Realidad Virtual. Práctica 5.

COLOR E ILUMINACIÓN

Índice

1. Objetivos

- Se presenta un modelo de iluminación que permite añadirle color a los objetos situados en la escena.
- El modelo planteado se conoce como modelo de Phong o modelo ADS, se calcula el color de cada punto como la suma de tres componentes:
  - Ambiental
  - Difusa
  - Especular
- El resultado final es similar al presentado en la práctica anterior, pero dotando de color a los cuerpos representados.
- Código de la práctica: https://www.uhu.es/francisco.moreno/gii_rv/practicas/practica05/practica05.rar

2. El modelo de iluminación de Phong

- OpenGL utiliza el formato RGBA para representar el color.
  - Describir un color por medio de 4 componentes.
  - Cada componente se representa por medio de un valor float normalizado.
  - El color negro se representa por el vector (0.0, 0.0, 0.0, 1.0)
  - El color blanco se representa por el vector (1.0, 1.0, 1.0, 1.0)
- En las prácticas anteriores el FragmentShader se limitaba a asignar el color rojo a todos los píxeles del fragmento.
  - Para generar figuras de colores diferentes una opción sencilla es introducir el color del objeto por medio de una variable uniforme. El problema es que esto dibuja las figuras con un color plano, siendo poco realista.
  - Para generar un coloreado natural es necesario estudiar la forma en la que la luz ilumina a las diferentes figuras. Es necesario planear un modelo de iluminación.

2. El modelo de iluminación de Phong

- El modelo de iluminación de Phong, propuesto por Bui Tuong Phong en su tesis doctoral, presentada en 1973 en la Universidad de Utah.
  - Este modelo constituye una mejora del modelo de Gouraud, que fue propuesto en 1971.
  - Estos modelos se basan en calcular el color con el que se ve un objeto como el resultado de la interacción de la luz (de un cierto color) con el material que forma el objeto (descrito también por un color)
  - Si los colores se expresan en coordenadas normalizadas. El resultado de iluminar un cuerpo de un color M con una luz de un color L es

$$\text{Color} = M \cdot L = (Mr, Mg, Mb, Ma) \cdot (Lr, Lg, Lb, La) = (Mr \cdot Lr, Mg \cdot Lg, Mb \cdot Lb, Ma \cdot La)$$

- Esta ecuación permite modificar el color con el que se ve un objeto al modificar el color de la luz.
  - Sin embargo, sigue generando el mismo color para todos los puntos del objeto (un color plano) lo que resulta antinatural.

2. El modelo de iluminación de Phong

- Para conseguir un colereado más natural se considera que el color final es la combinación de tres efectos:
  - Luz ambiental
  - Luz difusa
  - Luz especular
- La luz ambiental se considera como una luz que proviene de todas las direcciones y tras incidir en un punto se dispersa en todas las direcciones.
  - Este es el tipo de luz que ilumina la zona de sombra de un objeto (esta zona nunca es totalmente negra) y su fórmula corresponde a la descrita anteriormente.

2. El modelo de iluminación de Phong

- La luz difusa se considera como una luz que proviene de una dirección concreta y tras incidir en un punto se dispersa en todas las direcciones.
  - El efecto de esta luz tiene en cuenta la dirección de incidencia, de manera que si incide verticalmente sobre el objeto la iluminación es más intensa.
  - Mientras que si incide de forma inclinada la intensidad es menor.
  - El factor de incidencia se calcula como el coseno del ángulo que forma la dirección de la luz (Ldir) con el vector normal a la superficie (N). Si ambos vectores son unitarios el coseno corresponde al producto escalar. El efecto se calcula como:

$$\textbf{Color} = \textbf{M} \cdot \textbf{L} \cdot \text{Intensidad} = (\textbf{M} \cdot \textbf{L}) * (\textbf{Ldir} \cdot \textbf{N})$$

- La luz especular se considera como una luz que proviene de una dirección concreta y tras incidir en un punto se refleja, dispersándose principalmente en la dirección reflejada.
  - El efecto de esta luz tiene en cuenta la dirección de incidencia y la normal (para calcular la dirección reflejada) y la posición del observador.

