

# **Prácticas de Informática Gráfica**

Grado en Informática y Matemáticas. Curso 2019-20.



**UNIVERSIDAD  
DE GRANADA**

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



# Índice general.

<b>Índice.</b>	<b>3</b>
<b>0. Prerrequisitos, materiales y compilación</b>	<b>7</b>
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 . . . . .	9
0.2. Materiales y compilación . . . . .	9
0.2.1. Materiales para las prácticas. Entregas. . . . .	10
0.2.2. Compilación . . . . .	11
0.3. Clases auxiliares . . . . .	13
0.3.1. Clases para tuplas de valores enteros o reales con 2,3 o 4 valores . . . . .	13
<b>1. Visualización de modelos simples</b>	<b>15</b>
1.1. Objetivos . . . . .	15
1.2. Desarrollo . . . . .	15
1.3. Teclas a usar. Interacción. . . . .	15
1.4. Estructura del código: Clases para parámetros, objetos y escenas. . . . .	16
1.4.1. Contexto y modos de visualización y de envío . . . . .	16
1.4.2. Clase abstracta para objetos gráficos 3D ( <b>Objeto3D</b> ) . . . . .	17
1.4.3. Clase para mallas indexadas ( <b>MallaInd</b> ) . . . . .	18
1.4.4. La clase <b>Escena</b> . . . . .	19
1.4.5. Programación del cauce gráfico. Clase <b>Cauce</b> y derivadas. . . . .	19
1.4.6. Clases para los objetos de la práctica 1 . . . . .	19
1.5. Tareas . . . . .	20
1.5.1. En el archivo <b>escena.cpp</b> . . . . .	20
1.5.2. En el archivo <b>malla-ind.cpp</b> . . . . .	20
1.5.3. Clases nuevas para otros tipos de mallas indexadas . . . . .	21

<b>2. Modelos PLY y Poligonales</b>	<b>25</b>
2.1. Objetivos . . . . .	25
2.2. Desarrollo . . . . .	25
2.3. Algoritmo para la creación de malla por revolución . . . . .	27
2.4. Estructura del código. Clases. . . . .	29
2.4.1. La clase <b>Escena2</b> . . . . .	29
2.4.2. Clase <b>MallaPLY</b> : mallas creadas a partir de un archivo PLY . . . . .	29
2.4.3. Clase <b>MallaRevol</b> : mallas obtenidas por revolución de un perfil. . . . .	30
2.4.4. Clase <b>MallaRevolPLY</b> : revolución de un parfil leído en un PLY . . . . .	30
2.4.5. Clase: <b>Cilindro, Cono, Esfera</b> . . . . .	31
2.4.6. Archivos PLY disponibles. . . . .	31
2.5. Tareas . . . . .	32
<b>3. Modelos jerárquicos</b>	<b>33</b>
3.1. Objetivos . . . . .	33
3.2. Desarrollo . . . . .	33
3.3. Teclas a usar. Gestión de animaciones. . . . .	34
3.4. Diseño e implementación de un grafo de escena parametrizado original . . . . .	35
3.4.1. La clase <b>Escena3</b> . . . . .	35
3.5. Implementación del código de visualización. Colores. . . . .	35
3.6. Implementación de parámetros y animaciones . . . . .	36
3.6.1. Definición de clases para objetos parametrizados . . . . .	37
3.6.2. Definición de nodos del grafo de escena parametrizados . . . . .	38
3.7. Algunos ejemplos de modelos jerárquicos . . . . .	39
<b>4. Materiales, fuentes de luz y texturas</b>	<b>41</b>
4.1. Objetivos . . . . .	41
4.2. Desarrollo . . . . .	41
4.3. Teclas a usar . . . . .	42
4.4. Clases y métodos a añadir o completar. . . . .	42
4.4.1. Las clases <b>FuentesLuz</b> y <b>ColFuentesLuz</b> . . . . .	43

4.4.2. Clase <b>Textura</b> . . . . .	43
4.4.3. Clase <b>Material</b> . . . . .	44
4.4.4. Añadidos a la clase <b>NodoGrafoEscena</b> . . . . .	44
4.4.5. Añadidos a la clase <b>MallaIndy</b> y derivadas . . . . .	44
4.4.6. Añadidos a la clase <b>Escena</b> . . . . .	45
4.4.7. Clase <b>Escena4</b> . . . . .	46
4.4.8. Clase <b>LataPeones</b> . . . . .	46
4.4.9. Materiales y textura en el grafo de la práctica 3 . . . . .	47
4.5. Cálculo de tablas de normales y coordenadas de textura . . . . .	48
4.5.1. Clases <b>Cubo24</b> y <b>NodoCubo24</b> . . . . .	48
4.5.2. Clase <b>MallaRevol</b> . . . . .	49
4.5.3. Clases <b>Cubo</b> , <b>Tetraedro</b> y <b>MallaPLY</b> . . . . .	50



# 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 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).

#### OpenGL

Respecto de la librería OpenGL, es necesario tener instalado algún driver de la tarjeta gráfica disponible 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, para verificar la tarjeta instalada y ver los drivers recomendados y/o posibles para dicha tarjeta, se puede usar esta orden:

```
sudo ubuntu-drivers devices
```

Lo más fácil es instalar automáticamente el driver más apropiado, se puede usar la orden:

```
sudo ubuntu-drivers autoinstall
```

#### 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 (initialmente 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

#### CMake

Para compilar algunas librerías (y el propio proyecto) es necesario disponer de la orden **cmake**. Si no la tienes instalada, puedes descargar e instalar el archivo **.dmg** para macOS que se encuentra en esta página web:

 <https://cmake.org/download/>

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

```
cmake --version
```

#### 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).

## Librería GLFW

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 bibliotecas dinámicas con nombres que comienzan con `libglfw` y con extensión `.dylib`, )

### Librería JPEG

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 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

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` para compilar, así como la librería OpenGL preinstalada.

Respecto a las librerías GLEW, GLU, glfw, y de lectura de JPEGs, los archivos correspondientes se proporcionan entre los recursos entregados como material (en la carpeta `recursos/packages-win`), por tanto no hay que descargar nada.

## 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:

- **recursos**: esta carpeta contiene archivos de código fuente, imágenes, modelos 3D (archivos **.ply**) y otros, que se deben usar tal cual se entregan, sin que se deban modificar. Tiene estas sub-carpetas:
  - **src**: archivos de código fuente C++ (**.cpp**)
  - **include**: archivos de cabecera C++ (**.h**)
  - **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**)
  - **make**: varios archivos con información de configuración para el proceso de compilación.
  - **packages-win**: archivos de cabecera, **.lib** y **.dll** correspondientes a las librerías (GLEW, GLFW, JPEG) para el s.o. Windows.
- **plantilla**: esta carpeta contiene código fuente que debe ser completado por el alumno, así como otros archivos (imágenes o modelos 3D, por ejemplo), tiene estas sub-carpetas:
  - **src**: archivos de código fuente C++ (**.cpp**), para realizar las prácticas se debe completar y extender el código de esta carpeta.
  - **include**: archivos de cabeceras C++ (**.h**).
  - **plys**: archivos con modelos 3D en formato PLY (**.ply**), se proporciona vacío. Aquí el alumno puede situar sus propios archivos **.ply** que no se proporcionen en la carpeta **recursos/ply**.
  - **imgs**: archivos de imágenes para texturas en formato JPEG (**.jpg**), se proporciona vacío. Aquí el alumno puede situar sus propios archivos **.jpg** que no se proporcionen en la carpeta **recursos/imgs**.
  - **make-unix**: archivos para compilar la aplicación usando **make** en sistemas operativos . basados en Unix (es decir: Linux y macOS). Contiene dos sub-carpetas (**bin** y **objs**), inicialmente vacías.
  - **cmake-unix**: archivos para compilar la aplicación usando **cmake** en sistemas operativos basados en Unix. Contiene la subcarpeta **build** inicialmente vacía.
  - **cmake-win**: archivos para compilar la aplicación usando **cmake** en el sistema operativo Windows. También contiene la subcarpeta **build** inicialmente vacía.

