Introducción a la programación en C++

Introducción al C++ 2
- Tipos Básicos de datos 2
- Utilizando la librería estándar de C++ 2
- Polimorfismo 3
- Templates 4
- Operadores 5
- Otras utilidades 5
- Protocolo de programación 6

Protocolo de Inicio y Fin 7
Patrones de Diseño 9
- Singleton 9
- Observer – Subject 9
- Facade 9
- Factory 10
- Entidad – Componente 10

Estructura de un videojuego 12
Creación de un proyecto en Visual Studio 14
- Rellenando la Base 20
  - Añadiendo Maths 20
  - Añadiendo parser de XML 21

Introducción al C++

El objetivo de esta introducción a la programación en C++ es que refresquéis y consolidéis todos los conceptos básicos del C++ y al mismo tiempo acostumbraros a una metodología utilizada en las industrias de videojuegos actuales. La idea es que durante todo el proyecto del master os acostumbréis a una serie de patrones y protocolos de programación que se utilizan en la industria actual ya que de esta forma estaréis más preparados.

Tipos Básicos de datos

Los tipos básicos son los siguientes:

Tipo	Tam. Bits	Dígitos de precisión	Rango	
			Min	Max
Bool	8	0	0	1
Char	8	2	-128	127
Signed char	8	2	-128	127
unsigned char	8	2	0	255
short int	16	4	-32,768	32,767
unsigned short int	16	4	0	65,535
Int	32	9	-2,147,483,648	2,147,483,647
unsigned int	32	9	0	4,294,967,295
long int	32	9	-2,147,483,648	2,147,483,647
unsigned long int	32	9	0	4,294,967,295
long long int	64	18	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	64	18	0	18,446,744,073,709,551,615
Float	32	6	1.17549e-38	3.40282e+38
Double	64	15	2.22507e-308	1.79769e+308

const: indica que una variable no puede variar durante la ejecución del programa.
static: indica que una variable es única e invariable durante la ejecución de un programa y al mismo tiempo es global aunque está normalmente asociada a una clase.

Utilizando la librería estándar de C++

La stl es una librería estándar de C++ que se ha convertido en imprescindible para la mayoría de empresas del sector. Las funcionalidades que aporta son básicamente la de contenedores de varios tipos: vector, lista, mapa, sets, colas, pilas, etc.

En nuestro caso veremos solamente ejemplos del vector y del map, que son las estructuras más utilizadas ya que una vez se conoce su forma de funcionar ya puedes utilizar las demás estructuras.

Si abrimos el fichero *std_main.cpp* veremos que hay ejemplos de:

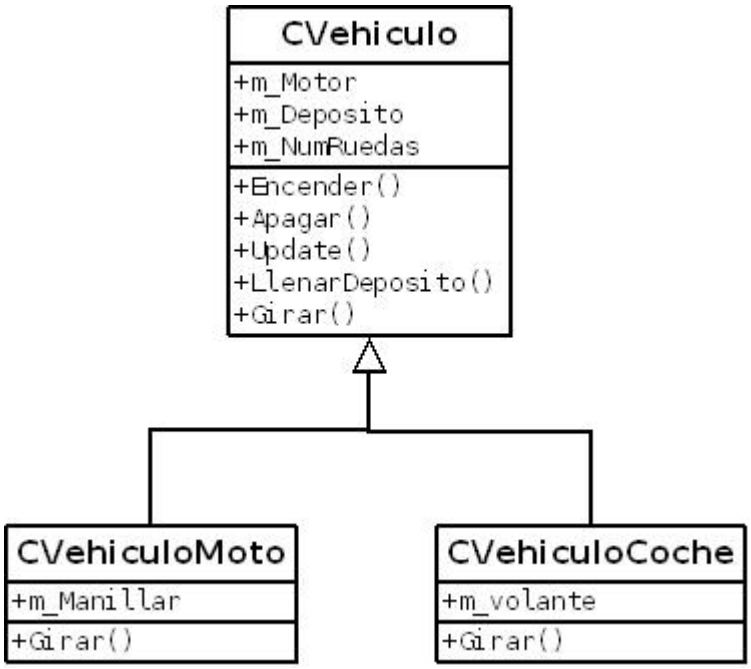
- Reservar memoria
- Añadir elemento
- Iterar / buscar elementos
- Modificar datos

- Eliminar dato
- Usar algoritmos de *algorithm.h*: *sort, find_if, for_each*
- Crear predicados

Polimorfismo

El polimorfismo consiste en definir tipos de clases con una similitud parecida pero con funcionalidades distintas. Una clase base o padre contendrá los tipos y funcionalidades comunes para todas sus hijas, las cuales contendrán funcionalidades y tipos específicos.

El polimorfismo tiene una regla básica que es el encapsulado de sus miembros. Una clase externa jamás debería poder acceder a un miembro de una clase directamente sino que debería acceder a través de funciones de la misma. Por esta misma razón los miembros de una clase los añadiremos como *protecteds*.



Si abrimos los ficheros *polimorfismo.h* y *.cpp* veremos un ejemplo claro de polimorfismo.

Siempre que se usa herencia es importantísimo hacer que el **destructor de la clase padre sea virtual** para que cuando se elimine el objeto hijo también se libere memoria de los datos contenidos por la clase padre.

Virtualidad y virtualidad pura

Si se quiere que una función de la clase del padre pueda ser sobre-escrita hará falta indicar delante de dicha función que es virtual. Si la función es virtual pura significa que las clases hijas están obligadas a implementar dicha función, sino el compilador fallará:

```
virtual bool   Init       ();
virtual bool   Init       () = 0;
```

← function virtual

← function virtual pura

Sobrecarga y sobre-escritura
Las funciones de la clase padre pueden ser sobre-escritas por las clases hijas. Por ejemplo imaginemos el método *Girar()*. La moto tendrá su propio método de *Girar()* con el manillar y en cambio el coche tendrá el suyo propio con el volante.

La sobrecarga por otro lado no es necesaria que ocurra bajo herencia sino que cualquier clase puede realizarla. Consiste en tener dos funciones con el mismo nombre pero con diferentes argumentos.

Downcast y upcast
Esto ocurre cuando en un objeto polimórfico se accede a la memoria de la clase padre o a la inversa. Miremos el siguiente ejemplo:

```
//-----------------------------
//-- definición clases
//-----------------------------
class A
{
};

class B : public A
{

};


//-----------------------------
//-- main.cpp
//-----------------------------
B* b = new B;
A* a = new A;

// Upcast
a = b;

// Downcast
c = static_cast<B*>(a);  a = b;
```

Templates
Los templates son muy útiles cuando se quiere crear una clase genérica, es decir una clase que realiza una misma funcionalidad para diferentes tipos de datos. De esta forma nos ahorramos el tener que definir una clase para cada tipo. Un ejemplo clarísimo serían los contenedores de la stl. El compilador automáticamente genera el código por ti cuando detecta una declaración de un template.

