Realidad Virtual. Práctica 4.

TRANSFORMACIONES GEOMÉTRICAS

Índice

1. Objetivos

- Uso de matrices de transformación para trabajar con varios sistemas de coordenadas.
- Definir los objetos en su propio sistema de coordenadas (local)
- Situarlos en una escala definida como un sistema de coordenadas del modelo (model)
- Mover el punto de vista del observador por medio de un objeto cámara que utiliza su propio sistema de coordenadas (view)
- Transformación entre este sistema de coordenadas y el sistema de coordenadas de Clip (esto se comnoce como proyección)

- Enlace al código de la práctica: https://www.uhu.es/francisco.moreno/gii_rv/practicas/practica04/practica04.rar

2. La clase CGFigure

- Para considerar las transformaciones que permitan situar figuras en diferentes posiciones dentro de una escena es necesario introducir cambios en CGFigure.
- Estos cambios se centran en incluir un campo location que almacena la matriz de transformación entre el sistema de coordenadas de la figura (local) y el sistema de coordenadas del modelo (model).
- Para facilitar la colocación de las figuras en la escena se añaden unos métodos para modificar la situación del objeto por medio de traslaciones y rotaciones.

2. La clase CGFigure

- 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
//
// DESCRIPCIÓN: Clase abstracta que representa un objeto descrito mediante
//              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;

  glm::mat4 location; // Model matrix

public:
  ~CGFigure();
  void InitBuffers();
  void ResetLocation();
  void Translate(glm::vec3 t);
  void Rotate(GLfloat angle, glm::vec3 axis);
  void Draw(CGShaderProgram* program, glm::mat4 projection, glm::mat4 view);
};
```

2. La clase CGFigure

- ResetLocation(): asigna la matriz unidad al campo location.
  - El sistema de coordenadas de la figura coincide con el sistema de coordenadas de la escena, es decir, la posición de la figura se encuentra en el origen de coordenadas de la escena y que su orientación corresponde a los ejes del sistema de coordenadas de la escena.
- Translate(): añade una traslación a la matriz de transformación
  - Matemáticamente significa multiplicar la matriz location por la derecha por una matriz de transformación.
  - El resultado es que la figura se dibujará desplazada sobre el sistema de coordenadas de la escena.
- Rotate(): añade una rotación a la matriz de transformación.
  - Matemáticamente esta función multiplica la matriz location por la derecha por una matriz de rotación.
  - El resultado es que la figura se dibujará girada con respecto al sistema de coordenadas de la escena.

2. La clase CGFigure

- Draw(): se ha modificado para recibir como atributos la matriz view (que transforma las coordenadas de la escena en coordenadas del observador) y la matriz projection (que transforma las coordenadas de la escena en coordenadas Clip).
- La transformación a desarrollar en el VertexShader es la composición de las tres transformaciones (model-view-projection).

2. La clase CGFigure

- Contenido de CGFigure.cpp

```
#include "CGFigure.h"

#include <GL/glew.h>

#include <glm/glm.hpp>

#include
<glm/gtc/matrix_transform.hpp
>



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);

}
```

2. La clase CGFigure

```
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);

location = glm::mat4(1.0f);
}
```

2. La clase CGFigure

```
void CGFigure::ResetLocation()
{
  location = glm::mat4(1.0f);
}

void CGFigure::Translate(glm::vec3 t)
{
  location = glm::translate(location, t);
}

void CGFigure::Rotate(GLfloat angle, glm::vec3 axis)
{
  location = glm::rotate(location, glm::radians(angle), axis);
}

void CGFigure::Draw(CGShaderProgram* program, glm::mat4 projection, glm::mat4 view)
{
  glm::mat4 mvp = projection * view * location;
  program->SetUniformMatrix4("MVP", mvp);

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

3. La clase CGGround

- Las figuras que vamos a situar en la escena son las mismas que las definidas en la practica anterior (cubo, pirámide, cono, cilindro, esfera, disco, toro e icosaedro).
- Como los cambios planteados son comunes a todas las figuras y se han incorporado a la clase CGFigure, no es necesario plantear modificaciones en estas clases.
- Para dibujar la escena se va a añadir un suelo formado por una superficie cuadrada. Este suelo requiere una nueva clase, CGGround.

