Realidad Virtual. Práctica 1.

OPENGL SOBRE MS-WINDOWS

Índice

1. Objetivos

- Estructura básica de un programa basado en OpenGL sobre MS-Windows.
- Prácticas siguientes características de OpenGL y no de la plataforma.
- Biblioteca de manejo de ventanas GLFW:
  - Construiremos ventana principal de la aplicación.
  - Permite configurar respuestas a eventos de la ventana y utilizar OpenGL sobre el contexto gráfico de la ventana.
- Código de la práctica:
  https://www.uhu.es/francisco.moreno/gii_rv/practicas/practica01/practica01.rar

2. OpenGL

- No es un lenguaje de programación
- Es una biblioteca de funciones escrita en C dedicadas a la generación de gráficos.
- La implementación genérica de OpenGL sobre MS-Windows no ha sido actualizada desde la versión 1.1 de OpenGL. Se encuentra en C:\Windows\System32\opengl32.dll
- El contenido de la biblioteca opengl32.dll depende de cada fabricante.
  - En algunos casos, cuando una determinada tarjeta gráfica no contiene algún proceso de aceleración gráfica incluido en la especificación de OpenGL, el fabricante lo suple con una implementación software de dicho proceso.
  - Además, suelen incluir opciones de aceleración que no se encuentran en el estándar de OpenGL.
  - La biblioteca contiene funciones para verificar si una cierta característica está soportada o no por el driver suministrado por el fabricante.
  - En el desarrollo de programas basados en OpenGL se suelen estudiar y aprovechar estas características.

2. OpenGL

- La primera versión de OpenGL (OpenGL 1.0) fue presentada en 1992.
  - Desde entonces se han presentado numerosas versiones que han ido ampliando el estándar.
  - 1.0, 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, 2.1, 3.0, 3.1, 3.2, 3.3, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5 y 4.6.
  - Esto provoca un problema a la hora de compilar los proyectos desarrollados en OpenGL.
  - En tiempo de ejecución la aplicación se en laza dinámicamente con la biblioteca opengl32.dll desarrollada por el fabricante de la tarjeta gráfica. Pueden aparecer conflictos.
  - Para evitar esto, se suele usar la biblioteca GLEW (OpenGL Extension Wrangler Library. Librería multiplataforma escrita en C/C++, destinada a ayudar en la carga y consulta de extensiones de OpenGL. GLEW incluye métodos eficientes, en tiempo de ejecución, para determinar qué extensiones de OpenGL son soportadas.
  - Todas las extensiones de OpenGL son listadas en un solo archivo de cabecera, generado automáticamente respecto a la lista oficial de extensiones.

2. OpenGL

- OpenGL no pretende ser un window manager.
  - No contiene funciones de manejo de ventanas (creación, configuración, etc)
  - Ni de control de eventos de la interfaz de usuario (control de teclado, ratón, sonido)
  - Ni de interfaz de entrada/salida (acceso a ficheros, directorios, …)
- Corresponde al sistema operativo nativo la gestión de ventanas y de estos eventos.
- Las funciones incluidas en OpenGL se limitan a trabajar sobre un contexto gráfico que permita dirigir las imágenes creadas por la tarjeta gráfica a un área determinada de una ventana controlada por el sistema operativo nativo.
- Para crear y manejar los aspectos de la ventana se suele utilizar alguna biblioteca que permita trabajar con ventanas de una forma más sencilla.
  - Como GLUT, FreeGLUT, CPW, FLTK, SDL o GLFW. Se usará en las prácticas GLFW.

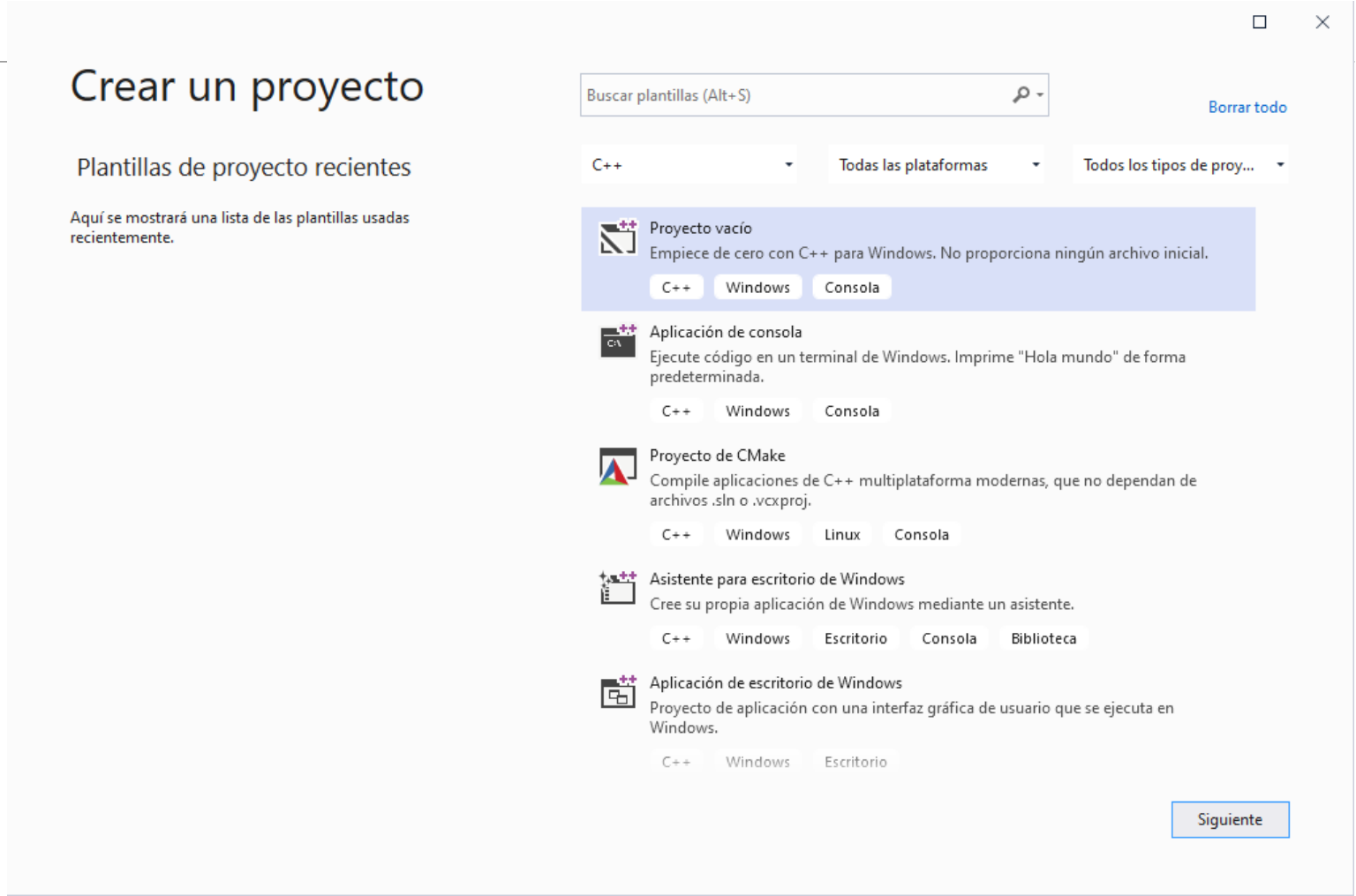
2. OpenGL

- La programación gráfica requiere trabajar tanto sobre el procesador principal (CPU) como sobre los procesadores de la tarjeta gráfica (GPU).
- La programación en la tarjeta gráfica se realiza en un lenguaje de programación denominado GLSL, que utiliza un conjunto de tipos de datos propio, que incluye tipos de datos vectoriales y matriciales, así como un conjunto de funciones predefinidas.
  - Para facilitar esto se utiliza la biblioteca GLM, que define sobre C/C++ los tipos de datos utilizados en GLSL, así cómo numerosas funciones y operadores asociados a estos tipos de datos.
- FreeImage es una biblioteca dedicada al tratamiento de imágenes que utilizaremos para cargar las texturas en nuestras aplicaciones.
- Estas librerías se encuentran en \ComputerGraphics\Tools.

