srcs-prac/FuenteLuz.cpp

```cpp
#include "FuenteLuz.hpp"

void ColeccionFL::activar() {
  glEnable(GL_LIGHTING);
  glEnable(GL_NORMALIZE);
  glDisable(GL_COLOR_MATERIAL);
  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
  glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

  for (unsigned f = 0; f < fuentes.size(); ++f) {
    fuentes.at(f)->activar();
  }
}

void FuenteLuz::activar() {
  glEnable(GL_LIGHT0 + index);

  glLightfv(GL_LIGHT0 + index, GL_AMBIENT, colores[0]);
  glLightfv(GL_LIGHT0 + index, GL_DIFFUSE, colores[1]);
  glLightfv(GL_LIGHT0 + index, GL_SPECULAR, colores[2]);

  if (posvec[3] == 1) { // => posicional
    glLightfv(GL_LIGHT0 + index, GL_POSITION, posvec);
  } else {
    // Entramos en la modelview
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glRotatef(longi, 0.0, 1.0, 0.0);
    glRotatef(lati, -1.0, 0.0, 0.0);
    glLightfv(GL_LIGHT0 + index, GL_POSITION, Tupla4f(0, 0, 1, 0));

    glPopMatrix();
  }
}

FuentePosicional::FuentePosicional(int index, const Tupla3f& posicion)
  :FuenteLuz(index, 0, 0, Tupla4f(posicion(X), posicion(Y), posicion(Z), 1)) {
  for (int i = 0; i < 3; i++)
    // Luz azulada
    colores[i] = Tupla4f(0.5, 0.7, 0.8, 1);
}

FuenteDireccional::FuenteDireccional(int index, float longi, float lati)
  :FuenteLuz(index, longi, lati, Tupla4f(0, 0, 0, 0)) {
  for (int i = 0; i < 3; i++)
    // Luz blanca
    colores[i] = Tupla4f(0.7, 0.7, 0.7, 1);
}

void FuenteDireccional::variarAngulo(unsigned angulo, float incremento) {
  if (angulo)
    longi += incremento;
  else
    lati += incremento;

  activar();
```

```
}

ColeccionFuentesP4::ColeccionFuentesP4() {
  fuentes.push_back(
    new FuenteDireccional(0, 0, 0)
  );
  fuentes.push_back(
    new FuentePosicional(1, Tupla3f(5, 5, 0))
  );
}
```

srcs-prac/main.cpp

```
// *****************************************************************
// **
// ** Informática Gráfica, curso 2014-15
// ** Carlos Ureña Almagro
// **
// ** Función 'main', inicialización y gestores de eventos
// **
// *****************************************************************

// includes de C/C++

#include <cctype>   // toupper
#include <string>   // std::string
#include <iostream> // std::cout
#include <fstream>  // ifstream
#include <cmath>    // fabs

// includes en ../include
#include "aux.hpp"  // include cabeceras de opengl / glut / glut / glew

// includes de archivos en el directorio de trabajo (de las prácticas)
#include "practica1.hpp"
#include "practica2.hpp"
#include "practica3.hpp"
#include "practica4.hpp"

// evita la necesidad de escribir std::
using namespace std ;

// *****************************************************************
// **
// ** Variables globales
// ** (se inicializan en las funciones de inicialización)
// **
// *****************************************************************

// variables que definen la posicion de la camara en coordenadas polares

float
   camara_angulo_x ,   // angulo de rotación entorno al eje X
   camara_angulo_y ;   // angulo de rotación entorno al eje Y

// ---------------------------------------------------------------------
// variables que definen el view-frustum (zona visible del mundo)

float
   frustum_factor_escala ,  // factor de escala homogeneo que se aplica al frustum (sirve para aleja
```

```
   frustum_ancho ,          // ancho, en coordenadas del mundo
   frustum_dis_del ,        // distancia al plano de recorte delantero (near)
   frustum_dis_tra ;        // distancia al plano de recorte trasero (far)

// ---------------------------------------------------------------------
// variables que determinan la posicion y tamaño inicial de la ventana
// (el tamaño se actualiza al cambiar el tamaño durante la ejecución)

int
   ventana_pos_x  ,  // posicion (X) inicial de la ventana, en pixels
   ventana_pos_y  ,  // posicion (Y) inicial de la ventana, en pixels
   ventana_tam_x  ,  // ancho inicial y actual de la ventana, en pixels
   ventana_tam_y  ;  // alto inicial actual de la ventana, en pixels

// ---------------------------------------------------------------------

unsigned
   modo_vis  ,  // modo de visualización (0,1,3,4)
   practica_actual ;  // practica actual (cambiable por teclado) (1,2,3,4,5)

const unsigned NUM_MODOS = 6;
const std::string modos[NUM_MODOS] = {
  "Puntos",
  "Alambre",
  "Relleno",
  "Ajedrez",
  "Sombreado plano",
  "Sombreado suave"
};

const unsigned NUM_PRACTICAS = 4;

// *********************************************************************
// **
// ** Funciones auxiliares:
// **
// *********************************************************************


// fija la transformación de proyeccion (zona visible del mundo == frustum)

void FijarProyeccion()
{
   const GLfloat ratioYX = GLfloat( ventana_tam_y )/GLfloat( ventana_tam_x );

   CError();

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

   // (3) proyectar en el plano de visión
   glFrustum
   (  -frustum_ancho,
      +frustum_ancho,
      -frustum_ancho*ratioYX,
      +frustum_ancho*ratioYX,
      +frustum_dis_del,
      +frustum_dis_tra
   );
```

```
   // (2) situar el origen (0,0,0), en mitad del view frustum
   //     (en la rama negativa del eje Z)
   glTranslatef( 0.0,0.0,-0.5*(frustum_dis_del+frustum_dis_tra));

    // (1) aplicar factor de escala
   glScalef( frustum_factor_escala, frustum_factor_escala,  frustum_factor_escala );

   CError();
}


// ---------------------------------------------------------------------
// fijar viewport y proyección (viewport ocupa toda la ventana)

void FijarViewportProyeccion()
{
   glViewport( 0, 0, ventana_tam_x, ventana_tam_y );
   FijarProyeccion() ;
}

// ---------------------------------------------------------------------
// fija la transformación de vista (posiciona la camara)

void FijarCamara()
{

   CError();
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();

   glRotatef(camara_angulo_x,1,0,0);
   glRotatef(camara_angulo_y,0,1,0);

   CError();
}

// ---------------------------------------------------------------------
// dibuja los ejes utilizando la primitiva grafica de lineas

void DibujarEjes()
{
   const float long_ejes = 30.0 ;

   // establecer modo de dibujo a lineas (podría estar en puntos)
   glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );

   // dibujar tres segmentos
   glBegin(GL_LINES);
      // eje X, color rojo
      glColor3f( 1.0, 0.0, 0.0 );
      glVertex3f( -long_ejes, 0.0, 0.0 );
      glVertex3f( +long_ejes, 0.0, 0.0 );
      // eje Y, color verde
      glColor3f( 0.0, 1.0, 0.0 );
      glVertex3f( 0.0, -long_ejes, 0.0 );
      glVertex3f( 0.0, +long_ejes, 0.0 );
      // eje Z, color azul
      glColor3f( 0.0, 0.0, 1.0 );
      glVertex3f( 0.0, 0.0, -long_ejes );
      glVertex3f( 0.0, 0.0, +long_ejes );
   glEnd();
```

```
   // bola en el origen, negra
   glPolygonMode( GL_FRONT_AND_BACK, GL_FILL );
   glColor3f(0.0,0.0,0.0);
   glutSolidSphere(0.01,8,8);

}

// ---------------------------------------------------------------------
// asigna el color de fondo actual a todos los pixels de la ventana

void LimpiarVentana()
{
   glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}

// ---------------------------------------------------------------------
// dibuja los objetos de la escena

void DibujarObjetos()
{
   switch( practica_actual )
   {
      case 1 :
         P1_DibujarObjetos( modo_vis ) ; // definido en 'practica1.hpp'
         break ;
      // falta: case 2: ... case 3: ..... case 4: ..... case 5: .....
