Índice de contenido

Herramientas necesarias

Ejemplos realizados con VC++ 2008 Express Edition en español:

http://www.microsoft.com/express/download/#webInstall

Otros compiladores c++ gratuitos:

http://bloodshed.net/devcpp.html

http://www.delorie.com/djgpp/

http://www.gnu.org/software/gcc/gcc.html

Listado de motores de juego:

http://gpwiki.org/index.php/Game_Engines

http://www.devmaster.net/engines/list.php

Tipos de Variable

Los ejemplos de programación utilizan la librería 'iostream' que incorporan cualquier IDE de c++, pero lo normal es que en la creación del juego solo utilices las librerías del motor de juego (lo cual significa que las funciones utilizadas son distintas a los ejemplos).

De todos modos, el sentido del código es entender algunos aspectos de c++ como punteros, referencias y creación de funciones.

Casi todos los tutoriales de c++ que he encontrado comienzan por explicar las variables, y este documento no va a a ser una excepción, pero lo voy a plantear de forma más resumida y justificando la utilización de punteros y referencias en c++, que desagradan a la mayoría.

Durante el transcurso de este tutorial yo también he aprendido, pues surgieron nuevas dudas, y tuve que informarme, estudiar y ensayar (no tengo estudios informáticos), procesos necesarios si quieres aprender c++.

Tipos de variables (más comunes) y rango cuando son 'signed' (con signo, positivo o negativo), en la plataforma windows 32-bits. En otras plataformas puede variar. Con la variable booleana bool (true o valor 1 significaría sí, false o valor 0 significaría no) no se suele aplicar este distintivo, pero si lo aplicas no saldrá error de compilación.

Tipo	Rango	Bytes Requeridos	Valor	Distintivo
`bool`	true (1), false (0)	1	`true(1), false(0)`	
`signed char`	[–128, 127]	1	`Entero (Texto)`	
`signed short`	[–32768, 32767]	2	`Entero`	
`signed int`	[–2147483648, 2147483647]	4	`Entero`	
`signed long`	[–2147483648, 2147483647]	4	`Entero`	`L ó l (final)`
`signed float`	$\pm [1.2 \cdot 10^{-38}, 3.4 \cdot 10^{38}]$	4	`Decimal, con 7 dígitos precisión`	`f ó F (final)`
`signed double`	$\pm [2.2 \cdot 10^{-308}, 1.8 \cdot 10^{308}]$	8	`Decimal, con 15 dígitos precisión`	

Tipos de variable y rango cuando son 'unsigned' (Sin signo. A efectos, tendrían valor positivo), en la plataforma windows 32-bits. en otras plataformas puede variar. Aquí apreciamos que sólo los valores enteros pueden ser 'unsigned'.

Tipo	Rango	Bytes Requeridos	Distintivo
`unsigned char`	[0, 255]	1	`U o u (final)`
`unsigned short`	[0, 65535]	2	`U o u (final)`
`unsigned int`	[0, 4294967295]	4	`U o u (final)`
`unsigned long`	[0, 4294967295]	4	UL, uL, Ul o ul

Para utilizar variables en c++ primero hay que declararlas. Esta sería la forma de declarar:

signed/unsigned **TipoVariable** *NombreVariable*;

Si NO se especifica nada delante de **TipoVariable** la IDE asigna 'signed' por defecto. (es decir, entiende que el número lleva signo).

Vemos que al final de la instrucción añadimos el **punto y coma** (;). Este signo es muy **importante** ya que indica al IDE el final de la instrucción (excepciones la sentencia #include, y funciones).

Las variables tipo `char` guardan los datos del tipo texto (cadenas y caracteres), donde las cadenas de texto van entre comillas ("*cadena de texto*") y los caracteres entre comillas simples ('*A*'). El juego de caracteres tiene valores entre [0-255] pero **no es necesario** preceder a `char` con `unsigned` , ya que el IDE se encarga de asignar el rango de valores necesario en este caso.

Para que las variables de texto tipo `char` contenga un carácter bastará con:

```
char cCaracter='A';
```

Pero si debe contener una cadena de texto, deberemos crear una "matriz" cuyo valor (número de elementos que forman esa matriz) va encerrado entre corchetes [ ]. Las dimensiones de esa matriz debe tener como mínimo 'el número de letras de la cadena' + 1 (para el valor /0 que indica final de cadena a c++). Ejemplo:

```
char cCadena[10]="123456789";  // Dimensión matriz de texto = Número de letras + 1
```

Normalmente, se utiliza los corchetes sin ningún valor dentro para que la IDE asigne el valor necesario. Ejemplo:

```
char cCadena[]="Cadena de Texto";
```

Como hemos visto en el ejemplo anterior, al mismo tiempo que declaramos una variable también podemos asignarle un valor, utilizando el signo igual (=) o introduciendo el valor de la variable entre paréntesis (*ValorVariable*). Ejemplos:

```
bool bInterruptor = true;
char cLetra('Z');
char cCadena[]="Cadena de Texto";
int iNum(5);
unsigned int uiNum = 5U;
```

```
long lNum = 10L;
unsigned long ulNum = 50UL;
float fNum = 2153.335f;
float PI = 3.14159F;
double dNum = 1237.112123425533;
```

Es buena praxis preceder al nombre de la variable con un distintivo que indique el tipo de valor que contiene, para reconocerla mejor en otras partes del programa. (por ejemplo preceder con 'i' las variables del tipo int).

Vemos como el IDE de VC++ asigna diferentes colores al texto: azul (palabras clave de c++), rojo (expresiones de texto), verde (comentarios, se verá continuación) y negro (resto).

Tipos de comentarios en el código, utilizando // o el conjunto /* */

```
int iVar1;  // Comentario de final de línea

int iVar2 /* Comentario entre línea */, iVar3 /* Otro comentario entre línea*/;
```

```
int iVar4;          /* Comentario
                    de
                     múltiples
                     líneas */
```

Según el lugar del programa donde se declare la variable existen varios tipos (del mismo modo, aplicable a funciones).

Globales: Se declara la variable al inicio del código, fuera del interior de una función y antes de declarar las funciones. También podría ir dentro de una estructura (struct) o dentro de una clase (class con public:). Estos conceptos ya se verán después. Estas variables pueden ser utilizadas por todas las partes del programa.

Locales: Se declaran dentro de una función (sólo puede ser utilizada por esa función). También podría ir dentro de una estructura (struct con private:) o dentro de una clase (class), donde sólo podrán ser utilizadas por esa estructura o clase respectivamente. Podría llamarse *semi-global* si se declara entre varias funciones (esto no se suele hacer). Ejemplos:

FORMA CORRECTA	FORMAS INCORRECTAS

bool bGlobal(true);
 	
void funcion1 (void)
{
bool local1, local2;
local1=bGlobal;
local2=local1;
 	
}
 	
bool bSemiGlobal(false);
 	
void funcion2 (void)
{
bool local3, local4;
local3=bGlobal;
local4=bSemiGlobal;
funcion1();
}
 	
void main ()
{
bool local5, local6, local7;
funcion1();
funcion2();
local5=bGlobal;
local6=bSemiGlobal;
local7=local6;
}
	bool bGlobal(false);
	
	void funcion1 (void)
	{
	bool local1, local2;
	local1=bGlobal;
	local2=bSemiGlobal; // NO
	funcion2(); //NO
	}
	
	bool bSemiGlobal(true);
	
	void funcion2 (void)
	{
	bool local3, local4;
	local3=bGlobal;
	local4=local5; // NO
	funcion1();
	}
	
	void main ()
	{
	bool local5, local6, local7;
	funcion1();
	funcion2();
	local5=local1; // NO
	local6=local4; // NO
	local7=bGlobal;
	}

En los ejemplo anteriores ya vemos un poco de código. El orden en c++ es muy importante, y todavía hay cosas que no se han comentado (declaración de funciones). Tan sólo voy a comentar algo sobre la función 'main' (es decir, 'principal'). Esta función está presente en todos los proyectos c++, y es la que utiliza el IDE como eje de todo el programa, ya que una vez finalizada esta función, finaliza el programa.

En algunos motores de juego utilizan otros nombres de **TipoVariable**, por ejemplo al tipo `float` le llaman **Float** , a `int` **Int**. Esto se hace con la orden typedef. Ejemplos:

```
typedef int Entero;
typedef char Cadena;

Entero eNum=5;  // Declaramos a eNum del tipo Entero=int
Cadena cTexto[]="Cadena de Texto";  // Declaramos a Ctexto  del tipo Cadena=char
```

Si quieres declarar una variable como constante utilizaremos const precediendo al **TipoVariable** en la declaración, con lo cual no podrá modificarse posteriormente. Ejemplo:

```
const float PI=3.141516f;
```

En algunos casos se utilizan "macros" para asignar valores a algunos parámetros del juego, como el número de vidas del jugador. Las "macros" se crean al inicio del código de programa con la sentencia #define del siguiente modo:

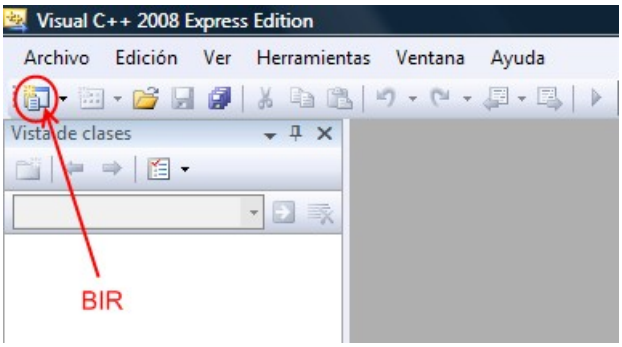
```
#define  NOMBREVARIABLE  Valor_de_Sustitucion;
```

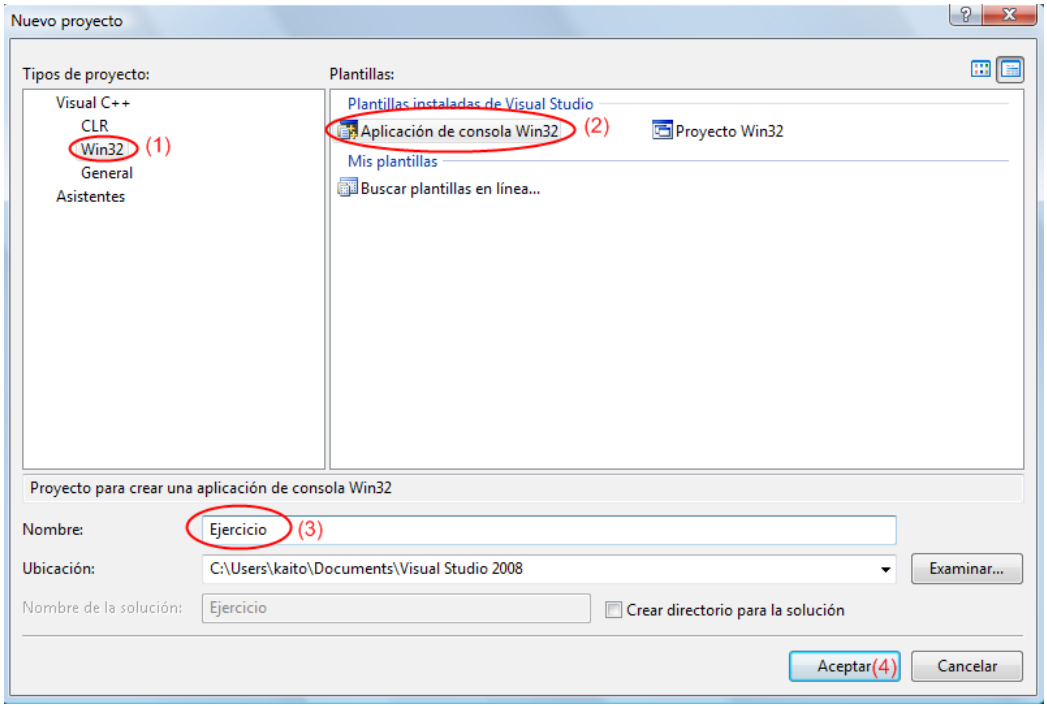
La almohadilla # indica que es una instrucción que se ejecuta antes que las demás durante el precompilado, y lo que hace es sustituir NOMBREVARIABLE por Valor_de_Sustitución. Ejemplo:

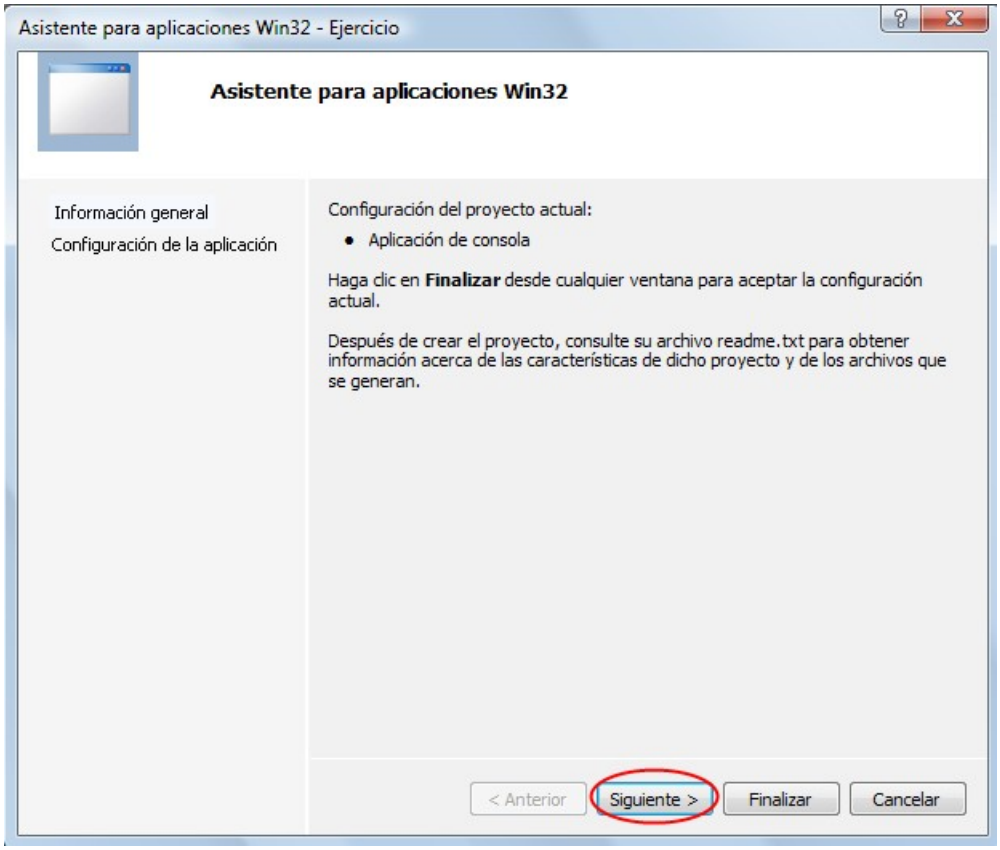
```
#define PI 3.141516F;   // Sustituye la palabra PI por 3.141516F en el resto de código del programa
```

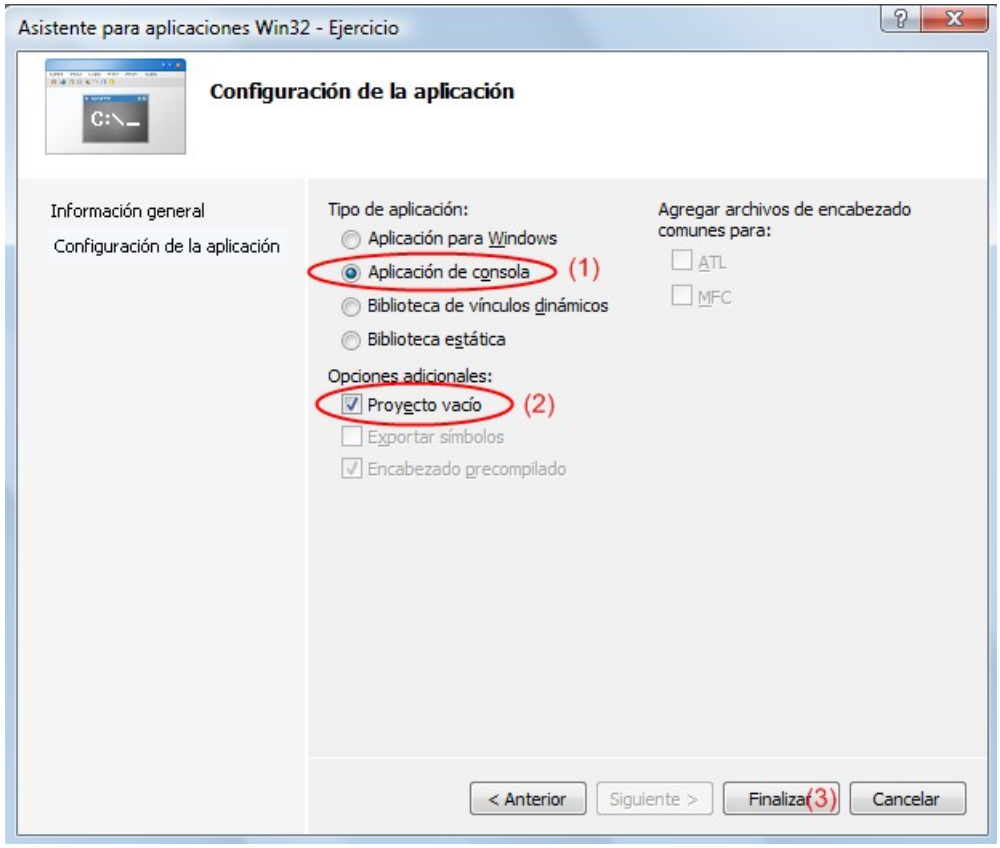
Operadores

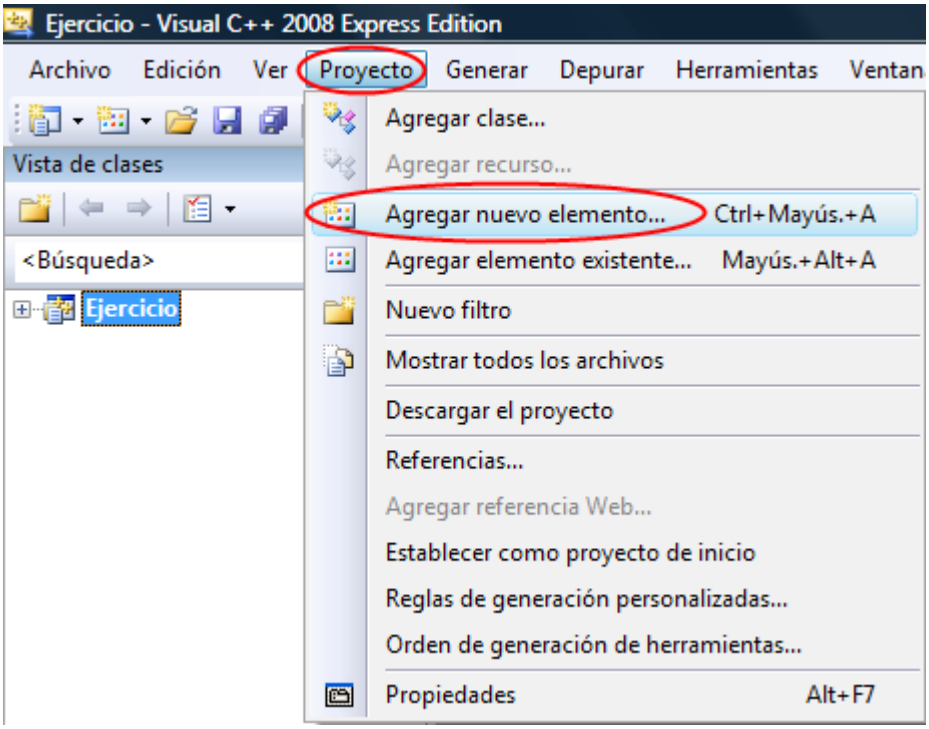
Esto lo tengo que explicar con ejemplos. Primero, creamos un proyecto nuevo en blanco con VC++. Aquí os dejo unas imágenes descriptivas.

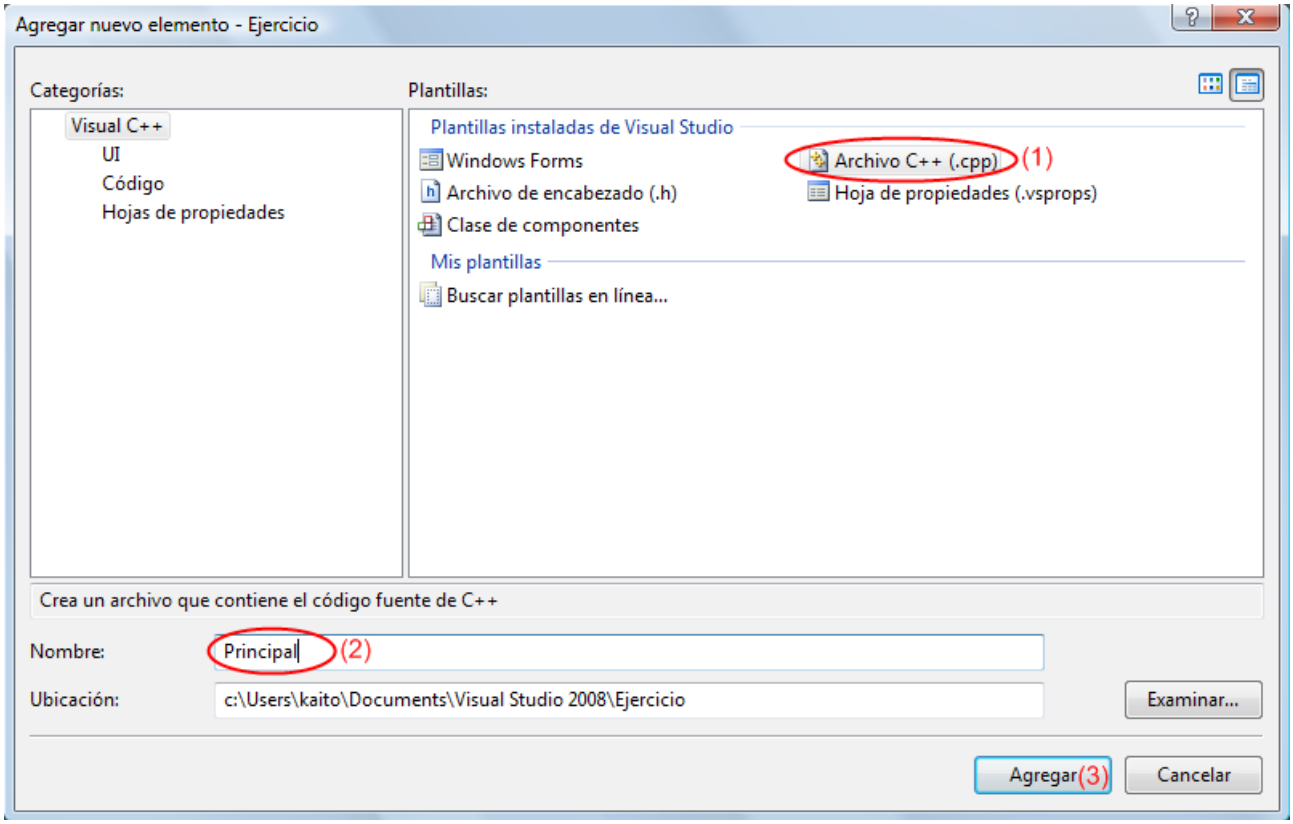





Asistente para aplicaciones Win32 - Ejercicio
Asistente para aplicaciones Win32
Información general
Configuración de la aplicación
Configuración del proyecto actual:
Aplicación de consola
Haga clic en Finalizar desde cualquier ventana para aceptar la configuración actual.
Después de crear el proyecto, consulte su archivo readme.txt para obtener información acerca de las características de dicho proyecto y de los archivos que se generan.
< Anterior
Siguiente >
Finalizar
Cancelar


Asistente para aplicaciones Win32 - Ejercicio
Configuración de la aplicación
Información general
Configuración de la aplicación
Tipo de aplicación:
Aplicación para Windows
Aplicación de consola
(1)
Biblioteca de vínculos dinámicos
Biblioteca estática
Opciones adicionales:
Proyecto vacío
(2)
Exportar símbolos
Encabezado precompilado
Agregar archivos de encabezado comunes para:
ATL
MFC
< Anterior
Siguiente >
Finalizar
(3)
Cancelar





Antes de seguir, recordar que c++ consta de unas pocas palabras clave (en comparación con otros lenguajes), el resto de sentencias o comandos se encuentran en librerías externas. Estas librerías externas constan de un archivo de biblioteca (.lib) que contiene las funciones u objetos precompilados y un archivo de inclusión (.h) que es utilizado en el código de programa donde se van a utilizar esas funciones.

Pero anteriormente tendremos que establecer en el IDE del compilador los directorios donde se encuentran esos archivos (.lib y .h). Esta operación ha sido tratada en otro documento.

Para los ejemplos voy a utilizar las funciones 'cout' y 'cin' que forman parte de la clase 'std' que se encuentra incluida en la librería 'iostream'.

Esta librería se encuentra preinstalada (preparada para ser utilizada) en el IDE y NO debemos realizar ninguna configuración adicional, por esta razón la voy a utilizar en los ejemplos.

Para poder utilizar esa librería en el programa debo incluirla en el código de programación haciendo referencia a su archivo de inclusión (es este caso, 'iostream' sin la extensión .h final) con la sentencia #include (es decir, incluir), en este caso quedaría:

```
#include <iostream>   /* Incluimos las declaraciones de las clases::funciones de la librería iostream para que puedan ser
                         utilizadas posteriormente*/
```

La sentencia #include es una excepción al no llevar el punto y coma (;) al final de línea. Encerrar el nombre del archivo de inclusión entre < > indica al compilador que busque el archivo en los directorios establecidos en el IDE. También podemos encontrar la siguiente forma:

```
#include "iostream"   // Así también incluimos las funciones de la librería iostream
```

Escribiendo el nombre del archivo de inclusión entre comillas “” hacemos que el IDE busque primero el archivo en la carpeta donde se encuentra el archivo de código .cpp, y si no lo encuentra continúa la búsqueda en los directorios del sistema del IDE.

Cuando se utiliza una función que se encuentra dentro de una clase (sería un objeto de esa clase) debemos hacer referencia *nombreclase::nombrefunción (Parámetros);*, donde '::' es un separador que utiliza c++ para diferencia el nombre de la clase (el primero) del nombre de la función (el segundo).

Para no especificar constantemente el nombre de la clase del objeto (función dentro de una clase) se utiliza el comando using namespace *NombreDeLaClase*; en este caso quedaría:

```
using namespace std;
```

Ahora podremos utilizar las funciones 'cout' que escribe datos (números, textos) en pantalla, y 'cin' que lee y asigna los datos introducidos por el teclado a una variable, sin mencionar la clase 'std'.

Al mismo tiempo estas funciones u objetos utilizan unos separadores ('<<' en el caso de 'cout', y '>>' en el caso de 'cin') entre los diferentes datos que manipulan (números, textos y variables).

Y tienen instrucciones propias, por ejemplo 'endl' (del inglés 'end line', es decir, fin de línea) que produce un retorno de carro (bajar a línea siguiente). No voy a mencionar todas estas instrucciones ya que posiblemente **en el motor de juego que utilices no las vas a necesitar**.

C++ también tiene la expresión **“\n”** para realizar un retorno de carro (va entre comillas con o sin texto), y la expresión **“\t”** para tabular.

Los código de ejemplo siguientes debes ejecutarlos desde el IDE **“PULSANDO CTRL+F5”** o menú 'Depurar-->Iniciar sin Depurar”, ya que no se solicita ningún dato al usuario, simplemente muestran una información en pantalla. De esta manera el IDE solicitará que pulses una tecla para continuar. Si pulsas 'F5' o menú 'Depurar-->Iniciar Depuración' sólo verás un pantallazo.

Todavía no se ha comentado como crear una función, pero en los ejemplos aparece la función '**main**', que aunque tiene los mismos requisitos o características que el resto de funciones presentes en un mismo programa, es una función "**especial para C++**", ya que es el cuerpo principal e indispensable de un programa en C++. De todos modos, en los ejemplos siempre aparecerá la misma forma de la función 'main' (void main()), para no liar más las cosas, y a continuación aparece un grupo de llaves { } que contiene el cuerpo o bloque de instrucciones (variables, operadores, llamada a otras funciones,...) de la función (en este caso, main).

Como dije anteriormente las funciones **NO** finalizan con el punto y coma (;). Lo reitero, porque estas puntualizaciones pueden provocar que el código no funcione cuando todo está perfectamente escrito (mayúsculas y minúsculas) y sólo se ha olvidado escribir, o sobra, (;).

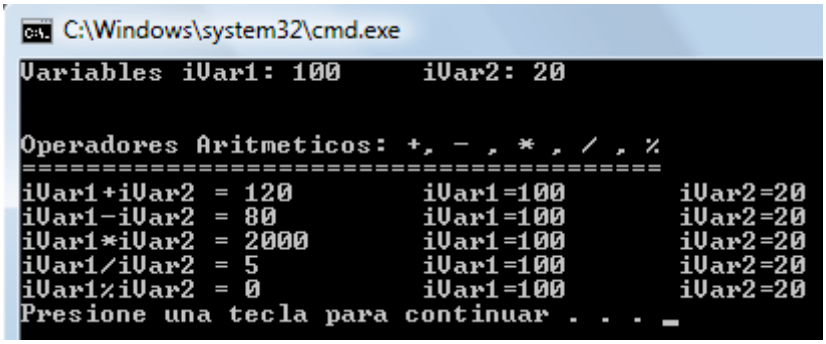
Sabiendo todo esto, seguimos con los operadores.

Existen 4 tipos: **Aritméticos**, de **Asignación**, **Relacionales** y **Lógicos**.

Aritméticos: + , - , * (multiplicación) , / (división) , %(resto de la división). Ejemplo:

```
#include <iostream>;
using namespace std;

void main()
{
int iVar1=100, iVar2(20);  // Variables Locales de la función "principal main"
cout << "Variables iVar1: " << iVar1 << "     iVar2: " << iVar2 << endl;
cout << endl << endl;
cout << "Operadores Aritmeticos: +, - , * , / , %" << endl;
cout << "=======================================" << endl;
cout << "iVar1+iVar2 = " << iVar1+iVar2 << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
cout << "iVar1-iVar2 = " << iVar1-iVar2 << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
cout << "iVar1*iVar2 = " << iVar1*iVar2 << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
cout << "iVar1/iVar2 = " << iVar1/iVar2 << "\t\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
cout << "iVar1%iVar2 = " << iVar1%iVar2 << "\t\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
}
```

Este sería el resultado:

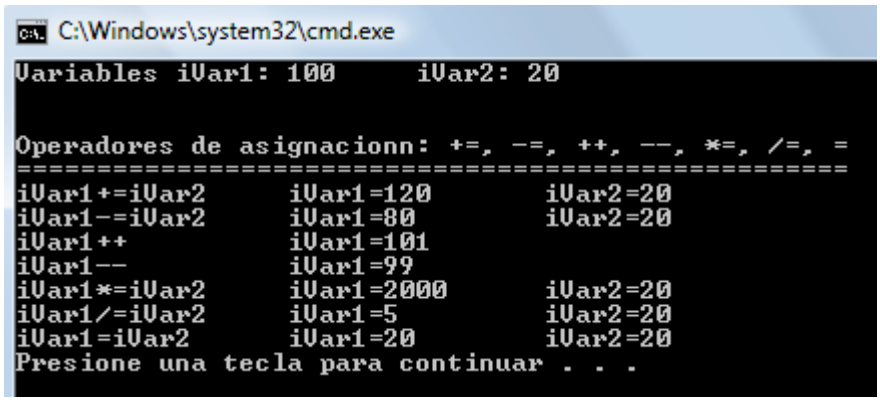


De **Asignación**: =, +=, -=, ++, --, *=, /=

Var1+=Var2 equivale a *Var1=Var1+Var2*
Var1-=Var2 equivale a *Var1=Var1-Var2*
Var1++ equivale a *Var1=Var1+1*
Var1-- equivale a *Var1=Var1-1*
Var1=Var2 equivale a Var1=Var1*Var2* ***(no confundir Var1* con un puntero)***
Var1+=Var2 equivale a Var1=Var1/Var2

```
#include <iostream>
using namespace std;

void main()
{
int iVar1=100, iVar2=20;  // Variables Locales de la función "principal main"
cout << "Variables iVar1: " << iVar1 << "     iVar2: " << iVar2 << endl;
cout << endl << endl;
cout << "Operadores de asignacion: +=, -=, ++, --, *=, /=, =" << endl;
cout << "=================================================" << endl;
iVar1+=iVar2;
cout << "iVar1+=iVar2" << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
iVar1=100; iVar2=20; iVar1-=iVar2;
cout << "iVar1-=iVar2" << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
iVar1=100; iVar2=20; iVar1++;
cout << "iVar1++" << "\t\t iVar1=" << iVar1 << endl;
iVar1=100; iVar2=20; iVar1--;
cout << "iVar1--" << "\t\t iVar1=" << iVar1 << endl;
iVar1=100; iVar2=20; iVar1*=iVar2;
cout << "iVar1*=iVar2" << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
iVar1=100; iVar2=20; iVar1/=iVar2;
cout << "iVar1/=iVar2" << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
iVar1=100; iVar2=20; iVar1=iVar2;
cout << "iVar1=iVar2" << "\t iVar1=" << iVar1 << "\t iVar2=" << iVar2 << endl;
}
```

Este sería el resultado:



Relacionales: == (igual que), < (menor que), > (mayor que), <= (menor o igual que), >= (mayor o igual que), != (distinto a).