3. Creación de un proyecto de aplicación gráfica

Visual Studio 2019

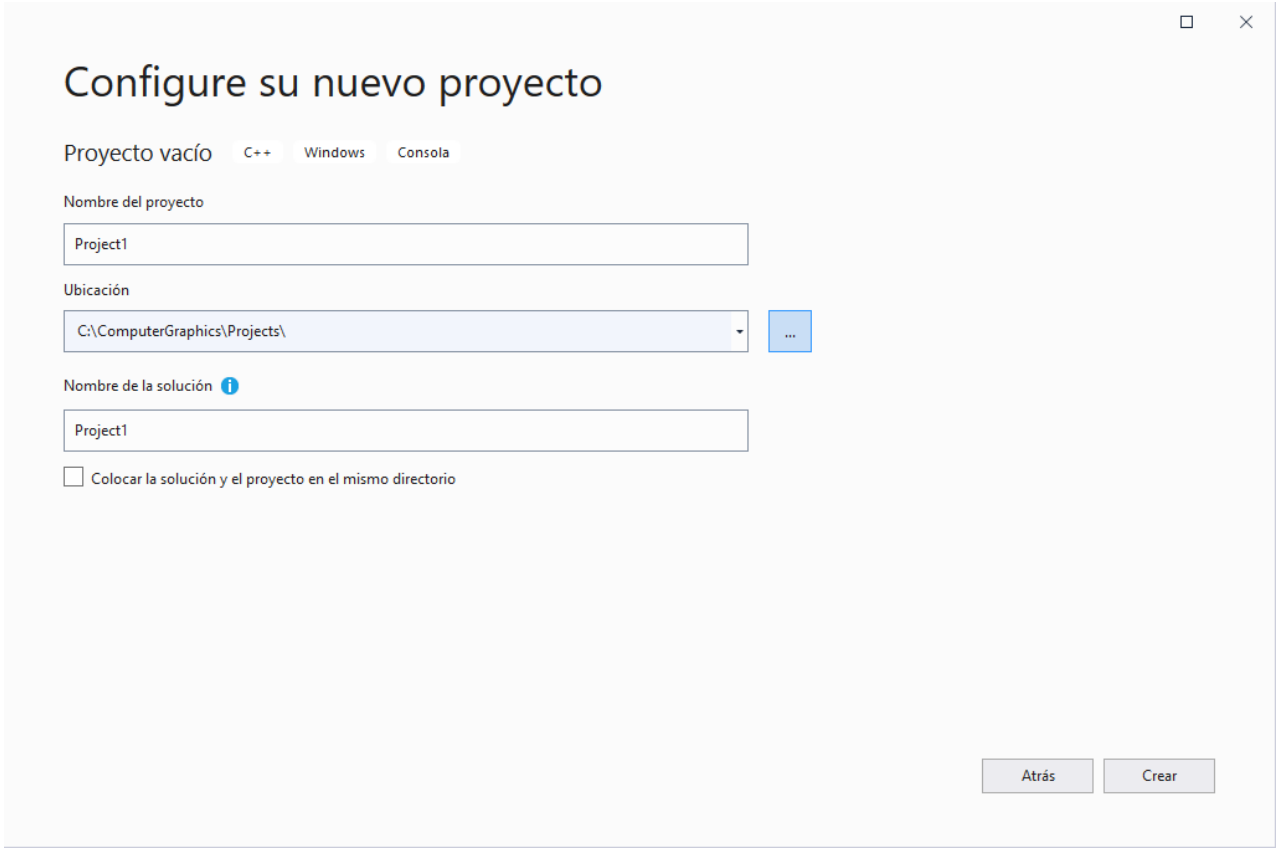
--> Proyecto Vacío 2019



3. Creación de un proyecto de aplicación gráfica

Nombre de proyecto

Y directorio de trabajo



3. Creación de un proyecto de aplicación gráfica

Crear main.cpp

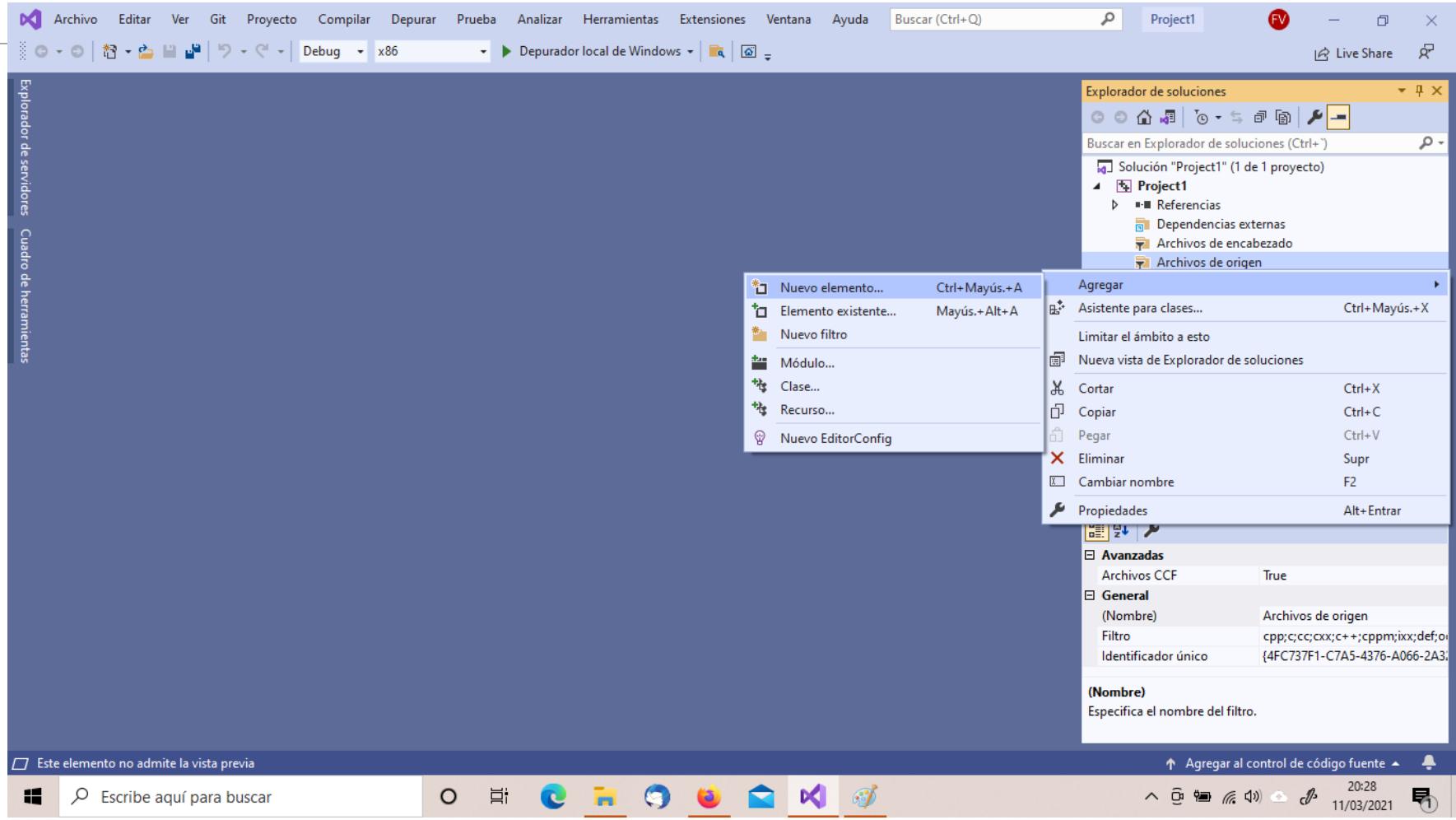
Explorador de soluciones

→Archivos de origen

→(Botón derecho)