Estas clases suelen desarrollarse enteritas en ficheros de cabecera *.h*. En algunos casos para mantener el código más legible se separa en ficheros *.h* (declaración de la clase) y *.inl* (cuerpo de los métodos de la clase).

Si abrís algún fichero de la librería de matemáticas que os he pasado veréis varios ejemplos (Base\Math\).

Operadores

Algunas veces nos será útil definir operadores específicos en nuestras clases. Un operador puede ser cualquier símbolo de estos: +, -, *, /, ^, etc.

La siguiente línea define un operador en la clase *CClassA* con el símbolo + donde el argumento de entrada es otra C*ClaseA* y el valor de retorno es una *CClassA*.

```
class CClassA
{
public:
  CClassA() {}
  CClassA(float _x) : x(_x) { }
  ~CClassA() {}
public:
  float x;

  CClassA  operator + (const CClassA& otro) const;
}


CClassA CClassA::operator + (const CClassA& otro) const
{
  return (CClassA(this.x + otro.x));
}
```

Un ejemplo claro es el que encontraremos dentro de la librería matemática en *Vector3* (Base\Math\).

Otras utilidades

Macros: las macros son trozos de código al que se le ha dado un nombre. Donde se use su nombre, éste será reemplazado por el código que la define.

```
#define CHECKED_DELETE(x) if(x!=NULL) {delete x; x=NULL;}
```

Pragma once: comando que se suele añadir al principio de los ficheros de cabecera *.h*. Su objetivo es impedir que el fichero sea compilado más de una vez. Agiliza la compilación de un proyecto.

```
#pragma once // para que sea incluido una sola vez
```

Precompiled headers: las cabeceras precompiladas suelen usarse para ahorrarse tiempo de compilación en proyectos de grande tamaño. Son ficheros de cabecera *.h* que contienen los includes de las clases más utilizadas, como por ejemplo los includes de la librería stl, y que se compilan una sola vez para todo el proyecto. El Visual Studio tiene una opción en propiedades del proyecto para poder añadir precompiled headers.

Declaraciones avanzadas: una declaración avanzada nos servirá para definir un tipo de dato que se utiliza en un fichero *.h* cabecera y ahorrarnos tener que añadir el incluye de la propia clase. Ahorra tiempo de compilación.

```
// declaración avanzada
class CAlumno;

// Includes
// …

//-------------------------------------------
//-- class
//-------------------------------------------
class CClassSample
{
public:

private:
  CAlumno*   m_pAlumno;
};
```

Protocolo de programación

En el proyecto del master la idea es que os acostumbréis a programar todos de la misma manera. De este modo si os intercambiáis código os será mucho más fácil entender lo que ha escrito vuestro compañero. Todas las empresas de la industria usan protocolos de programación y te obligan a adaptarte a ellos. En nuestro caso utilizaremos un protocolo parecido a la notación húngara.

Clases: *CNombreClase* → añadiremos C
Estructuras: *SNombreEstructura* → añadiremos S
Enumeradores: *ENombreEnumerador* → añadiremos E

Enteros: *iEntero* → añadimos i
Flotantes: *fFlotante* → añadiremos f
Booleano: *bBooleano* → añadiremos b
Carácter único: *cCaracter* → añadimos una sola c
Cadena de caracteres: *szCadena* → añadimos sz (string zero)
Punteros: pPointer → añadimos p

Miembros de una clase: *m_iMiembro* → añadimos m_ delante seguido del tipo.
Vectores, maps, listas: *AlumnosVector* → añadimos la palabra clave al final (vector, map o list)

Argumentos de entrada: *_ArgumentoIn* → añadiremos _ delante
Argumentos de salida: *ArgumentoOut_* → añadiremos _ al final

Funciones recursivas: *_FunciónRecursiva* → Añadiremos _ delante

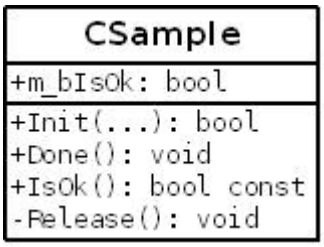
Nombres de funciones empiezan siempre con la primera letra en mayúsculas y si son varias palabras siempre la primera palabra en mayúsculas: *void EstoEsUnaFuncion(…)*

Tipos de datos typedef: *tAlumnosContainer* → añadimos la t

Variables globales: *g_GlobalVariable* → añadimos la g

Protocolo de Inicio y Fin

En vuestro proyecto utilizaréis un protocolo de Inicialización y Finalización de datos para tener un mayor control de la gestión de memoria en vuestras clases. De esta forma todos los programadores que trabajéis sobre el mismo proyecto sabréis dónde tendréis que ir cuando queráis inicializar o liberar memoria de una variable agilizando de este modo el desarrollo de vuestro juego. Cada clase tendrá los siguientes métodos y variables:



***m_bIsOk**:* Variable que controla si la clase se ha inicializado bien.
***Init**:* Función de inicialización. Retorna un trae si todo ha ido bien, sino false. Asigna la variable *m_bIsOk.*
***Done**:* Comprueba si la clase se ha inicializado bien y si es así llama a la función *Release*.
***Release**:* Libera la memoria de todas las variables de la clase. Es protected.
***IsOk**:* Retorna *m_bIsOk* y nos sirve para comprobar que la clase se ha inicializado correctamente.

Constructor: No recibirá parámetros de entrada y solamente asignaran valores a las variables miembro.
Destructor: simplemente llamará a la clase *Done()*.

```
class CSample
{
public:
  CSample() : m_bIsOk(false){}    // constructor
  virtual ~CSample() { Done(); } // destructor virtual por si polimorfismo

  // Protocolo inicio/fin
  bool           Init       (…);
  void           Done       ();
  bool           IsOk       () const { return m_bIsOk; }

protected:
  void          Release   ();

  // Members
  bool            m_bIsOk;
}
```

Si abrís los ficheros *classsample.h* y *.cpp* tenéis un ejemplo completo.

Si llega el caso en que tenéis una clase que hereda de otra entonces en la clase padre la función *Init(…)* sería virtual y la clase hija no contendría la variable *m_bIsOk* ya que ya la tendría el padre. Por otro lado el *.cpp* de la clase hija quedaría de la siguiente forma:

```
bool CChild::Init ()
{
  // Init de la clase padre
  bool bIsOk = Inherited::Init();

  if (!bIsOk)
  {
    // Llamamos a Done() en vez de Release() ya que tenemos que liberar la
    // memoria creada por la clase padre.
    Done();
  }

  return bIsOk;
}

void CChild::Done ()
{
  // Done() de la clase padre
  Inherited::Done();

  if (IsOk())
  {
    Release();
  }
}
```

Inherited: es un tipo definido en la clase hija que define la clase padre. De este modo nos olvidamos del nombre de la clase padre y siempre usamos inherited para dirigirnos a ella:

```
typedef CParent Inherited;
```

Tenéis un ejemplo funcional en los ficheros *polimorfismo.h* y *.cpp*.