3. La clase CGGround

- Cabecera de CGGround.h

```
#pragma once

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

//
// CLASE: CGGround
//
// DESCRIPCIÓN: Representa un cuadrado dibujado en el suelo
//
class CGGround : public CGFigure {
public:
  CGGround(GLfloat l1, GLfloat l2);
};
```

3. La clase CGGround

- Contenido de CGGround.cpp

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

CGGround::CGGround(GLfloat l1, GLfloat l2)
{
 numFaces = 2; // Number of faces
 numVertices = 4; // Number of vertices

 GLfloat p_vertices[4][3] = {
  { l1, 0.0f, l2 },
  { l1, 0.0f, -l2 },
  { -l1, 0.0f, -l2 },
  { -l1, 0.0f, l2 }
 };

 GLushort p_indexes[2][3] = {
  { 0, 1, 2 },
  { 0, 2, 3 }
 };

 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();
}
```

4. El programa gráfico

- El programa gráfico que vamos a utilizar es el mismo que el planteado en la práctica anterior.
- El VertexShader incluye tan sólo una única matriz de transformación llamada MVP, que debe contener la composición de transformaciones model-view-projection.

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

4. El programa gráfico

- El FragmentShader sigue siendo el mismo y se limita a dibujar de rojo los píxeles de cada fragmento.

```
#version 400

out vec4 FragColor;

void main()

{

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

}
```

5. Descripción de la escena (clase CGScene)

- La escena que queremos mostrar está formada por seis figuras situadas sobre un plano.
- El plano es un cuadrado con un lado de tamaño 100 generado por una figura CGGround.
- El sistema de coordenadas del modelo tiene su origen en el centro del cuadrado, siendo los ejes X y Z los que marcan la superficie del cuadrado y el eje Y la vertical.
- Las figuras incluidas en la escena son un cubo, un cono, un cilindro, un toro, una esfera y un icosahedro.
- Estas figuras se van a colocar formando dos hileras a una distancia de 25 sobre el eje X respecto al centro de la escena.
- Las figuras se van a rotar y desplazar verticalmente para que aparezcan apoyadas sobre el suelo.

5. Descripción de la escena (clase CGScene)

- El fichero CGScene.h contiene la descripción de la clase CGScene que describe la escena completa.
- La interfaz pública de la clase contiene el constructor de la clase, su destructor, y el método Draw() que debe generar la escena.
- Internamente la clase contiene siete objetos CGFigure, que almacenarán las figuras geométricas y el suelo.

5. Descripción de la escena (clase CGScene)

- Cabecera CGScene.h

```
#pragma once

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

class CGScene {
public:
  CGScene();
  ~CGScene();
  void Draw(CGShaderProgram* program, glm::mat4 proj, glm::mat4 view);

private:
  CGFigure* ground;
  CGFigure* fig0;
  CGFigure* fig1;
  CGFigure* fig2;
  CGFigure* fig3;
  CGFigure* fig4;
  CGFigure* fig5;
};
```

5. Descripción de la escena (clase CGScene)

- El fichero CGScene.cpp contiene el desarrollo de la clase CGScene que describe la escena.
- El constructor de la clase es el encargado de crear los objetos CGFigure incluidos en la escena y situarlos mediante llamadas a los métodos Translate() y Rotate().
- El destructor de la clase se limita a destruir los objetos que contienen las figuras.
- El método Draw() es el encargado de generar la imagen y se limita a ejecutar los métodos Draw() de las figuras que la forman.

5. Descripción de la escena (clase CGScene)

- Contenido CGScene.cpp

```
#include "CGScene.h"

#include <GL/glew.h>

#include <glm/glm.hpp>

#include <glm/gtc/matrix_transform.hpp>

#include "CGShaderProgram.h"

#include "CGFigure.h"

#include "CGCube.h"

#include "CGCone.h"

#include "CGCylinder.h"

#include "CGSphere.h"

#include "CGTorus.h"

#include "CGIcosahedron.h"

#include "CGGround.h"


//

// FUNCIÓN: CGScene::CGScene()