→Agregar

→Nuevo elemento …

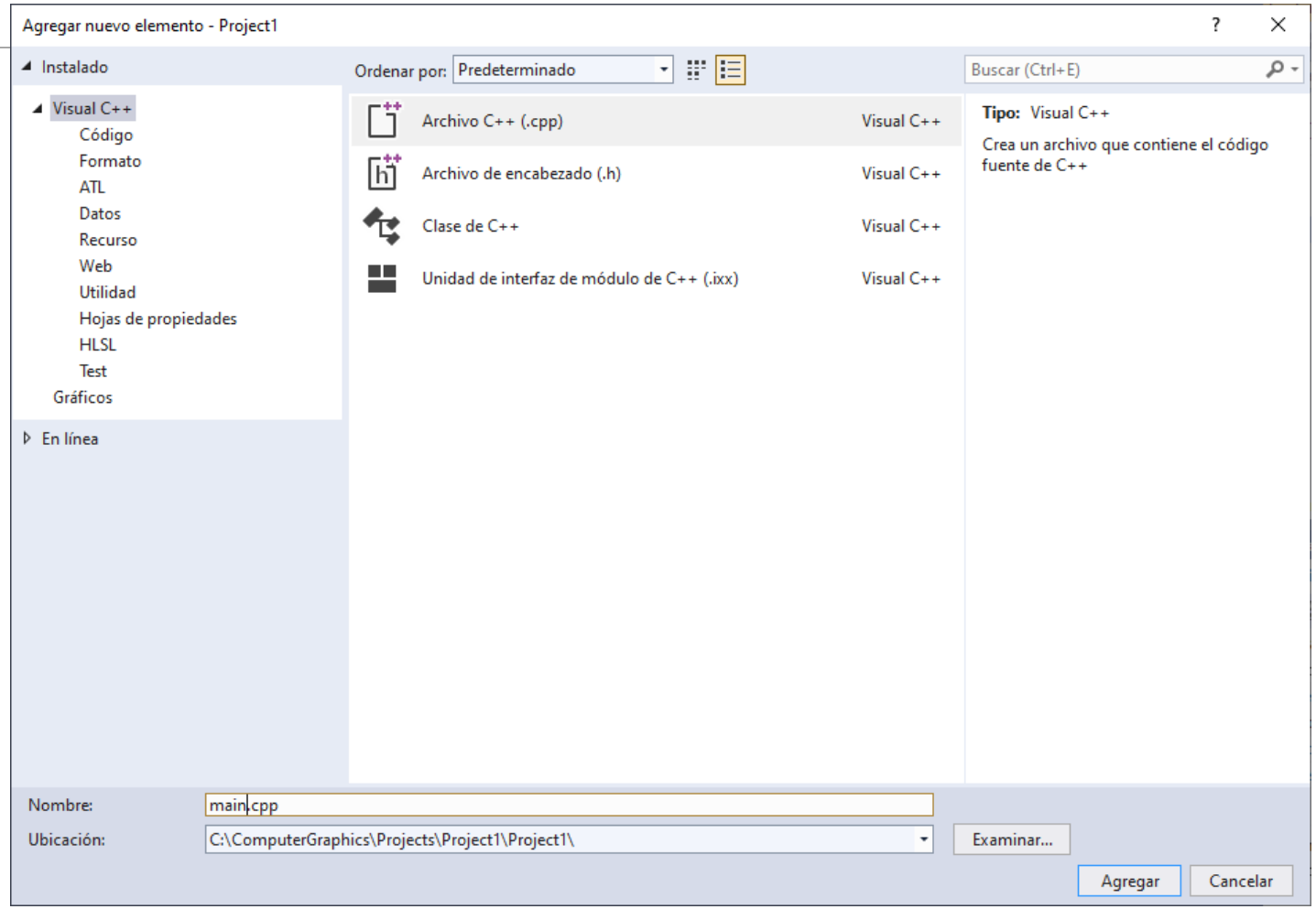


3. Creación de un proyecto de aplicación gráfica

Crear main.cpp

Seleccionar archivo .cpp

Nombrar como main.cpp

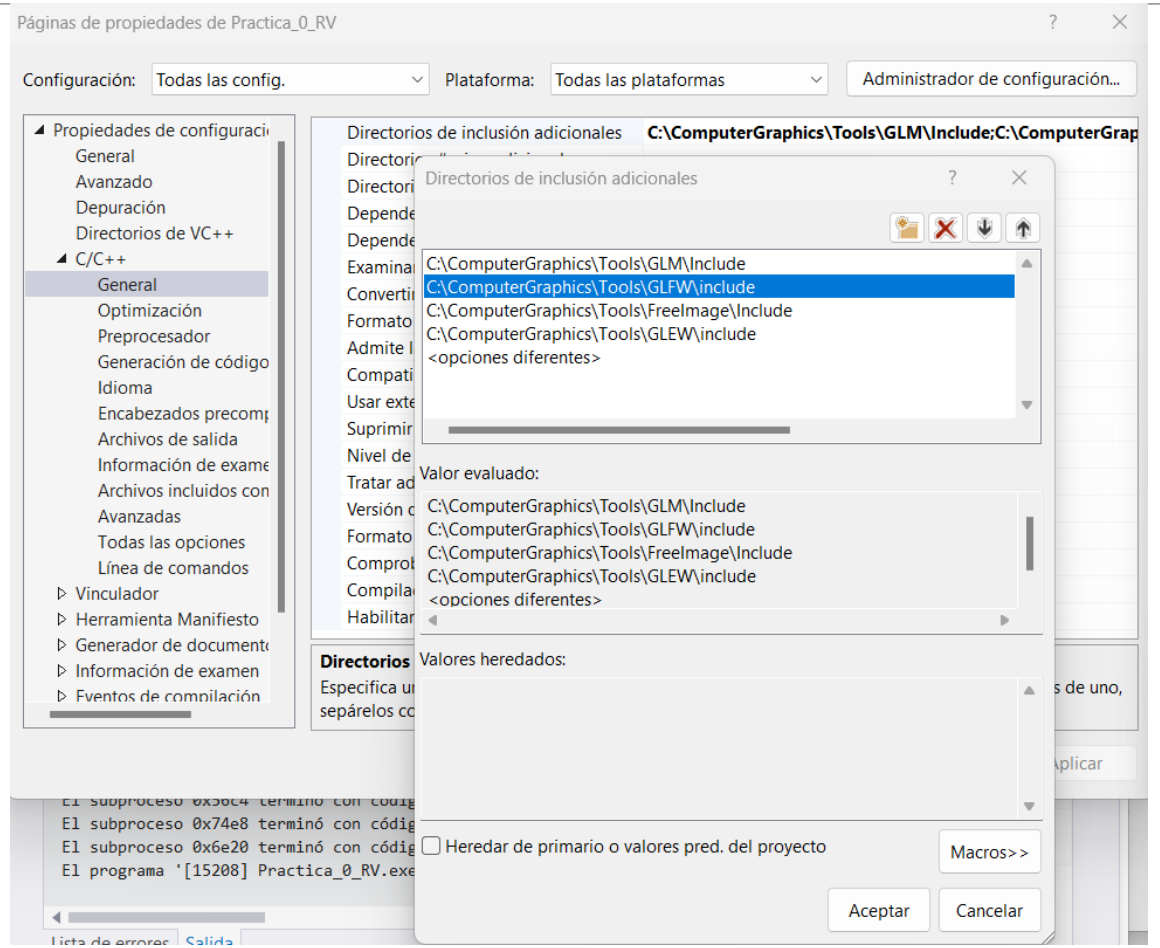


3. Creación de un proyecto de aplicación gráfica

- Para utilizar OpenGL y otras bibliotecas auxiliares, como FreeImage, GLFW y GLM, es necesario modificar las propiedades del proyecto.
- Las bibliotecas auxiliares se incluyen en C:\ComputerGraphics\Tools
- Propiedades del proyecto: Barra de menú --> Proyecto --> Propiedades de …
  - Configuración: (Seleccionar) Todas las config.
  - Plataforma: (Seleccionar) Todas las plataformas
  - C/C++: General --> Directorios de inclusión adicionales --> (Editar):
    - C:\ComputerGraphics\Tools\FreeImage\Include
    - C:\ComputerGraphics\Tools\GLEW\include
    - C:\ComputerGraphics\Tools\GLFW\include
    - C:\ComputerGraphics\Tools\GLM\Include

