes `apt` para la instalación de las herramientas y librerías necesarias.

En Linux es necesario tener instalado el compilador de C++ de GNU o del proyecto LLVM, esto permite invocar la orden `g++` y la orden `make`. Si no se tienen disponibles, estas herramientas se pueden instalar con la orden:

```
sudo apt install build-essential
```

Se debe usar `apt` para instalar `cmake`, que se usa para poder compilar desde la línea de órdenes, se hace con:

```
sudo apt install cmake
```

Finalmente se deben instalar los paquetes para las librerías GLEW, GLFW (version 3), GLM y JPEG, se puede hacer con:

```
sudo apt install libglew-dev libglfw3-dev libglm-dev libjpeg-dev
```

0.1.2. macOS

En ordenadores macOS hay que tener instalada la herramienta de desarrollo de *XCode*, disponible aquí:

developer.apple.com/xcode

Esta herramienta de desarrollo incorpora (entre otros) el compilador de C++ del proyecto LLVM adaptado por Apple, el IDE de desarrollo para Apple, así como el *framework* de OpenGL.

Una vez instalado XCode (y si no se ha hecho durante la instalación) es necesario instalar un componente adicional de XCode llamado Command line Tools (CLT), se puede hacer desde la propia línea de órdenes con:

```
xcode-select --install
```

Además de XCode, también debemos de usar el instalador de paquetes open source *Homebrew*

 brew.sh

Para instalarlo se deben seguir las instrucciones que podemos encontrar en esa página Web.

La librería OpenGL ya viene instalada con XCode, así que únicamente hará falta instalar la orden **cmake** y las librerías GLFW, GLM y JPEG. Estos paquetes se pueden instalar fácilmente con *Homebrew*, usando:

```
brew install cmake  
brew install glfw  
brew install glm  
brew install jpeg
```

0.1.3. Windows

En Windows hay que instalar *Microsoft Visual Studio*, disponible aquí:

visualstudio.microsoft.com

Es un entorno de desarrollo integrado (IDE) y una suite con compiladores e intérpretes para varios lenguajes de programación.

Este entorno de desarrollo incluye numerosos componentes para sus distintos lenguajes de programación. Para estas prácticas únicamente hay que instalar los componentes para desarrollo de aplicaciones de escritorio con C y C++. Este software incluye tanto **cmake** como **git**.

Para hacer la instalación de las librerías necesarias así como para compilar se debe usar una terminal de Windows, de tipo *Developer Powershell for VS*.

Los archivos de compilación están preparados para compilar en la línea de órdenes usando librerías instaladas con *vcpkg*, que es un instalador (de código abierto) de paquetes con librerías de C/C++ de Microsoft, disponible aquí:

 vcpkg.io

Para instalar vcpkg debes hacer **cd** a tu carpeta *home* (es decir, la carpeta **C:/Users/usuario**) y una vez en ella clonar el repositorio de vcpkg con:

```
git clone https://github.com/Microsoft/vcpkg.git
```

Si todo va bien se crea una carpeta de nombre **vcpkg** dentro de tu carpeta *home*. Para finalizar la instalación debes hacer **cd** a tu carpeta *home*, y ahí ejecutar:

```
.\vcppkg\bootstrap-vcppkg.bat
```

En la carpeta **vcppkg** quedará el archivo ejecutable **vcppkg.exe**, que se puede ejecutar directamente desde la línea de órdenes. En windows debemos de instalar las librerías GLEW y GLFW, se puede hacer cd a la carpeta vcppkg y ejecutar

```
.\vcppkg install glew --triplet x64-windows  
.vcppkg install glfw3 --triplet x64-windows  
.vcppkg install glm --triplet x64-windows  
.vcppkg install libjpeg-turbo --triplet x64-windows
```

El switch **--triplet** indica que se instalen las versiones de 64 bits dinámicas de estas librerías. La instalación de GLEW conlleva la instalación de la librería OpenGL.

0.2. Materiales y compilación

0.2.1. Materiales para las prácticas. Entregas.

Los archivos que se proporcionan se encuentran organizados en estas carpetas y sub-carpetas:

- **materiales**: esta carpeta contiene archivos de código fuente, imágenes, modelos 3D (archivos **.ply**) y otros, que se deben usar tal cual se entregan, sin que el alumno deba modificar ninguno de ellos. Tiene estas sub-carpetas:
 - **src-cpp**: archivos de código fuente C++ (**.cpp** y **.h**)
 - **src-shaders**: archivos de código fuente GLSL (**.glsl**)
 - **plys**: archivos con modelos 3D en formato PLY (**.ply**)
 - **imgs**: archivos de imágenes para texturas en formato JPEG (**.jpg** o **.jpeg**)
- **src**: esta carpeta contiene código fuente que debe ser completado por el alumno, son archivos **.cpp** y **.h**.
- **builds**: carpeta con archivos de configuración para compilar desde la línea órdenes (con **cmake**) y para editar el código fuente con VS Code de Microsoft. Tiene tres subcarpetas, se debe trabajar en la que corresponda al sistema operativo que usemos:
 - **linux**: para usar en Ubuntu u otros linux.
 - **macos**: para usar en macOS de Apple.
 - **windows**: para Windows de Microsoft.
- Dentro de cada una de estas carpetas se encuentran las subcarpetas **bin** (donde se aloja el archivo ejecutable al compilar) y **cmake** (usada para los archivos intermedios de la compilación). Más abajo se dan detalles del proceso de compilación.
- **archivos-alumno**: archivos recopilados o creados por el alumno, distintos de los archivos fuente en la carpeta **src**. Aquí el alumno debe incluir archivos de imagen o archivos con modelos PLY que él mismo haya recopilado. También debe incluir archivos PDF o imágenes que debe crear y que forman parte de las entregas, según se describen en el guión de prácticas.

Estas carpetas se deben incluir en una carpeta nueva, que llamaremos *carpeta raiz* y cuyo nombre puede ser cualquiera.

La entregas de prácticas consisten en subir los archivos fuente de la carpeta **src** y los archivos de la carpeta **archivos-alumno**, si hay alguno. En ningún caso se deben subir archivos objeto o ejecutables (resultados de la compilación que son específicos del s.o. y arquitectura hardware

usados por el alumno), tampoco ningún archivo que esté en la carpeta **materiales**, ni archivos de configuración de **.vscode**.

0.2.2. Compilación en la línea de órdenes

Se describe aquí el proceso de compilación en la línea de órdenes para macOS, Linux y Windows, en los tres casos usando el programa *CMake*. Se proporcionan los archivos **CMakeLists.txt** necesarios para ello.

El uso de **cmake** requiere crear inicialmente los archivos de configuración de compilación necesarios, una sola vez, o después cuando queremos regenerar dichos archivos por cualquier motivo.

0.2.2.1. Sistemas operativos Linux y macOS

En estos sistemas operativos podemos compilar en la línea de órdenes usando un terminal normal.

Para la generación de los archivos de compilación y la compilación en sí se usa cmake, para ello es necesario ir a la carpeta **builds/macOS** o **builds/linux**, según el sistema operativo. En esa carpeta debemos asegurarnos de que la sub-carpeta **cmake** está vacía (si no lo estaba ya, hay que borrar todos los archivos ahí). Para generar los archivos de compilación, hay que hacer entrar a la carpeta **cmake** y ahí escribir:

```
cmake ..
```

Esto hay que hacerlo una vez, o cada vez que se añadan nuevos fuentes o se quiera cambiar la configuración de compilación. Esto genera diversos archivos y carpetas dentro de la carpeta **cmake**. Después, para compilar los fuentes, hay que ejecutar (en esa misma carpeta):

```
make
```

Si la compilación va bien se genera el ejecutable, que tiene el nombre **debug_exe** y está en la carpeta bin. La orden **make** también se puede usar con un argumento para otros fines:

- **make clean** para eliminar el programa compilado y los archivos asociados.
- **make release_exe** para generar el ejecutable **release_exe** (también en **bin**), el cual no tiene los símbolos de depuración y además está optimizado (es más pequeño y puede que sea más rápido al ejecutarse)

Para forzar un recompilado de todos los fuentes, basta con vaciar la carpeta **cmake** y volver a hacer **cmake ..** en ella. Es necesario hacerlo si se añaden o quitan unidades de compilación o cabeceras de las carpetas con los fuentes.

0.2.2.2. Sistema operativo Windows

En Windows se debe que usar el terminal llamado *Developer PowerShell for VS*, es la aplicación de terminal para *PowerShell* de Microsoft, pero configurada con las variables de entorno necesarias para compilar desde la línea de órdenes.

Estos fuentes se deben compilar en la línea de órdenes con **cmake**, para ello es necesario ir a la carpeta **builds/windows**. En esa carpeta debemos asegurarnos de que la sub-carpeta **cmake**

está vacía. Si no lo estaba ya hay que borrar todos los archivos ahí. Para generar los archivos de compilación, dentro de cmake vacío escribimos:

```
cmake ..
```

Esto hay que hacerlo una vez, o cada vez que se añadan nuevos fuentes o se quiera cambiar la configuración de compilación. Esto genera diversos archivos y carpetas en cmake.

Una vez generados los archivos de compilación, cada vez que queramos recompilar los fuentes hay que ejecutar, en cmake, esta orden:

```
cmake --build .
```

Sila compilación va bien se genera el ejecutable, que tiene el nombre **pracs_ig.exe** y está situado en la sub-carpeta **Debug** dentro de la carpeta **bin**, dicha carpeta también incluye archivos **.dll** (librerías dinámicas de Windows) y un archivo **.pdb** para depuración.

Para forzar un recompilado de todos los fuentes, basta con vaciar la carpeta **cmake**, repetir la orden **cmake ..** en ella y finalmente compilar con **cmake --build .**

En Windows se genera por defecto una versión *Debug* del ejecutable, si se quiere generar una versión *Release*, el paso de compilación debe ser de esta forma:

```
cmake --build . --config Release
```

en este caso, el ejecutable (y sus archivos **.dll**) quedará en la subcarpeta **Release** dentro de **bin** (no se genera el **.pdb**).

Para eliminar el ejecutable de *debug* y todos los archivos intermedios producidos al compilarlo (archivos **.obj**), se puede usar:

```
cmake --build . --target clean
```

Para eliminar el ejecutable de *release* y todos los archivos **.obj** correspondientes:

```
cmake --build . --config release --target clean
```

En Windows, las sentencias **cout** o **printf** de C o C++ que contengan acentos o la eñe producen caracteres extraños en el terminal, ya que el terminal no asume por defecto que las cadenas que se imprimen están codificadas en UTF-8, mientras que los programas fuentes de este repositorio sí están codificadas en ese formato (deben estarlo así). Para solucionar este problema, hay que ejecutar una vez esta orden en el terminal Powershell (es una única línea y se debe incluir el carácter \$ inicial):

```
$OutputEncoding = [console]::InputEncoding = [console]::OutputEncoding =
New-Object System.Text.UTF8Encoding
```

Si no se quiere teclear esto en cada inicio de sesión, se puede añadir esa línea al archivo de script (tipo **.ps1**) que se ejecuta cada vez que se abre Powershell, el nombre completo de dicho archivo está en la variable de entorno **\$profile** de Powershell.

(Nota: todo esto se ha probado probado en Mayo-Junio de 2023 en Windows 11).

0.2.3. Edición y compilación con VS Code

El programa *VS Code* es un editor de código fuente de *Microsoft* gratuito y que puede usarse en Linux, macOS y Windows. Este programa no incorpora un compilador, en su lugar usa alguno que el usuario haya instalado en su ordenador, en nuestro caso usará los compiladores cuya instalación se ha descrito en el apartado anterior. Entre los archivos que se entregan hay archivos de configuración que facilitan la tarea de editar, compilar, ejecutar y depurar el código de prácticas usando *VS Code* y el compilador correspondiente.

La posibilidad de usar *VS Code* no excluye el uso de otros editores (p.ej. *atom*), y en cualquier caso el programa siempre se debe compilar en la línea de órdenes con **cmake**, como se ha explicado.

Para editar el código con *VS Code* se proporcionan (en las carpetas **builds/macOS**, **build/windows** y **builds/linux**) archivos de nombre **workspace.code-workspace** (además de las carpetas de nombre **.vscode**). Para trabajar en el *espacio de trabajo* (*workspace*) de las prácticas se debe abrir el archivo **.code-workspace** (el que corresponda al sistema operativo) con el programa *code*. Al abrir el espacio de trabajo se tiene la posibilidad de editar los códigos fuentes que se encuentran en la carpeta **src**, asimismo podemos ver (y no debemos editar), los fuentes que hay en **materiales**.

Al editar el código ocurre el subrayado automático de errores en cualquiera de los tres los sistemas operativos.

Para generar los archivos de compilación (si no estaban ya) se puede pinchar el sub-menu *Terminal*, dentro de eso la opción *Ejecutar tarea...* y finalmente seleccionar la opción *Generar archivos de compilación..* Esto hay que hacerlo una sola vez al principio, o bien cuando añadamos nuevos archivos a la carpeta **src**.

En el sub-menú *Terminal* hay otras tareas posibles, y además hay una opción llamada *Ejecutar tarea de compilación*. Esta opción permite compilar y ejecutar las prácticas sin abandonar *code*.

Todas las acciones de compilar y ejecutar que realiza *code* se ponen en marcha por dicho programa usando las mismas órdenes que hemos visto para la compilación y ejecución desde la línea de órdenes.

0.3. La librería auxiliar GLM

En esta sección se describen algunos tipos de datos de la librería GLM, que se usarán en las prácticas.

0.3.1. Tuplas de valores enteros o reales con 2, 3 o 4 valores

La librería GLM (namespace **glm**) permite usar varias clases para tuplas de valores numéricicos (reales en simple o doble precisión y enteros con o sin signo). Cada tupla contiene unos pocos valores del mismo tipo (entre 2 y 4). Por cada tipo de valores y cada número de valores hay un tipo de tupla distinto (**vec2**, **vec3**, **dvec2**, **dvec3**, **uvec3**, etc...). Los nombres de los tipos y las operaciones están tomadas de los tipos y operaciones equivalentes del lenguaje GLSL.

Aquí vemos ejemplos de uso de estos tipos:

```
// vec3 y dvec3: tuplas de 3 reales, adecuadas para coordenadas de puntos,  
// vectores o normales en 3D también para colores (R,G,B)  
  
vec3 v1 ; // tuplas de tres valores tipo float
```

```
dvec3 v2 ; // tuplas de tres valores tipo double
// uvec3: tuplas de 3 valores unsigned, adecuadas para triángulos
uvec3 v3 ;
// vec4 y dvec4: adecuadas para puntos o vectores en coordenadas homogéneas
// también para colores (R,G,B,A)
vec4 v5 ; // tuplas de cuatro valores tipo float
dvec4 v6 ; // tuplas de cuatro valores tipo double
// vec2 y dvec2: adecuadas para puntos o vectores en 2D, y coordenadas de textura
vec2 v7 ; // tuplas de dos valores tipo float
dvec2 v8 ; // tuplas de dos valores tipo double
```

Aquí vemos las diversas formas que podemos usar para declarar e inicializar estas tuplas (sintaxis alternativas de C++11)

```
// declaraciones con un valor inicial

vec3 a( 1.0, 2.0, 3.0 ) ;
vec3 b = { 1.0, 2.0, 3.0 } ;
vec3 c = vec3( 1.0, 2.0, 3.0 ) ;

// inicialización a partir de un puntero a un array nativo de C/C++

float arr3f[3] = { 1.0, 2.0, 3.0 } ;
unsigned arr3u[3] = { 1, 2, 3 } ;

vec3 e = make_vec3( arr3f );
uvec3 f = make_vec3( arr3u );
```

Para acceder a una componente de una tupla podemos usar varias alternativas

- La posición o índice de los componentes en la tupla, entre corchetes ([0],[1],[2],...),
- Los campos **.x**, **.y**, **.z** y **.w**
- Los campos **.r**, **.g**, **.b** y **.a**
- Los campos **.p**, **.s**, **.t** y **.q**

Aquí se ilustran estas posibilidades:

```
// accesos de lectura

float
    x1 = a[0], y1 = a[1], z1 = a[2], // acceso con indices (como array)
    x2 = a.x, y2 = a.y, z2 = a.z, // pensado para coordenadas (.x .y .z .w)
    re = c.r, gr = c.g, bl = c.b, // pensado para colores (.r .g .b .a)
    es = e.s, et = e.t, ep = e.p ; // pensado para coord.text (.s .t .p .q)
unsigned
    u0 = f[0], u1 = f[1], u2 = f[2] ;

// accesos de escritura

a[0] = x1 ;
```

```
a.x = x2 ;
c.g = gr ;
```

Podemos obtener un puntero al primer valor en la tupla (los demás están contiguos en memoria), esto es muy útil en funciones de OpenGL que esperan estos punteros:

```
float * p1 = value_ptr(a) ; // conv. a puntero de lectura/escritura
const float * p2 = value_ptr(b) ; // conv. a puntero de solo lectura
```

Las tuplas se pueden convertir a cadenas de caractéres para imprimirlas, por ejemplo:

```
cout << "la tupla 'a' vale: " << to_string(a) << endl ;
```

En C++ se pueden sobrecargar los operadores binarios y unarios usuales (+, -, etc...) para operar sobre las tuplas de valores reales:

```
// operadores binarios y unarios de asignación/suma/resta/negación
// (suponemos que a, b y c son tres tuplas del mismo tipo)
a = b ;
a = b+c ;
a = b-c ;
a = -b ;

// multiplicación y división por un escalar (deben ser tuplas de flotantes)
a = 3.0f*b ;
a = b*4.56f ;
a = b/34.1f ;
a = 3.0f*b +4.0f*c + 7.0f*c ;
```

Finalmente, podemos usar diversas funciones para operar con las tuplas. Aquí **s** y **l** son flotantes:

```
s = dot( a, b ) ; // producto escalar (usando método dot)
a = cross( b, c ) ; // producto vectorial (solo para tuplas de 3 valores)
l = length( a ) ; // calcular módulo o longitud de un vector
a = normalize( b ) ; // obtener copia normalizada (b/length(b))
```

0.3.2. Matrices de reales

GLM permite crear y operar con matrices (cuadradas y rectangulares) con valores **float** o **double**

```
// matrices cuadradas de 2x2, 3x3 o 4x4, de floats o de doubles (6 tipos)

mat2 m2x2f ; // 2x2 floats (nombre alternativo: mat2x2)
mat3 m3x3f ; // 3x3 floats (mat3x3)
mat4 m4x4f ; // 4x4 floats (mat4x4)

dmat2 m2x2d ; // 2x2 doubles (dmat2x2)
dmat3 m3x3d ; // 3x3 doubles (dmat3x3)
dmat4 m4x4d ; // 4x4 doubles (dmat4x4)

// matrices no cuadradas: tipo [d]matmxn, donde:
// - 'm' se sustituye por el número de columnas (2, 3 o 4)
// - 'n' se sustituye por el número de filas (2, 3 o 4)
// - el prefijo 'd' es opcional (si están son doubles)
```

```
mat2x3 m2x3f ; mat2x4 m2x4f ; dmat2x3 m2x3d ; dmat2x4 m2x4d ;
mat3x2 m3x2f ; mat3x4 m3x4f ; dmat3x2 m3x2d ; dmat3x4 m3x4d ;
mat4x2 m4x2f ; mat4x3 m4x3f ; dmat4x2 m4x2d ; dmat4x3 m4x3d ;
```

Se pueden inicializar usando varias sintaxis alternativas de C++. El acceso se hace con los corchetes, indicando primero la columna y luego la fila.

```
// inicialización de las matrices

mat4 m1 ; // valores indeterminados (no usar)
mat4 m4a = mat4( 1.0f ); // diagonal a 1, resto a cero (matriz identidad)
mat4 m4b = mat4( 34.0f ); // diagonal a 34, resto a cero
mat2 m2a = { 1.0f, 2.0f, 3.0f, 4.0f }; // valores dados por columnas
mat2 m2b = { {1.0f, 2.0f}, {3.0f, 4.0f} }; // valores dados por columnas

// accesos a las matrices

cout << "m2b[0][1] == " << m2b[0][1] << endl ; // columna 0, fila 1 -> valor 2.0

m4a[2][3] = 5.0f ; // escribir en celda en columna 2 fila 3 (empezando en 0)
float a = m4b[1][2] ; // leer el valor en columna 1 fila 2

// obtención de un puntero al primer valor (el resto están contiguos, por columnas)
float * p = value_ptr( m4a ) ;
```

Las matrices se pueden multiplicar entre ellas, también se puede multiplicar una matriz por un vector y obtener otro vector:

```
vec4 u, v ;
mat4 m1, m2, m3 ;

// multiplicación de matrices cuadradas entre ellas
// (deben ser de las mismas dimensiones y el mismo tipo)

m1 = m2 * m3 ;
m1 = m2 * m3 * m2 ;

// multiplicación de una matriz cuadrada por un vector (por la derecha)
// (debe ser de dimensiones coherentes y del mismo tipo)

u = m1 * v ;
u = m1 * m2 * v ;
```

La librería GLM da la posibilidad de crear diversos tipos de matrices muy usadas en gráficos

```
// matrices de transformación

float ar, ag ; vec3 v ;

mat4 mt = translate( v ), // traslación por v
        me = scale( v ), // escalado (v en la diagonal prin.)
        mr = rotate( ar, v ), // rotación entorno a v, ángulo ar (en radianes)
        mr1 = rotate( radians(ag), v ); // rotación en v (ángulo ag en grados)
```

```
// matrices de vista y proyección

float fy, a, l, r, t, b, n, f ;
vec3 eye, center, up ;

mat4 mv = lookAt( eye, center, up ), // matriz de vista
      mpo = ortho( l, r, b, t, n, f ), // proyección ortográfica
      mpf = frustum( l, r, b, t, n, f ), // proyección perspectiva
      mpp = perspective( fy, a, n, f ); // proyección perspectiva (v2)
```

1. Visualización de modelos simples.

1.1. Objetivos

Con esta práctica se quiere que el alumno aprenda:

- A crear estructuras de datos que permitan representar objetos 3D sencillos (mallas indexadas)
- A utilizar las órdenes para visualizar secuencias de vértices correspondientes a mallas indexadas.

1.2. Desarrollo

Para el desarrollo de esta práctica se entrega el esqueleto de una aplicación gráfica basada en eventos, mediante GLFW, y con la parte gráfica realizada por OpenGL. Para facilitar su uso, la aplicación permite abrir una ventana, mostrar unos ejes y mover una cámara básica.

El alumno deberá crear, como mínimo, el modelo de un **tetraedro** (el modelo de un **cubo** se da hecho a modo de ejemplo). Para ello, creará las estructuras de datos que permitan representarlo mediante sus vértices y caras (con una estructura de tipo *malla indexada*). Asimismo, escribirá el código necesario para visualizar mallas indexadas en general, usando *Vertex Buffer Objects* (VBOs) y *Vertex Array Objects* (VAOs) en **modo diferido**.

Las mallas se podrán visualizar usando cada uno de los tres **modos de visualización de polígonos**:

- **Modo puntos**: se visualiza un punto en la posición de cada vértice del modelo.
- **Modo alambre**: se visualiza como un segmento cada arista del modelo.
- **Modo sólido**: se visualizan los triángulos llenos todos de un mismo color (plano).

En el modo sólido, además, se podrá activar o desactivar el **visualización de aristas**. Cuando el dibujo de aristas está activado, en el modo sólido se ven los triángulos opacos y además las aristas del modelo sobre ellos (todas de color negro).

1.3. Teclas a usar. Interacción.

El programa permite pulsar las siguientes teclas:

- **tecla p/P**: cambia la escena actual (pasa a la siguiente, o de la última a la primera). Hay una escena por cada práctica (ver más abajo).
- **tecla o/O**: cambia el objeto activo dentro de la escena actual (pasa al siguiente, o del último al primero)
- **tecla m/M**: cambia el modo de visualización de polígonos actual (pasa al siguiente, o del último al primero)
- **tecla w/W**: activa o desactiva la visualización de aristas.
- **tecla i/I**: activa o desactiva la iluminación (no tiene efecto hasta que no se implemente la práctica 4, antes de eso no hay iluminación)
- **tecla f/F**: cambia entre uso de normales de triángulo y uso de normales de vértices interpoladas para iluminación (igualmente, es útil únicamente a partir de la práctica 4).

- **tecla e/E:** activa o desactiva el dibujado de los ejes de coordenadas.
- **tecla q/Q o ESC:** terminar el programa.
- **teclas de cursor:** rotaciones de la cámara entorno al origen.
- **teclas +/-, av.pág/re.pág.:** aumentar/disminuir la distancia de la camara al origen (zoom).

Los eventos correspondientes se gestionan en la función gestora del evento de pulsar o levantar una tecla (**FGE_PulsarLevantarTecla**), dentro del archivo **eventos-tecla.cpp** en la carpeta **src**.

También se da la posibilidad de gestionar la camara con el ratón:

- **desplazar el ratón con el botón derecho pulsado:** rotaciones de la cámara entorno al origen.
- **rueda de ratón (scroll):** aumentar/disminuir la distancia de la camara al origen (zoom).

Estos eventos se gestionan en las funciones gestoras de movimiento de ratón y de botones del ratón (**FGE_MovimientoRaton** y **FGE_PulsarLevantarTecla**, respectivamente), en el archivo **eventos-raton.cpp** de la carpeta **src** (la gestión de la cámara se estudiará en la práctica 5).

1.4. Estructura del código: clases, métodos y funciones.

El archivo **main.cpp** (carpeta **src**) incluye la función principal (**main()**) del código de prácticas, esta función, en primer lugar, crea una instancia de la clase **AplicacionIG**, a través de su constructor sin parámetros. La instancia queda referenciada desde un puntero (**aplicacionIG**) accesible desde cualquier fuente (que haga include de **aplicacion-ig.h**). La clase **AplicacionIG** está declarada en **aplicacion-ig.h** y su implementación está en **aplicacion-ig.cpp**. La instancia incluye punteros a diversos objetos usados en las prácticas y variables de instancia con los valores de diversos parámetros de configuración.

Una vez creado el objeto aplicación **aplicacionIG**, se invoca el método que incluye el bucle principal para gestión de eventos de GLFW (**bucleEventosGLFW**), que está implementado en **aplicacion-ig.cpp**. En cada iteración del bucle, se invoca la función **VisualizarEscena**, para visualizar la escena actual (inicialmente solo hay una escena, en las siguientes prácticas iremos añadiendo escenas con otros tipos de objetos).

En la primera práctica se completará el código de creación (constructor) de la clase **Escena1**, derivada de **Escena**. La clase **Escena1** contendrá varios objetos de clases derivadas de **MallaInd**. Cada uno de esos objetos contiene las tablas de coordenadas de vértices, atributos e indices correspondientes a una malla indexada.

A continuación se detalla la funcionalidad de las distintas clases relevantes.

1.4.1. Cauce, modos de visualización y de envío

Como se ha comentado, la instancia de **AplicacionIG** incluye diversos objetos útiles, entre otros el puntero **cauce** (puntero a un objeto de la clase **Cauce**). Este puntero es inicializado en el constructor de **AplicacionIG**.

Además, el objeto de la aplicación contiene una variable de instancia que codifica el *modo de visualización de polígonos* actual (puntos, alambre, sólido, etc...), en concreto está en la variable de instancia **modo_visu**, que es un valor del tipo enumerado **ModosVisu**, tipo que también se declara en ese archivo de cabecera. Las declaración del tipo enumerado es así:

```
enum class ModosVisu { relleno, lineas, puntos, num_modos } ;
```

La clase **AplicacionIG** contiene otros punteros a objetos y diversas variables de instancia con parámetros de configuración. Entre los objetos cabe destacar el puntero al objeto ventana GLFW (**ventana_glfw**) y un vector de punteros escenas (**escenas**), y entre los segundos, a modo de ejemplo un valor lógico que indica si la iluminación está activada o no. Todas estas variables las iremos viendo en las prácticas.

1.4.2. Clases para descriptores de VBOs y VAOs (**DescrVBOAtribs**, **DescrVBOInds** y **DescrVAO**).

Tal y como se ha visto en teoría, en las prácticas podemos usar las clases con descriptores de VBOs y VAOs (**DescrVBOAtribs**, **DescrVBOInds** y **DescrVAO**), clases que facilitan la creación y activación de VBOs y VAOs. Las clases se declaran en **vaos-vbos.h**, y se definen (implementan) en **vaos-vbos.cpp**.