//

// PROPÓSITO: Construye el objeto que representa la escena

//
```

5. Descripción de la escena (clase CGScene)

- Contenido CGScene.cpp

```
CGScene::CGScene()
{
  ground = new CGGround(50.0f, 50.0f);
  fig0 = new CGCone(5, 20, 5.0f, 5.0f);
  fig0->Translate(glm::vec3(25.0f, 5.0f, 25.0f));
  fig0->Rotate(-90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
  fig1 = new CGCube(5.0f);
  fig1->Translate(glm::vec3(-25.0f, 5.0f, 25.0f));
  fig1->Rotate(-90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
  fig2 = new CGCylinder(20, 20, 5.0f, 5.0f);
  fig2->Translate(glm::vec3(25.0f, 5.0f, 0.0f));
  fig2->Rotate(90.0f, glm::vec3(1.0f, 0.0f, 0.0f));
  fig3 = new CGTorus(20, 40, 3.0f, 5.0f);
  fig3->Translate(glm::vec3(-25.0f, 8.0f, 0.0f));
  fig4 = new CGSphere(20, 40, 8.0f);
  fig4->Translate(glm::vec3(25.0f, 8.0f, -25.0f));
  fig5 = new CGIcosahedron(5.0f);
  fig5->Translate(glm::vec3(-25.0f, 8.0f, -25.0f));
}
```

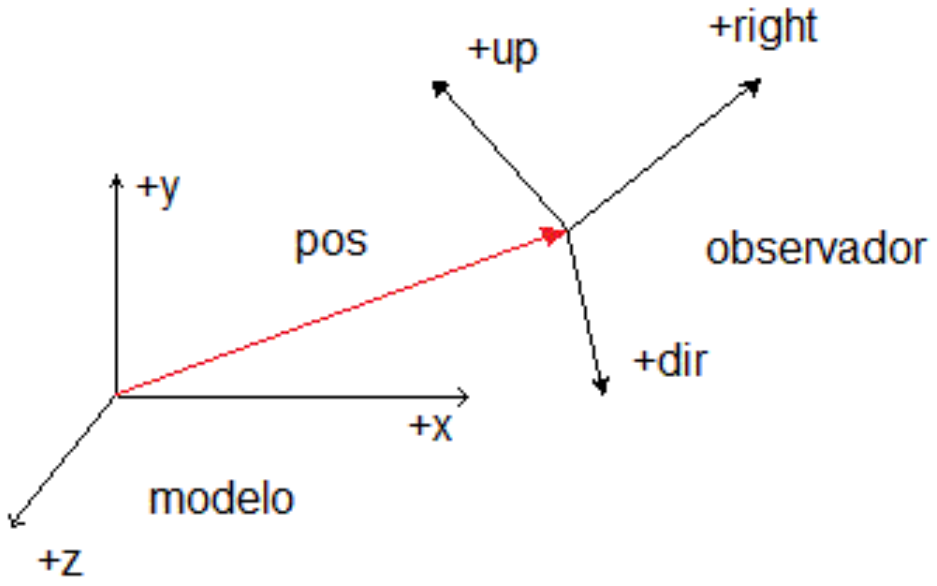
5. Descripción de la escena (clase CGScene)

- Contenido CGScene.cpp

```
CGScene::~CGScene()
{
  delete ground;
  delete fig0;
  delete fig1;
  delete fig2;
  delete fig3;
  delete fig4;
  delete fig5;
}

