Como crear un motor gráfico para videojuegos de cero.

Introducción.

Capítulo 1: Creado el esqueleto del motor gráfico.

Introducción.

El primer paso será crear el esqueleto de nuestro futuro motor gráfico. Dicho esqueleto se compondrá de los siguiente módulos:

Main: Es el archivo donde se inicializará el motor y el que controlará el iniio y la salida del mismo.

Application: Desde application se llamará a los diferentes módulos en un orden determinado y establecido para su ejecución.

ModuleWindow: En este módulo es donde se creará y mostrará la ventana de la aplicación.

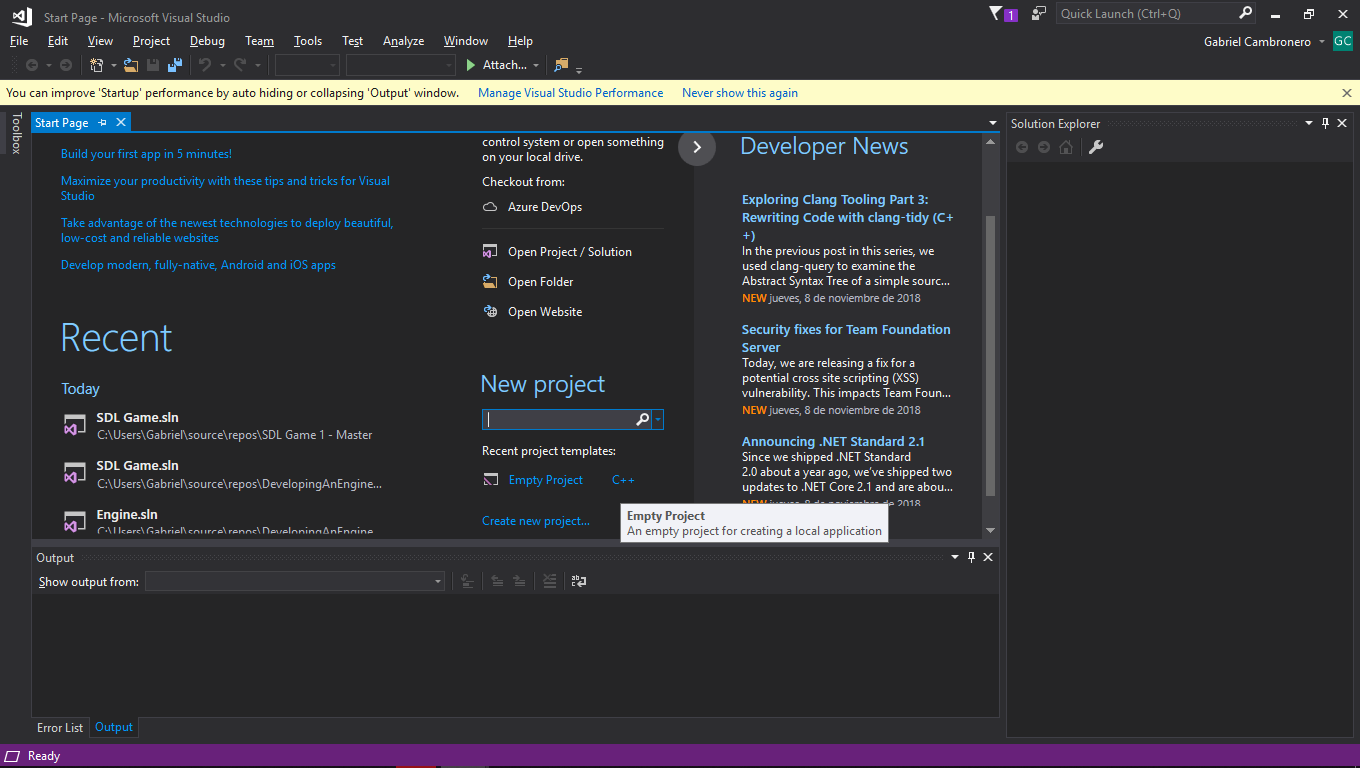
ModuleRender: Este módulo será el encargado de pintar en la ventana los objetos.

ModuleTextures: Este módulo será el encargado de cargar las diferenes texturas de los objetos.

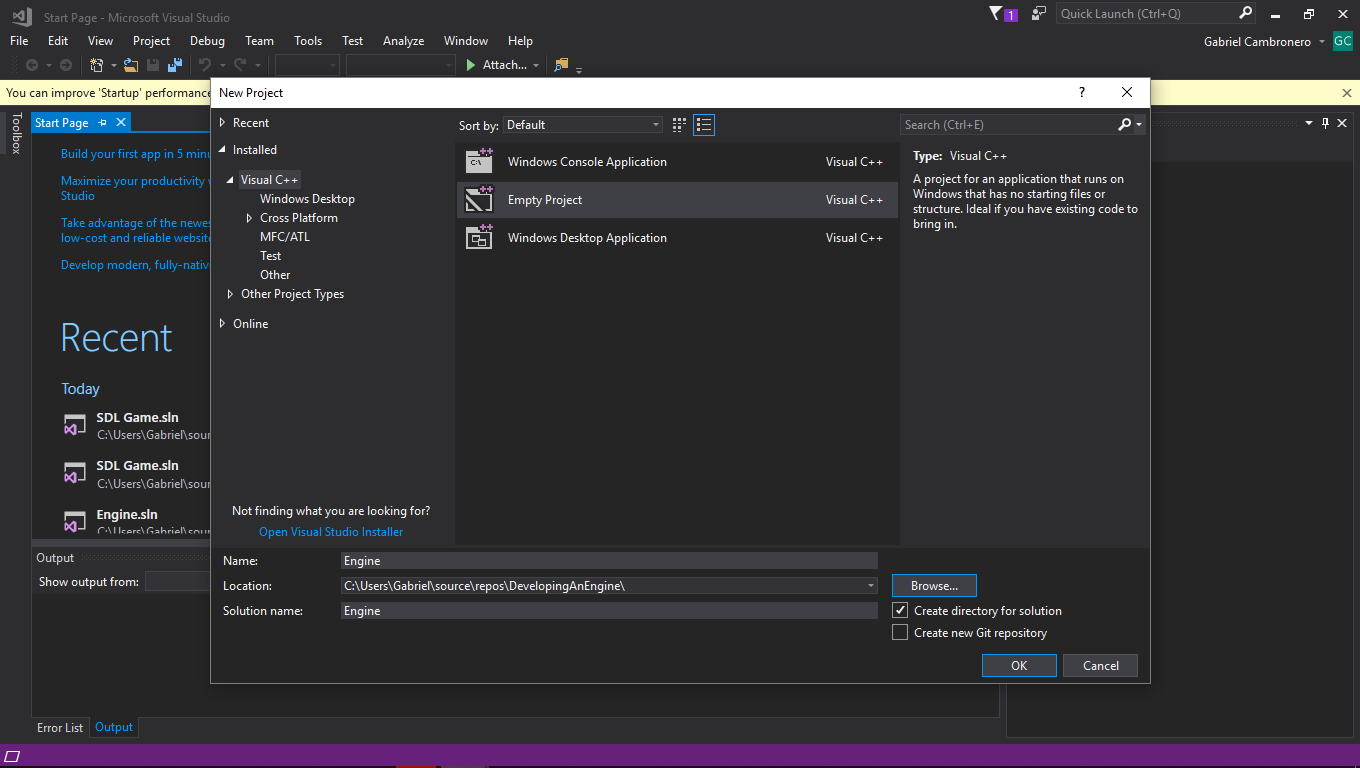
ModuleInput: Este módulo será el encargado de las actualizaciones de los diferentes módulos.

Creación del proyecto:

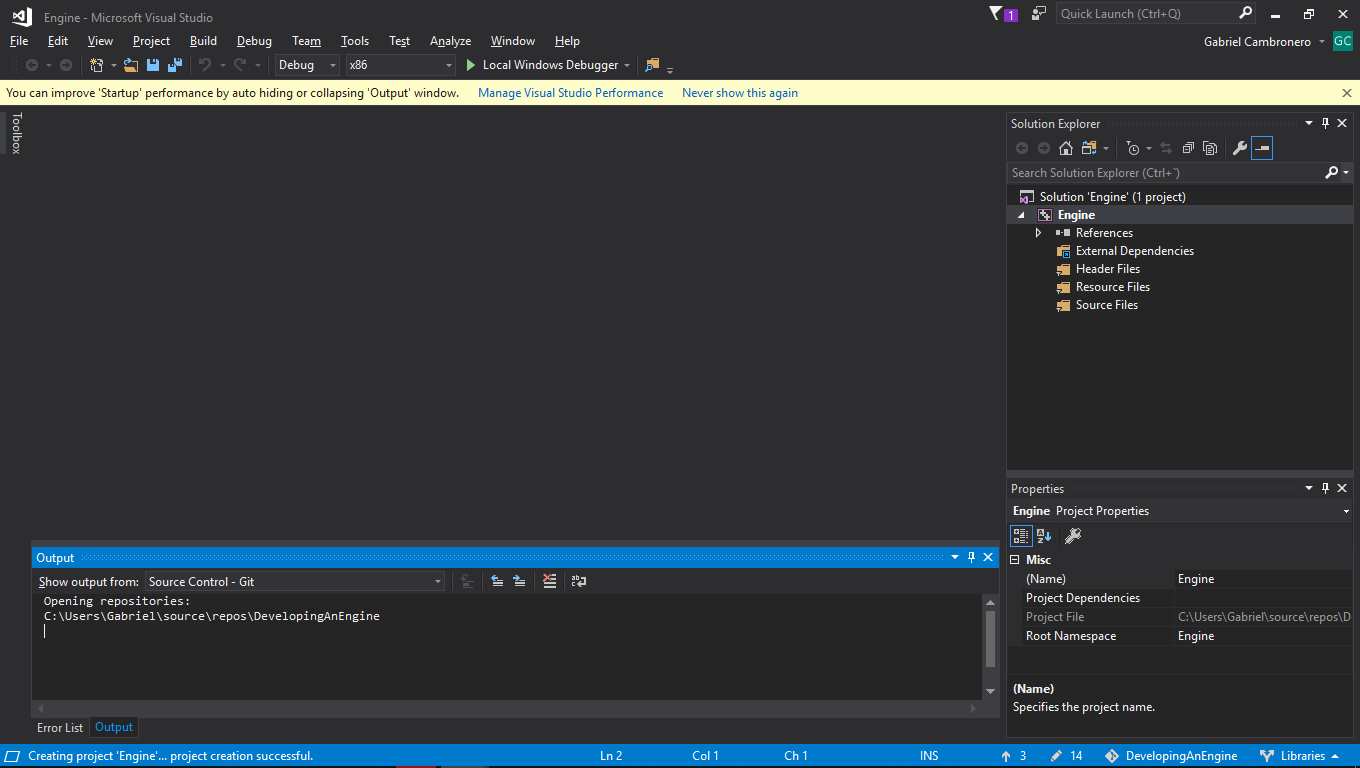
El primer paso será la creación del proyecto para desarrollar el motor. Para ello ejecutaremos Visual Studio. En la pantalla de presentación (Start page), por defecto de Visual Studio, se seleccionará la opción *Empty Projec*, figura 1.