      //
      case 2:
         P2_DibujarObjetos(modo_vis);
         break;
      case 3:
         P3_DibujarObjetos(modo_vis);
         break;
      case 4:
         P4_DibujarObjetos(modo_vis);
         break;

      default :
         cout << "El valor de 'practica_actual' (" << practica_actual  << ") es incorrecto" << endl
         break ;
   }
}


// *********************************************************************
// **
// ** Funciones gestoras de eventos
// **
// *********************************************************************

// F.G. del evento de redibujado (se produce cuando hay que volver a
// dibujar la ventana, tipicamente tras producirse otros eventos)

void FGE_Redibujado()
{
   using namespace std ;
   //cout << "redibujado......" << endl << flush ;
   FijarViewportProyeccion() ; // necesario pues la escala puede cambiar
   FijarCamara();
   LimpiarVentana();
```

```
   DibujarEjes() ;
   DibujarObjetos();
   glutSwapBuffers();
}

// ---------------------------------------------------------------------
// F.G. del evento de cambio de tamaño de la ventana

void FGE_CambioTamano( int nuevoAncho, int nuevoAlto )
{
   // guardar nuevo tamaño de la ventana
   ventana_tam_x  = nuevoAncho;
   ventana_tam_y  = nuevoAlto ;

   // reestablecer frustum, viewport y proyección
   FijarViewportProyeccion();

   // forzar un nuevo evento de redibujado, para actualizar ventana
   glutPostRedisplay();
}

// ---------------------------------------------------------------------
// F.G. del evento de pulsación de la tecla normal
//
// parámetros:
//       tecla: carácter corresondiente a la tecla (minúscula)
//       x_raton, y_raton : posición del ratón al pulsar

void FGE_PulsarTeclaNormal( unsigned char tecla, int x_raton, int y_raton )
{

   bool redibujar = true ; // true si al acabar de procesar el evento resulta que es necesario redibu
   switch (toupper(tecla))
   {
      case 'P':
      case 13: // Tecla ENTER
        practica_actual = 1 + practica_actual % NUM_PRACTICAS;
        std::cerr << "Práctica: " << practica_actual << std::endl;
        break;

      // Modo visualización: puntos, líneas, relleno y ajedrez
      case 'A':
        modo_vis = 0;
        break;
      case 'S':
        modo_vis = 1;
        break;
      case 'D':
        modo_vis = 2;
        break;
      case 'F':
        modo_vis = 3;
        break;
      case 'K':
        modo_vis = 4;
        break;
      case 'L':
        modo_vis = 5;
        break;

      case 'M':
```

```
        ++modo_vis %= NUM_MODOS;
        std::cerr << "Modo visualización: " << modos[modo_vis] << std::endl;
        break;

      case 'Q' :
         exit( 0 );
         break ;
      case '+' :
         frustum_factor_escala *= 1.05;
         break;
      case '-' :
         frustum_factor_escala /= 1.05;
         break;
      default:
         redibujar = false ;
         switch( practica_actual )
         {
            case 1 :
               redibujar = P1_FGE_PulsarTeclaNormal( tecla ) ; // true si es necesario redibujar
               break ;
            // falta: case 2, case 3, etc....
            case 2:
               redibujar = P2_FGE_PulsarTeclaNormal(tecla);
               break;
            case 3:
               redibujar = P3_FGE_PulsarTeclaNormal(tecla);
               break;
            case 4:
               redibujar = P4_FGE_PulsarTeclaNormal(tecla);
               break;

            default :
               redibujar = false ; // la tecla no es de la práctica activa (no es necesario redibuja
         }
         break ;
   }
   using namespace std ;
   //cout << "tecla normal....." << frustum_factor_escala << endl ;

   // si se ha cambiado algo, forzar evento de redibujado
   if (redibujar)
      glutPostRedisplay();
}

// ---------------------------------------------------------------------
// F.G. del evento de pulsación de la tecla especial
//
// parámetros:
//       tecla: código GLUT de la tecla pulsada
//       x_raton, y_raton : posición del ratón al pulsar


void FGE_PulsarTeclaEspecial( int tecla, int x_raton, int y_raton )
{
   bool redisp = true ;
   const float da = 5.0 ; // incremento en grados de ángulos de camara

   switch ( tecla )
   {
      case GLUT_KEY_LEFT:
         camara_angulo_y = camara_angulo_y - da ;
```

```
         break;
      case GLUT_KEY_RIGHT:
         camara_angulo_y = camara_angulo_y + da ;
         break;
      case GLUT_KEY_UP:
         camara_angulo_x = camara_angulo_x - da ;
         break;
      case GLUT_KEY_DOWN:
         camara_angulo_x = camara_angulo_x + da ;
         break;
      case GLUT_KEY_PAGE_UP:
         frustum_factor_escala *= 1.05;
         break;
      case GLUT_KEY_PAGE_DOWN:
         frustum_factor_escala /= 1.05;
         break;
      default:
         redisp = false ;
         break ;
    }
   using namespace std ;

   // si se ha cambiado algo, forzar evento de redibujado
   if ( redisp )
      glutPostRedisplay();
}

// *****************************************************************************
// **
// ** Funciones de inicialización
// **
// *****************************************************************************

// inicialización de GLUT: creación de la ventana, designar FGEs

void Inicializa_GLUT( int argc, char * argv[] )
{

   // inicializa variables globales usadas en esta función (y otras)
   ventana_pos_x  = 50 ;
   ventana_pos_y  = 50  ;
   ventana_tam_x  = 800 ;  // ancho inicial y actual de la ventana, en pixels
   ventana_tam_y  = 800 ;  // alto inicial actual de la ventana, en pixels

   // inicializa glut:
   glutInit( &argc, argv );

   // establece posicion inicial de la ventana:
   glutInitWindowPosition( ventana_pos_x, ventana_pos_y );

   // establece tamaño inicial de la ventana:
   glutInitWindowSize( ventana_tam_x, ventana_tam_y );

   // establece atributos o tipo de ventana:
   glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );

   // crea y visualiza una ventana:
   glutCreateWindow("Practicas IG GIM (15-16)");

   // establece función gestora del evento de redibujado:
   glutDisplayFunc( FGE_Redibujado );
```

```
   // establece función gestora del evento de cambio de tamaño:
   glutReshapeFunc( FGE_CambioTamano );

   // establece función gestora del evento de pulsación de tecla normal:
   glutKeyboardFunc( FGE_PulsarTeclaNormal );

   // establece función gestora del evento de pulsación de tecla especial:
   glutSpecialFunc( FGE_PulsarTeclaEspecial );
}

// ---------------------------------------------------------------------
// Inicialización de las variables globales del programa

void Inicializa_Vars( )
{
   // inicializar parámetros del frustum
   frustum_dis_del         = 0.1 ;
   frustum_dis_tra         = 10.0;
   frustum_ancho           = 0.5*frustum_dis_del ;
   frustum_factor_escala   = 2.0 ;

   // inicializar parámetros de la cámara
   camara_angulo_x = 0.0 ;
   camara_angulo_y = 0.0 ;

   // inicializar práctica actual y modo de visualización inicial
   practica_actual = NUM_PRACTICAS ;
   modo_vis = 0 ;
}

// ---------------------------------------------------------------------
// inicialización de OpenGL

void Inicializa_OpenGL( )
{
   // borrar posibles errores anteriores
   CError();

   // habilitar test de comparación de profundidades para 3D (y 2D)
   // es necesario, no está habilitado por defecto:
   // https://www.opengl.org/wiki/Depth_Buffer
   glEnable( GL_DEPTH_TEST );

   // establecer color de fondo: (1,1,1) (blanco)
   glClearColor( 1.0, 1.0, 1.0, 1.0 ) ;

   // establecer color inicial para todas las primitivas
   glColor3f( 0.7, 0.2, 0.4 ) ;

   // establecer ancho de línea
   glLineWidth( 1.0 );

   // establecer tamaño de los puntos
   glPointSize( 2.0 );

   // establecer modo de visualización de prim.
   // (las tres posibilidades son: GL_POINT, GL_LINE, GL_FILL)
   glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );

   // establecer la cámara, la proyección y el viewport
```

```
   FijarViewportProyeccion() ;
   FijarCamara() ;

   // imprimir datos del hardware y la implementación de OpenGL
   using namespace std ;
   cout  << "Datos de versión e implementación de OpenGL" << endl
         << "  implementación de : " << glGetString(GL_VENDOR)  << endl
         << "  hardware          : " << glGetString(GL_RENDERER) << endl
         << "  version de OpenGL : " << glGetString(GL_VERSION) << endl
         << "  version de GLSL   : " << glGetString(GL_SHADING_LANGUAGE_VERSION) << endl
         << flush ;

   // ya está
   CError();
}

// ---------------------------------------------------------------------
// Código de inicialización (llama a los dos anteriores, entre otros)

void Inicializar( int argc, char *argv[] )
{
   // inicializa las variables del programa
   Inicializa_Vars() ;

   // glut (crea la ventana)
   Inicializa_GLUT( argc, argv ) ;

   // opengl: define proyección y atributos iniciales
   Inicializa_OpenGL() ;

   // inicializar práctica 1.