void
CGScene::Draw(CGShaderProgram* program, glm::mat4 proj, glm::mat4 view)
{
  ground->Draw(program, proj, view);
  fig0->Draw(program, proj, view);
  fig1->Draw(program, proj, view);
  fig2->Draw(program, proj, view);
  fig3->Draw(program, proj, view);
  fig4->Draw(program, proj, view);
  fig5->Draw(program, proj, view);
}
```

6. Descripción de la cámara

- Para poder desplazarse a través de la escena es necesario establecer la posición y orientación del observador y realizar una transformación del sistema de coordenadas entre el sistema de coordenadas del modelo y el sistema de coordenadas del observador.



6. Descripción de la cámara

- La posición del observador, en coordenadas del modelo, viene dada por el vector pos.
- Los ejes de coordenadas del observador, en coordenadas del modelo, vienen dados por los vectores right, up y dir.
- Para transformar las coordenadas del modelo en coordenadas del observador el primer paso es realizar una traslación de (-pos). De esta forma, el origen de coordenadas del modelo pasa a estar situado en la posición (-pos) y el origen de coordenadas del observador pasa a estar situado en la posición (0,0,0).

$$M_{TR} = \begin{pmatrix} 1 & 0 & 0 & -pos_x \\ 0 & 1 & 0 & -pos_y \\ 0 & 0 & 1 & -pos_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6. Descripción de la cámara

- Para completar la transformación del sistema de coordenadas es necesario rotar los ejes.
- El resultado de esta rotación es que el vector right debe convertirse en el vector unitario del eje X en el sistema de coordenadas del observador, es decir, en el vector (1,0,0)
  - El vector up debe transformarse en el vector unitario sobre el eje Y, es decir, en el vector (0,1,0)
  - Y el vector dir debe convertirse en el vector unitario sobre el eje Z, es decir, en el vector (0,0,1)
- Matriz de rotación resultante:

$$M_{ROT} = \begin{pmatrix} right_x & right_y & right_z & 0 \\ up_x & up_y & up_z & 0 \\ dir_x & dir_y & dir_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

6. Descripción de la cámara

- La matriz de transformación completa se obtiene multiplicando la matriz de rotación y la matriz de traslación.
- Esta matriz de transformación se puede generar fácilmente utilizando la función glm::lookAt(eye,center,up) incluida en la biblioteca GLM.
  - El primer argumento es la posición del observador.
  - El segundo argumento es la posición de un punto situado sobre el eje Dir.
  - El tercer argumento de esta función es el vector up.
- Para controlar las propiedades del observador vamos a definir una nueva clase que denominaremos CGCamera.

6. Descripción de la cámara

- Esta clase nos permitirá almacenar la posición y orientación del observador y moverlo libremente a través de la escena.
- La descripción de la clase se encuentra en el archivo CGCamera.h.
- Los miembros de la clase, descritos en su zona privada, almacenan los valores de los vectores pos, right, up y dir.
- También se incluyen los campos moveStep y turnStep que contienen la magnitud de los desplazamientos y giros a realizar en cada paso de la animación.
- Los campos cosAngle y sinAngle almacenan el seno y el coseno de turnStep para que no sea necesario recalcularlos constantemente.

6. Descripción de la cámara

- La interfaz pública de la clase CGCamera consta del constructor, setters y getters para todos los campos, métodos Move..() para realizar desplazamientos sobre los ejes de coordenadas, métods Turn..() para realizar rotaciones sobre los ejes de coordenadas y el método ViewMatrix() que devuleve la matriz de transformación entre el sistema de coordenadas del modelo y el sistema de coordenadas del observador.

```
#pragma once

#include <GL/glew.h>

#include <glm/glm.hpp>

class CGCamera {

public:

  CGCamera();

  glm::mat4 ViewMatrix();
```

6. Descripción de la cámara

```
void SetPosition(GLfloat x, GLfloat y, GLfloat z);

void SetDirection(GLfloat xDir, GLfloat yDir, GLfloat zDir,
                  GLfloat xUp, GLfloat yUp, GLfloat zUp);

void SetMoveStep(GLfloat step);

void SetTurnStep(GLfloat step);


glm::vec3 GetPosition();

glm::vec3 GetDirection();

glm::vec3 GetUpDirection();

GLfloat GetMoveStep();

GLfloat GetTurnStep();


void MoveFront();

void MoveBack();

void MoveRight();

void MoveLeft();

void MoveUp();

void MoveDown();

void TurnRight();

void TurnLeft();

void TurnUp();

void TurnDown();

void TurnCW();

void TurnCCW();
```

6. Descripción de la cámara

```
private:
  glm::vec3 Pos;
  glm::vec3 Dir;
  glm::vec3 Up;
  glm::vec3 Right;

  GLfloat moveStep;
  GLfloat turnStep;
  GLfloat cosAngle;
  GLfloat sinAngle;
};
```

6. Descripción de la cámara

- El desarrolla de la clase CGCamera se encuentra en CGCamera.cpp
- El constructor de la clase inicializa la posición y orientación del observador de manera que coincide con el sistema de coordenadas del modelo.
- El método ViewMatrix() desarrolla la transformación de los sistemas de coordenadas utilizando la función lookAt() de la biblioteca GLM.
- Los métodos Get…() se limitan a obtener los valores de los campos internos de la clase.
- Con respecto a los métodos Set…() es importante señalar que la orientación se fija en un único método que sólo asigna los valores de dir y up.
  - Se asume que estos vectores son unitarios y perpendiculares entre sí.
- El vector right se cacula como el producto vectorial entre up y Dir.

6. Descripción de la cámara

- Los métodos Move…() generan traslaciones de módulo moveStep sobre cada uno de los ejes del sistema de coordendas del observador.
  - Esto supone modificar el vector pos añadiendo un desplazamiento de moveStep por el vector unitario de la traslación.
  - Por ejemplo: MoveRight() genera un desplazamiento en la dirección del eje X (vector right) y MoveFront() genera un desplazamiento hacia adelante (dirección –Z, es decir, -dir).
- Los métodos Tutn…() realizan giros de magnitud turnStep sobre cada uno de los ejes del sistema de coordendas.
- Para evitar que por problemas de redondeo se pierda la ortogonalidad de los vectores right, up y dir, los giros se calculan sobre un único vector y el otro afectado se recalcula mediante un producto vectorial.

6. Descripción de la cámara

- Los métodos TurnLeft() y TurnRight() realizan giros respecto al eje Y.
- Los métodos TurnUp() y TurnDown() realizan giros respecto al eje X.
- Los métodos TurnCW y TurnCCW() realizan giros respecto al eje Z.

6. Descripción de la cámara

