|  |
| --- |
| Software de entretenimiento y Videojuegos Jordán Pascual : pascualjordan@uniovi.es |

|  |  |
| --- | --- |
| 1 | Juego de Naves – Parte 1 |

# Introducción

Durante las siguientes sesiones implementaremos un videojuego simple de **naves**, una vez este implementado lo iremos modificando para convertirlo en un juego de tipo **plataformas**.

**C++**

Vamos a utilizar el entorno de programación **Visual Studio** y el lenguaje de programación **C++.**

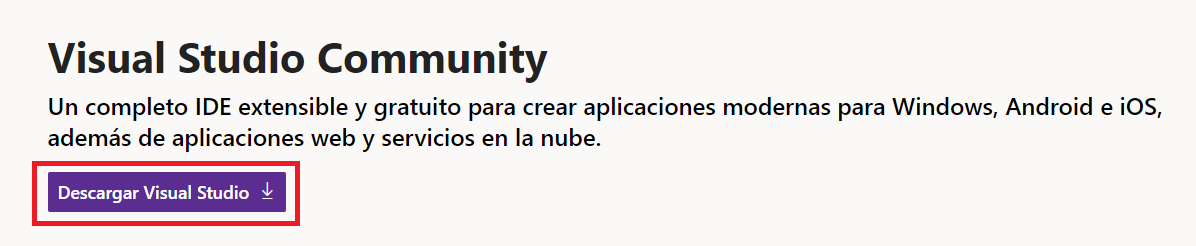
No vamos a utilizar un framework de desarrollo de videojuegos, sino una librería de gráficos, por lo tanto, toda la lógica y comportamiento propio del videojuego va a ser implementado a bajo nivel.

La librería gráfica que vamos a utilizar será SDL2 – Simple DirectMedia Layer, la librería está disponible para varios sistemas operativos (Windows, Linux, Mac, Android, iOS, entre otros <https://www.libsdl.org/> ). Para aumentar la compatibilidad lo máximo posible con los equipos de todos los alumnos vamos a utilizar la versión de **win32** de la librería (y de todas las demás).

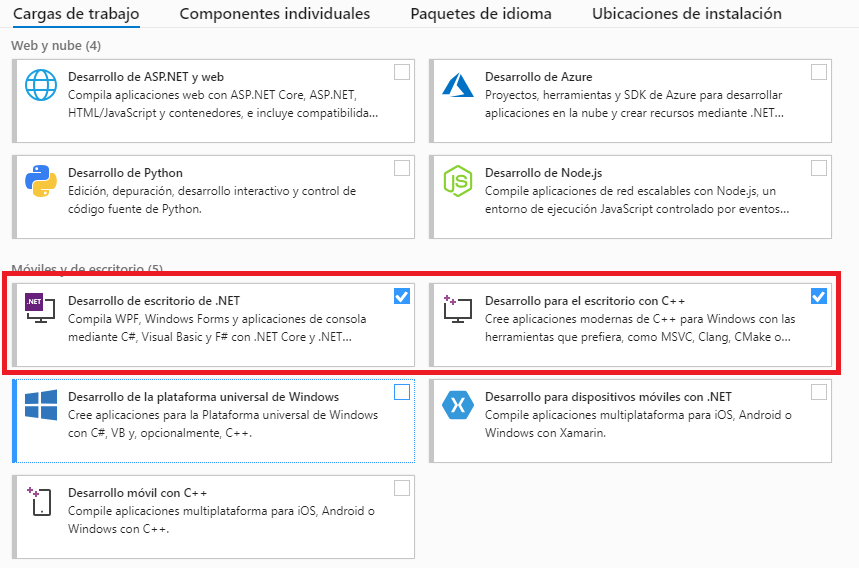
Se recomienda por tanto utilizar un equipo Windows (da igual si es de 32 o 64 bits), en caso de tener otro sistema operativo se recomienda utilizar una máquina virtual, realmente se trata de una librería multiplataforma, pero queremos reducir al máximo los problemas de configuración de los proyectos, lo que nos interesa es la programación. En el futuro y fuera de la asignatura el alumno podrá adaptar el proyecto a cualquiera de las plataformas soportadas.

**Requerimientos: Visual Studio Community**

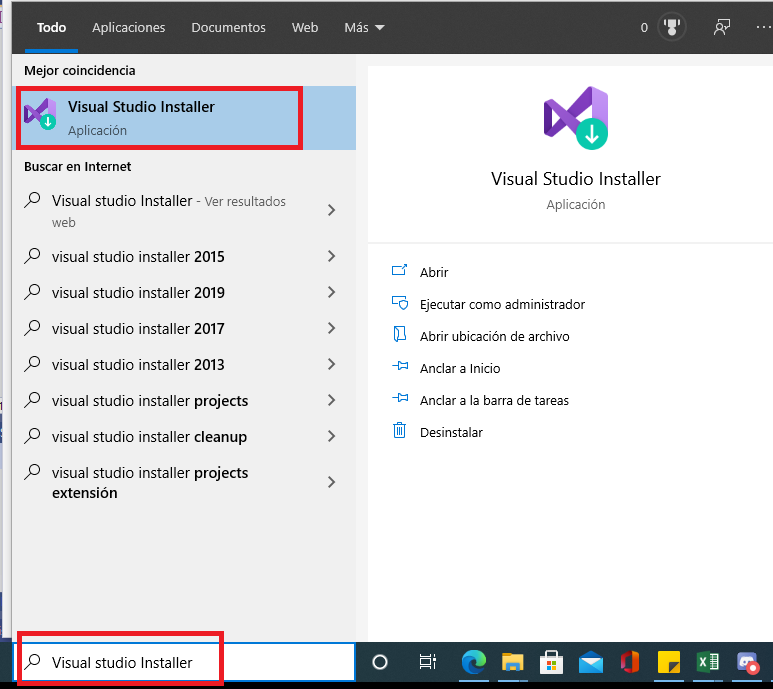
Descargamos el **Visual Studio Community** <https://visualstudio.microsoft.com/es/vs/community/>

****

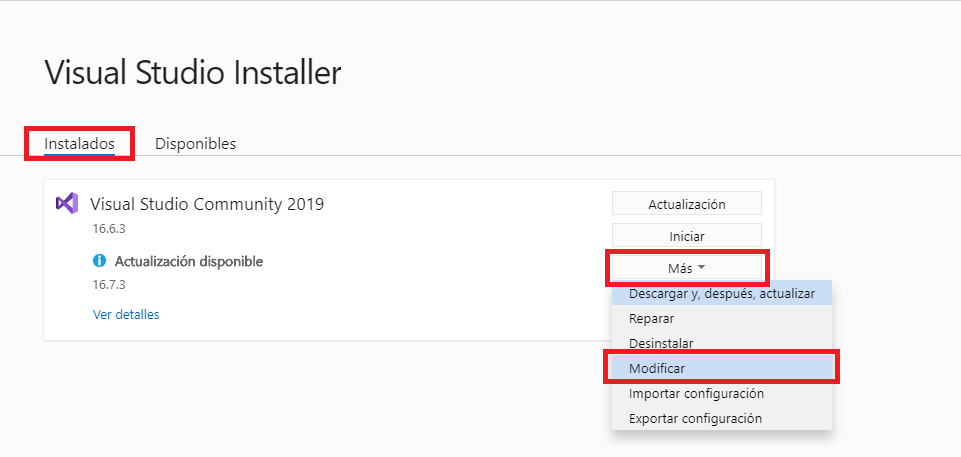
Hay que instalar el paquete **Desarrollo para el Escritorio con** **C++.** , seguramente para otras asignaturas también sea necesario el paquete **Desarrollo de escritorio .Net** aunque para está en concreto no vamos a utilizarlo.

****

Sí ya tenías instalado el **Visual Studio,** accede a la aplicación **Visual Studio Installer** para comprobar que **Cargas de trabajo** tienes instaladas (Necesitamos **Desarrollo de escritorio .Net y Desarrollo para el Escritorio con** **C++.)**

****

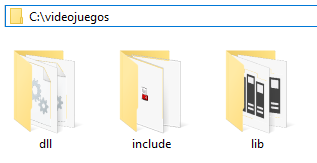
**El visual Studio Installer,** nos va a permitir Modificar la instalación actual para agregar los nuevos elementos

****

**Includes librerías y DLLs**

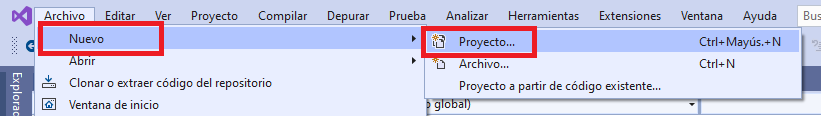
Descargamos el fichero **includesLibsDLLs.zip** del campus Virtual. Estas librerías pueden ser descargadas de las páginas oficiales, para simplificar este paso las hemos unido todas en un único zip.

Vamos a descomprimir las librerías en **C:/videojuegos** de forma que tengamos la siguiente estructura.

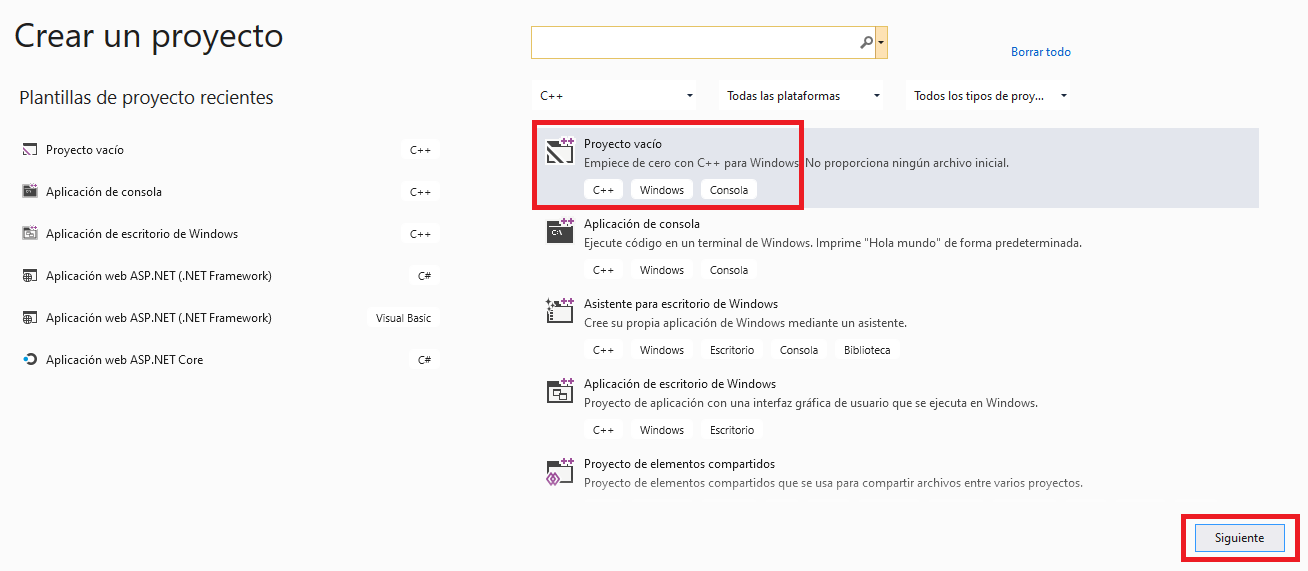


**Crear proyecto vacío**

Creamos un proyecto desde el Visual Studio.



Seleccionamos un proyecto de tipo **Proyecto vacío** de C++. Si no nos aparece este tipo de proyecto es que no hemos instalado correctamente los módulos del Visual Studio, buscarlo bien primero.



Vamos a llamarlo **Naves** y pulsamos en **Crear**.

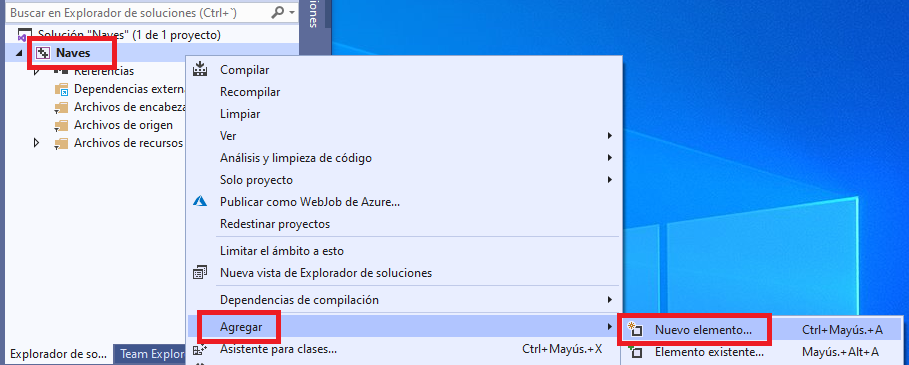


Por defecto nos va a crear una solución y un proyecto. Una solución es algo así como un “contenedor de proyectos”, aunque en este caso solo va a tener un proyecto dentro.