Podrás encontrar expresiones con << o >> que NO SON operadores relacionales de valores, sino de dirección de memoria. Es decir, compara la posición en la memora de dos variables (utilizado con *variables puntero*).

Lógicos: && (Y), || (O), ! (NO). Se sitúan entre (operadores &&, ||) o antes(el operador !) de grupos de paréntesis que contienen condiciones creados con los operadores relacionales.

Los operadores relacionales y lógicos son utilizados para expresar condiciones, donde los operadores relacionales se ocupan de comparar dos variables, y los lógicos agrupar dichas comparaciones. Para entender mejor los operadores utilizaré código con algunos comandos de c++ que utilizan estos operadores.

Sentencias

Sentencia condicional if / else if / else (*Si / en otro caso Si / en otro caso*):

if **((** (*Condicion1*) **OL** (*Condicion2*) **OL** ... **)** **OL** **(** (*Condicion1*) **OL** (*Condicion2*) **OL** ... **)** **OL** ...**)** {*Bloque1*;}
else if **(** (*Condicion1*) **OL** ... **)** {*Bloque2*;}
else if (...) {*Bloque...*;} else {*BloqueFinal*;}

OL: *Operador Lógico* (&&,||), el operador lógico ! puede preceder cada grupo de paréntesis para indicar lo contrario (sería NO) de lo que indica la condición.

Condición: Estará formado por expresiones compuestas de operadores *relacionales*.

{*Bloque*;} Aquí se encuentran las órdenes o instrucciones a realizar en caso de que se cumplan las condiciones. Las instrucciones que forman el *Bloque* van separadas por (;) y encerradas entre **llaves** { }.

No es obligatorio utilizar else if ni else, la fórmula general sólo indica la forma y posibilidades de esta sentencia.

Aunque no es muy complicado entender su funcionamiento, ya que'if' significa Si, else if significa 'en otro caso Si' y el último comando else hace que se ejecute el Bloque de instrucciones Final en caso de no cumplirse las condiciones anteriores debemos llevar la precaución de que los grupos de condiciones se encuentren perfectamente encerrados entre paréntesis ( ).

Además fíjate que sólo llevan punto y coma (;) las instrucciones de cada *Bloque*;.

Otra sentencia condicional, (alternativa a if), es switch / case / break / default (*Interruptor / caso / romper / por defecto*):

```
switch (Variable)
{
        case Valor1 : case Valor2: case ... : {Bloque1;} break;
        case ValorA : case ValorB: case ... : {Bloque2;} break;
        case ... : case ... : {Bloque ...;} break;
        default: {BloqueFinal;}
}
```

Esta sentencia utiliza la palabra switch (interruptor), donde utiliza el valor de *Variable* y lo compara con los case (caso) declarados. Se puede incluir el caso default (por defecto), aunque no es obligatorio, para ejecutar un bloque de instrucciones si no se cumplen los casos anteriores.

Se puede asignar varios casos (case) en la misma línea separados por dos puntos (:).

Con la sentencia break (romper) salimos del bloque creado con switch {...}

Vemos que las instrucciones de los distintos *Bloques* y la sentencia break van acompañadas de (;).

Sentencia para crear bucle (algo que se repite continuamente), for (*Para*):

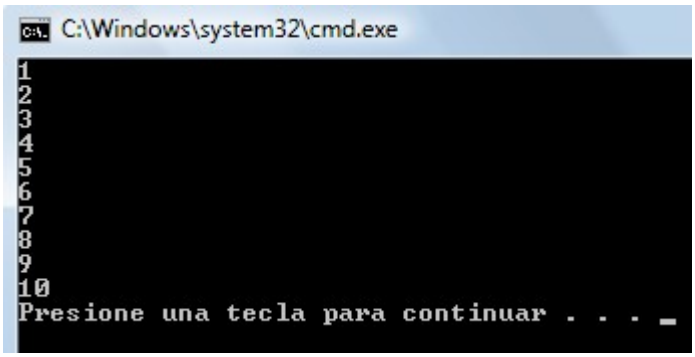
for (*TipoVariable* **NombreVariable**=*ValorInicial*; *ExpresiónDeControl*; *Variación*) {Cuerpo o bloque;}

ExpresiónDeControl: Aquí se utilizan los operadores *relacionales*, para delimitar la duración del bucle.

Variación: Aquí se utiliza los operadores de *asignación* o una fórmula matemática, para incrementar o disminuir el valor de la variable que utiliza for.(la declarada al inicio de la línea con **NombreVariable**).

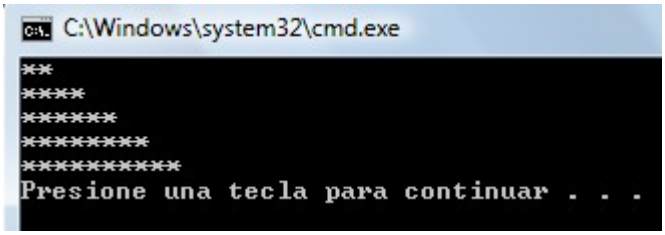
{Cuerpo o bloque;} Son las instrucciones que forman el bucle. Cada instrucción termina con (;). El bloque va encerrado entre llaves. (como en todos los casos comentados anteriormente). Ejemplos:

```
#include <iostream>
using namespace std;

void main()
{
	for (int i=1; i<11; i++)
	{
		cout << i << endl;
	}
}
```



Un ejemplo múltiple:

```
#include <iostream>
using namespace std;

void main()
{
	for (int i=2; i<11; i+=2)
	{
		for (int ii=0; ii<i; ii++)
		{
			cout << "*";
		}
		cout << endl;
	}
}
```



Otra sentencia que crea un bucle (alternativa a for), es while (Mientras):

while (*ExpresiónDeControl*) {*Bloque;*}

ExpresiónDeControl: Aquí se pueden utilizar operadores *relacionales* o *lógicos*, para delimitar la duración del bucle.

while repetirá las instrucciones contenidas en *Bloque* "mientras" se cumpla la condición de la *ExpresiónDeControl*.

Otra sentencia para crear un bucle (alternativa a for y while), se trata de do / while (Hacer / Mientras).

do {*Bloque;*} while (*ExpresiónDeControl*);

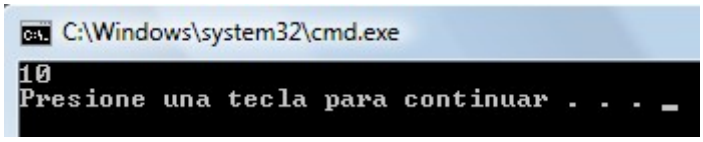
ExpresiónDeControl: Aquí se pueden utilizar operadores *relacionales* o *lógicos*, para delimitar la duración del bucle.

Vemos que se aplica (;) dentro del *Bloque* y tras el paréntesis de la *ExpresióndeControl*. Esta sentencia funciona de forma análoga a la anterior, pero utilizando do/while se habrá ejecutado *Bloque* al menos una vez. (Hace la comprobación de la condición al final, no al inicio).

Anteriormente se ha comentado la utilización de break; (romper) dentro del bloque de la sentencia switch, pero también puede ser utilizada dentro de cualquier bucle creado con for, while, do/while para finalizar su ejecución (salir). Existe otra palabra clave o sentencia de c++ llamada continue (continuar) que finaliza la ejecución del bloque pero sin salir del bucle (la ejecución vuelve al inicio del bucle como si se hubiera terminado de ejecutar todo el bloque).

Otra sentencia no muy conocida es goto (Ir a), que salta a a un lugar del programa señalado con una etiqueta. La etiqueta es un nombre terminado en dos puntos (:). Ejemplo:

```
#include <iostream>
using namespace std;

void main()
{
	int iVar1=10;
	if (iVar1==10) {goto salto1;}
	iVar1=5;
salto1:
	cout << iVar1 << endl;
}
```



Punteros y Referencias

Si has llegado hasta aquí, quisiera comentar que uno de los motivos por los que c++ es más rápido que otros lenguajes, razón por la que es utilizado ampliamente en la creación de videojuegos, es el acceso del programador a aspectos como punteros y referencias, ya que el IDE no se para a pensar en como asignar o colocar un tipo de dato en el programa dejando esa responsabilidad al programador, lo cual genera rechazo a algunos ya que es necesario un mayor conocimiento de programación (comprensible).

Pero si se hace el esfuerzo en comprender ciertas cosas obtendrás a cambio recursos gratuitos (IDE"s, motores de juego), y si entre todos hacemos otro esfuerzo en completar más información en nuestro idioma (para que no esté todo en inglés), quizás programar juegos en c++ no parezca una meta inalcanzable por la mayoría de programadores por “ocio”.

Dicho esto, comenzaré comentando los aspectos de la **variable puntero** y la **referencia,** ya que se suelen utilizar conjuntamente.

Una **variable puntero** *guarda* la dirección a memoria de otra variable que contiene un dato (la llamaré *variable dato*) pero, realmente, *maneja* dos datos (se detallará después):

- Dirección a memoria (**sin ***)
- Valor de dirección a memoria (**con ***)

La **referencia** es la dirección a memoria de una *variable dato*. **NO es una variable**.

Al ser una **variable**, el **puntero** debe ser declarado antes de ser utilizado, **precediendo** al nombre de la variable puntero con un **asterisco ***.

Para ser utilizada una **referencia** tan sólo hay que **preceder** al nombre de la *variable dato* con el signo **&**.

La *variable puntero* debe ser **del mismo tipo** que la *variable que contiene el dato*.

Declaración y asignación de una variable puntero

Existen dos formas de realizar este proceso:

1)**Declarar y asignar a la vez** la *variable puntero:* **Primero** declaramos la *variable dato* (naturalmente, podemos asignarle un valor si queremos) y **después** declaramos la *variable puntero* precedida con * y le asignamos la "*referencia"* de la *variable dato* (obligatorio hacer las dos cosas).

"*Esta forma se deberá aplicar **siempre que** se **asigne** la variable puntero **fuera de una función***". Los ejemplos incorrectos generan errores de compilación:

CORRECTO	INCORRECTO	INCORRECTO

int iVar=100; // Variable dato
float fVar;
 	
int *ipVar=&iVar; // Variable puntero
float *fpVar=&fVar;
float *fp2=fpVar;
 	
void main() { }
	int iVar=100, *ipVar;
	float fVar, *fpVar, *fp2;
	
	ipVar=&iVar;
	fpVar=&fVar;
	fp2=fpVar;
	
	void main() { }
		int iVar=100;
		float fVar;
		
		int *ipVar;
		float *fpVar;
		float *fp2;
		
		ipVar=&iVar;
		fpVar=&fVar;
		fp2=fpVar;
		
		void main() { }

En este caso se cumplirá la siguiente igualdad (algunos casos se verán después en otros ejemplos):

**puntero = otropuntero = &variabledato= variabledatoMATRIZ = &variabledatoMATRIZ[0]*

2) **Declarar y asignar por separado** la *variable puntero*: **Sólo** se podrá realizar esta operación **cuando** la **asignación** se realice **dentro** de una **función**.

CORRECTO	CORRECTO	CORRECTO

int iVar=100, *ipVar;
float fVar, *fpVar,*fp2; // Declaración
 	
void main()
{
ipVar=&iVar; // Asignación
fpVar=&fVar;
fp2=fpVar;
}
	int iVar=100;
	float fVar;
	
	int *ipVar; // Declaración
	float *fpVar,*fp2;
	
	void main()
	{
	ipVar=&iVar; // Asignación
	fpVar=&fVar;
	fp2=fpVar;
	}
		void main()
		{
		int iVar=100;
		float fVar;
		
		int *ipVar;
		float *fpVar,*fp2;
		
		ipVar=&iVar;
		fpVar=&fVar;
		fp2=fpVar;
		}

En este caso se cumpliría la siguiente igualdad (fijarse que el nombre *variable puntero* va sin *):

puntero = otropuntero = &variabledato = variabledatoMATRIZ = &variabledatoMATRIZ[0]

Sobre *variabledatoMATRIZ* se verán ejemplos más adelante. Son *variablesdato* compuestas de varios elementos o valores. Ejemplos:

```
int x[4]={5,2,7,4};        char cTex[]={T,e,x,t,o};        char cTex[]="Texto";
```

Utilización de los punteros

Después de haber sido declarada y asignada la variable puntero de la forma "*correcta*", podremos trabajar con ella de dos formas, con y sin asterisco (*).

La utilización del nombre de la *variable puntero* **sin *** indica dirección a memoria. Se suele utilizar para cambiar la dirección a memoria hacia otra variable dato. Se podría interpretar como una ***variable referencia***, por esta razón al otro lado de la igualdad encontraremos una *referencia* a *variable dato*.

La utilización del nombre de la variable puntero **con *** indica valor de la variable dato. Se suele utilizar para manipular el valor de la variable dato (modificar, mostrar en pantalla, ...) como si de la ***variable dato*** se tratara. Para entender mejor lo comentado voy a poner unas igualdades **después de haber sido declarada inicialmente, y dentro de una función (por ejemplo main):**

**unpuntero = *otropuntero = variabledato = valor*

unpuntero = otropuntero = &variabledato = variabledatoMATRIZ = &variabledatoMATRIZ[0]

Utilizando Variables Puntero y Referencias

```
#include <iostream>
using namespace std;

int iVar1=100, iVar2=20, iResultado;  // Variables dato Globales

int *ip1=&iVar1; // Variable puntero Global a iVar1(con &, para indicar dirección a memoria)
int *ip2=ip1,*ip3=ip1;  // Copiamos ip1(sin *, para indicar dirección a memoria) a otras variable puntero
                        // Todos las variables puntero apuntan a la variable dato iVar1
void main()
{
        // Vemos en pantalla los valores de las variable puntero (con *, para indicar valor de la direccion a memoria)
        cout << "ip1=" << *ip1 << "\tip2=" << *ip2 << "\tip3=" << *ip3 << endl;
        // Cambiamos de dirección o referencia a ip2 y a ip3
        ip2=&iVar2;  ip3=&iResultado; // Variable puntero (sin *, para indicar dirección a memoria)
                                      // = Variable dato  (con &, para indicar dirección a memoria o referencia)
        // Asignamos la suma iVar1+iVar2 a iResultado, utilizando sus variable puntero
        *ip3=*ip1+*ip2;  // Varible puntero (con *, para indicar valor de la dirección a memoria)
        // Vemos en pantalla los valores de las variable dato
        cout << "iVar1:" << iVar1 << "\t+\tiVar2:" << iVar2 << "\t=\tiResultado:" << iResultado << endl;
}
```

Sin utilizar Variables Puntero y Referencias

```
#include <iostream>
using namespace std;

int iVar1=100, iVar2=20, iResultado;  // Variables dato Globales

void main()
{
        // Asignamos la suma iVar1+iVar2 a iResultado
        iResultado=iVar1+iVar2;
        // Mostramos en pantalla el resultado
        cout << "iVar1:" << iVar1 << "\t+\tiVar2:" << iVar2 << "\t=\tiResultado:" << iResultado << endl;
}
```

En ambos códigos obtendríamos el mismo resultado. Claramente el segundo código tiene una sintaxis más clara y menos extensa.

Pero existen situaciones en c++ donde tendremos que utilizar *variables puntero* y *referencias* forzosamente. Esto se verá en las secciones siguientes: *Creación de funciones* y *Memoria dinámica*.

También recordar que las *variables puntero* de *números* y *texto* (tipo char) no muestran la misma información:

**punterotexto*=Letra que ocupa esa posición a memoria
punterotexto=Muestra *toda la cadena* de texto

**punteronúmero*=Valor que ocupa esa posición a memoria
punteronúmero=Muestra dirección a memoria

Creación de funciones

TvRetorno **NombreFuncion** (*TvP1 Parámetro1, TvP2 Parámetro2, TvP... Parámetro...*) {*Bloque;*}

TvRetorno: Tipo variable de retorno. La función devuelve un valor. Si no es así, será void (significa vacío). Es obligatorio completar este dato.

TvP: Tipo variable del Parámetro. Si la función no utiliza parámetros, será void o paréntesis ( ).

Ejemplos válidos de la función main que no retorna ningún valor ni utiliza parámetros:

```
void main(void)
```

```
void main()
```

Luego existen 2 formas de desarrollar una función en el código de programación.

1) *Declarar y construir función al mismo tiempo*: Se debe hacer **antes** de la función principal "**main**". Ejemplo:

CORRECTO	INCORRECTO

void funcion() // Declarar y Construir
{
int i=5;
}
 	
void main()
{
int iVar1=10;
funcion(); // LLAMADA A FUNCION
}
	void main()
	{
	int iVar1=10;
	funcion(); // LLAMADA A FUNCION
	}
	
	void funcion() // Declarar y Construir
	{
	int i=5;
	}

2) *Declarar la función antes de la función "main" y construirla después de la función "main":*

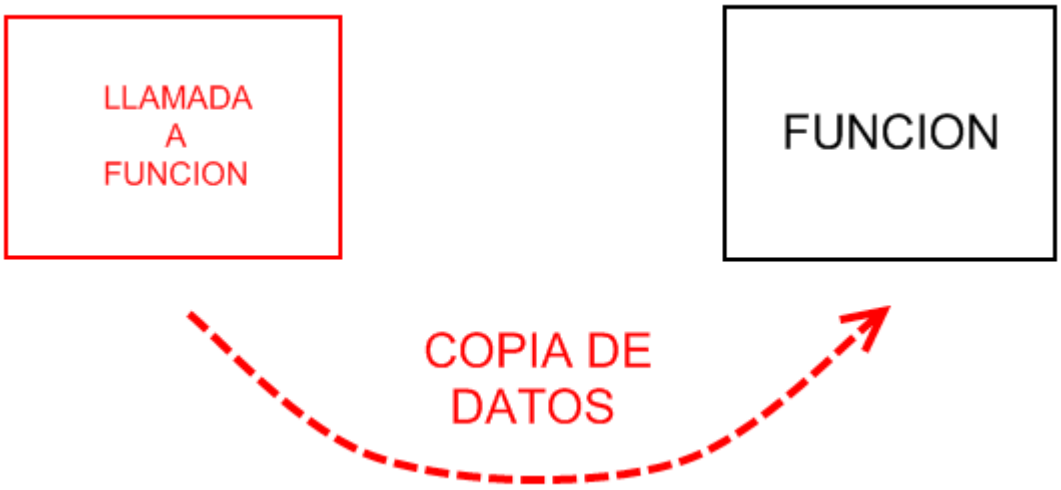
```
void funcion(); // Declaración

void main()
{
        int iVar1=10;
        funcion();  // LLAMADA A FUNCION
}

void funcion()  // Construcción
{
        int i=5;
}
```

Esta es la forma MÁS UTILIZADA, ya que permite declarar las funciones sin especificar las instrucciones que incluye (en el *Bloque* de la función). Normalmente, los *Bloques* de las funciones incluyen llamadas a otras funciones. Ejemplo:

INCORRECTO	CORRECTO

// Declaración y Construcción
void funcion1()
{
int z=0;
funcion2(); // Al llamar a funcion2 que no está
// declarada todavía produce error compilación
}
 	
void funcion2()
{
int w=5;
}
 	
void main(){ }
	void funcion1(); // Declaración
	void funcion2();
	
	void main(){ }
	
	void funcion1() // Construcción
	{
	int z=0;
	funcion2(); // LLAMADA
	}
	
	void funcion2()
	{
	int w=5;
	}

Utilizar funciones

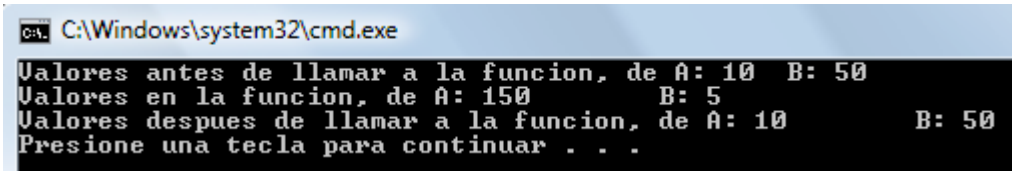


Importante, los datos (variables o valores) que pasemos en la llamada a la función , deben ser del mismo tipo de variable declarados en los parámetros de dicha función. Cuando indico llamada a función indico la instrucción por la cual se ejecuta la función.

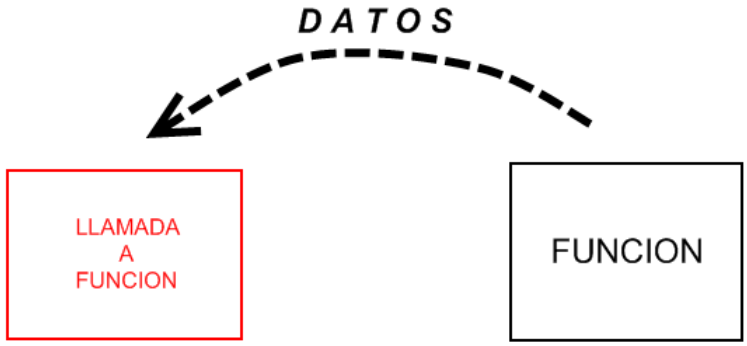
A la hora de utilizar una función es cuando se aprecia la necesidad o no de utilizar *variables puntero* y *referencias*. porque **los parámetros** que utiliza la función **son copias** de los datos que le pasamos en la llamada. Ejemplo:

```
#include <iostream>
using namespace std;

// Declaramos los parámetros A y B como datos del TipoVariable int
void funcion (int A, int B)  // Lo que realmente leemos es void funcion (int copiaA, int copiaB)
{
	A=B+100;
	B=5;
	cout << "Valores en la funcion, de A: " << A << "\tB: " << B << endl;
```

```
}

void main()
{
	int A=10, B=50;
	cout << "Valores antes de llamar a la funcion, de A: " << A << "\tB: " << B << endl;
	funcion (A,B);    // LLAMADA. Los datos de la llamada son del tipo int = tipo parámetro función
	cout << "Valores despues de llamar a la funcion, de A: " << A << "\tB: " << B << endl;
}
```



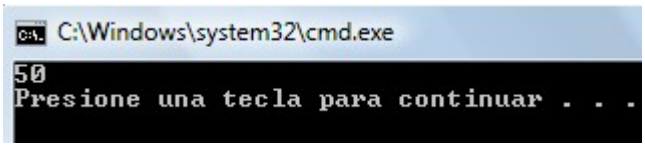
Este tipo de funciones cumplen acciones del juego que no requieren modificar ningún dato que le pasamos en la llamada o retornar algún valor. Por ejemplo establecer la configuración de pantalla.



Lo habitual suele ser que la función devuelva valores. Se puede enviar datos a la "llamada de la función" (desde donde se ejecuta la función) a través del *TvRetorno*, con la sentencia return (retornar), o a través de los parámetros, mediante un puntero (direcciones a memoria de las variables) y una referencia (dirección a memoria de *variable dato*).

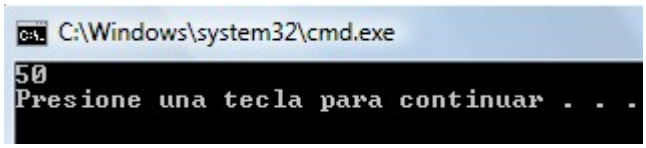
Utilizando return, debemos declarar un *TvRetorno*. Ejemplo:

```
#include <iostream>
using namespace std;

int funcion();  // Declarar funcion

void main()
{
	int local1=0;  // Tipo local1 = Tipo TvRetorno de funcion
	local1=funcion();  // local1 = LLAMADA
	cout << local1 << endl;
}

int funcion()  // Construir funcion
{
	int local2=50;
	return local2;   // La variable local2 es del tipo int = que TvRetorno de la función = que tipo local1
}                        // También sería válido:    return 50;    sin necesidad de crear variable local2
```



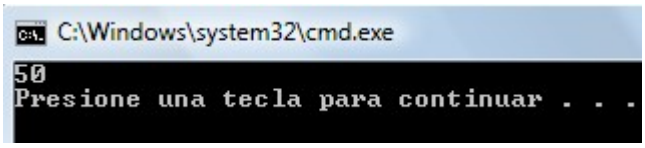
Utilizando una referencia(&) en la llamada y una variable puntero(*) en la función.

```
#include <iostream>
using namespace std;

void funcion(int *out);  // Declarar funcion

void main()
{
	int local1=0;  // Tipo local1 = Tipo parámetro función = int
	funcion(&local1);    // Le pasamos en la llamada una dirección a memoria (referencia a variable dato)
	cout << local1 << endl;
}
// Construir funcion
void funcion(int *out)    // Declaramos parámetro como variable puntero, para que pueda guardar la referencia.
{
	int local2=50;
	*out=local2;    // Variable puntero (en este caso, parámetro) con *, indicar valor de
}                       // esa dirección a memoria (referencia)
```



Utilizando una variable dato en la llamada y una referencia(&) en la función:

```
#include <iostream>
using namespace std;

void funcion(int &out);  // Declarar funcion

void main()
{
	int local1=0;  // Tipo local1 = Tipo parámetro función = int
	funcion(local1); // LLAMADA A FUNCION. Le pasamos una variable dato.
	cout << local1 << endl;
}
// Construir funcion
void funcion(int &out)  // Declarar parámetro como referencia (dirección a memoria) de la variable dato de la llamada
{
	int local2=50;
	out=local2;  // El parámetro asume el papel de la propia variable dato de la llamada. Los
}                    // cambios en el parámetro afectan de la misma forma a la variable dato de la llamada.
```

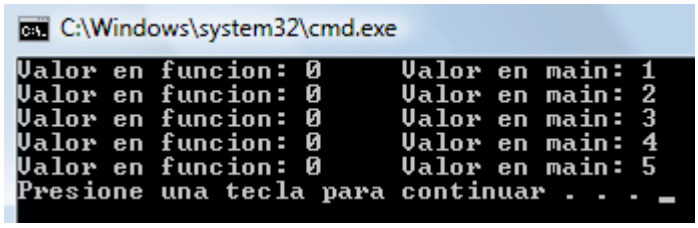


En los casos anteriores **los parámetros de la función siguen siendo copias** de los datos de la llamada, **pero** al **copia**r **direcciones a memoria**, modificamos los valores de las *variables de la llamada*.

Las variables locales de las funciones del programa se reinician (perdiendo los valores obtenidos desde que se llamó a la función por última vez) cada vez que se ejecutan o llaman. Ejemplo:

```
#include <iostream>
using namespace std;

int funcion(int x)
{
	int local=0;  // Es necesario iniciar la variable con (=0)
	cout << "Valor en funcion: " << local;  // Para que no salga error aquí
	local+=x;
	return local;
}
void main()
{
	for (int i=1; i<6; i++)
	{
		cout << "\tValor en main: " << funcion (i) << endl;
	}
}
```

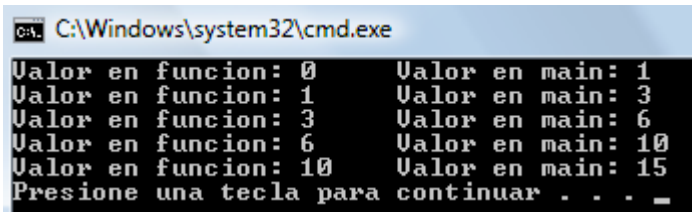
Obtendríamos este resultado:



Si queremos que la variable 'local' no se reinicie cada vez que se llama a la función le aplicamos la palabra clave static (estático) delante del tipo de variable. Ejemplo anterior:

```
#include <iostream>
using namespace std;

int funcion(int x)
{
	static int local=0;  // Es necesario iniciar la variable con (=0)
	cout << "Valor en funcion: " << local;  // Para que no salga error aquí
	local+=x;
	return local;
}
void main()
{
	for (int i=1; i<6; i++)
	{
		cout << "\tValor en main: " << funcion (i) << endl;
	}
}
```

Resultado:



Funciones y cadenas de texto

He dejado las variables de texto char para comentar al mismo tiempo las características de las funciones que manipulan cadenas de texto. Las cadenas de texto utilizan una matriz [ ] y no se pueden copiar las unas a las otras directamente. Ejemplos:

INCORRECTO		
// 15 caracteres + 1 para '/0' = 16 char c1[16]="Cadena de texto"; char c2[]=c1;	char c1[16]="Cadena de texto"; char c2[]=c1[16];	char c1[16]="Cadena de texto"; char c2[16]=c1[16];

La forma de copiar una *variable de texto* (tipo char) a otra, sería ir seleccionando uno a uno cada elemento de la matriz y copiarlo a otra matriz. Deberíamos crear una función llamada 'copiar' para manejar matrices de texto "completas", lo cual genera un problema ya que el parámetro que pasemos a la función no puede ser una *variable dato* matriz "completa".

La **solución** a este problema es utilizar una **variable puntero** que contenga la dirección a memoria del primer elemento de la matriz de la *variable texto* (es decir, *variable dato* matriz tipo char). Ejemplo:

CORRECTO

```
#include <iostream>
using namespace std;

// Declaración de parámetros como punteros
void copiar (char *a, char *b)
{
// Finalizar bucle cuando encuentre el último carácter '\0'
while (*a!='\0'){
// Var Puntero (con *). Asignar el valor de la dirección *a.
*b=*a;
// Var Puntero (sin *) direccion de memoria. Se desplaza a la
// siguiente posición memoria (hacia la derecha)
a++; b++;  //  a++ equivale a=a+1
}
// Var Puntero (con *). Asignar valor dirección de *a
// final de cadena '\0' antes de finalizar la función.
*b=*a;
}

void main()
{
char c1[]="Texto";
char copia[6]="";
char *p1=c1, *p2=copia;

cout << "Valor de Copia antes: " << copia << endl;
copiar (p1,p2);  // Var puntero (sin *) direcciones a memoria
cout << "Valor de Copia después: " << copia << endl;
}
```

CORRECTO

```
#include <iostream>
using namespace std;

char c1[]="Texto";
char copia[6]="";
char *p1=c1, *p2=copia;

void copiar (char *a, char *b)
{
        while (*a!='\0'){
        *b=*a;
        a++; b++;
        }
        *b=*a;
}

void main()
{
cout << "Valor de Copia antes: " << copia << endl;
copiar (p1,p2);
cout << "Valor de Copia después: " << copia << endl;
}
```

En el código de la izquierda la función manipula variables locales, y en el de la derecha globales. Al manejar direcciones a memoria la función puede manejar ambos tipos.

Los dos códigos producen los mismos resultados y son correctos.

También se podría realizar el mismo código utilizando *referencias*. Pero en este caso tenemos que pasarle la dirección del primer elemento de la matriz [0], o desde la posición que queramos copiar. Además voy a incluir la sentencia sizeof que puede ser utilizada para saber la dimensión de matrices de *variables dato* (tanto número como texto) para evitar copiar sobre matrices más pequeñas que la de origen.

```
#include <iostream>
using namespace std;


void copiar (char &r, char &rr)      // La función coge la dirección a memoria (referencia)
{                                    // de los datos introducidos en la llamada.(en main)
        char *a=&r, *b=&rr;          // Guardamos esas direcciones en variables puntero
        while (*a!='\0'){            // Para poder guardar valores (con *) y poder
        *b=*a;                       // movernos por la matriz (sin *)
        a++; b++;
        }
        *b=*a;
}

void main()
{
        char c1[]="Texto";
        char copia[6]="";
        cout << "Valor de Copia antes: " << copia << endl;
        if ((sizeof (c1)) <= (sizeof (copia))) {copiar (c1[0],copia[0]);} // Le paso los primeros elementos de la matriz
                                                                          // en la llamada
        cout << "Valor de Copia después: " << copia << endl;
}
```

Si en la función main cambias las línea char copia[6]=""; por char copia[5]=""; no se realizará la copia. Además podrías añadirle a la sentencia if un else para que mostrara un mensaje en pantalla comunicando que no se puede realizar la acción.