```
#include "CGCamera.h"
#include <GL/glew.h>
#include <math.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
CGCamera::CGCamera()
{
  Pos = glm::vec3(0.0f, 0.0f, 0.0f);
  Dir = glm::vec3(0.0f, 0.0f, 1.0f);
  Up = glm::vec3(0.0f, 1.0f, 0.0f);
  Right = glm::vec3(1.0f, 0.0f, 0.0f);

  moveStep = 0.1f;
  turnStep = 1.0f;
  cosAngle =
(GLfloat)cos(glm::radians(turnStep));
  sinAngle = (GLfloat)sin(glm::radians(turnStep));
}
```

6. Descripción de la cámara

```
glm::mat4 CGCamera::ViewMatrix()
{
  return glm::lookAt(Pos, Pos - Dir, Up);
}

void CGCamera::SetPosition(GLfloat x, GLfloat y, GLfloat z)
{
  Pos = glm::vec3(x, y, z);
}

void CGCamera::SetDirection(GLfloat xD, GLfloat yD, GLfloat zD,
                            GLfloat xU, GLfloat yU, GLfloat zU)
{
  Dir = glm::vec3(xD, yD, zD);
  Up = glm::vec3(xU, yU, zU);
  Right = glm::cross(Up, Dir);
}

void CGCamera::SetMoveStep(GLfloat step)
{
  moveStep = step;
}
```

6. Descripción de la cámara

```
void CGCamera::SetTurnStep(GLfloat step)
{
  turnStep = step;
  cosAngle = (GLfloat)cos(glm::radians(turnStep));
  sinAngle = (GLfloat)sin(glm::radians(turnStep));
}

glm::vec3 CGCamera::GetPosition()
{
  return Pos;
}
```

```
glm::vec3 CGCamera::GetDirection()
{
  return Dir;
}

glm::vec3 CGCamera::GetUpDirection()
{
  return Up;
}
```

6. Descripción de la cámara

```
GLfloat CGCamera::GetMoveStep()
{
  return moveStep;
}

GLfloat CGCamera::GetTurnStep()
{
  return turnStep;
}
```

```
void CGCamera::MoveFront()
{
  Pos -= moveStep * Dir;
}

void CGCamera::MoveBack()
{
  Pos += moveStep * Dir;
}
```

6. Descripción de la cámara

```
void CGCamera::MoveLeft()
{
  Pos -= moveStep * Right;
}

void CGCamera::MoveRight()
{
  Pos += moveStep * Right;
}
```

```
void CGCamera::MoveUp()
{
  Pos += moveStep * Up;
}

void CGCamera::MoveDown()
{
  Pos -= moveStep * Up;
}
```

6. Descripción de la cámara

```
void CGCamera::TurnRight()
{
  Dir.x = cosAngle * Dir.x - sinAngle * Right.x;
  Dir.y = cosAngle * Dir.y - sinAngle * Right.y;
  Dir.z = cosAngle * Dir.z - sinAngle * Right.z;

  // Right = Up x Dir
  Right = glm::cross(Up, Dir);
}
```

```
void CGCamera::TurnLeft()
{
  Dir.x = cosAngle * Dir.x + sinAngle * Right.x;
  Dir.y = cosAngle * Dir.y + sinAngle * Right.y;
  Dir.z = cosAngle * Dir.z + sinAngle * Right.z;

  // Right = Up x Dir
  Right = glm::cross(Up, Dir);
}
```

```
void CGCamera::TurnUp()
{
  Dir.x = cosAngle * Dir.x - sinAngle * Up.x;
  Dir.y = cosAngle * Dir.y - sinAngle * Up.y;
  Dir.z = cosAngle * Dir.z - sinAngle * Up.z;

  // Up = Dir x Right
  Up = glm::cross(Dir, Right);
}
```

6. Descripción de la cámara

```
void CGCamera::TurnDown()
{
  Dir.x = cosAngle * Dir.x + sinAngle * Up.x;
  Dir.y = cosAngle * Dir.y + sinAngle * Up.y;
  Dir.z = cosAngle * Dir.z + sinAngle * Up.z;

