

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

Índice.	3
0. Prerequisitos software	5
0.1. Sistema Operativo Linux	5
0.1.1. Compiladores	5
0.1.2. OpenGL	5
0.1.3. Librería GLEW	5
0.1.4. Librería GLFW	6
0.1.5. Librería JPEG	6
0.2. Sistema Operativo macOS	6
0.2.1. Compilador	6
0.2.2. Librería GLFW	6
0.2.3. Librería JPEG	7
1. Visualización de modelos simples	9
1.1. Objetivos	9
1.2. Desarrollo	9
1.3. Evaluación	9
1.4. Teclas a usar. Interacción.	10
1.5. Implementación	10
1.5.1. Inicialización y gestión de eventos	11
1.5.2. Contexto y modos de visualización	12
1.5.3. Clase abstracta para objetos gráficos 3d.	12
1.5.4. Clase para mallas indexadas.	13

1.5.5. Programación del cauce gráfico	13
1.5.6. Uso de OpenGL 2.0	14
1.5.7. Clases para los objetos de la práctica 1	14
1.5.8. Clases para tuplas de valores enteros o reales con 2,3 o 4 valores	15
1.6. Instrucciones para subir los archivos	16
2. Modelos PLY y Poligonales	17
2.1. Objetivos	17
2.2. Desarrollo	17
2.3. Creación del sólido por revolución	18
2.3.1. Lectura o creación del perfil inicial	20
2.3.2. Creación de la tabla de vértices	20
2.3.3. Creación de la tabla de caras (triángulos)	21
2.4. Teclas a usar	22
2.5. Implementación	22
2.5.1. Clase para mallas creadas a partir de un archivo PLY.	23
2.5.2. Clase para mallas creadas a partir de un perfil, por revolución.	24
2.5.3. Clases para: cilindro, cono y esfera	24
2.6. Lectura de archivos PLY	25
2.7. Archivos PLY disponibles.	26
2.8. Visualización en modo diferido	26
2.9. Instrucciones para subir los archivos	27
3. Modelos jerárquicos	29
3.1. Objetivos	29
3.2. Desarrollo	29
3.2.1. Reutilización de elementos	30
3.3. Animación	31
3.3.1. Gestión de los grados de libertad y sus velocidades	31
3.3.2. Animación mediante la función desocupado.	32
3.4. Teclas a usar	32

3.5. Implementación	33
3.5.1. Implementación de objetos jerárquicos parametrizados	34
3.5.2. Implementación de la animación	37
3.5.3. Instrucciones para subir los archivos	38
3.6. Algunos ejemplos de modelos jerárquicos	39

Práctica 0

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 con el driver apropiado para la tarjeta gráfica.

En cualquier caso, en ubuntu, para verificar la tarjeta instalada y ver los drivers recomendados y/o posibles para dicha tarjeta, se puede usar esta orden:

```
sudo ubuntu-drivers devices
```

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

```
sudo ubuntu-drivers autoinstall
```

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 puedan 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 esa carpeta vacía se crea una subcarpeta raíz (de nombre `glfw-...`). Después compilamos e instalamos la librería con estas órdenes:

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


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

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 -xzf 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.

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 visualiza 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`.

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 incorpora 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 cámara al origen (zoom).