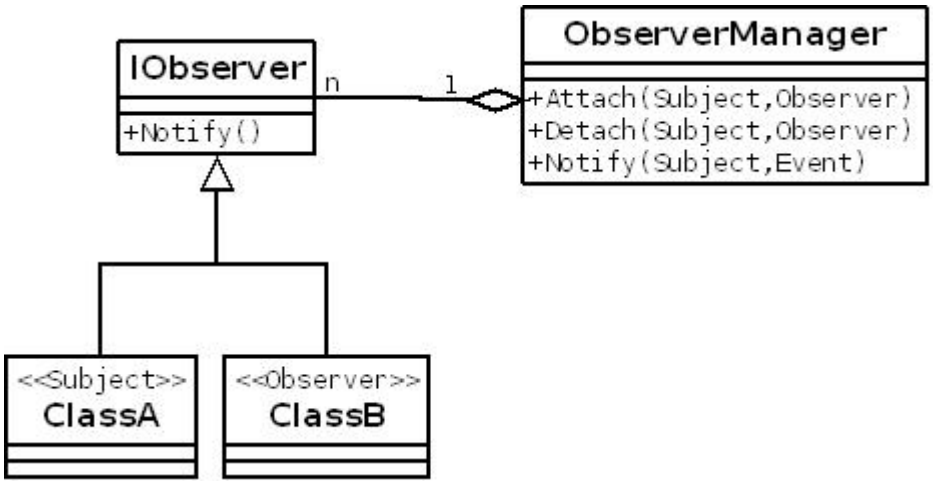
Patrones de Diseño

Singleton

El singleton es un patrón de los más comunes y se utiliza para poder manipular un objeto único instanciado en todo un proyecto sin necesidad de tener una referencia del mismo y pudiendo acceder a el de forma global. Tenemos un ejemplo en la clase *SingletonPattern.h.*

Observer – Subject

Este patrón se utiliza para notificar acciones entre clases. Cuando el *Subject* lo necesita envía una notificación al *Observador*. La forma más adecuada de desarrollar este patrón en proyectos grandes es añadiendo a la arquitectura un ObserverManager que gestiona los links entre Subject y Observer.

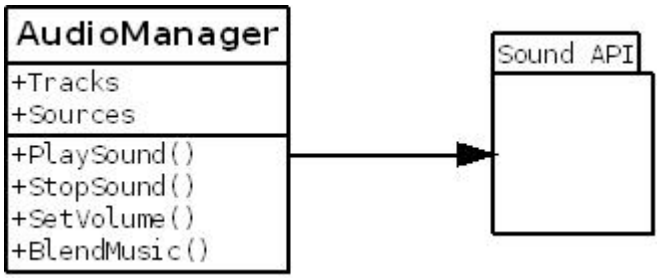


Las instrucciones serían algo así:

```
CObserverManager* pObsMgr = GetWorld()->GetObserverManager();
pObsMgr->Attach(pClassA, pClassB);
pObsMgr->Notify(pClassA, ON_ENTER_EVENT);
```

Facade

Este patrón consiste crear una interfaz amigable por encima de una API, es decir, sirve para encapsular las funcionalidades de una API para que su uso sea más cómodo. Este patrón lo veremos en varios sistemas: Audio, XML parser, etc. Nos facilita el uso de una API y al mismo tiempo nos permite poder añadir una o más APIs por debajo de la misma interfaz con lo que el código de la aplicación se mantendrá intacto y solo habrá que implementar la parte de la interfaz.



Factory

Este patrón centraliza la creación de objetos del mismo tipo en una sola clase. Pongamos el ejemplo de unas clases como coche, moto, tren, avión que heredan de una clase base vehículo. El factory se encargaría de crear memoria para el vehículo en concreto pasándole simplemente el tipo de vehículo que se desea. Sería algo así como:

```
CVehicle* pVehicleAirplane = m_pVehicleFactory->CreateVehicle(EVEHICLE_PLANE);
```

Internamente tendría la siguiente forma:

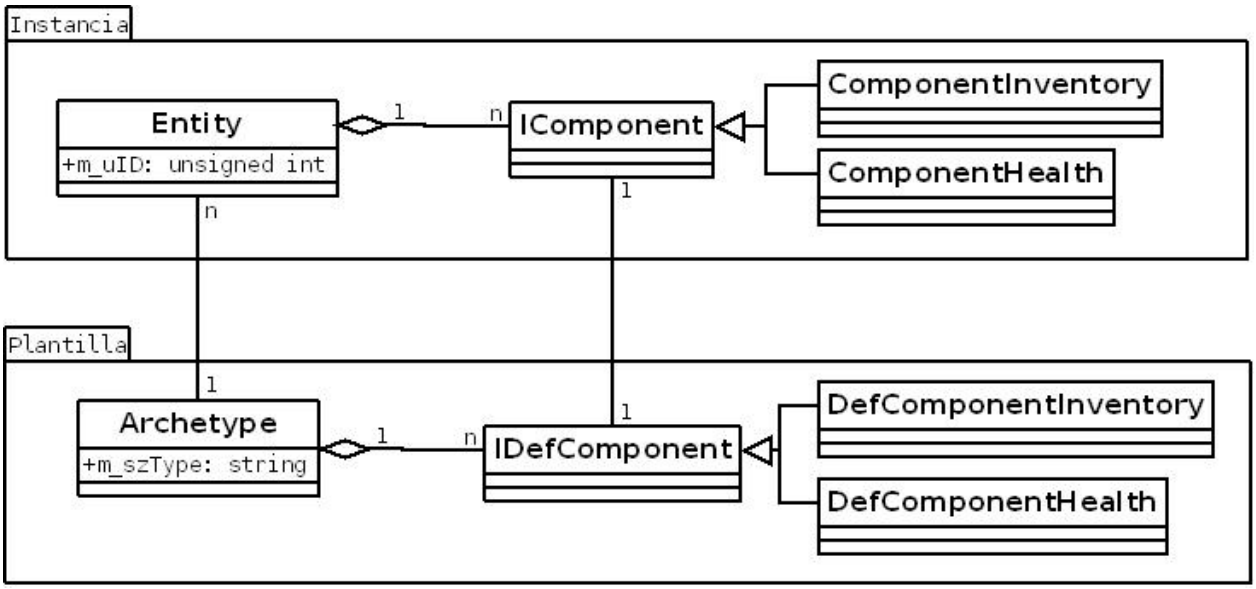
```
CVehicle* pVehicle = NULL;

  switch(eVehicleType)
  {
    case EVEHICLE_CAR:
      pVehicle = new CVehicleCar;
      break;
    case EVEHICLE_MOTO:
        pVehicle = new CVehicleMoto;
      break;
    case EVEHICLE_PLANE:
      pVehicle = new CVehiclePlane;
      break;
    default:
      // Log error: vehicle type undefined
  }

  if (pVehicle)
  {
    bool bIsOk = pVehicle->Init(/*....*/);

    if (!bIsOk)
    {
      // Error log
    }
  }

  return pVehicle;
```

