Realidad Virtual. Práctica 2.

PROGRAMAS GRÁFICOS

Índice

1. Objetivos

- Desarrollar una aplicación gráfica sencilla formada por un único triángulo en movimiento.
- Para ello, definir el contenido de las etapas programables del proceso de renderizado (vertex shader y fragment shader), generar los atributos de los vértices del triángulo y desarrollar la interfaz del modelo para responder a los distintos eventos recibidos en la ventana de la aplicación.
- Código de la práctica: https://www.uhu.es/francisco.moreno/gii_rv/practicas/practica02/practica02.rar

2. Un modelo 3D sencillo

- El código descrito en la práctica anterior (Práctica 1), permite generar una plantilla para generar aplicaciones gráficas con OpenGL.
- Para completar las aplicaciones debemos definir la clase CGModel, para dar contenido a las funciones que hemos utilizado en la plantilla.
- Por lo tanto, de la práctica 1, conservamos el main.cpp y CGApplication.h. Y vamos a implementar un nuevo GCModel.

2. Un modelo 3D sencillo

- Primeramente creamos el fichero de cabecera de GCModel.h, que contiene la definición de la clase.
  - Además de predefinir los métodos utilizados en la plantilla, la clase incluye una serie de campos que almacenarán la información necesaria para desarrollar el modelo.
  - posX y posY: contendrán la posición del centro del triángulo.
  - size: indicará el tamaño del triángulo
  - stepX y stepY: almacenarán la velocidad de movimiento del triángulo sobre los ejes X e Y.
- Para dibujar el triángulo se va a crear una clase específica CGTriangle que será descrita después.
- Para desarrollar el proceso de renderizado se utilizará la clase CGShaderProgram que describe un programa gráfico basado en shaders.

2. Un modelo 3D sencillo

- Plantilla CGModel.h

```
#pragma once

#include <GL/glew.h>
#include "CGShaderProgram.h"
#include "CGTriangle.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:
  GLfloat posX;
  GLfloat posY;
  GLfloat size;
  GLfloat stepX;
  GLfloat stepY;

  CGTriangle* triangle;
  CGShaderProgram* program;
};
```

2. Un modelo 3D sencillo

- initialize(): Como argumentos tiene el tamaño de la ventana principal de la aplicación, expresados en píxeles. Este método crea y compila el programa gráfico, inicializa los campos de la clase y configura las opciones generales de OpenGL.
  - En este punto se suelen cargar también las texturas que pueda necesitar el modelo, aunque en esta práctica aún no se ven.
  - Para crear el programa gráfico se ha utilizado en constructor basado en recursos, pero hay una función comentada basada en el nombre de los ficheros fuente de los shaders.
- finalize(): método que libera los recursos reservados por el modelo. Para este caso, se elimina el objeto que describe el triángulo y el programa gráfico.
- resize(): se ejecuta cuando se producen cambios en el tamaño de la ventana. Modifica el viewport, aunque normalmente también se modifica la matriz de proyección entre el sistema de coordenadas del modelo y el sistema de coordenadas *clip*.

2. Un modelo 3D sencillo

- render(): encargado de dibujar el modelo. Primero limpia la imagen llamando a glClear(). El color de fondo con el que se limpia se establece con glClearColor() al inicializar el modelo. A continuación, dibuja el triángulo llamando al método Draw() de ese objeto.
- update(): se ejecuta para actualizar el estado del modelo siguiendo la frecuencia de refresco utilizada en la aplicación (60 fps). En este caso, para generar la animación se incrementan los calores de posX y posY con los valores de velocidad stepX y stepY. Cuando la posición del triángulo alcanza los límites del Clip, se cambia el signo de la velocidad generando el efecto de rebote.
- key_pressed(): respuesta a eventos del teclado.
  - UP: aumenta el tamaño del triángulo
  - DOWN: disminuye el tamaño del triángulo
  - R: reinicializa la posición y el tamaño del triángulo.

2. Un modelo 3D sencillo

- Los métodos mouse_button() y mouse_move() son la respuesta a los eventos del ratón. En esta aplicación no se van a desarrollar.
- Código CGModel.cpp

```
#include "CGModel.h"

#include <GLFW\glfw3.h>

#include <iostream>
```

2. Un modelo 3D sencillo

```
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_TRUE) program->Use();
  else std::cout << program->GetLog() << std::endl;
  // Crea la escena
  triangle = new CGTriangle();
  posX = 0.0f;
  posY = 0.0f;
  size = 5.0f;
  stepX = 0.02f;
  stepY = 0.01f;

  // Asigna el viewport
  resize(w, h);

  // Opciones de dibujo
  glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
  glPolygonMode(GL_FRONT, GL_FILL);
}
```

2. Un modelo 3D sencillo

```
void CGModel::finalize()
{
  delete triangle;
  delete program;
}
void CGModel::resize(int w, int h)
{
  glViewport(0, 0, w, h);
}
```

```
void CGModel::render()
{
  // Limpia el framebuffer
  glClear(GL_COLOR_BUFFER_BIT);

  // Dibuja la escena
  triangle->Draw(program, posX, posY, size);
}
```

2. Un modelo 3D sencillo

```
void CGModel::update()
{
  GLfloat length = 0.01f * size;
  posX += stepX;
  if (posX - length < -1.0f) { posX -= stepX; stepX = 0.02f; }
  if (posX + length > 1.0f) { posX -= stepX; stepX = -0.02f; }

  posY += stepY;
  if (posY - length < -1.0f) { posY -= stepY; stepY = 0.01f; }
  if (posY + length > 1.0f) { posY -= stepY; stepY = -0.01f; }
}
```

2. Un modelo 3D sencillo

```
void CGModel::key_pressed(int key)
{
  switch (key)
  {
  case GLFW_KEY_R:
    posX = 0.0f;
    posY = 0.0f;
    size = 5.0f;
    break;
  case GLFW_KEY_UP:
    size += 0.5;
    if (size >= 50.0f) size = 50.0f;
    break;
  case GLFW_KEY_DOWN:
    size -= 0.5;
    if (size <= 1.0f) size = 1.0f;
    break;
  }
}
```