1.4.2.1. Descriptores de VBOs de atributos (clase **DescrVBOAtribs**)

Los descriptores de VBOs de atributos incluyen estas variables de instancia:

```
class DescrVBOAtribs
{
private:
    GLuint      buffer   = 0 ; // nombre del buffer en GPU (0 antes de crearlo, >0 después)
    GLuint      index    = 0 ; // índice de atributo (<número de tributos del cauce)
    GLenum      type     = 0 ; // tipo de los valores (GL_FLOAT o GL_DOUBLE)
    GLint       size     = 0 ; // numero de valores por tupla (usualmente 2,3, o 4)
    GLsizei     count    = 0 ; // número de tuplas en la tabla (>0)
    GLsizeiptr  tot_size = 0 ; // tamaño datos en bytes (=count*size*sizeof(c-type))
    const void * data    = nullptr ; // datos originales o su copia (nunca nulo)
    void *      own_data = nullptr ; // copia de los datos (nulo si no hay copia)
    .....
};
```

Los constructores de esta clase son estos tres:

```
DescrVBOAtribs( const unsigned p_index, const GLenum p_type,
                 const unsigned p_size,  const unsigned long p_count,
                 const void *p_data );

DescrVBOAtribs( const unsigned p_index,
                 const std::vector<glm::vec3> & src_vec );

DescrVBOAtribs( const unsigned p_index,
                 const std::vector<glm::vec2> & src_vec );
```

Estos constructores (alternativos) permiten crear el descriptor de VBO a partir de un puntero a los datos y algunos metadatos (el primero), o bien, para mayor facilidad, a partir de vectores de tuplas (los dos segundos).

En todos los casos, el constructor **creará una copia de los datos en la memoria**, esto implica

que el programador que llama a los constructores puede modificar o borrar los datos usados en la llamada, pues dichos datos no son leídos por los objetos descriptores una vez que se acaba el constructor.

Los estudiantes deben de completar el código del método `crearVAO` en esta clase. Este método se encarga de crear el VBO en la GPU e inicializarlo, transfiriendo los datos desde la memoria hacia la aplicación. Los detalles pueden verse en la subsección 1.5.1.

1.4.2.2. Descriptores de VBOs de índices (clase `DescrVBOInds`).

Los descriptores de VBOs de índices incluyen estas variables de instancia:

```
class DescrVBOInds
{
private:
    GLuint      buffer   = 0 ; // nombre del buffer en la GPU (0 antes de crearlo, >0 después)
    GLenum      type     = 0 ; // tipo de datos de cada índice (GL_UNSIGNED_BYTE,
                           // GL_UNSIGNED_SHORT, GL_UNSIGNED_INT)
    GLsizei     count    = 0 ; // número de índices en la tabla (>0)
    GLsizeiptr  tot_size = 0 ; // tamaño de los datos en bytes (=count*sizeof(c-type))
    const void * indices  = nullptr ; // datos originales o su copia (nunca nulo)
    void *       own_indices = nullptr ; // copia de los datos (nulo si no hay copia)
    ...
};
```

Los constructores (alternativos) de un descriptor de VBO de índices son:

```
DescrVBOInds( const GLenum p_type, const GLsizei p_count,
               const void * p_data );

DescrVBOInds( const std::vector<unsigned> & src_vec );
DescrVBOInds( const std::vector<glm::uvec3> & src_vec );
```

Estos constructores también permiten crear los descriptores a partir de un puntero o bien a partir de un vector de tuplas. Hacen una copia en memoria de los datos originales, que por tanto pueden ser borrados o modificados por el programador despues de llamar al constructor.

Al igual que en los descriptores de VBos de índices, los estudiantes deben de completar el código del método `crearVAO` (los detalles están en la subsección 1.5.1).

1.4.2.3. Descriptores de VAOs (clase `DescrVAO`)

Los descriptores de VAOs contienen información de los descriptores de VBOs que se han adjuntado, el estado de habilitado o deshabilitado de cada uno de ellos, así como el nombre o identificador del VAO en la GPU:

```
class DescrVAO
{
private:
    GLuint      array   = 0 ; // nombre del VAO en GPU (0 si no creado)
    GLuint      num_atribs = 0; // número máximo de tablas de atributos
    GLsizei     count    = 0 ; // número de vértices
```

```

GLsizei      idxs_count   = 0 ; // número de índices (0 si no hay)
GLenum       idxs_type    = 0;  // tipo de los índices (si hay)
DescrVBOInds * dvbo_indices = nullptr; // VBO de índices (nulo si no hay)

std::vector<DescrVBOAtribs *> dvbo_atributo; // vector descr. VBOs de atributos
std::vector<bool>             atrib_habilitado; // true si cada tabla habilitada
...
};

```

Respecto a los constructores y métodos la clase, se incluye un constructor que recibe dos parámetros: por un lado es el máximo número de tablas de atributos que se gestionan en el cauce. Ese número es actualmente 4 (posiciones, colores, normales y coordenadas de textura), aunque es mejor usar aquí la constante `numero_atributos_cauce` (declarada en el archivo `cauce.h`, dentro de la carpeta de materiales y con el valor 4). Se pueden crear VAOs con un número de tablas inferior a las que gestiona el cauce (quedarán algunas deshabilitadas y sin usar), pero nunca mayor. Por otro lado, el constructor recibe el descriptor del VBO de posiciones (ya que todo VAO debe tenerlo).

El método `crearVAO` se encarga de crear el VAO en la GPU, e invocar al método `crearVBO` de los descriptores de VBOs para cada una de las tablas de atributos que no estén vacías y para los índices (si hay), es decir, este método transfiere todas las tablas datos a la GPU.

Los descriptores de VAOs incluyen dos métodos para agregar al VAO un descriptores de VBOs. Uno de ellos agrega un descriptor de VBO de índices (se puede hacer una vez como mucho) y el otro un descriptor de VBO de atributos. Esta clase también tiene un método para habilitar o deshabilitar una de las tablas de atributos, dado su índice.

La declaración del constructor y los citados métodos es esta:

```

class DescrVAO
{
    ...
public:
    // Constructor (se da el descriptor de VBO de posiciones)
    DescrVAO( const unsigned p_num_atrib, DescrVBOAtribs * p_dvbo_posiciones ) ;

    // Crea el VAO
    void crearVAO() ;

    // Añade un descriptor de VBO de atributos
    void agregar( DescrVBOAtribs * p_dvbo_atributo ) ;

    // Añade el descriptor de índices, por tanto el VAO pasa a ser indexado.
    void agregar( DescrVBOInds * p_dvbo_indices ) ;

    // Habilita/deshabilita una tabla de atributos (index no puede ser 0)
    void habilitarAtrib( const unsigned index, const bool habilitar ) ;

    // Visualiza un VAO usando un tipo de primitiva (mode)
    void draw( const GLenum mode ) ;
};

```

Los estudiantes deben de completar el código de creación del VAO (método `crearVAO`), y el código del método `draw`, los detalles se incluyen en la subsección 1.5.1.

1.4.3. Clase abstracta para objetos gráficos 3D (`Objeto3D`)

La implementación de los diversos tipos de objetos 3D a visualizar en las prácticas se hará mediante la declaración de clases derivadas de una clase base, llamada `Objeto3D`, con un método virtual puro llamado `visualizarGL`. Las clases derivadas de `Objeto3D` deben implementar el método `visualizarGL`, cada una de ellas tendrá una implementación específica. En `objeto3d.h` (carpeta `src`) está la declaración de la clase, de esta forma:

```
class Objeto3D
{
protected:
    std::string nombre_obj ; // nombre asignado al objeto
public:
    // visualizar el objeto con OpenGL
    virtual void visualizarGL() = 0;
    // visualizar el objeto con OpenGL, visualizando únicamente la geometría.
    virtual void visualizarGeomGL() = 0 ;
    // devuelve el nombre del objeto
    std::string nombre() ;
    ....
}; ;
```

La implementación de esta clase está en `src/objeto3d.cpp` (solo están implementados algunos métodos no virtuales puros). Cualquier objeto tiene un nombre (una cadena de caracteres que lo describe). El nombre se puede fijar con el método `ponerNombre`, y se puede leer con `leerNombre`.

Esta clase incorpora un método para fijar el color de un objeto (`ponerColor`). Se puede llamar desde los constructores de las clases derivadas. Si no se llama, el objeto no tiene asignado color (se visualiza con el color por defecto que haya fijado en el cauce en el momento de la llamada a `visualizar`).

El método `visualizarGL` visualiza el objeto. Cualquier tipo de objeto que pueda ser visualizado en pantalla con OpenGL se implementará con una clase derivada de `Objeto3D`, que contendrá una implementación concreta del método virtual `visualizarGL`.

El método `visualizarGeomGL` se usa para visualizar únicamente la geometría del objeto. Es decir, en las clases derivadas se debe implementar para visualizar únicamente los triángulos del mismo, sin usar colores, normales o coordenadas de textura, y sin cambiar el color actual en el cauce. Se usará para visualizar las aristas del modelo en el modo sólido, cuando esté activado el modo de visualización de aristas. Este método es una versión simplificada de `visualizarGL`.

El método `visualizarNormalesGL` sirve para visualizar las normales de los vértices de los objetos, cada una de ellas se visualiza como un segmento de línea partiendo de dicho vértice.

En estas prácticas los métodos `visualizarGL` y `visualizarGeomGL` se implementarán para mallas indexadas (en esta práctica 1) y para nodos del grafo de escena (en la práctica 3). El método `visualizarNormalesGL` se implementará en la práctica 4 para mallas indexadas y nodos del grafo.

1.4.4. Clase para mallas indexadas (`MallaInd`)

Las mallas indexadas son mallas de triángulos modeladas con una tabla de coordenadas de vértices y una tabla de caras (más, opcionalmente, varias tablas de atributos: colores, normales, y coor-

nadas de textura, en estas prácticas). La tabla de caras o de triángulos contiene ternas de valores enteros, cada una de esas ternas tiene los tres índices de las coordenadas de los tres vértices de un triángulo (índices en la tabla de coordenadas de vértices). Se pueden visualizar con OpenGL en cualquiera de los tres modos de envío.

Para estas mallas indexadas se usa la clase **MallaInd** (derivada de **Objeto3D**), que está declarada en **include/malla-ind.h** y que se implementa en **src/malla-ind.cpp**:

```
#include "Objeto3D.hpp"

class MallaInd : public Objeto3D
{
protected:
    // declaraciones de tablas:
    // ...
public:
    virtual void visualizarGL() ;
    virtual void visualizarGeomGL() ;
    // ....
}; ;
```

La clase incluye:

- Como variables de instancia privadas:
 - Tabla de coordenadas de vértices (**vertices**).
 - Tabla de triángulos (**triangulos**) (es la tabla de índices, agrupados de tres en tres).
 - Tablas de atributos de vértices: **col_ver** con los colores, **nor_ver** las normales, **cc_tt_ver** las coordenadas de textura. Cada una de estas tablas puede estar vacía, o bien tener tantas entradas como vértices.
 - Tabla de normales de triángulos (**nor_tri**).
 - Puntero al descriptor de VAO que codifica esta secuencia.
- Como método público virtual, el método **visualizarGL**, que visualiza la malla teniendo en cuenta el parámetro que recibe (una referencia a una instancia de **ContextoVis**).
- Otro método público virtual es **visualizarGeomGL**, que también tiene como parámetro el contexto de visualización, y el cual, como se ha descrito, visualiza únicamente la geometría.

Aquí vemos las declaraciones de las citadas variables de instancia dentro de la declaración de **MallaInd**:

```
// tablas de posiciones, índices y atributos de la secuencia de vértices
std::vector<glm::vec3> vertices ;           // coordenadas de las posiciones de los vértices
std::vector<glm::uvec3> triangulos;          // triángulos (índices)

std::vector<glm::vec3> col_ver ;             // colores de los vértices (3 floats por vértice)
std::vector<glm::vec3> nor_ver ;             // normales de vértices (3 floats por vértice)
std::vector<glm::vec3> nor_tri ;             // normales de triángulos
std::vector<glm::vec2> cc_tt_ver ;           // coordenadas de textura de los vértices

// puntero al descriptor de VAO (creado en primera llamada a visualizarGL)
DescriVAO * dvaو = nullptr ;
```

1.4.4.1. El método `visualizarGL` de `MallaInd`

El método `visualizarGL` se encarga de visualizar la malla indexada usando el descriptor de VAO, y usando el color del objeto, si tiene alguno. Para realizar la visualización se usa el puntero al descriptor de VAO, dicho descriptor se crea la primera vez que se ejecuta el método. Después se usa el método `draw`.

En el caso de que el objeto malla tenga definido un color, se debe cambiar el color actual del cauce antes de dibujar. Si no tiene color, eso no es necesario, y entonces se usa el color actual del cauce, ya fijado antes de la llamada a este método. Si el color del cauce se ha cambiado, entonces después de dibujar se debe restaurar el color original. De esta forma, el método es *neutro* respecto a dicho color, es decir, siempre será el mismo al inicio y al final.

El código de este método debe ser completado por el estudiante, los detalles de esta tarea están en la subsección 1.5.2

1.4.4.2. El método `visualizarGeomGL` de `MallaInd`

El método `visualizarGeomGL` se encarga de visualizar únicamente la geometría de la malla indexada, sin usar colores, normales, texturas, y sin cambiar nunca el color actual en el cauce (se usa el que haya puesto en el momento de la llamada). Este método se invoca cuando se quiera visualizar las aristas del objeto actual de la escena.

Para visualizar, en este método se usa el mismo descriptor de VAO que se usa en el método `visualizarGL`, pero deshabilitando las tablas de atributos. Eso implica que, para cada objeto, este método únicamente puede llamarse después de `visualizarGL` (eso está garantizado ya únicamente se dibujan las aristas después de haber visualizado el objeto con `visualizarGL`).

El código de este método también debe ser completado por el estudiante, los detalles de la tarea están en la subsección 1.5.2.

1.4.5. La clase `Escena`

Una escena es básicamente un contenedor de elementos necesarios para visualizar objetos. Cada instancia de la clase `Escena` contiene: un vector de cámaras (inicialmente una sola de ellas, en la práctica 4 se añaden más), una colección de fuentes de luz, un material inicial o por defecto (ambos para iluminación, en la práctica 4) y un vector de objetos de tipo `Objeto3D`. En cada momento, la escena tiene un objeto actual (uno del vector de objetos) y una cámara actual, son los que se usan para visualizarla.

El objeto y la cámara actual se pueden cambiar usando los métodos `siguienteObjeto` y `siguienteCamara` respectivamente (activan el siguiente objeto o la siguiente cámara). Se pueden obtener punteros al objeto y la cámara actual (con `objetoActual` y `camaraActual`, respectivamente).

La clase escena tiene un método llamado `visualizarGL` que se encarga de visualizar el objeto actual usando la cámara actual. Este método es responsable de configurar OpenGL para que se pueda visualizar correctamente el objeto actual. Se invoca desde el método `visualizarFrame` de la clase `AplicacionIG` (archivo `aplicacion-ig.cpp`), inmediatamente después de limpiar la ventana.

El método `visualizarGL` de la clase escena se encarga básicamente de visualizar el objeto actual

de dicha escena, llamando a su vez al método `visualizarGL` de dicho objeto. Si corresponde visualizar las aristas, también hay que llamar a `visualizarGeomGL` de dicho objeto. Antes de ambas llamas, es necesario poner el cauce en un estado adecuado a la correspondientes llamadas.

Esta clase escena incorpora un constructor (sin parámetros), que se encarga de crear el vector de cámaras y (en la práctica 4), la colección de fuentes de luz y el material inicial.

En cada una de las prácticas se define una subclase derivada de la clase `Escena` (en `escena.h`). En concreto, para la primera práctica se define `Escena1`. Estas subclases únicamente definen un constructor que crea el vector de objetos de la escena (haciendo `push_back` sobre el vector `objetos`), añadiéndole los objetos que corresponda.

La clase `AplicacionIG` incluye entre sus variables de instancia un vector de escenas (una por cada práctica). En `aplicacion-ig.cpp`, al final del constructor de dicha clase, se crea un vector de escenas (una por cada práctica), haciendo `push_back` sobre la variable `escenas`. Durante la ejecución del programa es posible pulsar la tecla P para activar la siguiente escena.

El estudiante debe simplemente escribir el código que añade cada escena de cada práctica a dicho vector (ver subsección 1.5.3)

1.4.6. Uso de la instancia de la clase `Cauce`.

En el constructor de la clase `AplicacionIG` se crea una instancia de la clase `Cauce`, y se guarda un puntero a ella en dicha instancia (llamado `cauce`). Cada vez que se visualiza la escena, las funciones y métodos de visualización cambian el estado del cauce a través de este puntero. Para obtenerlo se usa la variable global con el puntero a la aplicación (`aplicacionIG`), es decir, siempre podemos recuperar el cauce mediante la expresión `aplicacionIG->cauce` (supuesto que se ha hecho `include` del archivo `aplicacion-ig.h`, lo cual se hace en la mayoría de los archivos `.cpp`).

Muchos métodos y funciones incluyen al inicio sentencias para leer el cauce en una variable local y comprobar que no es nula:

```
assert( aplicacionIG != nullptr );
Cauce * cauce = aplicacionIG->cauce ; assert( cauce != nullptr );
```

1.4.7. Clases para los objetos de la práctica 1

La clase `Cubo` contiene un cubo de 8 vértices, sin colores, normales ni coordenadas de textura. Ya se encuentra implementada en `malla-ind.cpp` (y declarada en `malla-ind.h`).

```
class Cubo : public MallaInd
{
public:
    Cubo() ; // crea las tablas del cubo, y le da nombre.
};
```

1.5. Tareas

Para realizar la práctica es necesario completar el código de visualización de malla indexadas y definir varias clases de clases derivadas de `MallaInd`, al igual que `Cubo` pero con otras formas o con otras características. En la plantilla se indica en comentarios los sitios donde se debe completar el

código. Además, se puede crear nuevos archivos `.cpp` en la carpeta `src` en el directorio de trabajo. Esos nuevos archivos se compilan junto con los demás, pero para ello es necesario volver a generar los archivos de compilación, es decir, vaciar la carpeta `cmake` y volver a ejecutar `cmake` y luego `make` para compilar.

A continuación se detallan las distintas tareas a realizar.

1.5.1. En el archivo `vaos-vbos.cpp`

En este archivo debemos de completar el código de creación de los descriptores de VBOs de atributos y de índices (clases `DescrVBOAtrib` y `DescrVBOInds`), en concreto hay que completar el método `crearVBO` de cada una de estas dos clases.

Respecto a los descriptores de VBOs de atributos, hay que añadir código para dar estos pasos:

1. Generar un nuevo identificador o nombre de VBO.
2. Fijar este buffer como buffer activo actualmente en el target `GL_ARRAY_BUFFER`.
3. Transferir los datos desde la memoria de la aplicación al VBO en GPU.
4. Registrar para este índice de atributo, la localización y el formato de la tabla en el buffer.
5. Desactivar el buffer.
6. Habilitar el uso de esta tabla de atributos.

Y respecto a los descriptores de VBOs de índices, hay que añadir código para dar estos pasos:

1. Crear un nuevo nombre o identificador de VBO.
2. Activar (hacer *bind*) el buffer en el target `GL_ELEMENT_ARRAY_BUFFER`.
3. Transferir los datos desde la memoria de la aplicación al VBO en GPU.

Hay que completar el método `crearVAO` de la clase `DescrVAO`, se deben escribir código que da estos pasos:

1. Crear el nombre de VAO y activarlo como VAO actual (hacer *bind*)
2. Para cada VBO de atributos adjunto al VAO (puntero en `dvbo_atributo` no nulo):
 - crear el VBO de atributos en la GPU (usar el método `crearVBO`).
3. Si hay índices (puntero `dvbo_indices` no nulo) entonces,
 - crear el VBO de índices (usar método `crearVBO`)
4. Para cada VBO de atributos adjunto al VAO (puntero en `dvbo_atributo` no nulo):
 - si la tabla está marcada como deshabilitada (`false` en su entrada del vector `atrib_habilitado`):
 - deshabilitarla en la GPU con `glDisableVertexAttribArray`

Finalmente, es necesario implementar el código del método `draw` de la clase `DescrVAO`, este método debe encargarse de visualizar el VAO. La primera vez que se ejecuta para un descriptor concreto, se hará la creación del VAO en la GPU, antes de visualizar. Al final del método, se debe desactivar el VAO para que no afecte a futuras operaciones.

Por tanto, el estudiante debe de escribir código para dar estos pasos:

1. Comprobar si el VAO se ha creado en la GPU o no (antes de crearse, `array` vale 0, después es > 0).
 - Si el array no se ha creado todavía: hay que crear el VAO en la GPU (método `crearVAO`), el VAO queda activado.
 - En otro caso (el array ya se ha creado): activar el VAO (con `glBindVertexArray`)
2. Comprobar si la secuencia es indexada o no lo es (si no es indexada `dvbo_indices` es nulo).

- Si la secuencia es indexada: visualizar con `glDrawElements`.
 - En otro caso (la secuencia no es indexada): visualizar con `glDrawArrays`.
3. Desactivar el VAO (activar el VAO 0 con `glBindVertexArray`).

1.5.2. En el archivo `malla-ind.cpp`

En este archivo es necesario completar el método `visualizarGL` de la clase `MallaInd`, según lo que ya se ha indicado sobre el funcionamiento requerido de este método. En concreto, se deben de añadir código para:

1. Comprobar si el objeto tiene un color asociado (lo cual se consulta con el método `tieneColor`), si lo tiene entonces hay que dar estos dos pasos:
 - 1.1. Hacer push del color actual del cauce (con el método `pushColor` de `Cauce`), y
 - 1.2. Leer el color del objeto (con `leerColor`) y usarlo para cambiar el color del cauce (con el método `fijarColor`).
2. Si el puntero `dva0` es nulo (ocurre en la primera llamada a este método para cada objeto malla), se debe de crear el descriptor de VAO con todos sus descriptores de VBOs asociados, para ello hay que:
 - 2.1. Crear el descriptor de VBO de posiciones, y usarlo para crear el descriptor VAO.
 - 2.2. Crear y añadir (con el método `agregar`) el descriptor del VBO de índices con la tabla de índices (es la tabla de triángulos).
 - 2.3. Para cada tabla de atributos (distintos de las posiciones de los vértices) que no esté vacía, se crea el correspondiente descriptor de VBO y se añade al descriptor del VAO (de nuevo con `agregar`).
3. Visualizar el VAO, usando el método `draw` del descriptor (se debe usar el tipo de primitiva adecuado).
4. Si el objeto tiene un color, es necesario restaurar el color anterior del cauce (con el método `popColor`).

También se debe escribir el código de `visualizarGeomGL`. Esta función debe asumir que el VAO completo ya está creado (en realidad esta función siempre se llama después de llamada `visualizarGL`, por eso se puede asumir). A continuación:

1. Desactivar todas las tablas de atributos en descriptor del VAO (que no estén vacías). Para ello se usa el método `habilitarAtrib` de la clase `DescrVAO`.
2. Visualizar el VAO (únicamente visualizará los triángulos), para ello se usa el método `draw`.
3. Volver a activar todos los atributos para los cuales la tabla no esté vacía (de nuevo con `habilitarAtrib`).

1.5.3. En el archivo `escena.cpp`

1.5.3.1. Método `visualizarGL` de la clase `Escena`

En el método `visualizarGL` de la clase `Escena` se dan distintos pasos para visualizar una escena. Después de fijar el color, se debe escribir el código para fijar el modo de visualización de polígonos (rellenos, aristas o puntos), usando la función `glPolygonMode` y la variable `modo_visu` que hay en la instancia de la aplicación.

A continuación se debe escribir el código para invocar el método `visualizarGL` de la clase `Objeto3D`, usando el puntero `objeto` al objeto actual de la escena.

Finalmente, cuando está activada la visualización de aristas (y el modo es el modo de relleno), es

necesario escribir el código para visualizar dichas aristas para el mismo objeto que acabamos de visualizar (**objeto**). Este código debe usar **glPolygonMode** para hacer que se dibujen únicamente las aristas de los triángulos, usar **fijarColor** del cauce para dibujar todas las aristas en color negro, y finalmente invocar el método **visualizarGeomGL** para el objeto actual, método que se encarga de visualizar únicamente la geometría, sin usar colores, normales ni coordenadas de textura, como se ha explicado antes.

1.5.3.2. Constructor de **Escena1**

La clase **Escena1** es una clase derivada de **Escena**, que simplemente añade un método constructor (**Escena1::Escena1**). En ese método constructor se puebla el array de objetos 3D (array **objetos**), que constituyen el catálogo de objetos de la escena (en cada momento se visualiza uno de ellos).

Para ello se debe de hacer **push_back** de los objetos (de tipo malla indexada) que se crean para esta práctica (cubo, tetraedro, etc...). Cada objeto se crea en memoria dinámica con **new**, y el puntero resultante se inserta en el vector.

1.5.4. Clases nuevas para otros tipos de mallas indexadas

En **malla-ind.cpp** ya está implementada la clase **Cubo**, derivada de **MallaInd**. El constructor (sin parámetros) crea un cubo de 6 caras, lado 2 unidades y centro en el origen (se extiende entre -1 y +1 en los tres ejes). Esta constructor no crea las tablas de atributos de vértices, que quedan vacías, lo cual significa que el cubo se dibuja del color por defecto fijado antes de visualizar el objeto.

Se debe crear una clase nueva de nombre **Tetraedro**, que tiene un tetraedro (no necesariamente regular), formado por 4 vértices y 4 caras (es la malla indexada más sencilla posible). Tampoco es necesario crear las tablas de atributos en esta clase. Sin embargo, en el constructor podemos fijar un color distinto del blanco para este tipo de objeto, usando el método **ponerColor** de la clase base **Objeto3D**. Este color se usará durante la visualización.

Adicionalmente se creará una clase llamada **CuboColores**, también derivada de **MallaInd**. En el constructor se inicializan las tablas de vértices y los triángulos igual que en la clase **Cubo**, pero además se definirá la tabla de colores de vértices. Cada color será una terna RGB, de forma que la componente R del color depende la componente X de la posición (si la X es -1, R es 0, y si X es +1, R es 1). Igualmente, G depende la componente Y y B de la componente Z.

1.6. Ejercicios adicionales

1.6.1. Ejercicio 1

(examen de la convocatoria ordinaria, curso 20-21)

Extiende los archivos **malla-ind.h** y **malla-ind.cpp** de tus prácticas para incluir (al final) las declaraciones e implementaciones de una nueva clase llamada **EstrellaZ** y derivada de **MallaInd**, cuyo constructor acepta un parámetro *n* (**unsigned**, *> 1*).