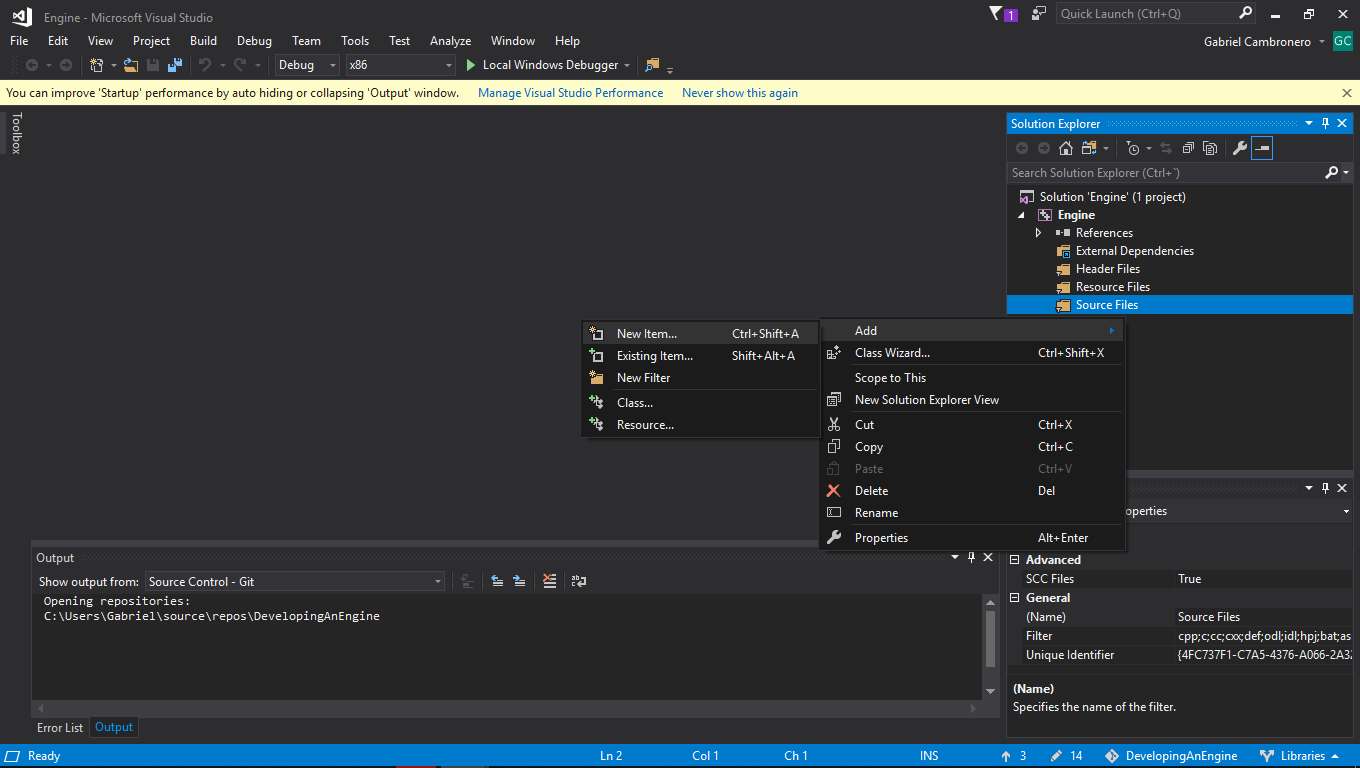
Esto nos mostrará una segunda ventana donde seleccionaremos la opción Empty Project.



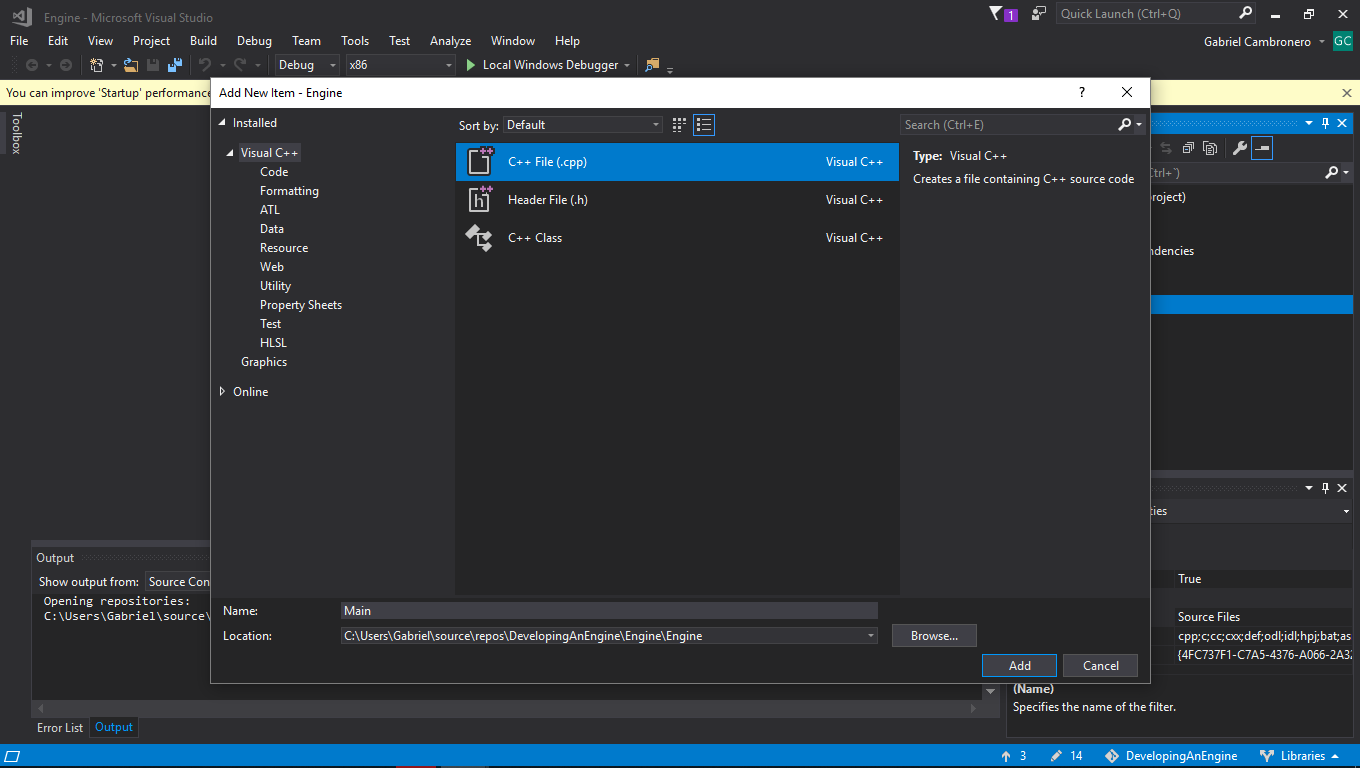
Con esto ya habremos creado el proyecto donde desarrollaremos el motor gráfico de videojuegos, tal y como se muestra en la figura 3.



El siguiente paso será la creación de Main.cpp. Para ello situaremos el ratón sobre la carpeta source files y pulsaremos el botón derecho del ratón seleccionando la opción Add-> New ítem…



Con o que aparecerá una nueva pantalla donde se seleccionará la opción C++ File (.cpp) y pulsaremos sobre Ad.



Con esto habremos creado el archivo Main.cpp. Otra forma de crear un archivo con Visual Studio es ir a File en el menú principal, seleccionar New y pulsar sobre File…

MAIN.CPP

Ahora que ya tenemos creado el nuevo archivo, pasaremos a introducir el código base para el módulo Main.cpp.

Lo primero será crear los includes para el fichero main, por lo que introduciremos el siguiene código:

*#include <stdlib.h>*

*#include "Application.h"*

*#include "Globals.h"*

*#include "SDL/include/SDL.h"*

*#pragma comment( lib, "SDL/libx86/SDL2.lib" )*

*#pragma comment( lib, "SDL/libx86/SDL2main.lib" )*

Con este Código estamos diciendo a Main que incluya los siguientes módulos:

Stdlib.h: Es la cabecera de una librería interna de Visual Studio.

Applicaion.h: Es la cabecera módulo donde se van llamando a los diferentes módulos del futuro motor gráfico.

Globals.h: Es el fichero de cabecera donde se declaran las variables globales que se usarán en los diferentes módulos.

SDL/include/SDL.h: Es la cabecera de la librería SDL. Antes de SDL.h se está especificando la ruta donde encontrar el archivo. Aunque esto no sea necesario, tal y como se explicará más adlante, si es muy recomendable, para informar que se trata de una librería externa y especificarse donde puede encontrarse por si hay un problema con la misma o se desea actualizar a una versión superior.