He aplicado la sentencia o palabra clave de c++ sizeof en la función main (donde se encuentran o están visibles las *variables dato* c1 y copia), y no en la función copiar que opera con direcciones a memoria. Con las matrices de número se emplea otra fórmula que se verá después.

Al utilizar referencias (&) en la función no tenemos necesidad de crear *variables puntero* para todas las matrices o *variables de texto*, bastará con las dos *variables puntero locales* de la función copiar.

No crean que todos los código los he copiado de un libro. Hay que practicar y compilar, hasta que no salga ningún error de compilación para entender la relación entre variables dato, matrices, variables puntero y referencias.

En este último ejemplo, he declarado el tipo variable de los parámetros de la función como direcciones a memoria (&) aunque los datos pasados en la llamada son *variables dato*.

```
void copiar (char &r, char &rr)
```

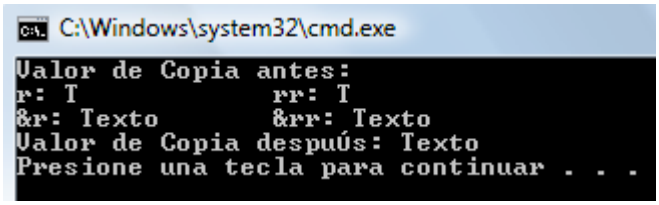
Posteriormente, cuando he declarado las *variable puntero locales* de la función he vuelto a añadir (&) a los *parámetros* aunque estaban ya declarados como *referencia* en el encabezamiento de la función.

```
char *a=&r, *b=&rr;
```

Si añadimos al final de la función copiar la línea de instrucciones:

```
cout << "r: " <<r << "\t\trr: " << rr << endl << "&r: " <<&r << "\t&rr: " << &rr << endl;
```

Podemos ver que el valor del *parámetro* (que es una *referencia*) es el primer carácter de la cadena y, en cambio, *&parámetro* corresponde a toda la *cadena de texto*.



Además, *&parámetro* es el único valor que acepta la declaración de la *variable puntero local* de la función copiar. Si cambias la línea de instrucciones de declaración de las *variable puntero* por char *a=r, *b=rr; saldrá un mensaje de error de compilación como este:

```
error C2440: 'inicializando' : no se puede realizar la conversión de 'char' a 'char *'
```

'char *' trata sobre la *variable puntero* , entonces ¿'char' que es el tipo de variable para texto es el *&parámetro* declarado en el encabezamiento de la función? Sí.

Anteriormente he utilizado otro ejemplo donde utilizaba *referencias* (&) en los parámetros de la función, y modificaba los **valores** de los datos pasados en la llamada desde la misma función como si fueran datos locales de la propia función.

Ahora necesito utilizar la direcciones a memoria de los *datos de la llamada* ya que tengo que manejar matrices. Para esto, vuelvo a hacer referencia con (&) sobre los parámetros de la función aunque estaban ya declarados como referencia en el encabezamiento de la función, porque manejan *valores* y necesito *direcciones a memoria*.

He realizado una tabla para simplificar un poco:

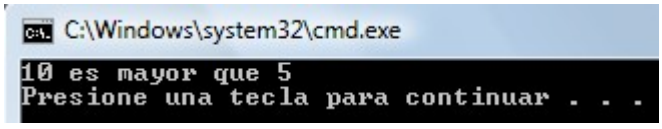
LLAMADA A FUNCIÓN	**PARÁMETRO FUNCIÓN**	**DENTRO FUNCIÓN**
variabledato (NO MATRIZ)	*&Parámetro*	*Parámetro*
&variabledato (NO MATRIZ)	**Parámetro*	**Parámetro* (con *, valor dir. mem.)
puntero (a *variable dato*) (sin *, dirección a memoria)	**Parámetro*	**Parámetro* (valor)
variabledato[0] (ES MATRIZ)	*&Parámetro*	**puntero* = *&Parámetro* **puntero* (valor) *puntero* (dirección)
puntero (a *variable dato* MATRIZ) (sin *, dirección a memoria)	**Parámetro*	**Parámetro* (valor) *Parámetro* (dirección)

Recordar que la tabla superior refleja los casos en los que se tiene que modificar la variable dato incluida en la llamada, o se envía una matriz en la llamada. No siempre será necesario, y se puede utilizar la instrucción return para que la función devuelva un valor.

Algunos ejemplos

Voy a desarrollar unos ejercicios más antes de pasar al siguiente tema. Un ejemplo donde se utilizan *variables puntero* de *variables dato* que no son matrices. La función mayor devuelve true(verdad) si el primer valor es mayor que el segundo valor o false si no es así utilizando return, y los datos de la llamada no son modificados, sólo consultados. Por lo tanto, aunque utilice una *variable puntero* en la llamada, me limito a pasarle el valor (con *) y no la dirección.

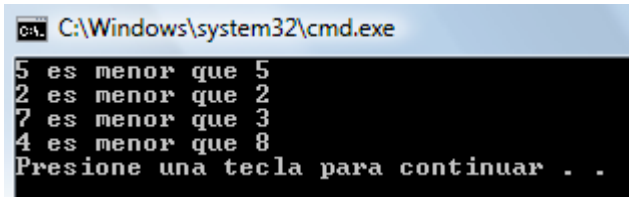
```
#include <iostream>
using namespace std;

bool mayor (int x, int y)
{
	if (x>y) {return true;} else {return false;}
}

void main()
{
	int x=10, y=5;
	int *px=&x, *py=&y;
	if (mayor(*px,*py))
	{ cout << *px << " es mayor que " << *py << endl;}
}
```



Se podría pensar que no tiene sentido hacer el código así, ya que se podría realizar perfectamente sin utilizar *variables puntero*, pero eso es porque *"todavía"* no hemos visto la *memoria dinámica*.

Si generamos el mismo código para unas *variables dato* matriz, sería:

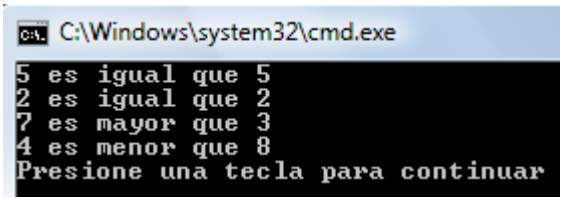
```
#include <iostream>
using namespace std;

void comparar (int *x, int *y, int z)
{
	for (int i=0; i<z; i++)
	{
		if (*x>>*y) {cout << *x << " es mayor que " << *y << endl;}
		else if (*x<<*y) {cout << *x << " es menor que " << *y << endl;}
		else {cout << *x << " es igual que " << *y << endl;}
		x++; y++; // (Sin *, direccion a memoria) Incremento +1
	}			// Desplazamiento hacia la derecha
}

void main()
{
	int x[]={5,2,7,4};
	int y[]={5,2,3,8};
	int *px, *py;
	px=x; py=y;
	comparar(px,py,(sizeof(x)/sizeof(x[0])));
}
```



Observando la imagen de resultados, el código funciona pero no como nosotros queremos (la computadora no se ha equivocado). El fallo se encuentra en la forma que he utilizado los operadores relacionales en la sentencia if. Voy a sustituir >> por >, y << por <. Esa parte del código quedaría:

```
if (*x>*y) {cout << *x << " es mayor que " << *y << endl;}
else if (*x<*y) {cout << *x << " es menor que " << *y << endl;}
else {cout << *x << " es igual que " << *y << endl;}
```

Obtendríamos:



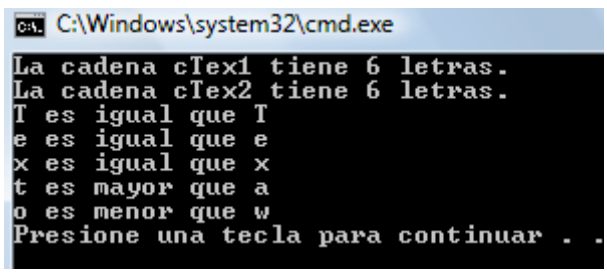
Al utilizar << y >> entre las *variable puntero*, comparaba las posiciones de memoria, y como la *variable dato* **x** fue declarada antes que **y** ocupaba unas posiciones anteriores y daba como resultado que los valores de ***x** eran menor a las direcciones de ***y**.

Sólo queda por mencionar el último dato de la llamada (sizeof(x)/sizeof(x[0])) que da como resultado el número de elementos de la matriz numérica **x**. A continuación veremos que las matrices de texto no se calculan igual. También vemos que he hecho constar ese dato en la función comparar como int z simplemente, pues se trata de un dato de consulta.

Si generamos el código para una *variable de texto* matriz, obtendremos:

```
#include <iostream>
using namespace std;

void comparar (char *x, char *y)  // Declaro los parámetros como variables puntero
{
	while (!(*x==0))  // Hasta llegar al final de la cadena (valor 0)
	{			// Sería: Mientras NO(valor puntero x = 0)
		if (*x>*y) {cout << *x << " es mayor que " << *y << endl;}
		else if (*x<*y) {cout << *x << " es menor que " << *y << endl;}
		else {cout << *x << " es igual que " << *y << endl;}
		x++; y++; // (Sin *, direccion a memoria) Incremento +1
	}			// Desplazamiento hacia la derecha
}

void main()
{
	char cTex1[]="Texto"; // Declarar y asignar variables de texto
	char cTex2[]="Texaw";
	char *px, *py;			// Declarar punteros tipo char
	cout << "La cadena cTex1 tiene " << sizeof (cTex1) << " letras." << endl;
	cout << "La cadena cTex2 tiene " << sizeof (cTex2) << " letras." << endl;
	px=cTex1; py=cTex2;   // Asignar punteros a las variables de texto
	comparar(px,py);// Llamada a funcion comparar, utilizo las direcciones a memoria.
}
```

Obtendríamos:



Ya había utilizado la sentencia sizeof con cadenas de texto anteriormente en otro ejemplo. Para saber las dimensiones de la matriz de texto no es necesario hacer la división por el primer elemento de la matriz, pero vemos que da como resultado 6 letras, cuando son 5 realmente (le suma el final de cadena '\0'). Si necesitas la dimensión real tendrás que restarle uno, pero si lo utilizas para comparar cadenas no es necesario restar nada ya que a todas les añade la misma cantidad.

Con las matrices de números sizeof calcula la dimensión exacta, pero aplicando la división realizada en el ejemplo anterior.

Existe otra forma de crear variables de texto en c++, pero deben ser utilizadas como CONSTANTES de texto. Esa forma consiste en asignar la cadena de texto a una *variable puntero* directamente (posteriormente utilizaré la expresión variable PUNTERO de texto).

```
char *cpFrase="Frase del Ejercicio";
```

o en forma de matriz: `char *cpFrase[]={"Frase del Ejercicio","Segunda Frase","Tercera Frase"};`

Si intentamos asignar nuevos valores a este tipo de variables, afectará a la estabilidad del programa.

En las variables DATO de texto podemos configurar la matriz a un tamaño superior a la cadena de texto asignada, por ejemplo `char cTex[100]="Texto";` en ese casos tenemos margen para asignar a la variable de texto cadenas mayores, pero utilizando `char *cTex="Texto";` NO.

Más diferencias. Al declarar y asignar *variables puntero* a matriz de cadenas de texto:

```
char *cpFrase[3]={"Frase del Ejercicio","Segunda Frase","Tercera Frase"}; // Utilizamos un índice menos
```

y después utilizar: `cpFrase[0]; cpFrase[1]; cpFrase[2];`

Declarar y asignar *variables dato* char;

```
char cfrase[3][50]={"Frase del Ejercicio","Segunda Frase","Tercera Frase"}; // Debemos constatar extensión de la
                                                                            // cadena en el subíndice más a la derecha
```

y después utilizar: `cpFrase[0]; cpFrase[1]; cpFrase[2];`

Utilización de sizeof en ambos casos: `(sizeof(cpFrase)/sizeof(cpFrase[0]))`

La instrucción sizeof aplicada de la forma indicada más arriba, devuelve el número de cadenas de texto de la matriz (en este caso 3) pero no la dimensión de la cadena. Tendríamos que utilizar una instrucción while por ejemplo y crear una variable en su interior que cuente los bucles creados hasta encontrar el final de cadena '\0'. (o utilizando *strlen* de la librería <string>, se verá más adelante).

MATRICES

En la creación de matrices de datos en c++ se detallan las dimensiones de la matriz entre corchetes [ ] a continuación del nombre de la matriz. Forma general:

TipoVariable **NombreVariable** [*índice*][*subíndice1*][*subíndice2*][*subíndice...*];

Las matrices pueden ser unidimensionales, bidimensionales, tridimensionales, etc... dependiendo del número de índices que conste.

La matriz básica es la ***unidimensional*** (de un sólo índice). A partir de esta se genera el resto. Los elementos(e) de la matriz unidimensional van separados por comas(,) y entre llaves {}. Forma de la matriz unidimensional:

TipoVariable **NombreVariable** [*índice*] = {e1,e2,e3,e4,e5,e6,e7,e8,...,e(índice)};

En la *declaración* el valor del índice *indica* el *número de elemento*s. Es decir:

número de elementos de matriz=índice

Después de haber sido declarada la variable matriz, *durante* la *asignación o lectura* el valor del índice de la matriz unidimensional *indica* la *posición del elemento*. Pero las posiciones en la matriz se empiezan a contar desde cero, por esta razón al primer elemento de la matriz le corresponde el índice cero, es decir, **NombreVariable[0]=e1** y el último elemento de la matriz será índice-1, es decir, **NombreVariable[indice-1]=e(índice)**.

A continuación, encontraríamos las matrices ***bidimensional***es. Forma general:

TipoVariable **NombreVariable** [*índice*][*subíndice*]= **{** {e1,e2,...,e(subíndice)}, *fila1*
{e1,e2,...,e(subíndice)}, *fila2*
{ .... }, *fila...*
{e1,e2,...,e(subíndice)} **};** *fila(índice)*

En este caso, durante la declaración el subíndice indica el número de elementos de la matriz unidimensional y él indice el número de filas (o veces que se repite la matriz unidimensional). En la asignación o lectura de valores debemos volver a tener en cuenta que el número de orden inicial es el cero por esta razón. Así quedaría el primer y último elemento de la matriz:

NombreVariable [0][0] = {fila1,e1}
NombreVariable [índice-1][subíndice-1] = {fila(índice),e(subíndice)}

La siguiente matriz sería la ***tridimensional***. Forma general:

TipoVariable **NombreVariable** [*índice*][*subíndice1*][*subíndice2*] = **{**

matriz1	{e1,e2,...,e(subíndice2)},	{ .... },	{e1,e2,...,e(subíndice2)},
matriz2	{e1,e2,...,e(subíndice2)},	{ .... },	{e1,e2,...,e(subíndice2)},
matriz...	{ .... },	{ .... },	{ .... },
matriz(índice)	{e1,e2,...,e(subíndice2)},	{ .... },	{e1,e2,..,e(subíndice2)}**};**
	fila1	*fila....*	*fila(subíndice1)*

Así quedaría el primer y último elemento de la matriz:

NombreVariable [0][0][0] = {matriz1,fila1,e1}
NombreVariable [*índice-1*][*subíndice1-1*][*subíndice2-1*] = {*matriz(índice)*, *fila(subíndice1)*,e(subíndice2)}

Forma general de una matriz de *cuatro dimensiones*:

TipVar NombreVariable [índice][subíndice1][subíndice2][subíndice3] = **{**

Grupo1

matriz1	{e1,e2,...,e(subíndice3)},	{ .... },	{e1,e2,...,e(subíndice3)},
matriz2	{e1,e2,...,e(subíndice3)},	{ .... },	{e1,e2,...,e(subíndice3)},
matriz...	{ .... },	{ .... },	{ .... },
matriz(subíndice)	{e1,e2,...,e(subíndice3)},	{ .... },	{e1,e2,...,e(subíndice3)},
	fila1	*fila....*	*fila(subíndice2)*

Grupo...

.....	.....	.....	.....

Grupo(índice)

matriz1	{e1,e2,...,e(subíndice3)},	{ .... },	{e1,e2,...,e(subíndice3)},
matriz2	{e1,e2,...,e(subíndice3)},	{ .... },	{e1,e2,...,e(subíndice3)},
matriz...	{ .... },	{ .... },	{ .... },
matriz(subíndice1)	{e1,e2,...,e(subíndice3)},	{ .... },	{e1,e2,..,e(subíndice3)}**};**
	fila1	*fila....*	*filasubíndice2*

Así quedaría el primer y último elemento de la matriz:

NombreVariable [0][0][0][0] = {Grupo1,matriz1,fila1,e1}
NombreVariable [*índice-1*][*subínd1-1*][*subínd2-1*][*subínd3-1*] = {*Grupo(índice)*,*matriz(subínd1)*, *fila(subínd2)*,e(subínd3)}

En el desarrollo de muchos juegos se suele crear una matriz bidimensional para delimitar las zonas transitables del mapa. Dicha matriz podría ser del tipo bool donde asignaríamos valores true a las zonas ocupadas y false a las zonas libres. Si queremos delimitar zonas especiales como accesos o puertas, terreno especiales (para reducir velocidad, vida,... del personaje) deberemos utilizar otro tipo de variable en la matriz, por ejemplo short, donde asignaríamos 0(zonas libres), 1(zonas ocupadas), 2(puertas), 3(fango), 4(agua), ...

Para crear una aventura gráfica se podría crear una matriz de 4 dimensiones para guardar los textos o mensajes del juego donde asignaríamos, por ejemplo, al índice el número de incidencia(objetos, acciones,situaciones), al subíndice1 sería el grupo o diferentes mensajes disponibles para esa incidencia, y finalmente subíndice2 serían los mensajes

Podremos asignar los valores a la matriz de dos formas:

1) *Durante la declaración*. Es la forma general que he citado anteriormente en cada tipo de matriz. Si desconocemos los valores a asignar durante la declaración, debemos iniciarla asignándole el valor {0} a las matrices numéricas "", y a las de texto (tipo char) {""} o {'\0'}. Ejemplo:

```
int iVar[5][5]={0};
char cVar[3][50]={""};
```

2)*Después de la declaración*. Esta opción es mucho más complicada puesto que debemos seleccionar uno a uno cada elemento de la matriz (creando un bucle) para asignarle el valor.

EJEMPLOS

```
#include <iostream>
#include <string>

using namespace std;

int iIniciada[3] = {0};   // Iniciar todos los elementos a cero
int iNoIniciada[3];                    // Declarar sin asignar valores. Se PUEDE hacer pero no es conveniente.

int iVar1[3] = {1,2,3};  // Matriz de 3 elementos
int iVar2[3][3] = { 1,2,3,   1,2,3,   1,2,3};  // Matriz de 3x3 elementos, total 9 elementos
int iVar3[3][3][3] = {1,2,3, 1,2,3, 1,2,3,   1,2,3, 1,2,3, 1,2,3,   1,2,3, 1,2,3, 1,2,3};

char cc[3][3] = {'\0'}; // Iniciar todos los elementos a cero
char ccc[3][3] = {""};
char cVar1[3] = {'1','2','\0'};  // Matriz de texto de 2 (3-1) caracteres
//char cVar1[3] = {'1','2','3'}; Se PUEDE pero NO SE DEBE hacer, porque hay que indicar fin de cadena '\0'
char cVar2[3] = "12";
//char cVar2[3] = "123"; DA ERROR

// El último subíndice tendrá el valor de la cadena más larga +1. En este caso "Tres" (4 caracteres+1=5)
char cVar3[3][5]   = {"Uno","Dos","Tres"};
// RECORDAR: Variable PUNTERO de TEXTO debe ser considerada como CONSTANTE
char *cpVar1 [3] = {"Uno","Dos","Tres"};   // LA variable PUNTERO de TEXTO NECESITA UN INDICE MENOS.
                                           // No necesitamos considerar la extensión de la cadena de texto
char cVar4 [3][3][5] =  {"Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres"};
char *cpVar2 [3][3]  =  {"Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres"};
char cVar5[3][3][3][5] = { "Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres",
                           "Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres",
                           "Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres"  };
char *cpVar3[3][3][3]  = { "Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres",
                           "Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres",
                           "Uno","Dos","Tres", "Uno","Dos","Tres", "Uno","Dos","Tres"  };

void main ()
{
// Asignar, dentro de funcion, valores de la matriz numerica. Si ya hemos declarado la variable
// debemos ir seleccionando uno a uno cada elemento de la matriz y asignarle un valor (utilizaré un bucle)
// La expresion de control del bucle será (i<subíndice) para que valor máximo del bucle sea (subíndice-1)
for (int i=0; i<3; i++) { iVar1[i]=(i+5); }
for (int i=0; i<3; i++) { for (int ii=0; ii<3; ii++) { iVar2[i][ii]=(ii+5); } }

// Variables DATO NUMERICAS. Especificamos TODOS los subíndice para escribir/asignar datos
// Primer y último elemento de la matriz.
cout << iVar2[0][0] << "\t" << iVar2[2][2] << endl;

// Variables DATO de TEXTO. A continuación, primer y último elemento de la matriz.
// PARA ESCRIBIR/ASIGNAR SE DESPRECIA EL ULTIMO SUBINDICE EN LAS VARIABLES DATO TEXTO.
cout << cVar1 << endl;  // En la matriz unidimensional sólo detallamos el nombre de la variable
cout << cVar4[0][0] << "\t" << cVar4 [2][2] << endl;  // Primer y último elemento de la matriz
cout << cVar5[0][0][0] << "\t" << cVar5 [2][2][2] << endl;

// Primer y último elemento de las variables PUNTERO de TEXTO (SIN *). Especificar TODOS los subíndices.
// Estas variables deben ser consideradas CONSTANTES y no podremos asignar nuevos valores.
cout << cpVar1[0] << "\t" << cpVar1 [2] << endl;
cout << cpVar3[0][0][0] << "\t" << cpVar3 [2][2][2] << endl;

//**********SOLO SE ESCRIBE UNA LETRA DE LA CADENA**********
cout << cVar1[0] << endl;   // AL AÑADIR ULTIMO SUBINDICE SOLO ESCRIBE UNA LETRA
cout << cVar5 [2][2][2][4] << endl;
```

```
// cout << *cpVar1[0]; (CON *) GENERA ERROR "el subíndice requiere una matriz o tipo de puntero"
// Para escribir una letra de una variable PUNTERO de TEXTO se utiliza *(puntero+posición).
cout << (*(cpVar1)) << endl;


// La Asignación de cadenas de texto(matrices de texto) se verá cuando lleguemos a la librería <string>
}
```

ANOTACIÓN: En matrices de muchos elementos poner **sólo** el **grupo de llaves** que **engloba todos** los **elementos** de la **matriz (como muestra este último ejemplo)** ya que, por lo menos a mí, surgen errores de compilación indicando “hay demasiados inicializadores” no siendo así. . Aunque al inicio he formulado las formas generales encerrando entre llaves cada matriz unidimensional que formaban las matrices bidimensiones tridimensionales,... En lugar de utilizar grupo de llaves para señalar cada grupo de datos, utiliza espacios y saltos de línea.

Todo lo comentado anteriormente está relacionado con variables de dimensiones fijas que ocupan un espacio en memoria inalterable hasta finalizar el programa, donde podemos modificar su contenido pero no sus dimensiones. Otra característica es que se crean al iniciar el programa.

Pero existe otro tipo de *memoria* llamada *dinámica*, que se crea o modifica sus dimensiones durante la ejecución del programa ya que no se sabe la cantidad de variables ni su dimensión exacta. Por ejemplo al cargar una partida guardada de un juego se generan nuevas variables con los datos alojados en un archivo.

Memoria Dinámica

ANOTACIÓN: No confundir las variables puntero de texto vistas anteriormente con las variables puntero, de memoria dinámica, que vamos a ver ahora tanto numérica como de texto.
Las variables que utilizan memoria dinámica son declaradas junto a la palabra clave new (es decir, nuevo). El comando new devuelve una *dirección a memoria*, por esta razón al otro lado de la igualdad aparece una *variable puntero*. Existen dos formas de declarar y asignar *memoria dinámica* (**se asigna espacio de memoria pero no valores**):

1) *Declarar y asignar a la vez:*

TipoVariable **NombreVariablePuntero* = new **TipoVariable;**

En forma de matriz (las matrices dinámicas son complicadas de definir y eliminar, pero no de asignar valores) Pondré un ejemplo de matriz bidimensional y tridimensional.

Bidimensional (se realiza en dos procesos):

TipoVariable ***NombreVariablePuntero* = new **TipoVariable** *[*Indice*];

NombreVariablePuntero = new **TipoVarialbe** [*Subíndice*] (Realizar dentro de una función)

Tridimensional o mayores. No he encontrado información con la que desarrollar algún ejemplo, y no serán tratadas aquí.

2) *Declarar primero la variable puntero y después asignar(espacio de memoria):*

TipoVariable **NombreVariablePuntero;* // Declarar variable puntero

NombreVariablePuntero = *new* **TipoVariable;** // Asignar sin *

No es sólo memoria dinámica porque se crea durante la ejecución del programa, sino también porque se puede eliminar. Esta sería la forma:

delete *NombreVariablePuntero*; // Libera el espacio que ocupa en memoria
NombreVariablePuntero=0; // (Sin *) Dirección de la variable puntero a 0

En forma de matriz:
delete[] *NombreVariablePuntero*;
NombreVariablePuntero=0;

Al ejecutar la orden delete se libera el espacio ocupado en memoria pero no se elimina la variable puntero, por esta razón se le asigna la dirección 0 (o NULL que es una macro de c++, #define NULL 0).

Además, al ser una variable puntero, el lugar desde donde realicemos la asignación del valor puede generar errores de compilación. Ejemplo:

INCORRECTO	CORRECTO	CORRECTO

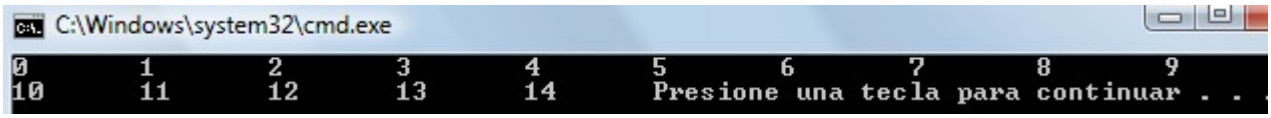
// Declarar y asignar fuera función
int *iVar1=new int;
// Asignar valor fuera funcion
*iVar1=5;
 	
void main() { }
	// Declarar y asignar fuera función
	int *iVar1=new int;
	
	void main()
	{ // Asignar valor dentro de funcion
	// Con *, valor dirección a memoria
	*iVar1=5;
	}
		void main()
		{
		int *iVar1=new int;
		*iVar1=5;
		}

En el caso de que la variable puntero apunta a un dato NO MATRIZ no habría problema en asignar un valor: Utilizaríamos **NombreVariablePuntero* (con *) para asignar un valor a esa dirección a memoria.

Pero si se trata de un dato tipo MATRIZ no podremos utilizar sizeof para saber el número de elementos que forma esa matriz (ya que utilizamos variable puntero, NO variable dato), sino que devolverá los bytes que ocupa el **TipoVariable**, por ejemplo 4 para int. No obstante, con las matrices de texto se puede utilizar el último elemento de la cadena de texto '\0' para saber su dimensión (si son unidimensionales).

Un problema añadido lo encontramos cuando queremos crear matrices de 2 dimensiones o más. Por ejemplo con una matriz unidimensional no habría relativamente problemas:

```
#include <iostream>
using namespace std;

int *puntero=0; // Declarar y asignar puntero

void main()
{
	puntero=new int[15];  // Matriz unidimensional de 15 valores
	for (int i=0; i<15; i++)
	{
		*(puntero+i)=i;
	}
	for (int i=0; i<15; i++)
	{
		cout << *(puntero+i) << "\t";
	}
	delete[] puntero;
	puntero=0;
}
```



Vemos que para seleccionar cada elemento de la matriz, le sumo a la *variable puntero* (sin *, dirección a memoria del primer elemento de la matriz, posición 0) una unidad, debido a que, estamos **utilizando la variable puntero real** (NO una copia como dentro de una función).

Además la sitúo entre paréntesis, precedida por un asterisco (*), para manipular el valor en la dirección a memoria.

El mismo ejemplo anterior en el que he añadido la función asignar donde sí que se cambia la dirección a memoria pero del parámetro o variable puntero de la función (copia de la llamada):

```
#include <iostream>
using namespace std;

int *puntero=0; // Declarar y asignar puntero
void asignar (int *var)
{
	for (int i=0; i<15; i++)
	{
		*var=i;  // Con *, valor de dirección a memoria.
		++var;  // Manejamos una copia que apunta a la misma dirección a memoria.
	}
}

void main()
{
	puntero=new int[15];  // Matriz unidimensional de 15 valores
	asignar(puntero);
	for (int i=0; i<15; i++)
	{
		cout << *(puntero+i) << "\t";
	}
	delete[] puntero;
	puntero=0;
}
```

Otra consideración es la numeración de cada elemento de la matriz respecto a su posición. Por ejemplo, el primer elemento de la matriz ocupa la posición cero, y el último en este caso sería el 14 o exactamente *(puntero+14), aunque en la declaración aparezca puntero=new int[15]; . ya que si contamos el intervalo 0-14 ambos inclusive, resulta 15 elementos.

Con las matrices de texto recordar que estará formada por “número de caracteres o letras de la cadena”+1. Donde el último valor de la cadena será 0. Esta última característica, como hemos visto en algunos ejemplos, hace que podamos crear un proceso (un bucle) capaz de detectar el final de cadena aplicando una sentencia while.

Librería string

En la práctica, para manipular cadenas de texto se utiliza otra **librería** que lleva implementada el IDE (no es necesario instalación) llamada '**string**' que incluye las siguientes funciones:

strcpy (copia, original); // Copia 'original' en 'copia'

*Con **strcpy** podemos asignar un valor a la variable pues hace una copia idéntica de original que puede ser una variable o texto(“Texto”, 'A',””) sin dejar rastro del contenido anterior de copia*

strlen (cadena); // Devuelve el número de caracteres de la cadena
strcat (resultado, añadido); // Añade 'añadido' a 'resultado'

Sirve para unir/añadir cadenas de texto

strcmp (cadenaA, cadenaB); //Compara cadenas teniendo en cuentas mayúsculas/minúsculas.
strcomp (cadenaA, cadenaB); //Compara cadenas **sin** tener en cuenta ” / “
strcpy_s(copia,tamaño,original); //Copia donde seleccionamos tamaño a copiar
strcat_s(resultado,tamaño,añadido); //Añadir donde seleccionamos tamaño a agregar

*Con strcpy_s y strcat_s establecemos el tamaño máximo de la cadena resultante en la operación(**Nunca** debe ser mayor que la dimensión-1 (tener en cuenta '\0') de la variable puntero).*

Además trabajan directamente con direcciones a memoria (*variable puntero* sin *). Estas instrucciones son muy útiles, una vez sabido lo que cuesta manipular matrices de texto. No las mencioné anteriormente para así poder ver aspectos sobre *variables puntero* y *referencias*.

Pero si comienzas a crear tu juego pronto te darás cuenta que la mayorías de acciones se realizan desde funciones externas al bloque de función donde te encuentras (en este caso main) y debemos solucionar como modificar la dimensión de las matrices de texto desde esas funciones, sabiendo que utilizan copias de los datos de la llamada y no las propias variables puntero donde se encuentran las cadenas.

Si no hubiera que modificar la matriz no habría tanto problema pues se podría utilizar return.

Ejemplo:

```
#include <iostream>
#include <string>

using namespace std;

char *cTex1=new char[4];
char *cTex2=new char[4];

void agregar (char *var1, char *var2)
{
	char *Nuevo=new char[strlen(var1)+strlen(var2)+1]; // Calcular dimension de la nueva cadena
	strcpy_s(Nuevo,strlen(Nuevo)+1,var1);
	strcat_s(Nuevo,strlen(Nuevo)+1,var2);
	delete[] var1;
	char *var1=new char[strlen(Nuevo)+1];  //** ERROR **
	strcpy_s(var1,strlen(var1)+1,Nuevo);
	delete[] Nuevo;
	Nuevo=0;
}

void main ()
{
	strcpy(cTex1,"Uno");  // Copia cadena "Uno" a cTex1
	strcpy(cTex2,"Dos");  // Copia cadena "Dos" a cTex2
	agregar(cTex1,cTex2); // Llamada a funcion agregar
	delete[] cTex1;
	delete[] cTex2;
	cTex1=cTex2=0;  // Esta asignación es válida
}
```

El código anterior produce el siguiente error.

```
error C2082: nueva definición del parámetro formal 'var1'
```

Ya que he intentado redimensionar el parámetro de la función var1. Es necesario redimensionar la matriz ya que la cadena resultante de agregar es mayor a las cadenas de origen. Debido a que en los

parámetros de la función recibimos datos y no los nombres de las variables, probé redimensionar el parámetro aún sabiendo que los parámetros son copias de los datos de la llamada.