El constructor inicializa las tablas de vértices, triángulos y colores de una malla indexada con $2n$ triángulos y $2n + 1$ vértices, en forma de estrella, plana (en el plano perpendicular al eje Z) y con *n* puntas. Los vértices tienen coordenadas entre 0 y 1 en X y en Y (y todos tienen Z igual a cero). El

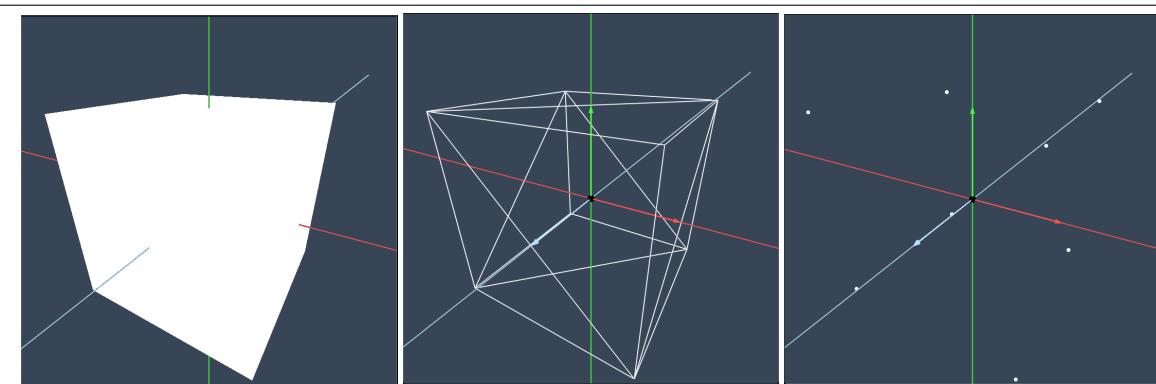


Figura 1.1: Cubo visualizado con los tres modos de visualización de polígonos: relleno (izquierda), aristas (centro) y puntos (derecha)

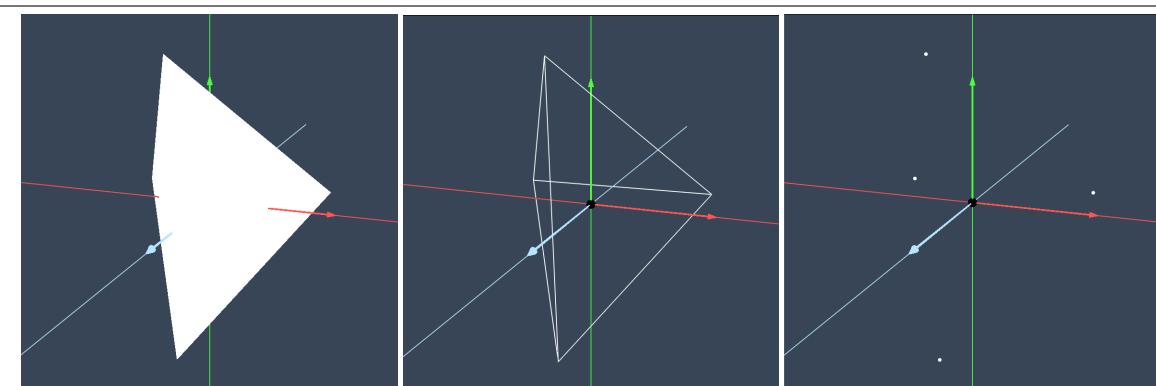


Figura 1.2: Tetraedro visualizado con los tres modos de visualización de polígonos: relleno (izquierda), aristas (centro) y puntos (derecha)

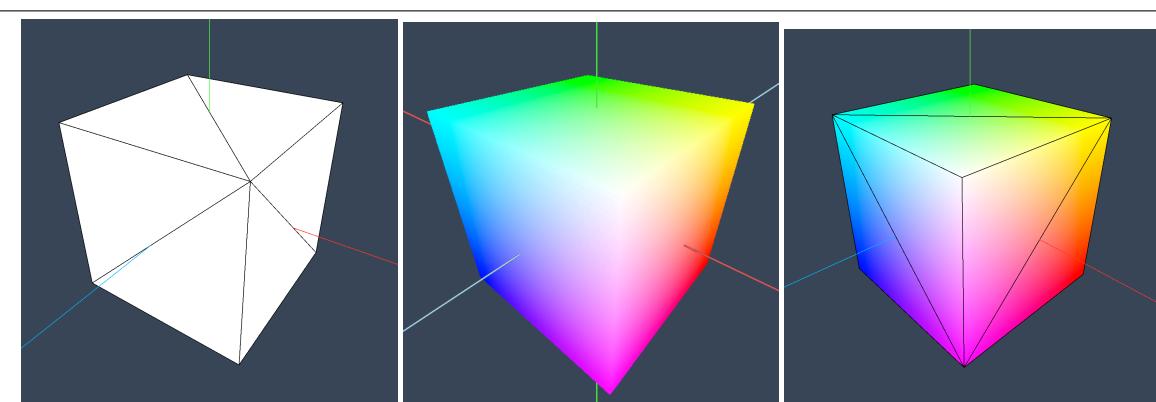
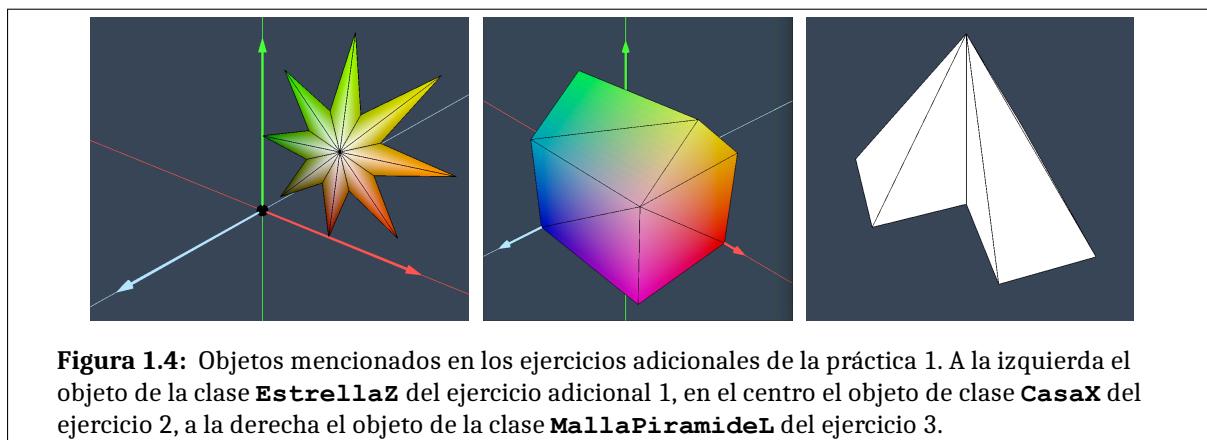


Figura 1.3: Cubo relleno con visualización de aristas (izquierda), cubo con colores (centro), cubo con colores y visualización de aristas (derecha).

centro de la estrella está en $(0,5,0,5)$ en X e Y, y los radios hasta las puntas son de longitud 0,5

El vértice central tiene color blanco. El resto de vértices tienen colores cuyas componentes R, G y B coinciden con sus coordenadas X, Y y Z, respectivamente.

En la figura 1.4 (izquierda) se observa como debe quedar este objeto una vez que se visualiza. Como



variantes, se propone hacer la estrella en un plano perpendicular al eje X o al eje Y.

1.6.2. Ejercicio 2

(examen de la convocatoria ordinaria, curso 20-21)

Extiende los archivos **malla-ind.h** y **malla-ind.cpp** de tus prácticas para incluir (al final) las declaraciones e implementaciones de la nueva **CasaX**, derivada de **MallaInd**, cuyo constructor (sin parámetros) es inicialmente una copia del cubo, pero que debes adaptar para (a) quitarle los dos triángulos de la base y los dos de la tapa (son los 4 triángulos perpendiculares al eje Y) y (b) añadirle en la parte superior 6 triángulos que forman una tejado a dos aguas, cuya arista superior es paralela al eje X. La casa es más alargada en el eje X que en el eje Z, pero tiene todas las coordenadas de todos los vértices entre 0 y 1 (ninguna negativa).

Cada vértice tiene un color RGB cuyas componentes son iguales a sus coordenadas XYZ. En la figura 1.4 (centro) se observa como queda el objeto de la clase **CasaX** al visualizarse.

1.6.3. Ejercicio 3

(examen de la convocatoria extraordinaria, curso 20-21)

Crea tres clases derivadas de **MallaInd** en **malla-ind.h** y **malla-ind.cpp** (al final de ambos archivos) con estos nombres y requerimientos:

- Clase **MallaTriangulo**: es una malla indexada compuesta de un único triángulo con tres vértices, el triángulo está en el plano perpendicular al eje Z, con centro de la base en el origen, la base tiene longitud unidad, y la altura la raíz de 2.
- Clase **MallaCuadrado**: es una malla indexada en forma de cuadrado, con 2 triángulos y 4 vértices. Está en el plano perpendicular al eje Z, con centro en el origen y lado 2 unidades.
- Clase **MallaPiramideL**: es una malla indexada con forma de pirámide cuya base tiene forma de L. Está formado por 7 vértices e incluye los triángulos de la base y los 6 triángulos adyacentes al ápice. Usa un número mínimo de triángulos para formar la base.

En la figura 1.4 (derecha) se observa como queda el objeto de la clase **MallaPiramideL** al visualizarse.

2. Modelos PLY y Poligonales.

2.1. Objetivos

Aprender a:

- A leer modelos guardados en ficheros externos en formato PLY (Polygon File Format) y su visualización.
- Modelar objetos sólidos poligonales mediante técnicas sencillas. En este caso se usará la técnica de modelado por revolución de un perfil alrededor de un eje de rotación. Se crearán varios tipos de objetos:
 - Objeto por revolución con el perfil almacenado en un archivo PLY (que contiene únicamente vértices)
 - **Cilindro**: con centro de la base en el origen, altura unidad.
 - **Cono**: con centro de la base en el origen, altura unidad.
 - **Esfera**: con centro en el origen, radio unidad.

2.2. Desarrollo

En este práctica se aprenderá a leer modelos de mallas indexadas usando el formato PLY. Este formato sirve para almacenar modelos 3D de dichas mallas e incluye la lista de coordenadas de vértices, la lista de caras (polígonos con un número arbitrario de lados) y opcionalmente tablas con diversas propiedades (colores, normales, coordenadas de textura, etc.). El formato fue diseñado por Greg Turk en la universidad de Stanford durante los años 90. Para más información sobre el mismo, se puede consultar:

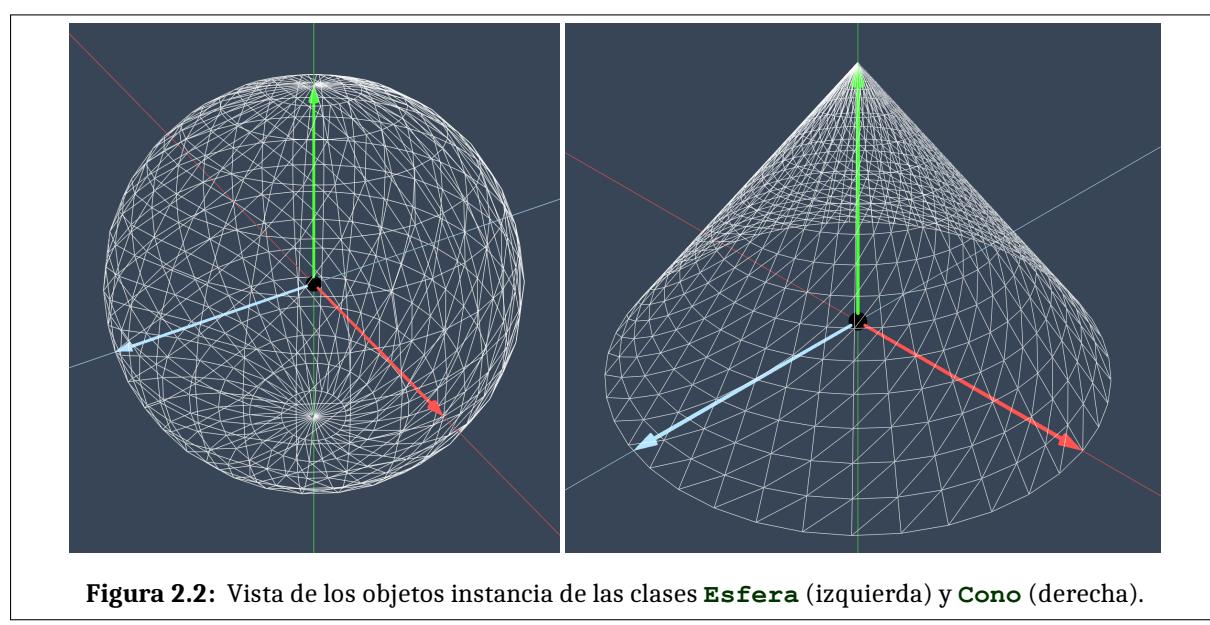
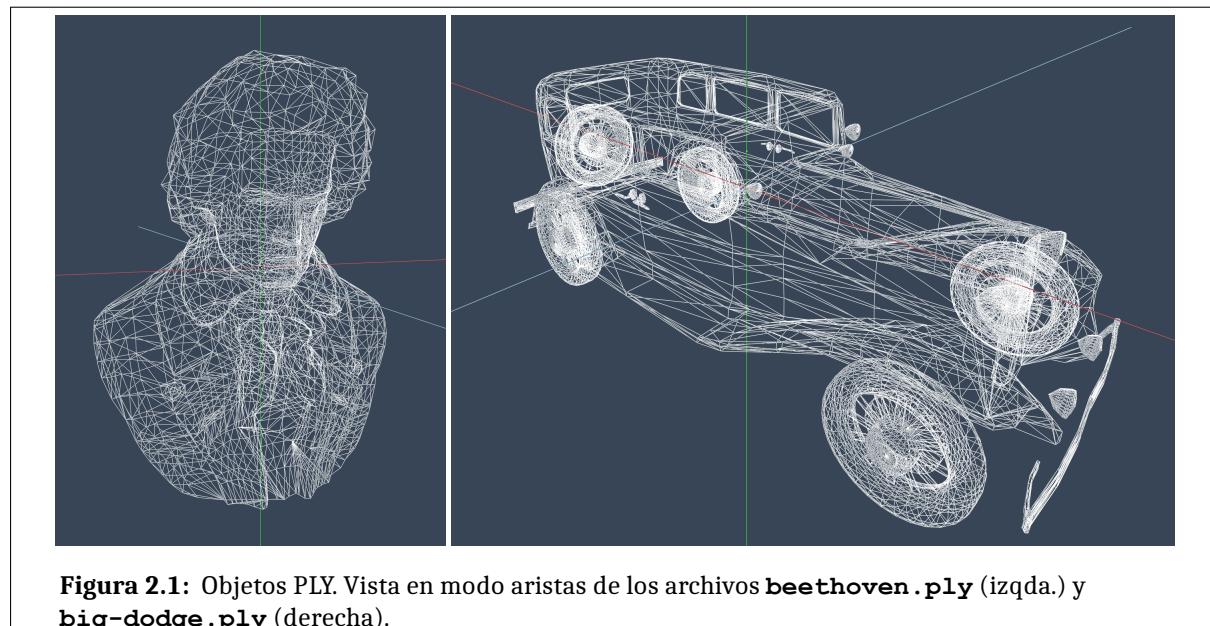
-  <http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html>

Para la realización de la práctica, en primer lugar, se visualizarán modelos de objetos guardados en formato PLY usando los modos de visualización implementados en la primera práctica. Para ello, se entregará el código de un lector básico de ficheros PLY para objetos únicamente compuestos por vértices y caras triangulares, que devuelve un vector de coordenadas de los vértices y un vector de los índices de vértices que forman cada cara. Se creará una estructura de datos que guarde los dos vectores anteriores.

En segundo lugar, se desarrollará un algoritmo para la generación procedural de una malla obtenida por revolución de un perfil alrededor del eje Y. Dicho algoritmo tiene como parámetros de entrada la secuencia de vértices que define dicho perfil, y el número de copias del mismo que servirán para crear el objeto. Como salida, se generará la tabla de vértices y la tabla de caras (triángulos) correspondientes a la malla indexada que representa al objeto.

En la sección 2.4 se detalla el algoritmo de creación del sólido por revolución (se implementa en un constructor, ver la sección 2.5.3 sobre la estructura del código). Partimos de un perfil inicial u original, es una secuencia de m tuplas de coordenadas de vértices en 3D, todas esas coordenadas con $z = 0$.

El perfil inicial se puede leer de un fichero PLY cuyo contenido sólo ha de tener las coordenadas de los vértices, ya que las caras se construyen después de leer los vértices. Este fichero PLY puede



escribirse manualmente, o bien se puede usar el que se proporciona en el material de la práctica, correspondiente a la figura de un peón, y que se muestra a continuación:

Además de leer el perfil original de un archivo PLY, también será posible crear el objeto de revolución a partir de un perfil original creado proceduralmente (es decir, usando código) en el propio programa. Esta será la opción que usaremos para crear los perfiles originales de los objetos de tipo cilindro, cono y esfera.

Usando este perfil base u original, queremos crear un total de n instancias o copias rotadas de dicho perfil. Esto implica insertar un total de nm vértices en la tabla de vértices, y después todas las caras (triángulos) correspondientes.

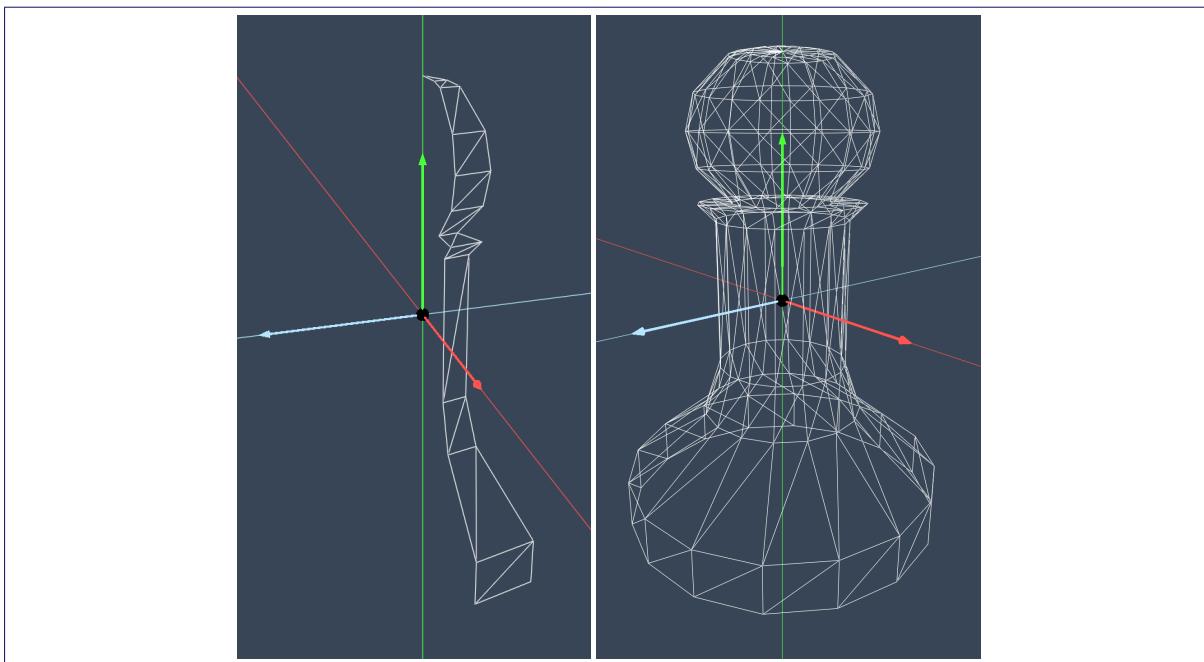


Figura 2.3: Vista de la primera ristra de triángulos entre las dos primeras copias del perfil original (izquierda) y del objeto *peon* obtenido del perfil almacenado en **peon.ply**

```

ply
format ascii 1.0
element vertex 11
property float32 x
property float32 y
property float32 z
element face 1
property list uchar uint vertex_indices
end_header
1.0 -1.4 0.0
1.0 -1.1 0.0
0.5 -0.7 0.0
0.4 -0.4 0.0
0.4 0.5 0.0
0.5 0.6 0.0
0.3 0.6 0.0
0.5 0.8 0.0
0.55 1.0 0.0
0.5 1.2 0.0
0.3 1.4 0.0
3 0 1 2

```

Figura 2.4: Texto del archivo PLY **peon.ply**.

2.3. Tareas

A modo de referencia, aquí se incluyen las diferentes tareas a completar:

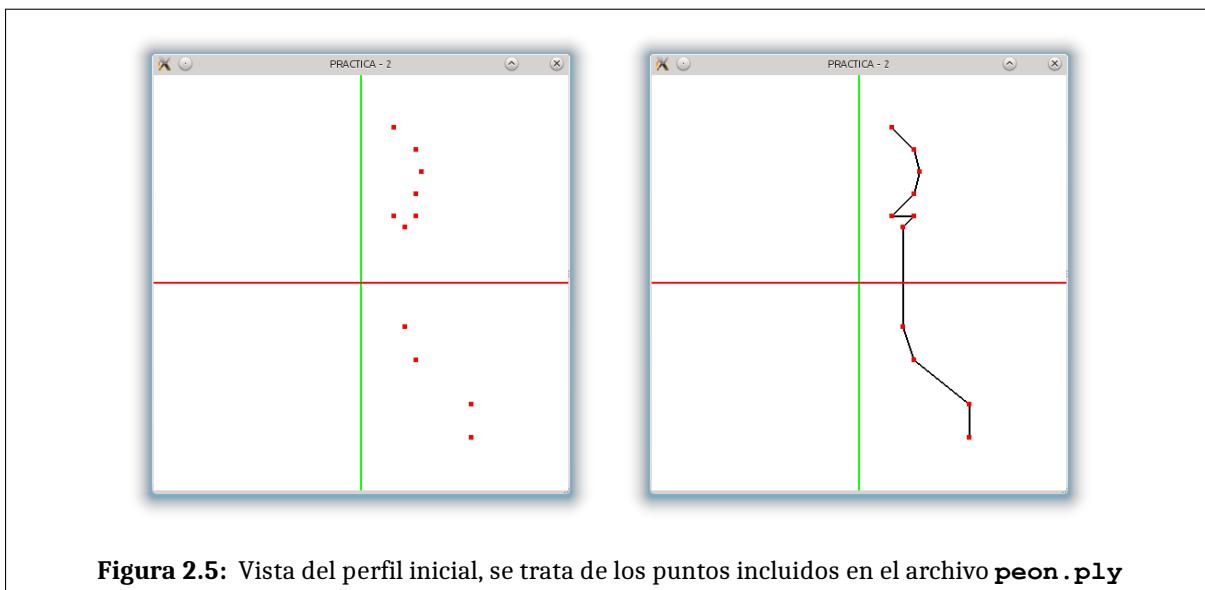


Figura 2.5: Vista del perfil inicial, se trata de los puntos incluidos en el archivo `peon.ply`

1. Añadir en el constructor de `AplicacionIG` la creación de una instancia de `Escena2` (en `aplicacion-ig.cpp`), esa instancia se debe añadir al vector de escenas de la aplicación (ver sección 2.5.1).
2. Declarar e implementar la clase `Escena2` en `escena.h` y `escena.cpp`, respectivamente. Añade un constructor que puebla el vector de objetos con objetos de los que se describen en esta práctica (ver la sección 2.5.1 para saber qué tipos son esos exactamente).
3. Completar el constructor `MallaPLY::MallaPLY` en `malla-ind.cpp`. Se encarga de leer los vértices y caras con la función `LeerPLY` (ver sección 2.5.2).
4. Completar el método `MallaRevol::inicializar` en `malla-revol.cpp`, en ese método se debe implementar el algoritmo de generación de las tablas de vértices y caras a partir de un perfil (ver sección 2.4 y subsección 2.5.3).
5. Completar el constructor `MallaRevolPLY::MallaRevolPLY` en `malla-revol.cpp`. Se debe de leer el perfil usando la función `LeerVerticesPLY`, y después usar el método `inicializar` para generar la malla (ver sección 2.5.4).
6. Declarar e implementar las clases `Cilindro`, `Cono` y `Esfera` (en `malla-revol.h` las declaraciones y en `malla-revol.cpp` las implementaciones) (ver sección 2.5.5).

2.4. Algoritmo para la creación de malla por revolución

Para la creación de un objeto de revolución, lo más fácil es crear, en primer lugar, la tabla de vértices (partiendo del perfil original) y en segundo lugar la tabla de triángulos. Hay que tener en cuenta que es necesario insertar una última copia del perfil que coincide en las mismas posiciones de vértices que la primera de ellas. Aunque se podría conectar con triángulos la última copia con la primera, ello no es posible pues en la práctica 4 veremos como eso haría imposible definir correctamente las normales o las coordenadas de textura. Este es un buen ejemplo de la necesidad de duplicar vértices debido a discontinuidades en los atributos (normal, coordenadas de textura) en la superficie que queremos aproximar con la malla.

Supongamos que el *perfil original* (leído de un PLY o almacenado en una tabla) tiene m vértices. Dicho vértices (dichas tuplas de coordenadas) los nombramos como $\{ \mathbf{p}_0, \dots, \mathbf{p}_{m-1} \}$. Se supone que los vértices se dan de abajo hacia arriba, en el sentido de que las coordenadas Y de esos vértices son

crecientes. Normalmente los vértices tienen coordenadas X estrictamente mayor que cero, pero el algoritmo funciona bien si algún vértice tiene coordenada X igual a cero (nunca negativa), en este caso se crean triángulos degenerados (sin área), que OpenGL ignora durante la rasterización.

De ese perfil original haremos $n > 3$ replicas rotadas (a cada una de ellas la llamamos una *instancia del perfil*). Cada uno de estas instancias del perfil forma un ángulo de $2\pi/(n - 1)$ radianes con la siguiente o anterior. Las instancias del perfil se numeran desde 0 hasta $n - 1$, por tanto, la i -ésima instancia forma un ángulo de $2\pi i/(n - 1)$ radianes con la instancia número 0. Las instancias 0 y $n - 1$ tienen sus vértices en las mismas posiciones que el perfil original.

Por supuesto se van a crear nm vértices en total. Dichos vértices se insertarán en la tabla de vértices por instancias del perfil (es decir, todos los vértices de una misma instancia aparecen consecutivos), además las instancias se almacenan en orden (empezando en la instancia 0 hasta la $n - 1$). Por tanto, sabemos que, en la tabla final de vértices, el j -ésimo vértice de la i -ésima instancia tendrá un índice en la tabla igual a $im + j$ (donde i va desde 0 hasta $n - 1$ y j va desde 0 hasta $m - 1$).

Para crear los vértices, por tanto, bastará con hacer un bucle doble que recorre todos los pares (i, j) y en cada uno de ellos crea el vértice correspondiente y lo inserta al final de la tabla de vértices.

El convenio anterior también facilita la creación de la tabla de caras. Para ello, basta hacer un bucle externo, con un índice i que recorre las instancias. Dentro habrá un bucle interno que recorrerá todos los vértices de la instancia número i , excepto el último de ellos. Por cada vértice visitado se insertan en la tabla de caras dos triángulos adyacentes (comparten la arista diagonal).

El pseudo-código, por tanto, para la creación de la tabla de vértices será como sigue:

- Partimos de la tabla de vértices vacía.
- Para cada i desde 0 hasta $n - 1$ (ambos incluidos)
 - Para cada j desde 0 hasta $m - 1$ (ambos incluidos)
 - Sea $\mathbf{q} =$ vértice obtenido rotando \mathbf{p}_j un ángulo igual a $2\pi i/(n - 1)$ radianes.
 - Añadir \mathbf{q} a la tabla de vértices (al final).

Por otro lado, el pseudo-código para la tabla de triángulos visita todos los vértices (excepto el último de cada instancia y los de la última instancia), y crea dos triángulos nuevos que son adyacentes a ese vértice:

- Partimos de la tabla de triángulos vacía
- Para cada i desde 0 hasta $n - 2$ (ambos incluidos)
 - Para cada j desde 0 hasta $m - 2$ (ambos incluidos)
 - Sea $k = im + j$
 - Añadir triángulo formado por los índices $k, k + m$ y $k + m + 1$.
 - Añadir triángulo formado por los índices $k, k + m + 1$ y $k + 1$.

2.5. Estructura del código. Clases.

En esta sección se detallan las clases a modificar, extender o crear en esta práctica 2, en concreto se trata de las clases con la colección de objetos (**Escena2**), la clase con mallas leídas de archivos PLY (**MallaPLY**), y las clases con objetos de revolución de diversos tipos (clases **MallaRevol**, **MallaRevolPLY** y derivadas).

2.5.1. La clase Escena2

Esta es una clase derivada de **Escena**, que añade únicamente un constructor, el cual añade a la escena los objetos de la práctica 2 (es igual que **Escena1** pero con otro constructor). Esta clase se debe declarar en **escena.h** e implementarse en **escena.cpp**. Los objetos de la práctica 2 serán:

- Al menos dos objetos **MallaPLY** (los archivados en **beethoven.ply** y **big_dodge.ply**), opcionalmente se podrán añadir otros objetos (ver subsección 2.5.2).
- Objetos obtenidos por revolución de un perfil, de dos tipos:
 - Con el perfil creado proceduralmente, al menos la esfera, el cono y el cilindro. Son instancias de **MallaRevol** o de clases derivadas (ver subsecciones 2.5.3 y 2.5.5).
 - Con un perfil leído de un archivo PLY, instancia de **MallaRevolPLY**. Al menos el peón, opcionalmente otros (ver subsección 2.5.4).