  // Up = Dir x Right
  Up = glm::cross(Dir, Right);
}
```

```
void CGCamera::TurnCW()
{
  Up.x = cosAngle * Up.x + sinAngle * Right.x;
  Up.y = cosAngle * Up.y + sinAngle * Right.y;
  Up.z = cosAngle * Up.z + sinAngle * Right.z;

  // Right = Up x Dir
  Right = glm::cross(Up, Dir);
}
```

```
void CGCamera::TurnCCW()
{
  Up.x = cosAngle * Up.x - sinAngle * Right.x;
  Up.y = cosAngle * Up.y - sinAngle * Right.y;
  Up.z = cosAngle * Up.z - sinAngle * Right.z;

  // Right = Up x Dir
  Right = glm::cross(Up, Dir);
}
```

7. Descripción del modelo 3D (clase CGModel)

- El modelo a representar en nuestra aplicación se desarrolla por medio de la clase CGModel.
- El modelo está formado simplemente por el programa gráfico, la cámara y la escena.
- También se almacena como campo la matriz de proyección.
- Y la interfaz pública de la clase es la que se utiliza en todas las prácticas.

7. Descripción del modelo 3D (clase CGModel)

```
#pragma once

#include <GL/glew.h>
#include "CGShaderProgram.h"
#include "CGScene.h"
#include "CGCamera.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;
  CGScene* scene;
  CGCamera* camera;

  glm::mat4 projection;
};
```

7. Descripción del modelo 3D (clase CGModel)

- El método initialize() de la clase CGModel es el encargado de crear los objetos CGShaderProgram, CGCamera y CGScene.
  - Inicialmente la cámara se coloca en una posición que permite apreciar la escena.
- El método finalize() se encarga de destruir los objetos program, camera y scene.
- El método render(), dedicado a generar la imagen, se limita a obtener la matriz de transformación de la cámara y dibujar la escena.
- La acción de actualización consiste en mover la cámara hacia el frente.
  - De esta manera, si el valor de moveStep en la cámara es positivo la cámara se mueve hacia delante de forma automática.

7. Descripción del modelo 3D (clase CGModel)

- El método key_pressed() asocia movimientos de la cámara a algunas teclas, de manera que el usuario pueda desplazar libremente la cámara a través de la escena.
  - Las teclas Q y A mueven la cámara arriba y abajo en un único paso.
  - Las teclas O y P mueven la cámara a la izquierda y derecha en un único paso.
  - Las teclas '+' y '-' permiten acelerar y frenar la cámara.
  - Para detener la cámara se utiliza la tecla S.
  - Los giros de la cámara se realizan con las flechas y las teclas K y L, dedicadas a girar a la derecha y a la izquierda.

7. Descripción del modelo 3D (clase CGModel)

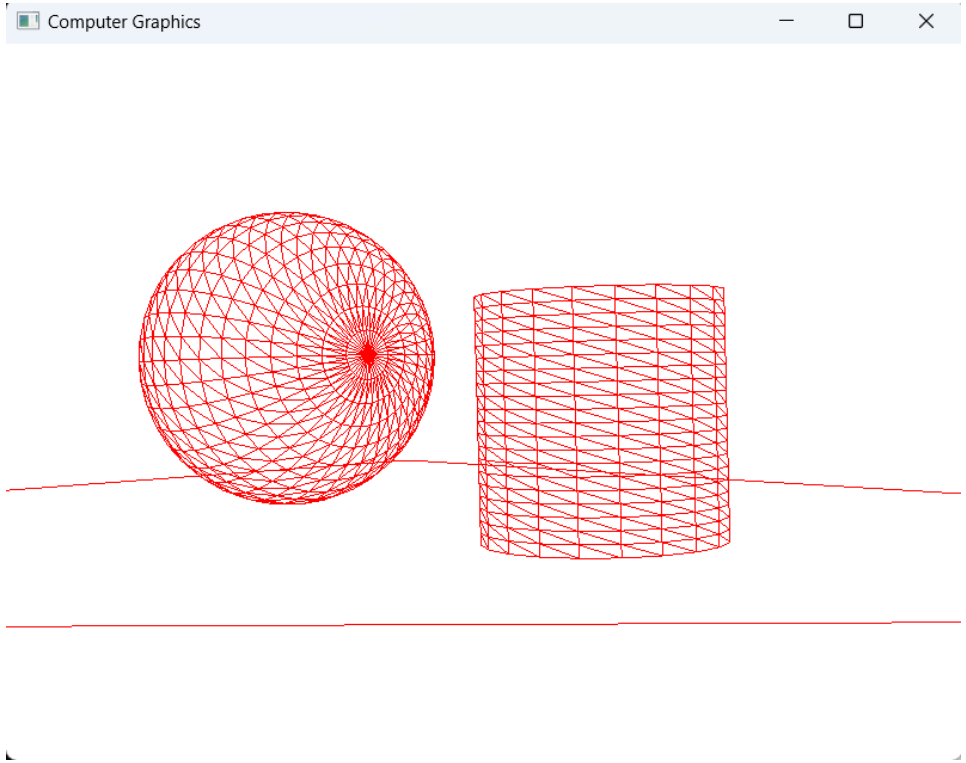
```
#include "CGModel.h"
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <GLFW/glfw3.h>
#include <iostream>
#include "CGCamera.h"
#include "CGScene.h"