2. El modelo de iluminación de Phong

- Si la dirección reflejada coincide con la dirección en la que se encuentra el observador la intensidad de este efecto será máxima
- Mientras que si la dirección reflejada se separa de la dirección del observador la intensidad disminuye.
- El factor de intensidad especular se calcula como el coseno del ángulo entre la dirección reflejada (Reflect) y la dirección del observador (Pos), elevado a un cierto coeficiente de brillo (Shininess)

$$\mathbf{Color} = \mathbf{M} \cdot \mathbf{L} \cdot \text{Intensidad} = (\mathbf{M} \cdot \mathbf{L}) * (\mathbf{Reflect} \cdot \mathbf{Pos})^{\textit{Shininess}}$$

$$\mathbf{Reflect} = 2 \cdot (\mathbf{Ldir} \cdot \mathbf{N}) \cdot \mathbf{N} - \mathbf{Ldir}$$

- El color final en el que se dibuja cada punto es la suma de los tres efectos:

$$\mathbf{Color} = \mathbf{Color\_ambiental} + \mathbf{Color\_difuso} + \mathbf{Color\_especular}$$

3. El programa gráfico

- Para desarrollar el modelo de iluminación de Phong es necesario calcular para cada píxel el valor de la normal a la superficie y la posición del punto asociado a ese píxel.
  - Estos vectores deben escribirse en coordenadas del observador.
  - Para obtener estos vectores en cada píxel, el VertexShader debe generarlos como salidas asociadas a cada vértice de forma que el proceso de interpolación genere los valores correspondientes a cada punto del fragmento.
  - Por tanto, los atributos asociados a cada vértice deben ser su posición y su normal.
  - La salida gl_Position contiene la posición en coordenadas homogéneas (multiplicando por la matriz MVP como vimos en la práctica anterior).
  - Las salidas Position y Normal contienen los vectores de posición y normal en coordenadas del observador (multiplicando en este caso por la matriz model-view).

3. El programa gráfico

- VertexShader

```
#version 400

layout(location = 0) in vec3 VertexPosition;
layout(location = 1) in vec3 VertexNormal;

uniform mat4 MVP;
uniform mat4 ViewMatrix;
uniform mat4 ModelViewMatrix;

out vec3 Position;
out vec3 Normal;

void main()
{
  vec4 n4 = ModelViewMatrix*vec4(VertexNormal, 0.0);
  vec4 v4 = ModelViewMatrix*vec4(VertexPosition, 1.0);
  Normal = vec3(n4);
  Position = vec3(v4);
  gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

3. El programa gráfico

- El FragmentShader debe desarrollar los cálculos del modelo de iluminación de Phong a partir de la interpolación de la posición y la normal (en coordenadas del observador) y de las propiedades de la luz y del material.
- Las propiedades de la luz se introducen por medio de una variable uniforme (Light) que responde a una estructura con cuatro campos:
  - Ldir es la dirección de la luz
  - La el color de la luz ambiental
  - Ld es el color de la luz difusa
  - Ls es el color de la luz especular

3. El programa gráfico

- Las propiedades del material se introducen por medio de una variable uniforme (Material) que responde a una estructura con cuatro campos:
  - Ka es el color del material frente a la luz ambiental
  - Kd es el color frente a la luz difusa
  - Ks es el color frente a la luz especualr
  - Shininess es el coeficiente de brillo
- La función auxiliar ads() desarrolla el modelo de Phong calculando los tres efectos:
  - Ambiental
  - Difusa
  - Especular

3. El programa gráfico

- FragmentShader

```
#version 400

in vec3 Position;
in vec3 Normal;

uniform mat4 ViewMatrix;

struct LightInfo
{
  vec3 Ldir;
  vec3 La;
  vec3 Ld;
  vec3 Ls;
};
uniform LightInfo Light;

struct MaterialInfo
{
  vec3 Ka;
  vec3 Kd;
  vec3 Ks;
  float Shininess;
};
uniform MaterialInfo Material;

out vec4 FragColor;

vec3 ads()
{
  vec4 s4 = ViewMatrix*vec4(Light.Ldir, 0.0);
  vec3 n = normalize(Normal);
  vec3 v = normalize(-Position);
  vec3 s = normalize(-vec3(s4));
  vec3 r = reflect(-s, n);
  float dRate = max(dot(s, n), 0.0);
  float sRate = pow(max(dot(r, v), 0.0), Material.Shininess);
  vec3 ambient = Light.La * Material.Ka;
  vec3 difusse = Light.Ld * Material.Kd * dRate;
  vec3 specular = Light.Ls * Material.Ks * sRate;
  return ambient + difusse + specular;
}

void main()
{
  vec3 Color = ads();
  FragColor = vec4(Color,1.0);
}
```

4. La clase CGLight

- Para describir las propiedades de la luz se va a crear una nueva clase denominada CGLight.
- La clase cuenta con los campos utilizados en el modelo Phong.
- El campo Ldir almacena la dirección de la luz en coordenadas del modelo.
- Los campos La, Ld y Ls contienen el color de las componentes ambiental, difusa y especular.

4. La clase CGLight

- Cabecera CGLight.h

