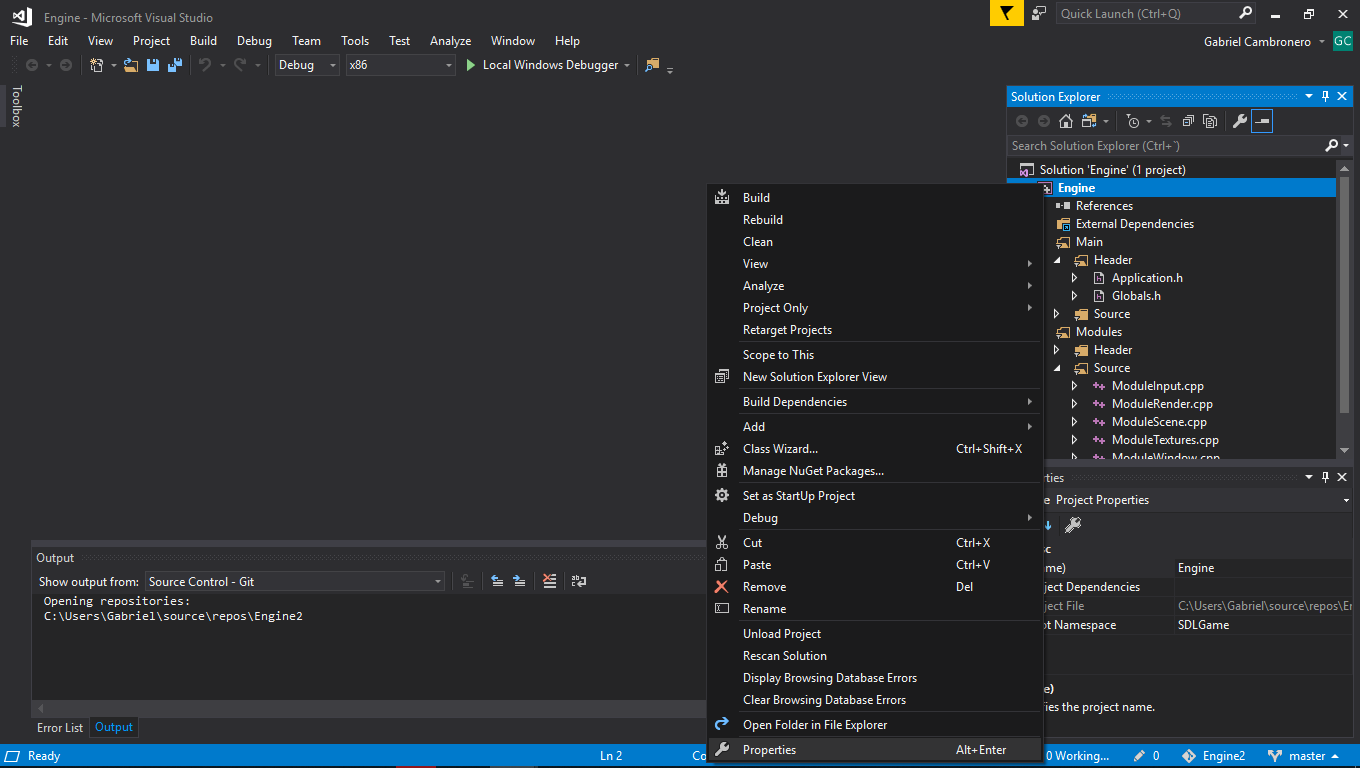
CONSTRUYENDO UN TRIÁNGULO

En este segundo capítulo dibujaremos un triángulo en la ventana que hemos creado en el anterior capítulo.