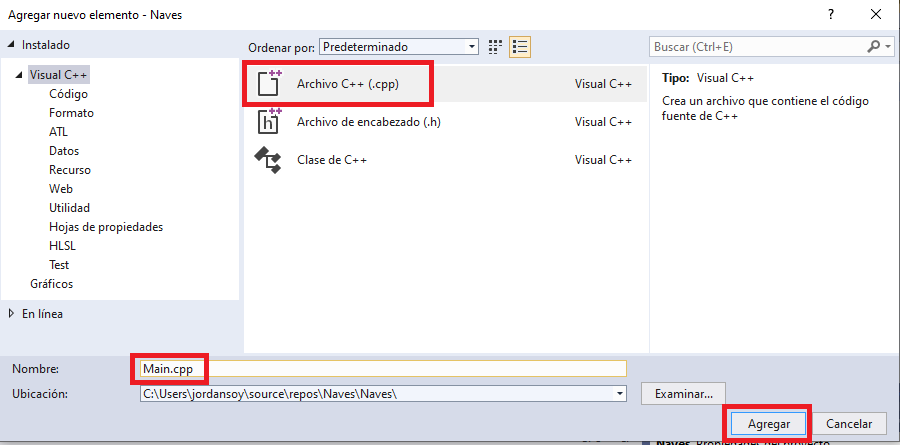
# Configuración de dependencias

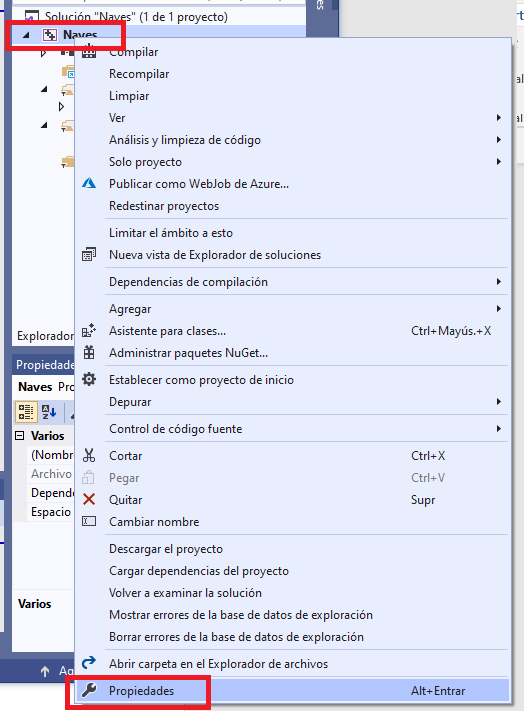
Siempre que creemos un nuevo proyecto vamos a tener que asociar el conjunto includes, dlls y librerías de SDL. Antes de agregar las dependencias el proyecto debe tener al menos un fichero de C++ , de esta forma Visual Studio sabrá que se trata de un proyecto de C++.

Para agregar un nuevo fichero de código C++, hacemos click con el botón derecho sobre el nombre del proyecto, luego **Agregar > Nuevo Elemento…**

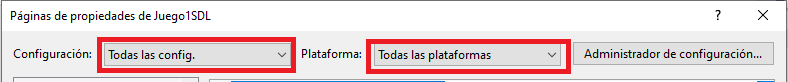


Seleccionamos un archivo de tipo **cpp** al que llamaremos **Main.cpp**



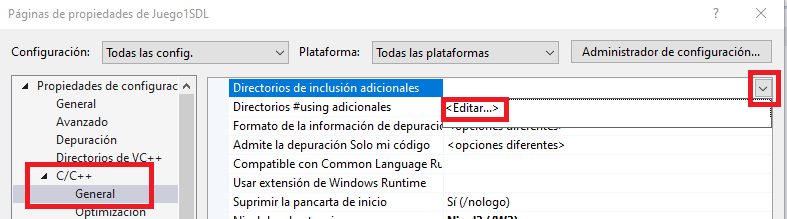


Para simplificar el proceso de configuración vamos a crear una única configuración, seleccionamos **Todas las configuraciones** y **Todas las plataformas**. Para las pruebas nosotros siempre usaremos win32 ya que las librerías están compiladas para este formato.

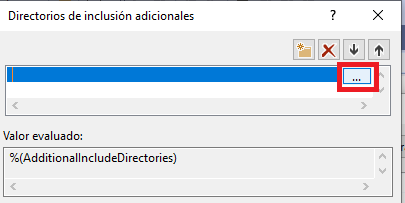


|  |
| --- |
| En otros casos podríamos querer disponer de diferentes configuraciones del proyecto para diferentes sistemas, por ejemplo, diferentes librerías para diferentes sistemas operativos, o diferenciar entre versiones de 32 y 64. |

Propiedades del proyecto, **C/C++** , categoría **General** buscamos la opción **Directorios de inclusión adicionales** , desplegamos la lista y pulsamos **<Editar…>**

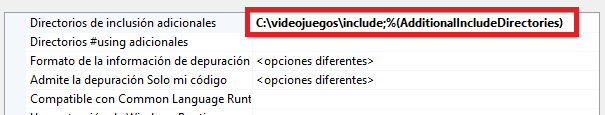
****

Seleccionamos la opción **[…]** para agregar nuevos directorios.

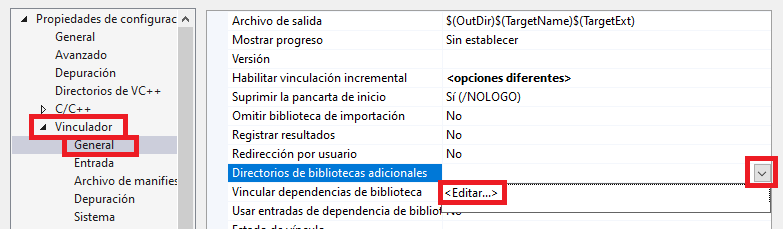


Buscamos la carpeta **C:/videojuegos/include** y pulsamos **Aceptar**. Recordamos que habíamos copiado contenido dentro de esta carpeta previamente.

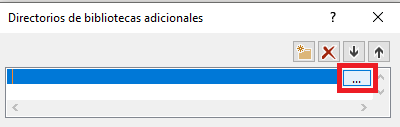
Hay que asegurarse bien de que la configuración ha quedado guardada.



Accedemos a la categoría, **Vinculador** y dentro de ella a **Entrada** y después a **General.** Tenemos que buscar la propiedad **Directorio de bibliotecas adicionales** y pulsar en **<Editar>.**

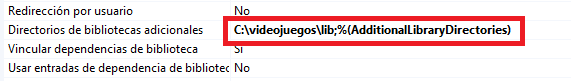


Pulsamos el botón […] para agregar un nuevo directorio.

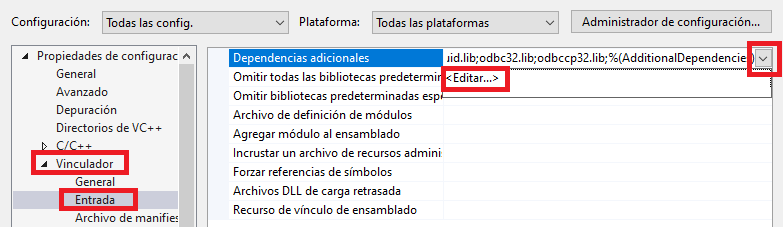


Seleccionamos el directorio **C:\videojuegos\lib,** en el que previamente habíamos copiado las librerías descargadas del campus.

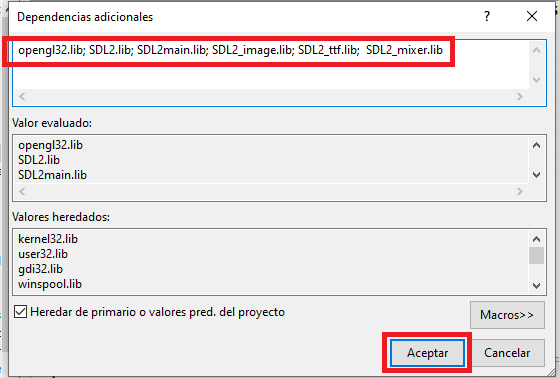
Hay que asegurarse de que la modificación haya quedado guardada.



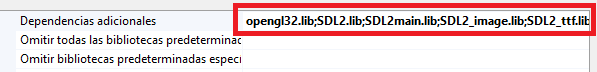
Ahora tenemos que indicar que librerías van a incluirse en la compilación. Accedemos a la categoría, **Vinculador** y dentro de ella a **Entrada**, ir a **Dependencias adicionales**

****Agregamos la siguiente línea ***opengl32.lib; SDL2.lib; SDL2main.lib; SDL2\_image.lib; SDL2\_ttf.lib; SDL2\_mixer.lib***

Estas son todas las librerías que vamos a utilizar a lo largo del bloque 1 de la asignatura, **SDL2** es la librería principal que está basada en **opengl, SDL2main** contiene la configuración inicial de librería, **SDL2\_image** y **SDL2\_ttf** son extensionesde SDL que sirven para cargar imágenes y fuentes de texto. La librería **SDL2\_mixer** sirve para cargar audio en diferentes formatos, por defecto SDL soporta pocos formatos de audio.

******

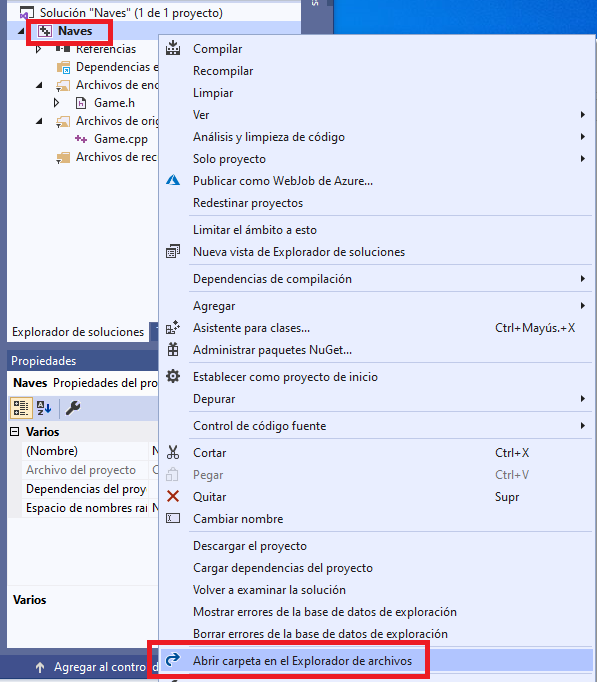
Comprobamos que la configuración se ha salvado correctamente.



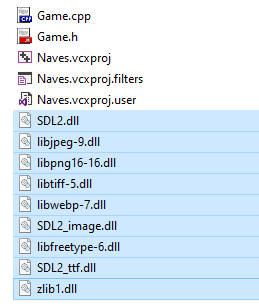
Nos falta un último paso que es añadir las **dll** que **NO** estén ya en nuestro operativo al proyecto, vamos a suponer que ninguna de las que vamos a usar estaba ya en nuestro equipo.

Abrimos la **carpeta raíz** del proyecto en el explorador de Windows.

Sí no tenemos claro dónde está la carpeta podemos hacer click con el botón derecho del ratón en el nombre del proyecto y pulsar **Abrir carpeta en el Explorador de archivos**.



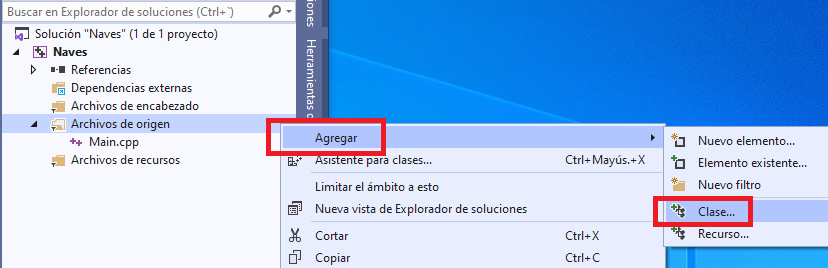
Ahora copiamos todos los ficheros **.dll** que nos hemos descargado del campus virtual (se encontraban dentro de la carpeta dll).

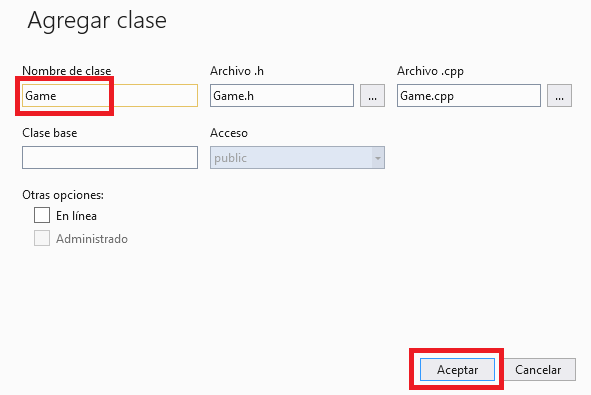


Con esto damos por finalizada la configuración inicial del proyecto, estos pasos deberían repetirse siempre que queramos crear un nuevo juego. También podemos guardar la carpeta de este proyecto y partir de él cuando queramos crear uno nuevo, basta con copiar la carpeta del proyecto desde el explorador de Windows.

**Clase principal motor del juego**

Creamos una nueva clase **Game**,hacemos **click derecho -> Agregar -> Clase…**



****

* **Game.h** es el fichero de cabeceras, en él se declara la clase, sus métodos y variables.
* **Game.cpp** es el fichero fuente, contiene la implementación de los métodos, tiene una dependencia #include del fichero Game.h , básicamente va a implementar los métodos declarados en su cabecera.

**¿Por qué dos ficheros?** Uno de los motivos principales es conseguir compilaciones más eficientes.

Accedemos al fichero **Game.h** para incluir una declaración

* #pragma once sirve para evitar duplicidades y que el código fuente sea incluido una sola vez, aunque se hagan más de un #include del mismo fichero.
* #include sirven para incluir código de vamos a utilizar en nuestra implementación, en este caso vamos a añadir ya todo lo que necesitamos <iostream>, <string>, <SDL.h>, <SDL\_image.h>, <SDL\_ttf.h>
* using namespace std. Un namespace es un bloque que agrupa funciones y objetos, por defecto los elementos “agrupados” bajo un namespace son accesibles utilizando <nombre\_namespace>::<nombre\_elemento> , por ejemplo para sacar un mensaje por consola usamos std::cout << “Hola”, cuando activamos el uso de un namespace hace que no sea necesario incluir el nombre del mismo bastaría con cout << “Hola”,
* #define sirve para crear constantes, en este caso el ancho y alto de la pantalla WIDTH y HEIGHT.

La declaración de la clase se realiza con la palabra clave **Class**, dentro se suelen incluir los bloques public: , private: protected: , para simplificarlo un poco nosotros vamos a optar por crearlo todo público, aunque no sea lo mejor desde un punto de vista de la arquitectura.