Además, en el constructor de **AplicacionIG** (que está en el archivo **aplicacion-ig.cpp**) será necesario añadir (con **push_back**) una instancia de **Escena2** al vector de escenas de la aplicación, igual que ya se hace con una instancia de **Escena1**.

2.5.2. Clase MallaPLY: mallas creadas a partir de un archivo PLY.

La implementación de los objetos tipo malla obtenidos a partir de un archivo PLY debe hacerse usando la clase (**MallaPLY**), derivada de la clase para **MallaInd**. La clase **MallaPLY** no introduce un nuevo método de visualización, ya que este tipo de mallas indexadas se visualizan usando el mismo método que ya se implementó en la práctica 1 para todas las demás, leyendo de las mismas tablas. La única diferencia de este tipo de mallas es como se construyen, y por tanto lo que hacemos es introducir un constructor específico nuevo, que construye la tablas de la malla indexada usando un parámetro con el nombre del archivo. La declaración de la clase, por tanto, es esta:

```
class MallaPLY : public MallaInd
{
public:
    MallaPLY( const std::string & nombre_arch ) ;
};
```

La declaración de esta clase está en **malla-ind.h**, y su implementación (incompleta) se encuentra en **malla-ind.cpp**. El código del constructor debe llamar a la función **LeerPLY**, que tiene como parámetros (1) el nombre del archivo (parámetro de entrada), (2) la tabla de vértices (de salida, un vector de **glm::vec3**) y (3) la tabla de triángulos (también de salida, un vector de **glm::uvec3**). Esta función ya está implementada y se encarga de añadir los vértices y las caras (leídos del PLY) a los correspondientes vectores.

El nombre del archivo no debe incluir el *path*, únicamente el nombre y la extensión (**.ply**). El archivo se buscará en la carpeta **materiales/ply** (donde están los archivos que se proporcionan), si no está ahí se busca en **archivos-alumno** (los archivos que ha buscado el alumno), y finalmente, si no se encuentra, se produce un error y el programa aborta.

2.5.3. Clase MallaRevol: mallas obtenidas por revolución de un perfil.

La implementación de los objetos tipo malla obtenidos a partir de un perfil, por revolución, debe hacerse usando clases derivadas de la clase **MallaRevol**, a su vez derivada de la clase para **MallaInd**. La clase **MallaRevol** ya está declarada en el archivo **malla-revol.h** y su imple-

mentación debe completarse en `malla-revol.cpp`. Esta clase sirve como base para clases concretas de objetos de revolución. No tiene constructor, ya que las clases derivadas deben llamar al método `inicializar` para crear la tabla de vértices y triángulos, desde sus propios constructores, a partir de algún perfil original.

```
class MallaRevol : public MallaInd
{
protected:
    MallaRevol() {} // solo usable desde clases derivadas con constructores específicos

    // Método que crea las tablas de vértices, triángulos, normales y cc.de.tt.
    // a partir de un perfil y el número de copias que queremos de dicho perfil.
    void inicializar
    (
        const std::vector<glm::vec3> & perfil,           // tabla de vértices del perfil original
        const unsigned                  num_copias      // número de copias del perfil
    );
};
```

Se debe implementar, en el método `inicializar` el algoritmo descrito (en la sección 2.4) para crear las tablas de vértices y caras (triángulos), a partir del perfil original.

2.5.4. Clase `MallaRevolPLY`: revolución de un perfil leído en un PLY

La clase `MallaRevolPLY` es una clase derivada de `MallaRevol` que incluye un constructor que debe leer los vértices del perfil de un archivo PLY y luego crea la malla llamando al método `inicializar`. Tiene esta declaración:

```
class MallaRevolPLY : public MallaRevol
{
public:
    MallaRevolPLY( const std::string & nombre_arch, // nombre del archivo PLY
                    const unsigned          nprofiles ) ; // número de perfiles.
};
```

Para la lectura de los vértices hay que usar la función `LeerVerticesPLY`, que es similar a `LeerPLY` pero no lee las caras (solo lee los vértices). Tiene un parámetro de entrada que es el nombre del archivo sin el path. Se buscará en `materiales/pls` y si no está en `archivos-alumnos`. También hay un parámetro de salida (un vector de `glm::vec3` con los vértices del perfil). Esta función se debe usar para leer los vértices del archivo `peon.ply`, que está en la carpeta `materiales/pls`.

2.5.5. Clase: Cilindro, Cono, Esfera

Para implementar estos objetos de revolución, crearemos clases derivadas de `MallaRevol`. Cada una de estas clases aporta un constructor específico, que crea el correspondiente vector con el perfil original, y luego invoca al método `inicializar` para crear las tablas. Estas clases se deben declarar en `malla-revol.h` e implementarse en `malla-revol.cpp`

```
// clases mallas indexadas por revolución de un perfil generado proceduralmente

class Cilindro : public MallaRevol
{
```

```

public:
// Constructor: crea el perfil original y llama a inicializar
// la base tiene el centro en el origen, el radio y la altura son 1
Cilindro
(
    const int      num_verts_per // número de vértices del perfil original (m)
    const unsigned nprofiles,      // número de perfiles (n)
)
;
} ;

class Cono : public MallaRevol
{
public:
// Constructor: crea el perfil original y llama a inicializar
// la base tiene el centro en el origen, el radio y altura son 1
Cono
(
    const int      num_verts_per // número de vértices del perfil original (m)
    const unsigned nprofiles,      // número de perfiles (n)
)
;
} ;

class Esfera : public MallaRevol
{
public:
// Constructor: crea el perfil original y llama a inicializar
// La esfera tiene el centro en el origen, el radio es la unidad
Esfera
( const int      num_verts_per // número de vértices del perfil original (M)
  const unsigned nprofiles,      // número de perfiles (N)
)
;
} ;

```

2.5.6. Archivos PLY disponibles.

Para buscar otros modelos PLY con mallas de polígonos, se pueden visitar estas páginas:

- Stanford 3D scanning repository
[☞ http://graphics.stanford.edu/data/3Dscanrep/](http://graphics.stanford.edu/data/3Dscanrep/)
- Sitio web de John Burkardt en Florida State University (FSU)
[☞ http://people.sc.fsu.edu/~jburkardt/data/ply/ply.html](http://people.sc.fsu.edu/~jburkardt/data/ply/ply.html)
- Sitio web de Robin Bing-Yu Chenen la National Taiwan University (NTU)
[☞ http://graphics.im.ntu.edu.tw/~robin/courses/cg03/model/](http://graphics.im.ntu.edu.tw/~robin/courses/cg03/model/)

2.6. Ejercicios adicionales

2.6.1. Ejercicio 1

(convocatoria ordinaria curso 20-21)

Extiende los archivos **malla-ind.h** y **malla-ind.cpp** de tus prácticas para incluir (al final) las declaraciones e implementaciones de la nueva clase **PiramideEstrellaZ**.

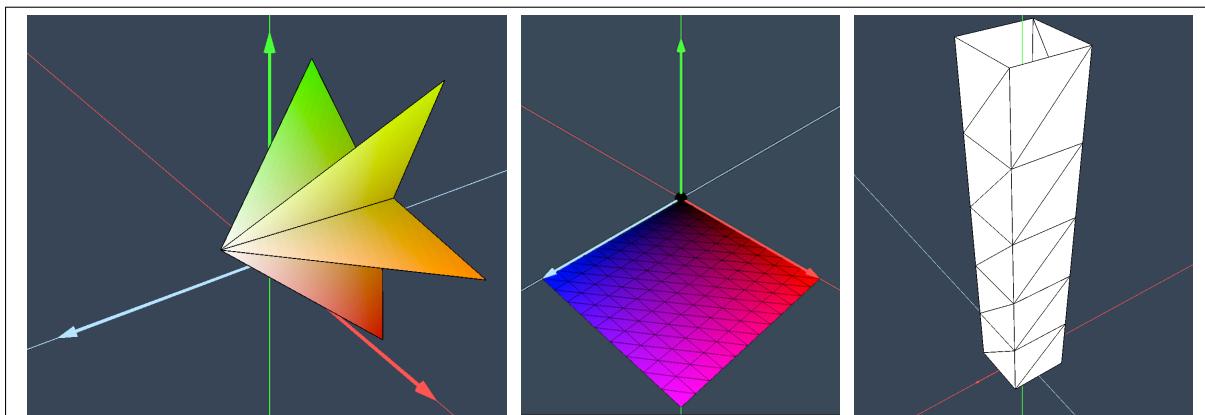


Figura 2.6: Objetos mencionados en los ejercicios adicionales de la práctica 2. A la izquierda el objeto de la clase **PiramideEstrellaZ** del ejercicio adicional 1, en el centro el objeto de clase **RejillaY** del ejercicio 2, a la derecha el objeto de la clase **Torre** del ejercicio 3.

Esta es una nueva clase, derivada de **MallaInd**, cuyo constructor acepta un parámetro n (**unsigned**, > 1). El constructor inicializa las tablas de vértices, triángulos y colores de una malla indexada con $4n$ triángulos y $2n + 2$ vértices, en forma de pirámide, con eje el eje Z, y cuya base es idéntica a la estrella descrita en el ejercicio adicional 1 de la práctica 1. El ápice de la pirámide es el único vértice adicional, tiene coordenadas $(0,5, 0,5, 0,5)$ y color blanco. En la figura 2.6 (izquierda) se observa el objeto.

Añade (como primer objeto de la escena de la práctica 2) una instancia de **PiramideEstrellaZ**.

2.6.2. Ejercicio 2

(convocatoria ordinaria curso 20-21)

Extiende los archivos **malla-ind.h** y **malla-ind.cpp** de tus prácticas para incluir (al final) las declaraciones e implementaciones de la nueva clase **RejillaY**.

Es una clase derivada de **MallaInd**, cuyo constructor acepta dos parámetros **unsigned**, ambos mayores estrictos que 1 (los llamamos m y n). El constructor crea una malla indexada compuesta de una rejilla, donde cada celda es un rectángulo formado por dos triángulos adyacentes. Todos los vértices tienen coordenada Y igual a 0, y las coordenadas X y Z están entre 0 y 1. La figura completa es cuadrada con lado unidad. Hay $m - 1$ celdas en una dirección y $n - 1$ en la otra. Por tanto, en total tiene nm vértices y $2(n - 1)(m - 1)$ triángulos.

Cada vértice tiene un color RGB cuyas componentes son iguales a sus coordenadas XYZ.

En la figura 2.6 (centro) se observa el objeto. Añade (como primer objeto de la escena de la práctica 2) una instancia de **RejillaY**

2.6.3. Ejercicio 3

(convocatoria extraordinaria curso 20-21)

Define al final de **malla-ind.h** y **malla-ind.cpp** la clase **MallaTorre**, derivada de **MallaInd**. El constructor de esta acepta un parámetro entero n y crea una malla indexada con $4(n + 1)$ vértices y $8n$ triángulos, con estos requerimientos:

- La malla está formada por n secciones o plantas puestas una encima de la otra.
- Cada sección esta formada por cuatro caras cuadradas de lado unidad (como las caras laterales de un cubo). Cada una de esas caras son dos triángulos (en total 8 triángulos por planta).
- La torre no tiene los triángulos de la base ni el techo.
- No se puede usar el código para generar objetos por revolución.

Se valora con el 60 % de la nota el hacer bien la tabla de coordenadas de vértices. Añade una instancia de la torre como único objeto de la práctica 2. Sube a prado un archivo llamado exactamente **p2.zip** con estos fuentes. En la figura 2.6 (derecha) se observa una vista del objeto, para $n = 5$.

3. Modelos jerárquicos.

3.1. Objetivos

Con esta práctica el alumno aprenderá a:

- Diseñar modelos jerárquicos parametrizados de objetos articulados.
- Implementar los modelos jerárquicos mediante estructuras de datos en memoria, incluyendo métodos para fijar valores de los parámetros o grados de libertad.
- Gestionar y usar una pila de transformaciones *modelview*.
- Implementar el control interactivo de los parámetros o grados de libertad.
- Implementar animaciones sencillas basadas en los grados de libertad.

3.2. Desarrollo.

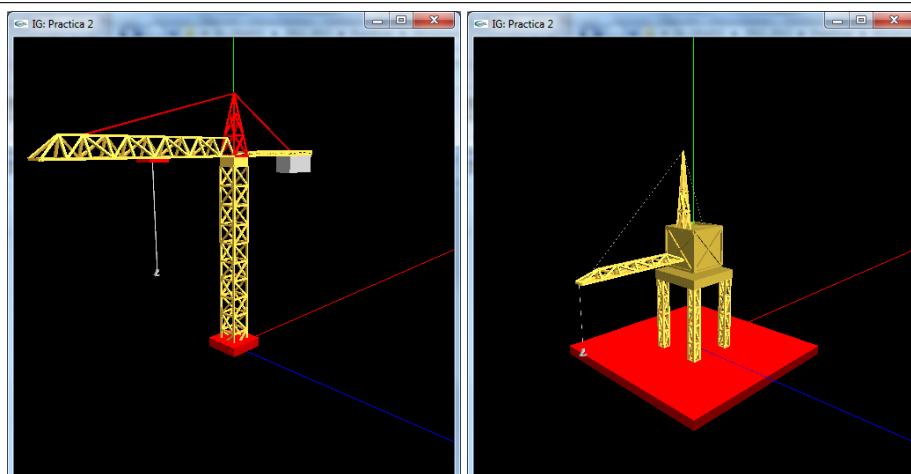


Figura 3.1: Ejemplos del resultado de la práctica 3.

En esta práctica se debe diseñar un modelo jerárquico con al menos 3 grados de libertad distintos (al menos deben aparecer giros y desplazamientos). Se puede tomar como ejemplo el diseño de una grúa semejante a las del ejemplo (ver figura 3.1). En el ejemplo, estas grúas tienen al menos tres grados de libertad: ángulo de giro de la torre, giro del brazo y altura del gancho. El modelo debe incluir las mallas indexadas creadas en las prácticas anteriores u otras de nueva creación (ver sección 3.7).

El diseño del modelo se debe materializar en un grafo de escena (tipo PHIGS) creado según la notación introducida en teoría, en un archivo PDF con un informe sobre el mismo (que tendrás que entregar cuando entregues la práctica). El archivo PDF debe incluir todas las transformaciones, indicaciones sobre los parámetros o grados de libertad, referencias a las mallas indexadas usadas como nodos terminales y una imagen de tu modelo en pantalla (ver subsección 3.7.1).

Se debe escribir el código necesario para visualizar un modelo jerárquico cualquiera, completando los métodos `visualizarGL` y `visualizarGeomGL` de la clase `NodoGrafoEscena` en el archivo `grafo-escena.cpp` (ver subsecciones 3.5.1 y 3.5.2). También es necesario completar el

método **agregar** y el método **leerPtrMatriz** de esa misma clase.

Para implementar el grafo de escena diseñado, se deben crear clases derivadas de **NodoGrafoEscena**, clases que añaden nuevos constructores, los cuales tendrán el código que agrega las entradas correspondientes en el array de entradas del nodo, y registra los punteros a las matrices asociadas a los parámetros.

Para la clase correspondiente al nodo raíz (al menos), también se deben implementar los métodos virtuales **leerNumParametros** y **actualizarEstadoParametro** (este último usa los punteros registrados para actualizar una matriz asociada a un parámetro en función del tiempo transcurrido). Estos métodos permiten modificar por teclado los parámetros del modelo y hacer las animaciones.

Para controlar los parámetros y animaciones se usa la tecla **A**, y por tanto, en la función gestora del evento de teclado (función **FGE_PulsarLevantarTecla** en el archivo **eventos-teclado.cpp**), cuando está pulsada la tecla **A** (y además se ha pulsado alguna otra tecla), se debe insertar una llamada a la función que procesa las teclas de animación (función **ProcesaTeclaAnimacion** en el archivo **animacion.cpp**, que ya está implementada).

3.3. Tareas.

A modo de resumen o referencia, estas son las tareas concretas que debe completar el estudiante:

1. En la clase **NodoGrafoEscena** hay que realizar las siguientes tareas:
 - Implementar la visualización de un nodo del grafo de escena en el método **visualizarGL**, ver subsección 3.5.1.
 - Implementar la visualización de un nodo, pero ignorando los colores en el método **visualizarGeomGL**, ver subsección 3.5.2).
 - Escribir el código del método **agregar** para añadir una entrada a un nodo, y devolver el índice de esa entrada. Ver subsección 3.5.3.
 - Escribir el código del método **leerPtrMatriz** para leer un puntero a una matriz en una entrada de un nodo (de tipo transformación). Ver subsección 3.5.4.
2. En la función gestora de eventos de teclado (**FGE_PulsarLevantarTecla**), añadir el código de soporte para las pulsaciones de teclas relacionadas con las animaciones de los grados de libertad del grafo. Ver sección 3.4.
3. Diseñar e implementar un grafo de escena parametrizado original, y escribir un informe (en PDF) sobre dicho diseño (ver sección 3.7). Esto incluye:
 - Declarar al menos una clase nueva, derivada de **NodoGrafoEscena**.
 - Implementar el constructor o constructores para crear el grafo asociado.
 - Implementar los métodos **leerNumParametros** y **actualizarEstadoParametro** de la(s) clase(s) nueva(s).
 Para más detalles, ver la sección 3.7.
4. Escribir la sentencia para añadir al vector de escenas una instancia de la clase **Escena3** (en el constructor de la clase **AplicacionIG**, archivo **aplicacion-ig.cpp**).

3.4. Teclas a usar. Gestión de animaciones.

En la función gestora de eventos de teclado (función **FGE_PulsarLevantarTecla** del archivo **eventos-teclado.cpp**) se detecta si está pulsada la tecla **A** cuando se ha pulsado otra, en ese caso se llama a una función gestora específica (**ProcesarTeclaAnimacion**), que ya está imple-

mentada en el archivo `animacion.cpp`. En esta función se gestionan pulsaciones de teclas que ocurren cuando está la tecla **A** pulsada, esas pulsaciones corresponden al control de los parámetros y las animaciones. El significado de cada tecla depende de si las animaciones están activadas o no.

Con las animaciones desactivadas, se usan las siguientes teclas:

- tecla **+** (en el teclado numérico): activa las animaciones.
- tecla **flecha izquierda**: activar el parámetro anterior (o el último).
- tecla **flecha derecha**: activar el siguiente parámetro (o el primero).
- tecla **flecha abajo**: incrementar el valor de tiempo del parámetro actual del objeto actual.
- tecla **flecha arriba**: decrementar el valor de tiempo del parámetro actual del objeto actual.

y con las animaciones activadas, estas:

- tecla **-** (en el teclado numérico): desactiva las animaciones.
- tecla **flecha izquierda**: activar el parámetro anterior (o el último).
- tecla **flecha derecha**: activar el siguiente parámetro (o el primero).

En el archivo `animaciones.cpp` se declara la variable lógica `animaciones_activadas`, que vale `true` cuando están activadas y `false` en otro caso. En ese archivo, la función `AnimacionesActivadas` permite consultar esa variable desde `aplicacion-ig.cpp`, en concreto desde el método que tiene el bucle principal (`buclePrincipalGLFW`). Si las animaciones están activadas y el objeto actual tiene algún parámetro animable, entonces en el bucle principal invoca, en cada iteración, a `ActualizarEstado` (en `animacion.cpp`) para actualizar el estado del objeto actual antes de volver a visualizar el cuadro.

En la función gestora del evento de teclas (`FGE_PulsarLevantarTecla`) ya hay código que detecta si la tecla **A** está pulsada en el momento de levantar otra tecla, si lo está es que el usuario quiere cambiar el estado de las animaciones. Se debe de completar el código que procesa la tecla pulsada, para ello se debe invocar la función `ProcesarTeclaAnimacion` (definida en `animaciones.cpp`), si devuelve `true`, se debe forzar revisualizar escena (asignando valor a `revisualizar_escena`).

Nota: en la plantilla entregada se aceptan las teclas **+** y **-** para activar o desactivar las animaciones, respectivamente, pero solo si se pulsan en el teclado numérico. Los códigos de teclas de GLFW para esas dos teclas son `GLFW_KEY_KP_ADD` y `GLFW_KEY_KP_SUBSTRACT`. Para ordenadores sin teclado numérico será necesario añadir otras teclas fuera de dicho teclado numérico, por ejemplo, la tecla con los símbolos ***** **+** **]** para activar las animaciones (código `GLFW_KEY_RIGHT_BRACKET`), y la tecla con los símbolos **-** **_** (código `GLFW_KEY_SLASH`) para desactivarlas. Esto puede hacerse modificando el código de la función `ProcesarTeclaAnimacion` en el archivo `animaciones.cpp`.

3.5. Visualización y gestión de nodos del grafo (clase `NodoGrafoEscena`)

Para poder visualizar los nodos del grafo de escena, es necesario completar la implementación de los métodos `visualizarGL` y `visualizarGeomGL` de la clase `NodoGrafoEscena`. Además, se deben completar otros dos métodos (`agregar` y `leerPtrMatriz`). Aquí se dan los detalles.

3.5.1. El método `visualizarGL`

Para completar la práctica es necesario implementar el método `visualizarGL` para los nodos del grafo de escena. Esta implementación recorre las entradas del nodo, y en cada entrada, si es un puntero a un sub-objeto, llamará recursivamente al método `visualizarGL` para ese sub-objeto,

y si es una matriz de transformación, debe componer la matriz con la modelview actual, usando la funcionalidad ofrecida por el cauce. Además, es necesario hacer *push* de la matriz *modelview* al inicio y *pop* al final. Todo esto se corresponde con lo que se ha explicado en teoría.

Además de esto, y al igual que en el caso de las malla indexadas, también se debe implementar la posibilidad de definir colores específicos de los objetos que sean nodos del grafo. Para ello, si el objeto nodo tiene color, usaremos los métodos para cambiar el color del cauce, y para guardar y restaurar una copia de dicho color. Cuando se visualiza un nodo que no tiene asignado color, se usará el color actual fijado en el cauce antes de la llamada a `visualizarGL`. Sin embargo, si el nodo tiene asociado un color entonces, antes de visualizar, se debe guardar el color actual del cauce y cambiarlo, y después de visualizar debemos de restaurarlo (de forma que en cualquier caso el color del cauce a la entrada es igual al color del cauce a la salida)

Así que el estudiante debe escribir código para dar estos pasos:

1. Si el objeto tiene un color asignado (se comprueba con `tieneColor`), hay que:
 - 1.1. hacer push del color actual del cauce (con `pushColor`) y después
 - 1.2. fijar el color en el cauce (con `fijarColor`) usando el color del objeto (se lee con `leerColor`).
2. Guardar copia de la matriz de modelado (con `pushMM`),
3. Para cada entrada del vector de entradas:
 - Si la entrada es de tipo objeto: llamar recursivamente a `visualizarGL`
 - Si la entrada es de tipo transformación: componer la matriz (con `compMM`)
4. Restaurar la copia guardada de la matriz de modelado (con `popMM`)
5. Si el objeto tiene color asignado, hay que restaurar el color original (con `popColor`).

3.5.2. El método `visualizarGeomGL`

Al igual que en las mallas indexadas, en los nodos del grafo el método `visualizarGeomGL` se debe encargar de visualizar únicamente la geometría de la malla indexada, sin usar colores, normales, texturas, y sin cambiar nunca el color actual en el cauce (se usa el que haya puesto en el momento de la llamada). Este método se invoca cuando se quiera visualizar las aristas del objeto actual de la escena.

El estudiante debe completar el código del método `visualizarGeomGL`. La implementación es similar a `visualizarGL`, pero más simple, ya que ahora se ignoran los colores de los objetos (y también, en la práctica 4, se ignorarán los materiales), y por tanto debemos simplemente de tener en cuenta la geometría, es decir, hacer *push* y *pop* de la matriz de modelado, y recorrer las entradas dibujando los sub-objetos y componiendo las matrices.

Por tanto, el estudiante debe escribir código para dar estos pasos:

1. Guardar copia de la matriz de modelado (con `pushMM`),
2. Para cada entrada del vector de entradas:
 - Si la entrada es de tipo objeto: llamar recursivamente a `visualizarGeomGL`.
 - Si la entrada es de tipo transformación: componer la matriz (con `compMM`).
3. Restaurar la copia guardada de la matriz de modelado (con `popMM`)

3.5.3. El método `agregar`

Este método (la versión que acepta como parámetro un `NodoGrafoEscena`) debe ser escrito por el estudiante, se trata simplemente de añadir (con `push_back`) la entrada al vector de entradas

del nodo, y luego devolver el índice de la entrada añadida (que será la última, y por tanto su índice coincide con el tamaño del vector menos uno).

3.5.4. El método `leerPtrMatriz`

Este método devuelve un puntero a la matriz de transformación que hay en una entrada dada del vector de entradas, a partir del índice de dicha entrada. En el método es necesario escribir código para comprobar que el índice no está fuera de rango (es decir, no es mayor o igual que el número de entradas del mismo), que la entrada es de tipo transformación, y que el puntero que hay en ella no es nulo. Si se dan esos requisitos, se devuelve el puntero.

3.6. Implementación de parámetros y animaciones

Si una clase derivada de `Objeto3D` representa un objeto parametrizado se asume que cada instancia de esa clase tiene asociado un número $n > 0$ de parámetros o grados de libertad (n es arbitrario, mayor que cero, puede ser distinto en cada instancia, pero en una instancia concreta no cambia nunca durante el tiempo de vida de dicha instancia).

A cada parámetro se le identifica por un **índice de parámetro**, que es un entero entre 0 y $n - 1$. Además, cada instancia de una clases parametrizada tiene asociado un **valor de tiempo** (t_i) para cada parámetro i . El valor de tiempo es un valor real en unidades de segundos, no necesariamente el mismo para cada parámetro. Cada clase parametrizada incluye código específico que permite modificar una instancia (la geometría de la instancia), en función del índice i de un parámetro y el valor de tiempo actual t_i de dicho parámetro i , a eso le llamamos *actualizar el estado del parámetro i al valor de tiempo t_i* .

Al crear un objeto parametrizado, todos los valores t_i son cero. Cuando se activan las animaciones y se está visualizando un objeto (el objeto actual de la escena actual), el bucle principal se encarga en cada iteración de incrementar el valor de tiempo de cada uno de los parámetros del objeto, según el tiempo real Δt transcurrido desde la última actualización o desde el inicio de las animaciones. Es decir, durante las animaciones, en cada cuadro se hace $t_i + \Delta t$, para cada índice de parámetro i del objeto que se está visualizando. Si el objeto actual tiene 0 parámetros, no se hace nada.

Además, cuando las animaciones están desactivadas, es posible modificar (incrementar o decrementar) el valor de tiempo de cualquier parámetro, según una constante k (podemos hacer t_i igual a $t_i + k$ o a $t_i - k$). El hecho de que los valores de tiempo se pueden cambiar manualmente de forma individual es lo que implica que no siempre todos los valores de tiempo sean el mismo para todos los parámetros de una instancia.

En esta sección se detalla como definir clases parametrizadas derivadas de `Objeto3D`, con énfasis en clases derivadas de `NodoGrafoEscena`, donde los parámetros o grados de libertad se concretan en matrices situadas en algún nodo, matrices que dependen del valor de tiempo de un parámetro. Por ejemplo, se puede diseñar un grafo de escena con un nodo que rota mediante una matriz de rotación, de forma que el ángulo de esa rotación es proporcional al valor de tiempo de un parámetro. Igual puede hacerse con desplazamientos o escalados dependientes del tiempo.

Lo típico es que un parámetro afecte a una matriz, pero a veces es necesario hacer depender más de una matriz de un único parámetro, cuando esas matrices están relacionadas. Por ejemplo, si queremos avanzar un coche por una carretera, girando sus ruedas sin que derrapen, en cada instante el ángulo de rotación de las ruedas y el desplazamiento del coche no son independientes, en este caso tendríamos una matriz de rotación y una de traslación que dependen de un único valor de tiempo.

3.6.1. Definición de clases para objetos parametrizados

La clase **Objeto3D** incorpora varios métodos (algunos de ellos virtuales), que se usan para gestionar los parámetros o grados de libertad y las animaciones. En las clases derivadas de **Objeto3D** se pueden redefinir los dos métodos virtuales para implementar parámetros y animaciones para las instancias de esa clase.