#pragma comment (lib, “SDL/libx86/SDL2.lib”): Es un comentario sobre la librería que se está utilizando.

#pragma comment (lib, “SDL/libx86/SDL2main.lib”): Es un comentario sobre la librería que se está utilizando.

Tras declarar los includes crearemos un enum con todos los estados que podemos tener durante la ejecución del motor gráfico. Para ello introduciremos las siguientes líneas a continuación:

*enum main\_states*

*{*

*MAIN\_CREATION,*

*MAIN\_START,*

*MAIN\_UPDATE,*

*MAIN\_FINISH,*

*MAIN\_EXIT*

*};*

Según esta sentencia, nuestro motor gráfico sólo puede tener los siguientes estados: MAIN\_CREATION: reacción del motor gráfico.

MAIN\_STAR: Inicialización del motor gráfico.

MAIN\_UPDATE: actualización del motor gráfico.

MAIN\_FINISH: Finalización del motor gráfico.

MAIN\_EXIT: Salida del motor gráfico.

Para no tener que teclear todo el rato la palabra Aplication, creamos un puntero que inicializaremos a null escribiendo la siguiente línea a continuación de la declaración del enum:

*Application\* App = NULL;*

Todas estas declaraciones se han realizado fuera del bloque main, el cual va a empezar a declararse ahora.

El bloque main debe devolvernos un integer (int), por lo que se declarará de la siguiente forma:

*int main(int argc, char \*\* argv)*

Con estas líneas se está indicando al bloque main que devolverá in integer y hemos de pasarle un integer y un char.

Lo siguiente que se debe hacer es declarar dos variables que serán necesaras para la ejecución del fichero main. Esto se hará introduciendo las siguientes líneas:

*int main\_return = EXIT\_FAILURE;*

*main\_states state = MAIN\_CREATION;*

En estas dos líneas estamos declarando la variable main\_return, que será de tipo int y tendrá el valor de la constante EXT\_FAILURE. También se está declarando la variable state, que será del tipo main\_state y tendrá el valor de MAIN\_CREATION. Debe notarse que main\_state es el enum que se ha declarado fuera del bloque main y que MAIN\_CREATION es uno de los valores que el enum main\_states tiene en su interior.

Tras estas declaraciones introduciremos la lógica que controlará el fichero y la ejecución del motor gráfico. Para ello utilizaremos un bucle while. Este bucle recibirá la cariable state que hará que dependiendo de su valor se ejecute o no e bucle while. Dentro de este bucle crearemos un bucle switch que es el que tomará la decisión de que parte debe ejecutarse dependiendo del valor de state.

En las siguientes líneas pueden verse los bucles while y switch:

*while (state != MAIN\_EXIT)*

*{*

*switch (state)*

*{*

*case MAIN\_CREATION:*

*LOG("Application Creation --------------");*

*App = new Application();*

*state = MAIN\_START;*

*break;*

*case MAIN\_START:*

*LOG("Application Init --------------");*

*if (App->Init() == false)*

*{*

*LOG("Application Init exits with error -----");*

*state = MAIN\_EXIT;*

*}*

*else*

*{*

*state = MAIN\_UPDATE;*

*LOG("Application Update --------------");*

*}*

*break;*

*case MAIN\_UPDATE:*

*{*

*int update\_return = App->Update();*

*if (update\_return == UPDATE\_ERROR)*

*{*

*LOG("Application Update exits with error -----");*

*state = MAIN\_EXIT;*

*}*

*if (update\_return == UPDATE\_STOP)*

*state = MAIN\_FINISH;*

*}*

*break;*

*case MAIN\_FINISH:*

*LOG("Application CleanUp --------------");*

*if (App->CleanUp() == false)*

*{*

*LOG("Application CleanUp exits with error -----");*

*}*

*else*

*main\_return = EXIT\_SUCCESS;*

*state = MAIN\_EXIT;*

*break;*

*}*

*}*

Atención a la línea: *App->Init()*. Aquí se está utilizando el operador -> para indicar al puntero App (puntero de Application) que acceda al atributo Init() que se está declarado en Application.cpp.

A continuación del bucle while se deben las siguientes líneas:

*delete App;*

*LOG("Bye :)\n");*

*return main\_return;*

En la primera línea se está borrando Application con la sentencia delete. Debe limpiarse memoria después de la ejecución de cada bloque y cuando este ya no sea necesario, ya que C/C++ no dispone de un recolector de basura como por ejemplo java.

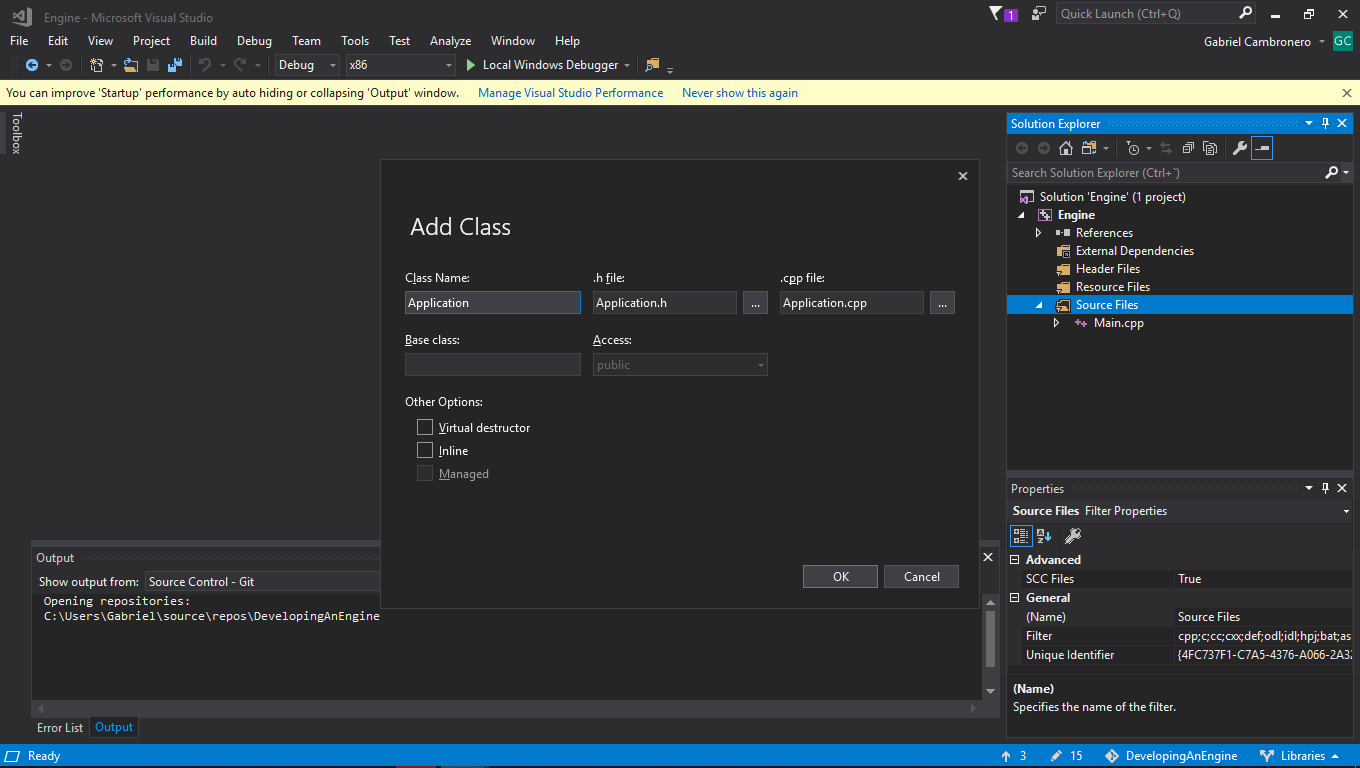
LOG nos devuelve mediante la escritura en log de un mensaje de despedida y return nos devuelve el la variable main\_return. Recordemos que en la declaración de main() se ha especificado que debía devolverse un int al término de su ejecución.

Para ver el Código entero entre aquí.

APPLICATION.CPP y APPLICATION.H

Para la creación de Application, el módulo principal del motor gráfico será necesario crear también un fichero de cabecera. Este ficher puede crearse de la misma forma que un fichero cpp o pueden crearse ambos simultáneamente si en vez de seleccionar New File… seleccionamos Clas…

Al realizar esto aparecerá la siguiente pantalla.



Donde deberá indicarse el nombre de la nueva clase. Si se ha realizado correctamente, se habrán generado dos archivos con el mismo nombre, uno con la terminación .cpp y otro con la terminación .h.

Ahora que tenemos creada la nueva clase Application, empezaremos introduciendo el código del fichero header (cabecera). En este fichero se para declarar las funciones y la mayoría de las variables que utilizaremos más tarde en el fichero Application.cpp.

La primera línea por defecto que ha sido insertada por Visual Studio al crear la clase es: #pragma once. Este uso de pragma no es estándar y lo modificaremos por las siguientes dos líneas de código:

*#ifndef \_\_Application\_h\_\_*

*#define \_\_Application\_h\_\_*

En estas líneas estamos utilizando las directivas *#ifndef* y *#define*. La primera comprueba si no está definida la macro-identificador, en este carp Application.h, y en caso de que no lo esté, (*#ifndef* = 1)la directiva *#define* la para poder utilizarla. La directiva *#ifndef* debe de cerrarse con *#endif*.

Tras estas líneas definiremos los include del archivo de cabecera:

*#include<list>*

*#include "Globals.h"*

*#include "Module.h"*

Tras estos entramos a declarar las clases ModuleRender, ModuleWindow, ModuleTextures y ModuleInput. Y definimos las clase Application. Esto se hace tal y como se muestra en las siguientes líneas:

*class ModuleRender;*

*class ModuleWindow;*

*class ModuleTextures;*

*class ModuleInput;*

*class Application*

Al inicio de la clase introduciremos la palabra *public*, y declararemos las funciones Application y su destructor, para ello incluiremos las siguientes líneas:

*public:*

*Application();*

*~Application();*

Tras estas líneas declaramos las funciones a utilizar de la siguiente forma:

*bool Init();*

*bool CleanUp();*

*update\_status Update();*

En las líneas anteriores se han declarado dos funciones que: *Init()* y *CleanUp()* que deben devolver un valor booleano (*bool*) al finalizar su ejecución. La tercera función: *Update()* debe devolver un valor *update\_status*. *Update\_status* no es un variable primitiva, y se explicará su función cuando se explique la clase Globals.