2. Un modelo 3D sencillo

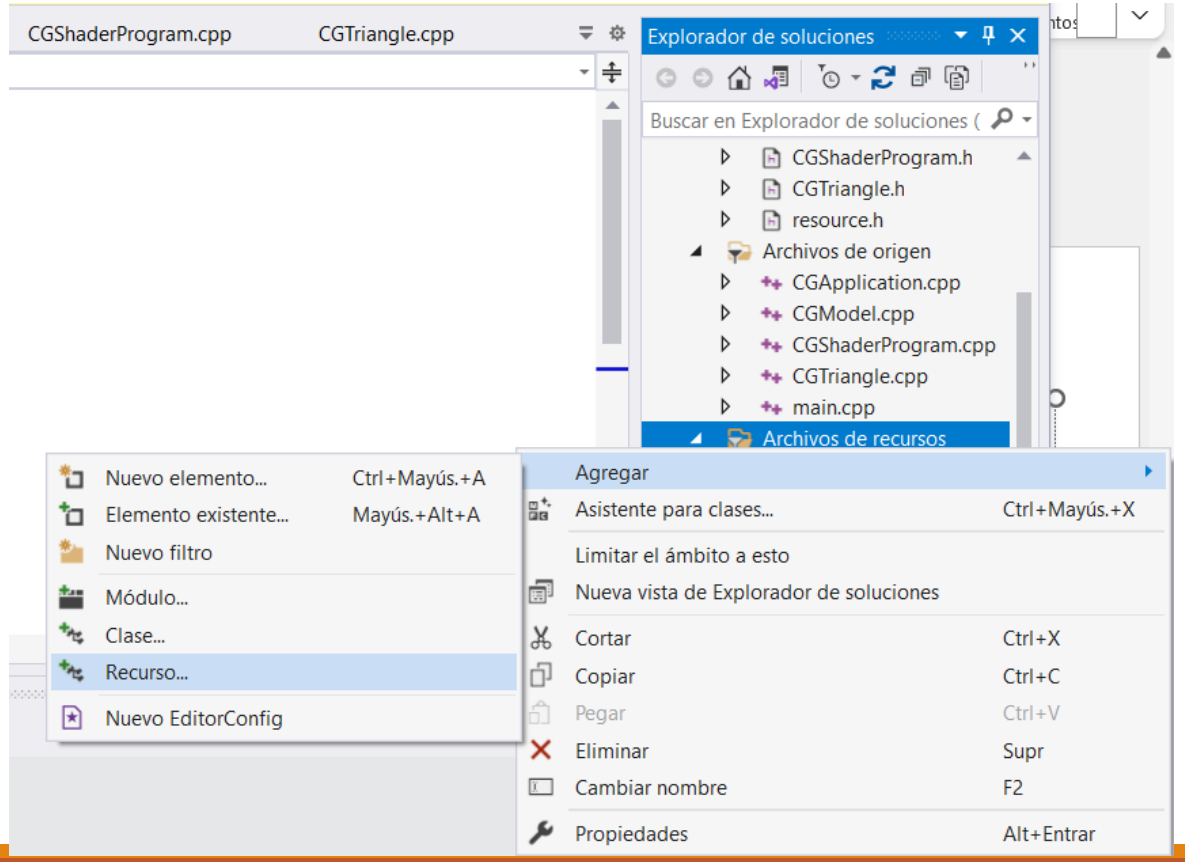
```
void CGModel::mouse_button(int button, int action)
{
}
void CGModel::mouse_move(double xpos, double ypos)
{
}
```

3. Programa gráfico

- Para realizar el programa gráfico es necesario crear el código del Vertex Shader y del Fragment Shader e incluirlos en el programa.
- Una de las formas de definir el código de estos shaders es incluirlo directamente en el código C++ en forma de string.
- Sin embargo, cuando el código del shader es largo, esta forma es incómoda.
- Una forma alternativa consiste en incluir el código de los shader en ficheros externos y cargarlos en tiempo de ejecución.
  - Esto obliga a distribuir los ficheros junto con el ejecutable de la aplicación y se corre el riesgo de que pueden ser manipulados externamente
- La solución propuesta utilizada en las prácticas, consiste en definir el código de los shaders en fichero externos, pero incluir estos ficheros como recursos de la aplicación. Se incrustan en el ejecutable.

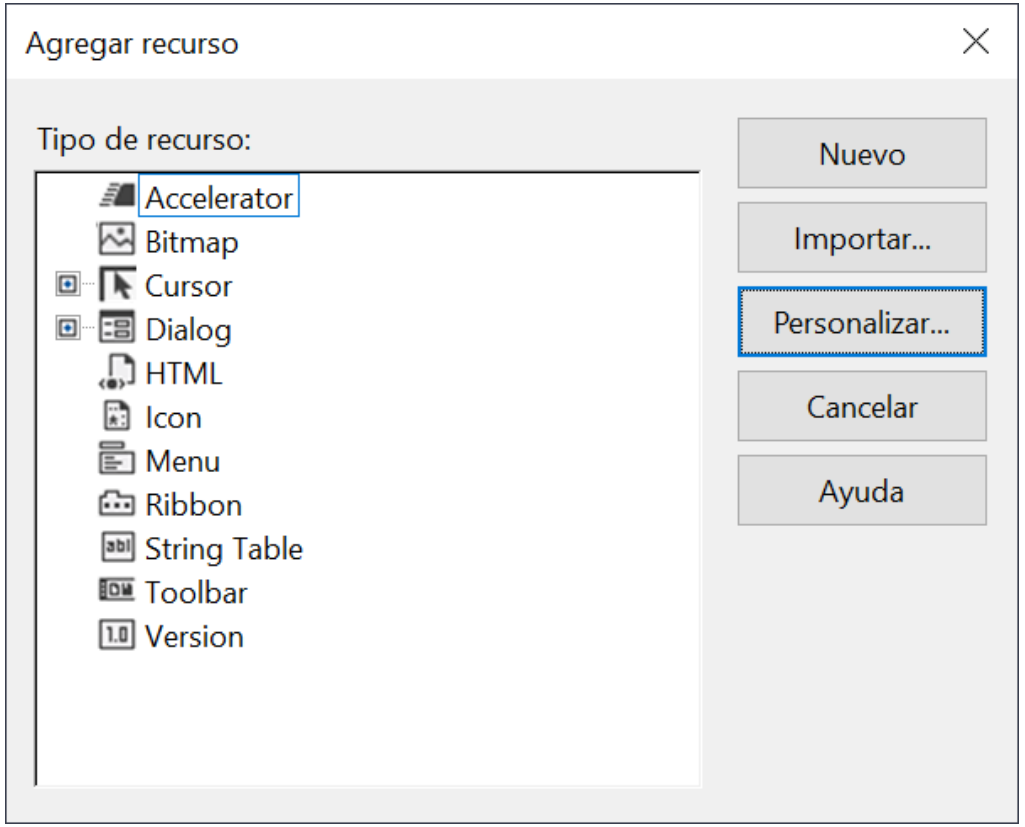
3. Programa gráfico

- En el explorador de soluciones --> Archivos de recursos --> (Clic Derecho) --> Agregar --> Recurso …