El primer método virtual (redefinible) de **Objeto3D** que veremos es **leerNumParámetros**: por defecto devuelve 0, pero las clases derivadas pueden redefinir el método y devolver un valor distinto. Es el número de parámetros o grados de libertad del objeto, un valor entero sin signo que en cada objeto concreto no varía durante la ejecución. Si una clase no redefine el método, al devolver 0 se asume que las instancias de esa clase no tienen parámetros. Cada objeto que devuelve un valor no nulo tiene asociados una serie de parámetros, cada uno de ellos tiene un índice y un valor de tiempo actual. Los valores de tiempo de un objeto se almacenan en un vector de flotantes declarado en **Objeto3D**, ese vector está por defecto vacío, pero si un objeto tiene parámetros, se crea el vector con una entrada por parámetro y se inicializa la entrada a cero (esto se hace *automáticamente*, antes de la primera vez que se quiera actualizar el estado de un parámetro)

El otro método virtual, redefinible, es **actualizarEstadoParametro** (no devuelve nada). Este método tiene como parámetros un índice de parámetro (**iParam**) y un real que representa un tiempo en unidades de segundos (**t_sec**), que es el valor de tiempo actual del parámetro con índice **iParam**. Por defecto, llamar a este método produce un error y aborta el programa. Si una clase redefine este método, entonces debe implementarse de tal forma que en **iParam** (índice de parámetro) espera valores entre 0 y $n - 1$, donde n es el valor que ese mismo objeto devuelve en **leerNumParametros**. La implementación debe entonces actualizar el estado del objeto para reflejar que el parámetro o grado de libertad designado se ha actualizado a un valor de tiempo. Esta actualización es arbitraria, pero en las prácticas consiste, en concreto, en actualizar una o varias matrices de los nodos de un grafo de escena.

Los dos métodos citados son los únicos que hay que implementar para definir una clase para objetos parametrizados y animables. Además de esto, la clase **Objeto3D** incluye una serie de métodos (ya implementados, no virtuales) para facilitar la gestión de los parámetros. Son estos:

- **modificarIndiceParametroActivo (d)**. Cada instancia de **Objeto3D** tiene un entero que el índice del parámetro activo, y designa a uno de sus parámetros (initialmente es 0). Este método permite incrementarlo o decrementarlo en una unidad (haciendo **d==+1** o **d==−1**) módulo el número de parámetros, de forma que cambia el parámetro activo del objeto
- **modificarParametro (i, d)**. Permite sumar el valor **d*k** al valor de tiempo del parámetro con índice **i**, y después llama a **actualizarEstadoParametro**. El valor **d** es un entero, típicamente +1 o −1. Se usa para modificar *manualmente* un parámetro.
- **modificarParametro (d)**. Igual que el anterior, pero afecta al parámetro activo actual del objeto.
- **actualizarEstado (d)**. Suma el valor real **d** a todos y cada uno de los valores de tiempo de los parámetros del objeto, y llama a **actualizarEstadoParametro**. Se usa durante las animaciones para actualizar el estado de un objeto una vez que ha transcurrido un intervalo de tiempo **d**.

Cuando cualquiera de estas funciones se invoca, se comprueba al inicio del método si la tabla de valores de tiempo del objeto está creada, y si no lo está se crea y se inicializan los valores a cero.

3.6.2. Definición de nodos del grafo de escena parametrizados

Para diseñar una clase para nodos de grafo de escena parametrizados, debemos de seleccionar que matriz (o matrices) del grafo dependen de cada parámetro, y además como depende cada matriz del valor de tiempo de su correspondiente parámetro. Respecto de esta dependencia, hay infinitas opciones, pero en estas prácticas se pueden usar dos de ellas, muy simples pero bastante versátiles. En concreto, se puede hacer depender un real v (que representa un ángulo de rotación o una distancia de desplazamiento o un factor de escala) de un valor de tiempo t de estas dos formas:

- **Linealmente:** hacemos $v = a + bt$.

Usamos dos valores a (valor inicial de v) y b (tasa de cambio de v por segundo). Se puede usar típicamente para rotaciones de velocidad angular constante, si v es un ángulo en radianes y b es $2\pi w$, donde w es la velocidad angular en ciclos por segundo. Se puede usar para traslaciones y escalados, pero no es aconsejable ya que el desplazamiento o el factor de escala crecerían indefinidamente con el tiempo.

- **Oscilante:** hacemos $v = a + b \sin(2\pi nt)$

Aquí $a = (v_{min} + v_{max})/2$ y $b = (v_{max} - v_{min})/2$. Ahora el valor v oscila entre v_{min} y v_{max} , con un número n (flotante) de oscilaciones por segundo. Se puede usar para rotaciones, traslaciones y escalados, de forma que sabemos que el valor v siempre está acotado por v_{min} y v_{max} .

Una vez que se ha calculado v se debe recalcular la matriz (o matrices) que depende de v en el grafo de escena, como corresponda.

En el caso de objetos de la clase **NodoGrafoEscena** (o sus derivadas) que sean objetos parametrizados, es posible implementarlos como se ha indicado, redefiniendo en esas clases los métodos **leerNumParametros** y **actualizarEstadoParametro**.

El método **actualizarEstadoParametro** contiene típicamente un **switch** que ejecuta código distinto en función de **iParam** (el índice del parámetro). En cada caso se recalcula una o varias matrices y se sobrescriben en el grafo de escena.

Hay que tener en cuenta que **actualizarEstadoParametro** debe usar punteros a las matrices que debe sobrescribir. Para ello se deben de registrar esos punteros como variables de instancia, es decir, durante la ejecución del constructor se deben de guardar en variables de instancia (de tipo puntero a **glm::mat4**) los punteros correspondientes. Cada vez que se añade una matriz al grafo, se puede obtener el puntero (con el método **leerPtrMatriz** de **NodoGrafoEscena**) usando el índice de la entrada (que es el valor devuelto por **agregar**).

El método **leerPtrMatriz** se debe implementar por el alumno.

3.7. Diseño e implementación de un grafo de escena parametrizado original

Se debe diseñar el grafo de escena y plasmarlo en un archivo PDF. Intenta hacer un diseño lo más completo posible de entrada, aunque es probable que durante la implementación del diseño te des cuenta de que debes cambiar dicho diseño. Al final, el archivo PDF tendrá los contenidos que se indican en la sección 3.7.1.

Tras diseñar el grafo de escena, para implementarlo debemos de definir una clase específica para dicho objeto. La clase (la llamamos, a modo simplemente de ejemplo, C) es una clase derivada de

NodoGrafoEscena. El nodo raíz de nuestro grafo de escena será un objeto de la clase *C*, y por tanto contiene todos los punteros a las matrices asociadas a los grados de libertad. Lo único que en principio debe de añadirse a *C* es un constructor. En dicho constructor se crean los subárboles del nodo raíz y se van añadiendo a la lista de entradas.

La clase *C* y el resto de clases necesarias se definirán en un archivo nuevo llamado `modelo-jer.cpp` y se declarará en el archivo `modelo-jer.h` (ambos en la carpeta `src`). Para añadir un nuevo archivo `.cpp` a la lista de archivos que se compilan, hay que vaciar y regenerar (con `cmake ..`) la correspondiente carpeta `cmake`. Al regenerarlo, se consideran todos los archivos `.cpp` que haya en la carpeta `src`, con lo cual se tiene en cuenta el nuevo `.cpp`.

La clase *C* debe redefinir los métodos `leerNumParametros` y `actualizarEstadoParametro` como corresponda al grafo de escena diseñado, según se ha indicado en las secciones anteriores.

3.7.1. Contenidos del archivo PDF con la documentación.

El archivo también debe de incluir la siguiente información:

- En la primera página, nombre de la asignatura y el curso académico, nombre y apellidos del autor, su titulación.
- Una captura de pantalla del modelo, donde se aprecien lo mejor posible todas las partes del mismo.
- El grafo de escena tipo PHIGS tal y como se ha visto en clase.
- Una lista con información de todos y cada uno de los nodos del grafo, para cada uno de ellos:
 - Nombre de la clase (si se ha definido una clase) o identificador de la variable u objeto (si no se ha definido una clase para ese nodo y simplemente se crea una instancia en el constructor del padre) (indicar si es una clase o un objeto). Estos nombres deben ser únicos.
 - Si el nodo tiene asociados parámetros o grados de libertad, los nombres de todos y cada uno de los parámetros (a cada parámetro del grafo se le asociará un nombre o identificador único).
 - Si tiene un color específico, color del nodo (como una terna RGB).
 - Nombre de los archivos `.h` y `.cpp` donde está declarada y definida la clase asociada al nodo o donde está el código que lo construye (si no tiene asociada una clase). Rango de líneas en el `.cpp` donde está el constructor de la clase o el código que construye el nodo.
- Una lista con información de todos y cada uno de los parámetros o grados de libertad del grafo, para cada uno de ellos:
 - Nombre o identificador único del parámetro o grado de libertad.
 - Nombre del nodo o los nodos donde está la matriz o matrices que dependen del parámetro.
 - Para cada una de esas matrices (normalmente es una única pero podrían ser más):
 - Breve descripción en texto sobre cómo cambia la matriz o matrices con el tiempo (por ejemplo: *es un desplazamiento oscilante en el eje X, con un período de 2 segundos y una amplitud de 1.4 unidades de distancia, o bien rotación entorno al centro del cubo, con una frecuencia de tres revoluciones por segundo*).
 - Para cada una de esas matrices, indicar la expresión que construye la matriz a partir del tiempo *t* en segundos (esa expresión se debe corresponder con el código que la actualiza). Usar la notación para las matrices que se ha visto en teoría.

3.7.2. La clase Escena3

Para poder visualizar el objeto jerárquico modelado es necesario declarar la clase **Escena3** (en **escena.h**) e implementar su constructor (en el archivo **escena.cpp**), al igual que hicimos con las clases **Escena1** y **Escena2**. En ese constructor debemos de añadir una instancia de la clase *C* a la lista de objetos de la escena.

Además de lo anterior, en el constructor de **AplicacionIG** (archivo **aplicacion-ig.cpp**) se debe de añadir una instancia de **Escena3** en el vector de escenas (**escenas**).

3.8. Algunos ejemplos de modelos jerárquicos

En las figuras desde la 3.3 hasta la 3.9 podéis ver algunos ejemplos de modelos jerárquicos que se pueden construir para la práctica (simplificando todo lo que se quiera los distintos elementos que los componen).

Estudiar con detalle cada uno y seleccionar el que os interese, o diseñar otro que tenga al menos 3 grados de libertad similares a los de las gruas que tenéis en el ejemplo.

3.9. Ejercicios adicionales

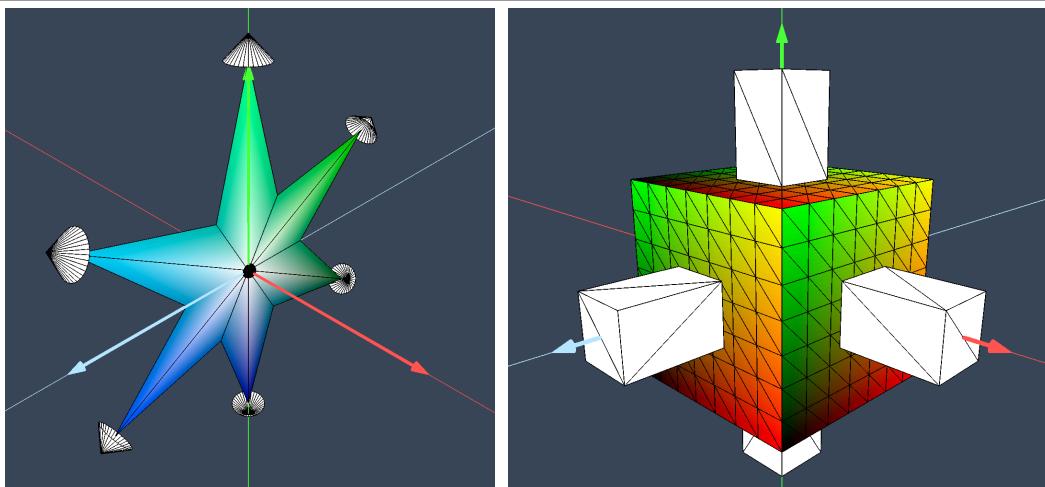


Figura 3.2: Objetos mencionados en los ejercicios adicionales de la práctica 3. A la izquierda el objeto de la clase **GrafoEstrellaX** del ejercicio adicional 1, a la derecha el objeto de la clase **GrafoCubos** del ejercicio 2 (los colores que tengan tus objetos pueden ser distintos a los que aparecen aquí)

3.9.1. Ejercicio 1

(convocatoria ordinaria curso 20-21)

Extiende los archivos **grafo-escena.h** y **grafo-escena.cpp** de tus prácticas para incluir (al final) las declaraciones e implementaciones de la clase **GrafoEstrellaX**, con estas especificaciones:

- El constructor de la clase acepta un parámetro llamado *n* (unsigned, > 1)

- En el constructor se construye un grafo de escena que tiene una instancia del objeto con una estrella plana de la práctica 1 (usa en tu grafo de escena una instancia de la misma clase que has creado para la práctica 1 en este examen, instanciala con el número de puntas indicado por n). Esa estrella es perpendicular al eje X. Tiene centro en el origen y el radio desde el centro a los vértices en las puntas es 1.3.
- En cada punta de la estrella hay una instancia de la clase **Cono** de la práctica 2, con radio de la base 0.14, y altura 0.15, y cuyo eje es el segmento que hace de radio desde el centro de la estrella a esa punta (intenta usar una única instancia del cono para todas las puntas, instanciada con distintas transformaciones)
- El objeto tiene un parámetro o grado de libertad, de forma que se puede rotar entorno al eje X (entorno al centro de la estrella). Si se activan las animaciones, gira a una velocidad de 2.5 vueltas por segundo.

En la figura 3.2 (izquierda) se observa el objeto jerárquico (los colores de tu estrella no tienen porque coincidir con los de la figura).

Añade una instancia de esta clase en el constructor de **Escena3**.

3.9.2. Ejercicio 2

(convocatoria ordinaria curso 20-21)

Extiende los archivos **grafo-escena.h** y **grafo-escena.cpp** de tus prácticas para incluir (al final) las declaraciones e implementaciones de la clase **GrafoCubos**, con estas especificaciones:

- La clase tiene un constructor sin parámetros, y usa la clase **Rejillay** (de los ejercicios adicionales de la práctica 2) y la clase **Cubo** (de la práctica 1).
- El objeto jerárquico tiene un cubo central, cuyas 6 caras son todas instancias de un único objeto de la clase rejilla de este examen. Ese cubo central tiene centro en el origen. En el centro de cada una de esas seis caras hay un cubo más pequeño. Esos cubos se construyen como 6 instancias de un objeto de la clase **Cubo**. Cada uno de esos cubos está alargado en la dirección de la línea que va desde su centro al origen (son paralelepípedos).
- El grafo tiene un único grado de libertad o parámetro. Cuando se anima, cada cubo pequeño rota entorno al eje que pasa por su centro y el origen (todos a la vez, ya que todos rotan con el mismo ángulo, al haber únicamente un grado de libertad).

Intenta crear un grafo con el mínimo número de nodos. Ten en cuenta que los cubos o paralelepípedos pequeños no entran dentro del grande.

En la figura 3.2 (derecha) se observa el objeto jerárquico (los colores de los vértices o de los objetos en tu solución no tienen porque coincidir con los de la figura).

Añade una instancia de esta clase en el constructor de **Escena3**.



Figura 3.3: Ejemplos de posibles modelos jerárquicos (1 de 7)



Figura 3.4: Ejemplos de posibles modelos jerárquicos (2 de 7)



Figura 3.5: Ejemplos de posibles modelos jerárquicos (3 de 7)



Figura 3.6: Ejemplos de posibles modelos jerárquicos (4 de 7)



Figura 3.7: Ejemplos de posibles modelos jerárquicos (5 de 7)



Figura 3.8: Ejemplos de posibles modelos jerárquicos (6 de 7)



Figura 3.9: Ejemplos de posibles modelos jerárquicos (7 de 7)

4. Materiales, fuentes de luz y texturas.

4.1. Objetivos

Con esta práctica el alumno aprenderá a:

- Incorporar a los modelos de escena información de aspecto o apariencia: normales y coordenadas de textura de vértices en las mallas indexadas, modelos sencillos de materiales y texturas en el grafo de escena, y modelos sencillos de las fuentes de luz.
- Escribir código de visualización que contemple no solo los modelos geométricos sino también los modelos de aspecto.
- Diseñar una escena incluyendo varios objetos con distintos modelos de aspecto.
- Permitir cambiar de forma interactiva algunos de los parámetros de los modelos de aspecto.

4.2. Desarrollo

En esta práctica incorporaremos a las mallas indexadas información de las normales y las coordenadas de textura, para eso el estudiante escribirá el código que calcula las normales de cada uno de los triángulos, y a partir de esas normales, calculará (promediando) las normales de todos los vértices.

Para poder verificar que las normales se están calculando de forma correcta, se escribirá el código que visualiza las normales de las mallas y los nodos del grafo de escena.

También se incorporará la capacidad de activar la iluminación, de forma que se usarán una o varias fuentes de luz, el modelo de iluminación local sencillo visto en teoría y los materiales y sus texturas. Eso supone definir y usar las clases que encapsulan los parámetros de las fuentes de luz, de colecciones de fuentes de luz, de los materiales y dentro de estos sus texturas.

Los estudiantes escribirán el código de activación de colecciones de fuentes de luz, materiales y texturas, es decir, el código que configura el cauce gráfico para que a partir de su activación se use un determinado material o una determinada colección de fuentes de luz.

Para poder usar texturas es necesario incluir una tabla de coordenadas de textura en los VAO, o bien activar la generación automática de coordenadas de textura. Los estudiantes escribirán código para generar modelos de mallas con tablas de coordenadas de textura, bien dando esas coordenadas explícitamente (para objetos sencillos), bien generándolas mediante un algoritmo (para los objetos de revolución). También se escribirá el código de activación de la generación de coordenadas de textura.

La asociación de diferentes materiales a los objetos en un grafo de escena se consigue contemplando entradas de tipo material en dichos nodos. Los estudiantes tendrán que tener en cuenta esas entradas para extender el código de visualización de los grafos de escena, que ahora las debe procesar adecuadamente.

El estudiante deberá añadir información sobre los materiales usados en el grafo de escena al documento PDF que describe dicho grafo y que fue producido en la práctica 3.

4.3. Tareas

A modo de referencia, aquí se incluyen las diversas tareas concretas que los estudiantes deben de realizar en esta práctica:

1. Se implementarán diversas estrategias para añadir tablas de normales y coordenadas de textura a las distintas clases de objetos de tipo malla indexadas (mallas leídas de un PLT, ver sección 4.6.3, y mallas de revolución, ver sección 4.6.2). Esta implementación se hará en los constructores de las clases derivadas de **MallaInd**, en algunos casos invocando métodos específicos.
2. Se añadirá código a las clases **MallaInd**, **NodoGrafoEscena** y **Escena** para visualizar las normales de vértices. Ver sección 4.5.7.
3. Se completará el código de las clases **Textura** (ver sección 4.5.2) y **Material** (sección 4.5.3), para cargar y enviar una textura a la GPU, para activar el uso de una textura o un material. Se completará asimismo el código de activación de la clase **ColFuentesLuz** (sección 4.5.1). Todo esto se hace en el archivo **materiales-luces.cpp**.
4. Se completará el código de la clase **Escena**, para añadir, en el constructor, la creación de una colección de fuentes y un material inicial. Se completará el método de visualización de dicha clase, para activar la iluminación, las fuentes y el material inicial (antes de visualizar los objetos, cuando está activada la iluminación), todo ello en **escena.cpp**. Para más detalles, ver la sección 4.5.6.
5. Se debe de añadir código en la función **FGE_PulsarLevantarTecla** (en el archivo **eventos-teclado.cpp**) para procesar las otras teclas que se pulsen cuando a la vez se mantiene pulsada la tecla **L**, invocando la función **ProcesarTeclaFuenteLuz** (ver sección 4.4).
6. Se creará una nueva clase (**Cubo24**) para una malla indexada con la geometría de un cubo, pero con una topología de 24 vértices, y con las normales a las caras y las coordenadas de textura correctas (ver sección 4.6.1).
7. Se creará una nueva clase (de nombre **LataPeones** en los archivos nuevos **latapeones.cpp** y **latapeones.h**) para un grafo de escena jerárquico que contiene objetos con distintos materiales y textura (la escena con la lata y los peones). La geometría y aspecto de esta escena se describe en este guion (ver sección 4.5.9), pero no los valores de los parámetros del material que producen ese aspecto.
8. Se definirá una nueva clase (**Escena4**) derivada de **Escena**, específica para la práctica 4, que contiene un objeto con el grafo jerárquico descrito en el punto anterior. La clase se declara en **escena.h** y se implementa en **escena.cpp**. Es necesario añadirla al vector de escenas en el constructor de **AplicacionIG** (archivo **aplicacion-ig.cpp**). Ver sección 4.5.8.
9. Se añadirán materiales y texturas al grafo de escena diseñado e implementado en la práctica 3, y se actualizará el archivo PDF con la documentación del grafo de escena para añadirle información de materiales. Ver sección 4.5.10.

4.4. Teclas a usar

Además del resto de teclas descritas en este guión, para esta práctica será necesario modificar (extender) la función **FGE_PulsarLevantarTecla** (en el archivo **eventos-teclado.cpp**). En dicha función, cuando se invoca (porque se ha pulsado alguna tecla) y se detecta que (además de la otra) la tecla **L** está pulsada, se debe de recuperar la colección de fuentes de luz de la escena actual (usando el método **colFuentes** de **Escena**, que devuelve un puntero), y des-

pués invocar la función `ProcesaTeclaFuenteLuz`, cuyo código ya está completo en el archivo `materiales-luces.cpp`. Cuando la función `ProcesarTeclaFuenteLuz` devuelve `true` indica que se ha cambiado algo de la configuración de las fuentes, y en ese caso se debe forzar a que se visualice la escena, poniendo a `true` la variable `revisualizar_escena` dentro del objeto `aplicacionIG`.

Las teclas que se procesan en la citada función permiten cambiar la fuente de luz actual de la colección de fuentes (siempre hay una *fuente actual* en una colección de fuentes), y cambiar su longitud y latitud (son los dos ángulos que forman las coordenadas esféricas del vector de dirección a dicha fuente). En concreto:

- Tecla **+**: la fuente de luz actual pasa a ser la siguiente fuente (o la primera)
- Tecla **-**: la fuente de luz actual pasa a ser la anterior fuente (o la última)
- Tecla **flecha izquierda**: incrementar el ángulo de longitud de la dirección de la fuente actual.
- Tecla **flecha derecha**: decrementar el ángulo de longitud de la dirección de la fuente actual.
- Tecla **flecha abajo**: decrementar el ángulo de latitud de la dirección de la fuente de luz actual.
- Tecla **flecha arriba**: incrementar el ángulo de latitud de la dirección de la fuente de luz actual.

Además de estas teclas, independientemente de que la tecla **L** este o no pulsada, se pueden usar estas otras dos:

- Tecla **I**: activa o desactiva la iluminación (inicialmente está activada)
- Tecla **F**: cambia entre uso de normales de triángulos y uso de normales interpoladas de vértices (inicialmente se usa interpolación).

4.5. Clases y métodos a añadir o completar.

En esta sección se incluye una descripción de las nuevas clases que se deben de crear en esta prácticas 4, y de los métodos que se deben crear o completar en las clases ya existentes.

4.5.1. Las clases `FuentesLuz` y `ColFuentesLuz`

Las clases `FuenteLuz` y `ColeccionFuenteLuz` ya están declaradas en el archivo `materiales-luces.h`, y parcialmente implementadas en el archivo `materiales-luces.cpp`.

La clase `FuenteLuz` encapsula los parámetros de una fuente de luz de tipo direccional. Las variables de instancia incluyen las coordenadas esféricas (ángulos de *longitud* y *latitud*) del vector de dirección que apunta a la fuente de luz. La fuente de luz puede manipularse interactivamente, cambiando ambos ángulos (con los métodos `actualizarLongi` y `actualizarLati`). El constructor admite como parámetros la longitud y latitud iniciales, así como el color de la fuente de luz. En esta práctica no es necesario extender el código de la clase.

La clase `ColFuentesLuz` encapsula un vector (una colección) de punteros a objetos de tipo `FuenteLuz`. Cuando se crea una de estas colecciones está inicialmente vacía. Podemos añadir fuentes de luz con el método `insertar`, ya implementado. Toda colección no vacía tiene siempre una *fuente de luz actual* que se puede consultar con `fuenteLuzActual` o modificar con `sigAntFuente`.

En esta práctica es necesario implementar el método `activar` de esta clase. Este método debe construir un vector con los vectores de dirección de las luces (calculados a partir de los dos ángulos

de cada una), y otro con los colores, y usar ambos vectores para invocar el método `fijarFuentesLuz` del cauce en uso (el cauce es un parámetro de `activar`). A partir de que una colección de fuentes se activa, las fuentes que incluye se usan para la evaluación del modelo de iluminación local.

4.5.2. Clase Textura

Las clases `Textura` y `Material` ya están declaradas en el archivo `materiales-luces.h`, y parcialmente implementadas en el archivo `materiales-luces.cpp`.

La clase `Textura` encapsula una imagen en memoria, con formato RGB, donde cada texel se codifica con tres bytes (3 datos de tipo `unsigned char`), dispuestos por filas, de abajo hacia arriba. Su constructor admite como parámetro el path y nombre (una cadena) del archivo de imagen, que debe ser de tipo JPEG. Se debe completar el código de dicho constructor, invocando a la función `LeerArchivoJPG`, que tiene como parámetro de entrada el path y nombre del archivo, como parámetros de salida el ancho y el alto, y como resultado un puntero a los bytes alojados en memoria dinámica.

También se debe completar el código para enviar la textura a la GPU, en el método `enviar`. Se debe crear un identificador de textura (que queda registrado como variable de instancia), y después enviar los bytes de la misma a la GPU (ver las transparencias de teoría).

Se debe de implementar el método `activar`, que se encarga de activar una textura en el cauce (a partir de su activación, se usará para todas las operaciones posteriores de visualización de objetos). Para esto debemos, en primer lugar, comprobar si la textura ya ha sido enviada a la GPU o no, y en caso negativo invocar el método `enviar`. Después se deben usar los métodos `fijarEvalText` y `fijarTipoGCT` del cauce gráfico para darle valor a las variables del cauce relacionadas con textura (activación de texturas, identificador de textura, modo de generación de coordenadas y coeficientes de generación).

Para asignar coordenadas de textura habrá que tener en cuenta que la librería que se usa para leer archivos JPEG (`libjpeg`) guarda en memoria las filas de arriba hacia abajo (es decir la primera fila en memoria es la fila superior de la imagen). Sin embargo, OpenGL interpreta las filas al revés, de abajo hacia arriba (asume que la primera fila en memoria es la inferior). Para solucionar esta discrepancia (sin tener que reorganizar las filas) se puede invertir el orden las coordenadas de textura (los detalles están en la sección 4.6.2).

En el archivo `materiales-luces.h` se declaran las clases `TexturaXY` y `TexturaXZ`, ambas derivadas de la clase `Textura`. Estas dos clases definen cada una un nuevo constructor de texturas, que deben ser implementados en `materiales-luces.cpp`. Esos constructores invocan el constructor de `Textura`, pero además configuran la textura de forma que se use generación automática de coordenadas de textura, en modo *coordenadas de objeto*.

En el caso de `TexturaXY`, las coordenadas de textura son proporcionales a las coordenadas X y Y de cada vértices. Igualmente, para la textura XZ, las coordenadas de textura serán proporcionales a la coordenadas X y Z del vértice. La generación automática de coordenadas de textura se usará para algunos de los objetos de esta práctica (ver figura 4.2, derecha). Puede que tengas que ajustar el factor de proporcionalidad para que la textura se vea como en la figura.