La clase Game define

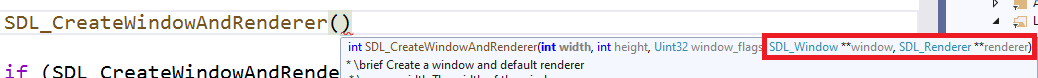
* Game() constructor sin parámetros.
* Método loop().
* Variables window (ventana del juego), renderer (superficie de dibujo/renderizado) y loopActive, booleano que indica si el juego está activo.

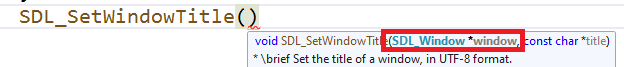
|  |
| --- |
| #pragma once  // Todas las librerias de SDL y otras utilidades  #include <iostream>  #include <string>  using namespace std;  #include <SDL.h>  #include <SDL\_image.h>  #include <SDL\_ttf.h>  // Valores generales  #define WIDTH 480  #define HEIGHT 320  class Game  {  public:  Game();  void loop();  SDL\_Window\* window; // ventana  SDL\_Renderer\* renderer; // renderizador  bool loopActive; // Juego activo  }; |

|  |
| --- |
| **C++ Punteros.** El \* que llevan las variables SDL\_Window\* y SDL\_Renderer\* indican que se trata de un puntero. Observaremos todas las variables no simples (objetos, clases, etc.) se van a referenciar mediante punteros.  Este sistema tiene algunas particularidades que iremos viendo, pero no dista demasiado de las referencias a objetos con las que solemos trabajar en Java. |

Abrimos el fichero **Game.cpp** para crear la implementación de los métodos. El fichero **.cpp** debe tener un #include de su fichero **.h** correspondiente. La implementación de los métodos lleva **<nombre\_clase>::Método**

* **Game()** constructor de la clase, llama a la función **SDL\_Init(SDL\_INIT\_EVERYTHING)** para iniciar la librería SDL con todas sus características. Una vez iniciada se crea la superficie de renderización con **SDL\_CreateWindowAndRenderer(ancho,alto,0,** \*\* **ventana,** \*\* **renderizado);** \*\* implica que necesita la dirección de memoría del puntero, la cual se obtiene utilizando el &, los parámetros querían de la siguiente forma: &window, &renderer.

****Le podemos colocar un título a la ventana con **SDL\_SetWindowTitle(\*window, const cha\*),** esta función solo recibe un **\*window**, es decir lo mismo que tenemos en nuestro puntero, basta con poner el nombre de la variable window.

****

Esta es una de las pocas veces en todo el juego en que nos encontramos con funciones que solicitan parámetros \*\*, pero hay que fijarse.

|  |
| --- |
| #include "Game.h"  Game::Game() {  if (SDL\_Init(SDL\_INIT\_EVERYTHING) < 0) {  cout << "Error SDL\_Init" << SDL\_GetError() << endl;  }  if (SDL\_CreateWindowAndRenderer(WIDTH, HEIGHT, 0, &window, &renderer) < 0) {  cout << "Error Window y Renderer" << SDL\_GetError() << endl;  }  SDL\_SetWindowTitle(window, "Juego de Naves");  // Escalado de imágenes de calidad  // https://wiki.libsdl.org/SDL\_HINT\_RENDER\_SCALE\_QUALITY  SDL\_SetHint(SDL\_HINT\_RENDER\_SCALE\_QUALITY, "1");  loopActive = true; // bucle activo  loop();  }  void Game::loop() {  } |

Implementamos el método **loop()**, lo que va a hacer es crear un bucle while infinito mientras el juego este activo. Las ejecuciones de este bucle deberían mantenerse constantes*,* ejemplo 30 por segundo.

Lo que hacemos es calcular los **ms** (**SDL\_GetTicks()**) al inicio y al final, cuando acaba hacemos una pausa **SDL\_Delay(ms)** con el tiempo necesario para mantener 30 ejecuciones por segundo (30 fps)

|  |
| --- |
| void Game::loop() {  int initTick; // ms de inicio loop  int endTick; // ms de fin de loop  int differenceTick; // fin - inicio  while (loopActive) {  initTick = SDL\_GetTicks();  // Controles  // Actualizar elementos  // Dibujar  endTick = SDL\_GetTicks();  differenceTick = endTick - initTick;  if (differenceTick < (1000 / 30)) {  SDL\_Delay((1000 / 30) - differenceTick);  }  }  } |

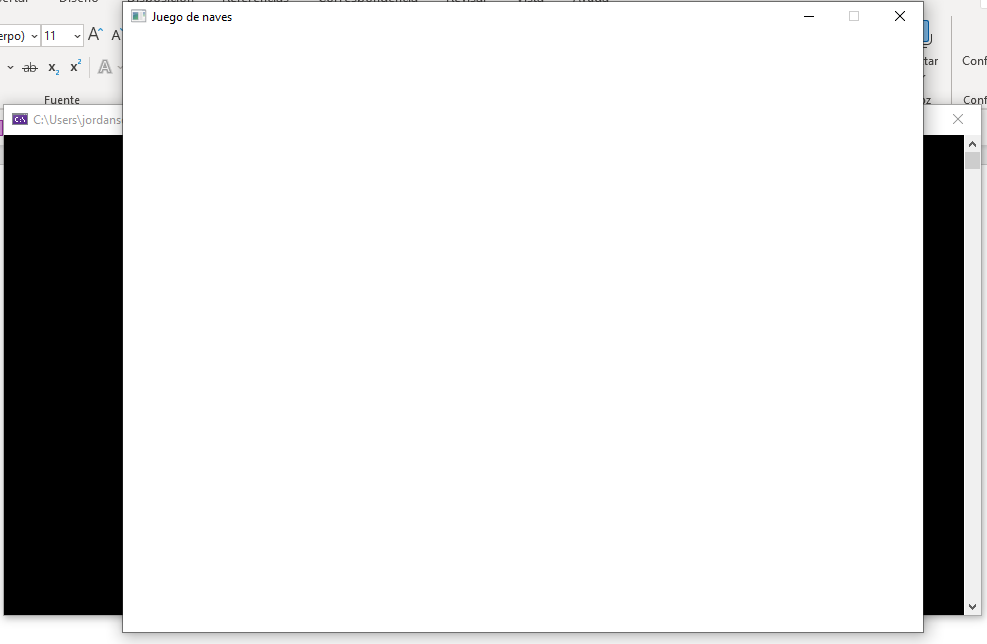
Accedemos al fichero **Main.cpp**. Este fichero NO es una clase únicamente contiene una función **main**. Sustituimos su implementación por el siguiente código:

|  |
| --- |
| #include "Game.h"  #undef main //esto es para quitar el main que hay en la Librería SDL\_main  int main() {  Game\* game = new Game();  return 0;  } |

Antes de pulsar en el ejecutar el proyecto nos aseguramos de tener seleccionada la configuración **x86** (Es la que se corresponde con las librerías **win32 / x86** que estamos utilizando).

****

Al ejecutar el proyecto se abre una consola de comandos y la ventana.

****

Para cerrar el programa pulsamos la opción de **detener** la depuración.

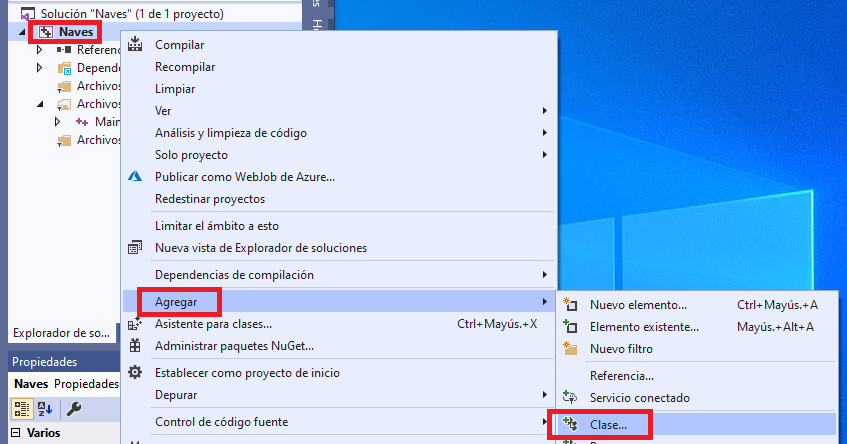


|  |  |  |  |
| --- | --- | --- | --- |
| **cout.** Utilizamos el **cout** (del espacio sdt) para mostrar mensajes por consola, el operador **<<** va incluyendo valores en la salida   |  | | --- | | cout << "hola Mundo";  cout << "Valor: " << posicionX ; | |  | |  | |

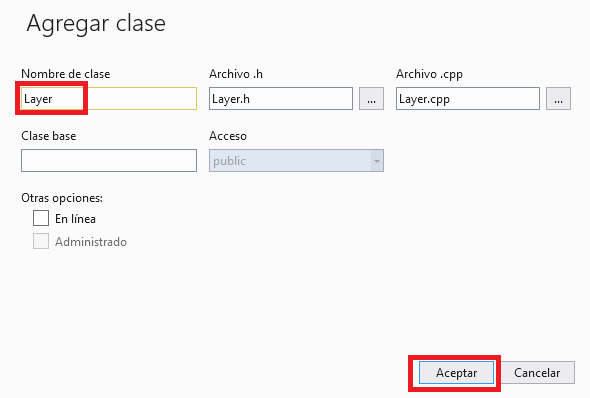
|  |
| --- |
| **Visual Studio ficheros .h y ficheros .cpp**  Al principio nos puede resultar un poco tedioso movernos entre los ficheros **.h** y **.cpp** es recomendable utilizar los atajos del Visual Studio  **Ir a la definición - Fichero cpp. (F12)**    **Ver la definición, en el propio fichero .h (ALT F12)**    **Agregar definición (en caso de que no la tenga ya).** Desde el fichero .h podemos agregar la implementación del cpp |

**Clase base Layer y GameLayer**

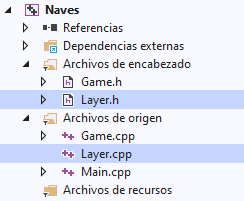
Vamos a agregar una nueva clase **Layer** que servirá para definir los elementos comunes capas/ventanas de nuestro juego. **Agregar -> Clase …**



Le ponemos el nombre **Layer**.

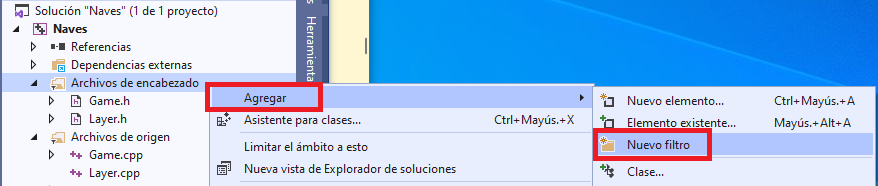


Podemos ver que ahora tenemos dos nuevos ficheros en nuestro proyecto.

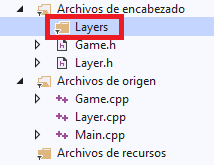


Existen varias maneras de organizar el código en un proyecto C++ en el Visual Studio, una de ellas es mediante el uso de **filtros**.

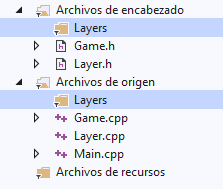
Hacemos click derecho sobre la “carpeta” **Archivos de Encabezado** pulsamos en **Agregar** , **Nuevo Filtro**.



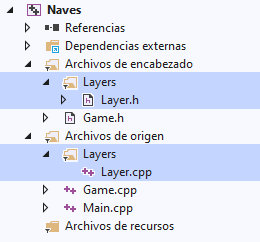
Haciendo click derecho sobre el filtro pulsamos en **cambiar nombre** y lo renombramos a **Layers**.



Se repite exactamente el mismo proceso sobre la carpeta **Archivos de origen**, la cual contiene los ficheros **.cpp**. El resultado debería ser el siguiente:



Finalmente arrastramos los ficheros **Layer.h** y **Layer.cpp** dentro de su correspondiente carpeta, de forma que consigamos una estructura de árbol.



Accedemos a **Layer.h,** realmente está clase no va a tener mucha funcionalidad, está pensada para definir una estructura y que otras hereden de ella.

Sí la clase va a tener métodos vacíos podemos crearlos en el propio fichero .h (todos menos el constructor). La palabra clave **virtual** indica que el método puede ser redefinido en una clase hija.

Va a tener un variable **Game\* (referencia al motor de juego),** para poder usar este objeto tenemos que hacer un#include “Game.h”veremos que el futuro esto implicara consideraciones adicionales al incluir una referencia cruzada (Game tiene una referencia a Layer y Layer una a Game).

|  |
| --- |
| #pragma once  #include "Game.h"  class Layer  {  public:  Layer(Game\* game);  virtual void init() {};  virtual void processControls() {};  virtual void update() {};  virtual void draw() {};  Game\* game;  }; |

Accedemos a la implementación **Layer.cpp** e implementamos el constructor, al igual que en Java cuando el parámetro recibido se llama igual que la variable tenemos que usar **this** para diferenciar cual es la variable de la clase y cual el parámetro.

|  |
| --- |
| #include "Layer.h"  Layer::Layer(Game\* game) {  this->game = game;  } |

|  |
| --- |
| **Acceso a variables y métodos de un puntero.** En nuestro juego casi todos los objetos serán punteros \* para acceder a los elementos declarados en él utilizamos **->** no utilizamos un . como en Java. Como es un fallo bastante común Visual Studio hace el cambio del . por un -> si pulsamos en la sugerencia. |

Probamos a ejecutar la aplicación, todavía no hace nada, pero es importante validar que no tiene errores.