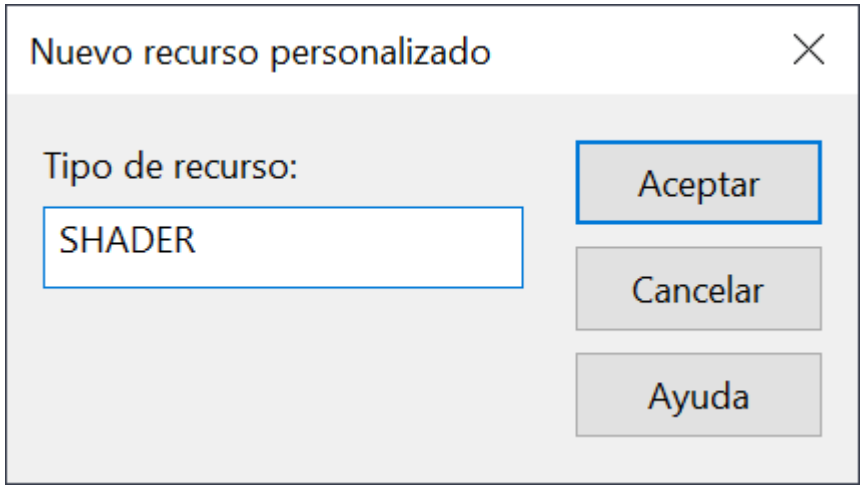
3. Programa gráfico

- Agregar recurso --> Personalizar …



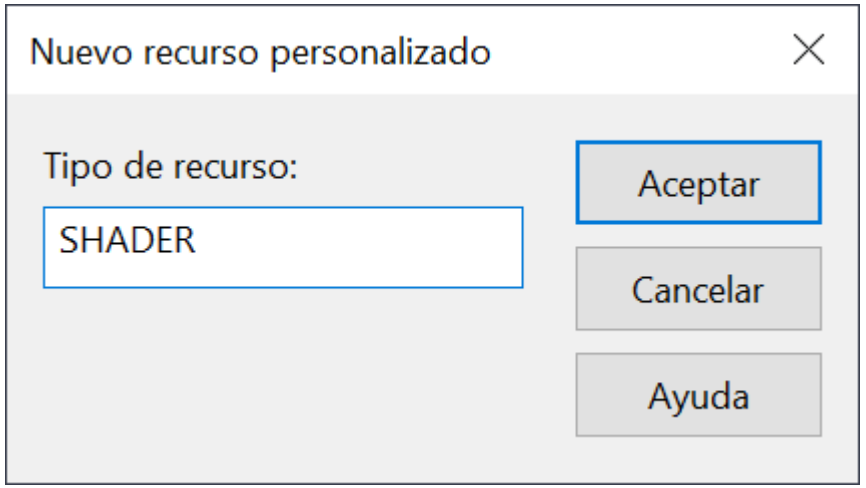
3. Programa gráfico

- Tipo de recurso: SHADER --> Aceptar



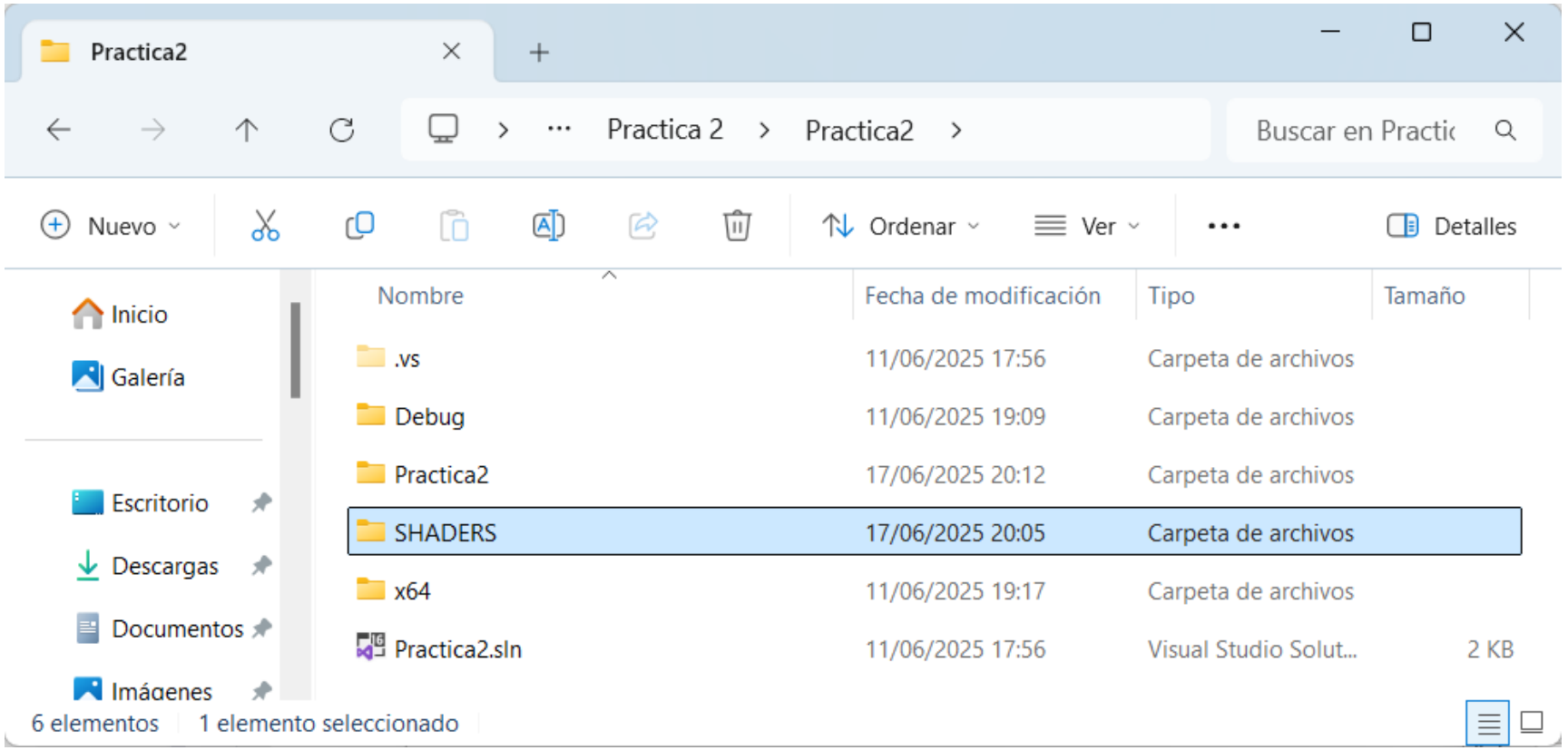
3. Programa gráfico

- Tipo de recurso: SHADER --> Aceptar



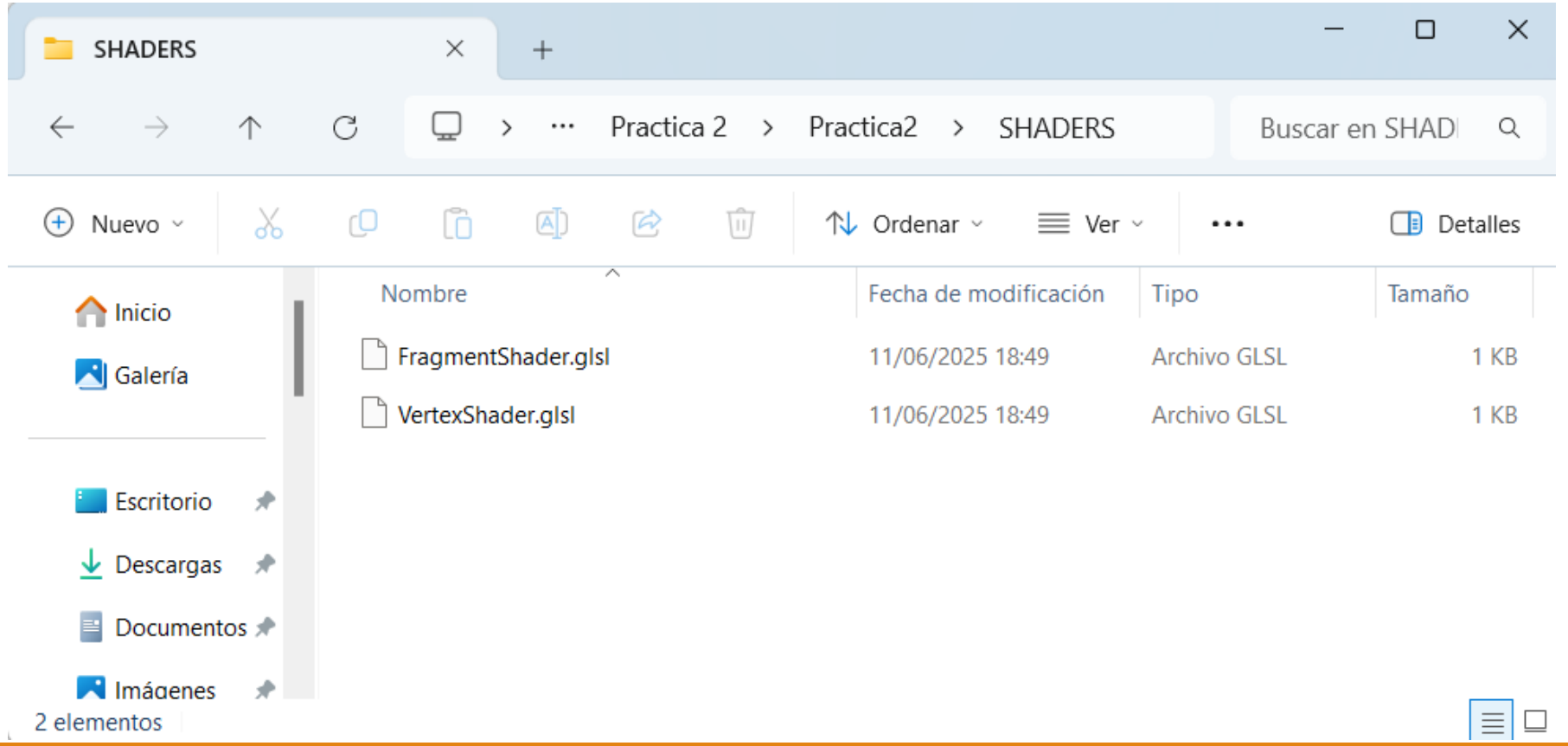
3. Programa gráfico

- Generar en una carpeta externa al proyecto el VertexShader.glsl y el FragmentShader.glsl



3. Programa gráfico

- Generar en una carpeta externa al proyecto el VertexShader.glsl y el FragmentShader.glsl



3. Programa gráfico

- VertexShader:
  - Un único atributo de entrada VertexPosition -> posición de los vértices del triángulo respecto del centro del triángulo
  - Para modificar la posición y el tamaño del triángulo, se consideran tres variables uniformes: posX y posY, describen posición del centro del triángulo en coordenadas Clip, y size, que describe el tamaño del triángulo.
