Realidad Virtual. Práctica 3.

Índice

1. Objetivos

2. Desarrollo del modelo

3. Programa gráfico

4. Figuras geométricas

5. Descripción de un cubo

6. Descripción de una pirámide

7. Descripción de un cono

8. Descripción de un cilindro

9. Descripción de una esfera

10. Descripción de un disco

11. Descripción de un toro

12. Descripción de un icosaedro

1. Objetivos

- Desarrollar un modelo 3D formado por figuras geométricas básicas.
- Se desarrollará un conjunto de clases que desarrollan estas figuras geométricas.
- El desarrollo de las clases requiere calcular la posición de los vértices de cada figura de manera que quede centrada en la escena.
- El programa gráfico añade una matriz de transformación de los vértices para poder incorporar rotaciones y traslaciones en los cuerpos.
- Código de la práctica: https://www.uhu.es/francisco.moreno/gii_rv/practicas/practica03/practica03.rar

2. Desarrollo del modelo

- El punto de partida de esta práctica es el código generado en la práctica 2, que desarrolla un proyecto en Visual Studio 2019, en el que se construye una ventana sobre MS-Windows y se modifica su configuración para poder generar el contenido de la ventana por medio de las funciones de OpenGL.
- Los ficheros main.cpp, CGApplication.h y CGApplication.cpp contienen el código encargado de configurar la ventana de MS-Windows y desarrollar la interfaz entre los eventos del sistema operativo y la parte gráfica de la aplicación (respuesta a eventos del teclado, a eventos del ratón, a cambios de tamaño, etc.). Estos ficheros no serán modificados.
- Los ficheros CGModel.h y CGModel.cpp contienen el código encargado de desarrollar el modelo 3D. En la práctica anterior era un triángulo que se desplazaba por la ventana. En esta práctica, modificaremos estos ficheros para generar algunas figuras geométricas y mostrarlas en la pantalla principal de la aplicación.

2. Desarrollo del modelo

- Contenido del fichero CGModel.h
- La interfaz pública de esta clase no se va a modificar, ya que se refiere a las funciones que se llaman desde CGApplication.cpp
- Respecto a la parte privada, modificación, incluyendo referencias a 8 objetos de la clase CGFigure, así cómo los campos figure, xAngle e yAngle.

- Contenido del fichero CGFigure.h
- Define un objeto que representa una figura geométrica.

2. Desarrollo del modelo

- En esta practica se van a definir 8 figuras geométricas diferentes:
  - Pirámide
  - Cubo,
  - Icosaedro
  - Cono
  - Cilindro
  - Disco
  - Esfera
  - Toro
- Campo figure indica la figura que se representa en cada momento.
- Para cambiar de figura se pulsa ‘F’, habrá que modificar el método key_pressed()

2. Desarrollo del modelo

- Los campos xAngle y yAngle contendrán la orientación de la figura. Para orientar la figura realizaremos en primer lugar un giro de ángulo xAngle sobre el eje X y, a continuación, un giro de tamaño yAngle sobre el eje Y.
- Se van a asociar las teclas de flechas para modificar estos ángulos.
- A partir de los valores xAngle e yAngle se calculará la matriz de transformación que se le enviará al VertexShader.
- El campo projection permite almacenar una matriz de proyección que se utiliza para transformar el espacio de representación de los objetos en el clipping volumen.
- El campo program contiene una referencia al programa gráfico a ejecutar.

2. Desarrollo del modelo

- Contenido CGModel.h

```
#pragma once
#include <GL/glew.h>
#include "CGShaderProgram.h"
#include "CGFigure.h"

class CGModel
{
public:
  void initialize(int w, int h);
  void finalize();
  void render();
  void update();
  void key_pressed(int key);
  void mouse_button(int button, int action);
  void mouse_move(double xpos, double
ypos);
  void resize(int w, int h);

private:
  CGShaderProgram* program;
  CGFigure* fig0;
  CGFigure* fig1;
  CGFigure* fig2;
  CGFigure* fig3;
  CGFigure* fig4;
  CGFigure* fig5;
  CGFigure* fig6;
  CGFigure* fig7;

  GLfloat xAngle;
  GLfloat yAngle;
  int figure;

  glm::mat4 projection;
};
```

2. Desarrollo del modelo

- Con respecto a CGModel.cpp, se van a modificar las funciones initialize(), finalize(), render(), key_pressed() y resize().
- Para la función update(), no se va a animar. Se eliminará el contenido.
- Las funciones mouse_move() y mouse_button() no tienen contenido.
- initialize(): comienza asignando los valores a los campos privados del modelo. Inicialmente la figura a mostrar será fig0 y los ángulos de orientación también serán 0. Para inicializar las figuras se hace una llamada a los constructores de cada una. Todas ellas son subclases de CGFigure y están declaradas en sus correspondientes archivos de cabecera. La inicialización también incluye una llamada a resize() para configurar el viewport y la matriz de proyección. Para terminar se incluyen algunas opciones de configuración de OpenGL, entre otras, se indica que los polígonos serán mostrados en modo arista para apreciar las caras.

2. Desarrollo del modelo

```
void CGModel::initialize(int w, int h)
{
  // Crea el programa
  program = new CGShaderProgram(IDR_SHADER1, IDR_SHADER2, -1, -1, -1);
  //program = new CGShaderProgram("shaders/VertexShader.glsl",
  // "shaders/FragmentShader.glsl", NULL, NULL, NULL);
  if (program->IsLinked() == GL_FALSE) return;
  program->Use();

// Inicializa la posición de las figuras
figure = 0;
xAngle = 0.0f;
yAngle = 0.0f;

// Crea las figuras
fig0 = new CGCube(25.0f);
fig1 = new CGPiramid(25.0f);
fig2 = new CGCone(5, 20, 25.0f, 25.0f);
fig3 = new CGCylinder(20, 20, 25.0f, 25.0f);
fig4 = new CGSphere(20, 20, 25.0f);
fig5 = new CGDisk(5, 20, 12.5f, 25.0f);
  fig6 = new CGTorus(10, 20, 10.0f, 25.0f);
  fig7 = new CGIcosahedron(15.0f);

  // Asigna el viewport y el clipping volume
  resize(w, h);

  // Opciones de dibujo
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_CULL_FACE);
  glFrontFace(GL_CCW);
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
}
```

2. Desarrollo del modelo

- finalize(): libera los recursos utilizados por el modelo

```
void CGModel::finalize()
{
  delete fig0;
  delete fig1;
  delete fig2;
  delete fig3;
  delete fig4;
  delete fig5;
  delete fig6;
  delete fig7;
  delete program;
}
```

2. Desarrollo del modelo

- resize(): respuesta ante modificaciones del tamaño de la ventana. También se llama dentro de initialize() para fijar el tamaño inicial.
  - En la practica anterior, esta función se limitaba a asignar el viewport de la imagen, ya que las coordenadas del triángulo se expresaban directamente en el sistema de coordenadas homogéneo.
  - Lo normal es que el modelo utilice su propio sistema de coordenadas y que se realice una transformación entre el volumen del modelo que se va a representar y el volumen Clip.
  - En este caso se va a realizar una transformación de tipo proyección en perspectiva.
  - La función glm::frustum() permite generar la matriz de transformación de esta proyección de manera sencilla.
  - Para que el aspecto de la imagen no se deforme, la proyección se calcula con la misma proporción de aspecto entre las coordenadas X e Y que existe en el tamaño de la ventana.