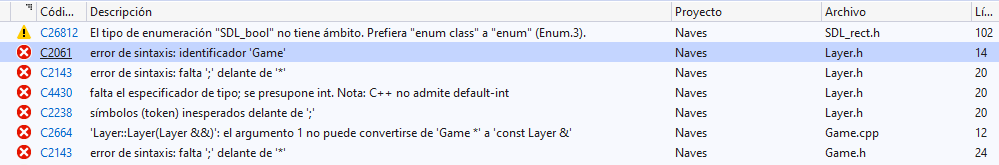
Vamos al **Game.h** e incluimos una nueva variable de tipo Layer, necesitamos hacer previamente un #include "Layer.h"

|  |
| --- |
| #define HEIGHT 320  #include "Layer.h"  class Game  {  public:  Game();  void loop();  SDL\_Window\* window; // ventana  SDL\_Renderer\* renderer; // renderizador  bool loopActive; // Juego activo  Layer\* gameLayer;  }; |

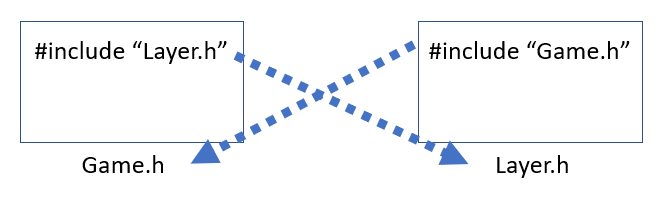
Nos dirigimos a la implementación de **Game.cpp,** en el constructor creamos la layer, y en el loop llamamos a los 3 métodos de la layer (procesar controles, actualizar, dibujar).

|  |
| --- |
| Game::Game() {  if (SDL\_Init(SDL\_INIT\_EVERYTHING) < 0) {  cout << "Error SDL\_Init" << SDL\_GetError() << endl;  }  if (SDL\_CreateWindowAndRenderer(WIDTH, HEIGHT, 0, &window, &renderer) < 0) {  cout << "Error Window y Renderer" << SDL\_GetError() << endl;  }  SDL\_SetWindowTitle(window, "Juego de Naves");  // Escalado de imágenes de calidad  // https://wiki.libsdl.org/SDL\_HINT\_RENDER\_SCALE\_QUALITY  SDL\_SetHint(SDL\_HINT\_RENDER\_SCALE\_QUALITY, "1");  gameLayer = new Layer(this);  loopActive = true; // bucle activo  loop();  }  void Game::loop() {  int initTick; // ms de inicio loop  int endTick; // ms de fin de loop  int differenceTick; // fin - inicio  while (loopActive) {  initTick = SDL\_GetTicks();  // Controles  gameLayer->processControls();  // Actualizar elementos  gameLayer->update();  // Dibujar  gameLayer->draw();  endTick = SDL\_GetTicks();  differenceTick = endTick - initTick;  if (differenceTick < (1000 / 30)) {  SDL\_Delay((1000 / 30) - differenceTick);  }  }  } |

Sí probamos a ejecutar el proyecto nos vamos a encontrar con varios errores, todos ellos derivados de que la clase **Layer** no consigue identificar la clase **Game**.



Este problema no se va a repetir más en el juego, pero acabamos de crear una dependencia cíclica. Game no encuentra el Layer porque está sin crear (ya que necesita el Game)



Vamos a solucionar el problema, modificando las cabeceras de las dos clases implicadas **Game.h** y **Layer.h**.

Abrimos el fichero **Game.h** y añadiendo dentro una declaración de la clase Layer, con ***class Layer;*** , de está forma ya encontrará la clase layer.

|  |
| --- |
| #include "Layer.h"  class Layer;  class Game  {  public: |

Abrimos **Layer.h** y hacemos justo lo contrario, añadimos una declaración de ***class Game;***

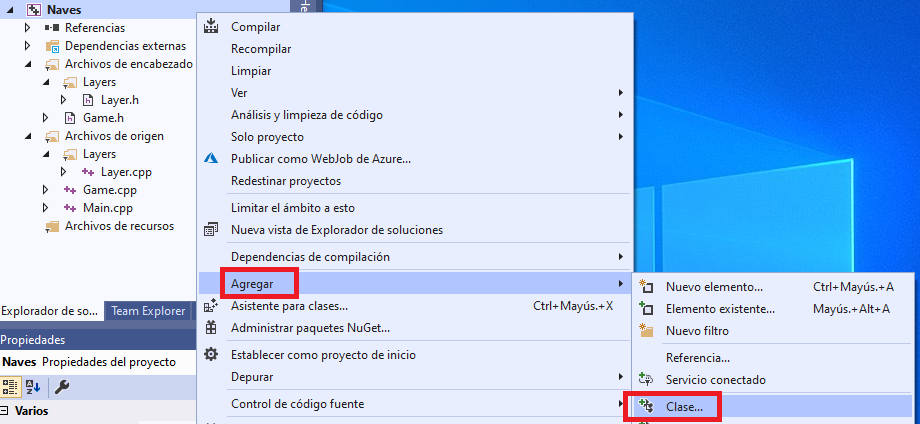
|  |
| --- |
| #include "Game.h"  class Game;  class Layer  {  public: |

Sí probamos a ejecutar ahora debería funcionar correctamente.

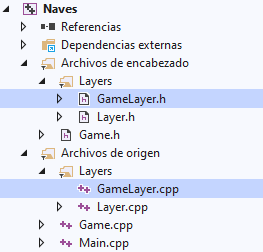
# GameLayer – pantalla de juego

Vamos a añadir una implementación especifica de una capa Layer, que representará nuestra capa del juego.

Creamos una nueva clase **GameLayer.**



Para continuar manteniendo una buena organización arrastramos los ficheros **GameLayer.h** y **GameLayer.cpp** dentro de sus correspondientes filtros.



**GameLayer.h,** va a heredar de Layer, y sobrescribir los métodos virtuales, para ello incluye la palabra clave **override** en la signatura.

|  |
| --- |
| #pragma once  #include "Layer.h"  class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  void init() override;  void processControls() override;  void update() override;  void draw() override;  }; |

Accedemos a **GameLayer.cpp** para completar la implementación. Para llamar al constructor de la clase padre usamos **: Layer(game).** Por el momento casi todos los métodos estarán vacíos.

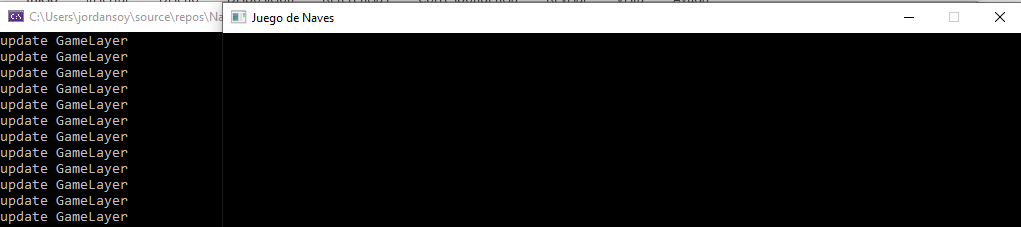
|  |
| --- |
| #include "GameLayer.h"  GameLayer::GameLayer(Game\* game)  : Layer(game) {  //llama al constructor del padre : Layer(renderer)  init();  }  void GameLayer::init() {  }  void GameLayer::processControls() {  }  void GameLayer::update() {  cout << "update GameLayer" << endl;  }  void GameLayer::draw() {  SDL\_RenderPresent(game->renderer); // Renderiza  } |

Vamos a que la capa que hay creada en el **Game** (motor del juego), **NO** sea una Layer genérica, sino una **GameLayer**, en la que posteriormente iremos incluyendo jugadores y enemigos.

Accedemos a **Game.cpp** , simplemente hacemos el #include “GameLayer.h” y cambiamos la construcción de la clase por **new GameLayer(this).**

|  |
| --- |
| #include "Game.h"  #include "GameLayer.h"  Game::Game() {  if (SDL\_Init(SDL\_INIT\_EVERYTHING) < 0) {  cout << "Error SDL\_Init" << SDL\_GetError() << endl;  }  if (SDL\_CreateWindowAndRenderer(WIDTH, HEIGHT, 0, &window, &renderer) < 0) {  cout << "Error Window y Renderer" << SDL\_GetError() << endl;  }  SDL\_SetWindowTitle(window, "Juego de Naves");  // Escalado de imágenes de calidad  // https://wiki.libsdl.org/SDL\_HINT\_RENDER\_SCALE\_QUALITY  SDL\_SetHint(SDL\_HINT\_RENDER\_SCALE\_QUALITY, "1");  gameLayer = new GameLayer(this);  loopActive = true; // bucle activo  loop();  } |

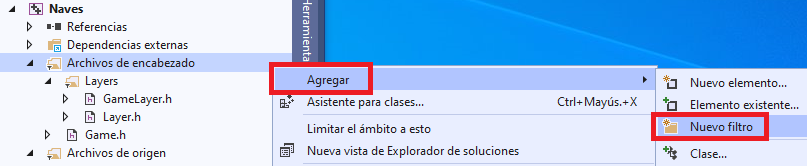
Sí probamos a ejecutar el juego veremos que se está utilizando correctamente el objeto de tipo GameLayer, lo sabemos porque estábamos imprimiendo por consola el mensaje *“update GameLayer”* en cada loop del juego.



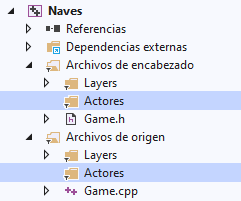
**Actor**

Un Actor va a ser cualquier elemento con representación gráfica en nuestro juego, un jugador, un enemigo, un botón, etc.

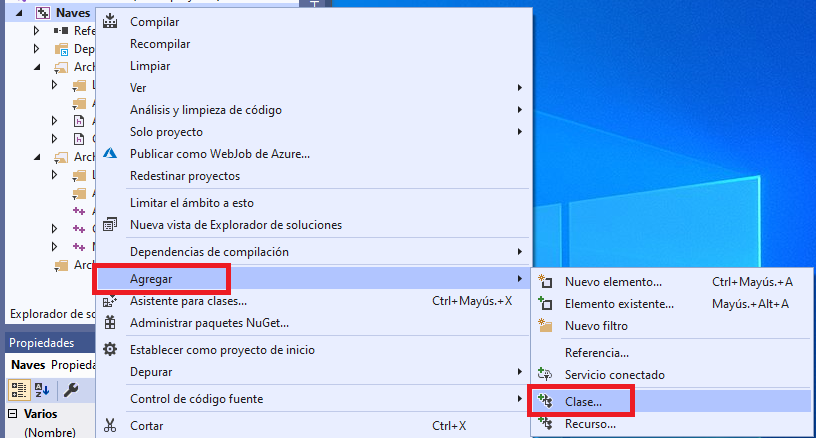
Creamos un nuevo filtro al que llamaremos **Actores**, un **filtro** dentro de **Archivos de encabezado**.



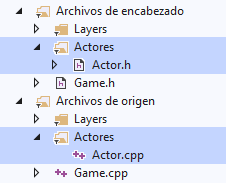
El otro filtro lo creamos dentro de **Archivos de origen**, el resultado final debería ser igual al que se muestra en la siguiente captura.

****

Creamos una nueva clase **Actor**, va a servir como clase base de cualquier elemento del juego. **Agregar -> Clase.**



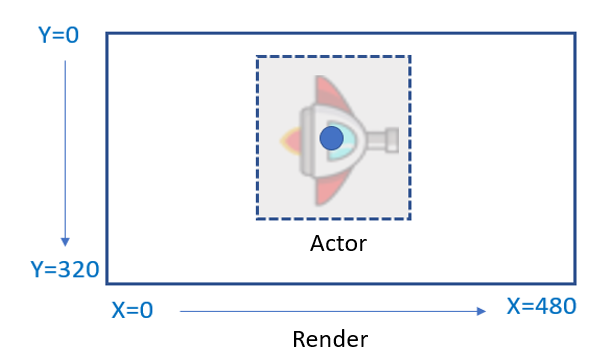
Arrastraremos los ficheros **Actor.h** y **Actor.cpp** a sus correspondientes carpetas.



**Actor.h** va a tener una referencia al motor de juego Game, pero el motor de juego Game no va a tener actores (no hay referencias cíclicas).

Un actor va a tener un **constructor**, un método **draw()** (que puede ser sobrescrito por sus clase hijas) y un conjunto de variables que tienen que ver con su representación gráfica y movimiento.

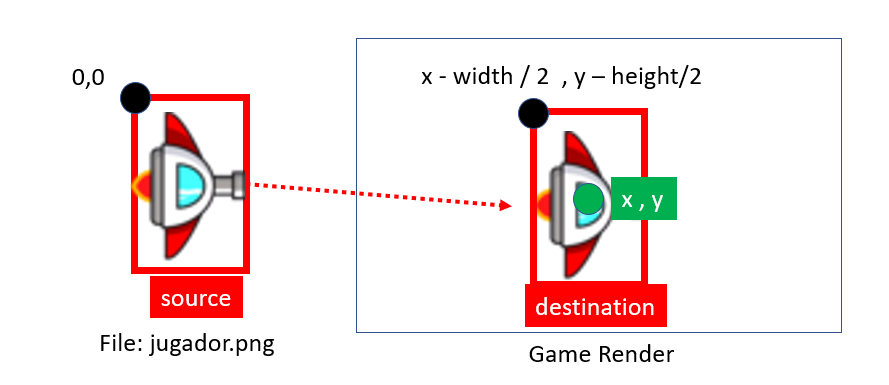
* **texture**, es la imagen que lo representa.
* **x** e **y** son las coordenadas en la superficie de renderizado.
* **vx** y **vy** son las velocidades (en caso de que se mueva).
* **width y height** es lo que mide en la superficie de renderizado.
* **fileWidth** y **fileHeight** es lo que mide el fichero de la imagen, normalmente pintamos al actor del mismo tamaño que la imagen fuente, pero no es obligatorio, podríamos pintarlo más grande o más pequeño.



|  |
| --- |
| #pragma once  #include "Game.h"  class Actor  {  public:  Actor(string filename, float x, float y, int width, int height, Game\* game);  virtual void draw();  SDL\_Texture\* texture;  int x;  int y;  float vx;  float vy;  int width;  int height;  int fileWidth;  int fileHeight;  Game\* game; // referencia al juego  }; |

Agregamos la implementación en **Actor.cpp.**

* El **constructor** simplemente guarda los valores de las variables recibidas. Lo más complejo que hace es crear la texture , este proceso se hace en dos pasos primero obteniendo una **SDL\_Surface** y luego creando la textura con **SDL\_CreateTextureFromSurface.**
* **draw()** básicamente crea dos rectángulos **SDL\_Rect:**
  + El primero se llama **source** y recorta el fichero de la imagen (muy sencillo comienza recortando en la coordenada 0,0 usando el tamaño de la imagen).
  + El segundo representa el **destination** donde se va a pegar la imagen sobre lasuperficie de renderización del juego. El rectángulo usa como referencia el punto superior izquierdo , como nosotros dijimos que nuestra x e y eran el punto central hay que “transformarlo”, es muy sencillo basta con restar la mitad del ancho y la mitad del alto (está es una cuenta que repetiremos en otros cálculos)



* + **SDL\_RenderCopyEx** es la función que se encarga de copiar en la superficie de renderización la textura.

