

Documentación de Prácticas de IG

Grado en Informática, grupo A, curso 2018-19.



**UNIVERSIDAD
DE GRANADA**

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

0. Índice general.

| | |
|--|-----------|
| Índice. | 3 |
| 1. Prerequisitos software | 4 |
| 1.1. Sistema Operativo Linux | 4 |
| 1.2. Sistema Operativo macOS | 5 |
| 2. Clases para tuplas | 7 |
| 2.1. Introducción | 7 |
| 2.2. Tipos de datos | 7 |
| 2.3. Creación, consulta y actualización | 7 |
| 2.4. Operaciones entre tuplas | 8 |
| 3. Práctica 1 | 9 |
| 3.1. Representación en memoria de Mallas Indexadas | 9 |
| 3.2. Visualización de Mallas Indexadas | 9 |
| 3.2.1. Modos de visualización y otros parámetros | 9 |
| 3.2.2. Modos de envío de primitivas | 9 |
| 3.2.3. Modo inmediato (glDrawElements) | 10 |
| 3.2.4. Modo diferido (glDrawElements) | 10 |
| 4. Práctica 2 | 13 |
| 4.1. Carga de modelos PLY | 13 |
| 4.2. Creación de malla por revolución | 13 |
| 4.3. Clases para mallas obtenidas por revolución | 14 |
| 4.3.1. Método de creación de la malla | 14 |
| 4.3.2. Clases para cilindros, conos y esferas | 15 |
| 5. Práctica 3 | 16 |
| 5.1. Archivos disponibles | 16 |
| 5.2. Teclas a usar | 16 |

| | |
|---|----|
| 5.3. Funciones o métodos a crear en los archivos existentes | 17 |
| 5.4. La clase ObjJerarquico | 18 |
| 5.5. La clase GrafoParam | 18 |

1. 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.

1.1. Sistema Operativo Linux

Compiladores

Para hacer las prácticas de esta asignatura en Ubuntu, en primer lugar debemos de tener instalado algún compilador de C/C++. Se puede usar el compilador de C++ open source CLANG o bien el de GNU (ambos pueden coexistir). Podemos usar `apt` para instalar el paquete `g++` (compilador de GNU) o bien `clang` (compilador del proyecto LLVM). Además, si no está disponible, es conveniente instalar el paquete `make` que proporciona la orden del mismo nombre. Todo esto lo podemos hacer con:

```
sudo apt install g++
sudo apt install make
```

OpenGL

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

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

```
sudo ubuntu-drivers devices
```

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

```
sudo ubuntu-drivers autoinstall
```

Librería GLEW

La librería GLEW es necesaria en Linux para que las funciones de la versión 2.1 de OpenGL y posteriores puedan ser invocadas (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
```

Librería GLUT

La librería GLUT se usa para gestión de ventanas y eventos de entrada. Se puede instalar con el paquete `debian freeglut3-dev`. En Ubuntu, se puede hacer con:

```
sudo apt install freeglut3-dev
```

Librería JPEG

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

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

1.2. Sistema Operativo macOS

Compilador

En primer lugar, para poder compilar los programas fuente en C++, debemos asegurarnos de tener instalado y actualizado **XCode** (conjunto de herramientas de desarrollo de Apple, incluye compiladores de C/C++ y entorno de desarrollo). Una vez instalado XCode, usaremos el compilador de C++ incorporado. Este compilador se invoca desde la línea de órdenes con la orden `clang++` (que es equivalente y tiene los mismos parámetros que la orden `g++` en Linux).

Librerías OpenGL, GLU, GLEW y GLUT

La librería GLEW no es necesaria en este sistema operativo. Respecto a las librerías OpenGL, GLU y GLUT, tanto las cabeceras como la implementación de OpenGL (driver) ya vienen instaladas con el entorno desarrollo XCode. Particular, y usando la terminología de Apple, esos archivos están incorporados en los *frameworks* OpenGL y GLUT.

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.

2. Clases para tuplas.

2.1. Introducción

En el archivo de cabecera `tuplasg.h` están disponibles varios tipos de datos para tuplas de valores reales o enteros. Entre otros, se proporcionan estos tipos:

- **Tupla3f**: tuplas de 3 valores reales (tipo `float`, que es equivalente a `GLfloat`). Son adecuadas para coordenadas 3D de vértices, colores RGB y vectores normales.
- **Tupla2f**: tupla de 2 valores `float`. Es útil para coordenadas de vértices en 2D, o bien para coordenadas de textura de los vértices.
- **Tupla3i**: tupla de 3 valores enteros (tipo `int`). Es útil para la tabla de caras de las mallas indexadas.

2.2. Tipos de datos

Aquí vemos los distintos tipos de datos disponibles:

```
// 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
```

2.3. Creación, consulta y actualización

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 ;

```

2.4. Operaciones entre tuplas

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)

```

3. Práctica 1.

3.1. Representación en memoria de Mallas Indexadas

Suponemos que las mallas indexadas se almacenan en memoria como instancias de la clase `ObjMallaIndexada`. Esta clase tiene como variables de instancia privadas, como mínimo, la tabla de coordenadas de vértices y la tabla de triángulos.

Estas tablas se pueden declarar usando vectores de la librería estándar de C++. En cada entrada contienen tuplas (usando los tipos para tuplas que ya hemos visto). Es decir, se declaran así:

```
std::vector<Tupla3f> vertices ;    // coords de vértices (3 valores 'float' por vértice)
std::vector<Tupla3i> triangulos ; // indices de vertices de triángulos
                                   // (3 valores 'int' por cada cara (triángulo))
```

3.2. Visualización de Mallas Indexadas

Aquí se dan detalles adicionales sobre visualización de mallas indexadas, especialmente la visualización en modo de envío diferido.

3.2.1. Modos de visualización y otros parámetros

La visualización de primitivas requiere, en primer lugar, modificar el estado de OpenGL para ponerlo en el modo de visualización requerido, en función del modo actual (que puede ser, como mínimo, el modo puntos, el modo líneas y el modo sólido).

Para esto se usa una variable entera global que indica el modo actual. Dicha variable se cambia pulsando la tecla **M** (cada vez que se pulsa se cambia al siguiente modo, o bien del último al primero).

Además de fijar el modo actual, es necesario configurar adecuadamente diversos parámetros de OpenGL. Esto se debe a que, aunque en algunos casos los valores por defecto iniciales de esos parámetros son adecuados para la prácticas, es posible que el código de visualización previo al de la malla indexada (p.ej. la visualización de los ejes), deje OpenGL en un estado incorrecto o no esperado.

Las funciones que se deben llamar, como mínimo, son:

- `glColor` para fijar el color con el que aparecerán las primitivas de cualquier tipo.
- `glPolygonMode`: para cambiar el modo de visualización de primitivas de tipo polígono, ponerlo en modo puntos, líneas o relleno (sólido).
- `glShadeModel` para fijar el modo de *shading* (sombreado) en modo *plano* (*flat*) (cada primitiva se visualiza con un único color en todos los píxeles en los que se proyecta, no con degradados). Es necesario asegurarnos que, al menos para esta práctica, el modo de sombreado esté fijado en modo plano antes de visualizar las primitivas.

3.2.2. Modos de envío de primitivas

En las librerías gráficas hay dos formas o modos de enviar las primitivas a la GPU para su visualización:

- **Modo inmediato:** cada vez que se quieren visualizar se envía a la GPU las coordenadas de los vértices, los atributos, y los índices de la tabla de triángulos. Esta opción, por tanto, es lenta.
- **Modo diferido:** antes de visualizar por primera vez, se envía a la GPU todos los datos de las primitivas. Cuando se quiere visualizar, se referencian los datos ya almacenados en la GPU. Esta opción, por tanto, es mucho más rápida si la geometría no cambia, lo cual es el caso de las mallas indexadas de estas prácticas.

En esta sección veremos como enviar en modo inmediato y en modo diferido (en ambos casos con `glDrawElements`) una malla indexada. Suponemos que las tablas usan los tipos para tuplas que ya hemos visto antes.

3.2.3. Modo inmediato (`glDrawElements`)

Asumiendo un objeto tipo malla indexada con tablas de tuplas como las descritas antes, la visualización de la mallas en modo inmediato se puede hacer usando `glDrawElements` como se indica aquí abajo.