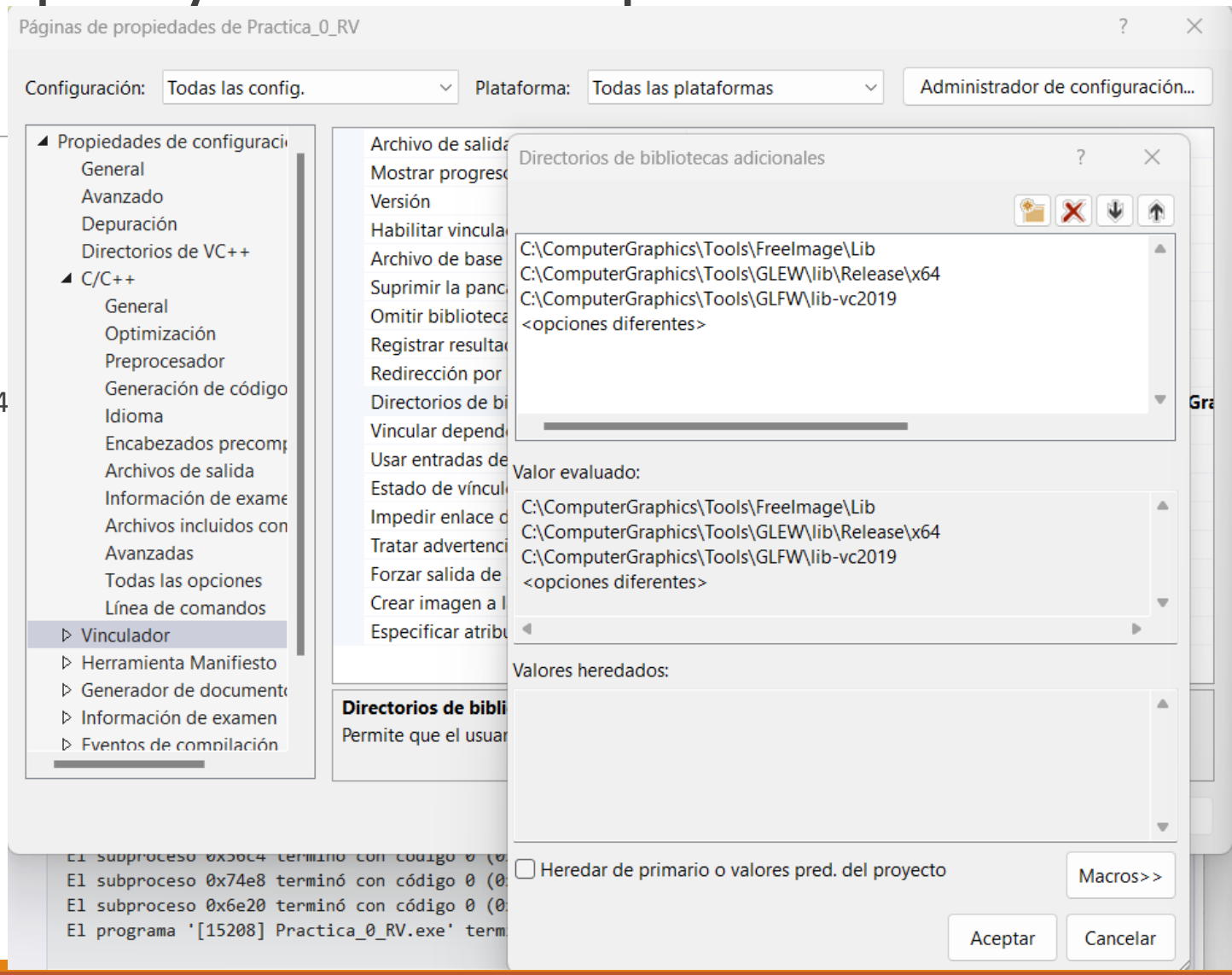
3. Creación de un proyecto de aplicación gráfica

- C/C++: General
- → Directorios de inclusión adicionales:
- → (Editar):
  - C:\ComputerGraphics\Tools\FreeImage\Include
  - C:\ComputerGraphics\Tools\GLEW\include
  - C:\ComputerGraphics\Tools\GLFW\include
  - C:\ComputerGraphics\Tools\GLM\Include



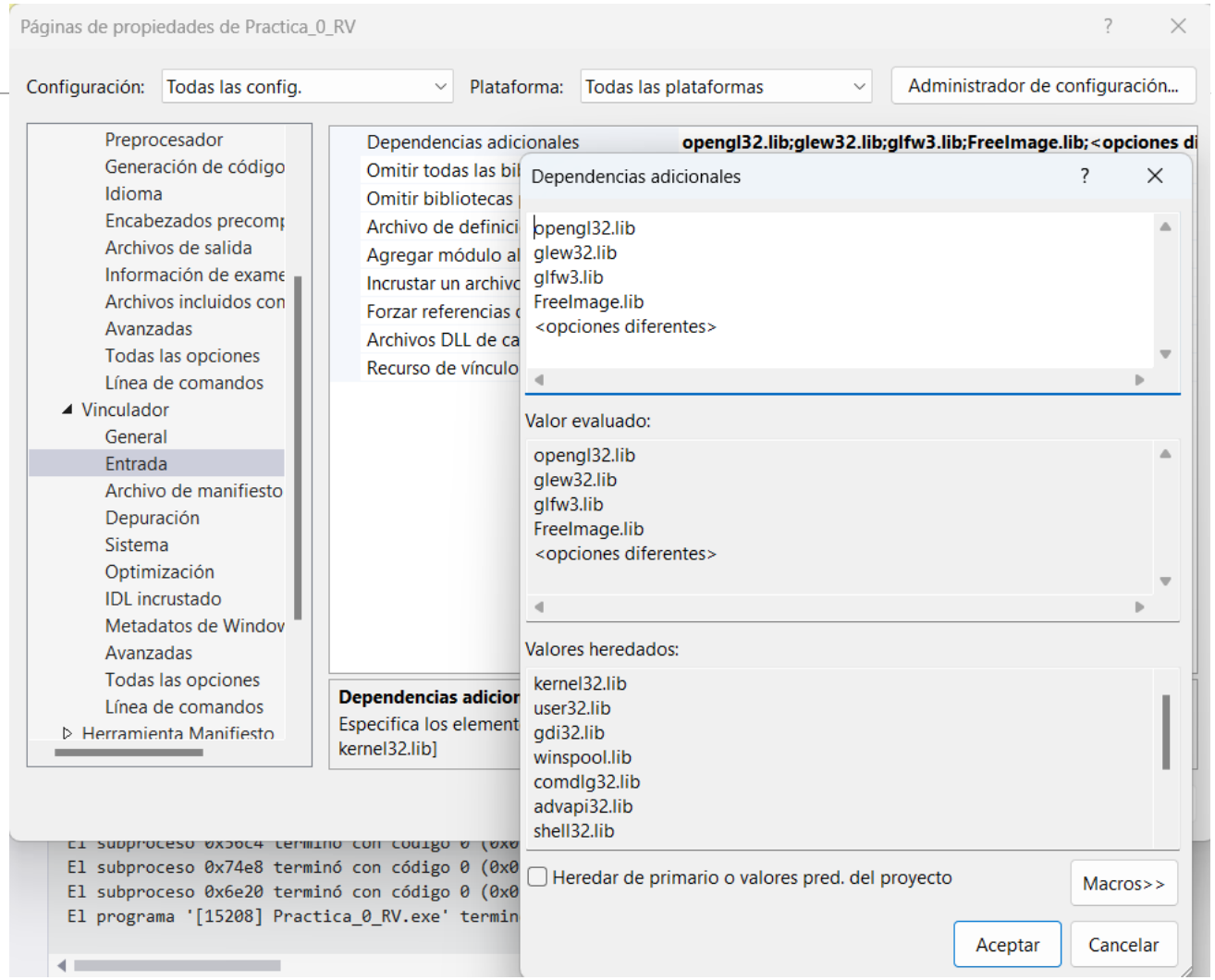
3. Creación de un proyecto de aplicación gráfica

- Vinculador: General
- → Directorios de bibliotecas adicionales
- → (Editar):
  - C:\ComputerGraphics\Tools\FreeImage\Lib
  - C:\ComputerGraphics\Tools\GLEW\lib\Release\x64
  - C:\ComputerGraphics\Tools\GLFW\lib-vc2019



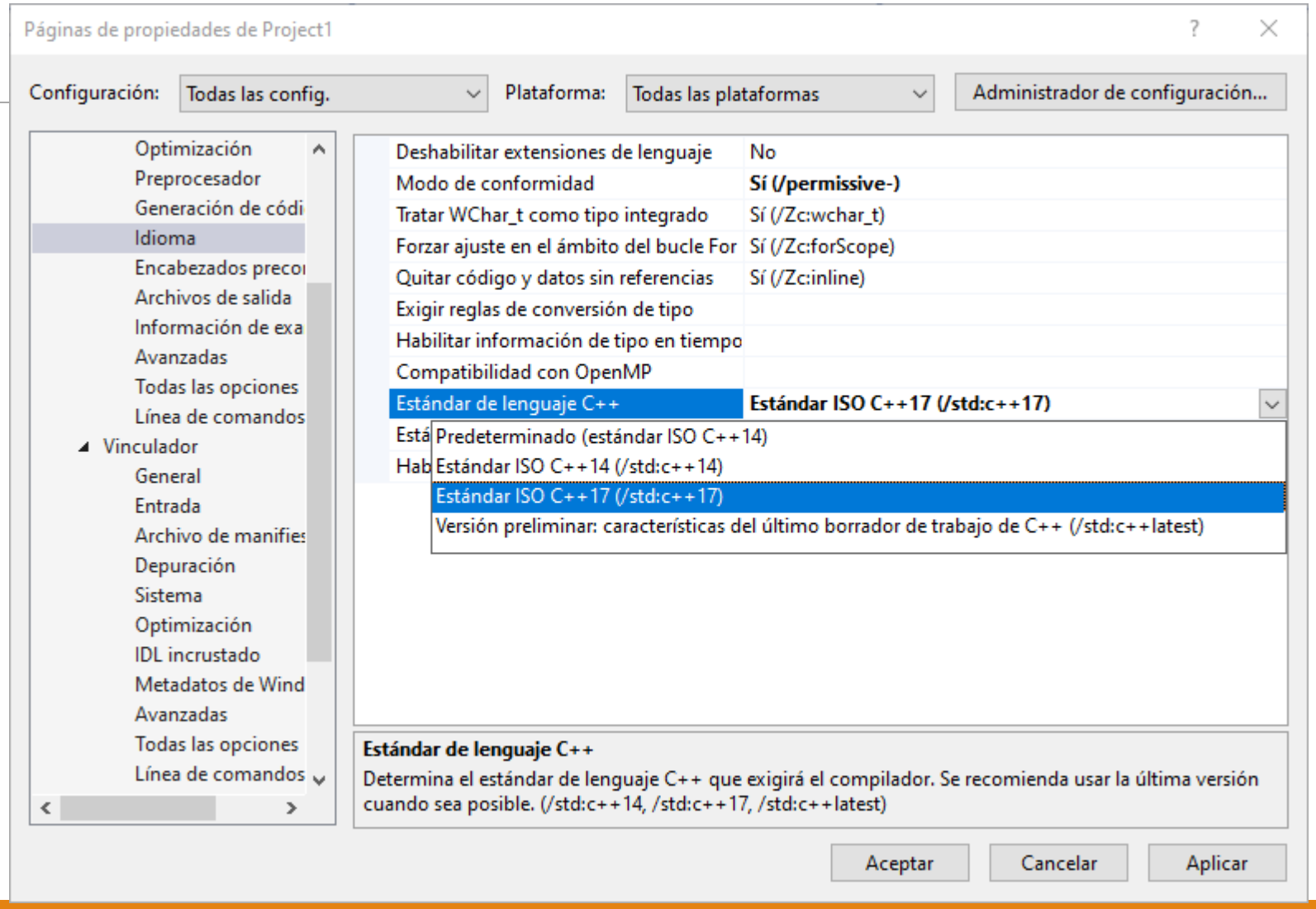
3. Creación de un proyecto de aplicación gráfica