Entidad – Componente

Este patrón es cada vez más común en las industrias de videojuegos ya que permite mucha flexibilidad y control sobre las entidades de una aplicación. Una entidad es un

ente sobre un mundo, cualquier cosa, sea física o no, que pueda instanciarse dentro de un mundo virtual. La entidad tendrá un ID que la identificará y un arquetipo asociado. El arquetipo consiste en una plantilla con un conjunto de definición de componentes. Estos componentes son los encargados de darle personalidad a la entidad. Una entidad estará definida por uno o varios componentes que nos especificarán su comportamiento dentro del mundo virtual.

Pongamos por ejemplo una caja: ¿Qué componentes le harán falta para ser una caja?
- Componente gráfico (que se vea en la pantalla la caja)
- Componente físico (que la caja pueda romperse y/o moverse)
- Componente de material (madera., metal, etc)
- etc.



Si quisiéramos que la caja fuera inteligente y se moviera le añadiríamos el componente Brain y así hasta que nuestra entidad vaya obteniendo la forma que deseamos.

La ventaja de este sistema es que todo nuestro mundo virtual se reduce a un conjunto de entidades, cada una con sus propiedades, con lo que podemos mantener todas estas entidades almacenadas en un vector con la información mínima de todas ellas que sería el ID de identificación, el tipo y un vector de componentes.

Estructura de un videojuego

Como ya habréis visto un Engine se compone de los siguientes módulos: Audio, Gráficos, Gameplay, GUI, Física, Network, etc. Nosotros vamos a ir más allá y vamos a ver qué es lo que realmente contiene cada uno de éstos módulos y vamos a organizarlos de forma que nos sirva más adelante para configurar de forma ordenada nuestra solución de visual studio.

Lo primero de todo y lo más importante es lo que llamaremos proyecto **Base**. Éste proyecto contendrá diferentes herramientas que nos serán útiles en cualquier otro módulo, como por ejemplo: gestión de input, librería de matemáticas, gestión de memoria, sockets, timers, parseador de XML, logger, y todo lo que se os pueda imaginar que pueda ser compartido entre diferentes módulos.

El módulo de **Gráficos** se encargará de encapsular todas las funcionalidades de renderizado, lectura de mallas, shaders, etc. Cuando decimos encapsular nos referimos a crear una capa o interfaz que nos esconda llamadas a piezas de código de una API en concreto, como por ejemplo Directx o OpenGL. Encapsular código es importante ya que nos ahorrará en un futuro tener que modificar miles de líneas de código por todos los proyectos si una API ha sido actualizada/modificada. Por otro lado nos será bastante más fácil añadir otras APIs sobre la misma interfaz sin necesidad de modificar el resto de código.

El módulo de **Física** encapsulará una API física como por ejemplo PhysX, AGEIA, Coldet, etc. Esta parte la rellenaréis más adelante.

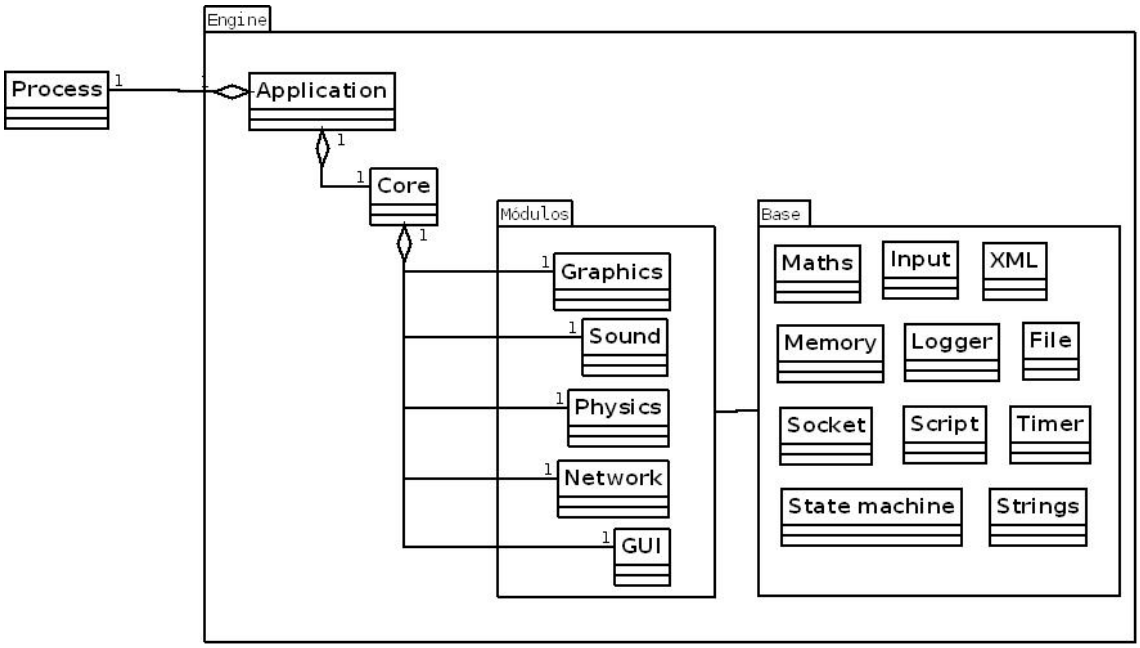
El módulo de **Audio** nos encapsulará la API de audio que en nuestro caso utilizaremos la librería BASS. Esta parte la realizaréis más adelante. Otras podrían ser FMOD, OpenAL, etc.

El módulo de **GUI** va a contener la gestión de ventanas, botones, scrolls y demás, que vosotros crearéis y utilizaremos para generar los menús y HUD de nuestro juego.

El módulo **Network** va a gestionar el tráfico de datos a través de la red entre un servidor y varios clientes. Esta parte la realizaréis también más adelante.

El proyecto **Core** es el que se encarga de inicializar los diferentes módulos y de gestionar el proceso principal de la aplicación.

