Prácticas de Informática Gráfica

Grado en Informática y Matemáticas. Curso 2018-19.

UNIVERSIDAD
DE GRANADA

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

Índice general.

Prerequisitos software.

En esta sección se detallan las herramientas software necesarias para realizar las prácticas en los sistemas operativos Linux (Ubuntu u otros) y macOS (de Apple). Respecto al sistema operativo Windows, las prácticas se pueden realizar en Ubuntu ejecutándose en una máquina virtual, o bien instalando Visual Studio.

0.1. Sistema Operativo Linux

0.1.1. 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).

0.1.2. 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 on 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
```

0.1.3. Librería GLEW

La librería GLEW es necesaria en Linux para que las funciones de la versión 2.1 de OpenGL y posteriores pueden ser invocadas (inicialmente esas funciones abortan al llamarlas). GLEW se encarga, en tiempo de ejecución, de hacer que esas funciones esten 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
```

0.1.4. 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
```

0.1.5. 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
```

0.2. Sistema Operativo macOS

0.2.1. 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 XCode, usaremos la implementación de OpenGL que se proporciona con XCode (la librería GLEW no es necesaria en este sistema operativo).

0.2.2. Librería GLFW

Para instalar esta librería, 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 `cmake`, se puede instalar GLFW. Para ello, comprueba ahora que tienes la orden `cmake` disponible en la shell. A continuación, debes acceder a la página de descargas del proyecto GLFW:

☞ http://www.glfw.org/download.html

aquí, 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 raiz (de nombre `glfw-...`). Después compilamos e instalamos la librería con estas órdenes:

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

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

0.2.3. Librería JPEG

Es necesario instalar los archivos correspondientes a la versión de desarrollo de la librería de lectura de jpegs. Respecto a esta librería para jpegs, se puede compilar el código fuente de la misma. Para esto, basta con descargar el archivo con el código fuente de la versión más moderna de la librería a un carpeta nueva vacía, y después compilar e instalar los archivos. Se puede hacer con estas 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 ordenes se refieren a la versión 9b de la librería, para futuras versiones habrá que cambiar `9b` por lo que corresponda. Si todo va bien, esto dejará los archivos `.h` en `/usr/local/`**`include`** y los archivos `.a` o `.dylib` en `/usr/local/lib`. Para poder compilar, debemos de asegurarnos de que el compilador y el enlazador tienen estas carpetas en sus paths de búsqueda, es decir, debemos de usar la opción `-I/usr/local/`**`include`** al compilar, y la opción `-L/usr/local/lib` al enlazar.

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 en modo inmediato y en modo diferido.

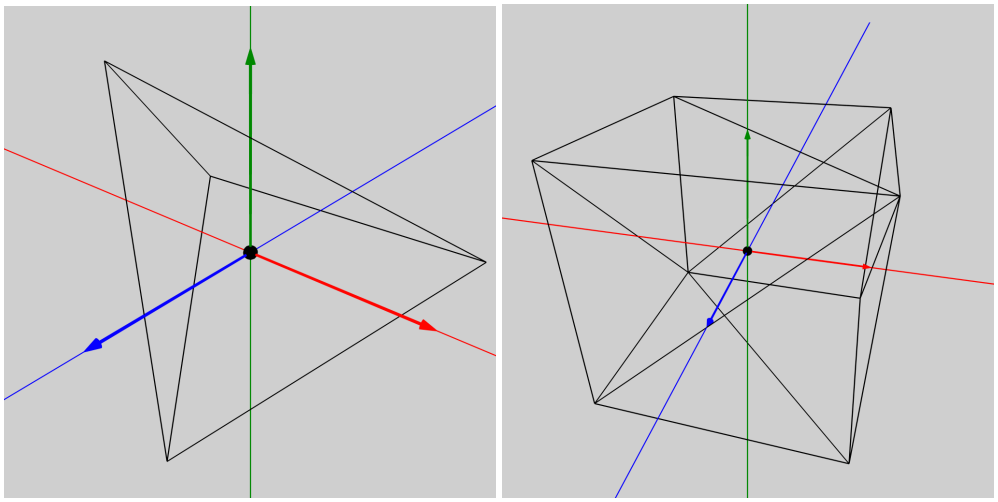
1.2. Desarrollo

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 y visualizar un **tetraedro** y un **cubo**. Para ello, creará las estructuras de datos que permitan representarlos mediante sus vértices y caras. Usando dicha información y las primitivas de dibujo de OpenGL los visualizará con los siguientes modos:

- Puntos: se visualiza un punto en la posición de cada vértice del modelo.
- Alambre: se viualiza como un segmento cada arista del modelo.
- Sólido: se visualizan los triángulos rellenos todos de un mismo color (plano).

Los alumnos escribirán código para visualizar las mallas usando el modo inmediato, tanto con `glBegin`/`glVertex`/`glEnd`, como con `glDrawElements`, y en modo diferido con `glDrawElements`.



(a) Tetraedro (b) Cubo

Figura 1.1: Tetraedro y cubo visualizados en modo alambre.

1.3. Evaluación

La evaluación de la práctica se hará mediante la entrega de las prácticas (via la plataforma PRADO), seguida de un sesión de evaluación en el laboratorio, en la cual se harán modificaciones sobre el código del alumno y el nuevo código se subirá a la plataforma PRADO.

- La nota máxima será de 10 puntos, si el alumno implementa todos los requerimientos descritos en este guión, y además hace visualización usando el cauce gráfico programable.
- Si el alumno implementa la visualización usando exclusivamente el cauce de la funcionalidad fija, la nota máxima será de 7 puntos.

Las modificaciones que se pidan durante la sesión de evaluación serán evaluadas entre 0 y la nota máxima descrita aquí arriba.

1.4. Teclas a usar. Interacción.

El programa permite pulsar las siguientes teclas:

- **tecla m/M**: cambia el modo de visualización activo (pasa al siguiente, o del último al primero)
- **tecla p/P**: cambia la práctica activa (pasa a la siguiente, o de la última a la primera).
- **tecla o/O**: cambia el objeto activo dentro de la práctica (pasa al siguiente, o del último al primero)

Además de estas teclas, la plantilla que se proporciona incopora otras teclas, válidas para todas las prácticas. En concreto, son las siguientes:

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

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

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

1.5. Implementación

Una vez descomprimido el archivo `.zip` con la plantilla de prácticas en una carpeta vacía, se crearán estas subcarpetas:

- `objs`: carpeta vacía donde se crearán los archivos `.o` al compilar.
- `bin`: carpeta vacía donde se guardará el archivo ejecutable (`prac_exe`) al compilar.
- `plys`: archivos ply de ejemplo para la práctica 2, proporcionados por el profesor.
- `imgs`: imágenes de textura para la práctica 4, proporcionadas por el profesor.
- **`include`**: archivos de cabecera de los módulos auxiliares (p.ej.: manejo de tuplas de valores reales para coordenadas y colores), proporcionados por el profesor.
- `srcs`: archivos fuente C/C++ de los módulos auxiliares (p.e.: lectura de plys, lectura de jpgs, shaders, etc...).
- `alum-srcs`: archivos fuente del programa principal, y de cada una de las prácticas (todos ellos son a completar o extender por el alumno).

- `alum-archs`: carpeta vacía, aquí el alumno incluirá archivos `.ply`, imágenes (`.jpg`), o de otros tipos, distintos de los proporcionados por el profesor, y que el alumno use en sus prácticas (quizás no sea necesario para la práctica 1, pero sí probablemente para otras)

Para realizar las prácticas es necesario trabajar en la carpeta `alum-srcs`. En esa carpeta se debe completar y extender el código que se proporciona en el archivo `practica1.cpp`

Para compilar el código, basta con teclear `make` (estando en la carpeta `alum-srcs`), esta orden leerá el archivos `makefile` y se encargará de compilar, enlazar y ejecutar el código, incluyendo los módulos auxiliares disponibles en la carpeta `srcs` (cabeceras en **`include`**). Si no hay errores, se producirá en la carpeta `bin` un ejecutable de nombre `prac_exe`, y a continuación se ejecuta.

No se debe de modificar en ningún caso el código de los archivos en las carpetas `srcs`, **`include`**, `bin`, `imgs`, `plys` y `objs`. Tampoco se debe añadir ningún archivo en esas carpetas. La revisión de las prácticas para evaluación se hará con el contenido no modifcado de dichas carpetas.

El archivo `makefile` que hay en `alum-srcs` se debe de modificar, pero exclusivamente para añadir nombre de unidades de compilación en la definición de la variable `units_alu`. Se deben añadir los nombres de las unidades (archivos `.cpp`) que estén en `alum-srcs` y que se quieren enlazar para crear el ejecutable.

1.5.1. Inicialización y gestión de eventos

En el archivo **`main`**`.cpp` (en la la función **`FGE_PulsarTeclaNormal`**) es necesario incluir el código necesario para gestionar el evento de teclado correspondiente a la pulsación de la tecla **M**, que permite cambiar el valor de la variable global `modoVis` (dentro de la estructura `contextoVis`), que determina el modo de visualización actual. También es necesario añadir el código que gestiona el evento de pulsación de la tecla **P**, que cambia la práctica actual (ahora mismo únicamente está activada la práctica 1, pero en las siguientes prácticas permite cambiar de una a otra).

La implementación requiere completar las siguientes funciones (en `practica1.cpp`):

- **`P1_Inicializar`**

  Sirve para crear las tablas de vértices y caras que se requieren para la práctica. Esta función se invoca desde **`main`**`.cpp` una única vez al inicio del programa, cuando ya se ha creado la ventana he inicializado OpenGL.

- **`P1_DibujarObjetos`**

  Sirve para dibujar las mallas, usando el parámetro **`cv`**, que contiene la variable que determina el tipo o modo de visualización de primitivas. Esta función se invoca desde **`main`**`.cpp` cada vez que se recibe el evento de redibujado.

- **`P1_FGE_PulsarTeclaNormal`**

  Está función se invoca desde **`main`**`.cpp` cuando se pulsa una tecla normal, la práctica 1 está activa, y la tecla no es procesada en el **`main`**`.cpp`. Sirve para cambiar entre la visualización del tetraedro y el cubo (cambiar el valor de la variable `objeto_activo`) cuando se pulsan alguna tecla. Debe devolver **`true`** para indicar que la tecla pulsada corresponde al cambio de objeto activo, y **`false`** para indicar que la tecla no corresponde a esta práctica.

1.5.2. Contexto y modos de visualización

En el archivo `practicas.hpp` (dentro de `srcs-alum`) se declara la clase **`ContextoVis`**, que contiene, como variables de instancia, distintos parámetros y variables de estado usados durante la visualización de objetos y escenarios en las prácticas. Inicialmente (para esta práctica 1), contiene únicamente el modo de visualización (puntos, alambre, sólido, etc...), en concreto está en la variable de instancia `modoVis`, que es un valor de un tipo enumerado **`ModoVis`**, tipo que también se declara en ese archivo de cabecera.

Las declaraciones son como se indica aquí:

```
// tipo enumerado para los modos de visualización:
typedef enum
      { modoPuntos, modoAlambre, modoSolido, modoAjedrez } ModoVis ;
// numero de modos distintos
const int numModosVisu = 4 ;
// clase para los distintos parámetros de la visualización
class ContextoVis
{
   public:
      ModoVis modoVis ;  // modo de visualización activo actualmente
} ;
```

Más adelante se definirán nuevos modos de visualización y otros parámetros en la clase **`ContextoVis`**.

1.5.3. Clase abstracta para objetos gráficos 3d.

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 llamado **`visualizarGL`**, con una declaración como esta (en el archivo **`Objeto3D`**`.hpp`)

```
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() ;
} ;
```

Cada clase concreta proveerá su propio método **`visualizarGL`**. Estos métodos tienen siempre un parámetro de tipo **`ContextoVis`**, que contendrá el modo de visualización que se debe usar (entre otras cosas).

Cualquier tipo de objeto que pueda ser visualizado en pantalla con OpenGL se implementará con una clase derivada de **`Objeto3D`**, que contendrá una implementación concreta del método virtual **`visualizarGL`**. El parámetro `modoVis` (dentro de **`cv`**) servirá para distinguir el modo de visualización que se requiere.

1.5.4. Clase para mallas indexadas.

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 modo inmediato usando la instrucción **glDrawElements** o bien **glBegin**/**glEnd**. También se pueden visualizar en modo diferido (con VBOs). Para implementar este tipo de mallas crearemos una clase (**MallaInd**), derivada de **Objeto3D** y que contiene:

- Como variables de instancia privadas, la tabla de coordenadas de vértices y la tabla de caras. La primera puede ser un vector stl con entradas de tipo **Tupla3f**, y la segunda un vector stl con entradas tipo **Tupla3i**.
- Como método público virtual, el método **visualizarGL**, que visualiza la malla teniendo en cuenta el parámetro modo, y usando las dos tablas descritas arriba.

El esquema puede ser como sigue:

```
#include "Objeto3D.hpp"

class MallaInd : public Objeto3D
{
   protected:
    // declarar aquí tablas de vértices y caras
    // ....
   public:
      virtual void visualizarGL( ContextoVis & cv ) ;
    // .....
} ;
```

La declaración de esta clase se puede poner en un archivo de nombre **MallaInd**.hpp, y su implementación en **MallaInd**.cpp. Es necesario añadir al archivo makefile el nombre **MallaInd** (en units_loc), para lograr que este archivo se compile y enlace con el resto.

Las tablas de vértices y caras se pueden implementar con arrays de C clásicos que contienen flotantes o enteros. No obstante, se recomienda usar vectores STL de tuplas de flotantes o enteros, ya que esto facilitará la manipulación posterior. En este guión, se describen más adelante los tipos que se proporcionan para tuplas de flotantes o enteros (tipos **Tupla3f** y **Tupla3i**).

1.5.5. Programación del cauce gráfico

Se tiene la opción de usar programación del cauce gráfico para realizar la visualización. Esta programación permitirá visualizar las primitivas de esta primera práctica usando para ello un *shader program* distinto del proporcionado en la funcionalidad fija de OpenGL.

El código fuente de este shader puede coincidir con el fuente sencillo visto en las transparencias de teoría para un *fragment shader* y un *vertex shader* básicos. También se pueden usar las funciones que hemos visto para cargar, compilar y enlazar los programas, que ya están disponibles en la unidad de compilación shaders que hay en las carpetas srcs (shaders.cpp) e **include** (shaders.hpp).

La implementación de esta funcionalidad requiere modificar **main**.cpp para incluir una variable global (de tipo **GLuint**) con el identificador del programa. Esta variable se usará para activar dicho

programa siempre antes de visualizar.

Los dos archivos `.glsl` requeridos deben de estar en `srcs-alum`, y se deben entregar junto con el resto de fuentes de este directorio.

1.5.6. Uso de OpenGL 2.0

En el caso de usar el sistema operativo Linux, no es posible invocar directamente las funciones que no existían en la versión 1.2 de OpenGL y que se han añadido en la versión 2.0 y posteriores. Si se hace, es posible que el programa aborte al intentar llamarlas (a pesar de haberse compilado y enlazado correctamente el programa). En particular, corresponden a OpenGL 2.1 las funciones relacionadas con el uso del modo diferido (uso de VBOs) y las relacionadas con la programación del cauce gráfico (compilar y ejecutar shaders).

El problema está en que esas funciones tienen asociado como punto de entrada (dirección en memoria de la primera instrucción ejecutable) un puntero nulo, lo cual hace que el sistema operativo aborte nuestro programa, ya que estamos intentando hacer un salto a la direción de memoria 0. Para evitar esto, en linux se puede instalar la librería GLEW, y llamar a la función `InicializarGLEW` al final de `Inicializa_OpenGL` en **`main`**`.cpp`. La función `Inicializa_GLEW` está declarada en `aux.hpp` y definida en `aux.cpp`. En el caso de ordenadores con sistema operativo macOS, este problema no existe, y no es necesario usar GLEW para esto (en macOS, la función `Inicializa_GLEW` no hace nada)

1.5.7. Clases para los objetos de la práctica 1

Los objetos cubo y tetraedro se implementarán usando dos clases derivadas de **`MallaInd`**, cada una de ellas definirá un nuevo constructor que construirá las dos tablas correspondientes a cada tipo de objeto. Estas clases se pueden declarar e implementar en un par de archivos nuevos, o se puede hacer en `practica1.hpp/.cpp`. En cualquier caso, en el archivo `practica1.cpp` habrá dos variables globales nuevas, una será una instancia del cubo y otra una instancia del tetraedro. El esquema para la clase Cubo (p.ej.) puede ser este:

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

En la función **`P1_Inicializar`** se crearán las instancias del cubo y el tetraedro. En la función **`P1_DibujarObjetos`** se visualizará el cubo o el tetraedro.