2. Desarrollo del modelo

```
void CGModel::resize(int w, int h)

{

  double fov = glm::radians(15.0);

  double sin_fov = sin(fov);

  double cos_fov = cos(fov);

  if (h == 0) h = 1;

  GLfloat aspectRatio = (GLfloat)w / (GLfloat)h;

  GLfloat wHeight = (GLfloat)(sin_fov * 0.2 /
cos_fov);

  GLfloat wWidth = wHeight * aspectRatio;


  glViewport(0, 0, w, h);

  projection = glm::frustum(-wWidth, wWidth, -
wHeight, wHeight, 0.2f, 400.0f);

  // projection = glm::perspective((float)fov,
(float)aspect, 0.2f, 400.0f);

}
```

2. Desarrollo del modelo

- render(): dibuja la escena. Aunque el dibujo a desarrollar es complejo, el contenido de esta función es muy simple, porque los aspectos del dibujo de cada figura son responsabilidad del método Draw() de cada objeto.
  - La función borra la escena (usando color blanco), selecciona el color rojo como el color de las figuras, realiza la traslación y las rotaciones respecto a los ejes X e Y y lanza el método Draw() de la figura seleccionada.
  - Para realizar las rotaciones se utiliza el método glm:rotate(), que genera una matriz de rotación considerando el ángulo de rotación (en radianes) y el eje de rotación.
  - Para realizar una traslación se utiliza el método glm:translate().
  - La matriz de transformación que se envía al VertexShader es el producto entre la matriz de proyección y la matriz de rotación y traslación.

2. Desarrollo del modelo

```
void CGModel::render()
{
  // Limpia el framebuffer
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  // Matriz de transformación
  glm::mat4 location = glm::mat4(1.0f);
  location = glm::translate(location, glm::vec3(0.0f, 0.0f, -180.0f));
  location = glm::rotate(location, glm::radians(yAngle),
glm::vec3(0.0f,1.0f,0.0f));
  location = glm::rotate(location, glm::radians(xAngle),
glm::vec3(1.0f,0.0f,0.0f));
  glm::mat4 transform = projection * location;
  switch (figure) {
  case 0: fig0->Draw(program, transform); break;
  case 1: fig1->Draw(program, transform); break;
  case 2: fig2->Draw(program, transform); break;
  case 3: fig3->Draw(program, transform); break;
  case 4: fig4->Draw(program, transform); break;
  case 5: fig5->Draw(program, transform); break;
  case 6: fig6->Draw(program, transform); break;
  case 7: fig7->Draw(program, transform); break;
  }
}
```

2. Desarrollo del modelo

- key_pressed(): respuesta del modelo a los eventos de teclado. Al pulsar los cursores arriba y abajo incrementaremos o decrementaremos el ángulo xAngle. Y cuando pulsemos los cursores izquierdo y derecho se modificará el ángulo yAngle. Al presionar ‘F’ se modificará el campo figure y mostrará otra figura.

```
void CGModel::key_pressed(int key)
{
 switch (key)
 {
 case GLFW_KEY_F:
  figure = (figure + 1) % 8;
  break;
 case GLFW_KEY_UP:
  xAngle += 5.0f;
  break;
 case GLFW_KEY_DOWN:
  xAngle -= 5.0f;
  break;
 case GLFW_KEY_LEFT:
  yAngle -= 5.0f;
  break;
 case GLFW_KEY_RIGHT:
  yAngle += 5.0f;
  break;
 }
}
```

2. Desarrollo del modelo

- Los métodos que quedan en blanco son:

```
void CGModel::mouse_button(int button, int action)
{
}
void CGModel::mouse_move(double xpos, double ypos)
{
}
```

```
//
// FUNCIÓN: CGModel::update()
//
// PROPÓSITO: Anima la escena
//
void CGModel::update()
{
}
```

2. Desarrollo del modelo

- Includes de CGModel.cpp:

```
#include "CGModel.h"
#include <GLFW\glfw3.h>
#include <iostream>
#include <glm/gtc/matrix_transform.hpp>
#include "resource.h"
```

3. Programa gráfico

- El programa a ejecutar en la tarjeta gráfica está compuesto por el VertexShader y el FragmentShader.
- Códgo del VertexShader.glsl:

```
#version 400
in vec3 VertexPosition;
uniform mat4 Transform;
void main()
{
   gl_Position = Transform * vec4(VertexPosition, 1.0);
}
```

3. Programa gráfico

- Con respecto a la práctica anterior se ha modificado el VertexShader para considerar una variable uniforme que contiene una matriz 4x4.
  - Esta matriz permite realizar una transformación sobre la posición de los vértices.
- Por su parte el FragmentShader sigue siendo el mismo que el utilizado en la práctica anterior. Sólo pinta en rojo los píxeles de cada fragmento.

```
#version 400

out vec4 FragColor;

void main()

{

  FragColor = vec4(1.0,0.0,0.0,1.0);

}
```

4. Figuras geométricas

- Todas las figuras geométricas que vamos a desarrollar en este proyecto van a ser subclases de una clase abstracta llamada CGFigure.
- Esta clase va a describir un objeto tridimensional formado por triángulos.
- Para construir los objetos se utilizará una lista de vértices (cuyas posiciones se van a almacenar en el campo vértices.
- Para evitar duplicidad de vértices, los triángulos se van a referenciar por medio de una lista de índices que se almacenan en el campo índices.
- Los campos numVertices y numFaces almacenarán el número de vértices y triángulos que forman el objeto
- Para definir los objetos en el programa gráfico se utiliza un VertexArrayObject (cuyo identificador se almacena en el campo VAO) y dos VertexBufferObjects (cuyos identificadores se almacenan en el campo VBO).

4. Figuras geométricas

- Cabecera CGFigure.h:

```
#pragma once

#include <GL/glew.h>
#include <glm/glm.hpp>
#include "CGShaderProgram.h"

#define VERTEX_DATA     0
#define INDEX_DATA      1

//
// CLASE: CGFigure
//   VAO para su renderizado mediante shaders
//
class CGFigure {
protected:
  GLushort* indexes;  // Array of indexes
  GLfloat* vertices;  // Array of vertices

  int numFaces;     // Number of faces
  int numVertices;  // Number of vertices
  GLuint VBO[2];
  GLuint VAO;

public:
  ~CGFigure();
  void InitBuffers();
  void Draw(CGShaderProgram* program, glm::mat4 transform);
};
```

4. Figuras geométricas

- Los métodos de la clase CGFigure, que utilizarán por herencia todas las figuras geométricas, son:
  - El destructor de la clase
  - El método InitBuffers()
  - El método Draw().
- Las subclases de CGFigure deberán crear el contenido de los campos vértices e índices en su constructor.
- A partir de esos valores, el método InitBuffers() se encarga de crear el VertexArrayObject y los dos VertexBufferObjects, almacenar el contenido de los buffers tomándolo de los campos vértices e índices y definir los atributos del VertexArrayObject.