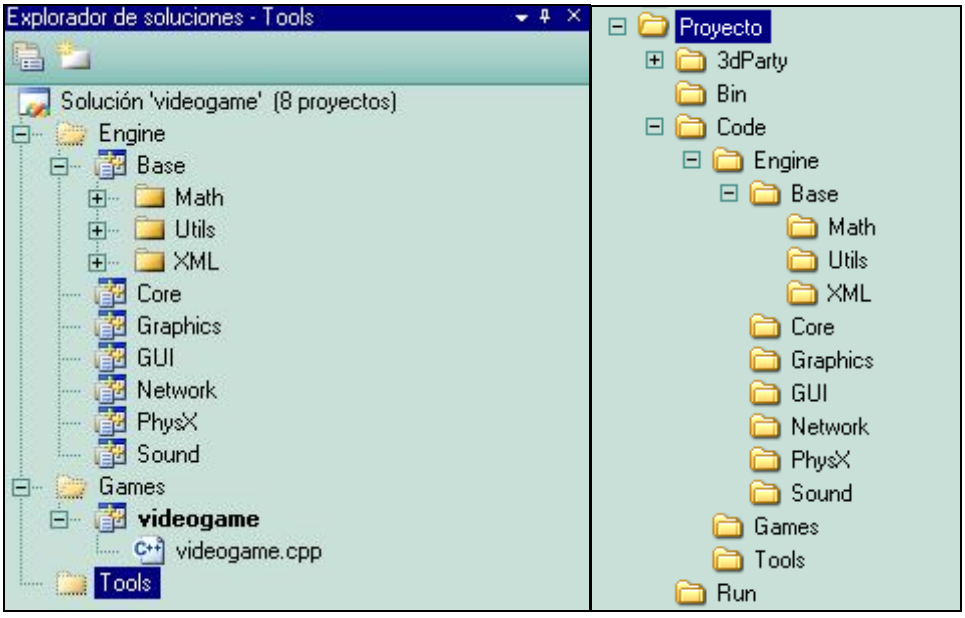
Dejando a un lado el Engine, necesitamos la parte del juego dónde se definen las reglas, es decir lo que hace que el juego sea jugable (qué es lo que pasa cuando el usuario le da al input). Esta parte o módulo suele llamarse Gameplay y es donde añadiremos todo lo relacionado con el player, enemigos, eventos, etc.



Pasemos pues a crear nuestra solución de Visual Studio con todos los módulos que hemos mencionado.

Creación de un proyecto en Visual Studio

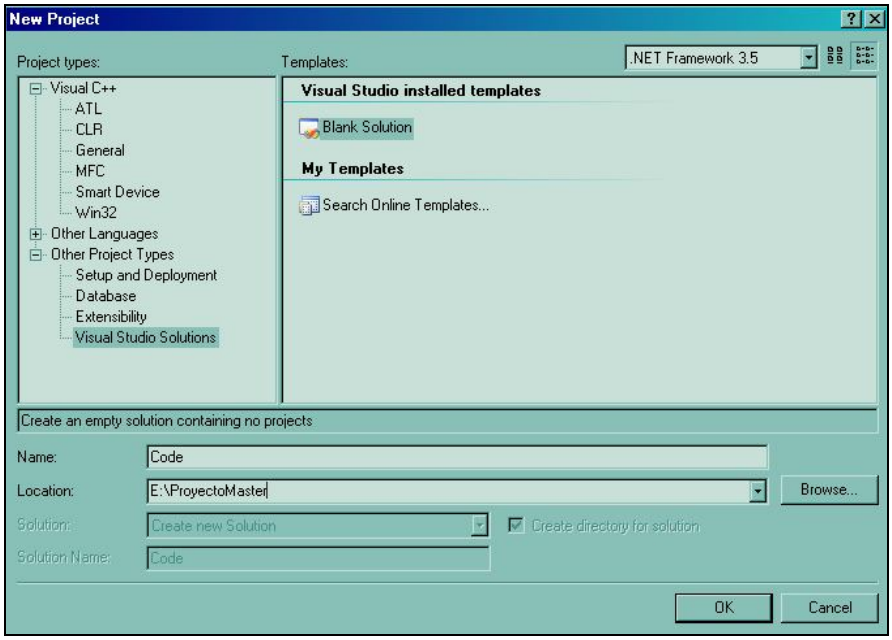
Nuestro objetivo es conseguir obtener una solución cómo la que observamos en la imagen de la izquierda y la estructura de carpetas de windows como la imagen de la derecha.



Solución y aplicación principal

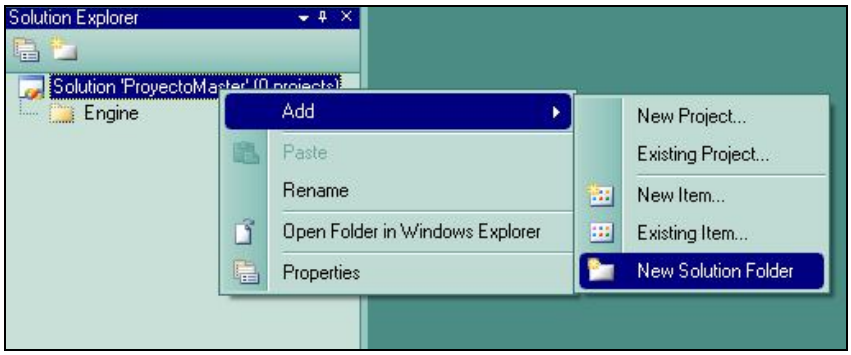
1. Creamos una carpeta que contendrá todo nuestro proyecto que llamaremos por ahora *ProyectoMaster*.

2. Creamos una solución vacía que llamaremos *Code*:

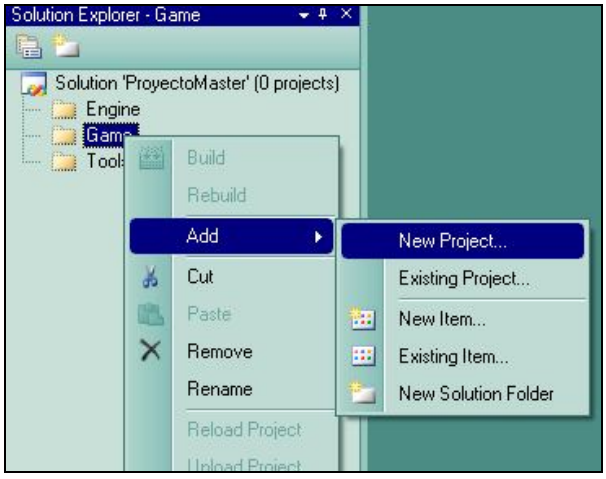


Si observamos en el explorador de windows se ha creado una carpeta llamada *Code* que contiene la solución del proyecto. Una solución puede contener tantos proyectos como se desee. Esto es muy útil cuando se trabaja en un equipo grande con varios programadores, de tal modo que cada uno trabaja en un proyecto específico. Eso hace que hayan muchos menos conflictos de código. Por otro lado es importante que tengamos el proyecto bien organizado para poder encontrar los ficheros de forma rápida y eficaz.