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

Haciendo *include* de `tuplasg.hpp`, están disponibles estos tipos de datos (clases):

```
// 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 predefinidias 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) ;   // apropidado 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)
```

1.6. Instrucciones para subir los archivos

Para entregar la práctica se creará y se subirá un único archivo `.zip`, de nombre igual a `P1.zip` siguiendo estas indicaciones:

- Hacer un zip (llamado `P1-fuentes.zip`) con todos los fuentes de la carpeta `alum-srcs`, incluyendo **`main`**`.cpp` o cualquier otro, así como los shaders si los hay (archivos `.glsl`). El `zip` debe hacerse directamente en `alum-srcs`, y no puede tener carpetas dentro de él, solo puede contener directamente los archivos indicados.
- La práctica debe poder compilarse con el mismo archivo `makefile` que se proporciona (al que se le añaden las unidades de compilación en `units_alu`)
- Incluir un archivo de texto ascii y de nombre `leeme.txt`, en ese archivo, incluir:
  - Si se ha hecho programación del cauce gráfico o no se ha hecho. En caso afirmativo, se debe de indicar el nombre de los archivos con los fuentes del shader (archivos `.glsl`).
  - Sistema operativo usado para compilar, y, si se ha usado algún entorno de desarrollo específico, indicarlo.
  - Todas las teclas que se pueden pulsar y utilidad de cada tecla.
  - Si se ha implementado alguna funcionalidad no descrita en este guión o no. En caso afirmativo, incluir la descripción de dicha funcionalidad (p.ej.: que tipo de objetos se han implementado, donde está el código, que parámetros configurables tiene, etc...)

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 contine 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.
- Opcionalmente, a visualizar mallas de triángulos usando el modo diferido, adicionalmente al modo inmediato.

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.

2.3. Creación del sólido por revolución

En esta sección se detalla el algoritmo de creación del sólido por revolución (se implementa en un constructor, ver la sección sobre implementación). Partimos de un perfil inicial u original, es una

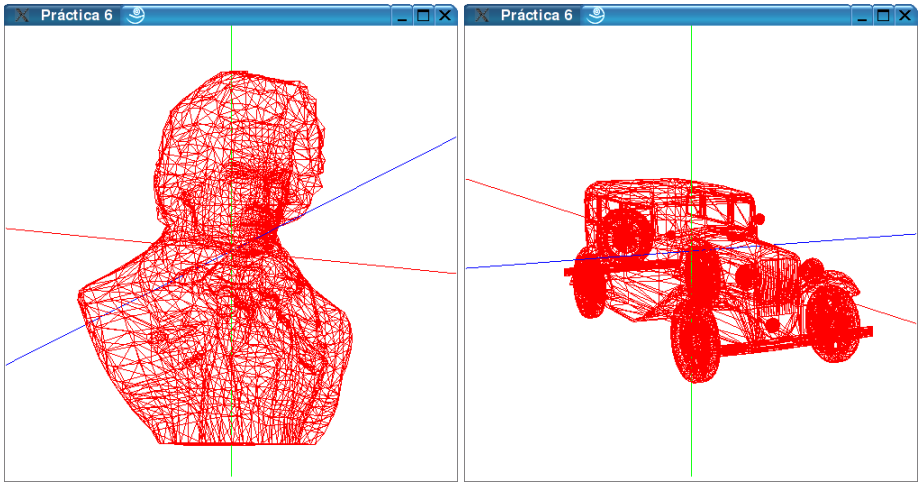
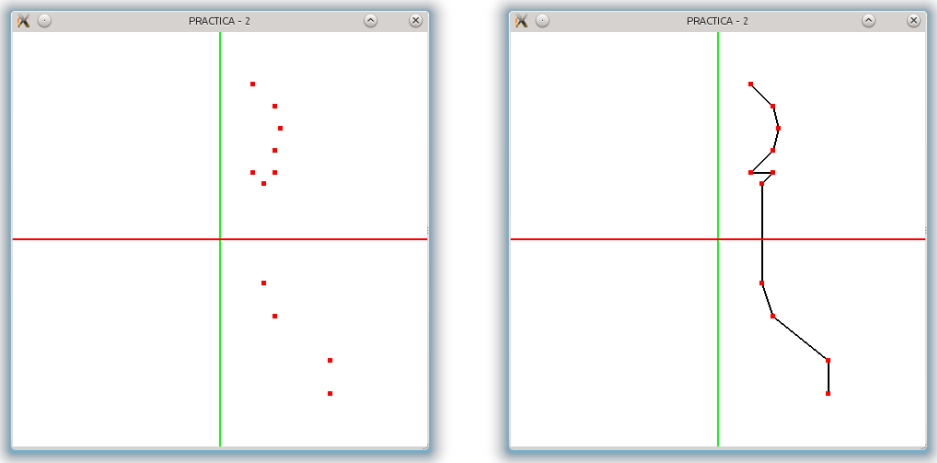


Figura 2.1: Objetos PLY.



(a) Puntos (b) Polilínea

Figura 2.2: Perfil inicial.

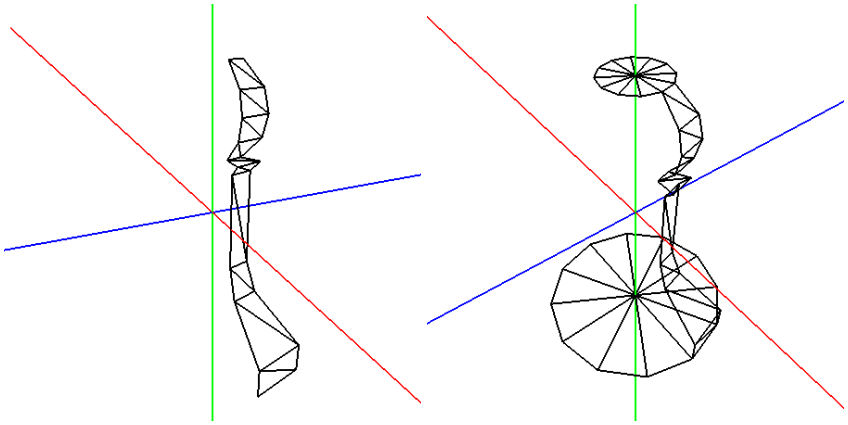
secuencia de m tuplas de coordenadas de vértices en 3D, todas esas coordenadas con $z = 0$

$$\mathbf{p}_0\ ,\ \mathbf{p}_1\ ,\ \cdots\ ,\ \mathbf{p}_{m-1}$$

(ver figura 2.2).

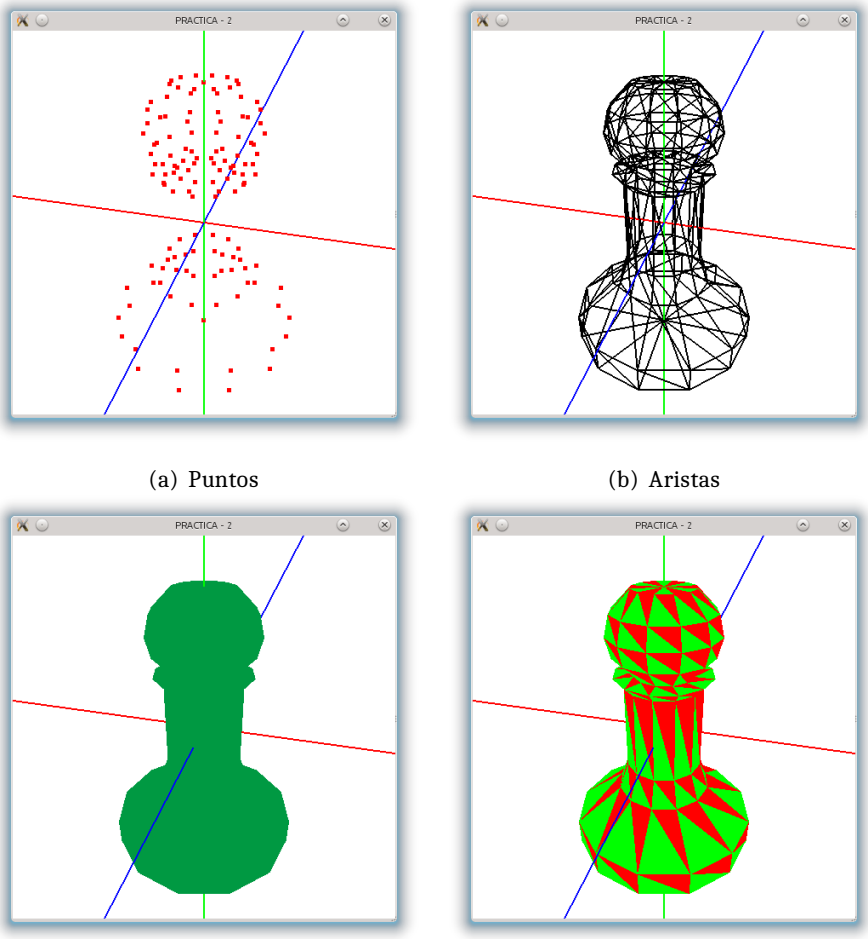
Usando este perfil base u original, queremos crear un total de n instancias o copias rotadas de dicho perfil. Esto implica insertar un total de nm vértices en la tabla de vértices, y después todas las caras (triángulos) correspondientes . Para ello se pueden dar los pasos que se detallan en las siguientes subsecciones.

El modelo poligonal finalmente obtenido también se podrá visualizar usando cualquiera de los distintos modos de visualización implementados para la primera práctica (ver figura 2.4).



(a) (b)

Figura 2.3: Caras del sólido a construir: (a) logitudinales (solo un lado es mostrado) y (b) incluyendo las tapas superior e inferior.



(a) Puntos (b) Aristas

(c) Sólido (d) Ajedrez

Figura 2.4: Sólido generado por revolución con distintos modos de visualización.

2.3.1. Lectura o creación del perfil inicial

El perfil inicial se puede leer de un fichero PLY cuyo contenido sólo ha de tener las coordenadas de los vértices (las caras no se leen) (ver la sección sobre lectura de PLYS). Este fichero PLY puede escribirse manualmente, o bien se puede usar el que se proporciona en el material de la práctica, correspondiente a la figura de un peón, y que se muestra a continuación:

```
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
```

Además de leer el perfil original de un arhivo 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.

2.3.2. Creación de la tabla de vértices

A partir del perfil original, creamos los n perfiles del objeto. El vértice número j del perfil número i (lo llamamos $\mathbf{q}_{ij}$) se obtiene rotando cada punto del perfil original un ángulo proporcional a i. Es decir:

$$\mathbf{q}_{ij} \;=\; R_i \mathbf{p}_j$$

donde i va desde 0 hasta $n-1$, y j va desde 0 hasta $m-1$. Suponemos que las coordenadas X de los vértices del perfil original son todas estrictamente mayores que cero. Asimismo, suponemos que las coordenadas Y en dichos perfiles originales son crecientes (la primera es la mínima y la última es la máxima).

El símbolo R_i representa una transformación afín de rotación entorno al eje Y, un ángulo igual a $i\alpha$, donde α es el ángulo entre dos perfiles consecutivos, ángulo cuyo valor exacto depende de n y de si se usa la opción de cerrar la malla o no se usa (ver más abajo). Dicha transformación se puede implementar, lógicamente, como una matriz de rotación.

Cada uno de los nm vértices obtenidos se inserta en la tabla o vector de vértices, según la estructura de datos creada en la práctica anterior. El índice k de cada vértice en el vector depende de los índices i y j usados para generar sus coordenadas. Lo más fácil es añadir de forma consecutiva todos los vértices de cada copia del perfil original. De esta forma sabemos que el j-ésimo vértice de la i-ésima copia del perfil se almacena en la entrada con índice $k = im + j$ del vector de vértices. Esto facilita la creación de la tabla de caras (ver el siguiente apartado).

2.3.3. Creación de la tabla de caras (triángulos)

Una vez creada la tabla de vértices, debemos de crear la tabla de caras (triángulos). Consideramos cada grupo de cuatro vértices adyacentes, tomando dos consecutivos de un pérfil y los otros dos vértices correspondientes del siguiente perfil. Con esos cuatro vértices se forman dos triángulos, que se insertan en la tabla de caras. En la figura 2.3(a) se muestran los triángulos así obtenidos solamente entre dos perfiles, para una mejor visualización. Los vértices de los triángulos tienen que estar ordenados en el sentido contrario a las agujas del reloj, según se observan desde fuera del objeto.

Habrá un parámetro lógico que indicará si la malla se debe *cerrar* o no. Aquí *cerrar* la malla significa no crear la última copia del perfil, a 360^o de la primera copia (a 0^o). En este caso, el último perfil (ahora a menos de 360^o) se une al primero. Por el contrario, si se decide no cerrar la malla, la ultima copia del perfil estará efectivamente a 360^o, y será igual a la primera. La opción de no cerrar la malla será útil para la práctica 4.

Para crear las caras, se recorren todos los pares (i, j), con i desde 0 hasta $n-1$, y con j desde 0 hasta $m-2$. Cada uno de estos pares (i, j) está asociado al j-ésimo vértice de la i-ésima copia del perfil. Para cada uno de esos vértices, creamos dos triángulos, usando los cuatro índices correspondientes a ese vértice y los otros vértices tres adyacentes. Los pares asociados a los cuatro vértices son:

$$\begin{array}{ll} (\,i,\,j+1\,) & (\,(i+1) \bmod n,\,j+1\,) \\ (\,i,\,j\,) & (\,(i+1) \bmod n,\,j\,) \end{array}$$

el módulo solo debe usarse si queremos cerrar la malla, de forma que conectamos el último perfil con el primero. Si no queremos cerrar la malla, entonces los cuatro pares asociados a los cuatro vértices son:

$$\begin{array}{ll} (\,i,\,j+1\,) & (\,i+1,\,j+1\,) \\ (\,i,\,j\,) & (\,i+1,\,j\,) \end{array}$$

en este caso (si no queremos cerrar la malla), i iría desde 0 hasta $n-2$, en lugar de hasta $n-1$.

A continuación creamos las tapas del sólido tanto inferior como superior (ver figura 2.3(b)). Para ello se han de añadir dos puntos al vector de vértices que se obtienen por la proyección sobre el eje de rotación del primer y último punto del perfil inicial. Estos dos vértices serán compartidos por todas las caras de las tapas superior e inferior. Esta creación de las tapas dependerá de un valor lógico (`crear_tapas`), que será un parámetro del procedimiento constructor de las mallas por revolución. El parámetro será `true` si queremos que se creen las tapas y `false` en caso contrario. Todas las caras creadas se añaden al vector de caras.

2.4. Teclas a usar

En esta práctica, al igual que en las demás prácticas, (e independientemente de otras que se usen para otras cosas) se deben usar estas teclas:

- **tecla m/M**: cambia el modo de visualización activo (pasa al siguiente, o del último al primero)
- **tecla p/P**: cambia la práctica activa (pasa a la siguiente, o de la última a la primera).
- **tecla o/O**: cambia el objeto activo dentro de la práctica (pasa al siguiente, o del último al primero)
- **tecla v/V**: activar o desactivaer el uso de modo diferido (VBOs) para visualización (esto es opcional, ver la sección sobre modo diferido)

Las dos primeras ya están impementadas (desde la práctica 1) en `main.cpp`, la tercera debe implementarse en la función gestora de pulsación de tecla normal específica de la práctica 2 (es decir, en la función `P2_FGE_PulsarTeclaNormal`). La cuarta debe de gestionarse en `main.cpp` (en la función `FGE_PulsarTeclaNormal`).

Para estas cuatro teclas, la función es la misma independientemente de que se pulsen en minúsculas o en mayúsculas.

2.5. Implementación

La implementación de esta práctica requiere la creación de dos archivos fuente C++ de nombres `practica2.cpp` y `practica2.hpp`, en carpeta `srcs-alum`. El primero contendrá la implementación o definición de las funciones y el segundo las declaraciones de las mismas, al igual que en la práctica 1. Asimismo, será necesario incluir `practica2` en la lista de unidades a compilar, en el archivo `makefile` (en la definición de la variable `units_loc`)

En el archivo `main.cpp` se debe gestionar la variable que indica cual es la práctica activa (variable `practica_activa`), de forma que mediante alguna tecla se puede conmutar entre las distintas prácticas (cambiar el valor de la variable). El procesamiento de la tecla debe añadirse a `main.cpp` (en la función `FGE_PulsarTeclaNormal`, o bien en `FGE_PulsarTeclaEspecial`, en base al tipo de tecla elegida). Inicialmente, la práctica activa será la 2, aunque por supuesto podrá cambiarse si el usuario quiere.

€s importante tener en cuenta que se debe de poner en `alum-archs` los archivos PLY que el alumno descarge de internet, distintos de los proporcionados en la plantilla se prácticas. Puesto que el binario ejecutable se ejeucta en la carpeta `alum-srcs` (hermana de `alum-archs`), el *path* y nombre usado para la lectura debe ser de esta forma: `../alum-archs/<nombre>.ply`.

La implementación requiere escribir las siguientes funciones (en `practica2.cpp`):

Función de inicialización: `void P2_Inicializar()`

Sirve para crear los objetos que se requieren para la práctica. Esta función se invoca desde `main.cpp` una única vez al inicio del programa, inmediatamente después de la llamada ya existente a `P1_Inicializar`, para crear los objetos.

Se crearán con `new` (es decir, en memoria dinámica) un objeto de tipo malla PLY y otro objeto de revolución, usando los constructores de las clases que se detallan más abajo. Los dos punteros a dichos objetos se guardan como variables globales de `practica2.cpp` (para evitar colisiones de nombres, se aconseja declaralos como `static`, y también se aconseja que esten inicializados a NULL en su declaración).

Los nombres de los dos archivos ply a cargar se escriben directamente en el código fuente, se pueden cambiar pero recompilando el programa. Hay que tener en cuenta que los nombres son nombres relativos a la carpeta `srcs_alum`, que es donde se ejecutan las prácticas. Por tanto, si se refie-

ren a archivos proporcionados en la plantilla, están en la carpta `plys`, y se usará prefijo `../plys/`, mientras que si son archivos buscados por el alumno, no se pone prefijo alguno.