4. Figuras geométricas

- El método Draw() es el encargado de lanzar el dibujo de la figura. Para ello, asigna la matriz de transformación como variable uniforme, activa el VertexArrayObject y lanza el método glDrawElements() considerando el número de triángulos a dibujar.
- El destructor de la clase se encarga de liberar los buffers.
- Contenido de CGFigure.cpp

```
#include "CGFigure.h"

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <glm/gtc/matrix_transform.hpp>
```

4. Figuras geométricas

```
CGFigure::~CGFigure()
{
  if (vertices != NULL) delete[] vertices;
  if (indexes != NULL) delete[] indexes;

  // Delete vertex buffer objects
  glDeleteBuffers(2, VBO);

  // Delete vertex array object
  glDeleteVertexArrays(1, &VAO);
}
```

4. Figuras geométricas

```
void CGFigure::InitBuffers()
{
// Create the Vertex Array Object
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// Create the Vertex Buffer Objects
glGenBuffers(2, VBO);

// Copy data to video memory
// Vertex data
glBindBuffer(GL_ARRAY_BUFFER, VBO[VERTEX_DATA]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * numVertices * 3, vertices, GL_STATIC_DRAW);

// Indexes
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, VBO[INDEX_DATA]);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLushort) * numFaces * 3, indexes, GL_STATIC_DRAW);

delete[] vertices;
delete[] indexes;

vertices = NULL;
indexes = NULL;

glEnableVertexAttribArray(0); // Vertex position
glBindBuffer(GL_ARRAY_BUFFER, VBO[VERTEX_DATA]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
}
```

4. Figuras geométricas

```
void CGFigure::Draw(CGShaderProgram* program, glm::mat4 transform)
{
  glm::mat4 matrix = transform;
  program->SetUniformMatrix4("Transform", transform);

  glBindVertexArray(VAO);
  glDrawElements(GL_TRIANGLES, numFaces * 3, GL_UNSIGNED_SHORT, NULL);
}
```

5. Descripción de un cubo

- Cabecera CGCube.h:

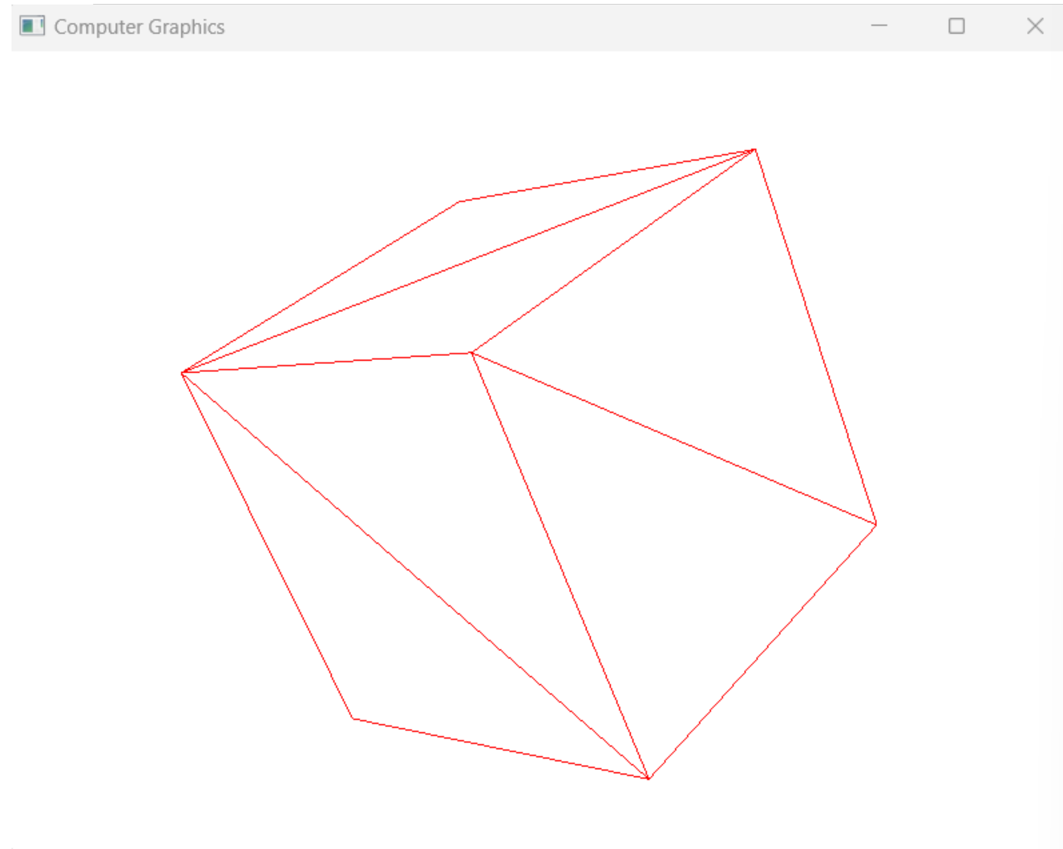
```
#pragma once

#include <GL/glew.h>
#include "CGFigure.h"

//
// CLASE: CGCube
//
// DESCRIPCIÓN: Representa un cubo de lado 2·s.
//
class CGCube : public CGFigure {
public:
  CGCube(GLfloat s);
};
```

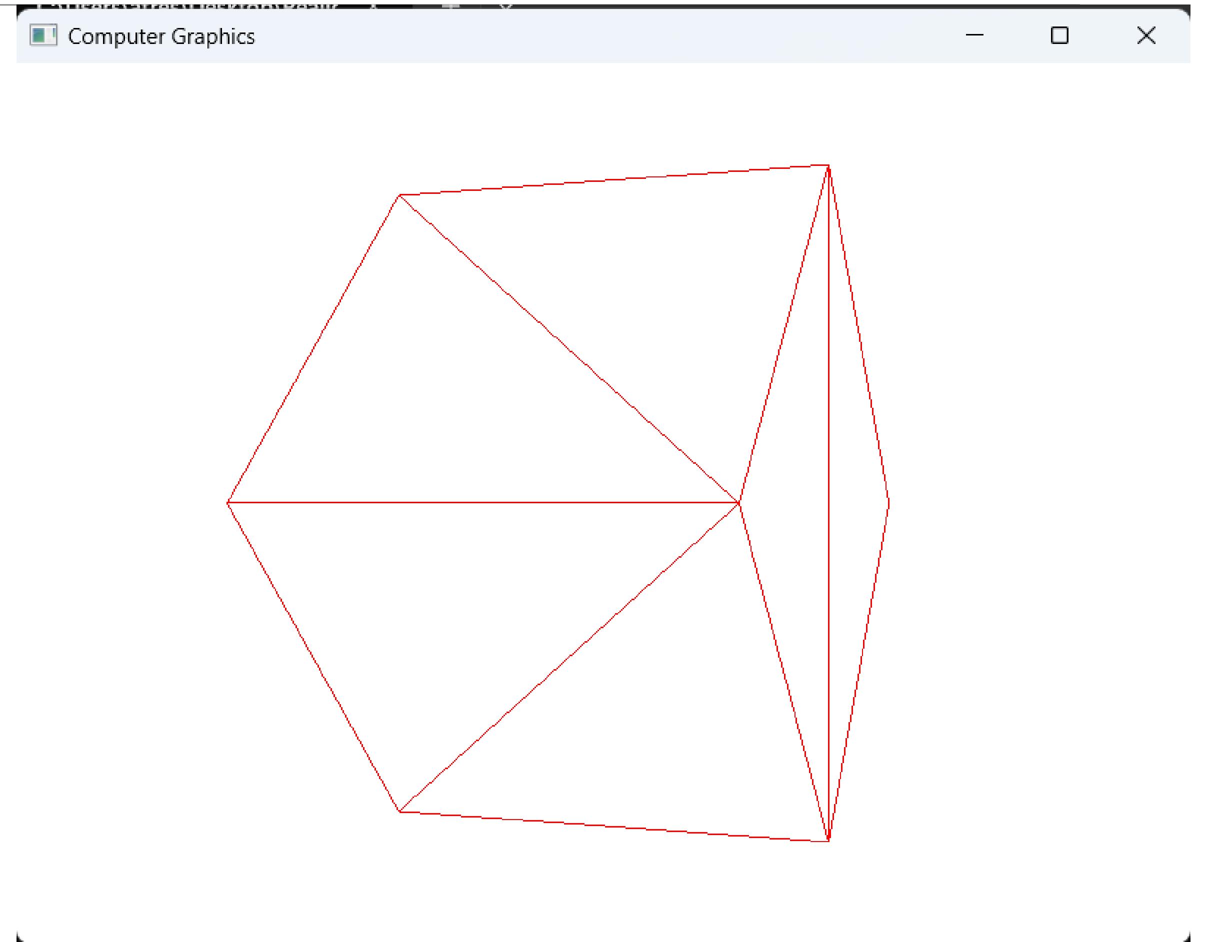
5. Descripción de un cubo

- Código CGCube.cpp:

```
#pragma once

#include <GL/glew.h>
#include "CGFigure.h"

//
// CLASE: CGCube
//
// DESCRIPCIÓN: Representa un cubo de lado 2·s.
//
class CGCube : public CGFigure {
public:
  CGCube(GLfloat s);
};
```