La única solución que he encontrado a esto consiste en crear matrices o índices en el nombre de la variable puntero, es decir cText[i]. Pero esta acción lleva un procedimiento que paso a comentar a continuación.

Crear un índice en el nombre de la variable (memoria dinámica)

Debemos crear o declarar un componente que no hemos utilizado hasta ahora, se trata de una ***variable puntero* a otra *variables puntero* (**)**. El ejemplo siguiente trata sobre matrices texto, pero es aplicable a números igualmente. La forma general sería:

TipoVariable ****NombreVariablePuntero*=new **TipoVariable *[*NElementos*];

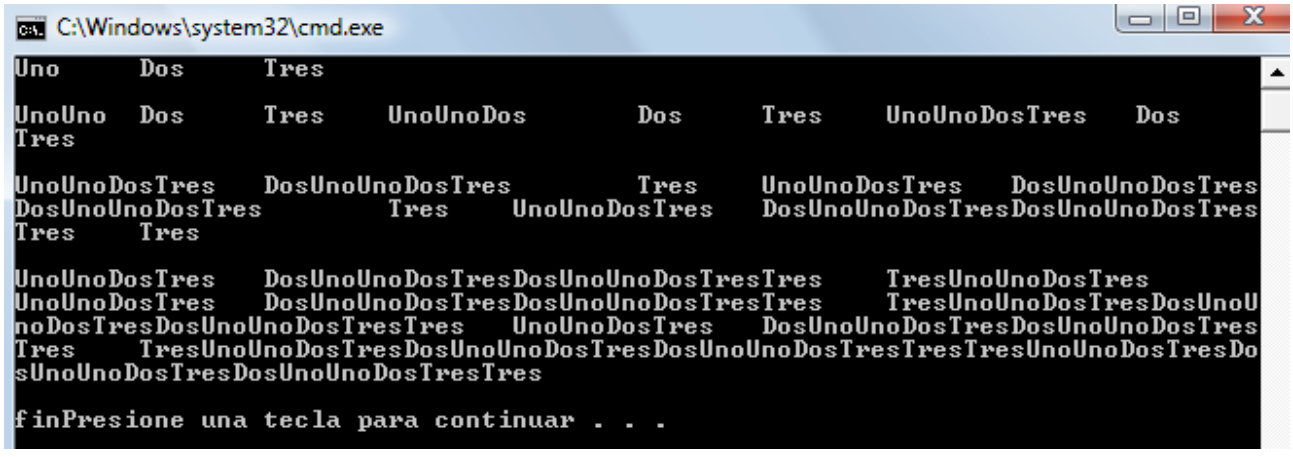
Posteriormente se asigna la dimensión de cada índice o fila de la matriz (dentro de una función) utilizando un bucle creado con for:

for (int i=0; i<*NElementos*; i++) { *NombreVariablePuntero*[i]=new **TipoVariable** [*Dim*];}

Sin más preámbulos muestro el ejemplo:

```
#include <iostream>
#include <string>

using namespace std;

char **cTex=new char *[3];  // Creamos un índice en el nombre de la variable de 3 elementos (serán 0,1,2)
char **cCopia=new char *[3];  // Aquí crearé una copia temporal para poder redimensionar cTex

void agregar (char *var1, char *var2);  // Declaración función

void main ()
{
	for (int i=0; i<3; i++) {cTex[i]=new char[5];}  // Le asigno una capacidad de 4 letras (4+'\0'=5)
	strcpy(cTex[0],"Uno");  // Ninguna asignación inicial sobrepasa los 4 caracteres
	strcpy(cTex[1],"Dos");
	strcpy(cTex[2],"Tres");
	for (int i=0; i<3; i++) {cout << cTex[i] << "\t";}
	cout << endl << endl;
	for (int ii=0; ii<3; ii++)
	{
		agregar (cTex[0],cTex[ii]);
		for (int i=0; i<3; i++) {cout << cTex[i] << "\t";}
	}
	cout << endl << endl;
	for (int ii=0; ii<3; ii++)
	{
		agregar (cTex[1],cTex[ii]);
		for (int i=0; i<3; i++) {cout << cTex[i] << "\t";}
	}
	cout << endl << endl;
		for (int ii=0; ii<3; ii++)
	{
		agregar (cTex[2],cTex[ii]);
		for (int i=0; i<3; i++) {cout << cTex[i] << "\t";}
	}
```

```
	cout << endl << endl << "fin";
	for (int i=0; i<3; i++) {delete[] cTex[i];} // Eliminamos las filas de la matriz cTex
	                                            // Las filas de cCopia ya han sido borradas en la funcion
	delete[] cTex;  // Eliminamos puntero a vectores
	delete[] cCopia;
}


void agregar (char *var1, char *var2)
{
	int iVar1=-1, iVar2=-1;  // Indicaran cuales son los índice pasados en la llamada
	int mx, mx2=0;  // mx maximo union de cadenas, mx2 maximo en cTex
	bool redimensionar=true;  // Condicional para redimensionar cTex
	for (int i=0; i<3; i++)
	{
		if (var1==cTex[i]) {iVar1=i;} //Comparar los dos valores de la
		if (var2==cTex[i]) {iVar2=i;} //llamada con todos los índices
	}
	if ((iVar1==-1) || (iVar2==-1)) {return;}  // No dejar pasar. Evitar ERRORES
	for (int i=0; i<3; i++)
	{
		mx=strlen(cTex[i]);  // Tengo que asignar el valor a una variable porque strlen es unsigned
		if (mx2<mx) {mx2=mx;}  // mx2 tomara el mayor valor de todo el índice
	}
	mx=strlen(cTex[iVar1])+strlen(cTex[iVar2]);  // mx toma la extensión de las cadenas de la llamada
	if (mx<=mx2) {mx=mx2; redimensionar=false;}  // Si mx<=mx2 no hace falta redimensionar cTex, y mx=mx2
	for (int i=0; i<3; i++) {cCopia[i]=new char[mx+1];} // Crear cCopia
	for (int i=0; i<3; i++) {strcpy(cCopia[i],cTex[i]);} // Copio cTex a cCopia
	strcat(cCopia[iVar1],cTex[iVar2]);  // Añado al valor del primer parámetro el segundo.
	if (redimensionar) { // Si la dimensión de cTex >= dimension Cadena1+Cadena2 no hace falta redimensionar
		for (int i=0; i<3; i++) {delete[] cTex[i];}    // Borramos valores de indices pero NO punteros a indices
		for (int i=0; i<3; i++) {cTex[i]=new char[mx+1];}  // Crear cTex con las nuevas dimensiones
	}
	for (int i=0; i<3; i++) {strcpy(cTex[i],cCopia[i]);}  //  Copiamos cCopia a cTex
	for (int i=0; i<3; i++) {delete[] cCopia[i];}  // Eliminamos valores de indices pero NO punteros a índices
}
```



Este ejercicios me sirvió para conocer más los comandos de la librería string. Probé con strcpy_s y strcat_s, pero tras varios errores de compilación, finalmente opté por las más comunes strcpy y strcat con la precaución de dimensionar correctamente la matriz.

En un principio sólo redimensionaba la variable modificada (la primera del parámetro) y tras varios errores más de compilación caí en la cuenta que tenía una fila con el doble de proporciones que el resto. Entonces añadí otro índice más cCopia y finalmente lo solucioné.

También me permitió conocer más sobre la función delete, pues no daba con la fórmula para eliminar sólo la fila y no toda la variable puntero.

Con esto ya pasamos al siguiente tema.

Clases y estructuras.

Es conocida mayoritariamente por clase, pero existen de dos tipos según el comando utilizado en su creación (struct o class):

struct Si no se expresa lo contrario (utilizando private:) los miembros de la estructura son públicos.

class Si no se expresa lo contrario (utilizando public:) los miembros de la clase son privados.

En este documento no voy a tratar las clases, sino las estructuras. Desarrollaré una estructura compuesta por variables y métodos. Los métodos son funciones que interactúan con las variables de la estructura.

Para desarrollar la estructura de un juego voy a utilizar las funciones de un motor de juego llamado ESENTHEL. Este motor no incluye una documentación propiamente dicha (por ahora), y tendremos que buscar la información necesaria entre los archivos de inclusión del motor y los archivos .cpp de ejemplo que adjunta.

Por esta razón, antes de seguir, voy a realizar unos comentarios para saber como instalar el motor, crear un nuevo proyecto con el IDE, una pequeña introducción al motor, y buscar esa información (funciones, macros, palabras reservadas del motor). Sitio de descarga del motor:

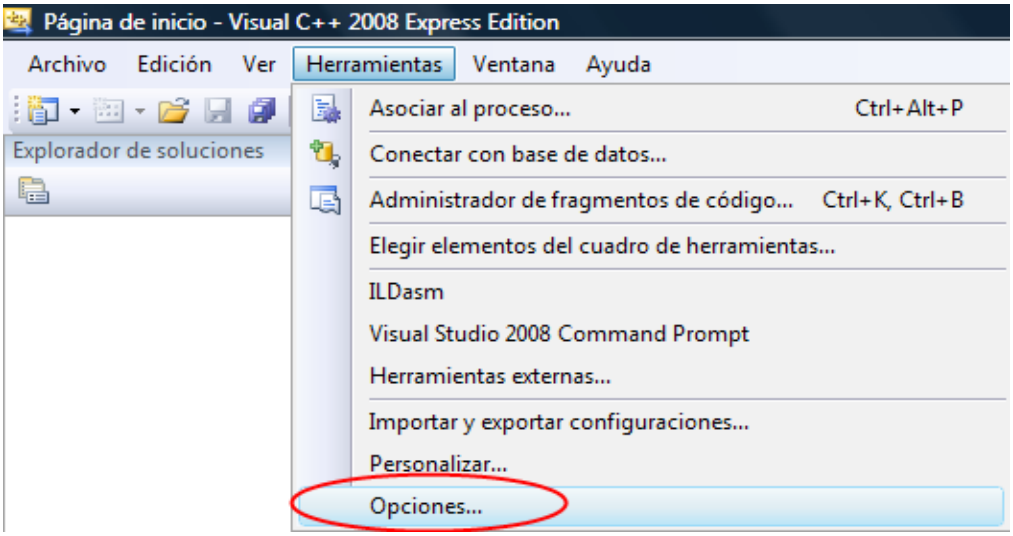
http://www.esenthel.com/en/engine/download.html

Después continuaré con las explicaciones sobre estructuras en c++.

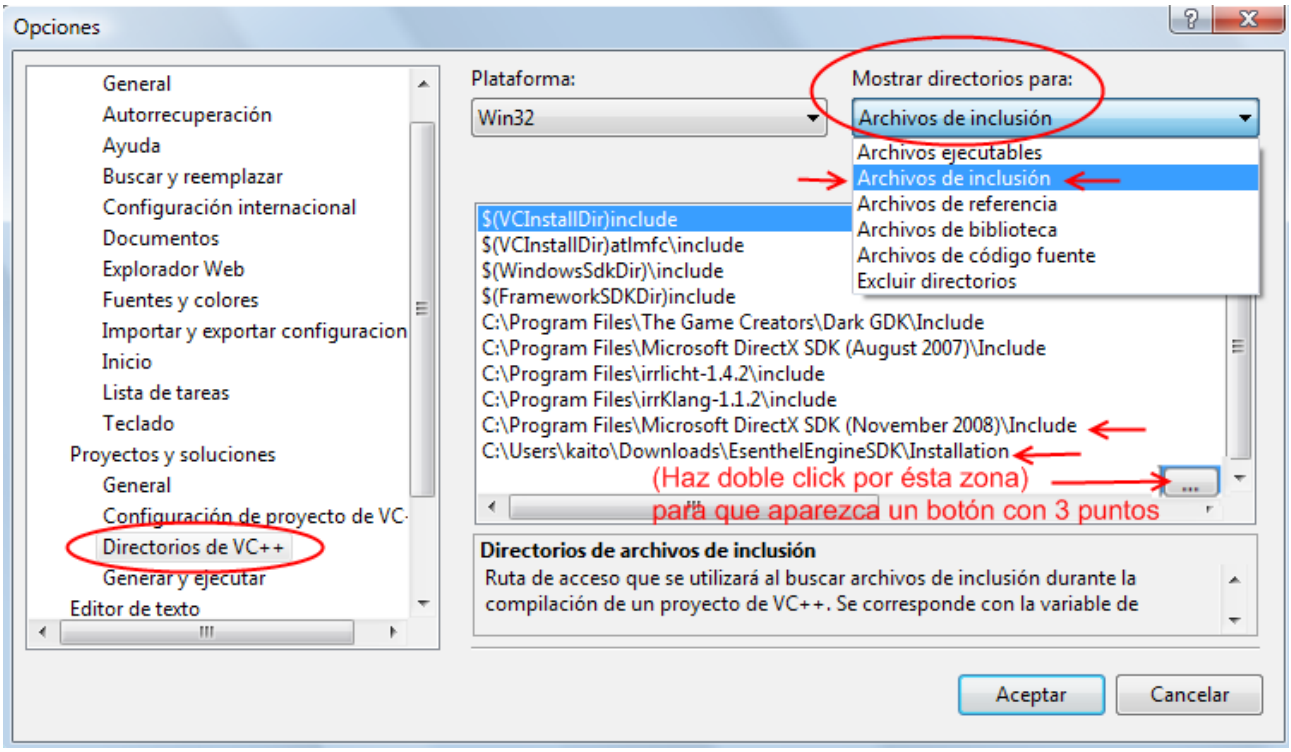
ESENTHEL

Instalación ESENTHEL.

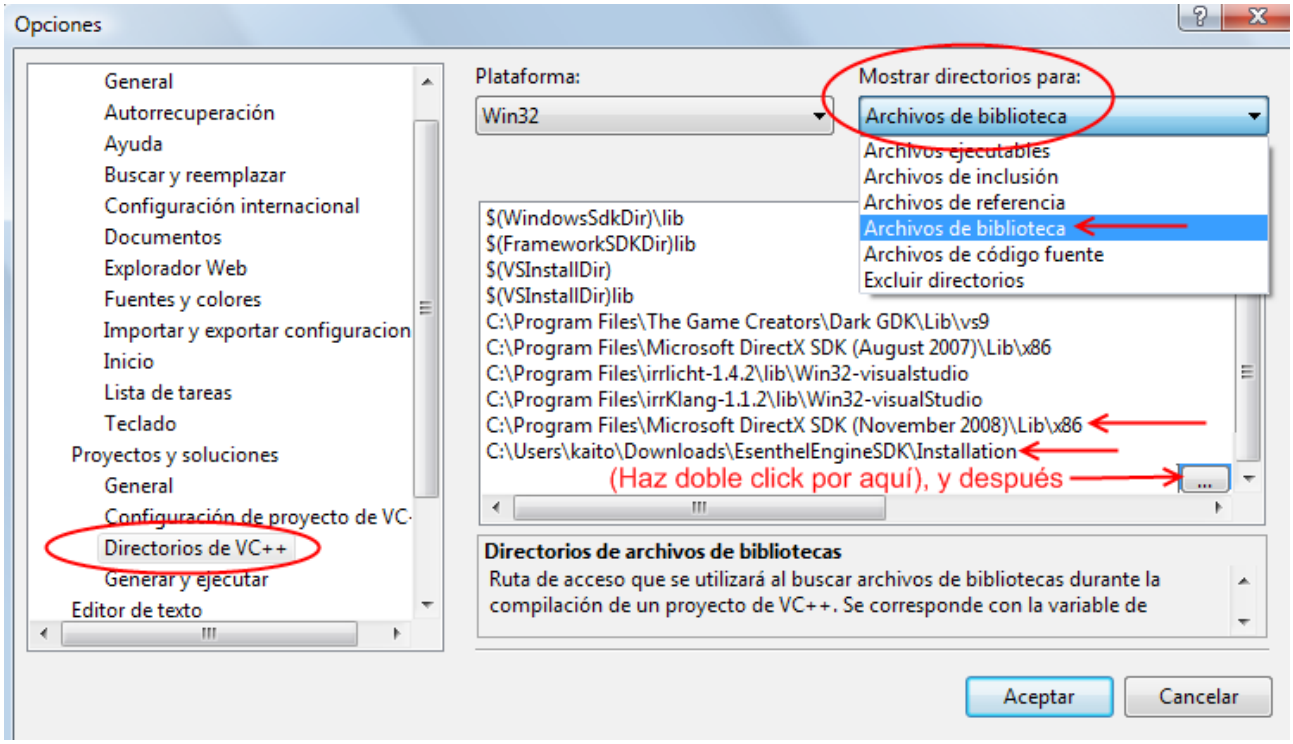
Después de instalar ESENTHEL (recomiendo NO instalar en la carpeta archivos de programa si utilizas windows Vista) junto a DirectX SDK noviembre 2008 y PhysX, debemos indicar al "IDE" (Integrated Development Environment – Entorno Integrado de Desarrollo) las rutas a las carpetas donde se encuentra las librerías (.lib) y archivos de inclusión (.h) del engine. Abrimos la IDE (en este caso Microsoft VC++ 2008 Express Edition) y vamos al menú 'Herramientas-->Opciones'.



Comenzamos indicando las rutas a los archivos de inclusión (.h) seleccionando 'Mostrar directorios para: Archivos de inclusión'. Con un doble click en el espacio en blanco, justamente debajo del último elemento de la lista, aparece un botón con 3 puntos seguidos. Lo seleccionamos para introducir las rutas a las carpetas con los archivos .h del engine y del DirectX SDK.

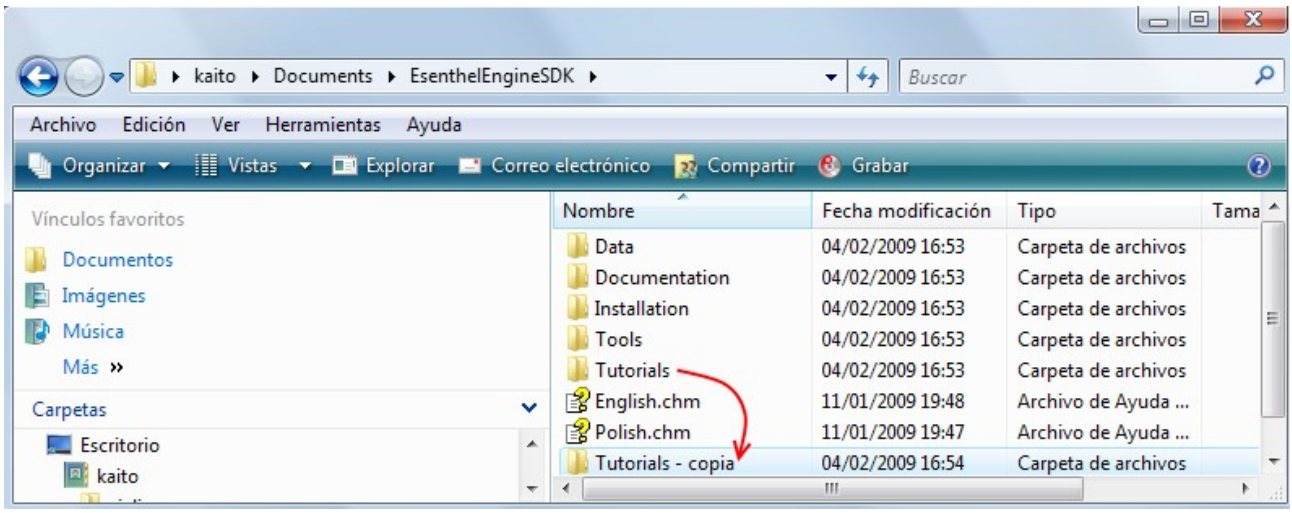


Procedemos a establecer los directorios para las librerías (.lib) cambiando 'Mostrar directorios para: Archivos de biblioteca', y establecemos del mismo modo que el paso anterior las rutas a las carpetas que contienen losr archivos (.lib) del engine y de DirectX SDK.



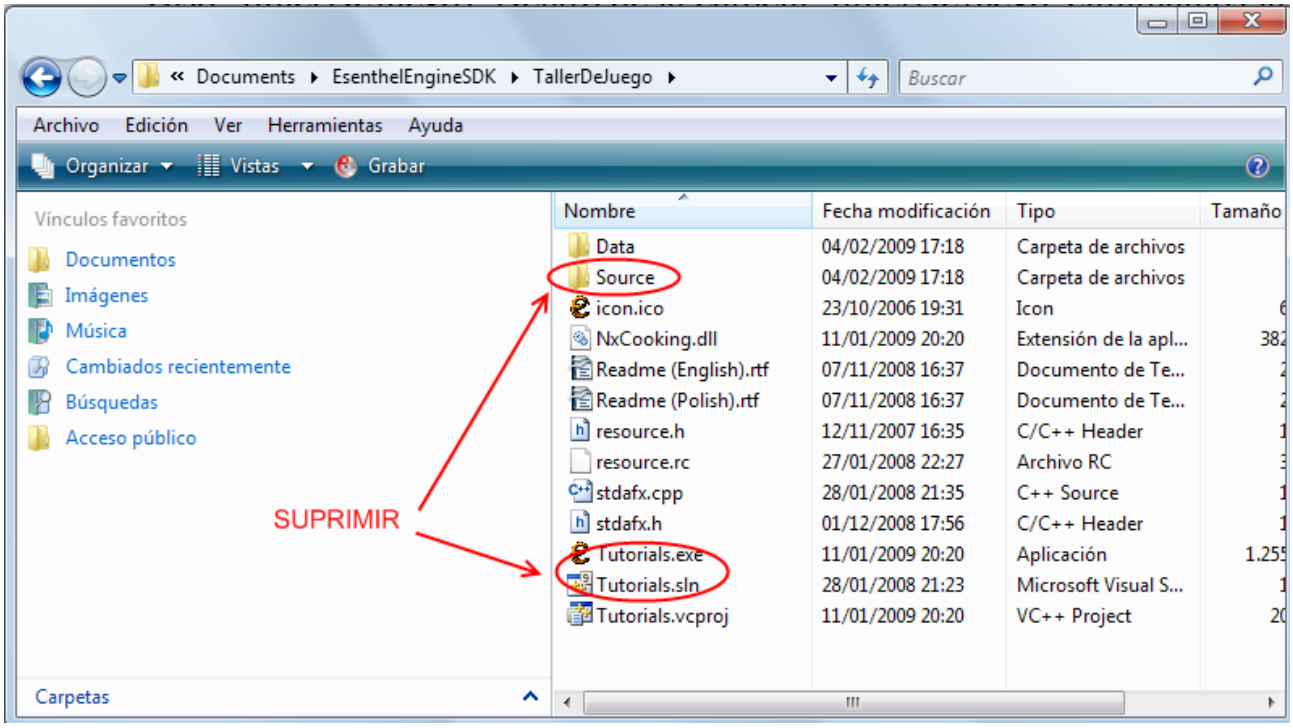
Pulsa el botón 'Aceptar' para finalizar y guardar los cambios. Por ahora ya hemos terminado con la IDE.

Ahora vamos a crear una carpeta para nuestro proyecto dentro de la misma carpeta de ESENTHEL SDK.

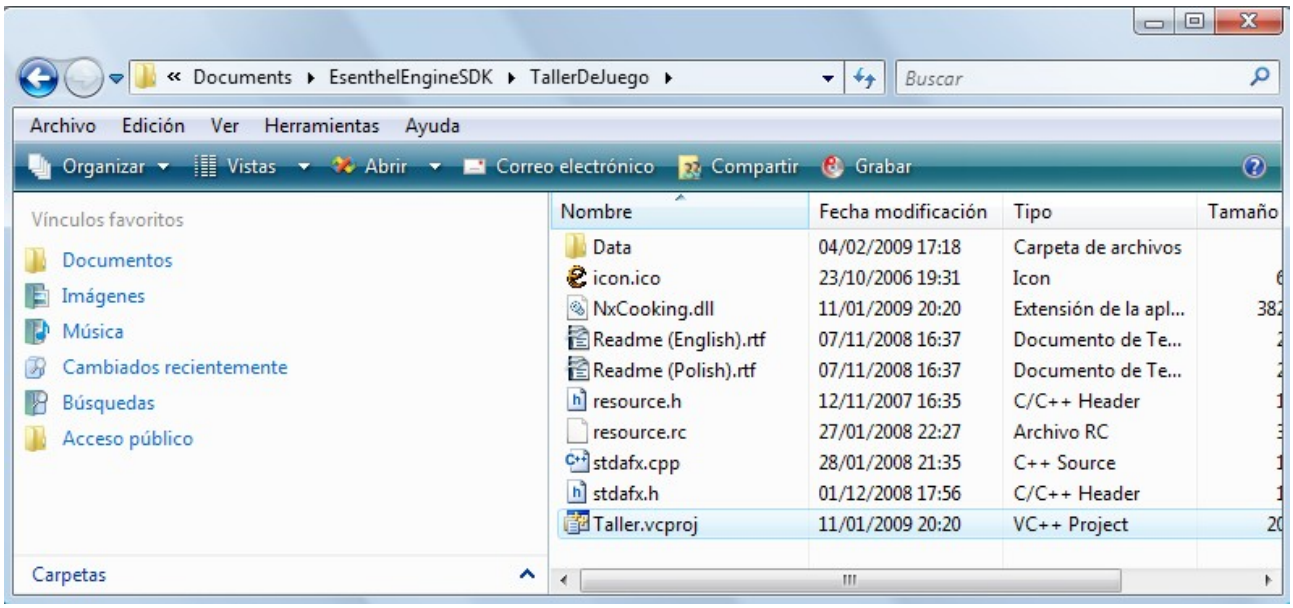


Cambiamos el nombre de la carpeta 'Tutorials – copia' por el nombre del juego o proyecto (en este caso, TallerDeJuego). Dentro de la carpeta 'TallerDeJuego' eliminamos la carpeta donde se encuentran los ejemplos o tutoriales del engine llamada 'Source', y también eliminamos los archivos 'Tutorials.exe' y 'Tutorials.sln'.

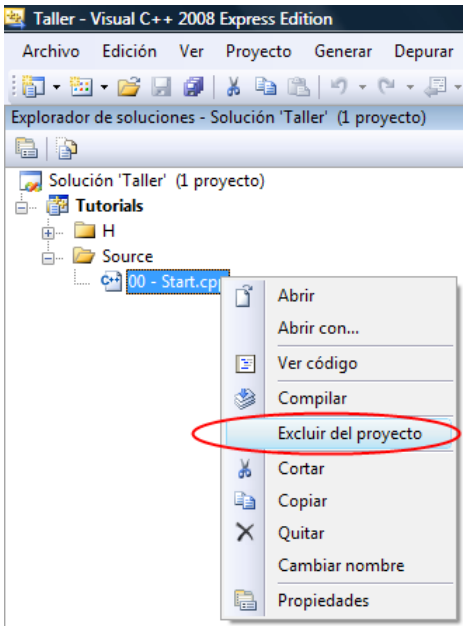
Imagen descriptiva.



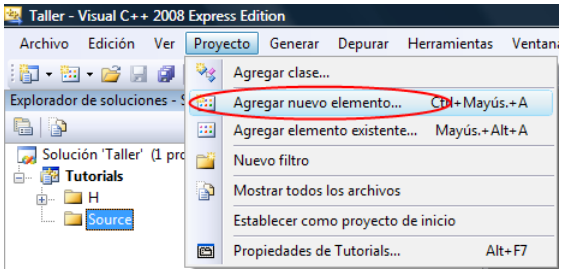
Cambiamos el nombre del archivo 'Tutorials.vcproj' (vcproj, quiere decir, visual C project o proyecto) por un nombre relacionado con el juego (en este caso, 'Taller.vcproj'). Este sería el aspecto de la carpeta.

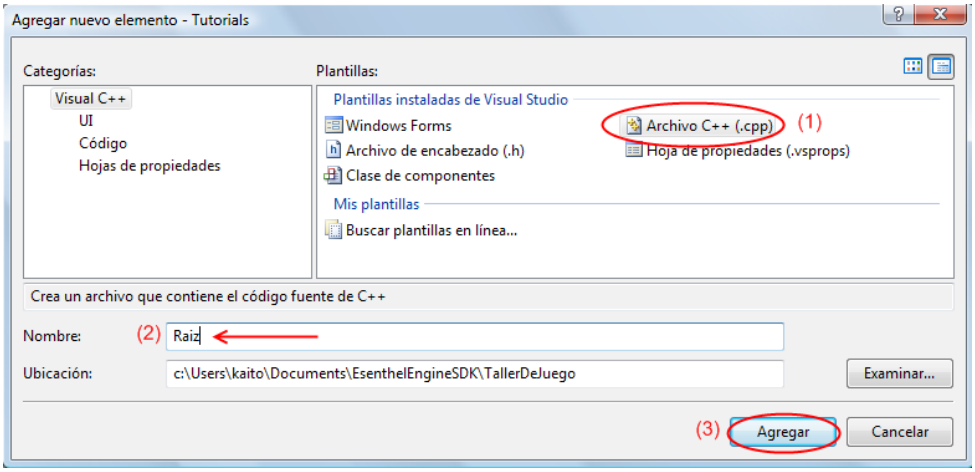


Al iniciar 'Taller.vcproj', abrimos la carpeta 'Source' del proyecto y con el BDR excluimos del proyecto el archivo .cpp que venía de ejemplo en la solución (en este caso, 'Start.cpp')

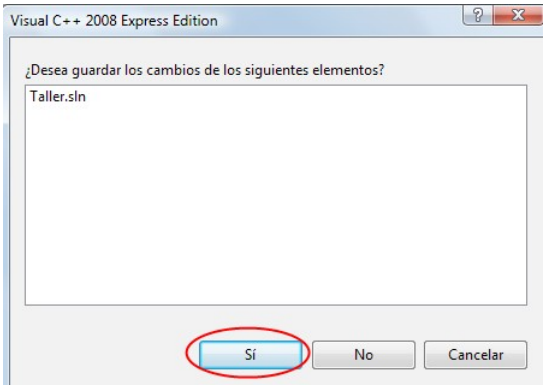


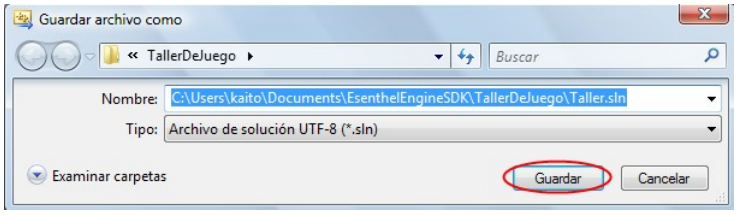
Ahora agregamos un nuevo archivo .cpp donde escribiremos el código del juego.



