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

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 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 el `.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 -DGLFW_USE_RETINA=ON .
make
sudo make install
```

Si no hay errores, esto debe instalar los archivos en `/usr/local/include` (cabeceras `.h`) y en `/usr/local/lib` (archivo objeto `.a`)

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.

Práctica 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`.

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:

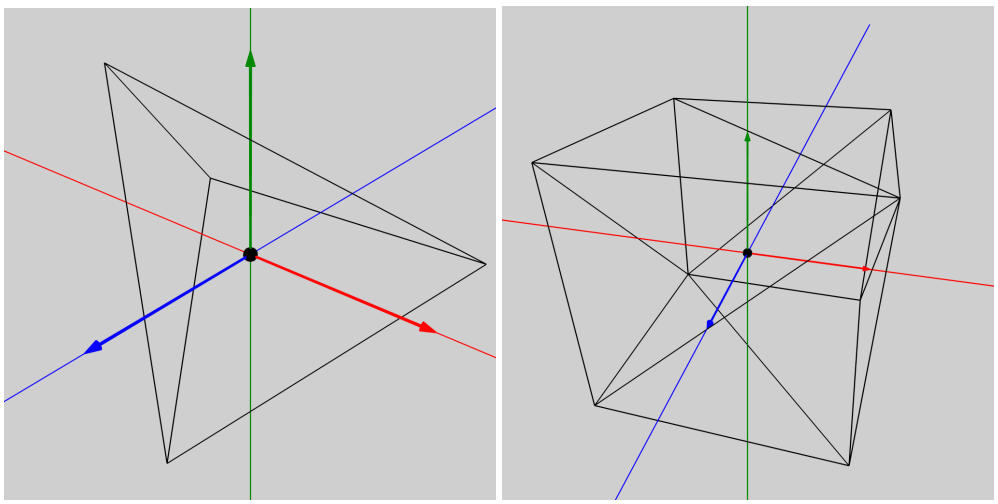
- **tecla q/Q o ESC**: terminar el programa.
- **teclas de cursor**: rotaciones de la cámara entorno al origen.
- **teclas +/-, av.pág/re.pág.**: aumentar/disminuir la distancia de la camara al origen (zoom).

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:



(a) Tetraedro (b) Cubo

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 `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...)

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 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.

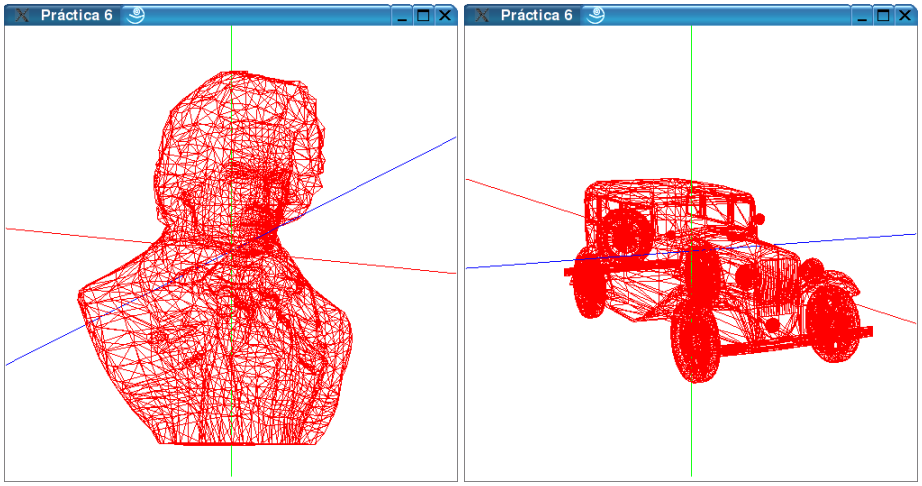
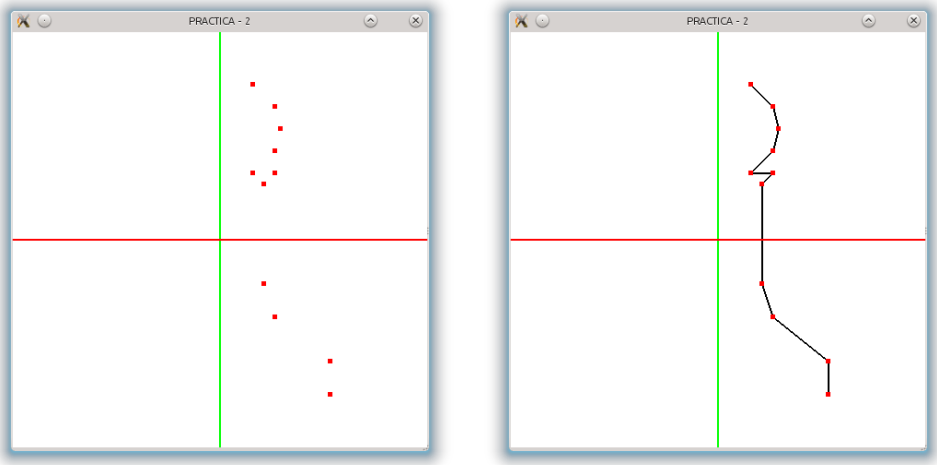


Figura 2.1: Objetos PLY.