```
#pragma once

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

class CGLight {

private:
  glm::vec3 Ldir; // Light direction
  glm::vec3 La; // Ambient intensity
  glm::vec3 Ld; // Difusse intensity
  glm::vec3 Ls; // Specular intensity

public:
  CGLight();
  void SetLightDirection(glm::vec3 d);
  void SetAmbientLight(glm::vec3 a);
  void SetDifusseLight(glm::vec3 d);
  void SetSpecularLight(glm::vec3 s);
  void SetUniforms(CGShaderProgram* program);
};
```

4. La clase CGLight

- La clase consta de métodos Set…() para asignar el valor de cada propiedad y del método SetUniforms() que asigna los valores a la variable uniforme utilizada en el programa gráfico.

4. La clase CGLight

- Contenido CGLight.cpp

```
#include "CGLight.h"
#include <glm/glm.hpp>

CGLight::CGLight()
{
  Ldir = glm::vec3(0.0f, 0.0f, -1.0f);
  La = glm::vec3(1.0f, 1.0f, 1.0f);
  Ld = glm::vec3(1.0f, 1.0f, 1.0f);
  Ls = glm::vec3(0.0f, 0.0f, 0.0f);
}
```

```
void CGLight::SetLightDirection(glm::vec3 d)
{
  Ldir = d;
}

void CGLight::SetAmbientLight(glm::vec3 a)
{
  La = a;
}
```

4. La clase CGLight

- Contenido CGLight.cpp

```
void CGLight::SetDifusseLight(glm::vec3 d)
{
  Ld = d;
}

void CGLight::SetSpecularLight(glm::vec3 s)
{
  Ls = s;
}
```

```
void CGLight::SetUniforms(CGShaderProgram* program)
{
  program->SetUniformVec3("Light.Ldir", Ldir);
  program->SetUniformVec3("Light.La", La);
  program->SetUniformVec3("Light.Ld", Ld);
  program->SetUniformVec3("Light.Ls", Ls);
}
```

5. La clase CGMaterial

- Para describir las propiedades de los materiales se va a crear una nueva clase denominada CGMaterial, que almacena las propiedades utilizadas en el modelo de Phong.
- Los campos Ka, Kd y Ks contienen el color del material ante las funetes de luz ambiental, difusa y especular, lo que se conoce como reflectividad.
- El campo Shininess contiene el factor de brillo, es decir, el exponente utilizado en el cálculo de la componente especular.

5. La clase CGMaterial

- Cabecera CGMaterial.h

```
#pragma once


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


class CGMaterial {


private:
  glm::vec3 Ka;       // Reflectividad ambiental (color ante la luz ambiental)
  glm::vec3 Kd;       // Reflectividad difusa    (color ante la luz difusa)
  glm::vec3 Ks;       // Reflectividad especular (color ante la luz especular)
  GLfloat Shininess;   // Factor de brillo (comportamiento ante la luz especular)


public:
  CGMaterial();
  void SetAmbientReflect(GLfloat r, GLfloat g, GLfloat b);
  void SetDifusseReflect(GLfloat r, GLfloat g, GLfloat b);
  void SetSpecularReflect(GLfloat r, GLfloat g, GLfloat b);
  void SetShininess(GLfloat f);
  void SetUniforms(CGShaderProgram* program);
};
```

5. La clase CGMaterial

- Al igual que la clase CGLight, la clase CGMaterial consta de métodos Set…() para asignar el valor de cada propiedad y del método SetUniforms() que asigna los valores a la variable uniforme utilizada en el programa gráfico.

5. La clase CGMaterial

- Contenido CGMaterial.cpp

