Ingeniería en Informática

Trabajo Profesional

Richard: una Game Engine con raíces académicas
Manual de usuario

Alumna: Milagros Virginia Cruz
Padrón: 101228
Email: mvcruz@fi.uba.ar

Tutor
Prof. Dr. Diego Corsi

Índice

1. Introducción **2**

2. Setup del ambiente **3**
2.1. Configuración del motor 5
2.2. Configuración del cliente 8

3. Proyecto base **12**

4. Input **15**

5. Renderizado **16**
5.1. Mesh 16
5.2. Shader 16
5.3. Texturas 16
5.4. Rendercommands 16
5.5. Ejemplos 16
5.5.1. Ejemplo de renderizado de cuadrado 16
5.5.2. Ejemplo de renderizado de una imagen 18
5.5.3. Ejemplo de renderizado de dos imágenes superpuestas 20

6. Operaciones matemáticas **24**

7. Físicas **25**

1. Introducción

Richard es un entorno de desarrollo que proporciona herramientas para la creación de videojuegos 2D. Su función es facilitarle al usuario la renderización de gráficos, la lectura de input y el uso de un sistema que simule las leyes de la física y detección de colisiones. Para ello, cuenta con:

- Un bucle de juego.
- Un sistema de renderización.
- Un sistema de físicas.
- Un sistema de lectura de input.
- Un sismtema de logging.

Por otra parte, también define una interfaz *Application*, que contiene los métodos mínimos que el usuario debe implementar para poder crear un juego. Si bien el motor ofrece la base, queda en el cliente definir qué objetos van a estar, cómo van a interactuar entre ellos, cómo van a ser renderizados, etc.

En este un manual de usuario se explicará detalladamente:

- Requerimientos necesarios para correr Richard.
- Cómo acceder al proyecto y preparar el ambiente para su uso.
- Cómo crear un nuevo proyecto compatible con Richard.
- Cómo renderizar objetos e imágenes.
- Cómo utilizar el sistema de lectura de input.
- Cómo utilizar el sistema de físicas.

Además, se proveerá de bibliografía que pueda resultarle útil al usuario a modo de recurso adicional para una mayor comprensión de los recursos que tiene y provee Richard y así, facilitar el desarrollo.

2. Setup del ambiente

Lo primero a realizar es descargar el repositorio, el cual puede ser encontrado https://github.com/Mivircruz/Richard
La branch master contiene la última versión estable del motor, por lo que es esta la que se debe descargar.

Para poder desarrollar un juego, se debe crear una solución dentro de un proyecto de Visual Studio que contenga a Richard. Este nuevo proyecto se deberá settear como start project, ya que será el encargado de producir el ejecutable utilizando el motor.

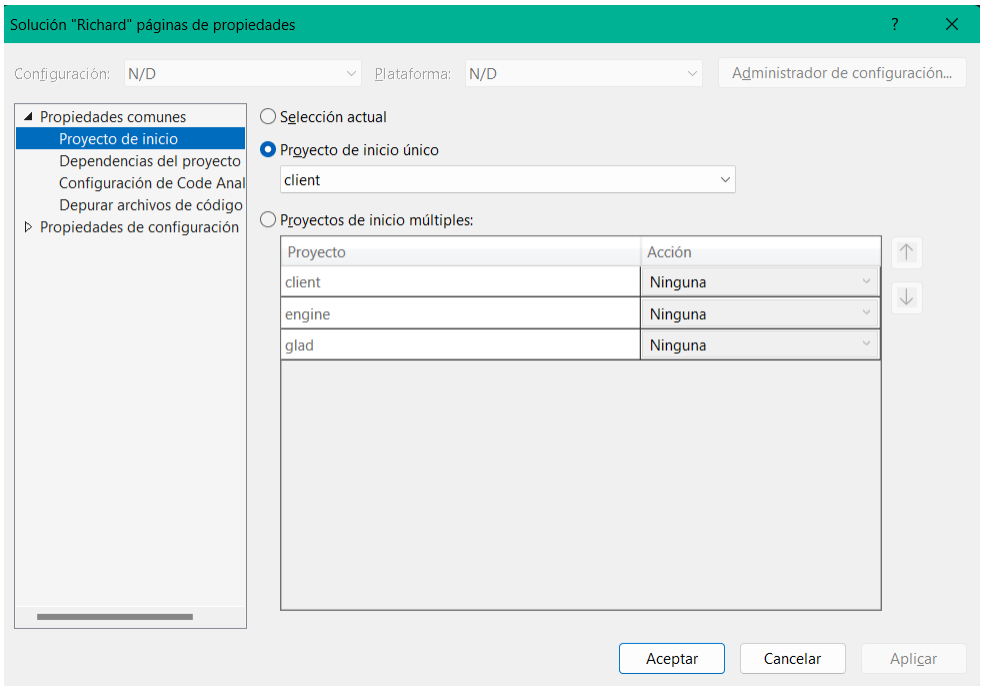

Figura 1: Proyecto de inicio

A su vez, es necesario indicarle al proyecto que la solución cliente depende de Richard. De esta forma, se compilará al motor previo a iniciar la compilación de la aplicación que lo usará.