```
// habilitar uso de un array de vértices
glEnableClientState( GL_VERTEX_ARRAY );

// indicar el formato y la dirección de memoria del array de vértices
// (son tuplas de 3 valores float, sin espacio entre ellas)
glVertexPointer( 3, GL_FLOAT, 0, vertices.data() );

// visualizar, indicando: tipo de primitiva, número de índices,
// tipo de los índices, y dirección de la tabla de índices
glDrawElements( GL_TRIANGLES, triangulos.size()*3,
                GL_UNSIGNED_INT, triangulos.data() );

// deshabilitar array de vértices
glDisableClientState( GL_VERTEX_ARRAY );
```

Para hacer la visualización usamos los métodos `data` y `size` de los vectores de la librería estándar de C++, que sirven, respectivamente, para obtener un puntero al primer elemento del vector, y para saber cuantos elementos tiene.

3.2.4. Modo diferido (`glDrawElements`)

En el modo diferido (mucho más eficiente), es necesario, una única vez, enviar las tablas de vértices y triángulos a la memoria de la tarjeta gráfica (la GPU), y después, cada vez que se quiera visualizar, se debe de referenciar esas tablas, de forma que no son transferidas de nuevo (la GPU las lee directamente de su memoria). Este es el modo usado normalmente en OpenGL, ya que evita transferencias innecesarias de datos.

El término VBO (*Vertex Buffer Object*) referencia a un bloque de memoria contigua en la GPU, y que se usa para almacenar una o varias tablas. Un VBO puede contener o bien una o varias tablas de atributos de vértices (coordenadas de posición, colores, coordenadas de textura, normales, etc...), o bien una tabla de índices de vértices. No se pueden mezclar en un VBO tablas de índices y de atributos.

Cada VBO tiene asociado un valor entero único (que llamamos *identificador del VBO*), mayor estricto que cero, y que se usa para referenciar un VBO. Cada tabla dentro de un VBO se identifica por

dos valores: el identificador del VBO, y el *offset* o desplazamiento en bytes del primer byte de la tabla, respecto del primer byte del VBO. Los identificadores VBOs son de tipo **GLuint** (equivalente a **unsigned**).

En estas prácticas vamos a usar un VBO distinto para cada tabla de datos (vértices y triángulos). Por tanto, será necesario almacenar en cada objeto de tipo malla indexada los dos identificadores de VBOs, como variables de instancia. Como cada VBO tiene una sola tabla, el *offset* de las tablas será 0. Los identificadores de VBO se deben inicializar a 0, de esta forma, la primera vez que se invoque al método para visualizar en modo diferido, se detectará que los VBOs no están creados (ya que sus identificadores son nulos) y será necesario crearlos, antes de la primera visualización. A partir de esa creación, los identificadores tendrán un valor distinto de 0. Si una malla no se llega a visualizar en modo diferido por cualquier motivo, los VBOs no se crean.

Para facilitar la creación de los VBOs, podemos añadir una función (**CrearVBO**) en el archivo de código fuente de las mallas indexadas. Esta función recibe como parámetros: el tipo de VBOs (un valor que indica si el VBO contiene tablas de atributos de vértice o una tabla de índices), el tamaño en bytes del VBO, y el puntero a la memoria RAM desde donde leer los datos para transferirlos. Como resultado, produce un identificador de VBO.

El tipo de VBO se identifica por un valor **GLuint** que solo puede valer **GL_ARRAY_BUFFER** (para VBOs con tablas de atributos de vértices) o bien **GL_ELEMENT_ARRAY_BUFFER** (para VBOs con una tabla de triángulos, es decir, de índices). El tamaño en bytes se puede calcular usando el método **size** de las tablas y la función **sizeof** aplicada a **float** o **int**. Para el puntero usamos el método **data** de las tablas.

```
GLuint CrearVBO( GLuint tipo_vbo, GLuint tamaño_bytes,
                GLvoid * puntero_ram )
{
    GLuint id_vbo ;                               // resultado: identificador de VBO
    glGenBuffers( 1, & id_vbo );                  // crear nuevo VBO, obtener identificador (nunca 0)
    glBindBuffer( tipo_vbo, id_vbo );              // activar el VBO usando su identificador