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

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

1.5. Implementación

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

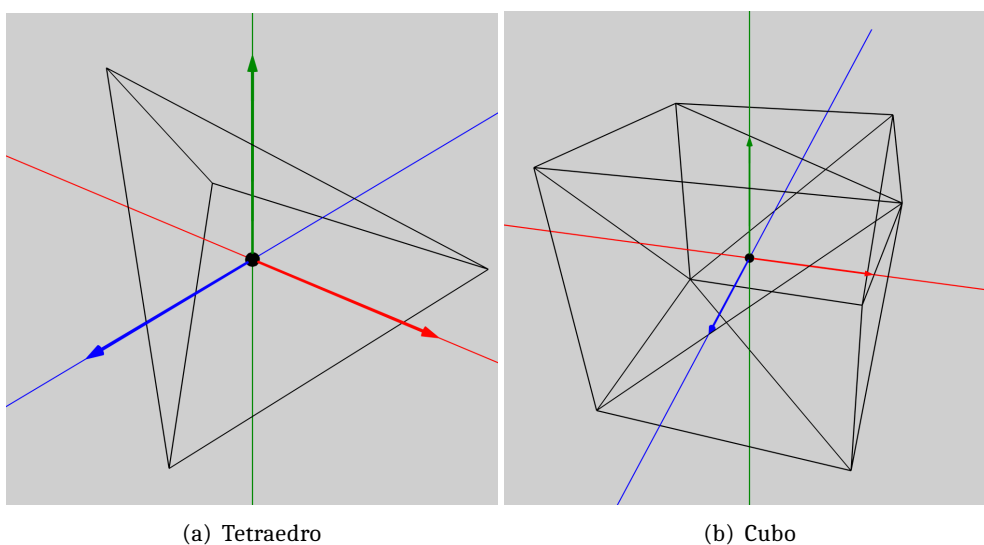


Figura 1.1: Tetraedro y cubo visualizados en modo alambre.

- `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 `practical.cpp`

Para compilar el código, basta con teclear `make` (estando en la carpeta `alum-srcs`), esta orden leerá el archivo `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 modificado 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 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 `practical.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 y se ha 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`

Esta 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 `practicass.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, modoSólido, 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 dirección 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 `practical.hpp/.cpp`. En cualquier caso, en el archivo `practical.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 predefinidas para coordenadas (X,Y,Z) o colores (R,G,B):
float x1 = a(0), y1 = a(1), z1 = a(2), //
        x2 = a(X), y2 = a(Y), z2 = a(Z), // apropiado para coordenadas
        re = c(R), gr = c(G), bl = c(B) ; // apropiado para colores

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

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

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

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

```
// declaraciones de tuplas y de valores escalares
```

```

Tupla3f  a,b,c ;
float    s,l ;

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

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

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

```

1.6. 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...)

Práctica 2

Modelos PLY y Poligonales

2.1. Objetivos

Aprender a:

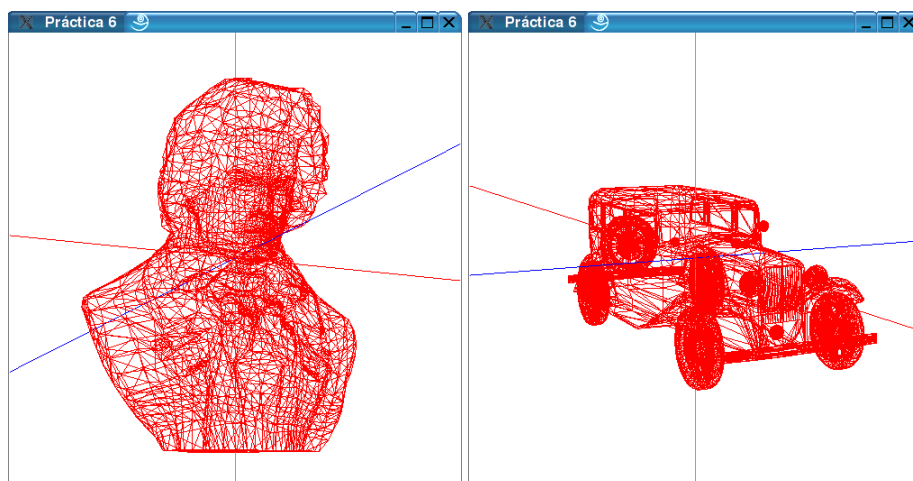
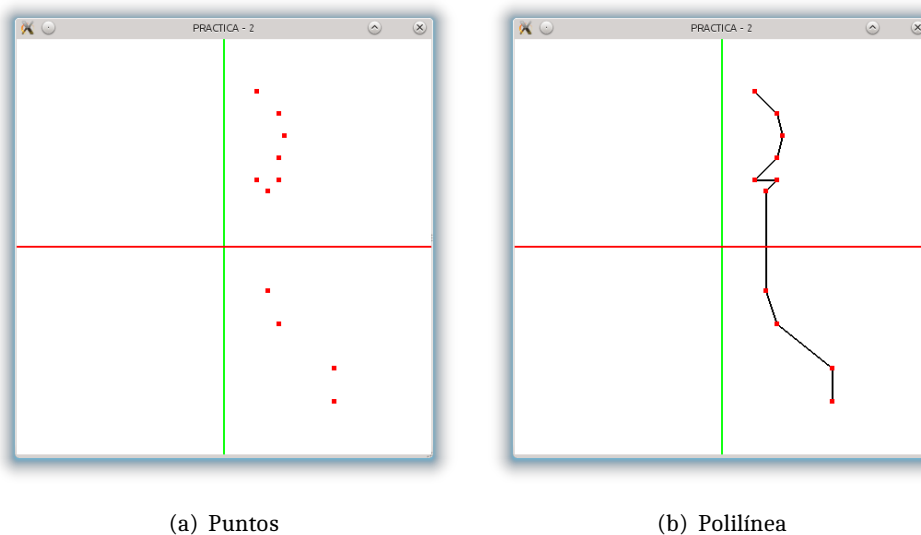
- A leer modelos guardados en ficheros externos en formato PLY (Polygon File Format) y su visualización.
- Modelar objetos sólidos poligonales mediante técnicas sencillas. En este caso se usará la técnica de modelado por revolución de un perfil alrededor de un eje de rotación. Se crearán varios tipos 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 esta 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.