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

Tras descargar los materiales, podemos copiar la carpeta **plantilla** a otra (hermana suya) de nombre, por ejemplo, **trabajo**. A esta carpeta la llamaremos en adelante la *carpeta de trabajo*. En la carpeta de trabajo será donde el alumno haga modificaciones, mientras mantiene como referencia la carpeta **plantilla**. El proceso de compilación y la ejecución requieren de la presencia de la carpeta **recursos** sin modificar.

La entregas de prácticas consisten en subir los archivos fuente, imágenes y modelos PLYS de la carpeta de la carpeta de trabajo (subcarpetas **src**, **include**, **plys** e **imgs**). 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), ni se debe subir ningún archivo que esté en la carpeta **recursos**.

## 0.2.2. Compilación

Se describe aquí el proceso de compilación en la línea de órdenes para macOS, Linux o Windows. En los tres casos también se pueden usar entornos de desarrollo para crear la aplicación, lo cual no se cubre en este documento. En particular, en macOS o Windows podemos usar las aplicaciones XCode o Visual Studio, respectivamente.

Los sistemas operativos macOS y Linux están ambos basados en Unix y tienen una *shell* y herramientas de desarrollo parecidas, por tanto, el proceso de compilación es también similar. Se puede hacer tanto usando **make** como **cmake**.

Usar **cmake** es preferible a **make** en lo que respecta a detectar las dependencias que hay entre los archivos **.cpp** y los archivos de cabecera que los primeros incluyen directa o indirectamente. Cuando se usa **make**, sera necesario limpiar y recomilar todo cuando modifiquemos algún archivo de cabecera.

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. Esto no es necesario cuando se usa **make**, ya que este programa directamente compila los fuentes.

En el sistema operativo Windows solo está disponible la opción de usar **cmake**.

### Sistemas operativos Linux y macOS, con CMake

Se deben de dar estos pasos:

- Ejecutar un terminal macOS o Linux, hacer **cd** a la carpeta de trabajo.
- Hacer **cd** a la subcarpeta **cmake-unix/build**, la cual está vacía inicialmente. En esa carpeta, escribir una sola vez esta orden:

```
cmake ..
```

Si no hay errores, esto generará en la carpeta **build** diversas sub-carpetas y archivos, entre ellos el archivo **Makefile**. Si se quiere volver a generar todo, bastaría con vaciar la carpeta **build** y de nuevo hacer **cmake ..**

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

```
make
```

Si no hay errores, esta orden debe de generar el archivo ejecutable **pracs\_ig\_exe** en la carpeta **build**. 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**

- Cada vez que se quiera ejecutar el programa hay que hacer **cd** a la carpeta de trabajo, y escribir:

```
cmake-unix/build/pracs_ig_exe
```

## Sistemas operativos Linux y macOS, con *make*

La orden **make** permite compilar y ejecutar el programa. Se deben de dar estos pasos:

- Ejecutar un terminal macOS o Linux, hacer **cd** a la carpeta de trabajo.
- Hacer **cd** a la carpeta **make-unix**, a su vez dentro de la carpeta de trabajo.
- Cada vez que se quiera compilar y ejecutar, en la carpeta **make-unix**, hay que escribir esta orden:

```
make
```

si no hay errores, esta orden debe de generar el archivo ejecutable **pracs\_ig\_exe** en la subcarpeta **bin** (junto con todos los archivos **.o**, en la sub-carpeta **objs**), después se ejecuta el programa.

- Para eliminar todos los archivos generados al compilar se puede ejecutar:

```
make clean
```

y luego de nuevo **make**. Es conveniente hacer esto cada vez que se modifique algún archivo **.h**, ya que con esta configuración no se captan las dependencias que hay entre cada archivo **.cpp** y todos los archivos **.h** que los primeros incluyen directa o indirectamente.

## Sistema operativo Windows con *CMake*

Asumimos que se dispone de una instalación actualizada de Windows 10, en un equipo de 64 bits, y que vamos a generar un ejecutable tipo **Debug** de Windows. Se deben de dar estos pasos:

- Ejecutar un terminal de desarrolladores para 64 bits, es una aplicación que se denomina **x64 native tools command prompt for VS 2019**. Este programa se puede encontrar en el menu inicio, en la carpeta correspondiente **Visual Studio 2019**, o bien buscando la palabra **command** y en la lista que se despliega seleccionar dicha aplicación. Esta aplicación es similar al **command prompt** de Windows (la terminal), pero con acceso a diversos programas de desarrollo de aplicaciones de Visual Studio, entre otros el propio compilador (**c1**) o **cmake**.
- Hacer **cd** hasta la carpeta **build** dentro de **cmake-win**,
- Escribir una sola vez este comando

```
cmake -DCMAKE_GENERATOR_PLATFORM=x64 ..
```

esto genera en **cmake-win/build** diversas carpetas y archivos (.vcxproj, .sln y otros). En ocasiones querremos limpiar todos los archivos de compilación para volver a compilar todo desde cero. Esto se hace vaciando la subcarpeta **build**, para ello, en **cmake-win**, hacemos:

```
rmdir /s /q build
mkdir build
```

y después de nuevo repetir, en la carpeta **build**, la orden citada arriba (**cmake -D....**).

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

```
cmake --build .
```

(si no hay errores, esta orden debe de generar el archivo **pracs\_ig\_exe.exe** en la subcarpeta **cmake-win\build\Debug**, junto con las DLLs necesarias y algún otro archivo)

- Cada vez que se quiera ejecutar hacer **cd** a la carpeta de trabajo, escribir:

```
cmake-win\build\Debug\pracs_ig_exe.exe
```

## 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 **tuplasg.h** se declaran varias clases para tuplas de valores numéricos (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**,**Tupla3i**, 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
Tupla3i t3 ; // tuplas de tres valores tipo int
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 arr3i[3] = { 1, 2, 3 } ;

// declaraciones e inicializaciones de tuplas
Tupla3f a( 1.0, 2.0, 3.0 ), b, c(arr3f) ; // b indeterminado
Tupla3i d( 1, 2, 3 ), e, f(arr3i) ; // 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 mallas indexadas de forma eficiente, tanto en modo inmediato y en modo diferido.

## 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 cada uno de los tres **modos de envío**:

- Modo inmediato con `glBegin/glVertex/glEnd`
- Modo inmediato con `glDrawElements`.
- Modo diferido con `glDrawElements` y usando un *Vertex Array Object* (VAO).

Independientemente de los modos de envío, 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).

## 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 b/B**: cambia el modo de envío actual de los vértices, que puede ser: modo inmediato con begin/end, modo inmediato con un VAO, o modo diferido con un VAO (pasa al siguiente, o del último al primero).
- **tecla f/F**: activa o desactiva el modo de sombreado plano.
- **tecla s/S**: cambia el cauce gráfico actual (de cauce programable a fijo o al revés).

- **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 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 cámara 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 cámara 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 cámara 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: Clases para parámetros, objetos y escenas.