   P1_Inicializar( argc, argv ) ;

   // inicializar práctica 2.
   P2_Inicializar( argc, argv ) ;

   // inicializar práctica 3.
   P3_Inicializar( argc, argv ) ;

   // inicializar práctica 4.
   P4_Inicializar( argc, argv ) ;
}

// ***********************************************************************
// **
// ** Función principal
// **
// ***********************************************************************


int main( int argc, char *argv[] )
{
   // incializar el programa
   Inicializar( argc, argv ) ;

   // llamar al bucle de gestión de eventos de glut, tiene el
   // control hasta el final de la aplicación
   glutMainLoop();

   // ya está
   return 0;
```

```
}
```

srcs-prac/MallaBarrido.cpp

```
#include "MallaBarrido.hpp"
#include "file_ply_stl.hpp"
#include <cmath>

void MallaBarrido::construir(unsigned num_traslaciones) {
  vertex_coords.erase(vertex_coords.begin(), vertex_coords.end());
  indexes.erase(indexes.begin(), indexes.end());

  const unsigned Nc = figura.size(), Nv = figura.size()/3;

  // Creamos los vértices trasladando la figura base por el vector
  // de traslación
  for (unsigned i = 0; i < Nc; i += 3) {
    Tupla3f punto_actual(figura[i], figura.at(i + 1), figura.at(i + 2));

    for (unsigned t = 0; t < num_traslaciones; t++)
      vertex_coords.push_back(punto_actual + (vector_traslacion * t));
  }

  // Unimos cada vértice con su lateral y los del trasladado
  for (unsigned i = 0; i < Nv - 1; i++)
    for (unsigned t = 0; t < num_traslaciones - 1; t++) {
      indexes.push_back(Tupla3i(
        i * num_traslaciones + t,
        i * num_traslaciones + t + 1, // enfrente
        (i + 1)%Nv * num_traslaciones + t + 1 // enfrente - siguiente
      ));
      indexes.push_back(Tupla3i(
        i * num_traslaciones + t,
        (i + 1) * num_traslaciones + t, // siguiente
        (i + 1) * num_traslaciones + t + 1 // enfrente - siguiente
      ));
    }
}

MallaBarrido::MallaBarrido(const char * filename, unsigned num_traslaciones, Tupla3f vect, std::strin
  nombre_obj = nombre;

  ply::read_vertices(filename, figura);
  vector_traslacion = vect;
  construir(num_traslaciones);
}
```

srcs-prac/MallaInd.cpp

```
#include "MallaInd.hpp"
#include "aux.hpp"
#include <algorithm>

/*************************
Método visualizar utilizando
la función glDrawElements.
*************************/
void MallaInd::visualizar(ContextoVis cv) {
  // Comprobación de parámetros
  if (cv.modo_vis > 5 || cv.modo_vis < 0) {
```

```
    cv.modo_vis = 2; // Asumimos sólido
  }

  const GLenum modo[] = {GL_POINT, GL_LINE, GL_FILL, GL_FILL, GL_FILL, GL_FILL};

  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(3, GL_FLOAT, 0, &vertex_coords.front());
  glPolygonMode(GL_FRONT_AND_BACK, modo[cv.modo_vis]);

  if (cv.modo_vis < 3) {
    glColor3f(color(R), color(G), color(B));
    glDrawElements(GL_TRIANGLES, 3 * indexes.size(), GL_UNSIGNED_INT, &indexes.front());
  } else if (cv.modo_vis == 3) { // Ajedrez
    for (unsigned i = 0; i < indexes.size(); i++) {
      float tone = 0.8 * (i % 2); // 0.8 if i is odd
      glColor3f(tone, tone, tone);
      glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, &indexes[i]);
    }
  } else if (cv.modo_vis == 4) {
    glShadeModel(GL_FLAT);

    if (!text_coords.empty()) {
      glTexCoordPointer(2, GL_FLOAT, 0, &text_coords.front());
      glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    }

    if (!normales_vertices.empty()) {
      glNormalPointer(GL_FLOAT, 0, &normales_vertices.front());
      glEnableClientState(GL_NORMAL_ARRAY);
    }

    glDrawElements(GL_TRIANGLES, 3 * indexes.size(), GL_UNSIGNED_INT, &indexes.front());

    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);

  } else if (cv.modo_vis == 5) {
    glShadeModel(GL_SMOOTH);

    if (!text_coords.empty()) {
      glTexCoordPointer(2, GL_FLOAT, 0, &text_coords.front());
      glEnableClientState(GL_TEXTURE_COORD_ARRAY);
    }

    if (!normales_vertices.empty()) {
      glNormalPointer(GL_FLOAT, 0, &normales_vertices.front());
      glEnableClientState(GL_NORMAL_ARRAY);
    }

    glDrawElements(GL_TRIANGLES, 3 * indexes.size(), GL_UNSIGNED_INT, &indexes.front());

    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
  }

  glDisableClientState(GL_VERTEX_ARRAY);
}

static Tupla3f normalizar(Tupla3f t) {
  return (t(X) != 0 || t(Y) != 0 || t(Z) != 0) ? t.normalized() : Tupla3f(1, 0, 0);
}
```

```cpp
void MallaInd::calcularNormales() {
  normales_vertices.resize(vertex_coords.size());
  std::fill(normales_vertices.begin(), normales_vertices.end(), Tupla3f(0, 0, 0));

  for (std::vector<Tupla3i>::iterator cara = indexes.begin(); cara != indexes.end(); ++cara) {
    Tupla3f
      p = vertex_coords[(*cara)(0)],
      q = vertex_coords[(*cara)(1)],
      r = vertex_coords[(*cara)(2)],

      // Vectores correspondientes a dos aristas
      a = q - p,
      b = r - p,

      // Vector ortogonal a la cara
      ortogonal = a.cross(b);

    normales_caras.push_back(ortogonal);

    // Sumamos ahora el normal a la cara a los vectores que
    // corresponden a los vértices
    for (int v = 0; v < 3; ++v)
      normales_vertices[(*cara)(v)] = normales_vertices[(*cara)(v)] + ortogonal;
  }

  // Normalizamos todos los vectores (para máxima eficiencia usamos transform)
  std::transform(normales_caras.begin(), normales_caras.end(),
                 normales_caras.begin(), normalizar);

  std::transform(normales_vertices.begin(), normales_vertices.end(),
                 normales_vertices.begin(), normalizar);
}

/*************************
Método visualizar utilizando
las funciones glBegin y glEnd.