- Vinculador: Entrada
- → Dependencias adicionales
- → (Editar):
  - opengl32.lib
  - glew32.lib
  - glfw3.lib
  - FreeImage.lib



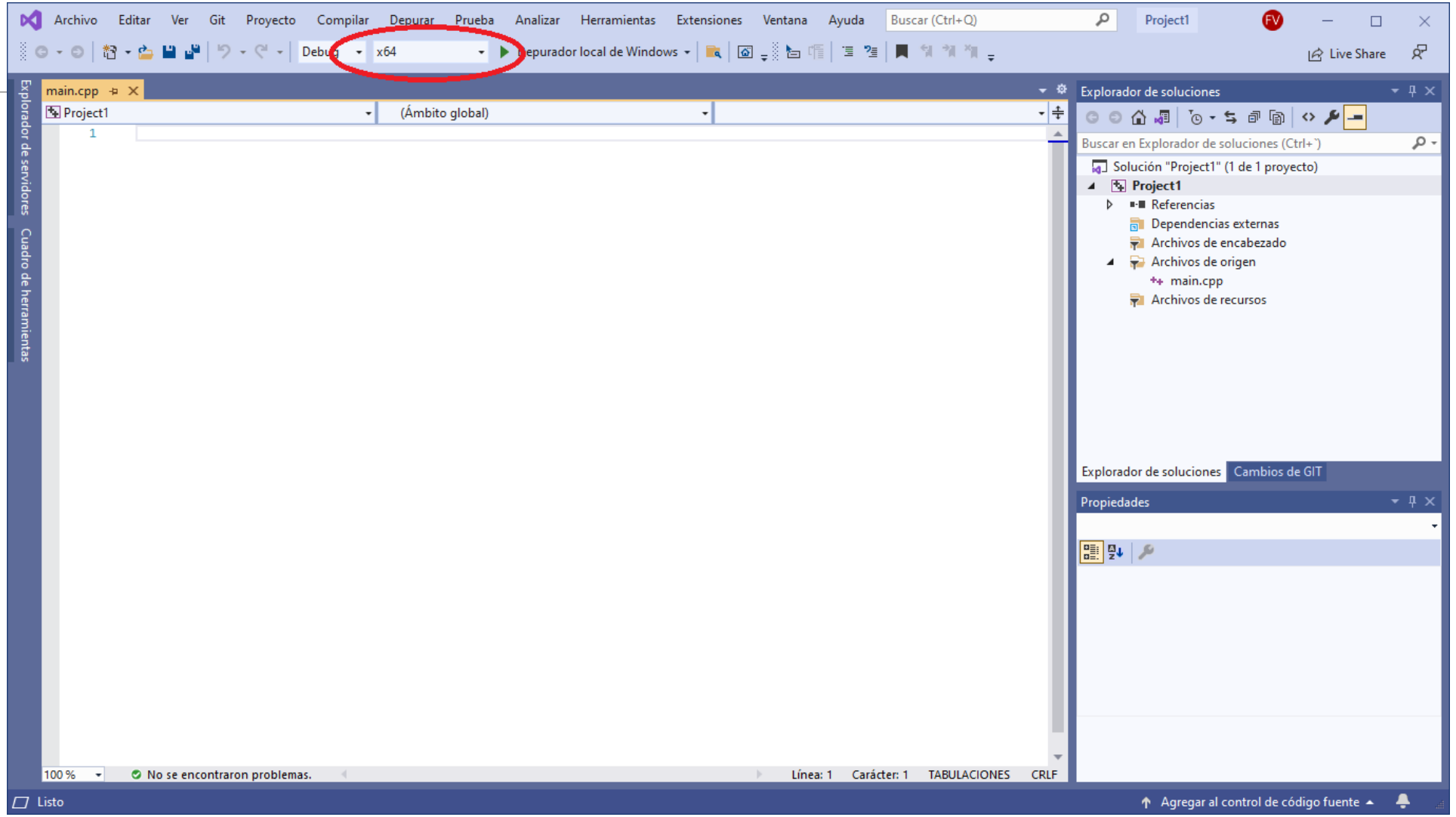
3. Creación de un proyecto de aplicación gráfica

- C/C++: Idioma
- → Estándar de lenguaje C++
- → (Seleccionar):
  - Estándar ISO C++ 17



3. Creación de un proyecto de aplicación gráfica

- Debug:
  - x64



4. La biblioteca GLEW

- Pasarela para utilizar las funciones OpenGL adaptándose a la versión utilizada en tiempo de ejecución, de forma transparente para el programador.
- Para utilizar las funciones de OpenGl hay que incluir el fichero de cabecera GL\glew.h
- Para inicializar la biblioteca es necesario ejecutar el método glewInit(), tras crear la ventana y el contexto gráfico sobre el que trabajan las funciones de OpenGL
- En la distribución de las aplicaciones gráficas generadas con GLEW es necesario incluir en la distribución el fichero glew.dll, que debe encontrarse en el directorio de ejecución de la aplicación.

5. La biblioteca GLFW

- Biblioteca de código abierto dedicada a la gestión de ventanas con soporte para OpenGL, OpenGL ES y Vulkan.
- Biblioteca escrita en C adaptada a múltiples plataformas (MS-Windows, Mac-OS, X11, Wayland).
- Las funciones de la biblioteca dan soporte a múltiples ventanas, monitores y eventos de teclado, ratón, joystick e incluso mando de consolas.
- Web oficial de la biblioteca: https://www.glfw.org/
- Podemos encontrar información así cómo la zona de descargas.
- Para las prácticas se incluye en el directorio \ComputerGraphics\Tools

5. La biblioteca GLFW

- glfwInit() primera función a utilizar para comenzar la ejecución de GLFW.
- glfwTerminate() al terminar la aplicación, libera todos los recursos que puedan haberser creado.
- GLFWwindow estructura en la biblioteca para describir ventanas.
  - glfwCreateWindow() crea una ventana
  - glfwDestroyWindow() destruye la ventana
- glfwMakeContextCurrent() para que las funciones de OpenGL utilicen el contexto gráfico de la ventana.

5. La biblioteca GLFW

- Las respuestas a los distintos eventos se configuran por medio de punteros a funciones:
  - Respuesta al evento de modificación del tamaño de ventana: glfwSetFramebufferSizeCallback().
  - Respuesta a los eventos del teclado: glfwSetKeyCallback().
  - Respuesta a los movimientos del ratón: glfwSetCursorPosCallback().
  - Respuesta a los botones del ratón: glfwSetMouseButtonCallback().
- Las funciones de respuesta a eventos deben poder acceder a la información de la aplicación que ha creado la ventana. Se usa: glfwSetWindowUserPointer(). Permite almacenar la referencia a un objeto de cualquier tipo. Así podemos acceder posteriormente a un objeto a través de la función: glfwGetWindowUserPointer().