Tras estas declaraciones pasamos a declarar los punteros que se utilizarán en el fichero Application.cpp. Para ello volveremos a introducir la palabra *public*. Si bien no es necesaria volver a introducir *public*, si que es recomendable, ya que hace que el código sea más entendible y legible, al establecer una separación entre la declaración de las funciones y de los punteros. Por tano, las siguientes líneas introducir serán:

*public:*

*ModuleRender\* renderer = nullptr;*

*ModuleWindow\* window = nullptr;*

*ModuleTextures\* textures = nullptr;*

*ModuleInput\* input = nullptr;*

En estas líneas se han declarado los punteros que se utilizarán en el fichero Application.cpp y a su vez, se han inicializado todos a *null*.

Tras declara las partes públicas se pasa a declarar las partes privadas de la clase. Esta parte privada sólo podrá ser accesible desde esta clase, no siendo visible para el resto de clases. Para declarar la parte privada se ha de utilizar la palabra reservada *private*. Se declarará como privada una lista de punteros de los módulos de la siguiente foram:

*private:*

*std::list<Module\*> modules;*

Ya solo nos queda realizar tres cosas; cerrar la clase *Application*, declarar el puntero *App* como una variable externa y cerrar el identificador *#ifndef*. Todo esto se realizará añadiendo las siguientes líneas de código:

*};*

*extern Application\* App;*

*#endif // \_\_Application\_h\_\_*

La palabra clave *extern* permite declara una variable de forma global y que será accesible desde cualquier clase debido a que se trata de una variable declarada como externar.

Si se observa el código detenidamente, se observará que Visual Studio marca errores en ciertas líneas, esto no nos debe de preocupar ahora mismo, pues se está haciendo referencias a clases que aún no han sido definidas como por ejemplo *ModuleWindow*.

Tras escribir el código del fichero *Application.h*, empezaremos a introducir el código del fichero *Application.cpp*. Así que guardemos el fichero *Application.h* y abramos el fichero *Application.cpp*.

Debido a que esta clase se llamará al inicio de la ejecución del programa por parte de main una única vez, la inicio del fichero deben ponerse las siguiente dos líneas:

*#ifndef \_\_Application\_cpp\_\_*

*#define \_\_Application\_cpp\_\_*

Estas dos líneas se podrán siempre al inicio de los fichero de cabecera de las clases, pero no en los fichero donde se definen las clases, siendo éste el único caso.

Tras estas líneas se declararán todos los includes que serán necesarios para la ejecución de esta clase. Esto se realizará incluendo las siguientes líneas de código:

*#include "Application.h"*

*#include "ModuleWindow.h"*

*#include "ModuleRender.h"*

*#include "ModuleTextures.h"*

*#include "ModuleInput.h"*

En estas líneas se están llamando a todos los ficheros de cabecera de las clases que se van a necesitar. El siguiente paso a realizar es declara el contexto donde se definirán los nombres de las variables, funciones, clases,… Para ello se utilizará la palabra reservada *namespace* de la siguiente forma:

*using namespace std;*

Existe cierta desavenencia sobre si es apropiado o no la inclusión de esta línea de código ya que en un futuro puede conflictos de nombre con otras librerías que sean incluidas y se dice que es mejor llamar a la parte de la librería que se vaya a utilizar directamente de la forma *std::cout*, por ejemplo. Pese a todo, aquí se ha decidido utilizarla ya que las librerías que se utilizarán no deberían entrar en conflicto en el espacio de nombres.

Tras esta línea se empieza a definir la función principal de la clase Application, que llevará su mismo nombre. Debe escribirse por lo tanto la siguiente línea:

*Application::Application()*

Tras esto se define el bloque de *Application*. En este bloque se realizarán las llamadas a los distintos módulos que compondrán nuestro futuro motor gráfico. El orden de llamada de los módulos es importante, ya que estos serán llamados en ese orden y serán destruidos en orden inverso. Así pues debe escribirse:

*// Order matters: they will Init/start/update in this order*

*modules.push\_back(window = new ModuleWindow());*

*modules.push\_back(renderer = new ModuleRender());*

*modules.push\_back(textures = new ModuleTextures());*

*modules.push\_back(input = new ModuleInput());*

Expliquemos un poco que es modules.push\_back(window = new ModuleWindow());. Modules es el nombre que se dio al listado de punteros en el fichero de cabecera y que ha sido definido como privado. En este caso el operador . (punto) posibilita el acceso a métodos específicos de la clase; aunque en este caso se está utilizando para formar un nombre completo. Los nombres completos especifican el espacio del nombre o la interfaz. Con push\_back estamos añadiendo a modules; así pues, con la sentencia modules.push\_back(window = new ModuleWindow()) lo que se está haciendo es añadir el módulo window a modules. Obsérvese que este módulo se crea en ese mismo momento.

Las siguientes dos líneas son comentarios donde deben resolverse dos pequeños ejercicios.

*// TODO 7: Create a new "scene" module that loads a texture and draws it on the screen*

*// Homework: Create a new module to handle music and sound effects*

Con esto se finaliza el bloque de Application. Y se inicia el bloque del desctructor. En este bloque se escribirá únicamente un comentario con un ejercicio a realizar tal y como se muestra a continuación:

*Application::~Application()*

*{*

*// TODO 6: Free module memory and check the result in Dr. Memory*

*}*

Atrás el destructor se declara el bloque de la función *Init()* donde se inicializará la función Application() mediante un bucle for. Debemos introducir las siguientes líneas de código:

*bool Application::Init()*

*{*

*bool ret = true;*

*for(list<Module\*>::iterator it = modules.begin(); it != modules.end() && ret; ++it)*

*ret = (\*it)->Init();*

*return ret;*

*}*

Comentemos un poco el bloque de la función. En el fichero de cabecera se ha declarado que la función *Init()* debía devolver un valor booleano, y aquí vuelve a recordarse ese hecho, por lo que antes de la función se ha incluido la palabra reservada bool. Dentro del bloque de la función se ha declarado y definido la variable *ret*, que como puede verse es una variable de tipo booleano que se ha inicializado como *true*.

Como puede verse, con el bucle for se inicializa odos y cada uno de los módulos. Una vez acabado el bucle for se devuelve la variable *ret*, que es booleana y es el valor que debe devolver la función *Init()*.

Tras este bloque se ha insertado un comentario con un ejercicio a realizar donde se habrán de implementarse dos nuevas funciones:

*// TODO 4: We need to have three updates, add them: PreUpdate Update PostUpdate*

Tras esta línea se definirá la función *Update()*. Esta función hará que los módulos se vayan actualizando cuando sea necesario. Para ello se utilizará un bucle *for* muy parecido al de la función *Init()*, y debe devolver un valor *update\_status*. Por tanto, introduciremos las siguientes líneas de código:

*update\_status Application::Update()*

*{*

*update\_status ret = UPDATE\_CONTINUE;*

*for(list<Module\*>::iterator it = modules.begin(); it != modules.end() && ret == UPDATE\_CONTINUE; ++it)*

*ret = (\*it)->Update();*

*return ret;*

*}*

De esta función cabe destacar la constante UPDATE\_CONTINUE. Esta constante se declarará en el fichero de cabecera Global que se describirá a continuación.

Para acabar con la clase *Application* sólo queda definir el método *CleanUp()* para realizar la limpieza de los módulos una vez estos ya no sean necesarios. Para ello se utilizará un bucle *for* tal y como se ha realizado con los métodos anteriores. Así pues, debe introducirse las siguientes líneas de código:

*bool Application::CleanUp()*

*{*

*bool ret = true;*

*for(list<Module\*>::reverse\_iterator it = modules.rbegin(); it != modules.rend() && ret; ++it)*

*ret = (\*it)->CleanUp();*

*return ret;*

*}*

Ahora sólo queda cerrar el bucle iniciado con el identificador *#ifndef* con *#endif* introuciendo la siguiente línea:

*#endif // #ifndef \_\_Application\_cpp\_\_*

GLOBALS.H

En este fichero de cabecera vamos a declarar las funciones y variables globales que se utilizarán a lo largo del motor gráfico.

Como en todos los ficheros de cabecera las dos primeras líneas del código del fichero de *Globals.h* serán los identificadores *#ifndef* y *#define*:

*#ifndef \_\_Application\_h\_\_*

*#define \_\_Application\_h\_\_*

Tras estas líneas, como siempre se escriben los *include* necesarios para el fichero de cabecera, en este caso ncesitaremos incluir las librerías de *stdio* y *windows*, ambas librerías de C++. Por ello se insertarán las siguientes líneas:

*#include <windows.h>*

*#include <stdio.h>*

Depués de declarar los *include* se definiremos las constantes necesarias mediante *#define*. La constante que va a definirse es *LOG*, esta constante escribirá una línea en un archio log durante la ejecución del programa. Esto nos permitirá conocer lo que va haciendo el programa y poder detectar fallos durante su ejecución. Para ellos incluiremos la siguiente línea:

*#define LOG(format, ...) log(\_\_FILE\_\_, \_\_LINE\_\_, format, \_\_VA\_ARGS\_\_);*

El siguiente paso es la declaración de la función *log()*. Esta función será del tipo void, es decir, no devolverá ningún tipo de valor y Se le podrán pasar una serie de valores. Para su declaración debe incluirse la siguiente línea:

*void log(const char file[], int line, const char\* format, ...);*

Esta función log() se definirá más adelante en otro fichero cabecera que llevará su nombre.