************************* /
void MallaInd::visualizar(ContextoVis cv) {
  // Comprobación de parámetros
  if (cv.modo_vis > 3 || cv.modo_vis < 0) {
    cv.modo_vis = 2;
  }

  const GLenum modo[] = {GL_POINTS, GL_LINES, GL_TRIANGLES, GL_TRIANGLES};

  glBegin(modo[cv.modo_vis]);
    for (unsigned i = 0; i < indexes.size(); i++) {
      for (unsigned j = 0; j < 3; j++) {
        float tone = 0.8 * (i % 2); // 0.8 if i is odd
        glColor3f(tone, tone, tone);
        glVertex3f(vertex_coords[indexes[i][j]][0], vertex_coords[indexes[i][j]][1], vertex_coords[i
      }
    }
  glEnd();
}

/* Método de impresión para depuración */
void MallaInd::imprimir() {
  std::cerr << "Tamaño " << indexes.size() << std::endl;
```

```
  for (unsigned i = 0; i < indexes.size(); i++) {
    Tupla3i cur = indexes[i];

    std::cerr << cur << ": ";

    for (unsigned j = 0; j < 3; j++) {
      std::cerr << vertex_coords[cur[j]] << " ";
    }

    std::cerr << std::endl;
  }
  std::cerr << "Hasta aquí" << std::endl;
}
```

srcs-prac/MallaPly.cpp

```
#include "MallaPly.hpp"
#include "file_ply_stl.hpp"

MallaPly::MallaPly(const char * filename, std::string nombre) {
  nombre_obj = nombre;

  std::vector<float> vertices;
  std::vector<int> caras;
  ply::read(filename, vertices, caras);

  for (int i = 0; i < vertices.size(); i += 3) {
    vertex_coords.push_back(Tupla3f(vertices[i], vertices.at(i + 1), vertices.at(i + 2)));
  }

  for (int i = 0; i < caras.size(); i += 3) {
    indexes.push_back(Tupla3i(caras[i], caras.at(i + 1), caras.at(i + 2)));
  }
}
```

srcs-prac/MallaRev.cpp

```
#include "MallaRev.hpp"
#include "file_ply_stl.hpp"
#include <cmath>

void MallaRevol::construir(unsigned num_perfiles, bool base_inferior, bool base_superior) {
  num_perfiles = num_perfiles;
  vertex_coords.erase(vertex_coords.begin(), vertex_coords.end());
  indexes.erase(indexes.begin(), indexes.end());

  const unsigned Nc = perfil.size(), Nv = perfil.size()/3;
  const double TAU = 6.2831853;

  // Para calcular todos los vértices tomamos el perfil y lo
  // rotamos a pasos de 2pi/num_perfiles

  // Hemos de duplicar vértices para usar las coordenadas de textura
  for (unsigned giro = 0; giro <= num_perfiles; giro++)
    for (unsigned i = 0; i < Nc; i += 3) {
      vertex_coords.push_back(Tupla3f(
        perfil[i] * cos(TAU * giro / num_perfiles),
        perfil.at(i + 1),
        perfil[i] * sin(TAU * giro / num_perfiles)
      ));
```

```
    }

  // Añadimos vértices para las bases superior e inferior
  vertex_coords.push_back(Tupla3f(0, perfil.at(1),      0)); // (num_perfiles + 1) * Nv
  vertex_coords.push_back(Tupla3f(0, perfil.at(Nc - 2), 0)); // (num_perfiles + 1) * Nv + 1

  // Unimos cada vértice con su lateral y los del siguiente perfil
  // Puesto que el último perfil y el primero están sobre los mismos
  // puntos, no es necesario unirlos
  for (unsigned giro = 0; giro < num_perfiles; giro++)
    for (unsigned i = 0; i < Nv - 1; i++) {
      indexes.push_back(Tupla3i(
        giro * Nv + i,
        (giro + 1) * Nv + i,
        (giro + 1) * Nv + i + 1
      ));
      indexes.push_back(Tupla3i(
        // seleccionamos vértices en el sentido positivo
        // (la regla del tornillo nos dice que la normal será hacia afuera)
        giro * Nv + i,
        (giro + 1) * Nv + i + 1,
        giro * Nv + i + 1
      ));
    }

  // Caras correspondientes a las bases:
  if (base_inferior)
    for (unsigned giro = 0; giro < num_perfiles; giro++) {
      indexes.push_back(Tupla3i(
        giro * Nv,
        (giro + 1) * Nv,
        (num_perfiles + 1) * Nv
      ));
    }

  if (base_superior)
    for (unsigned giro = 0; giro < num_perfiles; giro++) {
      indexes.push_back(Tupla3i(
        giro * Nv + (Nv - 1),
        (giro + 1) * Nv + (Nv - 1),
        (num_perfiles + 1) * Nv + 1
      ));
    }

  calcularNormales();
}

void MallaRevol::generar_coords_textura(bool invertir) {
  unsigned v_por_perfil = perfil.size()/3,
    total_perfiles = vertex_coords.size() / v_por_perfil;

  // Obtenemos las alturas normalizadas de los vértices del perfil
  std::vector<float> alturas;
  float norma = perfil.at(perfil.size() - 2) - perfil.at(1);

  alturas.push_back(0);
  for (unsigned j = 1; j < v_por_perfil; ++j) {
    float t = (vertex_coords.at(j)(Y) - perfil.at(1))/norma;
    if (invertir) t = 1 - t;
    alturas.push_back(t);
  }
```

```
  // Calculamos las coordenadas de textura
  for (unsigned i = 0; i < total_perfiles; i++) {
    float s = i / (float)(total_perfiles - 1);
    if (invertir) s = 1 - s;

    for (unsigned j = 0; j < alturas.size(); ++j) {
      text_coords.push_back(Tupla2f(
        s,
        alturas.at(j)
      ));
    }
  }
}

MallaRevol::MallaRevol(const char * filename, unsigned num_perfiles, bool base_inferior, bool base_su
  nombre_obj = nombre;

  ply::read_vertices(filename, perfil);
  construir(num_perfiles, base_inferior, base_superior);

  if (usar_textura)
    generar_coords_textura(invertir_textura);
}
```

srcs-prac/Material.cpp

```
#include "Material.hpp"

//-------------------------------------------------
// Activación de materiales
void MaterialEstandar::activar() {
  glEnable(GL_LIGHTING);
  glEnable(GL_NORMALIZE);
  glDisable(GL_COLOR_MATERIAL);

  glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
  glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR);
  glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

  glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, color[0]);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, color[1]);
  glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, color[2]);
  glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, color[3]);
  glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, exponente);

  glColorMaterial(GL_FRONT_AND_BACK, GL_EMISSION);
  glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT);
  glColorMaterial(GL_FRONT_AND_BACK, GL_DIFFUSE);
  glColorMaterial(GL_FRONT_AND_BACK, GL_SPECULAR);
  glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);

  if (text != NULL) {
    text->activar();
  } else {
    glDisable(GL_TEXTURE_2D);
  }
}

//-------------------------------------------------
```

```
// Gestión de texturas
void Textura::activar() {
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D, id_text);

  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

  if (mgct == 0) {
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
  } else {
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);

    if (mgct == 1) {
      glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
      glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);

      glTexGenfv(GL_S, GL_OBJECT_PLANE, cs);
      glTexGenfv(GL_T, GL_OBJECT_PLANE, ct);
    } else {
      glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
      glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);

      glTexGenfv(GL_S, GL_EYE_PLANE, cs);
      glTexGenfv(GL_T, GL_EYE_PLANE, ct);
    }
  }
}

