

Prácticas de Informática Gráfica

Grado en Informática y Matemáticas.

Grado en Informática y Administración y Dirección de Empresas
Curso 2022-23.



**UNIVERSIDAD
DE GRANADA**

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

Índice general.

Índice.	3
0. Prerrequisitos, materiales y compilación	7
0.1. Prerrequisitos software	7
0.1.1. Sistema operativo Linux	7
0.1.2. Sistema Operativo macOS	8
0.1.3. Sistema Operativo Windows	10
0.1.4. Uso de <i>VS Code</i> de Microsoft	10
0.2. Materiales y compilación	10
0.2.1. Materiales para las prácticas. Entregas.	10
0.2.2. Compilación en la línea de órdenes	11
0.2.3. Edición y compilación con <i>VS Code</i>	12
0.3. Clases auxiliares	12
0.3.1. Clases para tuplas de valores enteros o reales con 2,3 o 4 valores	12
1. Visualización de modelos simples	15
1.1. Objetivos	15
1.2. Desarrollo	15
1.3. Teclas a usar. Interacción.	15
1.4. Estructura del código: funciones y clases.	16
1.4.1. Contexto y modos de visualización y de envío	16
1.4.2. Funciones para crear y activar VBOs y VAOs	17
1.4.3. Clase abstracta para objetos gráficos 3D (Objeto3D)	17
1.4.4. Clase para mallas indexadas (MallaInd)	18
1.4.5. La clase Escena	20
1.4.6. Programación del cauce gráfico. Clase Cauce	21
1.4.7. Clases para los objetos de la práctica 1	21
1.5. Tareas	21

1.5.1. En el archivo escena.cpp	21
1.5.2. En el archivo malla-ind.cpp	22
1.5.3. Clases nuevas para otros tipos de mallas indexadas	22
1.6. Ejercicios adicionales	23
1.6.1. Ejercicio 1	23
1.6.2. Ejercicio 2	24
1.6.3. Ejercicio 3	25
2. Modelos PLY y Poligonales	27
2.1. Objetivos	27
2.2. Desarrollo	27
2.3. Algoritmo para la creación de malla por revolución	29
2.4. Estructura del código. Clases.	31
2.4.1. La clase Escena2	31
2.4.2. Clase MallaPLY : mallas creadas a partir de un archivo PLY	31
2.4.3. Clase MallaRevol : mallas obtenidas por revolución de un perfil	32
2.4.4. Clase MallaRevolPLY : revolución de un parfil leído en un PLY	32
2.4.5. Clase: Cilindro, Cono, Esfera	33
2.4.6. Archivos PLY disponibles.	33
2.5. Tareas	34
2.6. Ejercicios adicionales	34
2.6.1. Ejercicio 1	34
2.6.2. Ejercicio 2	35
2.6.3. Ejercicio 3	35
3. Modelos jerárquicos	37
3.1. Objetivos	37
3.2. Desarrollo	37
3.3. Teclas a usar. Gestión de animaciones.	38
3.4. Diseño e implementación de un grafo de escena parametrizado original	39
3.4.1. Contenidos del archivo PDF con la documentación.	39

3.4.2. La clase Escena3	40
3.5. Implementación del código de visualización. Colores.	40
3.6. Implementación de parámetros y animaciones	41
3.6.1. Definición de clases para objetos parametrizados	42
3.6.2. Definición de nodos del grafo de escena parametrizados	43
3.7. Algunos ejemplos de modelos jerárquicos	44
3.8. Ejercicios adicionales	44
3.8.1. Ejercicio 1	44
3.8.2. Ejercicio 2	45
4. Materiales, fuentes de luz y texturas	49
4.1. Objetivos	49
4.2. Desarrollo	49
4.3. Teclas a usar	50
4.4. Clases y métodos a añadir o completar.	50
4.4.1. Las clases FuentesLuz y ColFuentesLuz	50
4.4.2. Clase Textura	51
4.4.3. Clase Material	52
4.4.4. Añadidos a la clase NodoGrafoEscena	52
4.4.5. Añadidos a la clase MallaInd y derivadas	52
4.4.6. Añadidos a la clase Escena	53
4.4.7. Visualización de normales de vértices	54
4.4.8. Clase Escena4	54
4.4.9. Clase LataPeones	54
4.4.10. Materiales y textura en el grafo de la práctica 3	56
4.5. Cálculo de tablas de normales y coordenadas de textura	57
4.5.1. Clases Cubo24 y NodoCubo24	57
4.5.2. Clase MallaRevol	58
4.5.3. Clases Cubo , Tetraedro y MallaPLY	59
4.6. Ejercicios adicionales	61
4.6.1. Ejercicio 1	62

4.6.2. Ejercicio 2	63
5. Interacción: cámaras y selección.	65
5.1. Objetivos	65
5.2. Desarrollo	65
5.3. Teclas e interacción con el ratón.	66
5.4. Escena de la práctica 5.	66
5.5. Gestión interactiva de cámaras de 3 modos.	67
5.5.1. Uso del teclado	68
5.5.2. Uso del ratón	68
5.5.3. Código de actualización de la cámara actual	69
5.6. Selección.	69
5.6.1. Visualización de mallas y nodos en modo selección	70
5.6.2. La función Seleccion	70
5.6.3. Búsqueda recursiva de un identificador y cálculo del centro.	70
5.7. Documento con grafo de escena incluyendo identificadores de selección	72
5.8. Ejercicios adicionales	73
5.8.1. Ejercicio 1	73
5.8.2. Ejercicio 2	74

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

0.1.1. Sistema operativo Linux

Compiladores

Para hacer las prácticas de esta asignatura en Ubuntu, en primer lugar debemos de tener instalado algún compilador de C/C++. Se puede usar el compilador de C++ open source CLANG o bien el de GNU (ambos pueden coexistir). Podemos usar **apt** para instalar el paquete **g++** (compilador de GNU) o bien **clang** (compilador del proyecto LLVM).

Driver de la GPU

Respecto de la librería OpenGL, es necesario tener instalado algún driver de la tarjeta gráfica disponible (GPU) en el ordenador. Incluso si no hay tarjeta gráfica disponible (por ejemplo en un máquina virtual eso puede ocurrir), es posible usar OpenGL implementado en software (aunque es más lento). Lo más probable es que tu instalación ya cuente con el driver apropiado para la tarjeta gráfica.

En cualquier caso, en Ubuntu, si no tenemos instalado el driver y instalarlo, sería necesario en primer lugar averiguar el fabricante y modelo de la GPU, lo cual puede hacerse con esta orden:

```
sudo lshw -c video
```

Una vez conocido el fabricante y modelo de la GPU (nVidia, Intel, AMD), podemos seguir las instrucciones específicas de cada uno de los fabricantes para instalar el correspondiente driver.

Librería GLEW

La librería GLEW es necesaria en Linux para que las funciones de la versión 2.1 de OpenGL y posteriores puedan ser invocadas (inicialmente esas funciones abortan al llamarlas). GLEW se encarga, en tiempo de ejecución, de hacer que esas funciones estén correctamente enlazadas con su código.

Para instalarla, se puede usar el paquete debian **libglew-dev**. En Ubuntu, se usa la orden

```
sudo apt install libglew-dev
```

Librería GLFW

La librería GLFW se usa para gestión de ventanas y eventos de entrada. Se puede instalar con el paquete debian **libglfw3-dev**. En Ubuntu, se puede hacer con:

```
sudo apt install libglfw3-dev
```

Librería JPEG

Esta librería sirve para leer y escribir archivos de imagen en formato jpeg (extensiones **.jpg** o **.jpeg**). Se usará para leer las imágenes usadas como texturas. La librería se debe instalar usando el paquete debian **libjpeg-dev**. En Ubuntu, se puede hacer con la orden

```
sudo apt install libjpeg-dev
```

0.1.2. Sistema Operativo macOS

Compilador

En primer lugar, para poder compilar los programas fuente en C++, debemos asegurarnos de tener instalado y actualizado **XCode** (conjunto de herramientas de desarrollo de Apple, incluye compiladores de C/C++ y entorno de desarrollo). Una vez instalado, usaremos la implementación de OpenGL que se proporciona con XCode (la librería GLEW no es necesaria en este sistema operativo). Para verificar que se ha instalado correctamente, podemos probar a ejecutar en la línea de órdenes:

```
clang++ --version
```

(la versión es la 10.0.1 en julio de 2019).

Uso de **homebrew** para CMake, GLFW y librería JPEG

Homebrew es un gestor de paquetes para macOS que permite instalar fácilmente librerías de desarrollo. Nos permite instalar fácilmente la librería GLFW y la librería de lectura de archivos JPEG. Para compilar el código es necesario disponer de la orden **cmake**, la cual también se puede instalar con **Hombrew**.

Si no tenemos disponible **homebrew**, debemos de instalarlo antes según las instrucciones en la página Web ([brew.sh](#)).

Para instalar CMake, GLFW y la librería de JPEG simplemente será necesario usar los correspondientes paquetes ([cmake](#), [glfw](#) y [jpeg](#)), usando estas órdenes en la línea de comandos:

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

Si todo va bien, esto dejará los archivos `.h` en `/usr/local/include` (o en el caso de GLFW, en la subcarpeta `GLFW` dentro de la anterior), y los archivos `.a` o `.dylib` en `/usr/local/lib`. Para comprobar que `cmake` se ha instalado correctamente, teclea

```
cmake --version
```

Instalación de **CMake** sin *Hombrew*

La orden CMake se puede instalar con el archivo `.dmg` para macOS que se encuentra en esta página web:

☞ <https://cmake.org/download/>

Una vez descargado e instalado este archivo, se puede comprobar que está disponible en la línea de órdenes, escribiendo:

```
cmake --version
```

Instalación de **GLFW** sin *homebrew*

Para instalarla, se debe acceder a la página de descargas del proyecto GLFW:

☞ <http://www.glfw.org/download.html>

Aquí se descarga el archivo `.zip` pulsando en el recuadro titulado *source package*. Después se debe abrir ese archivo `.zip` en una carpeta nueva vacía. En ese carpeta vacía se crea una subcarpeta raíz (de nombre `glfw-...`). Después compilamos e instalamos la librería con estas órdenes:

```
cd glfw-....  
cmake -DBUILD_SHARED_LIBS=ON .  
make  
sudo make install
```

Si no hay errores, esto debe instalar los archivos en la carpeta `/usr/local/include/GLFW` (cabeceras `.h`) y en `/usr/local/lib` (archivos de librerías dinámicas con nombres que comienzan con `libglfw` y con extensión `.dylib`,)

Instalación de la librería **JPEG** sin *homebrew*

Es necesario instalar los archivos correspondientes a la versión de desarrollo de la librería de lectura de archivos de imagen en formato JPEG. Se debe compilar el código fuente de esta librería, para ello basta con descargar el archivo con el código fuente de la versión más moderna a un carpeta nueva vacía, y después compilar e instalar los archivos. Se puede hacer con esta secuencia de órdenes:

```
mkdir carpeta-nueva-vacia  
cd carpeta-nueva-vacia  
curl --remote-name http://www.ijg.org/files/jpegsrc.v9b.tar.gz  
tar -xzvf jpegsrc.v9b.tar.gz  
cd jpeg-9b
```

```
./configure  
make  
sudo make install
```

Estas órdenes se refieren a la versión 9b de la librería, para futuras versiones habrá que cambiar **9b** por lo que corresponda. Si todo va bien, esto dejará los archivos **.h** en **/usr/local/include** y los archivos **.a** o **.dylib** en **/usr/local/lib**. Para poder compilar, debemos de asegurarnos de que el compilador y el enlazador tienen estas carpetas en sus paths de búsqueda, es decir, debemos de usar la opción **-I/usr/local/include** al compilar, y la opción **-L/usr/local/lib** al enlazar.