El archivo **main.cpp** 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 índices 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.hpp** (dentro de **include**, en el directorio de trabajo) 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
} ;

```

Además, la clase **ContextoVis** contiene otra variable (**modo\_envio**), que es del tipo enumerado **ModosEnvio**, y que codifica el *modo de envío de polígonos* actual (puede ser: inmediato con **begin/end**, inmediato con **DrawElements**, y diferido con un VAO). El tipo se declara con:

```

enum class ModosEnvio
{
    inmediato_begin_end, inmediato_drawelements, diferido_vao,
    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 cauce gráfico activo, o un valor lógico que indica si la iluminación está activada o no (los iremos viendo en las prácticas).

#### 1.4.2. 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 **includes/objeto3d.h** 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 ;
    // 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**.

#### 1.4.3. 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, que contiene ternas de valores enteros, cada una de esas ternas tiene los tres índices de las coordenadas de los tres vértices del 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 ) ;
    // ...
};
```

La clase incluye:

- Como variables de instancia privadas:
    - tabla de coordenadas de vértices (**vertices**)
    - tablas de atributos (**col\_ver**, **nor\_ver**, **cct\_ver**)
    - tabla de triángulos (es la tabla de índices, agrupados de tres en tres).
    - identificadores de VBO de las distintas tablas (initialmente 0)
    - identificadores de VAO para modo inmediato (**id\_vao\_mi**) y para modo diferido (**id\_vao\_md**).
  - Como método público virtual, el método **visualizarGL**, que visualiza la malla teniendo en cuenta el parámetro modo, y usando las dos tablas descritas arriba. Este método (en función del modo de envío actual que está en **cv**) debe llamar a alguno de los métodos privados de visualización, específicos de cada modo de envío. Son los siguientes:
    - **visualizarGL\_MI\_BE**: inmediato con **begin/end**.
    - **visualizarGL\_MI\_DE**: inmediato con **DrawElements**.
    - **visualizarGL\_MD\_VAO**: diferido con VAO (y cn **DrawElements**).
  - Los métodos privados para creación y activación de VAOs y registro de tablas:
    - **crearActivarVAO\_MD**: crear (si no estaba creado) y activar VAO del modo diferido.
    - **crearVBOs**: crear todos los VBOs para las tablas (para modo diferido)
    - **registrarTablas\_MI**: registrar formato y localización de tablas en modo inmediato (cada vez antes de dibujar)
    - **registrarTablas\_MD**: registrar formato y localización de tablas en modo diferido (una vez antes de la primera visualización)
- estos métodos se invocan desde las correspondientes funciones de visualización.

Además, en el archivo `src/malla-ind.cpp` se incluyen tres funciones auxiliares (no miembro) para la creación de VBOs y registro de tablas:

- `CrearVBO`: crea un VBO
- `LocForTabla`: especifica localización y formato de una tabla
- `RegistrarTabla`: registra una tabla (especifica loc. y formato, y habilita/deshabilita).

#### 1.4.4. 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.

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.

#### 1.4.5. Programación del cauce gráfico. Clase Cauce y derivadas.

En `main.cpp`, al final de `InicializaOpenGL`, se crean dos objetos (dos instancias) de clases derivadas de `Cauce`. Punteros a esos dos objetos se guardan en el vector `cauces`. Uno de los objetos es de tipo `CauceProgramable` y otro `CauceFijo`. Ambos presentan el mismo interfaz público, pero incluyen distintas implementaciones de los métodos de dicho interfaz público.

En `main.cpp` se almacena un índice de cauce activo (un índice en el array de cauces). Durante la visualización, es posible pulsar la tecla S para conmutar el índice de cauce actual. Cada vez que se visualiza la escena, se usa el cauce actual activo.

#### 1.4.6. 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 fija el modo de sombreado, para ello es necesario llamar al método **fixarModoSombrPlano** del cauce activo (variable **cauce**) usando el valor lógico **sombr\_plano** almacenado en el el contexto de visualización (**cv**).

En ese mismo método, después de fijar el modo de sombreado, se debe también fijar el modo de visualización de polígonos (rellenos, aristas o puntos), usando la función **glPolygonMode** y la variable **modo\_visu** de **cv**.