5. La biblioteca GLFW

- La biblioteca GLFW permite también utilizar ventanas a pantalla completa, mediante glfwSetWindowMonitor(), en la que se debe incluir referencia al monitor, el modo de vídeo utilizado, la posición y el tamaño de la ventana.
  - El monitor se obtiene por: glfwGetPrimaryMonitor().
  - El modo de vídeo: glfwGetVideoMode().
  - El tamaño: glfwGetWindowSize()
  - La posición: glfwGetWindowPos()
- Para usar una ventana normal se usa glfwSetWindowMonitor(), en la que los parámetros de monitor y modo de vídeo están a nulo, y se asignan valores por defecto.
- Para crear la ventana a pantalla completa hay que indicar tanto el monitor como el modo de vídeo, así cómo el tamaño de la pantalla.

6. Generación de la aplicación gráfica

- Primer paso: función principal incluida en el fichero main.cpp.

```
#include "CGApplication.h"
#include <iostream>
#include <stdexcept>
int main()
{
   CGApplication app;
  try
  {
    app.run();
  }
  catch (const std::exception& e)
  {
    std::cerr << e.what() << std::endl;
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

6. Generación de la aplicación gráfica

- El código a desarrollar se basa en dos clases: la clase CGApplication y la clase CGModel.
- CGApplication: clase principal de la aplicación. El método run() ejecutará la aplicación completa.
  - Los miembros privados de esta clase contienen los métodos y campos necesarios para configurar el proceso de representación gráfica sobre una ventana creada con GLFW.
- CGModel: desarrolla el modelo gráfico.
  - Contiene métodos para inicializar el modelo, actualizarlo y modificarlo como respuestas a los eventos de teclado o ratón.

7. La clase GCApplication

- CGApplication: encargada de crear y gestionar la ventana sobre la que se generan los gráficos.
- Campos window y model almacenan la referencia a la ventana y al modelo gráfico.
- Campos *windowWidth*, *windowHeight*, *windowXpos* y *windowYpos* indican el tamaño y la posición de la ventana.
- *fullScreen* indica si la ventana se está mostrando a pantalla completa.
- *limitFPS* contiene la frecuencia de muestreo utilizada (*frames per second*).
- *lastTime* y *deltaTime* se usan para generar esa frecuencia de muestreo.

7. La clase GCApplication

- Declaración de la clase: CGApplication.h

```
#pragma once
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include "CGModel.h"
class CGApplication {
public:
  void run();
private:
  GLFWwindow* window;
  int windowWidth;
  int windowHeight;
  int windowXpos;
  int windowYpos;
  bool fullScreen;
  CGModel model;
  double limitFPS;
  double lastTime;
  double deltaTime;

  void initWindow();
  void initOpenGL();
  void initModel();
  void mainLoop();
  void timing();
  void cleanup();
  void swapFullScreen();

  static void keyCallback(GLFWwindow*
window, int key, int scan, int act, int mods);
  static void
mouseButtonCallback(GLFWwindow*
window, int bt, int action, int mods);
  static void
cursorPositionCallback(GLFWwindow*
window, double xpos, double ypos);
  static void
framebufferResizeCallback(GLFWwindow*
window, int width, int height);
};
```

7. La clase GCApplication

- El método run() encargado de ejecutar la aplicación. Inicializa la aplicación y se queda en un bucle de gestión de eventos de la ventana hasta recibir la orden de cierre de la aplicación.
  - initWindow(): encargado de crear y configurar la ventana por medio de las funciones de la biblioteca GLFW.
  - initOpenGL(): inicializa la biblioteca GLEW para comenzar a utilizar las funciones de OpenGL.
  - initModel(): se encarga de inicializar los campos de control de muestreo y de crear el modelo gráfico.
  - mainLoop(): bucle principal de aplicación. Consiste en un bucle infinito encargado de recibir y gestionar los eventos de la ventana y actualizar y mostrar la representación gráfica.
  - timing(): encargado de la temporización del modelo gráfico. Actualización del modelo en la tasa de muestreo indicada en limitFPS y lanza la representación del modelo gráfico.
  - cleanup(): se ejecuta al cierre de la aplicación y consiste en liberar el modelo gráfico y la ventana de ejecución.
  - swapFullScreen(): cambia el modo entre pantalla completa o ventana normal (pulsando F12).

7. La clase GCApplication

- Métodos Callback(): encargados de dar respuesta a los diferentes eventos de la ventana.
- En la ventana se almacena una referencia al objeto CGApplication que se puede obtener mediante la función glfwGetWindowUserPointer().
- Código en CGApplication.cpp

```
void CGApplication::run()
{
  initWindow();
  initOpenGL();
  initModel();
  mainLoop();
  cleanup();
}
```

7. La clase GCApplication

```
void CGApplication::initWindow()
{
  glfwInit();

  windowWidth = 800;
  windowHeight = 600;
  fullScreen = false;

  window = glfwCreateWindow(windowWidth, windowHeight, "Computer Graphics",
                            nullptr, nullptr);

glfwSetWindowUserPointer(window, this);

glfwSetFramebufferSizeCallback(window, framebufferResizeCallback);
  glfwSetKeyCallback(window, keyCallback);
  glfwSetCursorPosCallback(window, cursorPositionCallback);

glfwSetMouseButtonCallback(window, mouseButtonCallback);

glfwMakeContextCurrent(window);
}

void CGApplication::initOpenGL()
{
  glewInit();
}

void CGApplication::initModel()
{
  limitFPS = 1.0 / 60.0;
  lastTime = glfwGetTime();
  deltaTime = 0;

  int width, height;
  glfwGetFramebufferSize(window, &width, &height);
  model.initialize(width, height);
}
```

7. La clase GCApplication

```
void CGApplication::mainLoop()
{
  while (!glfwWindowShouldClose(window))
  {
    glfwPollEvents();
    timing();
    glfwSwapBuffers(window);
  }
}
```

```
void CGApplication::timing()
{
  double nowTime = glfwGetTime();
  deltaTime += (nowTime - lastTime) / limitFPS;
  lastTime = nowTime;

  while (deltaTime >= 1.0)
  {
    model.update();
    deltaTime--;
  }
  model.render();
}

void CGApplication::cleanup()
{
  model.finalize();
  glfwDestroyWindow(window);
  glfwTerminate();
}
```

7. La clase GCApplication

```
void CGApplication::swapFullScreen() {
  if (!fullScreen) {
    glfwGetWindowSize(window, &windowWidth, &windowHeight);
    glfwGetWindowPos(window, &windowXpos, &windowYpos);
    fullScreen = true;
    GLFWmonitor* monitor = glfwGetPrimaryMonitor();
    const GLFWvidmode* mode = glfwGetVideoMode(monitor);
    glfwSetWindowMonitor(window, monitor, 0, 0, mode->width, mode->height,
                                        mode->refreshRate);
}  else {
    fullScreen = false;
    glfwSetWindowMonitor(window, nullptr, windowXpos, windowYpos,
                                    windowWidth, windowHeight, nullptr);}
}
```

7. La clase GCApplication

```
void CGApplication::keyCallback(GLFWwindow* window, int key, int scancode,
                                int action, int mods)
{
  auto app = reinterpret_cast<CGApplication*>(glfwGetWindowUserPointer(window));
  if (action == GLFW_PRESS || action == GLFW_REPEAT)  {
    if (key == GLFW_KEY_F12) app->swapFullScreen();
    else app->model.key_pressed(key); }
}
void CGApplication::mouseButtonCallback(GLFWwindow* window, int button,
                                        int action, int mods)
{
  auto app = reinterpret_cast<CGApplication*>(glfwGetWindowUserPointer(window));
  app->model.mouse_button(button, action);
}
```

7. La clase GCApplication

```
void CGApplication::cursorPositionCallback(GLFWwindow* window, double xpos,
                                           double ypos)
{
  auto app = reinterpret_cast<CGApplication*>(glfwGetWindowUserPointer(window));
  app->model.mouse_move(xpos, ypos);
}
void CGApplication::framebufferResizeCallback(GLFWwindow* window, int width,
                                              int height)
{
  auto app = reinterpret_cast<CGApplication*>(glfwGetWindowUserPointer(window));
  if (height != 0) app->model.resize(width, height);
}
```

8. La clase GCModel

- CGModel: contiene la descripción del modelo gráfico.
- En las siguientes prácticas iremos modificando esta clase, para añadir la programación gráfica usando OpenGL.
- En este primer caso, vamos a desarrollar un modelo muy sencillo que tan sólo va a utilizar tres funciones básicas de OpenGL.
  - glViewport(): establece el tamaño de la imagen a generar que debe coincidir con el tamaño del área de dibujo de la ventana.
  - glClear(): borra toda la imagen utilizando un color de borrado:
  - glClearColor(): asigna el color de borrado

8. La clase GCModel

- Cabecera de CGModel.h

```
#pragma once

#include <GL/glew.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);
};
```

8. La clase GCModel

- initialize(): asigna el color blanco al fondo y llama a resize()
- resize(): se encarga de establecer el tamaño de la imagen
- render(): lanza el proceso de dibujo y se limita a limpiar la imagen con el color de fondo establecido.
- key_pressed(): respuesta a los eventos de teclado y analiza la tecla pulsada para modificar el color de fondo.
- Resto de métodos no tienen contenido.

8. La clase GCModel

- Código en CGModel.cpp

```
#include "CGModel.h"
#include <GLFW\glfw3.h>
#include <iostream>

void CGModel::initialize(int w, int h)
{
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
  resize(w, h);
}

void CGModel::finalize()
{
}

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

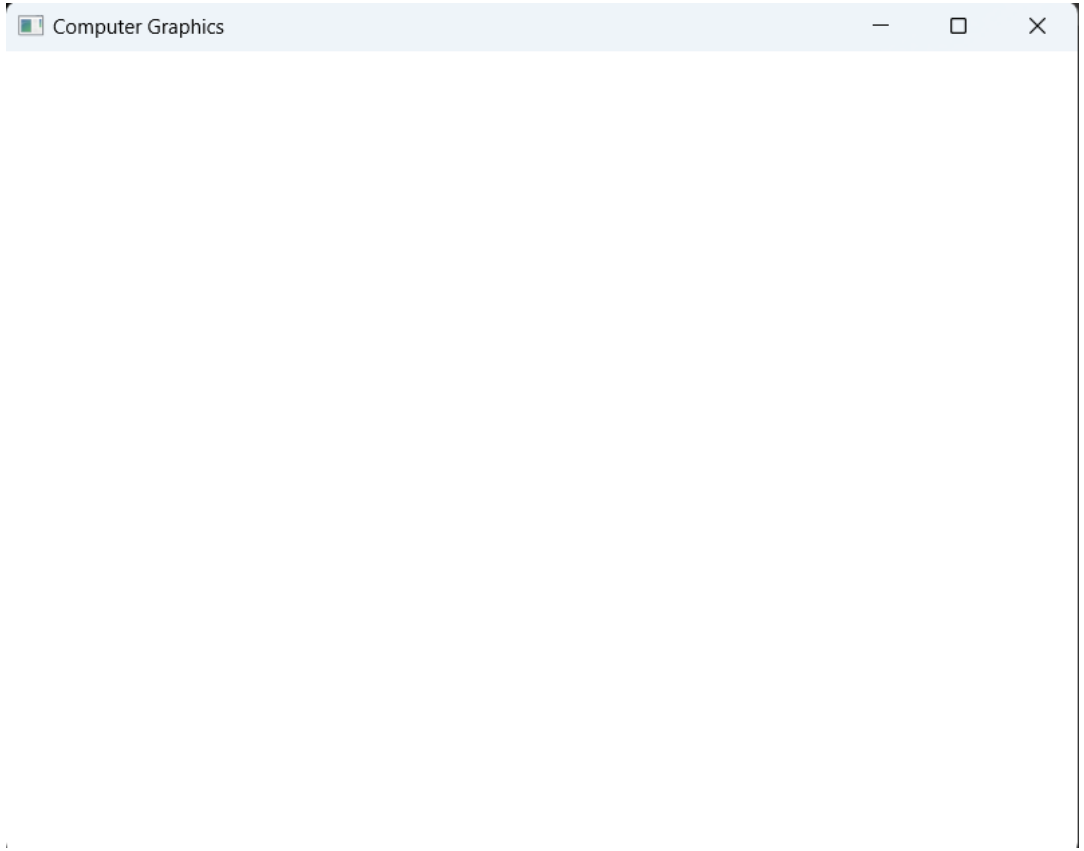
8. La clase GCModel

```
void CGModel::render()
{
  glClear(GL_COLOR_BUFFER_BIT);
}
void CGModel::update() {}

void CGModel::key_pressed(int key)
{
  switch (key)
  {
  case GLFW_KEY_R:
    glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
    break;
  case GLFW_KEY_G:
    glClearColor(0.0f, 1.0f, 0.0f, 1.0f);
    break;
  case GLFW_KEY_B:
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    break;
  case GLFW_KEY_W:
    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
    break;
  }
}
```

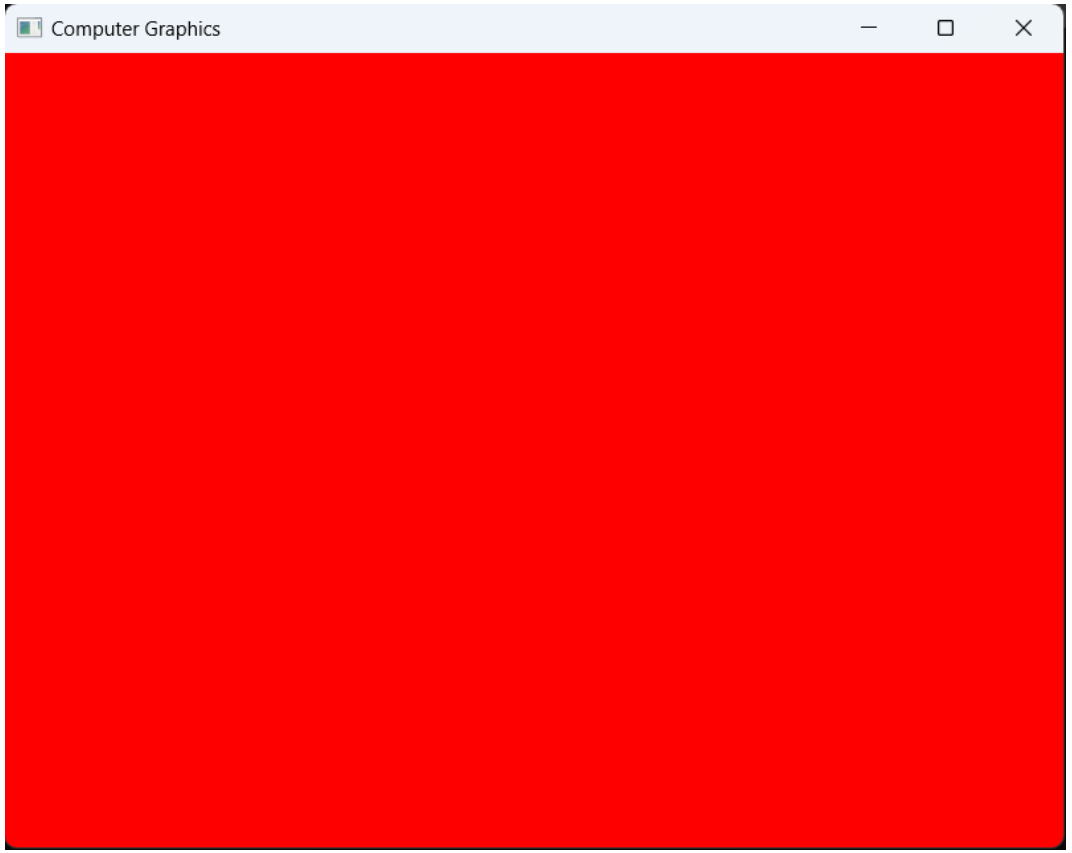
8. La clase GCModel

```
void CGModel::mouse_button(int button, int action)
{
}

void CGModel::mouse_move(double xpos, double ypos)
{
}
```