0.1.3. Sistema Operativo Windows

Las prácticas se pueden desarrollar en Windows.

Para compilar en Windows se puede usar *Microsoft Visual Studio* (la versión *Community* no tiene coste alguno). Se debe de seleccionar las componentes para desarrollo de aplicaciones de escritorio. Visual Studio incorpora **cmake**, así como los archivos de cabecera (para incluir) y librerías (para enlazar) de OpenGL.

Para compilar GLEW y GLFW se pueden obtener los archivos de cabeceras y librerías precompiladas para Windows de las páginas Web de estas librerías. Sin embargo, en el momento de redactar esto no están todavía preparados y probados completamente los archivos de configuración de cmake para windows.

0.1.4. Uso de VS Code de Microsoft

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. 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 a otros editores (p.ej. *atom*), y en cualquier caso el programa siempre se puede compilar en la línea de órdenes con **cmake** (lo vemos a continuación).

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 dos 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.
- En estas subcarpetas es donde se aloja el archivo ejecutable correspondiente al sistema operativo.
- **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 raíz* y cuyo nombre puede ser cualquiera.

Las 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 y Linux usando el programa *CMake*.

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.

Sistemas operativos Linux y macOS, con *CMake*

Se deben de dar estos pasos:

- Ejecutar un terminal macOS o Linux, hacer **cd** a la carpeta **builds/macOS** o **builds/linux**, según corresponda.
- Hacer **cd** a la subcarpeta **cmake**, la cual debe estar vacía inicialmente. En esa carpeta, escribir una sola vez esta orden:

```
cmake ..
```

Si no hay errores, esto generará en la carpeta **cmake** diversos archivos y sub-carpetas, entre ellos el archivo **Makefile**.

- Cada vez que se quiera compilar, en la carpeta **cmake**, hay que escribir esta orden:

```
make
```

Si no hay errores, esta orden debe de generar el archivo ejecutable **pracs_ig_macos_exe** o **pracs_ig_linux_exe** en la carpeta **cmake**. Para eliminar los archivos generados al compilar y forzar que se vuelva a compilar todo, se puede ejecutar:

```
make clean
```

y luego de nuevo **make**

- Para ejecutar el archivo resultado de la compilación, basta con hacer **cd** a la carpeta **cmake** y escribir

```
./pracs_ig_linux_exe
```

o bien

```
./pracs_ig_macos_exe
```

Hay que tener en cuenta que si añadimos algún archivo en la carpeta **src** (un nuevo archivo **.cpp** o **.h**) es necesario volver a generar los archivos de compilación en **cmake** para que el nuevo archivo se tenga en cuenta al compilar. Para eso, podemos vaciar la carpeta **cmake** y después volver a ejecutar **cmake ..** en ella.

0.2.3. Edición y compilación con VS Code

El código fuente se puede editar, compilar y ejecutar desde el editor *code*. Para ello se incluyen en **builds/macOS** y en **builds/linux** sendos archivos de nombre **pracs-ig.code-workspace** (además de las carpetas de nombre **.vscode**). Para trabajar en el *espacio de trabajo (workspace)* de las prácticas se debe abrir un 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**.

Para generar los archivos de compilación (si no estaban ya) hay que pinchar el sub-menu *Terminal*, dentro de eso la opción *Ejecutar tarea....* y finalmente seleccionar la opción *Regenerar 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. Clases auxiliares

En esta sección se describen algunas clases auxiliares que se usarán en las prácticas

0.3.1. Clases para tuplas de valores enteros o reales con 2,3 o 4 valores

En el archivo **tup_mat.h** se declaran 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 (**Tupla2f**,**Tupla3f**,**Tupla3u**, etc...).

Aquí vemos ejemplos de uso de esta

```
// adecuadas para coordenadas de puntos, vectores o normales en 3D
// también para colores (R,G,B)
Tupla3f t1 ; // tuplas de tres valores tipo float
Tupla3d t2 ; // tuplas de tres valores tipo double

// adecuadas para la tabla de caras en mallas indexadas
Tupla3u t4 ; // tuplas de tres valores tipo unsigned

// adecuadas para puntos o vectores en coordenadas homogéneas
// también para colores (R,G,B,A)
Tupla4f t5 ; // tuplas de cuatro valores tipo float
Tupla4d t6 ; // tuplas de cuatro valores tipo double

// adecuadas para puntos o vectores en 2D, y coordenadas de textura
Tupla2f t7 ; // tuplas de dos valores tipo float
Tupla2d t8 ; // tuplas de dos valores tipo double
```

Este trozo código válido ilustra las distintas opciones, para creación, consulta y modificación de tuplas:

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

// declaraciones e inicializaciones de tuplas
Tupla3f a( 1.0, 2.0, 3.0 ), b, c(arr3f) ; // b indeterminado
Tupla3u d( 1, 2, 3 ), e, f(arr3u) ; // e indeterminado

// accesos de solo lectura, usando su posición o índice en la tupla (0,1,2,...),
// o bien varias constantes predefinidas para coordenadas (X,Y,Z) o colores (R,G,B):
float x1 = a(0), y1 = a(1), z1 = a(2), // 
    x2 = a(X), y2 = a(Y), z2 = a(Z), // apropiado para coordenadas
    re = c(R), gr = c(G), bl = c(B) ; // apropiado para colores

// conversiones a punteros
float * p1 = a ; // conv. a puntero de lectura/escritura
const float * p2 = b ; // conv. a puntero de solo lectura

// accesos de escritura
a(0) = x1 ; c(G) = gr ;

// escritura en un 'ostream' (cout) (se escribe como: (1.0,2.0,3.0)
cout << "la tupla 'a' vale: " << a << endl ;
```

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

```
// declaraciones de tuplas y de valores escalares
Tupla3f a,b,c ;
float s,l ;

// operadores binarios y unarios de asignación/suma/resta/negación
a = b ;
a = b+c ;
a = b-c ;
a = -b ;
```

```
// multiplicación y división por un escalar
a = 3.0f*b ;      // por la izquierda
a = b*4.56f ;     // por la derecha
a = b/34.1f ;     // mult. por el inverso

// otras operaciones
s = a.dot(b)      ; // producto escalar (usando método dot)
s = a|b           ; // producto escalar (usando operador binario barra )
a = b.cross(c)    ; // producto vectorial (solo para tuplas de 3 valores)
l = a.lengthSq()  ; // calcular módulo o longitud al cuadrado
a = b.normalized() ; // hacer a= copia normalizada de b (a=b/modulo de b) (b no cambia)
```

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, los modelos de un **tetraedro** y un **cubo**. Para ello, creará las estructuras de datos que permitan representarlos mediante sus vértices y caras (con una estructura de tipo *malla indexada*). Asimismo, escribirá el código necesario para visualizar mallas indexadas 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 **src/main.cpp**.

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), también en el archivo **src/main.cpp** (la gestión de la cámara se estudiará en la práctica 5).

1.4. Estructura del código: funciones y clases.

El archivo **main.cpp** (carpeta **src**) incluye código de inicialización (creación de cauces y escenas, inicialización de GLFW y OpenGL, en ese orden), y después la llamada al bucle principal para gestión de eventos de GLFW (**BucleEventosGLFW**). 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. Contexto y modos de visualización y de envío

En el archivo **practicas.h** (dentro de la carpeta **src**) se declara la clase **ContextoVis** (por *Contexto de Visualización*), que contiene, como variables de instancia, distintos parámetros y variables de estado usados durante la visualización de objetos y escenarios en las prácticas. Los métodos encargados de visualizar objetos tienen como parámetro un puntero (**cv**) a una instancia de esta clase. Esto permite usar una instancia para fijar todos esos parámetros, de forma que sean accesibles desde los citados métodos de visualización.

Entre otras cosas, el contexto de visualización contiene una variable 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. La declaración del tipo enumerado es así:

```
enum class ModosVisu
{
    relleno, lineas, puntos,
    num_modos // al convertirlo a entero, da el número de modos
};
```

La clase **ContextoVis** tiene otras variables de instancia con diversos parámetros de configuración, por ejemplo: un puntero al objeto *cauce gráfico*, o un valor lógico que indica si la iluminación está activada o no (los iremos viendo en las prácticas).

1.4.2. Funciones para crear y activar VBOs y VAOs

Tal y como se ha visto en teoría, en las prácticas podemos usar las funciones para creación y activación de VBOs y VAOs. Las funciones se declaran en **ig-aux.h**, y se definen (implementan) en **ig-aux.cpp**, tienen estas declaraciones:

```
// VBOs de atributos
GLenum CrearVBOAtrib( GLuint ind_atributo, GLenum tipo_datos,
                      GLint num_vals_tupla, GLsizei num_tuplas,
                      const GLvoid * datos );
GLenum CrearVBOAtrib( unsigned ind_atributo,
                      const std::vector<Tupla3f> & tabla );
GLenum CrearVBOAtrib( unsigned ind_atributo,
                      const std::vector<Tupla2f> & tabla );

// VBOs de índices
GLenum CrearVBOInd( GLenum tipo_indices, GLint num_indices,
                     const void * indices );
GLenum CrearVBOInd( unsigned num_indices, const unsigned * indices );
GLenum CrearVBOInd( const std::vector<unsigned> & indices );
GLenum CrearVBOInd( const std::vector<Tupla3u> & indices );

// VAOs
GLenum CrearVAO();
```

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( ContextoVis & cv ) = 0;
    // visualizar el objeto con OpenGL, visualizando únicamente la geometría.
    virtual void visualizarGeomGL( ContextoVis & cv ) = 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 **leerFijarColVertsCauce** se encarga de leer el color actual en el cauce, fijar un color nuevo (usando el color del objeto, si tiene), y devolver color el anterior del cauce. Esto sirve para fijar el color actual al color del objeto, antes de visualizar el objeto y restaurarlo después, dejando el cauce en el mismo estado.

El método **visualizarGL** tiene un parámetro de tipo referencia a **ContextoVis**, que contendrá todos los parámetros necesarios para la visualización, entre otros: el modo de visualización de polígonos (**modo_visu**) y el modo de envío (**modo_envio**).

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.

En estas prácticas, este método se implementará para mallas indexada (práctica 1) y para nodos del grafo de escena (práctica 3), y en realidad consiste en una versión simplificada de **visualizarGL**.

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 coordenadas 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( ContextoVis & cv ) ;
    virtual void visualizarGeomGL( ContextoVis & cv ) ;
    // ....
};
```