Función de dibujo: `void P2_DibujarObjetos( ContextoVis & cv )`

Es para dibujar los mallas usando los punteros a los objetos descritos arriba, y usando también el parámetro `modoVis` (dentro de **`cv`**) para determinar el tipo o modo de visualización de primitivas (con la misma interpretación que en la práctica 1). Esta función se invoca desde **`main`**`.cpp` cada vez que se recibe el evento de redibujado, y la práctica activa es la práctica 2. Para ello, se debe insertar la nueva llamada en **`main`**`.cpp` (función **`FGE_Redibujado`**), en base a la práctica activa en cada momento (que ahora puede ser la 1 o la 2).

Función de tecla normal: `bool P2_FGE_PulsarTeclaNormal( unsigned char tecla )`

Esta función se invoca desde **`main`**`.cpp` cuando se pulsa una tecla normal, la práctica 2 está activa, y la tecla no es procesada en el **`main`**`.cpp` (hay que añadir la llamada en el **`main`**`.cpp`). La función sirve para cambiar entre la visualización de la malla leída de un ply y la malla obtenida por revolución, cuando se pulsan alguna tecla. Debe devolver **`true`** para indicar que la tecla pulsada corresponde al cambio de objeto activo, y **`false`** para indicar que la tecla no corresponde a esta práctica. Para gestionar cual es el objeto activo en cada momento, se puede usar una variable global en `practica2.cpp` llamada `p2_objeto_activo`.

2.5.1. Clase para 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 una nueva 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, puede quedar así:

```
// clase mallas indexadas obtenidas de un archivo PLY
class MallaPLY : public MallaInd
{
   public:
       // constructor
       // se debe especificar el nombre completo del archivo a leer
       MallaPLY( const std::string & nombre_arch ) ;
} ;
```

la declaración de esta nueva clase se puede hacer en su propio par de archivos fuente (`.hpp`/`.cpp`, en `srcs-alum`), o incorporarla a los archivos ya creados para las mallas.

2.5.2. Clase para mallas creadas a partir de un perfil, por revolución.

La implementación de los objetos tipo malla obtenidos a partir de un perfil, por revolución, debe hacerse usando una nueva clase (**`MallaRevol`**), derivada de la clase para **`MallaInd`**. La clase **`MallaRevol`**, al igual que en la otra clase descrita en esta práctica, no introduce un nuevo método de visualización. De nuevo, la única diferencia de este tipo de mallas es como se construyen, y por tanto tiene un constructor que construye la tablas usando, entre otros, un parámetro con el nom-

bre del archivo PLY con el perfil. Este constructor lee los vértices de un archivo PLY, construye un vector de tuplas, y después invoca la función `crearMallaRevol` (no está en la plantilla). Por tanto, la declaración de la clase puede quedar así:

```
// clase mallas indexadas obtenidas de un perfil, por revolución
class MallaRevol : public MallaInd
{
 protected:

 // constructor por defecto (vacío)
 MallaRevol() {}

 // crear la malla de revolución a partir del perfil original
 // (el número de vértices, M, es el número de tuplas del vector)

 // Método que crea las tablas vertices y triangulos
 void crearMallaRevol
 ( const std::vector<Tupla3f> & perfil_original,      // vértices del perfil original
   const unsigned       nperfiles,      // número de perfiles
   const bool           crear_tapas,    // true para crear tapas
   const bool           cerrar_malla    // true para cerrar la malla
 ) ;

 public:
 // constructor: crea una malla de revolución (lee PLY y llama a crearMallaRevol)

 MallaRevol
 ( const std::string & nombre_arch,    // nombre de archivo ply
   const unsigned      nperfiles,      // número de perfiles
   const bool          crear_tapas,    // true para crear tapas
   const bool          cerrar_malla    // true para cerrar la malla
 );
} ;
```

la declaración de esta nueva clase se puede hacer en su propio par de archivos fuente (`.hpp`/`.cpp`), o incorporarla a los archivos ya creados para las mallas.

2.5.3. Clases para: cilindro, cono y esfera

Para implementar estos objetos de revolución, crearemos clases derivadas de `MallaRevol`. Cada una de estas clases aporta un constructor especifico, que crea el correspondiente vector con el perfil original, y luego invoca al método `crearMallaRevol` para crear las tablas.

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

class Cilindro : public MallaRevol
{
 public:
 // Constructor: crea el perfil original y llama a crearMalla
 // 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 nperfiles,    // número de perfiles (N)
   const bool     crear_tapas,  // true para crear tapas
```

```
    const bool      cerrar_malla   // true para cerrar la malla
 ) ;
} ;

class Cono : public MallaRevol
{
 public:
 // Constructor: crea el perfil original y llama a crearMalla
 // 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 nperfiles,     // número de perfiles (N)
    const bool      crear_tapas,  // true para crear tapas
    const bool      cerrar_malla  // true para cerrar la malla
 ) ;
} ;

class Esfera : public MallaRevol
{
 public:
 // Constructor: crea el perfil original y llama a crearMalla
 // 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 nperfiles,     // número de perfiles (N)
    const bool      crear_tapas,  // true para crear tapas
    const bool      cerrar_malla  // true para cerrar la malla
 ) ;
} ;
```

2.6. Lectura de archivos PLY

Para leer los archivos PLY se proporcionan los archivos fuente `file_ply_stl.cpp/.hpp`, que se compilan junto con todos los demás (esto ya está incluido en el material proporcionado). La lectura se hace invocando las funciones `ply::read` (para la malla PLY, incluyendo vértices y caras), y `ply::read_vertices` (para el perfil del objeto de revolución, almacenado también en un archivo PLY, pero solo incluyendo los vértices).

Las funciones descritan leen las tablas como vectores de valores flotantes (vértices) y enteros (caras), exclusivamente de archivos PLY tipo ASCII (no binarios). Se debe implementar la construcción de las tablas de vértices y caras a partir de esos valores en los dos métodos constructores descritos arriba.

Para conocer los parámetros que tienen estas funciones y como se invocan, se puede consultar esta página:

- ☞ https://lsi.ugr.es/curena/varios/plys/

2.7. Archivos PLY disponibles.

En la carpeta `plys` se encuentran varios archivos PLY con mallas de polígonos. Estos archivos incluyen una lista de vértices (3 valores reales por vértice) y una lista de caras (3 enteros por vértice). Para crear el objeto obtenido de un archivo PLY, se puede usar uno de ellos o cualquier otro de los que se encuentran en internet con un formato similar. En la línea de comandos se puede usar un primer argumento con el nombre del archivo (si no se dan argumentos, se puede usar un o cualquiera de ellos). Los argumentos recibidos en **`main`** se pasan a **`P2_Inicializar`**, de forma que se pueda disponer de los nombres para cargar los PLYs.

Respecto a la construcción del objeto por revolución, se puede usar un archivo PLY que únicamente incluye la lista de vértices. Se puede construir manualmente, y además se puede probar con el que se proporciona (`peon.ply`), en la carpeta `plys`. El programa ejecutable puede aceptar un segundo parámetro en la línea de comandos con el nombre del archivo ply, si no se incluye dicho parámetros, se puede cargar este modelo (`peon.ply`)

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

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

2.8. Visualización en modo diferido

Para visualizar las mallas indexadas en modo diferido (es decir, para usar Vertex Buffer Objects, o VBOs), es necesario incluir en la clase **`ContextoVis`** un valor lógico que sea **`true`** para indicar que se debe usar modo diferido (VBOs) para visualizar dichas mallas, y **`false`** para indicar que se debe de usar el modo inmediato.

En **`main`**`.cpp`, en concreto en la función gestora de la pulsación de teclas normales, se debe gestionar la tecla `V`. Al pulsarla, el valor de la variable lógica citada arriba se cambia de **`true`** a **`false`** o al revés (y se imprime un mensaje indicando si se ha activado o desactivado el modo diferido).

Se debe añadir una nueva variable de instancia lógica (protegida) a la clase **`MallaInd`**. Esta variable indica si se han creado o no se han creado los VBOs correspondientes a esta instancia (inicialmente es **`false`**).

Al invocar a **`visualizarGL`** sobre un objeto **`MallaInd`** se debe comprobar en **`cv`** si está activado el modo diferido, y si no están todavía creados los VBOs de ese objeto. En ese caso se deben crear los VBOs necesarios, almacenar en la instancia los identificadores de VBO, y registrar que ya están creados los VBOs. A partir de entonces, cuando se quiera visualizar el objeto y esté activado el modo diferido, se visualizará la malla usando los VBOs que ya se han creado. Si no está activado el modo diferido, se visualiza en modo inmediato.

2.9. Instrucciones para subir los archivos

Se deben de seguir estas indicaciones:

- Hacer un zip con el nombre **`P2-fuentes.zip`**, con todos los fuentes de la carpeta **`alum-srcs`**, incluyendo **`main.cpp`** o cualquier otro, el **`zip`** debe hacerse directamente en **`alum-srcs`**, y no puede tener carpetas dentro de él, solo puede contener directamente los archivos indicados. Se debe incluir el archivo **`makefile`** con los nombres de la unidades de compilación específicas que el alumno haya usado para esta práctica. No incluir aquí archivos PLY.
- Si se han usado archivos PLY (distintos de los descargados en la plantilla de prácticas) hacer otro archivo ZIP, de nombre **`P2-archivos.zip`** con todos los archivos de cualquier tipo (PLY u otros) que el alumno haya usado y que no estén ya en las carpetas **`plys`** o **`imgs`** (descargadas de la web de la asignatura). Es decir, en **`alum-archs`** (y en el ZIP) se incluirán los archivos que el alumno haya buscado y usado por su cuenta, y de los cuales el profesor no dispone. Hacer un ZIP plano, sin carpetas dentro, solo los archivos directamente.
- Incluir un archivo de texto ascii y de nombre `leeme.txt`, en ese archivo, incluir:
  - Sistema operativo usado para compilar, y, si se ha usado algún entorno de desarrollo específico, indicarlo.
  - Todas las teclas que se pueden pulsar y utilidad de cada tecla.
  - Si se ha implementado o no la visualización en modo diferido (con VBOs).
  - Se se usa algún archivo PLY que no estuviera en la web de la asignatura. Indicar el nombre del archivo.
  - Si se ha implementado alguna funcionalidad no descrita en este guión o no. En caso afirmativo, incluir la descripción de dicha funcionalidad (p.ej.: que tipo de objetos se han implementado, donde está el código, que parámetros configurables tiene, etc...)

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

Para realizar un modelo jerárquico es importante la definición correcta de los grados de libertad o parámetros que presente el modelo.

Para modificar los parámetros asociados a los grados de libertad del modelo utilizaremos el teclado. Para ello tendremos que escribir código para modificar los parámetros como respuesta a la pulsación de teclas.

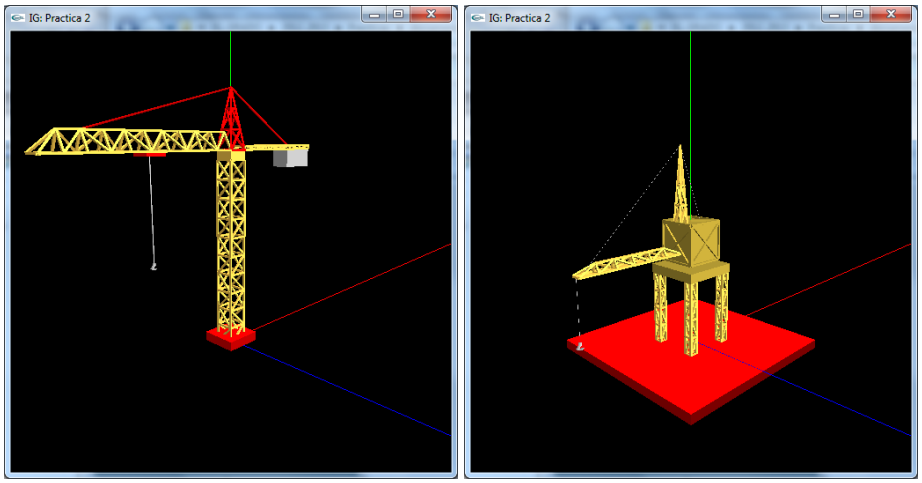


Figura 3.1: Ejemplos del resultado de la práctica 3.

Las acciones a realizar en esta práctica son:

1. Diseñar un modelo jerárquico con al menos 3 grados de libertad distintos (al menos deben aparecer giros y desplazamientos). Puedes tomar como ejemplo el diseño de una grúa semejante a las del ejemplo (ver figura 3.1). En el ejemplo, estas gruas tienen al menos tres grados de libertad: ángulo de giro de la torre, giro del brazo y altura del gancho.
2. Diseñar el grafo de escena (tipo PHIGS) correspondiente al objeto diseñado, determinando el tamaño de las piezas y las transformaciones geométricas a aplicar (tendrás que entregar el grafo del modelo en formato pdf cuando entregues la práctica).

3. Crear las estructuras de datos necesarias para almacenar el modelo jerárquico. El modelo debe contener la información necesaria para definir los parámetros asociados a la construcción del modelo (medidas de elementos, posicionamiento, etc.) y los parámetros que se vayan a modificar en tiempo de ejecución (grados de libertad, etc.). Hay que tener en cuenta que un modelo jerárquico esta formado por otros objetos, normalmente más sencillos, entre los que existen relaciones de dependencia hijo-padre, por lo que se deberá poder almacenar en la estructura de datos los distintos componentes que lo forman, así como las transformaciones que le afectan a cada uno con su tipo de transformación y sus parámetros.
4. Inicializar el modelo jerárquico diseñado para almacenar en la estructura de datos los componentes y parámetros necesarios para su construcción.
5. Crear el codigo necesario para visualizar el modelo jerárquico utilizando los valores de los parámetros del modelo almacenado en la estructura de datos. El alumno podra utilizar las funciones de visualización implementadas en las anteriores practicas que permiten visualizar los modelos con las distintas técnicas implementadas.
6. Incorporar código para cambiar los parámetros modificables del modelo y para controlar su movimiento. Añadir la ejecución de dicho código en las funciones de control de pulsación de teclas para modificar el modelo de forma interactiva. Hay que tener presente los límites de cada movimiento.
7. Incorporar código para poder realizar animaciones sencillas basadas en cambios repetidos de los parámetros.
8. Ejecutar el programa y comprobar que los movimientos son correctos.

3.2.1. Reutilización de elementos

En esta práctica para construir los modelos jerárquicos se deben utilizar otros elementos más sencillos que al combinarse mediante instanciación utilizando las transformaciones geométricas necesarias, nos permitirán construir modelos mucho más complejos. Se puede partir de cualquier primitiva que nos ofrezcan las propias librerías de OpenGL, o reutilizar los elementos implementados en las prácticas anteriores. En particular, será interesante poder definir objetos por revolución de un perfil, que puede estar definido por una tabla de puntos que aparece explicitamente en el código, en lugar de ser leido de un archivo ply.

En todos los casos, el código de visualización de los distintos tipos de objetos se incluirá en clases derivadas de `Objeto3D` o `MallaInd`, según corresponda.

3.3. Animación

Entre los objetivos de esta práctica está el realizar una animación sencilla del modelo. La animación se hará mediante la visualización repetida de la escena jerárquica parametrizada, de forma que en cada visualización se usa el mismo modelo jerárquico para producir una imagen o *cuadro* (*frame*). Sin embargo, en cada cuadro el valor de uno o varios parametros puede varíar respecto del anterior. La variación del valor de un parámetro animable entre dos cuadros sucesivos se llama *velocidad* del parámetro. En principio, en este texto se asume que cada parámetro es un valor real, y por tanto su velocidad también lo es. La animación de cada parámetro animable puede ser iniciada o parada por el usuario.

Para proceder a implementar esto:

- Añade al modelo lo que estimes necesario para almacenar una velocidad de movimiento para cada uno de los parámetros.

- Añade opciones en el control de teclas para fijar la velocidad de cada parámetro a un valor positivo, negativo o cero (consulta la sección sobre las teclas usar)
- Ahora puedes animar el modelo haciendo que el valor del parámetro que modifica cada grado de libertad se modifique segun su velocidad. Debes asegurarte que la función de redibujado se invoque repetidas veces de forma continua sin que haya intervención del usuario. Para ello, puedes usar la *función gestora del evento de desocupado* de GLUT, como se describe aquí abajo.

3.3.1. Gestión de los grados de libertad y sus velocidades

En el modelo jerárquico se guardan los valores actuales de los grados de libertad del modelo (cada grado de libertad viene descrito con una cadena de caracteres, que llamamos su *descripción*).

En la práctica 3 habrá una variable lógica que determinará si las animaciones están activadas o no lo están. El usuario puede activar o desactivar las animaciones (ver las teclas más abajo).

Cada grado de libertad o parámetro tiene asociado en cada instante de tiempo un valor real p, que llamaremos *valor base* actual del parámetro. El valor base es inicialmente 0 en todos los parámetros. Adicionalmente, cada parámetro tiene asociada una velocidad v (mayor que 0), que se especifica en su constructor, y que se usa para las animaciones. La velocidad tambien puede cambiar a lo largo del tiempo. Inicialmente, tiene un valor v_0, que es una constante igual para todos los parámetros.

Cuando las animaciones están desactivadas, el usuario puede variar interactivamente el valor base de cada parámetro. Para ello se usa un valor positivo de incremento $\Delta > 0$ (una constante igual para todos los parámetros). El parámetro activo puede incrementarse o decrementarse por teclado usando ese valor Δ, es decir, podemos hacer $p = p + \Delta$ o bien $p = p - \Delta$.

Cuando las animaciones están activadas, los valores base de todos los parámetros cambian entre cuadros, según la velocidad actual v de cada uno de ellos, es decir, hacemos $p = p + v$. Todos los parámetros cambian entre dos cuadros, es decir, al activar las animaciones todos los movimientos ocurren a la vez.

La velocidad actual v puede ser variada por el usuario (ver teclas). De esta forma, un parámetro puede acelerarse o decelerarse. Cada vez que se varía la velocidad, se incrementa o se decrementa usando un valor a (positivo) que llamamos *aceleración*, y que de nuevo es una constante igual para todos los parámetros. Cuando el usuario pulsa las teclas adecuadas, la velocidad de un parámetro puede cambiarse, haciendo $v = v + a$ (acelerar) o bien $v = v - a$ (decelerar). En el caso de la deceleración, si se produce un valor negativo de v, se debe truncar a 0, ya que la velocidad no puede ser negativa.

Hay dos tipos de parámetros: unos de ellos están no acotados y otros sí lo están. Los parámetros acotados crecen o decrecen indefinidamente, mientras que los acotados oscilan entre dos valores de forma periódica. Cada parámetro tiene asociados tres valores reales c, s y f (donde $f > 0$). Esos valores determinan el *valor actual* q del parámetro en función de su valor base actual p, en concreto, definimos q como sigue:

$$q \;=\; \begin{cases} c + s \cdot p & \text{si el parámetro no está acotado} \\ c + s \cdot \sin(f 2\pi p) & \text{si el parámetro está acotado} \end{cases}$$

de forma que c es el valor inicial de q (ya que siempre p es 0 al inicio). Para los parámetros acotados, c es también el valor central, s la semiamplitud, y f el número de ciclos que hace q por cada unidad que crece p (llamamos *frecuencia* a f). Vemos que q toma valores entre $c + s$ y $c - s$, según p crece. Para los parámetros no acotados, c es simplemente el valor inicial y s un factor de escala.

Cada parámetro tiene asociada una función F que produce una matriz de transformación a partir de un valor real. Dicha función F es usada para variar una de las matrices (M) del modelo jerárquico, de forma que cada vez que q varía, recalculamos la matriz (hacemos $M = F(q)$). Para poder actualizar la matriz del modelo, asociado a cada parámetro hay que tener un puntero a dicha matriz en la estructura de datos.

Recapitulando, la gestión de los parámetros esta determinada por estos valores:

- Constantes, iguales para todos los parámetros: v_0, Δ, a
- Constantes, distintas para cada parámetro: c, s, f, F (y el puntero a la matriz).
- Variables en el tiempo, distintas en cada parámetro: p, q, v

3.3.2. Animación mediante la función desocupado.

Se debe de implementar la visualización repetida y continua del modelo, sin que el usuario tenga para ello que provocar manualmente eventos de teclado en cada cuadro. Para ello, se puede designar una función que se invocará cuando no haya eventos de entrada que procesar y no sea necesario redibujar la ventana (es decir, cuando el programa queda desocupado). A dicha función la llamamos *función desocupado*.

Para implementar dicha funcionalidad en GLFW, es necesario tener esto en cuenta en el bucle principal de gestión de eventos de la librería. En concreto, si hay designada una función desocupado, en dicho bucle se debe invocar `glfwPollEvents` en lugar de `glfwWaitEvents`, y también se debe invocar dicha función desocupado. Al contrario que `glfwWaitEvents`, la función `glfwPollEvents` no espera a que se produzca un evento si al llamarla no hay ninguno pendiente de procesar. En la sección de implementación se detalla esto.

Dentro de esa función desocupado, la aplicación puede incluir código arbitrario para modificar el modelo y forzar después que se visualice el siguiente cuadro.

3.4. Teclas a usar

Respecto a las teclas específicas de la práctica 3, se usarán las que se indican aquí . Como se indica en el guión, en esta práctica se deben implementar varios grados de libertad en el modelo (como mínimo tres). Para gestionar los valores de los parámetros asociados, se definirá una variable entera local de la práctica con el grado de libertad actual. La variable tomará valores entre 0 y $n-1$, ambos incluidos, donde n es el número de grados de libertad. Las teclas son:

- Tecla **g/G**: activar grado de libertad siguiente al actual (o activar el primero si el actual era el último)
- Tecla **a/A**: activar o desactivar animación en la práctica (habrá una variable lógica en la práctica 3 que indicará si las animaciones están activadas o no). Se imprimirá un mensaje indicando si las animaciones quedan activadas o desactivadas.
- Tecla **r/R**: (reset) reinicializar todos los parámetros del modelo, desactivar animaciones.

Adicionalmente, se pueden usar las teclas $>$ y $<$. Su funcionamiento depende de si las animaciones están activadas o no lo están.

- Con las animaciones **desactivadas**:
  - Tecla $>$: incrementar el valor del grado de libertad actual
  - Tecla $<$: decrementar el valor del grado de libertad actual

Tras la pulsación de estas teclas, el programa debe imprimir el nombre y el valor del grado de libertad actual.

- Con las animaciones **activadas**:
  - Tecla >: aumentar la velocidad del grado de libertad actual (es decir, acelerarlo).
  - Tecla <: disminuir la velocidad del grado de libertad actual (es decir: decelerarlo).

  Tras la pulsación de estas teclas, el programa debe imprimir el nombre y la velocidad del grado de libertad actual.

Teclas generales

Al igual que ya se ha indicado en otras prácticas (independientemente de otras teclas que se usen para otras cosas) se deben usar estas teclas:

- tecla **m/M**: cambia el modo de visualización activo (pasa al siguiente, o del último al primero)
- tecla p/P: cambia la práctica activa (pasa a la siguiente, o de la última a la primera).
- tecla **o/O**: cambia el objeto activo dentro de la práctica (pasa al siguiente, o del último al primero).

Las dos primeras se implementan en **`main.cpp`**, la tercera en la función gestora del evento de teclado específica de la práctica 3.

3.5. Implementación

La implementación de esta práctica requiere completar el código de dos archivos fuente C++ de nombres **`practica3.cpp`** y **`practica3.hpp`**, en la carpeta **`alum-srcs`**. El primero contendrá la implementación o definición de las funciones y el segundo las declaraciones de las mismas, al igual que en la prácticas 1 y 2.

En el archivo **`main.cpp`** se gestiona la variable que indica cual es la práctica activa (variable **`practicaActual`**), de forma que mediante alguna tecla se puede conmutar entre las distintas prácticas (cambiar el valor de la variable). El procesamiento de la tecla ya está incluido **`main.cpp`**. Inicialmente, la práctica activa será la 3, aunque por supuesto podrá cambiarse si el usuario quiere.

La implementación requiere escribir las siguientes funciones (en **`practica3.cpp`**):

Función **`void P3_Inicializar( )`**

Sirve para crear los objetos que se requieren para la práctica. Esta función se invoca desde **`main.cpp`** una única vez al inicio del programa, inmediatamente después de la llamada ya existente a **`P2_Inicializar`**, para crear los objetos. No tiene parámetros.

Se crearán en memoria dinámica el objeto jerárquico a visualizar, y se guardará en una variable de la práctica un puntero al nodo raíz de dicho objeto (para evitar colisiones de nombres, se aconseja declarar este puntero como static, y también se aconseja que esten inicializado a **`nullptr`** en su declaración). El nodo raíz será de la clase **`NodoGrafoEscenaParam`**.

Función **`void P3_DibujarObjetos( ContextoVis & cv )`**

Sirve para dibujar el objeto jerárquico usando los punteros a los objetos descritos arriba, y usando también el parámetro modo para determinar el tipo o modo de visualización de primitivas (con la misma interpretación que en la prácticas 1 y 2). Esta función se invoca desde **`main.cpp`** cada vez

que se recibe el evento de redibujado, y la práctica activa es la práctica 3. Para ello, se debe insertar la nueva llamada en **main.cpp** (función **VisualizarFrame**), en base a la práctica activa en cada momento (que ahora puede ser la 1, la 2 o la 3).

Función `bool P3_FGE_PulsarTeclaNormal( unsigned char tecla )`

Está función se invoca desde **main.cpp** cuando se pulsa una tecla normal, la práctica 3 está activa, y la tecla no es procesada en el **main.cpp**. La función sirve para cambiar gestionar las teclas específicas de la práctica, que son las teclas A,G,R,O,> y <. Debe devolver true para indicar que la tecla pulsada corresponde a la práctica 3 y se debe redibujar el objeto (típicamente, porque se ha modificado un valor de un grado de libertad), y false para lo contrario (la tecla no corresponde a la práctica y no es necesario redibujar).

Para gestionar cual es el grado de libertad activo en cada momento, se puede usar una variable global estática en **practica3.cpp** llamada `p3_grado_libertad_activo`. El número de grados de libertad se puede obtener del nodo raíz del grafo de escena (ver más abajo la clase **NodoGrafoEscenaParam**)

3.5.1. Implementación de objetos jerárquicos parametrizados

En la plantilla se encuentran las siguientes clases (su código debe completarse):

- Clase **Parametro**, contiene los atributos de un parámetro o grado de libertad.
- Clase **NodoGrafoEscena**, para los nodos de los grafos de escena, tal y como se ha descrito en teoría.
- Clase **NodoGrafoEscenaParam**, es una clase derivada de la anterior, que añade un vector de parámetros.
- Una clase C derivada de **NodoGrafoEscenaParam**, para el objeto que es el nodo raíz de nuestro modelo.

Tal y como se ha visto en los ejemplos de teoría, el constructor de C debe crear el grafo de escena completo. Asimismo, debe poblar el vector de parámetros. A continuación se detallan las tres clases

Parámetros o grados de libertad

Cada parámetro o grado de libertad de nuestro modelo tiene asociado una instancia de la clase **Parametro**. Esa instancia guarda todos los valores o atributos del parámetro, junto con las operaciones para modificar su valor o velocidad. Se declara como sigue:

```
class Parametro
{
  private:
  const std::string
    descripcion ;  // descripción del grado de libertad (para seguimiento)
  const bool
    acotado ;      // true si el valor oscila entre dos valores, false si no
  const TFuncionCMF
    fun_calculo_matriz; // función que produce una nueva matriz a partir de un valor flotante
  const float
    c,   // valor inicial (y también central para grados acotados)
    s,   // semiamplitud (si acotado), o factor de escala (si no acotado)
    f;   // si acotado: frecuencia (ciclos por unidad del valor normalizado)
```

```
  Matriz4f * const
    ptr_mat ; // puntero a la matriz dentro del modelo
  float
    valor_norm, // valor actual, normalizado, no acotado (crece desde 0)
    velocidad ; // velocidad actual (se suma al valor_norm)

   public:
   // crear grado de libertad no acotado
   Parametro( const std::string & p_descripcion, // descripción
              Matriz4f *  p_ptr_mat,      // puntero a la matriz en modelo
              TFuncionCMF p_fcm,          // funcion de calculo de matriz
              bool        p_acotado,      // true si acotado, false sino
              float       p_c, float p_s, float p_f ); // ctes: c,s,f

   void  siguiente_cuadro(); // actualizar valor y matriz al siguiente frame
   void  reset();            // vuelve al estado inicial
   void  incrementar();      // incrementar el valor
   void  decrementar() ;     // decrementar el valor
   void  acelerar();         // acelerar (aumentar velocidad)
   void  decelerar();        // decelerar (disminuir la velocidad)
   float leer_valor_actual(); // devuelve el valor actual (valor de q)
   float leer_velocidad_actual(); // devuelve velocidad actual
   std::string leer_descripcion(); // devuelve descripción
};
```

Los métodos `reset`,`incrementar` o `decrementar` modifican el valor p del parámetro, y por tanto deben de recalcular la matriz y actualizarla en el modelo a través del puntero.

El tipo de datos `TFuncionCMF` sirve para las funciones de cálculo de una matriz a partir de un flotante, y puede declararse usando la plantilla `std::function` de C++11, como sigue:

```
#include <functional> // incluye std::function
....
// tipo para las funciones que calculan una Matriz4f a partir de un float
typedef std::function< Matriz4f( float )> TFuncionCMF  ;
```

Para crear un parámetro usamos, lógicamente, su constructor. El parámetro correspondiente a la función de cálculo de la matriz puede especificarse escribiendo directamente dicha función en la lista de parámetros actuales. A modo de ejemplo, vemos un parámetro que determina una rotación entorno al eje X, y que depende de un valor real que es un ángulo en grados, que a su vez oscila entre $30-25$ y $30+25$ grados:

```
Matriz4f * ptr_mat = leerPtrMatriz(...) ; // obtener puntero a la matriz en el grafo

Parametro param
  ( "nombre descriptivo del parámetro", // descripcion
    ptr_mat, // puntero a la matriz
    [=](float v) {return MAT_Rotacion( v, 1.0,0.0,0.0 );}, // f.calc.matr.
    true,     // sí acotado (es oscilante)
    30.0,     // valor central (inicial)
    25.0,     // semiamplitud
    4.0       // frecuencia
  );
```

Obtención de punteros a las matrices.

Hay que tener en cuenta que los métodos **agregar** de un nodo devuelven todos ellos un entero que es el índice de la entrada que se ha añadido al grafo de escena (índice en el vector de entradas del nodo). En los casos en los que se ha añadido un entrada de tipo matriz de transformación, ese índice puede usarse después para obtener el correspondiente puntero. Para ello se necesario implementar el método **leerPtrMatriz** de la clase **NodoGrafoEscena** (ya está la declaración, y la definición vacía). Este método debe comprobar que la entrada indicada es de tipo matriz, y que el puntero no es nulo. En caso de pasar esta comprobación, devuelve el puntero.

De esta forma, cuando queramos que un parámetro sirva para modificar una matriz del grafo, debemos de guardar el índice de esa matriz en su nodo, y usar ese índice para recuperar el puntero.

Clase para los nodos de los grafos de escena

Cada nodo del grafo de escena (excepto el nodo raiz) puede ser un objeto de la clase **NodoGrafoEscena**, o de alguna clase específica derivada de esta clase. Como se ha visto en teoría, esta clase sirve para nodos de grafos tipo PHIGS, y cada instancia tiene una lista de entradas con punteros a sub-árboles o matrices de transformación.

En esta prácticas, las partes del modelo tienen asociadas instancias de estos nodos. Los nodos se alojan en memoria dinámica (se crean con **new**). Se pueden crear vacíos y después poblar llamando a los métodos **agregar**.

Los métodos **agregar** devuelven el número de celda del vector de entradas donde se ha alojado el puntero o la transformación. Asimismo, se incorporará un método para obtener el puntero a una matriz de transformación (**Matriz4f**) alojada en una entrada del vector, dando su índice.

Algunas partes complejas del modelo pueden implementarse definiendo clases específicas derivadas de **NodoGrafoEscena**.

Clase para un nodo raiz parametrizado

El nodo raiz de nuestro grafo de escena es especial, ya que queremos alojar en dicho nodo un vector de parámetros, que describen los estado y atributos de los parámetros de nuestro objeto parametrizado. Por tanto, definimos una nueva clase, llamada **NodoGrafoEscenaParam**, derivada de **NodoGrafoEscena**. Puede tener esta estructura:

```
class NodoGrafoEscenaParam : public NodoGrafoEscena
{
  protected:
  std::vector<Parametro> parametros ; // vector de parámetros

  public:
  // devuelve el número de parámetros
  int numParametros();
  // devuelve puntero al i-ésimo parámetro (i <numParametros())
  Parametro * leerPtrParametro( unsigned i ) ;
  // actualiza el objeto para ir al siguiente cuadro (con animaciones)
  void siguienteCuadro();
} ;
```

Clase para el objeto de la práctica 3

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 **`NodoGrafoEscenaParam`**. El nodo raíz de nuestro grafo de escena será un objeto de la clase C, y por tanto contiene todos los parámetros del modelo. Lo único que en principio debe de añadirse a C es un constructor. En dicho constructor se crean los subárboles del nodo raiz y se van añadiendo a la lista de entradas.

Adicionalmente, es necesario poblar el vector de parámetros. Para ello se deben de construir los distintos parámetros que forman el mismo. Al crear los nodos del modelo, se deben de obtener los punteros a las matrices asociadas a los grados de libertad, y esos punteros deben de usarse para construir los objetos de tipo **`Parametro`**.

3.5.2. Implementación de la animación

En **`main.cpp`** (más concretamente en la función **`BucleEventosGLFW`**) el programa ejecuta el bucle principal de gestión de eventos de entrada. En cada iteración se espera a que se produzca un evento, se llama a la función gestora correspondiente, y si es necesario se redibuja la escena. Este es el esquema que se sigue cuando las animaciones están desactivadas, es decir, en las prácticas 1 y 2.

Sin embargo, si deseamos que se produzcan animaciones, entonces el bucle de gestión de eventos no puede quedar bloqueado esperando a que se produzca un evento de entrada (es decir, no debe llamar a **`glfwWaitEvents`**), sino que debe procesar los eventos pendientes (si hay alguno) llamando a **`glfwPollEvents`**, e inmediatamente después, si no es necesario redibujar la ventana, llamar a la función **`FGE_Desocupado`**, que se encuentra implementada en **`main.cpp`**.

Cuando se pulsa la tecla **P** y se activa la práctica 3, se hace la siguiente llamada:

```
FijarFuncDesocupado( FGE_Desocupado ) ;
```

Con esta llamada indicamos que cuando el programa no tenga eventos pendientes, debe invocar a **`FGE_Desocupado`**, y el bucle principal de gestión de eventos no queda bloqueado a la espera del siguiente.

La función **`FGE_Desocupado`** está declara en **`main.cpp`**, y se encarga (cuando la práctica activa es la 3), de invocar a la función **`P3_FGE_Desocupado`**. Tiene este código:

```
void FGE_Desocupado()
{
   bool desactivar = true ;
   if ( practicaActual == 3 )
      desactivar = ! P3_FGE_Desocupado();
   if ( desactivar )
      FijarFuncDesocupado( nullptr ); // desactivar para no saturar la CPU innecesariamente.
}
```

Como vemos, esta función llama a **`P3_FGE_Desocupado`** solo cuando es necesario. Cuando la práctica 3 no está activada, se desactiva la función desocupado (se usa el puntero nulo, **`nullptr`**), ya que aunque introduce un retraso muy corto, las repetidas llamadas mantendrían muy ocupada la CPU de forma innecesaria.

La función **`P3_FGE_Desocupado`** esta declarada en **`practica3.hpp`** y definida en **`practica3.cpp`**. Devuelve **`false`** cuando no es necesario invocarla (ver más abajo los detalles de esto). Como vemos,

si está función devuelve **false**, se desactivan las nimaciones, y el bucle de gestión de eventos quedará de nuevo bloqueado esperando eventos nuevos cuando no hay ninguno pendiente de procesar.

Hay que tener en cuenta que la visualización con OpenGL debe hacerse exclusivamente en la función **DibujarObjetos**, por tanto en la función de desocupado no se debe visualizar la escena, sino provocar que se haga después, lo cual se consigue poniendo **redibujar_ventana** a **true**.

Por todo lo dicho, debemos definir **P3_FGE_Desocupado** como sigue:

```
// Func. de gestión del evento de desocupado de la práctica 3
// debe devolver: false: si queremos que se desactive el evento
// true: si queremos que el evento permanezca activado

bool P3_FGE_Desocupado( )
{
   // no hacer nada si no es necesario, y desactivar
   if ( las animaciones están desactivadas )
      return false;    // provoca que bucle principal espere nuevos eventos

   // modificar los parámetros animables según sus velocidades actuales
   .......