    // esta instrucción hace la transferencia de datos desde RAM hacia GPU
    glBufferData( tipo_vbo, tamaño_bytes, puntero_ram, GL_STATIC_DRAW );

    glBindBuffer( tipo_vbo, 0 );                   // desactivación del VBO (activar 0)
    return id_vbo ;                                // devolver el identificador resultado
}
```

En el método **draw_modoS diferido** será necesario verificar si los identificadores de VBOs son 0, y, en ese caso, invocar a **CrearVBO** una vez por cada tabla.

A continuación se puede visualizar la malla, usando para ello **glDrawElements**, de una forma parecida a como se hace en el modo inmediato, excepto que en lugar de punteros a memoria RAM, usamos offsets dentro los VBOs. Estos offsets son siempre 0 pues únicamente alojamos una tabla por VBO, la cual, por tanto, siempre comienza al inicio de dicho VBO. Además, es necesario activar los VBOs antes de la llamada a **glVertexPointer** (para el VBO de vértices) o a **glDrawElements** (para el VBO de índices).

Si suponemos que el identificador del VBO de vértices es **id_vbo_ver**, y el de índices es **id_vbo_tri**, el código quedaría así:

```
// especificar localización y formato de la tabla de vértices, habilitar tabla
```

```
glBindBuffer( GL_ARRAY_BUFFER, id_vbo_ver ); // activar VBO de vértices
glVertexPointer( 3, GL_FLOAT, 0, 0 ); // especifica formato y offset (=0)
glBindBuffer( GL_ARRAY_BUFFER, 0 ); // desactivar VBO de vértices.
glEnableClientState( GL_VERTEX_ARRAY ); // habilitar tabla de vértices

// visualizar triángulos con glDrawElements (puntero a tabla == 0)

glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, id_vbo_tri ); // activar VBO de triángulos
glDrawElements( GL_TRIANGLES, 3*triangulos.size(), GL_UNSIGNED_INT, 0 );
glBindBuffer( GL_ELEMENT_ARRAY_BUFFER, 0 ); // desactivar VBO de triángulos

// desactivar uso de array de vértices
glDisableClientState( GL_VERTEX_ARRAY );
```

4. Práctica 2.

4.1. Carga de modelos PLY

En esta práctica usaremos las funciones para la carga de modelos PLY que se proporcionan en la plantilla de código fuente. Hay dos funciones que sirven para esto, ambas producen vectores de tuplas (con vértices y triángulos), a partir de un nombre de un archivo PLY (con o sin extensión).

Para leer la tabla de coordenadas de vértices y la tabla de triángulos se puede usar la función `ply::read`, sus parámetros son: el nombre del archivo PLY (parámetro de entrada) y las tablas de vértices y triángulos (parámetros de salida). Esta función, al inicio, vacía ambas tablas (si tenían algún contenido, se pierde), y luego inserta en ellas las coordenadas de vértices y los índices (triángulos) leídos del archivo PLY.

Otra opción (útil para leer, desde un archivo PLY, los perfiles de los objetos de revolución) es leer solo la tabla de vértices, usando `ply::read_vertices`. Las declaraciones de ambas funciones son se indica aquí:

```
void read
(
    const std::string &      nombre_archivo_pse, // entrada: nombre de archivo
    std::vector<Tupla3f> &   vertices,           // salida: vector de coords. de vert.
    std::vector<Tupla3i> &   caras               // salida: vector de triángulos (índices)
);