**Figura 2.1:** Objetos PLY.**Figura 2.2:** Perfil inicial.

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

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

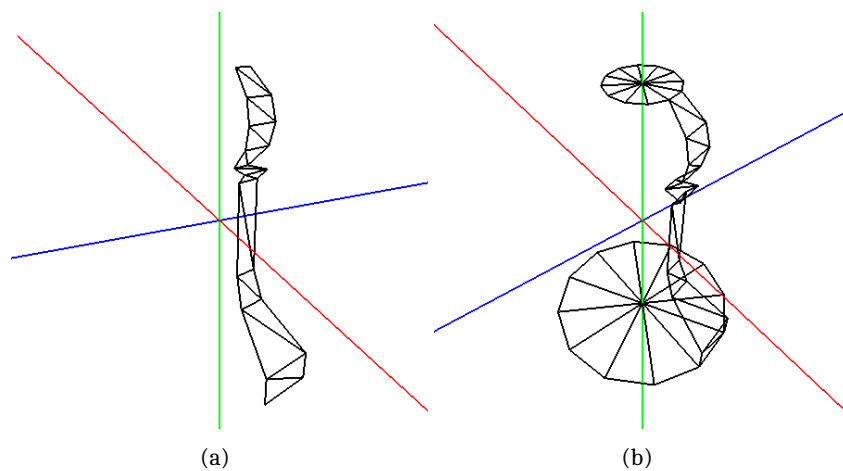


Figura 2.3: Caras del sólido a construir: (a) longitudinales (solo un lado es mostrado) y (b) incluyendo las tapas superior e inferior.