  - La posición final se calcula multiplicando los atributos por el tamaño y desplazando a la posición del centro del triángulo.

3. Programa gráfico

- Contenido VertexShader.glsl

```
#version 400

in vec3 VertexPosition;

uniform float posX;
uniform float posY;
uniform float size;

void main()
{
  gl_Position = vec4( size*VertexPosition.x + posX
                    , size*VertexPosition.y + posY
                    , VertexPosition.z, 1.0);
}
```

3. Programa gráfico

- FragmentShader:
  - Sencillo
  - Salida del FragmentShader es el color de cada pixel del fragmento
  - Todos los píxeles del interior del triángulo se dibujarán de rojo.

3. Programa gráfico

- Contenido FragmentShader.glsl

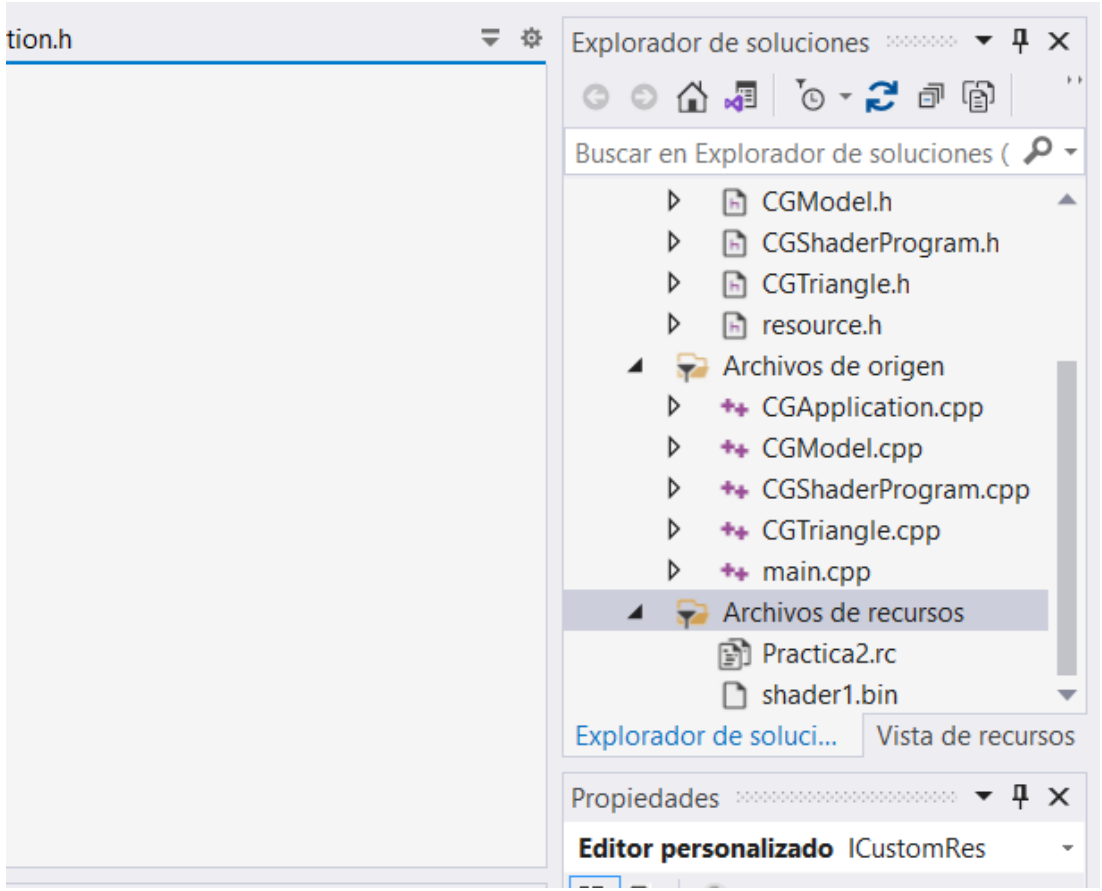
```
#version 400


out vec4 FragColor;


void main()
{
  FragColor = vec4(1.0,0.0,0.0,1.0);
}
```