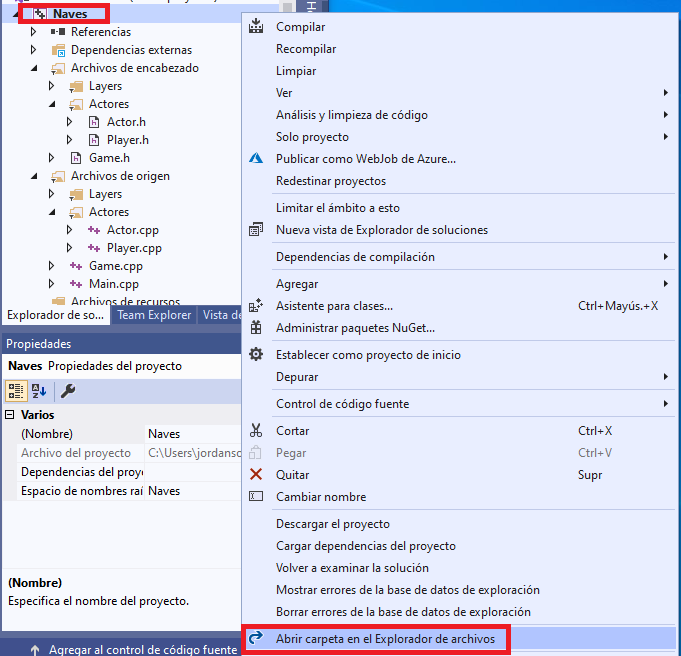
|  |
| --- |
| #include "Actor.h"  Actor::Actor(string filename, float x, float y, int width, int height, Game\* game) {  this->game = game;  SDL\_Surface\* surface = IMG\_Load(filename.c\_str());  texture = SDL\_CreateTextureFromSurface(game->renderer, surface);  this->x = x;  this->y = y;  // lo que mide el fichero  this->fileWidth = width;  this->fileHeight = height;  // lo que va a medir en el juego  this->width = width;  this->height = height;  }  void Actor::draw() {  // Recorte en el fichero de la imagen  SDL\_Rect source;  source.x = 0;  source.y = 0;  source.w = fileWidth; // texture.width;  source.h = fileHeight; // texture.height;  // Donde se va a pegar en el renderizador  SDL\_Rect destination;  destination.x = x - width / 2;  destination.y = y - height / 2;  destination.w = width;  destination.h = height;  // Modificar para que la referencia sea el punto central  SDL\_RenderCopyEx(game->renderer,  texture, &source, &destination, 0, NULL, SDL\_FLIP\_NONE);  } |

|  |  |  |
| --- | --- | --- |
| **Coordenada superior izquierda o central**. Recordamos que la transformación entre estos dos ejes es muy sencilla.   |  |  | | --- | --- | | // Donde se va a pegar en el renderizador  SDL\_Rect destination;  destination.x = x - width / 2;  destination.y = y - height / 2; |  | |

Probamos a ejecutar la aplicación para ver que compila.

# Recursos gráficos

Aun no disponemos de ficheros de imagen, descargamos el campus virtual el fichero **res.zip** . Abrimos la carpeta principal del proyecto desde el **explorador de archivos de Windows**.

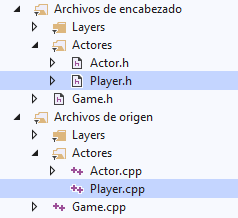


Descomprimimos dentro la carpeta **/res .** Asegurarse de que no hay ningún directorio dentro de la carpeta **/res**, dentro de /res deben estar directamente los recursos.

|  |  |
| --- | --- |
|  | res/jugador.png |

**Player**

Creamos una nueva clase **Player (Agregar -> Clase…)**. Arrastramos los ficheros **Player.h** y **Player.cpp** dentro de las carpetas **Actores.**

****

Definimos **Player.h**, como incluye Actor.h se heredan todos los #include declarados en su clase padre, no es necesario a ver un include de Game.

|  |
| --- |
| #pragma once  #include "Actor.h"  class Player : public Actor  {  public:  Player(float x, float y, Game\* game);  }; |

Para implementar **Player.cpp** , tenemos que ver las propiedades del fichero imagen para ver cuanto tiene de ancho y de alto, en el juego la pintaremos con esas mismas dimensiones (aunque podrían ser otras). En este caso es de 50 x 57.

|  |  |
| --- | --- |
|  |  |

|  |
| --- |
| #include "Player.h"  Player::Player(float x, float y, Game\* game)  : Actor("res/jugador.png", x, y, 50, 57, game) {  } |

Abrimos el **GameLayer.h** y añadimos una variable de tipo **Player.**

|  |
| --- |
| #include "Layer.h"  #include "Player.h"  class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  void init() override;  void processControls() override;  void update() override;  void draw() override;  Player\* player;  }; |

Ahora abrimos la implementación de **GameLayer.cpp**. Dentro del método **init()** instanciamos la variable **player**, dentro del método **draw()** lo dibujamos.

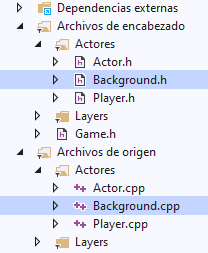
|  |
| --- |
| GameLayer::GameLayer(Game\* game) : Layer(game) {  //llama al constructor del padre : Layer(renderer)  init();  }  void GameLayer::init() {  player = new Player(50, 50, game);  }  void GameLayer::processControls() {  }  void GameLayer::update() {  cout << "update GameLayer" << endl;  }  void GameLayer::draw() {  player->draw();  SDL\_RenderPresent(game->renderer); // Renderiza  } |

Probamos a ejecutar la aplicación, el player parecerá en la coordenada **50,50.**



**Fondo**

Creamos una nueva clase **Background (Agregar -> Clase…),** arrastramos sus ficheros dentro de las carpetas **Actors** (ya que también va a heredar de Actor).



Para hacerlo más versátil vamos a permitir que especifiquen la ruta de laimagen en el constructor, de esta forma podemos construir diferentes fondos con la misma clase.

Declaramos **Background.h**

|  |
| --- |
| #pragma once  #include "Actor.h"  class Background : public Actor  {  public:  Background(string filename, float x, float y, Game\* game);  }; |

Implementamos ahora **Background.cpp**, añadiendo la implementación del constructor. Vamos a hacer que los fondos sean del mismo tamaño de la pantalla (WIDTH x HEIGHT en este juego), en este juego habíamos dicho que eran 320\*480, nos servirá cualquier imagen de esas dimensiones.

|  |
| --- |
| #include "Background.h"  Background::Background(string filename, float x, float y, Game\* game)  : Actor(filename, x, y, WIDTH, HEIGHT, game) {  } |

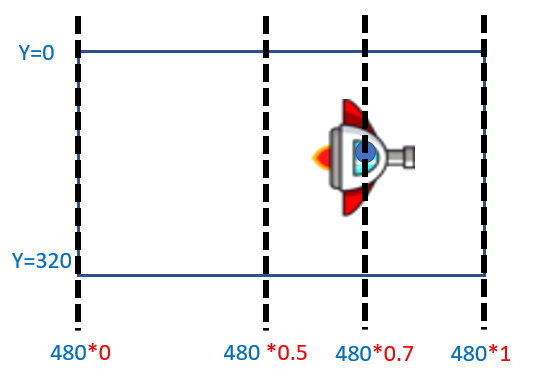
Solo falta añadir el fondo en la capa del juego GameLayer. Abrimos el **GameLayer.h** y creamos una nueva variable de tipo **Background**.

|  |
| --- |
| #pragma once  #include "Layer.h"  #include "Player.h"  #include "Background.h"  class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  void init() override;  void processControls() override;  void update() override;  void draw() override;  Player\* player;  Background\* background;  }; |

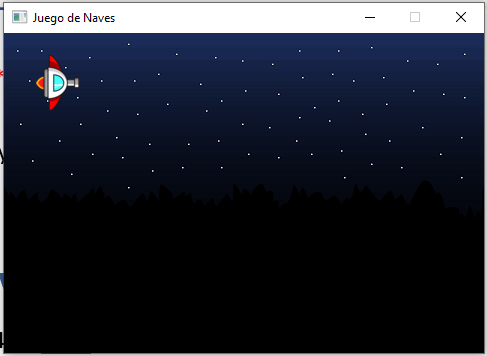
Accedemos a **GameLayer.cpp** e instanciamos el **background**, nos aseguramos de dibujarlo dentro del método **draw()** hay que dibujarlo antes que el **player**. El orden en el que dibujamos los actores tendrá mucha importancia.

|  |
| --- |
| void GameLayer::init() {  player = new Player(50, 50, game);  background = new Background("res/fondo.png", WIDTH\*0.5, HEIGHT\*0.5, game);  }  void GameLayer::processControls() {  }  void GameLayer::update() {  cout << "update GameLayer" << endl;  }  void GameLayer::draw() {  background->draw();  player->draw();  SDL\_RenderPresent(game->renderer); // Renderiza  } |

|  |
| --- |
| Para posicionar las coordenadas en la superficie de renderizado se puede multiplicar el total del **WIDTH** y el **HEIGHT** por un por un valor entre (0 y 1), por ejemplo, si lo multiplicamos por **0.5** estará en la mitad. |



Ejecutamos el proyecto y comprobamos que funcionando sin errores.

****

**Eventos de teclado y Controles**

Vamos a incluir un nuevo método y variables en la clase GameLayer, para ello abrimos el fichero **GameLayer.h**

El método **keysToControls();** va a ser el encargado de leer las teclas que hay pulsadas en el teclado y convertirlas a controles.

Un control va a ser una acción que el juego entienda, por ejemplo, disparar **controlShoot**, moverse en el eje x **controlMoveX** o moverse en el eje y **controlMoveY**. Un mismo **control** podría ser activado por diferentes eventos, un teclado, un click, una pulsación de mando, etc. Realmente da igual como se active, el juego realiza la misma acción, por ejemplo disparar.

|  |
| --- |
| class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  void init() override;  void processControls() override;  void update() override;  void draw() override;  void keysToControls(SDL\_Event event);  Player\* player;  Background\* background;  bool controlShoot = false;  int controlMoveY = 0;  int controlMoveX = 0;  }; |

Accedemos a la implementación de **GameLayer.cpp,** accedemos al método **processControls()** e incluimos una llamadaa **keysToControls().** Todos los eventos de la aplicación se guarda en un poll de eventos tipo SDL\_Event, lo que hacemos es recorrerlos todos utilizando un while, cada uno de esos eventos debe ser procesado, por **keysToControls().**

|  |
| --- |
| void GameLayer::processControls() {  // obtener controles  SDL\_Event event;  while (SDL\_PollEvent(&event)) {  keysToControls(event);  }  //procesar controles  }  void GameLayer::keysToControls(SDL\_Event event) {  } |

El objetivo del método **keysToControls** es detectar que teclas se han pulsado y levantado, para transformar esas teclas en controles (dar valor a las variables controlMoveX, controlMoveY, controlShoot). Habrá que comprobar de que tipo es el evento recibido como parámetro.

|  |
| --- |
| void GameLayer::keysToControls(SDL\_Event event) {  if (event.type == SDL\_KEYDOWN) {  int code = event.key.keysym.sym;  // Pulsada  }  if (event.type == SDL\_KEYUP) {  int code = event.key.keysym.sym;  // Levantada  }  } |

Vamos primero con las pulsaciones (eventos SDL\_KEYDOWN) , utilizamos las teclas:

* **d** y **a** para mover en el eje x
* **w** y **s** para mover en el eje y
* **espacio** para disparar

|  |
| --- |
| if (event.type == SDL\_KEYDOWN) {  int code = event.key.keysym.sym;  // Pulsada  switch (code) {  case SDLK\_d: // derecha  controlMoveX = 1;  break;  case SDLK\_a: // izquierda  controlMoveX = -1;  break;  case SDLK\_w: // arriba  controlMoveY = -1;  break;  case SDLK\_s: // abajo  controlMoveY = 1;  break;  case SDLK\_SPACE: // dispara  controlShoot = true;  break;  }  }  if (event.type == SDL\_KEYUP) { |

Levantar la pulsación (SDL\_KEYUP) también tiene efecto sobre los controles, si se estaba moviendo en una dirección dejará de moverse, si estaba disparando dejará de hacerlo, etc.

|  |
| --- |
| if (event.type == SDL\_KEYUP) {  int code = event.key.keysym.sym;  // Levantada  switch (code) {  case SDLK\_d: // derecha  if (controlMoveX == 1) {  controlMoveX = 0;  }  break;  case SDLK\_a: // izquierda  if (controlMoveX == -1) {  controlMoveX = 0;  }  break;  case SDLK\_w: // arriba  if (controlMoveY == -1) {  controlMoveY = 0;  }  break;  case SDLK\_s: // abajo  if (controlMoveY == 1) {  controlMoveY = 0;  }  break;  case SDLK\_SPACE: // dispara  controlShoot = false;  break;  }  } |

|  |
| --- |
| **Switch en C++.** Si tenemos pensado declarar diferentes variables locales dentro de los cases deberíamos utilizar llaves { } para delimitar cada case:  switch (code) {  case SDLK\_d: {  controlMoveX = 1;  break;  }  case SDLK\_a: {  controlMoveX = -1;  break;  } |

Comprobamos que la aplicación se ejecuta sin errores.