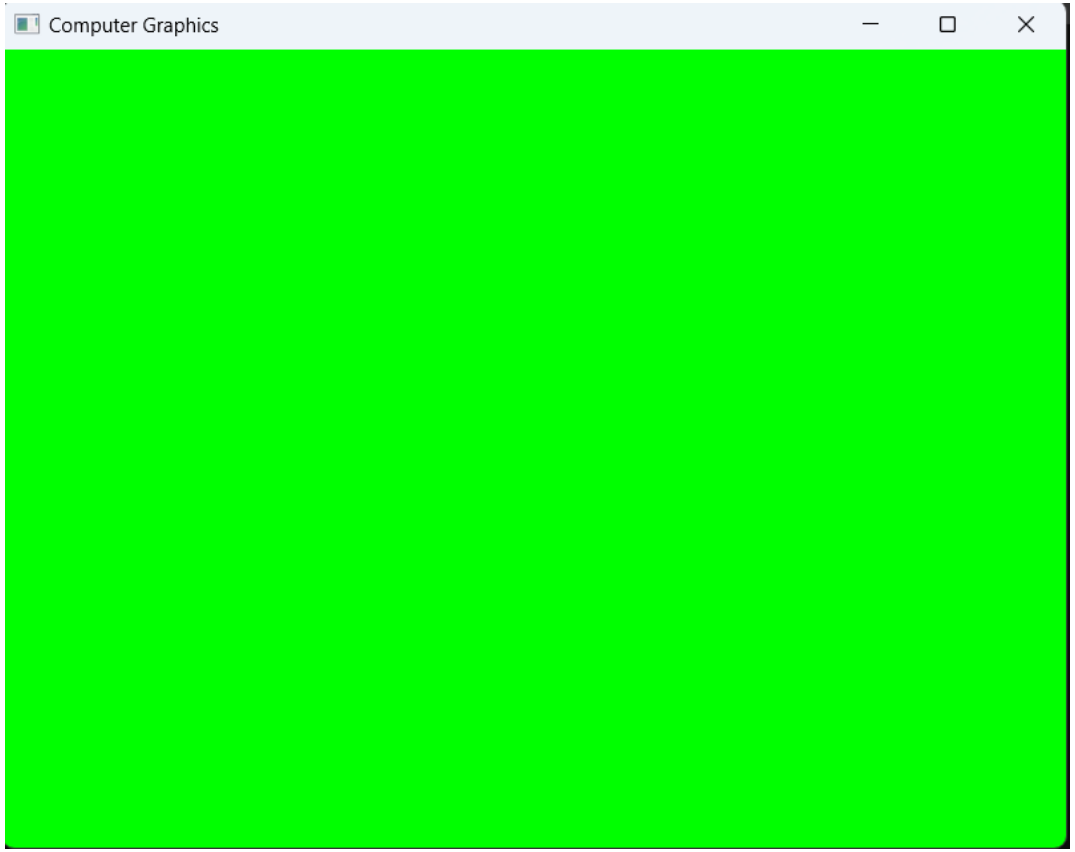
9. Aspecto Final: Arranque aplicación



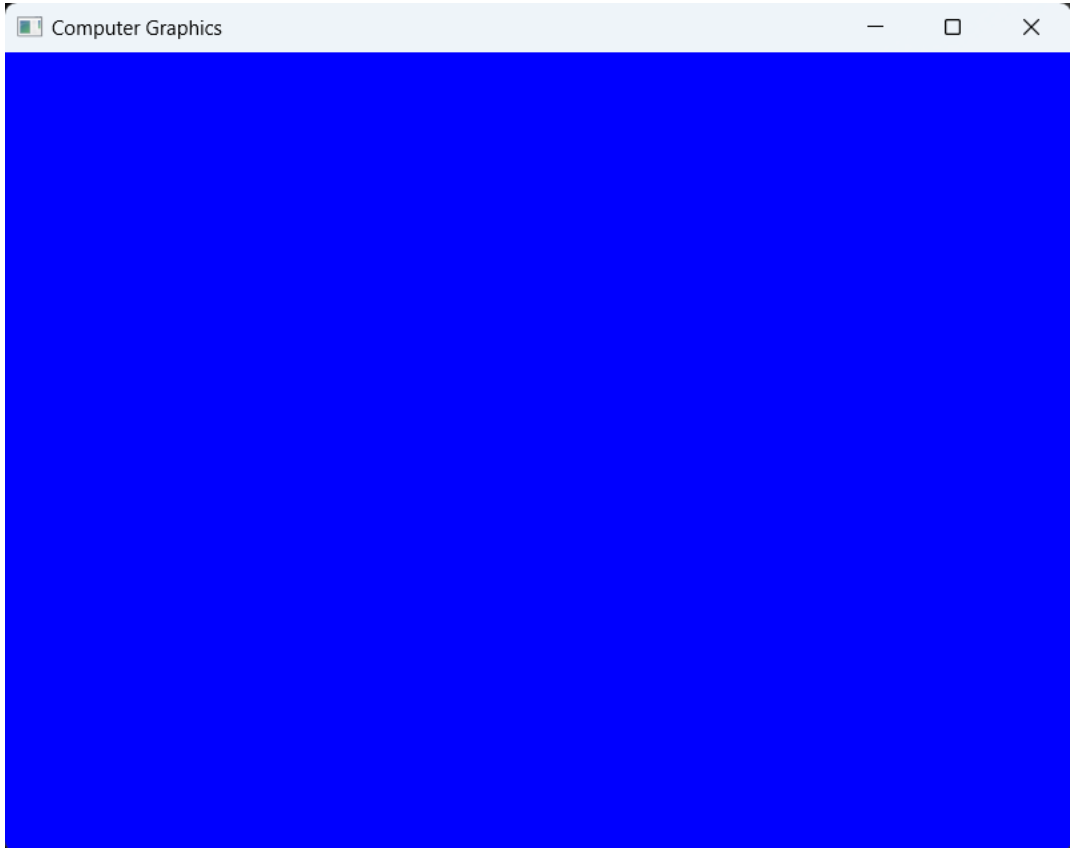
9. Aspecto Final: Pulsando tecla “r”



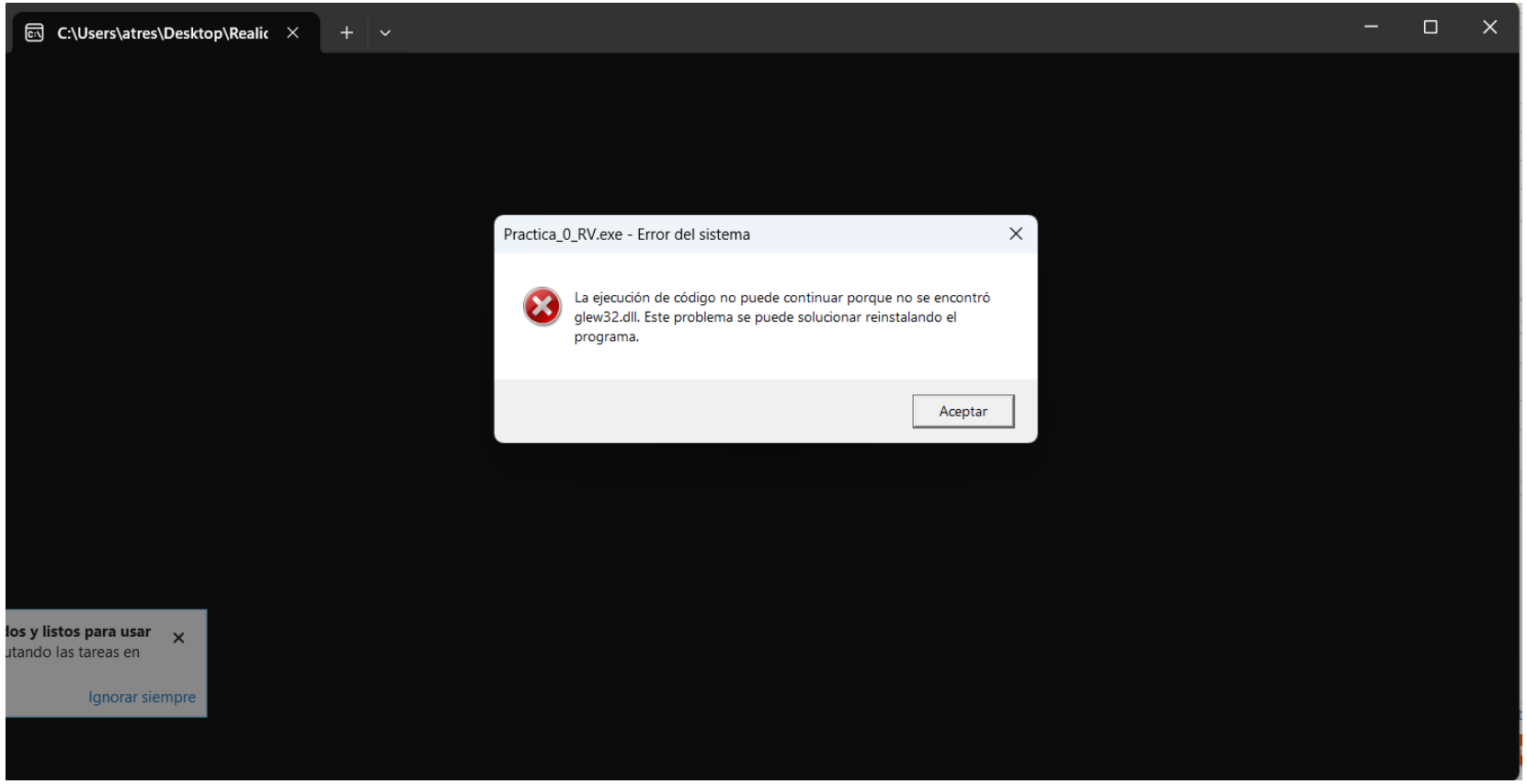
9. Aspecto Final: Pulsando tecla “g”



9. Aspecto Final: Pulsando tecla “b”



10. Si error glew32.dll



10. Si error glew32.dll, poner ruta glew32 y reiniciar visual studio 2019