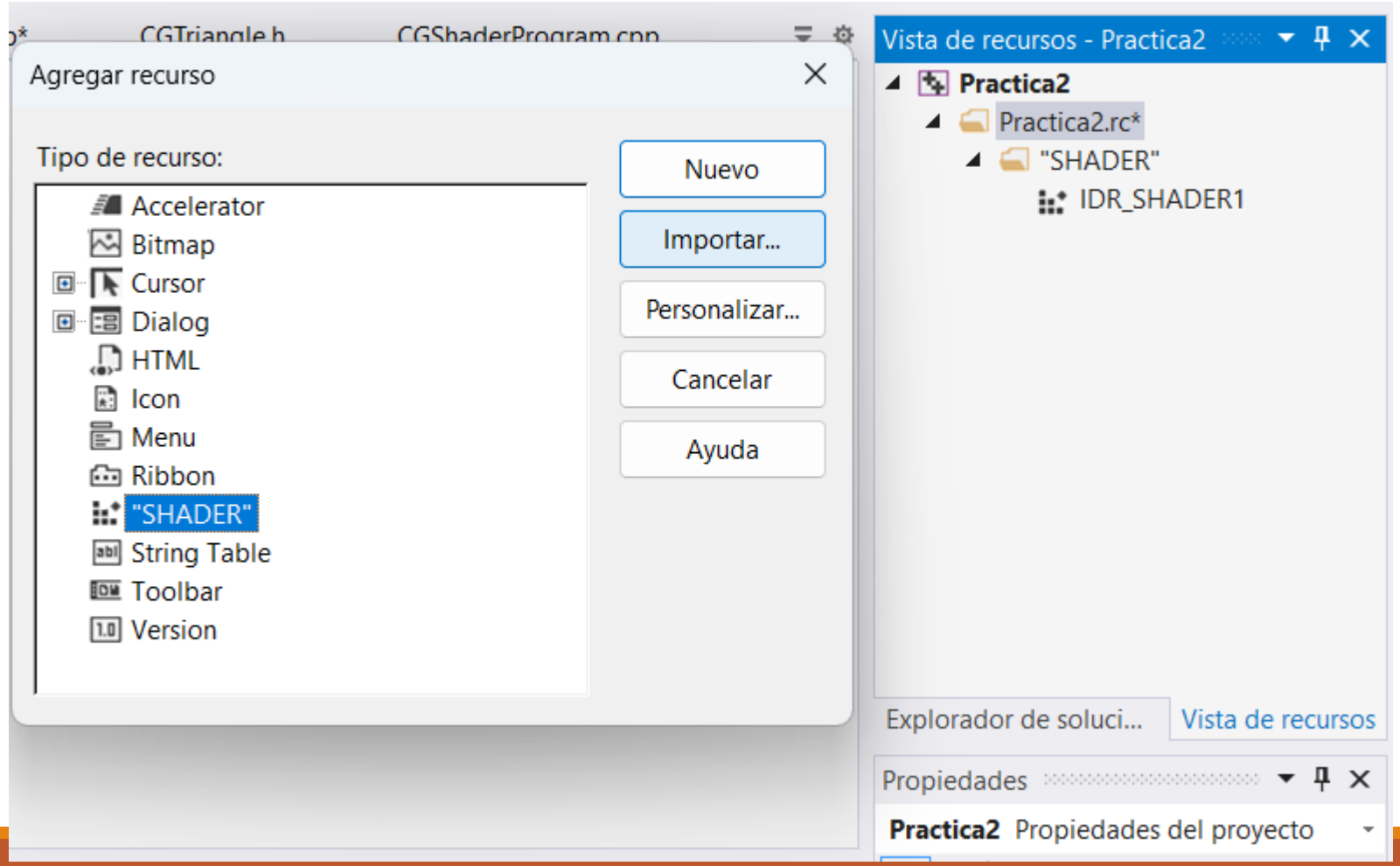
3. Programa gráfico

- Generará este resultado:
  - Eliminamos shader1.bin
  - Doble click izquierdo en Practica2.rc



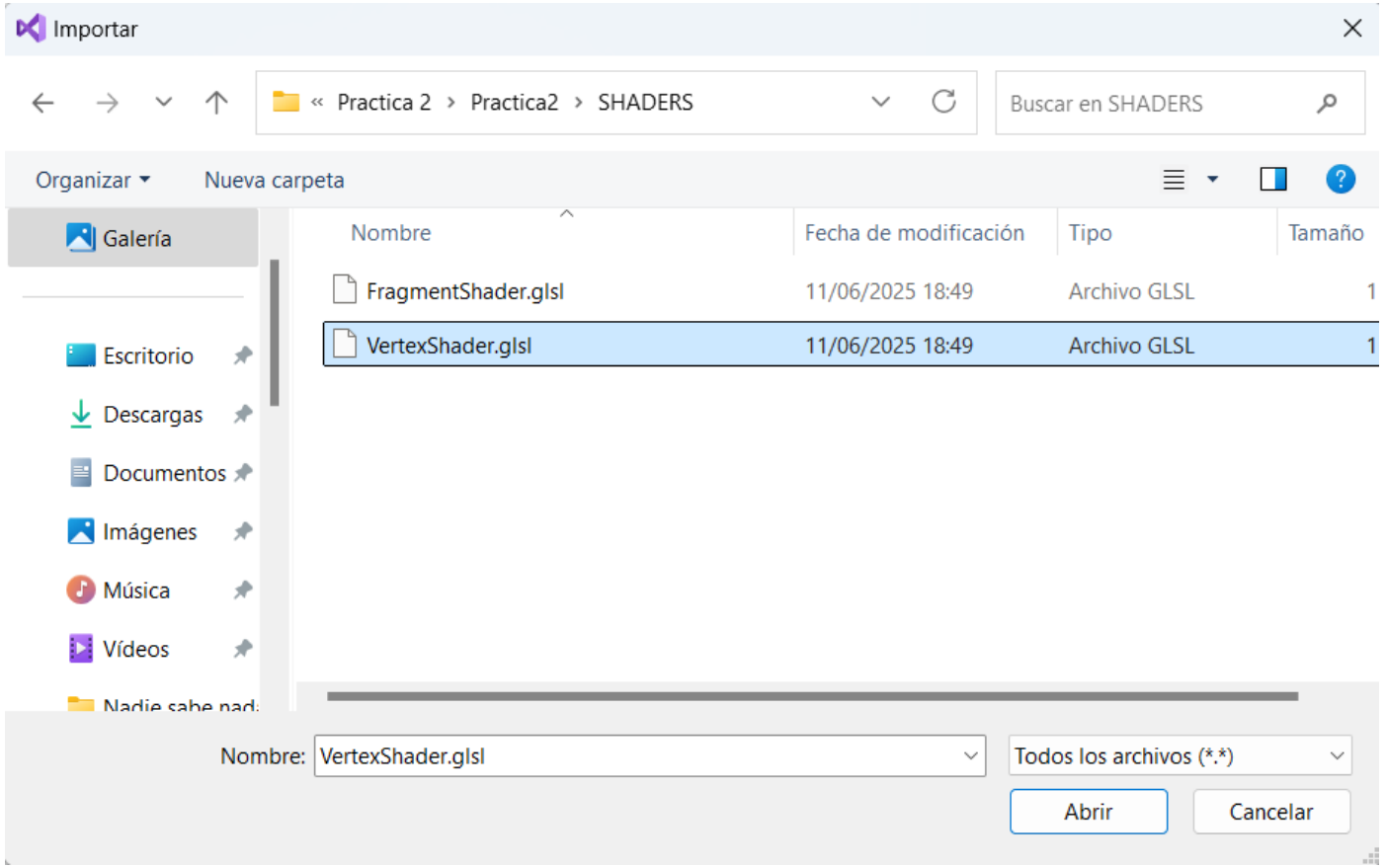
3. Programa gráfico

- Generará este resultado:
  - Click derecho en Practica2.rc
  - ➔ Agregar recurso
  - ➔ Importar …