Textura::Textura(const std::string& archivoJPG, unsigned mgct) :mgct(mgct) {
  img = new jpg::Imagen(archivoJPG);

  cs[0] = ct[2] = 1;
  cs[1] = cs[2] = cs[3] = ct[0] = ct[1] = ct[3] = 0;

  glGenTextures(1, &id_text);
  glBindTexture(GL_TEXTURE_2D, id_text);

  /*
  glTexImage2D(
    GL_TEXTURE_2D,
    0,
    GL_RGB,
    img->tamX(), img->tamY(),
    0, GL_RGB, GL_UNSIGNED_BYTE,
    img->leerPixels()
  );*/

  gluBuild2DMipmaps(
    GL_TEXTURE_2D,
    GL_RGB,
    img->tamX(), img->tamY(),
    GL_RGB, GL_UNSIGNED_BYTE,
    img->leerPixels()
  );

  glBindTexture(GL_TEXTURE_2D, 0);
```

```
}

//-------------------------------------------------
// Materiales concretos

MaterialLata::MaterialLata() {
  text = new Textura("../imgs/lata-coke.jpg", 0);

  Tupla4f blanco(1, 1, 1, 1);
  color[0] = blanco * 0.8;
  color[1] = blanco * 0.1;
  color[2] = blanco * 0.3;
  color[3] = blanco * 0.7;
  exponente = 3;
}

MaterialTapasLata::MaterialTapasLata() {
  text = NULL;

  Tupla4f blanco(0.9, 0.95, 1, 1);
  color[0] = blanco * 0.65;
  color[1] = blanco * 0.05;
  color[2] = blanco * 0.2;
  color[3] = blanco * 0.7;
  exponente = 3;
}

MaterialPeonMadera::MaterialPeonMadera() {
  text = new Textura("../imgs/text-madera.jpg", 1);

  Tupla4f blanco(1, 1, 1, 1);
  color[0] = blanco * 0.8;
  color[1] = blanco * 0.1;
  color[2] = blanco * 0.3;
  color[3] = blanco * 0.7;
  exponente = 3;
}

MaterialPeonBlanco::MaterialPeonBlanco() {
  text = NULL;

  Tupla4f blanco(1, 1, 1, 1);
  color[0] = blanco * 0.8;
  color[1] = blanco * 0.05;
  color[2] = blanco * 0.2;
  color[3] = blanco * 0.0;
  exponente = 3;
}

MaterialPeonNegro::MaterialPeonNegro() {
  text = NULL;

  Tupla4f blanco(1, 1, 1, 1);
  color[0] = blanco * 0.005;
  color[1] = blanco * 0.0;
  color[2] = blanco * 0.01;
  color[3] = blanco * 0.6;
  exponente = 3;
}

MaterialBaseR2::MaterialBaseR2() {
```

```
  text = NULL;

  Tupla4f blanco(1, 1, 1, 1);
  color[0] = blanco * 0.7;
  color[1] = blanco * 0.05;
  color[2] = blanco * 0.2;
  color[3] = blanco * 0.0;
  exponente = 3;
}

MaterialAzulR2::MaterialAzulR2() {
  text = NULL;

  Tupla4f azul(0.0, .2, 0.4, 1);
  color[0] = azul * 0.7;
  color[1] = azul * 0.05;
  color[2] = azul * 0.2;
  color[3] = azul * 0.0;
  exponente = 3;
}

MaterialCuerpoR2::MaterialCuerpoR2() {
  text = new Textura("r2d2.jpg", 0);

  Tupla4f blanco(1, 1, 1, 1);
  color[0] = blanco * 0.8;
  color[1] = blanco * 0.1;
  color[2] = blanco * 0.3;
  color[3] = blanco * 0.7;
  exponente = 3;
}
```

srcs-prac/NodoGrafoEscena.cpp

```
#include "NodoGrafoEscena.hpp"
#include "aux.hpp"

void NodoGrafoEscena::visualizar(ContextoVis cv) {
  // Operamos sobre la modelview
  glMatrixMode(GL_MODELVIEW);
  // Guarda modelview actual
  glPushMatrix();

  glColor3f(color(R), color(G), color(B));

  // Guarda material inicial
  Material * materialActivoInicial = cv.materialActivo;

  // Recorrer las entradas del nodo
  for (unsigned i = 0; i < entradas.size(); i++) {
    if (entradas[i].tipoE == 0) {
      // Visualizar los sub-objetos
      entradas[i].objeto->visualizar(cv);
    } else if (entradas[i].tipoE == 1) {
      // Componer las transformaciones
      glMultMatrixf(*(entradas[i].matriz));
    } else if (entradas[i].tipoE == 2) {
      if (entradas[i].material != cv.materialActivo) {
        cv.materialActivo = entradas[i].material;
        cv.materialActivo->activar();
```

```
      }
    }
  }

  // Recupera material inicial
  if (materialActivoInicial != cv.materialActivo) {
    cv.materialActivo = materialActivoInicial;

    if (cv.materialActivo != NULL)
      cv.materialActivo->activar();
  }

  glMatrixMode(GL_MODELVIEW);
  // Recuperar la modelview guardada
  glPopMatrix();
}

void NodoGrafoEscena::agregar(EntradaNGE * entrada) {
  entradas.push_back(*entrada);
}

void NodoGrafoEscena::agregar(Objeto3D * pObjeto) {
  entradas.push_back(EntradaNGE(pObjeto));
  pObjeto->setColor(this->color);
}

void NodoGrafoEscena::agregar(const Matriz4f& pMatriz) {
  entradas.push_back(EntradaNGE(pMatriz));
}

void NodoGrafoEscena::agregar(Material * p_material) {
  entradas.push_back(EntradaNGE(p_material));
}
```

srcs-prac/Objeto3D.cpp

```
#include "Objeto3D.hpp"

std::string Objeto3D::nombre() {
  return nombre_obj;
}
```

srcs-prac/practica1.cpp

```
// *********************************************************************
// **
// ** Informática Gráfica, curso 2015-16
// ** Práctica 1  (implementación)
// **
// *********************************************************************

#include "aux.hpp"
#include "tuplasg.hpp"   // Tupla3f
#include "practica1.hpp"
#include <cmath>

unsigned objeto_activo = 0; // objeto activo: cubo (0), tetraedro (1), otros....

// -----------------------------------------------------------------
// declaraciones de estructuras de datos....
```

```cpp
std::vector<MallaInd> figuras;
unsigned material, precision;

// Configuración del torp
const double
  DEFAULT_MAJOR_RADIUS = 1,
  DEFAULT_MINOR_RADIUS = 0.5;

// ---------------------------------------------------------------------
// Función para implementar en la práctica 1 para inicialización.
// Se llama una vez al inicio, cuando ya se ha creado la ventana e
// incializado OpenGL.

void P1_Inicializar(int argc, char *argv[]) {
  material = 1;
  precision = 1;
  figuras.push_back(Cubo());
  figuras.push_back(Tetraedro());
  figuras.push_back(Cono(precision));
  figuras.push_back(Cilindro(precision));
  figuras.push_back(Toro(DEFAULT_MAJOR_RADIUS, DEFAULT_MINOR_RADIUS, precision));
  figuras.push_back(Moebius(precision));
  figuras.push_back(Klein(precision));
}

// ---------------------------------------------------------------------
// Función invocada al pulsar una tecla con la práctica 1 activa:
// (si la tecla no se procesa en el 'main').
//
//  - devuelve 'true' si la tecla se usa en esta práctica para cambiar
//    entre el cubo, el tetraedro u otros objetos (cambia el valor de
//    'objeto_activo').
//  - devuelve 'false' si la tecla no se usa en esta práctica (no ha
//    cambiado nada)

bool P1_FGE_PulsarTeclaNormal(unsigned char tecla) {
  tecla = tolower(tecla);

  if (tecla >= '0' && tecla <= '9') {
    precision = tecla - '0' + 10 * (tecla == '0');
    figuras[2] = Cono(precision);
    figuras[3] = Cilindro(precision);
    figuras[4] = Toro(DEFAULT_MAJOR_RADIUS, DEFAULT_MINOR_RADIUS, precision);
    figuras[5] = Moebius(precision);
    figuras[6] = Klein(precision);
  }
  else if (tecla == ' ' || tecla == '.' || tecla == 'o') {
    ++objeto_activo %= figuras.size();
    std::cerr << "Objeto actual: " << figuras[objeto_activo].nombre() << std::endl;
  }
  else if (tecla == ',') {
    objeto_activo = (objeto_activo + figuras.size() - 1)% figuras.size();
  }
  else {
    return false;
  }

  return true;
}
```

```
// -----------------------------------------------------------------------
// Función a implementar en la práctica 1  para dibujar los objetos
// modo: 0 - puntos, 1 - alambre, 2 - sólido, 3 - sólido ajedrez , >=4 otros....