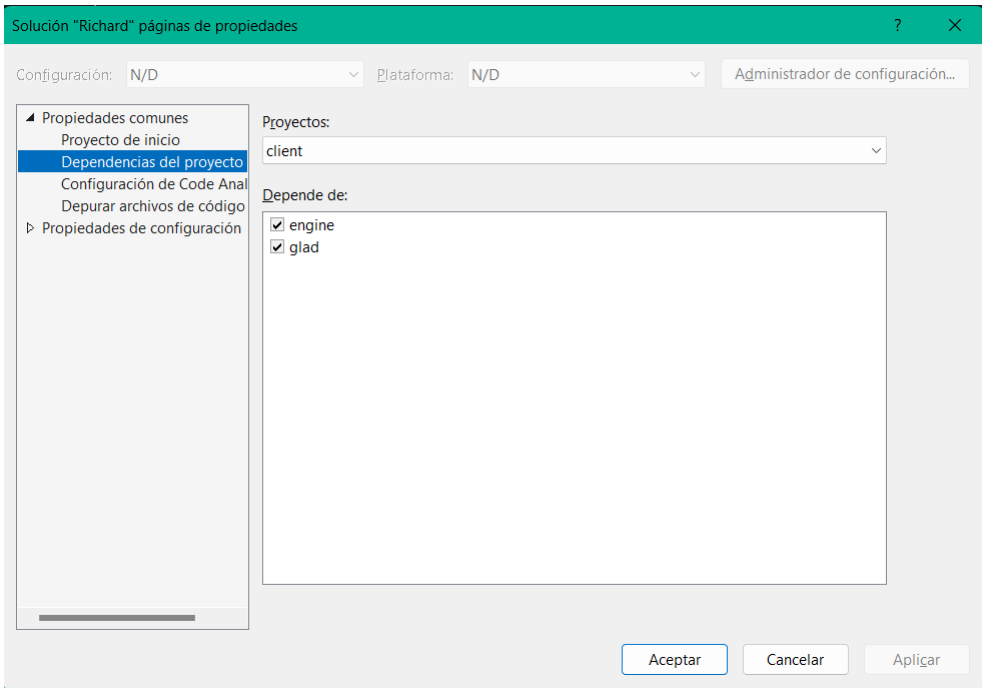

Figura 2: Dependencias del proyecto

Además, es necesario configurar ambas soluciones.

2.1. Configuración del motor

En la solución Richard se puede configurar, opcionalmente, los directorios de salida. Esto se puede modificar en la sección general de las propiedades de la solución.

Además, se debe configurar el output como .lib, ya que va a ser utilizado por la aplicación cliente que sí generará un ejecutable.

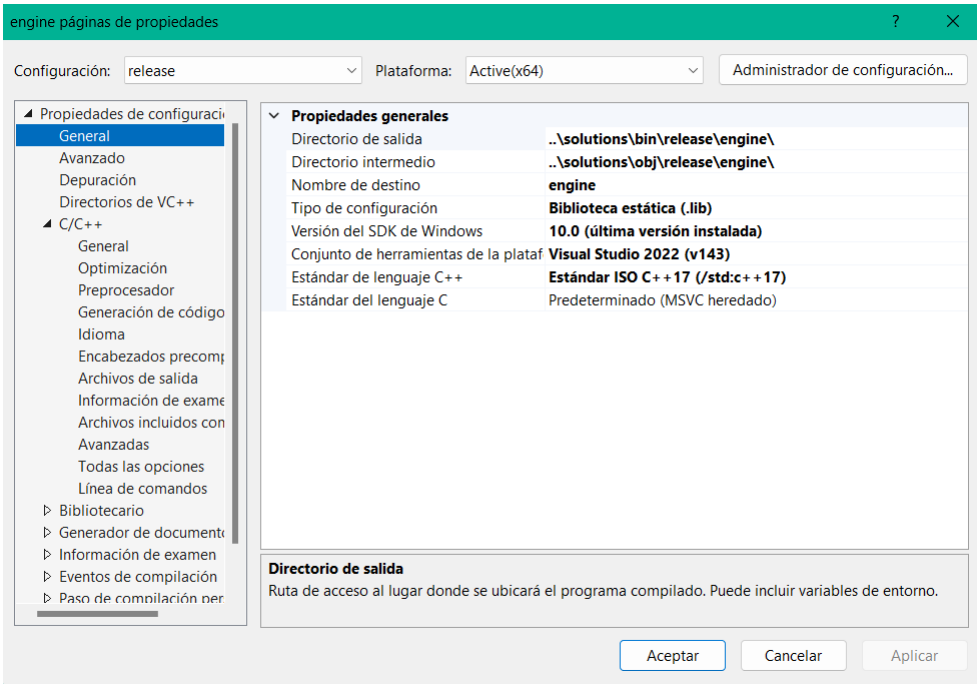

Figura 3: Propiedades generales del motor

Es obligatorio hacer visibles las dependencias externas para la engine Richard. En otras palabras, se debe incluir el directorio que incluye las dependencias externas, llamado dependencies. Esto se puede lograr cambiando los directorios de inclusión externos.