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();


  // Crea la cámara
  camera = new CGCamera();
  camera->SetPosition(0.0f, 5.0f, 30.0f);


  // Crea la escena
  scene = new CGScene();


  // 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);
}
```

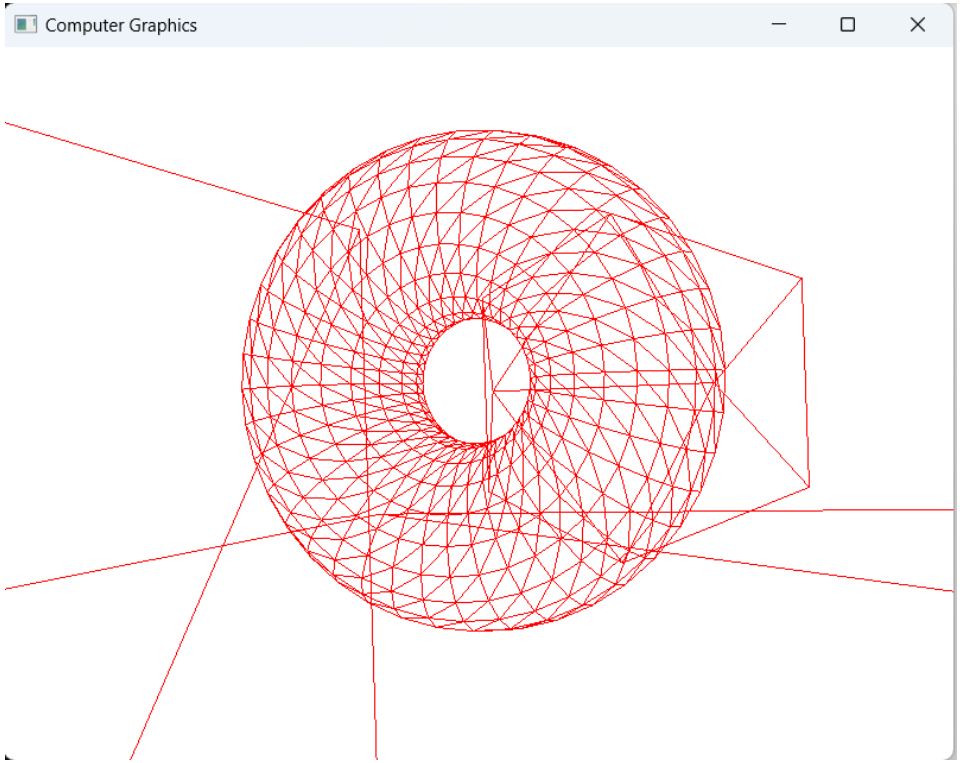
7. Descripción del modelo 3D (clase CGModel)

```
void CGModel::finalize()
{
  delete camera;
  delete scene;
  delete program;
}