void P1_DibujarObjetos(unsigned modo) {
  figuras[objeto_activo].visualizar(modo);
}

Cubo::Cubo(std::string nombre) {
  nombre_obj = nombre;

  vertex_coords.push_back(Tupla3f(0,0,1)); // 0
  vertex_coords.push_back(Tupla3f(1,0,0)); // 1
  vertex_coords.push_back(Tupla3f(1,1,0)); // 2
  vertex_coords.push_back(Tupla3f(0,1,0)); // 3
  vertex_coords.push_back(Tupla3f(0,0,0)); // 4
  vertex_coords.push_back(Tupla3f(1,0,1)); // 5
  vertex_coords.push_back(Tupla3f(1,1,1)); // 6
  vertex_coords.push_back(Tupla3f(0,1,1)); // 7

  // Orden: inferior, izquierda, posterior, superior, derecha y frontal

  indexes.push_back(Tupla3i(0,5,4));
  indexes.push_back(Tupla3i(4,5,1));

  indexes.push_back(Tupla3i(0,4,3));
  indexes.push_back(Tupla3i(3,7,0));

  indexes.push_back(Tupla3i(4,1,2));
  indexes.push_back(Tupla3i(3,4,2));

  indexes.push_back(Tupla3i(2,3,7));
  indexes.push_back(Tupla3i(7,6,2));

  indexes.push_back(Tupla3i(6,5,1));
  indexes.push_back(Tupla3i(1,2,6));

  indexes.push_back(Tupla3i(0,6,7));
  indexes.push_back(Tupla3i(0,5,6));
}

Tetraedro::Tetraedro(std::string nombre) {
  nombre_obj = nombre;

  // Base: (-sqrt(2)sin(15), 0, -sqrt(2)sin(15)), (1, 0, 0), (0, 0, 1)
  const float offset = -0.366025404;
  vertex_coords.push_back(Tupla3f(offset,0,offset));
  vertex_coords.push_back(Tupla3f(0,0,1));
  vertex_coords.push_back(Tupla3f(1,0,0));

  // Punto superior, offset 1 - 2/3 sqrt(2)sin(60)
  const float off2 = 0.183503419;
  const float height = 0.816496581;
  vertex_coords.push_back(Tupla3f(off2,height,off2));

  indexes.push_back(Tupla3i(0,1,2));
  indexes.push_back(Tupla3i(0,2,3));
  indexes.push_back(Tupla3i(0,1,3));
  indexes.push_back(Tupla3i(1,2,3));
}
```

```
Cono::Cono(unsigned prec, std::string nombre) {
  nombre_obj = nombre;

  const unsigned N = 10 * prec;
  const double PI = 3.1415926;

  const double height = 2;

  // Calculamos vértices en la circunferencia base
  for (unsigned i = 0; i < N; i++) {
    vertex_coords.push_back(Tupla3f(cos(i * 2 * PI / N), 0, sin(i * 2 * PI / N)));
  }
  vertex_coords.push_back(Tupla3f(0,0,0)); // N
  vertex_coords.push_back(Tupla3f(0,height,0)); // N + 1

  // Añadimos triángulos en el círculo base
  for (unsigned i = 0; i < N; i++) {
    indexes.push_back(Tupla3i(N, i, (i + 1)%N));
  }

  // Añadimos triángulos para la cara lateral
  for (unsigned i = 0; i < N; i++) {
    indexes.push_back(Tupla3i(N + 1, i, (i + 1)%N));
  }
}

Cilindro::Cilindro(unsigned prec, std::string nombre) {
  nombre_obj = nombre;

  const unsigned N = 10 * prec;
  const double PI = 3.1415926;

  const double height = 2;

  // Calculamos vértices en la circunferencia base inferior
  for (unsigned i = 0; i < N; i++) {
    vertex_coords.push_back(Tupla3f(cos(i * 2 * PI / N), 0, sin(i * 2 * PI / N)));
  }
  // Calculamos vértices en la circunferencia base superior
  for (unsigned i = 0; i < N; i++) {
    vertex_coords.push_back(Tupla3f(cos(i * 2 * PI / N), height, sin(i * 2 * PI / N)));
  }
  vertex_coords.push_back(Tupla3f(0,0,0)); // 2N
  vertex_coords.push_back(Tupla3f(0,height,0)); // 2N + 1

  // Añadimos triángulos en el círculo base inferior
  for (unsigned i = 0; i < N; i++) {
    indexes.push_back(Tupla3i(2*N, i, (i + 1)%N));
  }

  // Añadimos triángulos en el círculo base superior
  for (unsigned i = 0; i < N; i++) {
    indexes.push_back(Tupla3i(2*N + 1, N + i, N + (i + 1)%N));
  }

  // Añadimos triángulos para la cara lateral
  for (unsigned i = 0; i < N; i++) {
    // Triángulo con base abajo
    indexes.push_back(Tupla3i(i, (i + 1)%N, N + i));
    // Triángulo con base arriba
```

```
    indexes.push_back(Tupla3i(N + i, (i + 1)%N, N + (i + 1)%N));
  }
}

Tupla3f Toro::vertice(double theta, double phi) {
  return Tupla3f(
    (rad_ext + rad_int * cos(theta)) * cos(phi),
    (rad_ext + rad_int * cos(theta)) * sin(phi),
    rad_int * sin(theta)
  );
}

Toro::Toro(double R, double r, unsigned prec, std::string nombre)
  :rad_ext(R), rad_int(r) {
  nombre_obj = nombre;

  const unsigned N = 10 * prec;
  const double TAU = 6.2831853;

  // Añadimos vértices con la precisión dada
  for (unsigned i = 0; i < N; ++i) {
    double theta = (double)(i) / N * TAU;

    // Vértices i*N a i*N + (N - 1)
    for (unsigned j = 0; j < N; ++j) {
      // Vértice i*N + j
      double phi = (double)(j) / N * TAU;

      vertex_coords.push_back(vertice(theta, phi));
    }
  }

  // Añadimos caras
  // Recorriendo círculo exterior
  for (unsigned i = 0; i < N; ++i) {
    // Recorriendo círculo interior
    for (unsigned j = 0; j < N; ++j) {
      // Unir cada punto del círculo con el que está en la misma posición en
      // el siguiente círculo y con los adyacentes
      indexes.push_back(Tupla3i(i*N + j, ((i + 1)%N)*N + j, i*N + (j + 1)%N));
      indexes.push_back(Tupla3i(((i + 1)%N)*N + j, ((i + 1)%N)*N + (j + 1)%N, i * N + (j + 1)%N));
    }
  }

}

Tupla3f Moebius::vertice(double u, double v) {
  return Tupla3f(
    (1 + (v - 1)/2 * cos(u/2)) * cos(u),
    (1 + (v - 1)/2 * cos(u/2)) * sin(u),
    (v - 1)/2 * sin(u/2)
  );
}

Moebius::Moebius(unsigned prec, std::string nombre) {
  nombre_obj = nombre;

  const unsigned N = 10 * prec;
  const double TAU = 6.2831853;

  const double LIMIT_U = TAU;
```

```
  const double LIMIT_V = 2;

  // Añadimos vértices con la precisión dada
  for (unsigned i = 0; i < N; ++i) {
    double u = (double)(i) / N * LIMIT_U;

    // Vértices i*N a i*N + (N - 1)
    for (unsigned j = 0; j < N; ++j) {
      // Vértice i*N + j
      double v = (double)(j) / N * LIMIT_V;

      vertex_coords.push_back(vertice(u, v));
    }
  }

  // Añadimos caras
  for (unsigned i = 0; i < N - 1; ++i) {
    for (unsigned j = 0; j < N - 1; ++j) {
      // Unir cada punto del círculo con el que está en la misma posición en
      // el siguiente círculo y con los adyacentes
      indexes.push_back(Tupla3i(i*N + j, ((i + 1)%N)*N + j, i*N + (j + 1)%N));
      indexes.push_back(Tupla3i(((i + 1)%N)*N + j, ((i + 1)%N)*N + (j + 1)%N, i * N + (j + 1)%N));
    }
  }

  // La última fila de vértices hay que unirla con los opuestos de la primera!
  for (unsigned j = 0; j < N - 1; ++j) {
    indexes.push_back(Tupla3i((N - 1)*N + j, (N - j - 1)%N, (N - 1)*N + (j + 1)%N));
    indexes.push_back(Tupla3i((N - j - 2)%N, (N - j - 1)%N, (N - 1) * N + (j + 1)%N));
  }

}

Tupla3f Klein::vertice(double u, double v) {
  double cosu = cos(u), cosv = cos(v),
    sinu = sin(u), sinv = sin(v);

  // Ecuaciones paramétricas de la proyección tridimensional
  // de la botella de Klein obtenidas de Wikipedia:
  // https://en.wikipedia.org/wiki/Klein_bottle#Bottle_shape
  return Tupla3f(
    // x
    -(2.0/15) * cosu * (3 * cosv - 30 * sinu + 90 * pow(cosu, 4) * sinu
    - 60 * pow(cosu, 6) * sinu + 5 * cosu * cosv * sinu)

    ,// y
    -(1.0/15) * sinu * (3 * cosv - 3 * cosu * cosu * cosv - 48 * pow(cosu, 4) * cosv
    + 48 * pow(cosu, 6) * cosv - 60 * sinu + 5 * cosu * cosv * sinu
    - 5 * pow(cosu, 3) * cosv * sinu - 80 * pow(cosu, 5) * cosv * sinu
    + 80 * pow(cosu, 7) * cosv * sinu)

    ,// z
    (2.0/15) * (3 + 5 * cosu * sinu) * sinv
  );
}

Klein::Klein(unsigned prec, std::string nombre) {
  nombre_obj = nombre;

  const unsigned N = 10 * prec;
  const double TAU = 6.2831853;
```

```
  const double LIMIT_U = TAU/2;
  const double LIMIT_V = TAU;

  // Añadimos vértices con la precisión dada
  for (unsigned i = 0; i < N; ++i) {
    double u = (double)(i) / N * LIMIT_U;

    // Vértices i*N a i*N + (N - 1)
    for (unsigned j = 0; j < N; ++j) {
      // Vértice i*N + j
      double v = (double)(j) / N * LIMIT_V;

      vertex_coords.push_back(vertice(u, v));
    }
  }

  // Añadimos caras
  for (unsigned i = 0; i < N - 1; ++i) {
    for (unsigned j = 0; j < N; ++j) {
      // Unir cada punto del círculo con el que está en la misma posición en
      // el siguiente círculo y con los adyacentes
      indexes.push_back(Tupla3i(i*N + j, ((i + 1)%N)*N + j, i*N + (j + 1)%N));
      indexes.push_back(Tupla3i(((i + 1)%N)*N + j, ((i + 1)%N)*N + (j + 1)%N, i * N + (j + 1)%N));
    }
  }

  // La última fila de vértices hay que unirla con los opuestos de la primera
  for (unsigned j = 0; j < N; ++j) {
    indexes.push_back(Tupla3i((N - 1)*N + j, (N - (j + N/2)%N)%N, (N - 1)*N + (j + 1)%N));
    indexes.push_back(Tupla3i((N - (j + N/2)%N - 1)%N, (N - (j + N/2)%N)%N, (N - 1) * N + (j + 1)%N)
  }

}
```

srcs-prac/practica2.cpp

```
// *********************************************************************
// **
// ** Informática Gráfica, curso 2015-16
// ** Práctica 2  (implementación)
// **
// *********************************************************************

#include "aux.hpp"
#include "tuplasg.hpp"   // Tupla3f
#include "practica2.hpp"
#include "MallaInd.hpp"
#include "MallaPly.hpp"
#include "MallaRev.hpp"
#include "MallaBarrido.hpp"
#include <cmath>
#include <string>
#include <sstream>

static unsigned p2_objeto_activo = 0;
static const unsigned NUM_OBJETOS = 3;
static MallaInd * p2_figuras[NUM_OBJETOS] = {NULL};
```

```cpp
// -----------------------------------------------------------------------
// Función para implementar en la práctica 2 para inicialización.