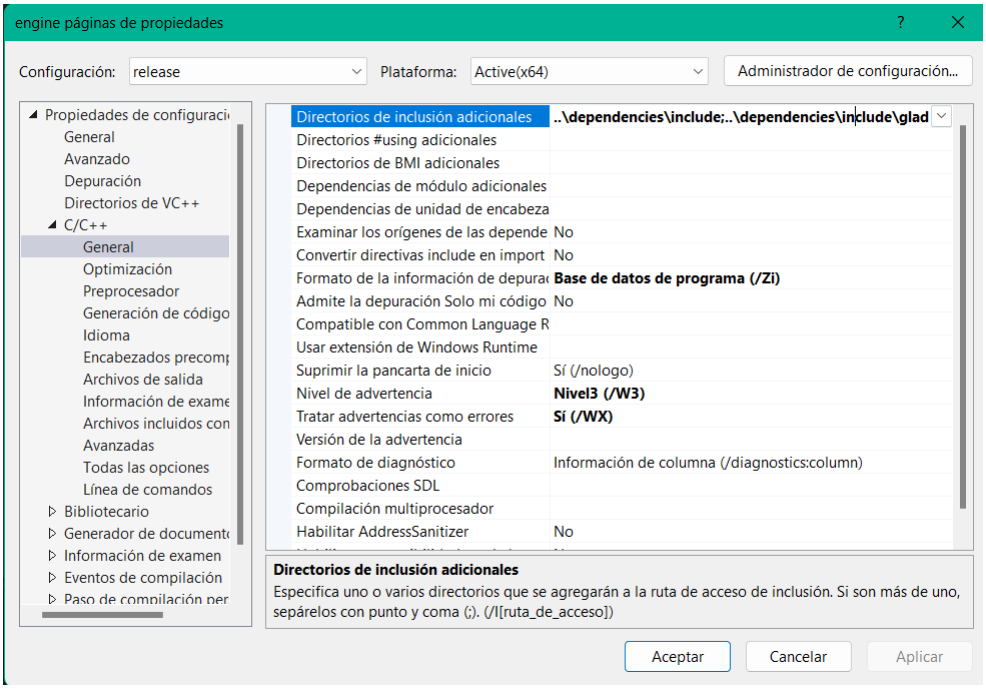

Figura 4: Directorio de inclusión externos de la solución Richard

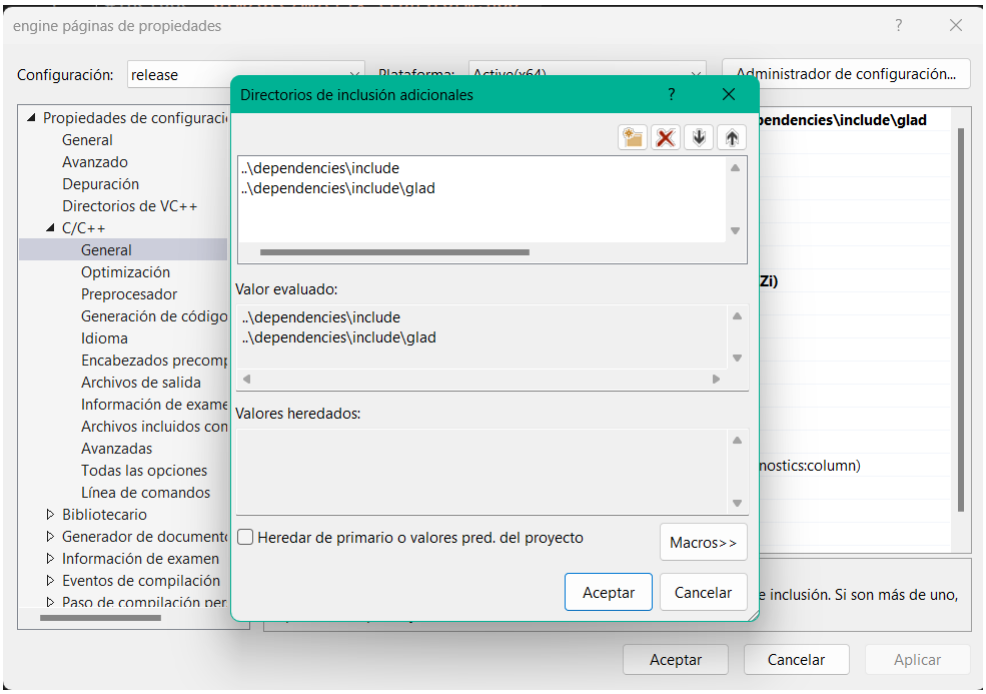

Figura 5: Directorio de inclusión externos de la solución Richard - Detalle

Finalmente, se debe seleccionar el tipo DLL multiproceso como biblioteca de ejecución para que sea coherente con la ejecución de las dependencias externas. Esto se puede cambiar en la sección Generación de código.

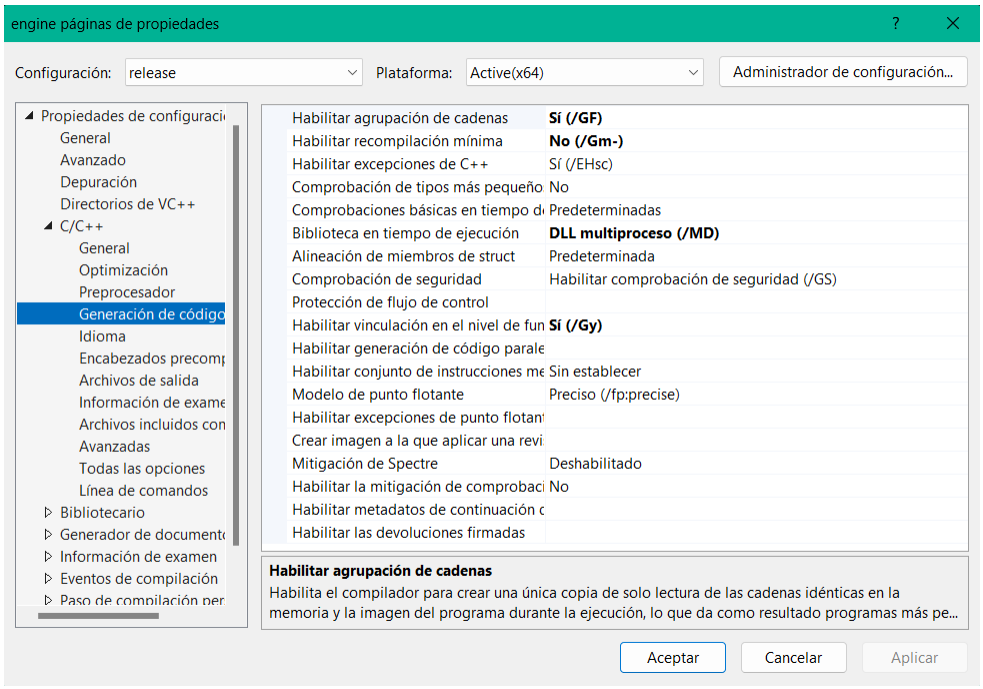

Figura 6: Configuración de la biblioteca de ejecución de Richard

2.2. Configuración del cliente

En la solución cliente se puede configurar, opcionalmente, los directorios de salida. Esto se puede modificar en la sección general de las propiedades de la solución.

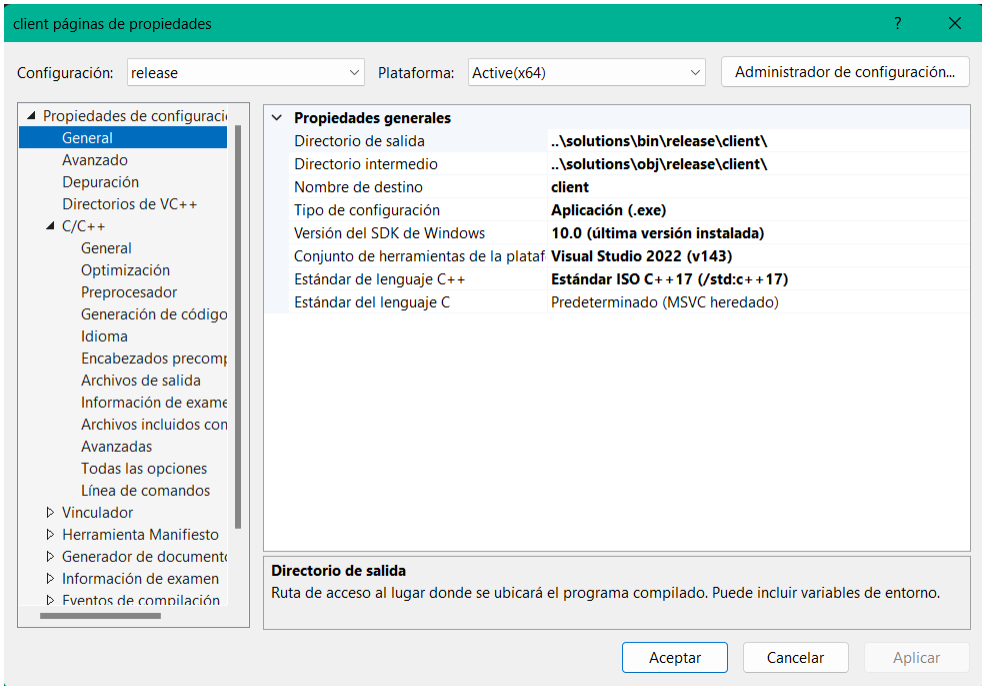

Figura 7: Propiedades generales de la solución cliente

Es obligatorio hacer visibles las dependencias de Richard para el cliente. Es decir, se deben incluir tanto el directorio que incluye las dependencias externas, llamado dependencies, como también el directorio que contiene el código del motor. Esto se puede lograr cambiando los directorios de inclusión externos.