5. Descripción de un cubo





6. Descripción de una pirámide

- Cabecera CGPiramid.h

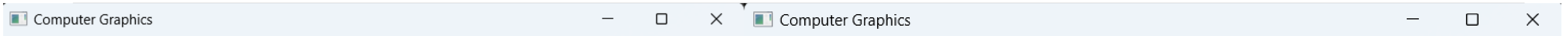
```
#pragma once


#include <GL/glew.h>

#include "CGFigure.h"


//

// CLASE: CGPiramid

//

// DESCRIPCIÓN: Representa una pirámide de base cuadrada de

//             lado 2·s y altura 2·s.

//

class CGPiramid : public CGFigure {

public:

  CGPiramid(GLfloat s);

};
```

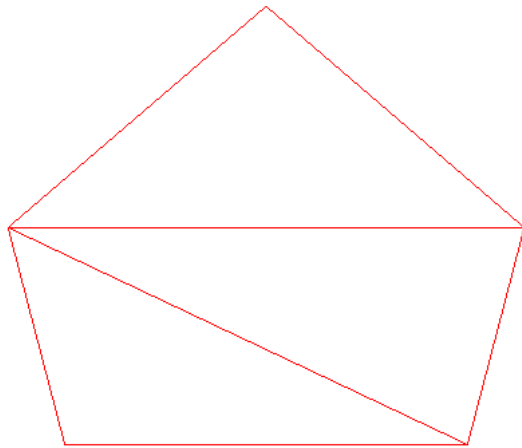
6. Descripción de una pirámide

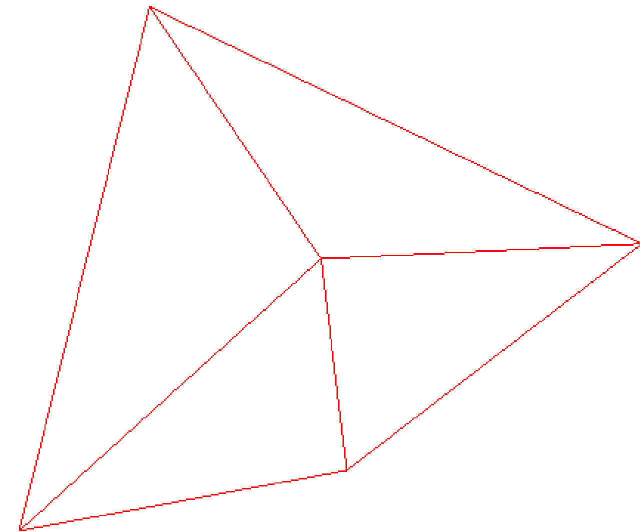
- Contenido CGPiramid.cpp

```
#include "CGPiramid.h"
#include <GL/glew.h>
#include <math.h>
#include "CGFigure.h"
CGPiramid::CGPiramid(GLfloat s)
  numFaces = 6; // Number of faces
  numVertices = 16; // Number of vertices
  GLfloat p_vertices[16][3] = {
    { 0.0f, 0.0f, s }, // E // Positive X
    { s, -s, -s }, // B
    { s, s, -s }, // A
    { 0.0f, 0.0f, s }, // E // Positive Y
    { s, s, -s }, // A
    { -s, s, -s }, // D
    { 0.0f, 0.0f, s }, // E // Negative X
    { -s, s, -s }, // D
    { -s, -s, -s }, // C
    { 0.0f, 0.0f, s }, // E // Negative Y
    { -s, -s, -s }, // C
    { s, -s, -s }, // B
    { s, s, -s }, // A // Negative Z
    { s, -s, -s }, // B
    { -s, -s, -s }, // C
    { -s, s, -s }, // D
  };
  GLushort p_indexes[6][3] = { // Array of indexes
    { 0, 1, 2 },
    { 3, 4, 5 },
    { 6, 7, 8 },
    { 9, 10, 11 },
    { 12, 13, 14 },
    { 12, 14, 15 }
  };
  vertices = new GLfloat[numVertices * 3];
  for (int i = 0; i < numVertices; i++)
    for (int j = 0; j < 3; j++)
      vertices[3 * i + j] = p_vertices[i][j];
  indexes = new GLushort[numFaces * 3];
  for (int i = 0; i < numFaces; i++)
    for (int j = 0; j < 3; j++)
      indexes[3 * i + j] = p_indexes[i][j];

  InitBuffers();
}
```

6. Descripción de una pirámide







7. Descripción de un cono

- Cabecera GCCone.h

```
#pragma once


#include <GL/glew.h>
#include "CGFigure.h"


//
// CLASE: CGCone
//
// DESCRIPCIÓN: Representa un cono de altura 'h', radio 'r',
//              dividido en 'm' líneas y 'p' rodajas.
//
class CGCone : public CGFigure {
public:
  CGCone(GLint p, GLint m, GLfloat h, GLfloat r);
};
```

7. Descripción de un cono

- Contenido GCCone.cpp

```
#include "CGCone.h"
#include <GL/glew.h>
#include <math.h>
#include "CGFigure.h"


CGCone::CGCone(GLint p, GLint m, GLfloat h, GLfloat r)
{
  numFaces = 2 * p * m; // Number of faces
  numVertices = (p+1)*m+2; // Number of vertices
  vertices = new GLfloat[numVertices * 3];
  indexes = new GLushort[numFaces * 3];

double module = sqrt(4 * h*h + r*r);
double xyN = (GLfloat)(2 * h / module);
double zN = (GLfloat)(r / module);

int verticesIndex = 0;
// Centro de la base
vertices[0] = 0.0f;
vertices[1] = 0.0f;
vertices[2] = -h;
verticesIndex += 3;

  // Vértices de la base
  for (int i = 0; i <m; i++)
  {
    GLfloat x = (GLfloat)cos(glm::radians(360.0 * i / m));
    GLfloat y = -(GLfloat)sin(glm::radians(360.0 * i / m));
    vertices[verticesIndex] = x*r;
    vertices[verticesIndex + 1] = y*r;
    vertices[verticesIndex + 2] = -h;
    verticesIndex += 3;
  }
```

7. Descripción de un cono

- Contenido GCCone.cpp

```
// Extremo del cono
  vertices[verticesIndex] = 0.0f;
  vertices[verticesIndex+1] = 0.0f;
  vertices[verticesIndex+2] = h;
  verticesIndex += 3;

  // Vértices de los lados
  for (int i = 1; i <= p; i++)
  {
    GLfloat xy = i*r / p;
    GLfloat z = h - 2 * i*h / p;
    for (int j = 0; j < m; j++)
    {
      GLfloat xN = (GLfloat)cos(glm::radians(360.0 * j / m));
      GLfloat yN = (GLfloat)sin(glm::radians(360.0 * j / m));
      GLfloat x = (GLfloat)(xN*xy);
      GLfloat y = (GLfloat)(yN*xy);
      vertices[verticesIndex] = x;
      vertices[verticesIndex + 1] = y;
      vertices[verticesIndex + 2] = z;
      verticesIndex += 3;
    }
  }

  int indicesIndex = 0;
  // Base
  for (int i = 0; i < m - 1; i++)
  {
    indexes[indicesIndex] = 0;
    indexes[indicesIndex + 1] = i + 1;
    indexes[indicesIndex + 2] = i + 2;
    indicesIndex += 3;
  }
```