(a) Puntos (b) Polilínea

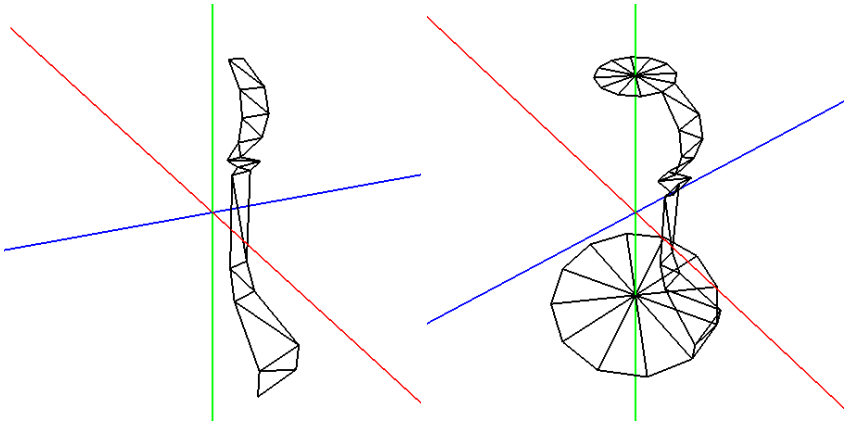
Figura 2.2: Perfil inicial.

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

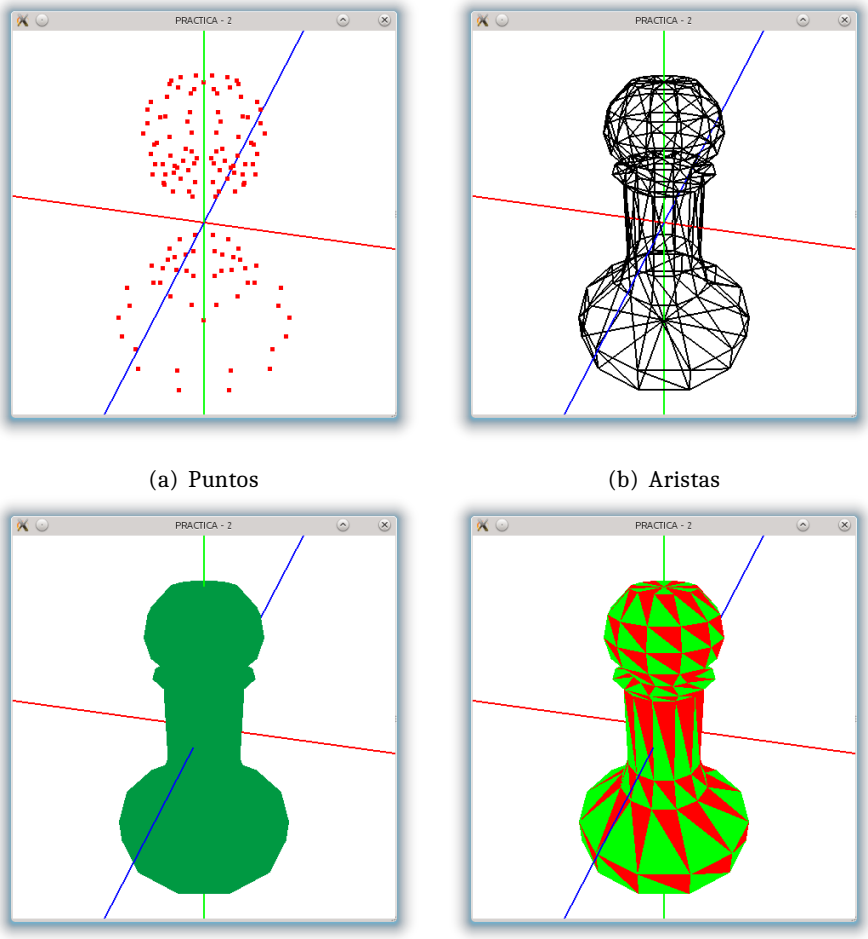
En esta sección se detalla el algoritmo de creación del sólido por revolución (se implementa en un constructor, ver la sección sobre implementación). Partimos de un perfil inicial u original, es una secuencia de m tuplas de coordenadas de vértices en 3D, todas esas coordenadas con $z = 0$

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



(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.

(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 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 1 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 tabla 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.

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 (es decir, esos vértices aparecerán duplicados), y habrá entonces más vértices (habrá $(n+1)m$ vértices), debido a esa copia adicional del perfil. La opción de no cerrar la malla será útil para la práctica 4.

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 refieren 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 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 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...)