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**)
 - nombres de los VBOs de las distintas tablas (inicialmente a 0).
 - nombre del VAO con la secuencia completa de vértices (inicialmente a 0).
 - nombre del VAO de geometría (inicialmente a 0).
- 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<Tupla3f> vertices ; // coordenadas de las posiciones de los vértices
std::vector<Tupla3u> triangulos; // triángulos (índices)

std::vector<Tupla3f> col_ver ; // colores de los vértices (3 floats por vértice)
std::vector<Tupla3f> nor_ver ; // normales de vértices (3 floats por vértice)
std::vector<Tupla3f> nor_tri ; // normales de triángulos
std::vector<Tupla2f> cc_tt_ver ; // coordenadas de textura de los vértices

// nombres de VAO y VBOs
GLenum nombre_vao = 0 , // nombre del VAO secuencia de vértices (VAO completo)
       nombre_vao_geom = 0 , // nombre del VAO para la geometría (VAO de geometría)
       nombre_vbo_pos = 0 , // VBO de posiciones (>0) una vez creado el VAO
       nombre_vbo_tri = 0 , // nombre del VBO de triángulos, >0
       nombre_vbo_col = 0 , // nombre del VBO de colores, si hay colores, si no 0
       nombre_vbo_nor = 0 , // nombre del VBO de normales, si hay normales, si no 0
       nombre_vbo_cct = 0 ; // nombre del VBO de coords de text., si hay, si no 0
```

El método **visualizarGL**

El método **visualizarGL** debe comprobar si el VAO está creado o no (mirando si su *nombre* es nulo). Si no está creado debe de crearlo e inicializarlo (ocurre en la primera llamada para crear este objeto). Se debe usar la función **CrearVAO**. Después, con el VAO activo, se usa **CrearVBOAtrib** para crear el VBO con la tabla de coordenadas de posiciones de vértices (**vertices**). Finalmente, para cada tabla de atributos no vacía (**col_ver, nor_ver, cct_ver**), hay que llamar a **CrearVBOAtrib** para crear el correspondiente VBO de ese atributo. El nombre del VAO y los nombre de los VBOs quedan en las correspondientes variables de instancia.

En el método **visualizarGL**, una vez que sabemos que el VAO ya está creado con seguridad, se usará la función **glDrawElements** para iniciar la visualización de los triángulos de la malla. Finalmente, se debe usar **glBindVertexArray** para desactivar el VAO (es decir, activar el VAO 0), para que no se quede activado y no interfiera con otras llamadas para visualizar objetos.

El método `visualizarGeomGL`

El método `visualizarGeomGL` se debe encargar de visualizar únicamente la geometría de la malla indexada, sin usar colores, normales, texturas, y sin cambiar el color actual en el cauce (se usa el que haya puesto en el momento de la llamada).

En el caso de la clase `MallaInd`, esto supone visualizar con `glDrawElements`, forzosamente usando algún VAO activo. Podríamos usar el mismo VAO que se usa en `visualizarGL` (lo llamamos el VAO *completo*), pero entonces habría que desactivar temporalmente las tablas de atributos (colores, normales y coordenadas de textura).

En lugar de esto, podemos usar un VAO específico (lo llamamos *VAO de geometría*) que tiene únicamente VBOs para las tablas de posiciones de vértices y la de triángulos (índices), pero no tiene tablas de colores, normales y coordenadas de textura. Para no duplicar la memoria de la GPU usada por las tablas de vértices y triángulos, usaremos en este VAO de geometría los mismos VBOs de vértices y triángulos que ya hemos creado para el VAO completo (OpenGL permite compartir VBOs entre distintos VAOs).

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 (por defecto una sola de ellas), 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 y una cámara actual.

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 la función `VisualizarEscena` de `main.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 incial.

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.

En `main.cpp`, al final de la función de inicialización (`Inicializar`), se crea un vector de escenas (una por cada práctica completada), haciendo `push_back` sobre la variable `escenas`. Durante la ejecución del programa es posible pulsar la tecla P para activar la siguiente escena.

1.4.6. Programación del cauce gráfico. Clase Cauce.

En `main.cpp`, al final de `InicializaOpenGL`, se crean una instancia de la clase `Cauce`, y se guarda un puntero a ella en el cauce actual (en `cv.cauce`). Cada vez que se visualiza la escena, las funciones y métodos de visualización cambian el estado del cauce a través de este puntero.

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 si se usa `cmake`, es necesario volver a generar los archivos de compilación, es decir, vaciar `build` y volver a ejecutar `cmake` y luego `make`). Esto no es necesario si se usa simplemente `make`)

A continuación se detallan las distintas tareas a realizar:

1.5.1. En el archivo `escena.cpp`

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` de `cv`.

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, y no se está en 'modo de selección'), 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.

Constructor de `Escena1`

La clase `Escena1` es una clase derivada de `Escena`, que simplemente añade un método constructor (`Escena1 : Escena`). 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.

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

En este archivo es necesario completar el método `visualizarGL` de la clase `MallaInd`. En concreto, se deben de añadir código para:

1. Asegurarnos que el VAO completo está creado:
 - Si no estaba creado ya, se debe crear e inicializar el VAO completo de la malla, incluyendo todos los VBOs correspondientes. Para ello se puede usar `CrearVAO`, `CrearVBOAtrib` y `CrearVBOInd` (por comodidad, se pueden usar las versiones que aceptan arrays tipo `std::vector`).
 - Si el array estaba creado, simplemente es necesario activarlo con `glBindVertexArray`.
2. Una vez que ya sabemos que el array de vértices está creado, debemos de visualizarlo usando el tipo de primitiva adecuado para una malla de triángulos, y teniendo en cuenta que se trata de una secuencia de vértices indexada (usar `glDrawElements`).
3. Finalmente, es necesario desactivar el VAO (activar el VAO 0).

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. Asegurarnos que el VAO de geometría está creado:
 - Si el VAO de geometría no estaba creado, se debe crear con `CrearVAO` (queda activado). A continuación deben de activarse y habilitarse el VBO de vértices y el VBO de triángulos (índices). Para ello no podemos usar `CrearVBOAtrib` ni `CrearVBOInd`, ya que en realidad basta con reutilizar los VBOs ya creados, cuyos nombres están guardados en `nombre_vbo_pos` y `nombre_vbo_tri`. Usaremos diversas llamadas a las funciones `glBindBuffer`, `glVertexAttribPointer` y `glEnableVertexAttribArray`.
 - Si el VAO de geometría ya está creado, simplemente lo activamos con `glBindVertexArray`.
2. Visualizar la malla con `glDrawElements`.
3. Desactivar el VAO.

1.5.3. 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.

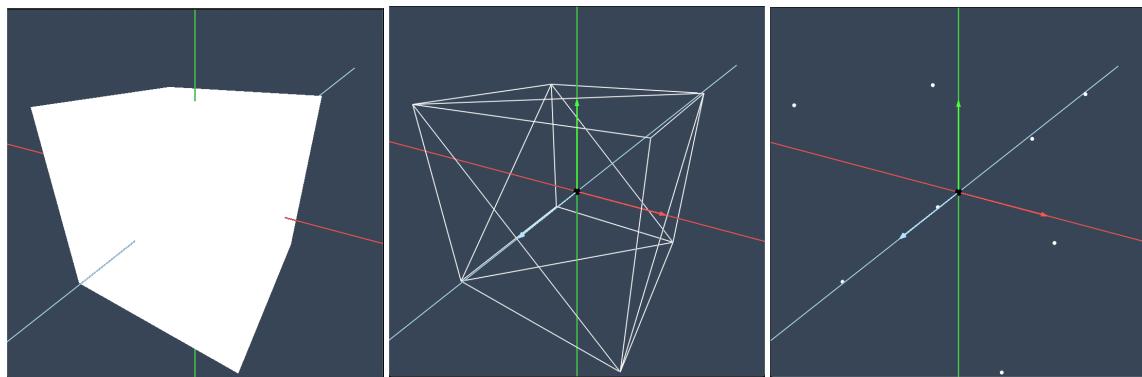


Figura 1.1: Cubo visualizado con los tres modos de visualización de polígonos: relleno (izquierda), aristas (centro) y puntos (derecha)

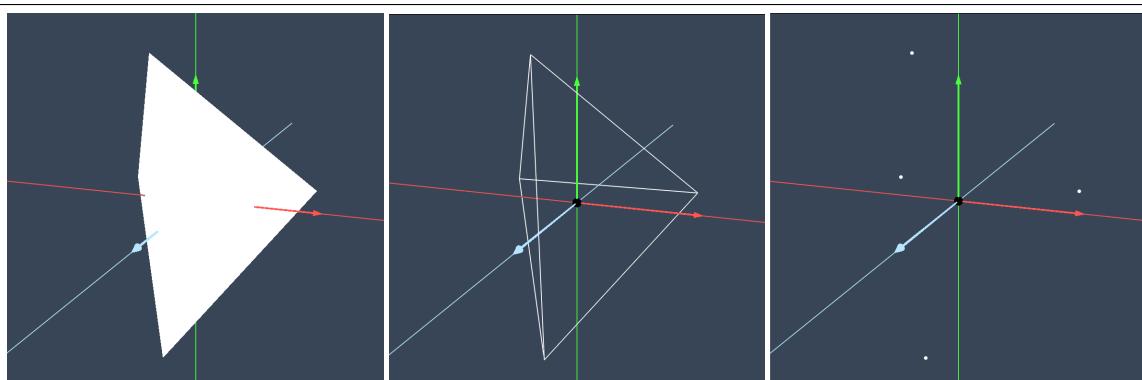


Figura 1.2: Tetraedro visualizado con los tres modos de visualización de polígonos: relleno (izquierda), aristas (centro) y puntos (derecha)

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

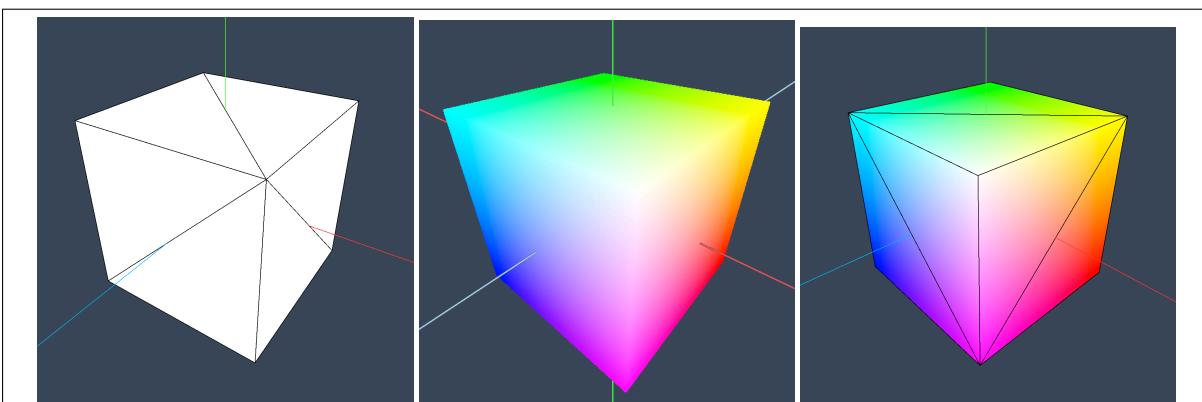


Figura 1.3: Cubo relleno con visualización de aristas (izquierda), cubo con colores (centro), cubo con colores y visualización de aristas (derecha).

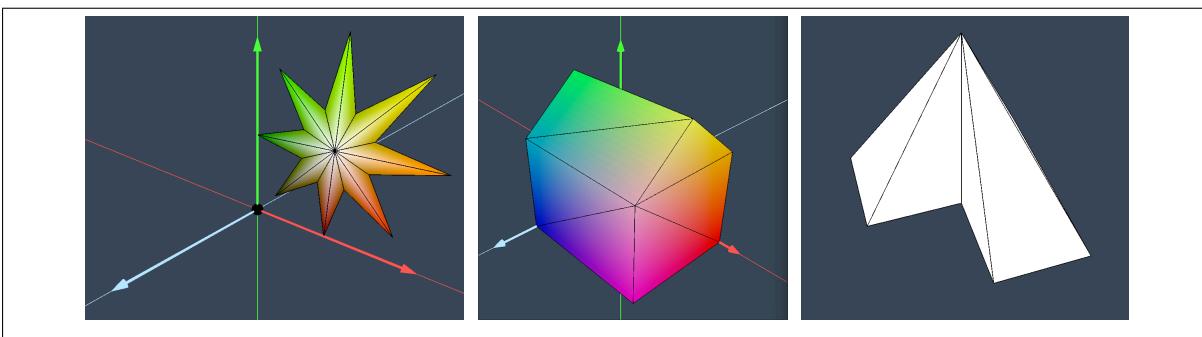


Figura 1.4: Objetos mencionados en los ejercicios adicionales de la práctica 1. A la izquierda el objeto de la clase **EstrellaZ** del ejercicio adicional 1, en el centro el objeto de clase **CasaX** del ejercicio 2, a la derecha el objeto de la clase **MallaPiramideL** del ejercicio 3.

n puntas. Los vértices tienen coordenadas entre 0 y 1 en X y en Y (y todos tienen Z igual a cero). El 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 esta sección se detalla el algoritmo de creación del sólido por revolución (se implementa en un constructor, ver la sección 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,

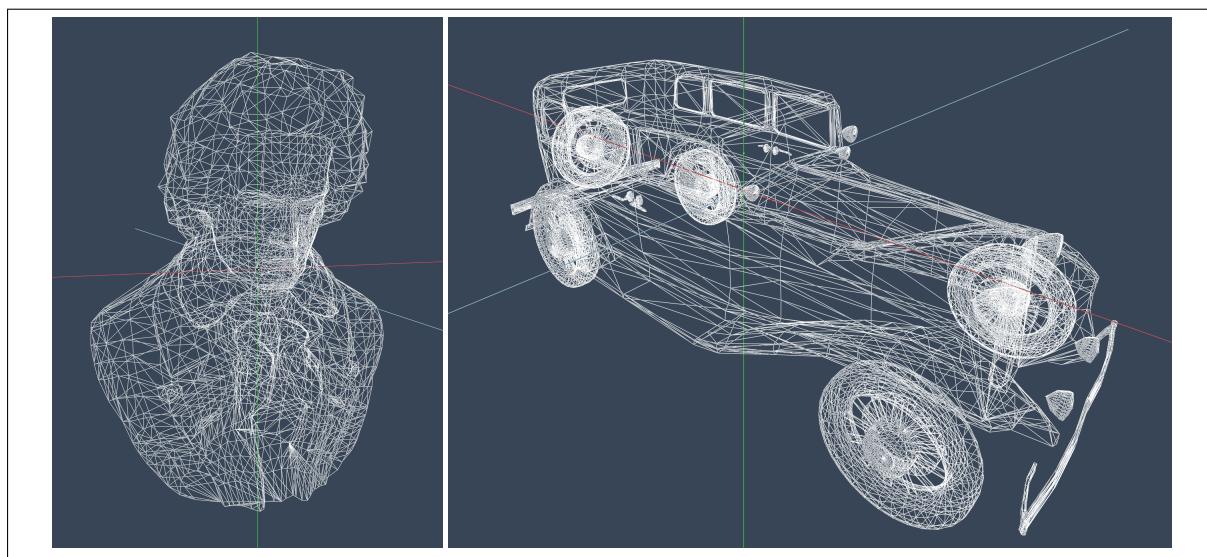


Figura 2.1: Objetos PLY. Vista en modo aristas de los archivos `beethoven.ply` (izqda.) y `big-dodge.ply` (derecha).

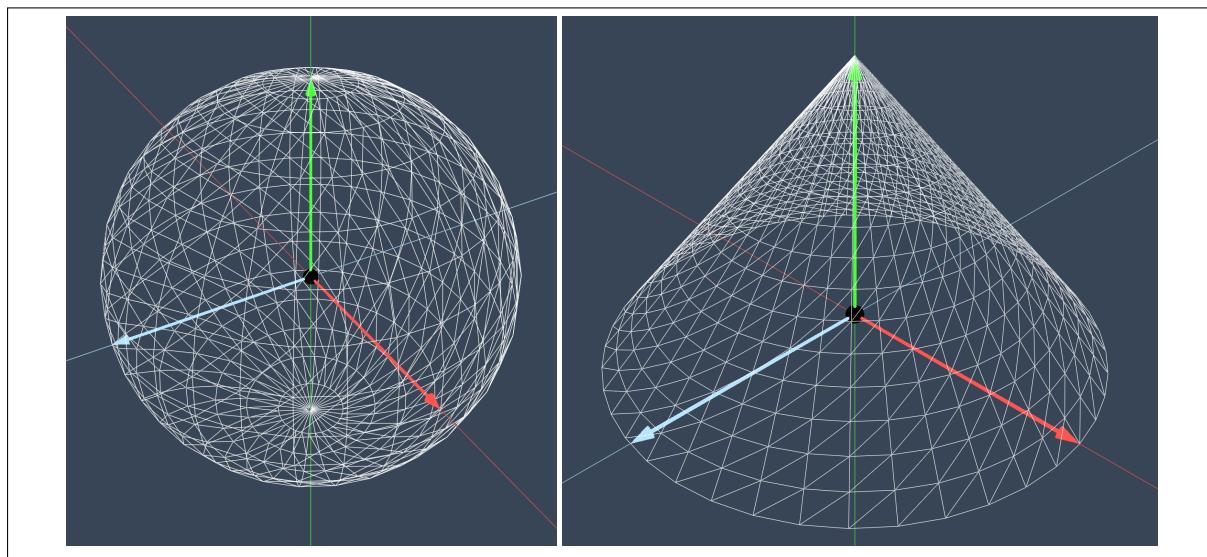


Figura 2.2: Vista de los objetos instancia de las clases `Esfera` (izquierda) y `Cono` (derecha).

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.

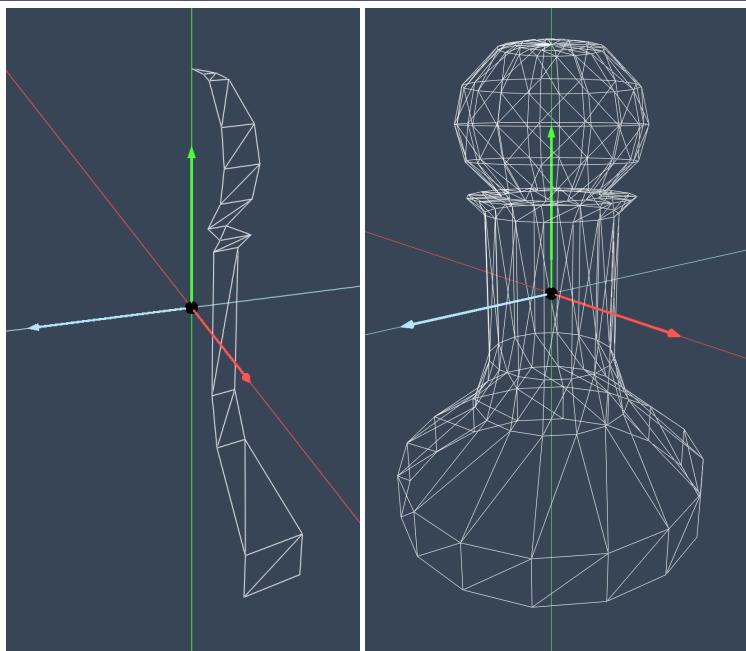


Figura 2.3: Vista de la primera ristra de triángulos entre las dos primeras copias del perfil original (izquierda) y del objeto *peon* obtenido del perfil almacenado en **peon.ply**