// Se llama una vez al inicio, cuando ya se ha creado la ventana e
// incializado OpenGL.

void P2_Inicializar(int argc, char *argv[]) {
  std::string ply_file, rev_file, bar_file;
  unsigned perfiles = 10;
  Tupla3f vec_dir(1, 1, 1);

  if (argc < 2) {
    ply_file = "../plys/beethoven.ply";
  } else {
    ply_file = argv[1];
  }
  if (argc < 3) {
    rev_file = "../plys/peon.ply";
  } else {
    rev_file = argv[2];
  }
  if (argc < 4) {
    bar_file = "../plys/peon.ply";
  } else {
    bar_file = argv[3];
  }
  if (argc >= 5) {
    std::stringstream conversor(argv[4]);
    conversor >> perfiles;
  }
  if (argc >= 8) {
    for (int i = 0; i < 3; i++) {
      std::stringstream conversor(argv[5 + i]);
      conversor >> vec_dir[i];
    }

    std::cout << vec_dir;
  }


  p2_figuras[0] = new MallaPly(ply_file.c_str());
  p2_figuras[1] = new MallaRevol(rev_file.c_str(), perfiles);
  p2_figuras[2] = new MallaBarrido(bar_file.c_str(), perfiles, vec_dir);
}

// -----------------------------------------------------------------------
// Función invocada al pulsar una tecla con la práctica 2 activa:
// (si la tecla no se procesa en el 'main').
//
//  - devuelve 'true' si la tecla se usa en esta práctica para cambiar
//    entre el cubo, el tetraedro u otros objetos (cambia el valor de
//    'objeto_activo').
//  - devuelve 'false' si la tecla no se usa en esta práctica (no ha
//    cambiado nada)

bool P2_FGE_PulsarTeclaNormal(unsigned char tecla) {
  tecla = tolower(tecla);

  if (tecla >= '0' && tecla <= '9') {
    unsigned num = (tecla - '0' + 10 * (tecla == '0'));
    static_cast<MallaRevol*>(p2_figuras[1])->construir(num * 10);
    static_cast<MallaBarrido*>(p2_figuras[2])->construir(num);
```

```
  } else if (tecla == ' ' || tecla == '.' || tecla == 'o') {
    ++p2_objeto_activo %= NUM_OBJETOS;
    std::cerr << "Objeto actual: " << p2_figuras[p2_objeto_activo]->nombre() << std::endl;
  } else if (tecla == ',') {
    p2_objeto_activo = (p2_objeto_activo + NUM_OBJETOS - 1) % NUM_OBJETOS;
  } else {
    return false;
  }

  return true;
}


// -------------------------------------------------------------------------
// Función a implementar en la práctica 2 para dibujar los objetos
// modo: 0 - puntos, 1 - alambre, 2 - sólido, 3 - sólido ajedrez , >=4 otros....

void P2_DibujarObjetos(unsigned modo) {
  p2_figuras[p2_objeto_activo]->visualizar(modo);
}
```

srcs-prac/practica3.cpp

```
// *********************************************************************
// **
// ** Informática Gráfica, curso 2015-16
// ** Práctica 3  (implementación)
// **
// *********************************************************************

#include "aux.hpp"
#include "tuplasg.hpp"   // Tupla3f
#include "practica3.hpp"
#include "NodoGrafoEscena.hpp"
#include "R2.hpp"

typedef void (R2::*R2Method)(float);

static unsigned p3_objeto_activo = 0;
static const unsigned NUM_OBJETOS = 3;
static R2 * p3_figura;

static const unsigned GRADOS_LIBERTAD = 4;
static unsigned p3_grado_libertad_activo = 0;
static float p3_valores[GRADOS_LIBERTAD] = {R2::DEFAULT_VALUE,R2::DEFAULT_VALUE,R2::DEFAULT_VALUE,R2
static R2Method funciones_cambios[GRADOS_LIBERTAD] = {
  &R2::girar_cabeza,
  &R2::extender_proj,
  &R2::girar_brazos,
  &R2::extender_brazos
};
static const std::string p3_grados_nombres[GRADOS_LIBERTAD] = {
  "Giro cabeza",
  "Desplazamiento proyector cabeza",
  "Giro brazos",
  "Desplazamiento brazos"
};

void P3_Inicializar(int argc, char *argv[]) {
  p3_figura = new R2();
```

```
}


bool P3_FGE_PulsarTeclaNormal(unsigned char tecla) {
  // Si pulsamos mayúsculas usamos signo negativo
  int signo = 1 - 2 * (tecla != tolower(tecla));
  tecla = tolower(tecla);

  if (tecla == 'g') {
    p3_grado_libertad_activo = (p3_grado_libertad_activo + 1) % GRADOS_LIBERTAD;
    std::cout << "Cambiado grado de libertad a '" << p3_grados_nombres[p3_grado_libertad_activo] << "
  } else if (tecla == '<' || tecla == '>' || tecla == 'z' || tecla == 'x' || tecla == 'c' || tecla ==
    unsigned grado = p3_grado_libertad_activo;

    if (tecla == '<' || tecla == '>')
      signo = 1 - 2 * (tecla == '<');
    else
      grado = (tecla == 'x') + 2 * (tecla == 'c') + 3 * (tecla == 'v');

    float nuevo = p3_valores[grado] + signo/100.0;

    if (nuevo >= 0 && nuevo <= 1) {
      (p3_figura->*funciones_cambios[grado])(nuevo);
      std::cout << "Ajustado grado de libertad '" << p3_grados_nombres[grado] << "' con valor " << nu
      p3_valores[grado] += signo/100.0;
    }

  } else {
    return false;
  }

  return true;
}


void P3_DibujarObjetos(unsigned modo) {
  p3_figura->visualizar(modo);
}
```

srcs-prac/practica4.cpp

```
// *********************************************************************
// **
// ** Informática Gráfica, curso 2015-16
// ** Práctica 4  (implementación)
// **
// *********************************************************************

#include "aux.hpp"
#include "tuplasg.hpp"   // Tupla3f
#include "practica4.hpp"
#include "matrices-tr.hpp"
#include "MallaRev.hpp"
#include "FuenteLuz.hpp"

static NodoGrafoEscena* p4_escena = NULL;
static ColeccionFL* p4_luces = NULL;
static unsigned current_longi = 1;

void P4_Inicializar(int argc, char *argv[]) {
```

```
  p4_escena = new Escena();
  p4_luces = new ColeccionFuentesP4();
  p4_luces->activar();
}

bool P4_FGE_PulsarTeclaNormal(unsigned char tecla) {
  if (toupper(tecla) == 'G') {
    current_longi = !current_longi;
    return true;
  } else if (tecla == '<') {
    static_cast<FuenteDireccional*>(p4_luces->fuentes.at(0))->variarAngulo(current_longi, -10);
    return true;
  } else if (tecla == '>') {
    static_cast<FuenteDireccional*>(p4_luces->fuentes.at(0))->variarAngulo(current_longi, 10);
    return true;
  }

  return false;
}

void P4_DibujarObjetos(ContextoVis modo) {
  glEnable(GL_LIGHTING);

  p4_escena->visualizar(modo);

  glDisable(GL_LIGHTING);
}