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);
}

void CGModel::render()
{
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glm::mat4 view = camera->ViewMatrix();
  scene->Draw(program, projection, view);
}
```

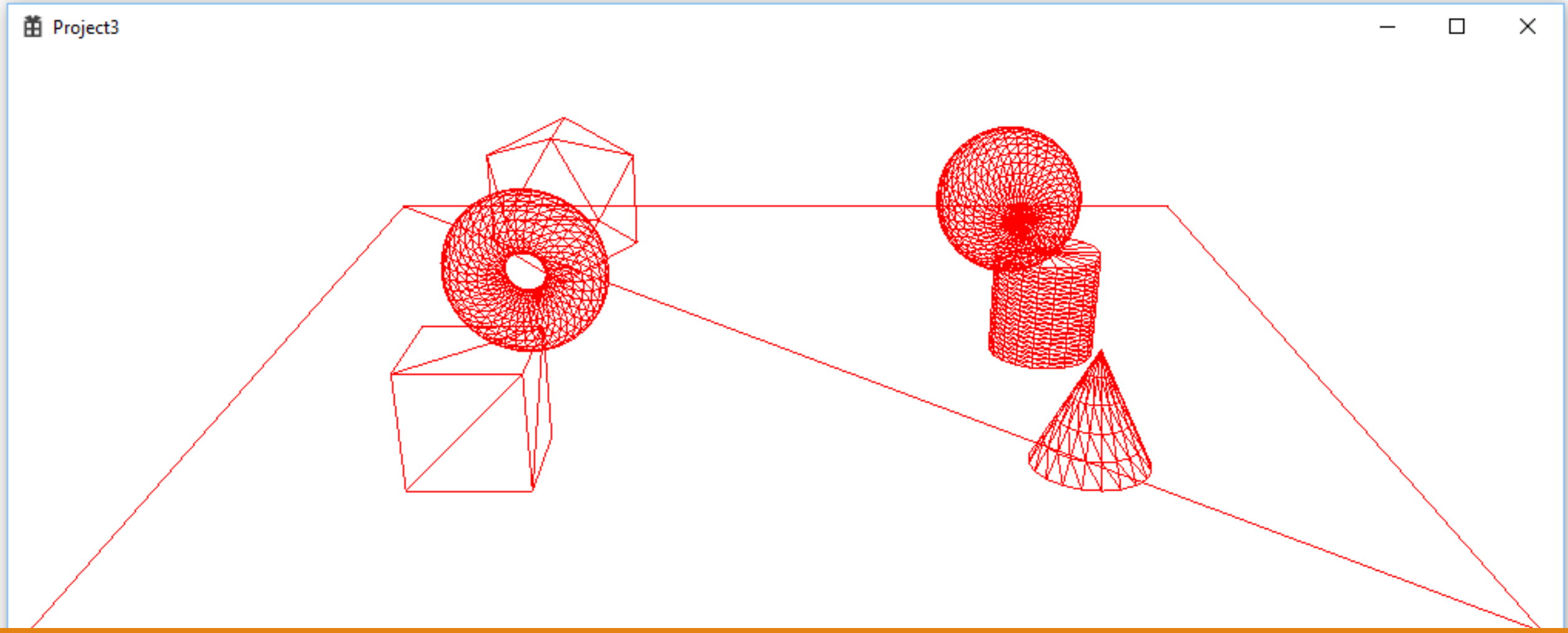
7. Descripción del modelo 3D (clase CGModel)

```
void CGModel::update()
{
  camera->MoveFront();
}
void CGModel::key_pressed(int key)
{
  switch (key)
  {
  case GLFW_KEY_UP:
    camera->TurnDown();
    break;
  case GLFW_KEY_DOWN:
    camera->TurnUp();
    break;
  case GLFW_KEY_LEFT:
    camera->TurnCCW();
    break;
  case GLFW_KEY_RIGHT:
    camera->TurnCW();
    break;
  case GLFW_KEY_S:
    camera->SetMoveStep(0.0f);
    break;
  case GLFW_KEY_RIGHT_BRACKET:
  case GLFW_KEY_KP_ADD:
    camera->SetMoveStep(camera->GetMoveStep() + 0.1f);
    break;
  case GLFW_KEY_MINUS:
  case GLFW_KEY_KP_SUBTRACT:
    camera->SetMoveStep(camera->GetMoveStep() - 0.1f);
    break;
  case GLFW_KEY_Q:
    camera->SetMoveStep(0.1f);
    camera->MoveUp();
    camera->SetMoveStep(0.0f);
    break;
  case GLFW_KEY_A:
    camera->SetMoveStep(0.1f);
    camera->MoveDown();
    camera->SetMoveStep(0.0f);
    break;
  case GLFW_KEY_O:
    camera->SetMoveStep(0.1f);
    camera->MoveLeft();
    camera->SetMoveStep(0.0f);
    break;
  case GLFW_KEY_P:
    camera->SetMoveStep(0.1f);
    camera->MoveRight();
    camera->SetMoveStep(0.0f);
    break;
  case GLFW_KEY_K:
    camera->TurnLeft();
    break;
  case GLFW_KEY_L:
    camera->TurnRight();
    break;
  }
}
```





8. Resultado final



8. Resultado final