```

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. 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

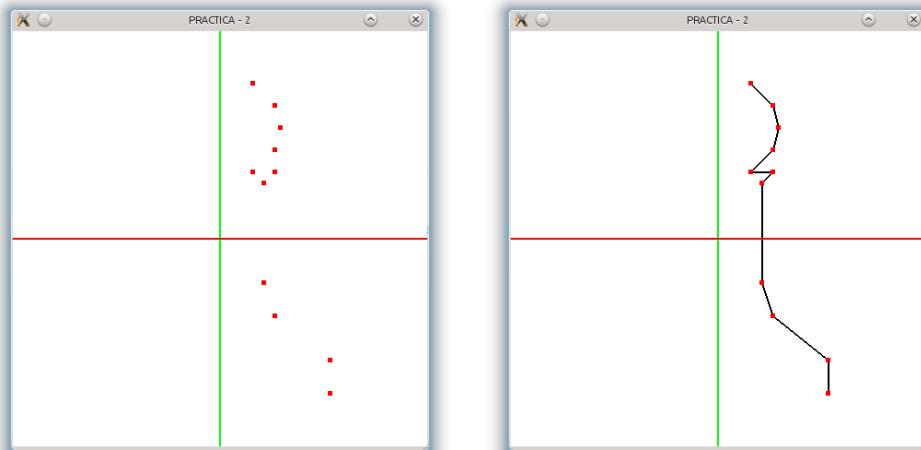


Figura 2.5: Vista del perfil inicial, se trata de los puntos incluidos en el archivo `peon.ply`

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 $k + 1$.

2.4. Estructura del código. Clases.

En el código se creará una clase para la escena de la práctica 2, que contendrá los objetos de revolución. Además se crearán clases para mallas indexadas leídas de un archivo PLY y mallas indexadas creadas por revolución de un perfil.

2.4.1. 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 prácticas 2 (es igual que **Escena1** pero con otro constructor). Esta clase se debe declarar en **escena.h** e implementarse en **escena.cpp**.