Finalmente, antes del final de método se debe invocar el método **visualizarGL** de la clase **Objeto3D**, usando el puntero **objeto** al objeto actual de la escena.

#### Constructor de **Escena1**

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

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

### 1.5.2. En el archivo **malla-ind.cpp**

En este archivo se deben de completar las distintas funciones y métodos que permiten visualizar una malla indexada, usando cualquiera de los tres modos de envío (inmediato con begin/end, in-

mediato con un VAO, y diferido con un VAO).

Todo esto debe realizarse según se ha descrito en teoría para visualizar secuencias arbitrarias de vértices (ya que en este caso estamos visualizando la tabla de coordenadas de vértices y las tablas de atributos, usando indexación) excepto en los siguientes puntos:

- La secuencia de vértices siempre es indexada.
- La secuencia de vértices codifica una serie de triángulos (cada tres índices consecutivos codifican un triángulo).
- En lugar de una tabla de enteros con los índices, se tiene la tabla de caras o tabla de triángulos (**triangulos**), con tantos elementos como triángulos. Esta tabla tiene en cada entrada una tupla con tres enteros (**Tupla3i**). Por tanto, el número de índices es tres veces el número de entradas.

## Funciones

Estas funciones se encuentran ya declaradas pero vacías, pendientes de completar:

- Función **CrearVBO**: crea un VBO y transfiere datos de una tabla.
- Función **LocForTabla**: especifica la localización y formato de una tabla.
- Función **RegistrarTabla**: habilita o deshabilita el uso de una tabla. Si habilita, especifica la localización y formato.

## Métodos de la clase **MallaInd**

En primer lugar hay que completar los métodos relacionados con registro de tablas, creación de VBOs y VAOs, son los siguientes:

- Método **crearVBOs**: crea todos los VBOs de todas las tablas.
- Método **registrarTablas\_MI**: registra todas las tablas en modo inmediato.
- Método **registrarTablas\_MD**: registra todas las tablas en modo diferido.
- Método **crearActivarVAO\_MI**: crear (si no lo estaba) y activar el VAO del modo inmediato.
- Método **crearActivarVAO\_MD**: crear (si no lo estaba) y activar el VAO del modo diferido.

Después es necesario completar los métodos para visualizar la malla:

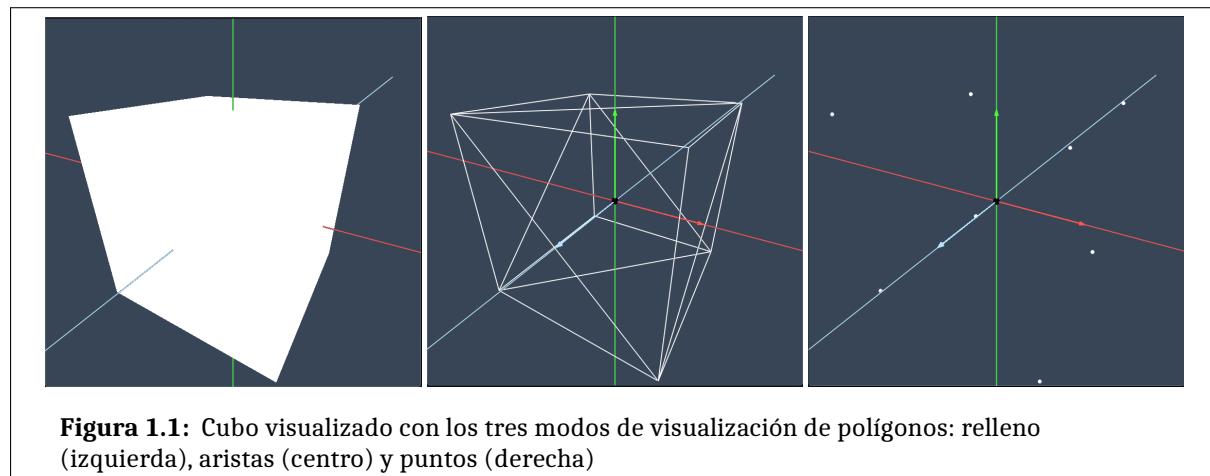
- Método **visualizarGL\_MI\_BE**: visualizar en modo inmediato con **begin/end**
- Método **visualizarGL\_MI\_VAO**: visualizar en modo inmediato con un VAO.
- Método **visualizarGL\_MD\_VAO**: visualizar en modo diferido con un VAO.
- Método **visualizarGL**: visualizar el objeto según el modo especificado en **modo\_visu**, dentro de **cv**, llamando a alguno de los métodos anteriores. En este método ya hay una llamada a **leerFijarColVertsCauce**, para fijar el color antes de visualizar y restaurarlo después.

### 1.5.3. Clases nuevas para otros tipos de mallas indexadas

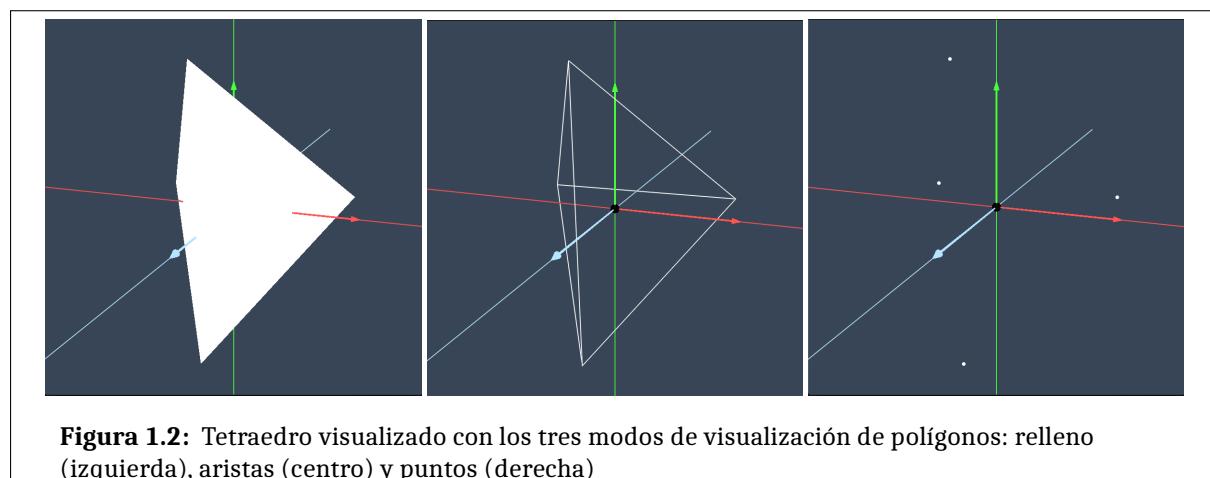
En **malla-ind.cpp** ya está implementada la clase **Cubo**, derivada de **MallaInd**. El constructor (sin parámetros) crea un cubo de 6 caras, lado 2 unidades y centro en el origen (se extiende entre -1 y +1 en los tres ejes). Esta constructor no crea las tablas de atributos de vértices, que quedan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

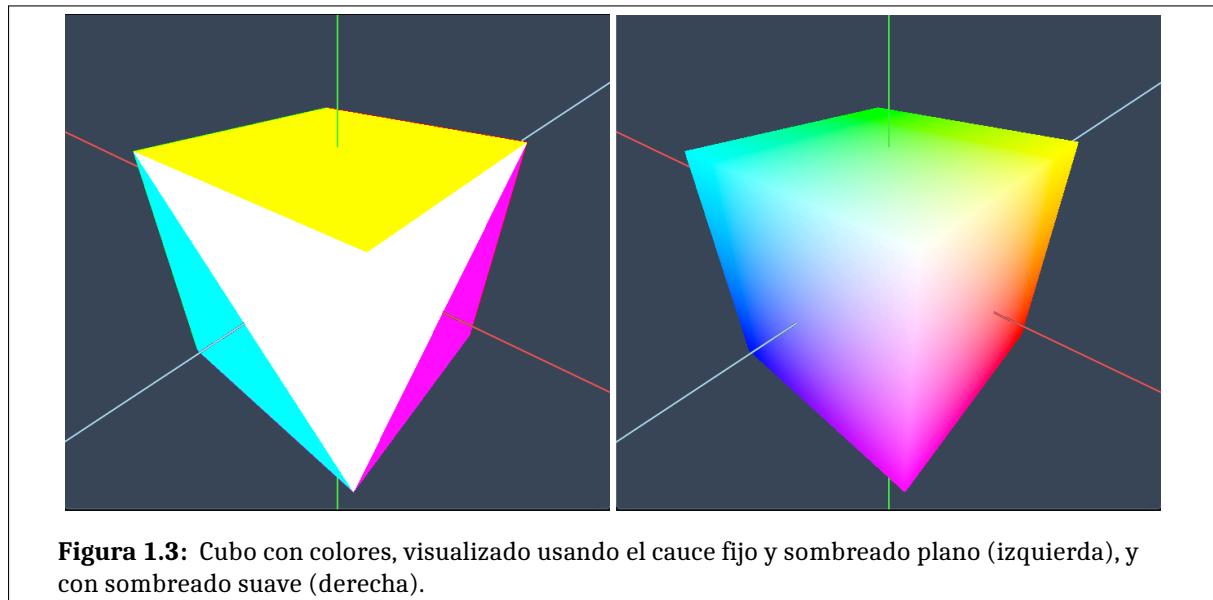
Adicionalmente se creará una clase llamada **CuboColores**, también derivada de **MallaInd**. En el constructor se inicializan las tablas de vértices y los triángulos igual que en la clase **Cubo**, pero además se definirá la tabla de colores de vértices. Cada color será una terna RGB, de forma que la 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)



**Figura 1.3:** Cubo con colores, visualizado usando el cauce fijo y sombreado plano (izquierda), y con sombreado suave (derecha).



## 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:

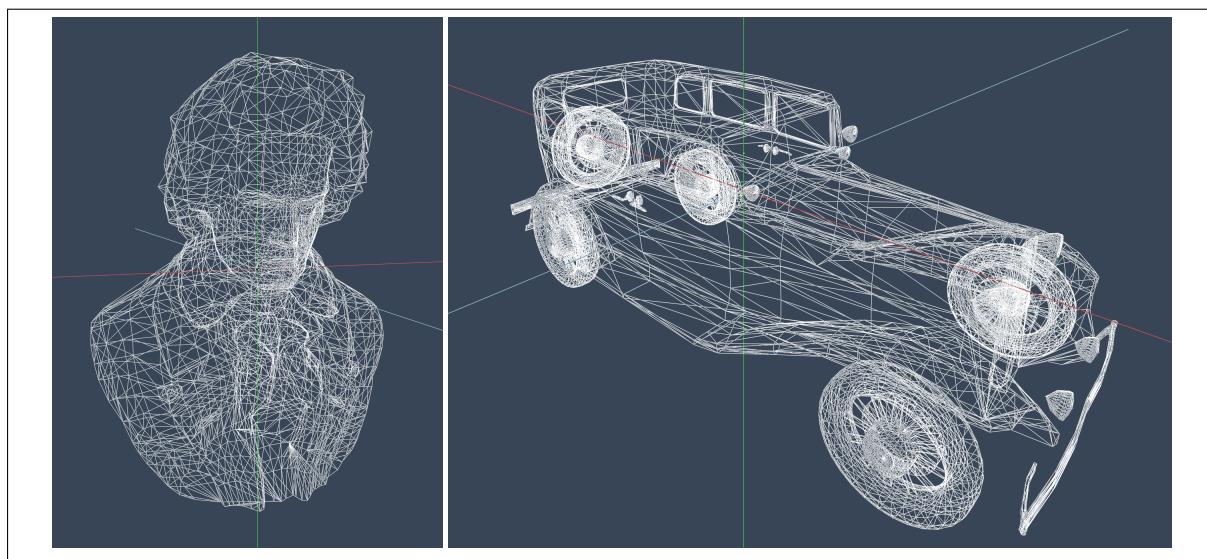
-  <http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html>

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.

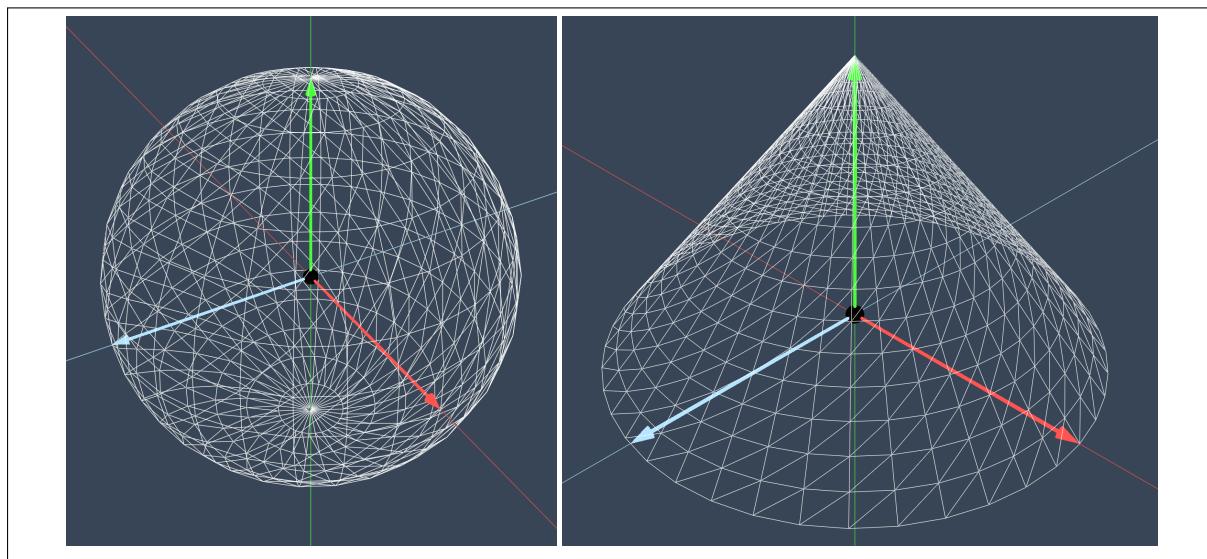
En segundo lugar, se desarrollará un algoritmo para la generación procedural de una malla obtenida por revolución de un perfil alrededor del eje Y. Dicho algoritmo tiene como parámetros de entrada la secuencia de vértices que define dicho perfil, y el número de copias del mismo que servirán para crear 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).

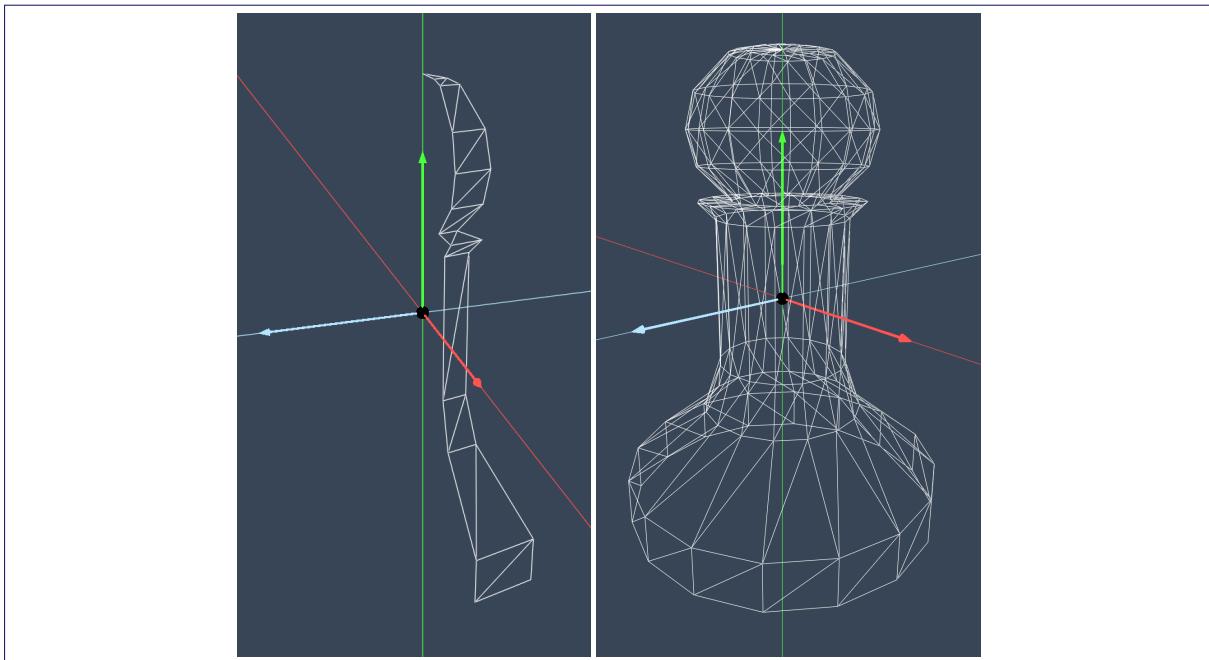


**Figura 2.2:** Vista de los objetos instancia de las clases `Esfera` (izquierda) y `Cono` (derecha).

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**