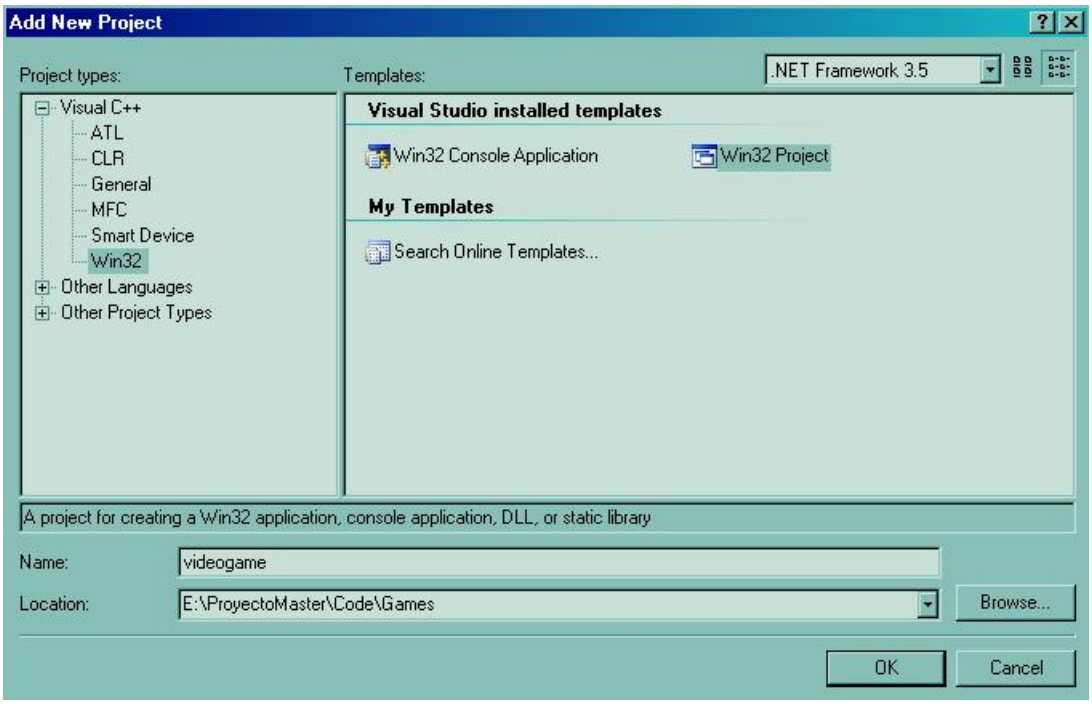
3. Ahora crearemos varias carpetas dentro de la solución: *Engine*, *Game* y *Tools*



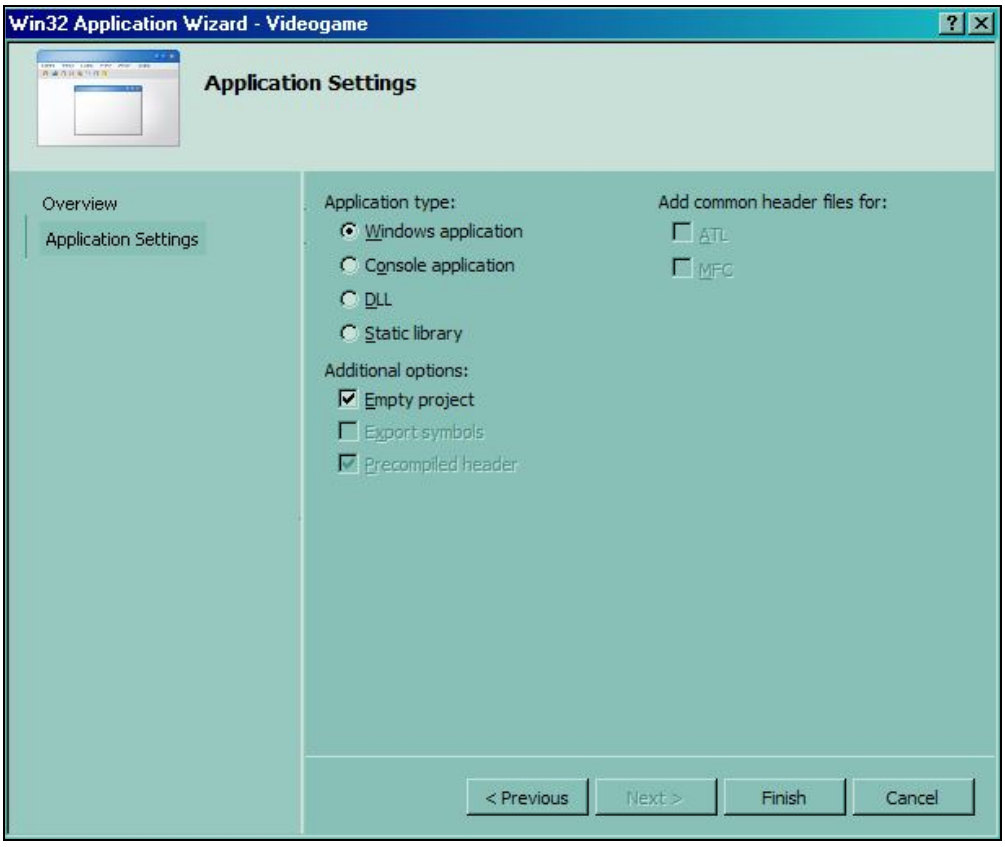
4. Y dentro de la carpeta *Game* crearemos un nuevo proyecto vacío *Win32* con el nombre de *Videogame*:



5. Tal y como muestra la imagen crearemos el proyecto *videogame* en la nuestra carpeta de windows *ProyectoMaster\Code\Game* y le daremos a *OK*:



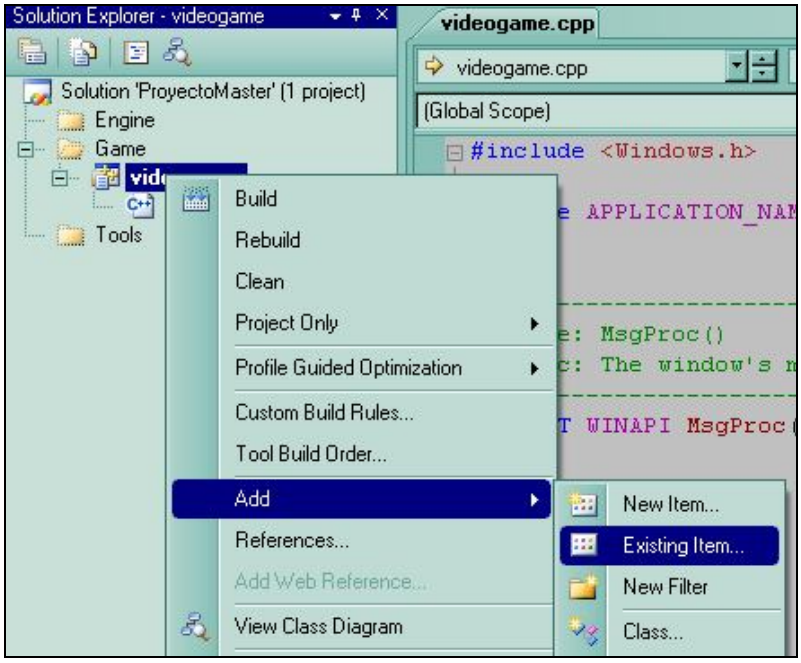
a. A continuación **NO** le damos a *Finish* sino que le damos a *Next* y marcamos las opciones que vemos en la siguiente imagen:



b. Le damos a *Finish*.