4.5.3. Clase Material

La clase **Material** encapsula una textura (opcionalmente) y además los parámetros del modelo de iluminación local que no está asociados a las fuentes de luz (estos son: los coeficientes de reflexión ambiental, difusa y especular, y el exponente de brillo, en total 4 valores de tipo **float**). Si el material tiene asociada una textura, incluye un puntero a la misma, en caso contrario ese puntero es nulo.

Se debe implementar el método **activar** de un material. En este método se debe activar la textura (si tiene), y después usar el método **fijarParamsMIL** del cauce.

La activación de un material debe de producir un error si se especifica un valor nulo o muy bajo del exponente de la componente pseudo-especular. Hay que tener en cuenta que no tiene sentido usar esos valores, y puede producir resultados no esperados. En general, se debe usar un valor no inferior a la unidad. Lo usual es usar valores muchos más altos.

4.5.4. Añadidos a la clase NodoGrafoEscena

El método **visualizarGL** de la clase **NodoGrafoEscena** debe ser completado para tener en cuenta ahora que puede haber un material activo al inicio, y que algunas entradas son de tipo material (además de punteros a objetos y transformaciones).

Para llevar a cabo esto se debe tener en cuenta que el puntero global a la aplicación **aplicacionIG** permite acceder a un valor lógico (**aplicacionIG->iluminacion**) que valdrá **true** si la iluminación está activada, y **false** si está desactivada. También se debe usar el puntero (**aplicacionIG->pila_mat**) a la pila de materiales.

Por tanto, el estudiante debe de escribir código para que cuando la iluminación esté activada, se haga esto:

1. Al inicio del método: guardar una copia del material activo (hacer *push* de la pila de materiales).
2. Durante el bucle que recorre las entradas: si una entrada es de tipo material, es necesario activarlo.
3. Al final del método: debemos de restaurar el material activo al inicio (hacer *pop* de la pila de materiales).

(ver las transparencias de teoría).

4.5.5. Añadidos a la clase MallaInd y derivadas

Será necesario extender los constructores de las clases derivadas de **MallaInd** para calcular las tablas de coordenadas de textura de los vértices (tabla **cc_tt_ver**, con entradas de tipo **glm::vec2**). Esta tabla contiene un par (s, t) con las coordenadas de textura de cada vértice.

También se calcularán las tablas de normales: la tabla de normales de triángulos (tabla **nor_tri**), con una entrada por triángulo, y la tabla de normales de vértices (tabla **nor_ver**), con una entrada por vértice. Ambas tablas tienen entradas de tipo **glm::vec3**. La primera de ellas es una tabla auxiliar para la construcción de la segunda, para algunos tipos de objetos (para otros no es necesario calcular las normales de triángulos). La tabla de vértices debe tener normales de longitud unidad.

En particular será necesario completar el código en estos puntos:

- Métodos `calcularNormalesTriangulos` y `calcularNormales` de `MallaInd`. Se deben calcular las normales de los vértices promediando las normales de las caras.
- Constructores de las clases `Cubo` y `Tetraedro` definidas en la práctica 1. Se deben calcular las normales al final de los mismos, invocando `calcularNormales`.
- Constructor de `MallaPLY`. Se debe invocar a `calcularNormales` al final.
- Método `inicializar` de la clase `MallaRevol`. En la práctica 2 se añadió el código que calcula la tabla de vértices (`vertices`) y la de triángulos (`triangulos`), y ahora se debe extender ese mismo código para calcular también las tablas de normales (`nor_ver`) y de coordenadas de textura (`cc_tt_ver`).

Los algoritmos necesarios para estos cálculos dependen del tipo de objeto y se detallan en la sección 4.6.

4.5.6. Añadidos a la clase Escena

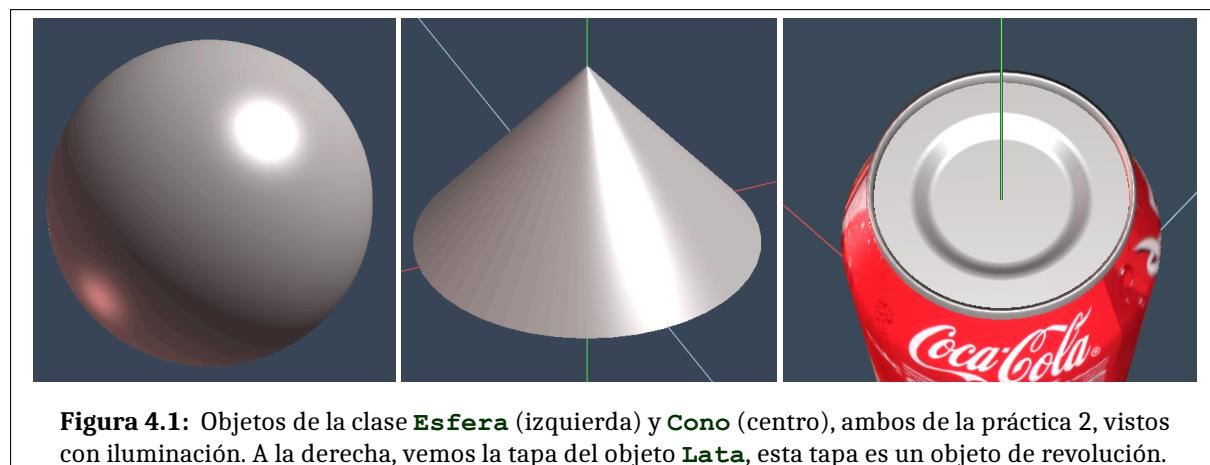


Figura 4.1: Objetos de la clase `Esfera` (izquierda) y `Cono` (centro), ambos de la práctica 2, vistos con iluminación. A la derecha, vemos la tapa del objeto `Lata`, esta tapa es un objeto de revolución.

Se definirá una nueva clase (`Escena4`) derivada de `Escena`, específica para la práctica 4, que contiene un objeto con el grafo jerárquico descrito en el punto anterior. La clase se declara en `escena.h` y se implementa en `escena.cpp`. Además, se debe de añadir una instancia de `Escena4` al vector de escenas, en el constructor de la clase `AplicacionIG` (archivo `aplicacion-ig.cpp`), al igual que hemos hecho para `Escena2` y `Escena3`.

La implementación de la clase `Escena` (en el archivo `escena.cpp`) debe extenderse, de forma que ahora, cuando se visualiza una escena, si la iluminación está activada, debemos de activar al inicio una colección de fuentes de luz y un material inicial (en este orden), antes de visualizar el objeto actual de la escena.

En el constructor de la clase `Escena` debemos de incluir código que inicialice las variables de instancia `col_fuentes` (un puntero a `ColFuentesLuz`) y `material_ini` (un puntero a `Material`). Para inicializar las fuentes de luz podemos usar una instancia de la clase `Col2Fuentes`, derivada de `ColFuentesLuz`, y ya implementada en el archivo `materiales-luces.cpp`. El constructor de `Col2Fuentes` añade dos fuentes de luz a la colección, con orientaciones y colores distintos (los colores suman (1, 1, 1) para que no se la imagen no aparezca como *sobreexpuesta*). Las variables de instancia citadas quedan entonces disponibles en todas las clases derivadas de `Escena` (a saber: `Escena1`, `Escena2`, etc...).

El método `visualizaGL` de la clase `Escena` se encarga de visualizar el objeto actual de la escena. Por tanto, este método debe extenderse. Ahora, si está activada la iluminación (es decir, si la variable

`iluminacion` dentro de `cv` está a `true`), se debe de habilitar la iluminación en el cauce (método `fijarEvalMIL` de `cv.cauce`), después activar la colección de fuentes de la escena, y finalmente activar el material inicial.

Al activar las luces y un material antes de visualizar cada escena, veremos los objetos de las prácticas 1,2 y 3 con la iluminación activada. Cuando se definan bien las normales de los objetos de revolución, podremos ver los objetos de la práctica 2 (como la **Esfera** y el **Cono**) con iluminación (ver figura 4.1).

4.5.7. Visualización de normales.

Pulsando la letra **N** se activa o desactiva la visualización de normales (se cambia la variable de instancia lógica `visualizar_normales` de la clase **AplicacionIG**). Cuando está activada dicha visualización de normales, después de dibujar la escena actual, se visualizan las normales de la misma. Para ello se invoca el método `visualizarNormalesGL` de la clase **Escena**, el cual configura el cauce y después invoca el método `visualizarNormalesGL` del **Objeto3D** actual de la escena. Puesto que ese método es un método virtual, el estudiante debe escribir las implementaciones del mismo para las clases **MallaInd** y **NodoGrafoEscena**.

En los objetos de la escena, cada vector normal \mathbf{n}_i de cada vértice en la posición \mathbf{p}_i se visualizará como un segmento de recta paralelo a la normal, con un extremo en la posición \mathbf{p}_i , y el otro en la posición $\mathbf{p}_i + a\mathbf{n}_i$, donde a es una constante ajustable y que puede depender de la escala de los objetos que se visualizan (prueba inicialmente con $a = 1$ y luego ajústalo).

La visualización de normales se hace sin iluminación, es decir, usando un mismo color plano para todos los segmentos. A continuación se dan más detalles para la implementación de este método en las clases **MallaInd**, **NodoGrafoEscena** y **Escena**.

4.5.7.1. El método `visualizarNormalesGL` en la clase **MallaInd**

Para las mallas indexadas, las normales requieren la creación de un descriptor de VAO no indexado con dos tuplas de coordenadas (dos tuplas `vec3`) por cada vértice de la malla, así que ese descriptor únicamente tendrá una tabla de posiciones (no tiene otros atributos), con el doble de entradas que vértices hay en la malla. Para crear el descriptor debemos de crear antes un vector o tabla de segmentos (un `std::vector<glm::vec3>`) e insertar en ese vector las posiciones de los extremos de los segmentos. La declaración de la clase **MallaInd** ya incluye dos variables de instancia para esto, se llaman `dvaonormales` para el VAO (initialmente es un puntero nulo) y `segmentos_normales` para la tabla (initialmente vacío).

En el método `visualizarNormalesGL` de **MallaInd**, la creación de la tabla y el VAO se hará bajo demanda, es decir, la primera vez que se invoque dicho método para cada objeto. Puesto que la creación del VAO supone hacer una copia de los datos, la tabla `segmentos_normales` se puede vaciar después de haber creado el VAO.

Así que el estudiante debe de completar el código que da estos pasos:

1. Si el puntero al descriptor de VAO de normales (`dvaonormales`) es nulo, debemos de crear dicho descriptor, con estos pasos:
 - 1.1. Para cada posición \mathbf{v}_i de un vértice en el vector `vertices`:
 - 1.1.1. Leer la correspondiente normal \mathbf{n}_i del vector de normales (`nor_ver`).
 - 1.1.2. Añadir \mathbf{v}_i al vector `segmentos_normales`.

- 1.1.3. Añadir $\mathbf{v}_i + a\mathbf{n}_i$ al vector **segmentos_normales**.
- 1.2. Crear el objeto descriptor del VAO de normales, para ello se usa el vector **segmentos_normales** y se tiene en cuenta que esa descriptor únicamente gestiona una tabla de atributos de vértices (la de posiciones, ya que las otras no se necesitan).
2. Visualizar el VAO de normales, usando el método **draw** del descriptor, con el tipo de primitiva **GL_LINES**.

4.5.7.2. El método **visualizarNormalesGL** en la clase **NodoGrafoEscena**

Este método hace un recorrido de las entradas del nodo, parecido a 'visualizarGL', pero ignorando los materiales y colores (y llamando recursivamente a **visualizarNormalesGL** para los nodos hijos). Por tanto, el alumno escribirá el código teniendo en cuenta estos puntos:

- Hacer *push* de la matriz de modelado al inicio y *pop* al fin (al igual que en **visualizarGL**)
- Escribir el bucle que recorre las entradas del nodo, llamando recursivamente a **visualizarNormalesGL** en los nodos u objetos hijos
- En ese bucle se deben ignorar el color o identificador del nodo (se supone que el color ya está prefijado antes de la llamada).
- También se debe ignorar las entradas de tipo material, y la gestión de materiales (se usa sin iluminación)

4.5.7.3. El método **visualizarNormales** en la clase **Escena**

En este método se deben visualizar las normales del objeto actual de la escena, pero configurando antes el cauce para que dichas normales aparezcan todas de un mismo color plano. Por tanto el estudiante debe escribir código para dar estos pasos:

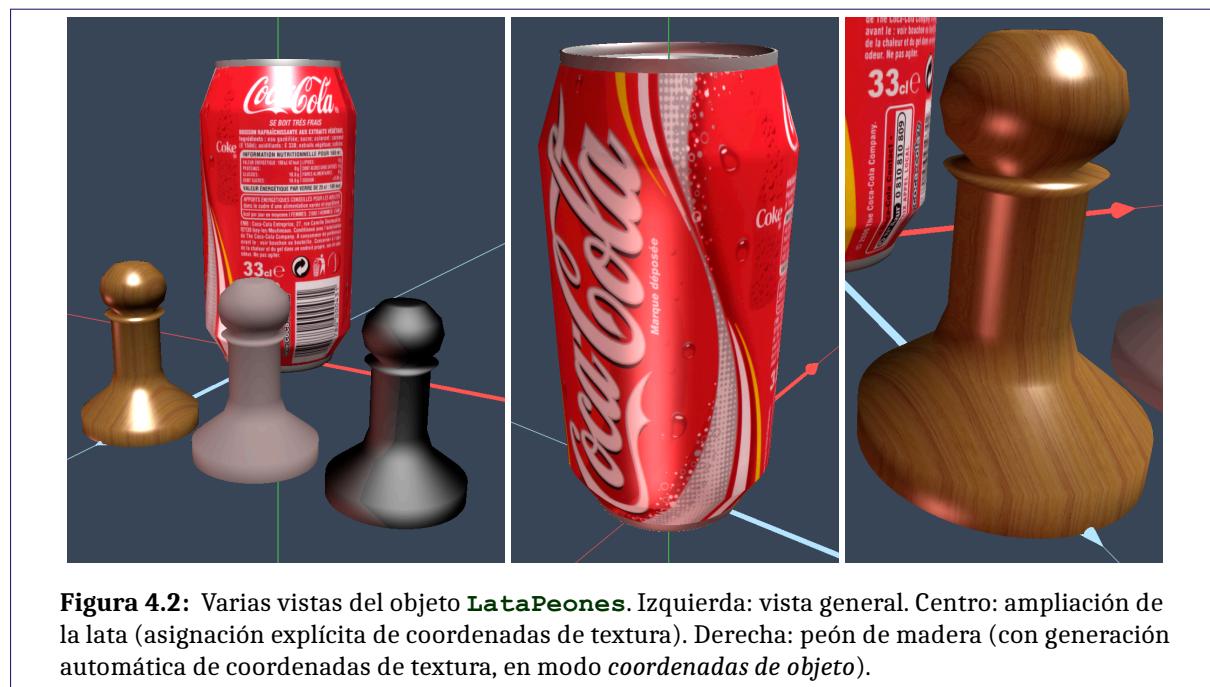
- Configurar el cauce de la forma adecuada, es decir:
 - Desactivar la iluminación (con **fijarEvalMIL**)
 - Desactivar el uso de texturas (con **fijarEvalText**)
 - Fijar el color (con **fijarColor**)
- Visualizar las normales del objeto actual de la escena (con el método **visualizarNormalesGL** de dicho objeto).

4.5.8. Clase **Escena4**

Para visualizar los objetos específicos de esta creará la clase **Escena4**, derivada de **Escena**. Al igual que las clases **Escena1**, **Escena2**, etc.... la nueva clase **Escena4** simplemente añade un nuevo constructor. Se declara en el archivo **escena.h** y el constructor se implementa en **escena.cpp**.

En el constructor de **AplicacionIG** (archivo **aplicacion-ig.cpp**) se debe de añadir una instancia de **Escena4** en el vector de escenas (escenas).

Esta escena contendrá un objeto de tipo **LataPeones** (ver la sección 4.5.9 y la figura 4.2, izquierda), y otro objeto de tipo **NodoCubo24** (ver la sección 4.6.1 y la figura 4.4). De esta forma, una vez que estemos viendo la escena 4, podemos comutar entre ambos objetos con la tecla O.



4.5.9. Clase **LataPeones**

Cada instancia de **Escena4** incluye un objeto de una nueva clase llamada **LataPeones** (derivada de **NodoGrafoEscena**), que se debe declarar e implementar en los archivos **latapeones.h** y **latapeones.cpp**, respectivamente, dentro de las carpetas **include** y **srs** del directorio de trabajo. En esta sección se detalla la estructura de esta clase.

Cada instancia de **LataPeones** incluye varios sub-objetos, en concreto se añadirán tres instancias del objeto peón de la práctica 2, con distintos materiales (uno pseudo-especular o metálico, otro difuso o mate y un tercero que es una combinación de ambos). También se añadirá un objeto nuevo (una lata de bebida), compuesto de tres sub-objetos (de revolución) con dos materiales distintos (un material metálico para la tapa y la base, y un material con textura para el cuerpo o parte central). La textura de la lata usa la tabla de coordenadas de textura, mientras que la textura del peón usa generación automática de coordenadas de textura. La escena se puede observar en la figura 4.2 (izquierda).

4.5.9.1. Objeto lata

Este objeto está compuesto de tres sub-objetos. Cada uno de ellos es una malla distinta, obtenida por revolución de un perfil almacenado en un archivo ply (en la carpeta de **materiales/plys**). En concreto:

- Archivo **lata-pcue.ply**: perfil de la parte central, la que incorpora la textura de la lata (archivo **lata-coke.jpg**). Es un material difuso-especular.
- Archivo **lata-psup.ply**: tapa superior metálica. No lleva textura, es un material difuso-especular de aspecto metálico. (derecha). Ver la figura 4.1 (derecha).
- Archivo **lata-pinf.ply**: base inferior metálica, sin textura y del mismo tipo de material que la tapa .

El aspecto de este objeto se puede observar en la figura 4.2, centro.

4.5.9.2. Objetos peón.

Son tres objetos obtenidos por revolución, usando el mismo perfil de la práctica 2. Los tres objetos comparten la misma malla, solo que instanciada tres veces con distinta transformación y material en cada caso:

- Peón **de madera**: con la textura de madera difuso-especular, usando generación automática de coordenadas de textura, de forma que la coordenada *s* de textura es proporcional a la coordenada X de la posición, y la coordenada *t* a Y (ver figura 4.2, derecha). La textura está en el archivo **text-madera.jpg** (ver figura 4.3, derecha).
- Peón **blanco**: sin textura, con un material puramente difuso (sin brillos especulares), de color blanco (ver figura 4.2, izquierda).
- Peón **negro**: sin textura, con un material especular sin apenas reflectividad difusa (ver figura 4.2, izquierda).

4.5.10. Materiales y textura en el grafo de la práctica 3

En esta práctica se incorporarán texturas y fuentes de luz al objeto jerárquico creado para la práctica 3. A dicho objeto se le pueden aplicar distintas texturas y/o materiales a las distintas partes de forma que se incremente su grado de realismo.

Para añadir materiales y texturas, será necesario extender los constructores de las clases derivadas de **NodoGrafoEscena**, de forma que invoquemos el método **agregar** para añadir entradas de tipo material. Esto requiere la construcción de materiales y texturas. Puedes usar tus propias imágenes de textura, pero recuerda que debes situarlas en la carpeta **archivos-alumno**, no en la carpeta **materiales**.

Los objetos de la escena deben incluir al menos dos objetos con texturas: uno de ellos con su tabla de coordenadas de textura y otro con generación automática de coordenadas de textura (en modo coordenadas de objeto). También debe haber objetos con colores planos (sin textura).

Hay que tener en cuenta que si en la práctica 3 para el grafo de escena se ha usado la clase **Cubo**, ahora debemos de usar en su lugar la clase **Cubo24**, que está pensada para iluminación y normales.

Se deben cubrir materiales de diversos tipos, tanto materiales eminentemente difusos, como materiales pseudo-especulares. Cada material usado se asociará con un identificador único (un identificador alfanumérico), esos identificadores alfanuméricos aparecerán en el grafo de escena, como se describe aquí abajo.

Se debe añadir información al archivo PDF con documentación sobre el grafo de escena que se elaboró para la práctica 3. La nueva información es relativa a los materiales y texturas en dicho grafo. Se debe de añadir (sin quitar nada de lo que había) un apartado o sección nueva al final del PDF, ese apartado o sección se debe titular **Práctica 4 - Materiales y texturas**. Dicho apartado contendrá la siguiente información:

- Una (o varias) captura de pantalla del modelo, con la iluminación activada, donde se aprecien lo mejor posible todas las partes del mismo, así como las diferentes texturas y materiales.
- Una nueva versión del grafo de escena, similar al de la práctica 3, pero que incluya las entradas de tipo material. Para cada entrada de ese tipo, se incluye el rótulo *Material <identificador>*, donde *<identificador>* es el identificador único del material usado en esa entrada.
- Una lista con información de todos y cada uno de los materiales usados en el grafo de escena, para cada uno de ellos:
 - Nombre o identificador único del material.
 - Nombre o nombres de los nodos que tienen entradas con ese material.
 - Valores de los coeficientes k_a , k_d , k_s y el exponente e que caracterizan al material.
 - Si ese material tiene asociada una textura o no la tiene. En caso de que la tenga, se debe indicar:
 - Nombre del archivo de textura, y una imagen de ese archivo.
 - Si la textura tiene asociada generación automática de coordenadas de textura (GACT) o no la tiene. Si la tiene, se deben dar los dos vectores de coeficientes asociados.

4.6. Cálculo de tablas de normales y coordenadas de textura

En esta sección se detalla la metodología a seguir para crear las tablas de normales (vector `nor_ver`) y coordenadas de textura (vector `cc_tt_ver`) de los objetos de clases derivadas de `MallaInd`. Para ello debemos de extender el código de los constructores de esas clases, de forma que ahora incluyan la creación de las mencionadas tablas.

4.6.1. Clases `Cubo24` y `NodoCubo24`

Queremos visualizar un objeto de tipo cubo, posiblemente asignando una imagen de textura a cada una de sus 6 caras, y con la iluminación correcta usando las normales a dichas caras.

La clase `Cubo`, creada en la práctica 1, no es apropiada para la iluminación y texturas, esto se debe a que las tablas de normales y coordenadas de textura presentan discontinuidades en las aristas de un cubo real. En el caso de las normales, en dichas aristas confluyen dos caras con distintas orientaciones, y en el caso de las coordenadas de textura, cada arista es adyacente a dos instancias de una imagen de textura (una en cada cara), de forma que no podemos asignar unas coords. de textura únicas a un punto en una arista.

Para poder usar cubos con iluminación y texturas en nuestros modelos será necesario definir una nueva clase, que llamaremos `Cubo24`, también derivada de `MallaInd`, de forma que su constructor construya un cubo de geometría similar al original, pero que ahora tiene 24 vértices en vez de 8. Además, se asignan valores a las tablas de coordenadas de textura, explícitamente, de forma que

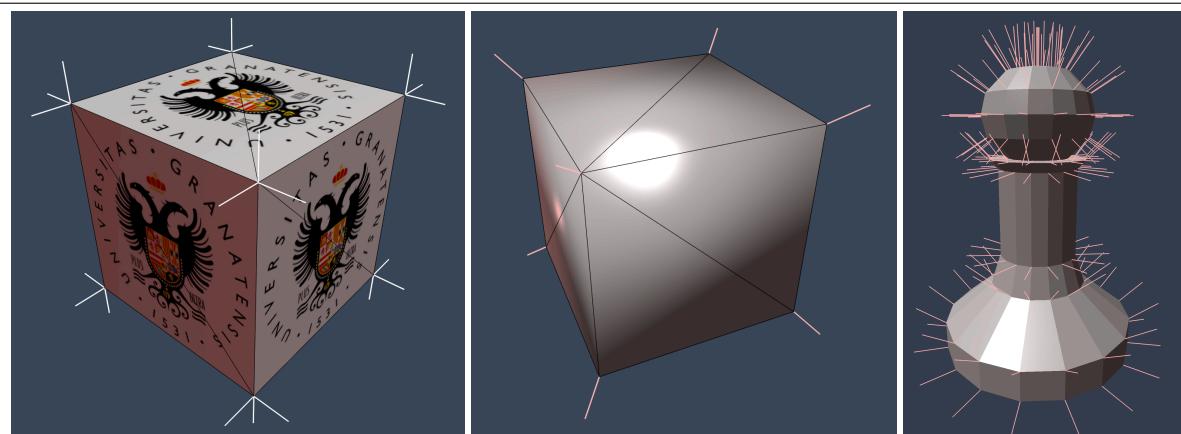


Figura 4.4: Varios objetos con sus normales de vértices también visualizadas. Izquierda: objeto **Cubo24** (cubo con 24 vértices, ver sección 4.6.1), con las normales y las texturas correctamente asignadas. Centro: objeto **Cubo** (8 vértices) con las normales promediadas (sección 4.6.3), se observa la iluminación errónea. Derecha: objeto de revolución de la práctica 2 (clase **Peon**, ver sección 4.6.2), con normales de triángulo (no interpoladas).

si se visualiza con una textura veamos la misma imagen replicada en cada una de las caras del cubo.

Para visualizar el cubo se insertará en un grafo de escena específico con un único nodo, de tipo **NodoCubo24** (una clase derivada de **NodoGrafoEscena**). Este nodo incluye una entrada material con textura y luego el cubo de 24 vértices. La imagen de textura será el archivo **window-icon.jpg** (disponible en la carpeta **materiales/imgs**). Debe quedar con el escudo de la Universidad de Granada en las 6 caras, tal y como se observa en la figura 4.4 (izquierda). El nodo con el cubo y su textura se insertará como un objeto en la escena de la práctica 4 (en el constructor de **Escena4**, ver sección 4.5.8).

4.6.2. Clase **MallaRevol**

En el caso de los objetos de revolución obtenidos a partir de un perfil, podemos asignarles fácilmente coordenadas de textura y normales a partir de los vértices de dicho perfil. Para ello se debe de extender el código que genera la malla de revolución, en el método **inicializar** de la clase **MallaRevol**, según se detalla a continuación.

4.6.2.1. Cálculo de normales

Supongamos que el perfil original consta de n vértices cuyas coordenadas son $\mathbf{v}_0, \mathbf{v}_1 \dots, \mathbf{v}_{n-1}$ (cada una de ellas tiene la componente Z a cero). Para calcular las normales de los vértices del perfil hacemos:

1. En primer lugar calculamos las normales (normalizadas) de las $n - 1$ aristas $\mathbf{m}_0, \dots, \mathbf{m}_{n-2}$. El vector \mathbf{m}_i tiene longitud unidad y está rotado (en el sentido de las agujas del reloj) 90° respecto de la i -ésima arista, paralela al vector $\mathbf{v}_{i+1} - \mathbf{v}_i$.
2. Calculamos las normales de los vértices ($\mathbf{n}_0, \dots, \mathbf{n}_{n-1}$). Para ello hacemos $\mathbf{n}_0 = \mathbf{m}_0$, también hacemos $\mathbf{n}_{n-1} = \mathbf{m}_{n-2}$. Para el resto de vértices, calculamos \mathbf{n}_i como $\mathbf{m}_{i-1} + \mathbf{m}_i$ (normalizado).

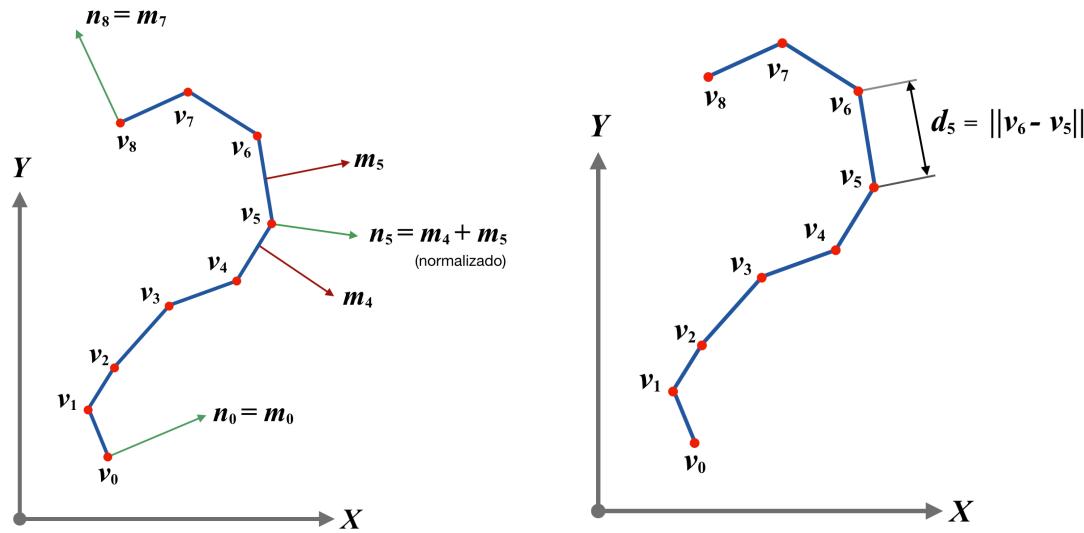


Figura 4.5: Izquierda: ejemplo de un perfil con las normales de sus aristas y sus vértices. Derecha: el mismo perfil, donde hemos señalado la distancia entre los vértices 5 y 6.