```

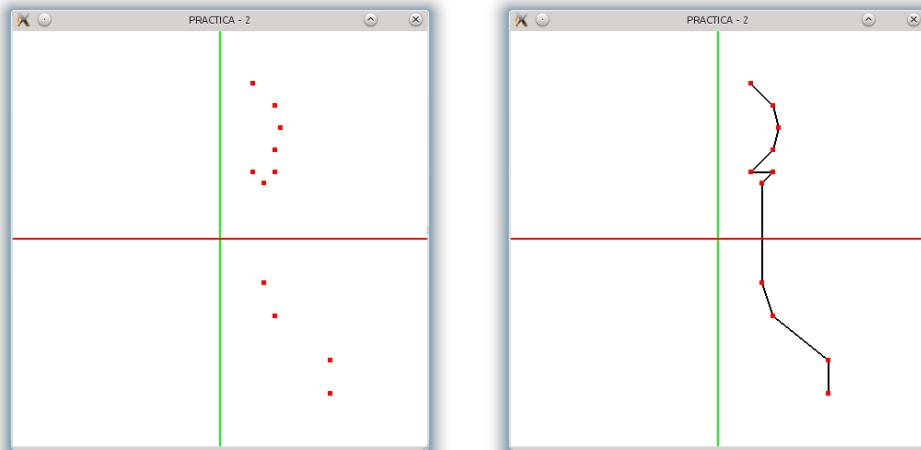
ply
format ascii 1.0
element vertex 11
property float32 x
property float32 y
property float32 z
element face 1
property list uchar uint vertex_indices
end_header
1.0 -1.4 0.0
1.0 -1.1 0.0
0.5 -0.7 0.0
0.4 -0.4 0.0
0.4 0.5 0.0
0.5 0.6 0.0
0.3 0.6 0.0
0.5 0.8 0.0
0.55 1.0 0.0
0.5 1.2 0.0
0.3 1.4 0.0
3 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 la 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 **Tupla3i**). Esta función ya está implementada y se encarga de añadir los vértices y las caras (leídos del PLY) a los correspondientes vectores.