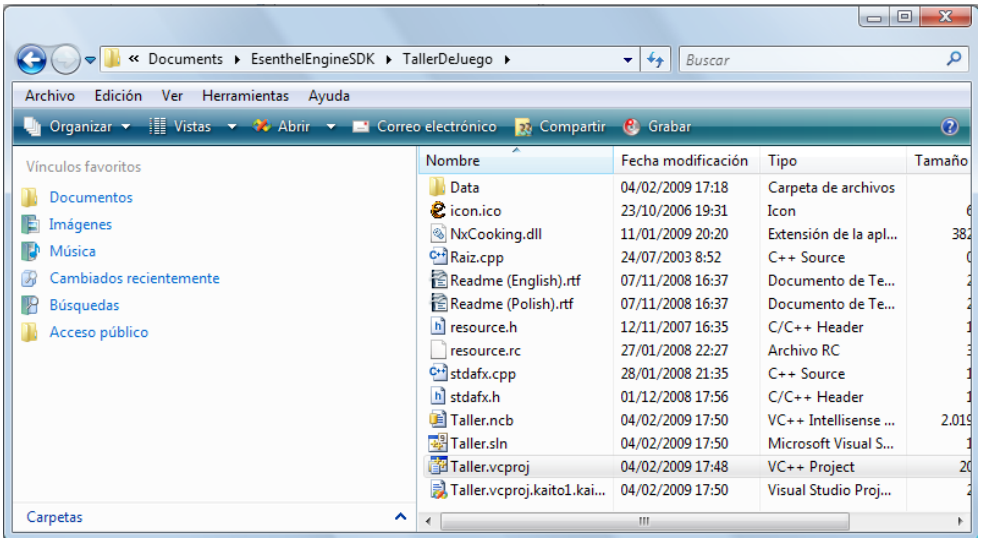


Ahora cerramos Visual C++  , y saldrá un mensaje como éste:



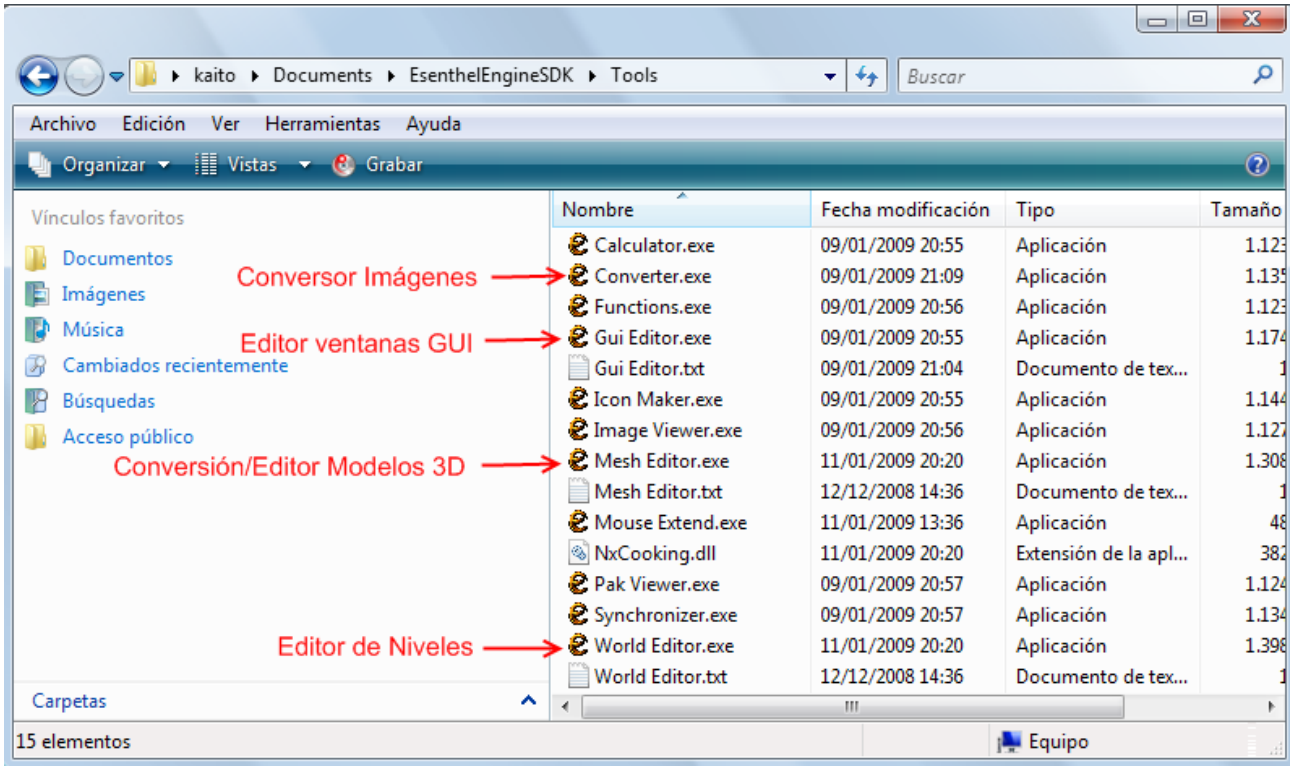


Este sería el aspecto de la carpeta o directorio.

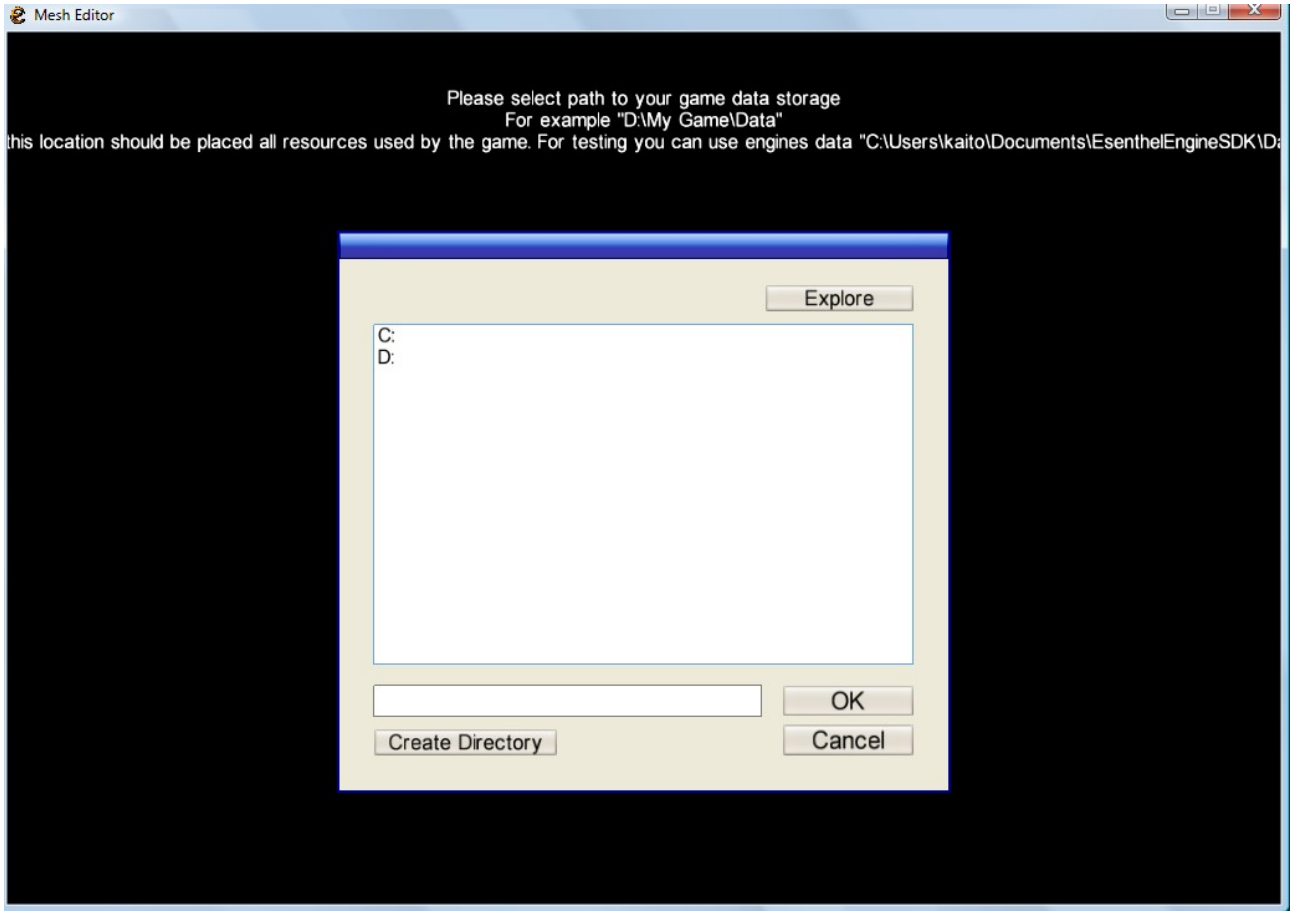


Vemos que dentro de la carpeta 'TallerDeJuego' existe otra carpeta llamada 'Data', que utilizaré para guardar los recursos específicos del juego. Dentro de la carpeta del engine 'EsenthelEngineSDK' existe otra carpeta 'Data' que incluye los recursos que utilizan los tutoriales .cpp y que también podremos utilizar.

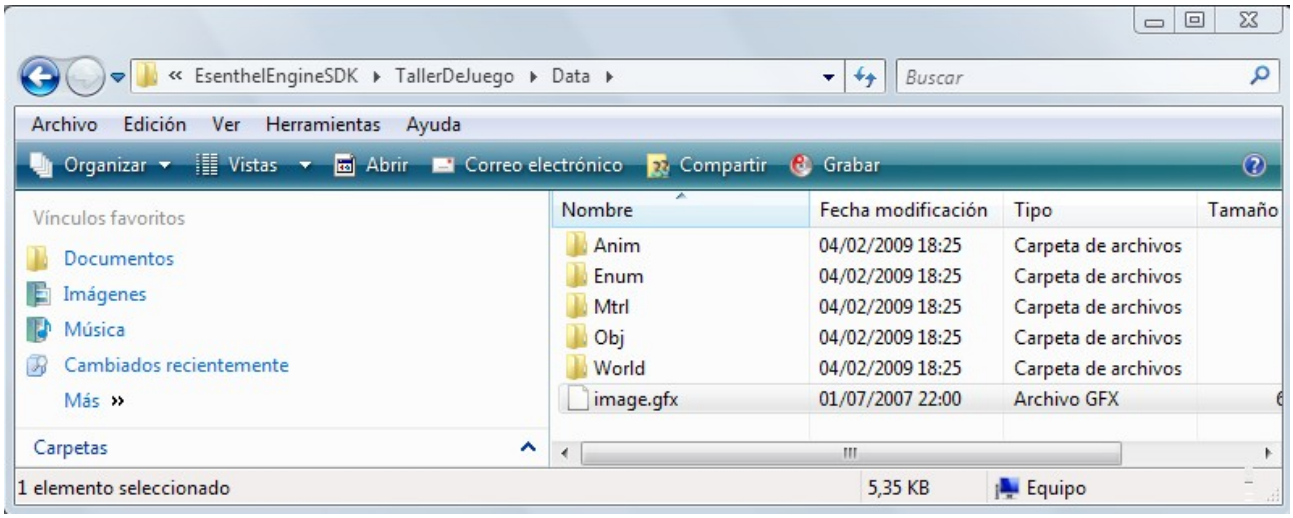
ESENTHEL utiliza un formato específico de gráficos llamado (.gfx) y otros formatos específicos para las mallas o modelos 3D, por eso utiliza un conversor de imágenes y otro conversor de modelos 3D. Estas herramientas se encuentran en la carpeta 'Tools' del engine. (es decir, ..\EsenthelEngineSDK\Tools)



La imagen inferior muestra la herramienta para manipular mallas (Mesh Editor).

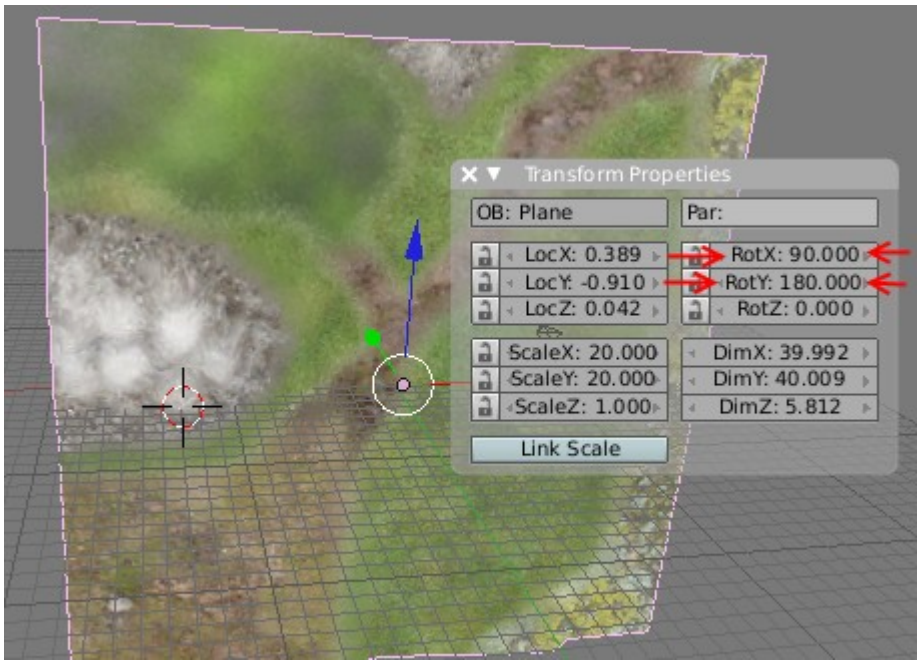


La primera vez que lo ejecutas te pregunta el directorio a utilizar para guardar los recursos creados. Le he asignado la carpeta 'Data' del directorio del 'TallerDeJuego'. Al hacer esto, creará un grupo de carpetas. Imagen inferior. El archivo 'image.gfx' ya existía anteriormente.

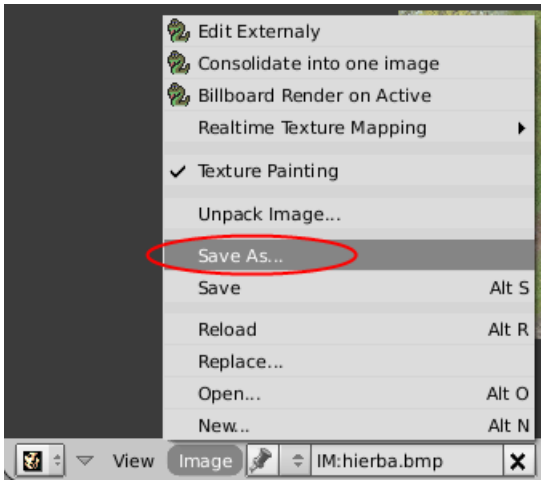


También vemos que en la carpeta 'Tools' el archivo 'Path.txt' ha guardado la ruta que hemos establecido anteriormente. Si lo borras, el 'Mesh Editor' volverá a preguntar la ruta de guardado. También podrías modificarlo si es necesario.

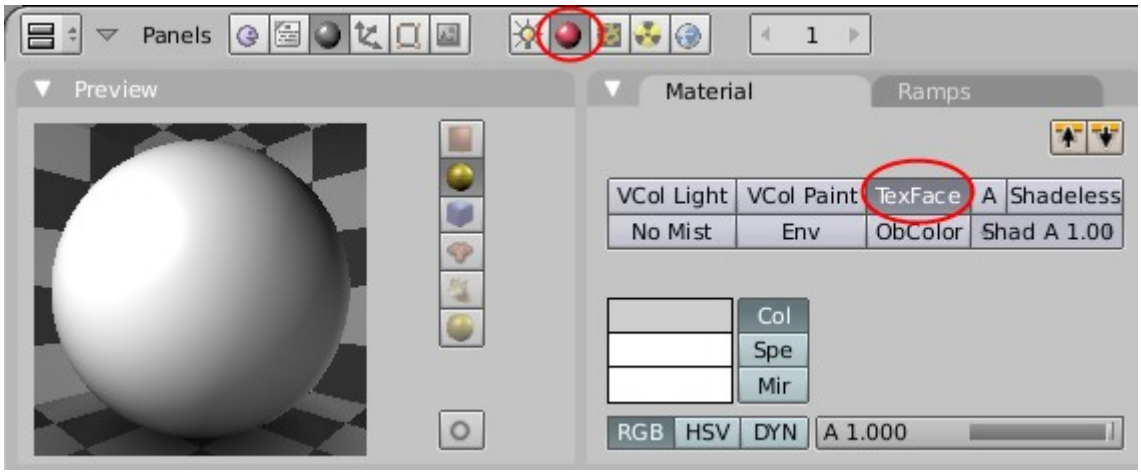
Ahora ya desde Blender preparamos el modelo 3D para verlo correctamente desde el editor del engine (en este caso, Mesh Editor). Debemos aplicar primero una rotación del eje X de +90º (RotX:0+90=90) y después una rotación del eje Y de +180º (Rot Y: 0+180=180).Utiliza el formulario, pulsa N.



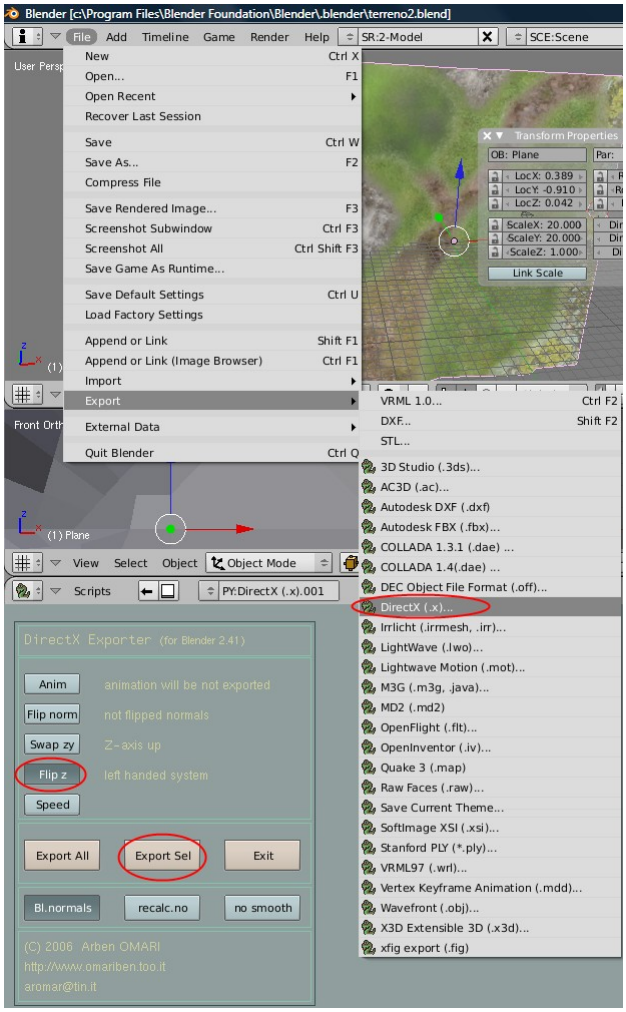
Voy a exportar la malla al formato DirectX (.x). Este exportador no te da la opción de guardar la imagen o textura que utiliza la malla junto al archivo (.x). Debemos guardar la textura por separado desde el 'UV/Image Editor', ejecutando la orden del menú 'Image-->Save As' (es decir, Guardar como, que te permite seleccionar un destino) y a continuación la guardamos en la carpeta 'Obj' que ha creado el 'Mesh Editor'.(en este caso sería, ..\TallerDeJuego\Data\Obj).



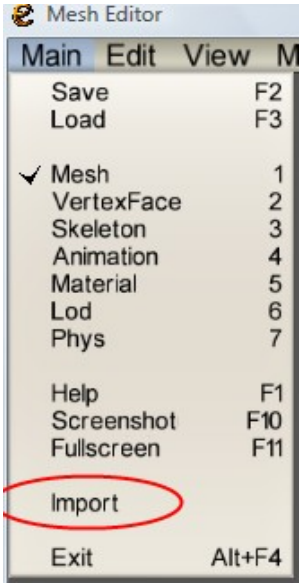
Antes de exportar también es conveniente asegurar que el material del modelo 3D tenga activado el botón 'TexFace' desde el panel 'Material Buttons' de la VENTANA 3.



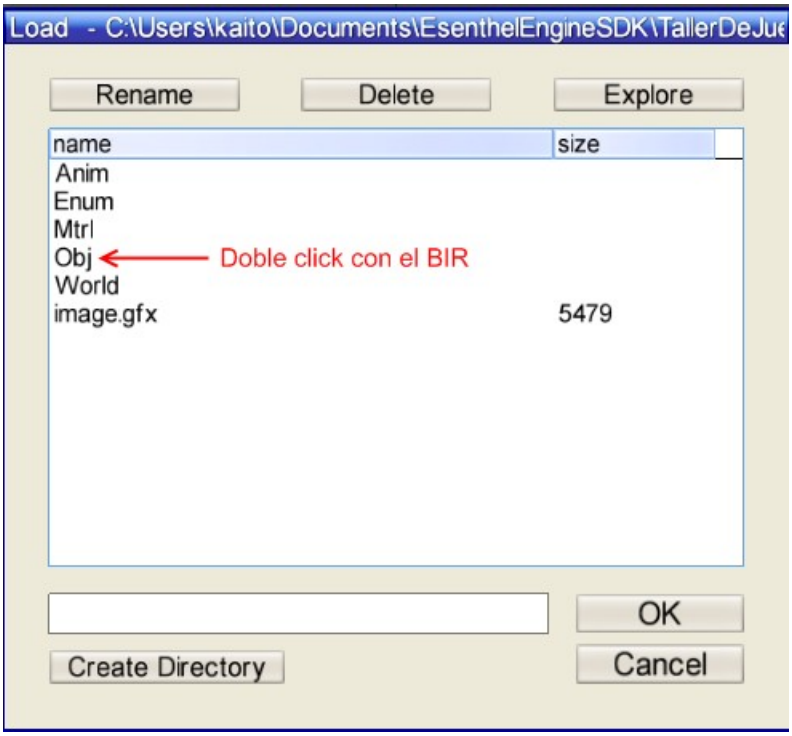
Ahora, ya desde el exportador del DirectX, con el botón 'Flip Z' activado guardamos el archivo (.x) en la misma carpeta donde guardamos la textura.



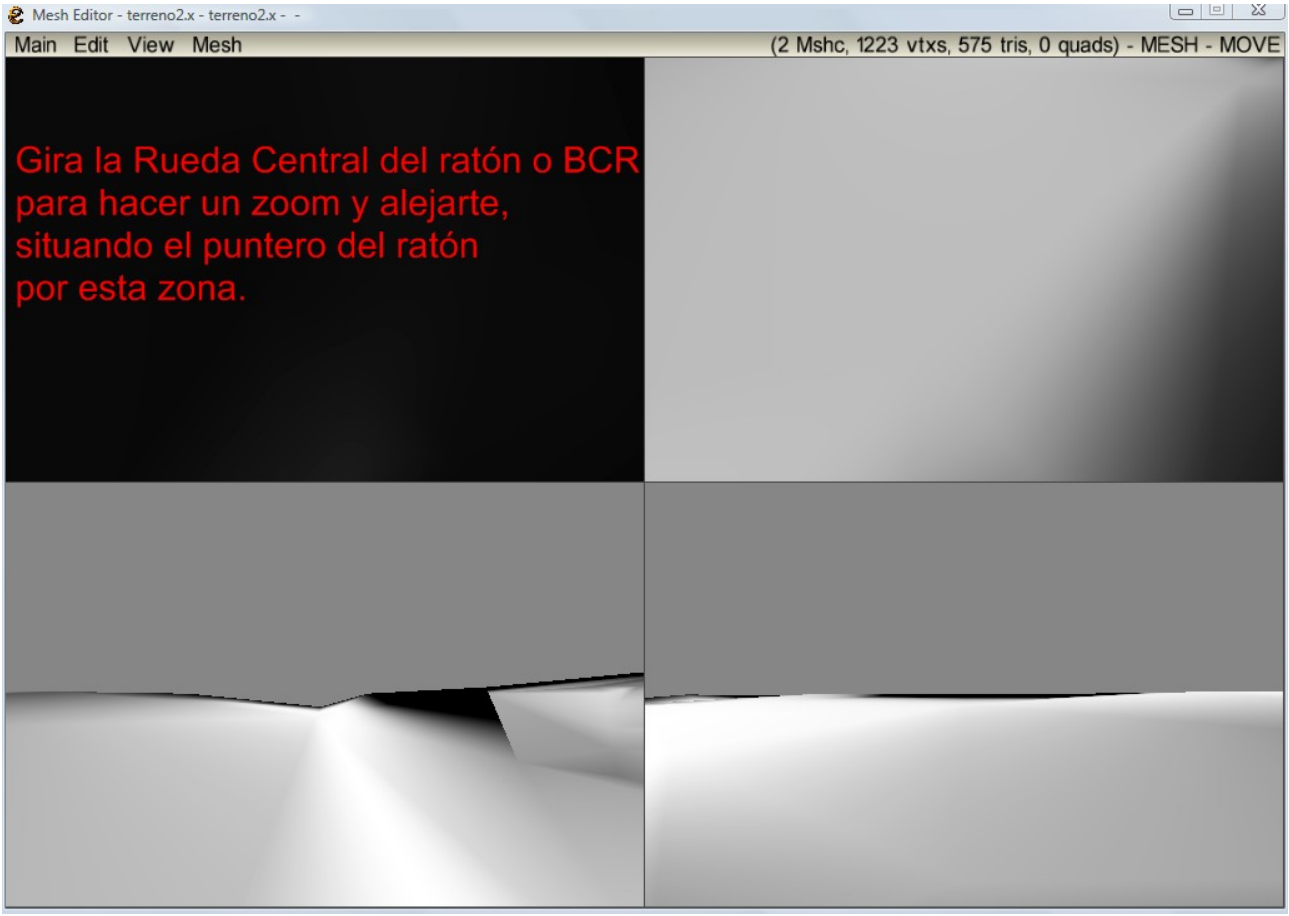
Abrimos el 'Mesh Editor' y ejecutamos la orden del menú 'Main-->Import'.



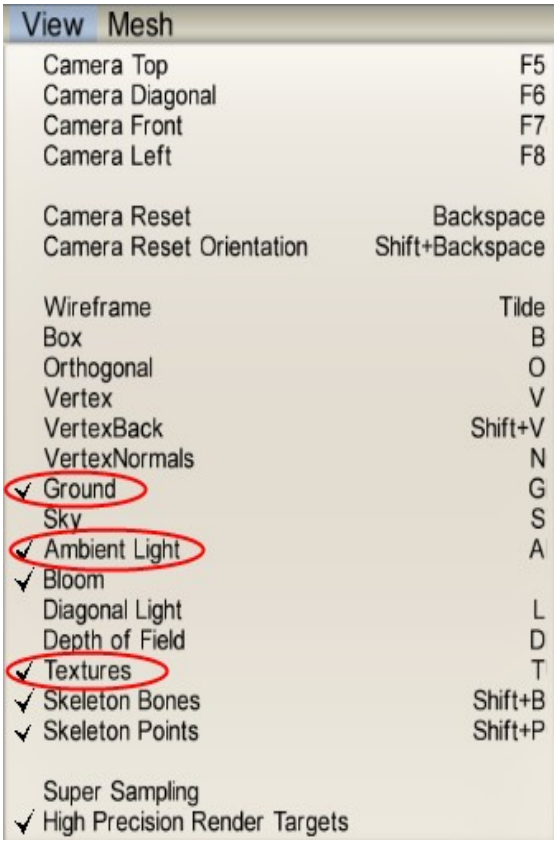
El 'Mesh Editor' muestra la ruta que establecimos en un principio. Entramos dentro de la carpeta 'Obj' para seleccionar el modelo 3D.



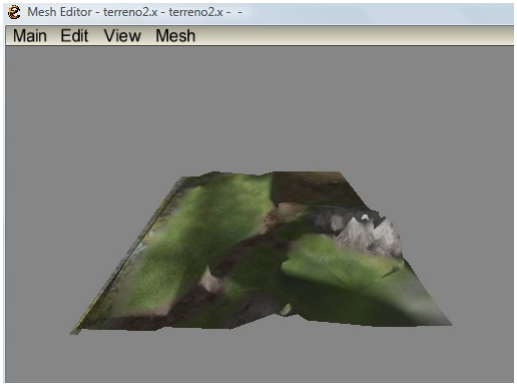
Una vez seleccionado veremos algo parecido a esto.



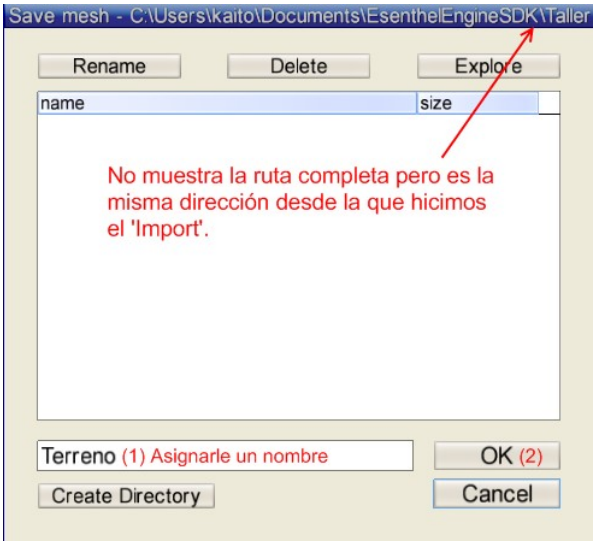
Además, desde el menú View selecciona o marca las opciones 'Ground' (es decir, Base) que muestra una rejilla que servirá de referencia tanto para las dimensiones como en la posición/orientación de la malla, 'Ambient Light' (es decir, luz ambiente) para dar más luz al escenario y 'Textures' (es decir, Texturas) para ver la malla (en inglés mesh) con la textura aplicada.

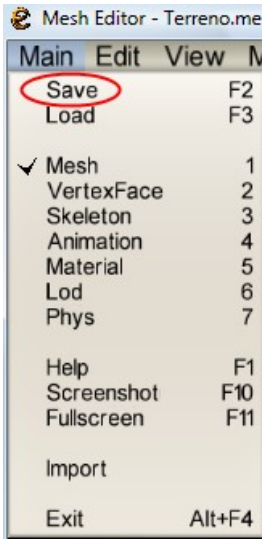


Observamos que la malla y la textura están extendidas de la misma forma que en Blender y de igual manera que la rejilla 'Ground'.

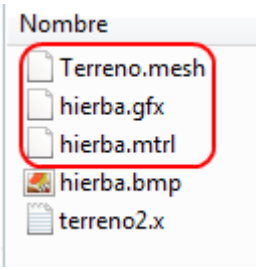


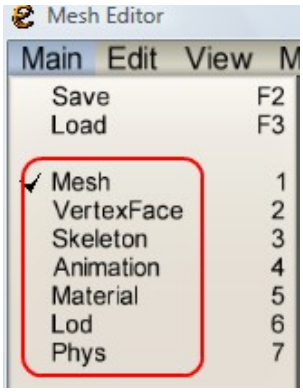
Para guardar la malla ejecutamos la orden del menú 'Main-->Save'





Este sería el aspecto de la carpeta 'Obj', con los archivos generados. El 'Mesh Editor' has creado un archivo (.mtrl) para el material, un archivo (.gfx) para la textura y el archivo al que haremos referencia en el código de programación (Terreno.mesh). Puedes eliminar 'hierba.bmp' y 'terreno2.x'.