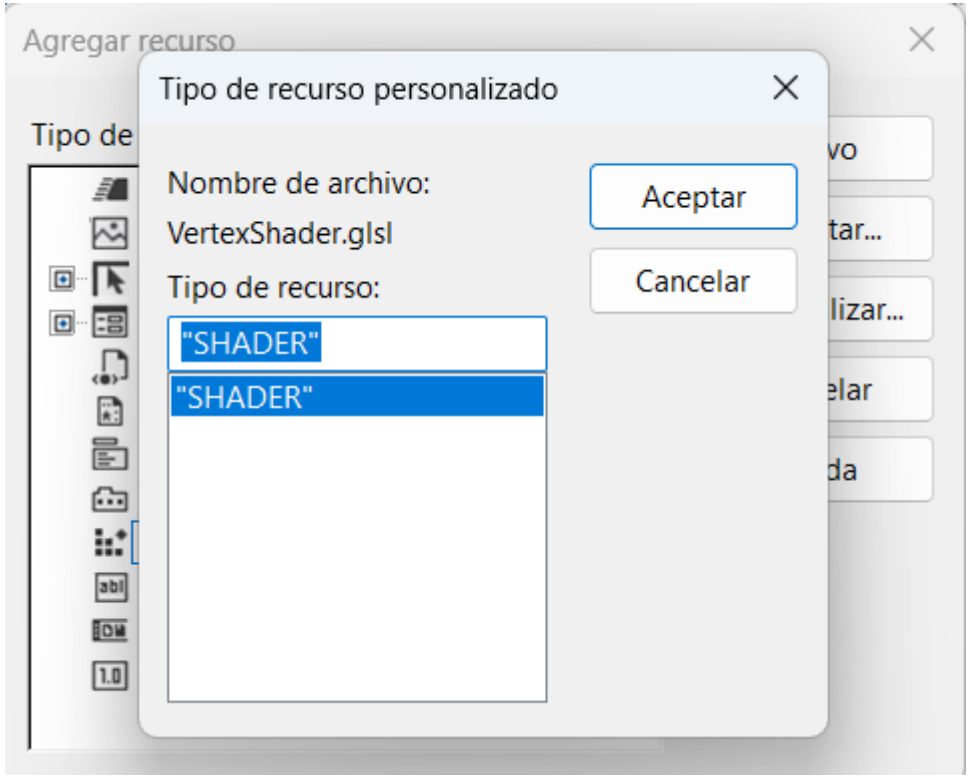
3. Programa gráfico

- Importar …
- Primero VertexShader.glsl



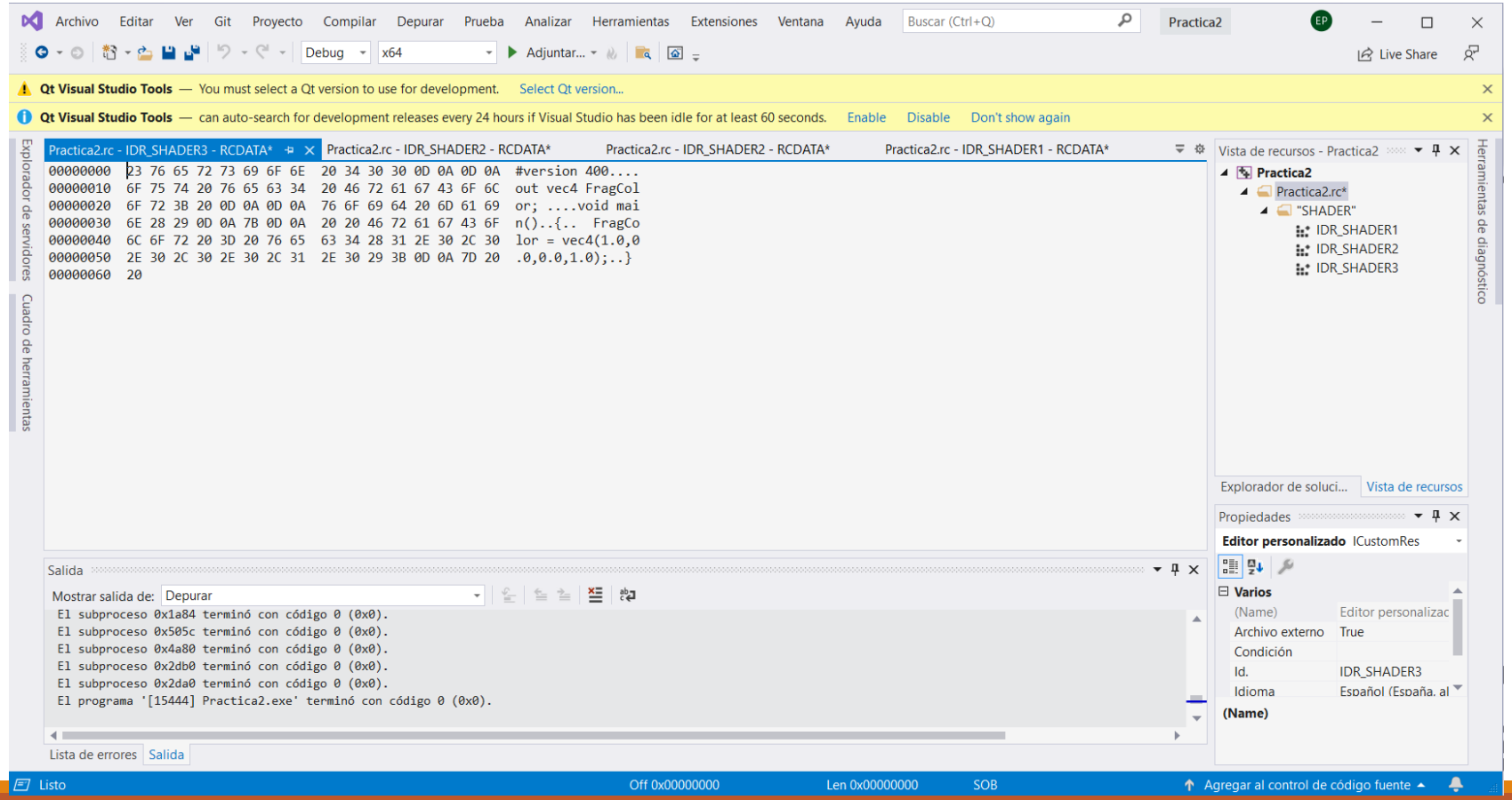
3. Programa gráfico

- ➔Importar …
- ➔Primero VertexShader.glsl
- ➔Elegir el tipo de recurso: SHADER
- ➔Repetir el proceso para el FragmentShader.glsl



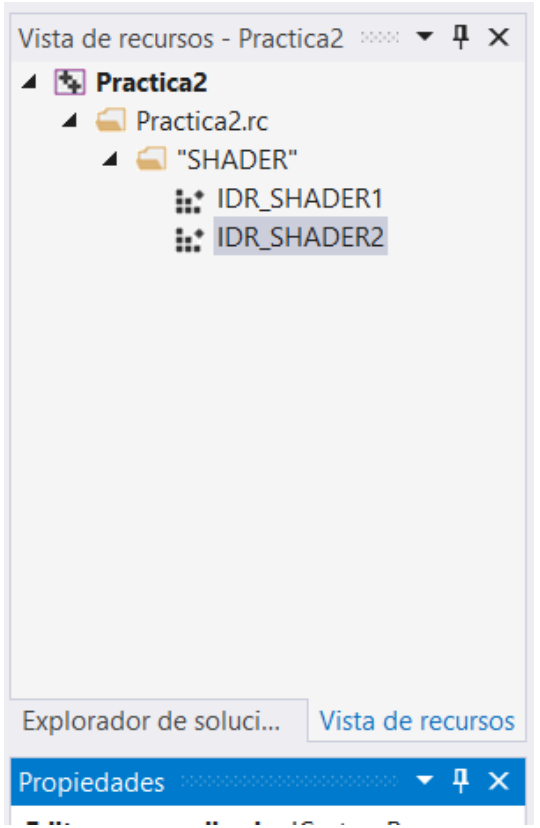
3. Programa gráfico

- Se habrán cargado los recursos y este es el resultado:



3. Programa gráfico

- Eliminar IDR_SHADER1 y renombrar IDR_SHADER2 a IDR_SHADER1 y IDR_SHADER3 a IDR_SHADER2.
- Guardar para aplicar los cambios.
- El Resultado debe ser el siguiente:



3. Programa gráfico

- Para describir programa gráfico se define la clase CGShaderProgram.

```
#pragma once
#include <GL/glew.h>
#include <glm/glm.hpp>
#define NO_SHADER 0xffff
class CGShaderProgram {
private:
  GLuint program;
  GLuint vertexShader;
  GLuint fragmentShader;
  GLuint geometryShader;
  GLuint tessControlShader;
  GLuint tessEvaluationShader;
  GLboolean linked;

  char* GetShaderCodeFromResource(int idr);
  char* GetShaderCodeFromFile(const char* filename);
  GLuint CreateShader(int mode, int idr);
  GLuint CreateShader(int mode, const char* filename);

public:
  CGShaderProgram(int vs, int fs, int gs, int tcs, int tes);
  CGShaderProgram(const char* vs, const char* fs, const char* gs,
                  const char* tcs, const char* tes);
  ~CGShaderProgram();
  GLboolean IsLinked();
  GLvoid Use();
  GLvoid SetUniformF(const char* name, GLfloat f);
  GLvoid SetUniformVec4(const char* name, glm::vec4 m);
  GLvoid SetUniformVec3(const char* name, glm::vec3 m);
  GLvoid SetUniformMatrix4(const char* name, glm::mat4 m);
  GLvoid SetUniformI(const char* name, GLint i);
};
```

3. Programa gráfico

- El desarrollo de la clase CGShaderProgram.cpp
- Constructor: crea los shaders indicados en los argumentos.
- A continuación, se crea el programa y se enlazan los shaders, estudiando si se producen errores al enlazar.
- El destructor elimina los shaders y el programa, cargados en la tarjeta gráfica.
- El método GetShaderCodeFromResource() se encarga de acceder a los ficheros de recursos y cargar el contenido en forma de string.
- El método GetShaderCodeFromFile() permite cargar el código directamente de un fichero externo no insertado como recurso.
- CreateShader(): crea un shader, carga su código y lo compila, se especifica el tipo de shader y el recurso o nombre del fichero en el que se encuentra el código GLSL.
- IsLinked(): informa del resultado del proceso de creación del programa.
- Use(): activa el uso del programa en la tarjeta gráfica.
- SerUniform…(): permite asignar valores a las variables uniformes.

3. Programa gráfico

```
#include <GL/glew.h>
#include <iostream>
#include <SDKDDKVer.h>
#include <Windows.h>
#include "CGShaderProgram.h"
#include "resource.h"

CGShaderProgram::CGShaderProgram(int vs, int fs, int gs, int tcs, int tes)
{
  vertexShader = NO_SHADER;
  fragmentShader = NO_SHADER;
  geometryShader = NO_SHADER;
  tessControlShader = NO_SHADER;
  tessEvaluationShader = NO_SHADER;
  linked = GL_FALSE;

  // Crea y compila los shaders
  if (vs != -1) vertexShader = CreateShader(GL_VERTEX_SHADER, vs);
  if (fs != -1) fragmentShader = CreateShader(GL_FRAGMENT_SHADER, fs);
  if (gs != -1) geometryShader = CreateShader(GL_GEOMETRY_SHADER, gs);
  if (tcs != -1) tessControlShader = CreateShader(GL_TESS_CONTROL_SHADER, tcs);
  if (tes != -1) tessEvaluationShader = CreateShader(GL_TESS_EVALUATION_SHADER, tes);
```

3. Programa gráfico

```
//Crea el programa y carga los shaders
  program = glCreateProgram();
  if (vertexShader != NO_SHADER)
glAttachShader(program, vertexShader);
  if (fragmentShader != NO_SHADER)
glAttachShader(program, fragmentShader);
  if (geometryShader != NO_SHADER)
glAttachShader(program, geometryShader);
  if (tessControlShader != NO_SHADER)
glAttachShader(program, tessControlShader);
  if (tessEvaluationShader != NO_SHADER)
glAttachShader(program, tessEvaluationShader);
  glLinkProgram(program);
  GLint status;
  glGetProgramiv(program, GL_LINK_STATUS, &status);
  if (status == GL_FALSE)
  {
    linked = GL_FALSE;
    GLint logLength;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH,
&logLength);
    char* logInfo = (char*)malloc(sizeof(char) *
logLength);
    GLsizei written;
    glGetProgramInfoLog(program, logLength, &written,
logInfo);
    std::cout << logInfo << std::endl;
    return;
  }

  linked = GL_TRUE;
}
```

3. Programa gráfico

```
CGShaderProgram::CGShaderProgram(const char* vs, const char* fs, const char* gs,
                                 const char* tcs, const char* tes)
{
  vertexShader = NO_SHADER;
  fragmentShader = NO_SHADER;
  geometryShader = NO_SHADER;
  tessControlShader = NO_SHADER;
  tessEvaluationShader = NO_SHADER;
  linked = GL_FALSE;
  // Crea y compila los shaders
  if (vs != NULL) vertexShader = CreateShader(GL_VERTEX_SHADER, vs);
  if (fs != NULL) fragmentShader = CreateShader(GL_FRAGMENT_SHADER, fs);
  if (gs != NULL) geometryShader = CreateShader(GL_GEOMETRY_SHADER, gs);
  if (tcs != NULL) tessControlShader = CreateShader(GL_TESS_CONTROL_SHADER, tcs);
  if (tes != NULL) tessEvaluationShader = CreateShader(GL_TESS_EVALUATION_SHADER, tes);

  //Crea el programa y carga los shaders
  program = glCreateProgram();
  if (vertexShader != NO_SHADER) glAttachShader(program, vertexShader);
  if (fragmentShader != NO_SHADER) glAttachShader(program, fragmentShader);
  if (geometryShader != NO_SHADER) glAttachShader(program, geometryShader);
  if (tessControlShader != NO_SHADER) glAttachShader(program, tessControlShader);
  if (tessEvaluationShader != NO_SHADER) glAttachShader(program, tessEvaluationShader);
  glLinkProgram(program);
```

3. Programa gráfico

```
 GLint status;
  glGetProgramiv(program, GL_LINK_STATUS, &status);
  if (status == GL_FALSE)
  {
    linked = GL_FALSE;
    GLint logLength;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &logLength);
    char* logInfo = (char*)malloc(sizeof(char) * logLength);
    GLsizei written;
    glGetProgramInfoLog(program, logLength, &written, logInfo);
    std::cout << logInfo << std::endl;
    return;
  }

  linked = GL_TRUE;
}
```