```
#include "CGMaterial.h"
#include <GL/glew.h>


CGMaterial::CGMaterial()
{
  Ka = glm::vec3(1.0f, 1.0f, 1.0f);
  Kd = glm::vec3(1.0f, 1.0f, 1.0f);
  Ks = glm::vec3(0.8f, 0.8f, 0.8f);
  Shininess = 16.0f;
}
```

```
void CGMaterial::SetAmbientReflect(GLfloat r, GLfloat g, GLfloat b)
{
  Ka = glm::vec3(r, g, b);
}


void CGMaterial::SetDifusseReflect(GLfloat r, GLfloat g, GLfloat b)
{
  Kd = glm::vec3(r, g, b);
}
```

5. La clase CGMaterial

- Contenido CGMaterial.cpp

```
void CGMaterial::SetSpecularReflect(GLfloat r, GLfloat g, GLfloat b)
{
  Ks = glm::vec3(r, g, b);
}

void CGMaterial::SetShininess(GLfloat f)
{
  Shininess = f;
}
```

```
void CGMaterial::SetUniforms(CGShaderProgram* program)
{
  program->SetUniformVec3("Material.Ka", Ka);
  program->SetUniformVec3("Material.Kd", Kd);
  program->SetUniformVec3("Material.Ks", Ks);
  program->SetUniformF("Material.Shininess", Shininess);
}
```

6. La clase CGFigure

- Para considerar los efectos dde iluminación es necesario añadir a las figuras la información respecto a las propiedades del material.
- También es necesario que las figuras almacenen el valor del vector normal (perpendicular) a la superficie de cada vértice, además de la posición de cada vértice.
- Esto supone realizar cambios sobre la clase CGFigure.

6. La clase CGFigure

- Cabecera CGFigure.h

```
#pragma once

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

#define VERTEX_DATA 0
#define INDEX_DATA 1
#define NORMAL_DATA 2
//
// 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
  GLfloat* normals; // Array of normals

  GLuint numFaces; // Number of faces
  GLuint numVertices; // Number of vertices
  GLuint VBO[3];
  GLuint VAO;

  glm::mat4 location; // Model matrix
  CGMaterial* material;
public:
  ~CGFigure();
  void InitBuffers();
  void SetMaterial(CGMaterial* mat);
  void ResetLocation();
  void Translate(glm::vec3 t);
  void Rotate(GLfloat angle, glm::vec3 axis);
  void Draw(CGShaderProgram* program, glm::mat4 projection, glm::mat4 view);
};
```

6. La clase CGFigure

- Los cambios en la clase CGFigure consisten en considerar un segundo atributo en el VAO así como las propiedades de material.
- Esto requiere modificar un poco el método InitBuffers() para incorporar el segundo atributo asociado a los vectores normales.
- Para tratar la información respecto al material se añade el método SetMaterial() y se modifica el método Draw() para asignar los valores a la variable uniforme correspondiente.
- También hay que tener en cuenta el programa gráfico, este necesita ahora no solo la matriz MVP, sino también las matrices ModelViewMatriz y ViewMatrix como variables uniformes.

6. La clase CGFigure

- Contenido 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;
  if (normals != NULL) delete[] normals;

  // Delete vertex buffer objects
  glDeleteBuffers(3, VBO);
  glDeleteVertexArrays(1, &VAO);
}
```

6. La clase CGFigure

- Contenido CGFigure.cpp

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

  // Create the Vertex Buffer Objects
  glGenBuffers(3, VBO);

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

  // Normal data
  glBindBuffer(GL_ARRAY_BUFFER, VBO[NORMAL_DATA]);
  glBufferData(GL_ARRAY_BUFFER, buffsize, normals, GL_STATIC_DRAW);

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

  delete[] vertices;
  delete[] indexes;
  delete[] normals;

  vertices = NULL;
  indexes = NULL;
  normals = NULL;

  glEnableVertexAttribArray(0); // Vertex position
  glBindBuffer(GL_ARRAY_BUFFER, VBO[VERTEX_DATA]);
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
  glEnableVertexAttribArray(1); // Vertex normals
  glBindBuffer(GL_ARRAY_BUFFER, VBO[NORMAL_DATA]);
  glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, 0);
  location = glm::mat4(1.0f);
}
```

6. La clase CGFigure

- Contenido CGFigure.cpp

```
void CGFigure::SetMaterial(CGMaterial* mat)
{
  material = mat;
}

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);
}
```

6. La clase CGFigure

- Contenido CGFigure.cpp