Además, en la función **Inicializar** de **main.cpp** será necesario añadir una instancia de **Escena2** al vector de escenas de la aplicación, igual que ya hemos hecho con una instancia de **Escena1**.

2.4.2. 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 **Tupla3f**) y (3) la tabla de triángulos (también de salida, un vector de **Tupla3u**). 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.

2.4.3. 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 implementación 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<Tupla3f> & 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 para crear las tablas de vértices y caras (triángulos), a partir del perfil original.

2.4.4. 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/plys** y si no está en **archivos-alumnos**. También hay un parámetro de salida (un vector de **Tupla3f** con los vértices del perfil). Esta función se debe usar para leer los vértices del archivo **peon.ply**, que está en la carpeta **materiales/plys**.

2.4.5. 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)
    ) ;
} ;
```

2.4.6. 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/>

- Sitio web de John Burkardt en Florida State University (FSU)
-  <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/>

2.5. Tareas

A modo de resumen, es necesario completar o hacer estos métodos o clases:

- Completar el constructor **MallaPLY::MallaPLY** en **malla-ind.cpp**. Se encarga de leer los vértices y caras con la función **LeerPLY**.
- Completar el método **MallaRevول::inicializar** en **malla-revol.cpp**. Se debe de implementar el algoritmo de generación de las tablas de vértices y caras a partir de un perfil.
- Completar el constructor **MallaRevولPLY::MallaRevولPLY** en **malla-revol.cpp**. Se debe de leer el perfil usando la función **LeerVerticesPLY**, y después usar el método **inicializar** para generar la malla.
- Declarar e implementar la clase **Escena2** en **escena.h** y **escena.cpp**, respectivamente. Simplemente añade un constructor que puebla el vector de objetos con varios objetos de revolución.
- Declarar e implementar las clases **Cilindro**, **Cono** y **Esfera** (en **malla-revol.h** las declaraciones y en **malla-revol.cpp** las implementaciones).
- Añadir en la función **Inicializar** la creación de una instancia de **Escena2** (en **main.cpp**), esa instancia se debe añadir al vector de escenas de la aplicación.

2.6. Ejercicios adicionales

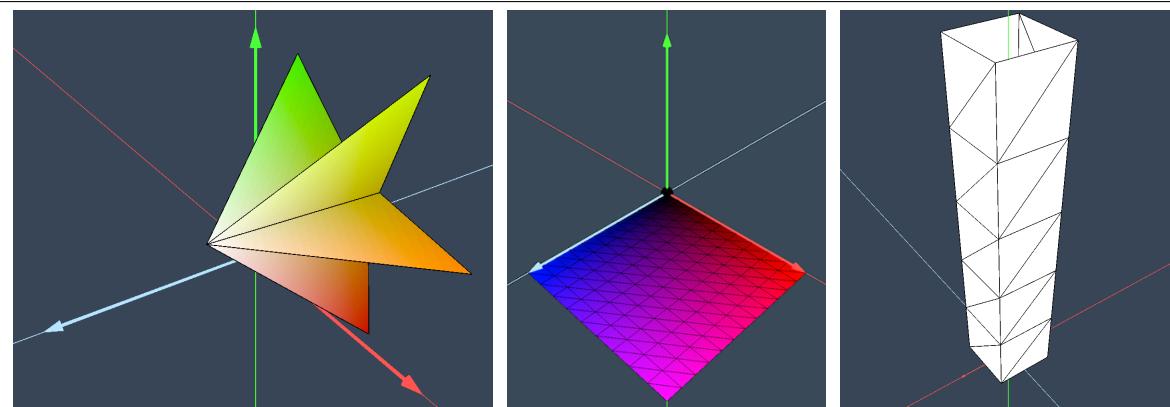


Figura 2.6: Objetos mencionados en los ejercicios adicionales de la práctica 2. A la izquierda el objeto de la clase **PiramideEstrellaZ** del ejercicio adicional 1, en el centro el objeto de clase **RejillaY** del ejercicio 2, a la derecha el objeto de la clase **Torre** del ejercicio 3.

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**.

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 está 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

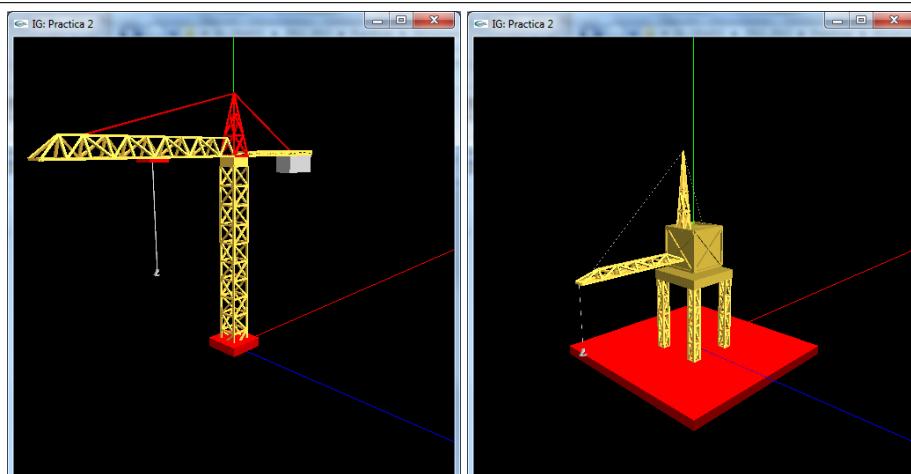


Figura 3.1: Ejemplos del resultado de la práctica 3.

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.

El diseño del modelo se debe materializar en un grafo de escena (tipo PHIGS) según la notación introducida en teoría, en un archivo PDF (que tendrás que entregar cuando entregues la práctica). El grafo 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 la subsección ?? para más detalle sobre los contenidos de este archivo).

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`. También es necesario completar el método `agregar` y el método `leerPtrMatriz`.

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 (en **main.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. Teclas a usar. Gestión de animaciones.

En la función gestora del evento de teclado, en **main.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 implementada 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 **main.cpp**, en concreto desde la función que tiene el bucle principal (**BucleEventosGLFW**). 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`.

3.4. 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 3.4.1

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 raiz y se van añadiendo a la lista de entradas.

La clase *C* y el resto de clases necesarias se definirán en un archivo nuevo `modelo-jer.cpp` y se declarará en `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 la secciones anteriores.

3.4.1. 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.
- Un 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).

3.4.2. 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 la función **Iniciarizar** de **main.cpp** se debe de añadir una instancia de **Escena3** en el vector de escenas (**escenas**).

3.5. Implementación del código de visualización. Colores.

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 activo. 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.

Adicionalmente, será necesario completar el código del método **visualizarGeomGL**. Este método 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). Esto se debe a que el método **visualizarGeomGL** se usa para visualización en modo alambre (únicamente aristas), y por tanto debemos simplemente tener en la geometría, es decir, hacer *push* y *pop* de la matriz de modelado, y recorrer las entradas dibujando los sub-objetos y procesando las entradas de tipo material.

Además de esto, se debe implementar la posibilidad de definir colores específicos de los objetos 3D. Para ello la clase **Objeto3D** tiene ya implementados tres métodos: uno de ellos es **ponerColor(c)**, este método sirve para asignarle un color **c** al objeto (una **Tupla3f**), otro es **tieneColor()**, que devuelve **true** si el objeto tiene asignado un color, y finalmente **leerColor**, que devuelve la tupla con el color actual, siempre que el objeto tenga un color (en otro caso aborta). Inicialmente un objeto no tiene asignado color alguno.

Cuando se visualiza un objeto que no tiene asignado color, se usará el color actual fijado en el cauce antes de la llamada a `visualizarGL`. Sin embargo, si el objeto tiene asociado un color (si `tieneColor` devuelve `true`), entonces se debe: (1) guardar el color actual en el cauce gráfico, (2) fijar el color actual en el cauce usando el color del objeto (3) visualizar y (4) restaurar el color actual en el cauce, fijándolo al valor anterior a la llamada a `visualizarGL`.

Para implementar fácilmente el comportamiento descrito en los dos párrafos anteriores, se debe usar (al inicio de `visualizarGL`) el método `leerFijarColVertsCauce` (de la clase `Objeto3D`, se le pasa `cv` como parámetro) sobre el nodo que estamos dibujando. Esta método ya está implementado en el archivo `objeto3d.cpp`. El método devuelve el color actual del cauce en `cv` (es una `Tupla4f`), y, si el nodo tiene un color definido, fija el color del cauce con el color del nodo. Al final de `visualizarGL` se debe restaurar el color previo leído al inicio.

Para implementar este comportamiento debemos de extender el código del método `visualizarGL` de las clases `MallaInd` y de `NodoGrafoEscena`, teniendo en cuenta lo descrito en el párrafo anterior. En principio no definiremos colores de los objetos de tipo `MallaInd`, pero sí es conveniente definirle colores a algunos nodos del grafo de escena. Esto se hace invocando a `ponerColor` en los constructores de esos nodos. La utilidad de esto es poder diferenciar visualmente en el modelo distintas partes o sub-árboles, usando colores distintos.

Lo dicho sobre los colores solo es efectivo para objetos de tipo `MallaInd` si ese objeto no tiene definida una tabla de colores de vértices (si la tabla `col_ver` está vacía). Si una malla tiene una tabla de colores de vértices no vacía, entonces se usará al visualizar, independientemente de cual sea el color actual en el cauce. Esto es independiente de lo descrito en los párrafos anteriores.

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, 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 del 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 sobreescriben en el grafo de escena.

Hay que tener en cuenta que **actualizarEstadoParametro** debe usar punteros a las matrices que debe sobreescribir. 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 **Matriz4f ***) 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. 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.

3.8. Ejercicios adicionales

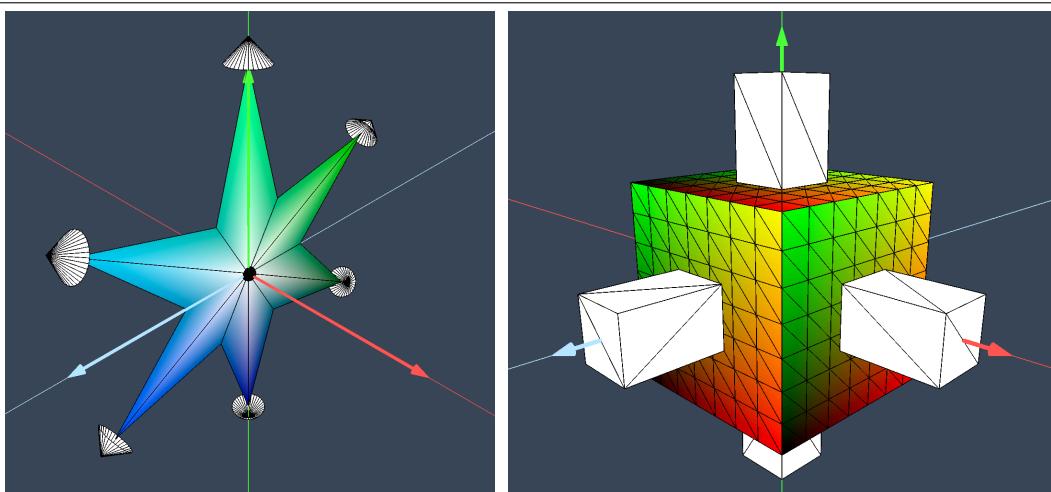


Figura 3.2: Objetos mencionados en los ejercicios adicionales de la práctica 3. A la izquierda el objeto de la clase **GrafoEstrellaX** del ejercicio adicional 1, a la derecha el objeto de la clase **GrafoCubos** del ejercicio 2 (los colores que tengan tus objetos pueden ser distintos a los que aparecen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

3.8.1. 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**.

3.8.2. 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, usaremos estructuras de datos para representar modelos de fuentes de luz, e incorporaremos a la estructura de datos que representa el grafo de escena modelos de materiales y texturas. En concreto:

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 PLY, ver sección 4.5.3, y mallas de revolución, ver sección 4.5.2). Esta implementación se hará en los constructores de las clases derivadas de **MallaInd**, en algunos casos invocando métodos específicos.