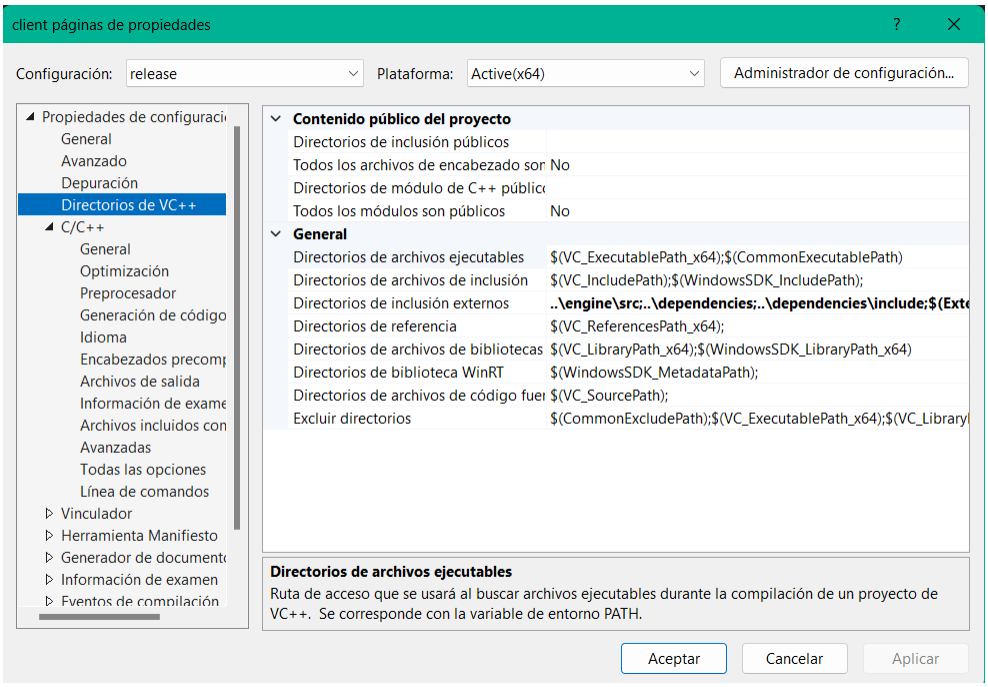

Figura 8: Directorio de inclusión externos de la solución cliente

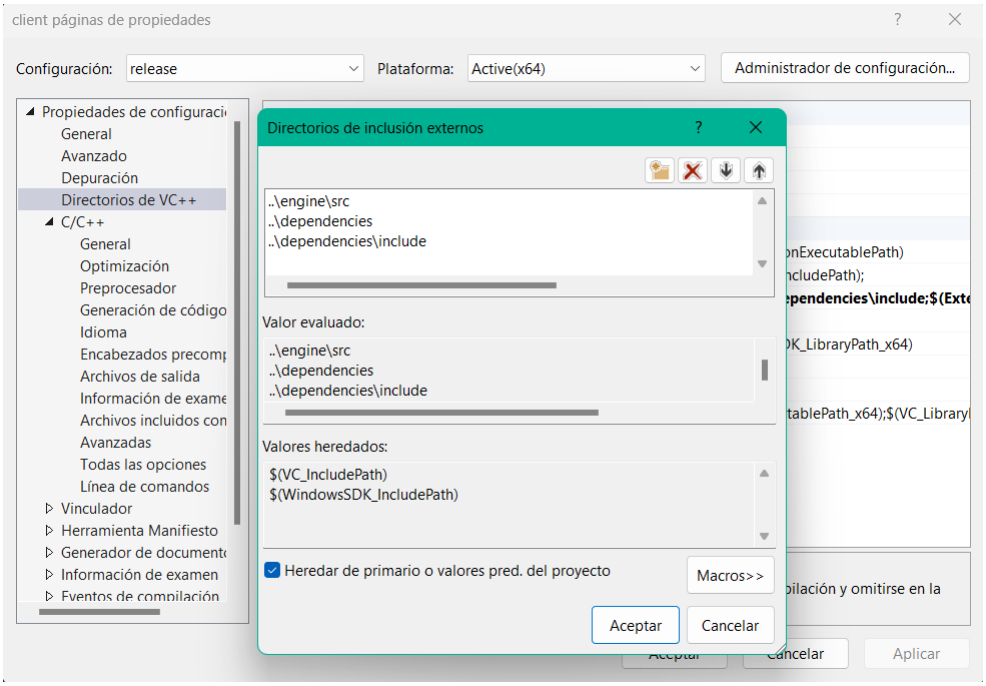

Figura 9: Directorio de inclusión externos de la solución cliente - Detalle

Se debe incluir particularmente GLM. Para ello, dentro de la sección General de las propiedades de C/C++ de la solución, se debe editar los directorios de inclusión adicionales y agregar la dependencia.