```
void CGFigure::Draw(CGShaderProgram* program, glm::mat4 projection, glm::mat4 view)
{
  glm::mat4 mvp = projection * view * location;
  program->SetUniformMatrix4("MVP", mvp);
  program->SetUniformMatrix4("ViewMatrix", view);
  program->SetUniformMatrix4("ModelViewMatrix", view * location);
  material->SetUniforms(program);
  glBindVertexArray(VAO);
  glDrawElements(GL_TRIANGLES, numFaces * 3, GL_UNSIGNED_SHORT, NULL);
}
```

7. La clase CGScene

- La escena a representar en el modelo se describe en la clase CGScene.
- Respecto a la versión de la práctica anterior, la única diferencia en la cabecera de la función es que se añade un campo light que almacena la información de la fuente de luz y los campos matg y material que almacenan los materiales del suelo y de las figuras.

7. La clase CGScene

- Cabecera CGScene.h

```
#pragma once


#include <GL/glew.h>
#include <glm/glm.hpp>
#include "CGShaderProgram.h"
#include "CGLight.h"
#include "CGMaterial.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;
  CGLight* light;
  CGMaterial* matg;
  CGMaterial* material;
};
```

7. La clase CGScene

- Con respecto al código, el constructor de la clase, además de generar las figuras y posicionarlas, debe inicializar los valores de la fuente de luz y de las propiedades de material de cada figura.
- El método Draw() debe asignar las variables uniformes asociadas a la luz además de realizar llamadas para dibujar los objetos que forman parte de la escena.

7. La 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 "CGLight.h"
#include "CGMaterial.h"
#include "CGCube.h"
#include "CGCone.h"
#include "CGCylinder.h"
#include "CGSphere.h"
#include "CGTorus.h"
#include "CGIcosahedron.h"
#include "CGGround.h"
```

7. La clase CGScene

- Contenido CGScene.cpp

```
CGScene::CGScene()
{
  glm::vec3 Ldir = glm::vec3(1.0f, -0.8f, -1.0f);
  Ldir = glm::normalize(Ldir);
  light = new CGLight();
  light->SetLightDirection(Ldir);
  light->SetAmbientLight(glm::vec3(0.2f, 0.2f, 0.2f));
  light->SetDifusseLight(glm::vec3(0.8f, 0.8f, 0.8f));
  light->SetSpecularLight(glm::vec3(1.0f, 1.0f, 1.0f));
  matg = new CGMaterial();
  matg->SetAmbientReflect(0.0f, 0.6f, 0.0f);
  matg->SetDifusseReflect(0.0f, 0.6f, 0.0f);
  matg->SetSpecularReflect(0.8f, 0.8f, 0.8f);
  matg->SetShininess(16.0f);
  material = new CGMaterial();
  material->SetAmbientReflect(1.0f, 0.0f, 0.0f);
  material->SetDifusseReflect(1.0f, 0.0f, 0.0f);
  material->SetSpecularReflect(0.8f, 0.8f, 0.8f);
  material->SetShininess(16.0f);
  ground = new CGGround(50.0f, 50.0f);
  ground->SetMaterial(matg);
  fig0 = new CGCone(5, 20, 5.0f, 5.0f);
  fig0->SetMaterial(material);
  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->SetMaterial(material);
  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->SetMaterial(material);
  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->SetMaterial(material);
  fig3->Translate(glm::vec3(-25.0f, 8.0f, 0.0f));
  fig4 = new CGSphere(20, 40, 8.0f);
  fig4->SetMaterial(material);
  fig4->Translate(glm::vec3(25.0f, 8.0f, -25.0f));
  fig5 = new CGIcosahedron(5.0f);
  fig5->SetMaterial(material);
  fig5->Translate(glm::vec3(-25.0f, 8.0f, -25.0f));
}
```

7. La clase CGScene

- Contenido CGScene.cpp

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

void
CGScene::Draw(CGShaderProgram* program, glm::mat4 proj, glm::mat4 view)
{
  light->SetUniforms(program);
  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);
}
```

8. Figuras geométricas

- Con respecto a las diferentes figuras geométricas, es necesario modificar su código para incorporar a los atributos de los vértices el valor del vector normal en cada vértice.
- En CGModel.cpp cambiar:

```
//glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```