2. Se añadirá código a la clase **MallaInd** y la clase **Escena** para visualizar las normales de vértices al activar dicha funcionalidad con la tecla **N**. Ver sección 4.4.7.
3. Se completará el código de las clases **Textura** (ver sección 4.4.2) y **Material** (sección 4.4.3), 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 4.4.1). 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 4.4.6.
5. Se añadirá código (en el archivo **main.cpp**) para procesar las teclas que permiten modificar interactivamente los vectores de dirección de las fuentes de luz de la escena activa, en concreto, se procesan las teclas que se pulsen a la vez que la tecla **L** (ver sección 4.3).
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 4.5.1).
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 ma-

teriales y textura (la escena con la lata y los peones). La geometría y aspecto de esta escena se describe en este guión (ver sección 4.4.9), 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 (en el archivo **escena.cpp**). Ver sección 4.4.8.
9. Se añadirán materiales y texturas al grafo de escena diseñado e implementado en la práctica 3, y se creará un archivo PDF con el grafo de escena nuevo (con materiales) y datos sobre los materiales y texturas usadas. Ver sección 4.4.10.

4.3. 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 **main.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 después invocar la función **ProcesaTeclaFuenteLuz**, cuyo código ya está completo en el archivo **materiales-luces.cpp**.

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).

4.4. 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.

4.4.1. 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.

4.4.2. 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 (cauce)**, que se encarga de activar una textura (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 4.5.2).

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 textu-

ras, 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.

4.4.3. 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 (cauce)` 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 alto.

En la plantilla, al final de `Material::activar` ya hay una sentencia que registra el material activado dentro del contexto de visualización (se escribe `cv.material_act`).

4.4.4. 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 la estructura `cv` que se pasa como parámetro incluye un valor lógico (`iluminacion`) que valdrá `true` si la iluminación está activada, y `false` si está desactivada. Debemos de escribir código para que cuando dicho valor sea `true` se den los siguientes pasos:

1. Al inicio del método: debemos de guardar puntero al material activo (que está registrado en en la variable `material_act` dentro de `cv`).
2. Durante el bucle que recorre las entradas: si una entrada es de tipo material, es necesario activarlo y actualizar `cv.material_act`.
3. Al final del método: debemos de restaurar el material activo al inicio (si es distinto del actual) (ver las transparencias de teoría).

4.4.5. 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

Tupla2f). 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_trí**), 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 **Tupla3f**. La primera de ellas es una tabla auxiliar en 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 de

- los métodos **calcularNormalesTriangulos** y **calcularNormales** de **MallaInd**, para calcular las normales de los vértices promediando las normales de las caras.
- los constructores de las clases **Cubo** y **Tetraedro** definidas en la práctica 1, para calcular las normales al final de los mismos, invocando **calcularNormales**.
- el constructor de **MallaPLY**, para invocar a **calcularNormales** al final.
- el método **inicializar** de la clase **MallaRevol**.

Los algoritmos necesarios para estos cálculos dependen del tipo de objeto y se detallan en la sección 4.5.

4.4.6. Añadidos a la clase Escena

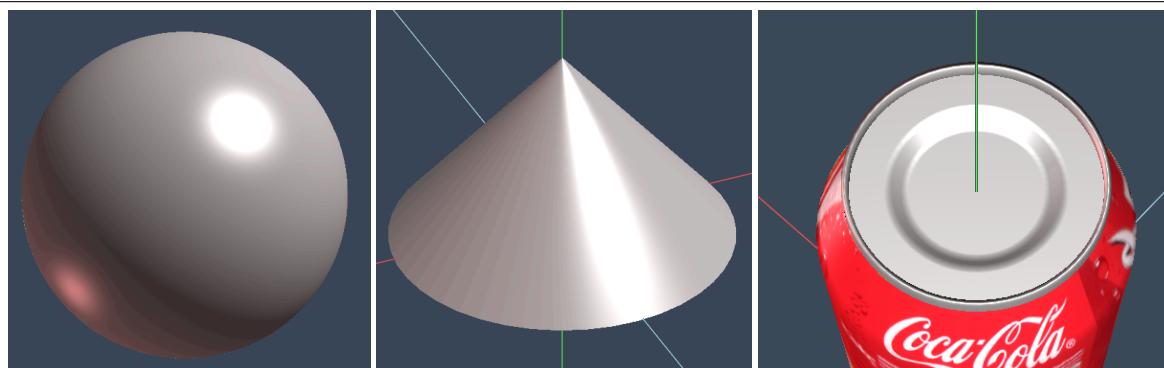


Figura 4.1: Objetos de la clase **Esfera** (izquierda) y **Cono** (centro), ambos de la práctica 2, vistos con iluminación. A la derecha, vemos la tapa del objeto **Lata**, esta tapa es un objeto de revolución.

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).

4.4.7. Visualización de normales de vértices

Se debe añadir código para visualizar las normales en las clases **MallaInd** (visualiza las normales de una malla indexada) y en **Escena** (visualiza las normales del objeto actual de la escena). Este código se activará pulsando la letra N.

La visualización de las normales requiere crear un VAO específico para cada malla indexada que tenga asociada una tabla de normales de vértices. El VAO contiene una secuencia de vértices, cada dos vértices forman los extremos de un segmento paralelo a la normal y con extremo en el correspondiente vértice de la malla. Como consecuencia, se debe visualizar el VAO con la primitiva **GL_LINES**.

Las instrucciones detalladas acerca de como incorporar este código se encuentran aquí

☞ <http://sl.ugr.es/visunorm>

4.4.8. 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**.

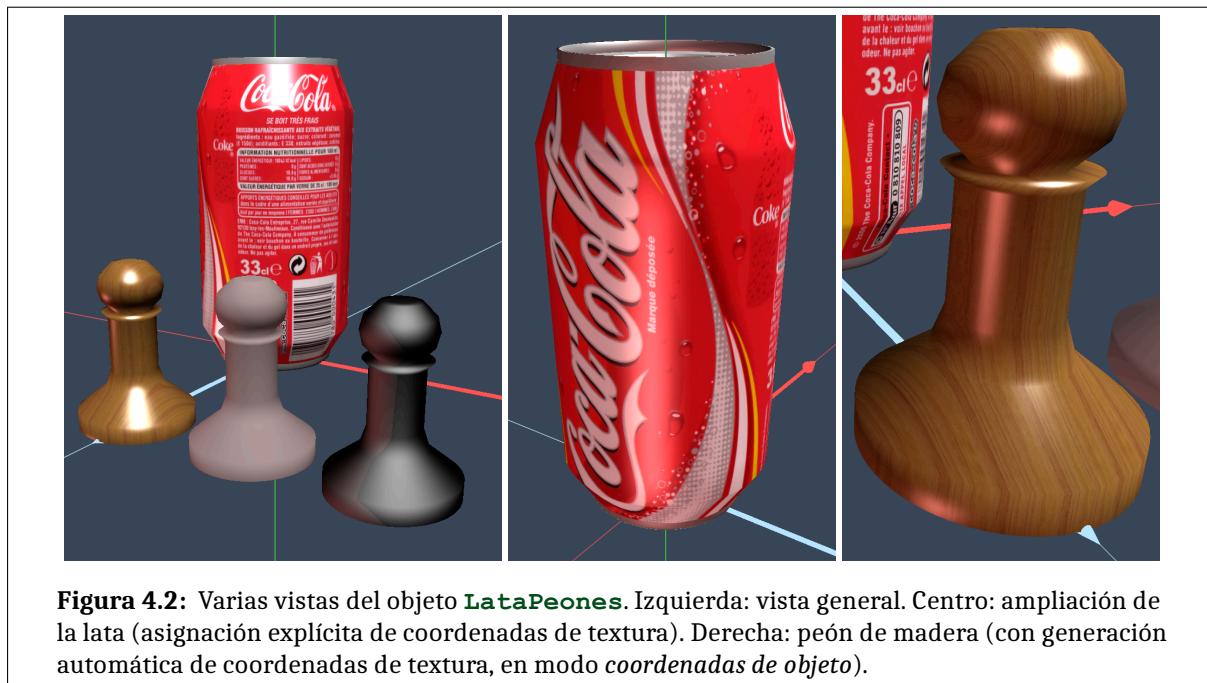
En la función **Iniciar** de **main.cpp** se debe de añadir una instancia de **Escena4** en el vector de escenas (escenas).

Esta escena contendrá un objeto de tipo **LataPeones** (ver la sección 4.4.9 y la figura 4.2, izquierda), y otro objeto de tipo **NodoCubo24** (ver la sección 4.5.1 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.

4.4.9. 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 **srcs** 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).

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.

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).

4.4.10. 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 el grafo de escena, como se describe aquí abajo.

Se debe producir un archivo PDF con documentación sobre los materiales en el grafo de escena, en concreto el archivo PDF contendrá:

- En la primera página, nombre de la asignatura y el curso académico, nombre y apellidos del autor, su titulación.
- 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.
- El grafo de escena tipo PHIGS incluyendo las entradas de tipo material. Para cada entrada material, 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.

4.5. 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 extender el código de los constructores de esas clases, de forma que ahora incluyan la creación de las mencionadas tablas.

4.5.1. Clases `Cubo24` y `NodoCubo24`

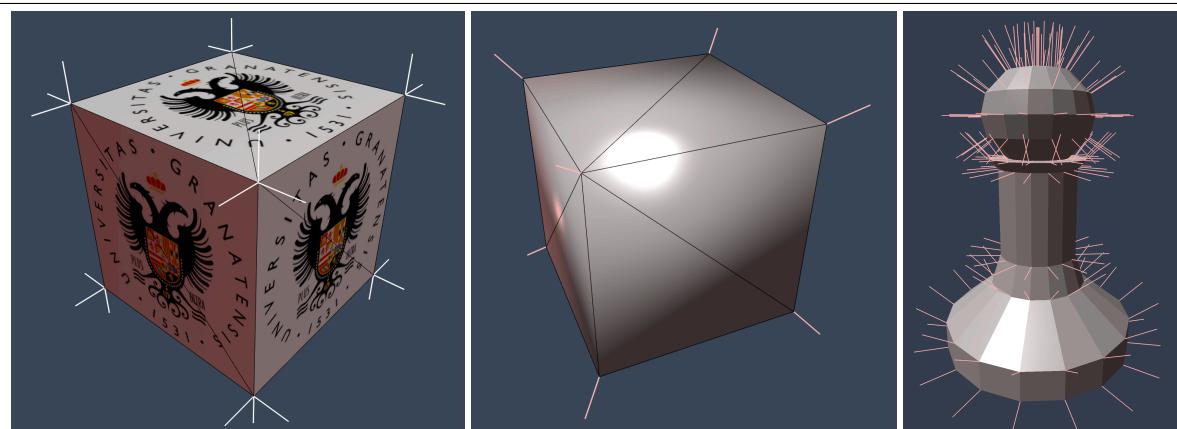


Figura 4.4: Varios objetos con sus normales de vértices también visualizadas. Izquierda: objeto `Cubo24` (cubo con 24 vértices, ver sección 4.5.1), con las normales y las texturas correctamente asignadas. Centro: objeto `Cubo` (8 vértices) con las normales promediadas (sección 4.5.3), se observa la iluminación errónea. Derecha: objeto de revolución de la práctica 2 (clase `Peon`, ver sección 4.5.2), con normales de triángulo (no interpoladas).

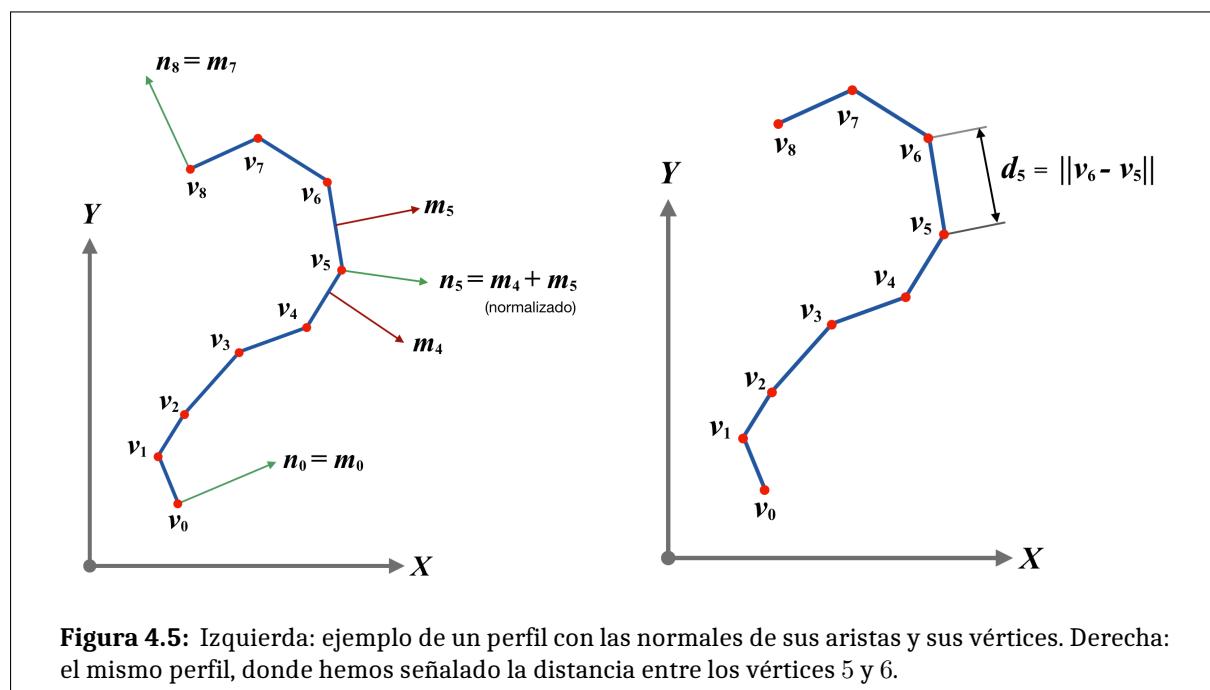
Queremos visualizar un objeto de tipo cubo, posiblemente asignando una imagen de textura a cada una de sus 6 caras, y con la iluminación correcta usando las normales a dichas caras.

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 nuestro 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 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 4.4.8).

4.5.2. 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.

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 $\mathbf{v}_{i+1} - \mathbf{v}_i$.

- Calculamos las normales de los vértices $(\mathbf{n}_0, \dots, \mathbf{n}_{n-1})$. Para ello hacemos $\mathbf{v}_0 = \mathbf{m}_0$, también hacemos $\mathbf{v}_{n-1} = \mathbf{m}_{n-2}$. Para el resto de vértices, calculamos \mathbf{n}_i como $\mathbf{m}_{i-1} + \mathbf{m}_i$ (normalizado).

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).

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 $d_i = \|\mathbf{v}_{i+1} - \mathbf{v}_i\|$ 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

$$t_i = \left(\sum_{j=0}^{i-1} d_j \right) / \left(\sum_{j=0}^{n-2} d_j \right)$$

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 guión 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 4.4.2.

4.5.3. 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 asignandole 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.

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`, de tipo vector de `Tupla3f`, 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 $\mathbf{a} = \mathbf{q} - \mathbf{p}$ y $\mathbf{b} = \mathbf{r} - \mathbf{p}$. El vector \mathbf{m}_c , perpendicular a la cara, se obtiene como el producto vectorial de las aristas, es decir, hacemos: $\mathbf{m}_c = \mathbf{a} \times \mathbf{b}$. Finalmente, el vector normal \mathbf{n}_c (de longitud unidad) se obtiene normalizando \mathbf{m}_c , esto es: $\mathbf{n}_c = \mathbf{m}_c / \|\mathbf{m}_c\|$.

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 $\mathbf{m}_c = \mathbf{b} \times \mathbf{a}$ en lugar del originalmente descrito, ya que el producto vectorial es anticomutativo.

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.

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