Llega el momento de implementar el procesamiento de los controles en **processControls()**, ya tenemos los valores de los controles ahora estos tienen que tener un efecto en el jugador.

Los métodos **moveX** y **moveY** del player están aún sin implementar, servirán para otorgarle velocidad en los ejes x e y, el movimiento en cada eje puede tomar 3 valores positivo, cero o negativo.

|  |
| --- |
| void GameLayer::processControls() {  // obtener controles  SDL\_Event event;  while (SDL\_PollEvent(&event)) {  keysToControls(event);    }  //procesar controles  // Disparar  if (controlShoot) {  }  // Eje X  if (controlMoveX > 0) {  player->moveX(1);  }  else if (controlMoveX < 0) {  player->moveX(-1);  }  else {  player->moveX(0);  }  // Eje Y  if (controlMoveY > 0) {  player->moveY(1);  }  else if (controlMoveY < 0) {  player->moveY(-1);  }  else {  player->moveY(0);  }  } |

En cada iteración **update()** de la capa se debe también actualizar al Player, para ello haremos una llamada al método **player->update()**, este método está aun sin implementar, se va a encargar de mover las coordenadas **x** e **y** del jugador.

|  |
| --- |
| void GameLayer::update() {  player->update();  cout << "update GameLayer" << endl;  } |

Para que el **Player** pueda cambiar de velocidad y actualizar su posición debemos agregarle los métodos **moveX(axis)**, **moveY(axis)** y **update().**

Accedemos a **Player.h** para declarar los nuevos métodos.

|  |
| --- |
| class Player : public Actor  {  public:  Player(float x, float y, Game\* game);  void update();  void moveX(float axis);  void moveY(float axis);  }; |

Vamos a la definición de **Player.cpp** para incluir la implementación, comenzamos por **update()** que modificará sus coordenadas **x** e **y** en función de sus velocidades **vx** e **vy**.

|  |
| --- |
| Player::Player(float x, float y, Game\* game)  : Actor("res/jugador.png", x, y, 50, 57, game) {    }  void Player::update() {  x = x + vx;  y = y + vy;  } |

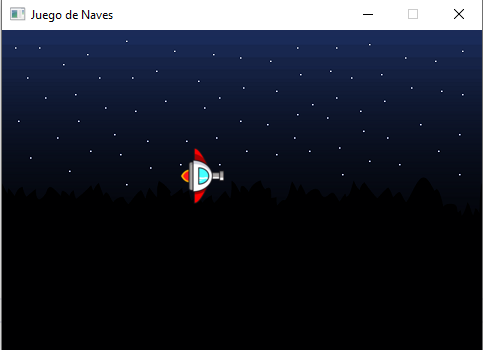
Implementamos los métodos encargados de variar la velocidad **moveX(axis**) y **moveY(axis)** , modificarán las velocidades dependiendo del parámetro **axis** recibido.

* axis > 0 se mueve de forma positiva en el eje. (derecha / abajo)
* axis < 0 se mueve de forma negativa en el eje. (izquierda / arriba)

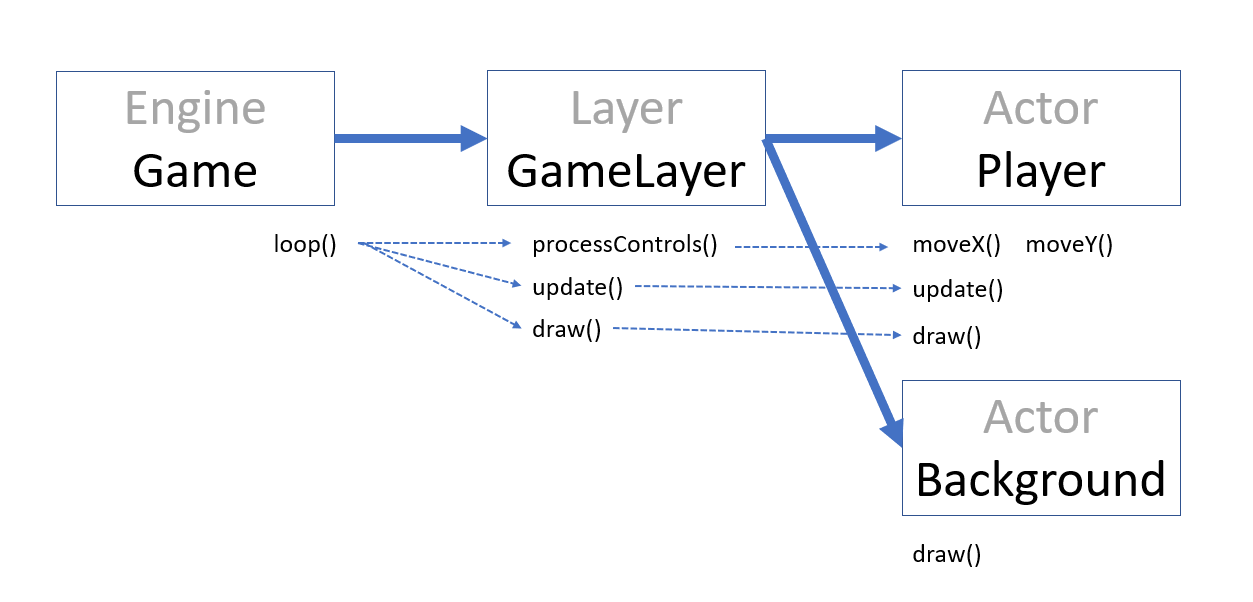
Para darle más velocidad multiplicamos por un número constante (por ejemplo 3) el valor del axis.

|  |
| --- |
| void Player::moveX(float axis) {  vx = axis \* 3;  }  void Player::moveY(float axis) {  vy = axis \* 3;  } |

Probamos a ejecutar la aplicación, nos movemos con las teclas **a d w** y **s.**



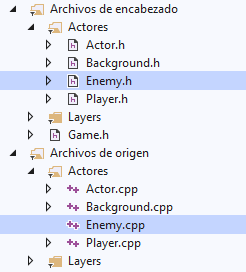
Repasamos la arquitectura actual del juego con un esquema.



* Los Actores más simples como el fondo solo se **dibujan** - **draw()**
* Los Actores “dinámicos/móviles” también se **actualizan - update()**
* Los Actores “controlables” tienen funciones **move()** que dependen de los **controles**

**Enemigos**

Creamos una nueva clase **Enemy** y arrastramos los ficheros **Enemy.h** y **Enemy.cpp** a sus respectivas carpetas **Actors**

****

El **Enemy** también será **Actor**, realmente esta clase se parecerá a Player**.** Va a tener un **constructor** y un método **update()** (más todo lo que hereda de Actor).

|  |
| --- |
| #pragma once  #include "Actor.h"  class Enemy : public Actor  {  public:  Enemy(float x, float y, Game\* game);  void update();  }; |

Abrimos el fichero **Enemy.cpp** y añadimos el **constructor**, este método va a llamar al constructor de su clase padre actor y usar la imagen “res/enemigo.png”), las dimensiones de esta imagen son 36 x 40.

El método **update()**, va a utilizar el valor de **vx** para modificar la **x.** Además, va a comprobar si se sale de los límites de la superficie de renderizado y en ese caso va a invertir su velocidad **vx**.

|  |
| --- |
| #include "Enemy.h"  Enemy::Enemy(float x, float y, Game\* game)  : Actor("res/enemigo.png", x, y, 36, 40, game) {  vx = 1;  }  void Enemy::update() {  x = x + vx;  if (x <= 0 || x >= WIDTH) {  vx = vx \* -1;  }  } |

Ahora hay que añadir los enemigos en la capa del juego **GameLayer**, para ello entramos en **GameLayer.h** y declaramos una lista de enemigos. Es necesario hacer #include tanto de “Enemy.h” como de <list>.

|  |
| --- |
| #include "Layer.h"  #include "Player.h"  #include "Background.h"  #include "Enemy.h"  #include <list>  class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  void init() override;  void processControls() override;  . . .  int controlMoveY = 0;  int controlMoveX = 0;  list<Enemy\*> enemies;  }; |

Ahora entramos en la implementación de **GameLayer.cpp**, nos situamos en el método **init()** y creamos 2 instancias de enemigos y las introducimos en la lista ( lista.push\_back(objeto) ). En el futuro vamos a usar el método **init()** para reiniciar el juego, es conveniente asegurarnos de que la lista de enemigos está vacía.

|  |
| --- |
| void GameLayer::init() {  player = new Player(50, 50, game);  background = new Background("res/fondo.png", WIDTH\*0.5, HEIGHT\*0.5, game);  enemies.clear(); // Vaciar por si reiniciamos el juego  enemies.push\_back(new Enemy(300, 50, game));  enemies.push\_back(new Enemy(300, 200, game));  } |

Los enemy se deben actualizar y dibujar, por eso debemos hacer que el método **update()** de la capa llame al método **update()** de cada enemigo, recordamos que en este método ejecuta la inteligencia y cambia la posición, el enemigo no espera procesar ningún comando del usuario.

|  |
| --- |
| void GameLayer::update() {  player->update();  for (auto const& enemy : enemies) {  enemy->update();  }  cout << "update GameLayer" << endl;  }  void GameLayer::draw() {  background->draw();  player->draw();  for (auto const& enemy : enemies) {  enemy->draw();  }  SDL\_RenderPresent(game->renderer); // Renderiza  } |

Ejecutamos y comprobamos que funciona sin errores.



**NOTA**. Podríamos evitar que los enemigos se salgan parcialmente de la pantalla. Para ello al mover el enemigo hay que comprobar:

* La esquina izquierda **x – width/2** cuando se mueve hacia la izquierda.
* La esquina derecha **x + width /2** cuando se mueve a la derecha



|  |
| --- |
| void Enemy::update() {  x = x + vx;  if (x - width/2 <= 0 || x + width/2 >= WIDTH) {  vx = vx \* -1;  }  } |

**Colisiones**

En general los juegos necesitan calcular muchas colisiones entre diferentes Actores, incluimos en el **Actor.h** la declaración nuevo método **isOverlap(actor)**, va a comprobar la colisión del propio Actor con otro recibido como parámetro.

|  |
| --- |
| class Actor  {  public:  Actor(string filename, float x, float y, Game\* game);  void draw();  bool isOverlap(Actor\* actor);  SDL\_Texture\* texture;  int x; |

Accedemos a **Actor.cpp** e implementamos el cuerpo del método **isOverlap(actor).**

Aplicamos la fórmula para ver si dos cuadrados colisionan (Existen otros algoritmos más complejos para calcular superposiciones, también se podrían utilizar otras formas geométricas en lugar de cuadrados)

|  |  |
| --- | --- |
| MI lado izquierdo más pequeño que TU lado derecho |  |
| MI lado derecho más grande que TU lado izquierdo |  |
| MI lado de arriba más pequeño que TU lado de abajo  *(recordar que la Y negativa va hacia arriba)* |  |
| MI lado de abajo más grande que TU lado de abajo  *(recordar que la Y positiva va hacia abajo)* |  |

|  |
| --- |
| bool Actor::isOverlap(Actor\* actor) {  bool overlap = false;  if (actor->x - actor->width / 2 <= x + width / 2  && actor->x + actor->width / 2 >= x - width / 2  && actor->y + actor->height / 2 >= y - height / 2  && actor->y - actor->height / 2 <= y + height / 2) {  overlap = true;  }  return overlap;  } |

Abrimos **GameLayer.cpp** y ampliamos el método **update()** para que realice detección de colisiones **.** Sí detecta alguna colisión entre el **Player** y un **Enemy** se vuelve a iniciar la capa.

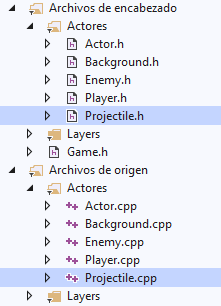
|  |
| --- |
| void GameLayer::update() {  player->update();  for (auto const& enemy : enemies) {  enemy->update();  }  // Colisiones  for (auto const& enemy : enemies) {  if (player->isOverlap(enemy)) {  init();  return; // Cortar el for  }  }  cout << "update GameLayer" << endl;  } |

Ejecutamos la aplicación y comprobamos que la capa se reinicia al detectar la colisión.

****

**Disparos**

Creamos una nueva clase **Projectile**, arrastramos los ficheros **Projectile.h** y **Projectile.cpp** dentro sus respectivas carpetas **Actors.**



Comenzamos accediendo a **Projectile.h** para incluir un método **constructor** y un **update().**

|  |
| --- |
| #pragma once  #include "Actor.h"  class Projectile : public Actor  {  public:  Projectile(float x, float y, Game\* game);  void update();  }; |

Accedemos a **Projectile.cpp** e implementamos, su **constructor**, que hará uso de la imagen “res/disparo\_jugador.png” de 18 x 6. También su método **update()** que lo único que hará será modificar la coordenada **x** según la **vx**.

|  |
| --- |
| #include "Projectile.h"  Projectile::Projectile(float x, float y, Game\* game) :  Actor("res/disparo\_jugador.png", x, y, 18, 6, game) {  vx = 9;  }  void Projectile::update() {  x = x + vx;  } |

Adaptamos el **GameLayer** para que maneje elementos de tipo **Projectile**, como van a ser varios lo que haremos será crear una lista.