6. En el proyecto *videogame* añadimos un fichero existente .cpp que os pasaré llamado videogame.cpp que contiene el main principal de la aplicación. Lo **copiáis** dentro de *ProyectoMaster\Code\Game\Videogame*

7. Lo añadimos a nuestro proyecto *videogame* de Visual Studio:



8. Ahora vais a las propiedades del proyecto y en General cambiáis el juego de caracteres de *unicode* a *multibyte* **para todas las configuraciones Debug y Release**. Esto lo tendréis que hacer con cada nuevo proyecto que creéis y nos permitirá usar caracteres de 1 o 2 bytes.

9. Comprobad que el programa se compila y ejecuta correctamente tanto en Release como en Debug.

10. En las propiedades del proyecto en General:

    a. Aplicaciones windows (EXE)
        i. Directorio de salida → $(SolutionDir)\bin\$(ProjectName)\
    b. Librerías (LIB)
        i. Directorio de salida → $(SolutionDir)\bin\
    c. Directorio Intermedio → $(SolutionDir)\bin\Intermediate\$(ProjectName)$(Configuration)\

Esto nos generara los ejecutables en la carpeta *Bin* a la misma altura que la carpeta *Code*. Esto lo haremos para cada uno de los proyectos que creemos. **Para todas las configuraciones Debug y Release.**

11. En las propiedades del proyecto en Depuración → Comando
    a. En Debug ponemos $(ProjectName)_d.exe
    b. En Release ponemos $(ProjectName).exe

12. En las propiedades del proyecto en Depuración → Directorio de trabajo
    a. En Debug y en Release ponemos $(SolutionDir)\Bin\$(ProjectName)\
13. Ejecutad y comprobad que se os genera la carpeta *Bin* a la misma altura que *Code* tanto en Debug como en Release.

14. Seguimos en Propiedades vamos a C/C++ → General → Tratar advertencias como errores. Marcamos Si. Nos será útil para prevenir los errores de precisión durante la programación del código.

15. Seguimos en Propiedades vamos a C/C++ → Generación de código y lo cambiamos:
    a. En Release lo ponemos en Multiproceso
    b. En Debug lo ponemos en depuración Multiproceso.

Esto lo haremos para cada uno de los proyectos que creemos.

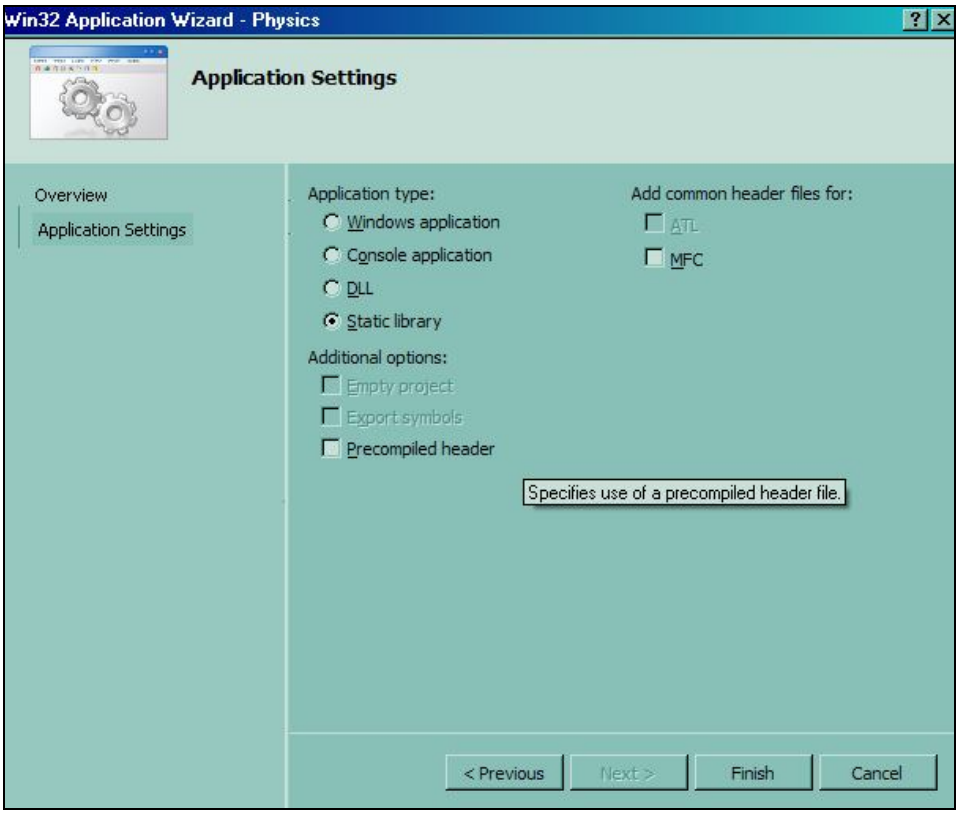
16. Vamos a Vinculador → Depuración → Generar archivo de base de datos de programa
    a. En Debug ponemos $(TargetDir)$(TargetName)_d.pdb
    b. En Release ponemos $(TargetDir)$(TargetName).pdb

17. Vamos a Vinculador → General → Archivo de resultados:
    a. Aplicaciones windows (EXE)
        i. Debug → $(OutDir)\$(ProjectName)_d.exe
        ii. Release → $(OutDir)\$(ProjectName).exe
    b. Librerías (LIB)
        i. Debug → $(OutDir)**Lib**\$(ProjectName)_d.lib
        ii. Release → $(OutDir)**Lib**\$(ProjectName).lib

Proyectos del Engine

Ya tenemos preparado el proyecto principal de nuestra solución. Ahora pasaremos a crear los proyectos que conjuntamente darán forma a nuestro motor del juego.

18. Crearemos más proyectos desde visual studio en la carpeta Engine con el siguiente nombre: Core, Base, Graphics, GUI, Network, PhysX, Sound. Deberán ser de tipo Proyecto Win32 y Biblioteca estática **SIN** encabezado precompilado.