Estas normales de los vértices del perfil se calculan y guardan en un vector al inicio del método **inicializar**, y se usarán para crear las normales de los vértices de la malla completa. Para ello tenemos en cuenta que la normal de cualquier vértice de la malla completa es igual a la normal (rotada) del correspondiente vértice del perfil original. Por tanto, cada vez que se añade un vértice a la tabla **vertices** podemos también añadir su normal a la tabla **nor_ver** (ver figura 4.5, izquierda).

4.6.2.2. Cálculo de coordenadas de textura

Para el cálculo de las coordenadas de textura de todos los vértices, en primer lugar calculamos el vector de valores reales d_0, \dots, d_{n-2} , donde $d_i = \|v_{i+1} - v_i\|$ es la distancia entre el vértice i y el $i + 1$ en el perfil original. Después calculamos el vector con los valores t_0, \dots, t_{n-1} , donde $t_0 = 0$ y

$$t_i = \sum_{j=0}^{i-1} d_j \Bigg/ \sum_{j=0}^{n-2} d_j$$

es la distancia, medida a lo largo del perfil, entre el vértice 0 y el vértice i (normalizada de forma que $t_{n-1} = 1$) (ver figura 4.5, derecha).

Los valores t_0, \dots, t_{n-1} se calculan y se guardan en un vector al inicio del método **inicializar**. Después, en un bucle doble añadimos cada vez el i -ésimo vértice de la j -ésima copia de perfil, donde j va desde 0 hasta $n - 1$, ambos incluidos (n es el número de copias del perfil, ver guion de la práctica 2). Al añadir el vértice a la tabla **vertices**, también podemos añadir sus coordenadas de textura (una tupla de 2 flotantes) a **cc_tt_ver**. Para ello usamos como coordenada S el valor $j/(n - 1)$ (división real), y como coordenada T el valor $1 - t_i$. El motivo de usar $1 - t_i$ en lugar de t_i es que de esa manera se invierte el orden en vertical de la textura, lo cual es necesario por la discrepancia en el orden vertical que se mencionó en la sección 4.5.2.

4.6.3. Clases Cubo, Tetraedro y MallaPLY

Los objetos PLY son mallas leídas de un archivo de las cuales, en general, desconocemos cual es exactamente la superficie que aproximan. Por tanto, debemos de calcular sus normales de vértices haciendo la suposición de que aproximan una superficie de normal continua. Esto lo llevamos a cabo asignándole a cada vértice la normal promedio (suma normalizada) de las caras adyacentes a dicho vértice. Lo mismo hacemos para objetos de otras clases, como **Cubo** o **Tetraedro**, en las cuales no vamos a modificar la topología (como ocurre en la clase **Cubo24**) y por tanto usaremos las normales de vértice promediadas.

Para crear las tablas de normales en estas clases, será necesario añadir una llamada al método **calcularNormales** al final del constructor.

Respecto a las coordenadas de textura, no se calcularán para los objetos PLY, ya que aunque podríamos estudiar algoritmos para hacerlo, son complejos para estas prácticas. Si se quiere asignar texturas a alguno de estos objetos, se podrá usar generación automática de coordenadas de textura. Respecto a las clases **Cubo** y **Tetraedro**, tampoco las calculamos, pues en este tipo de objetos no tiene sentido asignarle explícitamente coordenadas a los vértices.

Para implementar el cálculo de normales debemos, en primer lugar, calcular las normales a las caras, y después las normales de los vértices. A continuación se detalla como extender el código para hacerlo.

4.6.3.1. Normales de las caras: método **calcularNormalesTriangulos** de **MallaInd**

La tabla de normales de las caras o triángulos (en coordenadas de objeto o maestras) es una variable de instancia protegida de **MallaInd**, se llama **nor_tri**, es de tipo **std::vector<glm::vec3>**, con tantas entradas como caras, e inicialmente vacía en todos los objetos. Esta tabla se calcula en el método **calcularNormalesTriangulos** de **MallaInd**.

En este método se deben de recorrer la tabla caras que hay en la malla. En cada cara se consideran las posiciones (coordenadas de objeto) de sus tres vértices, sean estas, por ejemplo **p,q** y **r**. A partir de estas coordenadas se calculan los vectores **a** y **b** correspondientes a dos aristas, haciendo **a = q - p** y **b = r - p**. El vector **m_c**, perpendicular a la cara, se obtiene como el producto vectorial de las aristas, es decir, hacemos: **m_c = a × b**. Finalmente, el vector normal **n_c** (de longitud unidad) se obtiene normalizando **m_c**, esto es: **n_c = m_c/||m_c||**.

Hay que tener en cuenta que, para objetos cerrados, es necesario que todas las normales apunten hacia el exterior del objeto, y que en los objetos abiertos, todas ellas apunten hacia el mismo lado de la superficie. Para ello la selección de los tres vértices **p,q** y **r** de la cara debe hacerse de forma coherente, es decir, siempre en orden de las agujas del reloj, o siempre en el contrario (visto desde un lado de la superficie).

Aunque se haga el cálculo de normales de forma coherente, según lo indicado, las normales pueden quedar todas ellas apuntando al lado equivocado, si este fuese el caso, se puede cambiar el orden del producto vectorial, es decir, hacer **m_c = b × a** en lugar del originalmente descrito, ya que el producto vectorial es anti-comutativo.

Un problema que puede haber es que algunos triángulos están *degenerados* (son triángulos en los cuales dos o más vértices están en la misma posición), y al calcular el producto vectorial de las aristas se produzca un vector de longitud nula, lo cual provoca un error al intentar normalizarlo. Para

prevenir esto, cuando la longitud de ese vector sea cero, no se intenta normalizar, y le asignamos al triángulo el vector nulo como vector normal.

4.6.3.2. Normales de los vértices: método `calcularNormales` de `MallaInd`

El cálculo de las normales de vértices se debe implementar en el método `calcularNormales` de `MallaInd`. Al inicio se debe invocar el método `calcularNormalesTriangulos` para calcular las normales de las caras.

Una vez calculadas las normales de caras, se obtienen las normales de los vértices. Para cada vértice, el vector \mathbf{m}_v , es un vector aproximadamente perpendicular a la superficie de la malla en la posición del vértice. Se puede definir como la suma de los vectores normales de todas las caras adyacentes a dicho vértice, es decir:

$$\mathbf{m}_v = \sum_{i=0}^{k-1} \mathbf{u}_i$$

donde \mathbf{u}_i es el vector perpendicular a la i -ésima cara adyacente al vértice (suponemos que hay k de ellas). Al igual que con las caras, el vector normal al vértice \mathbf{n}_v se define como una versión normalizada de \mathbf{m}_v , es decir: $\mathbf{n}_v = \mathbf{m}_v / \|\mathbf{m}_v\|$.

Una implementación básica (derivada directamente de esta definición de \mathbf{n}_v) requeriría recorrer la lista de vértices (en un bucle externo), y en cada uno de ellos buscar sus caras adyacentes, recorriendo para ello la lista de caras completa (en un bucle interno). Esta implementación, por tanto, tiene complejidad en tiempo en el orden del producto del número de caras y de vértices (o cuadrática con el número de vértices, que es proporcional al de caras para la inmensa mayoría de las mallas).

Se recomienda diseñar e implementar un método más eficiente con complejidad en tiempo en el orden del número de caras. Este método está basado en recorrer las caras en el bucle externo, en lugar de los vértices, y en eliminar el bucle interno. Esto es posible debido a que obtener los vértices de una cara se puede hacer de forma inmediata (en $O(1)$) sin más que consultar la entrada correspondiente de la tabla de caras.

4.7. Ejercicios adicionales

4.7.1. Ejercicio 1

(convocatoria ordinaria curso 20-21)

La figura 4.6 incluye un trozo de código con la implementación de las clases `MallaDiscoP4` (derivada de `MallaInd`) y `NodoDiscoP4` (derivada de `NodoGrafoEscena`). Escribe las declaraciones de ambas clases y copia esta implementación en tu código de prácticas. El objeto `NodoDiscoP4` aparecerá como se observa en la figura 4.7 (izquierda).

Usa la imagen de textura que aparece en la figura 4.7 (derecha) para crear un material con textura que se debe añadir como primera entrada de la clase `NodoDiscoP4`. A continuación, completa el código (sin eliminar nada del código que ya hay) de la clase `MallaDiscoP4` para crear la tabla de coordenadas de textura en el constructor, de forma que el disco aparezca con textura como se observa en la figura 4.8 (izquierda).

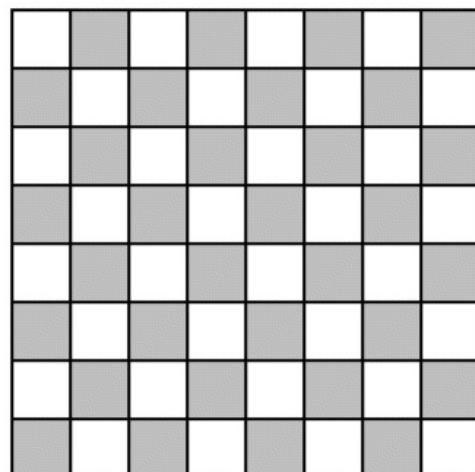
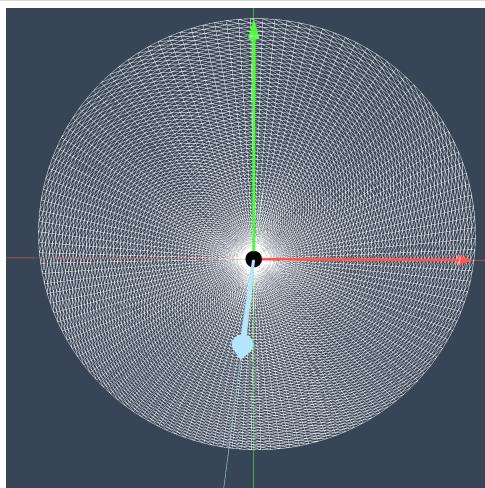
```

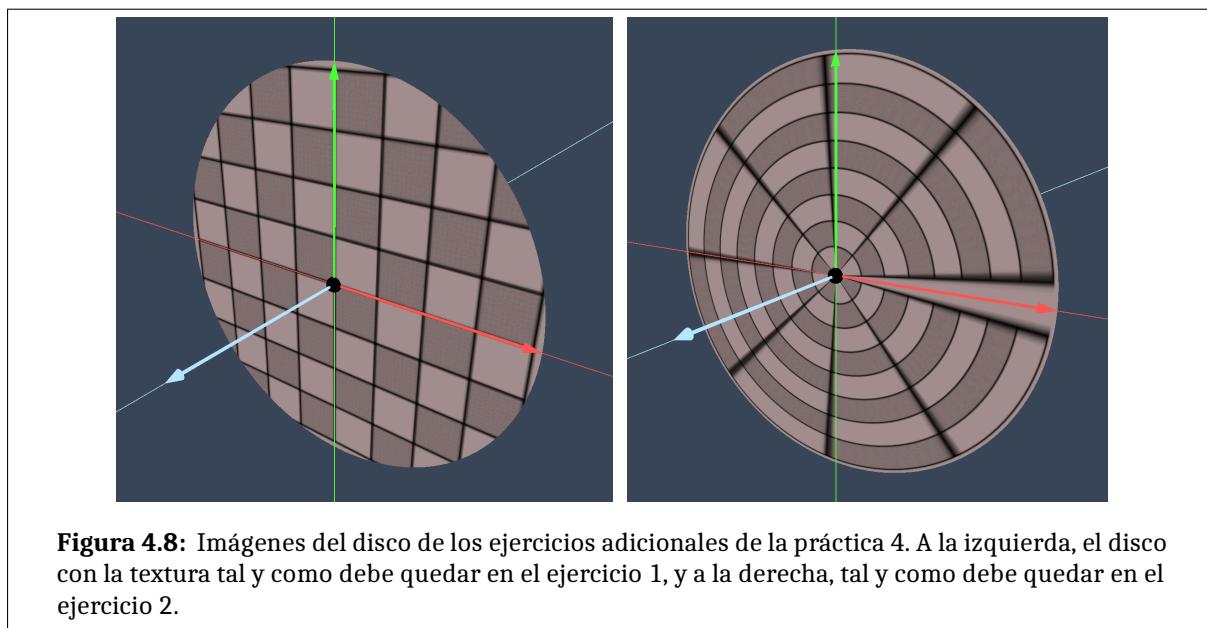
MallaDiscoP4::MallaDiscoP4()
{
    ponerColor({1.0, 1.0, 1.0});
    const unsigned ni = 23, nj = 31;

    for( unsigned i= 0 ; i < ni ; i++ )
    for( unsigned j= 0 ; j < nj ; j++ )
    {
        const float
            fi = float(i)/float(ni-1),
            fj = float(j)/float(nj-1),
            ai = 2.0*M_PI*fi,
            x = fj * cos( ai ),
            y = fj * sin( ai ),
            z = 0.0 ;
        vertices.push_back({ x, y, z });
    }
    for( unsigned i= 0 ; i < ni-1 ; i++ )
    for( unsigned j= 0 ; j < nj-1 ; j++ )
    {
        triangulos.push_back({ i*nj+j, i*nj+(j+1), (i+1)*nj+(j+1) });
        triangulos.push_back({ i*nj+j, (i+1)*nj+(j+1), (i+1)*nj+j });
    }
}

NodoDiscoP4::NodoDiscoP4()
{
    ponerNombre("Nodo ejercicio adicional práctica 4, examen 27 enero");
    agregar( new MallaDiscoP4() );
}

```

Figura 4.6: Código con la implementación de las clases **MallaDiscoP4** y **NodoDiscoP4****Figura 4.7:** Imágenes relativas a los ejercicios adicionales de la práctica 4. A la izquierda el objeto de la clase **NodoDiscoP4**, a la derecha la imagen de textura **ea-textura-cuadricula.jpeg**.



4.7.2. Ejercicio 2

(convocatoria ordinaria curso 20-21)

Repite el ejercicio anterior, pero esta vez el código que genera la tabla de coordenadas debe calcular dichas coordenadas de forma que el disco se vea como aparece en la figura 4.8 (derecha).

4.7.3. Ejercicio 3

(añadido en Diciembre de 2024).

Crea una clase derivada de **NodoGrafoEscena** para un grafo con un único nodo, ese nodo contiene una textura y una instancia de una malla almacenada en un PLY (en concreto, el archivo PLY del busto de Beethoven, **beethoven.ply**).

La textura es una instancia de una clase derivada de **Textura**, en el constructor de esa clase debe configurar la textura para que haga uso de la generación automática de coordenadas de textura (en modo *coordenadas de objeto*), usando la imagen de la madera (igual que en el peón de madera). El constructor de la textura debe tener un parámetro booleano, que permita seleccionar si las vetas de la madera aparecen en vertical o en horizontal. Ese mismo parámetro lo tiene el constructor de la clase derivada de **NodoGrafoEscena**.

Añade a la colección de objetos de la práctica 4 dos instancias de este Beethoven de madera, una de ellas con las vetas de la madera en horizontal y otra con las vetas en vertical.

En la figura 4.9 se observan imágenes con las dos instancias del objeto descrito.

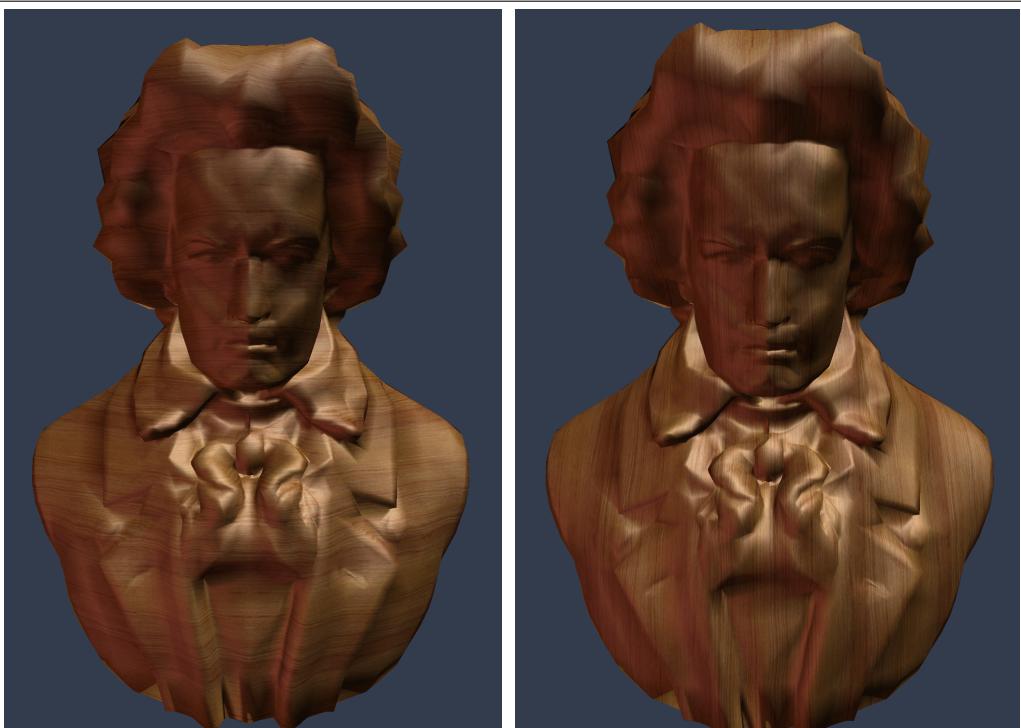


Figura 4.9: Imágenes de la malla indexada con textura de madera que se detalla en el ejercicio adicional 3 de la práctica 4, a la izquierda las vetas aparecen en horizontal y a la derecha en vertical.

5. Interacción: cámaras y selección..

5.1. Objetivos

El objetivo de esta práctica es:

- Aprender a desarrollar aplicaciones gráficas interactivas, gestionando los eventos de entrada de ratón.
- Aprender a realizar operaciones de selección de objetos en la escena.
- Afianzar los conocimientos de los parámetros de la cámara para su correcta ubicación y orientación en la escena.

5.2. Desarrollo

Partiendo de las prácticas anteriores, se añadirá funcionalidad relativa a la gestión de cámaras y a la selección de objetos, usando las técnicas descritas en el tema 4 de teoría.

5.2.1. Gestión de cámaras

Se añadirán varios objetos de tipo cámara a la clase **Escena**. Las nuevas cámaras serán cámaras interactivas, modificables con el teclado y el ratón, instancias de la clase **Camara3Modos**. Cada instancia de la clase **Escena** tendrá en cada momento una cámara activa. Cada cámara tendrá en cada momento uno de los tres modos posibles (*modo primera persona con rotaciones*, *modo primera personal con traslaciones*, y *modo orbital* o *modo examinar*).

Se añadirá código para la modificación interactiva de la cámara actual de la escena actual usando el ratón y el teclado. En cualquier de los tres modos, los parámetros que definen la cámara actual de la escena actual (ángulos, origen y punto de atención) podrán cambiarse usando el teclado (flechas) y/o el ratón (usando movimientos con el botón derecho pulsado). También será posible cambiar la cámara actual de la escena y cambiar el modo actual de dicha cámara.

5.2.2. Selección de objetos.

Se harán asignaciones de identificadores en los constructores de los objetos de tipo malla indexada o de tipo nodo del grafo de escena, para los diversos objetos y escenas de las prácticas anteriores.

Una vez asignados los identificadores, se podrá realizar la operación de selección, usando un *frame-buffer object* (FBO) creado a tal efecto. Para ello se usará el click con el botón izquierdo del ratón, a lo que sigue una visualización (sobre dicho FBO) en modo de selección y una búsqueda del nodo que contiene el identificador en el árbol jerárquico. Una vez que se seleccione un objeto o sub-objeto, se ejecutará su método **cuandoClick**, lo cual (por defecto) pasará la cámara actual pasará a modo examinar y el centro del objeto se proyectará en el centro de la imagen (ese comportamiento puede cambiarse).

5.2.3. Documento del grafo es escena con identificadores de selección.

Se debe producir un archivo PDF, basándonos en el realizado para la práctica 4, en el cual se añade a la documentación del grafo de escena información de los identificadores de selección asignados a los nodos e información adicional de dichos identificadores (ver sección 5.6).

5.3. Tareas.

A modo de referencia, se incluyen aquí las diversas tareas concretas que el alumno debe de realizar. Respecto a definir una nueva escena, las tareas son:

1. Declarar e implementar la clase **Escena5** (derivada de **Escena**), añadirle una instancia de **VariasLatasPeones** en su constructor. Añadir esa escena al vector de escenas de la aplicación (en el constructor de **AplicacionIG**, en el archivo **aplicacion-ig.cpp**), al igual que hemos hecho con las prácticas anteriores.
2. Implementar la clase **VariasLatasPeones**, según se describe en la sección 5.5.
3. Definir una versión específica del método **cuandoClick** para los peones que hay en el grafo de la clase **VariasLatasPeones**, de forma que al hacer click en un peón, en lugar de apuntar la cámara al mismo, se desplace (ver sección 5.5).
4. Definir identificadores de objetos para los distintos objetos (mallas y grafos) de las prácticas 3 y 4 (ver sección 5.5). En particular, se deben de definir identificadores de selección para el grafo original que se diseñó en la práctica 3 y se extendió en la práctica 4 (ver sección 5.6).
5. Producir una nueva versión del archivo PDF con la documentación del grafo de escena, a partir del PDF de la práctica 4, añadiéndole la información sobre los identificadores de selección que se han añadido en esta práctica 5 al grafo original de las dos prácticas anteriores (ver sección 5.6).

Las tareas relacionadas con la gestión de cámaras son:

6. En el constructor de la clase **Escena**, añadir al vector de cámaras de la escena varias instancias de la clase **Camara3Modos** con diversas cámaras perspectivas y ortogonales (ver sección 5.7), esas cámaras sustituyen, en todas las prácticas, a la cámara tipo **CamaraOrbitalSimple** que hemos venido usando hasta ahora.
7. Completar el código de los métodos **desplRotarXY**, **moverZ** y **mirarHacia** de la clase **Camara3Modos**. Esos métodos permiten la manipulación de cámaras en los diversos modos (ver sección 5.7.3).
8. Completar el método **cuandoClick** de la clase **Objeto3D** para añadirle código que haga que la cámara apunte al centro del objeto en coordenadas del mundo (ver subsección 5.7.4).

Finalmente, las tareas relacionadas con la implementación de la selección son:

9. Implementar el método **visualizarModoSeleccionGL** de la clase **MallaInd** (ver subsección 5.8.2.1).
10. Implementar el método **visualizarModoSeleccionGL** de la clase **NodoGrafoEscena** (ver subsección 5.8.2.2).
11. Implementar el método **visualizarGL_Seleccion** de la clase **Escena** (ver subsección 5.8.3).
12. Implementar (en el archivo **seleccion.cpp**) el método **seleccion** de la clase **AplicacionIG** (ver sección 5.8.4).
13. Implementar el método **buscarObjeto** de la clase **NodoGrafoEscena** (ver sección 5.8.5).

5.4. Teclas e interacción con el ratón.

En la función `FGE_PulsarLevantarTecla` del archivo `eventos-teclado.cpp` ya se encuentra implementado el código de gestión de las siguientes teclas relacionadas con la gestión de las cámaras. En todo momento hay una escena actual, la cual a su vez tiene una cámara actual. Las teclas son:

- **flecha izquierda/derecha**: desplazamiento o rotación de la cámara actual en horizontal.
- **flecha arriba/abajo**: desplazamiento o rotación de la cámara actual en vertical.
- **tecla +**: acercamiento o desplazamiento de la cámara actual hacia adelante.
- **tecla -**: alejamiento o desplazamiento de la cámara actual hacia detrás.
- **tecla c/C**: activar el siguiente modo de cámara (o el primero) en la cámara actual.
- **tecla v/V**: activar la siguiente cámara de la escena actual (o la primera)

Además de estas teclas, en `eventos-raton.cpp` ya se ha añadido código para las diversas funciones gestoras de eventos de ratón:

- **FGE_PulsarLevantarBotonRaton**: pulsar o levantar un botón del ratón.
- **FGE_RatonMovido**: movimiento del puntero del ratón.
- **FGE_Scroll**: girar la rueda del ratón (también se llama hacer *scroll*)).

Las posibles interacciones (que se implementan usando estas funciones) modifican el estado de la cámara actual, de estas formas:

- **rueda del ratón o scroll hacia adelante**: acercamiento o desplazamiento de la cámara actual hacia adelante.
- **rueda del ratón o scroll hacia detrás**: alejamiento o desplazamiento de la cámara actual hacia detrás.
- **click con el botón izquierdo**: seleccionar el objeto que se proyecta en el pixel sobre el que se ha pulsado (si hay algún objeto en el pixel), ese objeto pasa a proyectarse en el centro de la imagen.
- **movimiento del ratón con el botón derecho pulsado**: desplazamiento o rotación de la cámara actual en horizontal y en vertical.

5.5. Escena de la práctica 5. La clase `VariasLatasPeones`.

Para esta práctica se creará una nueva clase (`Escena5`, derivada de `Escena`) que tendrá al menos un objeto de tipo `NodoGrafoEscena`. Dicho objeto será una instancia de la nueva clase `VariasLatasPeones`. Esta nueva clase será una copia de la clase `LataPeones` de la práctica 4, a la cual le añadiremos un par de latas más (ver figura 5.1). Se implementará en el archivo `latapeones.cpp`.

En este nueva clase nos aseguraremos que los distintos objetos que aparecen tienen todos ellos definidos nombres descriptivos, para ello usaremos el método `ponerNombre` en los respectivos nodos. También usaremos el método `ponerIdentificador` para asociarle distintos identificadores a dichos objetos. Los nombre de dichos objetos son:

- *Peón de madera* (usa textura de madera)
- *Peón blanco* (material difuso blanco)
- *Peón negro* (material especular, sin apenas componente difusa)
- *Lata de Coca-Cola* (textura de lata de Coca-Cola)
- *Lata de Pepsi* (textura de lata de Pepsi, archivo `lata-pepsi.jpg`).



Figura 5.1: Escena de la práctica 5 con los 6 objetos que se indican en la sección 5.5

- *Lata de la UGR* (textura con el escudo o logotipo de la Univ. de Granada, archivo `window-icon.jpg`).

Es importante asignarle estos nombres a los objetos, de forma que sea fácil después depurar el código de selección. Las texturas necesarias ya se encuentran disponibles en la carpeta `materiales/imgs`.

Se deben de implementar el método `cuandoClick` para los tres peones de esta escena, de forma que cuando se haga click en un peón, se ejecute código que cambie la matriz de transformación de dicho peón y se desplace cierta distancia en la dirección y sentido de la rama positiva del eje Z. Para eso se debe de definir una clase específica para estos peones, que puede ser derivada de `NodoGrafoEscena`, pero que tendrá esa matriz de desplazamiento y además tendrá redefinido el método `cuandoClick`. Puesto que el método `cuandoClick` de estos peones no es ya el método por defecto, ahora al pulsar en ellos no se cambia la cámara (pero sí cabia el grafo).

Asimismo, el estudiante debe de asignar identificadores y nombres a los distintos objetos para las escenas de las prácticas 3 y 4, de forma que se pueden seleccionar partes de dichos modelos. Respecto a extender el grafo original que se diseñó en la práctica 3, se dan más detalles en la sección 5.6.