Accedemos a **GameLayer.h** y añadimos una nueva lista de **projectiles,** no se nos debe olvidar el #include "Projectile.h" para poder utilizar la clase.

|  |
| --- |
| #include "Enemy.h"  #include "Projectile.h"  #include <list>  class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  void init() override;  . . .  . . .  int controlMoveX = 0;  list<Enemy\*> enemies;  list<Projectile\*> projectiles;  }; |

Ahora abrimos la implementación **GameLayer.cpp**, vamos al método **init()** y limpiamos la lista de proyectiles, para asegurarnos que siempre que se reinicia el juego está vacía.

|  |
| --- |
| void GameLayer::init() {  player = new Player(50, 50, game);  background = new Background("res/fondo.png", WIDTH\*0.5, HEIGHT\*0.5, game);  projectiles.clear(); // Vaciar por si reiniciamos el juego  enemies.clear(); // Vaciar por si reiniciamos el juego |

Al igual que hicimos con los enemigos, habrá que hacer **update()** de todos los proyectiles. , para que estos se muevan.

|  |
| --- |
| void GameLayer::update() {  player->update();  for (auto const& enemy : enemies) {  enemy->update();  }  for (auto const& projectile : projectiles) {  projectile->update();  }  // Colisiones  for (auto const& enemy : enemies) { |

también debemos hacer **draw()** de todos los proyectiles. Aunque el orden de dibujado no es crítico dibujar los **proyectiles** antes que el **player** nos dará un buen resultado visual (parecerá que salen de debajo del jugador)

|  |
| --- |
| void GameLayer::draw() {  background->draw();  for (auto const& projectile : projectiles) {  projectile->draw();  }  player->draw(); |

**¿Cuándo se van a crear los proyectiles?** Nuestra capa ya cuenta con un método **processControls()** y con un control específico **controlShoot** que indica que el jugador quiere disparar(se activaba con la barra espaciadora).

Vamos a delegar la creación del disparo en el **Player**, llamando al método **Player->shoot(),** que nos devolverá un **Projectile** el cual meteremos en la lista **projectiles**. Completamos el método **processControls()**.

|  |
| --- |
| void GameLayer::processControls() {  // obtener controles  keysToControls();  //procesar controles  // Disparar  if (controlShoot) {  Projectile\* newProjectile = player->shoot();  if (newProjectile != NULL) {  projectiles.push\_back(newProjectile);  }  } |

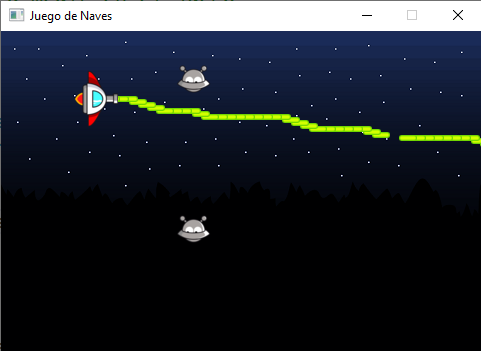
Ahora debemos crear el método **shoot()** en la clase **Player**. Accedemos a **Player.h** y colocamos la signatura del método, debemos recordar el #include “Projectile.h”.

|  |
| --- |
| #include "Actor.h"  #include "Projectile.h"  class Player : public Actor  {  public:  Player(float x, float y, Game\* game);  Projectile\* shoot();  void update();  void moveX(float axis); |

Accedemos a **Player.cpp** para incluir la implementación de **shoot().** Más adelante mejoraremos su funcionamiento.

|  |
| --- |
| Projectile\* Player::shoot() {  return new Projectile(x, y, game);  } |

Ejecutamos la aplicación y comprobamos que no tenga errores, aunque el disparo que se accione pulsando la barra espaciadora probablemente no funcione como esperábamos.

****

**Cadencia de disparo**

La versión actual permite generar un proyectil por iteración del **loop()/update()** . La cadencia hace referencia al tiempo mínimo que debe pasar entre disparo y disparo.

Añadimos dos variables a **Player.h** .

* **shootCadence** mínimo de ejecuciones del update() que deben pasar desde el último disparo para poder crear uno nuevo. Vamos a decir que son 30, este valor es fijo.
* **shootTime** ejecuciones de update() que hay que esperar para poder hacer un nuevo disparo, cuando llega a 0 significa que ya podemos disparar, inicialmente va a ser 0 (no hace falta esperar).

|  |
| --- |
| class Player : public Actor  {  public:  Player(float x, float y, Game\* game);  Projectile\* shoot();  void update();  void moveX(float axis);  void moveY(float axis);  int shootCadence = 30;  int shootTime = 0; |

Accedemos a **Player.cpp** para modificar la implementación de **shoot().** Solo disparamos si no nos hace falta esperar **shootTime==0,** al crear un nuevo proyectil hay que volver a esperar tanto como indique **shootCadence**.

|  |
| --- |
| Projectile\* Player::shoot() {  if (shootTime == 0) {  shootTime = shootCadence;  return new Projectile(x, y, game);  }  else {  return NULL;  }  } |

En cada ejecución de **update()** se debe restar 1 al tiempo de espera **shootTime**.

|  |
| --- |
| void Player::update() {  if (shootTime > 0) {  shootTime--;  }  x = x + vx;  y = y + vy;  } |

Ejecutamos la aplicación comprobamos que la cadencia funciona.



**Eliminar enemigos (Overlap entre Enemy y Shoot)**

El **GameLayer** se va a encargar de detectar las colisiones entre actores, entramos en **GameLayer.cpp** y nos dirigimos al método **update(),** anteriormente ya comprobamos Overlaps entre el **Player y Enemy,** vamos a incluir una nueva comprobación de colisión nueva entre: **Enemy y Projectile**.

|  |
| --- |
| **Recorrer listas y eliminar elementos.** En general eliminar elementos a la vez que se recorre una lista puede traernos algunos problemas, si no lo hacemos con suficiente cuidado se pueden originar excepciones por acceso a elementos eliminados.  Para tratar de reducir la posibilidad de fallos (sobre todo cuando extendamos la funcionalidad del juego), vamos a recorrer las listas, anotar los elementos que queremos eliminar en una **lista auxiliar** en la que se almacenan los disparos a eliminar y luego recorrer esa lista auxiliar, eliminando de la lista original cada uno de los elementos de la auxiliar. |

Vamos a crear dos listas temporales **deleteEnemies** y **deleteProjectiles**, las vamos a utilizar para introducir en ella los enemigos y proyectiles que queramos eliminar.

|  |
| --- |
| // Colisiones  for (auto const& enemy : enemies) {  if (player->isOverlap(enemy)) {  init();  return; // Cortar el for  }  }  // Colisiones , Enemy - Projectile    list<Enemy\*> deleteEnemies;  list<Projectile\*> deleteProjectiles;  for (auto const& enemy : enemies) {  for (auto const& projectile : projectiles) {  if (enemy->isOverlap(projectile)) {  deleteEnemies.push\_back(enemy);  deleteProjectiles.push\_back(projectile);  }  }  }    cout << "update GameLayer" << endl;  } |

Importante, es muy posible que en otras partes del código queramos eliminar elementos (introducirlos en las listas **deleteEnemies** y **deleteProjectiles),** cabría la posibilidad remota de que se metiese el mismo elemento en la lista 2 veces (por ejemplo, si se elimina por dos condiciones diferentes, chocó contra un disparo y una bomba). Por lo tanto para evitar futuros problemas y excepciones graves vamos a comprobar que el objeto no estaba ya en la lista, para ello podemos usar la función **std::find()**

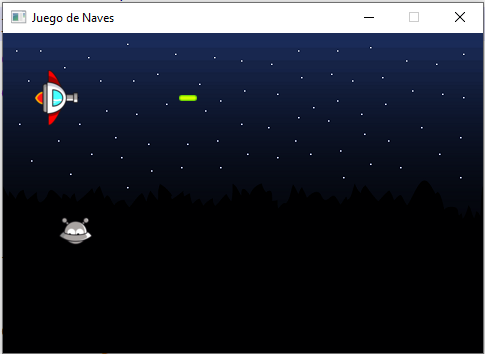
|  |
| --- |
| for (auto const& enemy : enemies) {  for (auto const& projectile : projectiles) {  if (enemy->isOverlap(projectile)) {  bool pInList = std::find(deleteProjectiles.begin(),  deleteProjectiles.end(),  projectile) != deleteProjectiles.end();  if (!pInList) {  deleteProjectiles.push\_back(projectile);  }  bool eInList = std::find(deleteEnemies.begin(),  deleteEnemies.end(),  enemy) != deleteEnemies.end();  if (!eInList) {  deleteEnemies.push\_back(enemy);  }  }  }  } |

Una vez hemos recorrido las dos listas ya disponemos de todos los elementos que queremos borrar en las listas temporales **deleteEnemies** y **deleteProjectiles.** Habrá que recorrer estas listas y eliminar los elementos de sus listas originales **enemies** y **projectiles**.

|  |
| --- |
| for (auto const& enemy : enemies) {  for (auto const& projectile : projectiles) {  if (enemy->isOverlap(projectile)) {  bool pInList = std::find(deleteProjectiles.begin(),  deleteProjectiles.end(),  projectile) != deleteProjectiles.end();  if (!pInList) {  deleteProjectiles.push\_back(projectile);  }  bool eInList = std::find(deleteEnemies.begin(),  deleteEnemies.end(),  enemy) != deleteEnemies.end();  if (!eInList) {  deleteEnemies.push\_back(enemy);  }  }  }  }  for (auto const& delEnemy : deleteEnemies) {  enemies.remove(delEnemy);  }  deleteEnemies.clear();  for (auto const& delProjectile : deleteProjectiles) {  projectiles.remove(delProjectile);  }  deleteProjectiles.clear(); |

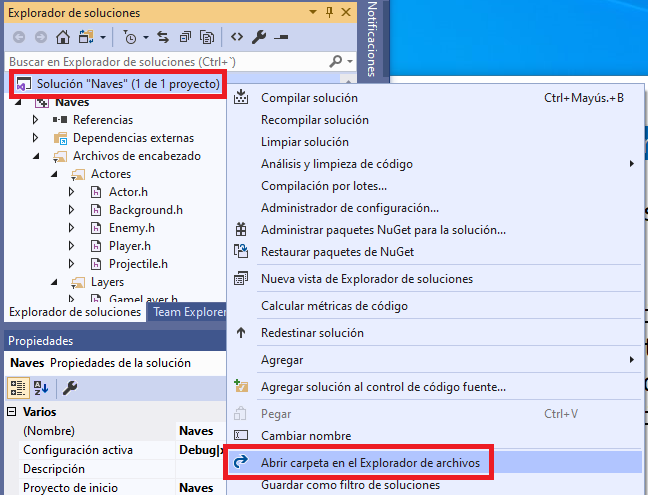
|  |
| --- |
| **Optimizaciones al recorrer colecciones.** Dado el escaso tiempo del que disponemos y la simpleza del juego no vamos a poner énfasis en “optimizar” el juego, pero lo ideal sería recorrer las colecciones el menor número de veces posible y así reducir el consumo de memoria, si se puede hay que reaprovechar un mismo bucle for para comprobar varias cosas. |

Ejecutamos la aplicación y comprobamos que funciona de la forma esperada y sin errores.

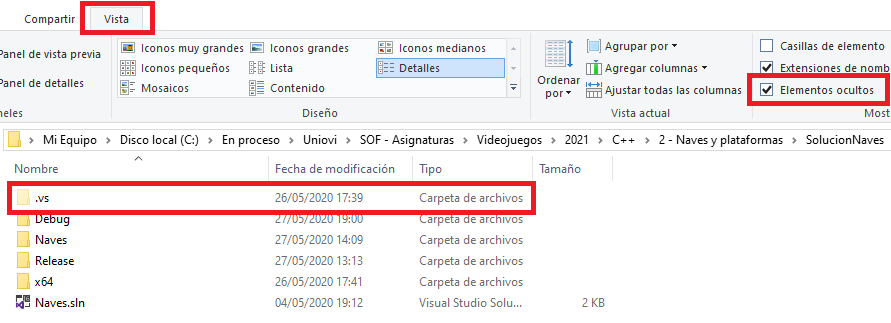


**\*Llevar proyecto a otro equipo**

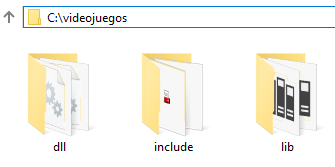
Basta con copiar la carpeta raíz en la que está la solución desde el propio Windows, si no recordamos cual se puede obtener desde Visual Studio.



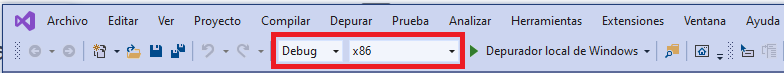
**La carpeta ocupa 1GB o más!,** es por culpa de la carpeta .vs (oculta) que Visual Studio utiliza para meter compilaciones y otros archivos, se puede borrar sin problema.

****

**\*Recordar:** el ordenador al que lleve el proyecto debe tener los dll, include y libs que usa el proyecto como dependencias situado en C:\videojuegos

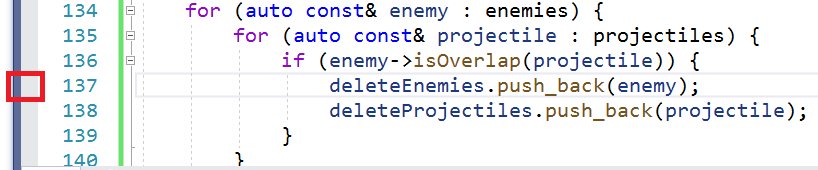


En el momento de ejecutar el proyecto hay que seleccionar la opción **x86** (probablemente se haya desmarcado, tenemos que hacerlo porque elegido las librerías de 32.