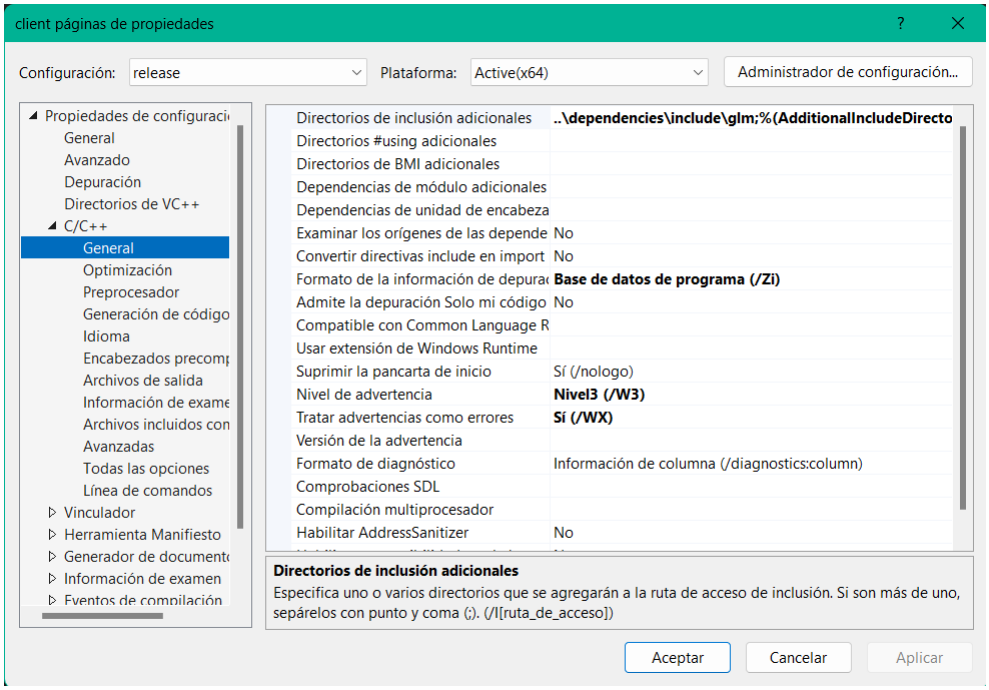

Figura 10: Directorios de inclusión adicionales

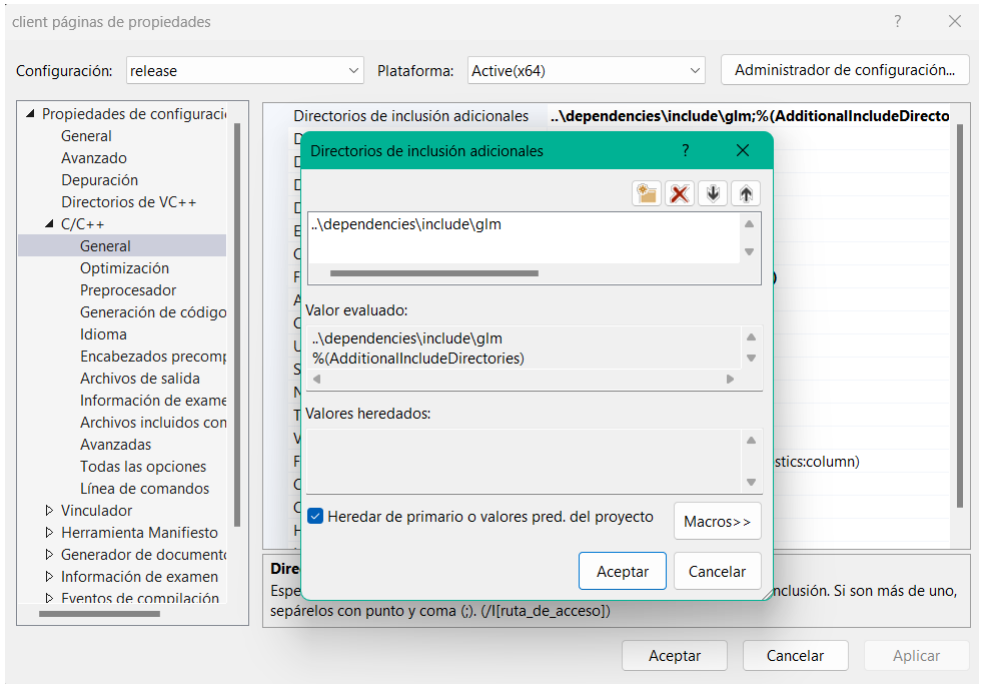

Figura 11: Directorios de inclusión adicionales - Detalle

GLFW es una dependencia particular ya que se debe incluir su ejecutable. Para ello, se debe agregar la carpeta dependencies/lib en el directorio de bibliotecas adicionales del linker.

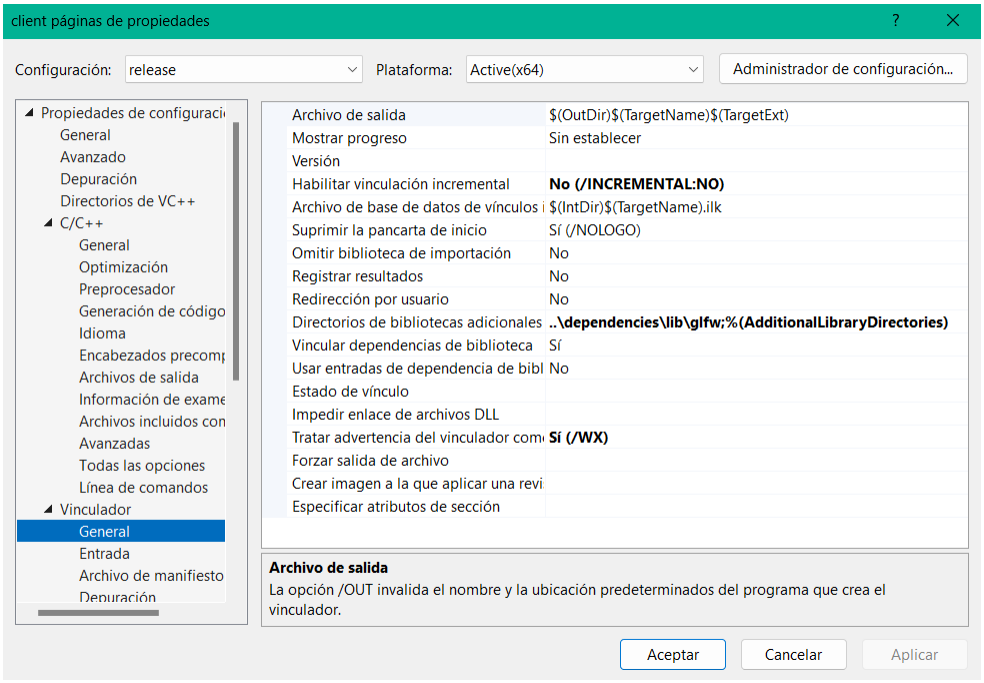

Figura 12: Directorios de bibliotecas adicionales

3. Proyecto base

Una vez que el ambiente está listo, para poder obtener un proyecto base se debe implementar la interfaz Application.h. Esta define los métodos que debe tener, como mínimo, un programa que vaya a utilizar Richard.

```
class Application {
        public:
            /*Methods*/
            Application() {}
            ~Application() {}

            virtual void Initialize() {}
            virtual void Shutdown() {}

            virtual void Update() {}
            virtual void Render() {}
    };
```

Además, se debe de implementar un método para crear la aplicación que tenga la siguiente firma:

```
Application* CreateApplication();
```

Esto se debe a que Richard define un main en el archivo *main.h* en donde se instancia a la aplicación y se pone a correr la engine:

```
int main(void)
{
    Application* app = CreateApplication();
    Engine::GetInstance()->Run(app);
    delete app;
    return 0;
}
```

Teniendo estos métodos y sin ningún tipo de código adicional, se puede generar un ejecutable que, al correr, muestra una ventana con el color default: negro.