El nombre de archivo debe ser una nombre relativo (relativo a la carpeta donde se ejecuta el programa) que nombrará alguno de los archivos de la carpeta **ply** en **recursos**, por ejemplo puede ser **../recursos/ply/beethoven.ply**. Si el archivo no es de los ue se proporcionan en la carpeta de recursos, entonces debe estar en la carpeta de trabajo (subcarpeta **plys**), y debemos de usar el nombre relativo **./recursos/ply/nombre.ply**

#### 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 parfil 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 (el nombre del archivo) y otro de salida (un vector de **Tupla3f** con los vértices del perfil). Se debe usar para leer los vértices del archivo **peon.ply** en la carpeta **plys** (en **recursos**).

#### 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/~jb Burkardt/data/ply/ply.html>
- Sitio web de Robin Bing-Yu Chen en 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 **escena.h** las declaraciones y en **escena.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.

## 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

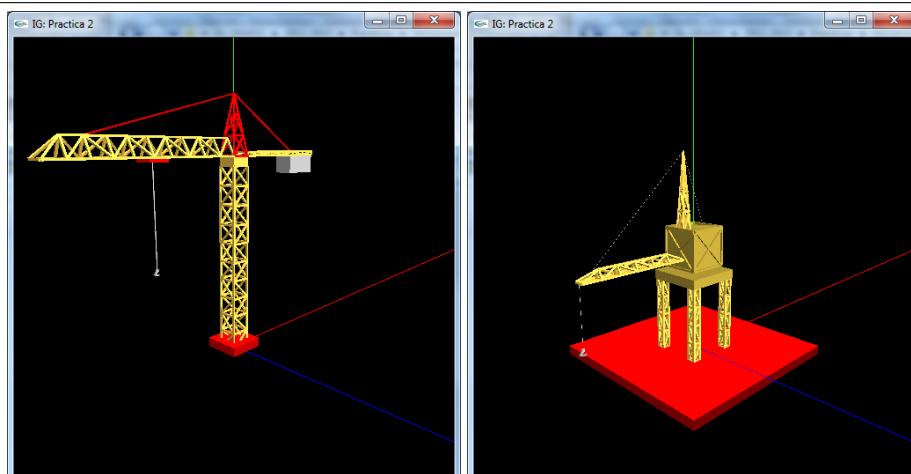


Figura 3.1: Ejemplos del resultado de la práctica 3.

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, y referencias a las mallas indexadas usadas como nodos terminales.

Se debe escribir el código necesario para visualizar un modelo jerárquico cualquiera, completando el método `visualizarGL` 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.

*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 como se ha dicho. El archivo también debe de incluir información de los parámetros o grados de libertad. Para cada uno de ellos se debe incluir: una breve descripción con unas pocas líneas, cual es la matriz afectada en el modelo y como depende esa matriz del valor de tiempo.

Tras diseñar el grafo de escena, para implementarlo debemos de definir una clase específica para dicho objeto. La clase (la llamamos *C*) es una clase derivada de **NodoGrafoEscena**. El nodo raíz de nuestro grafo de escena será un objeto de la clase *C*, y por tanto contiene todos los punteros a las matrices asociadas a los grados de libertad. Lo único que en principio debe de añadirse a *C* es un constructor. En dicho constructor se crean los subárboles del nodo raíz y se van añadiendo a la lista de entradas.

La clase *C* y el resto de clases necesarias se definirán en un archivo nuevo **modelo-jer.cpp** (en la carpeta **src** en la carpeta de trabajo) y se declarará en **modelo-jer.h** (en la carpeta **include** en la carpeta de trabajo). Esto implica que si usas **cmake** para compilar, habría que vaciar y regenerar la correspondiente carpeta **build**.

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

#### 3.4.1. 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 **Inicializar** 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.

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 leer el color actual en el cauce podemos usar el método `leerColorActual()` de la clase `Cauce` (devuelve una `Tupla3f`). Para cambiar el color actual del cauce podemos usar directamente la función `glColor3fv(c)` de OpenGL (donde `c` es una `Tupla3f`).

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 clase 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  $\Delta t$  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 y  $b = 2\pi wt$ , 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.

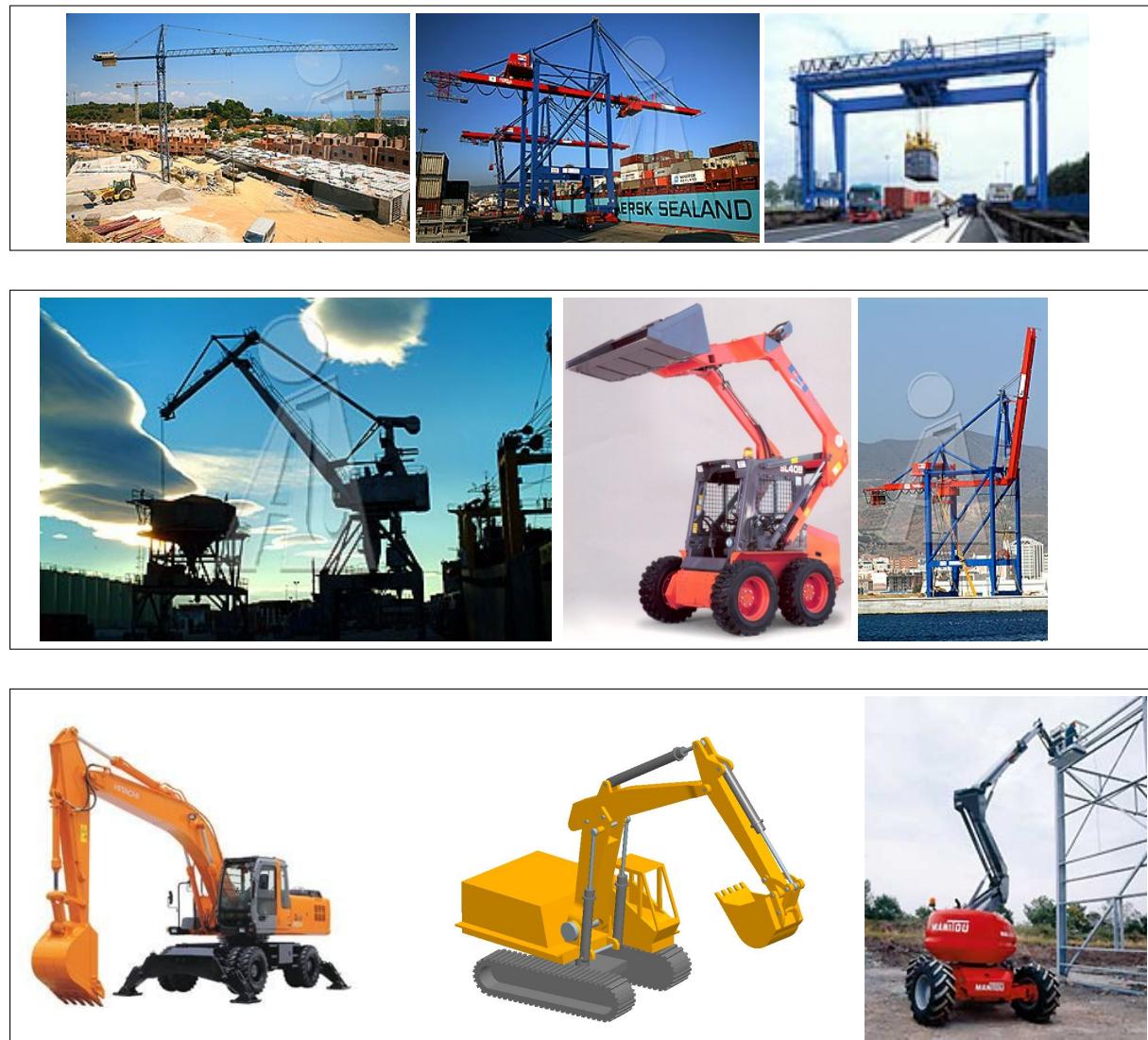
Hay que tener en cuenta que **actualizarEstadoParametro** debe usar punteros a las matrices que debe sobreescribir. Para ello se deben de registrar esos punteros como variables de instancia, es decir, durante la ejecución del constructor se deben de guardar en variables de instancia (de tipo **Matriz4f \***) 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 siguientes 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.