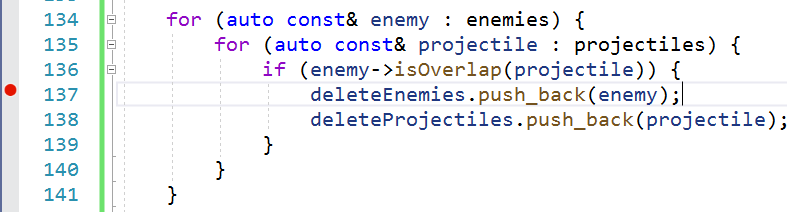


**\*Depurar desde Visual Studio (Para hacer en clase)**

Para depurar lo único que tenemos que hacer es colocarlos en la línea en la que nos interese detener la ejecución y hacer click en la zona gris.



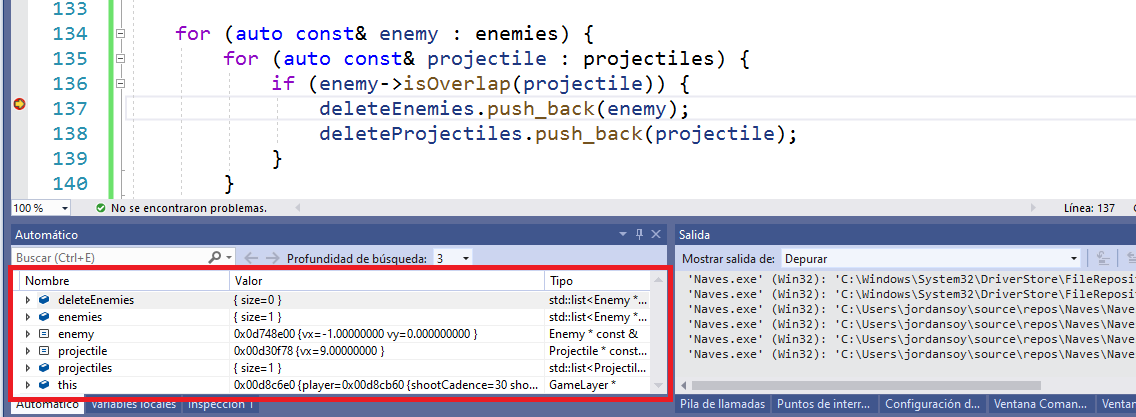
Veremos que aparece un punto rojo, indica que la ejecución se detendrá en ese punto de ruptura.



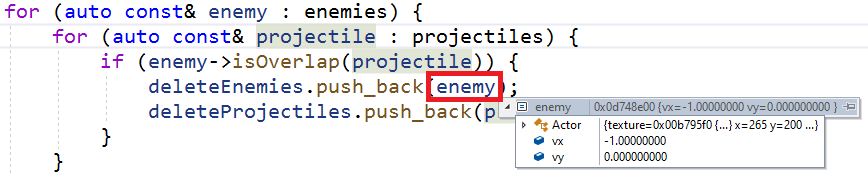
A partir de este momento si ejecutamos el proyecto el modo debug la ejecución se detendrá al llegar a la línea marcada (justo después de detectar la colisión entre un enemigo y un proyectil).



En el juego disparamos a un enemigo para que se detenga la ejecución. Sabremos que ha llegado a la línea porque el punto rojo cambiará al icono . En la parte inferior izquierda podemos inspeccionar el valor de todas las variables.



Sí colocamos el puntero del ratón sobre una variable en el código podremos inspeccionar su valor (aparece en una ventana emergente), por ejemplo, revisamos el valor de enemy.



Las acciones más comunes para permitir que continúe la ejecución son:



**Continuar**, que deja correr la ejecución hasta llegar al siguiente punto de ruptura.



**Paso a paso por procedimiento**, avanza a la siguiente línea de código, sin entrar dentro de la implementación de los métodos.



**Paso a paso por instrucciones,** avanza a la siguiente línea de código, entrando dentro de la implementación de los métodos.

|  |
| --- |
| **Nuevos puntos de interrupción, en** cualquier momento incluso durante la ejecución podemos añadir y eliminar nuevos puntos de interrupción, para eliminar un punto solo hay que hacer click sobre él. |

**\*Lógica simple para generar Enemigos (Para hacer en clase)**

Vamos a modificar el movimiento del enemigo, accedemos **Enemy.cpp** y modificamos la implementación de **update(),** únicamente se moverán hacia la izquierda.

|  |
| --- |
| void Enemy::update() {  vx = -1;  x = x + vx;  } |

Ahora vamos a modificar el **GameLayer** para hacer que aparezcan nuevos enemigos cada 110 ejecuciones del **update()**.

Lo primero es acceder a **GameLayer.h** y añadir la variable **newEnemyTime**.

|  |
| --- |
| class GameLayer : public Layer  {  public:  GameLayer(Game\* game);  . . .  . . .  void keysToControls();  int newEnemyTime = 0; |

Ahora abrimos **GameLayer.cpp** y modificamos la implementación de **update().** La variable **newEnemyTime** va a actuar como una “cuenta atrás” (es básicamente como la cadencia de los disparos), en cada ejecución reducimos uno su valor, cada vez que llega a **0** creamos un enemigo y establecemos su valor a 110 (para que haya que volver a esperar otras **110** ejecuciones).

Para crear los enemigos vamos a usar valores **x** e **y** aleatorios

* Por ejemplo, una **x** entre 500 y 600 (El lado derecho de la superficie de renderizado es 480)
* Por ejemplo, una **y** entre 60 y 300 (La superficie de renderizado mide 320 de alto, así no estarán cerca del borde)

|  |
| --- |
| void GameLayer::update() {  // Generar enemigos  newEnemyTime--;  if (newEnemyTime <= 0) {  int rX = (rand() % (600-500)) + 1 + 500;  int rY = (rand() % (300-60)) + 1 + 60;  enemies.push\_back(new Enemy(rX, rY, game));  newEnemyTime = 110;  }  player->update(); |

Como los **enemigos** se crean en la parte derecha (fuera de la superficie de renderizado) y se mueven hacia la izquierda parecerá que entran en la pantalla.



|  |
| --- |
| **Mejoras, generar más enemigos a medida que más matamos.** Podríamos guardar en una variable en número de enemigos eliminados , de forma que cuantos más enemigos hayamos eliminado más aparezcan, o que aparezcan cada menos tiempo. |

|  |
| --- |
| **Mejoras, eliminar enemigos.** Podríamos hacer que los enemigos que se salen por la parte izquierda de la pantalla fueran eliminados, algo similar se podría hacer con los disparos (pero con la parte derecha). |

**\*Botón para salir (Para hacer en clase)**

Vamos a hacer que al pulsar la tecla escape se salga del juego, para que el juego deje de ejecutarse basta con poner a false la variable **loopActive** que hay dentro de **Game**.

Prácticamente todas las clases tienen una referencia a Game, vamos a entrar en **GameLayer.cpp** y modificar el método **keysToControls**.

|  |
| --- |
| void GameLayer::keysToControls(SDL\_Event event) {  if (event.type == SDL\_KEYDOWN) {  int code = event.key.keysym.sym;  // Pulsada  switch (code) {  case SDLK\_ESCAPE:  game->loopActive = false;  break;  case SDLK\_d: // derecha |

|  |  |
| --- | --- |
| **Botón x de cerrar ventana.**  Podemos añadir otra comprobación para ver si el evento es de tipo SDL\_QUIT que hará que funcione el botón de cerrar la ventana, no es una tecla sino un tipo de evento diferente.   |  | | --- | | if (event.type == SDL\_QUIT) {  game->loopActive = false;  } | |

**\*Maximizar pantalla (Para hacer en clase)**

Vamos a implementar un sistema que permita maximizar la pantalla pulsando la **tecla 1**. Para ello necesitamos y dos variables y un nuevo método en el motor del juego **Game**.

Accedemos a **Game.h** y agregamos el método **scale()** y la variable booleana **scaledToMax** . Creamos también la variable **scaleLower** que va a almacenar el actual **factor de escalado**, ahora mismo 1, mucho más adelante usaremos esta variable desde otras clases para transformar coordenadas de ratón.

|  |
| --- |
| class Game  {  public:  Game();  void loop();  void scale();  bool scaledToMax = false;  float scaleLower = 1;  SDL\_Window\* window; // ventana  SDL\_Renderer\* renderer; // renderizador  bool loopActive; // Juego activo  Layer\* gameLayer;  }; |

Accedemos a la implementación en **Game.cpp** e implementamos el nuevo método **scale()**.

* Cada vez que se ejecuta el método se invierte la variable **scaledToMax** que indica si la pantalla está escalada o no.
* Si hay que escalar el juego se obtiene la pantalla actual con **SDL\_GetCurrentDosplayMode**, una vez tenemos la pantalla obtenemos su resolución y la dividimos entre el WIDTH y el HEIGHT actual del juego. Esto nos va a dar cuanto hay que escalar la superficie de renderizado para que ocupe toda la pantalla. Debemos tener cuidado porque probablemente nos dará un **scaleX** y un **scaleY** diferente, es muy raro que la dimensión de la pantalla sea proporcional a nuestro WITH y HEIGHT.

Para que la renderización del juego no se “deforme” hay que quedarse con la escala más pequeña de las dos (entre **scaleX** y **scaleY**), la almacenaremos en la variable **scaleLower** y la aplicaremos en ambos ejes, redimensionando primero la ventana **SDL\_SetWindowSize** y luego el render con **SDL\_RenderSetScale.**

* Si no hay que escalar la pantalla basta con volver a la resolución por defecto WITH y HEIGHT que se corresponde con el factor de escalado 1.

|  |
| --- |
| void Game::scale(){  scaledToMax = !scaledToMax;  if (scaledToMax) {  SDL\_DisplayMode PCdisplay;  SDL\_GetCurrentDisplayMode(0, &PCdisplay);  float scaleX = (float)PCdisplay.w / (float)WIDTH;  float scaleY = (float)PCdisplay.h / (float)HEIGHT;  // Necesitamos la menor de las 2 escalas para no deformar el juego  scaleLower = scaleX;  if (scaleY < scaleX) {  scaleLower = scaleY;  }  // Cambiar dimensiones ventana  SDL\_SetWindowSize(window, WIDTH \* scaleLower, HEIGHT \* scaleLower);  // Cambiar escala del render  SDL\_RenderSetScale(renderer, scaleLower, scaleLower);  }  else { // Escala Original  scaleLower = 1;  // Cambiar dimensiones ventana  SDL\_SetWindowSize(window, WIDTH, HEIGHT);  // Cambiar escala del render  SDL\_RenderSetScale(renderer, 1, 1);  }  } |

Ahora vamos a hacer que, desde el **GameLayer**, al pulsar la **tecla 1** se active el escalado. Accedemos a GameLayer.cpp y buscamos el método **keysToControls()**, añadimos un nuevo caso para la tecla 1 (SDLK\_1) a través de la referencia a **game** llamamos al método **game->scale().**

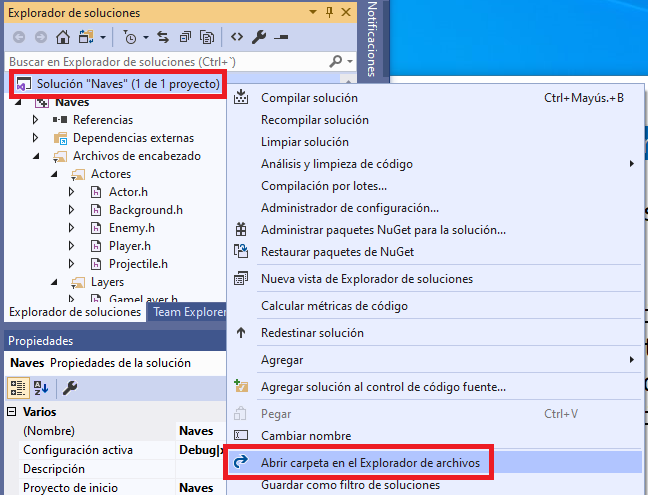
|  |
| --- |
| void GameLayer::keysToControls(SDL\_Event event) {  if (event.type == SDL\_KEYDOWN) {  int code = event.key.keysym.sym;  // Pulsada  switch (code) {  case SDLK\_ESCAPE:  game->loopActive = false;  break;  case SDLK\_1:  game->scale();  break;  case SDLK\_d: // derecha  controlMoveX = 1; |

**Entrenamiento (No hay que entregarlo, es para practicar)**

Antes de continuar con el guion vamos a proponer algunas modificaciones para entrenarnos.

Debemos crear un **proyecto nuevo** y **copiar el contenido a nivel de fichero** de este en el otro para no estropearlo. **En las siguientes sesiones necesitamos el proyecto sin ampliaciones**

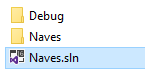
Para sacar una copia del proyecto basta con acceder a la carpeta raíz en la que está **la solución** proyecto en el explorador de Windows y copiarla.



Basta con sacar una copia.



Si abrimos la carpeta de la solución encontramos el fichero **Naves.sln** que se abre directamente la solución.



**Controlar que la Nave no se salga de los limites**

Ajustar el método **actualizar()** para que la “x” no pudiera ser menor que 0 ni mayor de HEIGHT, y la “y” no pudiera ser menos que 0 ni mayor de WIDTH.

**Elementos recolectables**

Crear un nuevo Actor que represente un “elemento recolectable”, si la nave colisiona con él este desaparecerá.

**Disparos finitos**

Crea un contador de disparos en el enemigo, solo podrá disparar N veces. Cada vez que se obtiene un elemento recolectable permitimos realizar un disparo más.

**Enemigos aguantan 3 disparos**

Incluir una variable vida en los enemigos, que se reduce en uno cada vez que reciben un disparo, solo se eliminan del juego cuando su vida llega a 0. Es recomendable eliminar el **Projectile** cuando colisiona con el **Enemy**, así no continúa colisionando más veces.