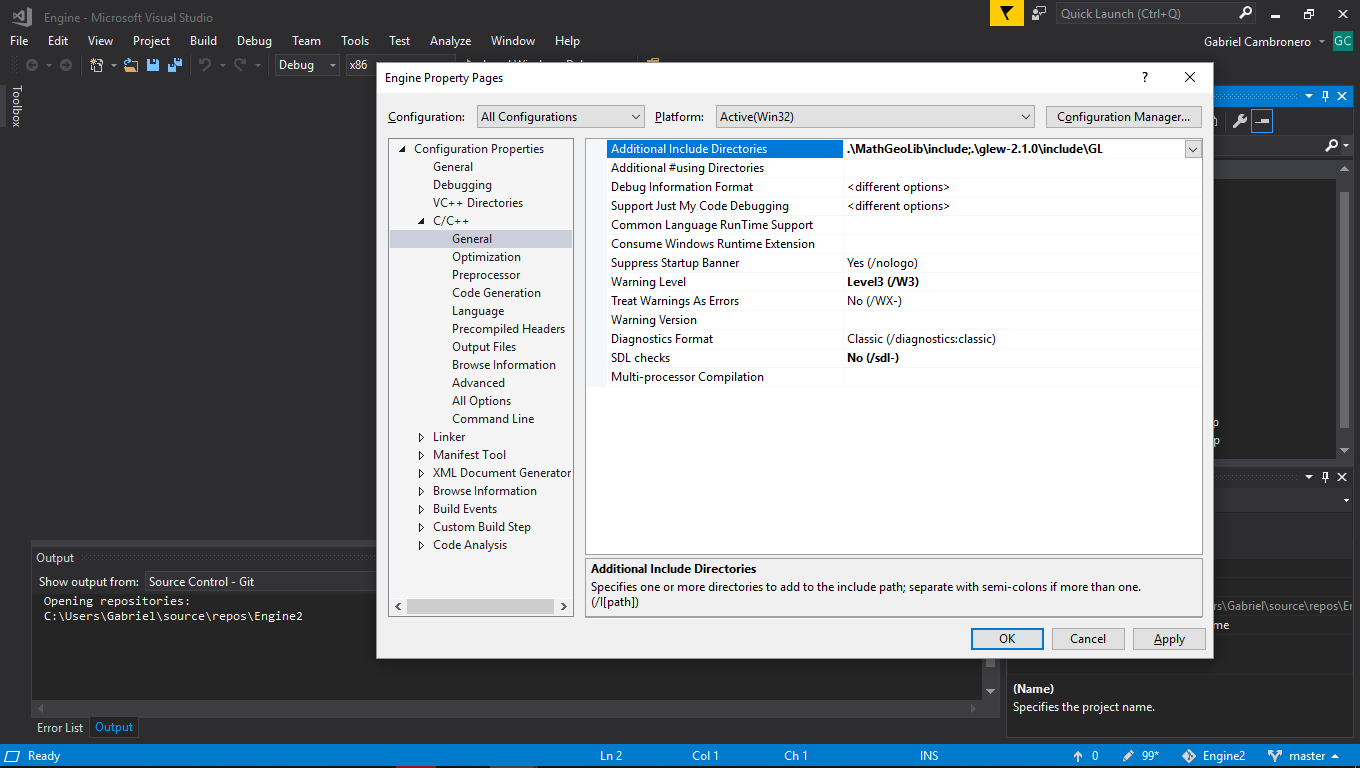
Para este sealizar este capítulo será necesario instalar dos librerías externas: *Glew-2.1.0* y *MathGeoLib*. En el siguiente apartado se explicará como instalar Glew-2.1.0.