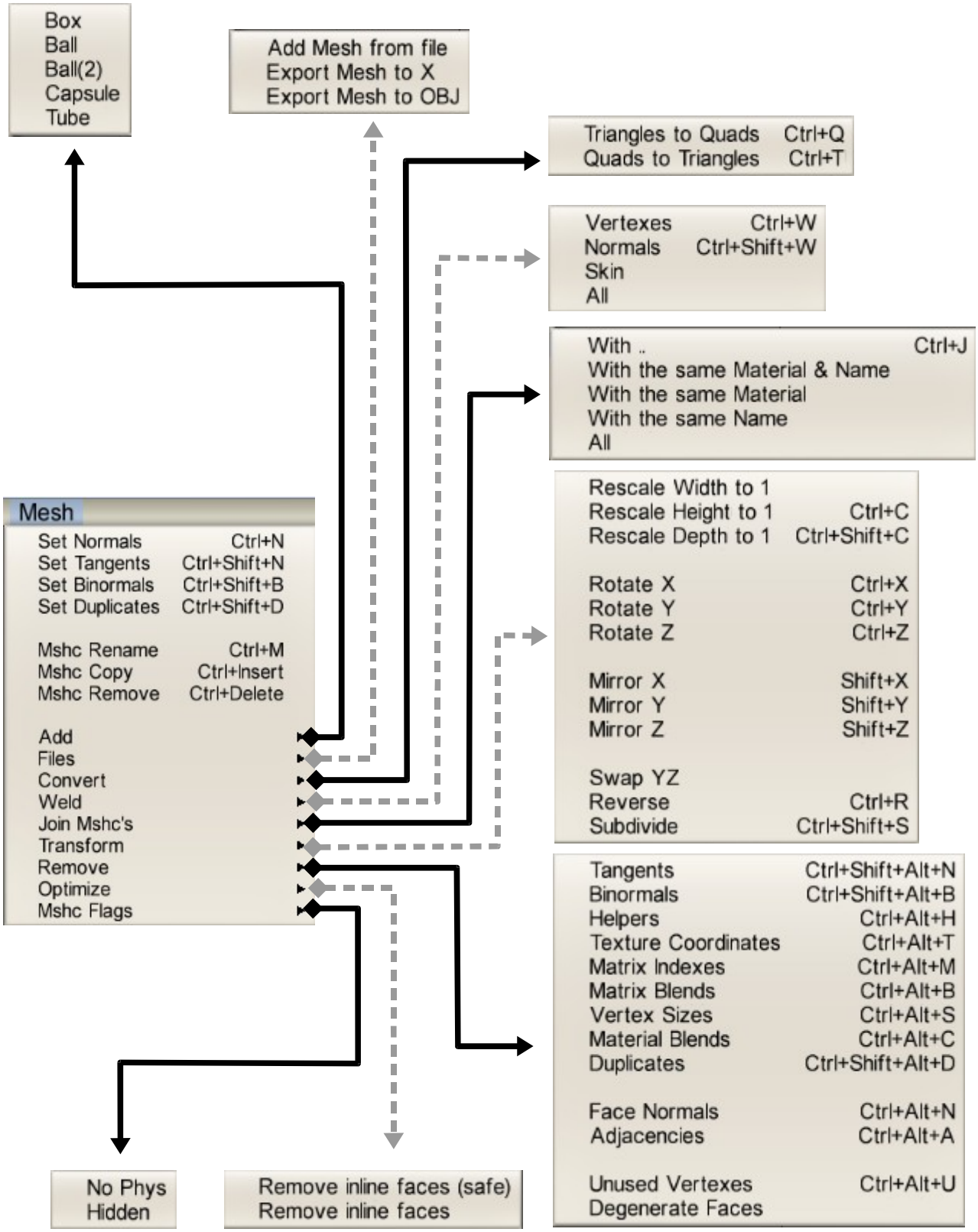


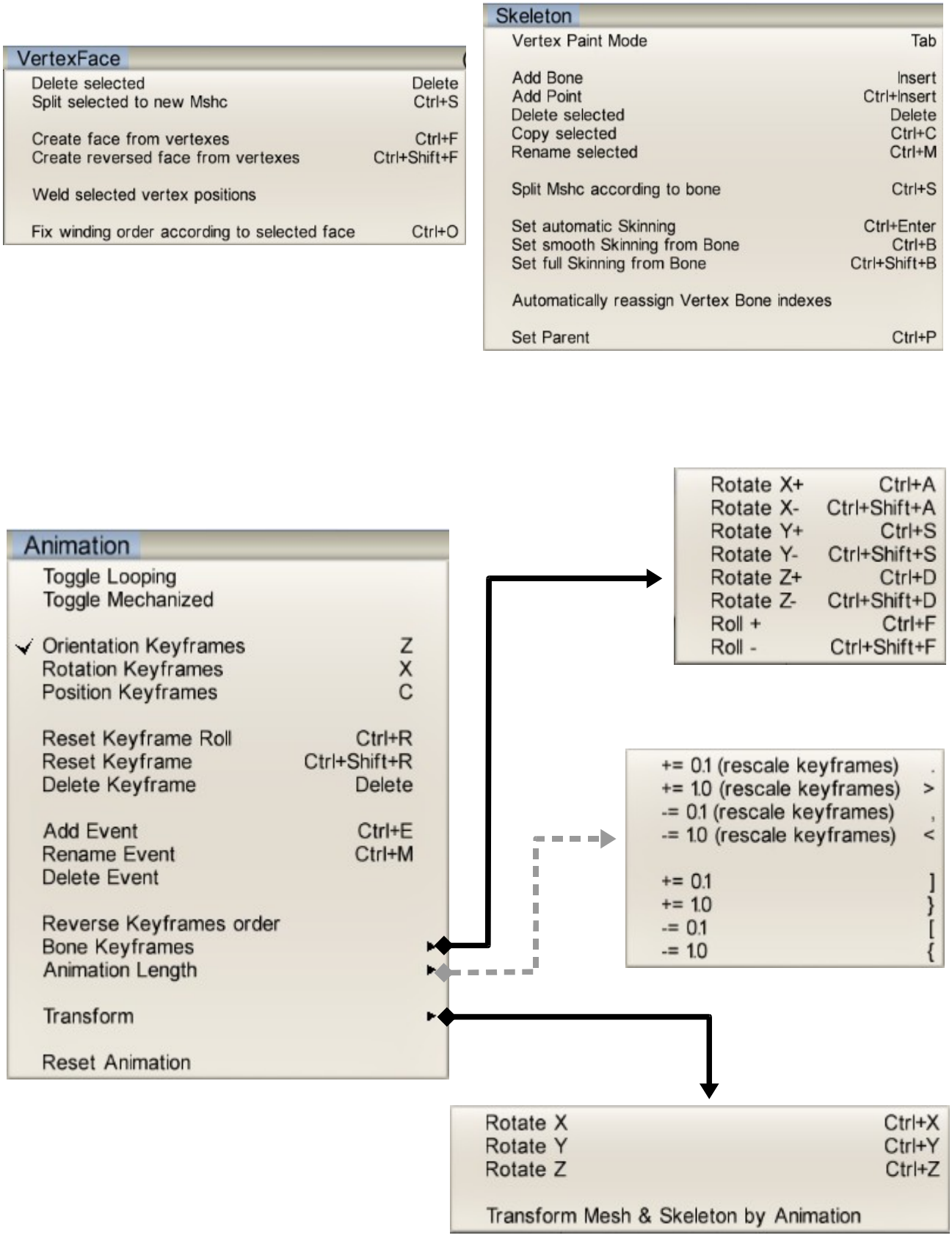
Desde el 'Mesh Editor' podemos modificar/crear modelos 3D. Para utilizar estas características primero debemos seleccionar el modo de trabajo desde el menú 'Main'. Puedes elegir entre: Mesh (malla), Vertex Face (Vértice/Cara), Skeleton (Esqueleto), Animation (Animación), Material, Lod (Level Of Detail, es decir, Nivel de detalle, relacionado con los terrenos 3D), Phys (de Physic, es decir, Física).

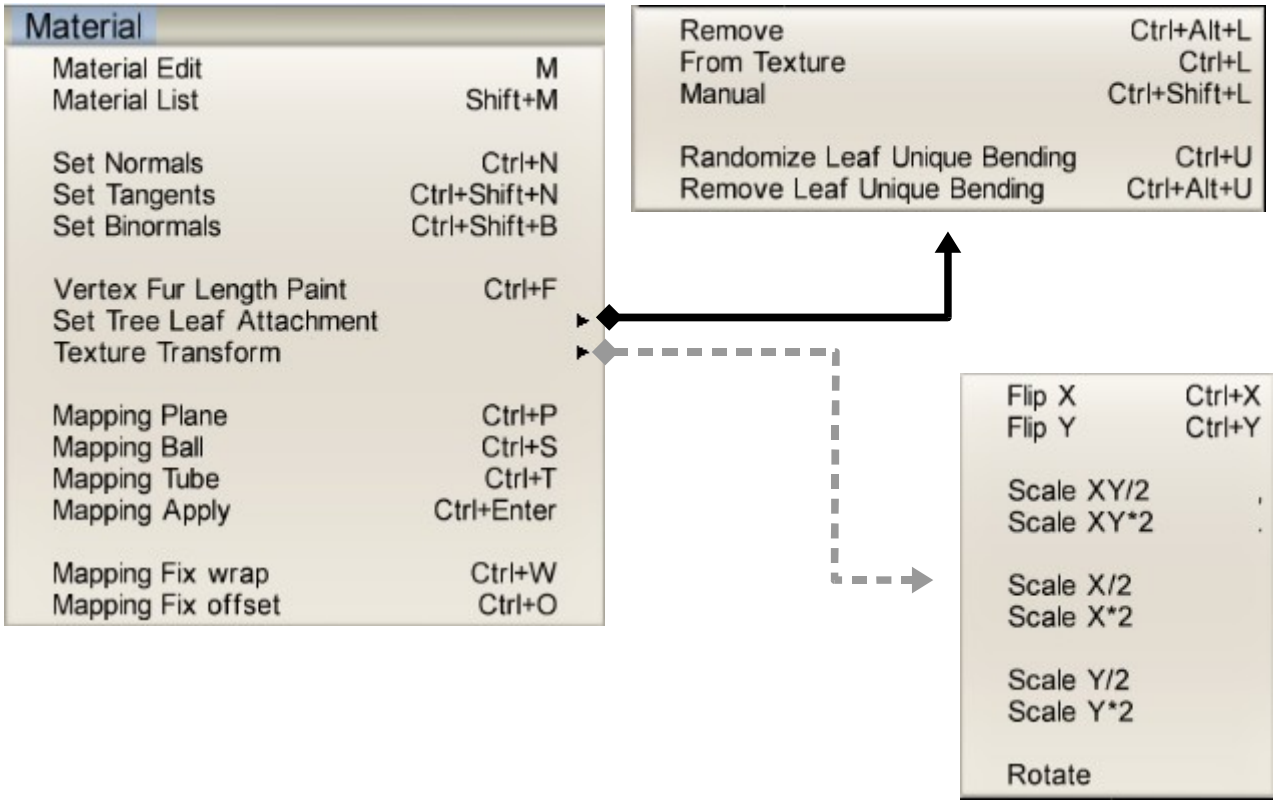
Otros términos ingleses utilizados en los menús: Weld (Soldar/Unir), Fix Winding Order (Fijar el orden de devanado/enrollado)

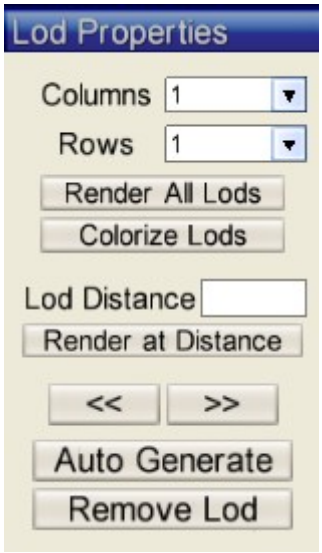
LOD. Más información: http://sabia.tic.udc.es/gc/Contenidos%20adicionales/trabajos/Programas3D/Terrenos3D/

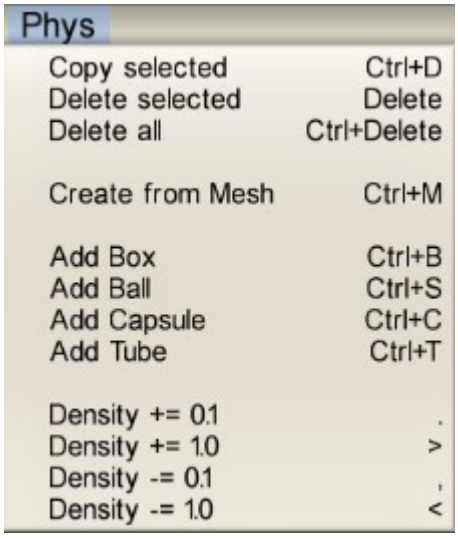
A continuación, incluyo unas capturas de los diferentes menú del 'Mesh Editor'.




VertexFace
Delete selected Delete
Split selected to new Mshc Ctrl+S
Create face from vertexes Ctrl+F
Create reversed face from vertexes Ctrl+Shift+F
Weld selected vertex positions
Fix winding order according to selected face Ctrl+O
Skeleton
Vertex Paint Mode Tab
Add Bone Insert
Add Point Ctrl+Insert
Delete selected Delete
Copy selected Ctrl+C
Rename selected Ctrl+M
Split Mshc according to bone Ctrl+S
Set automatic Skinning Ctrl+Enter
Set smooth Skinning from Bone Ctrl+B
Set full Skinning from Bone Ctrl+Shift+B
Automatically reassign Vertex Bone indexes
Set Parent Ctrl+P
Animation
Toggle Looping
Toggle Mechanized
Orientation Keyframes Z
Rotation Keyframes X
Position Keyframes C
Reset Keyframe Roll Ctrl+R
Reset Keyframe Ctrl+Shift+R
Delete Keyframe Delete
Add Event Ctrl+E
Rename Event Ctrl+M
Delete Event
Reverse Keyframes order
Bone Keyframes
Animation Length
Transform
Reset Animation
Rotate X+ Ctrl+A
Rotate X- Ctrl+Shift+A
Rotate Y+ Ctrl+S
Rotate Y- Ctrl+Shift+S
Rotate Z+ Ctrl+D
Rotate Z- Ctrl+Shift+D
Roll + Ctrl+F
Roll - Ctrl+Shift+F
+= 0.1 (rescale keyframes) .
+= 1.0 (rescale keyframes) >
-= 0.1 (rescale keyframes) ,
-= 1.0 (rescale keyframes) <
+= 0.1]
+= 1.0 }
-= 0.1 [
-= 1.0 {
Rotate X Ctrl+X
Rotate Y Ctrl+Y
Rotate Z Ctrl+Z
Transform Mesh & Skeleton by Animation


Material
Material Edit M
Material List Shift+M
Set Normals Ctrl+N
Set Tangents Ctrl+Shift+N
Set Binormals Ctrl+Shift+B
Vertex Fur Length Paint Ctrl+F
Set Tree Leaf Attachment
Texture Transform
Mapping Plane Ctrl+P
Mapping Ball Ctrl+S
Mapping Tube Ctrl+T
Mapping Apply Ctrl+Enter
Mapping Fix wrap Ctrl+W
Mapping Fix offset Ctrl+O
Remove Ctrl+Alt+L
From Texture Ctrl+L
Manual Ctrl+Shift+L
Randomize Leaf Unique Bending Ctrl+U
Remove Leaf Unique Bending Ctrl+Alt+U
Flip X Ctrl+X
Flip Y Ctrl+Y
Scale XY/2 ,
Scale XY*2 .
Scale X/2
Scale X*2
Scale Y/2
Scale Y*2
Rotate


Lod Properties
Columns 1
Rows 1
Render All Lods
Colorize Lods
Lod Distance
Render at Distance
<<
>>
Auto Generate
Remove Lod


Phys
Copy selected Ctrl+D
Delete selected Delete
Delete all Ctrl+Delete
Create from Mesh Ctrl+M
Add Box Ctrl+B
Add Ball Ctrl+S
Add Capsule Ctrl+C
Add Tube Ctrl+T
Density += 0.1 .
Density += 1.0 >
Density -= 0.1 ,
Density -= 1.0 <

Programación en C++. Un poco de teoría-práctica

Algunas extensiones utilizadas en los proyectos VC++:

(.lib) Librerías de código objeto. Aquí se encuentran las funciones precompiladas.
(.h) Archivos de inclusión. Son archivos de código fuente (igual que .cpp) que pueden ser utilizados en distintos proyectos. Por ejemplo: código fuente para declarar funciones de un motor de juego (game engine), particiones del código fuente que hace el programador para agrupar procesos del juego (configurar entorno, variables y funciones relacionadas con el teclado, variables y funciones relacionadas con la cámara del juego, etc...)
(.cpp) Archivo de código fuente de C++. Son archivos de texto (se pueden abrir con el Bloc de Notas de windows) donde se guarda el código de programación del software o juego.

Después de instalar y configurar el motor de juego (game engine) en la IDE, para utilizarlo en un proyecto, en este caso, sólo es necesario incluir el archivo de inclusión (.h) del motor ('stdafx.h') con la orden o comando '#include' (es decir, incluir) al comienzo del archivo (.cpp), que lo que hace coloquialmente hablando, es insertar el texto (código fuente) del archivo (.h) en el archivo (.cpp) donde se encuentra.

Esta sería la primera línea del archivo (.cpp) que recoge el código fuente del programa:

```
// Esto es un comentario
#include "stdafx.h"
```

El color verde indica que es un comentario. El texto va precedido por // o entre /* ... */
El color azul indica que es una palabra clave (reservada) o comando de C++.
El color rojo indica que es una cadena de texto o carácter. Las cadenas de texto van encerradas entre comillas (“ ”) , y una letra o carácter va encerrado entre comillas simples (' ').
El color negro indica que es un término no incluido en los apartados anteriores. Suelen ser funciones o declaraciones del motor (engine), nombres de variables, operadores, ...

Las palabras claves precedidas por el símbolo # indica que se ejecutan durante el PRECOMPILADO, es decir, antes de compilar el código fuente del archivo (.cpp) principal.

El archivo 'stdafx.h' incluye las declaraciones de los términos (clases, funciones, estructura de datos, ...) utilizados por el motor (engine) para que puedan ser utilizadas posteriormente.

Este motor agrupa el código del juego en 5 funciones “generales”:

void InitPre () {}	Acciones a llevar a cabo antes de Iniciar el motor.
Bool Init () {}	“ ... después de Iniciar el motor.
void Shut () {}	“ ... al cerrar el motor.
Bool Main () {}	Principal. Actualización.
void Draw () {}	Principal. Dibujo.

Ejemplo incluido en el motor, Start.cpp:

```
#include "stdafx.h"

// Aquí se declaran las variables,objetos y otras funciones del juego

void InitPre() // Iniciar antes del inicio del motor o engine
{
  // Aquí tendrás que establecer la configuración inicial del motor/engine
  // como el nombre de la aplicación, opciones, resolución de pantalla,...

  App.name="Start";           // Título de la ventana
  App.icon=(Char*)IDI_ICON1;    // Establecer el icono de la ventana
  App.flag=APP_NO_FX;          // Establecer opción APP_NO_FX le dice al motor/engine que sólo cargue los métodos básicos de dibujo (no 3D ni Efectos Especiales)
  PakAdd("../data/engine.pak"); // Cargar archivo de datos(recursos) .pak del juego, específico de ESENTHEL
}
/*****************************************************************************/
Bool Init() // Iniciar después del incio del motor/engine
{
  // Desde quí, cuando el engine esté funcionando puedes cargar los datos/recursos del juego
  // devuelve 'false' (falso) cuando ocurre un error (return false)
  return true;
}
/*****************************************************************************/
void Shut() // Cerrar y salir
{
  // Esta sección recoge las acciones a llevar a cabo cuando el motor/engine está cerrando la aplicación.
}
/*****************************************************************************/
Bool Main() // Principal. Actualización
{
  // Aquí tendrás que tratar lo que ocurre en cada frame(escena) actualizada. Ejemplo: Teclado, IA del juego, ...

  if(Kb.bp(KB_ESC))return false; // Si la tecla 'Escape' es presionada SALIR. Devuelve falso (false)
  return true;                // Continuar. Devuelve verdad (true)
}
/*****************************************************************************/
void Draw() // Principal. Dibujo
{
  // Aquí tendrás que indicar al motor/engine qué dibujar en pantalla. También recoge condicionantes, por ejemplo dibujar una imagen o efecto si el cursor está encima o si es pulsada una tecla, ...

  D.clear(TURQ);                                    // Limpiar/borrar pantalla con el color tuquesa
  D.text (0, 0.1,"Este es el primer Tutorial");          // Dibujar texto en la posición (0, 0.1)
  D.text (0,-0.1,"Susitituir el CPP con algún otro archivo para ver un Tutorial diferente"); // Dibujar texto en la posición (0,-0.1)
}
/*****************************************************************************/
```

Algunas funciones “generales” van precedidas con las palabra clave 'void' (es decir, vacío) que indican que dicha función no devuelve (en inglés 'return') ningún valor.

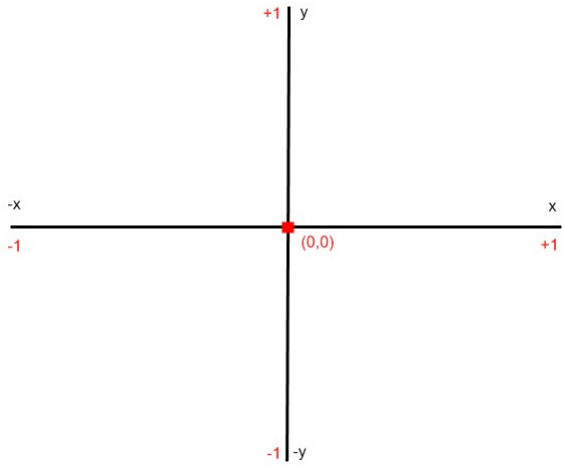
En cambio otras van precedidas de 'Bool' en color negro, que es diferente a la palabra clave 'bool'. (C++ es sensible a las mayúsculas y minúsculas). Por otra parte 'Bool' es un tipo de dato específico del motor/engine, por eso no aparece en azul representando una palabra clave.

Existen dos funciones indispensables para que el motor funcione. Son:

```
App.name="Nombre de la Aplicación. Servirá de título para la ventana";
PakAdd("../data/engine.pak"); //Archivo donde se encuentran los recursos del engine
```

Si no se utilizan estas dos funciones la aplicación no funcionará.

Antes de continuar voy a comentar el sistema de coordenadas 2D y 3D que utiliza ESENTHEL. Los ejes de coordenadas 2D (Vec2) serán utilizadas cuando creamos/manipulamos imágenes, texto o ventanas GUI.



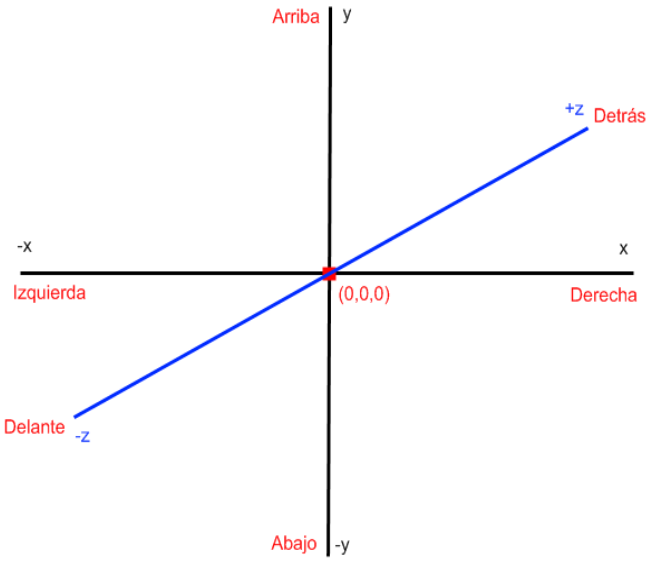
Son coordenadas relativas, donde el punto central de la ventana o pantalla será (0,0), y los extremos serán los valores -1,+1.

En una resolución de pantalla de (ResX, ResY), la posición (PosX, PosY) en este sistema de coordenadas será:

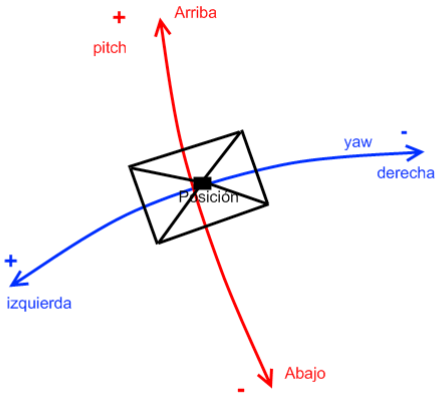
$$\frac{ResX-(2\times PosX)}{-ResX} \qquad \frac{ResY-(2\times PosY)}{-ResY}$$

Ejemplo, en una resolución de pantalla de (800,600) si queremos dibujar un texto o imagen en la posición (105,430) su eje de coordenadas será (-0.737, 0.433). En la Posición podemos utilizar una precisión de 3 dígitos.

Esta sería una imagen explicativa del eje de coordenadas 3D en ESENTHEL. (imagen de la derecha).



Y esta otra de abajo, detalla los diferentes parámetros de la cámara.



Los ángulos que maneja el motor son radianes, NO grados. También manejan radianes las fórmulas trigonométricas (cos, sen, tan, ...). Si queremos girar la cámara 90 grados debemos convertilos a radianes primero (Sería 1.57) mediante

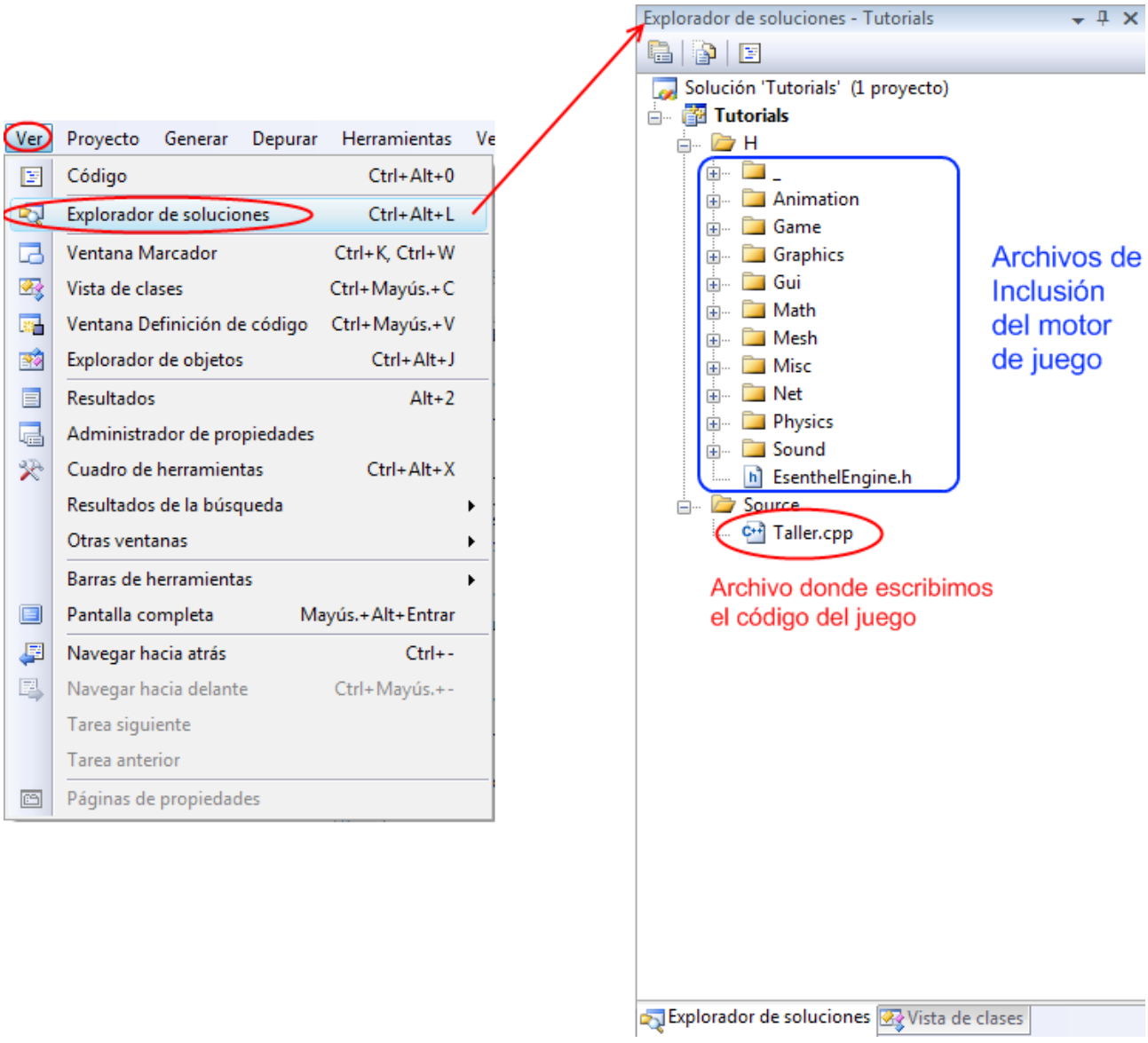
$$Radianes=\frac{Grados\times \Pi}{180} \qquad Grados=\frac{Radianes\times 180}{\Pi}$$

Comenzando a utilizar un motor de juego

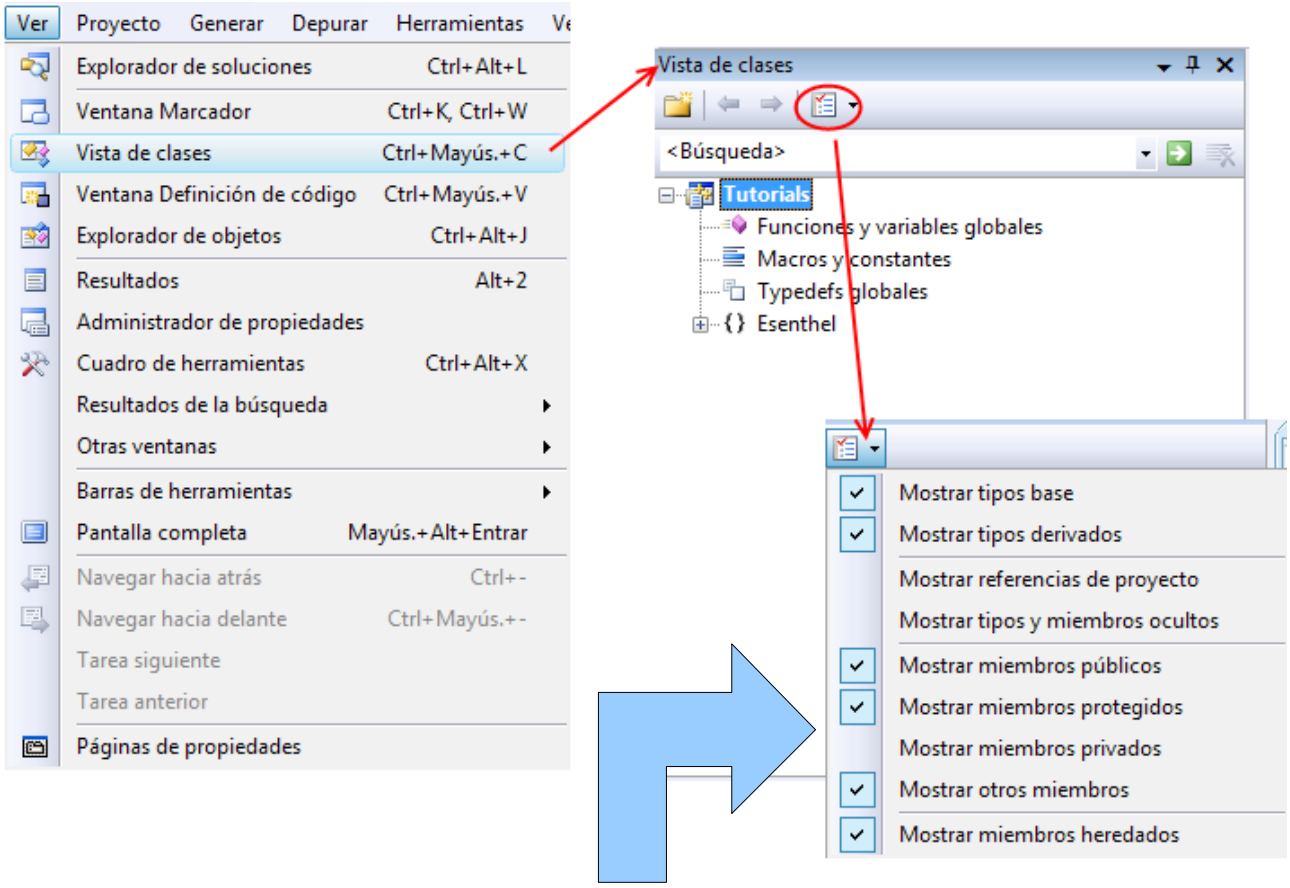
Al utilizar por primera vez un motor de juego, necesitaremos un período de adaptación hasta familiarizarnos con las palabras reservadas que utiliza dicho motor.

Las palabras reservadas del motor están compuestas por el nombre de clases y funciones, macros (creadas con #define) , enumeraciones (creadas con enum) y tipos definidos (creados con typedef).

Esta sería la vista normal del IDE:

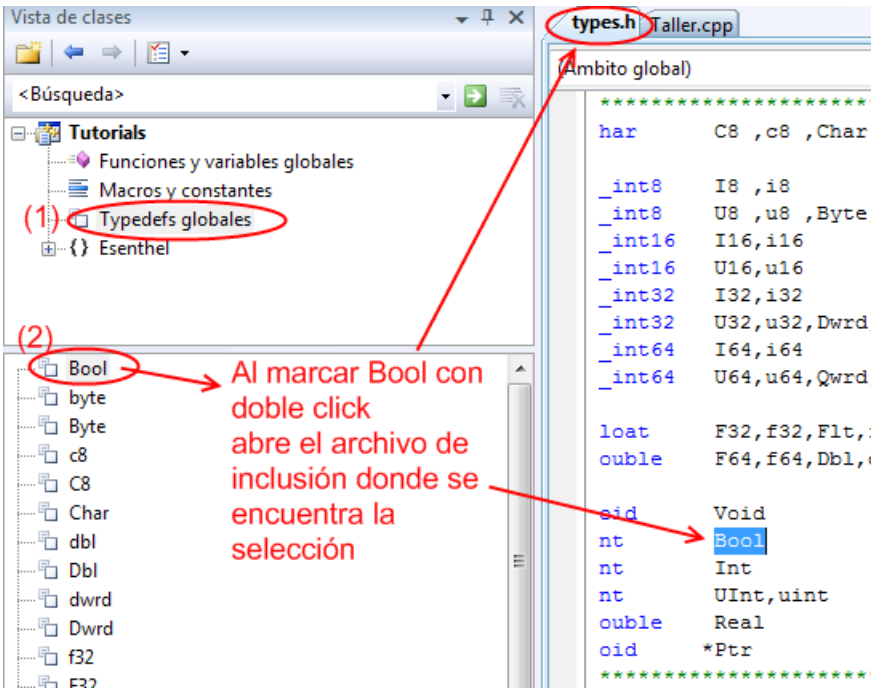


Pero existe una vista mejor para localizar esas palabras reservadas. La Vista de clases.