## 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 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**.
3. 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.
4. 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).
5. 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).
6. Se creará una nueva clase (de nombre **LataPeones** en los archivos nuevos **latapeones.cpp** y **latapeones.h**) para un grafo de escena jerárquico que contiene objetos con distintos materiales y textura (la escena con la lata y los peones). La geometría y aspecto de esta escena se describe en este guión (ver sección 4.4.8), pero no los valores de los parámetros del material que producen ese aspecto.

7. 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 4.4.7.
8. Se añadirán materiales y texturas al grafo de escena diseñado e implementado en la práctica 3. Ver sección 4.4.9.

### 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**: activa o desactiva el sombreado plano (inicialmente está desactivado).

Respecto al sombreado plano, hay que tener en cuenta que ambos tipos de cauce contemplan el modo de sombreado plano activado y el modo de sombreado plano desactivado (variable **sombr\_plano** del contexto de visualización), que se cambia con el método **fijarModoSombroPlano** del cauce. En el cauce fijo esto se traduce en que se usa *flat shading* (variable a **true**) y *Gouraud o smooth shading* (variable a **false**). Sin embargo, en el cauce programable, siempre se hace sombreado en los pixels, así que en este cauce la diferencia entre un modo u otro es que se usa la normal del triángulo (variable a **true**) o la normal interpolada (variable a **false**).

### 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 `FuenteLuz` y `ColFuentesLuz`

Las clases `FuenteLuz` y `ColecciónFuenteLuz` 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).

Finalmente 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).

#### 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. Hay que tener en cuenta que los coeficientes del material son valores reales, mientras que las reflectividades que se deben usar para **fijarEvalMIL** son tuplas RGB, por lo cual debemos de construir las tuplas con el mismo valor replicado en las tres componentes (ver las transparencias de teoría).

#### 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.

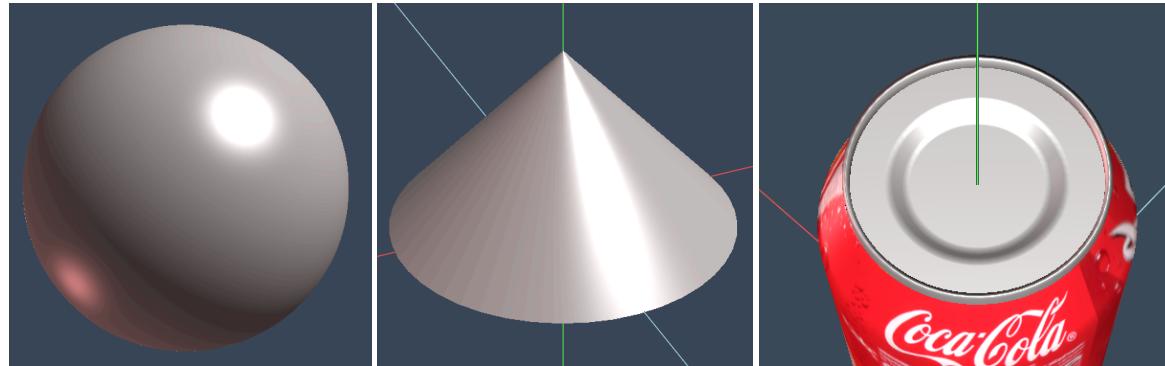
En particular será necesario completar el código de

- 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 initialize 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).

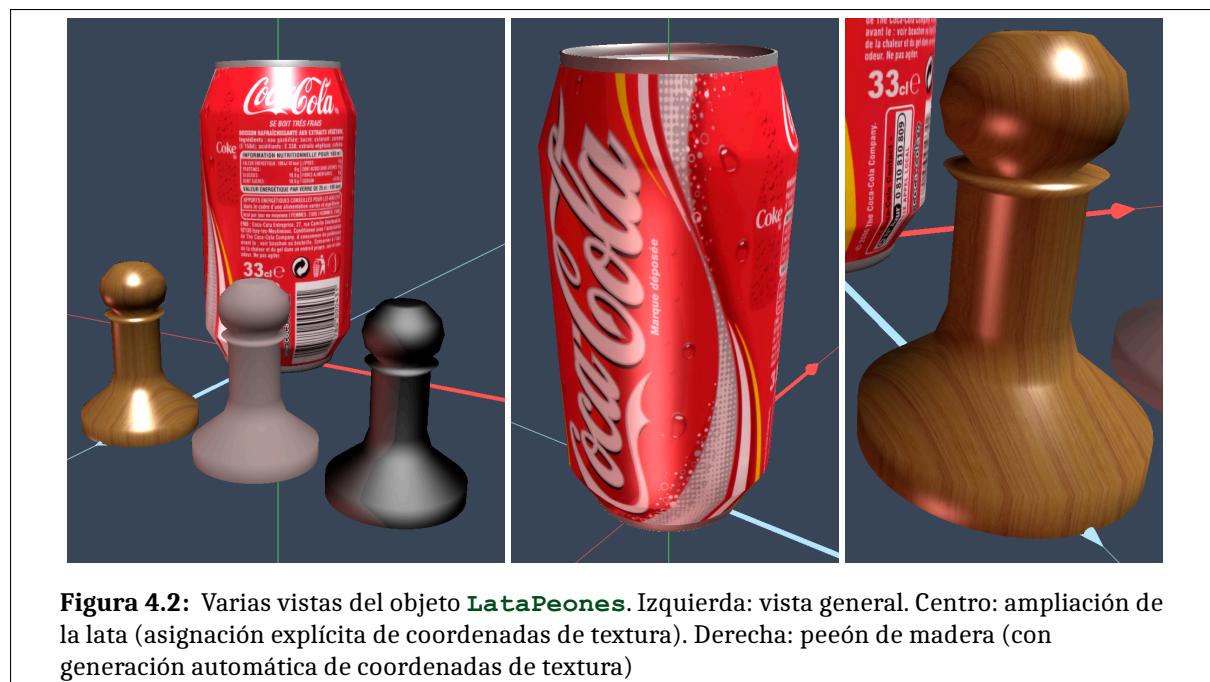
#### 4.4.7. Clase Escena4

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 4.4.8 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.