AÑADIENDO LIBRERÍAS EXTERNAS

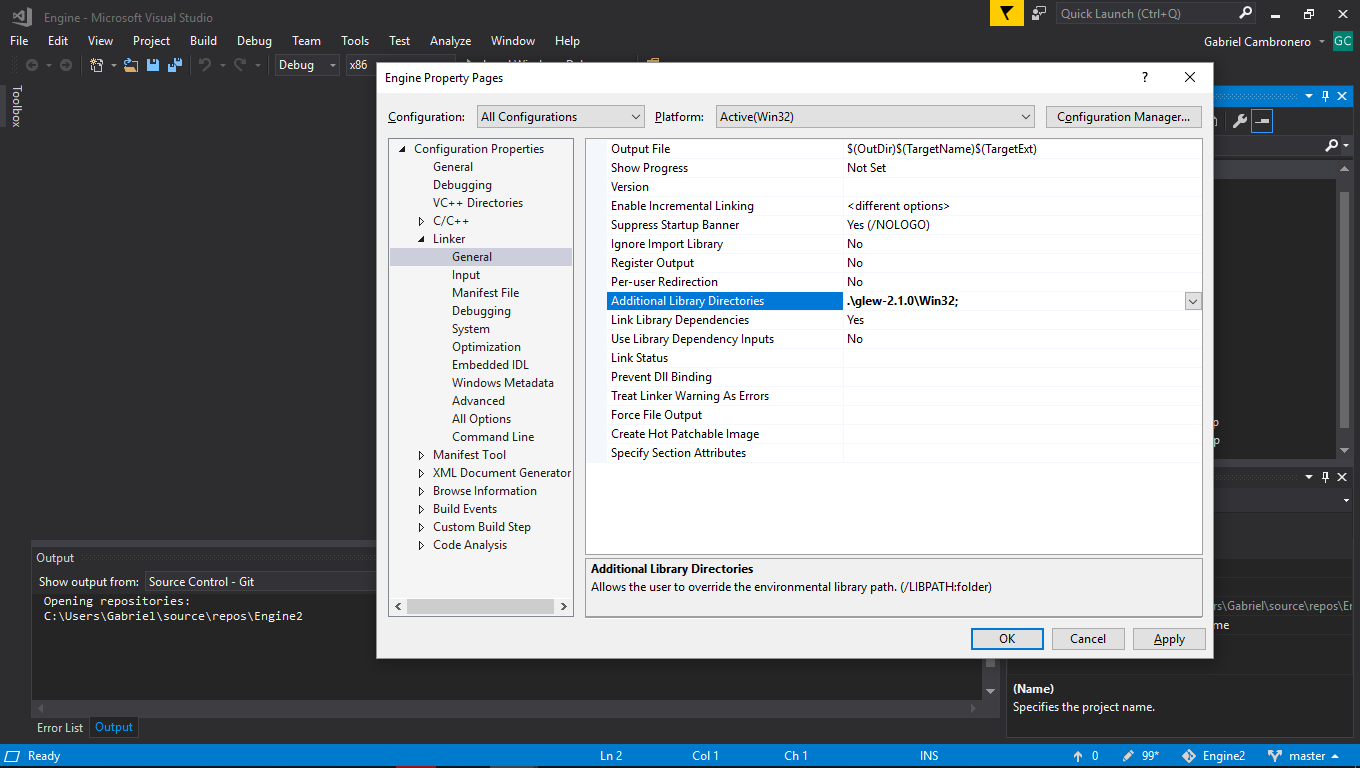
Ante todo debemos descargar la librería de Glew que puede encontrarse dentro de la carpeta Librerías del repositorio o en su página web. Una vez descargada la librería se debe incluir la carpeta Glew-2.1.0 con las subcarpetas include y lib en la carpeta raíz del proyecto. Una vez hecho esto vamos a Visual Studio y abrimos nuestro Engine.

Situamos el ratón sobre el nombre del proyecto: *Engine* y pulsamos el botón derecho. En el menú contextual que aparece seleccionamos la opción *Properties* situada al final del menú. 