El siguiente paso es la definición del *enum update\_status*. Esta enumeración (*enum*) contendrá todos los posibles estados (*status*) que puede tener nuestro motor gráfico. Si recordamos del apartado anterior, hemos visto que se declaraban unas variables de tipos *update\_status*. Es en esta enumeración donde declaramos como deben ser ese tipo de variables. Para definir la enumeración introduciremos las siguiente líneas:

*enum update\_status*

*{*

*UPDATE\_CONTINUE = 1,*

*UPDATE\_STOP,*

*UPDATE\_ERROR*

*};*

Si se observa el contenido de la enumeración, puede observarse que el primer elemento se ha igualado a 1. Esto no significa que el primer elemento de la enumeración tenga valor 1, lo que se está haciendo es asignar el valor 1 al primer índice de la enumeración. Esto nos facilitará poder recorrer la enumeración de una forma más sencilla mediante sus índices.

Tras la enumeración declararemos una serie de constantes para la configuración de la ventana donde se visualizará la cámara de nuestro motor. Para ello introduciremos las siguientes líneas:

*// Configuration -----------*

*#define SCREEN\_WIDTH 640*

*#define SCREEN\_HEIGHT 480*

*#define FULLSCREEN false*

*#define VSYNC true*

*#define TITLE "Super Awesome Game"*

*#define SPRITE\_FILE "sprites.png"*

En estas líneas estamos declarando el ancho y el alto de la ventana, si se puede extender a toda la pantalla, su sincronización, el título que mostrará la ventana y su forma.

Ahora sólo hemos de finalizar el archivo de la forma correcta, eso es con la sentencia:

*#endif // #ifndef \_\_Globals\_h\_\_*

LOG.H

Tras el fichero *Globals.h*, declararemos la clase l*og()*. Para ello crearemos un fichero con el nombre *log.cpp* e introduciremos el siguiente código.

Las dos primeras líneas serán la declaración de los identificadores, tal y como se ha efectuado en los archivos de cabecera anteriores:

*#ifndef \_\_log\_cpp\_\_*

*#define \_\_log\_cpp\_\_*

Tras esto incuiremos el fichero de cabecera de Globals:

*#include "Globals.h"*

Y posteriormente definiremos la clase log de la siguiente manera:

*void log(const char file[], int line, const char\* format, ...)*

*{*

*static char tmp\_string[4096];*

*static char tmp\_string2[4096];*

*static va\_list ap;*

*// Construct the string from variable arguments*

*va\_start(ap, format);*

*vsprintf\_s(tmp\_string, 4096, format, ap);*

*va\_end(ap);*

*sprintf\_s(tmp\_string2, 4096, "\n%s(%d) : %s", file, line, tmp\_string);*

*OutputDebugString(tmp\_string2);*

*}*

La función de esta clase se recoger los datos que se han pasado e imprimirlos en un fichero de log para posteriormente poder revisarlo. Al inicio de la clase se declaran tres variables estáticas, dos del tipo *char* y una de tipo *va\_list*. Posteriormente se crean un valor de tipos *string* para imprimir en el archivo *log*. El valor del *string* creado depende de los argumentos que se pasen a la clase *log*.

Para cerrar el archivo debe añadirse la línea

*#endif // #ifndef \_\_log\_cpp\_\_*

MODULES.H

En los ficheros anteriores se ha creado el núcleo de lo que será nuestro motor gráfico. Ahora se crearán los módulos para nuestro futuro motor. El primer módulo a crear será *Modules*. En esta clase se declararán las clases de forma virtual.

Como siempre, primero introduciremos las dos líneas con los identificadores, en este caso serán:

*#ifndef \_\_Modules\_h\_\_*

*#define \_\_modules\_h\_\_*

Tras esta líneas declaramos la clase Application, que es por así decirlo la clase madre de Modules.

*class Application;*

Trs esta declaración pasamos a la definición de la clase *Módules*. Esta clase tendrá una componente pública donde se definen los métodos *Module()* y *Init()*. Este último método se declarará virtual.

*class Module*

*{*

*public:*

*Module()*

*{*

*}*

*virtual bool Init()*

*{*

*return true;*

*}*

Tras esta líneas insertaremos un comentario para un futuro ejercicio que consitirá en la declaración de dos nuevas clases:

*// TODO 3: Create a new method "Start()" that should be calling all modules just before the first frame*

A continuación declaramos dos nuevos métodos, ambos virtuales, para la actualización y el limpiado. Estos métodos serán *Update()* y *CleanUp()*. Para ello se introducirán las siguiente líneas:

*virtual update\_status Update()*

*{*

*return UPDATE\_CONTINUE;*

*}*

*virtual bool CleanUp()*

*{*

*return true;*

*}*

*};*

Ahora, como siempre que iniciamos un fichero con el identificador *#ifndef*, debemos cerrar el archivo con *#endif*.

*#endif // ifndef \_\_Modules\_h\_\_*

MODULEINPUT.H Y MPDULEINPUT.CPP

Como siempre, lo primero que debe hacerse es incluir los identificadores al inicio del fichero y luego los include necesarios. En este caso, se añadirán los siguiente include: Module y Globals. Por lo que el aspecto de las primeras líneas del fichero será el siguiente:

*#ifndef \_\_ModuleInput\_h\_\_*

*#define \_\_ModuleInput\_h\_\_*

*#include "Module.h"*

*#include "Globals.h"*

Tras estas líneas en la siguiente línea utilizaremos la palabra reservada typedef para asignar un nombre alternativo. Esto se hará de la siguiente manera:

*typedef unsigned \_\_int8 Uint8;*

Al incluir esta línea, se le está diciendo al compilador que cuando delante de una variable se coloque uint8, se está creando una variable del tipo int de 8 bitssin signo.

Tras esta línea ya iniciamos la declaración de la clase ModelInput. Esta clase tendrá por el momento una parte pública donde se declararán las funciones del constructor y el destructor, y la de inicialización, limpieza y una función llamada Update(). En la parte privada declararemos un puntero tipo unit8, recordad que hemos definido ste tipo mediante typedef anteriormente, que será un puntero del teclado.

*class ModuleInput : public Module*

*{*

*public:*

*ModuleInput();*

*~ModuleInput();*

*bool Init() override = 0;*

*bool CleanUp() override = 0;*

*update\_status Update() override = 1;*

*private:*

*const Uint8 \*keyboard = NULL;*

*};*

Debe prestarse atención a dos aspectos de la clase que acaba de definirse;

* Todas las variables han sido inicializadas. Esto se ha hecho para evitar problemas de alguna variable nula.
* Los métodos Init(), CleanUp() y Update() han sido sobreescritos. A estos métodos se les ha añadido la palabra reserva override para poder sobreescribir el método. Esto se ha hecho por que los me´todos padres han sido declarados en la clase Application. Al hacerse esto se favorece la interpretación del código a la vez que se ahorran posibles problemas de escritura, como por ejemplo, olvidar de poner una letra en mayúscula en el nombre del método.

Como el fichero se ha iniciado con el identificador #ifndef, éste debe cerrarse con #endif. Así pues la última línea del fichero será:

Una vez definido el fichero de cabecera pasaremos a definir el fichero ModuleInput.cpp. Como siempre, al inicio del fichero se definirán los include necesarios para la clase. En este caso serán: Application, Globals, ModuleInput y SDL.

*#include "ModuleInput.h"*

*#include "Application.h"*

*#include "Globals.h"*

*#include "SDL/include/SDL.h"*

SDL no es un módulo definido por nosotros, se trata de una librería que incluiremos posteriormente. Los detalles de esta librería serán descritos cuando realicemos su inclusión en nuestro motor.

Tras los includes definiremos el constructor y el destructor que por el momento estará vacíos:

*ModuleInput::ModuleInput()*

*{}*

*// Destructor*

*ModuleInput::~ModuleInput()*

*{}*

Tras ellos definiremos el método *Init()* donde se inicializará la clase. Este método constará de un bucle if y de dos LOGs. El primer LOG escribirá en el fichero al inicializarse correctamente el SDL, mientras que el segundo escribirá en el LOG en caso de que se entre en el bucle if. Esto ocurrirá cuando haya un error de inicialización del SDL.

*// Called before render is available*

*bool ModuleInput::Init()*

*{*

*LOG("Init SDL input event system");*

*bool ret = true;*

*SDL\_Init(0);*

*if(SDL\_InitSubSystem(SDL\_INIT\_EVENTS) < 0)*

*{*

*LOG("SDL\_EVENTS could not initialize! SDL\_Error: %s\n", SDL\_GetError());*

*ret = false;*

*}*

*return ret;*

*}*

Tal y como se ha especificado en el comentario, este método se llamará antes de que la clase render está lista. Tras este método se define el método Update() que será llamado cada vez que se actualice el pintado.

En la actualidad este método está vacío, pero en él deberá implementarse el cerrado de la ventana de la aplicación.

*// Called every draw update*

*update\_status ModuleInput::Update()*

*{*

*SDL\_PumpEvents();*

*keyboard = SDL\_GetKeyboardState(NULL);*

*// TODO 1: Make the application properly close when ESC is pressed (do not use exit())*

*// Homework: Make the application close up when pressing “X” button of the window*

*return UPDATE\_CONTINUE;*

*}*

Por último se definirá el método de limpieza de la clase. Este método limpiará la librería SL de la memoria.

*// Called before quitting*

*bool ModuleInput::CleanUp()*

*{*

*LOG("Quitting SDL input event subsystem");*

*SDL\_QuitSubSystem(SDL\_INIT\_EVENTS);*

*return true;*

*}*

MODULEWINDOW.H Y MODULEWINDOW.CPP

Tras el módulo input pasaremos a crear el ModuleWindow. Este módulo nos creará una ventana en el dispositivo.

Como siempre empezaremos con el fichero de cabecera y por lo tanto lo primero que introduciremos en el código es el identificador y los include necesarios.