5.6. Extensión del grafo original. Documentación.

Como se ha indicado arriba, el estudiante debe de asignar identificadores y nombres a los distintos objetos para las escenas de las prácticas 3 y 4, de forma que se pueden seleccionar partes de dichos modelos. En particular, se deben añadir identificadores de objetos para las distintas partes o componentes del grafo original que se definió en la práctica 3 y que después se extendió con materiales en la práctica 4.

Una vez asignados identificadores al grafo original, se debe producir un archivo PDF que documente dichos identificadores. Para ello se parte del PDF del documento de la práctica 4, extendiéndolo con

la información relativa a los distintos identificadores de selección que se hayan usado (sin borrar nada de lo que ya había). En concreto el archivo PDF contendrá una nueva sección al final, titulada **Práctica 5 - Identificadores de selección**. Esa sección tendrá los siguientes contenidos:

- El grafo de escena tipo PHIGS incluyendo las transformaciones (práctica 3), las entradas de tipo material (práctica 4) y además, para cada nodo u objeto, se mostrará el identificador de selección usado para el nodo (un valor entero). Se incluirá en la parte superior de cada nodo, incluyendo los casos en los que dicho identificador es 0 (no seleccionable), o es -1 (se hereda del padre).
- Una lista con información de todos y cada uno de los identificadores de selección que se han añadido al grafo, para cada uno de ellos:
 - Breve descripción o nombre del identificador y descripción de las partes del modelo a las que se asocia.
 - Valor numérico entero del identificador, y, si hay en el programa alguna constante con ese valor, nombre de la constante.
 - Para cada uso de este identificador en un nodo u objeto del grafo de escena: nombre de la clase del objeto, nombre de archivo y línea donde se asocia el identificador con el objeto de clase (mediante una llamada a **ponerIdentificador**)

5.7. Gestión interactiva de cámaras de 3 modos.

La clase **Escena** incluye un vector (**camaras**) de punteros a objetos de clases derivadas de **CamaraInteractiva**. En el constructor de la clase **Escena** (en **Escena.hpp** en **escena.cpp**) añadiremos a dicho vector varias instancias de la clase **Camara3Modos**, y quitaremos la instancia de la clase **CamaraOrbitalSimple**. Por tanto, a partir de ahora cada instancia de **Escena** incluirá más de una cámara.

Al menos se habrán de colocar tres cámaras, ofreciendo inicialmente las vistas clásicas de frente, alzado y perfil de la escena completa. Al menos una de ellas deberá tener proyección ortogonal y al menos otra proyección en perspectiva. Usando el teclado se podrá cambiar la cámara actual, pasando a la siguiente, o de la primera a la última (ver la sección sobre teclas e interacción).

Para añadir las cámaras de distintas características usaremos el constructor de la clase **Camara3Modos** que acepta 5 parámetros: (1) tipo de proyección (perspectiva u ortográfica), (2) origen del marco de cámara (posición del observador), (3) ratio del viewport (alto/ancho), (4) punto de atención y (5) apertura vertical de campo (en grados). Podemos usar distintos valores de dichos parámetros. La apertura de campo vertical debe ser un valor real en grados, mayor que 0° y menor que 180° (entre 50° y 80° es razonable).

Hay que tener en cuenta que el constructor de la clase **Camara3Modos** ya está implementado en **camara.hpp**. En ese constructor se crean correctamente una cámara perspectiva u ortográfica. En la figura 5.2 se observa la diferencia entre ambos tipos de cámaras.

5.7.1. Uso del teclado

Usando las teclas es posible cambiar los parámetros de las cámaras, según se detalla más arriba en la sección sobre las teclas. En el archivo **eventos-teclado.hpp** la función gestora de evento de teclado (**FGE_PulsarLevantarTecla**) ya incluye el código necesario para actualizar la cámara actual de la escena actual, invocando a los métodos correspondientes de la clase **Camara**: el método **desplRotarXY** para las teclas de flechas (izquierda, derecha, arriba o abajo), y el método **moverZ** para las teclas + y -. Las teclas izquierda y derecha producen desplazamiento o rotación

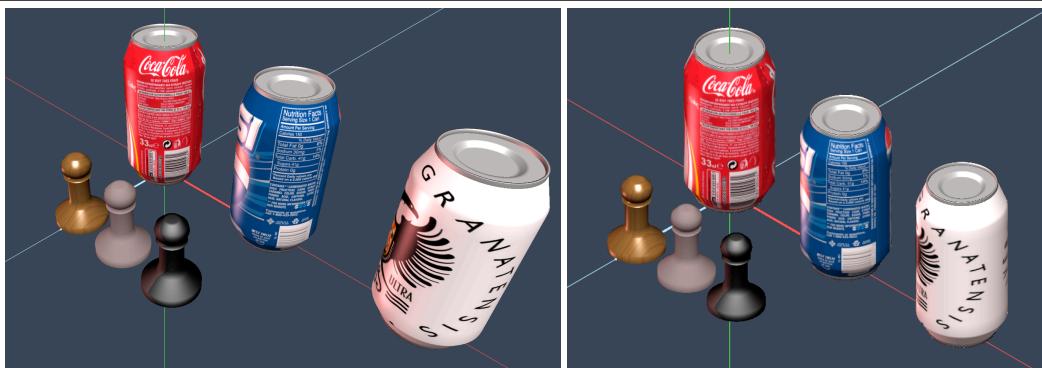


Figura 5.2: Escena de la práctica 5, vista con una cámara con proyección perspectiva (izquierda) y con una cámara con proyección ortográfica (derecha).

en horizontal (en función del modo de la cámara), y las teclas arriba o abajo en vertical.

5.7.2. Uso del ratón

Se podrá editar la cámara activa usando el ratón, en concreto manteniendo pulsado el botón derecho. Cuando se pulsa el botón derecho, se registran las coordenadas (enteras) del pixel donde se ha pulsado el citado botón. Hasta que se levante, la aplicación está en modo *arrastrar*. Cuando se levanta el botón derecho, se abandona el modo *arrastrar*.

En el modo *arrastrar*, cada vez que se produce un evento de movimiento del ratón (mientras se mantiene pulsada su tecla derecha), se actualiza la cámara actual. Para ello se usa el desplazamiento relativo del ratón (en unidades de pixels) desde el pixel donde se pulsó el botón derecho hasta el pixel actual. Ese desplazamiento se traduce en dos números enteros, positivos o negativos, que indican cuantas filas o columnas de pixels se ha movido el ratón. Esos desplazamientos se usan para seleccionar las distancias a desplazar la cámara (en modo primera persona), o bien los dos ángulos de rotación (en modo examinar).

La gestión del movimiento del ratón ya se encuentra programada en la función `FGE_MovimientoRaton` de `eventos-raton.cpp`. Cuando se mueve con el botón derecho pulsado (modo arrastrar), se llama al método `desplRotarXY` de la cámara actual de la escena actual. También es posible usar la rueda del ratón o los gestos de *scroll* en el touchpad, si está disponible. La función gestora de este último tipo de eventos (`FGE_Scroll`) invoca el método `moverZ` de la cámara actual.

5.7.3. Código de actualización de la cámara actual

El código de los métodos `desplRotarXY` y `moverZ` de la clase `Camara3Modos` (en el archivo `camara.cpp`) debe de completarse por el estudiante, en concreto se debe de implementar la actualización de las variables de instancia de la cámara en función de los parámetros `da` y `db` del método. Estos parámetros son proporcionales los desplazamientos horizontal y vertical (respectivamente) del ratón o los desplazamientos asociados a las teclas de las flechas (izquierda, derecha, arriba y abajo). En concreto, se deben de actualizar las variables de instancia (las tres de tipo `glm::vec3`):

- **org_polares**: coordenadas esféricas del origen de la cámara, relativas al punto de atención (el vector tiene dos ángulos: latitud y longitud, y un radio o modulo del vector).
- **org_cartesianas**: coordenadas (del mundo) cartesianas del origen de la cámara, relativas

al punto de atención.

- **punto_atencion**: coordenadas (del mundo) del punto de atención.

Para todo esto se deben usar los algoritmos que se detallan en las transparencias del tema 4 de teoría. Será necesario invocar las funciones **Cartesianas** y **Esféricas** para convertir entre coordenadas cartesianas y esféricas en ambos sentidos. Ambas funciones ya se encuentran implementadas en el archivo **camara.cpp**. Una vez actualizadas estas tres variables, si ha cambiado la orientación de la cámara, se debe llamar al método **actualizarEjesMCV** (ya implementado) para recalcular los ejes del marco de cámara (no es necesario hacerlo si la modificación de la cámara únicamente supone desplazamiento del origen, pero la cámara mantiene la orientación).

También es necesario implementar el método **mirarHacia** de la clase **Camara3Modos**. Este método se usará tras seleccionar un objeto, y sirve para poner la cámara en modo examinar y fijar el nuevo punto de atención como igual al vector que se pasa como parámetro, todo ello sin mover la posición del observador (sin cambiar el origen de la cámara), pero obviamente cambia la orientación de la cámara. También se describe en las transparencias como hacerlo.

5.7.4. Apuntar la cámara al objeto al que se hace click.

El estudiante debe de añadir código a la implementación método virtual **cuandoClick** de la clase **Objeto3D**, ese método se ejecuta por defecto cuando se hace click sobre un objeto con un identificador de selección propio (si el objeto es de una clase que no redefine el método). En ese método ya se imprime el nombre del objeto.

En concreto, el código a añadir debe usar el puntero a la escena actual (variable **escena** local de ese método) para recuperar su cámara actual, con el método **camaraActual**. Después se hace que esa cámara apunte al punto central de este objeto (es el parámetro **centro_wc** del método), usando para ello el método **mirarHacia** de esa cámara.

5.8. Selección.

Usando el ratón (pulsando con el botón izquierdo en un pixel de la ventana) se podrá seleccionar alguno de los objetos en la escena.

Para implementar la selección, el programa contempla que cada vez que se pulse el botón izquierdo del ratón se ejecute la FGE de nombre **FGE_PulsarLevantarBotonRaton**, ya implementada en **eventos-raton.cpp**. En esa FGE se invoca el método **seleccion** de la instancia **aplicacionIG** (clase **AplicacionIG**), este método se encuentra implementado en el archivo **seleccion.cpp**. Dicho método recibe como parámetros: las coordenadas del pixel donde se ha pulsado, un puntero a la escena actual, y el contexto de visualización actual, y se encarga de gestionar el click de la forma adecuada, lo cual implica invocar el método **cuandoClick** del objeto sobre el que se ha hecho click, si hay alguno.

El estudiante debe completar el código del método **seleccion**, dando los pasos que se indican en la sub-sección 5.8.4. Será asimismo necesario implementar el código del método **visualizarModoSeleccion** de las clases **MallaInd** y **NodoGrafoEscena** (ver la sección 5.8.2), y el método **visualizarGL_Seleccion** de la clase **Escena**. Finalmente, en el archivo **seleccion.cpp** será necesario completar el código del método **seleccion**.

5.8.1. El método `cuandoClick` de la clase `Objeto3D`.

El método `seleccion` de la clase `AplicacionIG` debe invocar el método `cuandoClick` de un objeto, tras haber sido clicado ese objeto, como se ha indicado antes. Este método es un método virtual de la clase `Objeto3D`, y recibe como parámetro un `vec3` con el punto central del objeto (`centro_wc`). Por defecto, si no se redefine, el comportamiento es que se describe aquí:

- Se imprime el nombre del objeto.
- La cámara activa de la escena actual pasará a modo examinar (si no lo estaba ya) y se centrará en `centro_wc` (el punto de atención de dicha cámara quedará fijado en ese punto, para eso se usa el método `mirarHacia` de la cámara).

El valor devuelto por el método es un valor lógico, cuando es `true` indica que el estado del objeto (o de algún parámetro de configuración) ha cambiado de forma que es necesario volver a visualizar la imagen de la escena actual. La versión por defecto devuelve `true`, ya que modifica la cámara actual.

Este método puede cambiar el estado del objeto (o de la aplicación en general) de forma arbitraria, por ejemplo puede cambiar el color del objeto, o alguna de sus transformaciones, o puede imprimir información del objeto. Para ello debe ser redefinido en clases derivadas de `Objeto3D`, y devolver `true` cuando haya cambiado alguna variable y por tanto haya que revisualizar.

5.8.2. Visualización de mallas y nodos en modo selección

El método `visualizarModoSeleccionGL` de `MallaInd` y de `NodoGrafoEscena` se encarga de visualizar la malla o el nodo en *modo de selección*, esto implica visualizar la geometría, pero usando un colores obtenidos a partir de los identificadores de los objetos. A continuación se describen estos métodos en más detalle.

5.8.2.1. El método `visualizarModoSeleccionGL` en la clase `MallaInd`.

En este método, para la visualización de la geometría se debe usar el método `visualizarGeomGL`. Por tanto, el estudiante debe escribir código que da estos pasos:

1. Leer el identificador del objeto (con `leerIdentificador`). Si el objeto tiene identificador (es decir, si su identificador no es -1), entonces:
 - Hacer push del color del cauce, con `pushColor`.
 - Fijar el color del cauce (con `fijarColor`) usando un color obtenido a partir del identificador (con `ColorDesdeIdent`).
2. Invocar `visualizarGeomGL` para visualizar la geometría.
3. Si tiene identificador: hacer pop del color, con `popColor`.

5.8.2.2. El método `visualizarModoSeleccionGL` en la clase `NodoGrafoEscena`.

En este método se hace igual que en el anterior en cuanto a hacer *push* (al inicio) y *pop* (al final) del color del cauce cuando el nodo tiene un identificador. Se debe recorrer el vector de entradas, para las de tipo sub-objeto se invoca recursivamente a `visualizarModoSeleccionGL`. Por supuesto, se debe gestionar adecuadamente la matriz de modelado y las entradas transformación.

El estudiante debe de escribir código para dar estos pasos:

1. Leer el identificador del nodo (con `leerIdentificador`), si el identificador no es -1:
 - Guardar una copia del color actual del cauce (con `pushColor`)
 - Fijar el color del cauce de acuerdo al identificador, (usar `ColorDesdeIdent`).
2. Guardar una copia de la matriz de modelado (con `pushMM`).
3. Recorrer la lista de nodos y procesar las entradas transformación o subobjeto:
 - En las entradas subobjeto: invocar recursivamente a `visualizarModoSeleccionGL`.
 - Para las entradas transformación, componer la matriz (con `compMM`).
4. Restaurar la matriz de modelado original (con `popMM`).
5. Si el identificador no es -1, restaurar el color previo del cauce (con `popColor`).

5.8.3. Visualización de la escena en modo de selección

Cuando se hace click sobre un pixel, es necesario visualizar la escena en modo de selección, usando la misma cámara actual, pero configurando el OpenGL y el cauce de forma adecuada a la selección. Todo esto se hace en el método `visualizarGL_Selección` de la clase `Escena`.

La configuración de OpenGL y del cauce requiere activar el modo de visualización relleno, desactivar la iluminación y las texturas, activar la cámara actual de la escena. La visualización se hace usando el método `visualizarModoSeleccionGL` del objeto raíz.

El estudiante debe escribir código para dar estos pasos:

1. Configurar estado de OpenGL:
 - Fijar el viewport (con la función `glViewport`), usando el tamaño de la ventana actual (está en `AplicacionIG`),
 - Fijar el modo de polígonos a *relleno*, con `glPolygonMode`.
2. Activar y configurar el cauce:
 - Activar el cauce (con su método `activar`).
 - Desactivar iluminación y texturas.
 - Poner el color actual a 0 (por defecto los objetos no son seleccionables).
3. Limpiar el framebuffer (color y profundidad), usando el color RGB (0,0,0), ya que al inicio en ningún pixel debe haber nada seleccionable.
4. Recuperar la cámara de la escena (con `camaraActual`) y activar dicha cámara en el cauce (con `activar`)
5. Recuperar (con `objetoActual`) el objeto raíz actual de esta escena y visualizarlo con su método `visualizarModoSeleccionGL`.

5.8.4. Gestión de clicks en el método `seleccion` de `AplicacionIG`

El método `seleccion` de la clase `AplicacionIG` se ejecutará cuando se haga click en un pixel. El método recibe las coordenadas del pixel. En el método se creará un objeto framebuffer (FBO) y se hará sobre dicho framebuffer la visualización de la escena, en modo selección. Después se lee el identificador que haya quedado en el pixel del FBO donde se ha pulsado, y si hay algún identificador se busca ese identificador en el objeto actual, si se encuentra se ejecuta el método `cuandoClick` del mismo.

Por tanto, el estudiante debe escribir código para dar estos pasos:

1. Crear (la primera vez) y activar el objeto framebuffer (puntero `fbo` de la clase `AplicacionIG`), se dan estos dos pasos:
 - Si el puntero `fbo` es nulo, crear el objeto framebuffer usando su constructor (necesita

- el tamaño actual de la ventana en dos parámetros).
- Activar el framebuffer, usando su método `activar` (a partir de ahora, las llamadas de visualización se hacen sobre dicho framebuffer).
2. Visualizar la escena actual en modo selección. Se usará el método `visualizarGL_Selection` de la clase `Escena`.
 3. Leer el identificador del pixel en las coordenadas del click, se usa `LeerIdentEnPixel`.
 4. Desactivar el FBO (vuelve a activar el FBO por defecto, el FBO 0), se usa el método `desactivar` del objeto `fbo`.
 5. Si el identificador del pixel es 0, imprimir mensaje y terminar (devolver `false`).
 6. Buscar el identificador en el objeto actual de la escena, se usa el método `buscarObjeto` de `Objeto3D`. Si se encuentra un objeto con ese identificador, hay que ejecutar el método `cuandoClick` de dicho objeto y hacer `return` del valor devuelto por `cuandoClick`.

5.8.5. Búsqueda recursiva de un identificador y cálculo del centro.

En esta práctica se requiere ser capaz de identificar objetos de la escena (asociar algún tipo de identificador entero a cada objeto, para que, conocido un identificador, podamos encontrar el objeto), y ser capaz de conocer el centro de cada objeto (para que al seleccionarlo la cámara pase al modo examinar, viendo dicho centro del objeto en el centro de la pantalla).

Estos requerimientos hacen necesario añadir información a los objetos 3D. Para ello, en cada instancia de la clase `Objeto3D`, se han añadido estas dos variables de instancia privadas:

- **Identificador:** es un entero entre 0 y 2^{24} (cabe en 3 bytes), no necesariamente único. Puede valer:
 - -1 para indicar que el objeto no tiene asociado identificador propio. Si el objeto es parte de una jerarquía, este valor indica que el identificador es el mismo del padre o del ancestro más cercano con identificador distinto de -1 (esto permite referenciar un nodo desde distintos padres con distintos identificadores en los padres).
 - 0 indica que el objeto no es seleccionable, es decir, que se debe ignorar en los procesos de búsqueda.
 - > 0 el objeto tiene un identificador determinado y es seleccionable.
- Este identificador se puede fijar con el método `ponerIdentificador` y se puede consultar con `leerIdentificador` (ambos métodos ya están implementados). Inicialmente es -1 para todos los objetos.
- **Centro:** es una tupla con tres flotantes que contiene un punto central al objeto, en coordenadas de objeto. Por defecto vale (0, 0, 0) para cualquier objeto. Se puede cambiar con el método `ponerCentroOC` y se puede consultar con `leerCentroOC` (ambos métodos ya están implementados).

Para poder buscar un nodo dando un identificador (mayor estricto que cero), se debe usar un nuevo método de la clase `Objeto3D`, que se encarga de buscar dicho identificador (dentro del objeto) y devolver un puntero al primer sub-objeto u objeto encontrado que lo tenga.

```
virtual bool buscarObjeto( const int ident_busc,
                           const Matriz4f & mmodelado,
                           Objeto3D ** objeto,
                           glm::vec3 & centro_wc );
```

La función (virtual), devuelve `true` si el objeto se ha encontrado, o `false` en otro caso. Los pará-

metros son:

- **ident_busc**: es el identificador (mayor estricto que cero) del objeto que queremos buscar. El valor de **ident_busc** no puede ser 0 ni negativo, en ese caso se debe producir un error.
- **mmodelo**: es un parámetro de entrada, contiene la matriz de modelado del objeto padre (si el objeto es la raíz de un grafo, será la matriz identidad).
- **objeto**: parámetro de salida. Si es un puntero nulo, no sirve para nada. Si apunta a un puntero, y el resultado es **true**, se escribirá en el puntero el puntero al objeto encontrado.
- **centro_wc**: parámetro de salida. Si el resultado es **true**, contiene la posición en coordenadas de mundo del punto central del objeto seleccionado.

El método **buscarObjeto** tiene una implementación genérica por defecto para la clase **Objeto3D**, en dicha implementación simplemente se comprueba si el identificador que se busca coincide con el identificador del objeto, y se devuelve lo que corresponda (ver la implementación en **Objeto3D.cpp**).

Se debe completar la implementación distinta del método **buscarObjeto** en la clase **NodoGrafoEscena** (en **grafo-escena.cpp**). Se dan estos pasos:

1. Invocar a **calcularCentroOC** para calcular el centro del objeto (ese método no hace nada si el centro ya estaba calculado).
2. Comparar el identificador del nodo con el identificador que se está buscando. Si coinciden ya se ha encontrado el identificador, se debe de escribir **objeto** y **centro_wc** y terminar.
3. En otro caso, se debe invocar el método recursivamente para cada uno de las entradas con un puntero a objeto, en orden, terminando cuando se encuentre. En cada llamada hay que componer la matriz de modelado actual (que va cambiando con cada entrada de tipo de transformación). Las entradas de tipo material se ignoran.

El método **calcularCentroOC** de **Objeto3D** no hace nada por defecto, así que en la mayoría de los objetos, si no se redefine este método, se usará el centro inicial, que es el punto (0, 0, 0) en coordenadas de objeto. Esto funciona bien para la mayoría de las mallas indexadas y objetos de revolución, que suelen tener en su interior este punto (si se quiere cambiar esto se puede, opcionalmente, declarar y redefinir **calcularCentroOC** para la clase correspondiente).

Para el caso de objetos de tipo **NodoGrafoEscena**, se debe completar el cálculo del centro de forma recursiva, es decir, hay que completar **calcularCentroOC** en el archivo **grafo-escena.cpp**. Este cálculo se hace solo la primera vez que se invoca el método, al hacerlo se pone a **true** la variable de instancia **centro_calculado**, y después ya no se repite (se comprueba dicha variable al inicio del método).

Si es necesario calcular el centro de un nodo del grafo, se recorren recursivamente todas las entradas y se calcula el centro de todas las que sean de tipo sub-objeto (llamando recursivamente al método y transformando el punto obtenido por la correspondiente matriz de modelado de esa entrada). Al final, se puede fijar como centro del nodo el centro de la caja englobante de los centros de los sub-objetos, o el promedio de dichos centros.

5.9. Ejercicios adicionales

5.9.1. Ejercicio 1

(Nota: convocatoria ordinaria del curso 20-21, enunciado adaptado para usar **cuandoClick** y algún detalle más).

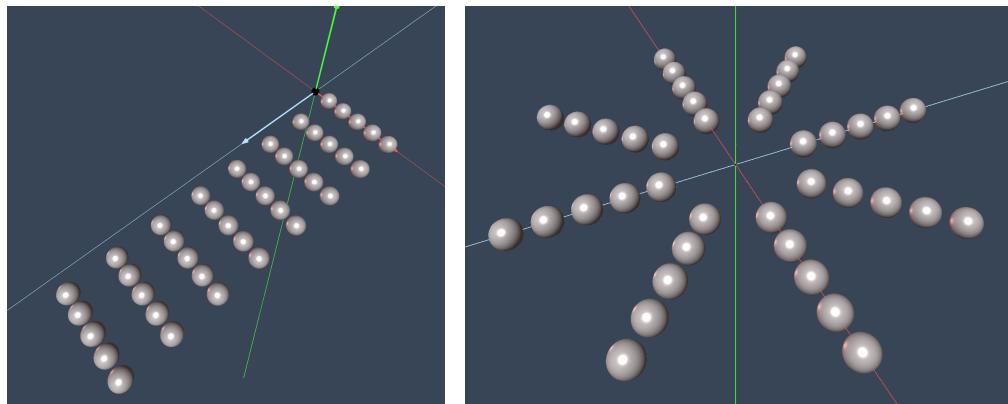


Figura 5.3: Imágenes de los grafos de escena mencionados en el ejercicio adicional 1 (izquierda) y 2 (derecha) de la práctica 5.

Escribe una clase llamada **GrafoEsferasP5**, derivada de **NodoGrafoEscena**, cuyo constructor produce un grafo de escena formado por varias filas de esferas. Cada esfera es una instancia de la nueva clase **EsferaGrafoP5**, derivada de **MallaInd**, según se puede observar en la figura 5.3 (izquierda). Copia el código de aquí:

```
GrafoEsferasP5::GrafoEsferasP5()
{
    const unsigned
        n_filas_esferas      = 8,
        n_esferas_x_fila     = 5 ;
    const float
        e = 0.4/n_esferas_x_fila ;

    agregar( MAT_Escalado( e,e,e ) );

    for( unsigned i = 0 ; i < n_filas_esferas ; i++ )
    {
        NodoGrafoEscena * fila_esferas = new NodoGrafoEscena() ;

        for( unsigned j = 0 ; j < n_esferas_x_fila ; j++ )
        {
            MiEsferaEl * esfera = new MiEsferaEl(i,j) ;
            fila_esferas->agregar( MAT_Traslacion( 2.2, 0.0, 0.0 ) );
            fila_esferas->agregar( esfera );
        }
        agregar( fila_esferas );
        agregar( MAT_Traslacion( 0.0, 0.0, 5.0 ) );
    }
}
```

Añade (como último objeto de la escena) una instancia de **GrafoEsferasP5** en el constructor de **Escena5**.

Añádele al constructor de **GrafoEsferasP5** las líneas de código necesarias para añadirle identificadores de selección al grafo de escena. Debes definir una clase nueva (**MiEsferaEl**), derivada de **NodoGrafoEscena**, para las esferas. Añade a esa clase su propia versión del método virtual **cuandoClick**, de forma que se imprima un mensaje como este:

Se ha seleccionado la esfera número E de la fila número F

Donde *E* es el número de esfera en su fila (comenzando en 1), y *F* es el número de fila (tmb comenzando en 1).

5.9.2. Ejercicio 2

(Nota: es de la convocatoria ordinaria del curso 20-21, enunciado adaptado para usar **cuandoClick** y algún detalle más).

Escribe una clase llamada **GrafoEsferasP5_2**, derivada de **NodoGrafoEscena**, cuyo constructor produce un grafo de escena formado por varias filas de esferas, según se puede observar en la figura 5.3 (derecha). Copia el código de aquí:

```
GrafoEsferasP5_2::GrafoEsferasP5_2()
{
    const unsigned
        n_filas_esferas      = 8,
        n_esferas_x_fila     = 5 ;
    const float
        e = 2.5/n_esferas_x_fila ;

    agregar( MAT_Escalado( e, e, e ));

    for( unsigned i = 0 ; i < n_filas_esferas ; i++ )
    {
        NodoGrafoEscena * fila_esferas = new NodoGrafoEscena() ;
        fila_esferas->agregar( MAT_Traslacion( 3.0, 0.0, 0.0 ));

        for( unsigned j = 0 ; j < n_esferas_x_fila ; j++ )
        {
            MiEsferaE2 * esfera = new MiEsferaE2() ;
            fila_esferas->agregar( MAT_Traslacion( 2.5, 0.0, 0.0 ));
            fila_esferas->agregar( esfera );
        }
        agregar( fila_esferas );
        agregar( MAT_Rotacion( 360.0/n_filas_esferas, { 0.0, 1.0, 0.0 }));
    }
}
```

Añade (como último objeto de la escena) una instancia de **GrafoEsferasP5_2** en el constructor de **Escena5**.

Añádele al constructor de **GrafoEsferasP5_2** las líneas de código necesarias para añadirle identificadores de selección al grafo de escena. Define una clase nueva (**MiEsferaE2**), derivada de **NodoGrafoEscena**, para las esferas. Añade a esa clase su propia versión del método virtual **cuandoClick**, de forma que al ejecutarse se cambie el color de la esfera de blanco a rojo (o de rojo a blanco, si estaba rojo).