Se puede modificar tanto el tamaño inicial de la ventana como su color. Cabe aclarar que una vez renderizada la ventana, se la puede resizear.

Ejemplo de código en donde se cambia de color a la ventana:

```
#include "client/application.h"
#include "core/engine.h"

class ClientApp : public Richard::Application {
public:
    void Initialize() override {
      Engine::GetInstance()->GetRenderer()->SetClearColor(158/255.f, 0.f, 0.f, 1.f);
    }

    void Shutdown() override {
    }

    void Update() override {
    }
```

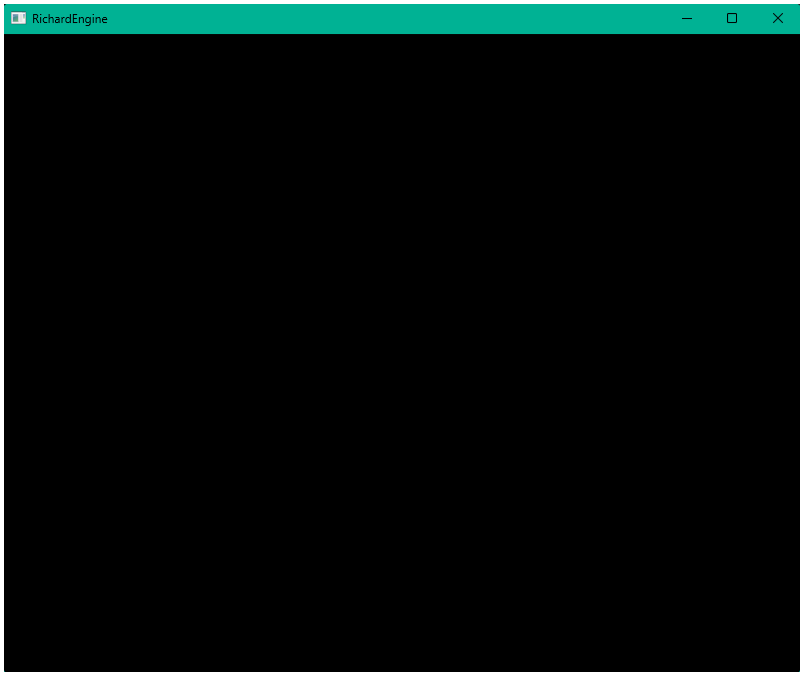

Figura 13: Ventana de base.

```
    void Render() override {
    }
};

Richard::Application* CreateApplication() {
    return new ClientApp();
}
```

Ejemplo de código en donde se cambia el tamaño inicial de la ventana.

```
#include "client/application.h"
#include "core/engine.h"

class ClientApp : public Richard::Application {
public:

    ClientApp {
        Engine::GetInstance()->GetWindow()->SetSize(600, 600);
    }

    void Initialize() override {
    }

    void Shutdown() override {
    }

    void Update() override {
    }
```

```
    void Render() override {
    }
};

Richard::Application* CreateApplication() {
    return new ClientApp();
}
```

4. Input

Richard cuenta con dos clases para el manejo del input **Mouse** y **Keyboard**. Con ellas se puede saber si una tecla/botón está siendo presionada/o o no. En el caso particular del mouse, también se puede obtener la posición del cursor.

Para ser utilizado, sólo se debe importar los headers **mouse.h** y **keyboard.h** respectivamente dentro de la carpeta **input** y ya se pueden utilizar los métodos mencionados.

En el caso de Mouse, los métodos disponibles son:

```
static std::pair<float, float> GetPosition();

static bool IsButtonPressed(int buttonName);
```

Es importante destacar que el segundo método recibe un int que representa un botón del mouse. Estos enteros están definidos como constantes dentro del header.

En el caso de Keyboard, el método disponibles es:

```
static bool IsKeyPressed(int keyName);
```

De forma análoga al mouse, el método recibe un int que representa una tecla y los mismos se encuentran definidos como constantes en el header.

Por último, cabe mencionar que todos los métodos son estáticos, por lo que no hace falta instanciar la clase para poder usarlos.

5. Renderizado

Para poder renderizar objetos, se necesita de un **Mesh** y de un **Shader**.

5.1. Mesh

El Mesh es el encargado de inicializar los VAO, VBO y EBO. Tiene varios constuctores y debe recibir, como mínimo, un array de posiciones con su respectivo tamaño y dimensión.

También se le puede agregar vértices que tengan información de colores y texturas como así también un array de índices (elementos)

5.2. Shader

La clase Shader encapsula los dos shaders necesarios para renderizar imágenes utilizando OpenGL. Para poder crear un shader de Richard, se debe proporcionar tanto el vertex shader como el fragment shader utilizando GLSL. Para ello, se deben crear dos archivos con los códigos correspondientes y pasar la ruta a dichos archivos como argumento.

5.3. Texturas

Las texturas son necesarias para renderizar imágenes más no así para renderizar figuras geométricas. Para crear una textura se cuenta con la clase **Texture** y se le debe proporcionar una ruta a una imagen.

Un feature interesante es que se pueden crear varias texturas y superponerlas entre sí. Además, utilizando la dependencia GLM, que ya viene integrada con Richard, se pueden aplicar transformaciones para modificar las imágenes.

5.4. Rendercommands

Para poder renderizar una textura y/o un mesh, hace falta indicarle al Render pipeline qué comandos se desea ejecutar. Para poder hacer eso, se debe crear el comando correspondiente, ya sea **RenderMesh** o **RenderTexture** y enviarlo al **Renderer** mediante el método **Submit()**. Finalmente, se debe ejecutar el pipeline con el método **Execute()**

5.5. Ejemplos

A modo informativo se proveerán algunos ejemplos.