   // forzar llamada a VisualizarFrame en la próxima iteración del bucle
   redibujar_ventana = true ;

   // terminar, manteniendo activada la gestión del evento
   return true ;
}
```

3.5.3. Instrucciones para subir los archivos

Para entregar la práctica se creará y se subirá un único archivo **.zip**, de nombre **P3.zip**, siguiendo estas indicaciones:

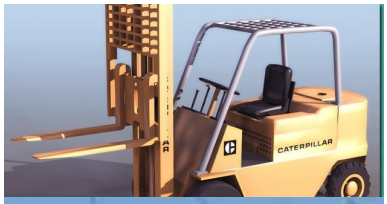
- Se redactará un único documento (que se entregará en PDF) con esta información:
  - Grafo de escena, en formato PHIGS (según lo visto en teoría). Cada nodo se etiquetará usando el nombre de la clase derivada de **NodoGrafoEscena** que lo implementa.
  - Lista de grados de libertad del modelo: para cada grado de libertad se describirá el tipo de nodo (el nombre de la clase), el nodo del grafo al que afecta, y el tipo de transformación asociada. También se incluirán los atributos del parámetro: valor inicial, velocidad inicial, incremento, aceleración, descripción, minimo o máximo, etc....
- Hacer un zip (de nombre **P3-fuentes.zip** con todos los fuentes de la carpeta **alum-srcs**, incluyendo **main.cpp** o cualquier otro, el zip debe hacerse directamente en **alum-srcs**, y no puede tener carpetas dentro de él, solo puede contener directamente los archivos indicados. El zip también contendrá el archivo PDF con la documentación descrita arriba.
  No incluir en el zip archivos **.ply** ni **.jpg** (ya proporcionados), ni los **.o** ni el ejecutable, ni **include.make**. Se incluirán los archivos de código fuente, el archivo **makefile** (con la unidades de compilación).
- Si se han usado archivos PLY (distintos de los descargados en la plantilla de prácticas) hacer otro archivo ZIP, de nombre **P3-archivos.zip** con todos los archivos de cualquier tipo (PLY u otros) que el alumno haya usado y que no estén ya en las carpetas **plys** o **imgs** (descargadas de la web de la asignatura). Es decir, en **alum-archs** (y en el ZIP) se incluirán los archivos que el alumno haya buscado y usado por su cuenta, y de los cuales el profesor no

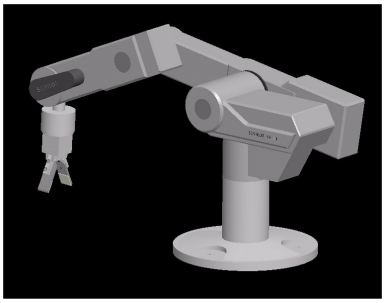
dispone. Hacer un ZIP plano, sin carpetas dentro, solo los archivos directamente.

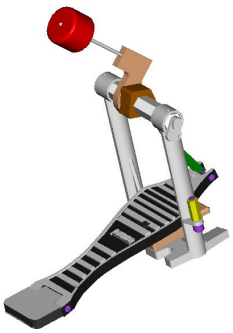
3.6. Algunos ejemplos de modelos jerárquicos

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.


CATERPILLAR





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, incluyendo las nomales de las mallas y modelos sencillos de fuentes de luz, materiales y texturas.
- Visualizar, usando la funcionalidad fija de OpenGL, escenas incluyendo varios objetos con distintos modelos de aspecto, usando tanto sombreado plano (*flat shading*) como sombreado suave (*Gourdoud shading*).
- Permitir cambiar de forma interactiva los parámetros de los modelos de aspecto de la escena anterior.

4.2. Desarrollo

En esta práctica incorporaremos a las mallas de triángulos información de las normales y las coordenadas de textura, así como crearemos estructuras de datos para representar modelos de fuentes de luz, materiales y las texturas, posibilitando la visualización de objetos con distintos modelos de aspecto. El código descrito aquí debe añadirse al código existente para las prácticas desde la 1 hasta la 3, sin eliminar nada de la funcionalidad ya implementada.

Se definirá un grafo de escena con varios objetos y materiales. La geometría de esta escena se describe en este guión. Se añadirán materiales y texturas al grafo de escena de la práctica 3, en este guión se especifica el aspecto de dichos materiales, pero no los valores de los parámetros del material que producen ese aspecto.

Se crearán el código para dos nuevos modos de visualización (*modos con iluminación*), adicionales a los que se indican en el guión de la práctica 1. En ambos modos (y para las mallas que dispongan de ellas) se enviarán junto con cada vértice sus coordenadas de textura, pero no se enviarán los colores de los vértices ni de los triángulos. Respecto a las normales, se deben usar las tablas calculadas con el mismo código de las prácticas anteriores. Se procede según el modo:

- Modo con iluminación y sombreado plano: se activa el modo de sombreado plano de OpenGL, y se envían los triángulos, usando la tabla de normales de triángulos. Para esto es necesario visualizar en modo inmediato usando las órdenes `glBegin`/`glVertex`/`glEnd`, ya que es la única forma de poder asignar una normal (o cualquier otro atributo) a un triángulo (para ello tenemos que enviar cada vértice tantas veces como triángulos adyacentes tenga, cada vez con la normal correspondiente a un triángulo).
- Modo con iluminación y sombreado de suave (Gouroud): se activa el modo de sombreado suave, y se envían los vértices usando la tabla de normales de vértices.

En el método para visualizar mallas indexadas, cuando el modo de visualización es alguno de estos dos, se supone que el material correspondiente ya está activado antes de la llamada al método. Igual pasa con las fuentes de luz, las suponemos activadas. Por tanto, en estos modos, la visualización de una malla simplemente supone enviar los vértices y/o caras. En los otros modos, distintos de estos dos, no hay que modificar nada. En cualquier caso, al pulsar la tecla **m/M** se debe rotar entre todos los modos.

Tras incorporar la práctica 4, hay que revisar el código que visualiza en los modos sólido, alambre y ajderez, ya que ahora debemos de estar seguros de que para visualzar en esos modos se desactivan la iluminación y las texturas.

4.2.1. Cálculo y almacenamiento de normales.

A partir de la tabla de coordenadas de vértices y la tabla de caras de una malla, se deben calcular las dos tablas de normales (normales de caras y normales de vértices). Cada tabla de normales es un array que en cada entrada contiene una tupla de 3 reales que representa (en coordenadas maestras o de objeto) el vector perpendicular a la cara o a la superficie de la malla en el vértice. Las dos tablas de normales quedarán almacenadas en la estructura o instancia de clase que representa la malla, junto con el resto de tablas (coordenadas de vértices, de textura, etc...).

Para el cálculo de las tablas de normales se puede seguir este procedimiento:

1. En primer lugar se calcula tabla de normales de las caras. Para ello se debe de recorrer la tabla caras que hay en la malla. En cada cara se consideran las posiciones de sus tres vértices, sean estas, por ejemplo $\vec{p}$,$\vec{q}$ y $\vec{r}$. A partir de estas coordenadas se calculan los vectores $\vec{a}$ y $\vec{b}$ correspondientes a dos aristas, haciendo $\vec{a} = \vec{q} - \vec{p}$ y $\vec{b} = \vec{r} - \vec{p}$. El vector $\vec{m}_c$, perpendicular a la cara, se obtiene como el producto vectorial de las aristas, es decir, hacemos: $\vec{m}_c = \vec{a} \times \vec{b}$. Finalmente, el vector normal $\vec{n}_c$ (de longitud unidad) se obtiene normalizando $\vec{m}_c$, esto es: $\vec{n}_c = \vec{m}_c / \|\vec{m}_c\|$.
   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 $\vec{p}$,$\vec{q}$ y $\vec{r}$ 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 $\vec{m}_c = \vec{b} \times \vec{a}$ en lugar del originalmente descrito, ya que el producto vectorial es anticonmutativo.
2. Una vez calculadas las normales de caras, se obtienen las normales de los vértices. Para cada vértice, el vector $\vec{m}_v$, 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:

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

   donde $\vec{u}_i$ 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 $\vec{n}_v$ se define como una versión normalizada de $\vec{m}_v$, es decir: $\vec{n}_v = \vec{m}_v / \|\vec{m}_v\|$.
   Una implementación básica (derivada directamente de esta definición de $\vec{n}_v$) 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, método basado en recorrer las caras en el bucle externo, en lugar de los vértices, y en eliminar el bucle interno. Esto es posible debido a que obtener los

vértices de una cara se puede hacer de forma inmediata (en $O(1)$) sin más que consultar la entrada correspondiente de la tabla de caras.

4.2.2. Almacenamiento y visualización de coordenadas de textura

El siguiente paso será aumentar las estructuras de datos que representan las mallas en memoria y extender el código de visualización para calcular y visualizar las coordenas de textura.

Para ello, se añadirá a las mallas la tabla de coordenadas de textura, que será un array con n_v pares de valores valores de tipo `GLfloat`, donde n_v es el número de vértices. Esta tablas se usarán para asociar coordenadas de textura a cada vértice en algunas mallas. En las mallas que no tengan o no necesiten coordenadas de textura, dicha tabla estará vacía (0 elementos) o será un puntero nulo.

4.2.3. Asignación de coordenadas de textura en objetos obtenidos por revolución.

Una vez definidas las tablas de coordenadas de textura, se creará una nueva versión modificada del código o función que crea **objetos por revolución** de la práctica 3. En los casos que así se especifique, dicho código nuevo incluirá la creación de la tabla de coordenadas de textura.

Supongamos que el el perfil de partida tiene M vértices, numerados comenzando en cero. Las posiciones de dichos vértices serán: $\{p_0, p_1, \ldots, p_{M-1}\}$. Si suponemos que hay N copias del perfil, y que en cada copia hay M vértices, entonces el j-ésimo vértice en la i-ésima copia del perfil será $q_{i,j}$, con $i \in [0 \ldots N-1]$ y $j \in [0 \ldots M-1]$ y tendrá unas coordenadas de textura (s_i, t_j) (dos valores reales entre 0 y 1. La coordenada S (coordenada X en el espacio de la textura) es común a todos los vértices en una copia del perfil.

El valor de s_i es la coordenada X en el espacio de la textura, y está entre 0 y 1. Se obtiene como un valor proporcional a i, haciendo $s_i = i/(N-1)$ (la división es real), de forma que s_i va desde 0 en el primer perfil hasta 1 en el último de ellos.

El valor de t_j es la coordenada Y en el espacio de la textura, y también está entre 0 y 1. Su valor es proporcional a la distancia d_j (medida a lo largo del perfil), entre el primer vértice del mismo (vértice 0), y dicho vértice j -ésimo. Las distancia se definen como sigue: $d_0 = 0$ y $d_{j+1} = d_j + \|p_{j+1} - p_j\|$, y se pueden calcular y almacenar en un vector temporal durante la creación de la malla. Conocidas las distancias, la coordenada Y de textura (t_j) se obtiene como $t_j = d_j/d_{M-1}$

Hay que tener en cuenta que en este caso, la última copia del perfil (la copia $N-1$) es la que corresponde a una rotación de 2π radianes o 360^o. Esta copia debe ser distinta de la primera copia, ya que, si bien sus vértices coinciden con aquella, las coordenadas de textura no lo hacen (en concreto la coordenada S o X), debido a que en la primera copia necesariamente dicha coordenada es $s_0 = 0$ y en la última es $s_{N-1} = 1$. Este es un ejemplo de duplicación de vértices debido a las coordenadas de textura.

4.2.4. Fuentes de luz

El siguiente paso será diseñar e implementar las **fuentes de luz** de la escena (al menos dos). Al menos una de las dos fuentes será direccional y tendrá su vector de dirección definido en coordenadas esféricas por dos ángulos α y β , donde β es el ángulo de rotación en torno al eje X (latitud), y α es la rotación en torno al eje Y (longitud). Cuando α y β son ambas 0, la dirección coincide con la rama positiva del eje Z. el vector de dirección de la luz se considerará fijado al sistema de referencia de la camara (la luz se "mueveçon el observador).

Para implementar las fuentes se definen en el programa la instancias de clase que almacenan los parámetros de cada una de las fuentes de luz que se planean usar. Para cada fuente de luz, se debe guardar: su color S_i (una terna RGB), su tipo (un valor lógico que indique si es direccional o posicional), su posición (para las fuentes posicionales), y su dirección en coordenadas esféricas (para las direccionales). Al menos para la fuente dada en coordenadas esféricas, se guardarán asimismo los valores α y β.

Habrá método de la clase para las fuentes de luz que se encarge de activar una instancia. Aquí *activar* una fuente significa que en OpenGL se habilite su uso y que se configuran sus parámetros en función del valor actual de las variables de instancia que describen dicha fuente.

4.2.5. Carga, almacenamiento y visualización de texturas.

A continuación se diseñarán e implementarán las **texturas** de la escena. Se incluirán en el programa las instancias de clase que almacenan los parámetros de cada una de las texturas que se planean usar. Para cada textura, se debe guardar un puntero a los pixels en memoría dinámica, el identificador de textura de OpenGL, un valor lógico que indique si hay generación automática de coordenadas de textura o se usa la tabla de coordenadas de textura, y finalmente los 8 parámetros (dos arrays de 4 flotantes cada uno) para la generación automática de texturas.

Después se definirá en el programa el método para *activar* una textura. *Activar* aquí significa habilitar las texturas en OpenGL, habilitar el identificador de textura, y si la textura tiene asociada generación automática de coordenadas, fijar los parámetros OpenGL para dicha generación.

Hay que tener en cuenta que, la primera vez que se intente activar una textura, se debe *crear* la textura, esto significa que se deben leer los texels de un archivo y enviarlos a la GPU o la memoria de vídeo, inicializando el identificador de textura de OpenGL. Es importante no repetir la creación de la textura una vez en cada cuadro, sino hacerlo exclusivamente la primera vez que se intenta activar.

4.2.6. Materiales.

El siguiente paso será diseñar e implementar los **materiales** de la escena, entendiendo como un *material* a un conjunto de valores de los parámetros del modelo de iluminación de OpenGL relativos a la reflectividad y brillo de la superficie de los objetos.

Será necesario definir en las instancias de la clase `Material` que almacenan los parámetros de cada uno de los materiales que se planean usar. Para cada material, se almacenará la reflectividad ambiental (M_A) la difusa (M_D), la especular (M_S) y el exponente de brillo (e). Cada reflectividad es un array con 4 valores `float`: las tres componentes RGB más la cuarta componente (opacidad) puesta a 1 (opaco). Asimismo, un material puede llevar asociada una textura o no llevarla. Si la lleva, junto con el material se almacenan los parámetros de dicha textura (un puntero a ellos), descritos más arriba en el documento. Si el material no lleva textura, el citado puntero será nulo.

Después se definirán funciones o métodos para activar cada uno de los materiales. Cuando se activa un material, se habilita la iluminación en OpenGL, y se configuran los parámetros de material de OpenGL usando las variables o instancias descritas en el párrafo anterior. Para los materiales que lleven asociada una textura, se llamará a la función o método para activar dicha textura, descrita anteriormente. Si el material no lleva textura, se deben deshabilitar las texturas en OpenGL.

4.2.7. Grafo de la escena completa.

Figura 4.1: Vista de la escena completa (en modo materiales suave o modo *Gouroud*)

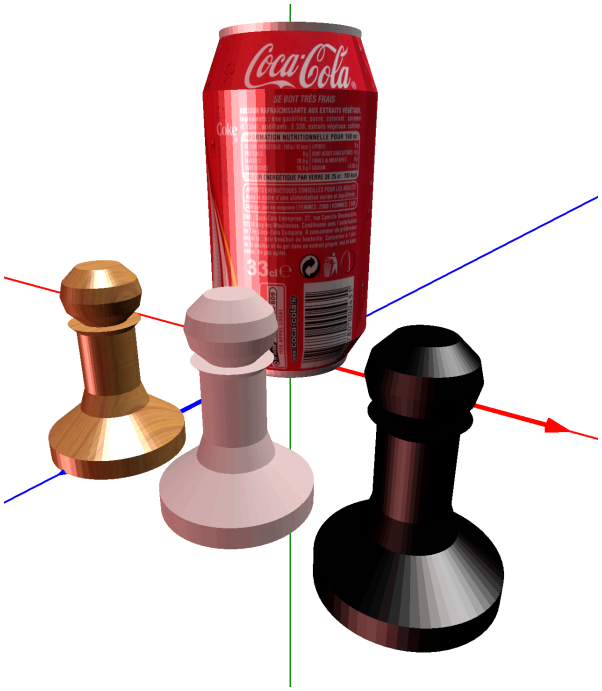


Figura 4.2: Vista de la escena completa (en modo materiales plano)

El último paso será definir la **función principal de visualización** de la escena para esta **práctica**

4. En esta función se visualizará un grafo de escena que contiene objetos y materiales (que a su vez contienen texturas).

La escena que se visualiza estará compuesta de varios objetos (ver figura 4.1), en concreto los siguientes:

- Objeto **lata** (ver figura 4.3). 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 concreto:
  - **`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.
  - **`lata-psup.ply`** : tapa superior metálica. No lleva textura, es un material difuso-especular de aspecto metálico (ver figura 4.4, derecha).
  - **`lata-pinf.ply`** : base inferior metálica, sin textura y del mismo tipo de material (ver figura 4.4, izquierda).
- 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 4.5, izquierda). La textura está en el archivo `text-madera.jpg` (ver figura 4.5, derecha).
  - Peón **blanco**: sin textura, con un material puramente difuso (sin brillos especulares), de color blanco (ver figura 4.6, izquierda).
  - Peón **negro**: sin textura, con un material especular sin apenas reflectividad difusa (ver figura 4.6, izquierda).

Figura 4.3: Objeto lata obtenido con tres mallas creadas por revolución partir de tres perfiles en archivos ply (izquierda). La malla correspondiente al cuerpo incorpora la textura de la imagen de la derecha.

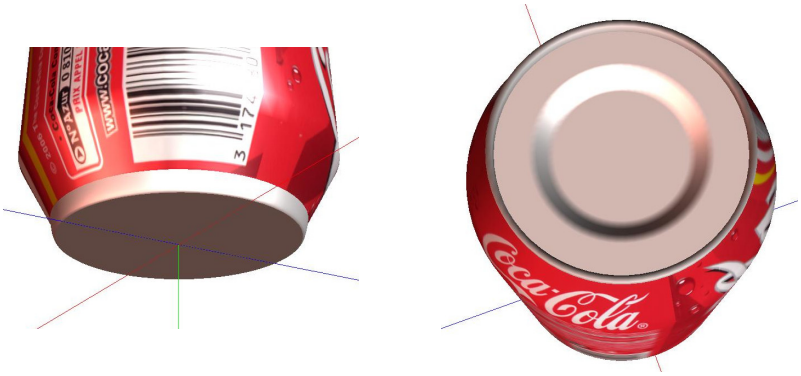


Figura 4.4: Vista ampliada de las tapas metálicas inferior (izquiera) y superior (derecha) de la lata (ambas sin textura)

Figura 4.5: Vista del objeto tipo peón (izquierda) y la textura de madera que se le aplica (derecha). La textura se aplica usando generación automática de coordenadas de textura (la textura se proyecta en el plano XY).

4.3. Grafo de escena 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, así como definir fuentes de luz para que se hagan visibles los materiales.

4.4. Implementación

La implementación de la práctica 4 sigue la línea de las anteriores. Se tendrán en cuenta las siguientes indicaciones:

1. La práctica se implementa en un archivo llamado **`practica4.cpp`**.
2. Para evitar colisiones de nombres, es conveniente que **todas** las funciones declaradas en **`practica4.hpp .cpp`** tengan un nombre que comienze con **`P4_....`** (en todas las prácticas se debe seguir el mismo convenio).

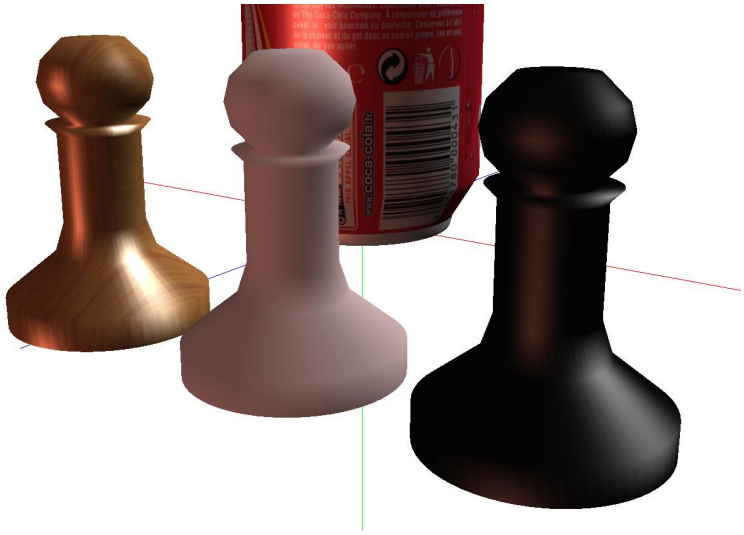


Figura 4.6: Vista ampliada de los tres objetos tipo peón.

3. Para evitar duplicar código, es conveniente que todas las funciones que se usen en más de una prácticas (por ejemplo, la función para activar un material, textura, fuente de luz o similares) no se dupliquen: esto se puede hacer moviendo todas las funciones comunes a un archivo **`.cpp`** específico, que implemente, por ejemplo, todo lo relacionado con los materiales, fuentes de luz y texturas.
4. En el **`makefile`** se debe de añadir **`practica4`** a la lista de archivos a compilar y enlazar para producir el ejecutable (en **`units_loc`**).
5. En esta práctica se debe usar el código fuente que sirve para leer y escribir archivos de imágenes en formato JPEG. Se proporcionan los archivos **`.hpp`** y **`.cpp`** correspondientes (todos ellos tienen nombres que comienzan con **`jpg_`** Estos archivos usan la librería *libjpeg*. Es necesario instalar un package con la versión de desarrollo de *libjpeg*, en distribuciones de linux con paquetes debian se puede usar el package **`libjpeg-dev`**
6. En **`practica4.cpp`** es necesario declarar como variables globales las variables que guardan los parámetros de los materiales y el MIL. Como se indica más abajo, estas variables se gestionan exclusivamente desde **`practica4.cpp`**.
7. La función **`P4_Inicializar`** debe usarse para cualquier inicialización que requiera la práctica 4. Como mínimo, habrá que cargar las texturas y crear los materiales y las fuentes de luz (dandoles valores iniciales a las estructuras o clases relacionadas). Todos estos objetos se deben añadir a un **grafo de escena de la práctica 4**. Al final, se guarda en una variable el puntero al nodo raíz de dicho grafo.
8. En el archivo **`main.cpp`** se debe llamar a las cuatro funciones de inicialización (**`P1_Inicializar`**, **`P2_Inicializar`**, **`P3_Inicializar`** y **`P4_Inicializar`**), una vez al inicio. Para esto es necesario añadir el **`include`** de **`practica4.hpp`** al principio de **`main.cpp`**
9. En el archivo **`main.cpp`** se debe permitir ahora que la práctica activa sea la práctica 4.
10. Es conveniente no complicar mucho las funciones gestoras del evento de tecla normal y de tecla especial, que ahora tendrán muchos casos, y que además, para la práctica 4, necesitan acceder y modificar los valores actuales de los parámetros que determinan el material y las fuentes de luz, valores que se declaran en **`practica4.cpp`**. Para lograr todo esto, lo más fácil usar dos funciones gestoras de eventos de teclado (las llamamos **`P4_FGE_PulsarTeclaNormal`** y **`P4_FGE_PulsarTeclaEspecial`**):
    a) Se declaran en **`practica4.hpp`**, y se implementan en **`practica4.cpp`**

b) Ambas funciones tienen los mismos parámetros que las funciones gestoras de eventos correspondientes que hay en **`main.cpp`**
c) Son invocadas desde f.g.e. de teclado del archivo **`main`** cuando la práctica 4 está activa y la tecla pulsada no corresponde a las teclas comunes. Así que los eventos de teclado específicos de la práctica 4 son redirigidos a **`practica4.cpp`** y se gestionan en dicho archivo.
d) Ambas funciones devuelven un valor lógico (**`true`**/**`false`**), que indica si el evento de teclado corresponde o no corresponde a una de las teclas de la práctica 4 (y por tanto ha cambiado algo del modelo jerárquico). En las f.g.e. de teclado del **`main.cpp`** se lee el valor devuelto y en base a dicho valor se llama o no se llama a `glfwPostRedisplay`.

De esta forma, la gestión del teclado específica de la práctica 4 se hace exclusivamente en **`practica4.cpp`** y no en el **`main.cpp`**, de forma que cualquier cambio en dicha gestión de teclado (p.ej., en como se gestionan las fuentes de luz o los materiales) afectará exclusivamente al archivo **`practica4.cpp`**, pero no al **`main.cpp`** o a otras prácticas.

nota: para poder compilar la práctica, es necesario añadir **`jpg_imagen`** en la definición del símbolo **`units`** que hay en el archivo **`makefile`** (por error no estaba). Además, es necesario descargar de la web y copiar los archivos ply de los perfiles en la carpeta plys.

4.4.1. Teclas de la práctica

Respecto a las teclas específicas de la práctica 4, se usarán las que se indican aquí. Estas teclas permiten incrementar y decrementar los valores de los ángulos que definen la dirección de la fuente de luz direccional. Son estas:

- Tecla **g/G**: conmutar entre el ángulo *alpha* y el ángulo *beta* (en cada instante hay uno de los dos activo, a ese le llamamos *ángulo actual*)
- Tecla **>** aumentar el valor del ángulo actual
- Tecla **<** disminuir el valor del ángulo actual

Para esta práctica, hay que tener en cuenta que en **`main.cpp`** se deberán de gestionar dos nuevos modos de visualización. La tecla **m/M** sigue gestionando los modos, pero ahora se da la posibilidad de que el modo activo sea uno de los dos nuevos. En las prácticas anteriores, los nuevos modos se asimilan al modo sólido, mientras que en esta práctica, se procesarán adecuadamente.

4.4.2. Cálculo y almacenamiento de normales

Respecto de las normales, la clase **`MallaInd`** debe de incorporar:

- Una tabla de normales de caras, es un vector stl o un puntero a un array C/C++, con elementos de tipo **`Tupla3f`**, contiene tantas entradas como triángulos hay en la malla, para cada triángulo contiene su normal.
- Una tabla de normales de vértices, es un vector stl o un puntero a un array de C/C++, con elementos de tipo **`Tupla3f`**. Contiene tantas entradas como vértices, para cada vértice contiene su normal.
- El método **`void calcularNormales()`**, sirve para calcular las tablas de normales de las caras y los vértices, usando las tablas de caras y vértices. Se puede invocar para cualquier malla indexada, en los constructores de las clases derivadas (mallas de revolución y mallas PLY), después de haber creado las tablas de caras y vértices.

4.4.3. Cálculo y almacenamiento de coordenadas de textura.

Respecto de las coordenadas de textura, la clase **`MallaInd`** debe de incorporar un vector stl o un puntero a un array de C/C++ con las coordenadas de textura. Tendrá tantas entradas como vértices, y en cada una de ellas se guardan una tupla de dos valores flotantes (**`Tupla2f`**). Si una malla no tiene coordenadas de textura (porque no lleva coordenadas de textura o bien porque se usa generación automática de coordenadas), el vector stl estará vacío o el puntero al array es nulo.

Cuando se crea una malla por revolución de un perfil, se puede especificar si se desean generar las coordenadas de textura o no se desean. Esto se hace mediante un parámetro booleano que se añade al constructor de la clase **`MallaRevol`**. En dicho constructor, si se desean coordenadas de textura, dichas coordenadas se calcularán como se describe en el guión.

Hay que tener en cuenta que, cuando se requieren coordenadas de textura, la primera y última copia del perfil no conciden, tal y como se indica en el guión.

4.4.4. Implementación de materiales, texturas y fuentes de luz

Para esta práctica se deben de implementar las clases ue se han visto en las transparencias de teoría, en concreto:

- Clase **`Material`**: abstracta, sirve para materiales de cualquier tipo.
- Clase **`MaterialEstandar`**: material con los parámetros del MIL que hemos visto en teoría, es derivada de **`Material`**
- Clase **`FuenteLuz`**: una fuente de luz posicional o direccional, con el vector de posición y el color.
- Clase **`ColeccionFL`**: una clase contenedora con un vector de hasta 8 fuentes de luz

más adelante se detallan las clases concretas que se derivan de estas, y que también deben implementarse en la práctica.

4.4.5. Carga de texturas y envío a la memoria de vídeo o la GPU

Para la lectura de los texels de un archivo de imagen, se puede usar, entre otras, la funcionalidad que se proporciona en la clase **`jpg::Imagen`**, que sirve para cargar JPGs y se usa con el siguiente esquema:

```
#include "jpg_imagen.hpp"
....

// declara puntero a imagen (pimg)
jpg::Imagen * pimg = NULL ;
....

// cargar la imagen (una sola vez!)
pimg = new jpg::Imagen("nombre.jpg");
.....

// usar con:
tamx = pimg->tamX();          // núm. columnas (unsigned)
tamy = pimg->tamY();          // núm. filas (unsigned)
texels = pimg->leerPixels(); // puntero texels (unsigned char *)
```

En memoria, cada texel es una terna rgb (**GL_RGB**), y cada componente de dicha terna es de tipo **GL_UNSIGNED_BYTE**.

Para poder usar estas funciones, es necesario tener estos archivos fuente:

- Las cabeceras (ya están en la carpeta **include**)
  - **jpg_imagen.hpp** declaración de métodos públicos.
  - **jpg_jinclude.hpp** declaración de algunas funciones auxiliares
- Unidades de compilación (se deben compilar y enlazar con el resto de unidades). Estos archivos ya están en la carpeta **srcs**:
  - **jpg_imagen.cpp**
  - **jpg_memsrc.cpp**
  - **jpg_readwrite.cpp**

  Estas unidades se encuentran en el símbolo **units** del **makefile** que hay en **srcs-prac**.

Este código usa la librería **libjpeg**, que debe enlazarse tambien (con el *switch* **-ljpeg** en el enlazador/compilador de GNU). Esta librería puede instalarse en cualquier distribución de linux usando paquetes tipo rpm o o debian. Al hacer la instalación se debe usar la versión de desarrollo (incluye las cabeceras).

4.4.6. Materiales concretos usados en la práctica

En la práctica serán necesarios 5 tipos de materiales: el de la lata (con textura de Coca-Cola), el del peón de madera (con la textura de madera), y el de las tapas de la lata, el del peón negro y el peón blanco (todos ellos sin textura). Para cada tipo de material definiremos una clase derivada de **Material**, clase que únicamente añade un constructor.En cada constructor se le dan valores a los atributos del material, y si es necesario se crea la textura. Por tanto, serán necesarias estas declaraciones:

```
class MaterialLata : public MaterialEstandar
{
  public:
    MaterialLata() ;
};
class MaterialTapasLata : public MaterialEstandar
{
  public:
    MaterialTapasLata() ;
};
class MaterialPeonMadera : public MaterialEstandar
{
  public:
    MaterialPeonMadera() ;
};
class MaterialPeonBlanco : public MaterialEstandar
{
  public:
    MaterialPeonBlanco() ;
};
class MaterialPeonNegro : public MaterialEstandar
{
  public:
    MaterialPeonNegro() ;
};
```

4.4.7. Construcción de las fuentes de luz

Para construir las fuentes de luz, podemos definir dos clases derivadas de la clase **FuenteLuz**, una para las direccionales y otra para las posicionales. Asimismo, definimos una clase para la colección concreta de las dos fuentes de luz (como minimo) que se usan en esta práctica 4. Al igual que los materiales específicos, estas clases solo añaden constructores:

```
class FuenteDireccional : public FuenteLuz
{
  public:
    // inicializar la fuente de luz
    FuenteDireccional( float alpha_inicial, float beta_inicial ) ;