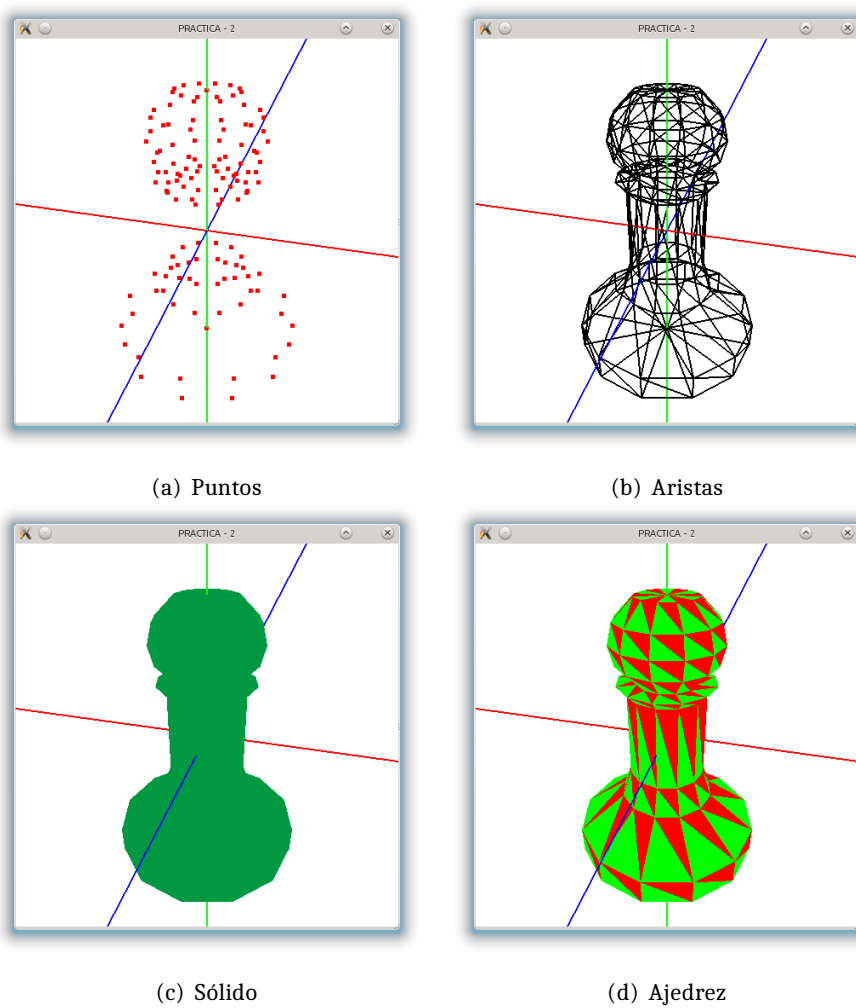


Figura 2.4: Sólido generado por revolución con distintos modos de visualización.

(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).

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

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 perfil 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.

Habrà 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° de la primera copia (a 0°). En este caso, el último perfil (ahora a menos de 360°) se une al primero. Por el contrario, si se decide no cerrar la malla, la última copia del perfil estará efectivamente a 360° , 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 desactivar el uso de modo diferido (VBOs) para visualización (esto es opcional, ver la sección sobre modo diferido)

Las dos primeras ya están implementadas (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 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.

Es 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 de prácticas. Puesto que el binario ejecutable se ejecuta 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 declararlos 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 refieren a archivos proporcionados en la plantilla, están en la carpeta `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 las 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 nombre 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:
    // 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 específico, 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 describen cómo leer 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 cómo 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 Chen en 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...)

Práctica 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.

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

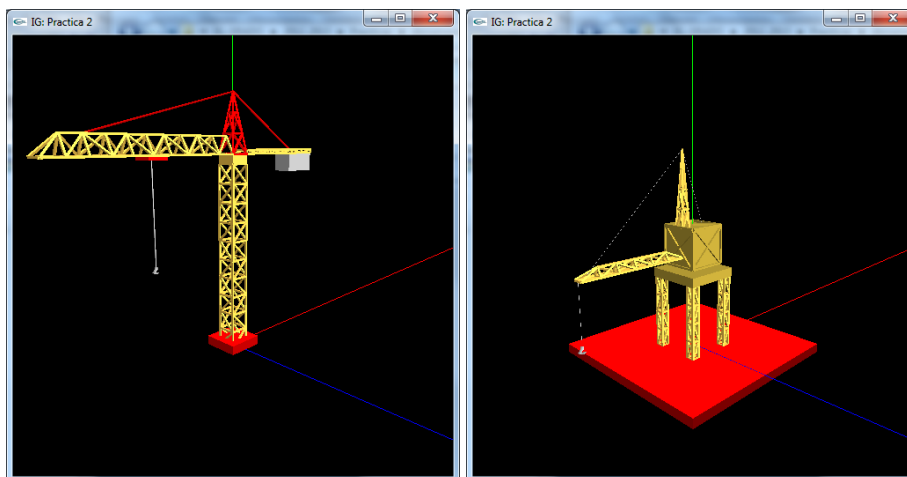


Figura 3.1: Ejemplos del resultado de la práctica 3.

- 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 código necesario para visualizar el modelo jerárquico utilizando los valores de los parámetros del modelo almacenado en la estructura de datos. El alumno podrá utilizar las funciones de visualización implementadas en las anteriores prácticas 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 explícitamente en el código, en lugar de ser leído 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 parámetros puede variar 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 según 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 también 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:

$$q = \begin{cases} c + s \cdot p & \text{si el parámetro no está acotado} \\ c + s \cdot \sin(f2\pi p) & \text{si el parámetro está acotado} \end{cases}$$

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.

Resumiendo, 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-srscs**. 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)`

Esta 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 = ..... ; // 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
);
```

Clase para los nodos de los grafos de escena

Cada nodo del grafo de escena (excepto el nodo raíz) 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 raíz parametrizado

El nodo raíz 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 raíz 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 declarada 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 esta función devuelve **false**, se desactivan las animaciones, 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, mínimo 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.