5.5.1. Ejemplo de renderizado de cuadrado

Vertex shader

```
#version 410 core
layout (location = 0) in vec3 position;
void main() {
    gl_Position = vec4(position, 1.0);
}
```

Fragment shader

```
#version 410 core
out vec4 outColor;
void main() {
    outColor = vec4(1.0);
}
```

Main

```
    #include "client/main.h"

#include <iostream>
#include <vector>

#include "client/application.h"
#include "core/engine.h"
#include "graphics/mesh.h"
#include "graphics/rendermesh.h"
#include "graphics/shader.h"

using namespace std;

class ClientApp : public Richard::Application {
public:

    ClientApp() {
        Engine::GetInstance()->GetWindow()->SetSize(800, 600);
    }

    void Initialize() override {
        // Define the square
        // We will define two triangles that togehter make the square
        float vertices[] = {
            0.5f,   0.5f,  0.f,
            0.5f,  -0.5f,  0.f,
            -0.5f, -0.5f,  0.f,
            -0.5f,  0.5f,  0.f
        };
        uint32_t indices[]{
            0, 3, 1, // first triangle
            1, 3, 2  // second triangle
        };
        mMesh = make_shared<Richard::Graphics::Mesh>
        (&vertices[0], 4, 3, &indices[0], 6);

        mShader = make_shared<Richard::Graphics::Shader>
        ("resources/shaders/square_vs.txt", "resources/shaders/square_fs.txt");

    }

    void Shutdown() override {
    }

    void Update() override {
    }

    void Render() override {
```

```
        auto renderCommand = make_unique<Richard::Graphics::RenderMesh>
        (mMesh, mShader);
        Engine::GetInstance()->GetRenderer()->Submit(move(renderCommand));
        Engine::GetInstance()->GetRenderer()->Execute();
    }

private:
    shared_ptr<Richard::Graphics::Mesh> mMesh;
    shared_ptr<Richard::Graphics::Shader> mShader;
};

Richard::Application* CreateApplication() {
    return new ClientApp();
}
```

Fragment shader

```
#version 410 core
out vec4 outColor;
void main() {
    outColor = vec4(1.0);
}
```

5.5.2. Ejemplo de renderizado de una imagen

Vertex shader

```
#version 410 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;
out vec3 ourColor;
out vec2 TexCoord;
void main() {
gl_Position = vec4(aPos, 1.0);
ourColor = aColor;
TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Fragment shader

```
#version 410 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D texture1;
uniform sampler2D texture2;
void main() {
        FragColor = mix(texture(texture1, TexCoord),
        texture(texture2, TexCoord), 0.2);
}
```

Main

```
#include "client/main.h"

#include <iostream>
#include <vector>

#include "client/application.h"
#include "core/engine.h"
#include "graphics/mesh.h"
#include "graphics/rendertexture.h"
#include "graphics/shader.h"
#include "graphics/texture.h"

using namespace std;

class ClientApp : public Richard::Application {
public:

    ClientApp() {
        Engine::GetInstance()->GetWindow()->SetFullscreen();
    }

    void Initialize() override {
        std::cout << "ClientApp Initialize" << std::endl;

        mShader = make_shared<Richard::Graphics::Shader>
        ("resources/shaders/texture_vs.txt", "resources/shaders/texture_fs.txt");

        // Define the square
        float vertices[] = {
            // positions          // colors           // texture coords
             0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,
             0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,
            -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,
            -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f
        };

        uint32_t indices[]{
            0, 1, 3, // first triangle
            1, 2, 3  // second triangle
        };

        // Define the quad that will contain the image to load
        float textureCoodinates[] = {
            1.f, 1.f,
            1.f, 0.f,
            0.f, 0.f,
            0.f, 1.f
        };
```

```
        mMesh = make_shared<Richard::Graphics::Mesh>
        (&vertices[0], 4, 3, 3, 2, &indices[0], 6);


        // Define the texture
        shared_ptr<Richard::Graphics::Texture> texture =
        make_shared<Richard::Graphics::Texture>
        ("resources/images/poro.jpg", T_COLOR_RGB);

        mTexture = texture;
        mShader->SetUniformInt("texture", 0);
    }

    void Shutdown() override {
    }

    void Update() override {
    }

    void Render() override {

        auto renderCommand = make_unique
        <Richard::Graphics::RenderTexture>(mMesh, mShader);
        renderCommand->AddTexture(mTexture);
        Engine::GetInstance()->GetRenderer()->Submit(move(renderCommand));
        Engine::GetInstance()->GetRenderer()->Execute();
    }

private:
    shared_ptr<Richard::Graphics::Mesh> mMesh;
    shared_ptr<Richard::Graphics::Shader> mShader;
    shared_ptr<Richard::Graphics::Texture> mTexture;
};

Richard::Application* CreateApplication() {
    return new ClientApp();
}
```

5.5.3. Ejemplo de renderizado de dos imágenes superpuestas

Vertex shader

```
#version 410 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;
out vec3 ourColor;
out vec2 TexCoord;
void main() {
gl_Position = vec4(aPos, 1.0);
ourColor = aColor;
```

```
TexCoord = vec2(aTexCoord.x, aTexCoord.y);
}
```

Fragment shader

```
#version 410 core
out vec4 FragColor;
in vec3 ourColor;
in vec2 TexCoord;
uniform sampler2D texture1;
uniform sampler2D texture2;
void main() {
        FragColor = mix(texture(texture1, TexCoord), texture(texture2, TexCoord), 0.2
}
```

Main

```
#include "client/main.h"

#include <iostream>
#include <vector>

#include "client/application.h"
#include "core/engine.h"
#include "graphics/mesh.h"
#include "graphics/rendertexture.h"
#include "graphics/shader.h"
#include "graphics/texture.h"

using namespace std;

class ClientApp : public Richard::Application {
public:

    ClientApp() {
        Engine::GetInstance()->GetWindow()->SetFullscreen();
    }

    void Initialize() override {
        std::cout << "ClientApp Initialize" << std::endl;

        mShader = make_shared<Richard::Graphics::Shader>
        ("resources/shaders/texture_vs.txt", "resources/shaders/texture_fs.txt");

        // Define the square
        float vertices[] = {
            // positions          // colors           // texture coords
             0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,
             0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,
            -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,
            -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f
```

```
    };

    uint32_t indices[]{
        0, 1, 3, // first triangle
        1, 2, 3  // second triangle
    };