#### 4.4.8. 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 recursos). En concreto:

- Archivo **lata-pcue.ply**: perfil de la parte central, la que incorpora la textura de la lata (archivo **text-lata-1.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).

### 4.4.9. 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.

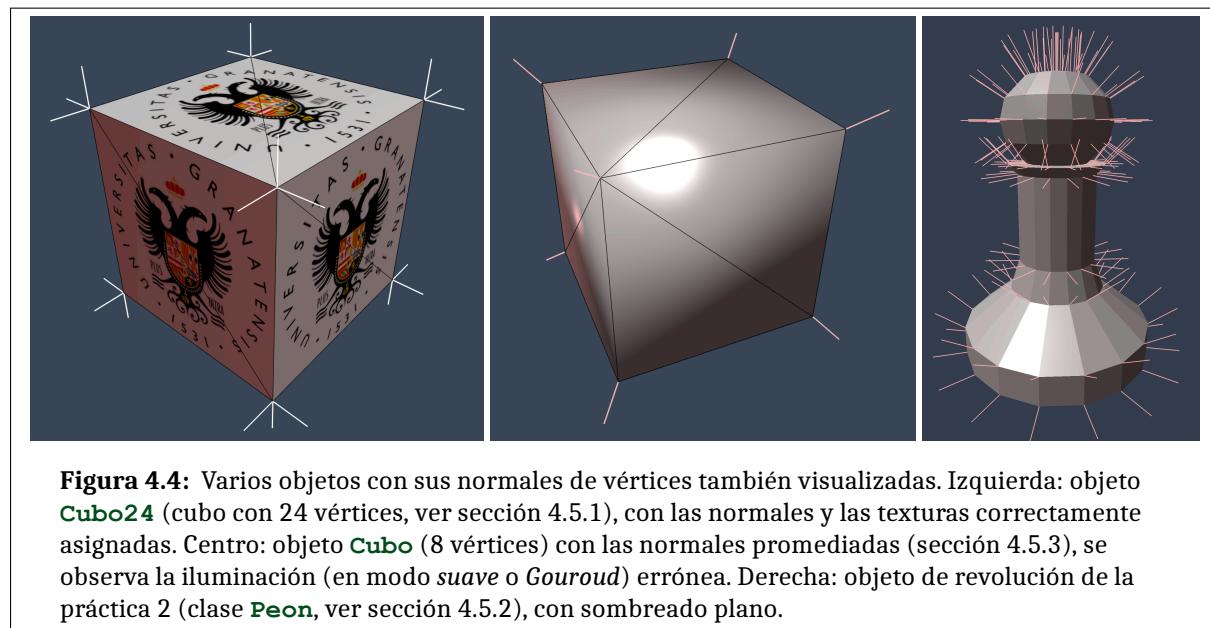
Para añadir materiales y texturas, será necesario extender los constructores de las clases deriva-

das 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 **imgs** dentro de la carpeta de trabajo, no en la carpeta de recursos.

## 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**



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.

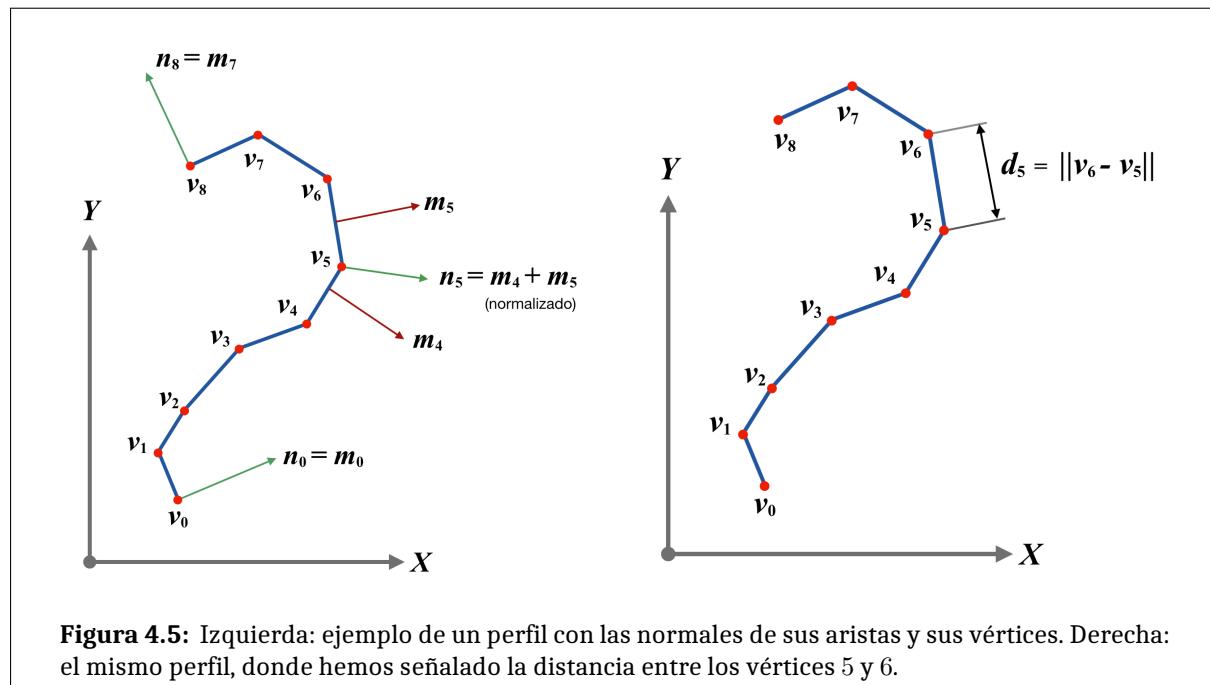
La clase **Cubo**, creada en la práctica 1, no es apropiada para la iluminación y texturas, esto se debe a que las tablas de normales y coordenadas de textura presentan discontinuidades en las aristas de un cubo real. En el caso de las normales, en dichas aristas confluyen dos caras con distintas orientaciones, 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 mate-

rial con textura y luego el cubo de 24 vértices. La imagen de textura será el archivo **window-icon.jpg** (disponible en la carpeta **recursos/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 4.4.7).

#### 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)  $90^\circ$  respecto de la  $i$ -ésima arista, paralela al vector  $\mathbf{v}_{i+1} - \mathbf{v}_i$ .
2. Calculamos las normales de los vértices ( $\mathbf{n}_0, \dots, \mathbf{n}_{n-1}$ ). Para ello hacemos  $\mathbf{v}_0 = \mathbf{m}_0$ , 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 (ro-

tada) 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

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

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

Los valores  $t_0, \dots, t_{n-1}$  se calculan y se guardan en un vector al inicio del método **inicIALIZAR**. Después, en un bucle doble añadimos cada vez el  $i$ -ésimo vértice de la  $j$ -ésima copia de perfil, donde  $j$  va desde 0 hasta  $n - 1$ , ambos incluidos ( $n$  es el número de copias del perfil, ver 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  $\mathbf{p}, \mathbf{q}$  y  $\mathbf{r}$ . A partir de estas coordenadas se calculan los vectores  $\mathbf{a}$  y  $\mathbf{b}$  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  $\mathbf{p}, \mathbf{q}$  y  $\mathbf{r}$  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 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.