    // Cambiar ángulo:
    // (angulo==0 ->variar alpha, angulo==1 ->variar beta)
    void variarAngulo( unsigned angulo, float incremento ) ;
};
class FuentePosicional : public FuenteLuz
{
  public:
    FuentePosicional( const Tupla3f & posicion ) ;
};
class ColeccionFuentesP4 : public ColeccionFL
{
  public:
    ColeccionFuentesP4() ;
};
```

4.5. Instrucciones para subir los archivos

Para entregar la práctica se creará y se subirán uno dos archivos **.zip**, siguiendo estas indicaciones:

- Hacer un zip (de nombre **P4-fuentes.zip** con todos los fuentes de la carpeta **alum-arcs**, incluyendo **main.cpp** o cualquier otro, el **zip** debe hacerse directamente en **alum-srcs**, y no puede tener carpetas dentro de él, solo puede contener directamente los archivos indicados.
- Hacer otro zip (de nombre **P4-archivos.zip**) con los archivos **.ply** o **.jpg** que no estén ya en las carpetas **plys** o **imgs** (descargadas de la web de la asignatura). No se deben de subir los archivos jpg con las texturas, obtenidos de la web de la asignatura. Es decir, en **alum-archs** (y en el zip) se incluirán los archivos que el alumno haya buscado y usado por su cuenta, y de los cuales el profesor no dispone. El código debe cargar las texturas usando el path así: **"../alum-archs/nombre.jpg"**.
- La práctica debe poser compilarse con los archivos **makefile** e **include.make** que se proporcionan en la web (para que puedan ser evaluadas). Se debe subir el archivo **makefile** modificado con los nombres de la unidades de compilación (archivos **.o**) específicas que el alumno haya usado para esta práctica, tal y como se indica más arriba. En ningún caso se debe subir el archivo **include.make**.
- Incluir un archivo de texto ascii y de nombre **leeme.txt**, en ese archivo, incluir:
  - Indicar si se han añadido materiales al grafo de escena de la práctica 3 o no se ha hecho. En caso afirmativo, indicar las clases de tipo material creadas, así como los nodos del grafo de escena donde se situan.
  - Sistema operativo usado para compilar, y, si se ha usado algún entorno de desarrollo

específico, indicarlo.

- Todas las teclas que se pueden pulsar y utilidad de cada tecla
- Si se ha implementado alguna funcionalidad no descrita en este guión o no. En caso afirmativo, incluir la descripción de dicha funcionalidad (p.ej.: que tipo de objetos se han implementado, donde está el código, que parámetros configurables tiene, etc...)

5. Interacción.

5.1. Objetivos

El objetivo de esta práctica es:

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

5.2. Funcionalidad

Partiendo de las prácticas anteriores, se añadirá la siguiente funcionalidad:

1. Gestión de varios objetos de tipo cámara para visualizar la escena desde distintos puntos de vista.
2. Modificación interactiva de cada una de las cámaras usando el ratón y el teclado, tanto en **modo primera persona** como en modo **orbital** (también llamado modo **examinar**).
3. Selección de objetos o partes de la escena, de forma que se pueda fijar una cámara sobre el objeto seleccionado (en modo examinar).

Se podrá usar para esta práctica el grafo de escena de la práctica 3, el de la práctica 4 o uno nuevo definido por el alumno. En cualquier caso, el grafo de escena contendrá varios objetos independientes, cada uno de ellos seleccionable con el ratón (mediante un identificador entero numérico asignado a cada uno de ellos en el grafo de escena).

5.2.1. Manipulación interactiva de cámaras

Se añadirá al código un vector de cámaras y se incluirá en la escena un mecanismo de control para saber qué cámara de las existentes está activa.

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

Cada cámara podrá funcionar en modo **examinar** o en modo **primera persona**. En cualquiera de los dos modos, las cámaras podrán manipularse usando teclado o ratón. Mediante estas manipulaciones se podrá cambiar la posición y orientación del marco de coordenadas de vista de la cámara activa. En las siguientes secciones se detallan los distintos modos y como se conmuta entre ellos.

5.2.2. Selección de objetos en la escena

Usando el ratón (pulsando con el botón izquierdo en un pixel de la ventana) se podrá seleccionar alguno de los objetos en la escena. Cuando esto ocurre, la cámara activa pasará a modo examinar (si no lo estaba ya). El punto de atención de dicha cámara quedará fijado a un punto central al objeto,

y a partir de entonces los movimientos de la cámara se harán rotando alrededor de dicho punto, según se describe en las transparencias de teoría, hasta que se conmute el modo de la cámara activa o bien se cambie la cámara activa.

5.3. Modificación interactiva de cámaras

En esta sección se detalla la interacción con el usuario para la modificación interactiva de las cámaras, bien usando el teclado, bien usando el ratón

5.3.1. Uso del teclado

Cuando la práctica activa es la práctica 5, se podrá usar el teclado para cambiar la cámara activa, cambiar el modo de funcionamiento de dicha cámara, y para cambiar el marco de coordenadas de vista. Las teclas permitirán desplazar (en modo primera persona) o rotar (en modo examinar) el marco de coordenadas de vista, en horizontal o en vertical. También permiten desplazar la cámara hacia delante o hacia detrás (en la dirección del eje Z del marco de coordenadas de vista), en ambos modos.

En concreto, se dispondrá de las siguientes teclas:

- Tecla **c/C**
  Cambia la camara activa, pasa a la siguiente, o de la última a la primera (se debe imprimir un mensaje indicando cual es la nueva cámara activa).
- Tecla **v/V**
  Si la camara activa está en modo examinar, ponerla en modo primera persona. Si la cámara activa está en modo primera persona, pasa a modo examinar. En cualquier caso se indica en un mensaje el nuevo modo, y cual es la cámara activa.
- **Flecha izquierda**
  Desplazar o rotar la cámara activa en horizontal, hacia la izquierda.
- **Flecha derecha**
  Desplazar o rotar la cámara activa en horizontal, hacia la derecha.
- **Flecha arriba**
  Desplazar o rotar la cámara en vértical, hacia arriba.
- **Flecha abajo**
  Desplazar o rotar la cámara en vértical, hacia abajo.
- Tecla **+** (o página arriba)
  Desplazar la camara en la direción de la vista (eje Z de la cámara), hacia adelante.
- Tecla **-** (o página abajo)
  Desplazar la cámara en la direccion de la vista (eje Z de la cámara), hacia detrás.

5.3.2. Uso del ratón

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

En el modo *arrastrar*, cada vez que se produce un evento de movimiento del ratón (mientras se mantiene pulsada su tecla derecha), se actualiza la cámara actual. Para ello se usa el desplazamiento del ratón desde el pixel donde se pulso el botón derecho hasta el pixel hasta donde se ha movido.

Ese desplazamiento se traduce en dos números enteros, positivos o negativos, que indican cuantas filas o columnas de pixels se ha movido el ratón. Esos desplazamientos se usan para seleccionar las distancias a desplazar la cámara (en modo primera persona), o bien los dos ángulos de rotación (en modo examinar). El ratón no permite hacer desplazamientos en el eje Z del marco de coordenadas de la cámara.

5.4. Implementación

En esta sección se incluyen detalles sobre la implementación de esta práctica, en línea con lo que se ha hecho para las anteriores.

5.4.1. Creación de archivos y funciones de la práctica 5

Al igual que en las otras prácticas, en esta se deben crear los archivos y funciones específicos de la práctica. A continuación se detalla esto:

1. La práctica se implementa en un archivo llamado **`practica5.cpp`**, de la plantilla que se proporciona.
2. Para evitar colisiones de nombres, es conveniente que **todas** las funciones declaradas en **`practica5.hpp (.cpp)`** tengan un nombre que comienze con **`P5_`**`....` (en todas las prácticas se debe seguir el mismo convenio).
3. En el **`makefile`** se debe de añadir **`practica5`** a la lista de archivos a compilar y enlazar para producir el ejecutable (en **`units_loc`**).
4. En **`practica5.cpp`** es necesario declarar como variables globales las variables que guardan las cámaras (en un array o vector de objetos de tipo **`CamaraInteractiva`**). Como se indica más abajo, estas variables se gestionan exclusivamente desde **`practica5.cpp`**. También se declarará una variable de tipo **`Viewport`** (visto en teoría, tema 3). Esta variable contiene las dimensiones del viewport actual, que se inicializa en **`P5_Inicializar`** y se actualiza en la función **`P5_FijarMVPOpenGL`**, cada vez que se va a dibujar.
5. La función **`P5_Inicializar`** debe usarse para cualquier inicialización que requiera la práctica 5. Como mínimo, habrá que crear los objetos (nodos de grafo de escena) que se usarán en la práctica (dandoles valores iniciales a las estructuras o clases relacionadas). Habrá que crear un array de objetos de tipo cámara, y darle un valor inicial a una variable entera que designa cual de ellas es la cámara activa. También se la variable de tipo **`Viewport`** (se asume que el origen del viewport es $0, 0$), con los datos del viewport actual.
   La función **`P5_Inicializar`** debe recibir como parámetros el ancho y alto incial de la ventana que se crea en **`main`**, ya que la inicialización del viewport depende de esas dimensiones (la inicialización de las cámaras también usa el tamaño del viewport).
6. En el archivo **`main.cpp`** se debe llamar a las cinco funciones de inicialización (incluyendo **`P5_Inicializar`**), una vez al inicio. Para esto es necesario añadir el **`include`** de **`practica5.hpp`** al principio de **`main.cpp`**. En **`main`** se pueden usar las variables **`ventana_tam_x`** y **`ventana_tam_y`**, (que contienen el tamaño actual de la ventana de la aplicación) como parámetros de **`P5_Inicializar`**.
7. En el archivo **`main`**`.cpp` se debe permitir ahora que la práctica activa sea la práctica 5.
8. En esta práctica se calcula la cámara usada para visualizar la escena en cada momento, de forma distinta al resto de la prácticas, esto implica que ha sido necesario modificar la función **`VisualizarFrame`** que hay en **`main.cpp`**. En la nueva versión, si la práctica activa es la 5, no se invocan las funciones **`FijarViewportProyeccion`** ni **`FijarCamara`**. En lugar de esto, se hacen estas llamadas:

```
if ( la practica activa es la 5 )   // fijar matrices y viewport en P5:
```

```
    P5_FijarMVPOpenGL( ventana_tam_x, ventana_tam_y ) ;
else // hacer lo mismo que antes:
{   FijarViewportProyeccion() ;
    FijarCamara();
}
```

la función `P5_FijarMVPOpenGL` está declarada en **`practica5.hpp`** y debe definirse (implementarse) en **`practica5.cpp`**, según las indicaciones que hay en el guión de la práctica. Tiene como parámetros el tamaño actual de la ventana (ancho y alto).

9. Es conveniente no complicar mucho las funciones gestoras del evento de tecla normal y de tecla especial, que ahora tendrán muchos casos, y que además, para la práctica 5, necesitan acceder y modificar los parámetros de las cámaras, parámetros que se declaran en **`practica5.cpp`**. Para lograr todo esto, lo más fácil usar dos funciones gestoras de eventos de teclado (las llamamos `P5_FGE_PulsarTeclaNormal` y `P5_FGE_PulsarTeclaEspecial`):
   a) Están declaradas en **`practica5.hpp`**, y se implementan en **`practica5.cpp`**
   b) Ambas funciones tienen los mismos parámetros que las funciones gestoras de eventos correspondientes que hay en **`main.cpp`**
   c) Son invocadas desde f.g.e. de teclado del archivo **`main`** cuando la práctica 5 está activa y la tecla pulsada no corresponde a las teclas comunes. Así que los eventos de teclado específicos de la práctica 5 son redirigidos a **`practica5.cpp`** y se gestionan en dicho archivo.
   d) Ambas funciones devuelven un valor lógico (**`true`**/**`false`**), que indica si el evento de teclado corresponde o no corresponde a una de las teclas de la práctica 5 (y por tanto ha cambiado algo del modelo jerárquico). En las f.g.e. de teclado del **`main.cpp`** se lee el valor devuelto y en base a dicho valor se llama pone a **`true`** o no la variable `redibujar_ventana`, para redibujar la escena.

   De esta forma, la gestión del teclado específica de la práctica 5 se hace exclusivamente en **`practica5.cpp`** y no en el **`main.cpp`**, de forma que cualquier cambio en dicha gestión de teclado (p.ej., en como se gestionan las cámaras) afectará exclusivamente al archivo **`practica5.cpp`**, pero no al **`main.cpp`** o a otras prácticas.

5.4.2. Clase para cámaras interactivas. Métodos.

En esta práctica se declararán un nuevo tipo de objetos, los objetos de tipo o clase **Camara**. La descripción de la estructura de estos objetos se encuentra en las transparencias del tema 3 (subsección 1.6: *representación de los parámetros de las transformaciones*).

Para la realización de la práctica definiremos un tipo específico de cámaras, que añaden la funcionalidad necesaria de la práctica. La clase `Camara` y su clase derivada `CamaraInteractiva` tienen esta declaración:

```
class Camara
{  public:
   MarcoCoorVista  mcv ; // marco de coordenadas de la vista
   ViewFrustum     vf  ; // parámetros de la proyección
   Camara() ;            // usa constructores por defecto para mc y vf
   void activar() ;   // fijar matrices MODELVIEW y PROJECTION de OpenGL
} ;
```

```
class CamaraInteractiva : public Camara
{
 public: // todo es público
 bool    examinar;      // modo: true ->examinar, false ->primera persona
```

```
bool    perspectiva; // tipo proy.: true ->perspectiva, false ->ortográfica
float   ratio_yx_vp; // aspect ratio del viewport (alto/ancho)
float   longi,       // ángulo (en grados) en torno al eje Y (longitud)
        lati ;       // ángulo (en grados) en torno al eje X (latitud)
Tupla3f aten ;       // punto de atención (inicialmente el origen)
float   dist ;       // distancia entre el punto de atención y el observador

// constructor de cámaras interactivas, los parámetros son:
// * examinar_ini: fija modo examinar (true) o modo primera persona (false)
// * ratio_yx_vp_ini: ratio del viewport (alto/ancho)
// * longi_ini_grad, lati_ini_grad: longitud y latitud iniciales (en grados)
// * aten_ini: punto de atención inicial
// * pers_ini: fija proy. perspectiva (true) u ortográfica (false)
CamaraInteractiva( bool examinar_ini, int ratio_yx_vp_ini,
                   float longi_ini_grad, float lati_ini_grad,
                   const Tupla3f & aten_ini, bool pers_ini );

// calcula el view-frustum (y la matriz de proyección) en vf
void calcularViewfrustum( ); // lee: perspectiva, dist, ratio_yx_vp

// calcula el marco de referencia de la camara (y la matriz de vista), en mcv
void calcularMarcoCamara( ); // lee: longi, lati, dist, aten

// métodos para manipular (desplazar o rotar) la cámara
void moverHV( int nh, int nv ); // desplazar o rotar la cámara en horizontal/vertical
void desplaZ( int nz );         // desplazar en el eje Z de la cámara (hacia adelante o hacia detrás)

// métodos para cambiar de modo
void modoExaminar( const Tupla3f & pAten ); // fija punt. aten. y activa modo examinar
void modoExaminar();                        // pasa a modo examinar (mantiene p.aten.)
void modoPrimeraPersona() ;                 // pasa al modo primera persona

} ;
```

Las variables de instancia `longi`, `lati` y `dist`, contienen las coordenadas esféricas del punto del observador en relación al punto de atención, para la cámara en modo examinar.

La métodos `moverHV` y `desplaZ` funcionan de una forma que depende del modo de la cámara, a continuación se detalla el funcionamiento de ambos métodos en ambos modos.

En modo primera persona:

En este caso, los métodos suponen desplazamiento del observador, es decir, del origen del marco de coordenadas de vista:

Método `moverHV( int nh, int nv )`

Desplaza el origen (posición del observador) del marco de coordenadas de la cámara en el sentido del eje X (horizontal) de dicho marco, un número de unidades igual a `nh*udesp`, y en el sentido del eje Y (vertical), un número de unidades igual a `nv*udesp` Los ejes del marco no cambian.

El valor de `udesp` (unidad de desplazamiento) es un real, positivo (nunca cero), y se implementa como una constante definida en el programa. Podrá ajustare para que el uso de la cámara sea cómodo, y podría depender de la escala de la escena.

El parámetro `nh` puede ser positivo (implica movimiento a la derecha), o negativo (implica movimiento a la izquierda), igualmente, el valor de `nv` puede ser positivo (movimiento hacia arriba) o negativo (hacia abajo). Si ambos son cero, no se cambia nada.

Método `desplaZ( int nz )`

Igual a la anterior, pero desplaza el punto del observador o foco exclusivamente en la dirección del eje Z del marco de la cámara (`nz*udesp` unidades). Esto implica que el desplazamiento es en la dirección en la que el usuario está mirando la escena. El signo del parámetro `nz` determina si el movimiento es hacia adelante (`nz` positivo) o hacia detrás (`nz` negativo). Si `nz` es cero, no se hace nada.

En modo examinar:

En este caso, `moverHV` supone rotar la cámara entorno al punto de atención, y `desplaZ`, supone alejarla o acercarla al mismo.

Método `moverHV( int nh, int nv )`

Incrementa el valor de `long` (longitud) un número de radianes igual a `nh*urot`, y el valor de `lati` (latitud) un número de radianes igual a `nv*urot`. Esto implica que se debe recalcular el marco de coordenadas de la cámara, de forma que el punto de atención se mantega fijo (se proyecte en el centro de la imagen), pero cambien los ejes y la posición del observador.

El valor de `urot` (unidad de rotación) es un real en radianes, positivo (nunca cero), y se implementa como una constante definida en el programa. Podrá ajustare para que el uso de la cámara sea cómodo.

Método `desplaZ( int nz )`

Cambia la distancia entre el observador y el punto de atención, desplazando el observador en la dirección del eje Z del marco de la camara. Cuando `nz` es positivo, la distancia disminuye, acercando el observador el punto de atención (se mueve hacia delante). Cuando es negativa, la distancia aumenta, alejando el observador de dicho punto de atención (se mueve hacia detrás). En cualquier caso, la distancia entre ambos puntos nunca será menor que un valor mínimo (`dmin`) determinado.

Los incrementos o aumentos pueden hacerse actualizando la distancia de esta forma:

```
dist = dmin + (dist-dmin)*(1.0-nz*porc/100.0)
```

Donde `porc` es un factor real positivo (un tanto por ciento) que indica cuanto cambia el exceso de distancia (exceso en el sentido de por encima del mínimo). El valor `porc` puede hacerse, por ejemplo, igual a la unidad (es decir, el cambio mínimo es del 1 %) y luego incrementarlo o decrementarlo para hacer cómoda la navegación. El valor de `dmin` podría incorporarse a la cámara y hacerse depender del tamaño de cada objeto, o bien simplemente dejarlo como una constante predeterminada.

Puesto que esta función modifica el valor de `dist`, después de cambiarlo se debe llamar a `calcularMarcoCamara` para actualizar el marco de la cámara.

5.4.3. Activación de cámaras: fijar las matrices OpenGL

A diferencia de las otras prácticas, en esta se debe implementar el método `P5_FijarMVPOpenGL` (llamado desde `main.cpp`) para fijar la matriz de vista (*modelview* inicial) y la matriz de proyección (*projection*) de OpenGL (se usarán después al visualizar los objetos). Esto se debe hacer usando la camara activa en el momento de la llamada. La declaración de este método es como sigue:

```
void P5_FijarMVPOpenGL( int vpx, int vpy );
```

los parámetros **vpx** y **vpy** contienen las dimensiones del viewport actual, esto es necesario ya que en la práctica 5 es necesario tener registradas esas dimensiones. Por tanto, se usan esos valores para actualizar el objeto **Viewport** que hay en la práctica 5. El view frustum de la cámara activa debe adaptarse a las proporciones del viewport, de forma que los objetos no aparezcan deformados (se debe actualizar **ratio_yx_vp** y después llamar a **calcularViewfrustum** de la cámara activa, que recalcula su view-frustum en función del **ratio_yx_vp**, **dist** y **vf.persp**).

Para implementar **P5_FijarMVPOpenGL** la clase **Camara** incorpora un método para fijar las matrices OpenGL. Este método se denomina **activar** (no tiene parámetros), según se describe en las transparencias de teoría (ver tema 3, sección relativa a la representación de los parámetros de las transformaciones, donde se introduce la clase cámara).

El efecto de la llamada al método **activar** es establecer la matriz *modelview* y la matriz *projection* del cauce de OpenGL, en función del estado de la cámara. Para ello se usa la matriz **matrizVista** que hay en el marco de coordenada de vista, y la matriz **matrizProy** que hay en el view-frustum (dentro de cada objeto instancia de **Camara**).

Lo anterior requiere recalcular adecuadamente las matrices **matrizVista** y **matrizProy** cada vez que se invoca a cualquier método de la cámara que cambie sus parámetros, en concreto cuando se llama a **moverHV** y **desplaZ**, o también cuando se llama a **calcularViewfrustum** (que afecta a la matriz de proyección).

5.4.4. Movimiento de la cámara con eventos de ratón.

La cámara activa se puede controlar con el teclado, pero además también podrá gestionarse moviendo el ratón con el botón derecho pulsado. Esto requiere definir en el archivo **main.cpp** las funciones gestoras de eventos de ratón, registrarlas al inicio, y usarlas para ejecutar código de la práctica 5 cuando se produzcan dichos eventos. Para continuar con el esquema que ya tenemos de otras prácticas, haremos que las funciones gestoras generales (en **main**.cpp) invoquen a las específicas de la práctica 5 (en **practica5.cpp**) cuando esta práctica este activa.

Por tanto en **main.cpp** se han insertado estas llamadas a las f.g.e. específicas de la práctica 5, así:

```
// función gestora del evento de pulsar/levantar tecla del ratón
// (se registra con glfwSetMouseButtonCallback)

void FGE_ClickRaton( GLFWwindow* window, int button, int action, int mods )
{
   .....
   if ( practicaActual == 5 )
   {  if ( P5_FGE_ClickRaton( button, action, x, y ) )
         redibujar_ventana = true ;
      return ;
   }
   ........
}
```

```
// función gestora del evento de ratón movido a una nueva posición
// (se registra con glfwSetCursorPosCallback)

void FGE_RatonMovido( GLFWwindow* window, double xpos, double ypos )
{
   ......
   if ( practicaActual == 5 )
```

```
   {  if ( P5_FGE_RatonMovidoPulsado( int(xpos), int(ypos) ) )
         redibujar_ventana = true ;
      return ;
   }
   ......
}
```

```
// función gestora del evento de hacer scroll (movimiento rueda del ratón)
// (se registra con glfwSetScrollCallback

void FGE_Scroll( GLFWwindow* window, double xoffset, double yoffset  )
{
   .....
   if ( practicaActual == 5 )
   {  if ( P5_FGE_Scroll( direction ) )
         redibujar_ventana = true ;
      return ;
   }
   .......
}
```

Estas funciones, si la práctica 5 está activa, invocarán a las correspondientes funciones gestoras de la práctica 5, en otro caso no harán nada. Al gestionar el evento en la práctica 5, se determina si se requiere redibujar la escena o no es así, y se devuelve **`true`** en el primer caso y **`false`** en el segundo (por ejemplo, al seleccionar, si no se ha seleccionado ningún objeto por haberse pinchado en el fondo, no se debe redibujar la escena).

Dichas funciones gestoras de la práctica 5 se declaran en `practica5.hpp` y se definirán en **`practica5.cpp`**, con estos nombres y parámetros:

```
bool P5_FGE_ClickRaton( int button, int state, int x, int y );
bool P5_FGE_RatonMovidoPulsado( int x, int y );
```

Al procesar el click de ratón (**`P5_FGE_ClickRaton`**), se debe determinar si se ha pulsado un botón, y de que botón se trata.

- Si se ha bajado el botón izquierdo, se ejecuta el código para seleccionar un objeto y pasar la cámara activa actualmente al modo examinar (si se ha seleccionado algún objeto). Este código se encuentra en esta función:

  ```
  // calcula objeto sobre el que se ha hecho el click, si hay alguno seleccionado pone
  // la cámara activa mirando a dicho objeto en modo examinar y devuelve true,
  // en otro caso devuelve false
  bool P5_ClickIzquierdo( int x, int y ) ;
  ```

- Si se trata de una pulsación del botón derecho debe quedar registrada la posición del ratón (en dos variables: **`xant`** e **`yant`**). Para eso se usa la función:

  ```
  // entra en modo arrastrar, registra donde se ha iniciado el movimiento
  void P5_InicioModoArrastrar( int x, int y ) ;
  ```

- Si se trata de levantar el botón derecho, hay que registrar que ya no es está en modo arrastrar, llamando a

  ```
  // acaba el modo arrastrar
  void P5_FinModoArrastrar( int x, int y ) ;
  ```

Al procesar el movimiento del ratón con el botón pulsado (en `P5_FGE_RatonMovidoPulsado`), se debe de verificar si el botón pulsado es el botón derecho, en ese caso se debe invocar esta función:

```
// en modo arrastrar, procesa la nueva posición del ratón
void P5_RatonArrastradoHasta( int x, int y ) ;
```

En esta función, se considera la posición actual del ratón (`x` e `y`), y se calcula el desplazamiento en pixels del ratón desde que se pulsó el botón derecho o desde el anterior movimiento, ya con el botón pulsado. Esto se hace calculando `dx` como `x-xant` (desplazamiento en horizontal) y `dy` como `y-yant` (desplazamiento en vértical), suponiendo que `xant` y `yant` es la posición anterior (o la inicial, registrada al pulsar el botón derecho).

Los valores `dx` y `dy` se usan como argumentos para el método `moverHV` de la cámara activa, para modificarla. Finalmente, se actualiza `xant` e `yant`, haciéndose iguales a `x` e `y`.

Finalmente también queremos controlar las cámaras con la rueda de scroll del ratón, suponiendo que el ratón lo tenga. Para eso es necesario escribir el código de la función correspondiente, declarada así:

```
// procesa evento de movimiento de scroll
// (acerca/aleja el observador de la cámara activa)
void P5_FGE_Scroll( int direction ) ;
```

Hay que tener en cuenta que las funciones `P5_ClickIzquierdo`,`P5_InicioModoArrastrar` y `P5_FGE_RatonMovidoPulsado` deben recibir la coordenadada Y en coordenadas de dispositivo OpenGL (no negativas), cuya coordenada Y crece desde 0 de abajo arriba. Sin embargo, las funciones `P5_FGE_ClickRaton` y `P5_FGE_RatonMovidoPulsado` reciben las coordenadas producidas por el sistemas de ventanas, cuya coordenada Y suele crecer desde 0 de arriba abajo.

Por tanto, en `P5_FGE_ClickRaton` y en `P5_FGE_RatonMovidoPulsado` es necesario transformar la coordenada Y desde coordenadas del gestor de ventanas a coordenadas de dispositivo de OpenGL, antes de llamar a las otres tres funciones. Para ello, se debe usar la información disponible del viewport (en concreto, debemos de usar el alto del viewport).

5.4.5. Identificación de objetos. Punto central.

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

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

```
int      identificador;  // identificador del objeto
Tupla3f  centro_oc;      // punto central del objeto, en coordenadas de objeto
```

El identificador es un valor entero, entre 0 y 2^{24}. Se usará el valor 0 para los nodos que no puedan ser seleccionados, al pinchar en estos objetos se ignora el click (igual que al pinchar en el fondo). Se usará el valor -1 para los objetos de tipo nodo (`NodoGrafoEscena`) que tengan un identificador similar al del padre (esto permite referenciar un nodo desde distintos padres con distintos identificadores en los padres).

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

```
virtual bool buscarObjeto( const int         ident_busc,
                           const Matriz4f & mmodelado,
                           Objeto3D **       objeto,
                           Tupla3f &         centro_wc   );
```

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

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

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

En la clase `NodoGrafoEscena`, hay una implementación distinta del método `buscarObjeto`. En este caso, se comprueba el identificador con el identificador del nodo, si coinciden ya se ha encontrado el objeto, y acaba, en otro caso se busca recursivamente en los nodos hijos del actual (si hay alguno).

En la construcción de los objetos de la escena para esta práctica, debemos de asignar un identificador único y distinto de 0 a los objetos que queramos que se pueda seleccionar (puede ser -1 en algunos casos). Para los que no queramos seleccionar, se debe inicializar explicitamente el identificador a 0 (es importante no dejarlo sin inicializar para ningún nodo). Asimismo, para cada uno de esos nodos, debemos de inicializar la tupla que representa el centro del nodo (en coordenadas de objeto).

El cálculo del punto central de un objeto es simple para objetos de la clase `MallaInd` y derivadas (es el centro de la caja englobante más pequeña que contiene a todos los vértices). Para el caso de objetos de tipo `NodoGrafoEscena`, ese cálculo puede hacerse recursivamente una sola vez, en concreto la primera vez que se invoque el procedimiento `buscarObjeto` de un nodo. Para ello se debe incorporar en `NodoGrafoEscena` una variable booleana que indique si el centro ha sido ya calculado o no. Además, se necesita de un método recursivo de cálculo del punto central (calcula el punto central de un nodo y de todos los nodos descendientes del primero, lo cual solo se hace una vez).

5.4.6. Selección de objetos

La función `P5_ClickIzquierdo` se invoca cuando se desea hacer la selección. La selección se puede implementar de dos formas, segun se ha visto en teoría (ver el apartado sobre selección del tema 4). Una de ellas usa el **buffer de selección** de OpenGL, mientras que la otra simplemente dibuja en un **frame-buffer invisible**, típicamente se usa el **frame-buffer trasero**.

1. Fijar el color del fondo de pantalla a (0,0,0) (usando tres *unsigned byte* puestos a cero), y limpiar la imagen.
2. Activar el *modo selección* (ver más abajo), y visualizar la escena, llamando a `visualizarGL` para el nodo raíz.
3. Determinar si se ha seleccionado algún objeto o no. Esto depende del modo del selección. Sabemos que no se ha seleccionado ningún objeto cuando:
   - buffer de selección: el buffer de selección está vacío, o bien tiene varios identificadores, todos ellos 0 (es decir: no hay ningún identificador mayor estricto que 0 en dicho buffer).
   - frame-buffer invisible: el color del pixel sobre el que se ha hecho click es 0 (color de fondo y de los objetos no seleccionables).
4. Si no se ha seleccionado ningún objeto, entonces se imprime un mensaje indicado esto, se devuelve false y no ocurre nada más (ocurre al pinchar en el fondo o en un objeto no seleccionable, con identificador 0).
5. Si se ha seleccionado algún objeto, se habrá obtenido un identificador (entero) del mismo, distinto de 0. Se debe buscar el correspondiente nodo en el grafo de escena, y obtener un puntero al mismo. Se debe imprimir un mensaje con el nombre del nodo seleccionado. Si no se encuentra el objeto, esto indicaría una situación errónea que en principio no debe ocurrir, en este caso se imprimiría un mensaje de error que puede servir para depurar el programa (no se haría nada más).
6. Pasar la cámara activa al modo examinar, centrándola en el punto central del nodo seleccionado.

5.4.7. Visualización del grafo de escena en modo *selección*

Para poder conocer el identificador del objeto sobre el que se ha hecho click, es necesario redibujar la escena (esto es cierto tanto si se usa el buffer de selección como si se usa el frame-buffer trasero). En nuestro caso, puesto que la escena está es un árbol de nodos, debemos de usar un nuevo modo de visualización (lo llamaremos *modo selección*), que se debe de tener en cuenta en el método `visualizarGL` de los nodos. Cuando la visualización se hace en este modo:

- Antes de visualizar, se deben de desactivar la iluminación y las texturas. También se debe activar dibujado en modo relleno (de caras delanteras y traseras), y activar el modo de sombreado plano. El color actual de dibujo debe ser el (0,0,0) (se debe fijar con `glColor3ub`). Si se usa el buffer de selección, se debe invocar el método para iniciar dicho modo (ver transparencias).
- Hay que tener en cuenta que se debe usar el mismo viewport y la misma cámara activa actual, ya que de otra forma no estarían alineadas las vista normal y la vista creada para la selección. En la práctica 5 se encuentra la cámara activa, y también se tiene registrado el tamaño del viewport (se actualiza al llamar a `P5_fijarMVPOpenGL`).
- Antes de visualizar cada entrada de un nodo N, se debe de activar el identificador del nodo N, si dicho identificador no es -1. Esto depende del método de selección que se use:
  - Con buffer de selección: se usará la función para modificar el nombre actual (`glLoadName`), usando como argumento el identificador de N.
  - Con frame-buffer invisible: se usará la función que se detalló en teoría para modificar el color actual, a partir del identificador de N (ver función `FijarColorIdent`).

  Cuando el identificador de un nodo es -1, se supone que el identificador a usar ya está activado antes de visitar el nodo, en algún nodo ancestro del actual.
  Es importante que se cambie el color o el nombre **antes de cada entrada**, no solo una única vez al inicio del nodo (en otro caso el identificador de un nodo podría afectar erróneamente a sus hermanos posteriores en el nodo padre)

- Durante la visualización, se deben ignorar (no activar) la entradas de tipo material, ya que no estamos visualizando en pantalla. Por supuesto, las entradas de tipo transformación deben procesarse igual que en el resto de los modos.

5.5. Instrucciones para subir los archivos

Para entregar la práctica se creará y se subirá un único archivo **`.zip`**, de nombre **`P5.zip`**, siguiendo estas indicaciones:

- Hacer un zip con todos los fuentes de la carpeta **`srcs-alum`**, incluyendo **`main.cpp`** o cualquier otro, el **`zip`** debe hacerse directamente en **`srcs-alum`**, y no puede tener carpetas dentro de él, solo puede contener directamente los archivos indicados.
- En el zip que se indica más arriba se deben de incluir los archivos **`.ply`** o **`.jpg`** que no estén ya en las carpetas **`plys`** o **`imgs`** (descargadas de la web de la asignatura). No se deben de subir los archivos ply con los perfiles, obtenidos de la web de la asignatura. Es decir, en `srcs-alum` (y en el zip) se incluirán los archivos que el alumno haya buscado y usado por su cuenta, y de los cuales el profesor no dispone.
- La práctica debe poder compilarse con los archivos **`makefile`** e **`include.make`** que se proporcionan en la web (para que puedan ser evaluadas). Se debe subir el archivo `makefile` modificado con los nombres de la unidades de compilación (archivos **`.o`**) específicas que el alumno haya usado para esta práctica. En ningún caso se debe subir el archivo **`include.make`**.
- Incluir un archivo de texto ascii y de nombre **`leeme.txt`**, en ese archivo, incluir:
  - Indicar la funcionalidad que se ha implementado (ver la lista de características evaluables que hay en la sección de evaluación de este guión).
  - Sistema operativo usado para compilar, y, si se ha usado algún entorno de desarrollo específico, indicarlo.
  - Si se puede usar alguna tecla adicional a las descritas en el guión, se debe indicar que tecla es y su funcionalidad.
  - Si se ha implementado alguna funcionalidad no descrita en este guión o no. En caso afirmativo, incluir la descripción de dicha funcionalidad.