void read_vertices
(
    const std::string &      nombre_archivo_pse, // entrada: nombre de archivo
    std::vector<Tupla3f> &   vertices           // salida: vector de coords. de vert.
);
```

4.2. Creación de malla por revolución

Para la creación de un objeto de revolución, lo más fácil es crear, en primer lugar, la tabla de vértices, en segundo lugar la tabla de triángulos, y después, si es necesario crear las tapas, añadir los vértices en el polo norte y sur y finalmente las caras correspondientes.

Supongamos que, tal y como se indica en el guión, el *perfil original* (leído de un PLY o almacenado en una tabla) tiene M vértices. Dicho vértices (dichas tuplas de coordenadas) los nombramos como $\{\mathbf{o}_0, \dots, \mathbf{o}_{M-1}\}$. Se supone que los vértices se dan de abajo hacia arriba, en el sentido de que las coordenadas Y de esos vértices son crecientes. También se asume que ninguno está en el eje Y , es decir, que todos ellos tienen coordenada X estrictamente mayor que cero.

De ese perfil original haremos N replicas rotadas (a cada una de ellas la llamamos una *instancia del perfil*). Cada uno de estas instancias del perfil forma un ángulo de $2\pi/N$ radianes con la siguiente o anterior. Las instancias del perfil se numeran desde 0 hasta $N - 1$, por tanto, la i -ésima instancia forma un ángulo de $2\pi i/N$ radianes con la instancia número 0, la cual coincide con el perfil original.

Por supuesto se van a crear NM vértices en total. Dichos vértices se insertarán en la tabla de vértices por instancias del perfil (es decir, todos los vértices de una misma instancia aparecen consecuti-

tivos), además las instancias se almacenan en orden (empezando en la instancia 0 hasta la $N - 1$). Por tanto, sabemos que, en la tabla final de vértices, el j -ésimo vértice de la i -ésima instancia tendrá un índice en la tabla igual a $Mi + j$ (donde i va desde 0 hasta $N - 1$ y j va desde 0 hasta $M - 1$).

Para crear los vértices, por tanto, bastará con hacer un bucle doble que recorre todos los pares (i, j) y en cada uno de ellos crea el vértice correspondiente y lo inserta al final de la tabla de vértices (quedará en la entrada con índice $Mi + j$).

El convenio anterior también facilita la creación de la tabla de caras. Para ello, basta hacer un bucle externo, con un índice i que recorre las instancias. Dentro habrá un bucle interno que recorrerá todos los vértices de la instancia número i , excepto el último de ellos. Por cada vértice visitado se insertan en la tabla de caras dos triángulos adyacentes (comparten la arista diagonal). Los único que hay que tener en cuenta es que la instancia siguiente a la última es la primera.

El pseudo-código, por tanto, para la creación de la tabla de vértices será como sigue:

- Partimos de la tabla de vértices vacía.
- Para cada i desde 0 hasta $N - 1$ (ambos incluidos)
 - Para cada j desde 0 hasta $M - 1$ (ambos incluidos)
 - Sea \mathbf{v} = vértice obtenido rotando \mathbf{o}_j un ángulo igual a $2\pi i/N$ radianes.
 - Añadir \mathbf{v} a la tabla de vértices (al final).

Por otro lado, el pseudo-código para la tabla de triángulos visita todos los vértices (excepto el último de cada instancia), y crea los dos triángulos correspondientes:

- Partimos de la tabla de triángulos vacía
- Para cada i desde 0 hasta $N - 1$ (ambos incluidos)
 - Para cada j desde 0 hasta $M - 2$ (ambos incluidos)
 - Sea $a = Mi + j$
 - Sea $b = M((i + 1) \bmod N) + j$
 - Añadir triángulo formado por los índices a, b y $b + 1$.
 - Añadir triángulo formado por los índices $a, b + 1$ y $a + 1$.

Finalmente, si se desea añadir las tapas a la malla, se insertarán el vértice en el *polo sur* (tendrá índice NM) y en el *polo norte* (que tendrá índice $NM + 1$). Después se pueden insertar los N triángulos en la tapa inferior, que conectan el vértice en el polo sur con los N primeros vértices de cada instancia (el primer vértice de la i -ésima instancia tiene índice Mi , para i desde cero hasta $N - 1$). Finalmente, se insertan los triángulos de la tapa superior, que conectan el vértice en el polo norte con el último vértice de cada instancia (el último vértice de la i -ésima instancia tiene como índice $M(i + 1) - 1$, de nuevo con i entre 0 y $N - 1$).

4.3. Clases para mallas obtenidas por revolución

4.3.1. Método de creación de la malla

En la plantilla se encuentra declarada la clase `ObjRevolucion`, derivada de la clase de las mallas indexadas. Será necesario definir en esa clase un nuevo método protegido (`protected`), con esta declaración:

```
void crearMalla( const std::vector<Tupla3f> & perfil_original,
                const int                    num_instancias_perf ) ;
```

Este método usa el vector de tuplas en `perfil_original` (cuyo tamaño se corresponde con M) y construye las tablas de vértices y caras, como se indica arriba (usando `num_instancias_perf` como valor de N , el número de instancias del perfil original).

La clase `ObjRevolucion` ya tiene un constructor que usa un nombre de un archivo PLY. Para implementar este constructor, es necesario usar la función `ply::read_vertices`, obtener el perfil original, y luego invocar el método `crearMalla`.

4.3.2. Clases para cilindros, conos y esferas

Adicionalmente, el guión pide que se creen los objetos de revolución correspondientes a un cilindro, un cono y una esfera. Estos tres objetos se implementan en tres clases derivadas de `ObjRevolucion`, llamadas `Cilindro`, `Cono` y `Esfera`, con las declaraciones en `malla.h` y la implementación en `malla.cc`.

Cada una de estas clases tiene un constructor propio que crea el correspondiente perfil original, y después llama al método `crearMalla` para generar las tablas de vértices y caras. Los tres constructores tienen como parámetros: el número de vértices en el perfil original (M) y el número de instancias del perfil (N). Por tanto, estas tres clases se declaran como sigue:

```
class Cilindro : public ObjRevolucion
{
public:
    Cilindro( const int num_vert_perfil,
              const int num_instancias_perf );
} ;
class Cono : public ObjRevolucion
{
public:
    Cono( const int num_vert_perfil,
          const int num_instancias_perf );
} ;
class Esfera : public ObjRevolucion
{
public:
    Esfera( const int num_vert_perfil,
            const int num_instancias_perf );
} ;
```


5. Práctica 3.

5.1. Archivos disponibles

En esta práctica se debe usar el archivo `jerarquico.h` y `jerarquico.c` que se proporcionan por el profesor, sustituyendo tal cual se descargan a los de la plantilla original (no se deben modificar). En estos archivos se encuentra la definición e implementación completa de la clase `ObjJerarquico`. En dicha clase se gestiona el estado de los valores de los parámetros o grados de libertad, su animación y modificación, y se guarda un puntero a un objeto de tipo `GrafoParam`, que tiene el grafo de escena (parametrizado).

Por otro lado, también se deben de descargar los archivos `grafoparam.cc` y `grafoparam.h`, que implementan la clase `GrafoParam`, la cual incluye el código para un ejemplo muy sencillo de grafo de escena. Dicho código se encapsula totalmente en la citada clase `GrafoParam` (es un *grafo de escena parametrizado*). Este ejemplo debe ser modificado por los estudiantes para sustituirlo por un modelo más complejo con más grados de libertad y un grafo con más nodos. En el ejemplo, se implementan 4 grados de libertad o parámetros (más abajo se dan los detalles).

En la clase `Escena` (archivo `escena.cc`) se creará un puntero a un objeto jerarquico (puntero a `ObjJerarquico`). Dicho objeto se visualiza (cuando es el objeto activo) igual que el resto de objetos, invocando a su método `draw`. Es necesario añadir el puntero al objeto, y la llamada a `draw` en el `switch` que hay en el método `dibujar_objeto_actual` de la clase `Escena`.

En este texto se usa el término *parámetro* del modelo jerárquico como sinónimo de *grado de libertad*.

5.2. Teclas a usar

En esta práctica intentaremos reducir el número de teclas distintas a usar, y asimismo intentaremos escribir código que sea lo más independiente posible del número de grados de libertad o parámetros del modelo implementado. Por ello se define el *parámetro actual* del objeto jerárquico como un número entero que indica cual es el grado de libertad o parámetro que se modificará manualmente con las teclas correspondientes, en lugar de definir un par de teclas específicas para incrementar o decrementar cada uno de los grados de libertad del modelo. Con una tecla podemos cambiar el parámetro actual. La correspondiente variable es la variable (privada) `i_param_act` en la clase `ObjJerarquico`.

Por tanto el estudiante definirá en `escena.cc` acciones para las siguientes teclas (en el método `teclaPulsada`):

- Tecla **P** (mayúscula o minúscula): activar el parámetro siguiente al actual (o el primero si el actual es el último). Esto se hace llamando al método `siguienteParametro` de la clase `ObjJerarquico`.
- Tecla **A** (mayúscula o minúscula): activa o desactiva las animaciones. Esto se hace llamando al método `conmutarAnimaciones` de la clase `Escena`, el cual debe ser implementado por el estudiante en `escena.cc`
- Tecla **Z**: incrementa el valor del parámetro actual del objeto jerárquico. Esto se hace llamando al método `incrementaParamAct` de `ObjJerarquico` (ya implementado). Solo funciona si

las animaciones están desactivadas.

- Tecla **z**: decrementa el valor del parámetro actual del objeto jerárquico. Esto se hace llamando al método `decrementaParamAct` de `ObjJerarquico` (ya implementado). Solo funciona si las animaciones están desactivadas.
- Tecla **>**: incrementar el valor de incremento/decremento usado para las animaciones y las modificaciones del parámetro actual. Esto se hace llamando al método `acelerar` de `ObjJerarquico` (ya implementado).
- Tecla **<**: decrementar el valor de incremento/decremento usado para las animaciones y las modificaciones del parámetro actual. Esto se hace llamando al método `decelerar` de `ObjJerarquico` (ya implementado).

Estas teclas solo funciona cuando el objeto actual es el objeto jerarquico.

5.3. Funciones o métodos a crear en los archivos existentes

El estudiante deberá definir los siguientes nuevos métodos o funciones (todos ellos devuelven `void` y ninguno tiene parámetros)

Función `funcion_desocupado` (en `practicass.cc`)

Este función será la función gestora del *evento desocupado* (*idle event*), es decir, es la función que se llamará periódicamente con las animaciones activadas. En este función simplemente se invocará al método `mgeDesocupado` de la clase `Escena` (ver más abajo). Si la escena no hubiera sido creada al ser llamado, no hace nada.

Es conveniente añadir la declaración de esta función en `aux.h` (simplemente añadir una línea con: `void funcion_desocupado() ;`

Método `mgeDesocupado` de la clase `Escena`

Este método (público) se llamará cada cierto tiempo en los intervalos de tiempo en los que las animaciones están activadas, desde `funcion_desocupado`. Se encarga de actualizar el estado de los parámetros del objeto jerárquico, lo cual se hace llamando al método `actualizarEstado` de la clase `ObjJerarquico` (ya implementado).

Después, será necesario llamar a la función `glutPostRedisplay` para forzar la visualización de la escena, con el modelo ya actualizado al siguiente cuadro.

Método `conmutarAnimaciones` de la clase `Escena`.

Este método (privado) de la clase `Escena` se ejecuta cuando el usuario pulsa la tecla correspondiente a activar o desactivar las animaciones.

Para implementar este método es necesario incluir una variable de instancia (privada) en `Escena`, que valgo `true` si las animaciones están activadas y `false` en caso contrario.

Si el objeto actual de la escena no es el objeto jerárquico, no se hace nada (se puede imprimir un mensaje de advertencia). Cuando el objeto actual es el objeto jerárquico, se niega el valor de la variable lógica citada.

Si las animaciones se han activado (la variable ha pasado a `true`), se debe invocar el método `inicioAnimaciones` de la clase `ObjJerarquico`, y a continuación se usa GLUT para activar la función desocupado, haciendo:

```
glutIdleFunc( funcion_desocupado );
```

Si las animaciones se han desactivado (la variable ha pasado a **false**), se debe de desactivar la función desocupado, con:

```
glutIdleFunc( nullptr );
```

5.4. La clase **ObjJerarquico**

La clase **ObjJerarquico** ya se encuentra completamente implementada en el archivo **jerarquico.cc** (que se descarga de PRADO). Esta clase encapsula un puntero al grafo de escena (variable de instancia privada **objparam**), y tiene un método **draw** que simplemente invoca al método **draw** del grafo de escena parametrizado apuntado por **objparam**. En el constructor de **ObjJerarquico** se debe crear dicho objeto **objparam**, llamando a su constructor.

La funcionalidad principal de la clase **ObjJerarquico** es gestionar las actualizaciones de los valores de los parámetros para las modificaciones por teclado y las animaciones. Para ello se ha diseñado un esquema que permite que el código de **ObjJerarquico** sea independiente del número y la naturaleza de los parámetros del grafo de escena parametrizado, simplemente se asume que dicho grafo tiene un número de parámetros determinado, cada uno de ellos es un valor real.

Para lograr el objetivo anterior, el grafo de escena parametrizado tiene un método **numParametros** que permite conocer cuantos parámetros o grados de libertad tiene. Con este dato, en **ObjJerarquico** se crea un vector de flotantes (**float**), con tantas entradas como parámetros, e inicializados a 0. A cada uno de estos valores lo llamamos *valor no acotado* del correspondiente parámetro. Los nombraremos como $\{t_0, t_1, \dots, t_{n-1}\}$, donde n es el número de parámetros.

Cuando las animaciones están activadas, cada vez que actualiza el estado del objeto jerárquico (en el método **actualizarEstado**), se añade a cada valor no acotado t_i el valor $s\Delta$, donde Δ es una variable (que inicialmente vale la unidad), y s es el tiempo real en segundos (**float**) transcurrido desde que el instante en que se activaron las animaciones o desde la última vez que se llamó a **actualizarEstado** (lo último que haya ocurrido). De esta forma, los valores no acotados de los parámetros crecen en Δ unidades por segundo todos ellos, durante los intervalos en los que las animaciones están activadas y se invoca periódicamente a **actualizarEstado**. Por eso estos valores no acotados se pueden asimilar al tiempo durante las animaciones. Cuando un parámetro se modifica individualmente por teclado, su valor no acotado se incrementa o decrementa en $\Delta/20$ unidades. Este es el motivo por el cual en un instante determinado puede ocurrir que no todos los t_i sean iguales.

Cuando se usan las teclas para acelerar o decelerar las animaciones, lo que se hace es incrementar o decrementar el valor de Δ .

Los valores no acotados de los parámetros se usan para calcular los llamados *valores efectivos* de dichos parámetros. Los valores efectivos se calculan en la clase **GrafoParam**, y son estos valores efectivos los que se usan para definir las transformaciones que dependen de los parámetros (ver la siguiente sub-sección).

5.5. La clase **GrafoParam**

Los estudiantes deben de modificar la clase **GrafoParam** para definir un modelo más complejo, partiendo del ejemplo sencillo que se proporciona.