7. Descripción de un cono

- Contenido GCCone.cpp

```
indexes[indicesIndex] = 0;
  indexes[indicesIndex + 1] = m;
  indexes[indicesIndex + 2] = 1;
  indicesIndex += 3;

  // Extremo
  for (int i = 0; i < m - 1; i++)
  {
    indexes[indicesIndex] = m + 1;
    indexes[indicesIndex + 1] = m + 2 + i;
    indexes[indicesIndex + 2] = m + 3 + i;
    indicesIndex += 3;
  }
  indexes[indicesIndex] = m + 1;
  indexes[indicesIndex + 1] = 2*m + 1;
  indexes[indicesIndex + 2] = m + 2;
  indicesIndex += 3;

  // Lados
  for (int j = 1; j < p; j++)
  {
    for (int i = 0; i < m - 1; i++)
    {
      indexes[indicesIndex] = j*m + 2 + i;
      indexes[indicesIndex + 1] = (j+1)*m + 2 + i;
```

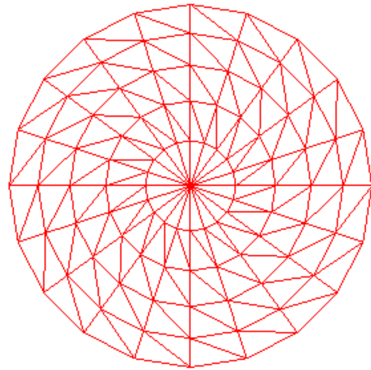
7. Descripción de un cono

- Contenido GCCone.cpp

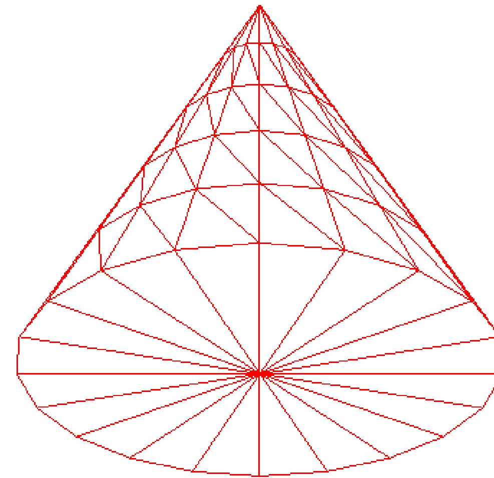
```
  indexes[indicesIndex + 2] = (j+1)*m + 3 + i;
        indicesIndex += 3;

        indexes[indicesIndex] = j*m + 2 + i;
        indexes[indicesIndex + 1] = (j + 1)*m + 3 + i;
        indexes[indicesIndex + 2] = j*m + 3 + i;
        indicesIndex += 3;
      }

      indexes[indicesIndex] = (j+1)*m + 1;
      indexes[indicesIndex + 1] = (j + 2)*m + 1;
      indexes[indicesIndex + 2] = (j + 1)*m + 2;
      indicesIndex += 3;

      indexes[indicesIndex] = (j+1)*m + 1;
      indexes[indicesIndex + 1] = (j + 1)*m + 2;
      indexes[indicesIndex + 2] = j*m + 2;
      indicesIndex += 3;
    }

  InitBuffers();
}
```

7. Descripción de un cono

Computer Graphics



Computer Graphics



8. Descripción de un cilindro

- Cabecera CGCylinder.h

```
#pragma once

#include <GL/glew.h>
#include "CGFigure.h"

//
// CLASE: CGCylinder
//
// DESCRIPCIÓN: Representa un cilindro de radio 'r', longitud '2·l',
//              dividido en 'p' capas y 'm' líneas.
//
class CGCylinder : public CGFigure {
public:
  CGCylinder(GLint p, GLint m, GLfloat r, GLfloat l);
};
```

8. Descripción de un cilindro

- Contenido CGCylinder.cpp

```
#include "CGCylinder.h"
#include <GL/glew.h>
#include <math.h>
#include "CGFigure.h"

CGCylinder::CGCylinder(GLint p, GLint m, GLfloat r, GLfloat l)
{
  numFaces = 2 * m * (p + 1); // Number of faces
  numVertices = (m + 1)*(p + 3); // Number of vertices
  vertices = new GLfloat[numVertices * 3];
  indexes = new GLushort[numFaces * 3];

  int verticesIndex = 0;
  int indexesIndex = 0;

  /* northern polar cap*/
  vertices[0] = 0.0f;
  vertices[1] = 0.0f;
  vertices[2] = l;
  verticesIndex += 3;

  for (int j = 0; j < m; j++)
  {
    GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
    GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
```

8. Descripción de un cilindro

- Contenido CGCylinder.cpp

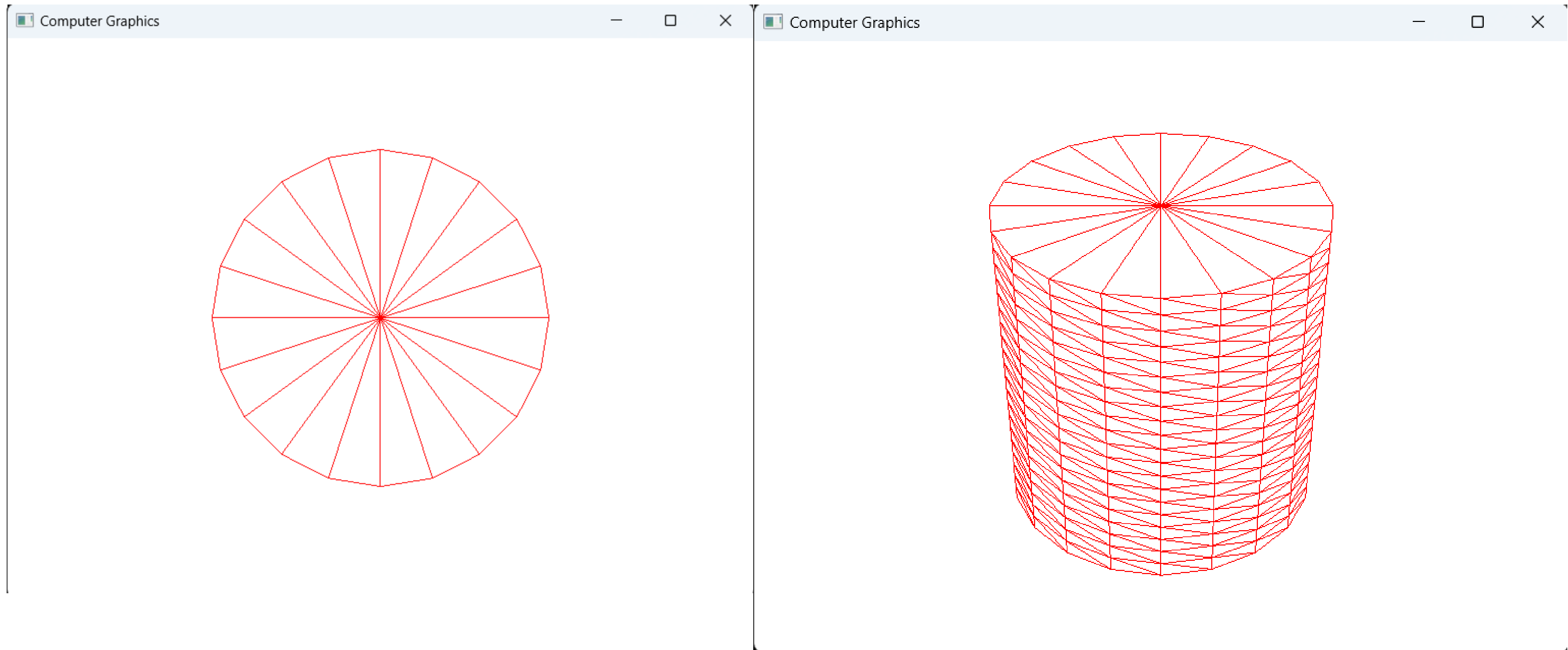
```
 vertices[verticesIndex] = mCos * r;
      vertices[verticesIndex + 1] = mSin * r;
      vertices[verticesIndex + 2] = l;
      verticesIndex += 3;

      indexes[indexesIndex] = 0; // center
      indexes[indexesIndex + 1] = j + 1;
      indexes[indexesIndex + 2] = (j + 2>m ? 1 : j
+ 2);
      indexesIndex += 3;
   }
 /* southern polar cap*/
  vertices[verticesIndex] = 0.0f;
  vertices[verticesIndex + 1] = 0.0f;
  vertices[verticesIndex + 2] = -l;
  verticesIndex += 3;

  for (int j = 0; j < m; j++)
  {
     GLfloat mCos =
(GLfloat)cos(glm::radians(360.0 * j / m));
     GLfloat mSin =
(GLfloat)sin(glm::radians(360.0 * j / m));
     vertices[verticesIndex] = mCos * r;
     vertices[verticesIndex + 1] = -mSin * r;
      vertices[verticesIndex + 2] = -l;
      verticesIndex += 3;

      indexes[indexesIndex] = m + 1; // center
      indexes[indexesIndex + 1] = j + m + 2;
      indexes[indexesIndex + 2] = (j + 2>m ? m +
2 : j + m + 3);
      indexesIndex += 3;
   }
```