```

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**

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.

4.6. Ejercicios adicionales

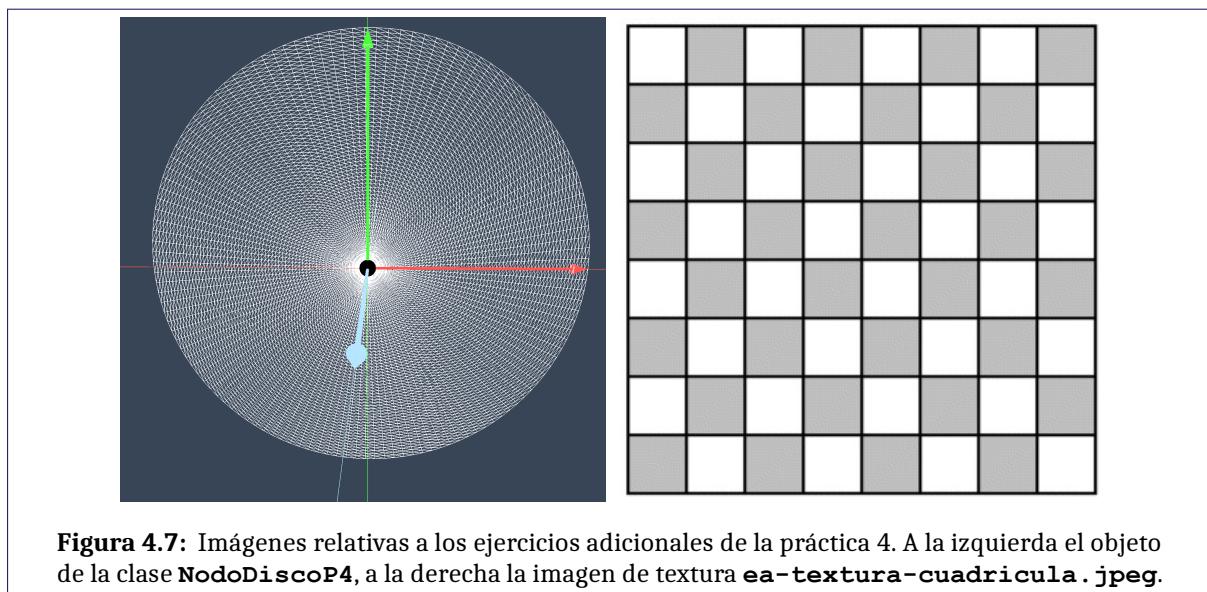


Figura 4.7: Imágenes relativas a los ejercicios adicionales de la práctica 4. A la izquierda el objeto de la clase **NodoDiscoP4**, a la derecha la imagen de textura **ea-textura-cuadricula.jpeg**.

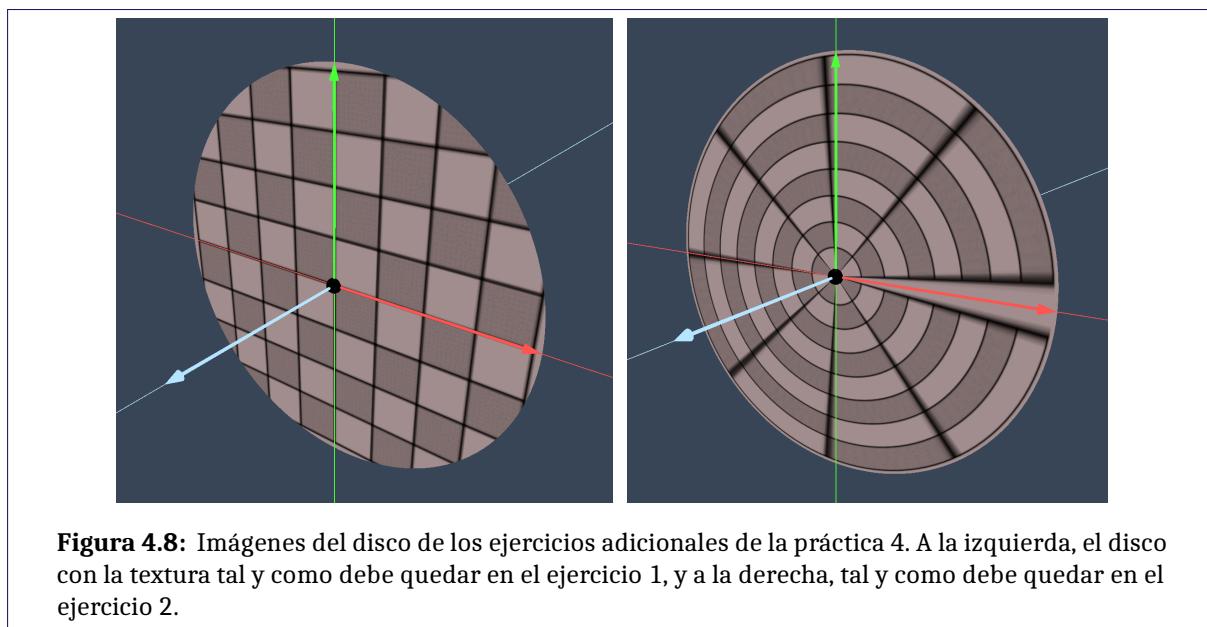


Figura 4.8: Imágenes del disco de los ejercicios adicionales de la práctica 4. A la izquierda, el disco con la textura tal y como debe quedar en el ejercicio 1, y a la derecha, tal y como debe quedar en el ejercicio 2.