El constructor de **GrafoParam** debe crear las mallas indexadas necesarias, y que forman los nodos terminales del grafo de escena.

El modelo se visualiza mediante el código del método **draw**, que es invocada a su vez desde el método **draw** de **ObjJerarquico**. Esta visualización usa las funciones de OpenGL para gestionar la pila de transformaciones que permite ir actualizando la matriz *modelview*, como se detalla en los apuntes. Se deben de definir métodos que visualizan sub-grafos, por claridad en el código y a veces para evitar replicar el mismo código en varios sitios (se valorará que el grafo de escena no tenga muchos más nodos de los necesarios).

El método **numParametros** de la clase **GrafoParam** debe de modificarse para devolver el número de parámetros o grados de libertad del modelo.

Los valores de los parámetros se declaran como variables de instancia **float** (privadas) de la clase **GrafoParam**. Estos valores codifican ángulos, distancias, factores de escala, etc....(ver en el ejemplo las correspondientes variables de instancia). Se usan para definir las transformaciones que dependen de los grados de libertad. A estos valores se les llama *valores efectivos* de los parámetros.

Los valores efectivos de los parámetros se calculan a partir de los valores no acotados de los mismos. Esto se hace en el método **actualizarValorEfe** de **GrafoParam** (cambia el valor de uno de los parámetros). Este método es el único que modifica los valores efectivos.

Cada valor efectivo de un parámetro (llamemosle p_i) depende de su correspondiente valor no acotado t_i , mediante alguna función f_i , de forma que $p_i = f_i(t_i)$. Cada función p_i puede ser una dependencia simplemente lineal, por ejemplo, de la forma:

$$p_i = a + vt_i$$

de manera que, inicialmente, p_i vale a (ya que al principio del programa t_i será cero), y al activar las animaciones p_i crece v unidades por segundo de tiempo real (mientras no se cambie el valor inicial de Δ , que es la unidad).

En otros casos, nos interesa que el valor efectivo oscile entre dos valores distintos, llamemosles m y M , con $m < M$ (típicamente ocurre con desplazamientos o distancias que no queremos que crezcan indefinidamente, o por ejemplo el ángulo de giro de un péndulo). En este caso, el valor efectivo p_i ya sí estará acotado, a diferencia del valor no acotado. Se pueden usar varias funciones para esto, lo más fácil es usar el seno, que produce oscilaciones suaves (en el sentido que la velocidad baja a cero al acercarnos a los valores extremos). En este caso, la relación entre p y t sería esta:

$$p_i = a + b \sin(w2\pi t_i)$$

donde $a = (M + m)/2$ es el valor inicial (central) y $b = (M - m)/2$ es la semiamplitud del movimiento. El valor real w es la frecuencia, es decir, es el número de oscilaciones por segundo de animación (cuando $\Delta = 1$), y puede ser mayor o menor que la unidad (o incluso negativo, nunca 0).

El estudiante puede definir otras funciones más complejas para calcular p_i a a partir de t_i .

Para comprender mejor la relación entre los parámetros no acotados y los efectivos, se puede ver el código de **actualizarValorEfe** del ejemplo que se proporciona, que tiene cuatro parámetros: dos ángulos y dos distancias, y en los cuales hay valores efectivos no acotados (el ángulo de un cubo), y otros son oscilantes.