8. Descripción de un cilindro

- Contenido CGCylinder.cpp

```
/* body */
  for (int i = 0; i <= p; i++)
  {
    for (int j = 0; j <= m; j++)
    {
      GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
      GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
      vertices[verticesIndex] = mCos * r;
      vertices[verticesIndex + 1] = mSin * r;
      vertices[verticesIndex + 2] = l - 2 * l*i / p;
      verticesIndex += 3;
    }
  }

  int base = 2 * m + 2;
  for (int i = 0; i < p; i++)
  {
    for (int j = 0; j < m; j++)
    {
      indexes[indexesIndex] = base + (m + 1)*i + j;
      indexes[indexesIndex + 1] = base + (m + 1)*(i + 1) + j;
      indexes[indexesIndex + 2] = base + (m + 1)*(i + 1) + j + 1;
      indexesIndex += 3;
      indexes[indexesIndex] = base + (m + 1)*i + j;
      indexes[indexesIndex + 1] = base + (m + 1)*(i + 1) + j + 1;
      indexes[indexesIndex + 2] = base + (m + 1)*i + j + 1;
      indexesIndex += 3;
    }
  }

  InitBuffers();
}
```

8. Descripción de un cilindro



9. Descripción de una esfera

- Cabecera CGSphere.h

```
#pragma once


#include <GL/glew.h>
#include "CGFigure.h"


//
// CLASE: CGSphere
//
// DESCRIPCIÓN: Representa una esfera de radio 'r', dividida en 'p'
//             capas (paralelos) y 'm' líneas (meridianos).
//
class CGSphere : public CGFigure {
public:
  CGSphere(GLint p, GLint m, GLfloat r);
};
```

9. Descripción de una esfera

- Contenido CGSphere.cpp

```
#include "CGSphere.h"
#include <GL/glew.h>
#include <math.h>
#include "CGFigure.h"



CGSphere::CGSphere(GLint p, GLint m, GLfloat r)
{
  numFaces = 2 * m*(p - 1); // Number of faces
  numVertices = (m + 1)*(p + 1); // Number of vertices
  vertices = new GLfloat[numVertices * 3];
  indexes = new GLushort[numFaces * 3];

  int verticesIndex = 0;
  int indexesIndex = 0;
  /* northern polar cap*/
  for (int j = 0; j <= m; j++)
  {
    vertices[verticesIndex] = 0.0;
    vertices[verticesIndex + 1] = 0.0f;
    vertices[verticesIndex + 2] = r;
    verticesIndex += 3;
  }

  for (int i = 1; i < p; i++)
  {
    for (int j = 0; j <= m; j++)
```

9. Descripción de una esfera

- Contenido CGSphere.cpp

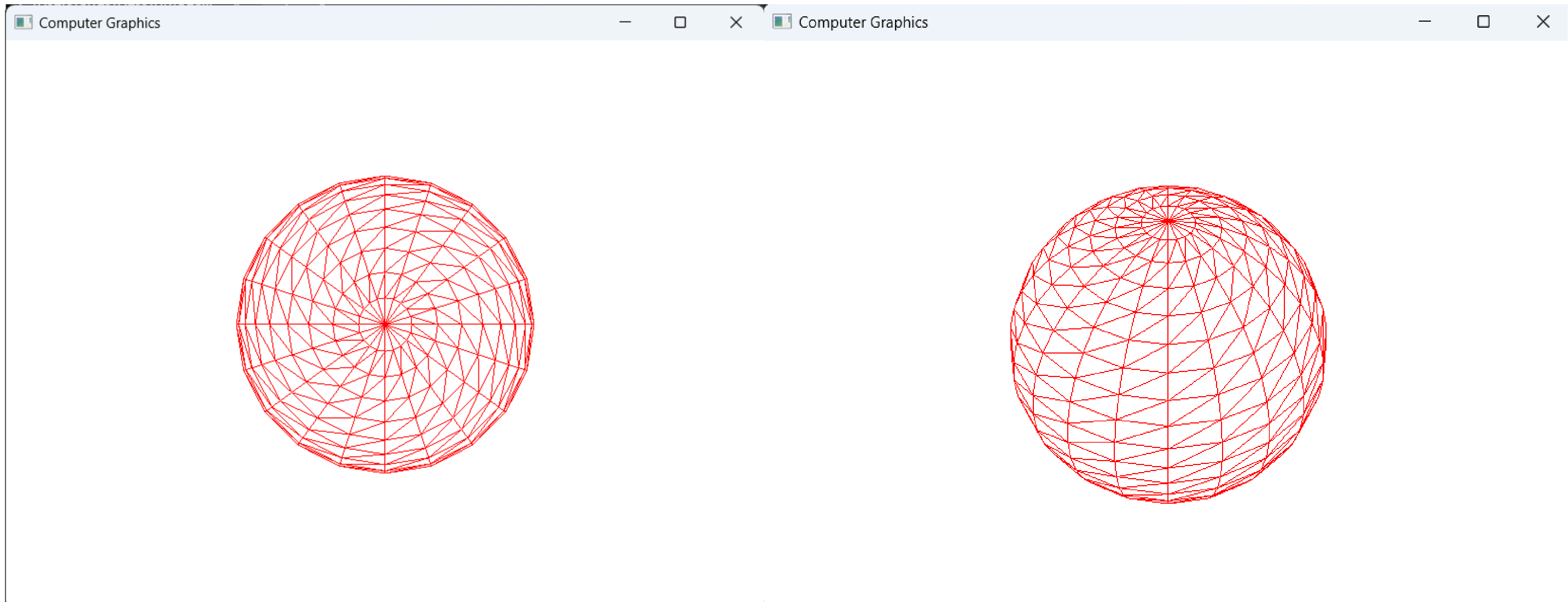
```
{
	GLfloat pCos = (GLfloat)cos(glm::radians(180.0 * i / p));
	GLfloat pSin = (GLfloat)sin(glm::radians(180.0 * i / p));
	GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
	GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));

	vertices[verticesIndex] = pSin*mCos*r;
	vertices[verticesIndex + 1] = pSin*mSin*r;
	vertices[verticesIndex + 2] = pCos*r;
	verticesIndex += 3;
  }
}
/* southern polar cap*/
 for (int j = 0; j <= m; j++)
 {
   vertices[verticesIndex] = 0.0;
   vertices[verticesIndex + 1] = 0.0f;
   vertices[verticesIndex + 2] = -r;
   verticesIndex += 3;
 }
```