Cada parte del engine creará una lib que nosotros deberemos linkar en nuestra aplicación principal para poder usar sus funciones (lo veremos más adelante).

19. Para cada nuevo proyecto haced los pasos 8, 10, 14, 15 y 17 para todos estos nuevos proyectos (el paso 14 y 15 solamente se podrá realizar cuando se añada al proyecto ficheros de código .cpp). Si seleccionáis todos los proyectos en Visual Studio y le dais a propiedades con el botón derecho podréis modificarlos todos a la vez.

Ya tenemos los módulos básicos de un engine definidos cada uno con sus proyectos. Durante todo el master los iremos rellenando poco a poco y os iréis acostumbrando a esta estructuración. La configuración de un proyecto puede parecer muy complicada pero es una parte esencial ya que un proyecto bien configurado puede ahorrarnos tiempo, problemas futuros y conflictos entre programadores.

20. Pasemos ahora a crear las carpetas donde guardaremos los recursos gráficos, xmls, scripts y demás. También crearemos una carpeta para añadir documentación, si fuera necesario y otras dos carpetas más: una para las APIs externas (bass, physx, libxml, etc) y otra para las posibles Tools que vayamos generando durante el curso (visualizador de mallas, consola de errores, etc). Iremos pues a la carpeta que contiene la carpeta *Code* y añadiremos: *Run*, *Run\Data*, *3dParty*, *Docs* y *Tools*.

Rellenando la Base

Añadiendo Maths

Empezaremos añadiendo al proyecto Base una librería básica de geometría que llamaremos Math. Entre los ficheros que os paso veréis que hay una llamada Maths y otra Utils. Las copiáis y las pasteáis en /Base/. Maths está compuesta por las clases templatizadas:

- Matrix33, Matrix34, Matrix44
- Vector2, Vector3 y Vector4
- En *MathTypes.h* hay todos los tipos de matrices y vectores definidos, entre otras utilidades como el número PI.
- En *MathUtils.h* tenemos utilidades como funciones de trigonometría, de redondeo, etc.
- Finalmente en *MathConstants.h* tenemos una serie de constantes definidas que podremos utilizar en nuestro código.
- En Utils tenemos *Types.h* donde hemos definido los diferentes tipos de enteros que nos pueden ser útiles.

Desde visual studio añadís en el proyecto Base una carpeta que llamaremos Maths y otra Utils y añadiremos todos los ficheros que hemos mencionado. Recordemos que al añadir ficheros de código al proyecto, en la ventana de propiedades del mismo se nos ha creado una nueva pestaña llamada C/C++ que deberemos configurar adecuadamente como ya indicamos en los pasos 14 y 15.

Si compilamos veremos que no se encuentran algunos incluyes. Esto es porque tenemos que añadir en el proyecto Base → Propiedades → C/C++ → General → Directorios de inclusión adicionales, la dirección absoluta de la carpeta Base → ..\Base\. Haced esto tanto para Release como para Debug y ya debería compilar correctamente.

Ahora vayamos al *videogame.cpp* y en el método WinMain añadamos la siguiente declaración: `Mat33f m;`

Si compilamos veremos que no se encuentra el valor de Mat33f. Esto es porque nos hace falta añadir la cabecera *MathTypes.h* al inicio del documento. Lo haremos así:

```
#include "..\Engine\Base\Math\Matrix33.h"
```

Si compilamos el compilador nos dirá que no encuentra esa cabecera con lo que deberemos indicarle en qué carpeta se encuentra a través de las propiedades del proyecto videogame: Base → Propiedades → C/C++ → General → Directorios de inclusión adicionales, añadimos ..\Engine\Base para Debug y Release.

Finalmente añadiremos en Vinculador→General→Directorios de bibliotecas adicionales, añadiremos ..\..\Bin\lib\ tanto para Release como Debug. Además, en Vinculador→Entrada→Dependencias adicionales, añadiremos Base.lib para Release y Base_d.lib para Debug.

Esto debería ser suficiente para poder usar el código de Maths junto con sus constantes y funcionalidades.

Añadiendo parser de XML

Empezaremos añadiendo al proyecto Base un parser de XML que nos será muy útil durante todo el proyecto. Veréis que en la carpeta que os paso XML hay 3 ficheros:

- Example.cpp
- XMLTreeNode.h
- XMLTreeNode.cpp

Tenemos que copiar esta carpeta junto con esos tres ficheros dentro de la carpeta de nuestro proyecto Code/Engine/Base/ y desde visual añadirlas al proyecto Base pero dentro de una carpeta que crearemos llamada XML.

Una vez hecho esto le daremos sobre el fichero Example.cpp con el botón derecho e iremos a propiedades donde en Propiedades de configuración → General → Excluir de la generación, marcaremos Si para todas las configuraciones Release y Debug (al ser un ejemplo no queremos que se compile).

Copiaremos también la carpeta 3dParty y la pastearemos en el mismo sitio donde se encuentra la carpeta Code. Como este parser utiliza la API libxml necesitaremos añadir sus includes en propiedades del proyecto. Vamos pues a las propiedades de Base y en C/C++ → General → Directorios de inclusión adicionales, añadimos con path relativo las carpetas de includes de iconv y libxml para todas las configuraciones Release y Debug. Finalmente copiad de las carpetas de 3dParty iconv y libxml las dll's y las copiáis en la carpeta \Bin\. Nos hará falta para poder ejecutar el exe de la aplicación que generemos.

Si compiláis dándole a F7 os debería funcionar correctamente, sino es que algo habéis hecho mal.

Utilizando el parser de XML

Ahora el programa ya nos encontrará la clase *XMLTreeNode* para poder trabajar con ella. No obstante nos hará falta también añadir los includes de iconv y libxml como ya hicimos anteriormente para el proyecto Base y también los libs de libxml que lo haremos en Vinculador → General (path de la carpeta lib de libxml) y Vinculador → Entrada (añadiremos lixml2.lib). Recordad que esto se hará tanto para Release como para Debug.

Ahora ya estamos a punto para poder utilizar la herramienta de XML. Hagamos una prueba para ver si todo funciona correctamente. Id al fichero Example.cpp y copiad todo ese código, pasteándolo en videogame.cpp en donde pone `// Añadir aquí el Init de la applicacioón` y ejecutais el programa dándole a F5. Si todo ha ido bien se os habrá generado un fichero llamado test.xml dentro de la carpeta /Game/.

```
- <Companies>
  - <Company name="FLO" location="Marina 13">
    - <Workers number="2">
        <worker name="Raul" surname="Riera" age="27" salary="232.399994" dead="true"/>
        <worker name="Juan" surname="Navas" age="46" salary="122.321404" dead="false"/>
      </Workers>
    </Company>
  </Companies>
```