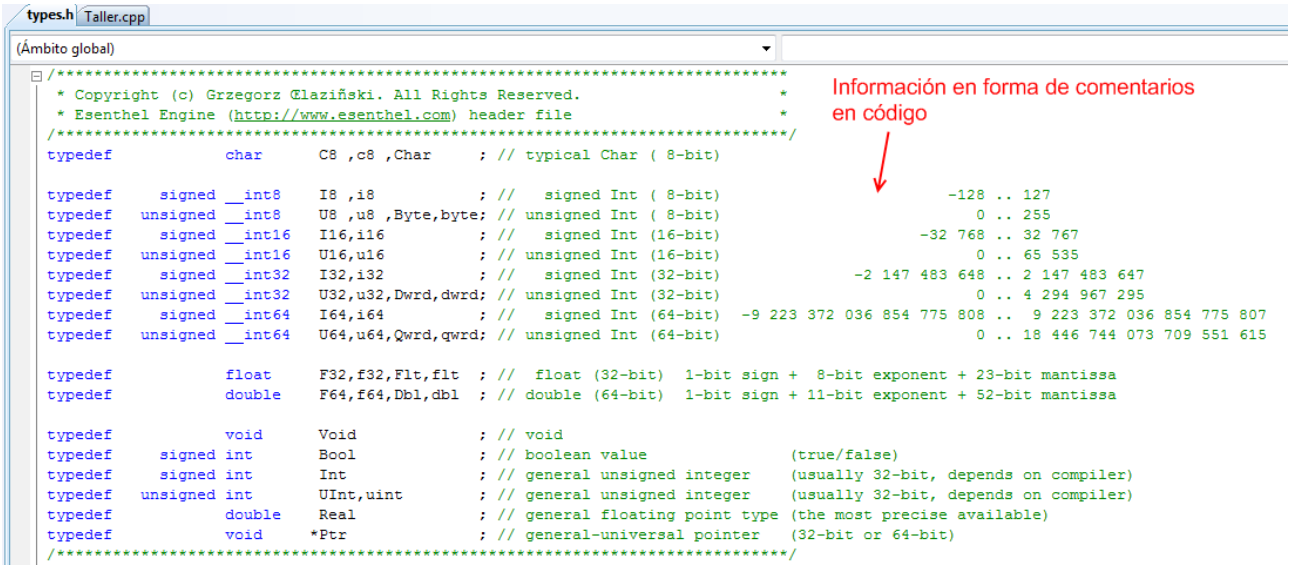


La imagen superior muestra la configuración mínima de la ventana 'Configuración vistas de clases' (la que muestra opciones marcadas con aspas), puedes marcarlas todas si quieres.

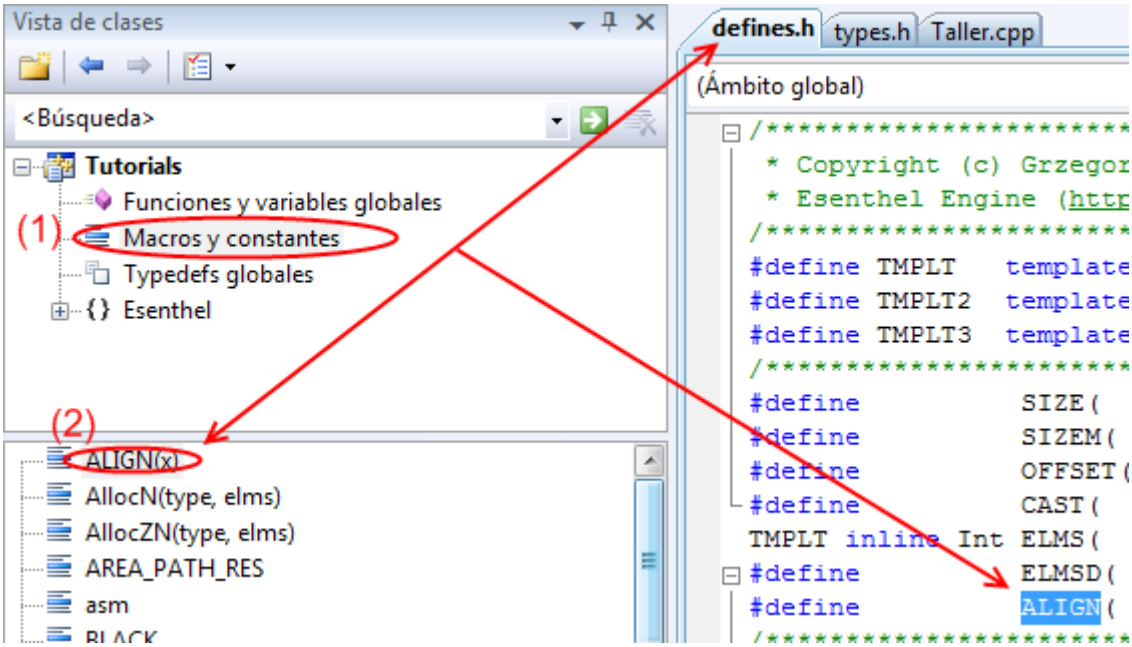
Comenzaremos por los tipos definidos en el motor:



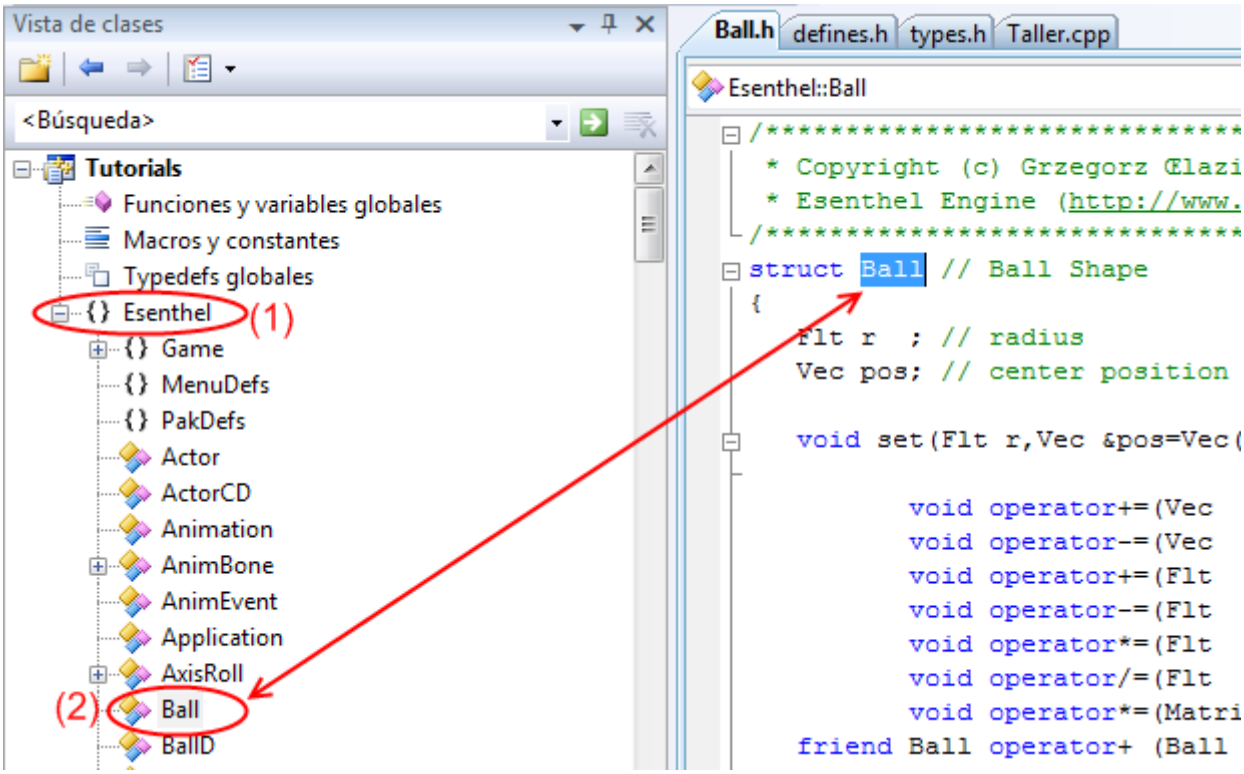
En el archivo de inclusión correspondiente encontraremos información:



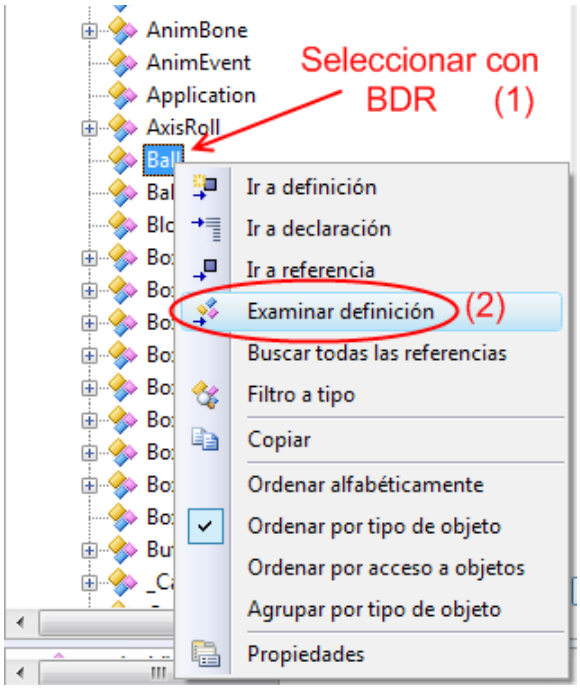
Seguimos con las macros definidas con #define:



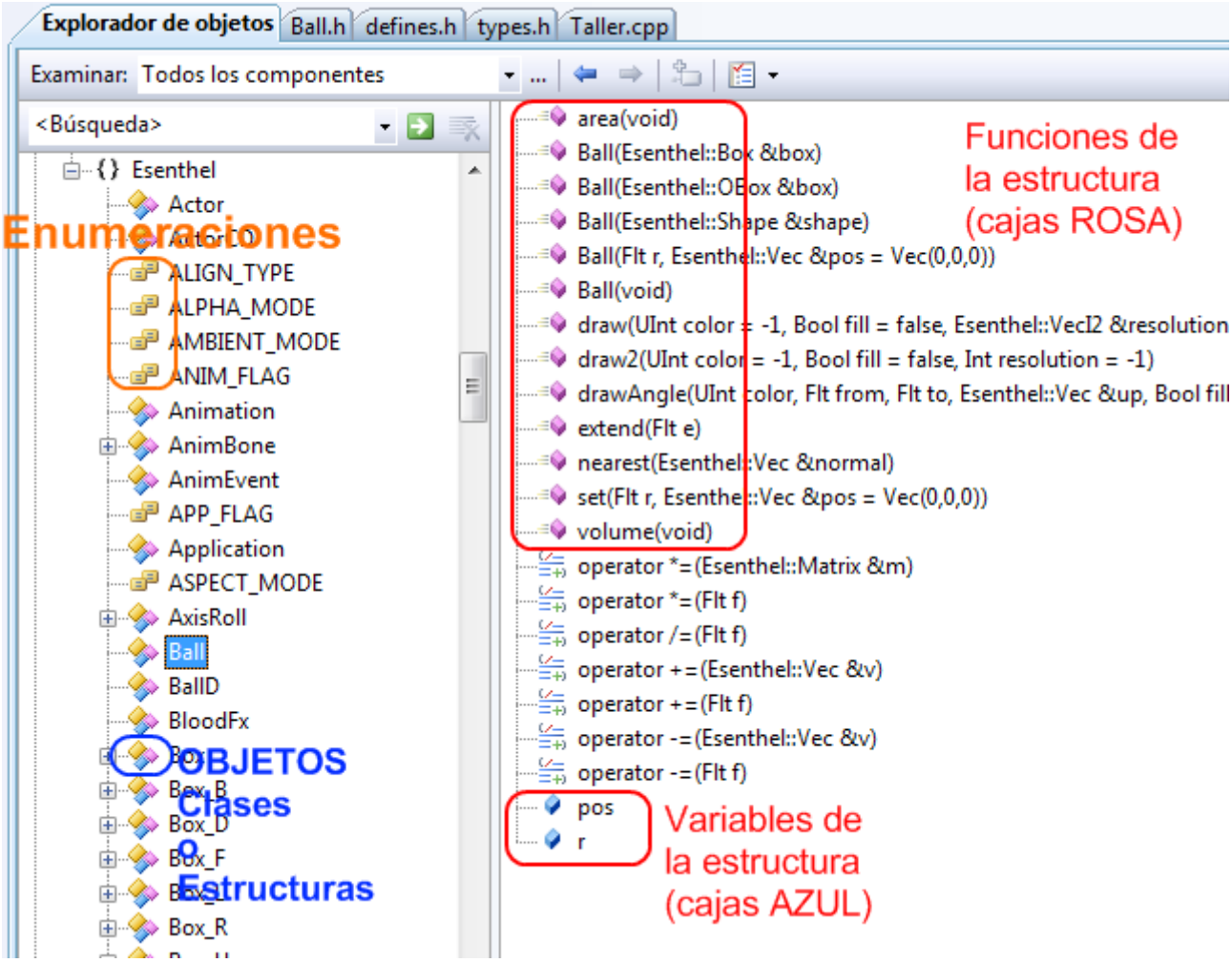
Seguimos con las funciones que, en este caso, se encuentran en **{} Esenthel**. Las funciones no se muestran directamente ya que se encuentran dentro de clases o estructuras. Los nombres de las clases y estructuras guardan relación con la labor que desempeñan en el juego. Por ejemplo, si queremos crear un objeto 3D tipo esfera, primero debemos saber su traducción inglesa, en este caso Ball. Después buscamos una clase o estructura con ese nombre y con el doble click accedemos al archivo de inclusión donde se encuentra para leer el comentario en el código adyacente.



En este caso vemos que se trata de una estructura (declarada con struct). Ahora debemos encontrar la función que muestra en pantalla la esfera (Ball). Para esto, hacemos lo que muestra la imagen siguiente.



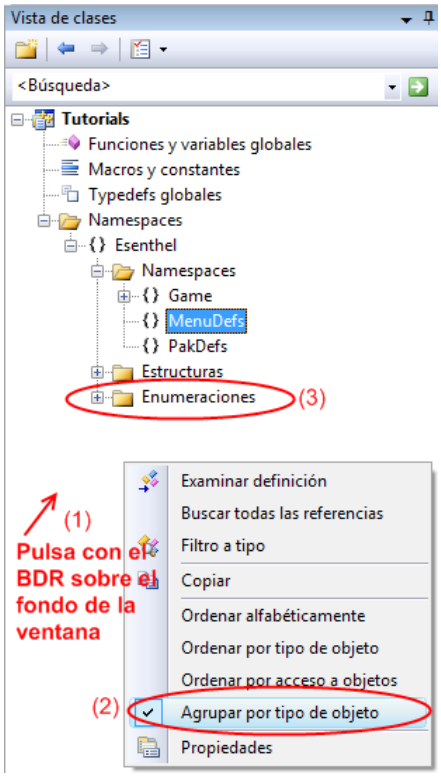
Y aparecerá la imagen siguiente:



De este modo podemos comparar mejor cada nombre de las distintas funciones. Si hacemos doble click sobre algún elemento, automáticamente accedemos a la posición que ocupa en el archivo de inclusión y así poder el comentario adyacente a la declaración de la función o variable.

Finalmente, las *enumeraciones* son un tipo especial de variables que tienen la propiedad de que su rango de valores es un conjunto de constantes enteras. Dicho en otras palabras, es un listado o enumeración de "palabras que llevan un orden" (por ejemplo días de la semana o meses del año). Se declara esa lista o enumeración con la instrucción enum y a la primera "palabra" de esa lista se le asigna la primera posición (el cero) y así sucesivamente. Nosotros en el código de programación utilizaremos esa "palabra" mientras que para el IDE (que entiende números) utilizará el orden que ocupa en esa lista.

Realizando el proceso que muestra la imagen de la derecha para acceder a las diferentes enumeraciones del motor.



Creación de Estructuras

Para desarrollar un juego necesitamos un motor de juego que nos aportará clases, estructuras y funciones para manipular los diferentes recursos del juego (modelos 3d, imágenes, sprites, sonido, ...).

Y además, "nosotros" deberemos desarrollar "nuevas" funciones para desarrollar los mecanismos del juego (fin de turno, siguiente nivel, ...).

Naturalmente, estos mecanismo podemos desarrollarlos solamente con funciones sin necesidad de crear nuevas "clases" o "estructuras" (si el juego no es muy complicado).

Pero en juegos más complejos donde se desarrolla un argumento en el cual aparecen muchos personajes o acciones, el número de variables y complejidad de dichas funciones se elevaría demasiado.

Por esta razón utilizaremos estructuras (similar a las clases) que engloba las variables y funciones que manipulan esas variables (llamadas métodos de la estructura) de un determinado objeto del juego (el jugador, enemigotipo1, enmigotipo2, accesotipo1, ...).

En la creación de la estructura no crearé distinciones entre publico (public:) y privado (private:), y será todo público así que no tenemos que especificar nada, ya que de forma predeterminada la estructura es pública. Forma general:

```
struct  NombreEstructura {


TipoVariable Variable1;  // Se declara variables SIN ASIGNAR VALORES
....
TipoVariable Variable...;

void NombreEstructura::crear () {  // La estructura debe constar de una FUNCION CONSTRUCTORA
Variable1=Valor1;                  // a la que he llamado 'crear'. Aquí asignamos los valores iniciales
...                                // de la estructura.
Variable...=Valor...;
}

TvRet NombreEstructura::funcion1 (Tv Par1, ..., Tv Par...) {  // Funciones o METODOS de la estructura
...
}
                              .............

TvRet NombreEstructura::funcion... (Tv Par1, ..., Tv Par...) {
...
}

} CopiaEstructura1, CopiaEstructura2, CopiaEstructura... ;  // Al final de la estructura podemos crear las
copias de la estructura a la que podemos dar nombre propios (p.e. seat600, seatCoupe,...) o una matriz (por ejemplo
orco[10] habrá creado 10 copias: orco[0], orco[1],...., orco[9])
```

No es obligatorio realizar la copia de la estructura al final de la declaración de esta. Podemos realizar copias en otras partes del programa utilizando al nombre de la estructura como si de un tipo de variable se tratase:

NombreEstructura *Copia1*, *Copia2*, ... ;

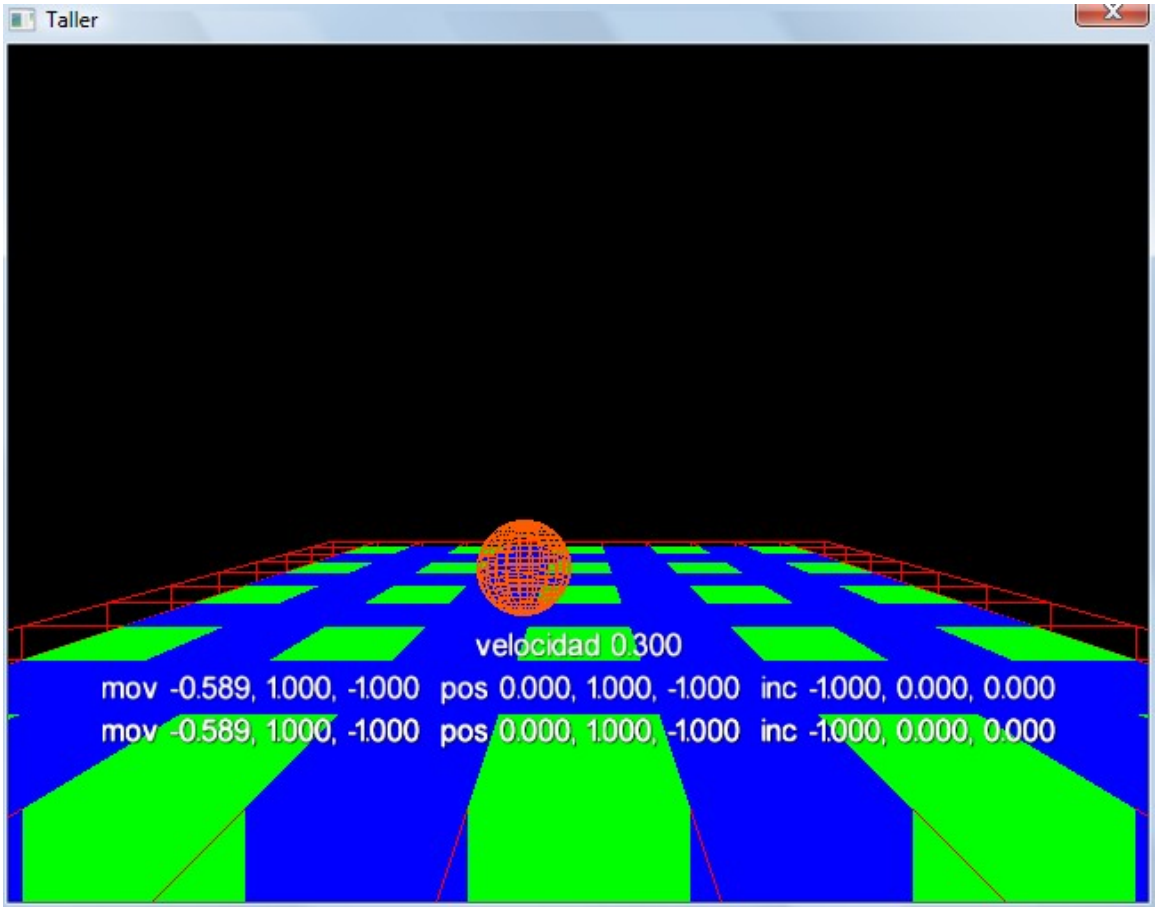
En la creación del código del juego, comenzaremos configurando las variables y funciones básicas sin prestar demasiada atención al aspecto gráfico. Una vez que consigamos que el código funcione iremos sustituyendo o añadiendo elementos, realizando compilaciones para comprobar que todo encaja correctamente. Por ejemplo, este sería el primer código:

```
/*****************************************************************************/
#include "stdafx.h"
/*****************************************************************************/
// Fuciones y variables anexas por el usuario
Flt distancia(Vec pos1,Vec pos2) // Funcion que calcula la distancia entre dos puntos 3D
{
Flt xc=pos1.x,yc=pos1.y,zc=pos1.z,xp=pos2.x,yp=pos2.y,zp=pos2.z;
return Sqrt(Sqr(xc-xp)+Sqr(yc-yp)+Sqr(zc-zp));
}

struct esf {          // Estructura esfera (esf)

        Vec posicion;   // Vec. Vector 3D (x,y,z) guardara la posicion de la esfera
        Vec incremento;  // Guarda el tipo de movimiento en curso
        Vec mov;                    // Situación de la esfera cuando se encuentra en movimiento
        Flt radio;                  // Variable que guarda el radio de la esfera. Asignando el
        Flt velocidad;      // Variable para regular la velocidad de movimiento
        Byte opc;                   // Variable opcion en curso.

void esf::dibujar () {

D.text(0,-0.6,S+"mov "+mov+"  pos "+posicion+"  inc "+incremento);  // PUNTO DE CONTROL (Si opc=0)

if (opc==0) {Ball(radio,posicion).draw(ColorHue(Tm.time()/3)); return;}

D.text(0,-0.4,S+"velocidad "+velocidad);

mov=Vec(mov.x+(incremento.x*Tm.d*velocidad),mov.y+(incremento.y*Tm.d*velocidad),mov.z+
(incremento.z*Tm.d*velocidad));

D.text(0,-0.5,S+"mov "+mov+"  pos "+posicion+"  inc "+incremento);  // PUNTO DE CONTROL (en movimiento)
// Si la distantcia entre el punto de origen y el punto actual(mov) es mayor a la distancia entre el punto de
// origen corregido con el incremento (Ha sobrepasado el destino). Ha finalizado el movimiento (opc=0)
if
((distancia(posicion,mov))>=(distancia(posicion,Vec(posicion.x+incremento.x,posicion.y+incremento.y,posicion.z+incr
emento.z))))
{posicion=Vec(posicion.x+incremento.x,posicion.y+incremento.y,posicion.z+incremento.z); mov=posicion; opc=0;}

Ball(radio,mov).draw(ColorHue(Tm.time()/3));

D.text(0,-0.6,S+"mov "+mov+"  pos "+posicion+"  inc "+incremento);  // PUNTO DE CONTROL (después de if)

return;
}
        void esf::crear (Vec pos, Flt rad=0.5f, Flt vel=0.3f) {
                posicion=pos;
                radio=rad;
                velocidad=vel;
```

```
		opc=0;
		incremento=Vec(0,0,0);
		mov=posicion;
		dibujar();   // Mostrar en pantalla
	}
	void esf::actualizar() {	// Aplica acción necesaria según la tecla pulsada

		if (Kb.br(KB_NPSUB)) if (velocidad>0.1f) {velocidad-=0.1f;}  // -
		if (Kb.br(KB_NPADD)) if (velocidad<1.5f) {velocidad+=0.1f;}		// +

		if (opc>0) {return;}   // Si se no ha terminado una accion anterior volver

		if (Kb.b(KB_UP))  {opc=5;}
		if (Kb.b(KB_DOWN))  {opc=6;}
		if (Kb.b(KB_LEFT))  {opc=7;}
		if (Kb.b(KB_RIGHT))  {opc=8;}
		if ((Kb.b(KB_UP)) && (Kb.b(KB_LEFT)))  {opc=1;}
		if ((Kb.b(KB_UP)) && (Kb.b(KB_RIGHT)))  {opc=2;}
		if ((Kb.b(KB_DOWN)) && (Kb.b(KB_LEFT)))  {opc=3;}
		if ((Kb.b(KB_DOWN)) && (Kb.b(KB_RIGHT)))  {opc=4;}

		if (opc==0) {return;}  // Si no se ha pulsado ninguna tecla anterior volver

		switch (opc) {
			case 1 : {incremento=Vec(-1,0,1);} break;
			case 2 : {incremento=Vec(1,0,1);} break;
			case 3 : {incremento=Vec(-1,0,-1);} break;
			case 4 : {incremento=Vec(+1,0,-1);} break;
			case 5 : {incremento=Vec(0,0,1);} break;
			case 6 : {incremento=Vec(0,0,-1);} break;
			case 7 : {incremento=Vec(-1,0,0);} break;
			case 8 : {incremento=Vec(1,0,0);}
		}

		return;
	}
} esfera;  // Copia esfera. Será el termino a utilizar en el codigo de programa.
/*****************************************************************************/
void InitPre()
{
  App.name="Taller";
  App.flag=APP_NO_FX;
  PakAdd("../data/engine.pak");
}
/*****************************************************************************/
Bool Init()
{	// Configurar posición cámara.
	Cam.matrix.pos+=Cam.matrix.y*2; Cam.matrix.pos+=Cam.matrix.z*-6;
	// Al modificar los parámetros de la cámara hay que actualizar cámara
	Cam.setAngle(Cam.matrix.pos,Cam.yaw,Cam.pitch,Cam.roll).set();
	//Constructor de esfera
	esfera.crear(Vec(0,1,0),0.5f,0.3f);

  return true;
}
/*****************************************************************************/
void Shut()
{

}
/*****************************************************************************/
Bool Main()
{
```

```
        if(Kb.bp(KB_ESC))return false;
        esfera.actualizar();  //actualizar. funcion que se encarga del teclado
        return true;
}
/*****************************************************************************/
void Draw()
{
        D.clear(BLACK);  // Esta debe ser la primera instrucción, porque es la que borra/limpia

        for (int i=-5;i<6;i++){
        Box(0.5,Vec(i,0,5)).draw(RED,false); //opción de dibujo false. Dibujo modo de alambre
        Box(0.5,Vec(i,0,-5)).draw(RED);              // opción por defecto
        Box(0.5,Vec(-5,0,i)).draw(RED);
        Box(0.5,Vec(5,0,i)).draw(RED);
        }
        for (int i=4;i>-5;i--){
                for (int ii=-4;ii<5;ii++){   // Si ii y i son pares dibujar verde
                        if (((ii%2)==0)&&((i%2)==0)) {Box(0.5,Vec(ii,0,i)).draw(GREEN,true);}
                        else {Box(0.5,Vec(ii,0,i)).draw(BLUE,true);}
                }                                   //La opcion de dibujo true, dibuja un cuerpo sólido
        }

        esfera.dibujar(); //dibujar. funcion que se encarga del dibujo
}
/*****************************************************************************/
```

Imagen del código anterior:



Las funciones “del motor” que he utilizado son:

Teclado:
Kb.b: Botón/tecla activo (actualmente)
Kb.bp: Botón/tecla ha sido pulsado(aunque no sea en este momento).
Kb.br: Botón/tecla soltada (para ejecutar una instrucción sólo una vez)

Escritura:
D.text: Escribe texto en pantalla. Si debemos escribir texto junto a datos numéricos, encabezamos la línea con 'S+'.

Dibujo:
Ball.draw: Función de la estructura "del motor" Ball que dibuja una esfera.
Box.draw: Función de la estructura Box que dibuja un cubo.

Cámara:
Cam.matrix.pos: No es una función, es una variable de la cámara que guarda la posición de esta. Es peculiar a la hora de modificarla. Fíjate en el código.
Cam.set(): Establece los cambios realizados en los parámetros de la cámara.

Para ajustar la velocidad de movimiento el motor incorpora una "variable" llamada Tm.d (Time delta). Es un factor tiempo, que varía según la velocidad de cada computadora o los procesos activos en ese momento. Sustituye la línea de código siguiente, que se encuentra en la función dibujar de la estructura que hemos creado:

```
mov=Vec(mov.x+(incremento.x*Tm.d*velocidad),mov.y+(incremento.y*Tm.d*velocidad),mov.z+
(incremento.z*Tm.d*velocidad));
```

por esta otra, donde he eliminado la variable Tm.d para comprobar el tiempo de ejecución del programa:

```
mov=Vec(mov.x+(incremento.x*velocidad),mov.y+(incremento.y*velocidad),mov.z+(incremento.z*velocidad));
```

Al ejecutarlo comprobareis la velocidad de procesado de c++. Una pulsación que no llega al segundo equivale a un desplazamiento de 40-60.