*#ifndef \_\_ModuleWindow\_h\_\_*

*#define \_\_ModuleWindow\_h\_\_*

*#include "Module.h"*

*#include "SDL/include/SDL.h"*

Después de los include declararemos la clase ModuleWindow, la cual como siempre tendrá los métodos del constructor, el desctructor y un método para la limpieza: CleanUp(). Pero antes debemos declarar la clase Application de la que dependerá la clase MpduleWindow, por lo que introduciremos la siguiente línea:

*class Application;*

Tras estos declararemos la clase ModuleWindow. Esta clase tendrá el constructor y el constructor declarados como públicos y dos métodos más que será Init() y CleanUp(), que como hasta el momento, deberán devolver un elemento de tipo boolean.

A parte de estros étodos declararemos dos punteros como públicos. Estos dos punteros apuntarán a dos clases de SDL que nos permitirán controlar la pantalla.

*class ModuleWindow : public Module*

*{*

*public:*

*ModuleWindow();*

*// Destructor*

*virtual ~ModuleWindow();*

*// Called before quitting*

*bool Init() override = 0;*

*// Called before quitting*

*bool CleanUp()* override = 0*;*

*public:*

*//The window we'll be rendering to*

*SDL\_Window\* window = NULL;*

*//The surface contained by the window*

*SDL\_Surface\* screen\_surface = NULL;*

*};*

Tras la declaración de la clase y sus métodos finalizaremos el fichero con el cierre del identificador con el que hemos empezado el fichero.

*#endif // \_\_ModuleWindow\_h\_\_*

Una vez declarada la clase ModuleWindow empezaremos a definirla. Para estas clase necesitaremos las clases Globals y Application.

*#include "ModuleWindow.h"*

*#include "Application.h"*

*#include "Globals.h"*

Tras los includes definiremos el construcor de la clase y su desctrutor, que por ahora, dejaremos vacíos. Tras los cuales definiremos el método para Init(). Este método será llamao antes de que la clase render esté disponible. Lo primero que hará este método es escribir en el LOG que SDL se ha inicializado y creará una variable booleana que se convertirá en false si SDL no puede inicializarse. En caso de que se logre inicializar el método creará una nueva ventana.

*// Called before render is available*

*bool ModuleWindow::Init()*

*{*

*LOG("Init SDL window & surface");*

*bool ret = true;*

*if(SDL\_Init(SDL\_INIT\_VIDEO) < 0)*

*{*

*LOG("SDL\_VIDEO could not initialize! SDL\_Error: %s\n", SDL\_GetError());*

*ret = false;*

*}*

*else*

*{*

*//Create window*

*int width = SCREEN\_WIDTH;*

*int height = SCREEN\_HEIGHT;*

*Uint32 flags = SDL\_WINDOW\_SHOWN;*

Tras la creación de la ventana se llama a un bucle if donde se ha insertado un comentario para que posteriormente se rellene con código que nos permitirá controlar la ventana. Ahora mismo, se creásemos la ventana no se podría cerrar la misma de ninguna forma.

*if(FULLSCREEN == true)*

*{*

*flags |= SDL\_WINDOW\_FULLSCREEN;*

*}*

*// TODO 2: Create options for RESIZABLE, SDL\_WINDOW\_BORDERLESS, SDL\_WINDOW\_RESIZABLE,*

*// SDL\_WINDOW\_FULLSCREEN\_DESKTOP (same way as with FULLSCREEN)*

*window = SDL\_CreateWindow(TITLE, SDL\_WINDOWPOS\_UNDEFINED, SDL\_WINDOWPOS\_UNDEFINED, width, height, flags);*

Tras este bucle se inicia otro bucle if en el que se entrará en el caso de que el puntero del módulo sea null y en caso contrario inicializará la ventana. Recordemos que el método debe devolver una variable de tipo booleana; por lo que anter de cerrarlo se cevuelve la misma.

*if(window == NULL)*

*{*

*LOG("Window could not be created! SDL\_Error: %s\n", SDL\_GetError());*

*ret = false;*

*}*

*else*

*{*

*//Get window surface*

*screen\_surface = SDL\_GetWindowSurface(window);*

*}*

*}*

*return ret;*

*}*

Posteriormente a este método se declara el método de limpieza donde se escribirá en el fichero LOG y se destruirá la ventana en caso de que este haya sido creado. A parte de eso se sale del método de SDL.

*// Called before quitting*

*bool ModuleWindow::CleanUp()*

*{*

*LOG("Destroying SDL window and quitting all SDL systems");*

*//Destroy window*

*if(window != NULL)*

*{*

*SDL\_DestroyWindow(window);*

*}*

*//Quit SDL subsystems*

*SDL\_Quit();*

*return true;*

*}*

MODULERENDER.H Y MODULERENDER.CPP

Tras el módulo window pasaremos a crear el ModuleRender. Este módulo será el que utilizaremos para pintar en pantalla el resto de módulos.

Como siempre empezaremos definiendo el fichero de cabecera del módulo. Las primeras dos líneas serán como siempre los identificadores.

*#ifndef \_\_ModuleRender\_h\_\_*

*#define \_\_moduleRender\_h\_\_*

Como siempre, después de los identificadores definiremos los includes necesarios para utilizar la clase Render. En este caso serán necesarios: Module y Globals.

*#include "Module.h"*

*#include "Globals.h"*

Tras los defines declararemos la palabra clave struct para definir tres estructuras de SDL:

*struct SDL\_Texture;*

*struct SDL\_Renderer;*

*struct SDL\_Rect;*

Tras esto empezaremos la definición de la clase Render. En este caso la clase sólo contendrá los métodos públicos del constructor, destructor y los tres métodos que nos encontraremos en la mayoría de las clases del motor: Init(), CleanUp() y Update(), a parte de estos méodos, también se deberá declara el método Blit() y variables publicas render, que se trata de un puntero.

*class ModuleRender : public Module*

*{*

*public:*

*ModuleRender();*

*~ModuleRender();*

*bool Init();*

*update\_status Update();*

*bool CleanUp();*

*bool Blit(SDL\_Texture\* texture, int x, int y, SDL\_Rect\* section);*

*public:*

*SDL\_Renderer\* renderer = NULL;*

*};*

Recordemos que al inicio del fichero se ha utiliza el identificador; por lo que debemos cerrarlo al final del mismo.

*#endif // \_\_ModuleRender\_h\_\_*

Tras la definición del fichero de cabecera pasaremos a definir el fichero de la clase Render. Recordemos que esta clase será utilizada para pintar en pantalla cada una de las pares de nuestro futuro motor.

Como siempre, declararemos los include necesarios, en este caso serán: Application, Globals, Window y SDL, y tras ellos el constructor y el destructor de la clase que por el momento se dejarán vacíos.

*#include "ModuleRender.h"*

*#include "Application.h"*

*#include "Globals.h"*

*#include "ModuleWindow.h"*

*#include "SDL/include/SDL.h"*

*ModuleRender::ModuleRender()*

*{*

*}*

*// Destructor*

*ModuleRender::~ModuleRender()*

*{*

*}*

El siguiente método al que se llamará es el método Init() que será llamado antes de que render esté disponible. Este método, como todos los anteriores escribirá en el fichero LOG informando de la creación del contexto y llama a la creación de la ventana donde posteriormente se pintará.

*// Called before render is available*

*bool ModuleRender::Init()*

*{*

*LOG("Creating Renderer context");*

*bool ret = true;*

*Uint32 flags = 0;*

*if(VSYNC == true)*

*{*

*flags |= SDL\_RENDERER\_PRESENTVSYNC;*

*}*

*renderer = SDL\_CreateRenderer(App->window->window, -1, flags);*

El bucle if ira si el puntero de render no es null, en caso de serlo escribe en el fichero LOG este hecho. Como debe ser, antes de acabar el método se devuelve una variable de tipo booleano.

*if(renderer == NULL)*

*{*

*LOG("Renderer could not be created! SDL\_Error: %s\n", SDL\_GetError());*

*ret = false;*

*}*

*return ret;*

*}*

Tras este método se llama al método Update() que pintará en la ventana que se ha llamado en el método anterior. Lo primero que realiza el método es limpiar la ventana

*// Called every draw update*

*update\_status ModuleRender::Update()*

*{*

*SDL\_RenderClear(renderer);*

Tras esta línea se escriba un comentario para un posterior ejercicio donde se deberá mostrar el preupdate y el postupdate.

*// TODO 5: Now that we have PreUpdate/PostUpdate/Update move things around so we can render :)*

Posteriormente se utiliza una función que actualiza el pintado que se muestra en pantalla y se devuelve la variable de retorno del étodo Update().

*SDL\_RenderPresent(renderer);*

*return UPDATE\_CONTINUE;*

*}*

Posteriormente a este método se implementará el método de limpieza que destruirá el render

*// Called before quitting*

*bool ModuleRender::CleanUp()*

*{*

*LOG("Destroying renderer");*

*//Destroy window*

*if(renderer != NULL)*

*{*

*SDL\_DestroyRenderer(renderer);*

*}*

*return true;*

*}*

Depués del método de limpieza se implementará el método Blit() al que será necesario pasarle la textura, dos variables integer que serán las direcciones de los ejes de coordenadas, y una variable de SDL. Este método pintará la tetura en pantalla.

*// Blit to screen*

*bool ModuleRender::Blit(SDL\_Texture\* texture, int x, int y, SDL\_Rect\* section)*

*{*

*bool ret = true;*

*SDL\_Rect rect;*

*rect.x = x;*

*rect.y = y;*

*if(section != NULL)*

*{*

*rect.w = section->w;*

*rect.h = section->h;*

*}*

*else*

*{*

*SDL\_QueryTexture(texture, NULL, NULL, &rect.w, &rect.h);*

*}*

*if(SDL\_RenderCopy(renderer, texture, section, &rect) != 0)*

*{*

*LOG("Cannot blit to screen. SDL\_RenderCopy error: %s", SDL\_GetError());*

*ret = false;*

*}*

*return ret;*

*}*

MODULETEXTURES.H Y MODULETEXTURES.CPP

Este modulo será el último módulo que crearemos para nuestro esqueleto del futuro motor gráfico que estamos desarrollado. Esta clase será la encargada de cargar las texturas que serán utilizadas.