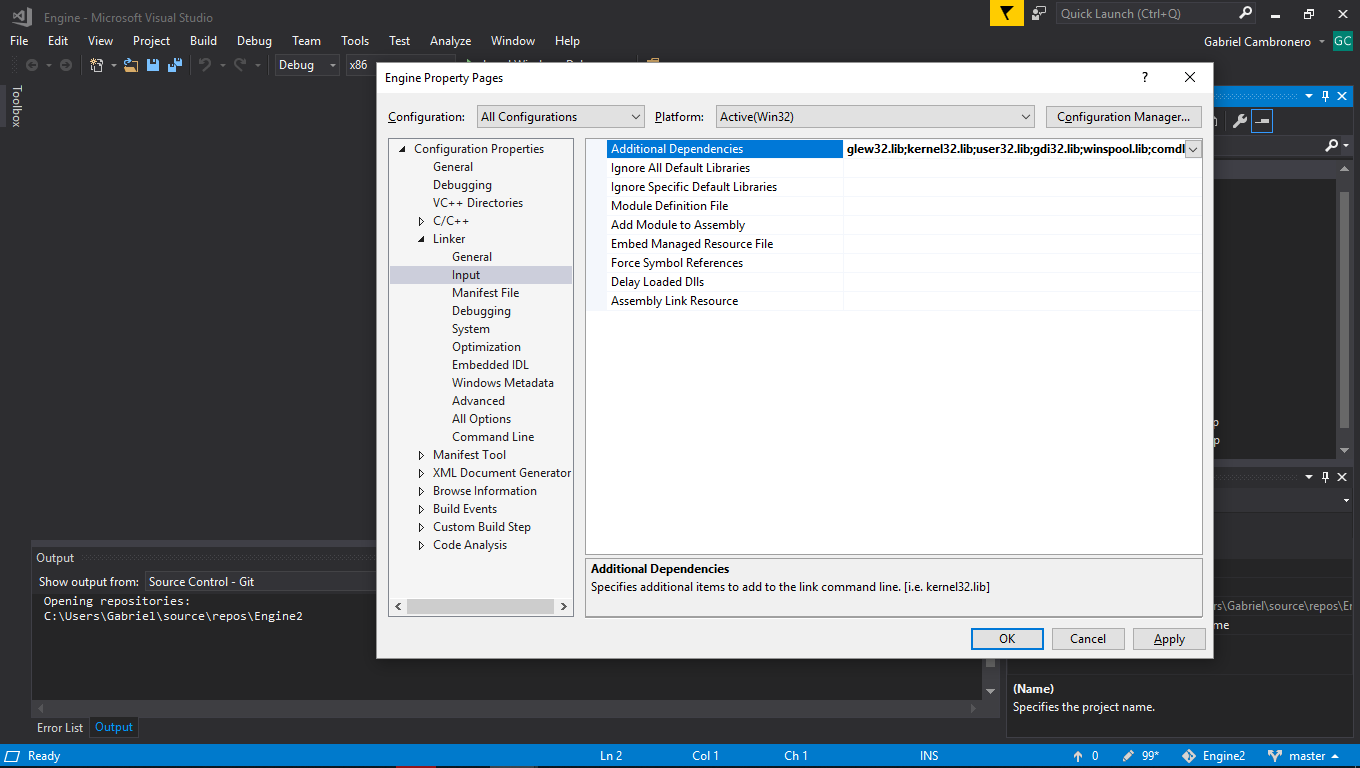
Tras esto nos aparecerá una nueva ventana donde seleccionaremos *C/C++* -> *General* del menú lateral derecho. Una vez dentro de esta opción nos posicionaremos sobre la casilla Aditional dependencias Directories donde deberemos incluir las direcciones de las carpetas de las dos bibliotecas que queremos incluir.



Tras esto iremos al menú *Linker* -> *General* y nos posicionaremos sobre la casilla *Additional Librery Directories* donde debemos introducir la dirección de las librerías a incluir. En este caso sólo debemos incluir la biblioteca de *Glew*, ya que *MathGeoLib* no tiene un fichero *.lib*.



Tras esto sólo nos queda ir a *Linker* -> *Input* y posicionarnos en la casilla *Additional Dependencies* donde introduciremos el nombre del fichero *.lib* que queremos incluir.



Con esto ya habremos incluido las librerías externas que necesitaremos para nuestro motor; pero aún nos queda un último paso para finalizar. Este último caso consiste en añadir los archivos de cabecera al proyecto.

También tendremos que incluir los archivos *.dll* de las librerías a la carpeta donde apunta nuestro motor. En caso de no saber a que carpeta apunta el proyecto, deben incluirse estos archivos *.dll* a la carpeta donde estás los ficheros de nuestro motor.