9. Descripción de una esfera

- Contenido CGSphere.cpp

```
/* northern polar cap*/
  for (int j = 0; j < m; j++)
  {
    indexes[indexesIndex] = j;
    indexes[indexesIndex + 1] = m + j + 1;
    indexes[indexesIndex + 2] = m + j + 2;
    indexesIndex += 3;
  }
  for (int i = 1; i < p - 1; i++)
  {
    for (int j = 0; j < m; j++)
    {
      indexes[indexesIndex] = i*(m + 1) + j;
      indexes[indexesIndex + 1] = (i + 1)*(m + 1) + j;
      indexes[indexesIndex + 2] = i*(m + 1) + j + 1;
      indexes[indexesIndex + 3] = (i + 1)*(m + 1) + j;
      indexes[indexesIndex + 4] = (i + 1)*(m + 1) + j + 1;
      indexes[indexesIndex + 5] = i*(m + 1) + j + 1;
      indexesIndex += 6;
    }
  }
  for (int j = 0; j < m; j++)
  {
    indexes[indexesIndex] = (p - 1)*(m + 1) + j;
    indexes[indexesIndex + 1] = p*(m + 1) + j;
    indexes[indexesIndex + 2] = (p - 1)*(m + 1) + j + 1;
    indexesIndex += 3;
  }

  InitBuffers();
}
```

9. Descripción de una esfera



10. Descripción de un disco

- Cabecera CGDisk.h

```
#pragma once

#include <GL/glew.h>
#include "CGFigure.h"

//
// CLASE: CGDisk
//
// DESCRIPCIÓN: Representa un disco de radio interior 'r0' (puede ser cero),
//              radio exterior 'r1', dividido en 'p' capas y 'm' sectores.
//
class CGDisk : public CGFigure {
public:
        CGDisk(GLint p, GLint m, GLfloat r0, GLfloat r1);
};
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
#include "CGDisk.h"
#include <GL/glew.h>
#include <math.h>
#include "CGFigure.h"

CGDisk::CGDisk(GLint p, GLint m, GLfloat r0, GLfloat r1)
{
  if (r0 == 0.0f)
  {
      numFaces = 2*(2 * m * p - m); // Number of faces
      numVertices = 2*(m* p + 1); // Number of vertices
      vertices = new GLfloat[numVertices * 3];
      indexes = new GLushort[numFaces * 3];

      int verticesIndex = 0;
      int indexesIndex = 0;
vertices[0] = 0.0f;
vertices[1] = 0.0f;
vertices[2] = 0.0;
verticesIndex += 3;

for (int j = 0; j < m; j++)
{
  GLfloat r = (GLfloat) (r1 / p);
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
 GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
        GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
        vertices[verticesIndex] = mCos * r;
        vertices[verticesIndex + 1] = mSin * r;
        vertices[verticesIndex + 2] = 0.0f;
        verticesIndex += 3;

        indexes[indexesIndex] = 0; // center
        indexes[indexesIndex + 1] = j + 1;
        indexes[indexesIndex + 2] = (j + 2>m ? 1 : j + 2);
        indexesIndex += 3;
    }
for (int i = 2; i <= p; i++)
    {
        for (int j = 0; j < m; j++)
        {
            GLfloat r = (GLfloat) (r1*i / p);
            GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
            GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
            vertices[verticesIndex] = mCos * r;
            vertices[verticesIndex + 1] = mSin * r;
            vertices[verticesIndex + 2] = 0.0f;
            verticesIndex += 3;
        }
    }
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
for (int i = 0; i < p - 1; i++)
    {
      for (int j = 0; j < m; j++)
      {
        indexes[indexesIndex] = m * i + j + 1;
        indexes[indexesIndex + 1] = m *(i + 1) + j + 1;
        indexes[indexesIndex + 2] = (j+1 == m ? m*(i+1) + 1 : m*(i+1) + j + 2);
        indexesIndex += 3;

        indexes[indexesIndex] = m *i + j + 1;
        indexes[indexesIndex + 1] = (j+1 == m ? m*(i+1) + 1 : m*(i+1) + j + 2);
        indexes[indexesIndex + 2] = (j+1 == m ? m*i + 1 : m*i + j + 2);
        indexesIndex += 3;
      }
    }

int base = verticesIndex/3;

vertices[base+0] = 0.0f;
vertices[base+1] = 0.0f;
vertices[base+2] = 0.0;
verticesIndex += 3;
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
for (int j = 0; j < m; j++)
    {
        GLfloat r = (GLfloat) (r1 / p);
        GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
        GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
        vertices[verticesIndex] = mCos * r;
        vertices[verticesIndex + 1] = mSin * r;
        vertices[verticesIndex + 2] = 0.0f;
        verticesIndex += 3;
        indexes[indexesIndex] = base + 0; // center
        indexes[indexesIndex + 1] = base+ (j + 2>m ? 1 : j + 2);
        indexes[indexesIndex + 2] = base + j + 1;
        indexesIndex += 3;
    }

    for (int i = 2; i <= p; i++)
    {
        for (int j = 0; j < m; j++)
        {
            GLfloat r = (GLfloat) (r1*i / p);
            GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
            GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
            vertices[verticesIndex] = mCos * r;
            vertices[verticesIndex + 1] = mSin * r;
            vertices[verticesIndex + 2] = 0.0f;
            verticesIndex += 3;
        }
    }
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
for (int i = 2; i <= p; i++)
   {
     for (int j = 0; j < m; j++)
     {
       GLfloat r = (GLfloat) (r1*i / p);
       GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
       GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
       vertices[verticesIndex] = mCos * r;
       vertices[verticesIndex + 1] = mSin * r;
       vertices[verticesIndex + 2] = 0.0f;
       verticesIndex += 3;
     }
   }

   for (int i = 0; i < p - 1; i++)
   {
     for (int j = 0; j < m; j++)
     {
       indexes[indexesIndex] = base + m * i + j + 1;
       indexes[indexesIndex + 1] = base + (j + 1 == m ? m*(i+1)+1 : m*(i+1)+j+2);
       indexes[indexesIndex + 2] = base + m *(i + 1) + j + 1;
       indexesIndex += 3;

       indexes[indexesIndex] = base + m *i + j + 1;
       indexes[indexesIndex + 1] = base + (j + 1 == m ? m*i + 1 : m*i + j + 2);
       indexes[indexesIndex + 2] = base + (j + 1 == m ? m*(i+1)+1 : m*(i+1)+j+2);
       indexesIndex += 3;
     }
   }
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
}
  else
  {
    numFaces = 2*(2 * m * p); // Number of faces
    numVertices = 2*(m* (p + 1)); // Number of vertices
    vertices = new GLfloat[numVertices * 3];
    indexes = new GLushort[numFaces * 3];
int verticesIndex = 0;
int indexesIndex = 0;

for (int i = 0; i <= p; i++)
{
  GLfloat r = r0 + (r1 - r0)*i / p;
  for (int j = 0; j < m; j++)
  {
    GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
    GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
    vertices[verticesIndex] = mCos * r;
    vertices[verticesIndex + 1] = mSin * r;
    vertices[verticesIndex + 2] = 0.0f;
    verticesIndex += 3;
  }
}

for (int i = 0; i < p; i++)
{
```