Como siempre lo primero que crearemos serán el fichero de cabecera que se inicializará de la misma forma que hemos hecho hasta ahora. En esta ocasión las clases que debemos cargar serán; Module y Globals. A parte de estas clases se incluirá una librería de C++ que cargará la clase list (listas).

*#ifndef \_\_ModuleTextures\_h\_\_*

*#define \_\_moduleTextures\_h\_\_*

*#include<list>*

*#include "Module.h"*

*#include "Globals.h"*

Tras los includes se declarará la variable SDL\_Texture que será de tipo struct., y por tanto, será una estructura o registro.

*struct SDL\_Texture;*

Posteriormente, se definirá la clase MOduleTexture que se compondrá de un constructor, un destructor y los siguientes métodos: Init(), CleanUp() y un puntero a SDL\_Texture

*bool Init();*

*bool CleanUp();*

*SDL\_Texture\* const Load(const char\* path);*

Tras estos métodos se declará como privada una lista que se llenará con punteros.

*private:*

*std::list<SDL\_Texture\*> textures;*

private:

std::list<SDL\_Texture\*> textures;

Como siempre se deberá cerrar el fichero de cabecera con:

*#endif // \_\_ModuleTextures\_h\_\_*

Tras la creación del fichero de cabecera empezaremos a definir la clase textures. Para ello lo primero que ecribiremos serán los includes. En este case llamaremos a las siguientes clases: Application, Globals, Render, Textures, SDL y SDL\_image.

*#include "ModuleTextures.h"*

*#include "Application.h"*

*#include "Globals.h"*

*#include "ModuleRender.h"*

*#include "ModuleTextures.h"*

*#include "SDL/include/SDL.h"*

*#include "SDL\_image/include/SDL\_image.h"*

*#pragma comment( lib, "SDL\_image/libx86/SDL2\_image.lib" )*

La última línea es un comentario (#pragma comment) sobre la localización de la librería SDL2\_image. Con #pragma comment indicamos a Visual C++ wue deje un comentario en el fichero de generación del objeto. En este caso estamos indicando al Linker que añada el nombre de esta librería a lista de dependencias.

Posteriormente introducimos el espacio de nombre mediante:

*using namespace std;*

Posteriormente declararemos el constructor de la clase, que estará vacío y el destructor, el cual esta vez no estará vacío, sino que llamará al método IMG\_Quit(). Esta función limpia lo que se ha cargado dinámicamente.

*ModuleTextures::ModuleTextures()*

*{*

*}*

*// Destructor*

*ModuleTextures::~ModuleTextures()*

*{*

*IMG\_Quit();*

*}*

Posteriormente se declara el método Init() que cargará una función que nos permitirá utilizar imágenes en formato png para las texturas. Como siempre, este método realizará escrituras en el fichero de LOG dependiendo de si ha habido errores o no.

*// Called before render is available*

*bool ModuleTextures::Init()*

*{*

*LOG("Init Image library");*

*bool ret = true;*

*// load support for the PNG image format*

*int flags = IMG\_INIT\_PNG;*

*int init = IMG\_Init(flags);*

*if((init & flags) != flags)*

*{*

*LOG("Could not initialize Image lib. IMG\_Init: %s", IMG\_GetError());*

*ret = false;*

*}*

*return ret;*

*}*

Tras el método Init() se implementará el método de limpieza CleanUp() donde se limpiarán odas lastexturas que se hayan cargado.

*// Called before quitting*

*bool ModuleTextures::CleanUp()*

*{*

*LOG("Freeing textures and Image library");*

*for(list<SDL\_Texture\*>::iterator it = textures.begin(); it != textures.end(); ++it)*

*SDL\_DestroyTexture(\*it);*

*textures.clear();*

*return true;*

*}*

El ultimo método de la clase será el método Load() que se encargará de cargar la textura. Para ello se le ha de pasar al método la dirección de donde se encuentra almacenada la textura que desea cargarse.

*// Load new texture from file path*

*SDL\_Texture\* const ModuleTextures::Load(const char\* path)*

*{*

*SDL\_Texture\* texture = NULL;*

*SDL\_Surface\* surface = IMG\_Load(path);*

*if(surface == NULL)*

*{*

*LOG("Could not load surface with path: %s. IMG\_Load: %s", path, IMG\_GetError());*

*}*

*else*

*{*

*texture = SDL\_CreateTextureFromSurface(App->renderer->renderer, surface);*

*if(texture == NULL)*

*{*

*LOG("Unable to create texture from surface! SDL Error: %s\n", SDL\_GetError());*

*}*

*else*

*{*

*textures.push\_back(texture);*

*}*

*SDL\_FreeSurface(surface);*

*}*

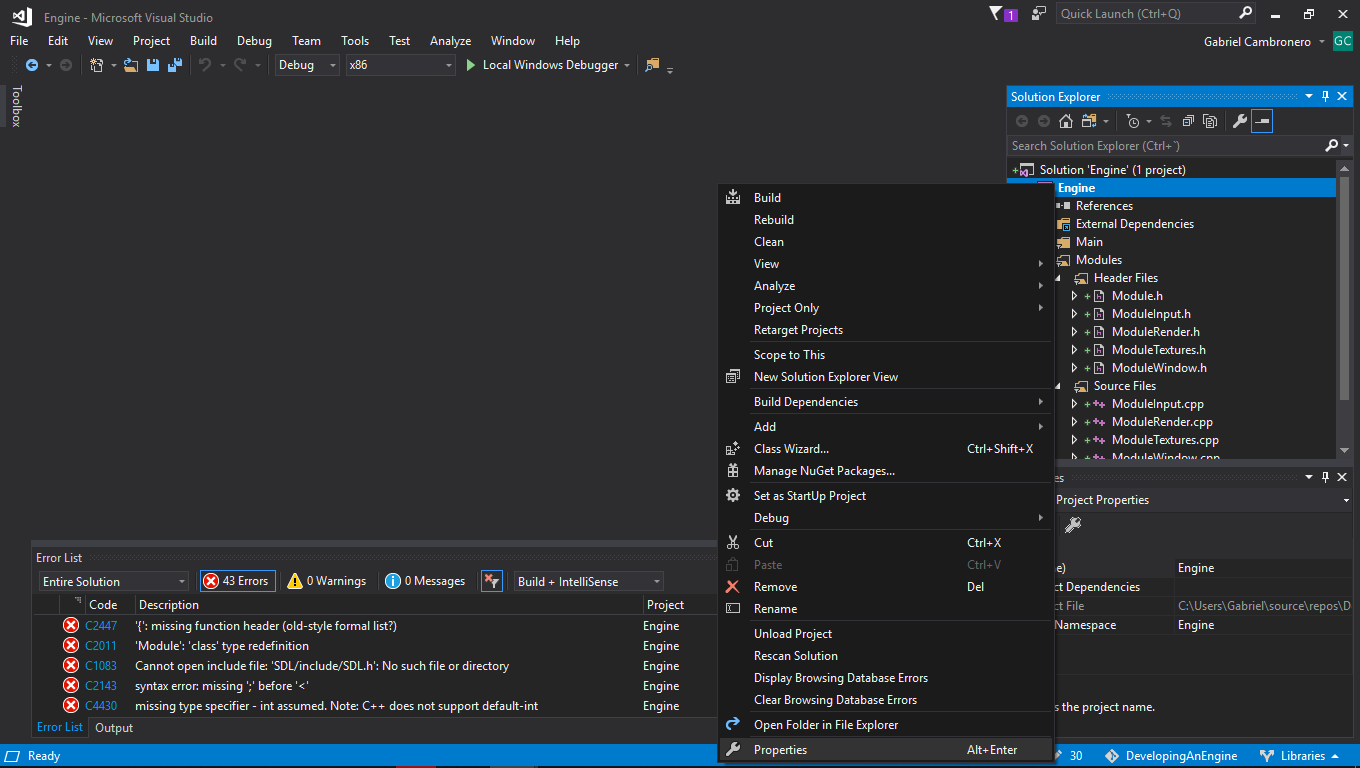
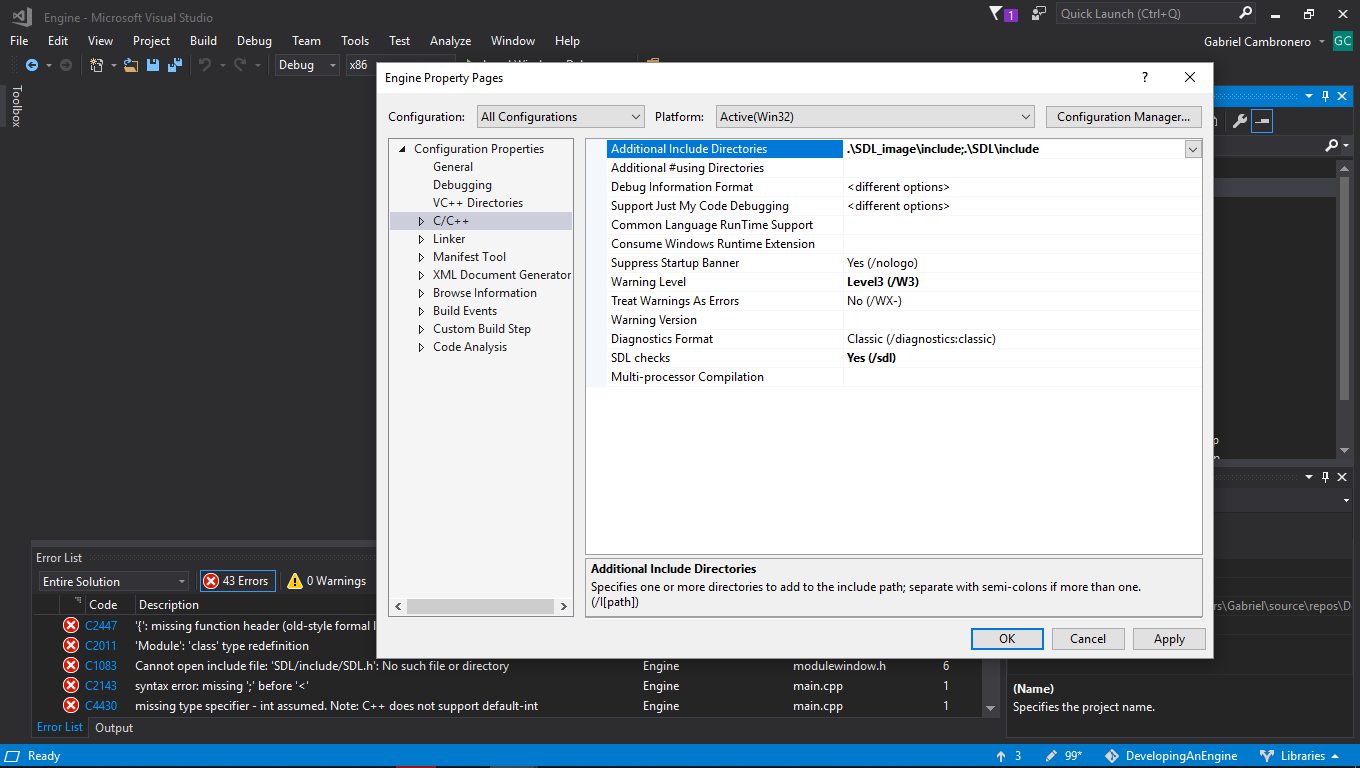
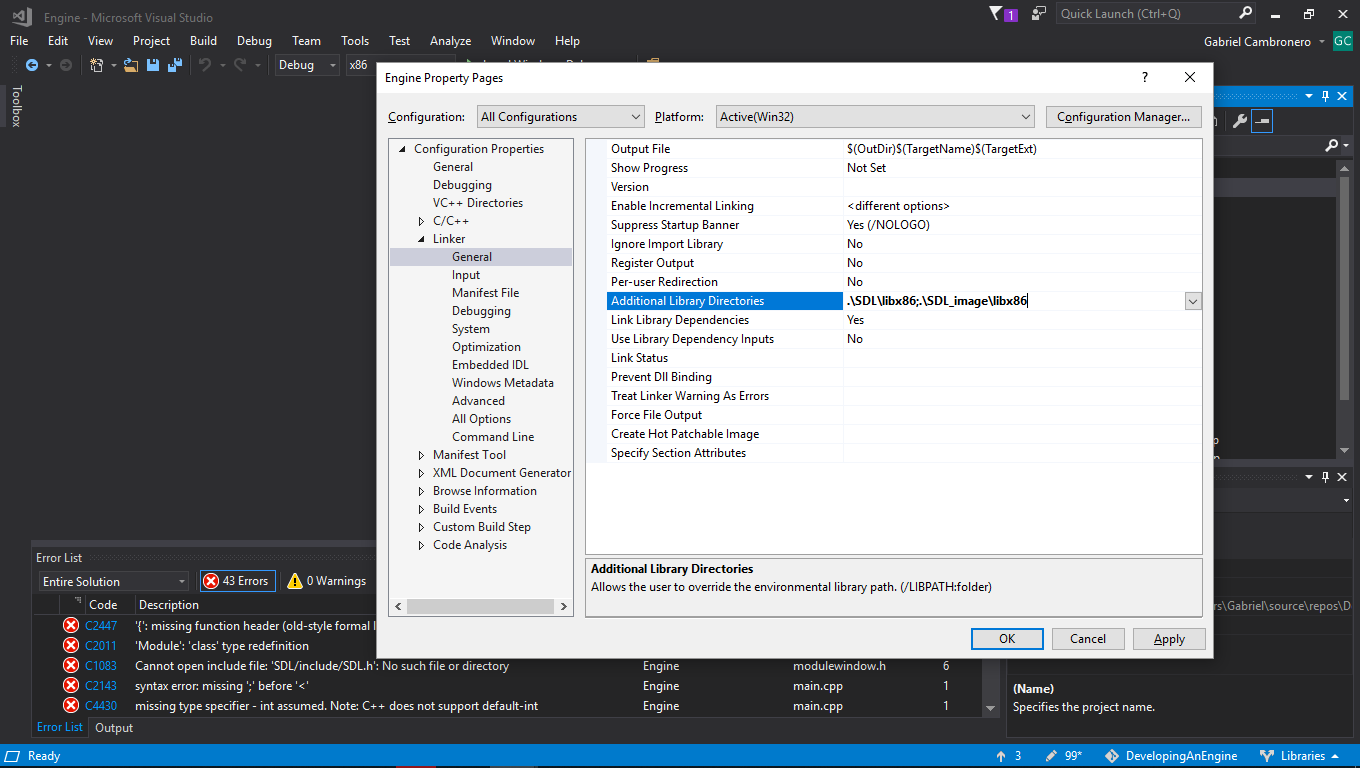
*return texture;*