3. Programa gráfico

```
GLuint CGShaderProgram::CreateShader(int mode, int idr)
{
  GLint status;
  GLuint shader = glCreateShader(mode);
  char* code = GetShaderCodeFromResource(idr);
  glShaderSource(shader, 1, &code, NULL);
  glCompileShader(shader);
  free(code);
  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  if (status == GL_FALSE)
  {
    GLint logLength;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLength);
    char* logInfo = (char*)malloc(sizeof(char) * logLength);
    GLsizei written;
    glGetShaderInfoLog(shader, logLength, &written, logInfo);
    std::cout << logInfo << std::endl;
    return NO_SHADER;
  }
  return shader;
}
```

3. Programa gráfico

```
GLuint CGShaderProgram::CreateShader(int mode, const char* filename)
{
  GLint status;
  GLuint shader = glCreateShader(mode);
  char* code = GetShaderCodeFromFile(filename);
  glShaderSource(shader, 1, &code, NULL);
  glCompileShader(shader);
  free(code);
  glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
  if (status == GL_FALSE)
  {
    GLint logLength;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLength);
    char* logInfo = (char*)malloc(sizeof(char) * logLength);
    GLsizei written;
    glGetShaderInfoLog(shader, logLength, &written, logInfo);
    std::cout << logInfo << std::endl;
    return NO_SHADER;
  }
  return shader;
}
```

3. Programa gráfico

```
char* CGShaderProgram::GetShaderCodeFromResource(int idr)
{
  HRSRC shaderHandle = FindResource(NULL, MAKEINTRESOURCE(idr), L"SHADER");
  HGLOBAL shaderGlobal = LoadResource(NULL, shaderHandle);
  LPCTSTR shaderPtr = static_cast<LPCTSTR>(LockResource(shaderGlobal));
  DWORD shaderSize = SizeofResource(NULL, shaderHandle);
  char* shaderCodeLine = (char*)malloc((shaderSize + 1) * sizeof(char));
  memcpy(shaderCodeLine, shaderPtr, shaderSize);
  shaderCodeLine[shaderSize] = '\0';
  FreeResource(shaderGlobal);
  return shaderCodeLine;
}
```

3. Programa gráfico

```
char* CGShaderProgram::GetShaderCodeFromFile(const char* filename)
{
  FILE* file = NULL;
  fopen_s(&file,filename, "r");
  fseek(file, 0L, SEEK_END);
  int size = ftell(file);
  fseek(file, 0L, SEEK_SET);
  char* code = (char*)malloc(sizeof(char) * (size + 1));
  int read = fread(code, sizeof(char), size, file);
  code[read] = '\0';
  fclose(file);
  return code;
}
```

3. Programa gráfico

```
CGShaderProgram::~CGShaderProgram()
{
  if (vertexShader != NO_SHADER)
glDeleteShader(vertexShader);
  if (fragmentShader != NO_SHADER)
glDeleteShader(fragmentShader);
  if (geometryShader != NO_SHADER)
glDeleteShader(geometryShader);
  if (tessControlShader != NO_SHADER)
glDeleteShader(tessControlShader);
  if (tessEvaluationShader != NO_SHADER)
glDeleteShader(tessEvaluationShader);
  glDeleteProgram(program);
}

GLboolean CGShaderProgram::IsLinked()
{
  return linked;
}

GLvoid CGShaderProgram::Use()
{
  glUseProgram(program);
}
```

3. Programa gráfico

```
void CGShaderProgram::SetUniformF(const char * name, GLfloat f)
{
  GLuint location = glGetUniformLocation(program, name);
  if (location >= 0) glUniform1f(location, f);
}

GLvoid CGShaderProgram::SetUniformMatrix4(const char *name, glm::mat4 m)
{
  GLuint location = glGetUniformLocation(program, name);
  if (location >= 0) glUniformMatrix4fv(location, 1, GL_FALSE, &m[0][0]);
}

void CGShaderProgram::SetUniformVec4(const char * name, glm::vec4 v)
{
  GLuint location = glGetUniformLocation(program, name);
  if (location >= 0) glUniform4fv(location, 1, &v[0]);
}
```

3. Programa gráfico

```
void CGShaderProgram::SetUniformVec3(const char * name, glm::vec3 v)
{
  GLuint location = glGetUniformLocation(program, name);
  if (location >= 0) glUniform3fv(location, 1, &v[0]);
}

void CGShaderProgram::SetUniformI(const char * name, GLint i)
{
  GLuint location = glGetUniformLocation(program, name);
  if (location >= 0) glUniform1i(location, i);
}
```

4. Creación de un triángulo

- Clase que describe el triángulo CGTriangle.h
- Clase formada por dos campos:
  - Identificador del VertexArrayObject del objeto
  - Identficador del VertexBufferObject que almacenará los valores del atributo VertexPosition
- La clase incluye el método Draw() que se encarga de activar el VAO y lanzar la primitiva que dibuje el triángulo.

4. Creación de un triángulo

- Implementación CGTriangle.h

```
#pragma once

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

class CGTriangle {
private:
  GLuint VBO;
  GLuint VAO;

public:
  CGTriangle();
  ~CGTriangle();
  void Draw(CGShaderProgram* program,
GLfloat posX, GLfloat posY, GLfloat size);
};
```

4. Creación de un triángulo

- Desarrollo en CGTriangle.cpp
- Para construir el objeto se crea un array de valores con las coordenadas de los vértices que formarán el triángulo.
- A partir de ahí, se crea el VAO y el VBO y se introduce el contenido.
- El destructor de la clase elimina el VAO y el VBO, liberando la memoria que ocupan en la tarjeta gráfica.
- El método Draw() asigna los valores a las variables uniformes y a continuación, activa el VAO y lanza una primitiva GL_TRIANGLES indicando que se tomen tres vértices, por lo que se dibujará un único triángulo.

4. Creación de un triángulo

- Implementación CGTriangle.cpp

```
#include "CGTriangle.h"
#include <GL/glew.h>
CGTriangle::CGTriangle()
{
 int numVertices = 3;
 int numFaces = 1;
 GLfloat vertices[9] = {
  -0.01f, -0.01f, 0.5f,
  0.01f, -0.01f, 0.5f,
  0.0f, 0.01f, 0.5f
 };
 // Create the Vertex Array Object
 glGenVertexArrays(1, &VAO);
 glBindVertexArray(VAO);

 // Create the Vertex Buffer Objects
 glGenBuffers(1, &VBO);
 // Vertex data
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat) * numVertices * 3, vertices,
                                                    GL_STATIC_DRAW);
 glEnableVertexAttribArray(0); // Vertex position
 glBindBuffer(GL_ARRAY_BUFFER, VBO);
 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
}
```

4. Creación de un triángulo

- Implementación CGTriangle.cpp

```
CGTriangle::~CGTriangle()
{
  // Delete vertex buffer objects
  glDeleteBuffers(1, &VBO);
  // Delete vetex array object
  glDeleteVertexArrays(1, &VAO);
}
void CGTriangle::Draw(CGShaderProgram* program, GLfloat posX, GLfloat posY, GLfloat size)
{
  program->SetUniformF("posX", posX);
  program->SetUniformF("posY", posY);
  program->SetUniformF("size", size);

  glBindVertexArray(VAO);
  glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

5. Aspecto Final