Escena::Escena() {
  agregar(new Lata());

  agregar(new MaterialPeonMadera());
  agregar(MAT_Traslacion(-3, 1.4, 3));
  agregar(new Peon(true));

  agregar(new MaterialPeonBlanco());
  agregar(MAT_Traslacion(3, 0, 0));
  agregar(new Peon());

  agregar(new MaterialPeonNegro());
  agregar(MAT_Traslacion(3, 0, 0));
  agregar(new Peon());
}

Lata::Lata() {
  agregar(MAT_Escalado(4, 4, 4));

  agregar(new MaterialTapasLata());
  agregar(new BaseInfLata());
  agregar(new BaseSupLata());

  agregar(MAT_Rotacion(90, 0, 1, 0));
  agregar(new MaterialLata());
  agregar(new LateralLata());
}

LateralLata::LateralLata() {
  // La malla de revolución para el cuerpo de la lata la
  // generamos sin bases y con una textura, invirtiendo
  // el cálculo de las coordenadas de textura:
  agregar(new MallaRevol("lata-pcue.ply", 40, false, false, true, true));
```

```
}

BaseInfLata::BaseInfLata() {
  agregar(new MallaRevol("lata-pinf.ply", 40, false));
}

BaseSupLata::BaseSupLata() {
  agregar(new MallaRevol("lata-psup.ply", 40, false));
}

Peon::Peon(bool usar_textura) {
  agregar(new MallaRevol("../plys/peon.ply", 40, usar_textura));
}
```

srcs-prac/R2.cpp

```
#include "R2.hpp"
#include "matrices-tr.hpp"
#include "practica1.hpp"
#include "MallaPly.hpp"
#include "MallaRev.hpp"

CabezaR2::CabezaR2(float giro, float offset) {
  color = Tupla3f(0.66, 0.66, 0.66);
  agregar(MAT_Rotacion(0, 0, 1, 0)); // Vacía (usamos el método)
  agregar(MAT_Traslacion(0, 2.48, 0));

  agregar(new Camara());

  agregar(MAT_Escalado(0.00777, 0.00777, 0.00777));
  agregar(new MallaPly("sphere.ply"));

  agregar(pr = new ProjR2(offset));

  girar(giro);
  extender(offset);
}

void CabezaR2::girar(float giro) {
  entradas.at(0) = EntradaNGE(MAT_Rotacion(-30 + 60 * giro, 0, 1, 0));
}

Camara::Camara() {
  agregar(new MaterialAzulR2());
  agregar(MAT_Rotacion(-30, 1, 0, 0));
  agregar(MAT_Escalado(0.35, 0.35, 0.35));
  agregar(MAT_Traslacion(-0.5, -0.5, 1.8));
  Cubo * cuadrado = new Cubo();
  agregar(cuadrado);

  agregar(new MaterialPeonNegro());
  agregar(MAT_Traslacion(0.5, 0.5, 0.75));
  agregar(MAT_Escalado(0.00368, 0.00368, 0.00368));
  MallaPly * camara = new MallaPly("sphere.ply");
  agregar(camara);
}

ProjR2::ProjR2(float offset) {
  color = Tupla3f(0.66, 0.66, 0.66);
  agregar(MAT_Rotacion(15, 0, 1, 0));
```

```
  agregar(MAT_Traslacion(0, 20, 120));
  agregar(MAT_Escalado(0.1, 0.1, 0.1));
  MallaPly * base_camara = new MallaPly("sphere.ply");
  agregar(base_camara);

  agregar(MAT_Rotacion(85, 1, 0, 0));
  agregar(MAT_Escalado(1, 1, 1)); // Vacío (usamos el método)
  agregar(new Cilindro(5));
  agregar(new MaterialPeonNegro());
  agregar(MAT_Escalado(0.9, 1.01, 0.9));
  Cilindro * interior_camara = new Cilindro(5);
  agregar(interior_camara);
}

void ProjR2::extender(float offset) {
  entradas.at(5) = EntradaNGE(MAT_Escalado(80, 60 * offset + 80, 80));
}

CuerpoR2::CuerpoR2() {
  color = Tupla3f(0.86, 0.86, 0.9);
  agregar(MAT_Escalado(1, 1.2, 1));
  agregar(MAT_Rotacion(90, 0, 1, 0));
  agregar(new MaterialCuerpoR2());
  agregar(new MallaRevol("perfil_cilindro.ply", 50, true, true, true));
}

BrazoSuperior::BrazoSuperior() {
  color = Tupla3f(0.9, 0.9, 0.95);
  agregar(MAT_Rotacion(90, 0, 1, 0));
  agregar(MAT_Rotacion(90, 1, 0, 0));
  agregar(MAT_Escalado(0.4, 0.14, 0.4));
  agregar(new Cilindro(5));

  agregar(new MaterialPeonNegro());
  agregar(MAT_Traslacion(0, 1, 0));
  agregar(MAT_Escalado(0.7, 1, 0.7));
  Cilindro * junta = new Cilindro(5);
  agregar(junta);

  agregar(MAT_Traslacion(0, -1.01, 0));
  agregar(MAT_Escalado(0.5, 2, 0.5));
  Cilindro * agujero = new Cilindro(5);
  agregar(agujero);
}

LadoBrazo::LadoBrazo() {
  color = Tupla3f(0.65, 0.67, 0.7);
  agregar(MAT_Traslacion(0.05, -1.5, -0.18));
  agregar(MAT_Escalado(0.2, 1.5, 0.35));
  agregar(new Cubo());

  agregar(new MaterialAzulR2());
  agregar(MAT_Traslacion(-0.3, 0, 0.2));
  agregar(MAT_Escalado(1.2, 1, 0.6));
  Cubo * lateral = new Cubo();
  agregar(lateral);

  agregar(MAT_Traslacion(-0.15, -0.25, -0.5));
  agregar(MAT_Escalado(1.5, 0.25, 2));
  agregar(new Cubo());
```

```
}

BrazoR2::BrazoR2(float giro, float offset) {
  // Coloca el brazo a un lado y traslada hacia arriba
  agregar(MAT_Traslacion(-1.35, 2, 0));
  agregar(MAT_Rotacion(0, 1, 0, 0)); // Vacía (giramos con el método)

  agregar(new BrazoSuperior());

  agregar(MAT_Traslacion(0, 0, 0)); // Vacía (usamos el método)
  agregar(new LadoBrazo());

  // Coloca la pata
  agregar(MAT_Rotacion(12, 1, 0, 0));
  agregar(MAT_Traslacion(0.1, -2.3, 0.4));
  agregar(new PataR2());

  girar(giro);
  extender(offset);
}

void BrazoR2::girar(float giro) {
  // Gira el brazo según el parámetro dado
  entradas.at(1) = EntradaNGE(MAT_Rotacion(60 * giro, 1, 0, 0));
}

void BrazoR2::extender(float offset) {
  // Permite extender el brazo
  entradas.at(3) = EntradaNGE(MAT_Traslacion(0, (offset - 1)/3.0, 0));
}

PataR2::PataR2() {
  color = Tupla3f(0.8, 0.8, 0.85);
  agregar(MAT_Escalado(0.3, 0.5, 0.5));
  agregar(new MallaPly("truncated_pyramid.ply"));
}

R2::R2() {
  agregar(new MaterialBaseR2());
  // Coloca la pata inferior
  agregar(MAT_Traslacion(0, -0.35, 0.5));
  agregar(new PataR2());
  agregar(MAT_Traslacion(0, 0.35, -0.5));

  // Gira todo el robot para adoptar la posición típica de R2
  agregar(MAT_Rotacion(-12, 1, 0, 0));

  // Coloca la cabeza y el cuerpo
  agregar(cabeza = new CabezaR2(R2::DEFAULT_VALUE, R2::DEFAULT_VALUE));
  agregar(new CuerpoR2());

  // Coloca los dos brazos: uno al lado derecho de R2 y realizando una simetría
  // (componiendo una rotación de pi y un escalado inverso) para el izquierdo
  agregar(izquierdo = new BrazoR2(R2::DEFAULT_VALUE, R2::DEFAULT_VALUE));
  agregar(MAT_Rotacion(180, 0, 1, 0));
  agregar(MAT_Escalado(1, 1, -1));
  agregar(derecho = new BrazoR2(R2::DEFAULT_VALUE, R2::DEFAULT_VALUE));

}

const float R2::DEFAULT_VALUE = 0.5;
```