Continuemos. Si queremos crear una malla y aplicarle una textura deberemos crear un objeto del tipo 'Mesh' y otro tipo 'Material' al inicio del programa, para que sea global y pueda ser utilizadas en las distintas funciones generales del motor.

```
/*****************************************************************************/
#include "stdafx.h"
/*****************************************************************************/
// Variables Globales
Mesh suelo;                          // Objeto Mesh. Guarda vértices, aristas, caras, normales
Material *tsuelo;  // textura del suelo
```

Debemos quitar el distintivo APP_NO_FX, al utilizar mallas 3d. La función general void InitPre quedaría así:

```
void InitPre()
{
  App.name="Taller";
  PakAdd("../data/engine.pak");
}
```

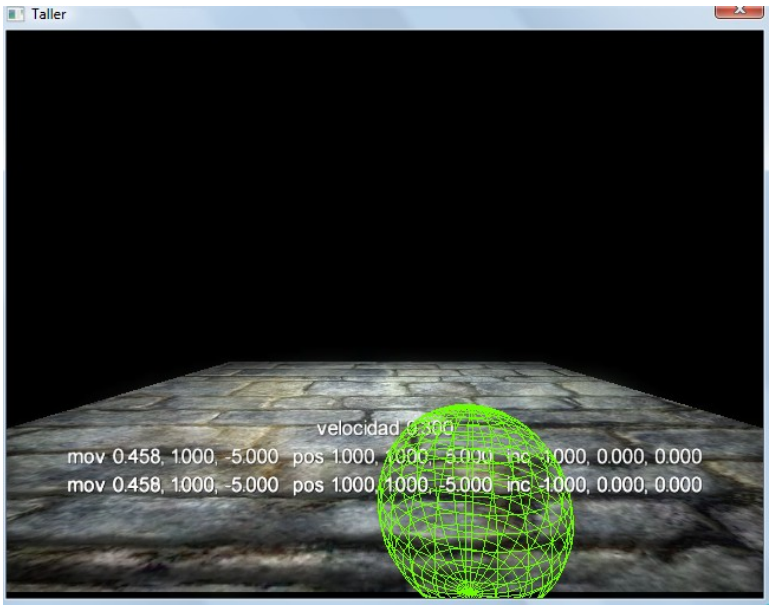
La creación de la malla se desarrolla en la función general 'Bool Init'.

```
Bool Init()
{           // Configurar posición cámara.
            Cam.matrix.pos+=Cam.matrix.y*2; Cam.matrix.pos+=Cam.matrix.z*-6;
            // Al modificar los parámetros de la cámara hay que actualizar cámara
            Cam.setAngle(Cam.matrix.pos,Cam.yaw,Cam.pitch,Cam.roll).set();
            //Constructor de esfera
            esfera.crear(Vec(0,1,0),0.5f,0.3f);
            tsuelo=Materials("../data/mtrl/brick/0.mtrl");  // Asignar material a tsuelo
            suelo.create(1).B(0).create(Box(10,1,10,Vec(0,0,0)),VTX_TX0|VTX_NRM|VTX_TNG); //Crear malla
            suelo.setMaterial(tsuelo).setRender().setBox();  // Aplicar material a la malla
  return true;
}
```

En la función general void Draw se emplea otro sistema de dibujo. Ya no es necesario aplicar la instrucción D.Clear para limpiar/borrar pantalla. El motor utiliza un sistema o método de renderizado llamado Renderer (Render); donde Render es una función (declarada antes de void Draw) que maneja los diferentes método de renderizado (sólido, luz,...).

```
void Render() // Método de Renderizado
{
  switch(Renderer()) // El método de renderizado sera llamado con diferentes modos
  {
    // En el modo de "renderizado solido" se añaden la instrucciones de dibujo de
            // las mallas del programa
    case RM_SOLID:
      suelo.draw();
    break;

    // Desde el modo "renderizado de luz" añadimos las instrucciones
            // relacionadas con el renderizado de luz
    case RM_LIGHT:
      LightPoint(40,Vec(0,5,0)).add();  // 40 alcance, con Vec asignamos una posición.
    break;
  }
}


void Draw()
{
            Renderer(Render);   // Llamada a Método de Renderizado
            esfera.dibujar();       //dibujar. funcion que se encarga del dibujo
}
```

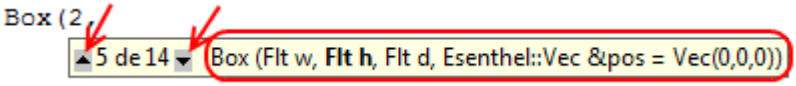
Este sería el resultado.



La función que he utilizado para crear la malla(sin textura) ha sido:

suelo.create(1).B(0).create(Box(10,1,10,Vec(0,0,0)),VTX_TX0|VTX_NRM|VTX_TNG);

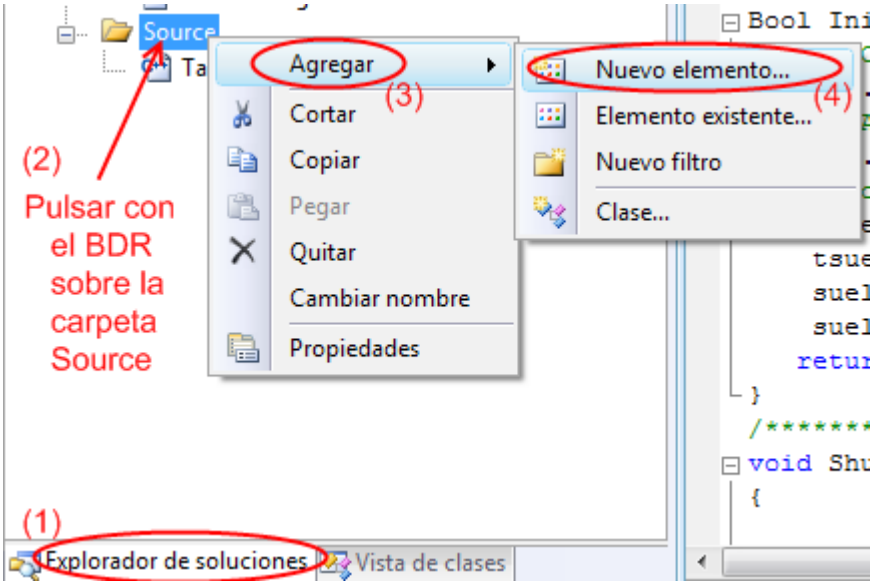
Para entender el funcionamiento de esta instrucción debemos buscar entre los diferentes tutoriales .cpp de motor y leer los comentarios adyacentes. Una vez determinada la función que necesitamos, escribimos su nombre en el IDE de VC++ y al escribir el primer paréntesis aparece un mensaje de ayuda indicando las diferentes opciones de la función (la función está sobrecargada, se han creado diferentes funciones con el mismo nombre, pero diferentes parámetros). Desde este mensaje podemos buscar los parámetros de la función que mejor se adapten a nuestras necesidades.

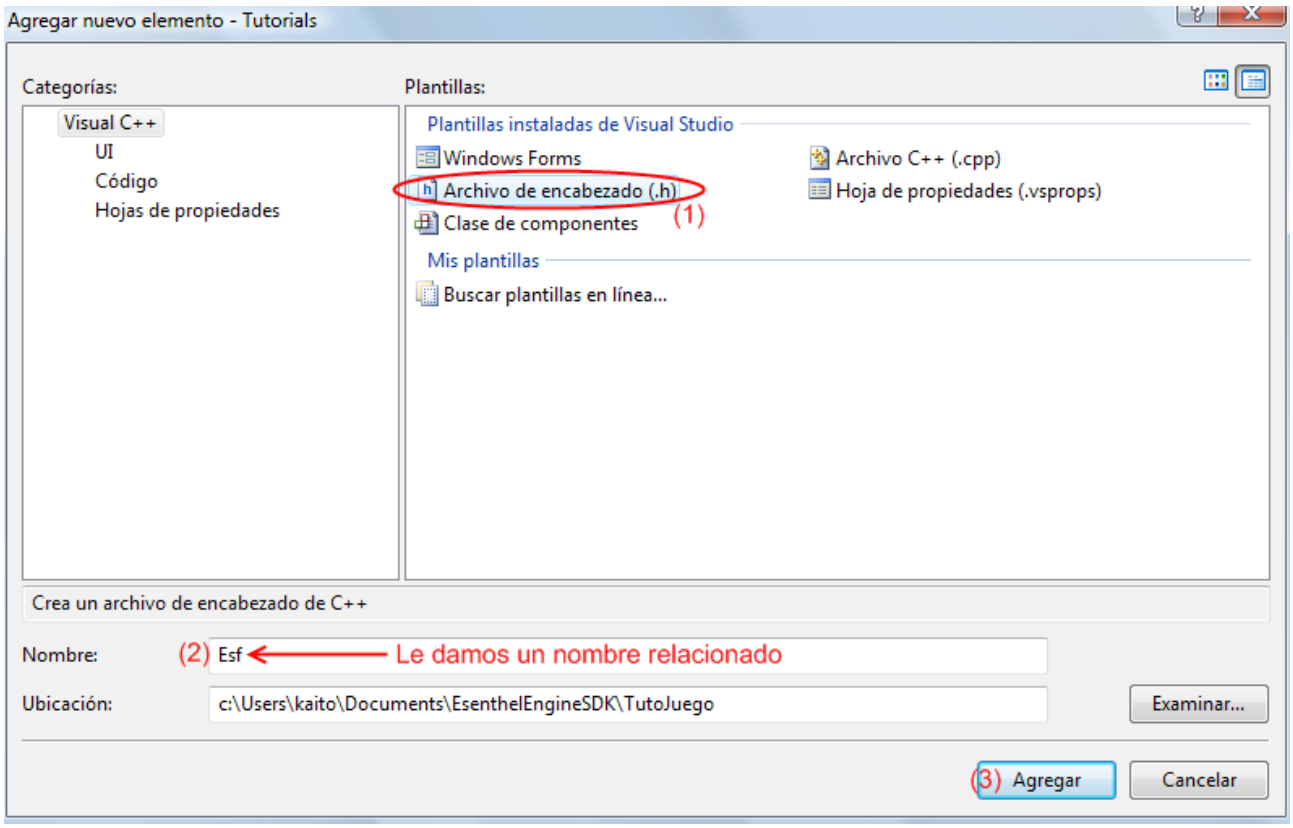


Finalmente seleccione la forma donde se detalla w(de width, ancho (eje x)), h (de height, altura (eje y)), y d (de depth, es decir, profundidad (eje z)), y un vector de posicion. Mirar también archivo de inclusión Box.h.

Voy a concluir explicando como guardar la estructura y la función creadas durante esta demostración para ser utilizadas posteriormente en otros proyectos. Este proceso consiste en guardar las estructuras y funciones creadas en un archivo de inclusión que podrá ser utilizado con la instrucción #include.

Comenzamos creando un archivo de inclusión como muestra la imagen:





Dentro del nuevo archivo creado Esf.h moveríamos (es decir, cortar y pegar) la función y la estructura de la demostración. Este sería el resultado:

```
// Esf.h
//
//
Flt distancia(Vec pos1,Vec pos2) // Funcion que calcula la distancia entre dos puntos 3D
{
Flt xc=pos1.x,yc=pos1.y,zc=pos1.z,xp=pos2.x,yp=pos2.y,zp=pos2.z;
return Sqrt(Sqr(xc-xp)+Sqr(yc-yp)+Sqr(zc-zp));
}

// Como ejemplo (NO ES NECESARIO HACER ESTA MODIFICACION) voy a declarar y construir la
// estructura por separado.
// Declaración de la estructura
struct esf {          // Estructura esfera (esf)

        Vec posicion;   // Vec. Vector 3D (x,y,z) guardara la posicion de la esfera
        Vec incremento;  // Guarda el tipo de movimiento en curso
        Vec mov;                    // Situación de la esfera cuando se encuentra en movimiento
        Flt radio;                  // Variable que guarda el radio de la esfera. Asignando el
        Flt velocidad;      // Variable para regular la velocidad de movimiento
        Byte opc;                   // Variable opcion en curso.

        void esf::dibujar ();  // Dibujar esfera en pantalla
        void esf::crear (Vec pos, Flt rad, Flt vel);  // Método Constructor
        void esf::actualizar(); // Aplica acción necesaria según la tecla pulsada
};


// Construcción de la estructura
        void esf::dibujar () {

                D.text(0,-0.6,S+"mov "+mov+"  pos "+posicion+"  inc "+incremento);  // PUNTO DE CONTROL (Si
```

```
opc=0)

                if (opc==0) {Ball(radio,posicion).draw(ColorHue(Tm.time()/3)); return;}

                D.text(0,-0.4,S+"velocidad "+velocidad);

                mov=Vec(mov.x+(incremento.x*Tm.d*velocidad),mov.y+(incremento.y*Tm.d*velocidad),mov.z+
(incremento.z*Tm.d*velocidad));

                D.text(0,-0.5,S+"mov "+mov+"  pos "+posicion+"  inc "+incremento);  // PUNTO DE CONTROL (en
movimiento)
                // Si la distantcia entre el punto de origen y el punto actual(mov) es mayor a la distancia entre el punto
de
                // origen corregido con el incremento (Ha sobrepasado el destino). Ha finalizado el movimiento
(opc=0)
                if
((distancia(posicion,mov))>=(distancia(posicion,Vec(posicion.x+incremento.x,posicion.y+incremento.y,posicion.z+incr
emento.z))))
                {posicion=Vec(posicion.x+incremento.x,posicion.y+incremento.y,posicion.z+incremento.z);
mov=posicion; opc=0;}

                Ball(radio,mov).draw(ColorHue(Tm.time()/3));

                D.text(0,-0.6,S+"mov "+mov+"  pos "+posicion+"  inc "+incremento);  // PUNTO DE CONTROL
(después de if)

                return;
        }

        void esf::crear (Vec pos, Flt rad=0.5f, Flt vel=0.3f) {
                posicion=pos;
                radio=rad;
                velocidad=vel;
                opc=0;
                incremento=Vec(0,0,0);
                mov=posicion;
                dibujar();
        }

        void esf::actualizar() {      // Aplica acción necesaria según la tecla pulsada

                if (Kb.br(KB_NPSUB)) if (velocidad>0.1f) {velocidad-=0.1f;}  // -
                if (Kb.br(KB_NPADD)) if (velocidad<1.5f) {velocidad+=0.1f;}           // +

                if (opc>0) {return;}   // Si se no ha terminado una accion anterior volver

                if (Kb.b(KB_UP))  {opc=5;}
                if (Kb.b(KB_DOWN))  {opc=6;}
                if (Kb.b(KB_LEFT))  {opc=7;}
                if (Kb.b(KB_RIGHT))  {opc=8;}
                if ((Kb.b(KB_UP)) && (Kb.b(KB_LEFT)))  {opc=1;}
                if ((Kb.b(KB_UP)) && (Kb.b(KB_RIGHT)))  {opc=2;}
                if ((Kb.b(KB_DOWN)) && (Kb.b(KB_LEFT)))   {opc=3;}
                if ((Kb.b(KB_DOWN)) && (Kb.b(KB_RIGHT)))   {opc=4;}

                if (opc==0) {return;}  // Si no se ha pulsado ninguna tecla anterior volver

                switch (opc) {
                        case 1 : {incremento=Vec(-1,0,1);} break;
                        case 2 : {incremento=Vec(1,0,1);} break;
                        case 3 : {incremento=Vec(-1,0,-1);} break;
                        case 4 : {incremento=Vec(+1,0,-1);} break;
                        case 5 : {incremento=Vec(0,0,1);} break;
```

```
                        case 6 : {incremento=Vec(0,0,-1);} break;
                        case 7 : {incremento=Vec(-1,0,0);} break;
                        case 8 : {incremento=Vec(1,0,0);}
                }
                return;
        }
```

En el archivo principal debemos hacer dos modificaciones, una será incluir el archivo 'Esf.h' y otra será declarar una copia de la estructura 'esf' llamada esfera. Este sería el código:

```
// Taller.cpp  ARCHIVO PRINCIPAL
/*************************************************************************/
#include "stdafx.h"
#include "Esf.h"  // ***********(PRIMERA MODIFICACION)*************
/*************************************************************************/
// Variables y objetos Globales
Mesh suelo;                             // Objeto Mesh. Guarda vértices, aristas, caras, normales
Material *tsuelo;   // textura del suelo
esf esfera;                             // Objeto tipo esfera ********(SEGUNDA MODIFICACION)********
/*************************************************************************/
void InitPre()
{
  App.name="Taller";
  PakAdd("../data/engine.pak");
}
/*************************************************************************/
Bool Init()
{       // Configurar posición cámara.
        Cam.matrix.pos+=Cam.matrix.y*2; Cam.matrix.pos+=Cam.matrix.z*-6;
        // Al modificar los parámetros de la cámara hay que actualizar cámara
        Cam.setAngle(Cam.matrix.pos,Cam.yaw,Cam.pitch,Cam.roll).set();
        //Constructor de esfera
        esfera.crear(Vec(0,1,0),0.5f,0.3f);
        tsuelo=Materials("../data/mtrl/brick/0.mtrl");  // Asignar material a tsuelo
        suelo.create(1).B(0).create(Box(10,1,10,Vec(0,0,0)),VTX_TX0|VTX_NRM|VTX_TNG); //Crear malla
        suelo.setMaterial(tsuelo).setRender().setBox();  // Aplicar material a la malla
  return true;
}
/*************************************************************************/
void Shut()
{

}
/*************************************************************************/
Bool Main()
{
        if(Kb.bp(KB_ESC))return false;
        esfera.actualizar();  //actualizar. funcion que se encarga del teclado
        return true;
}
/*************************************************************************/
void Render() // Método de Renderizado
{
  switch(Renderer()) // El método de renderizado sera llamado con diferentes modos
  {
    // En el modo de "renderizado solido" se añaden la instrucciones de dibujo de
          // las mallas del programa
    case RM_SOLID:
      suelo.draw();
    break;
```

```
    // Desde el modo "renderizado de luz" añadimos las instrucciones
          // relacionadas con el renderizado de luz
    case RM_LIGHT:
      LightPoint(30,Vec(0,5,0)).add();  // 40 alcance, con Vec asignamos una posición.
    break;
  }
}


void Draw()
{
/*        D.clear(BLACK);  // Esta debe ser la primera instrucción, porque es la que borra/limpia

          for (int i=-5;i<6;i++){
          Box(0.5,Vec(i,0,5)).draw(RED,false); //opción de dibujo false. Dibujo modo de alambre
          Box(0.5,Vec(i,0,-5)).draw(RED);              // opción por defecto
          Box(0.5,Vec(-5,0,i)).draw(RED);
          Box(0.5,Vec(5,0,i)).draw(RED);
          }
          for (int i=4;i>-5;i--){
                    for (int ii=-4;ii<5;ii++){

                              if (((ii%2)==0)&&((i%2)==0)) {Box(0.5,Vec(ii,0,i)).draw(GREEN,true);}
                              else {Box(0.5,Vec(ii,0,i)).draw(BLUE,true);}
                    }
          }
*/
          Renderer(Render);
          esfera.dibujar(); //dibujar. funcion que se encarga del dibujo
}
/*****************************************************************************/
```

Espero que este documento haya sido de vuestro interés.

ANEXO Cámara

```
/
**********************************************************************************************
**********************************************************************************************
**********

Los diferentes modo de cámara que podremos configurar son:
Cam.setAngle: Al mismo tiempo que modificamos la posicion de la cámara (Cam.matrix.pos) variamos la posicion
enfocada o a la que mira (Look 'at') llamada Cam.at, si bien, dicha posicion (la de Cam.at)
junto a la distancia entre la cámara y la posicion enfocada (Cam.dist) no pueden ser modificadas directamente, sino que
es modificada al variar el resto de variables (Cam.yaw, Cam.pitch y Cam.roll).
Para enfocar a una posicion determinada desde este modo debemos calcular el Cam.yaw y el Cam.pitch necesarios.

Cam.setSpherical: Desde este modo movemos la camara alrededor de un punto fijo, girando alrededor del punto
enfocado cambiando los valores Cam.yaw y Cam.pitch. Si queremos desplazar horizontalmente
o verticalmente esta cámara, debemos mover la posicion enfocada (Cam.at).

Cam.setFromAt: Es parecida a la anterior, ya que enfoca una posicion fija (naturalmente podremos modificarla) pero
desde este modo podremos desplazar la cámara directamente modificando (Cam.matrix.pos)
sin cambiar a la posicion a la que mira o enfoca (Cam.at).

Cam.setPosDir: Desde este modo situamos la camara tomando como referencia la posicion del jugador (en este caso,
PosEsfera) situandola por encima y desplazando hacia atrás, de forma que siga en
todo momento los movimientos de dicho personaje. Utilizan un Vector dir por defecto de (0,1,0) y un Vector up de
(0,0,1).

/
**********************************************************************************************
**********************************************************************************************
**********/

#include "stdafx.h"

/
**********************************************************************************************
**********************************************************************************************
**********/

// Declaración de variables globales

Vec PosEsfera; // Vec. Variable 3D (x,y,z) donde guardaremos la posicion de la malla 3D (en este caso una esfera (ball))
Int iVar1=0,iVar2(0); // Variables de una valor entero utilizado para guardar los parámetros seleccionados
Flt tiempo,fVar; // Variable tipo float donde guardar el tiempo transcurrido en un momento determinado (tiempo)
// Otra variable tipo float donde guardaré los incrementos a aplicar sobre los parámetros (fVar)

/
**********************************************************************************************
**********************************************************************************************
**********/

// Funciones anexas al programa, creadas por el usuario

Flt Yaw(Vec &pos1,Vec &pos2) //Función que calcula la demora(Yaw) entre dos posiciones 3D, fijando como norte
(demora 0) el eje +Z
{
Flt xc=pos1.x,zc=pos1.z,xp=pos2.x,zp=pos2.z; // Variables Locales
```

```
if (xp==xc && zp>=zc) {return 0;} // Se encuentra justo al frente. Un giro completo de 360º equivale a una variación +6 (si es hacia la izquierda) o -6 (si es hacia la derecha) en el Cam.yaw
if (xp==xc && zp<zc) {return 3;} // Se encuentra justo detrás
if (zp==zc && xp>xc) {return -1.5;} // Se encuentra justo por el costado de estribor (derecho) o través de estribor. Es de signo negativo porque las variaciones en Cam.yaw a la derecha son negativas.
if (zp==zc && xp<xc) {return 1.5;} // Se encuentra justo por el costado de babor (izquierdo) o través de babor. Es de signo positivo porque las variaciones en Cam.yaw a la izquierda son positivas.
if (zp>zc) {return Atan((xc-xp)/(zp-zc));} // Se encuentra delante, por la amura de babor (será positivo) o estribor (será negativo)
if (xp>xc) {return ((Atan((zc-zp)/(xc-xp)))-1.5);} // Se encuentra detrás por la banda de estribor (derecho) o aleta de estribor
else {return ((Atan((zc-zp)/(xc-xp)))+1.5);} // Se encuentra detrás por la banda de babor (izquierdo) o aleta de babor
}

Flt Pitch(Vec &pos1,Vec &pos2) // Función que calcula el angulo de elevación (Pitch) entre dos posiciones 3D, fijando como nivel (elevación 0) el plano (x-z)o valor del eje 'y' de la posicion principal (pos1)
{
Flt xc=pos1.x,yc=pos1.y,zc=pos1.z,xp=pos2.x,yp=pos2.y,zp=pos2.z; // Variables Locales
if (yp==yc) {return 0;} // Se encuentran al mismo nivel o altura.
return Atan((yp-yc)/(Sqrt(Sqr(xp-xc)+Sqr(zp-zc)))); // Tan(A)=Co/Ca (Co=Cateto opuesto, Ca=Cateto adyacente)
} // Si la segunda posicion (pos2) se encuentra debajo de la posicion de referencia o principal (pos1) tendrá un valor negativo.

Flt Distance(Vec &pos1,Vec &pos2) // Función que calcula la distancia entre dos posiciones 3D
{
Flt xc=pos1.x,yc=pos1.y,zc=pos1.z,xp=pos2.x,yp=pos2.y,zp=pos2.z; // Variables Locales
return Sqrt(Sqr(xc-xp)+Sqr(yc-yp)+Sqr(zc-zp)); // h^2=C1^2+C^2 (la hipotenusa al cuadrado es igual a la suma de sus catetos al cuadrados (Teorema de Pitagoras)
}

/* Más información sobre TRIGONOMETRIA:
http://personal5.iddeo.es/ztt/For/F7_Triangulos.htm
http://descartes.cnice.mec.es/materiales_didacticos/Vectores3D_d3/vectores3D_03.htm */


/
**********/

void InitPre()
{
// Configuracion General
App.name="TallerCamara"; // Nombre de la aplicacion y titulo de la ventana
App.flag=APP_FULL_TOGGLE; // Permite redimensionar a pantalla completa pulsado Alt+ENTER. Para minimizar la ventana volver a pulsar Alt+ENTER
IOPath="../data/"; // Directorio por defecto(si no se detalla una ruta específica del archivo (imagen, malla,...) donde busca los recursos necesarios en la aplicación.
PakAdd("engine.pak"); // Archivo .pak (paquete de archivos y recursos) utilizado por la aplicación. Creado anteriormente con la función CreatePak() (ver tutorial Pak Create.cpp)
// Configurar ventana 800x600
D.mode(800,600);

}
/
**********/
```

```
Bool Init()
{
/*
Configuración de la Cámara por defecto (es decir, si no se especifica nada)

Cam.matrix.pos=(0,0,1) // Posición inicial por defecto (0,0,1) debido a que mantiene una distancia de 1 metro respecto a
Cam.at (hacia donde mira)
Cam.at=(0,0,0) // Posición hacia la que enfoca la cámara
Cam.dist=1 // Distancia entre la posición de la cámara y el punto enfocado
Cam.pitch=0 // Inclinación 0
Cam.yaw =0 // Orientación 0
Cam.roll =0 // Rotación eje x-z de la cámara
*/
// Guardamos el tiempo transcurrido en este momento
tiempo=Tm.time();

// Posición inicial de la esfera
PosEsfera=Vec(0,0,5);

return true;
}

/
**********************************************************************************************
**********************************************************************************************
**********/

void Shut() // Función general de salida o cierre del motor
{
}

/
**********************************************************************************************
**********************************************************************************************
**********/

Bool Main() // Funcion principal del motor, donde se establecen las operaciones necesarias para el juego.
{
if (Kb.bp(KB_ESC)){return false;} // Se aplica return false; para terminar con la aplicación. A continuación el engine se
dirije a void Shut()
if (Kb.bp(KB_TAB)) {iVar1++; return true;} // Al presionar la tecla TAB modificamos la variable iVar y salimos (sin
terminar la aplicación) de la función general Bool Main() y el motor se dirije a la función general void Draw()
if (Kb.bp(KB_SPACE)) {iVar2++; return true;}
// Al pulsar ENTER centramos la vista de la camara sobre el objetivo (en este caso una esfera(Ball)). Dependiendo del
modo utilizado de cámara hay que realizar diferentes ajustes
if (Kb.bp(KB_ENTER)) if (iVar2==0) {Cam.yaw=Yaw(Cam.matrix.pos,PosEsfera);
Cam.pitch=Pitch(Cam.matrix.pos,PosEsfera);
Cam.setAngle(Cam.matrix.pos,Cam.yaw,Cam.pitch,Cam.roll).updateVelocities().set(); return true;}
else if (iVar2==1) {Cam.at=PosEsfera;
Cam.setSpherical(Cam.at,Cam.yaw,Cam.pitch,Cam.roll,Cam.dist).updateVelocities().set(); return true;} else if
(iVar2==2) {Cam.at=PosEsfera; Cam.setFromAt(Cam.matrix.pos,Cam.at,Cam.roll).updateVelocities().set(); return
true;}
// Detecta si se ha pulsado las teclas de dirección. Si no es así, finalizar la función general (return true;)
if ((Kb.b(KB_LEFT)) || (Kb.b(KB_DOWN))) {fVar=-Tm.d*1.05;} else if ((Kb.b(KB_RIGHT)) || (Kb.b(KB_UP)))
{fVar=Tm.d*1.05;} else {return true;}
// Según la opción seleccionada iVar1 se modifican diferentes valores.
switch (iVar1) {
case 0 : if ((Kb.b(KB_LEFT )) || (Kb.b(KB_RIGHT))) {Cam.matrix.pos+=Cam.matrix.x*fVar;} else if
```

```
(Kb.b(KB_LCTRL)) {Cam.matrix.pos+=Cam.matrix.z*fVar;} else {Cam.matrix.pos+=Cam.matrix.y*fVar;} break;
case 1 : if ((Kb.b(KB_UP )) || (Kb.b(KB_DOWN ))) {Cam.dist-=fVar;} break;
case 2 : if ((Kb.b(KB_LEFT )) || (Kb.b(KB_RIGHT))) {Cam.yaw-=fVar;} break;
case 3 : if ((Kb.b(KB_UP )) || (Kb.b(KB_DOWN ))) {Cam.pitch+=fVar;} break;
case 4 : if ((Kb.b(KB_LEFT )) || (Kb.b(KB_RIGHT))) {Cam.roll+=fVar;} break;
case 6 : if ((Kb.b(KB_LEFT )) || (Kb.b(KB_RIGHT))) {Cam.at.x+=fVar;} else if (Kb.b(KB_LCTRL))
{Cam.at.z+=fVar;} else {Cam.at.y+=fVar;} break;
default : if ((Kb.b(KB_LEFT )) || (Kb.b(KB_RIGHT))) {PosEsfera.x+=fVar;} else if (Kb.b(KB_LCTRL))
{PosEsfera.z+=fVar;} else {PosEsfera.y+=fVar;}
}

// Según la opción seleccionada iVar2 se aplica un diferente modo de camara.
switch (iVar2) {
case 0 : Cam.setAngle(Cam.matrix.pos,Cam.yaw,Cam.pitch,Cam.roll); break;
case 1 : Cam.setSpherical(Cam.at,Cam.yaw,Cam.pitch,Cam.roll,Cam.dist); break;
case 2 : Cam.setFromAt(Cam.matrix.pos,Cam.at,Cam.roll); break;
default : Cam.setPosDir(Vec(PosEsfera.x,(PosEsfera.y)+1,(PosEsfera.z)-1));
}

Cam.updateVelocities().set(); // Actualizar y establecer cambios realizados en la Camara
return true; // Finalizar función general, sin cerrar apliación. A continuación se dirije a void Draw() como se ha
comentado anteriormente.
}

/
**********************************************************************************************
**********************************************************************************************
***********/

void Draw() // Funcion principal de dibujo del motor (funcion general)
{
// Si el valor de la opcion seleccionada excede el número de opciones existentes, reiniciar a cero
if (iVar1==7) iVar1=0;
if (iVar2==4) iVar2=0;

D.clear(WHITE); //Limpiar pantalla pintando todo de blanco.
Ball(1,PosEsfera).draw(ColorHue(Tm.time()/3)); // Dibuja una esfera de diámetro 1 metro de un color generado a partir
del tiempo transcurrido
// Configurar modo de escritura
TextDS estilo; // Declarar formato de texto

estilo.color=GREY; // Establecer el color del texto a gris
estilo.scale/=1.2; // Reducir el tamaño de texto (se utiliza /=)
estilo.align.set(-1,0); // Configurar alineación de texto justificación izquierda
estilo.shadow=1; // Activar sombra de texto (=1)
D.text(estilo,1.3,0.92,"Pulsa BARRA ESPACIADORA para cambiar seleccion"); // Escribir texto aplicando el formato
de texto 'estilo' en la posición (1.3,0.92)

estilo.align.set(1,0); // Configurar alineación de texto justificación derecha
D.text(estilo,-1.3,0.92,"Pulsa TAB para cambiar seleccion");

estilo.color=BLACK;
estilo.scale/=1.4;
estilo.shadow=0; // Desactivar sombra de texto (=0)
D.text(estilo,-1.2,0.85,S+"Posicion Camara (x,y,z): "+Cam.matrix.pos); // Para escribir texto junto a otros valores (float,
double, ...) se utiliza S+
D.text(estilo,-1.2,0.8,S+"Distancia: "+Cam.dist);
D.text(estilo,-1.2,0.75,S+"Orientacion (Yaw): "+Cam.yaw);
```

```
D.text(estilo,-1.2,0.7,S+"Inclinacion (Pitch): "+Cam.pitch);
D.text(estilo,-1.2,0.65,S+"Rotacion (Roll): "+Cam.roll);
D.text(estilo,-1.2,0.6,S+"Mover esfera. Posición Esfera: "+PosEsfera);
D.text(estilo,-1.2,0.55,S+"Enfoca a la posicion: "+Cam.at);
D.text(estilo,-1.3,0.45,S+"Distancia entre la camara y el objeto: "+Distance(Cam.matrix.pos,PosEsfera));
estilo.align.set(-1,0);
D.text(estilo,1.2,0.85,"Cam.setAngle");
D.text(estilo,1.2,0.8,"Cam.setSpherical");
D.text(estilo,1.2,0.75,"Cam.setFromAt");
D.text(estilo,1.2,0.7,"Cam.setPosDir");
estilo.color=RED;
if ((Tm.time()-tiempo)>0.4f) { // Si ha transcurrido más de 0.4 segundos desde 'tiempo', entonces escribir texto
D.text(estilo,1.3,(0.85-(0.05*iVar2)),"<<<");
estilo.align.set(1,0);
D.text(estilo,-1.3,(0.85-(0.05*iVar1)),">>>");
}

estilo.scale*=1.2; // Aumentar de tamaño el texto (se utiliza *=, que en este caso equivale a
estilo.scale=estilo.scale*1.2). No se trata de un puntero de C++. Para ser puntero, el * debe ir delante del nombre de la
variable.
estilo.align.set(0,0); // Justificación centrada, horizontal y verticalmente.
switch (iVar1) {
case 0 : case 5 : case 6 : D.text(estilo,0,-0.8,"Utiliza teclas de direccion izquierda/derecha(x), arriba/abajo(y), manten
pulsado Control Izquierdo(z)"); break;
case 1 : case 3 : D.text(estilo,0,-0.8,"Utiliza teclas de direccion arriba/abajo"); break;
default : D.text(estilo,0,-0.8,"Utiliza teclas de dirección izquierda/derecha");
}
estilo.scale*=1.4;
estilo.shadow=1;
if ((Tm.time()-tiempo)>1.8f) {tiempo=Tm.time();} else if ((Tm.time()-tiempo)>1.1f) {estilo.color=WHITE;}

D.text(estilo,0,-0.9,"Pulsa INTRO para enfocar la camara a la malla 3D");

// Al llegar al final de la función general void Draw() el motor vuelve a la función general Bool Main()
}
/
**********************************************************************************************
**********************************************************************************************
**********/
```