10. Descripción de un disco

- Contenido CGDisk.cpp

```
for (int j = 0; j < m; j++)
        {
          indexes[indexesIndex] = m *i + j;
          indexes[indexesIndex + 1] = m *(i + 1) + j;
          indexes[indexesIndex + 2] = (j + 1 == m ? m *(i + 1) : m*(i + 1) + j + 1);
          indexesIndex += 3;

          indexes[indexesIndex] = m *i + j;
          indexes[indexesIndex + 1] = (j + 1 == m ? m *(i + 1) : m*(i + 1) + j + 1);
          indexes[indexesIndex + 2] = (j + 1 == m ? m *i : m *i + j + 1);
          indexesIndex += 3;
        }
    }
int base = verticesIndex/3;

    for (int i = 0; i <= p; i++)
    {
      GLfloat r = r0 + (r1 - r0)*i / p;
      for (int j = 0; j < m; j++)
      {
          GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * j / m));
          GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * j / m));
          vertices[verticesIndex] = mCos * r;
          vertices[verticesIndex + 1] = mSin * r;
          vertices[verticesIndex + 2] = 0.0f;
          verticesIndex += 3;
      }
    }
```

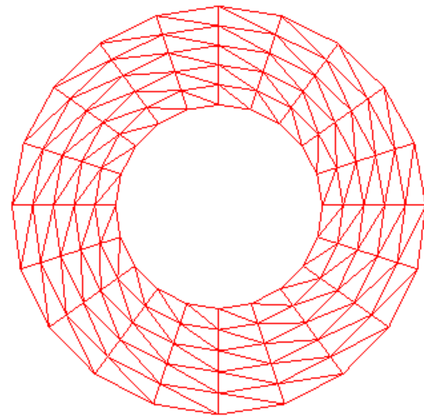
10. Descripción de un disco

- Contenido CGDisk.cpp

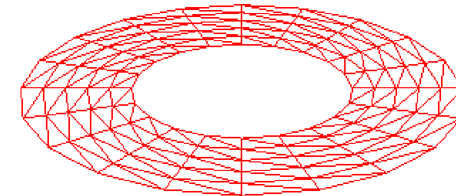
```
for (int i = 0; i < p; i++)
    {
      for (int j = 0; j < m; j++)
      {
        indexes[indexesIndex] = base + m *i + j;
        indexes[indexesIndex + 1] = base + (j+1 == m ? m *(i+1) : m*(i+1)+j+1);
        indexes[indexesIndex + 2] = base + m *(i + 1) + j;
        indexesIndex += 3;

        indexes[indexesIndex] = base + m *i + j;
        indexes[indexesIndex + 1] = base + (j+1 == m ? m *i : m *i + j + 1);
        indexes[indexesIndex + 2] = base + (j+1 == m ? m *(i+1) : m*(i+1)+j+1);
        indexesIndex += 3;
      }
    }
  }

  InitBuffers();
}
```

10. Descripción de un disco

Computer Graphics



Computer Graphics



11. Descripción de un toro

- Cabecera CGTorus.h

```
#pragma once


#include <GL/glew.h>
#include "CGFigure.h"


//
// CLASE: CGTorus
//
// DESCRIPCIÓN: Representa un toro con radio interior 'r0', radio exterior 'r1',
//              dividido en 'p' capas y 'm' sectores.
//
class CGTorus : public CGFigure {
public:
  CGTorus(GLint p, GLint m, GLfloat r0, GLfloat r1);
};
```

11. Descripción de un toro

- Contenido CGTorus.cpp

```
#include "CGTorus.h"
#include <GL/glew.h>
#include <math.h>
#include "CGFigure.h"

CGTorus::CGTorus(GLint p, GLint m, GLfloat r0, GLfloat r1)
{
  numFaces = 2 * m * p; // Number of faces
  numVertices = (m + 1)*(p + 1); // Number of vertices
  vertices = new GLfloat[numVertices * 3];
  indexes = new GLushort[numFaces * 3];

  int verticesIndex = 0;
  int indexesIndex = 0;

  for (int i = 0; i <= m; i++)
  {
    for (int j = 0; j <= p; j++)
    {
      GLfloat pCos = (GLfloat)cos(glm::radians(360.0 * j / p));
      GLfloat pSin = (GLfloat)sin(glm::radians(360.0 * j / p));
      GLfloat mCos = (GLfloat)cos(glm::radians(360.0 * i / m));
      GLfloat mSin = (GLfloat)sin(glm::radians(360.0 * i / m));

      vertices[verticesIndex] = (r1 + r0*pCos)*mCos;
      vertices[verticesIndex + 1] = (r1 + r0*pCos)*mSin;
      vertices[verticesIndex + 2] = r0*pSin;
      verticesIndex += 3;
    }
  }
```

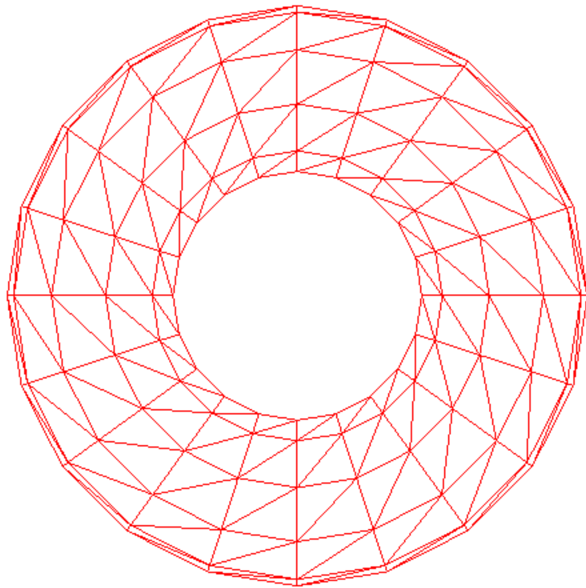
11. Descripción de un toro

- Contenido CGTorus.cpp

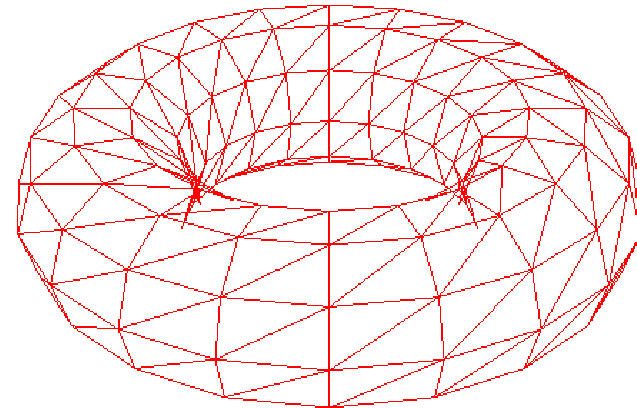
```
for (int i = 0; i < m; i++)
  {
    for (int j = 0; j < p; j++)
    {
      indexes[indexesIndex] = (p + 1)*i + j;
      indexes[indexesIndex + 1] = (p + 1)*(i + 1) + j;
      indexes[indexesIndex + 2] = (p + 1)*(i + 1) + j + 1;
      indexesIndex += 3;

      indexes[indexesIndex] = (p + 1)*i + j;
      indexes[indexesIndex + 1] = (p + 1)*(i + 1) + j + 1;
      indexes[indexesIndex + 2] = (p + 1)*i + j + 1;
      indexesIndex += 3;
    }
  }

  InitBuffers();
}
```

11. Descripción de un toro

Computer Graphics



Computer Graphics



12. Descripción de un icosaedro