    // Define the quad that will contain the image to load
    float textureCoodinates[] = {
        1.f, 1.f,
        1.f, 0.f,
        0.f, 0.f,
        0.f, 1.f
    };

    mMesh = make_shared<Richard::Graphics::Mesh>
    (&vertices[0], 4, 3, 3, 2, &indices[0], 6);


    // Define the textures
    shared_ptr<Richard::Graphics::Texture> texture1 =
    make_shared<Richard::Graphics::Texture>
    ("resources/images/poro.jpg", T_COLOR_RGB);

    shared_ptr<Richard::Graphics::Texture> texture2 =
    make_shared<Richard::Graphics::Texture>
    ("resources/images/pusheen.png", T_COLOR_RGBA);

    mTextures.push_back(texture1);
    mTextures.push_back(texture2);
    mShader->SetUniformInt("texture1", 0);
    mShader->SetUniformInt("texture2", 1);

    Engine::GetInstance()->GetRenderer()->SetClearColor(2.0f / 255.0f, 97.0f / 25
}

void Shutdown() override {
}

void Update() override {
}

void Render() override {

    auto renderCommand = make_unique<Richard::Graphics::RenderTexture>
    (mMesh, mShader);
    renderCommand->AddTexture(mTextures.at(0));
    renderCommand->AddTexture(mTextures.at(1));
    Engine::GetInstance()->GetRenderer()->Submit(move(renderCommand));
    Engine::GetInstance()->GetRenderer()->Execute();
```

```
    }

private:
    shared_ptr<Richard::Graphics::Mesh> mMesh;
    shared_ptr<Richard::Graphics::Shader> mShader;
    vector<shared_ptr<Richard::Graphics::Texture>> mTextures;
};

Richard::Application* CreateApplication() {
    return new ClientApp();
}
```

6. Operaciones matemáticas

En las sección anterior se vio cómo crear objetos, colorearlos y/o darles una apariencia detallada usando texturas, pero se puede seguir trabajando para modificarlos. Una forma eficiente de transformar un objeto es utilizando (múltiples) objetos matriz y modificándolas utilizando operaciones conocidas como rotación, traslación, multiplicación, etc.

Una de las dependencias externas que Richard tiene es **GLM** (OpenGL Mathematics), la cual es una biblioteca de matemática para utilizar en gráficos y justamente se puede utilizar para trabajar con matrices.

Además, la clase Shader de Richard provee métodos para pasarle al shader variables uniformes del tipo matrices. Las variables uniformes son aquellas que no cambian a menudo y que son las mismas para todos los vértices o fragmentos procesados por el shader. Se utilizan para pasar información desde la aplicación a los shaders.

Para poder usar esta biblioteca, no es necesario incluir ningún header ya que con el set up inicial la aplicación del cliente tendrá incluida esta dependencia. Lo único que hace falta es utilizar el namespace adecuado.

Si se desea consultar los detalles de lo que ofrece GLM, se puede visitar su repositorio oficial.

7. Físicas

Richard provee de una clase **Gameobject.h** que el cliente puede utilizar como clase base para que los objetos del juego la hereden. Esta clase posee los atributos que suelen tener los objetos de los juegos junto con algunos métodos ya implementados que resultan de utilidad.

Sin embargo, también posee métodos virtuales que el cliente debe implementar como Render() y Update(), puesto que estos varían dependiendo del objeto y del tipo de juego que se esté desarrollando.

Una vez creados los objetos, se deben enviar al **GameObjectManager.h**, quien es el encargado de ejecutar los métodos Render() y Update() dentro de los métodos con el mismo nombre de la engine. A modo de ejemplo, se muestra a continuación la creación de un objeto *Ball* y su adición al pipeline de manejo de físicas.

Primero se implementa el objeto, el cual hereda de la clase GameObject:

```
#include "ball.h"

#include "core/engine.h"
#include "graphics/rendermesh.h"
#include "input/keyboard.h"

// Define mesh
static float Vertices[] = {
        // positions
         0.5f,  0.5f, 0.f,     // top right
         0.5f, -0.5f, 0.f,     // bottom right
        -0.5f, -0.5f, 0.f,     // bottom left
        -0.5f,  0.5f, 0.f,     // top left
};

static uint32_t Indices[]{
        0, 3, 1, // first triangle
        1, 3, 2  // second triangle
};

Ball::Ball(std::pair<double, double> position, std::pair<double, double> size)
        : GameObject(position, size, make_pair(0.f, 0.f), nullptr, nullptr) {

        mMesh = make_shared<Graphics::Mesh>(&Vertices[0], 4, 3, &Indices[0], 6);

        mShader = make_shared<Graphics::Shader>
        ("resources/shaders/pong_vs.txt", "resources/shaders/pong_fs.txt");
}

void Ball::Render() {
        // Create identity matrix
        glm::mat4 model = glm::mat4(1.f);

        // Make a tranlsation to move the object
        model = glm::translate(model, { mPosition.first, mPosition.second, 0.f });

        // Scale the matrix to fit the screen
        model = glm::scale(model, { mSize.first, mSize.second, 1.f });
```

```
        mShader->SetUniformMat4("model", model);
        auto renderCommand = make_unique<Graphics::RenderMesh>(mMesh, mShader);
        Engine::GetInstance()->GetRenderer()->Submit(move(renderCommand));
}

void Ball::Update() {
        MoveWithConstantVelocity();
}
```

Luego, en la aplicación cliente se lo agrega al Manejador.

```
#include "client/main.h"

#include <memory>
#include <iostream>
#include <vector>

#include "client/application.h"
#include "core/engine.h"
#include "graphics/mesh.h"
#include "graphics/shader.h"
#include "input/keyboard.h"
#include "physics/gameobject.h"

#include "ball.h"

using namespace std;

class ClientApp : public Richard::Application {
public:

    void Initialize() override {
        // Ball creation
        mBall = make_shared<Ball>(make_pair(0.f, 0.f), make_pair(0.075f, 0.1f));

        // Submit object to manager
        Engine::GetInstance()->GetGameObjectManager()->Submit("ball", mBall);
    }

    void Shutdown() override {
    }

    void Update() override {
        if (Input::Keyboard::IsKeyPressed(KEY_Q)) {
            Engine::GetInstance()->Quit();
            return;
        }
```

```
    }

    void Render() override {
    }



private:
    shared_ptr<Ball> mBall;
};

Richard::Application* CreateApplication() {
    return new ClientApp();
}
```