Una vez realizado esto ya podremos empezar a dibujar el triángulo en nuestra ventana.

Utilizaremos la clase *ModuleScene* que creamos en el último punto del capítulo anterior para dibujar nuestro triángulo.

Como siempre, empezaremos por definir los includes. Para pintar el triángulo necesitaremos incluir las clases Application, ModuleScene, ModuleWindow, Glew y SDL. Por lo que incluiremos las siguientes líneas:

*#include "Application.h"*

*#include "ModuleScene.h"*

*#include "ModuleWindow.h"*

*#include "GL/glew.h"*

*#include "SDL.h"*

Tras esto, dejaremos el constructor y el destructor de la calse vacíos, tal y como estaban, y eliminaremos el método *Start()* que se había definido y los sustituiremos por un método *Init()* que se definirá de la siguiente manera:

*bool ModuleScene::Init()*

*{*

*float vertex\_buffer\_data[] = {*

*-1.0f, -1.0f, 0.0f,*

*1.0f, -1.0f, 0.0f,*

*0.0f, 1.0f, 0.0f,*

*};*

*glGenBuffers(1, &vbo);*

*glBindBuffer(GL\_ARRAY\_BUFFER, vbo);*

*glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertex\_buffer\_data), vertex\_buffer\_data, GL\_STATIC\_DRAW);*

*glBindBuffer(GL\_ARRAY\_BUFFER, 0);*

*return vbo; // Vetor Buffer Object*

*}*

En este método estamos definiendo las coordenadas que tendrá los vértices de nuestro triángulo dentro del *array* de *floats vertex\_buffer\_data*. Expliquemos un poco más el código línea a línea.

*glGenBuffers(1, &vbo);*

En esta línea generamos un *buffer object* que se guarda en el Aray *vbo*.

*glBindBuffer(GL\_ARRAY\_BUFFER, vbo);*

En esta línea estamos enlazando nuestro *array vbo* con un *target* de *OpenGL*, en este caso un *GL\_ARRAY\_BUFFER*, es decir, un *array* de *buffers*.

*glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertex\_buffer\_data), vertex\_buffer\_data, GL\_STATIC\_DRAW);*

En esta línea se crea un nuevo almacenaje de datos para un buffer object. En este caso, el *buffer object* utiliza la unión al *target*.

Por ´último, cerramos el enlace mediante un array de ceros:

*glBindBuffer(GL\_ARRAY\_BUFFER, 0);*

Tras la definición del método *Init()* definiremos el método *Update()*. En este método es donde pintaremos realmente el triángulo que hemos definido en el anterior método y se definirá de la siguiente forma:

*update\_status ModuleScene::Update()*

*{*

*glEnableVertexAttribArray(0);*

*glBindBuffer(GL\_ARRAY\_BUFFER, vbo);*

*glVertexAttribPointer(*

*0, // attribute 0*

*3, // number of componentes (3 floats)*

*GL\_FLOAT, // data type*

*GL\_FALSE, // should be normalized?*

*0, // stride*

*(void\*)0 // array buffer offset*

*);*

*glDrawArrays(GL\_TRIANGLES, 0, 3); // Starting from vertex 0; 3 vertices total -> 1 // triangle*

*glDisableVertexAttribArray(0);*

*glBindBuffer(GL\_ARRAY\_BUFFER, 0);*

*return UPDATE\_CONTINUE;*

*}*

Expliquemos un poco que estamos haciendo dentro de este método.

En la primera línea: *glEnableVertexAttribArray(0);* estamos diciendo a OpenGL que habilite el array de atributos de los vértices genéricos. En nuestro caso eto lo estamos haciendo mediante la unión del *vertex array object*.

Lo que hace la segunda línea ya se explicó en el bloque anterior, así que se pasará a describir que realiza la tercera línea del bloque: *glVertexAttribPointer()*. En esta línea se está especificando la localización y el formato del array de los atributos de los vértices en el índice que se usará para renderizar, el tamaño especifica el número de componentes por atributo, el tipo indica el tipo de datos que contiene el array en su interior y el paso indica el paso de bytes de un atributo a otro permitiendo que los atributos y los vértices sean empaquetados en un único array o en varios.