*}*

CARGA DE LA LIBRERÍA DE SDL Y SDL\_IMAGE

Ahora ya tenemos cargados todas clases y los métodos del esqueleto de nuestro futuro motor. Pero si pulsamos F5 para realizar la compilación del mismo Visual Studio nos devolverá un error de compilación. Esto se debe a que no hemos cargado las librerías de SDL que estamos utilizando.

Para cargar las librerías debe hacerse lo siguiente:

1. Con el ratón sobre la carpeta del proyecto pulsar el botón derecho y seleccionar Porperties (al final de la lista). 
2. Ir a la opción C/C++ y pulsar sobre Additional Include Directories. Aquí debe introducirse la dirección de las carpetas donde están los archivos delas librerías de SDL. Es mejor que se introduzac una ruta relativa. 
3. Ir a la opción Linker->General y pulsar sobre Addoitional Library Directories. Aquí debe escribirse la ruta a las librerías de SDL (ficheros .lib). 
4. Aún nos queda una última cosa a hacer en Visual Studio. Ir a la opción Linker->Input y en el campo Aditional Dependencies escribir el nombre de los archivos .lib que componen las librerías.
5. Ahora sólo queda tenemos que copiar los archivos .dll de las librerías en la carpeta a la que apunta nuestro proyecto. En caso de no saber a que carpeta apunta el proyecto, se pueden copiar en la misma carpeta donde se encuentras los archivos .h y .cpp del proyecto.

EJERCICIOS

Ahora tenemos el esqueleto de nuestro motor, pero s lo ejecutamos veremos que se muestra una ventana de color negro pero que no podemos cerrarla, ni maximizarla ni hacer nada con ella.

Durante la creaciñon de la base de nuestro futuro motor hemos ido incluyendo líneas de comentario con la palabra TODO. Para realizar una búsqueda en Visual Sudio debe utilizarse la siguiente combinación de teclas Control+ Shift + F.

Utilizando esta combinación de teclas ahora deben buscarse los comentarios TODO y realizar los ejercicios propuestos.

TODO 1: HACER QUE LA APLICACIÓN SE CIERRE.

Para hacer que la aplicación se cierre al pulsar la tecla de escape (Esc), deben utilizarse métodos de SDL.

El primer paso será la creación de una constante mediante su definición después de los includes. Para ello incluiremos la siguiente línea:

*#define MAX\_KEYS 300*

Para poder utilizar el teclado se introducirá la siguiente línea de comandos dentro de *ModuleInput()*:

*keyboard = new KeyState[MAX\_KEYS];*

Ahora pasaremos al método *Update()* donde se cambiarán las líneas:

*SDL\_PumpEvents();*

*keyboard = SDL\_GetKeyboardState(NULL);*

Por las siguientes:

*static SDL\_Event event;*

*const Uint8\* keys = SDL\_GetKeyboardState(NULL);*

En estas líneas hemos declarado una variable *event* de tipo *SDL\_Event* y una variable denominada *keys*. Tras estas líneas se introducirá un bucle *for*. Este bucle recorrerá las pulsaciones introducidas en el teclado.

*for (int i = 0; i < MAX\_KEYS; ++i)*

*{*

*if (keys[i] == 1)*

*{*

*if (keyboard[i] == KEY\_IDLE)*

*keyboard[i] = KEY\_DOWN;*

*else*

*keyboard[i] = KEY\_REPEAT;*

*}*

*else*

*{*

*if (keyboard[i] == KEY\_REPEAT || keyboard[i] == KEY\_DOWN)*

*keyboard[i] = KEY\_UP;*

*else*

*keyboard[i] = KEY\_IDLE;*

*}*

*}*

Tras este bucle *for*, introduciremos un bucle *if* que cerrará la ventana en caso de que la tecla escape sea pulsada:

*if (GetKey(SDL\_SCANCODE\_ESCAPE) == KEY\_DOWN)*

*return UPDATE\_STOP;*

Este bucle *if* depende del método *GetKey()*, el cual no ha sido declarado por el momento. Así que nuestro próximo paso será la declaración y definición de dicho método.

Así que pasaremos al fichero *ModuleInput.h* para la declaración y definición (ya que se trata de un método de una línea.

Pero antes realizaremos un *include* para poder utilizar los métodos de SDL:

*#include "SDL/include/SDL\_scancode.h"*

En el fichero *ModuleInput.h* introduciremos como *public* las siguientes líneas:

*// Check key states (includes mouse and joy buttons)*

*KeyState GetKey(int id) const*

*{*

*return keyboard[id];*

*}*

Como Podemos ver el método *GetKey()* es del tipo *KeyState*, que es un *enum*. Para definirlo primeros definiremos un puntero dentro de la parte *private*. Allí incluiremos la siguiente línea:

*KeyState\* keyboard;*

Una vez hemos realizado esto, sólo nos queda declara y definir el *enum KeyState* justo después de los *define* que ya tenemos definidos de la siguiente manera:

*enum KeyState*

*{*

*KEY\_IDLE = 0,*

*KEY\_DOWN,*

*KEY\_REPEAT,*

*KEY\_UP*

*};*

Ahora, si ejecutamos la aplicación veremos como la ventana de la aplicación se cierra al pulsar sobre la tecla Esc. Si se pulsa sobre la cruz presente en el borde superior derecho de la ventana, se verá como ésta no reacciona a la pulsación. Esto se debe a que no hay ningún evento escuchando y tampoco se le ha asociado ninguna acción.