4.6.1. 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).

4.6.2. 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).

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.

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.

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, la cámara actual pasará a modo examinar, y el centro del objeto se proyectará en el centro de la imagen.

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 al grafo de escena los identificadores de selección asignados a los nodos e información adicional de dichos identificadores (ver sección 5.7).

5.3. Teclas e interacción con el ratón.

En la función `FGE_PulsarLevantarTecla` del archivo `main.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 `main.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.

5.4. Escena de la práctica 5.

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 seguiremos 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:



Figura 5.1: Escena de la práctica 5 con los 6 objetos que se indican en la sección 5.4

- *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`).
- *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 depurar el código de selección. Las texturas necesarias ya se encuentran disponibles en la carpeta `materiales/imgs`.

También debemos de hacer lo mismo (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 jerárquicos.

5.5. 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.cpp`. 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.

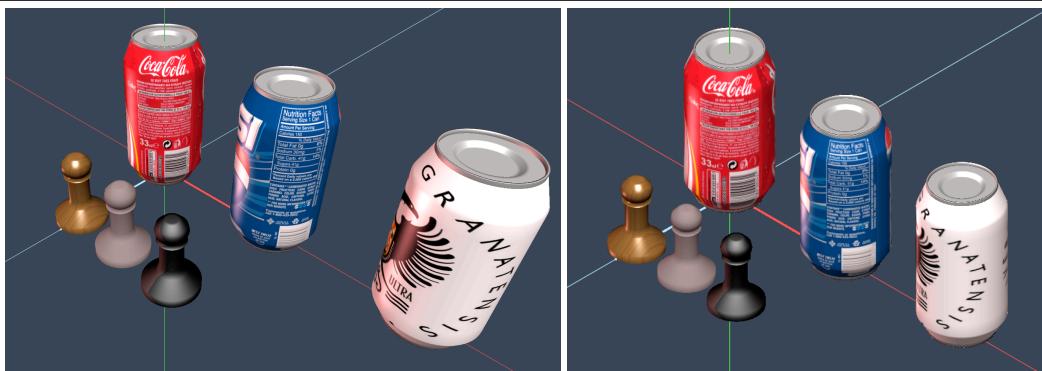


Figura 5.2: Escena de la práctica 5, vista con una cámara con proyección perspectiva (izquierda) y con una cámara con proyección ortográfica (derecha).

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

5.5.2. 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 `main.cpp`. Cuando se mueve con el botón derecho pulsado (modo arrastrar), se llama al méto-

do `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.

5.5.3. 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, 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:

- **`org_polares`**: coordenadas esféricas del origen de la cámara, relativas al punto de atención (la tupla 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 `Esfericas` 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 a la tupla 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.

5.6. 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. Cuando esto ocurre, la cámara activa pasará a modo examinar (si no lo estaba ya). El punto de atención de dicha cámara quedará fijado a un punto central al objeto, y a partir de entonces los movimientos de la cámara se harán rotando alrededor de dicho punto, según se describe en las transparencias de teoría, hasta que se commute el modo de la cámara activa o bien se cambie la cámara activa.

Cada vez que se pulsa el botón izquierdo del ratón se ejecuta la FGE de nombre `FGE_PulsarLevantarBotonRat` ya implementada en `main.cpp`. En esa FGE se invoca la función `Seleccion` en el archivo `seleccion.cpp`. Dicha función recibe como parámetros: las coordenadas del pixel donde se ha pulsado, un puntero a la escena actual, y el contexto de visualización actual. Se debe completar el código de esta función, dando los pasos que se indican en la sub-sección 5.6.2.

Será necesario extender el código del método `visualizarGL` de las clases `MallaIndy` y `NodoGrafoEscena` (ver la sección 5.6.1). En el archivo `seleccion.cpp` será necesario completar el código de las funciones `FijarColVertsIdent` y `LeerIdentEnPixel`, según se ha detallado en las transpa-

rencias de teoría.

5.6.1. Visualización de mallas y nodos en modo selección

El método `visualizarGL` de `MallaInd` y de `NodoGrafoEscena` debe estar adaptado para funcionar correctamente en modo selección (es decir, cuando `cv.modo_seleccion` vale `true`). Para asegurarnos de esto debemos de comprobar que en esos dos métodos hay llamadas (al inicio) al método `leerFijarColVertsCauce (cv)` de la clase `Objeto3D`. Este método ya está implementado. Al ejecutarse, cuando `cv.modo_seleccion` es `true`, usa el identificador del objeto (en lugar de su color) para cambiar el color actual de cauce gráfico (es decir, codifica el identificador como color actual). Al igual que antes, hay que tener la precaución de restaurar el color previo siempre antes del acabar `visualizarGL`, pero después de visualizar.

5.6.2. La función Selección

En esta función se hará la visualización en modo selección, y si se ha seleccionado un objeto se apunta la cámara actual hacia el centro de dicho objeto. Se deben implementar estos pasos:

1. Crear (si es necesario) el *framebuffer object* (FBO) de selección. Se usa la variable `fbo` declarada en `seleccion.cpp`. Si dicha variable es `nullptr`, se creará el objeto en memoria dinámica, usando el tamaño de la ventana que hay en el contexto de visualización.
2. Crear un objeto `ContextoVis` nuevo, en ese objeto:
 - a) Activar modo selección (poner `modo_seleccion` a `true`), desactivar iluminación (poner `iluminacion` a `false`), fijar el modo de visualización (`modo_visu`) a modo relleno.
 - b) Fijar el cauce (`cauce_act`) al mismo que había.
 - c) Fijar el tamaño de la ventana.
3. En el cauce, fijar el color actual al identificador 0 (usar `FijarColVertsIdent`), que es el identificador inicial de los objetos (y que indica que los objetos no son seleccionables).
4. Activar el FBO (usar el método `activar` de la clase `Framebuffer`), activar el cauce y fijar la matriz del viewport (`glviewport`). Configurar cauce: ponerlo en modo sólido relleno, sin iluminación ni texturas. Limpiar el FBO usando el 0 como color de fondo.
5. Activar la cámara actual en el cauce (la cámara actual se debe leer de la escena con `camaraActual`).
6. Visualizar el objeto raíz actual (se debe leer de la escena con el método `objetoActual`).
7. Leer el identificador en el pixel donde se ha hecho click (usar `LeerIdentEnPixel`).
8. Desactivar el FBO de selección (usar el método `desactivar`).
9. Si el identificador es el 0, en ese pixel no hay ningún objeto seleccionable, no habría que hacer nada más.
10. Si el identificador es > 0 , buscar ese identificador en el objeto actual de la escena. Si se encuentra, poner la cámara actual mirando hacia el centro de dicho objeto, usando el método `mirarHacia` de la cámara (ver la sección 5.6.3). Es importante imprimir en el terminal si se ha seleccionado un objeto o no, cual es su identificador, y el nombre del objeto seleccionado (esto permite depurar el proceso de selección de forma independiente a la gestión de cámaras).

5.6.3. 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,
Tupla3f & centro_wc );
```

La función (virtual), devuelve **true** si el objeto se ha encontrado, o **false** en otro caso. Los parámetros 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.
- **mmodelado**: 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.

5.7. Documento con grafo de escena incluyendo identificadores de selección

Se debe producir un archivo PDF con el grafo de escena, partiendo del documento de la práctica 4, pero extendiéndolo con la información relativa a los distintos identificadores de selección que se hayan usado. En concreto el archivo PDF contendrá:

- En la primera página, nombre de la asignatura y el curso académico, nombre y apellidos del autor, su titulación.
- Una (o varias) captura de pantalla del modelo.
- 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**)

5.8. Ejercicios adicionales

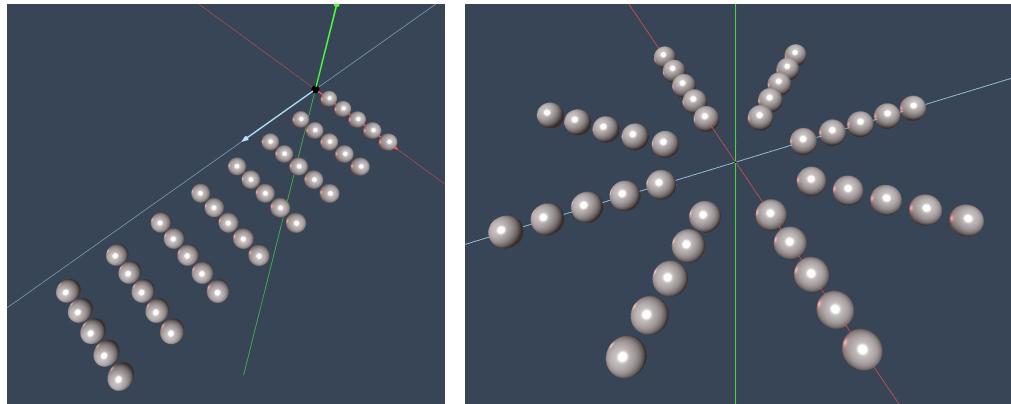


Figura 5.3: Imágenes de los grafos de escena mencionados en el ejercicio adicional 1 (izquierda) y 2 (derecha) de la práctica 5.

5.8.1. Ejercicio 1

(convocatoria ordinaria curso 20-21)

Escribe una clase llamada **GrafoEsferasP5**, 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 (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++ )
        {
            MallaInd * esfera = new Esfera() ;
            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. Modifica el código de la función **Seleccion** de forma que si se pincha con el ratón en una de las esferas, se imprima (además de lo que ya se imprime) un mensaje de esta forma:

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). El resto del código de selección funciona igual (en particular, la cámara se debe centrar en el centro de la esfera concreta seleccionada).

Recuerda que únicamente debes añadir código al constructor **GrafoEsferasP5** y a la función de selección, en ningún caso debes eliminar código.

5.8.2. Ejercicio 2

(convocatoria ordinaria curso 20-21)

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++ )
        {
            MallaInd * esfera = new Esfera() ;
            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. Modifica el código de la función **Seleccion** de forma que si se pincha con el ratón en una de las esferas, se imprima (además de lo que ya se imprime)

un mensaje de esta forma:

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 (también comenzando en 1). El resto del código de selección funciona igual (en particular, la cámara se debe centrar en el centro de la esfera concreta seleccionada). Recuerda que únicamente debes añadir código al constructor **GrafoEsferasP5_2** y a la función de selección, en ningún caso debes eliminar código.