En la siguiente línea del boque: *glDrawArrays(GL\_TRIANGLES, 0, 3);* es cuando realmente pintamos el triángulo en pantalla. En este comando estamos diciendo que pinte el triángulo empezando por el vértice 0 y que contrndrá 3 vérices.

Por último, antes de realizar la senencia return, indicamos a OpenGL que deshabilite el atributo de los vértices que habíamos habilitado en la primera línea del bloque y lo enlazamos con el array que contiene la información.

Para concluir la clase, como casi siempre, hemos de crear el método de limpieza *CleanUp()*. Este método debe limpiar la memoria los buffers ocupados por el elemento *vbo* que hemos creado anteriormente y por tanto será definido de la siguiente manera:

*bool ModuleScene::CleanUp()*

*{*

*if (vbo != 0)*

*{*

*glDeleteBuffers(1, &vbo);*

*}*

*return true;*

*}*

Ahora solo debemos ir al fichero de cabecera de la clase donde declararemos los nuevos métodos para completar la clase. En el fichero de cabecera eliminaremos la declaración del método *Start()* de *public* y añadiremos los siguientes métodos:

*bool Init();*

*bool CleanUp();*

También añadiremos como *private* la variable *vbo*:

*private:*

*unsigned vbo;*

Antes de ejecutar el código debemos hacer algunas modificaciones.

MODIFICACIONES APPLICATION.

Abrimos la case *Application()* donde modificaremos el destructor para utilizar un bucle *for* que vaya borrando las clases que se han creado. El bucle *for* será de la siguiente forma:

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

*{*

*delete \*it;*

*}*

También entraremos dentro del método *Init()* donde borraremos o eliminaremos las siguientes líneas:

*if (ret)*

*{*

*ret = Start();*

*}*

Después se eliminará o comentará todo el método *Start()*. Y por último, se modificará el método *Update()* de la clase para que llame a los métodos *PreUpdate()*, *Update()* y *PostUpdate()* de las otras clases. Nuestro método *Update()* ahora será de la siguiente manera:

*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)->PreUpdate();*

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

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

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

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

*return ret;*

*}*

Ahora iremos al fichero de cabecera donde borraremos la declaración de la clase *Start()* que ha sido eliminada.

MODIFICACIONES EN MODULE.H

Como hemos eliminado el método *Star()*, para completar su eliminación debemos entrar en el fichero *Module.h* y eliminar o comentar su declaración.

MODIFICACIONES EN MODULEINPUT

En esta clase debemos eliminar el método *Start()*. Aprovechando que estamos en este módulo, vamos a limpiar un poco el código y haremos que la ventana sólo pueda cerrarse si se cierra mediante el ratón. Para ello comentaremos o eliminaremos la constrnte *MAX\_KEYS* y vaciaremos de contenido el constructor de la clase. También redefiniremos el método *Update()* que quedará de la siguiente manera:

*update\_status ModuleInput::Update()*

*{*

*SDL\_PumpEvents();*

*keyboard = SDL\_GetKeyboardState(NULL);*

*return UPDATE\_CONTINUE;*

*}*

Por último, eliminaremos o comentaremos el método *GetWindowEvent*.

Del fichero de cabecera eliminaremos o comentaremos los dos *enum* que teníamos declarados. También eliminaremos los métodos *Start()* y *GetWindowEvent()*. Y en la parte *private* sólo nos quedará la siguiente variable:

*private:*

*const Uint8 \*keyboard = NULL;*

MODIFICACIÓN EN MODULETEXTURES

Tras modificar la clase *Input()* iremos a la clase  *Texture()* donde volveremos a eliminar el método *Start()*.

MODIFICACIÓN EN MODULEWINDOW

En la clase Window volveremos a eliminar el método *Start()*, tal y como hemos realizado en todas las clases anteriores.

MODIFICACIONES EN MODULERENDER

Este es el módulo encargado de pintar el pantalla. Así pues, las modificaciones en él serán algo mayores. Lo primero que haremos será incluir el fichero de cabecera de la librería de *Glew*:

*#include "glew-2.1.0\include\GL\glew.h"*

Acto seguido redefiniremos todo el método Init() de la siguiente manera:

*bool ModuleRender::Init()*

*{*

*LOG("Creating Renderer context");*

*SDL\_GL\_SetAttribute(SDL\_GL\_CONTEXT\_MAJOR\_VERSION, 3);*

*SDL\_GL\_SetAttribute(SDL\_GL\_CONTEXT\_MINOR\_VERSION, 0);*

*SDL\_GL\_SetAttribute(SDL\_GL\_CONTEXT\_PROFILE\_MASK, SDL\_GL\_CONTEXT\_PROFILE\_CORE);*

*SDL\_GL\_SetAttribute(SDL\_GL\_DOUBLEBUFFER, 1);*

*SDL\_GL\_SetAttribute(SDL\_GL\_DEPTH\_SIZE, 24);*

*SDL\_GL\_SetAttribute(SDL\_GL\_STENCIL\_SIZE, 8);*

*context = SDL\_GL\_CreateContext(App->window->window);*

*glewInit();*

*glHint(GL\_PERSPECTIVE\_CORRECTION\_HINT, GL\_NICEST);*

*glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);*

*glEnable(GL\_DEPTH\_TEST);*

*glEnable(GL\_CULL\_FACE);*

*glFrontFace(GL\_CCW);*

*glEnable(GL\_TEXTURE\_2D);*

*glClearDepth(1.0f);*

*glClearColor(0.3f, 0.3f, 0.3f, 1.f);*

*int width, height;*

*SDL\_GetWindowSize(App->window->window, &width, &height);*

*glViewport(0, 0, width, height);*

*return true;*

*}*

Expliquemos el código del bloque con un poco más de detalle. En las primeras 6 líneas, aquellas que empiezan con *SDL\_GL\_SetAttribute SDL\_GL\_SetAttribute* se está obteniendo información de diferentes atributos de SDL como puede ser la versión, la profundidad, … Posteriormente, en la línea *context = SDL\_GL\_CreateContext(App->window->window)*, se está definiendo el contexto de la ventana y tras esto pasa a inicializarse Glew mediante la sentencia *glewInit()*.

Con glHint estamos controlando la corrección de la perspectiva que se ha definido como la más precisa posible.

Con *glBlendFunc* estamos pintando los píxeles de pantalla y posteriormente con glEnable estamos habilitando la comparación y actualización de los buffers de profundidad y polígonos de desecho. También se están habilitando texturas de 2D.

Con *glFrontFace* estamos eliminando los polígonos de las caras ocultas lo que nos da una mayor velocidad en el pintado.

Con *glClearDepth* estamos especificando la profundidad del buffer de limpieza. Mientras que con *glClearColor* estamos especificando la profundidad del rojo, verde, azul y canal Alpha.

Posteriormente declaramos las variables de altura y anchura de la ventana y le indicamos que las obtenga del módulo ventana en la siguiente línea. Por último, con *glViewport* estamos pasando la coordenada del vértice inferior izquierdo en píxeles e indicando la altura y anchura de la misma.

Tras esto, eliminamos el método *Start()* y dejamos los métodos *PreUpdate()*, *Update()* y *PostUpdate()* tal y como están. Para definir a continuación el método *WindowResized()* que será el encargado de redimensionar la ventan de la siguiente manera:

*void ModuleRender::WindowResized(unsigned width, unsigned height)*

*{*

*glViewport(0, 0, width, height);*

*}*

Por último, borraremos o comentaremos el método *Blit()*.

Si hemos realizado todos los pasos de forma correcta, al ejecutar la alicación nos deberá aparecer una ventan con un triángulo en su interior como el de la siguiente figura:

