Diseño y Programación Orientados a Objetos

UNA VISIÓN APLICADA

Diego Rodríguez-Losada
Miguel Hernando

UNIVERSIDAD POLITÉCNICA DE MADRID

Diego RODRÍGUEZ LOSADA, Miguel HERNANDO
Diseño y Programación Orientados a Objetos. Una Visión Aplicada.

ISBN: 978-84-96737-51-8

Formato: 210 x 290 mm
Páginas: 198
Editorial: FUNDACIÓN GENERAL DE LA UNIVERSIDAD POLITÉCNICA DE MADRID
C/Pastor nº3. 28003 MADRID.

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Al Arkanoid, al Amstrad CPC6128, al Basic, a Ghost'n Goblins, al Mario Bros y al Donkey Kong, al tetris, a los marcianitos, al comecocos…

A mi familia.

Diego

A mi familia y a aquellos alumnos que al terminar pensaron que esto no valía tanto, pero que al descubrir el valor de lo aprendido han vuelto para animarnos a continuar con este esfuerzo.

Miguel

PRÓLOGO

Siempre que se empieza un libro, especialmente en el tema de programación, donde tantas buenas referencias se pueden encontrar, hay que valorar si realmente es necesario. Este libro parte de nuestra experiencia como investigadores, programadores, ingenieros software, y especialmente docentes, intentando llenar un hueco existente en la enseñanza del Diseño Orientado a Objetos y el lenguaje C++, a la vez que buscando incentivar la motivación y creatividad de los aprendices en la materia. Este libro es el trabajo de más de tres años, no solo de los autores, sino de todos los profesores de la asignatura de Informática Industrial de la EUITI-UPM (gracias especialmente a Carlos Platero, por dejarnos hacer, y por su interés, su constancia, su esfuerzo en revisarlo y sus numerosas contribuciones), y en gran medida de todos los alumnos que han pasado estos años por la asignatura, con una respuesta muy positiva al enfoque presentado en este libro.

Aunque utilizamos este libro para nuestra enseñanza presencial en la Universidad Politécnica de Madrid, creemos que también es un excelente material de estudio para el autoaprendizaje, ya que está escrito siguiendo la filosofía de un tutorial. El lector puede encontrar todo el código necesario en la página Web www.elai.upm.es.

En este libro se desarrolla un clásico juego de computador, partiendo desde cero, e incrementando capítulo a capítulo el mismo, con explicaciones, código, diagramas, ejercicios para el lector, etc. En dichos capítulos se incluyen explicaciones teóricas básicas sobre los conceptos fundamentales y la sintaxis del lenguaje, pero obviamente no se intenta abordar una cobertura exhaustiva de los mismos. Sin embargo, se hace especial hincapié en un buen diseño Orientado a Objetos desde el principio, se establecen conexiones directas constantes entre diagramas UML, el código C++ y el resultado visual de la aplicación, y se describen algunos patrones de diseño utilizados, en un enfoque conjunto que creemos altamente beneficioso para el estudiante.

Mediante esta metodología, la exposición a menudo árida de la filosofía y de la teoría del diseño software, cobra vida y se ve reflejada de forma directa en el programa que poco a poco se va desarrollando ayudando a adquirir y consolidar una disciplina tan importante en el campo de la ingeniería actual.

La motivación es muy importante a la hora de aprender. Creemos que el resultado final es altamente atractivo e interesante. Nosotros lo hemos pasado bien desarrollándolo, creemos que gran parte de nuestros alumnos se divierten con el enfoque planteado y esperamos que lo mismo suceda con el lector.

1. INTRODUCCIÓN AL ENTORNO DE PROGRAMACIÓN VISUAL C++. DEPURACIÓN DE UNA APLICACIÓN

1.1. INTRODUCCIÓN

A lo largo de los siguientes temas se diseñará y programará una aplicación gráfica interactiva (juego) en C++, utilizando el gestor de ventanas GLUT y las librerías OpenGL para renderizar escenas tridimensionales. El entorno de desarrollo utilizado es el Visual Studio 6.0 (Visual C++ 2008) de Microsoft, aunque podría ser otro entorno, incluso otro sistema operativo, ya que el código final de nuestra aplicación será totalmente portable. Aunque se supone que el lector tiene un conocimiento adecuado del lenguaje C, no necesariamente tiene que haber sido adquirido en este entorno, por lo que este primer capítulo se centra en la descripción del mismo, con una introducción a su funcionalidad básica de desarrollo y depuración.

1.2. REALIZACIÓN DEL PRIMER PROGRAMA EN VISUAL C

Para realizar nuestro primer programa se deberán seguir los siguientes pasos:

1. Arrancar Microsoft Visual C++ perteneciente al grupo de programas de Microsoft Visual Studio
2. Utilizaremos el "*Project Wizard*" para generar un nuevo proyecto (un proyecto contiene la información necesaria de configuración para compilar y generar un fichero ejecutable o aplicación), yendo al menú *File -> New -> Project*

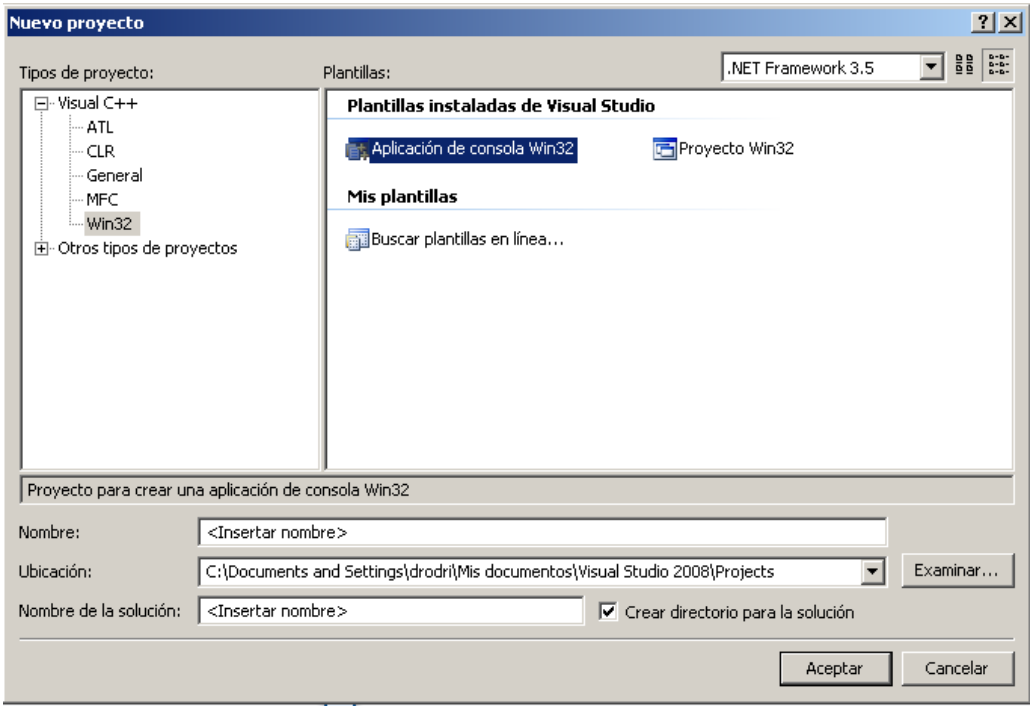


Figura 1-1. Asistente para la creación de un nuevo proyecto

3. Dar un nombre al proyecto (*Project Name*), seleccionar la carpeta donde será almacenado (*Location*), y seleccionar como tipo de proyecto "*Win32 Console Application*". Pulsar OK
4. Pulsar "Siguiente". Marcar "Proyecto vacio" y desmarcar "Encabezado precompilado". Pulsar "Finalizar".

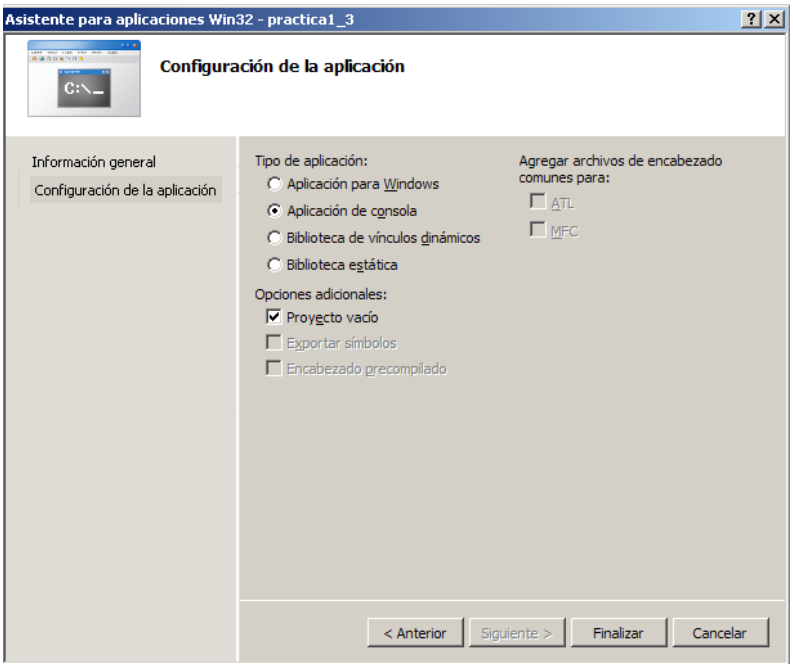


Figura 1-2. Crear un proyecto vacio sin cabeceras precompiladas.

5. Crear un fichero nuevo, llamado con extensión *.cpp. Para ello pulsando en la carpeta de ficheros de código fuente -> Agregar -> Nuevo Elemento. Seleccionamos *C++ Source File*. Introducir el nombre, por ejemplo ***principal.cpp*** y pulsar OK. (NOTA: Importante la extensión, ya que si ponemos otra extensión como .c, el archivo se compilara con el Compilador de C, en vez de el compilador de C++, lo que produciría errores)

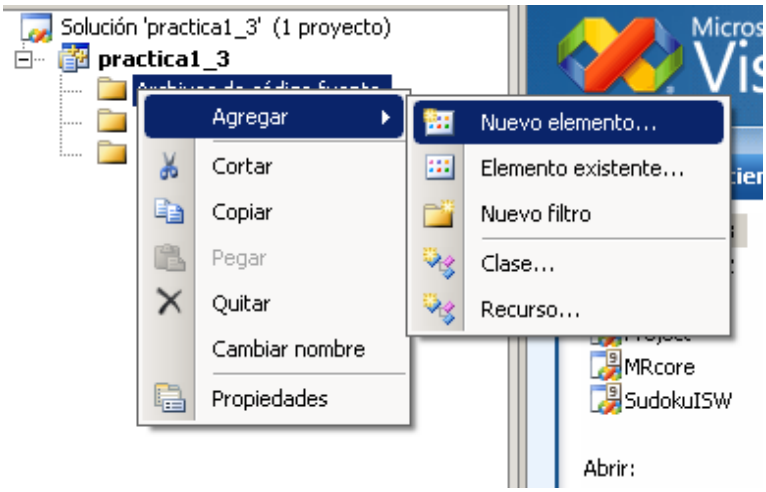


Figura 1-3. Asistente para la creación de un fichero fuente nuevo

6. Obsérvese como se puede abrir el fichero desde la etiqueta de *Explorador de soluciones.* Para ello cierra la ventana del archivo principal.cpp y haga doble click sobre el nombre del archivo en el navegador de ficheros "*Explorador de soluciones*"
7. Teclee el código de la siguiente aplicación

```
/*
 *  programa: Hola Mundo.cpp
 *    fecha:      7-10-2008
 *    comentarios: Mi primera aplicacion
 */

#include <stdio.h>

void main(void)
{
      printf("Hola Mundo \n");
}
```

8. Ver en *ClassView* como aparece el objeto "main" dentro del subconjunto de funciones globales. Cerrar el fichero y hacer doble click en "main" de *ClassView*.
9. Compilar el programa con *Build -> Build Practica1*
10. Ejecutar *Depurar -> Iniciar sin depurar (Ctrl + F5)*
11. Identificar los comandos de compilación y ejecución en la barra de herramientas

Realizados estos pasos, nos detendremos brevemente a examinar los distintos elementos y carpetas que componen el proyecto.

1.3. ESTRUCTURA DE ARCHIVOS Y DIRECTORIOS DEL PROYECTO

Dentro de la carpeta principal del proyecto podemos encontrar los siguientes archivos y carpetas importantes

- **Practica1.sln**: La solución de Visual Studio. Este es el archivo que tenemos que abrir (doble click) si queremos volver a abrir esta solución. Una solución puede contener varios proyectos, aunque en este libro utilizaremos exclusivamente 1 proyecto por solución.
- **Practica1**: La subcarpeta que contiene el proyecto, incluido el fichero de código fuente principal.cpp. Un proyecto contiene la informacion de cómo se compilan y enlazan los archivos de código fuente.
- **Release**: Estas subcarpetas (una a nivel global y otra para cada subproyecto) contienen los módulos objeto ".obj" y archivos auxiliares de compilación del proyecto así como el ejecutable en su versión Release (Ver seccion Configuraciones de Proyecto).
- **Debug**: Estas subcarpetas (una a nivel global y otra para cada subproyecto) contienen los módulos objeto ".obj" y archivos auxiliares de compilación del proyecto así como el ejecutable en su versión Debug (Ver seccion Configuraciones de Proyecto).
- **Principal.cpp**: El archivo de código fuente. Generalmente los archivos de código fuente y de cabecera se encuentran también en esta carpeta, a excepción de aquellos ficheros de código o librerías desarrollados por terceros o en otros proyectos.

1.4. EL PROCESO DE CREACIÓN DE UN EJECUTABLE

El compilador genera un fichero o modulo objeto (binario) por cada uno de los ficheros fuentes contenidos en el proyecto. Estos modulos objeto no necesitan para ser compilados más que el fichero fuente de origen, aunque se referencien funciones externas a dicho fichero.

El proceso de enlazado une los modulos objeto resolviendo las referencias entre ellos, asi como las referencias a posibles librerias[1] externas al proyecto, y generando el archivo ejecutable.

El sistema operativo es el encargado de unir el ejecutable con las librerias dinamicas cuando el programa es cargado en memoria.

[1] El término inglés utilizado para denominar al conjunto de funciones ya compiladas y disponibles para el uso por parte de otros programas es library cuya traducción correcta sería "biblioteca". Sin embargo, en la jerga de programación española, se ha extendido el término "librería", por lo que de aquí en adelante se utilizará este último.

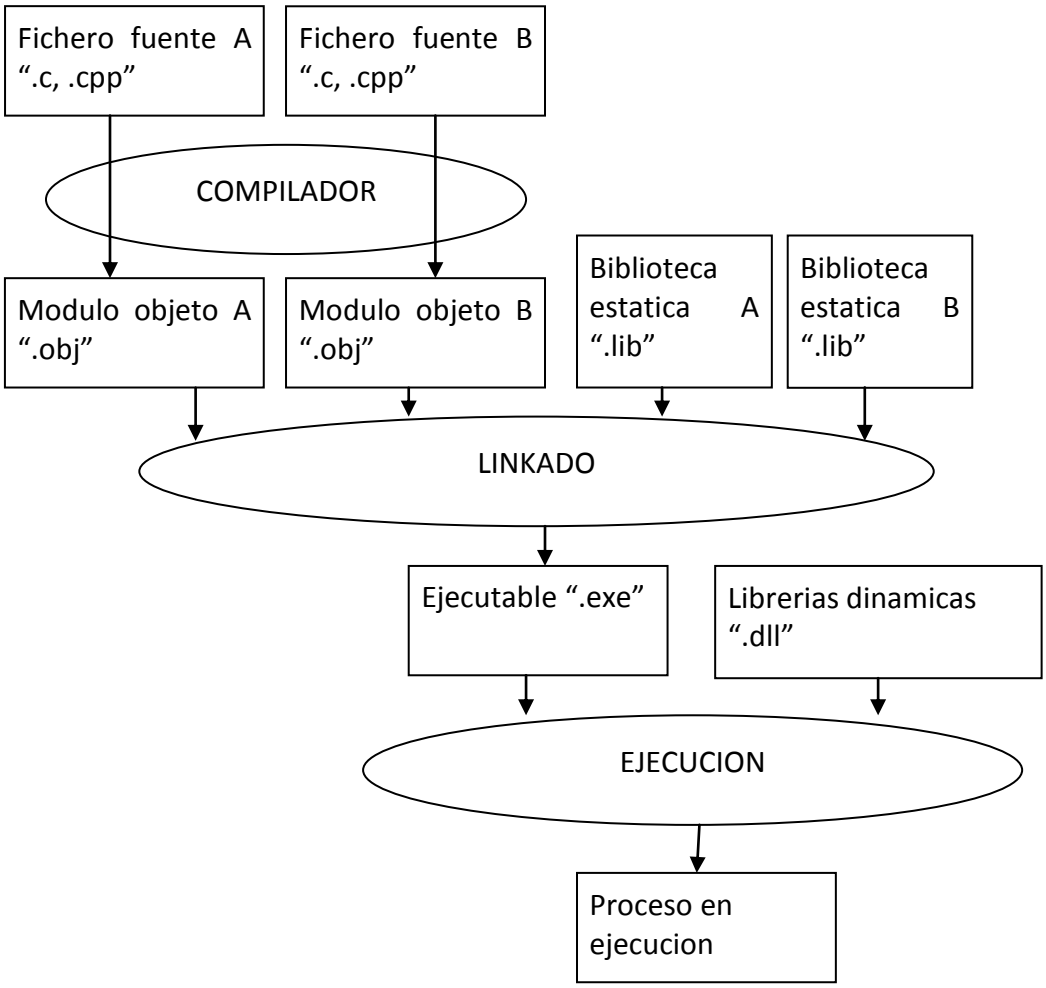


Figura 1-4. El proceso de creación de un ejecutable

1.5. CONFIGURACIONES DE PROYECTO (PROJECT CONFIGURATION)

El proyecto puede ser compilado en dos modos distintos: **Release y Debug**

La versión **Debug** es la versión de desarrollo, ya que permite depurar el código en tiempo de ejecución (mediante un depurador[2] integrado en el entorno). El ejecutable generado no está optimizado para ser rápido ni para ocupar menor tamaño. Además, para poder evaluar y seguir el valor de las variables así como la evolución del programa, es necesario incluir ciertas instrucciones adicionales que hacen el programa más lento y pesado. Sin embargo es un precio que se paga gustosamente dada la facilidad con la que mediante esta herramienta es posible corregir los inevitables errores que a la hora de programar se producen. Los ficheros de código objeto "*.obj" generados así como el

[2] En ingles *debugger*

ejecutable se encuentran en la subcarpeta ***Debug*** dentro de la carpeta principal del proyecto.

La versión **Release** es para generar un ejecutable optimizado para mayor velocidad, pero que no permite la depuración del código en busca de errores. Tanto los objetos ".obj" así como el ejecutable se encuentran en la subcarpeta ***Release*** dentro de la carpeta principal del proyecto.

En cualquier momento es posible seleccionar entre una u otra configuración. Esto se realiza mediante el menú desplegable en la barra de herramientas y seleccionar la que se desee. La configuración por defecto es la **Debug**

A continuación, y para finalizar el apartado, se comprobará que el programa ejecutable ".exe" generado es independiente del entorno de trabajo. Para ello, nos desplazamos hasta la carpeta correspondiente y lo ejecutamos haciendo doble click sobre él. De igual forma es posible su ejecución desde la interfaz de comandos:

1. Abrir una consola
2. Ir al directorio donde está el ejecutable
3. C:\directorio_propio>cd practica1\debug
4. Ejecutarlo con su nombre
5. C:\directorio_propio>practica1

Obsérvese la diferencia de tamaño entre el ejecutable generado en versión Debug y en versión Release

1.6. LOS PARÁMETROS DEL PROYECTO (PROJECT PROPERTIES)

Siguiendo con el entorno en el que desarrollamos el programa, vamos a ver el lugar en el que se establecen las distintas opciones que permiten generar soluciones de distintos tipos, así como acceder a funcionalidades más avanzadas.

En el menú *Project -> Properties*, se abre la ventana mostrada en la figura 1-3 , la cual contiene numerosos e importantes parámetros de configuración, compilación y enlazado del proyecto. Esta ventana nos permite definir por ejemplo si vamos a utilizar alguna librería (aparte de las estándar de visual, que son incluidas por el "Wizard" al seleccionar el tipo de proyecto), entre otras muchas cosas.

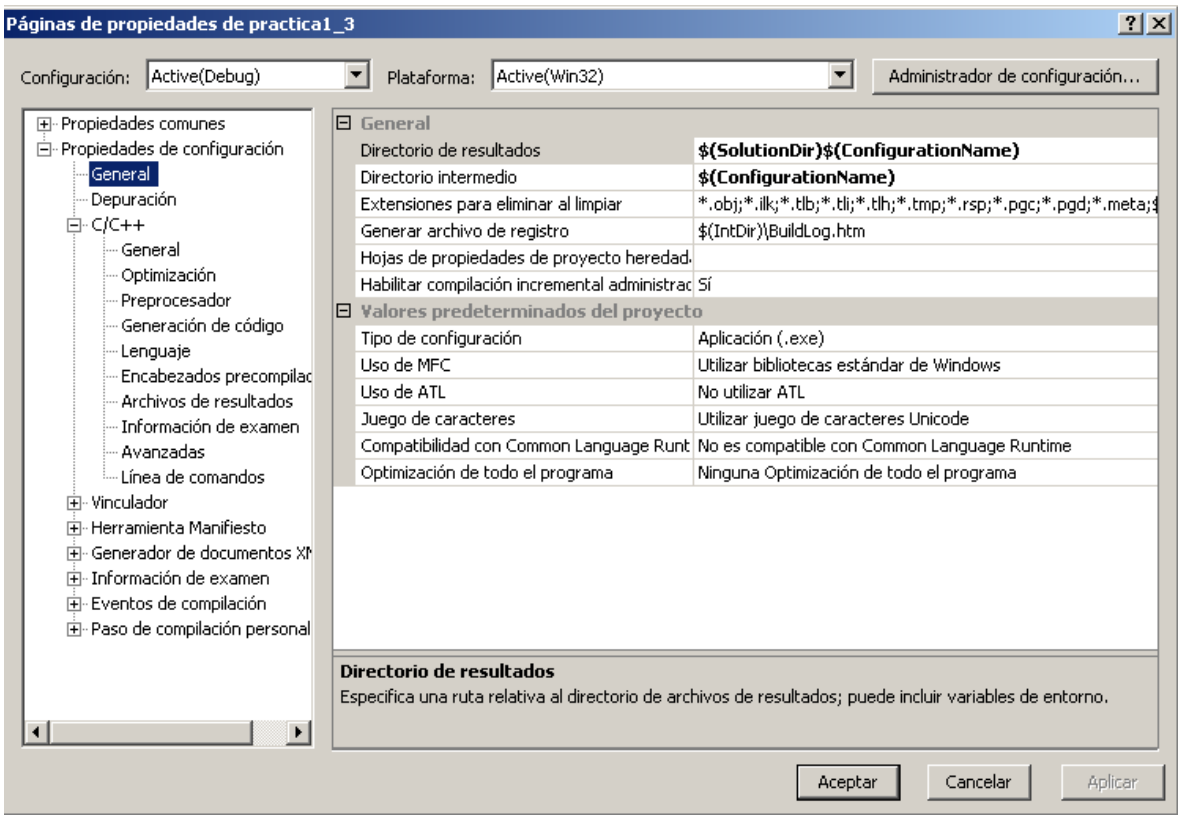


Figura 1-5. Parámetros o configuración del proyecto

Obsérvese y compruébese que los cambios que se hacen a estos parámetros afectaran solo a la versión Debug, a la Release o a ambas, en función de lo seleccionado en el *combobox* situado en la parte superior izquierda.

En el árbol de la izquierda se puede seleccionar el elemento o elementos sobre los que se quiere ver o cambiar la configuración, ya sea el proyecto entero o uno o varios ficheros.

Si se selecciona el proyecto entero se puede ver en la etiqueta Link la ruta y el nombre del fichero ejecutable, que se puede cambiar si se desea. En la etiqueta C/C++ se puede comprobar cómo las optimizaciones de código quedan desactivadas en el modo Debug.

Es importante conocer que todos estos parámetros se encuentran en las opciones del proyecto y que son modificables, Sin embargo, dada su extensión y en algunos casos su complejidad, se irán explicando en la medida en que vaya siendo necesaria su modificación.

1.7. TIPOS DE ERROR

Básicamente se pueden diferenciar dos tipos de errores en un programa: los errores en tiempo de ejecución y los errores en tiempo de compilación. Vamos a ver la diferencia entre ambos:

- Errores en tiempo de compilación.

Son errores, principalmente de sintaxis. El compilador los detecta y nos informa de ello, no produciendo un ejecutable. Vamos a provocar un error de este estilo. Realizamos el cambio:

```
printf("Hola Mundo \n");
```

Quitamos el punto y coma del final:

```
printf("Hola Mundo \n")
```

Y compilamos de nuevo. El mensaje que aparece en la ventana de *Output*, etiqueta *Build* es el siguiente:

```
\Leccion1\leccion1.c(14) : error C2143: syntax error : missing ';' before '}'
```

Haciendo doble click sobre el mensaje en cuestión, el editor abrirá el archivo correspondiente y marcará la línea en la que se produce este error.

- Errores en tiempo de ejecución.

También llamados errores lógicos o *run-time error*. Es un error que no es capaz de detectar el compilador porque no es un fallo en la sintaxis, pero que produce un malfuncionamiento durante la ejecución del programa por un fallo lógico. Por ejemplo, una división por cero, sintácticamente no es un error en el programa, por lo que el compilador construirá el ejecutable sin problema, sin embargo en el momento de ser ejecutado el programa e intentar la división, se producirá un error en tiempo de ejecución.

```
/*
 *  programa: división por cero
 *    fecha:      7-10-2003
 *    comentarios: Error en tiempo de ejecución
 */

#include <stdio.h>

void main(void)
{

int dividendo,divisor,cociente;
dividendo=3;
divisor=0;
cociente=dividendo/divisor;
printf("El cociente de %d y %d es %d \n",dividendo,divisor,cociente);

}
```

Para ilustrar esto último introducimos y compilamos este programa (en versión *Debug*) y lo ejecutamos. El programa fallará y saldrá un mensaje informándonos de ello. Al tener el sistema operativo un depurador, este mensaje nos da la opción de Depurar para buscar este error. De momento no pulsaremos este botón, sino que cancelaremos y depuraremos desde el entorno de programación el programa, hasta que el depurador nos informe de la división por cero.

También cabe la posibilidad de que un fallo en el código del programa produzca un comportamiento no deseado, pero que éste no suponga la finalización brusca del

programa. Para éste tipo de errores es especialmente útil el uso del depurador del entorno de programación

1.8. DEPURACIÓN DE LA APLICACIÓN

Para introducirnos en el procedimiento de depuración vamos a teclear el siguiente programa. La funcionalidad pretendida del mismo es que tras solicitar del usuario las coordenadas que definen dos puntos (x,y) del espacio bidimensional calcula la distancia entre ellos. Si lo probamos para dos puntos cuya distancia es fácilmente obtenible directamente (por ejemplo (0,10) y (0,5)) se observa que la respuesta no es correcta. Obviamente hay un error que a primera vista no es fácil detectar por lo que usaremos el depurador para descubrirlo

```
/*
 *  programa: distancia entre dos puntos
 */

#include <stdio.h>
#include <math.h>

void main(void)
{
      float x1,y1,x2,y2;
      printf("Introduzca las coordenadas del primer punto\n");
      scanf("%f %f",&x1,&y1);
      printf("Introduzca las coordenadas del segundo punto\n");
      scanf("%f %f",&x2,&y2);

      float norma=(x2-x1)*(x2-x1)-(y2-y1)*(y2-y1);
      float dist;//Notese la declaracion aqui, no permitida en C
      dist=sqrt(norma);
      printf("Distancia entre los puntos=%f\n",dist);
}
```

Una vez introducido, hay que asegurarse que se está compilando en la versión *Debug*, la cual como se dijo anteriormente nos permite utilizar el depurador.

En primer lugar introduciremos un punto de parada (*breakpoint*). Para ello pinchamos en la línea de código en la que queremos que se detenga la ejecución del programa (al comienzo del mismo) y pulsamos sobre el icono con aspecto de mano que aparece en la barra de herramientas. Aparecerá un punto gordo en el margen izquierdo de la línea.

Pulsamos *F5* para arrancar el proceso de depuración. El programa comienza a ejecutarse y se para en el *breakpoint*. Una flecha amarilla muestra la siguiente instrucción que se va a ejecutar. El `printf` aun no se ha ejecutado, por lo que si se mira en la consola, no aparecerá aun ningún mensaje.

Ejecutamos *Debug->Step Over* (pulsando F10 también vale) para ir ejecutando sentencia a sentencia el programa. Como consecuencia la flecha avanza una línea. Ahora si vemos el mensaje en pantalla, dado que se ha ejecutado la instrucción `printf`.

Si volvemos a pulsar F10, el programa se queda esperando (como consecuencia de la instrucción `scanf`) a que introduzcamos el punto, y no avanzara aunque volvamos a pulsar F10. Cuando hayamos introducido las coordenadas del punto, la flechita amarilla saltará a la línea siguiente.

Existe una ventana llamada *Watch* que permite ver los valores de las variables durante la depuración. Si no estuviera abierta es posible abrirla mediante *Debug Windows->Watch*.

Si escribimos el nombre de una variable en la primera casilla de la ventana *Watch* se puede ver su valor. En concreto, si introducimos `x1`, veremos el valor de la coordenada *x* que hemos introducido para el primer punto.

Como ejercicio se propone continuar la ejecución del programa pulsando la tecla F10, viendo cómo funciona el programa así como la salida por la consola. Encontrar el error y corregirlo

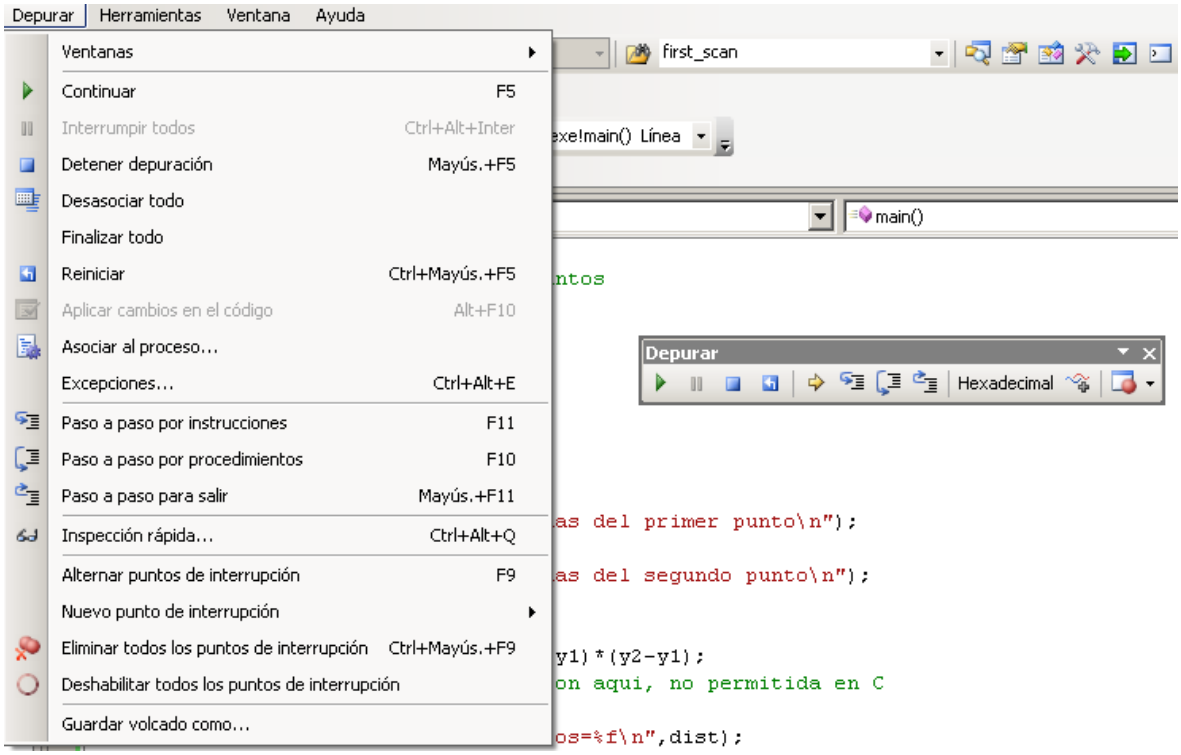


Figura 1-6. Comandos para compilar y ejecutar un programa

1.8.1. Otros comandos de depuración

Teclear el código siguiente.

```
#include <stdio.h>
#include <math.h>

float Distancia(float x1,float y1,float x2,float y2)
{
      float delta_x=x2-x1;
```

```
	float delta_y=y2-y1;
	float norma=delta_x*delta_x+delta_y*delta_y;
	float dist=sqrt(norma);
	return dist;
}

void main(void)
{
	float x1,y1,x2,y2;
	printf("Introduzca las coordenadas del primer punto\n");
	scanf("%f %f",&x1,&y1);
	printf("Introduzca las coordenadas del segundo punto\n");
	scanf("%f %f",&x2,&y2);
	float d=Distancia(x1,y1,x2,y2);
	printf("Distancia entre los puntos=%f\n",d);
}
```

Comenzar la depuración del mismo con un *Breakpoint* en la primera línea. Ejecutando paso a paso (F10, *Step Over*)

Entrar en el código de la función `Distancia` mediante el comando *StepInto* (F11). Si utilizáramos el comando *StepOver*, no se entraría en la depuración de la función `Distancia()`, sino que se ejecutaría como una sentencia de la función main.

Colocar el cursor en la última línea de la función y ejecutar *RunToCursor*, para no tener que ejecutar paso a paso toda la función, sino solo ver el valor final.

Repetir todo el proceso de depuración, pero una vez dentro de la función `Distancia` salir de la misma mediante la utilización del comando *StepOut*.

1.9. USO DE LA VENTANA WATCH

El contenido de las variables del programa se puede ver tanto en la ventana "*Watch*" *Debug Windows->Watch* como en la ventana "*Variables*" *View->Debug Windows->Variables.* Estas ventanas pueden ser abiertas una vez esta iniciada la depuración del programa, no antes. La segunda de ellas selecciona automáticamente las variables de las que muestra su valor, así como el contexto en el que se encuentra (la función).

Para explorar las posibilidades de la ventana *watch* escríbase el siguiente programa, que realiza la misma función que antes, pero utilizando estructuras

```
#include <stdio.h>
#include <math.h>

typedef struct
{
	float x;
	float y;
} punto;

float Distancia(punto p1,punto p2)
{
	float delta_x=p1.x-p2.x;
```

```
      float delta_y=p1.y-p2.y;
      float norma=delta_x*delta_x+delta_y*delta_y;
      float dist=sqrt(norma);
      return dist;
}

void main(void)
{
      punto punto1,punto2;
      printf("Introduzca las coordenadas del primer punto\n");
      scanf("%f %f",&punto1.x,&punto1.y);
      printf("Introduzca las coordenadas del segundo punto\n");
      scanf("%f %f",&punto2.x,&punto2.y);
      float d=Distancia(punto1,punto2);
      printf("Distancia entre los puntos=%f\n",d);
}
```

A continuación, depúrese el programa desde la primera línea de código y obsérvese en la ventana *watch* el valor que van tomando los datos. Para ello, introdúzcase los nombres de las variables en las casillas de la izquierda de la ventana *watch*.

Los datos en rojo indican los valores modificados por la última instrucción. Compruébese la indirección de los punteros, y su valor cuando no han sido inicializados, añadiendo las líneas

```
      punto* p;
      p=&punto1;
```

1.10. EJERCICIO

- Modificar el programa anterior para que realice la comprobación de si dos circunferencias en el plano colisionan entre ellas o no:
- Modificar la estructura, añadiendo el radio.
- Añadir una función que solicite al usuario los datos de una circunferencia por teclado
- Comprobar con el depurador el error si no se pasa por referencia
- Añadir una función que devuelva 1 o 0 en función de si hay o no hay colisión. Esta función puede utilizar la función distancia, o incluir el cálculo de la distancia dentro de ella.

1.11. EJERCICIO PROPUESTO

Aumentar el programa anterior, de tal forma que permita calcular si hay colisión entre un segmento, definido por sus puntos extremos y una circunferencia:

1. Añadir una estructura denominada “Segmento”, que contenga los dos puntos extremos
2. Añadir una función que le pida al usuario los datos de un segmento
3. Añadir una función que calcule la distancia de un punto a un segmento
4. Añadir una función que compruebe la colisión, devolviendo 1 o 0
5. Realizar la comprobación de la mayor velocidad de ejecución de una aplicación en modo *Release* y en modo *Debug*. Para ello, el programa principal debe medir el tiempo que tarda en ejecutar una función auxiliar que realice un cálculo matemático intensivo, como comprobar un millón de veces si hay colisión entre un segmento y una circunferencia cualquiera

2. PROGRAMACIÓN DE GRÁFICOS 3D CON OPENGL Y EL GESTOR DE VENTANAS GLUT

2.1. OPENGL Y GLUT

Lo primero que queremos destacar al comenzar este capítulo, es que el mismo no pretende ser una guía exhaustiva de como dibujar en OpenGL. Existen muchos manuales, tutoriales y otros recursos para el aprendizaje de OpenGL. Este capítulo trata de dar los conocimientos mínimos para poder trabajar y desarrollar nuestra aplicación. Por lo tanto, muchas cosas y gran parte de la potencia de OpenGL quedan sin explicar aquí.

OpenGL es una librería de funciones que nos permiten visualizar gráficos 3D en nuestra aplicación. La gran ventaja de esta librería es que nos aísla del hardware disponible; por tanto, si instalamos una tarjeta aceleradora, no hará falta que cambiemos nuestro programa para aprovechar la potencia de la tarjeta.

En este libro, no sólo vamos a trabajar con la librería OpenGL. Trabajaremos además con otras dos librerías: la GLU y la GLUT. La librería GLU contiene funciones gráficas de más alto nivel, que permiten realizar operaciones más complejas (algo análogo a la relación existente entre el ensamblador y el C) En cambio, la librería GLUT es un paquete auxiliar para construir aplicaciones de ventanas independientemente del sistema operativo en el que se esté trabajando, además de incluir algunas primitivas geométricas auxiliares. Por tanto, la gran ventaja de este paquete es que el mismo código podrá ser compilado sobre Windows™ o sobre Linux, además de simplificar mucho el código fuente del programa, como veremos en los ejemplos.

Estas tres librerías están escritas en C, y por lo tanto no son POO. La renderización 3D es una herramienta gráfica de visualización que nos permite cómodamente representar nuestros gráficos, y por tanto hacer atractiva la programación desde el primer momento. Sin embargo, nos gustaría insistir en que lo importante y por tanto en lo que se debe centrar el esfuerzo es en entender y aplicar correctamente los principios de la POO y de C++. Este capítulo introduce brevemente la forma de trabajar e interaccionar con estas librerías, y se deja al usuario la profundización en OpenGL, que puede realizar leyendo otros manuales disponibles en la Web.

Las funciones de las tres librerías se distinguen por su nomenclatura:

- GLUT. Las funciones se denominan comenzando por "glut" como `glutCreateWindow(…)` Estas funciones sirven en general para gestionar la ventana, crearla, gestionar eventos como el ratón o pulsaciones de teclado, así como algunas primitivas gráficas de alto nivel, como dibujar una esfera, un toro o una tetera.
- GLU (OpenGL Utility library). Funciones de dibujo de nivel medio, que empiezan por "glu" como `gluLookAt(…)`, que sirve para definir el punto de vista de la escena gráfica.
- GL. (OpenGL library) Funciones de dibujo de bajo nivel, que empiezan por "gl" como `glColor3f(…)` que sirve para definir colores.

Las librerías GL y GLU vienen con la distribución de Visual Studio C++ por defecto, así que no es necesario hacer nada para utilizarlas, exceptuando los `#include` que pudieran ser necesarios. Sin embargo el gestor de ventanas GLUT no forma parte del entorno de desarrollo por lo que es necesario conseguirlo, principalmente mediante descarga (legal y gratuita) desde la Web.

2.2. ESTRUCTURA DE UN PROGRAMA BASADO EN GLUT

En el código adjunto se suministra un proyecto que ya realiza los pasos descritos a continuación, por lo que no es necesario seguirlos para la consecución del capítulo, pero muestra el orden en el que se ha creado dicho proyecto.

2.2.1. Configuración del proyecto

Para empezar a manejar la GLUT, empezaremos creando un proyecto tipo consola (Win32 Console Application), y un fichero denominado ***principal.cpp*** en el que programamos una función `main()` con los parámetros habituales.

Para utilizar la librería GLUT y OpenGL necesitamos los siguientes componentes:

- Fichero de cabecera ***glut.h***
- Librería estática ***glut32.lib***
- Librería dinámica ***glut32.dll***

Estos ficheros se encuentran ya dentro de la carpeta del proyecto suministrada. Si se esta creando un proyecto nuevo, hay que conseguir estos ficheros (la Web) y copiarlos dentro de la carpeta.

Para poder utilizar las funciones de la GLUT pondremos un `#include "glut.h"` para tener acceso a las mismas:

```
#include "glut.h"

int main(int argc,char* argv[])
{
      return 0;
}
```

2.2.2. Creación de una ventana gráfica

El siguiente paso es pedirle al gestor de ventanas GLUT que nos cree una ventana gráfica. Las siguientes líneas de código inicializan la librería GLUT y crean una ventana de 800x600, que se denomina "MiJuego" y con las propiedades necesarias para que podamos dibujar. Asimismo, se inicializan también las luces y se definen algunas propiedades del dibujo como la perspectiva.

```
int main(int argc,char* argv[])
{
      //Inicializar el gestor de ventanas GLUT
      //y crear la ventana
      glutInit(&argc, argv);
      glutInitWindowSize(800,600);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutCreateWindow("MiJuego");

      //habilitar luces y definir perspectiva
      glEnable(GL_LIGHT0);
      glEnable(GL_LIGHTING);
      glEnable(GL_DEPTH_TEST);
      glEnable(GL_COLOR_MATERIAL);
      glMatrixMode(GL_PROJECTION);
      gluPerspective( 40.0, 800/600.0f, 0.1, 150);

      return 0;
}
```

Nótese que aunque este código compila, aun no se muestra ninguna ventana al ejecutar el programa. No es el objetivo de este libro que el usuario entienda y domine estas funciones. De hecho, con esta configuración de dibujo de OpenGL trabajaremos todo el libro, sin necesidad de modificar esta parte del código. Si el lector desea realizar gráficos más avanzados o sofisticados, puede referirse a los manuales y referencias de la bibliografía.

2.2.3. Registrando las funciones necesarias

La librería GLUT se encarga de llamar a unas funciones que programa el usuario cuando suceden eventos del sistema o cuando necesita realizar alguna tarea. De esta forma, el usuario declara y programa varias funciones e indica al gestor de ventanas la función que se debe ejecutar según cada evento. En este libro utilizaremos las funciones de dibujo (obligatoria de implementar en todo programa basado en GLUT), la función de temporización (timer) para realizar animaciones y movimiento, y la función de teclado, que es invocada cuando el usuario pulsa una tecla normal (no especial). Así añadimos a nuestro fichero ***principal.cpp*** las siguientes declaraciones

```
void OnDraw(void);
void OnTimer(int value);
void OnKeyboardDown(unsigned char key, int x, int y);
```

Cuando el usuario pulsa una tecla, GLUT se encargara de llamar a la función `OnKeyboardDown`. Cuando transcurra una temporización, GLUT llamara a `OnTimer` y cuando necesite refrescar el dibujo de la pantalla, llamara a `OnDraw`.

Por supuesto, estas funciones hay que implementarlas. De momento dejamos su implementación (cuerpo de la función) vacía, y en la próxima sección se explicara cómo hacerlo.

Para que GLUT sepa a qué funciones tiene que llamar en cada caso, hay que decírselo, en un proceso que se llama "registrar un *callback*", lo que hacemos en la función `main`:

```
//Registrar los callbacks
glutDisplayFunc(OnDraw);
glutTimerFunc(25,OnTimer,0);
glutKeyboardFunc(OnKeyboardDown);
```

Existen muchos otros "*callbacks*" que se pueden utilizar para otros eventos: ratón, redimensionado de pantalla, teclas especiales, etc. tal y como se describe en los manuales de la librería, o bien deducir del código del fichero de cabecera ***glut.h***

2.2.4. La función glutMainLoop()

Una vez que se ha inicializado la GLUT, y se han registrado los *callbacks*, se pasa el control definitivamente a la GLUT mediante la llamada a:

```
glutMainLoop();
```

Esta función hace entrar a la GLUT en un bucle infinito, que va invocando nuestras funciones `OnDraw()`, `OnKeyboardDown()`, etc., cuando es necesario

El código de nuestro proyecto queda por tanto como sigue:

```
#include "glut.h"

void OnDraw(void);
void OnTimer(int value);
void OnKeyboardDown(unsigned char key, int x, int y);

int main(int argc,char* argv[])
{
      //Inicializar el gestor de ventanas GLUT
      //y crear la ventana
      glutInit(&argc, argv);
      glutInitWindowSize(800,600);
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
      glutCreateWindow("MiJuego");

      //habilitar luces y definir perspectiva
      glEnable(GL_LIGHT0);
      glEnable(GL_LIGHTING);
      glEnable(GL_DEPTH_TEST);
      glEnable(GL_COLOR_MATERIAL);
      glMatrixMode(GL_PROJECTION);
      gluPerspective( 40.0, 800/600.0f, 0.1, 150);

      //Registrar los callbacks
      glutDisplayFunc(OnDraw);
      glutTimerFunc(25,OnTimer,0);//25 ms
      glutKeyboardFunc(OnKeyboardDown);
```

```
	//pasarle el control a GLUT,que llamara a los callbacks
	glutMainLoop();
	return 0;
}
void OnDraw(void)
{
}
void OnTimer(int value)
{
}
void OnKeyboardDown(unsigned char key, int x_t, int y_t)
{
}
```

2.3. DIBUJANDO OBJETOS

En este punto, si compilamos y ejecutamos el proyecto, veremos que se crea una ventana, pero en ella no se dibuja nada. Para dibujar algo en el contexto de OpenGL es necesario implementar la función `OnDraw()`, borrando la pantalla, definiendo el punto de vista, dibujando lo que queramos y por último llamando a `glutSwapBuffers()` función que implementa un redibujado de pantalla sin parpadeo.

La función `OnDraw()` será ejecutada de forma automática por la GLUT siempre que sea necesario repintar la ventana.

```
void OnDraw(void)
{
	//Borrado de la pantalla
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	//Para definir el punto de vista
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

	gluLookAt(0.0, 10, 20,  // posicion del ojo
		0.0, 0, 0.0,	// hacia que punto mira  (0,0,0)
		0.0, 1.0, 0.0);	// definimos hacia arriba (eje Y)

	//en este caso dibujamos solo un cubo de alambre
	//aqui es donde hay que poner el código de dibujo

	glutWireCube(2);

	//Al final, cambiar el buffer (redibujar)
	//no borrar esta linea ni poner nada despues
	glutSwapBuffers();
}
```

Al igual que antes, no es el objetivo de este libro que el usuario entienda todo lo referente a OpenGL. Las únicas partes de este código que deben de ser modificadas son los parámetros de la función `gluLookAt(…)` en caso de que se desee definir otro punto de vista de la escena y la parte de la función `glutWireCube(…)` que puede ser sustituida para dibujar otras cosas.

Compilar en este punto el proyecto y ejecutarlo. En pantalla deberemos ver algo como lo siguiente, con un cubo de alambre:

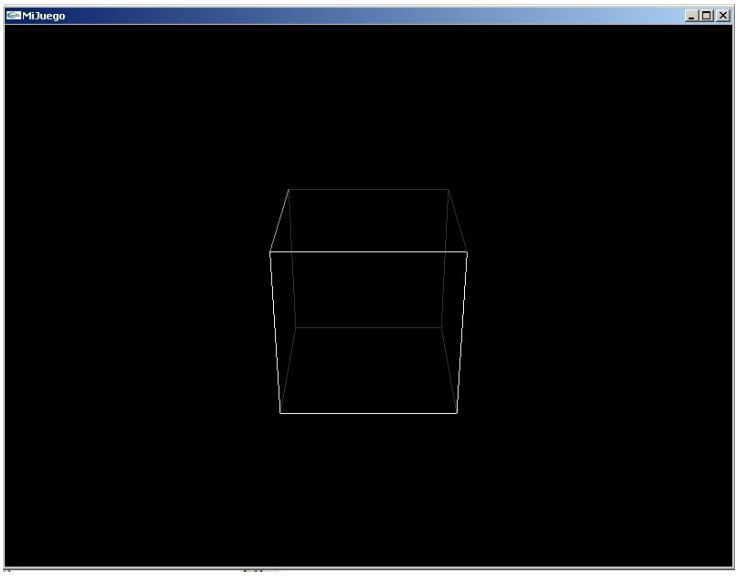


Figura 2-1. Aplicación con OpenGL y el gestor de ventanas GLUT

2.3.1. Primitivas de dibujo

El cubo lo hemos dibujado con la función `glutWireCube(…)` a la que hemos pasado el lado como parámetro. Utilizando distintas funciones de GLUT podemos dibujar distintas primitivas.

```
glutWireSphere( radius,  slices,  stacks);
glutSolidSphere(radius,  slices,  stacks);
glutWireCone( base,  height,  slices,  stacks);
glutSolidCone( base,  height,  slices,  stacks);
glutWireCube( size);
glutSolidCube( size);
glutWireTorus( innerRadius,  outerRadius,  sides,  rings);
glutSolidTorus( innerRadius,  outerRadius,  sides,  rings);
glutWireDodecahedron();
glutSolidDodecahedron();
glutWireTeapot( size);
glutSolidTeapot( size);
glutWireOctahedron();
glutSolidOctahedron();
glutWireTetrahedron();
glutSolidTetrahedron();
glutWireIcosahedron();
glutSolidIcosahedron();
```

Cuando una de estas primitivas tiene un parámetro tipo `size` o `radius`, define el tamaño del objeto. Si no lo tiene, siempre dibujara el objeto del mismo tamaño. Algunos de los parámetros de estas funciones corresponden a la discretización que se hace del dibujo de los objetos. Los parámetros `slices` o `stacks` corresponden a meridianos y paralelos de una esfera, y de forma similar `sides` y `rings`, son la discretización de un toro. Pruébese a cambiar estos valores y apreciará su efecto. Valores entre 10 y 20 para estos últimos parámetros dan un resultado adecuado para nuestros propósitos.

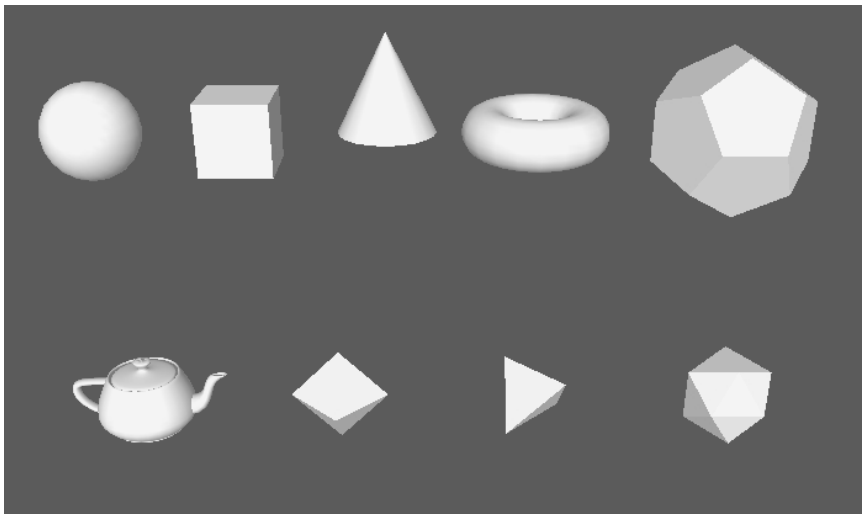


Figura 2-2. Primitivas de dibujo

Ejercicio: Probar en este punto a dibujar los siguientes objetos (cada uno de forma independiente, no hace falta que estén configurados como en la figura) en la pantalla en vez del cubo, y probar distintos valores de sus parámetros: Una esfera, un cono, un toro, un dodecaedro, una tetera, un tetraedro. Dibujar los objetos sólidos y con modelo de alambre

2.3.2. Colores

Un color en OpenGL viene definido de la forma más sencilla por sus 3 componentes de color, la componente roja (R=red), la componente verde (G=green) y la componente azul (B=blue).

Existen diferentes funciones para definir el color, y cuyos parámetros tienen diferentes rangos. Utilizaremos la siguiente función, (en la que los tres parámetros son de tipo `unsigned char`) para definir un color:

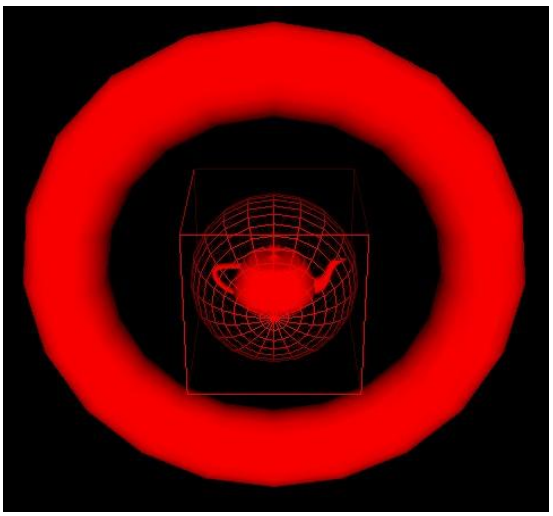
```
glColor3ub(rojo, verde, azul);
```

Los valores de estas variables, en el rango 0-255 pueden ser combinados, dando lugar a más de 16 millones de colores diferentes. Teniendo en cuenta que el espacio de color es aditivo, se muestran en la siguiente tabla algunas combinaciones básicas:

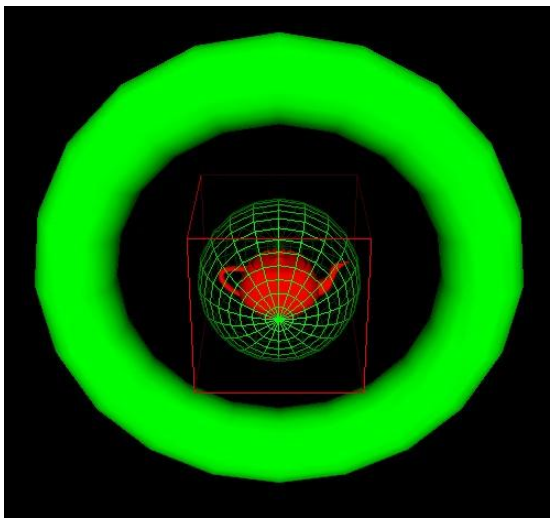
Tabla 1. Combinaciones de color

Componente			
Rojo	**Verde**	**Azul**	**Color**
255	255	255	Blanco
0	0	0	Negro
255	0	0	Rojo
0	255	0	Verde
0	0	255	Azul
255	255	0	Amarillo
0	255	255	Cyan
255	0	255	Magenta

OpenGL funciona como máquina de estados. Esto significa que una vez definido un color, todos los objetos que se dibujen a continuación, irán en ese color, hasta que no se especifique lo contrario. Nótese la diferencia en la siguiente figura. En el lado derecho se puede apreciar que el cubo y la tetera son dibujados en rojo, mientras que la esfera y el toro son dibujados en verde.



```
glColor3ub(255,0,0);
glutWireCube(4);
glutSolidTeapot(1);
glutWireSphere(2,20,20);
glutSolidTorus(1,5,20,20);
```



```
glColor3ub(255,0,0);
glutWireCube(4);
glutSolidTeapot(1);
glColor3ub(0,255,0);
glutWireSphere(2,20,20);
glutSolidTorus(1,5,20,20);
```

Figura 2-3. Colores

2.3.3. Definición del punto de vista

IMPORTANTE: Cuando trabajamos con la librería OpenGL, realmente no se dibuja en la pantalla, sino que se define el mundo o escena gráfica en coordenadas reales tridimensionales. Se puede pensar por lo tanto en un espacio 3D cuando se dibuja. Luego la librería se encarga de calcular y representar en la pantalla la vista que tendría de esa escena, un observador situado en unas coordenadas y que mirara hacia un punto. Cambiar las coordenadas del observador, cambiaria lo que se ve en pantalla sin modificar la escena o mundo.

La definición de la vista se realiza en nuestra función de dibujo `OnDraw()` con la llamada a la función `gluLookAt()`, que recibe los siguiente parámetros:

```
gluLookAt(x_ojo, y_ojo, z_ojo,      // posicion del ojo
          x_obs, y_obs, z_obs,    // hacia que punto mira
          0.0, 1.0, 0.0);       // definimos hacia arriba (eje Y)
```

Donde `x_ojo, y_ojo, z_ojo` son las coordenadas tridimensionales del punto donde se situaría el ojo del espectador o la cámara que visualiza la escena. Las coordenadas `x_obs,y_obs, z_obs` corresponden al punto al que dicho ojo está mirando. Por último, es necesario definir un vector que defina la orientación del mundo, el equivalente a un vector perpendicular al suelo o superficie terrestre.

Tal como se indica en la figura siguiente, se ha escogido un sistema de referencia tal que el eje Y es el que define dicho sentido, mientras que el plano paralelo al suelo es el plano XZ. Esto se ha hecho así, porque luego nuestra aplicación tendrá todos los objetos contenidos en un único plano, el plano XY, lo que facilita el desarrollo del mismo.

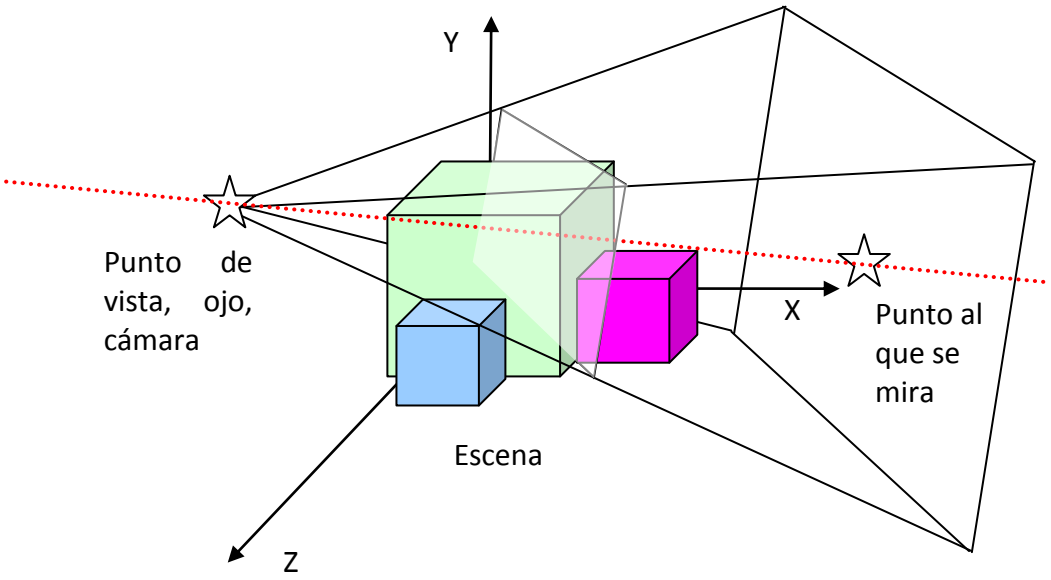


Figura 2-4. Definición del punto de vista

Con el ejemplo anterior se puede ver fácilmente el efecto que tiene cambiar el punto de vista, mirando siempre al origen de coordenadas:

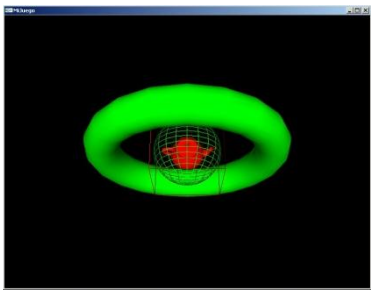
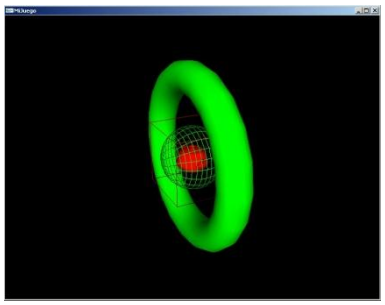
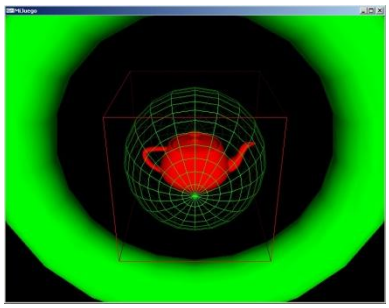
		
Cámara en (0,20,10)	Cámara en (20,10,10)	Cámara en (0,5,10)

Figura 2-5. Distintos puntos de vista de la misma escena

2.3.4. Desplazamiento de las figuras

Como se describe anteriormente, en OpenGL no dibujamos, sino que lo que hacemos es posicionar objetos (esferas, teteras, etc.) en una escena, en un mundo virtual. Las primitivas geométricas vistas, dibujan siempre dicho objeto en el origen de coordenadas. Sin embargo, lo que nosotros deseamos es poder posicionar dichos objetos en cualquier parte de la escena. Esto se consigue con la función:

```
glTranslatef(float x, float y, float z);
```

Recuérdese que OpenGL trabaja como una máquina de estados. Esto quiere decir que si hacemos una translación a un punto (x,y,z), todos los objetos que dibujemos a continuación estarán ubicados o referenciados a este punto, como si hubiéramos cambiado el origen de coordenadas. Si se desea volver a emplear el origen de coordenadas inicial, es necesario realizar la transformación inversa, tal y como se muestra en el ejemplo.

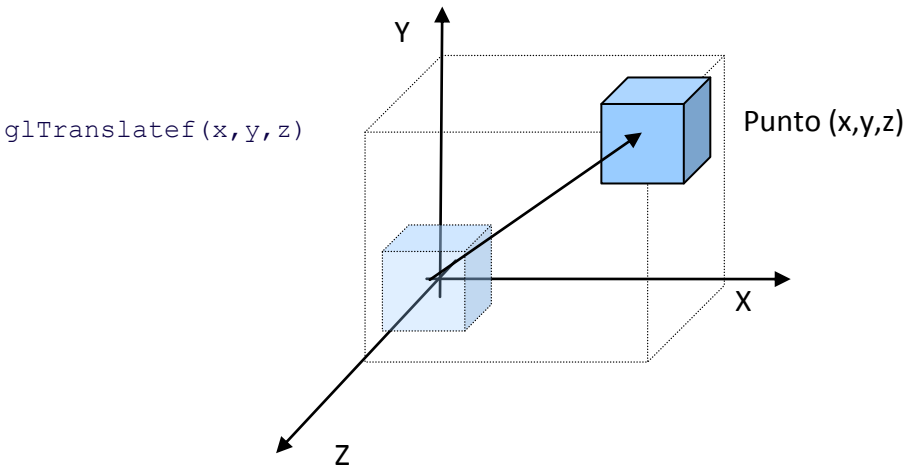


Figura 2-6. Desplazamiento de una figura en la escena

Este ejemplo dibuja primero una tetera y una esfera en las coordenadas (3,0,0) y finalmente otra tetera en el origen de coordenadas (situando otra vez el punto de vista en (0,10,20).

```
    glTranslatef(3,0,0);
    glutSolidTeapot(1);
    glutWireSphere(2,10,10);
    glTranslatef(-3,0,0);

    glutSolidTeapot(1);
```

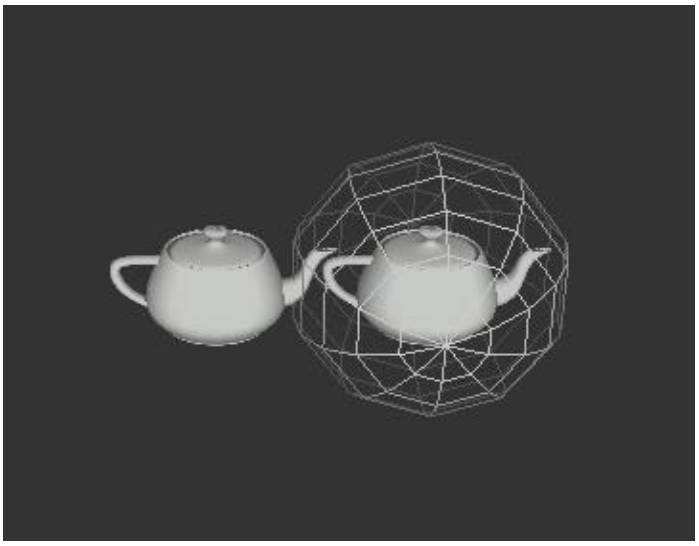
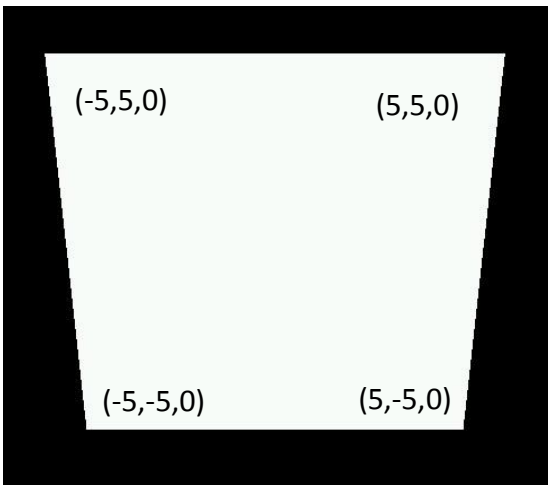


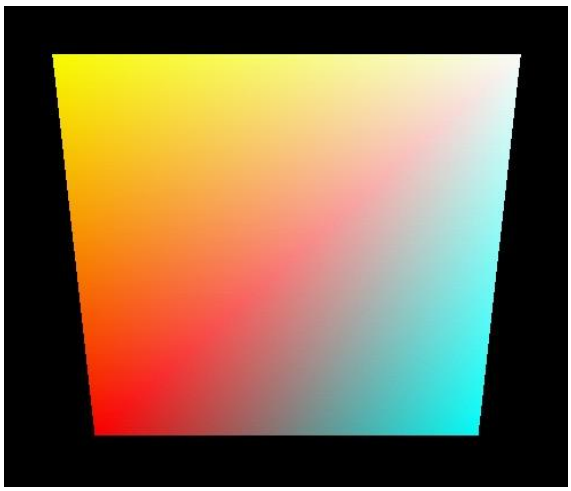
Figura 2-7. Translación de objetos en la escena

2.3.5. Dibujando formas simples.

Hasta ahora hemos utilizado primitivas de dibujo de la librería GLUT, que nos permitían utilizar objetos complejos como esferas, teteras, toros, etc. La librería OpenGL permite también dibujar formas simples como polígonos, rectas, triángulos, puntos.

En el siguiente ejemplo se dibuja un cuadrado en la pantalla, primero de color uniforme y luego definiendo un color diferente para cada vértice. La interpolación de colores del cuadrado la hará automáticamente OpenGL..





```
glColor3ub(255,255,255);
glBegin(GL_POLYGON);
      glVertex3f(-5.0f,-5.0f,0.0f);
      glVertex3f(-5.0f,5.0f,0.0f);
      glVertex3f(5.0f,5.0f,0.0f);
      glVertex3f(5.0f,-5.0f,0.0f);
glEnd();
```

```
glBegin(GL_POLYGON);
      glColor3ub(255,0,0);
      glVertex3f(-5.0f,-5.0f,0.0f);
      glColor3ub(255,255,0);
      glVertex3f(-5.0f,5.0f,0.0f);
      glColor3ub(255,255,255);
      glVertex3f(5.0f,5.0f,0.0f);
      glColor3ub(0,255,255);
      glVertex3f(5.0f,-5.0f,0.0f);
glEnd();
```

Figura 2-8. Dibujo de un cuadrado

La definición de vértices del polígono viene delimitada entre las funciones `glBegin()` y `glEnd()`. En cada función `glVertex3f()` se definen las coordenadas de un punto en el espacio 3D de la escena, siendo los parámetros de tipo `float`.

Nótese que los vértices tienen que ir definidos en un sentido, no pueden ir definidos en cualquier orden. Definirlos en sentido horario u antihorario define el lado principal o cara principal del polígono, aunque este detalle no nos afecte para nuestra aplicación. En el caso de que no se muestren los colores adecuadamente suele ser un problema de iluminación. Lo que haremos para dibujar polígonos es desactivar dicha iluminación, volviéndola a activar al terminar de dibujar el polígono, de la siguiente forma:

```
glDisable(GL_LIGHTING);
glBegin(GL_POLYGON);
      glVertex3f(-5.0f,0,-5.0f);
      glVertex3f(-5.0f,0,5.0f);
      glVertex3f(5.0f,0,5.0f);
      glVertex3f(5.0f,0,-5.0f);
glEnd();
glEnable(GL_LIGHTING);
```

También es importante darse cuenta, que la función `glTranslatef()` no es necesaria, ya que se puede especificar directamente las coordenadas de los objetos.

El parámetro de la función `glBegin()`, define el tipo de objeto que se va a dibujar. Sus posibles valores son:

Tabla 2. Valores del parámetro de la función glBegin()

GL_POINTS	Puntos individuales aislados
GL_LINES	Cada par de puntos corresponde a una recta
GL_LINE_STRIP	Segmentos de recta conectados
GL_LINE_LOOP	Segmentos de recta conectados y cerrados
GL_TRIANGLES	Cada tres puntos un triángulo
GL_QUADS	Cada cuatro puntos un cuadrado
GL_POLYGON	Un polígono

Ejercicio: Dibujar diferentes objetos geométricos simples, cambiando el parámetro de glBegin(). Conviene para ello consultar la ayuda de Visual.

2.4. INTERACCIONANDO CON LOS OBJETOS DE LA ESCENA

Hasta ahora hemos visto como dibujar objetos estáticos en el mundo, pero si queremos hacer una aplicación interactiva en la que el usuario cambia la escena tenemos que implementar más cosas. Como ejemplo vamos a ampliar el programa, de tal forma que nos permita controlar una esfera por la escena con las siguientes pulsaciones de teclado:

- 'a' mueve la esfera a la izquierda (en el plano XY)
- 'd' mueve la esfera a la derecha
- 'w' mueve la esfera hacia arriba (en el plano XY)
- 's' mueve la esfera hacia abajo
- 'r' cambia el color de la esfera a rojo
- 'g' cambia el color de la esfera a verde
- 'b' cambia el color de la esfera a azul
- '+' incrementa el radio de la esfera en 0.5 unidades, sin superar las 3 uds.
- '-' decrementa el radio de la esfera en 0.5 unidades, sin bajar de 1 ud.

Comenzaremos por lo más sencillo, mover la posición de la esfera. Cuando el usuario pulsa una tecla en la ventana gráfica abierta por la GLUT, esta se encarga de llamar automáticamente a la función o *callback* que habíamos registrado y que había quedado vacía:

```
void OnKeyboardDown(unsigned char key, int x_t, int y_t)
{

}
```

El primer parámetro de esta función corresponde a la tecla pulsada, mientras que los dos siguientes son irrelevantes y no los utilizaremos. Obviamente, si queremos que nuestro programa responda a eventos de teclado, tendremos que poner código dentro de esta función. Por otra parte, dentro de esta función NO se pueden poner funciones de dibujo. Solo se puede poner código de dibujo dentro de la función `OnDraw()`. Por lo tanto tiene que haber información común entre la función `OnDraw()` y la función `OnKeyboardDown()`.

Añadimos por tanto a nuestro programa una variable global, que se denomine por ejemplo "`radio_esfera`", y que utilizaremos tanto en la función `OnDraw()` como en `OnKeyboardDown()`. El código queda como sigue (se sustituyen trozos de código con puntos suspensivos):

```
#include "glut.h"

float radio_esfera=2.0f;

//declaraciones
…


int main(int argc,char* argv[])
{
      …
```

```
	glutMainLoop();
	return 0;
}


void OnDraw(void)
{
	…

	gluLookAt(0.0, 10, 20,  // posicion del ojo
		0.0, 0, 0.0,	// hacia que punto mira  (0,0,0)
		0.0, 1.0, 0.0);	// definimos hacia arriba (eje Y)

	glutSolidSphere(radio_esfera,20,20);

	…
}

void OnTimer(int value)
{
	…
}

void OnKeyboardDown(unsigned char key, int x_t, int y_t)
{
	//poner aqui el código de teclado
	if(key=='+' && radio_esfera<3)
		radio_esfera+=0.5f;
	if(key=='-' && radio_esfera>1)
		radio_esfera-=0.5f;

	glutPostRedisplay();
}
```

La llamada `glutPostRedisplay()` sirve para notificar a OpenGL que hemos modificado la escena y que tiene que redibujar. No es una llamada directa a la funcion de dibuja.

Procedemos de la misma forma para cambiar la posición y el color de la esfera. Se muestra solo el código modificado:

```
#include "glut.h"

float radio_esfera=2.0f;
float x_esfera=0.0f;
unsigned char rojo_esfera=255;
unsigned char verde_esfera=255;
unsigned char azul_esfera=255;
…

void OnDraw(void)
{
	//Borrado de la pantalla
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	//Para definir el punto de vista
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
```

```
	gluLookAt(0.0, 10, 20,  // posicion del ojo
		0.0, 0, 0.0,	// hacia que punto mira  (0,0,0)
		0.0, 1.0, 0.0);		// definimos hacia arriba (eje Y)

	//aqui es donde hay que poner el código de dibujo
	glColor3ub(rojo_esfera,verde_esfera,azul_esfera);
	glTranslatef(x_esfera,0,0);
	glutSolidSphere(radio_esfera,20,20);
	glTranslatef(-x_esfera,0,0);

	//no borrar esta linea ni poner nada despues
	glutSwapBuffers();
}
void OnKeyboardDown(unsigned char key, int x_t, int y_t)
{
	//poner aqui el código de teclado
	if(key=='+' && radio_esfera<3)
		radio_esfera+=0.5f;
	if(key=='-' && radio_esfera>1)
		radio_esfera-=0.5f;
	if(key=='r')
	{
		rojo_esfera=255;
		verde_esfera=0;
		azul_esfera=0;
	}
	if(key=='a')//izq, X negativo
		x_esfera-=0.1f;

	glutPostRedisplay();
}
```

Ejercicio: Se deja en este punto al lector que complete el programa según lo indicado anteriormente.

2.5. ESTRUCTURANDO EL PROGRAMA

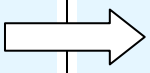
El programa anterior puede (y debe) ser escrito de una manera más adecuada y estructurada. Se tiene un conjunto de variables globales, todas con el nombre "esfera". El buen programador de C se dará cuenta que toda la información relacionada con una esfera es mejor que este almacenada en una estructura `struct`.

Realmente, la utilización de esta estructura requeriría en ANSI C, una definición de la misma de la siguiente forma:

```
//ANSI C
typedef struct _Esfera
{
	float radio;
	float x;
	float y;
	unsigned char rojo;
	unsigned char verde;
	unsigned char azul;
} Esfera;
```

No obstante C++ permite una definición y utilización más sencilla que no requiere del `typedef`, y que utilizaremos tanto por su simplicidad así como por su facilidad de convertirla a una clase como se verá en el tema posterior. La definición es la siguiente:

```
float radio_esfera=1.0f;
float x_esfera=0.0f;
float y_esfera=0.0f;
unsigned char rojo_esfera=255;
unsigned char verde_esfera=255;
unsigned char azul_esfera=255;
```



```
struct Esfera
{
      float radio;
      float x;
      float y;
      unsigned char rojo;
      unsigned char verde;
      unsigned char azul;
};

Esfera esfera={1,0,0,255,255,255};
```

Nótese como se ha eliminado de los campos de la estructura el nombre “esfera”, ya que al estar dentro de la estructura, sería redundante. El código de la función `OnKeyboardDown()` quedaría como sigue:

```
      if(key=='+' && esfera.radio<3)
            esfera.radio+=0.5f;
      if(key=='-' && esfera.radio>1)
            esfera.radio-=0.5f;
      if(key=='r')
      {
            esfera.rojo=255;
            esfera.verde=0;
            esfera.azul=0;
      }
      if(key=='a')
            esfera.x-=0.1f;
      if(key=='d')
            esfera.x+=0.1f;
      if(key=='s')
            esfera.y-=0.1f;
      if(key=='w')
            esfera.y+=0.1f;
```

Asimismo el código de dibujo de la función `OnDraw()` podría ser:

```
      glColor3ub(esfera.rojo,esfera.verde,esfera.azul);
      glTranslatef(esfera.x,esfera.y,0);
      glutSolidSphere(esfera.radio,20,20);
      glTranslatef(-esfera.x,-esfera.y,0);
```

Si quisiera dibujarse otra esfera, habría que repetir este código. Sin embargo, es mucho mejor programar una función que se encargue de esta tarea e invocarla desde `OnDraw()`, tal y como se muestra en el siguiente código:

```
#include "glut.h"

struct Esfera
{
      float radio;
      float x;
      float y;
      unsigned char rojo;
      unsigned char verde;
      unsigned char azul;
};
```

```
Esfera esfera={1,0,0,255,255,255};
Esfera esfera2={1,3,0,0,255,255};

//declaraciones de funciones
void Dibuja(Esfera e);

main(…)
{
…
}

void Dibuja(Esfera e)
{
      glColor3ub(e.rojo,e.verde,e.azul);
      glTranslatef(e.x,e.y,0);
      glutSolidSphere(e.radio,20,20);
      glTranslatef(-e.x,-e.y,0);
}
void OnDraw(void)
{
      …

      //aqui es donde hay que poner el código de dibujo
      Dibuja(esfera);
      Dibuja(esfera2);

      …
}
```

2.6. ANIMACIÓN: LA FUNCIÓN TIMER

Hemos registrado un *callback*, la función `OnTimer()` que será ejecutada automáticamente por la GLUT aproximadamente cada 25 milisegundos, pero que de momento hemos dejado vacía. Si dentro de esta función ponemos un código que modifique los datos de la esfera, conseguiremos una animación. Por ejemplo, en el siguiente código añadimos un pequeño incremento del radio de la esfera, hasta que llega un máximo, momento en el que vuelve a un radio pequeño. Este efecto lo aplicamos inicialmente solo a la segunda esfera:

```
void OnTimer(int value)
{
      //poner aqui el código de animacion
      esfera2.radio+=0.1f;
      if(esfera2.radio>3)
            esfera2.radio=0.5f;

      //no borrar estas lineas
      glutTimerFunc(25,OnTimer,0);
      glutPostRedisplay();
}
```

La llamada `glutPostRedisplay()` en el *callback* `OnTimer()` sirve para informar que después de cambiar las cosas hay que redibujar. Además, se vuelve a

registrar el *callback* para volver a fijar la temporización para dentro de otros 25 milisegundos

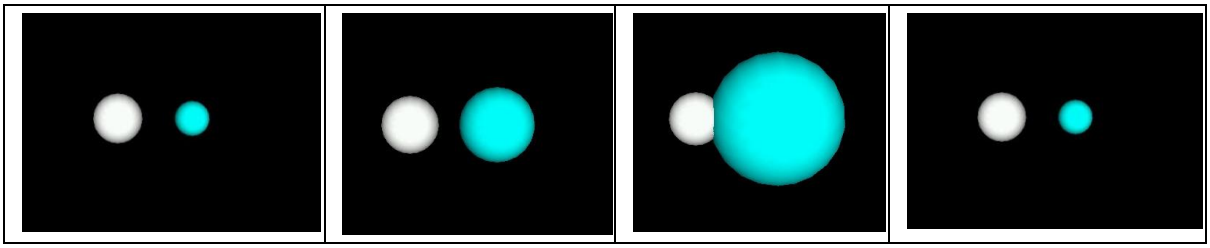


Figura 2-9. Animación que cambia el radio de una esfera.

Si quisiéramos hacer que las dos esferas pulsaran, sería conveniente hacer de forma similar a la función dibujo una función que se encargara de realizar el código de animación de las esferas. Llamamos a esta función `Mueve()`.

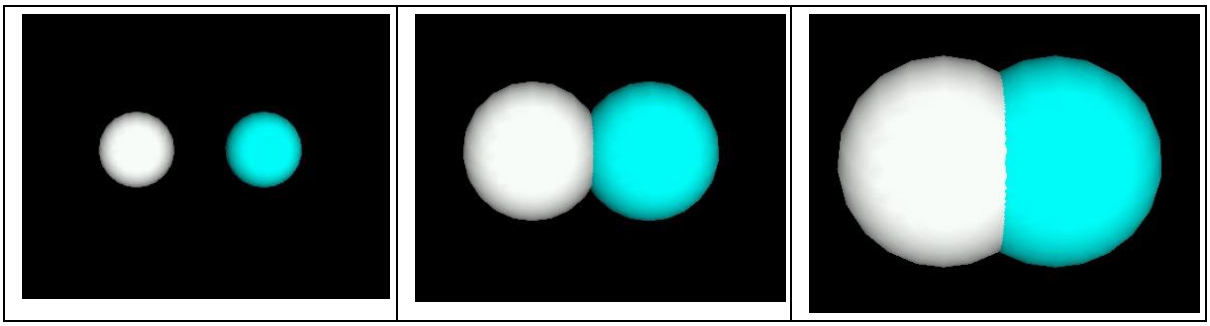


Figura 2-10. Animación que cambia el radio de 2 esferas

Como se aprecia en el código, a la función `Mueve()` hay que pasarle el parámetro de tipo `Esfera` por referencia (puntero) ya que la función tiene que modificar los valores de la esfera que se le pasa como parámetro, y esto solo se puede conseguir así.

```
…
void Mueve(Esfera* e);

int main(int argc,char* argv[])
{
      …
}
…
void OnTimer(int value)
{
      //poner aqui el código de animacion
      Mueve(&esfera);
      Mueve(&esfera2);

      glutTimerFunc(25,OnTimer,0);
      glutPostRedisplay();
}
void Mueve(Esfera* e)
{
      e->radio+=0.1f;
      if(e->radio>3)
            e->radio=0.5f;
}
```

EJERCICIO: Cambiar el color de la esfera según crece en la animación

2.7. IMPORTANTE: DESACOPLAR EL DIBUJO DE LA PARTE LÓGICA

Cabe resaltar en este momento una forma de trabajar importante:

La función de dibujar no realiza ninguna acción lógica, no cambia valores de las variables ni hace ninguna operación sobre ellas. Simplemente consulta su valor para dibujar las cosas en la posición y con el tamaño y color adecuados

Las funciones de animación y de respuesta a eventos como teclado simplemente modifican los valores de las variables, sin tener en cuenta para nada ninguna función de dibujo.

De esta forma la parte lógica y la parte gráfica de la aplicación están muy desacopladas, de tal forma que podemos trabajar de forma más cómoda y la aplicación es más portable.

Esta forma de desacoplar la parte lógica y la parte gráfica debe ser mantenida al desarrollar aplicaciones gráficas.

2.8. EJERCICIO PROPUESTO

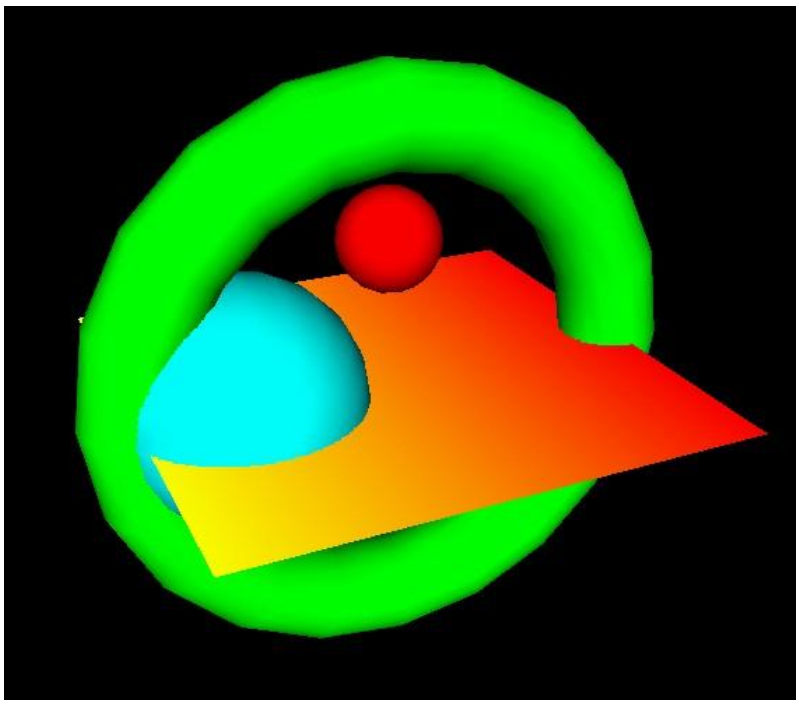


Figura 2-11. Ejercicio propuesto: Un escena interactiva y animada

Realizar un programa con la GLUT siguiendo el esquema presentado con las siguientes funcionalidades:

- Hay un plano de colores en el suelo, centrado en el origen y de lado 10.
- Hay un toro de color verde, como el mostrado en la figura
- Una esfera de color cyan esta quieta en las coordenadas (3,0,0) pero va cambiando su radio en el tiempo como se ha realizado en este capítulo.

- La otra esfera se puede mover en el plano XY, cambiar el radio y el color, todo ello con el teclado, como se ha descrito en el capítulo.
- El punto de vista orbita lenta y continuamente alrededor del centro de la imagen, el punto 0, 0,0. Para ello:
  - Crear una estructura "Mundo" que contenga las variables de posición 3D de la cámara, e instanciar una variable global de esta clase.
  - Cambiar en la función `OnDraw()` la función `gluLookAt()` para que utilice estos valores
  - Calcular en la función `OnTimer()` una nueva posición de la cámara, que sigue una trayectoria circular paralela al suelo, mirando siempre al origen. Seguir los siguientes pasos:
    - Calcular la distancia "`d`" del punto de vista al eje Y
    - Calcular el ángulo plano "`theta`" formado con el eje X (`atan2(z,x)`)
    - Incrementar este ángulo un diferencial
    - Volver a calcular el punto de vista, manteniendo la coordenada Y y calculando `x=d*cos(theta)` y `z=d*sin(theta);`

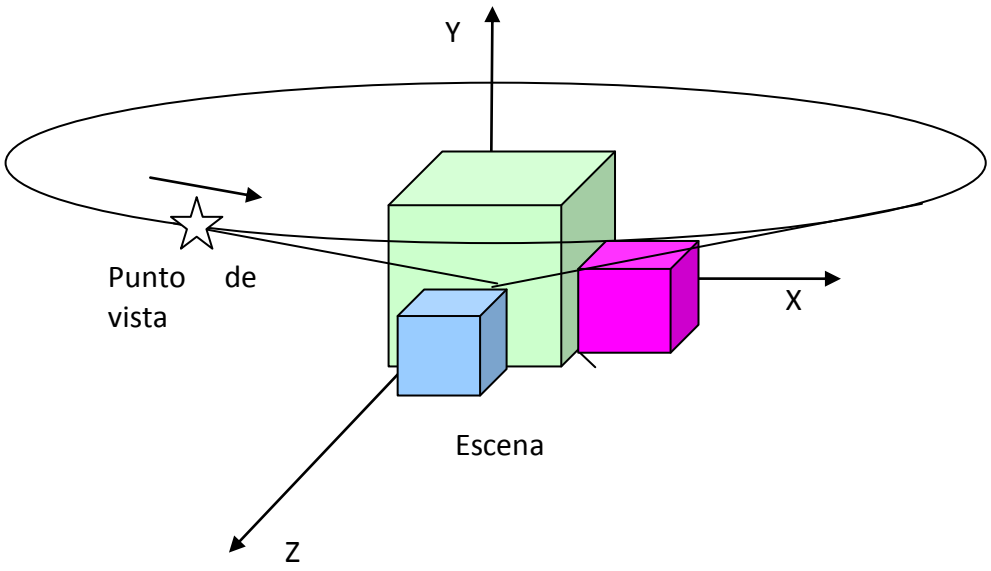


Figura 2-12. Cámara orbitando una escena

3. CLASES DE C++

En este capítulo se comienzan a introducir los conceptos de clase, objeto y encapsulamiento y se comienza a trabajar en la idea de la Programación Orientada a Objetos, precisamente con la introducción del elemento básico: el objeto. Se pretende la familiarización con el uso y la forma de los programas que utilizan objetos. No es finalidad el comenzar a diseñar nuestros propios objetos, sino más bien el de descubrir lo fácil que se vuelven operaciones complejas gracias al uso de los objetos. En concreto se observará como unas pocas líneas de código pueden tener ahora una potencia extraordinaria.

Un objeto es una unidad que engloba en sí mismo datos y procedimientos necesarios para el tratamiento de esos datos. Hasta ahora se han realizado programas en los que los datos y las funciones estaban perfectamente separados. Cuando se programa con objetos esto no es así, cada objeto contiene datos y funciones. Y un programa se construye como un conjunto de objetos. Los objetos se agrupaban en clases, de la misma forma que en la realidad, existiendo muchas mesas distintas (la de mi cuarto, la del comedor, la del laboratorio, la del compañero o la mía) agrupamos todos esos objetos reales en un concepto más abstracto denominado mesa. De tal forma que podemos decir que un objeto específico es una realización o instancia de una determinada clase.

Por ejemplo, en los programas de Windows, cada botón clásico de una interfaz es un objeto. Todos los botones son **instancias** o realizaciones de la clase `CButton` de Windows. Luego lo que realmente existe son los objetos, mientras que las clases se pueden considerar como patrones para crear objetos. Pero además, estos botones pertenecen a un objeto más grande que es el tapiz sobre el que están pintados. Este objeto de Windows habitualmente no es más que una instancia de la clase `CDialog` que permite crear objetos que contienen objetos que son capaces de interaccionar con el usuario de forma gráfica. Muchas de las interfaces que estamos acostumbrados a manejar en Windows son instancias de esta clase de objetos.

¿Por qué se trabaja de esta forma? ¿Qué ventajas tiene? Pues la ventaja más evidente es que al programar basándose en objetos, es posible reutilizar con facilidad el código realizado por otras personas. Así, cuando creamos un botón en Windows prácticamente lo único de lo que nos tenemos que preocupar es de indicar el título o texto que aparece en el centro del botón. Sin embargo, toda la gestión de dibujar el rectángulo, la sombra, de reaccionar ante un clic del ratón, etc. no es necesario que lo programemos puesto que ya lo ha hecho alguien por nosotros al constituir características comunes a todos los botones.

3.1. LAS CLASES EN C++

En el fondo, se puede decir que una clase en C++ es un mecanismo que permite al programador definir sus propios tipos de objetos. Un ejemplo clásico es el de los números complejos. Es bien conocido que en C no existe un tipo de datos para definir variables de tipo complejo. Por ello lo que se hacía era definir estructuras que contuviesen un campo que representara la parte real como un número de tipo `float`, y una parte imaginaria también de tipo `float` que representara la parte imaginaria. De esta forma se podía agrupar bajo un mismo identificador el conjunto de datos necesarios para definir un complejo.

En C, esto se realizaba por medio del uso de estructuras con estos dos campos:

```
struct complex
{
      float real;
      float imag;
};
void main()
{
      complex a;
      a.real=5.0f;
      a.imag=3.0f;
}
```

Posteriormente, si se quisiera trabajar cómodamente con este nuevo tipo de datos se definían una serie de funciones pensadas para manipular o extraer información de los mismos. Así, sería posible definir una función que permitiera extraer el módulo de cualquier complejo mediante el siguiente código:

```
#include <stdio.h>
#include <math.h>
struct complex
{
      float real;
      float imag;
};

float modulo(complex a)
{
      float m;
      m=sqrt(a.real*a.real+a.imag*a.imag);
      return m;
}

void main()
{
      complex a;
      a.real=5.0f;
      a.imag=3.0f;
      float m=modulo(a);
      printf("El modulo es: %f\n",m);
}
```

Se observa que mediante este sistema por un lado están los datos y por otro las funciones, aunque estas funciones estén definidas exclusivamente para trabajar con el tipo de datos `complex`.

Supóngase que existe un tipo de objetos en C++ llamado `complejo`. En ese caso el programa antes mostrado en C se podría reescribir, añadiendo el cálculo del módulo, de la siguiente forma:

```
#include "complejo.h"

void main()
{
      complejo a(2,3); //(1)
      float modulo = a.modulo(); //(2)
      cout<<a<<"tiene por modulo "<<modulo<<endl; //(3)
}
```

Fíjese en las distintas instrucciones propias de C++ que se han utilizado y que se han ido marcando con números:

1. Este modo de asignar los valores iniciales del número complejo recuerda a la llamada de una función de C. De hecho, como se verá más adelante en C++ la creación de cualquier variable (de una clase o de cualquier otra cosa) supondrá la llamada a una función especializada en inicializar dicho elemento, y por tanto, a veces es necesario pasar unos parámetros. En este caso, la función que construye el objeto sólo se encarga de asignar a la parte real e imaginaria los valores 2 y 3 respectivamente.
2. En C++ se pueden realizar asignaciones no constantes a variables recién creadas. A la variable módulo se le asigna el valor del módulo del número complejo. Observe bien como se hace. Se llama a una función que pertenece al objeto de tipo complejo y que se llama módulo y que no recibe argumentos. De esta forma, todas las funcionalidades de un objeto van adosados al mismo. Un número complejo, por el hecho de serlo, puede calcular su módulo.
3. Como se habrá visto en la exposición teórica, C++ tiene un nuevo modo de sacar y recoger datos de los periféricos. En este caso, el complejo a sabe como se tiene imprimir por pantalla, al igual que el número real modulo y que la cadena de caracteres del mensaje.

3.2. ELEMENTOS DE UNA CLASE EN C++

Las clases en C++ se pueden considerar como la evolución de las estructuras. Las clases permiten no sólo agrupar los datos –como ocurre en las estructuras- sino que además nos permiten incluir las funciones que operan con estos datos. Aunque más adelante se irán desgranando los componentes que se mencionan a continuación, es interesante el que aparezcan por primera vez enumerados en este guión.

Las clases en C++ tienen los siguientes elementos o atributos:

- **Un conjunto de datos miembro**. En el caso de los complejos, el número real que representa la parte real y el número real que representa la parte imaginaria, serán datos miembro de la clase `complex`. La información guardada en estos datos son lo que harán distinto un objeto de otro dentro de una misma clase. Aunque normalmente siempre tenemos datos en una clase, no es necesario que existan.

- **Un conjunto de métodos o funciones miembro**. Como se ha mencionado antes, serán un conjunto de funciones que operarán con objetos de la clase. Puede haber desde cero hasta tantos métodos como se consideren necesarios.
- **Unos niveles de acceso a los datos y métodos de la clase**. Supongamos que se ha creado una clase que permite almacenar un conjunto de números, de tal forma que es como una especie de saco, en donde se van agregando números, y después se pueden ir extrayendo de uno en uno o se puede preguntar cuantos números hay almacenados. En la clase existirán por tanto un conjunto de datos que permitirá ir almacenando los números que se introducen (*data*) y un dato que nos permite conocer cuántos números se han introducido hasta ese momento (*num*). No sería conveniente que en esta clase, el programador pudiera modificar libremente el valor de *num*, puesto que este valor indica el número de objetos que se han introducido y no un valor arbitrario. Por ello en C++ se puede establecer que algunos de los datos y funciones miembro de una clase no puedan ser utilizados por el programador, sino que están protegidos o son datos privados de la clase ya que sirven para su funcionamiento interno, mientras que otros son públicos o totalmente accesibles al programador. Esta cualidad aplicable a los datos o a las funciones miembro es lo que se denomina como nivel de acceso.
- **Un identificador o nombre asociado a la clase**. Al igual que en las estructuras, será necesario asignar un nombre genérico a la clase para poder hacer referencia a ella. Las palabras `CButton`, `CDialog`, ó `complejo` utilizadas en los ejemplos anteriores serán los identificadores de las clases que permiten crear objetos de tipo botón, diálogo o complejo.

3.3. INTRODUCCIÓN AL USO DE CLASES. EJEMPLOS STL.

3.3.1. La clase *string*

En este capítulo como introducción teórica es suficiente. El siguiente ejercicio, pretende mostrar la facilidad con la que se puede reutilizar el código en C++. Para ello utilizaremos una de las clases definidas en la librería **Standard Template Library** (STL), una librería estándar del lenguaje C++ que se puede encontrar en todas sus implementaciones, independientemente de la plataforma, el entorno o el sistema operativo, con la cual iremos viendo como nos permite manejar las cadenas de caracteres con gran comodidad. Por este motivo, en paralelo, iremos viendo como quedaría el mismo código si se realiza en C.

1. En primer lugar crearemos un proyecto de tipo consola, al cual añadiremos un fichero de código llamado ***principal.cpp*** que es sobre el que vamos a trabajar.
2. Para comenzar escribiremos el código siguiente en ***principal.cpp***:

```
//Ejemplo manejo string
#include <iostream>     //para usar cout
#include <string>

using namespace std;

void main()
{
      char cadenaOld[]="Hola Mundo";
      cout<<cadenaOld<<endl;
      string micadena("Hola Mundo");
      cout<<micadena<<endl;
}
```

Además del código introducido en el interior de la función `main`, ha sido necesario incluir ficheros de cabecera adicionales. Uno de lo que más utilizaremos a lo largo del curso `iostream` que podríamos decir que es el equivalente a `stdio.h` en C pero para C++. Para no complicar más, lo que se ha hecho es utilizar el sistema de salida aportado por C++, en la parte de C. El otro fichero de cabecera necesario para poder utilizar la clase `string` de las *STL* para manejar cadenas de caracteres.

Tras compilar el programa se ve que aparentemente hacen lo mismo ambas partes del programa.

En esta primera versión simplemente se saca por pantalla el mensaje “Hola Mundo”.

Se propone ir haciendo los siguientes añadidos tanto en C como en C++.

- **Calcular su longitud** (`length()` y `strlen()`)

```
      cout<<strlen(cadenaOld)<<endl;

      cout<<micadena.length()<<endl;
```

- **Comparar 2 cadenas**

```
      char cadenaOld1[]="Hola Mundo";
      char cadenaOld2[]="Hola Mundo";

      string cadena1("Hola Mundo");
      string cadena2("Hola Mundo");

      if(strcmp(cadenaOld1,cadenaOld2)==0)
            cout<<"Iguales"<<endl;

      if(cadena1==cadena2)
            cout<<"Iguales tambien, pero mas facil, no?"<<endl;
```

- **Concatenar dos cadenas.**

Hasta ahora, no se ha tenido en cuenta el hecho de que de antemano no se suele conocer la longitud de la cadena de caracteres que se va a utilizar. En este caso para hacer equivalentes ambas operaciones se va a escribir el código realmente equivalente entre la parte de C y la C++.

```
      char cadenaOld[]="Hola Mundo";
      string cadena("Hola Mundo");

//Codigo C
      char cadenaOld2[]=" Virtual",*pcadena;
      int tam_total=strlen(cadenaOld)+strlen(cadenaOld2)+1;
```

```
      pcadena=(char *) (malloc(sizeof(char)*tam_total));
      strcpy(pcadena,cadenaOld);
      strcat(pcadena,cadenaOld2);
      cout<<pcadena<<endl;
      free(pcadena);

//Codigo C++ con clase string de STL
      string cadena2(" Virtual"),resultado;
      resultado=cadena+cadena2;
      cout<<resultado<<endl;
```

Ahora comienza a percibirse lo cómodo de la programación orientada a objetos. Es interesante observar como la gestión del tamaño y la memoria necesaria para almacenar la información en C es responsabilidad del programador, mientras que en C++ pasa a ser responsable el objeto. De esta forma, lo único que se indica en C++ es que se quiere construir resultado como el resultado de sumar el contenido de dos cadenas. La `string resultado` ya se preocupa de preparar lo necesario para poder recibir esa información. En cambio en C, el programador debe medir la longitud de las cadenas, pedir espacio para almacenar una cadena tan larga como las dos más el carácter nulo, y proceder a hacer la copia de caracteres. El programador debe tener un conocimiento de cómo se trabaja con las cadenas de caracteres que no es necesario en C++, dado que el objeto es el responsable de hacer las operaciones que hacen referencia a la información que contiene.

Por eso, en el fondo se puede hacer lo que los métodos de la clase dejan hacer. Con este motivo se proponen los siguientes dos ejercicios para resolver, tanto en C como en C++.

1. **Buscar** una palabra dentro de la cadena de caracteres.
2. Insertar una subcadena dentro de otra.

Muy resumidamente estas son algunas de las operaciones que nos ofrece el objeto `string`:

- `operator[int i]` obtiene la letra "i" (igual que las cadenas de C)
- `operator+=` añadir a la cadena
- `insert(int pos, string str)` Inserta la cadena "str" en la posicion "pos"
- `erase(int n,int m)` borra "m" caracteres empezando desde el "n"
- `replace` reemplaza parte de una cadena
- `c_str` Obtiene el equivalente C de la cadena
- `find` Encuentra una subcadena, devuelve la posicion en la cadena, que sera menor que length() si la encuentra
- `substr` Obtiene una subcadena

3.3.2. La clase CFileDialog

Mucho más sorprendente puede ser el trabajar con elementos aun más complicados. Para poder escribir estas líneas de código en nuestro programa será necesario incluir el fichero de cabecera ***afxdlgs.h***. Esto nos permite utilizar objetos del tipo `CFileDialog`. Con la filosofía de la programación orientada a objetos, las cuatro siguientes líneas tienen un efecto realmente sorprendente:

```
//Ejemplo de dialogo de seleccion de ficheros
//Si sale una ventana de error, pulsar "Omitir"
#define _AFXDLL
#include <afxdlgs.h>
#include <iostream>

using namespace std;

void main()
{
      CFileDialog midlg(TRUE);
      CString micadena;
      midlg.DoModal();
      micadena=midlg.GetPathName();
      MessageBox(NULL,micadena,CString("Has seleccionado"),MB_OK);
}
```

3.4. INTRODUCIENDO CLASES EN LA APLICACIÓN

Partimos de la situación en la que se dejó el juego en el tema anterior. El proyecto consiste de una simulación en la que tenemos dos esferas, una de las cuales está animada, y un punto de vista que rota alrededor de la escena. Como se comentó anteriormente, en el código se observa como la parte lógica y la parte gráfica están desacopladas, y como la información se agrupó con la ayuda de dos estructuras. Una primera, llamada `Esfera`, encargada de contener la información relativa a una esfera, y la otra que contenía la información relativa al conjunto de la simulación, al cual denominamos como `Mundo`.

En este capítulo, sin que se realicen cambios aparentes importantes, se va a proceder a transformar el programa anterior en un programa de C++, en el que en vez de estructuras, tengamos clases, y en vez de variables globales, objetos. Se va a ir realizando la transformación poco a poco.

3.4.1. Conversión de estructuras a clases

En la práctica, se puede decir que una estructura es como una clase que contiene solo datos visibles por todo el mundo

Como se ha visto en la introducción los objetos pueden contener información privada, de forma que impiden que alguien desde el exterior modifique el contenido de esta información. Cuando un dato contenido en un objeto es accesible, se dice que es

público. A continuación se va a modificar el código de forma que en vez de estructuras, tengamos clases.

1. Cambiamos `struct` por `class` y añadimos antes de la enumeración de campos –pasan a llamarse atributos- el indicador de que son públicos: `public:`

De esta forma, la zona de código que contenía las estructuras pasa a ser:

```
class Esfera
{
public:
	float radio;
	float x;
	float y;
	unsigned char rojo;
	unsigned char verde;
	unsigned char azul;
};

class Mundo
{
public:
	float x_ojo;
	float y_ojo;
	float z_ojo;
};
```

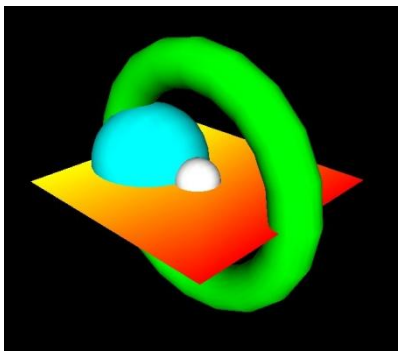


Figura 3-1. La aplicación no cambia, solo la estructuración de código

Si se compila y se ejecuta, se observa que no ha cambiado absolutamente nada. Es decir que el concepto de clase, incluye al concepto de estructura puesto que contiene datos. De igual forma el acceso a los atributos de la clase se realiza de igual forma a como se realiza el acceso a los campos de una estructura.

2. Comentamos la línea `public:` y observamos el mensaje del compilador:

Se generan unos cuantos errores, de entre los que cabe destacar los que tienen la forma siguiente:

```
'rojo' : cannot access private member declared in class 'Esfera'
```

Esto es porque por defecto se considera que los atributos de una clase son privados por lo que no se puede acceder a estos mediante el operador '`.`'.

Una vez visto esto, procedemos a quitar el comentario a la línea `public:`

3. Con la idea de ir generando una estructura de ficheros más acorde con la POO, vamos a trasladar la definición de las clases a unos ficheros independientes. La clase `Esfera` la declaramos en ***Esfera.h***, y la clase `Mundo` en ***Mundo.h***, por lo

que hemos de incluir ambos ficheros al principio de ***principal.cpp*** para que este siga funcionando. Para ello añadimos estos ficheros, pinchando con el botón derecho en el nombre del proyecto *Add->New Item*. Seleccionamos "*C/C++ Header File*", damos el nombre correspondiente al fichero.

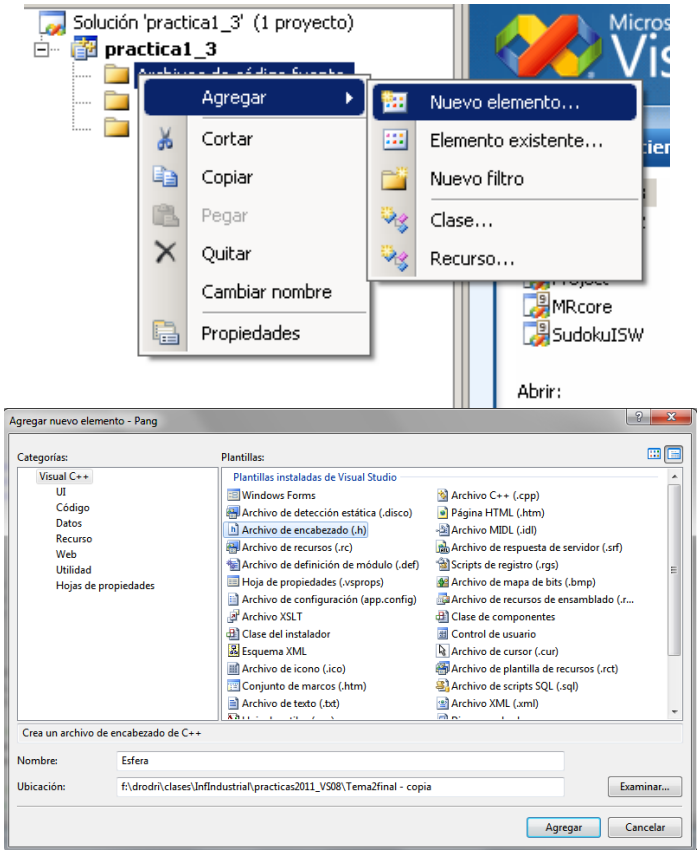


Figura 3-2. Añadir ficheros de cabecera " .h"

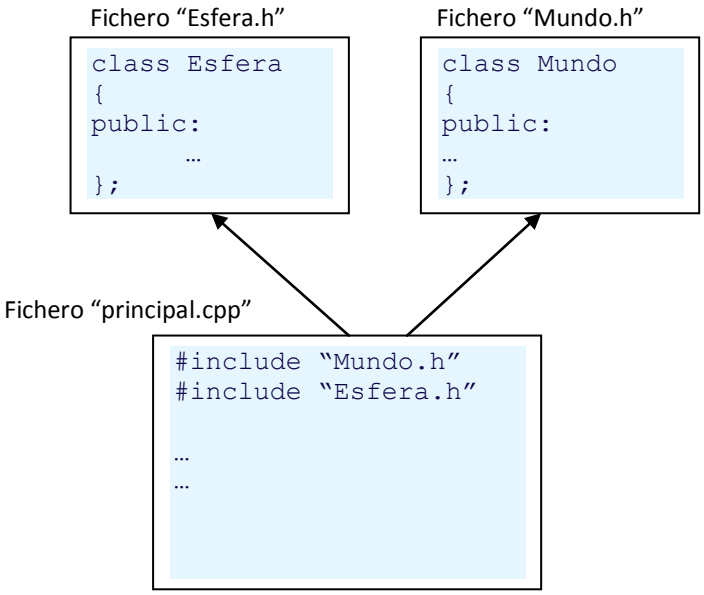


Figura 3-3. Estructuración del código en ficheros de cabecera

3.4.2. Métodos de una clase

Una clase contiene métodos que utilizan los datos del objeto para realizar operaciones. Una de las principales ventajas de una clase, es que puede contener métodos o funciones aparte de datos o variables miembro. Estos métodos son capaces de operar sobre los miembros de la clase, sin necesidad de que estos sean pasados por parámetro.

3.4.2.1 Establecer el color de una esfera

Veamos primero el ejemplo del método `SetColor()`**.** Lo normal es que cuando queremos indicar el color de una esfera, indiquemos el valor de las tres componentes de color dado que el color resultante es resultado de la combinación de estas tres componentes. Comenzamos por tanto con lo más sencillo que es crear un método para darle color a la esfera.

Dentro de la declaración de la clase escribimos una función denominada `SetColor()` con las siguientes líneas:

```
class Esfera
{
public:
	float radio;
	float x;
	float y;
	unsigned char rojo;
	unsigned char verde;
	unsigned char azul;
	//escribimos nuestro primer método
	void SetColor( unsigned char r,unsigned char v, unsigned char a)
	{
		rojo=r;
		verde=v;
		azul=a;
	}
};
```

Obsérvese, que un método debe ser ejecutado por un objeto de tipo `Esfera`. Dentro de la función, se hace referencia directa a los atributos del objeto que ha invocado el método y por tanto no se utiliza el punto. Es decir, la expresión `rojo=r` significa *al atributo rojo del objeto que me ha llamado se le asigna el valor del parámetro de la función r*.

Se va a utilizar ahora este método. En la función `main`, justo antes de comenzar el bucle principal de control (`glutMainLoop`), incluimos las líneas siguientes que se encargan de redefinir el color de las esferas que serán ahora de color rojo y verde:

```
	…
	glutKeyboardFunc(OnKeyboardDown);
	//inicialización de los datos de la simulación
	esfera.SetColor(200,0,0);
	esfera2.SetColor(0,200,0);

	//pasarle el control a GLUT,que llamara a los callbacks
	glutMainLoop();
```

Se podría decir que las llamadas a la función tienen la siguiente equivalencia, en la que se puede como la función opera sobre los miembros del objeto en cuestión:

```
esfera.SetColor(200,0,0);
// equivale a esfera.rojo=200;esfera.verde=0;esfera.azul=0;
esfera2.SetColor(0,200,0);
//equiavale a esfera2.rojo=0;esfera2.verde=200;esfera2.azul=0;
```

Esta función también se puede utilizar para escribir de forma más compacta la función `OnKeyBoardDown():`

```
      if(key=='r')
            esfera.SetColor(255,0,0);
      if(key=='g')
            esfera.SetColor(0,255,0);
      if(key=='b')
            esfera.SetColor(0,0,255);
```

Conviene separar la declaración de una clase de su definición, en particular si el proyecto es extenso. Al igual que en C, lo normal es tener un fichero de cabecera que sirve para informar de las funciones que existen en los distintos ficheros de código c, en C++ se procede de igual forma. El código conviene casi siempre que esté en un fichero "cpp". Se va a proceder ahora a pasar el código de la clase a un nuevo fichero denominado "***Esfera.cpp***". Para ello procedemos de la siguiente forma.

1. En *Menu->File->New*, escogemos crear un "*C++ Source File*", al que denominamos como "***Esfera.cpp***":
2. Agregado el fichero, incluimos al principio de "***Esfera.cpp***" la línea que incluye el fichero de cabecera que declara la clase esfera: `#include "Esfera.h"`
3. Copiamos el código de los métodos definidos en "***Esfera.h***", y añadimos delante del nombre de la función lo siguiente: "`Esfera::`", que sirve para indicar que es un método de esfera lo que se va definir. Con estas dos operaciones, en el fichero de código debemos tener lo siguiente:

```
#include "Esfera.h"
void Esfera::SetColor( unsigned char r,unsigned char v, unsigned char a)
{
      rojo=r;
      verde=v;
      azul=a;
}
```

4. Quitamos del fichero de cabecera el cuerpo de las funciones, dejando sólo el prototipo con un ';' al final. Por tanto el fichero de cabecera quedará como sigue:

```
#include "glut.h"

class Esfera
{
public:
      float radio;
      float x;
      float y;
      unsigned char rojo;
      unsigned char verde;
      unsigned char azul;
      void SetColor( unsigned char r,unsigned char v, unsigned char a);
};
```

3.4.2.2 Tamaño y posición de una esfera

De igual forma procedemos a crear un método `SetRadio()` para dar valor al radio, y e inicializamos el radio de las esferas junto con el color. La función `SetRadio()` asegurará que el radio que se le pasa es mayor que 0.1, y si no es así le asignará a la esfera este radio mínimo.

```
void Esfera::SetRadio(float r)
{
      if(r<0.1F)
            r=0.1F;
      radio=r;
}
```

Para poder modificar la posición x e y directamente, creamos el método `SetPos()` e inicializamos los valores de posición en el mismo sitio.

```
void Esfera::SetPos(float ix,float iy)
{
      x=ix;
      y=iy;
}
```

Los prototipos de métodos a añadir en la declaración de la clase `Esfera` en el fichero "***Esfera.h***" son:

```
      void SetRadio(float r);
      void SetPos(float ix,float iy);
```

Por tanto finalmente en el `main()`, aparecerán las líneas siguientes.

```
      //inicialización de los datos de la simulación
      esfera.SetColor(200,0,0);
      esfera.SetRadio(1);
      esfera.SetPos(2,0);
      esfera2.SetColor(0,200,0);
      esfera2.SetRadio(1);
      esfera2.SetPos(3,0);
      ...
```

3.4.2.3 Pintar una esfera

Si nos fijamos en la función que dibuja la esfera, observamos que la operación de pintar depende exclusivamente de datos que están contenidos todos dentro de la esfera. Es lógico entonces pensar que la esfera podría responsabilizarse ella misma de su dibujo. Añadimos el siguiente método a la clase `Esfera`, resultante de copiar el código de la función `Dibuja()` para esferas, y eliminamos la función antigua que recibía la esfera como un parámetro:

```
//eliminamos la antigua version
void Dibuja(Esfera e)
{
  glColor3ub(e.rojo,e.verde,e.azul);
  glTranslatef(e.x,e.y,0);
  glutSolidSphere(e.radio,20,20);
  glTranslatef(-e.x,-e.y,0);
}
```

```
//mejor como método de Esfera
void Esfera::Dibuja()
{
  glColor3ub(rojo,verde,azul);
  glTranslatef(x,y,0);
  glutSolidSphere(radio,20,20);
  glTranslatef(-x,-y,0);
}
```

Figura 3-4. Transformación de función con parámetro a método de clase

Fíjese en como el código de dentro de la función, se simplifica, al no tener que preceder a cada dato del nombre del objeto en cuestión, ya que al ser un método de clase, tiene acceso directo a dichos miembros.

Después de eliminar la antigua función dibuja, modificamos el código de pintado de la función `OnDraw()` de forma que ahora las esferas llaman a su método de pintado, en vez de pasar a una función de pintado genérica una esfera como parámetro. Nótese como cambian las llamadas a la función:

```
//llamadas antiguas

      Dibuja(esfera);
      Dibuja(esfera2);

      glutSwapBuffers();
```

```
//llamadas a los métodos de Esfera

      esfera.Dibuja();
      esfera2.Dibuja();

      glutSwapBuffers();
```

Figura 3-5. Invocación a métodos de una clase

Puesto que estamos usando en el fichero ***Esfera.cpp*** las funciones de la librería GLUT, es necesario incluir el fichero de cabecera ***glut.h***, para que si cambiamos el orden en la inclusión de los ficheros no haya ningún problema en la compilación.

En este punto es importante destacar lo siguiente. Imagínese que ahora que se comienza un proyecto nuevo. Si quisiéramos utilizar las mismas esferas del proyecto actual, nos bastaría con incluir el fichero "***Esfera.h***" en este nuevo proyecto, dado que el código va incluido con la clase. Si esto mismo lo hiciéramos con la estructura utilizada en C, tendríamos que trasladar también las funciones que están definidas a lo largo del código y que sirven para la realización de las operaciones básicas de las esferas.

3.4.2.4 Mover una esfera

De igual forma a como ocurre con la operación de pintado, la esfera se mueve (se hincha y deshincha) atendiendo sólo a sus datos. Se realiza la modificación del código para que la operación mover pase a pertenecer a la clase como método y de esta forma limpiar el código en "***principal.cpp***" de funciones que se refieren a las esferas.

```
//eliminamos la antigua version

void Mueve(Esfera* e)
{
      e->radio+=0.1f;
      if(e->radio>3)
            e->radio=0.5f;
}
```

```
//mejor como método de Esfera

void Esfera::Mueve()
{
      radio+=0.01f;
      if(radio>2)
            radio=0.5f;
}
```

La llamada a la función `Mueve()` también se simplifica:

```
//llamada antigua

      Mueve(&esfera2);
```

```
//llamada a método de Esfera

      esfera2.Mueve();
```

3.4.3. La clase Esfera

La clase Esfera queda declarada en el fichero de cabecera "***Esfera.h***" como sigue:

```
#include "glut.h"

class Esfera
{
public:
	float radio;
	float x;
	float y;
	unsigned char rojo;
	unsigned char verde;
	unsigned char azul;
	//tras los atributos, escribimos nuestros métodos
	void SetColor( unsigned char r,unsigned char v, unsigned char a);
	void SetRadio(float r);
	void SetPos(float ix,float iy);
	void Dibuja();
	void Mueve();
};
```

El fichero de implementación "***Esfera.cpp***" queda como sigue:

```
#include "Esfera.h"

void Esfera::SetColor(unsigned char r,unsigned char v,unsigned char a)
{
	rojo=r;
	verde=v;
	azul=a;
}
void Esfera::SetRadio(float r)
{
	if(r<0.1F)
		r=0.1F;
	radio=r;
}

void Esfera::SetPos(float ix,float iy)
{
	x=ix;
	y=iy;
}

void Esfera::Dibuja()
{
	glColor3ub(rojo,verde,azul);
	glTranslatef(x,y,0);
	glutSolidSphere(radio,20,20);
	glTranslatef(-x,-y,0);
}

void Esfera::Mueve()
{
	radio+=0.01f;
	if(radio>2)
		radio=0.5f;
}
```

3.4.4. La clase Mundo

Una vez vista la estructura, y comprobando su correcto funcionamiento, se va a pasar a realizar la misma operación para la clase Mundo. En este caso directamente escribiremos el código en un nuevo fichero ***Mundo.cpp*** que contendrá los métodos de la clase. Al finalizar la clase `Mundo` debe tener el siguiente fichero de cabecera:

```
class Mundo
{
public:
      float x_ojo;
      float y_ojo;
      float z_ojo;
      void RotarOjo();
};
```

Implementar el cuerpo del método RotarOjo() de la clase Mundo y realizar la llamada necesaria al método desde el "principal.cpp"

La estructuración final del código en distintos ficheros deberá ser la reflejada por el esquema de la figura 3.8:

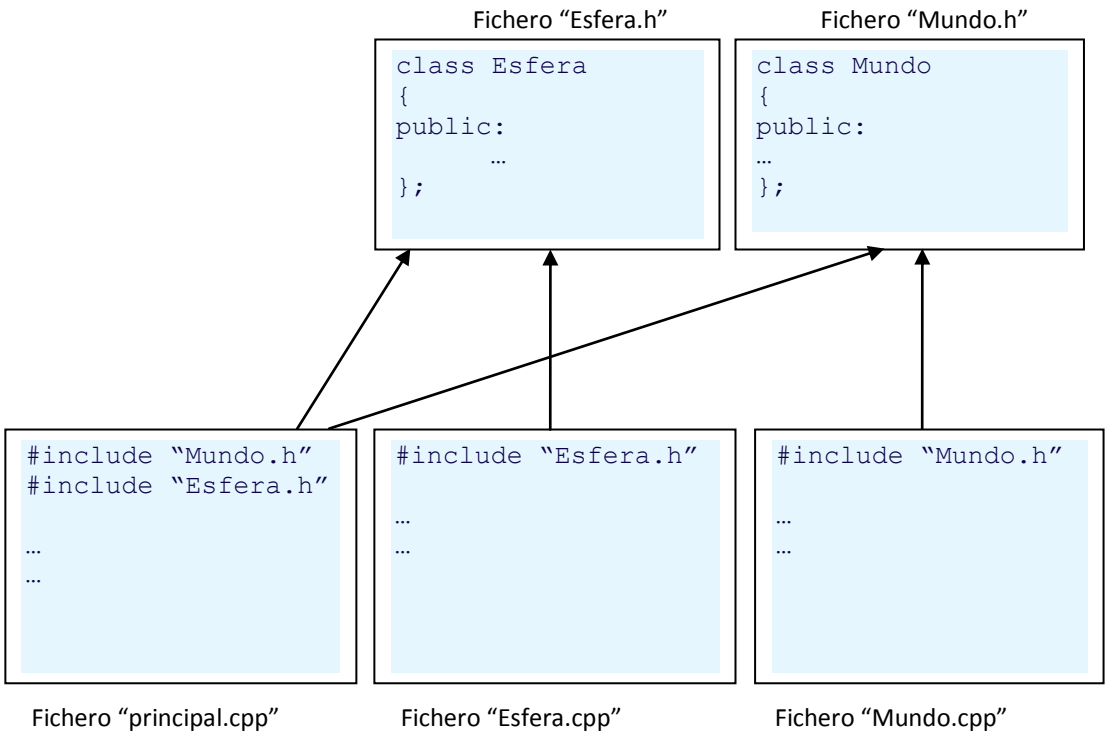


Figura 3-6. Estructuración de código fuente en ficheros

3.5. INTRODUCCIÓN A LA ENCAPSULACIÓN

A pesar de que hemos implementado la función `Esfera::SetRadio()` que asegura que el radio no puede ser negativo, aun son correctas las sentencias:

```
Esfera e;
e.radio=-3;
```

Lo que no tiene mucho sentido. Para proteger las variables miembro o atributos de una clase, existen los niveles de acceso a las variables y métodos de las clases. Cualquier atributo o método de una clase tiene uno de los siguientes accesos:

- Publico (`public`). El acceso al método o atributo esta permitido tanto desde fuera como desde dentro de la clase.
- Privado (`private`). El acceso al método o atributo únicamente esta permitido desde dentro de la clase.
- Protegido (`protected`). El acceso al método o atributo únicamente esta permitido desde dentro de la clase o desde dentro de una clase derivada de la misma

En nuestro caso, el objetivo es conseguir que el usuario (programador) de la clase `Esfera` pueda utilizar la clase, pero que no pueda cometer errores asignando un valor de radio negativo. Para ello convertimos los atributos de la clase `Esfera` en privados, mientras que dejamos los métodos como públicos:

```
#include "glut.h"

class Esfera
{
private:
	float radio;
	float x;
	float y;
	unsigned char rojo;
	unsigned char verde;
	unsigned char azul;

public:
	void SetColor( unsigned char r,unsigned char v, unsigned char a);
	void SetRadio(float r);
	void SetPos(float ix,float iy);
	void Dibuja();
	void Mover();
};
```

Si realizamos este cambio e intentamos compilar, el compilador nos informa de que en la función `OnKeyboardDown( )` se esta intentando acceder a una variable privada:

```
principal..cpp(60) : error C2248: 'radio' : cannot access private member declared in class 'Esfera'
```

De momento, eliminamos todas las referencias a atributos privados de la clase `Esfera` en la función `OnKeyboardDown( )`.

```
void OnKeyboardDown(unsigned char key, int x_t, int y_t)
{
	if(key=='r')
		esfera.SetColor(255,0,0);
	if(key=='g')
		esfera.SetColor(0,255,0);
	if(key=='b')
		esfera.SetColor(0,0,255);

}
```

A medida que necesitemos acceder a los datos internos privados de la clase, ya sea para consultarlos desde fuera o para modificarlos, desarrollaremos métodos para ellos. Por ejemplo si necesitáramos conocer el radio de una esfera añadiríamos el método `GetRadio()` a la clase `Esfera`. Esta forma de proceder se suele denominar como implementar métodos `Get` y `Set` para los atributos privados o protegidos

```
float Esfera::GetRadio()
{
	return radio;
}
```

3.6. EJERCICIOS PROPUESTOS

Poco a poco hemos limpiado el código, y cada vez cada objeto es más autónomo. El ejercicio propuesto sigue el razonamiento siguiente:

Si tenemos una simulación, esta solo tiene sentido si sabe lo que se está simulando. La clase `Mundo`, es quien debe saber que cosas se van pintar y que cosas no. Por tanto, lo lógico es que las esferas pertenezcan a la clase `Mundo`, y que el programa principal vaya pidiendo el mundo que se pinte o que se actualice.

Por tanto:

1. Incluir dos atributos de tipo `Esfera` en la clase `Mundo`. Para evitar los problemas derivados de la posible redefinición de clases, en los ficheros *.h incluir al principio del fichero las directivas de preprocesador siguientes:

```
#ifndef _ESFERA_H
#define _ESFERA_H

#include "glut.h"

class Esfera
{
public:
	…
};

#endif
```

2. Crear un método en la clase `Mundo` denominado `Inicializa()` que se encargue de dar valores iniciales al color posición y radio de las esferas que contiene la simulación, así como de los valores iniciales de la posición del ojo.

Llamar este método justo antes de llamar a `glutMainLoop` en el `main` del programa.

3. Crear un método `Dibuja()` de la clase Mundo que se llamará en la función `OnDraw()` del fichero "***principal.cpp***", y que será responsable de situar la vista, y de pintar las esferas que contiene el Mundo.
4. Crear el método `Mueve()` de la clase `Mundo` que se llamará en el `Timer()` del fichero "***principal.cpp***" para actualizar tanto la posición de la vista como del movimiento de las esferas.
5. Crear el método `Tecla(unsigned char key)` de la clase Mundo que se llamara desde el `OnKeyboardDown()` del fichero "***principal.cpp***", pasándole el valor de la tecla pulsada por el usuario
6. Limpiar el código de forma que lo único que se ve desde las funciones definidas en "***principal.cpp***" es el objeto `mundo` de la clase `Mundo`.

4. ANÁLISIS Y DISEÑO ORIENTADO A OBJETOS (A/DOO). UML

4.1. INTRODUCCIÓN

En este tema se introduce el Análisis y Diseño Orientado a Objetos y el UML (dentro del contexto del Proceso Unificado) utilizando una aplicación conocida como objetivo: la programación del juego del Pang.

En este capítulo se aborda el comienzo de este juego desde la perspectiva del A/DOO, y se realizará un diseño no completo y una implementación preliminar, sobre los que se irá trabajando en sucesivos capítulos hasta la consecución final del juego. Se recuerda que el Proceso Unificado es un proceso iterativo, y por lo tanto no pretende hacer un diseño exhaustivo y perfecto antes de la implementación, sino ir mejorando en sucesivas iteraciones. En este proyecto iremos iterando en los sucesivos capítulos a medida que se introducen nuevos conceptos (generalización y herencia, dependencias, asociaciones, creación y destrucción de objetos, diseño con patrones, etc.) hasta llegar al resultado final. Por lo tanto el punto de partida de cada capítulo será el resultado del capítulo anterior, y sobre ello se diseñará y se añadirán, cambiarán, modificarán, suprimirán y mejorarán distintos aspectos.

El objetivo de este capítulo es que el lector se familiarice con las técnicas y diagramas UML más comunes del A/DOO, pero también que realice la implementación de los mismos de forma paralela, de forma que se practique la creación de clases nuevas, la implementación de relaciones, la adición de atributos y métodos, etc.

4.1.1. Configuración del proyecto

En el capítulo anterior hemos concluido con un proyecto en el que se implementan las funciones básicas para trabajar con la GLUT, así como las clases `Esfera` y `Mundo`. También teníamos una escena en la que el punto de vista gira entorno al origen, y en la escena se ve un toro, un par de esferas y un plano. En la carpeta de código correspondiente a este tema, se ha eliminado el toro, la rotación respecto del origen y las esferas. La razón es que en este capítulo se explica primero el diseño y luego la implementación de dicho diseño, para muchas de las clases necesarias en el juego, así

que se realizará el desarrollo de la clase `Esfera` en paralelo con el resto, siguiendo la metodología que se expone. Además, se presenta una nueva estructuración de ficheros de código en subcarpetas, lo que requeriría demasiados cambios, así que es más sencillo implementar la clase desde cero, copiando fragmentos de código si fuera necesario.

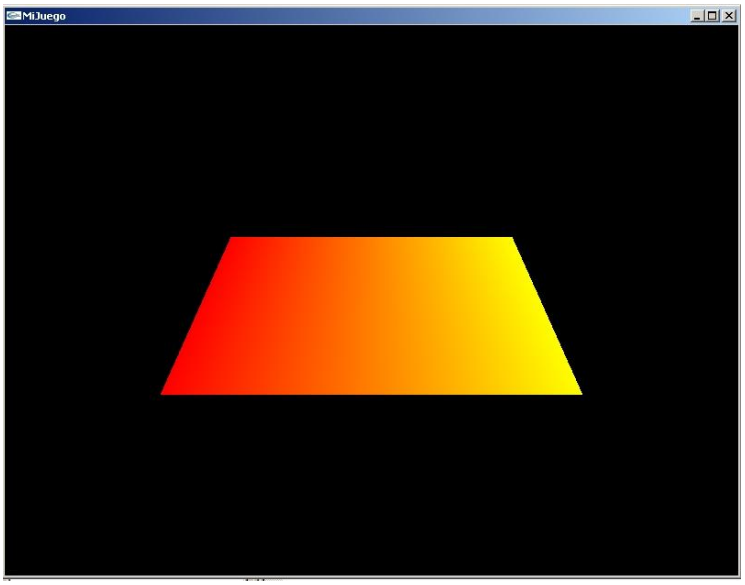


Figura 4-1. Punto de partida de este capítulo

Dentro de la carpeta del proyecto se puede encontrar la siguiente estructura de subcarpetas y ficheros, típica de la mayoría de proyectos software:

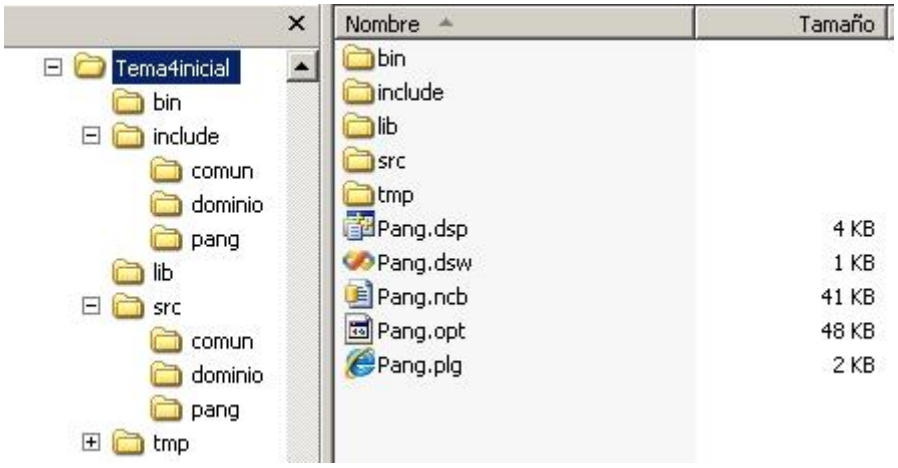


Figura 4-2. Estructura de directorios del proyecto

En la carpeta raíz solo tenemos los archivos de Visual. La subcarpeta ***"bin"*** contiene el ejecutable ***"Pang.exe"***, así como la librería dinámica necesaria para su ejecución ***"glut32.dll"***. Estos serían los archivos estrictamente necesarios para la ejecución del juego.

La carpeta ***"include"*** contiene todos los ficheros de cabecera ".h" del proyecto, estructurados a su vez en 3 subcarpetas:

- comun: archivos comunes a distintas posibles aplicaciones que desarrollemos aparte de los juegos. El fichero de cabecera de la GLUT ***"glut.h"*** por ejemplo va

aquí. Cuando posteriormente desarrollemos una clase `Vector2D` para las operaciones vectoriales, el fichero ***"Vector2D.h"*** también lo pondremos aquí.

- dominio: ficheros correspondientes a elementos básicos del dominio, como la `Esfera`. Se podría decir que los objetos del dominio son altamente reutilizables cuando estemos en el mismo ámbito, es decir desarrollando juegos similares.
- pang: ficheros específicos del juego Pang y particularizaciones que no son altamente reutilizables en otros juegos.

La carpeta ***"lib"*** contiene las librerías estáticas necesarias en el proyecto, como la librería ***"glut32.lib"***.

La carpeta ***"src"*** contiene todos los ficheros de código fuente ***".cpp"*** del proyecto, estructurados con los mismos criterios que la carpeta ***"include"***.

Por último la carpeta ***"tmp"*** contiene todos los archivos temporales de compilación, los objetos binarios y archivos necesarios por el Visual C++ para la compilación del proyecto. Como indica su nombre (temporal = tmp) el contenido de esta carpeta no es imprescindible. Podemos borrar su contenido completo sin problemas para transportar o almacenar nuestro proyecto si necesidad de gastar espacio extra.

Obviamente, toda esta estructuración requiere la modificación del proyecto a través de los *"Project Settings"*. Esta tarea ya ha sido realizada para comodidad del lector, pero los detalles se pueden encontrar en el anexo.

Además, cuando realicemos inclusiones en nuestro código, deberemos indicar en cuales de las subcarpetas se incluyen los ficheros, tal y como se muestra a continuación:

```
#include "pang/Mundo.h"
#include "comun/glut.h"
```

Esta subdivisión común-dominio-pang corresponde a la división del proyecto en paquetes, tal y como muestra el diagrama siguiente:

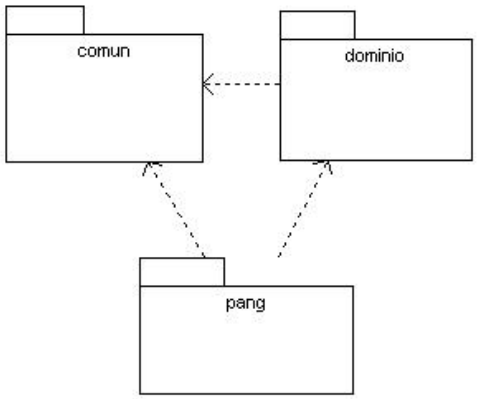


Figura 4-3. Diagrama UML de paquetes

Este diagrama también ilustra las dependencias entre los distintos paquetes. Así el paquete común no depende de los otros, sino al contrario. Es decir, el Pang necesita conocer las funciones de GLUT, pero no al contrario, así como del dominio, que a su vez necesita conocer funciones comunes.

4.1.2. El juego del Pang

El juego del Pang consiste en que el jugador debe destruir una serie de pelotas que rebotan en la pantalla, mediante unos disparos. Cada vez que uno de estos disparos impacta en una pelota, la divide por la mitad, excepto si la pelota es muy pequeña, que desaparece. Si una de las pelotas impacta al jugador, entonces le mata, el jugador pierde la vida y vuelve a comenzar la pantalla. Una pantalla termina cuando se logran destruir todas las pelotas sin haber sido impactado, y como consecuencia se pasa a otra pantalla posterior de mayor dificultad. En la pantalla hay plataformas, que son planos en los que rebotan las pelotas. Las pelotas también rebotan entre ellas y con los límites de la pantalla. La mejor forma de comprender el juego es verlo en ejecución (incluido en el código de apoyo del capítulo):

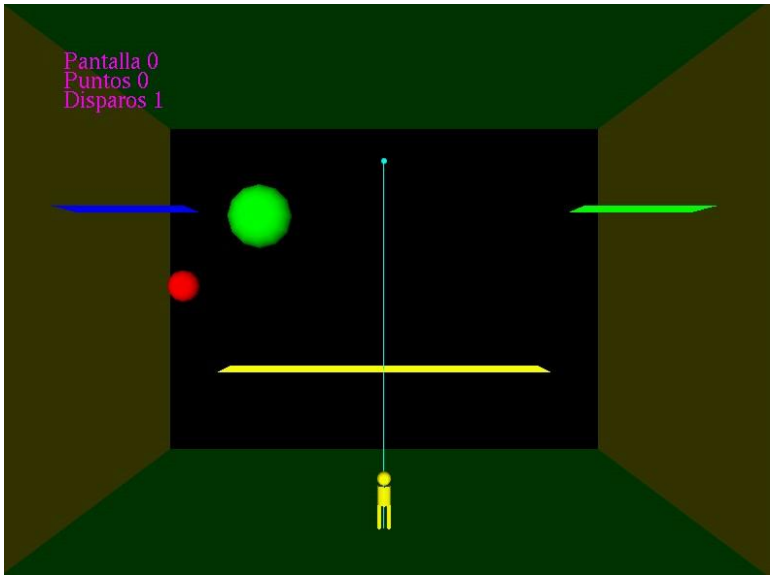


Figura 4-4. Juego del Pang

4.2. REQUISITOS

Dado que esta es una aplicación muy sencilla, de la que ya tenemos un ejemplo "para ver que es lo que queremos", y además el cliente final somos nosotros mismos, no se hará mucho hincapié en esta disciplina. Se recuerda no obstante que la disciplina de requisitos tiene una gran importancia, ya que gran cantidad de los costes asociados a los proyectos software tienen su origen en los requisitos.

Descubrir el actor principal es fácil: el jugador que juega al juego. El caso de uso del sistema también es obvio: el jugador lo que quiere del sistema es entretenerse un rato: "jugar". El caso de uso en formato breve podría ser el siguiente:

El jugador comienza una partida. Se desplaza de izquierda a derecha esquivando unas pelotas que rebotan por la pantalla. El jugador dispara para destruir las pelotas, que se dividen en dos al ser alcanzadas, o desaparecen si son muy pequeñas. El jugador gana la partida si consigue destruir todas las pelotas sin ser alcanzado por ellas.

El diagrama UML que representa este caso de uso es el siguiente:

Figura 4-5. Caso de uso principal de la aplicación

4.3. OBJETIVOS DEL CAPÍTULO

En este capítulo no se aborda el caso de uso visto anteriormente en su totalidad, sino sólo una parte. Se puede decir que los objetivos o requisitos parciales de esta iteración son los siguientes:

- Crear una estructura básica del proyecto, consistente en una serie de clases con sus atributos y métodos.
- Se tiene que representar en la pantalla (puede ser simplificado) todo lo existente en la aplicación de muestra, es decir se tiene que ver al menos una pelota, un bonus, una plataforma, la caja contenedora de la pantalla, el jugador y un disparo. No hace falta que el dibujo del jugador sea muy detallado, se puede aproximar por ejemplo por una esfera o un cilindro.
- El movimiento básico de los cuerpos debe ser implementado, suponiendo que no hay interacciones entre ellos. Los objetos en la pantalla serán libres, no chocaran unos con otros, el disparo sale de unas coordenadas iniciales cualesquiera (dadas en código), no del jugador, y lo mismo sucede con el bonus.

4.4. ANÁLISIS ORIENTADO A OBJETOS (AOO)

El análisis orientado a objetos (AOO) consiste en el estudio del problema y su entorno, así como su modelado y concreción en la documentación, fundamentalmente en el artefacto conocido como Modelo del Dominio, que es un diagrama de clases UML. Es importante destacar que en esta parte no se está escribiendo software, ni diseño del software, sólo estudiando el dominio del problema.

4.4.1. Identificación de algunas clases

El primer paso en el AOO es la detección de las clases de objetos que hay en el mundo o entorno del problema. En este caso, el mundo o problema es el mundo virtual del juego, en el que existen pelotas, disparos, plataformas, etc. Se puede pensar en el juego como si fuera un juego físico de verdad, como el tenis o el fútbol. En este sencillo ejemplo identificar un primer conjunto de clases es muy sencillo.

- Pared: las paredes en las que rebotan las pelotas.
- Bonus: el regalo que cae de vez en cuando al dividir una esfera en dos con un disparo
- Hombre: el muñequito que se controla con las teclas
- Esfera: las pelotas que rebotan y que podrían impactar con el jugador o ser alcanzadas por un disparo.
- Disparo: el disparo que efectúa el jugador que le sirve para ir destruyendo las pelotas.
- Caja: Los limites en los que jugamos. Se podría considerar que esta compuesta por un conjunto de paredes.

Nótese como la clasificación se realiza con nombres de clase en **singular**. Con estas clases hemos establecido un mecanismo que permite clasificar cualquier objeto del juego. Es decir, podemos señalar un objeto del juego y decir “esto es una Esfera” (en **singular**), o “esto otro es un disparo” (también en **singular**). Además, todos los objetos del entorno pueden ser clasificados en alguna de la clases, no existe nada que no sepamos lo que es.

4.4.2. Identificación de relaciones entre clases

El siguiente paso es analizar las relaciones que aparecen entre las clases que hemos identificado anteriormente. Las relaciones entre las clases también aparecen de forma intuitiva en este ejemplo:

- Un disparo puede impactar con una esfera
- Una esfera puede impactar con el hombre
- Las esferas rebotan con las paredes (sean de la caja o no)
- Las esferas sueltan bonus
- El hombre realiza disparos
- El hombre no se puede salir de la caja
- La caja está compuesta por cuatro paredes

En el AOO no es demasiado importante encontrar todas las relaciones entre clases y detallarlas cuidadosamente. No hay que perder demasiado tiempo en esta tarea, intentando especificar todas las relaciones, analizar sus multiplicidades o cualificarlas como asociaciones, composiciones, etc.

4.4.3. Modelo del Dominio

El artefacto o documento más importante del AOO es el modelo del dominio, en el que figuran de forma gráfica las clases y relaciones descubiertas anteriormente. Este diagrama es un modelo conceptual del problema a resolver con nuestro software. Con la información anterior construimos el diagrama del Modelo del Dominio, teniendo en cuenta que:

- El diagrama no es perfecto, se irá perfeccionando en el desarrollo iterativo, a medida que se descubran nuevas clases y relaciones.
- Como se ha comentado, no es importante dedicar tiempo a detallar las relaciones, ni especificar si son agregaciones, composiciones, generalizaciones, realizaciones o dependencias (aunque se podría), ya que lo que añadiría este trabajo no es significativo. Por lo tanto todas las relaciones se dejan tal y como se detallan en el diagrama (Figura 4-6).

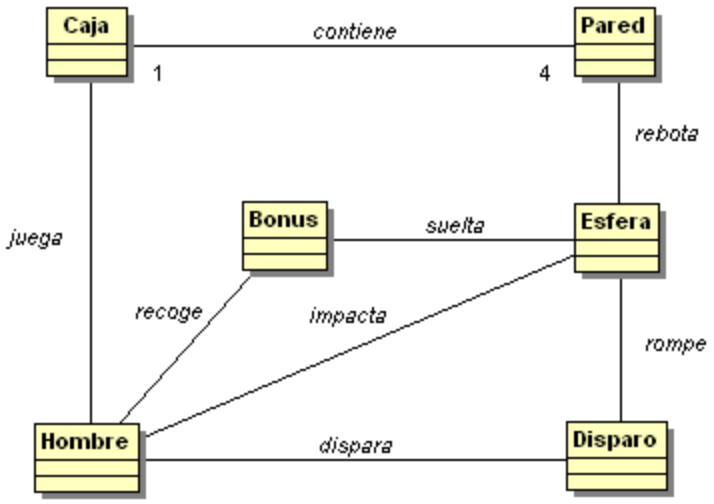


Figura 4-6. Modelo del Dominio

4.5. Diseño Orientado a Objetos (DOO)

En este punto comenzamos a diseñar e implementar nuestra solución software para éste problema en concreto: el desarrollo del juego del Pang. En este apartado se va a utilizar el DOO para empezar a diseñar la aplicación, siguiendo una serie de pasos. En cada uno de ellos se irán traduciendo a código los diagramas desarrollados, de forma que se pueda apreciar la relación existente entre ambos. Nótese bien que esto no es una práctica habitual en DOO ni el proceso unificado, sino una ayuda al alumno para asimilar los conceptos. Generalmente se haría primero todo el diseño (y los diagramas) presentados en este apartado, y una vez completo (que no perfecto, recuérdese que el diseño software es un proceso iterativo) se implementa el código. Por ese motivo a lo largo del capítulo se irá diferenciando lo que consiste en diseño y lo que es una implementación.

4.5.1. Primer paso: Clases

En primer lugar vamos a decidir cuántas y qué clases de diseño software va a tener nuestra aplicación, las cuales serán posteriormente implementadas en código C++,. El artefacto o diagrama fundamental para este propósito es el **Diagrama de Clases de Diseño (DCD)**, en el que se mostrarán nuestras clases y nuestro diseño. Es como una especie de plano de lo que más tarde se implementará procediéndose de forma análoga como ocurre en el mundo de la construcción. Como en la construcción de una casa, primero se dibuja el plano y luego se construye la casa, basándose en ese plano.

En este punto es necesario el uso de una herramienta CASE[3], en concreto una herramienta de desarrollo de diagramas UML. Los diagramas de este capítulo han sido desarrollados con BOUML[4] que es una herramienta gratuita y lo suficientemente completa para nuestros objetivos. La opción más estándar es seguramente el uso de la herramienta *Rational Rose*. En cualquier caso, el ADOO es una disciplina independiente de la herramienta, lo importante es saber pensar, analizar y diseñar y luego transmitirlo a los demás por medio de unos diagramas. El escoger una u otra herramienta es irrelevante.

4.5.1.1 Diseño

Vamos a comenzar nuestro diseño creando una clase (ahora si estamos hablando de diseño software) para cada uno de los objetos existentes en el Modelo del Dominio, y que además se denominen igual que estas. De esta forma, es mucho más comprensible el diseño, al coincidir con la nomenclatura y conceptos existentes en el dominio, lo que se conoce como "**reducción en el salto de la representación**". No obstante, hay que tener en cuenta que son unas clases diferentes, las clases de nuestro DCD no tienen porque corresponder 100% con el modelo del dominio.

De hecho, si nos fijamos en nuestro proyecto, tenemos una clase denominada Mundo, que nos sirve para definir algunas variables auxiliares como el punto de vista de la cámara y para contener todos los objetos del mundo. Esta clase que no aparece en el Modelo del Dominio, pero si aparece en nuestro código, es parte de nuestra solución y es necesario incluirla también en nuestro DCD.

Al añadir las clases nuestro diagrama de clases de diseño DCD queda como sigue:

[3] Las herramientas CASE (Computer Aided Software Engineering, Ingeniería de Software Asistida por Ordenador) son diversas aplicaciones informáticas destinadas a aumentar la productividad en el desarrollo de software reduciendo el coste de las mismas en términos de tiempo y de dinero

[4] http://bouml.free.fr

Figura 4-7. Diagrama de Clases de Diseño (DCD) solo con clases

4.5.1.2 Implementación

La mayoría de herramientas CASE son capaces de generar código automáticamente desde un diagrama de clases de diseño DCD, recomendamos para el principiante no hacerlo, sino desarrollar manualmente sus clases desde la herramienta de desarrollo correspondiente. En nuestro caso, procederemos en este capítulo a crear manualmente las clases para familiarizarse y practicar con las utilidades del propio Visual Studio para crear clases, añadir funciones y atributos.

Para crear las clases utilizaremos las ayudas del Visual Studio. Pulsando con el botón derecho del ratón en el *Workspace->ClassView->"…… classes"*, aparece la opción de *New Class*, que muestra el siguiente dialogo de la figura 4-8:

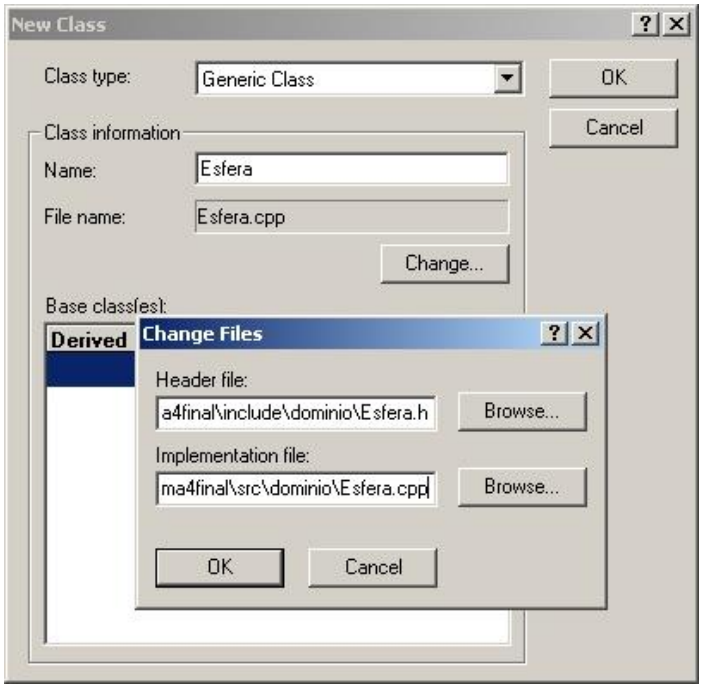


Figura 4-8. Asistente para creación de nuevas clases en Visual Studio 6.0

Poner el nombre de la clase deseado, y pulsar OK. Se crearan automáticamente dos ficheros, uno de cabecera .h y otro .cpp, que contienen la declaración de la clase y el cuerpo del constructor y destructor de la clase. El nombre de los ficheros tiene el mismo nombre de la clase, excepto si la primera letra es C (de Clase), que se omite. Así si a una clase la denominamos `Cuadrado`, los ficheros que define automáticamente serán

"uadrado.*". Si se quiere corregir esto, denominar la clase `CCuadrado`, o definir manualmente los ficheros con el botón *"Change…"*

En nuestro caso, es **importante** pulsar este botón para definir la localización de nuestros archivos. Seleccionar las respectivas subcarpetas dentro de ***src*** e ***include*** para cada nueva clase. Todas las clases generadas en este punto pertenecen al dominio. Recuérdese que la clase `Mundo` ya existe y no hay que crearla

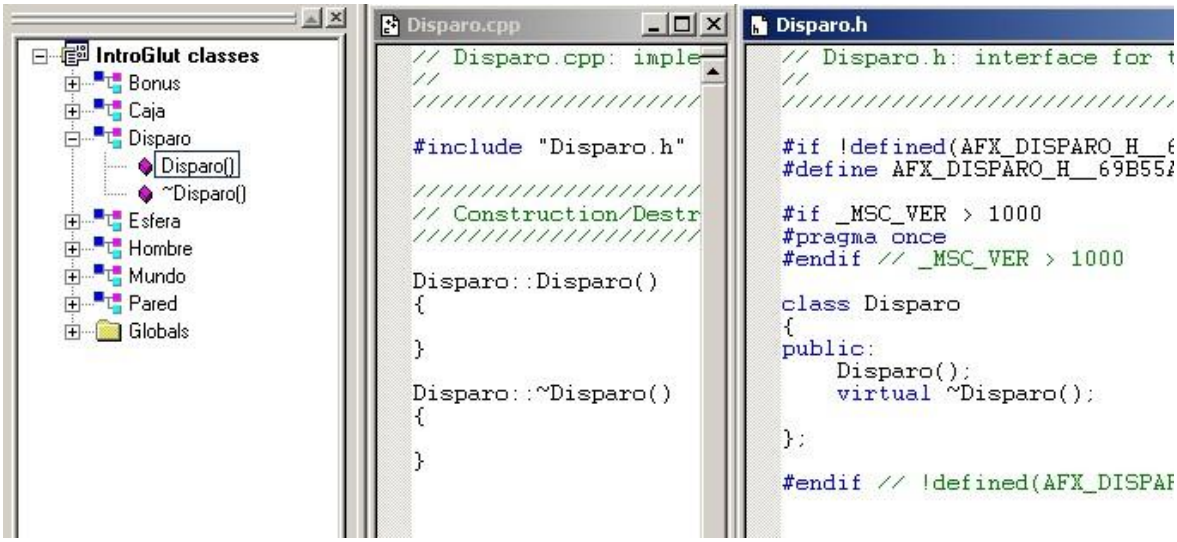


Figura 4-9. Clases software

El resultado final en nuestro proyecto de Visual Studio es el mostrado en la figura anterior. Nótese además como haciendo doble click con el ratón en el espacio de trabajo *Workspace->ClassView* en las clases y métodos se puede navegar por los ficheros de código fuente rápidamente.

4.5.2. Segundo paso: Atributos

Las clases en el apartado anterior están vacías tanto de atributos como de métodos. En este apartado diseñamos que atributos tendrá cada clase, y finalmente realizaremos la implementación de dichos atributos

4.5.2.1 Diseño

4.5.2.1.1 Atributos de tipos primitivos

Está claro que las esferas tienen distinto radio, por lo que uno de los atributos de la clase `Esfera` debe ser el radio, que es una magnitud real (utilizamos un `float`). También hay esferas de diferente color, con lo cual el color es otro atributo. Utilizamos para representarlo tres variables rojo (red), verde (green) y azul (blue) para almacenar las tres componentes de color (en el rango 0-255), y por tanto de tipo `unsigned char` (uchar). Todo esto ya fue realizado en el capítulo anterior. Estas variables son de tipo público (`public`). Los indicadores en un diagrama UML para representar el acceso a miembros de una clase es (+) para acceso público, (-) para acceso privado y (#) para acceso protegido.

También las paredes tienen distintos colores, mientras que los disparos y el hombre tienen siempre el mismo color, y el bonus una variación aleatoria de colores. Por tanto solo añadimos las variables de color necesarias a la clase `Pared`. Hay que tener en cuenta que no es incorrecto añadirlas a las clases `Hombre` y `Disparo`, pero de momento no se considera necesario (puede que más adelante si sea necesario, si queremos efectuar disparos de diferentes colores, por ejemplo).

Asimismo, el hombre tendrá un tamaño (altura), los disparos un calibre (radio) y el bonus (dibujado con un cuadrado) un lado. Todos ellos de tipo `float`. De momento podemos dejar los atributos como públicos, por simplificar, aunque en la mayoría de los casos es recomendable que sean privados o protegidos y acceder a ellos mediante las funciones correspondientes (tipo `Set` y `Get`). El DCD queda como sigue:

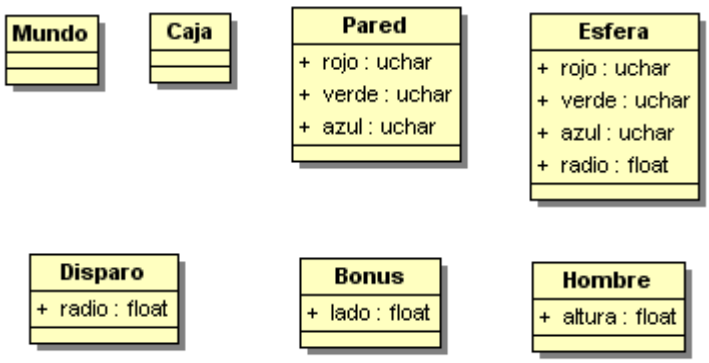


Figura 4-10. DCD con atributos de tipos simples

4.5.2.1.2 Atributos de tipos compuestos.

Falta por representar en las clases anteriores quizás la información más importante: las cosas del juego tienen una posición en la pantalla, y muchas de ellas también una velocidad. Una posición (en el plano XY en el que se desarrolla el juego) se puede representar por dos atributos de tipo `float` en las distintas clases, tal y como habíamos hecho anteriormente con la esfera. De la misma forma podemos utilizar dos atributos `float` para almacenar la información de velocidad de un objeto. Por ejemplo:

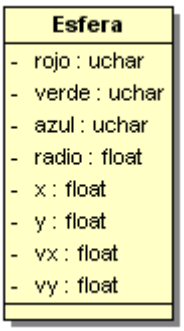


Figura 4-11. Clase Esfera con variables (x, y) para la posición y (vx,vy) para la velocidad

Las ecuaciones de movimiento de una esfera en un campo gravitacional se pueden escribir como:

$$\begin{cases} x = x + v_x t \\ y = y + v_y t - \frac{1}{2} g t^2 \end{cases}$$

$$\begin{cases} v_x = v_x \\ v_y = v_y - gt \end{cases}$$

Donde *t* es el tiempo transcurrido y *g* la aceleración de la gravedad terrestre considerando como vertical el eje Y. No obstante, nuestros conocimientos de física nos permiten formular las anteriores ecuaciones de una forma más compacta y a la vez genérica si usamos para ello la notación vectorial:

$$\mathbf{x} = \mathbf{x} + \mathbf{v}t + \frac{1}{2}\mathbf{a}t^2$$

$$\mathbf{v} = \mathbf{v} + \mathbf{a}t$$

donde

$$\mathbf{x} = \begin{pmatrix} x \\ y \end{pmatrix}; \mathbf{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}; \mathbf{a} = \begin{pmatrix} a_x \\ a_y \end{pmatrix}$$

Y en el caso concreto de un móvil en tiro parabólico terrestre $a_x = 0$ y $a_y = -9.8$.

Observemos que el segmento tendría dos pares de coordenadas representando sus extremos, y los otros objetos móviles también tienen que tener definida su posición, velocidad y aceleración en un mundo bidimensional. Podemos observar que esta información bidimensional 2D tiene entidad propia, es un tipo de dato en si mismo y por lo tanto puede y debe ser representada mediante la clase correspondiente. Esta clase es la clase `Vector2D`, que permite almacenar las componentes X e Y de un vector (o las coordenadas de un punto) en el espacio. Se podría decir que la clase `Vector2D` es como el tipo de dato simple `float`, pero bidimensional. Además, se verá en el futuro como esta clase permite encapsular las operaciones matemáticas relacionadas con vectores gracias a la sobrecarga de operadores, de tal forma que el código que realicemos dentro de las otras clases quedara mucho más compacto y fácil de entender. Es decir, utilizar esta clase nos permite compactar el código de igual forma que compactamos la ecuación anterior.

De momento, creamos una nueva clase denominada `Vector2D` que tiene dos atributos de tipo `float`, denominados `x` e `y`. Este es un buen ejemplo de una clase "Software" que no existe en el Modelo del Dominio. De hecho, es una clase claramente reutilizable en otros proyectos distintos a los juegos, como en cálculos físicos, científicos, etc.

Añadimos un atributo de posición, velocidad y aceleración a cada uno de los objetos móviles, siendo estos atributos de tipo `Vector2D`. También añadimos los dos atributos `limite1` y `limite2` a la clase Pared, que constituyen la definición de sus extremos, e igualmente son de tipo `Vector2D`. Seguimos dejando por comodidad todos los atributos como públicos. Obviamente, eliminamos los atributos de posición (x, y) y velocidad (vx, vy) de la esfera. El diagrama de clases de diseño DCD quedará entonces como sigue:

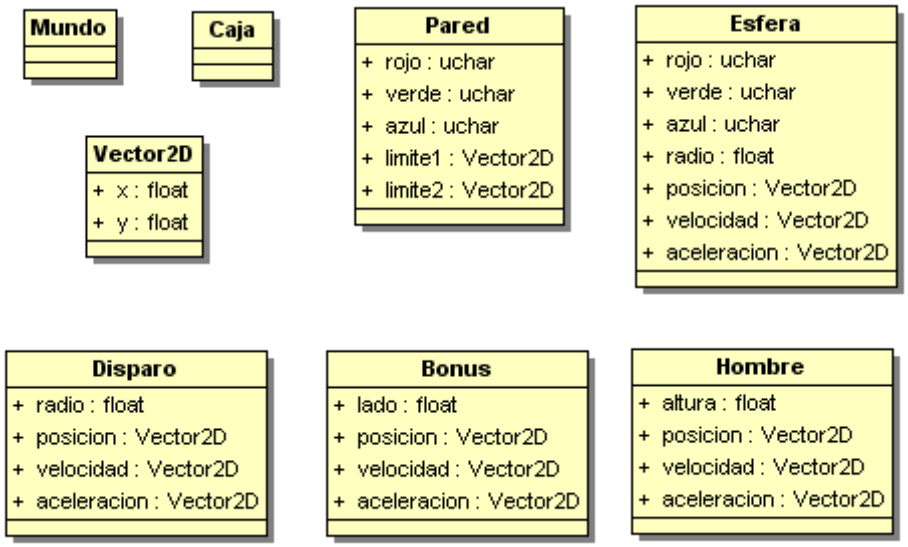


Figura 4-12. DCD con atributos de tipos compuestos (mejor)

Aquí, la relación entre las clases `Esfera` y `Vector2D` (por ejemplo) aparece solo implícitamente: al añadir un atributo de tipo `Vector2D` a la clase, obviamente se establece una relación entre ellas. Esta relación se puede entender y representar como un tipo de relación muy común: la relación de **composición**. Si en nuestro diseño consideramos importante resaltar que nuestro modelo de una esfera está compuesto por el modelo de su posición (y por otras cosas como su radio), podemos representarlo en el diagrama de clases de la siguiente forma, con un rombo lleno en el lado de la clase "compuesta-todo" y una flecha en el lado de la clase "parte de".

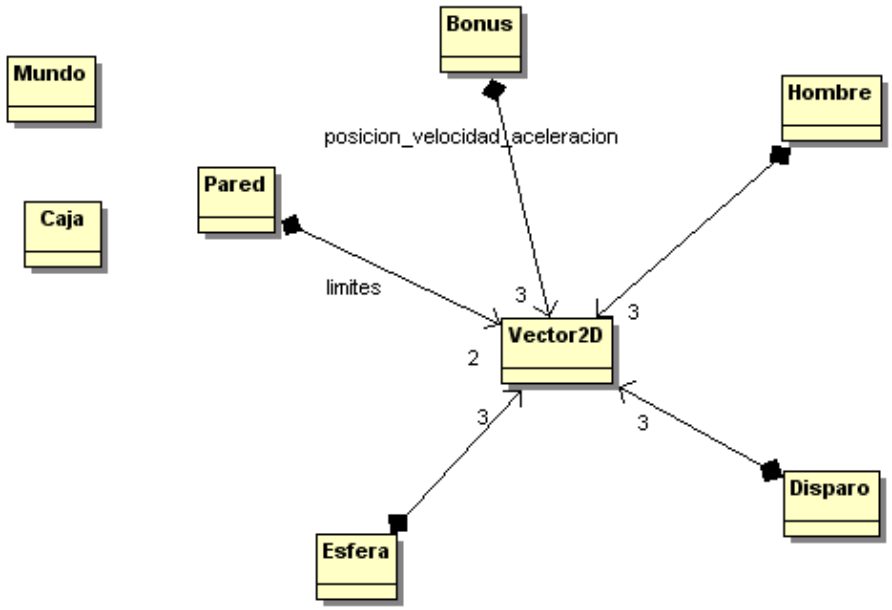


Figura 4-13. DCD con atributos de tipos compuestos como relación de composición (peor)

Elegir una forma de representarlo u otra depende de lo que quiera detallar en el diagrama el diseñador, aunque como regla general se puede decir que la primera forma

se aplica cuando la clase en cuestión representa un tipo "estándar" de dato compuesto matemático, geométrico y/o lógico, y la segunda cuando no lo es. Ej.: Clases `Complejo`, `Vector`, `Punto`, `Matriz`, `Cadena`, `CString`, etc. Por tanto, en este caso se prefiere la primera forma, mostrada en la figura 4-12.

Falta por completar el diagrama con la relación existente entre la clase `Caja` y la clase `Pared`. En este caso es muy evidente la relación de composición, ya que la caja está compuesta por cuatro paredes (multiplicidad 4) que constituyen los bordes (el rol de `Pared` en la relación) de la `Caja`.

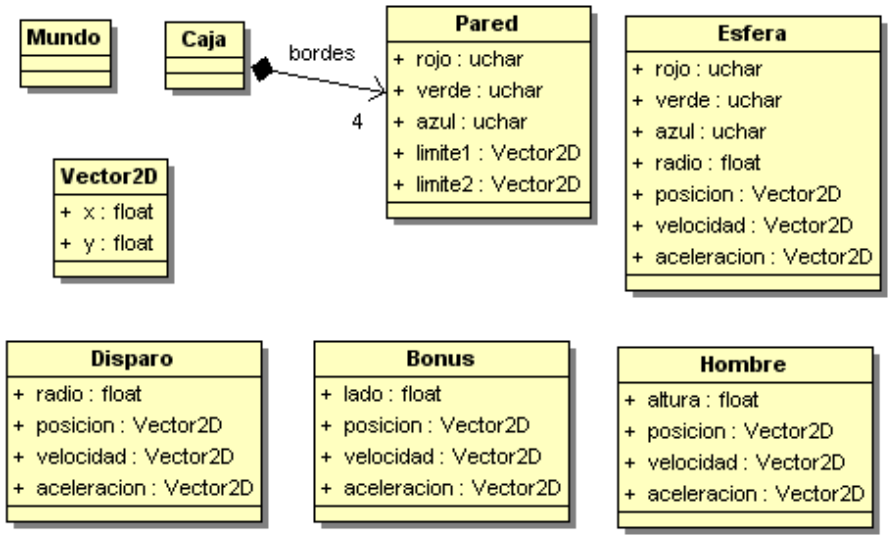


Figura 4-14. DCD mostrando la relación de composición Caja-Pared

4.5.2.2 Implementación

Implementamos en este punto el DCD anterior de la figura 4-14. El primer paso es crear una nueva clase denominada `Vector2D`, tal y como se ha visto anteriormente, que contenga las variables x e y. **Importante**: Por lo comentado anteriormente, la ubicación correcta de los ficheros de esta clase es dentro de las subcarpetas ***"comun"*** (***"Vector2D.h"*** en ***include/comun*** y ***"Vector2D.cpp"*** en ***src/comun***)

```
class Vector2D
{
public:
      Vector2D();
      virtual ~Vector2D();
      float x;
      float y;
};
```

Además, como se explicó anteriormente es una buena práctica dar valores por defecto a los atributos de una clase, en el constructor de la misma. En este caso podemos asumir el vector (0,0) como vector por defecto (posición cero = origen, velocidad cero = parado, aceleración cero = sin aceleración):

```
Vector2D::Vector2D()
{
      x=y=0;
}
```

El segundo paso es añadir los atributos de posición, velocidad y aceleración a las clases de objetos móviles. Para añadir los atributos, se puede hacer directamente en el fichero de cabecera ".h" correspondiente a la clase deseada:

```
#include "Vector2D.h"
class Hombre
{
public:
      Hombre();
      virtual ~Hombre();
      float altura;
      Vector2D posicion;
      Vector2D velocidad;
      Vector2D aceleracion;
};
```

```
#include "comun/Vector2D.h"
class Disparo
{
public:
      Disparo();
      virtual ~Disparo();
      float radio;
      Vector2D posicion;
      Vector2D velocidad;
      Vector2D aceleracion;
};
```

```
#include "comun/Vector2D.h"

class Bonus
{
public:
      Bonus();
      virtual ~Bonus();

      float lado;
      Vector2D posicion;
      Vector2D velocidad;
      Vector2D aceleracion;

};
```

```
#include "comun/Vector2D.h"

class Esfera
{
public:
      Esfera();
      virtual ~Esfera();
      unsigned char rojo;
      unsigned char verde;
      unsigned char azul;
      float radio;

      Vector2D posicion;
      Vector2D velocidad;
      Vector2D aceleracion;
};
```

Nótese que hay que realizar el `#include "comun/Vector2D.h"` correspondiente para poder utilizar la clase `Vector2D`. Visual Studio nos permite realizar este paso directamente. Si pulsamos con el botón derecho encima de la clase (en *ClassView*) y seleccionamos *New Member Variable* sale la ventana siguiente, que rellenada adecuadamente, realiza el trabajo por nosotros.

Figura 4-15. Asistente de Visual C para la adición de variables miembro a una clase.

Ejecutando este asistente dos veces para añadir el "`limite1`" y "`limite2`" de la clase `Pared`, nos queda:

```
#include "comun/Vector2D.h"

class Pared
{
```

```
public:
      Pared();
      virtual ~Pared();

      unsigned char rojo;
      unsigned char verde;
      unsigned char azul;
      Vector2D limite1;
      Vector2D limite2;
};
```

Nótese que la implementación de la **relación de composición** se realiza mediante la declaración de una variable miembro de la clase ("parte de") en la clase "compuesta por". Así, la clase `Caja` queda como sigue:

```
//IMPLEMENTACION DE LA RELACION DE COMPOSICION
#include "Pared.h"      //el include no es necesario que ponga "dominio"
                        //porque ya estamos en dicha carpeta.

class Caja
{
public:
      Caja();
      virtual ~Caja();

      Pared suelo;
      Pared techo;
      Pared pared_izq;
      Pared pared_dcha;
};
```

Al igual que hemos hecho anteriormente, damos valores por defecto a los atributos en los constructores de las clases. Por ejemplo, podemos decir que una esfera por defecto será de color blanco, tendrá un radio de 1 y una aceleración gravitatoria normal. Su posición y su velocidad, como son `Vector2D`, por defecto serán 0, es decir en el origen y con velocidad inicial nula.

```
Esfera::Esfera()
{
      rojo=verde=azul=255; //blanco
      radio=1.0f;
      aceleracion.y=-9.8f;
}
```

Nótese que para realizar esta inicialización en el constructor, es necesario declarar dicho constructor en el fichero de cabecera "***Esfera.h***", cosa que hace el Visual Studio automáticamente si utilizamos el asistente para crear la clase:

```
class Esfera
{
…
public:
      Esfera();
…
};
```

EJERCICIO

Realizar la declaración e inicialización a valores por defecto de todos los atributos existentes en el Diagrama de Clases de Diseño.

- El color por defecto de todos los objetos es blanco.

- Los disparos tienen un movimiento uniforme ascendente vertical, no afectado por la gravedad. Su radio por defecto es 0.25f.
- El hombre tampoco se ve afectado por la gravedad. Su altura es de 1.8f.
- La caja mide 15 de alto por 20 de ancho. El origen de coordenadas se sitúa en el punto medio del suelo. El color de las paredes de la caja es verde, aunque hay que dar distintos colores (tonos) para el suelo y las paredes.

4.5.3. Tercer paso: Métodos

4.5.3.1 Diseño

En el paso anterior ya tenemos la estructura de datos necesaria, pero aun no hemos añadido funcionalidad a nuestras clases. El diseño orientado a objetos tiene que identificar qué métodos es necesario añadir, y en que clases.

Las entidades `Hombre`, `Esfera`, `Pared`, `Bonus`, `Caja` y `Disparo` necesitan ser pintadas en la pantalla. Ya hemos añadido los atributos necesarios, pero ahora hay que añadir la funcionalidad para que sean representadas. Para ello añadimos un método a cada una de estas clases que se denomine `Dibuja()` y que no reciba parámetros ni devuelva nada. ¿Por qué no es necesario que el método reciba parámetros? Porque tiene las variables miembro que contienen la información necesaria. Para dibujar una esfera necesitamos solo su posición, su radio y su color, todo ello variables miembro de la clase `Esfera`.

Las clases `Hombre`, `Esfera`, `Bonus` y `Disparo` contienen información de velocidad. La posición debe ser actualizada en función de esta velocidad. Para ello añadimos un método a cada una de estas clases que se denomine `Mueve(t: float)` y que reciba un parámetro de tipo `float` que será el intervalo de tiempo transcurrido entre un instante y el siguiente, el cual haremos coincidir con el definido por el *timer* que establezcamos en la GLUT, que recordemos, establece indirectamente la frecuencia a la que se actualiza la escena.

El DCD queda como sigue:

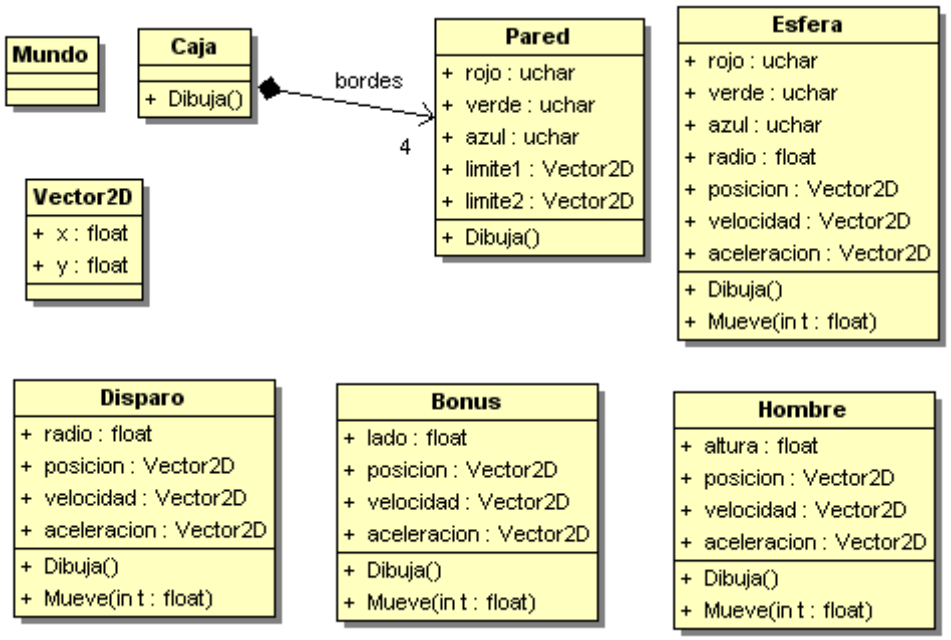


Figura 4-16. DCD con los métodos Dibuja() y Mueve()

Se observa que para dibujar la `Caja`, no va a ser necesario realizar nuevo código OpenGL, sino que basta con decirle a cada una de las cuatro paredes que la componen que se pinten. Esto se representa típicamente en un Diagrama de Secuencias como el reflejado en la figura 4-17.

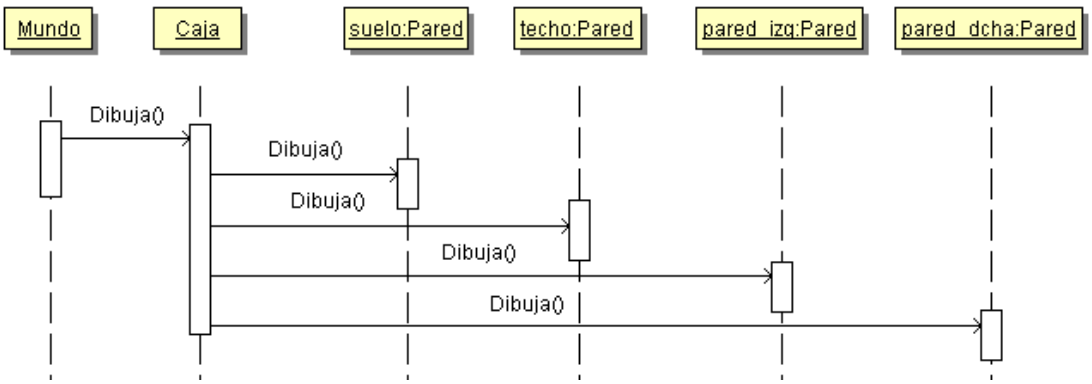


Figura 4-17. Diagrama de secuencias para dibujar la Caja

4.5.3.2 Implementación

Para implementar estos métodos, pinchamos con el botón derecho sobre la clase correspondiente y seleccionamos *Add Member Function*. Lo rellenamos adecuadamente y pulsamos OK. Esto añadirá automáticamente la declaración en el fichero ".h" y el cuerpo de la función en el ".cpp".

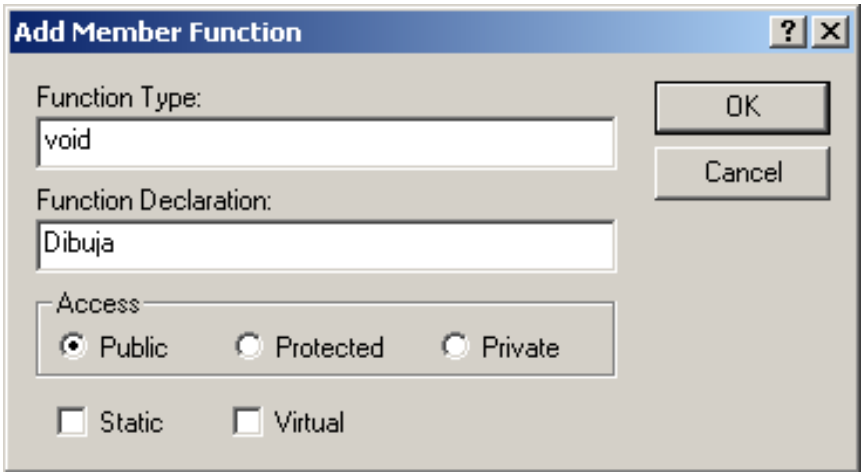


Figura 4-18. Asistente Visual Studio para la adición de un método a una clase

En el dibujo del `Bonus`, seleccionamos un color aleatorio utilizando la función `rand()` (`#include <stdlib.h>`) y el dibujo del hombre lo simplificamos de momento con una esfera.

Será necesario realizar el `#include "comun/glut.h"` correspondiente (en los ficheros ***.cpp***) para poder hacer uso de las funciones de dibujo de esta librería. La implementación resultante de estas funciones queda como sigue.

```
void Bonus::Dibuja()
{
      glPushMatrix();
      glTranslatef(posicion.x,posicion.y,0);
      glRotatef(30,1,1,1);
      glColor3f(  rand()/(float)RAND_MAX,rand()/(float)RAND_MAX,
                  rand()/(float)RAND_MAX);//color aleatorio
      glutSolidCube(lado);
      glPopMatrix();
}
void Caja::Dibuja()
{
      suelo.Dibuja();
      techo.Dibuja();
      pared_izq.Dibuja();
      pared_dcha.Dibuja();
}
void Disparo::Dibuja()
{
      glColor3f(0.0f,1.0f,1.0f);
      glPushMatrix();
      glTranslatef(posicion.x,posicion.y,0);
      glutSolidSphere(radio, 20, 20);
      glPopMatrix();
}
void Esfera::Dibuja()
{
      glColor3ub(rojo,verde,azul);
      glTranslatef(posicion.x,posicion.y,0);
      glutSolidSphere(radio,20,20);
      glTranslatef(-posicion.x,-posicion.y,0);
}
void Hombre::Dibuja()
```

```
{
	glPushMatrix();
	glTranslatef(posicion.x,posicion.y,0);
	glColor3f(1.0f,0.0f,0.0f);
	glutSolidSphere(altura, 20, 20);
	glPopMatrix();
}
void Pared::Dibuja()
{
	glDisable(GL_LIGHTING);
	glColor3ub(rojo,verde,azul);
	glBegin(GL_POLYGON);
		glVertex3d(limite1.x,limite1.y,10);
		glVertex3d(limite2.x,limite2.y,10);
		glVertex3d(limite2.x,limite2.y,-10);
		glVertex3d(limite1.x,limite1.y,-10);
	glEnd();
	glEnable(GL_LIGHTING);
}
```

Y las funciones de mover de todos los objetos quedan exactamente iguales. Se incluye aquí la función de la clase `Bonus`, pero debe de ser implementada igual para el resto de clases.

```
void Bonus::Mueve(float t)
{
	posicion.x=posicion.x+velocidad.x*t+0.5f*aceleracion.x*t*t;
	posicion.y=posicion.y+velocidad.y*t+0.5f*aceleracion.y*t*t;
	velocidad.x=velocidad.x+aceleracion.x*t;
	velocidad.y=velocidad.y+aceleracion.y*t;
}
```

4.5.3.3 Patrón Experto en Información

¿Por qué hemos decidido poner la función `Mueve()` y la función `Dibuja()` en las clases respectivas? ¿No podríamos haber puesto unas funciones globales, o haber puesto las funciones como miembro de la clase mundo, y pasarle como parámetro el objeto que queremos mover o dibujar?

Si implementamos la funcionalidad de dibujo en la clase `Mundo`, podríamos intentar hacer algo como:

```
void Mundo::Dibuja(Esfera e)
{
	glColor3ub(e.rojo, e.verde, e.azul);
	glTranslatef(e.posicion.x, e.posicion.y,0);
	glutSolidSphere(e.radio,20,20);
	glTranslatef(-e.posicion.x,- e.posicion.y,0);
}
```

No obstante las variables a las que queremos acceder podrían ser privadas. En cualquier caso, aunque fueran públicas, el código queda mucho más complejo, comparado a lo expuesto anteriormente. Toda la información necesaria para pintar una esfera esta dentro de la clase `Esfera`. Se dice que el **Experto en Información** es la clase `Esfera`, y por tanto ella debe de ser la responsable de pintarse a si misma puesto que tiene toda la información requerida para realizar esta operación. Por tanto el método `Dibuja()` debe de pertenecer a la clase `Esfera`, no a ninguna otra.

4.5.4. Cuarto paso: Objetos

En este punto ejecutamos la aplicación. ¿Qué tenemos? Aun nada. Con todo el código que hemos realizado ¿Por qué no hay nada? Porque no existen objetos de las clases anteriores. Hemos declarado e implementado las clases pero no hemos creado instancias de las mismas. Los objetos de nuestra aplicación se pueden representar en un Diagrama de Colaboraciones. En este instante en el diagrama de colaboraciones de nuestra aplicación solo existiría un objeto "mundo" de la clase "`Mundo`", porque es lo único que hemos instanciado como variable global:

Si queremos utilizar el código realizado en los pasos anteriores tenemos que instanciar objetos de dichas clases. Esta instancia la podemos realizar en la clase `Mundo`, de tal forma que la clase `Mundo` contenga una instancia de cada uno de los objetos, para poder probarlos.

4.5.4.1 Diseño

Nuestro diseño tiene que contemplar fundamentalmente 3 cosas: se deben instanciar objetos de las clases desarrolladas, se tiene que decir a esos objetos que se pinten y se tiene que decir a algunos de dichos objetos que se muevan. Es decir, la funcionalidad que hemos implementando en las clases tiene que ser invocada desde algún sitio, y en algún orden.

Los diagramas de clases vistos anteriormente sirven para las tareas comunes de diseño, identificación de atributos y métodos, relaciones genéricas entre clases, pero no sirven para especificar los objetos concretos de esas clases, ni las interacciones entre ellos. En este apartado veremos los dos tipos existentes de diagramas de interacciones entre objetos: diagramas de secuencia y diagramas de colaboraciones. Estos dos tipos de diagramas son básicamente complementarios, ambos presentan la misma información desde dos puntos de vista diferentes.

Los diagramas de secuencias se centran en la evolución temporal de los mensajes o interacciones entre objetos, mientras que en los diagramas de colaboraciones se muestra más claramente los vínculos entre objetos en vez del orden temporal de los mensajes.

Vamos a analizar primero la interacción de nuestro código con la GLUT y con el usuario del programa, cuyas acciones van canalizadas a través de la GLUT. Tal como hemos configurado la GLUT, un *timer* que actúa cada 25 milisegundos invoca mediante un *callback* a la función `OnTimer()` existente en el fichero principal. Aunque este *callback* es parte de nuestro código, realmente lo único que hace es enviar un mensaje a la clase `Mundo` para decirle que ha transcurrido el intervalo de tiempo fijado por el temporizador. Este mensaje es el mensaje `Mundo::Mueve()`. Posteriormente, la propia GLUT envía un mensaje a la clase `Mundo` para indicarle que tiene que pintar. Es el mensaje `Mundo::Dibuja()`. Cuando el usuario pulsa una tecla, la GLUT invoca el método `Tecla()` de la clase `Mundo`. Es el mensaje `Mundo::Tecla()`. El siguiente diagrama de secuencias ilustra este comportamiento.

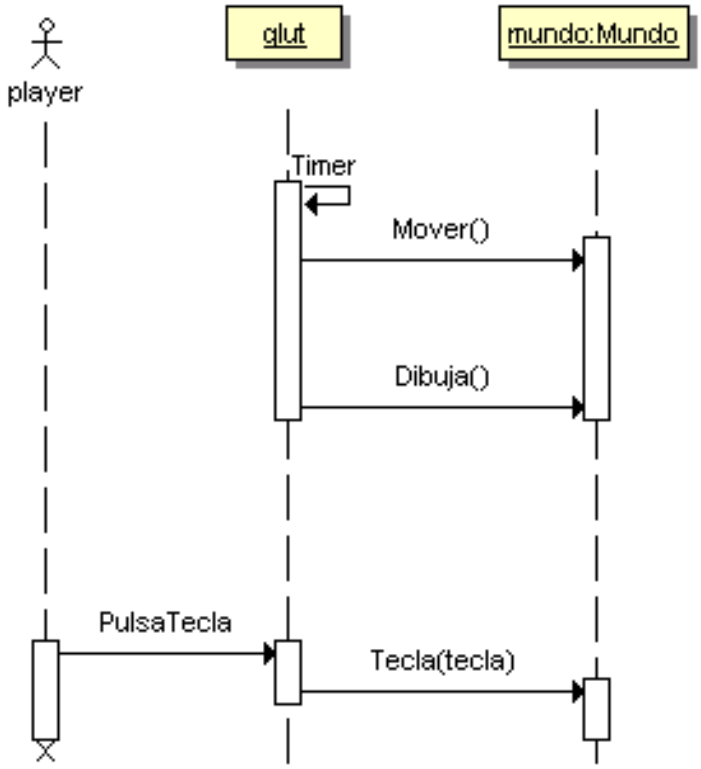


Figura 4-19. Diagrama de secuencias del funcionamiento de la GLUT

Hacemos ahora que el mundo contenga un objeto de cada clase, para poder probar la funcionalidad de dichas clases. Como esto es una situación de prueba provisional, ni siquiera la representamos en un diagrama de clases. Cuando la clase `Mundo` instancia un objeto de cada clase, se establece entre el objeto mundo y dichas clases un **vínculo** que permite al objeto "`mundo`" enviar mensajes a dichos objetos. Tal y como se realiza esta prueba, el vinculo es unidireccional, la clase `Mundo` puede invocar los métodos de las clases contenidas, pero no al revés.

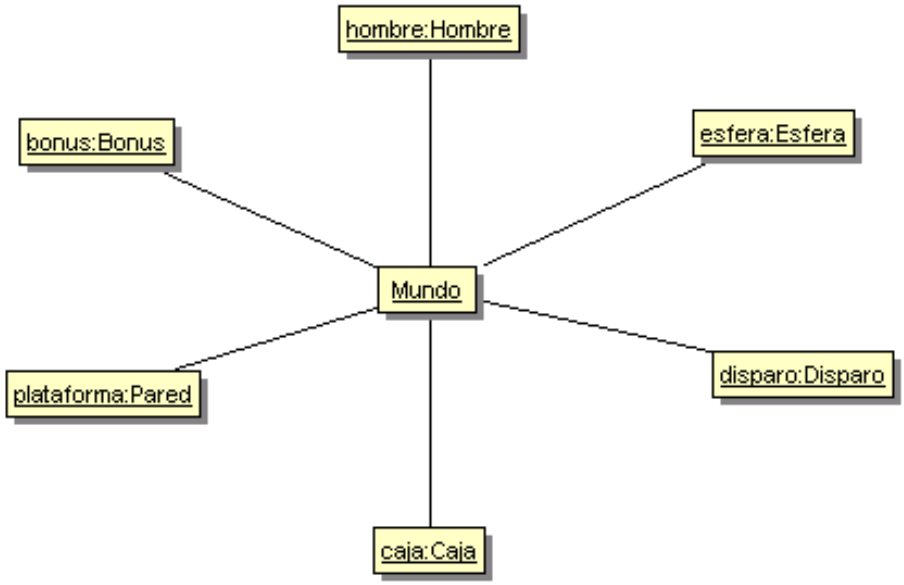


Figura 4-20. Vínculos entre los objetos

Obviamente, cuando a la clase `Mundo` le llega el mensaje de `Mueve()`, ella a su vez tiene que comunicar a los objetos (solo a los móviles) que se tienen que mover. Esto se puede representar mediante el siguiente diagrama de secuencias:

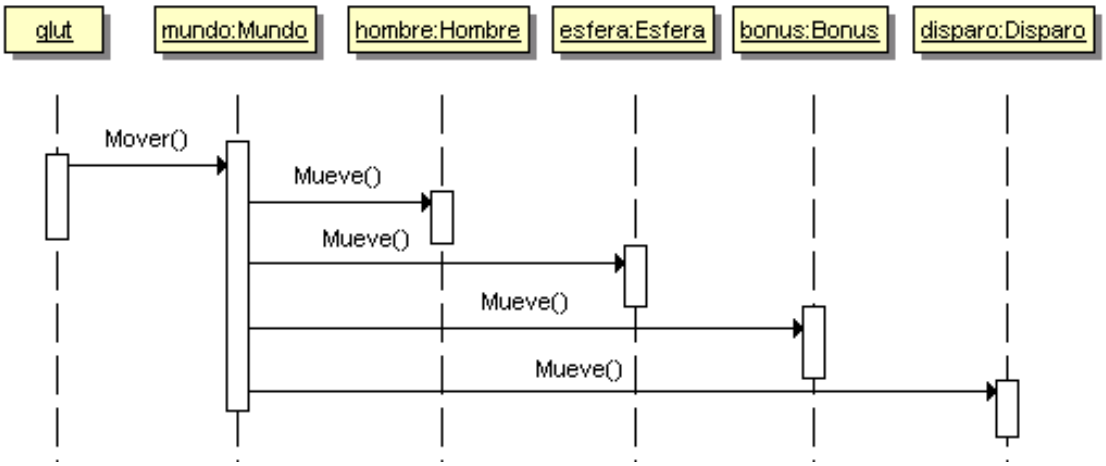


Figura 4-21. Diagrama de secuencias de animación de los objetos del mundo

Cuando no interesa mucho el orden temporal de los mensajes, o quizás no es cómodo trabajar en un diagrama de secuencias porque se expande mucho a la derecha, se puede utilizar un diagrama de colaboraciones. Por ejemplo, para expresar como se distribuye el pintado de los objetos podemos realizar el siguiente diagrama de colaboraciones, en el que el orden temporal de los mensajes viene mostrado por números (Figura 4-22):

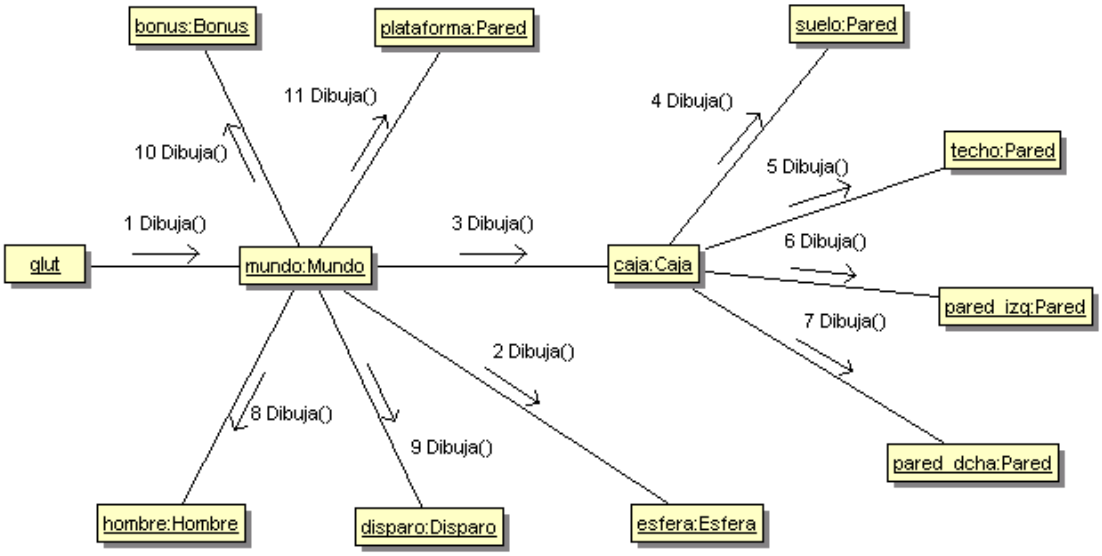


Figura 4-22. Diagrama de colaboraciones del dibujado de la escena

4.5.4.2 Implementación

Realizamos la implementación, instanciando un objeto de cada tipo en la clase `Mundo`:

```
#include "dominio/Caja.h"
#include "dominio/Hombre.h"
#include "dominio/Esfera.h"
#include "dominio/Bonus.h"
#include "dominio/Disparo.h"
class CMundo
{
public:
      Disparo disparo;
      Esfera esfera;
      Hombre hombre;
      Caja caja;
      Bonus bonus;
      Pared plataforma;
…
};
```

Por el hecho de haber implementado de esta forma esta relación, en el momento en el que se instancia el objeto "`mundo`" de la clase "`Mundo`", automáticamente se crean los objetos "`disparo`", "`esfera`", "`hombre`", "`caja`", "`bonus`" y "`plataforma`" de las clases respectivas. Además, existe un vinculo creado automáticamente entre estos objetos y el objeto "`mundo`", de tal forma que el objeto "`mundo`" puede enviar mensajes (invocar funciones) a los objetos que contiene. De la misma forma, el objeto "`caja`" instancia automáticamente sus objetos "`suelo`", "`techo`", "`pared_izq`", "`pared_dcha`", con los que automáticamente tiene un vinculo. Nótese bien que este vínculo solo se crea automáticamente en este tipo de asociación de composición, pero no en todas las asociaciones.

Compilamos y ejecutamos el programa. ¿Por qué sigue sin verse nada en la pantalla? El hecho de tener los objetos no implica que se pinten solos ni se muevan solos, tenemos

que interaccionar con los objetos para decirles cuando se tienen que pintar y cuando se tienen que mover.

Lo que hay que hacer es que el objeto "mundo" interaccione con los otros objetos a través de sus vínculos, enviándoles mensajes de cuando se tienen que mover y cuando se tienen que pintar.

```
void Mundo::Dibuja()
{
      gluLookAt(x_ojo, y_ojo, z_ojo,
                0.0, y_ojo, 0.0,     //NOTESE QUE HEMOS CAMBIADO ESTO
                0.0, 1.0, 0.0);    //PARA MIRAR AL CENTRO DE LA ESCENA
      esfera.Dibuja();
      caja.Dibuja();
      hombre.Dibuja();
      disparo.Dibuja();
      plataforma.Dibuja();
      bonus.Dibuja();
}

void Mundo::Mover()
{
      hombre.Mueve(0.025f);
      esfera.Mueve(0.025f);
      bonus.Mueve(0.025f);
      disparo.Mueve(0.025f);
}
```

Para ver las objetos en distintas posiciones iniciales, diferentes a las de por defecto, establecer los valores deseados en la función `Inicializa()` del `Mundo`:

```
void Mundo::Inicializa()
{
      x_ojo=0;
      y_ojo=7.5;
      z_ojo=30;

      esfera.posicion.x=2;
      esfera.posicion.y=4;
      esfera.radio=1.5f;
      esfera.rojo=0;
      esfera.verde=0;
      esfera.azul=255;

      bonus.posicion.x=5.0f;
      bonus.posicion.y=5.0f;

      disparo.posicion.x=-5.0f;
      disparo.posicion.y=0.0f;

      plataforma.limite1.x=-5.0f;
      plataforma.limite2.x=5.0f;
      plataforma.limite1.y=9.0f;
      plataforma.limite2.y=9.0f;
}
```

4.6. EJERCICIOS PROPUESTOS

1. Modificar el diseño y la implementación del programa realizado, añadiendo atributos y modificando métodos, para que el disparo se represente con su estela. Nota: Añadir como atributo el punto de origen del disparo, denominándolo "origen"
2. Volver a convertir en privados los atributos de la clase `Esfera`. Implementar los métodos `SetColor()`, `SetRadio()`, `SetPos()` tal y como se vio en el tema anterior, para poder establecer los valores, de tal forma que la inicialización de las misma quede:

```
esfera.SetColor(0,0,255);
esfera.SetRadio(1.5f);
esfera.SetPos(2,4);
```

3. Convertir en privados los atributos de color de la clase `Pared`. Implementar el método `SetColor()` que permita definir el color de una pared, de tal forma que el constructor de la caja quede como sigue:

```
Caja::Caja()
{
      suelo.SetColor(0,100,0);
      techo.SetColor(0,100,0);
      pared_dcha.SetColor(0,150,0);
      pared_izq.SetColor(0,150,0);
}
```

5. INTERACCIONES

5.1. INTRODUCCIÓN

En el capítulo anterior hemos finalizado con un programa que al ejecutar muestra varios objetos moviéndose libremente. Sin embargo, para realizar nuestro juego, necesitamos que los objetos interactúen entre ellos. También necesitamos que el usuario del programa o jugador sea capaz de controlar por teclado el hombre de la pantalla.

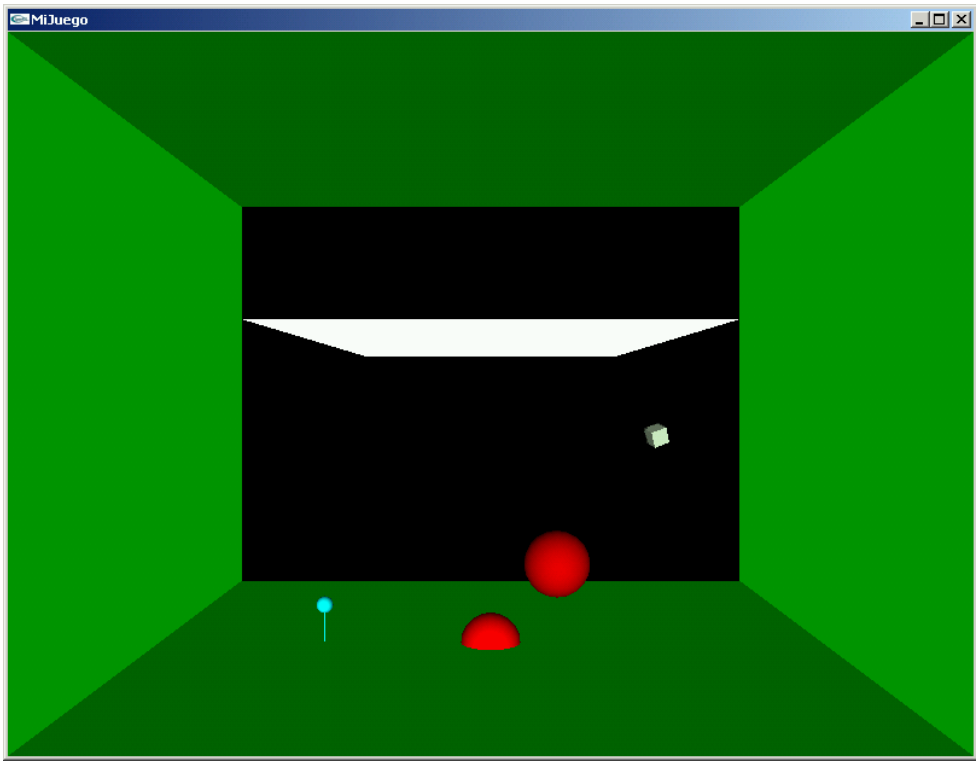


Figura 5-1. Los objetos se mueven libremente

A lo largo de este capítulo se van a abordar dos aspectos fundamentalmente. En primer lugar, se realizarán una serie de modificaciones aparentemente pequeñas al código que ya tenemos para lograr un programa más compacto, entendible y seguro.

Fundamentalmente se realizará por medio de dos metodologías. En primer lugar se hará uso de métodos de interfaz, que además nos llevará a revisar e implementar la característica de encapsulamiento que deben tener las clases de un buen programa.

En segundo lugar, se procederá a modificar la clase `Vector2D` de forma que no sea un simple contenedor para las componentes de vectores de dos dimensiones, sino que además nos de una serie de operaciones básicas como son la suma, el producto escalar, o el producto por un escalar. Lo más interesante de esto, es que se realizará mediante la redefinición de los operadores a los cuales ya estamos acostumbrados (*,+, etc.). A esto se lo denomina como **sobrecarga de operadores**.

Una vez compactado y aclarado el código, se procederá a agregar nuevas funcionalidades a las clases y al programa. Se comenzará a introducir cierta interacción tanto entre el jugador y el juego como entre los distintos objetos de la escena. Para ello en primer lugar se implementará el movimiento del jugador en función de las pulsaciones del teclado. A continuación, se establecerá una elegante estructura que nos permitirá programar con facilidad las interacciones que se dan entre dos objetos, como por ejemplo que una esfera debe rebotar contra una pared, o que dos esferas rebotan al chocar.

Finalmente, y como medida de la bondad de la metodología de programación seguida, se procederá a definir múltiples objetos a la escena con muy poco esfuerzo.

5.2. HACIENDO CLASES MÁS SEGURAS: EL ENCAPSULAMIENTO

Con la introducción del concepto de clase se ha hablado de un aspecto especialmente importante en la POO, que es la **encapsulación**. Recordemos que la idea que subyace en esta metodología de programación es la de lograr objetos reutilizables y seguros por sí mismos incluso ante errores del programador.

Principalmente esto se logra poniendo limitaciones a las posibles modificaciones de los datos contenidos en el objeto (**atributos**) de forma que se asegure que la información del objeto es válida y coherente.

En el capítulo tercero se observaba como es interesante asegurar que el radio de una esfera sea un valor positivo, dado que no tiene sentido hablar de un radio negativo. Si un atributo es accesible públicamente, de forma que le podemos asignar cualquier valor, no habría forma de controlar a priori que se cumple esta condición obvia sobre los posibles valores del atributo. Por tanto se procede a esconder este dato, y que su modificación se realice por medio de los métodos de la clase, que a su vez se asegurarán que el radio asignado es válido y en caso de que no lo sea, adoptarán la medida que al programador le parezca más oportuna.

Como norma de buena programación, cuando un objeto no es básico y tiene cierta complejidad, la tendencia es la de hacer todos sus atributos privados o protegidos. De esta forma la modificación y consulta de los datos es controlada por medio de métodos públicos de la clase.

En el capítulo anterior ya se había procedido a establecer como privados todos los atributos de la clase `Esfera`. A continuación procedemos a realizar lo mismo para el resto de clases definidas, salvo para el `Vector2D` que consideraremos como básico y por tanto difícilmente incoherente en cuanto a la información contenida.

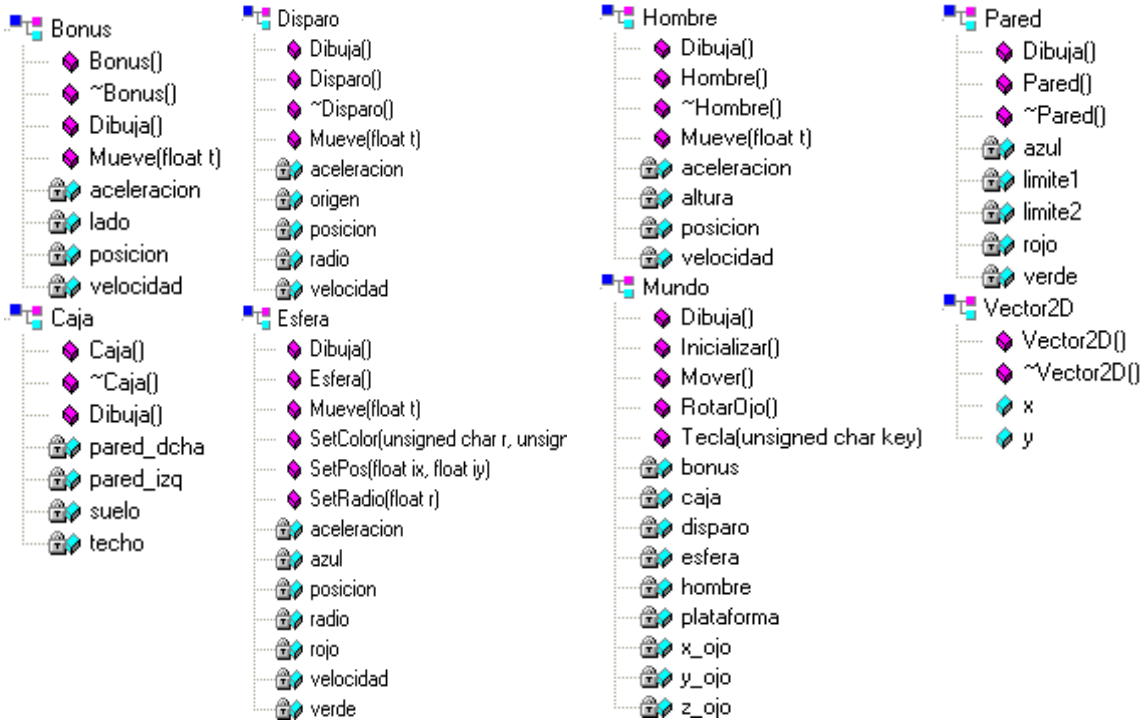


Figura 5-2. Clases, atributos, métodos y niveles de acceso tras hacer privados los atributos.

Realizar esta operación es tan sencillo como recorrer todas las declaraciones de las clases y anteponer `private:` antes del conjunto de atributos. Por cuestiones de estilo, se suelen poner los atributos y los métodos en dos grupos claramente diferenciados. Se recomienda utilizar esta estructura. Si intentamos compilar ahora el código resultante, se observa que se producen errores de compilación:

```
mundo.cpp(44) : error C2248: 'posicion' : cannot access private member declared in class 'Bonus'
mundo.cpp(47) : error C2248: 'posicion' : cannot access private member declared in class 'Disparo'
mundo.cpp(49) : error C2248: 'origen' : cannot access private member declared in class 'Disparo'
...
```

Como era de esperar, los intentos de modificación de los atributos desde código externo a la clase, generan un error: *no es posible acceder al miembro privado declarado en la clase Bonus.*

Se va a aprovechar esta circunstancia para crear un método en cada clase cuya finalidad es la de inicializar los atributos según unos parámetros. Comenzamos introduciendo el método `SetPos(float x1,float y1,float x2, float y2)` para la clase `Pared`, de forma que se asigne al `limite1` las coordenadas `x1` e `y1`, y al `limite2` las coordenadas `x2` e `y2`. Utilizamos para ello la opción del menú contextual *Add member function…* pinchando sobre la clase `Pared`.

Como consecuencia se habrá agregado el siguiente código en el fichero ***Pared.cpp***:

```
void Pared::SetPos(float x1, float y1, float x2, float y2)
```

```
{
	limite1.x=x1;
	limite1.y=y1;
	limite2.x=x2;
	limite2.y=y2;
}
```

Dado que la caja contiene cuatro paredes que además son inicializadas al construirla, el compilador ha generado muchos errores como consecuencia. Con el uso de estos dos nuevos métodos, modificamos el constructor de la clase `Caja`, quedando mucho más compacto y sencillo:

```
Caja::Caja()
{
	suelo.SetColor(0,100,0);
	suelo.SetPos(-10.0f,0,10.0f,0);

	techo.SetColor(0,100,0);
	techo.SetPos(-10.0f,15.0f,10.0f,15.0f);

	pared_dcha.SetColor(0,150,0);
	pared_dcha.SetPos(-10.0f,0,-10.0f,15.0f);

	pared_izq.SetColor(0,150,0);
	pared_izq.SetPos(10.0f,0,10.0f,15.0f);
}
```

EJERCICIO

De igual forma procedemos a definir e implementar el método de la clase `Bonus` `void SetPos(float ix, float iy)` que se encarga de posicionar el `Bonus` y `Disparo`, teniendo en cuenta en este último caso que además de posicionar el disparo inicializa el atributo origen con el mismo valor. De esta forma, el método `Inicializa()` de la clase `Mundo` quedará, una vez hecho uso de estos métodos, como sigue:

```
void Mundo::Inicializa()
{
	x_ojo=0;
	y_ojo=7.5;
	z_ojo=30;
	esfera.SetColor(0,0,255);
	esfera.SetRadio(1.5f);
	esfera.SetPos(2,4);
	bonus.SetPos(5.0f,5.0f);
	disparo.SetPos(-5.0f,0.0f);
	plataforma.SetPos(-5.0f,9.0f,5.0f,9.0f);
}
```

Que como se observa, es de nuevo mucho más claro y compacto que el código anterior. Tras este paso, ya se puede volver a compilar, y se ve que aparentemente no se ha cambiado nada, pero ahora el código es más compacto y las clases más cerradas y seguras.

5.3. CLASE VECTOR2D: LA SOBRECARGA DE OPERADORES

En C++ es posible escribir varias funciones que teniendo el mismo identificador son diferenciadas por el tipo de los argumentos que se utilizan. A este mecanismo se le denomina **sobrecarga**. De esta forma se facilita la comprensión del código y la facilidad de programación al evitar tener que utilizar nombres distintos para tipos de argumentos distintos cuando la funcionalidad es parecida. El compilador decide a cual de las funciones que tienen el mismo nombre nos referimos precisamente porque las distingue por el tipo de datos que se introducen como parámetros.

Dentro de una clase ocurre lo mismo. Es posible tener varios métodos que se llamen igual y que se diferencien por el tipo de argumentos que se esperan. Un caso especial de sobrecarga es el que C++ nos permite realizar con los operadores normales. Así es posible definir una suma o un producto especial para la clase `Vector2D`. De hecho, es posible sobrecargar la operación producto (*) con más de un método. Se podrá definir un producto de un `Vector2D` con un `float`, distinto que el producto de dos `Vector2D` entre sí, aun siendo usado en ambos casos el mismo símbolo (*).

En C++ los operadores pasan a ser unas funciones como otra cualquiera, pero que permiten para su ejecución una notación especial. Todos ellos tienen asignada una función por defecto que es la que hasta ahora habíamos utilizado en C. Por ejemplo, el operador '+' realizará la suma aritmética o de punteros de los operandos que pongamos a ambos lados.

A partir de ahora, podremos hacer que el operador '+' realice acciones distintas en función de la naturaleza de los operandos sobre los que trabaja. De hecho, en realidad, la mayoría de los operandos en C++ están sobrecargados. El ordenador realiza operaciones distintas cuando divide enteros que cuando divide números en coma flotante. Un ejemplo más claro sería el del operador *, que en unos casos trabaja como multiplicador y en otros realiza la operación de indirección.

En C++ el programador puede sobrecargar casi todos los operadores para adaptarlos a sus propios usos. Para ello disponemos de una sintaxis específica que nos permite definirlos o declararlos al igual que ocurre con las funciones:

Prototipo para los operadores:

```
<tipo> operator <operador> (<argumentos>);
```

Definición para los operadores:

```
<tipo> operator <operador> (<argumentos>)
{
 <sentencias>;
}
```

Antes de ver lo interesante de su aplicación como operadores miembros de una clase, es necesario mostrar algunas limitaciones en la sobrecarga de operadores, así como su uso como funciones externas a una clase.

Existen unas limitaciones a la sobrecarga de operadores:

- Se pueden sobrecargar todos los operadores excepto ".", ",", ".*" , "::" ,"?:".
- Los operadores "=","[ ]","->","`new`" y "`delete`", sólo pueden ser sobrecargados cuando se definen como miembros de una clase.

- Los argumentos para los operadores externos deben ser tipos enumerados o estructurados: `struct, union` o `class`.

Cuando se redefine un operador **dentro de una clase**, la sintaxis utilizada es muy parecida, pero se entiende que **el primer operando es el objeto que llama al método**, y el segundo operando –en caso de haberlo- es el que se pasa como argumento. Así, por ejemplo, la operación + para `Vector2D` se podría declarar y definir como sigue:

```
//Prototipo para la suma dentro de Vector2D
Vector2D operator+(Vector2D op);

//Definición para los operadores:
Vector2D Vector2D::operator+(Vector2D op)
{
      Vector2D resultado;
      resultado.x=x+op.x;
      resultado.y=y+op.y;
      return resultado;
}
```

Con esta definición, el siguiente código ejecutaría el operador escrito, siendo `v1` y `v2` objetos de la clase `Vector2D`. Es importante destacar que lo que está ocurriendo es que `v1` llama a su método operador + al cual se le pasa como argumento `v2`, y que el resultado de dicha operación es un `Vector2D` dado por el valor de retorno de este particular método.

```
      Vector2D suma;
      suma=v1+v2;
```

Una forma de verlo más claramente es realizar una llamada explícita al método `operator+`. El siguiente código, aunque menos estético, es exactamente equivalente al anterior:

```
      Vector2D suma;
      suma=v1.operator+(v2);
```

Al igual que las funciones los operadores son sobrecargables. Se recuerda que en C++ las llamadas a funciones con el mismo nombre pueden diferenciarse gracias al número y al tipo de los parámetros de la función. Esto permite que definamos una operación diferente para el caso en que se sume un vector a otro vector, o que nos planteemos lo que significa sumarle a un vector un valor real. Por ejemplo, una posible sobrecarga del operador suma sería la siguiente:

```
//Prototipo para la sobrecarga de suma dentro de Vector2D
Vector2D operator+(float real);

//Definición de este nuevo operador:
Vector2D Vector2D::operator+(float real)
 {
      Vector2D resultado;
      resultado.x=x+real;
      resultado.y=y;
      return resultado;
```

```
}
```

Teniendo simultáneamente definidos estos operadores, el siguiente código ejecuta uno u otro basándose en qué es lo que se suma al vector.

```
Vector2D suma1,suma2;
suma1=v1+v2;
suma2=v1+5.0f;
```

5.3.1. Operadores y métodos de la clase Vector2D

A continuación procedemos a sobrecargar los operadores de la clase `Vector2D`. Reescribimos la definición de esta clase, en el fichero ***Vector2D.h***:

```
class Vector2D
{
public: //atributos
      float x;
      float y;
public: //métodos
      Vector2D(float xv=0.0f,float yv=0.0f); // (1)
      virtual ~Vector2D();

      float modulo();                  // (2) modulo del vector
      float argumento();               // (3) argumento del vector
      Vector2D Unitario();             // (4) devuelve un vector unitario
      Vector2D operator - (Vector2D &);  // (5) resta de vectores
      Vector2D operator + (Vector2D &);  // (6) suma de vectores
      float operator *(Vector2D &);      // (7) producto escalar
      Vector2D operator *(float);        // (8) producto por un escalar
};
```

En primer lugar se observa que se ha modificado el constructor. Anteriormente el constructor no tenía parámetros, y se ocupaba de poner a cero las componentes del vector cada vez que se crea un objeto de este tipo. Con la línea (1) indicamos que ahora el constructor tiene dos argumentos, que nos servirán para inicializar las componentes del vector. Sin embargo, se dan unos valores por defecto, de forma que si se omite el segundo o los dos argumentos, los valores omitidos se considerarán nulos.

Como consecuencia, la implementación de este constructor en ***Vector2D.cpp***, sustituye al anterior, y deberá escribirse como sigue:

```
Vector2D::Vector2D(float xv,float yv)
{
      x=xv;
      y=yv;
}
```

El siguiente método no implementado es el indicado en la línea (2). La finalidad del mismo es obtener el módulo del vector. Por tanto el valor m que retornará la ejecución de este método quedará determinado por la siguiente expresión matemática:

$$m = \sqrt{x^2 + y^2}$$

Siendo x e y las componentes del vector. El código que a continuación se incluye implementa esta operación:

```
float Vector2D::modulo()
{
      return (float)sqrt(x*x+y*y);
}
```

Hay que destacar la inclusión de una conversión forzosa de tipo de datos para evitar la aparición de una advertencia en la compilación. Esto es porque la función raíz cuadrada tiene como valor de retorno un `double`, mientras que este método retorna un `float`.

No hay que olvidar que para poder hacer uso de la función raíz cuadrada (`sqrt`) es necesario incluir la librería ***math.h***.

De igual forma procedemos a implementar el cálculo del argumento. Esta operación matemática sobre vectores de dos dimensiones viene determinada en el primer cuadrante por la siguiente expresión:

$$\text{argumento} = \text{atan}\left(\frac{y}{x}\right)$$

La información que nos permite obtener el argumento para los cuatro cuadrantes viene determinada por los signos de x e y. De forma que si x es positivo e y es negativo, nos encontramos en el cuarto cuadrante, y el argumento es el siguiente:

$$\text{argumento}=\text{-atan}\left(\frac{|y|}{|x|}\right)$$

Con razonamientos similares se puede obtener la información del ángulo para las distintas combinaciones de signo entre las componentes x e y. Por suerte, en la librería matemática existe la función cuyo prototipo es `double atan2(double num, double den)` que realiza todas estas operaciones directamente. Además, si la componente x es nula, en cuyo caso el cociente sería infinito, esta función nos indica correctamente el ángulo del vector ($\frac{\pi}{2}$ ó $-\frac{\pi}{2}$).

Por tanto la implementación del método argumento para la clase `Vector2D` es como sigue:

```
float Vector2D::argumento()
{
      return (float)atan2(y,x);
}
```

El siguiente método obtiene el vector director del objeto que lo ha llamado, y por tanto realiza la siguiente operación matemática:

$$\vec{u} = \frac{\vec{v}}{|\vec{v}|}$$

La cual implementamos con el siguiente código:

```
Vector2D Vector2D::Unitario()
{
      Vector2D retorno(x,y);
      float mod=modulo();
      if(mod>0.00001)
      {
            retorno.x/=mod;
```

```
            retorno.y/=mod;
      }
      return retorno;
}
```

Nótese las precauciones tomadas a la hora de realizar el cociente. Como cabe la posibilidad de que el módulo del vector sea nulo, lo cual nos llevaría a una indeterminación matemática, se comprueba que este supera un valor que se considera despreciable (0.00001) antes de proceder a realizar la división.

Comenzamos a implementar a continuación la operación suma y la operación resta, ambas muy parecidas. En el prototipo de estas operaciones (5) y (6) , se observa que se ha incluido un paso de parámetro por referencia, en vez de por valor. La razón de proceder de esta forma, a pesar de que la variable con la que se opera no es modificada, es de eficiencia. El paso por referencia es más rápido que el paso por valor. Esta eficiencia es mayor en la medida en que más grande es la información almacenada por el objeto. En este caso no es mucha, por lo que su inclusión es casi por motivos exclusivamente docentes, dado que este modo de implementar los operadores es muy habitual.

Siguiendo la sintaxis anteriormente expuesta, el código que realiza estas operaciones es el siguiente:

```
Vector2D Vector2D::operator - (Vector2D &v)
{
      Vector2D res;
      res.x=x-v.x;
      res.y=y-v.y;
      return res;
}


Vector2D Vector2D::operator + (Vector2D &v)
{
      Vector2D res;
      res.x=x+v.x;
      res.y=y+v.y;
      return res;
}
```

EJERCICIO: Finalmente, se deja como ejercicio completar las operaciones de producto escalar de vectores y del producto por un escalar. Se observa que es una operación sobrecargada, dado que se ejecutará un código diferente si el operando que va después del * es un vector o es una variable real.

5.3.2. Utilizando los nuevos operadores de la clase Vector2D.

La redefinición de los operadores permite una escritura mucho más compacta y clara del código. Se recuerda que las ecuaciones del movimiento uniformemente acelerado, descrito en base a las componentes de la posición, la velocidad y la aceleración, se describieron con las ecuaciones (4.1) y se implementaron en los distintos objetos móviles como sigue:

```
void Esfera::Mueve(float t)
{
```

```
        posicion.x=posicion.x+velocidad.x*t+0.5f*aceleracion.x*t*t;
        posicion.y=posicion.y+velocidad.y*t+0.5f*aceleracion.y*t*t;

        velocidad.x=velocidad.x+aceleracion.x*t;
        velocidad.y=velocidad.y+aceleracion.y*t;
}
```

Sin embargo, utilizando las operaciones con vectores, tal y como se expone en las ecuaciones del capítulo 4 la notación es mucho más compacta y clara:

```
void Esfera::Mueve(float t)
{
        posicion=posicion+velocidad*t+aceleracion*(0.5f*t*t);
        velocidad=velocidad+aceleracion*t;
}
```

Dado que las operaciones binarias son ejecutadas por el objeto que se encuentra a la izquierda del operador, nótese el cuidado que se ha tenido para que los vectores se encuentren siempre en esa posición relativa al operador. Si en vez de `velocidad*t` se hubiera escrito `t*velocidad`, el compilador generaría un error, al no tener los objetos de tipo `float` definida la operación `*` con un `Vector2D`.

De igual forma procedemos a modificar las ecuaciones de movimiento de las clases `Hombre`, `Disparo` y `Bonus`.

En este punto, compilamos el programa para verificar que no hay ningún error. De nuevo no existe una modificación aparente, dado que lo que se ha venido realizando es un ordenamiento del código y dotando de herramientas para el consiguiente desarrollo del programa.

5.4. INTERACCIONANDO CON EL USUARIO: MOVIENDO EL JUGADOR

Continuamos ahora dotando de nuevas funcionalidades al código. En primer lugar se va a proceder a implementar que el jugador responda al movimiento de las teclas, de forma que cuando se pulse la flecha de la izquierda, vaya hacia la izquierda, y cuando se pulse la derecha vaya a la derecha.

Vamos a generar un método sencillo perteneciente a la clase `Hombre`, que permita definir su velocidad, de forma similar a como hemos hecho con la posición de otras clases:

```
void Hombre::SetVel(float vx, float vy)
{
        velocidad.x=vx;
        velocidad.y=vy;
}
```

Esta función deberá ser llamada por la clase `Mundo` cada vez que se pulsen las teclas de los cursores a derecha e izquierda. En un principio no parece tener especial complejidad salvo por el hecho de que la librería GLUT trata de forma distinta las teclas de función y de los cursores que el resto de teclas. Para ellas se define una función de *callback* especial. Por ello es necesario realizar las siguientes modificaciones:

1. Agregamos una función a la clase `Mundo` cuya misión es la de realizar las acciones oportunas cuando se pulse una tecla especial. En este caso, lo único que hay que hacer es incrementar la posición del objeto hombre.

```
void Mundo::TeclaEspecial(unsigned char key)
{
	switch(key)
	{
	case GLUT_KEY_LEFT:
		hombre.SetVel (-5.0f, 0.0f);
		break;
	case GLUT_KEY_RIGHT:
		hombre.SetVel (5.0f, 0.0f);
		break;
	}
}
```

2. Crear una función de *callback* según el formato especificado por la librería GLUT que servirá para llamar al método correspondiente de `Mundo`. Esta modificación la se realiza en el fichero ***Pang.cpp***, y se sigue la estructura de declaración y definición utilizada para el resto de funciones *callback* que se incluyeron en los primeros capítulos:

```
void OnSpecialKeyboardDown(int key, int x, int y)
{
	mundo.TeclaEspecial(key);
}
```

3. Indicamos a la GLUT que ésta es la función que se debe ejecutar cada vez que se pulse alguna de las teclas especiales, lo cual realizamos en el cuerpo del `main`:

```
	…
//Registrar los callbacks
	glutDisplayFunc(OnDraw);
	glutTimerFunc(25,OnTimer,0);//le decimos que dentro de 25ms llame 1
				    //vez a la función OnTimer()
	glutKeyboardFunc(OnKeyboardDown);
	glutSpecialFunc(OnSpecialKeyboardDown); //gestion de los cursores
	//Inicializacion de la escena
	mundo.Inicializa();
	…
```

4. Compilamos y ejecutamos el programa para comprobar que funciona correctamente y que ahora el jugador puede moverse libremente de izquierda a derecha. Se observa de inmediato que nadie ha puesto límites a este movimiento, por lo que se saldrá de la caja si así se lo ordenamos.

Es importante observar que en programación no se hará ninguna operación que no se indique en el código. Si deseamos que el jugador quede confinado en el interior de la caja será necesario indicarlo. En un principio, una solución inmediata, aunque no correcta desde el punto de vista de una programación ordenada, sería no incrementar la posición salvo en un sentido si el jugador se encuentra en el límite. Algo así como lo siguiente:

```
void Hombre::Mueve(float t)
{
	posicion=posicion+velocidad*t+aceleracion*(0.5f*t*t);
	velocidad=velocidad+aceleracion*t;
```

```
      //NO es la forma adecuada de resolverlo
      if(posicion.x>10.0f)posicion.x=10.0f;
      if(posicion.x<-10.0f)posicion.x=-10.0f;
}
```

Se observa que con esta modificación el hombre queda confinado en el interior de la caja, sin embargo se está haciendo uso de una información –los límites de la caja- que no son del objeto lo cual no es correcto. Como no es la solución que se va a adoptar, **eliminamos** las dos líneas que se han introducido.

Para realizar las interacciones entre objetos que son del mismo nivel, la caja y el hombre, es conveniente acudir a una estructura como la que se expone a continuación:

5.5. INTERACCIÓN ENTRE OBJETOS

Se va a crear a continuación una clase cuya única misión es la de contener las posibles funciones que gestionarán las interacciones entre los distintos objetos de la escena. Esta clase se llamará `Interaccion`, y no tendrá en si entidad, puesto que no tendrá datos, sino que simplemente será responsable de agrupar todas estas funciones. Esta clase implementará las interacciones típicas (rebotes, colisiones) entre pares de objetos pertenecientes al dominio, luego parece lógico ubicar esta clase en las carpetas "dominio"

Comenzamos creando la clase y un método que gestione el movimiento del `Hombre` dentro de la `Caja`. Utilizando el menú contextual sobre el *ClassView*, creamos la clase `Interaccion`, lo que provocará la aparición de dos nuevos ficheros en el proyecto: ***Interaccion.cpp*** e ***Interaccion.h***. **Importante**: recordar definir la ubicación correcta en las subcarpetas "dominio".

Agregamos el método `static void rebote()`, que tiene como argumentos las referencias a un hombre y una caja, y cuya función es la de confinar al hombre en el interior de la caja, de forma que si éste se sale de los límites fuerza a que esté en el interior:

```
void Interaccion::Rebote(Hombre &h, Caja c)
{
      float xmax=c.suelo.limite2.x;
      float xmin=c.suelo.limite1.x;
      if(h.posicion.x>xmax)h.posicion.x=xmax;
      if(h.posicion.x<xmin)h.posicion.x=xmin;
}
```

En la declaración del método indicamos que es un método `static`. Esto es porque es una función que queremos utilizar independientemente de que haya o no un objeto. En el fondo indicamos que es una función disponible para todo el mundo, pero que queda agrupado dentro de una clase de la que no vamos a generar objetos a la que hemos llamado `Interaccion`. También ponemos los `includes` necesarios:

```
#include "Hombre.h"
#include "Caja.h"

class Interaccion
{
```

```
public:
      static void Rebote(Hombre& h, Caja c);
      Interaccion();
      virtual ~Interaccion();
};
```

En este caso, hemos considerado que el hombre sólo se puede mover de izquierda a derecha, por lo que no comprobamos si la posición en y está fuera de los límites. Se observa que con esta estructura, el hombre no tiene porque saber las dimensiones de la caja, sino que esta es una información que tiene este último objeto. De esta forma, si se decidiera modificar el tamaño y posición de la misma, no habría que modificar el código de interacción.

Este método deberá ser llamado cada vez que sea movido el hombre. Por tanto un buen lugar para incluir la ejecución de este método es en la función recién creada en la clase mundo que decide mover el hombre cada vez que se pulsa una tecla de los cursores:

```
void Mundo::Mueve()
{
      hombre.Mueve(0.025f);
      esfera.Mueve(0.025f);
      bonus.Mueve(0.025f);
      disparo.Mueve(0.025f);
      Interaccion::Rebote(hombre,caja);
}
```

Obsérvese el uso del operador `::`, que indica que se trata del método estático de la clase `Interaccion`. Evidentemente, para poder usar esta clase, es necesario incluir su fichero de cabecera.

Sin embargo, si intentamos compilar el programa surgirán muchos errores debido a que el método ejecutado está intentando acceder a partes privadas de la clase `Hombre`, `Caja` y `Pared`. Aquí es donde introducimos por vez primera un concepto nuevo que es el concepto de amistad (`friend`).

A partir de ahora, todos los objetos susceptibles de interaccionar con otros vamos a establecer que se considerarán amigos de la clase `Interaccion`, dando de esta forma permiso de acceso a sus atributos y métodos privados. Esta operación es tan sencilla como incluir la siguiente línea en las declaraciones de las clases que permiten el acceso a los métodos de interacción:

```
      friend class Interaccion;
```

Por tanto, al final de la declaración de `Pared`, `Caja` y `Hombre`, agregamos este código. No importa que nos encontremos en la zona privada o pública, puesto que es una concesión de permisos. Tampoco es necesario incluir los ficheros de cabecera de interacción para declarar la relación de amistad. A partir de este momento, cualquier método que incluyamos en interacción podrá acceder a la parte protegida de las clases que la han declarado como amiga.

5. Ahora sí, compilamos el código y verificamos que funciona correctamente.

5.5.1. El patrón indirección

Cuando hemos creado la clase `Interaccion`, realmente estamos utilizando un patrón de diseño GRASP. Nos hemos inventado una clase que no pertenece al dominio, es

decir hemos empleado otro patrón GRASP denominado Fabricación Pura. El objetivo de inventarnos esta clase es evitar el acoplamiento directo (patrón GRASP Bajo Acoplamiento) entre las distintas clases.

Para implementar la funcionalidad anterior, se podría haber añadido una función a la clase `Hombre` que hiciera esta tarea:

```
#include "Caja.h"
class Hombre
{
…
      void Chequea(Caja caja);
};
```

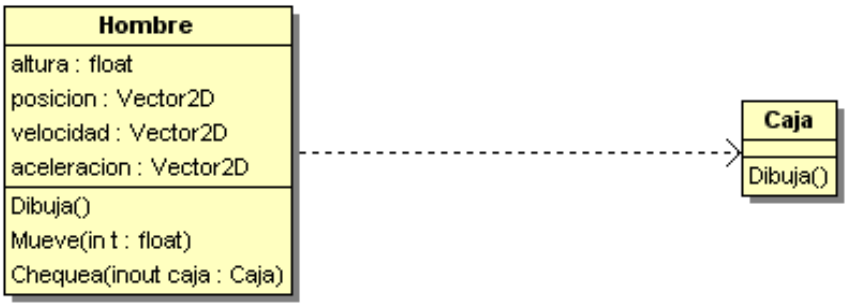


Figura 5-3. Acoplamiento directo entre Hombre y Caja

o tal vez añadiendo una función a la clase `Caja`, con el mismo propósito, que podría tener el siguiente prototipo:

```
#include "Hombre.h"
class Caja
{
…
      void Chequea(Hombre& hombre);
};
```

Figura 5-4. Acoplamiento directo entre Caja y Hombre

Llevándonos la funcionalidad a la clase intermedia `Interaccion`, conseguimos que las clases `Hombre` y `Caja` no estén relacionadas entre ellas directamente. Nótese como no existen `#include`s entre dichas clases.

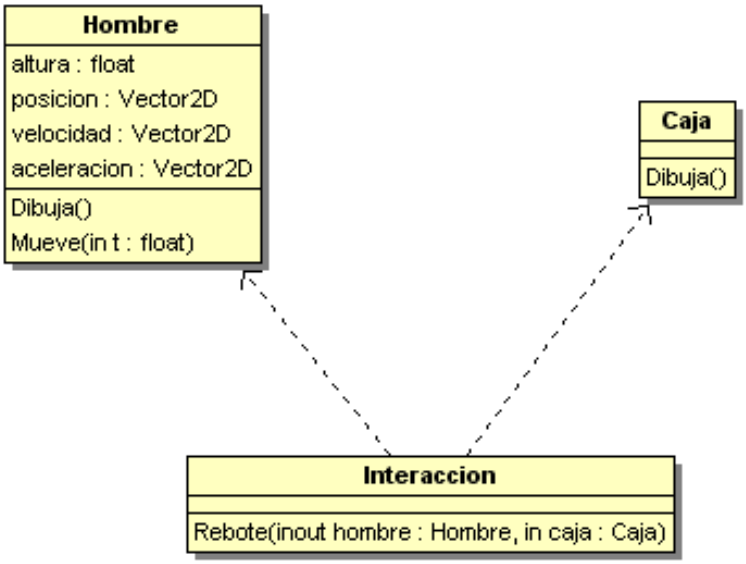


Figura 5-5. El patrón GRASP Indirección evita el acoplamiento entre Hombre y Caja

5.5.2. Interacción de la esfera con las paredes.

La interacción del hombre con la caja, es posiblemente la más sencilla tanto desde el punto de vista de la programación como matemáticamente. A continuación vamos a codificar el rebote de una esfera con una pared, para después poder realizar el rebote de las esferas con las paredes de la caja. El prototipo de esta función será:

```
static bool Rebote(Esfera &e,  Pared p);
```

y devolverá `true` si se ha producido una colisión entre ambos objetos, y como consecuencia ha tenido que ser modificada la velocidad de la esfera.

Las matemáticas que se han de implementar en caso de impacto son las siguientes:

$$\overrightarrow{V_{out}} = \overrightarrow{V_{in}} + 2(\overrightarrow{V_{in}} \cdot \vec{n})\vec{n}$$

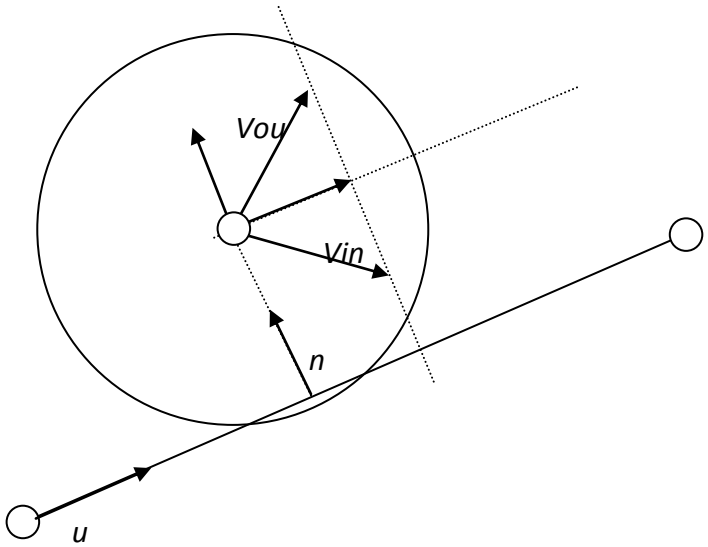


Figura 5-6. Rebote de una esfera en una pared

Luego dado el vector de velocidad, y el vector director de la recta utilizada para medir la distancia entre el segmento y la esfera (en el sentido desde el segmento a la esfera), la obtención de la nueva velocidad de la esfera es inmediata.

Por tanto, antes de codificar el rebote, se va incluir una función que permita obtener la distancia de un punto a una pared, y que además nos informe de la dirección con la que se ha medido esta distancia. De esta forma será posible calcular el efecto del rebote.

1. Agregamos el siguiente método en la clase `Pared`:

```
float Distancia(Vector2D punto, Vector2D *direccion=0);
```

El código que lo implementa y que ya se puso como ejercicio anteriormente es el siguiente:

```
//Calculo de distancia de una pared a un punto, adicionalmente
//se modifica el valor de un vector direccion opcional que contendrá
//el vector unitario saliente que indica la direccion de la
//recta más corta entre el punto y la pared.
float Pared::Distancia(Vector2D punto, Vector2D *direccion)
{
      Vector2D u=(punto-limite1);
      Vector2D v=(limite2-limite1).Unitario();
      float longitud=(limite1-limite2).modulo();
      Vector2D dir;
      float valor=u*v;
      float distancia=0;

      if(valor<0)
            dir=u;
      else if(valor>longitud)
            dir=(punto-limite2);
      else
            dir=u-v*valor;
      distancia=dir.modulo();
      if(direccion!=0) //si nos dan un vector…
            *direccion=dir.Unitario();
      return distancia;
}
```

2. Una vez incluido este método, procedemos a codificar la función de la clase `Interaccion` que gestionará el rebote entre esferas y paredes. Su prototipo es el siguiente:

```
static bool Rebote(Esfera &e, Pared p);
```

y el código que implementa la función según la matemática indicada es el siguiente:

```
bool Interaccion:: Rebote(Esfera &e, Pared p)
{
      Vector2D dir;
      float dif=p.Distancia(e.posicion,&dir)-e.radio;
      if(dif<=0.0f)
      {
            Vector2D v_inicial=e.velocidad;
            e.velocidad=v_inicial-dir*2.0*(v_inicial*dir);
            e.posicion=e.posicion-dir*dif;
            return true;
      }
      return false;
}
```

No hay que olvidar indicar que `Interaccion` es clase amiga de la clase `Esfera`.

EJERCICIO: Una vez preparada la matemática que calcula el rebote entre una esfera y cualquier pared, se propone como ejercicio, agregar la función static void Rebote(Esfera &e, Caja c) que gestionará el rebote de la esfera con las paredes de la caja.

3. Finalmente, llamamos a esta función tras calcular el movimiento de los distintos objetos en la clase `Mundo`:

```
void Mundo::Mueve()
{
	hombre.Mueve(0.025f);
	esfera.Mueve(0.025f);
	bonus.Mueve(0.025f);
	disparo.Mueve(0.025f);
	Interaccion::Rebote(hombre,caja);
	Interaccion::Rebote(esfera,caja);
}
```

Compilamos y observamos el resultado en el que se observa como la esfera rebota contra el suelo. Para comprobar que la esfera rebota contra todas las paredes de la caja, se propone modificar la velocidad inicial de la misma, de forma que parta en un movimiento diagonal y rápido. Para poder dar una velocidad inicial a la `Esfera` será necesario crear un método de interfaz que nos permita modificar este valor. Se propone dar un impulso de 5, 15.

Con ello se observa que la esfera rebota contra las cuatro paredes que conforman la caja tal y como se ha programado. Sin embargo también se pone de manifiesto que la esfera no rebota contra la plataforma que está en medio. Esto es debido a que no lo hemos indicado en el código. Al seguir un método de programación orientado a objetos, en donde las responsabilidades están claramente definidas, y la información está encapsulada en las unidades que son afectadas por la misma, agregar esta funcionalidad es tan sencillo como añadir la línea siguiente al código del método `Mueve()` de la clase `Mundo`:

```
Interaccion::Rebote(esfera,plataforma);
```

5.6. EJERCICIO: INCLUYENDO MÁS DE UNA ESFERA.

En el juego del Pang, se gestionan varias esferas y varias plataformas, que además interaccionan entre sí. Así, si tenemos dos esferas, ambas pueden chocar en el aire y como consecuencia rebotarán modificando sus respectivas trayectorias. Para poder plantear este problema vamos a introducir una segunda esfera en la escena. Para ello agregamos un nuevo objeto a la clase `Mundo`, llamado `esfera2` de tipo `Esfera`, y añadimos el método `SetVel()` a la misma, que permita cambiar su velocidad.

Ahora debemos preocuparnos de inicializar, pintar, mover y comprobar las interacciones de este nuevo objeto para que aparezca coherentemente en la escena. Modificamos por tanto el método `Inicializa()` de la clase `Mundo`:

```
void Mundo::Inicializa()
{
	x_ojo=0; y_ojo=7.5;
	z_ojo=30;
	esfera.SetColor(0,0,255);
	esfera.SetRadio(1.5f);
	esfera.SetPos(2,4);
	esfera.SetVel(5,15);
	esfera2.SetRadio(2);
	esfera2.SetPos(-2,4);
	esfera2.SetVel(-5,15);
	bonus.SetPos(5.0f,5.0f);
	disparo.SetPos(-5.0f,0.0f);
	plataforma.SetPos(-5.0f,9.0f,5.0f,9.0f);
}
```

Y agregamos la línea que pinta esta segunda esfera en el método `Dibuja`:

```
void Mundo::Dibuja()
{
	gluLookAt(x_ojo, y_ojo, z_ojo,  // posicion del ojo
		  0.0, y_ojo, 0.0,      // hacia que punto mira  (0,0,0)
		  0.0, 1.0, 0.0);       // definimos hacia arriba (eje Y)
	esfera.Dibuja();
	esfera2.Dibuja();
	caja.Dibuja();
	hombre.Dibuja();
	disparo.Dibuja();
	plataforma.Dibuja();
	bonus.Dibuja();
}
```

De igual forma introducimos el código correspondiente en el método `Mueve()`.

```
void Mundo::Mueve()
{
	hombre.Mueve(0.025f);
	esfera.Mueve(0.025f);
	esfera2.Mueve(0.025f);
	bonus.Mueve(0.025f);
	disparo.Mueve(0.025f);
	Interaccion::Rebote(hombre,caja);
	Interaccion::Rebote(esfera,caja);
	Interaccion::Rebote(esfera,plataforma);
	Interaccion::Rebote(esfera2,caja);
	Interaccion::Rebote(esfera2,plataforma);
}
```

Se observa, tal como muestra la figura, que tenemos ahora ambas esferas funcionando perfectamente, pero que entre ellas no existe interacción. Evidentemente, no hemos codificado la función correspondiente, que deberá ser llamada una vez que las esferas han sido movidas.

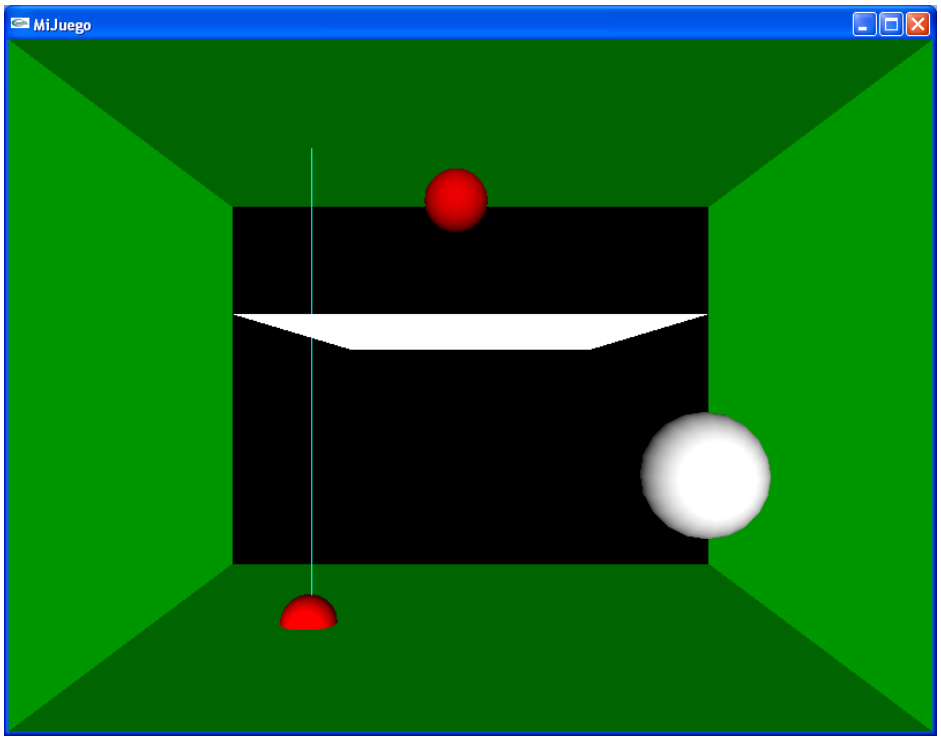


Figura 5-7. Estado final del Juego tras el capítulo 5.

5.7. EJERCICIO PROPUESTO

Por ello, finalmente se propone como ejercicio para continuar en el siguiente capítulo la codificación y uso de la función de la clase `Interaccion`, que calcule el rebote entre dos esferas, de forma que este siga la mecánica del impacto elástico entre dos objetos. Es decir, que se cumpla la conservación vectorial de la cantidad de movimiento y el principio de conservación de la energía, considerando la masa proporcional al área del objeto (en vez de al volumen).

6. CREACIÓN Y DESTRUCCIÓN DE OBJETOS

Con el tema anterior se completó el marco básico de los elementos que constituyen el armazón del juego del Pang, así como sus interacciones básicas. En este capítulo se desarrolla la gestión de conjuntos de objetos, principalmente el conjunto de esferas, a través de algunos mecanismos de la POO.

En el código suministrado para el desarrollo de este capítulo se ha incluido el método estático `Interaccion::Rebote(Esfera& e1, Esfera& e2)`, cuya finalidad es la de calcular el rebote entre dos esferas simulando el comportamiento físico real. Su complejidad más que en la codificación se encuentra en la física y geometría implicada en el cálculo de las velocidades resultantes tras el choque.

Para demostrar su uso, se ha incluido una llamada a dicha función, de forma que se observa el rebote entre las dos esferas que hasta el momento constituían la escena. Esta llamada está situada en el cuerpo de la función `Mueve()` de la clase `Mundo`:

```
void Mundo::Mueve()
{
      hombre.Mueve(0.025f);
      esfera.Mueve(0.025f);
      esfera2.Mueve(0.025f);
      bonus.Mueve(0.025f);
      disparo.Mueve(0.025f);

      Interaccion::Rebote(hombre,caja);
      Interaccion::Rebote(esfera,caja);
      Interaccion::Rebote(esfera,plataforma);
      Interaccion::Rebote(esfera2,caja);
      Interaccion::Rebote(esfera2,plataforma);
      Interaccion::Rebote(esfera,esfera2);
}
```

Si se ejecutase el juego ya terminado, se observaría que las esferas son objetos que varían en su cantidad de forma continuada. A veces se tienen dos, tres o cuatro esferas. No importa cuantas tengamos que cada una de ellas rebotará contra las paredes y si son impactadas por un disparo se duplican, y las esferas resultantes vuelven a tener el comportamiento programado para una esfera individual. Se muestra a continuación, el código que haría falta agregar si en vez de dos esferas tuviésemos cuatro:

```
void Mundo::Mueve()
{
```

```
        hombre.Mueve(0.025f);
        bonus.Mueve(0.025f);
        disparo.Mueve(0.025f);
        //mover esferas
        esfera.Mueve(0.025f);
        esfera2.Mueve(0.025f);
        esfera3.Mueve(0.025f);
        esfera4.Mueve(0.025f);

        //chocar esfera con la caja
        Interaccion::Rebote(esfera,caja);
        Interaccion::Rebote(esfera2,caja);
        Interaccion::Rebote(esfera3,caja);
        Interaccion::Rebote(esfera4,caja);

        //chocar esfera con la plataforma
        Interaccion::Rebote(esfera,plataforma);
        Interaccion::Rebote(esfera2,plataforma);
        Interaccion::Rebote(esfera3,plataforma);
        Interaccion::Rebote(esfera4,plataforma);

        //choque de esferas entre sí
        Interaccion::Rebote(esfera,esfera2);
        Interaccion::Rebote(esfera,esfera3);
        Interaccion::Rebote(esfera,esfera4);
        Interaccion::Rebote(esfera2,esfera3);
        Interaccion::Rebote(esfera2,esfera4);
        Interaccion::Rebote(esfera3,esfera4);

        Interaccion::Rebote(hombre,caja);
}
```

Los comentarios indican la intención del programador, pero el código se va extendiendo de forma progresiva, de una forma que obviamente no es generalizable para cualquier número de esferas. Se observa un patrón en el código que lo que manifiesta es que hay una serie de operaciones que deben aplicarse a cada esfera del conjunto de esferas. ¿Podría crearse algún tipo de objeto que las agrupase, de forma que se pudiera decir: detectar choque de cualquier esfera contra la caja?

6.1. CREANDO LA CLASE LISTAESFERAS

Observamos que hay un elemento especialmente dinámico en cuanto a creación, destrucción, número e iteraciones, dentro del programa. Este es el caso de la `Esfera`. De alguna manera, el programa tiene que trabajar con un número continuamente variable de esferas, ya que estas se duplican y se destruyen continuamente, y mientras tanto hay que estar comprobando si son impactadas por el disparo, si se chocan entre ellas o si chocan contra las paredes.

Por ello parece útil el diseño de una clase que contenga las esferas y que se preocupe de gestionar su aparición y desaparición, así como las operaciones como pintar o mover que se realizan en todas ellas.

A esta clase la denominaremos `ListaEsferas` y deberá suministrar la siguiente funcionalidad:

- Gestionará una lista de esferas, permitiendo agregar, quitar o eliminar esferas creadas externamente.
- Nos dará un acceso elegante a cada una de las esferas.
- Realizara automáticamente operaciones comunes a todo el conjunto de las esferas tales como Pintar, Mover o rebotar contra una pared, o contra la caja.
- Realizará operaciones que supongan la interacción exclusiva de los objetos contenidos entre si. Es decir, gestionará el rebote entre las esferas.

Nótese que según esta funcionalidad, la clase *contenedora* que estamos diseñando, en un principio no crea ni destruye los objetos, sino que los recibe y los agrupa. Es posible, sin cometer un error de diseño, dar a la clase la responsabilidad de producir o destruir objetos, siempre que estos sean consecuencia directa de una petición desde el exterior. También es importante notar que al contrario que el resto de clases, cuyos conceptos aparecen en singular (`Esfera, Disparo`), la clase `ListaEsferas` parece plural, pero no lo es. El concepto es también singular (podemos tener una lista de esferas)

Aunque se podría plantear el crear una clase que pudiera contener todas las esferas que quisiéramos, por simplificar de momento el código se va a limitar su número a un máximo fijo.

Por tanto, mediante el botón derecho sobre el icono que representa al proyecto accedemos a *New Class* y creamos la clase genérica `ListaEsferas`, que guardaremos en las subcarpetas “dominio”:

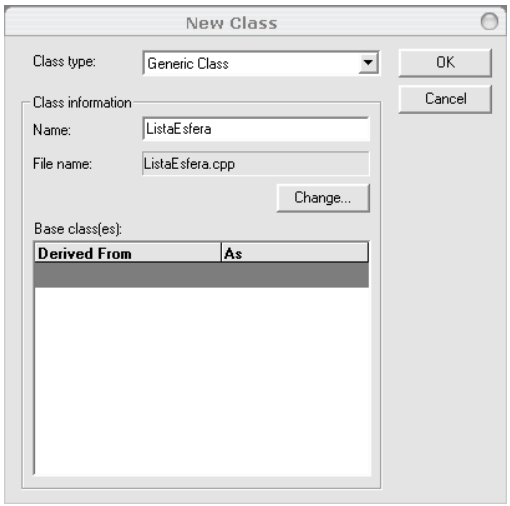


Figura 6-1. Adición de la clase ListaEsferas

Puesto que las esferas serán creadas externamente a la clase, lo que va a hacer esta es mantener una lista con los sitios de la memoria en donde se encuentran cada una de las distintas esferas que debe manejar. Esto se implementará mediante la inclusión de dos atributos **privados**:

```
class ListaEsferas
{
public:
      ListaEsferas();
      virtual ~ListaEsferas();
private:
```

```
        Esfera * lista[MAX_ESFERAS];
        int numero;
};
```

El primero es un vector de punteros a objetos de tipo `Esfera`. Es decir que tendremos la posibilidad de almacenar hasta un número `MAX_ESFERAS` de posiciones de memoria en las que se encuentra una esfera. El segundo atributo, es necesario para llevar cuenta de cuantas esferas llevamos apuntadas. De esta forma es posible recorrer sólo los elementos del vector que tienen direcciones válidas, porque se han rellenado con posiciones de memoria de objetos ya creados.

El valor de `MAX_ESFERAS`, se puede definir al comienzo del fichero de cabecera de la clase, asignando un valor de `100` por ejemplo:

```
#define MAX_ESFERAS 100
```

En el caso de los contenedores, es especialmente importante proteger los atributos que llevan cuenta de los objetos. Esto es así porque si fueran públicos sería posible modificar la cuenta de objetos que se tienen almacenados desde el exterior de la clase con el problema de funcionamiento que esto puede generar.

Conceptualmente, la clase `ListaEsferas` es una agregación de esferas (nótese la diferente representación respecto de la relación de composición), lo que se puede representar en un diagrama UML (en el diagrama de clases de diseño DCD) de nuestra aplicación como:

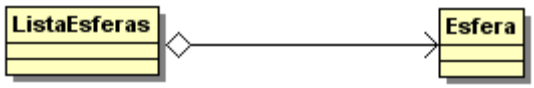


Figura 6-2. La clase ListaEsferas como agregación de Esfera

Una vez preparados los atributos y el armazón de la clase se seguirán los siguientes pasos:

6.1.1. Inicialización de la lista

En primer lugar se asegurará que al principio la información contenida es ninguna, y por tanto, al crear una instancia de la clase `ListaEsferas`, es necesario dar valor cero al atributo `numero`. De igual forma, es conveniente indicar que ningún puntero apunta de momento a nada, puesto que no se ha agregado ninguna esfera al conjunto. Esto se hace poniendo a cero el valor de cada uno de los 100 punteros. Para codificar la inicialización, lógicamente, rellenaremos el código del constructor por defecto, que el compilador ya ha preparado al generar la clase:

```
ListaEsferas::ListaEsferas()
{
        numero=0;
        for(int i=0;i<MAX_ESFERAS;i++)
                lista[i]=0;
}
```

6.1.2. Adición de esferas

Una vez inicializados los atributos de la clase se va a dotar de la función que nos permita añadir esferas al conjunto. Las operaciones que se deben realizar son las siguientes. Es posible que se llegue a la situación de que no quepan más esferas en el contenedor porque se haya alcanzado su capacidad máxima. Es ese caso es conveniente que la función informe de que la esfera no ha podido ser agregada. Según lo descrito el prototipo del método deberá adoptar la siguiente forma:

```
bool Agregar (Esfera *e);
```

De forma que si se agrega la esfera, el método retornará el valor `true`, y devolverá `false` en caso contrario. Lógicamente, puesto que lo que se va a almacenar son direcciones, lo que recibe la función será también una dirección.

Una vez comprobado que se puede agregar una esfera, se deben realizar los pasos que se describen a continuación. En primer lugar se almacena la dirección en el último puesto del vector sin rellenar verificando que no se ha superado la capacidad máxima del contenedor. En segundo lugar se indica que el número de esferas apuntadas por el contenedor se ha incrementado en uno. Tanto en C como en C++ esto se suele realizar en una sola sentencia aprovechando el modo de funcionamiento del operador post incremento. Por tanto, el código que se ha de implementar es el siguiente:

```
bool ListaEsferas::Agregar (Esfera *e)
{
      if(numero<MAX_ESFERAS)
            lista[numero++]=e;
      else
            return false;
      return true;
}
```

6.1.3. Dibujo y movimiento de las esferas

Ahora la clase `ListaEsferas` ya puede recibir objetos de tipo esfera. Recuérdese que la función principal de esta clase es la de agrupar operaciones, de forma que con una sola instrucción se pueda hacer que todas las esferas contenidas, se pinten o se muevan Estas dos funciones son las que se implementarán, y básicamente consistirá en recorrer las esferas e ir diciéndole a cada una que ejecute su método correspondiente. El código de las mismas es el que se pone a continuación, que como se ve es sencillo.

```
void ListaEsferas::Dibuja()
{
      for(int i=0;i<numero;i++)
            lista[i]->Dibuja();
}

void ListaEsferas::Mueve(float t)
{
      for(int i=0;i<numero;i++)
            lista[i]->Mueve(t);
}
```

Puesto que lo que se almacenan son direcciones, para poder acceder a la ejecución de un método de la esfera apuntada es necesario recurrir al operador `->`. El código, lo

único que hace es recorrer las `numero` direcciones de esferas apuntadas, e ir ejecutando sus métodos `Dibuja` y `Mueve` respectivamente. Como para mover una esfera es necesario pasar el parámetro del intervalo de tiempo, la función mover de `ListaEsferas`, también requerirá de dicho argumento.

6.2. USANDO LA CLASE LISTAESFERAS

Con esto ya tenemos la funcionalidad mínima para poder probar el funcionamiento de lo que se lleva programado. Agregamos un atributo de tipo `ListaEsferas` a la clase `Mundo`, al cual vamos a llamar `esferas`.

Obviamente, el `include` correspondiente a la clase `ListaEsferas` es necesario en el fichero ***Mundo.h.***

```
class Mundo
{
	…
private:
	ListaEsferas esferas;
	…
};
```

Agregamos la llamada a las funciones `Dibuja y Mueve` de esferas en las funciones respectivas de la clase mundo, de igual forma a como se procedió cuando añadimos el resto de objetos de la escena:

```
void Mundo::Dibuja()
{
	…
	esferas.Dibuja();
}

void Mundo::Mueve()
{
	…
	esferas.Mueve(0.025f);
	…
}
```

Evidentemente, si ejecutamos el código, no se observa ningún cambio, puesto que el contenedor está vacío de objetos. Finalmente para probarlo, vamos a agregar unas cuantas esferas al contenedor, y observamos lo que ocurre. Para ello, en la inicialización del mundo, creamos una serie de esferas -en concreto seis- y las introducimos en `esferas`:

```
void Mundo::Inicializa()
{
	…

	for(int i=0;i<6;i++)
	{
		Esfera* aux=new Esfera;
		aux->SetPos(i,1+i);
```

```
            aux->SetVel(i,i);
            aux->SetRadio(0.75+i*0.25);
            esferas.Agregar(aux);
      }
}
```

6.2.1. Sobrecarga de constructores

En general, cada vez que queremos crear una nueva esfera, será necesario indicar una serie de atributos básicos que definen el objeto. Algunos son claramente auxiliares, como puede ser el color, pero otros casi siempre hemos de indicarlos tras haber creado el objeto.

En el caso anterior se observa claramente como es conveniente especificar el radio, la posición y la velocidad, para diferenciar las distintas esferas que se han creado.

Se podría compactar mucho más el código si permitimos definir estos atributos en el momento de creación del objeto. Esto se puede hacer gracias a la sobrecarga del constructor, que en C++ se permite.

Por ello, a continuación, se va a implementar un nuevo constructor para la clase `Esfera` que espera la inclusión de parámetros como el radio y la posición, y cuyo prototipo sería:

```
Esfera(float rad, float x=0.0f, float y=0.0f,
       float vx=0.0f, float vy=0.0f);
```

En donde indicamos que tanto la posición como la velocidad asumirán por defecto el valor nulo si no se utilizan cuando se invoque al constructor. Al igual que cualquier otra función, los constructores admiten la definición de valores por defecto. Los argumentos asumirán el valor indicado en caso de que el programador no los defina. Por el modo de proceder de este mecanismo, lo normal es poner como primeros argumentos aquellos que tengan más posibilidades de ser definidos explícitamente por el programador, y después por orden decreciente de importancia, los que puedan ser asumidos por defecto.

Esto nos permitiría construir esferas de maneras muy diversas. Los siguientes, son ejemplos de las distintas sentencias válidas:

```
Esfera miesfera;              //constructor por defecto…sin argumentos
Esfera miesfera1(8.0F);       //nuevo constructor: radio 8
Esfera miesfera2(3.0F,2,5);         //nuevo constructor: radio 3  y
                                    //posición 2,5
Esfera miesfera3(2.0F,i,i*2,i+4,3);
Esfera *aux=new Esfera(5.0f);       //creación dinámica con el nuevo
                                    //constructor
```

La implementación del constructor quedaría como sigue:

```
Esfera::Esfera(float rad, float x, float y, float vx, float vy)
{
      radio=rad;
      posicion.x=x;
      posicion.y=y;
      velocidad.x=vx;
      velocidad.y=vy;

      rojo=verde=255;
      azul=100; //color distinto
```

```
        aceleracion.y=-9.8;
}
```

Rescribimos ahora el código de creación de las esferas utilizando este nuevo constructor:

```
        for(int i=0;i<6;i++)
        {
                Esfera* aux=new Esfera(0.75+i*0.25,i,1+i,i,i);
                esferas.Agregar(aux);
        }
```

6.3. REBOTES

Cuando ejecutamos el programa, nos damos cuenta que las esferas recién creadas se van de la pantalla. Esto se debe obviamente a que no estamos invocando las funciones de rebote correspondientes para cada una de las esferas de la clase `ListaEsferas`. Vamos a programar en esta sección esta funcionalidad.

6.3.1. Rebote con la caja

Lo primero que deseamos es que las esferas no salgan de la caja que define el área de juego. Es decir tenemos que implementar la funcionalidad de rebote de la lista de esferas y la caja. Para ello podríamos seguir diferentes alternativas. Por ejemplo podríamos decidir implementar un método en la clase `Caja`, que admita un parámetro de la clase `ListaEsferas` por referencia, para poder modificarlo si fuera necesario:

```
class Caja  //UNA POSIBLE OPCION
{
        friend class Interaccion;
public:
        Caja();
        virtual ~Caja();
        void Dibuja();
        void Rebote(ListaEsferas& lista_esferas);
};
```

No obstante esta alternativa violaría la encapsulación de `ListaEsferas`, al no poder acceder fácilmente a sus miembros privados. También podríamos intentar añadir un método a la clase `Interaccion`, aunque a priori también nos encontraríamos con el mismo problema:

```
class Interaccion  //OTRA POSIBLE OPCION
{
public:
        static bool Rebote(ListaEsferas& lista, Caja c);
};
```

Además, esta última opción rompe ligeramente el criterio adoptado cuando se desarrolló la clase `Interaccion`, que es que dicha clase se encarga de simular o calcular las interacciones físicas entre pares de objetos. Lo importante cuando se diseña

es establecer criterios y pautas. El criterio seguido en nuestro caso es el siguiente:

> La clase `ListaEsferas` es la responsable de agrupar un conjunto de esferas y de repetir para cada una de ellas tareas individuales, como que se pinte o que se mueva cada esfera. Parece por tanto lógico que la clase `ListaEsferas` sea la encargada de repetir las acciones de rebotes para cada una de sus esferas.

No queremos decir con ello que este sea el único criterio valido. De hecho se propone en el anexo un diseño más avanzado y arquitectónicamente más correcto. No obstante se mantiene el criterio anterior para los siguientes desarrollos por simplicidad y a título didáctico para el programador novel.

Construimos pues a continuación un método que compruebe el rebote de cada una de las esferas contenidas con una caja que se pasa por parámetro. El código resultante, es muy parecido al del apartado anterior, pero en este caso, hacemos uso de la función de la clase `Interacción` que nos permite calcular el rebote entre una caja y una esfera:

```
#include "DOMINIO\Interaccion.h"
…
void ListaEsferas::Rebote(Caja caja)
{
      for(int i=0;i<numero;i++)
            Interaccion::Rebote(*(lista[i]),caja);
}
```

Obviamente, es necesario invocar la función:

```
void Mundo::Mueve()
{
      …
      esferas.Rebote(caja);
      …
}
```

Obtendremos el resultado siguiente:

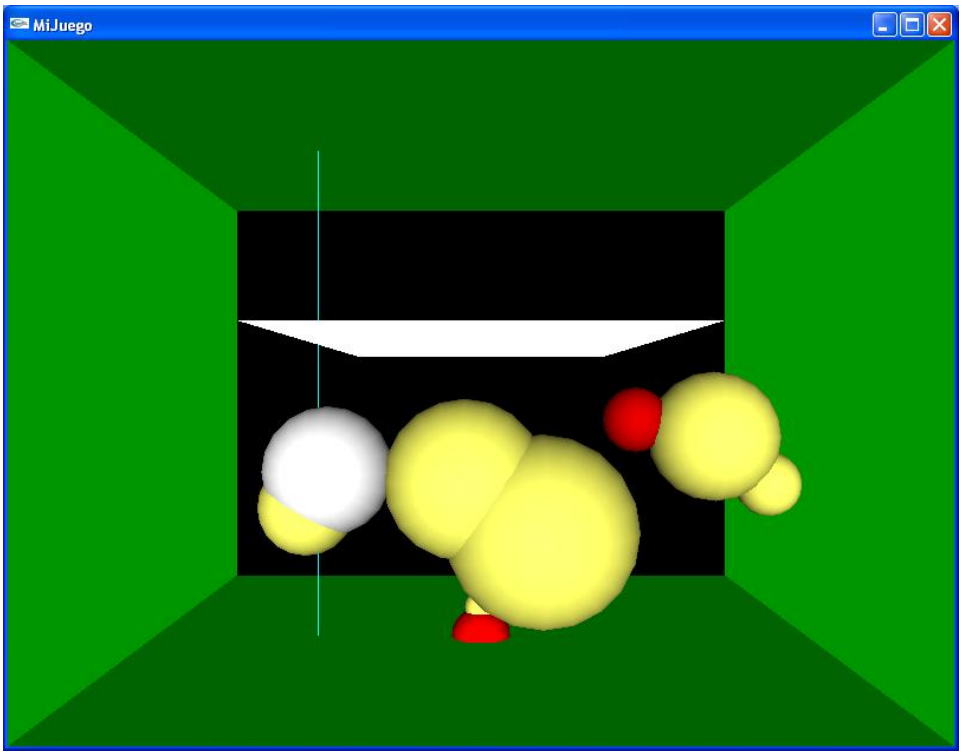


Figura 6-3. La clase ListaEsferas en funcionamiento

De nuevo, aunque la estructura es igual a los casos anteriores se observa una nueva diferencia. La función `Interaccion::Rebote(Esfera& e, Caja c)`, espera como argumentos una variable de tipo `Esfera` y otra variable de tipo `Caja`. En cuanto a la caja no hay ninguna dificultad, puesto que se recibe directamente como argumento de la función, sin embargo lo que se ha almacenado en la clase contenedora no son objetos de tipo `Esfera`, sino sus direcciones. Por ello, si se quiere acceder al objeto cuando lo que se tiene es la dirección es necesario hacer uso del operador contenido (*), tal y como se refleja en `*(lista[i])`.

6.3.2. Sobrecarga de los rebotes

Al igual que se ha sobrecargado un constructor, es posible sobrecargar otros métodos. Por ejemplo, en el caso de la lista de esferas, se observa, que interesa que se gestione no sólo el robote con la caja, sino también el rebote de las esferas con cualquier pared, y de las esferas entre si. Esto se puede realizar utilizando el mismo identificador para el método, pero cambiando el tipo de argumento que se utilizará.

Ahora procedemos a definir el método `void ListaEsferas::Rebote(Pared p)` cuya función es la de hacer que todas las esferas contenidas en la lista, reboten contra la pared que se pasa como argumento:

```
void ListaEsferas::Rebote(Pared p)
{
      for(int i=0;i<numero;i++)
            Interaccion::Rebote(*(lista[i]),p);
}
```

Para que comience a funcionar, será necesario incluir la llamada a este método en la función `Mueve()` de la clase `Mundo`. Aunque aparentemente la llamada a la función es la misma, el compilador diferencia el código que se debe ejecutar en función del tipo de parámetro que se pasa. Esto hace que el código sea muy legible, y facilita la escritura de código por parte del programador.

Más interesante, por ser un poco distinto, es plantearse cómo hacer el código de la función rebotar que gestiona el rebote de las distintas esferas entre sí. Puesto que es la clase contenedora la que contiene toda la información necesaria para este cálculo, deberá ser una función que no recibe ningún argumento.

EJERCICIO: Programar el rebote de las esferas entre si, mediante un doble bucle for() anidado

Codificar el método de la clase `ListaEsferas`, cuyo prototipo sea el siguiente, y que gestione el rebote entre las esferas contenidas:

```
void ListaEsferas::Rebote()
```

De esta forma el código de la función `Mueve()` finalmente contenido en `Mundo` y que gestiona las interacciones de las esferas contenidas en el contenedor es el siguiente:

```
    esferas.Rebote();
    esferas.Rebote(caja);
    esferas.Rebote(plataforma);
```

Al ejecutar el código se observa que hay dos conjuntos de esferas que no interaccionan entre sí, aunque sí que lo hacen con el resto de objetos de la escena. En el fondo eso es lo que dice el programa escrito, dado que tratamos de forma independiente a `esfera1` y `esfera2`, y por otro lado a las esferas contenidas en la lista de esferas.

6.4. EL DESTRUCTOR Y EL ENCAPSULAMIENTO

A estas alturas del libro, el lector ya habrá observado que el modo habitual de proceder en POO es tal que las clases suelen tener como parte privada –y por tanto no accesible desde el exterior- prácticamente todos los atributos de la clase, mientras que lo más normal es que los métodos sean públicos. Esta forma de implementación intenta compartimentar el código de forma que podemos asegurar que un objeto funciona por si mismo, incluso si se le piden cosas erróneas. Por ejemplo, la clase `ListaEsferas` tal y como está actualmente diseñada, impide que en una lista de una capacidad determinada, se introduzcan más objetos de los posibles.

Un aspecto importante de este modo de trabajar con objetos cerrados y protegidos es el control de su creación y su destrucción, por medio de los constructores y destructores. Gracias al constructor nos ha sido posible definir que inicialmente un lista de esferas no contiene ninguna esfera. Ahora vamos a diseñar el modo en que podemos destruir las esferas contenidas en la lista.

Hay que destacar que puesto que la clase `ListaEsferas` no ha sido la responsable de la creación de las esferas que contiene, deberá ser la clase creadora, en nuestro programa la clase `Mundo`, la que debe asumir esta responsabilidad, aunque lo haga a través de la petición de una acción a la clase `ListaEsferas`.

Lo vamos a ir haciendo por partes para ver posibles errores que se pueden cometer a la hora de diseñar el proceso de destrucción.

Declarar y definir el método `void ListaEsferas::destruirContenido()` cuya función es la de destruir todas las esferas cuya dirección se contiene, e inicializar la lista, dejándola preparada para volver a contener nuevas direcciones de esferas.

```
void ListaEsferas::DestruirContenido()
{
      for(int i=0;i<numero;i++)
            delete lista[i];

      numero=0;
}
```

Al terminar la ejecución de la clase `Mundo`, se ejecutará su destructor. Por tanto igual se debe indicar en este punto que se destruyan las esferas que habiendo sido creadas por esta clase, se almacenaron en la instancia de la clase `ListaEsferas`. Por tanto, editamos el contenido del destructor de `Mundo` y escribimos el siguiente código:

```
Mundo::~Mundo()
{
      esferas.DestruirContenido();
}
```

Aparentemente no ha habido ningún cambio significativo en el programa, sin embargo, se ha comenzado a gestionar correctamente la memoria. Buscamos asegurar que la clase `Mundo` destruye, o limpia, todo lo que ha ido creando.

Sin embargo, vamos a ver ahora uno de los posibles errores que se puede cometer con la destrucción de objetos. Para ello, vamos a hacer que `esfera1` y `esfera2` estén contenidos dentro de la lista de esferas, de forma que ya no hay que llamar a sus métodos de pintado, de mover y de interacción de forma independiente.

Por tanto, al inicializar el objeto de la clase `Mundo`, agregaremos:

```
      esferas.AgregarEsfera(&esfera1);
      esferas.AgregarEsfera(&esfera2);
```

Por otro lado, vamos limpiando el código de `Mundo`, de forma que `esfera` y `esfera2`, ya no son gestionados de forma independiente. Eliminamos por tanto las llamadas específicas de estos dos objetos en las funciones `Dibuja` y `Mueve` de la clase `Mundo`. Compilamos y ejecutamos. Parece que todo va bien, hasta el momento en que cerramos el programa. En ese momento se produce un error de ejecución.

Si vemos por medio del *Debugger* que es lo que está pasando, se observa que el error es producido precisamente por el destructor que acabamos de escribir. La razón es que estamos intentando destruir a `esfera` y `esfera2`, que son atributos de la clase `Mundo`. Recuérdese la regla:

> Lo que se reserva con `new`, se destruye con `delete`, y lo que se reserva con `new []` se destruye con `delete []`, pero NUNCA SE DEBE LLAMAR A DELETE sobre algo que no ha sido creado con NEW.

Al ser atributos no reservados dinámicamente mediante `new`, el sistema detectará que no es una zona de memoria que admita una operación de `delete` y da un error.

Lo solucionamos eliminando los atributos `esfera` y `esfera2` de la cabecera de la clase `Mundo`, de forma que ya no constituyan un estamento privilegiado en el código del programa. Quedando visualmente igual, y habiendo eliminado los atributos y **todo el código** que usaba estos datos, el método `Inicializa` de la clase `Mundo` quedará finalmente:

```
void Mundo::Inicializa()
{
	x_ojo=0;
	y_ojo=7.5;
	z_ojo=30;

	bonus.SetPos(5.0f,5.0f);
	disparo.SetPos(-5.0f,0.0f);
	plataforma.SetPos(-5.0f,9.0f,5.0f,9.0f);

	Esfera *e1=new Esfera(1,2,4,5,15);
	e1->SetColor(200,0,0);
	esferas.Agregar(e1); //esfera

	Esfera *e2=new Esfera(2,-2,4,-5,15);
	e2->SetColor(255,255,255);
	esferas.Agregar(e2); //esfera2

	for(int i=0;i<6;i++)
	{
		Esfera* aux=new Esfera(0.75+i*0.25,i,1+i,i,i);
		esferas.Agregar(aux);
	}
}
```

Un error solucionable por medio de la encapsulación es el que se produce cuando se intenta agregar una misma esfera dos veces. Si se ordenara la destrucción del contenido de lista esferas, daría error, puesto que se llamaría dos veces a `delete` sobre la misma dirección. En ese caso se producirá un error en la destrucción de nuevo porque se intenta destruir lo ya destruido.

Ejercicio: Implementar la funcionalidad necesaria en ListaEsferas::Agregar() para que no pueda ser agregada la misma esfera (el mismo puntero) mas de una vez.

6.5. AGREGANDO Y ELIMINANDO DINÁMICAMENTE ESFERAS

Durante la evolución del juego, se observa que una esfera grande impactada por un disparo, es eliminada y sustituida por otras dos de menor tamaño, que pasan a formar parte del escenario de juego. En el caso de que la esfera sea pequeña esta es destruida sin más. La agregación de esferas ya la tenemos resuelta, sin embargo, la destrucción y el acceso a las esferas contenidas en la lista no. Vamos a codificar esta funcionalidad:

Permitiremos la eliminación de esferas identificadas por su posición en el vector o directamente por su dirección. Por tanto, codificamos la siguiente función que permite borrar una esfera según su índice:

```
void ListaEsferas::Eliminar(int index)
```

```
{
	if((index<0)||(index>=numero))
		return;
	delete lista[index];
	numero--;
	for(int i=index;i<numero;i++)
		lista[i]=lista[i+1];
}
```

Y de la misma forma, otra función que permita borrar una esfera según su dirección de memoria, que se apoya en la función anterior:

```
void ListaEsferas::Eliminar(Esfera *e)
{
	for(int i=0;i<numero;i++)
		if(lista[i]==e)
		{
			Eliminar(i);
			return;
		}
}
```

Para poder probarlas vamos a hacer que cuando una esfera se choque con el hombre, sea destruida ("comida"). Aunque realmente este no es el comportamiento final del juego, es un paso intermedio útil.

Para ello comenzaremos por codificar un nuevo método de la clase `Interaccion` que nos informe de cuando una esfera colisiona con el hombre (la función devuelve `true` si hay colisión y `false` si no la hay). El método es una cruda aproximación, en la que solo se coge la posición central del hombre, pero realmente no se tiene en cuenta sus dimensiones. Nótese que también ha sido necesario implementar los métodos `GetAltura()` y `GetPos()` de `Hombre`, para acceder a sus datos privados:

```
bool Interaccion::Colision(Esfera e, Hombre h)
{
	Vector2D pos=h.GetPos(); //la posicion de la base del hombre
	pos.y+=h.GetAltura()/2.0f; //posicion del centro

	float distancia=(e.posicion-pos).modulo();
	if(distancia<e.radio)
		return true;
	return false;
}
```

Este método es utilizado por `ListaEsferas` para informarnos de la primera esfera de la lista que choca con la que se pasa como argumento.

```
Esfera * ListaEsferas::Colision(Esfera &e)
{
	for(int i=0;i<numero;i++)
	{
		if(Interaccion::colision(e,*(lista[i])))
			return lista[i];
	}
	return 0; //no hay colisión
}
```

Hay que destacar que el valor de retorno de la función es utilizado además para indicar si hay o no colisión. Si se devuelve un cero, (un puntero a NULL en C), indicamos que ninguna de las esferas contenidas choca con la pasada como argumento.

Modificamos el método `Mundo::Mueve()` de forma que utilizando dicha función, obtenemos la posible esfera que colisiona con el hombre, y la eliminamos de la clase `ListaEsferas`.

```
void Mundo::Mueve()
{
	hombre.Mueve(0.025f);
	bonus.Mueve(0.025f);
	disparo.Mueve(0.025f);

	esferas.Mueve(0.025f);
	esferas.Rebote();
	esferas.Rebote(plataforma);
	esferas.Rebote(caja);
	Esfera *aux=esferas.Colision(hombre);
	if(aux!=0)//si alguna esfera ha chocado
		esferas.Eliminar(aux);

	Interaccion::Rebote(hombre,caja);
}
```

Agregamos la funcionalidad de las teclas '1','2','3' y '4' en `Mundo`:

```
void Mundo::Tecla(unsigned char key)
{
	switch(key)
	{
		case '1':
			esferas.Agregar (new Esfera(0.5f,0,10));
			break;
		case '2':
			esferas.Agregar (new Esfera(1.0f,0,10));
			break;
		case '3':
			esferas.Agregar (new Esfera(1.5f,0,10));
			break;
		case '4':
			esferas.Agregar (new Esfera(2.0f,0,10));
			break;
	}
}
```

Ahora es cuando realmente podremos sorprendernos de lo cómodo que ha sido definir una clase contenedora que se encargue de gestionar con seguridad la evolución de las esferas. Si ejecutamos el programa podremos obtener resultados tan sorprendentes como el de la figura:

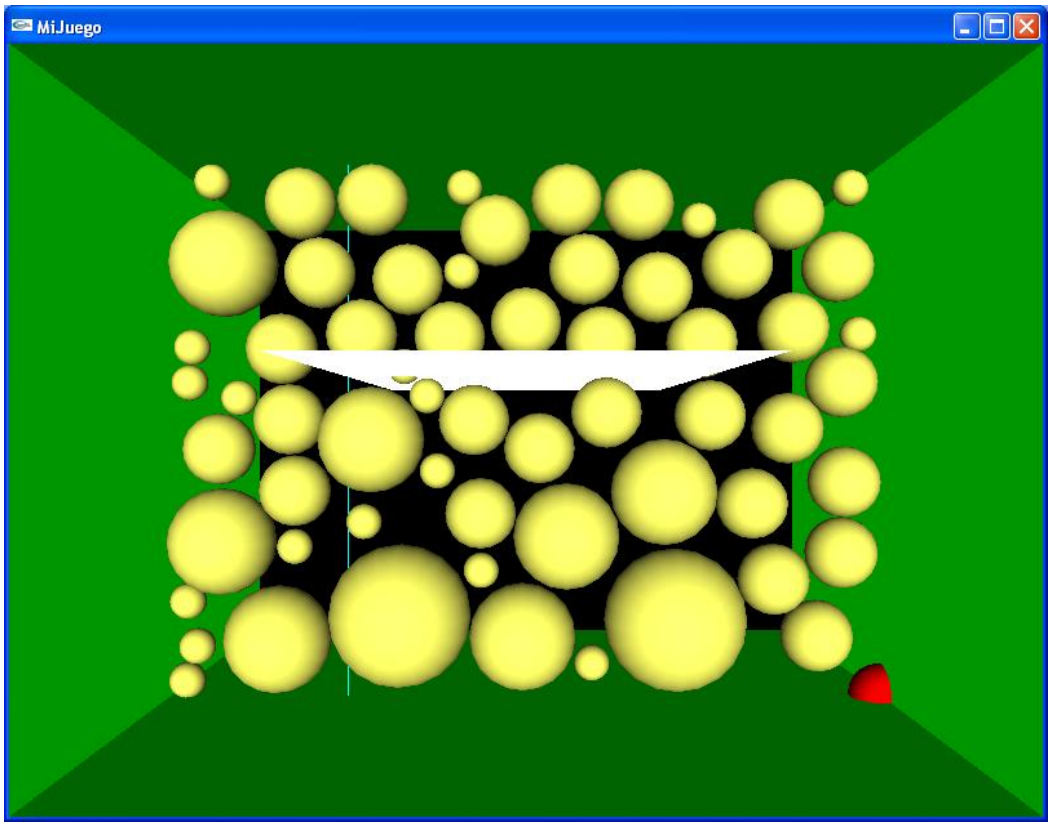


Figura 6-4. La utilidad de la clase ListaEsferas

6.6. ACCESO A LOS ELEMENTOS DE LISTAESFERAS

Para terminar, vamos a agregar dos métodos a la clase, que nos permitirán consultar las esferas contenidas en el contenedor.

Por un lado vamos a sobrecargar el operador `[]` de forma que nos dé la esfera situada en la posición indicada en el interior de los corchetes.

```
Esfera *ListaEsferas::operator [](int i)
{
	if(i>=numero)//si me paso, devuelvo la ultima
		i=numero-1;

	if(i<0) //si el indice es negativo, devuelvo la primera
		i=0;

	return lista[i];
}
```

Y por otro necesitaremos conocer el número de esferas contenidas en la lista, lo cual realizaremos directamente a través de una función `get inline` (en el fichero de cabecera ***ListaEsferas.h***):

```
int GetNumero(){return numero;}
```

Con ello es posible realizar operaciones específicas para cada esfera. Como ejemplo se puede codificar en la clase `Mundo` que las esferas que superen la altura 11 pasen a color rojo, y las que no a color blanco

```
      for(int i=0;i<esferas.GetNumero();i++)
            if((esferas[i]->GetPos()).y>11)
                  esferas[i]->SetColor(255,0,0);
            else
                  esferas[i]->SetColor(255,255,255);
```

Aparte de haber logrado un bonito cuadro "roji-blanco", y en contra de lo que parece, acabamos de construir una buena explicación del efecto de cómo es posible que el agua se evapore a temperaturas inferiores a 100 ºC. Cuando se pruebe este efecto, eliminar el código anterior, de tal forma que las esferas mantengan su color.

6.7. EJERCICIO PROPUESTO: LISTADISPAROS

Implementar la clase `ListaDisparos`, con la siguiente definición:

```
#define MAX_DISPAROS 10
#include "Disparo.h"
#include "Caja.h"

class ListaDisparos
{
public:
      ListaDisparos();
      virtual ~ListaDisparos();

      bool Agregar(Disparo* d);
      void DestruirContenido();
      void Mueve(float t);
      void Dibuja();

      void Colision(Pared p);
      void Colision(Caja c);

private:
      Disparo * lista[MAX_DISPAROS];
      int numero;

};
```

Para su correcto funcionamiento también se tienen que implementar los métodos:

```
class Interaccion
{
…
      static bool Colision(Disparo d, Pared p);
      static bool Colision(Disparo d, Caja c);

};
```

La clase `Disparo` debe de ser también completada para permitir acceso a sus variables:

```
class Disparo
{
…
```

```
        void SetVel(float vx, float vy);
        float GetRadio();
        Vector2D GetPos();

};
```

Los disparos se realizarán cuando el usuario pulse el espacio:

```
void Mundo::Tecla(unsigned char key)
{
        switch(key)
        {
                case ' ':
                        {
                                Disparo* d=new Disparo();
                                Vector2D pos=hombre.GetPos();
                                d->SetPos(pos.x,pos.y);
                                disparos.Agregar(d);
                                break;
                        }
…
}
```

Obviamente, el disparo existente debe de ser eliminado y se debe instanciar un objeto de la clase `ListaDisparos` en la clase `Mundo`. El resultado final es el siguiente, en el que los disparos se paran cuando colisionan con la caja o la pared.

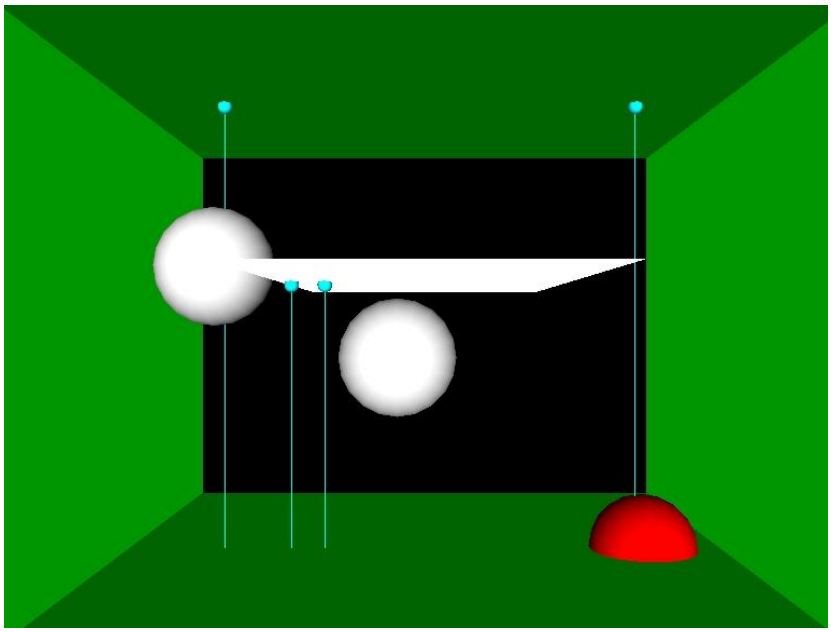


Figura 6-5. La lista de disparos en funcionamiento

6.8. ANEXO: ACERCA DE LA STL

Hemos implementado el vector interno a la clase `ListaEsferas` como un vector estático de una dimensión máxima definida por `MAX_ESFERAS`. C++ tiene una librería estándar, la *Standard Template Library* (STL) que permite (entre otras muchas cosas) el manejo de vectores dinámicos de cualquier tipo de objeto que pueden crecer o decrecer

en su tamaño. Se plantea en este anexo como se implementaría la funcionalidad anterior mediante el uso de las STL. No obstante, seguiremos usando en sucesivos capítulos nuestras clase `ListaEsferas` tal y como ha quedado implementada en los puntos anteriores, si bien gracias a la encapsulación se podría cambiar y el resto del programa no se vería afectado.

Como su propio nombre indica, la STL se construye principalmente haciendo uso de las plantillas (*templates*). Este es un mecanismo avanzado de la programación con C++, que queda fuera del alcance de este libro. De momento baste este ejemplo para ilustrar cómo se utiliza esta librería para implementar contenedores de objetos.

La declaración de un vector de la STL sería de la forma:

```
#include <vector>
class ListaEsferas
{
public:
      ListaEsferas();
      virtual ~ListaEsferas();
private:
      std::vector<Esfera*> lista;
};
```

Literalmente podemos traducir la última línea como “lista es un vector de punteros a Esfera”.

El vector inicialmente está vacío, y no es necesario inicializarlo:

```
ListaEsferas::ListaEsferas()
{
}
```

Para añadir una esfera basta con agregar al final del vector el elemento (push-back):

```
bool ListaEsferas::AgregarEsfera(Esfera *esf)
{
      lista.push_back(esf);
      return true;
}
```

Es importante darse cuenta, que en condiciones normales (no superemos la memoria física del computador), podremos añadir todas las esferas que queramos, por lo que la función siempre devuelve `true`.

La dimensión del vector se puede conocer con la función `size()`, y el acceso a las componentes del vector se realiza gracias a la sobrecarga de operadores como si de un vector ordinario se tratase. Por tanto, el código del método `Dibuja()` lo podemos escribir como sigue:

```
void ListaEsferas::Dibuja()
{
      for(int i=0;i<lista.size();i++)
            lista[i]->Dibuja();
}
```

La eliminación de esferas conlleva dos operaciones. Por un lado hay que eliminar de la memoria el objeto, y por otro hay que quitar el puntero de la lista. Para esta última

operación es necesario seguir la sintaxis escrita en la última sentencia para evitar fallos de indexación:

```
void ListaEsferas::EliminarEsfera(int ind)
{
	if((ind<0)||(ind>=numero))
		return;

	delete lista[ind];

	lista.erase(lista.begin()+ind);
}
```

6.9. ANEXO: DISEÑO DE LA GESTIÓN DE INTERACCIONES ENTRE LISTAS DE OBJETOS

A lo largo del capitulo se ha utilizado un criterio y una solución que no necesariamente es la idónea desde el punto de vista arquitectónico de la aplicación. Cuando se añaden las clases que implementan las listas sin incluir en ellas la gestión de las interacciones de los elementos de las mismas, se tiene un diagrama de clases como el de la figura:

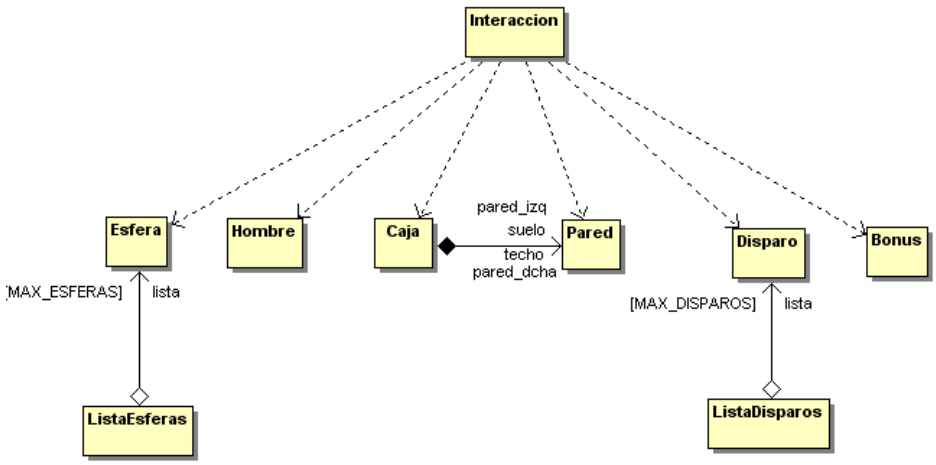


Figura 6-6. Diagrama de clases sin implementar la gestión de interacciones de las listas

La solución adoptada en el capítulo, asigna las responsabilidades de gestionar las interacciones de los conjuntos a las clases contenedoras de los objetos. Así, la clase `ListaEsferas` es la encargada de invocar el rebote de cada una de ellas con una pared dada mediante:

```
void ListaEsferas::Rebote(Pared p)
{
	for(int i=0;i<numero;i++)
		Interaccion::Rebote(*(lista[i]),p);
}
```

Esto en el diagrama se traduce en una dependencia de la clase `ListaEsferas` a la clase `Pared`. Si se representan en el diagrama todas las dependencias que aparecen en funciones similares de `ListaEsferas` y `ListaDisparos` nos encontramos con el diagrama de la figura:

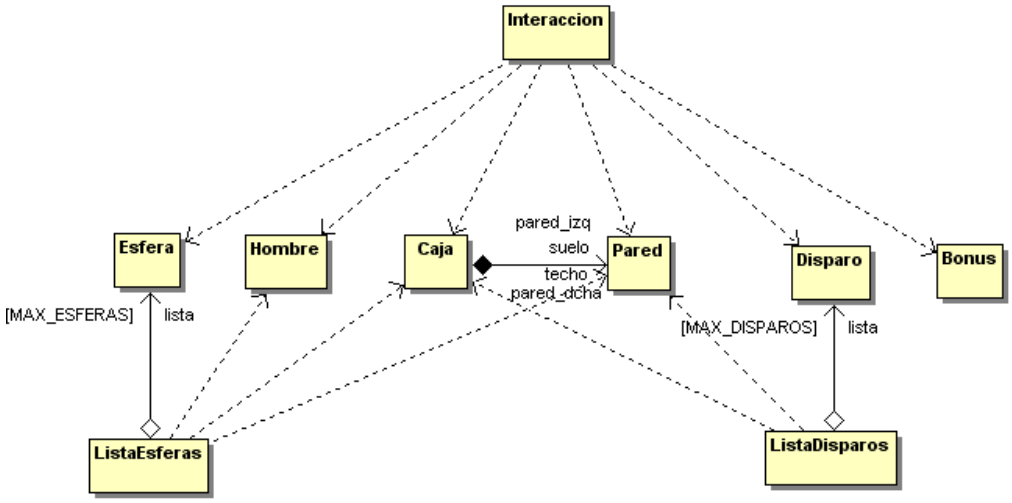


Figura 6-7. Dependencias generadas en la solución adoptada

No obstante, se puede conseguir un diagrama más limpio si utilizamos de nuevo el patrón **Indirección**, y nos llevamos toda la funcionalidad que implica las interacciones de listas de objetos a una nueva clase que podemos llamar `InteraccionListas`. Si la clase `Interaccion` calculaba interacciones entre pares de objetos, la clase `InteraccionListas` hace algo similar, pero gestionando las listas de objetos existentes en nuestro juego.

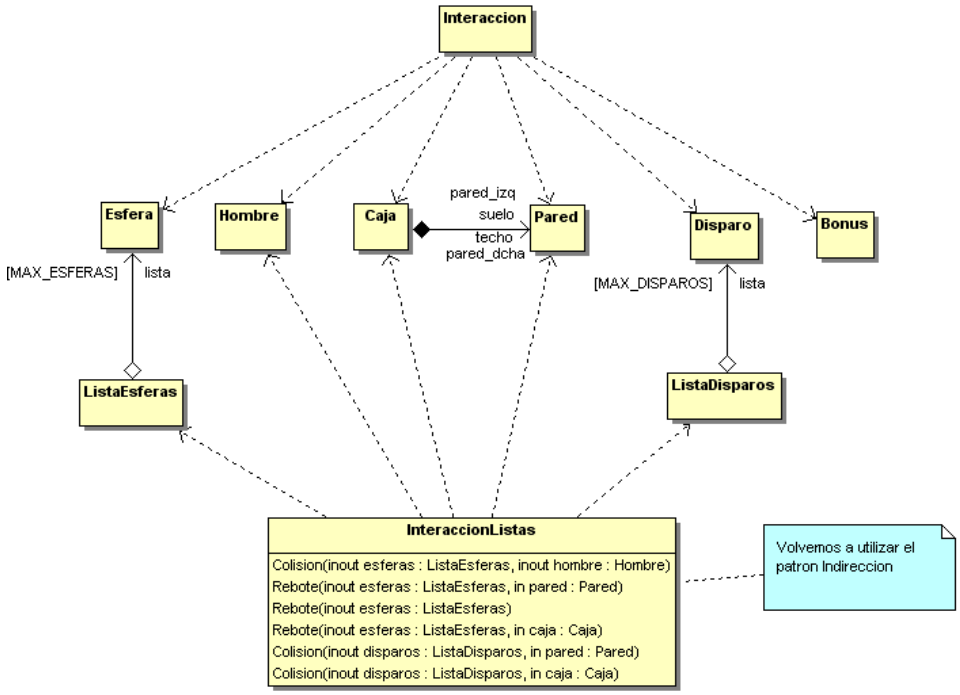


Figura 6-8. Un diseño más estructurado usando el patrón Indirección de nuevo

7. GENERALIZACIÓN Y ESPECIALIZACIÓN MEDIANTE HERENCIA

7.1. INTRODUCCIÓN

En capítulos anteriores hemos visto la mayor parte del desarrollo del juego del Pang. Terminar el juego es solo cuestión de desarrollar algunos puntos más de forma muy similar a como ya se ha explicado.

No obstante, hay una característica muy importante en la programación orientada a objetos en C++ (y en cualquier lenguaje OO) que facilita la reutilización de código, la abstracción de conceptos y permite un gran número de buenas prácticas en la Ingeniería del Software: Es la **Herencia**, con la que se puede implementar una relación de **Generalización**.

En este tema abordamos la Herencia en C++, y los mecanismos con los que se descubre: la relación de generalización en sentido ascendente y la especialización en sentido descendente.

En anteriores temas se han mostrado diagramas UML, pero no se ha pedido al lector que los implemente mediante alguna herramienta CASE. En este tema se realizará una ingeniería inversa sobre el código existente, y luego la ingeniería directa sobre los nuevos diagramas que incluyan algunos ejemplos de la relación de generalización.

7.2. HERENCIA EN C++

La herencia, entendida como una característica de la programación orientada a objetos y más concretamente del C++, permite definir una clase modificando una o más clases ya existentes. Estas modificaciones consisten habitualmente en añadir nuevos miembros (variables o funciones), a la clase que se está definiendo, aunque también se pueden redefinir variables o funciones miembro ya existentes.

La clase de la que se parte en este proceso recibe el nombre de **clase base**, y la nueva clase que se obtiene se denomina **clase derivada**. Ésta a su vez puede ser clase base en un nuevo proceso de derivación, iniciando de esta manera una jerarquía de clases. De ordinario las clases base suelen ser más generales que las clases derivadas. Esto es así

porque a las clases derivadas se les suelen ir añadiendo características, en definitiva variables y funciones que diferencian concretan y particularizan.

En algunos casos una clase no tiene otra utilidad que la de ser clase base para otras clases que se deriven de ella. A este tipo de clases base, de las que no se declara ningún objeto, se les denomina clases base abstractas (*Abstract Base Class*) y su función es la de agrupar miembros comunes de otras clases que se derivarán de ellas. Por ejemplo, se puede definir la clase vehículo para después derivar de ella coche, bicicleta, patinete, etc., pero todos los objetos que se declaren pertenecerán a alguna de estas últimas clases; no habrá vehículos que sean sólo vehículos.

Las características comunes de estas clases (como una variable que indique si está parado o en marcha, otra que indique su velocidad, la función de arrancar y la de frenar, etc.), pertenecerán a la clase base y las que sean particulares de alguna de ellas pertenecerán sólo a la clase derivada (por ejemplo el número de platos y piñones, que sólo tiene sentido para una bicicleta, o la función embragar que sólo se aplicará a los vehículos de motor con varias marchas).

Este mecanismo de herencia presenta múltiples ventajas evidentes a primera vista, como la posibilidad de reutilizar código sin tener que escribirlo de nuevo. Esto es posible porque todas las clases derivadas pueden utilizar el código de la clase base sin tener que volver a definirlo en cada una de ellas.

Veamos un sencillo ejemplo de herencia, viendo primero el diagrama UML, en el que se representa la herencia por una flecha vacía. Este diagrama se puede ver abriendo en Rational Rose el fichero ***Poligonos.mdl***.

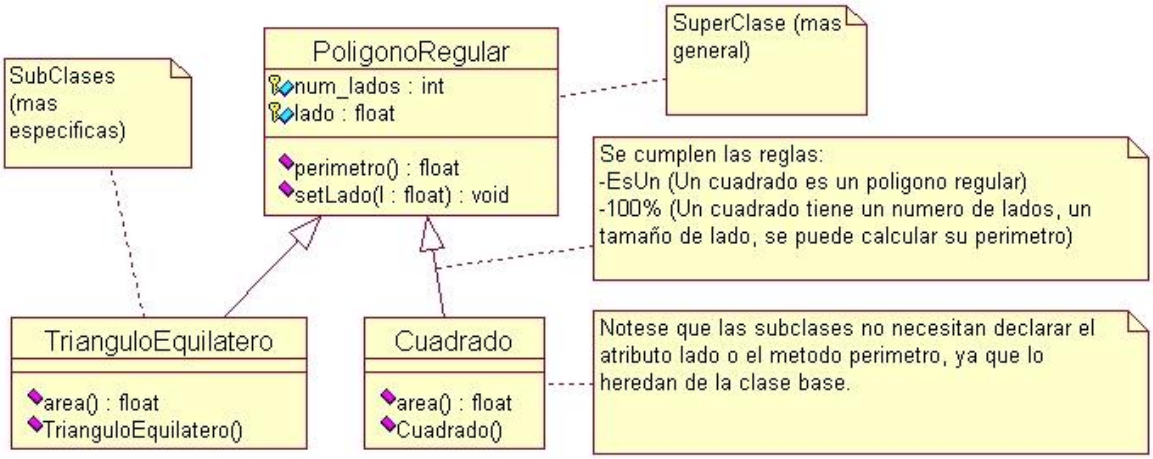


Figura 7-1. Herencia en un diagrama UML

Las subclases `TrianguloEquilatero` y `Cuadrado` heredan de la clase `PoligonoRegular`, que es más general que ellas. Además, la clase `PoligonoRegular` será abstracta, ya que no existen polígonos regulares genéricos, sino instancias de las subclases: triángulos, cuadrados, etc. La clase `PoligonoRegular`, puede calcular el perímetro de cualquier polígono regular, ya que conoce su número de lados y la longitud de su lado.

Nótese que los atributos `lado` y `num_lados` de la clase `PoligonoRegular` son protegidos, y por tanto se tiene acceso desde la propia clase `PoligonoRegular` y sus subclases, pero no desde fuera. De esta forma, como no se añade un método que sea

cambiar el numero de lados, no se puede cambiar el numero de lados de un `Cuadrado` por ejemplo, lo que sería un error lógico. Esto es encapsulación.

En los constructores respectivos de las subclases se inicializa el número de lados al número correspondiente a la figura de que se trate, 3 para el triángulo, 4 para un cuadrado, etc.

Mientras que las subclases pueden utilizar el método `perimetro()` sin necesidad de implementarlo, ya que lo heredan de la clase `PoligonoRegular`, el método `area()` es específico para cada subclase, y por tanto tienen que implementarlo ellas mismas.

EJERCICIO: Abrir el proyecto de Visual C++ en la carpeta *Poligonos*, compilarlo y ejecutarlo para ver el resultado. Explorar el código. El código de este proyecto ha sido generado automáticamente desde Rational Rose, a excepción del contenido de las funciones y el cuerpo del `main`.

IMPORTANTE: Nótese que la misma funcionalidad se podría haber conseguido con una sola clase `PoligonoRegular` que implementara una función `area()` con el siguiente aspecto. No obstante, esto no es correcto desde el punto de vista de la POO

```
float PoligonoRegular::area()
{
	if(num_lados==3)
	{
		float altura=lado*(float)sqrt(3)/2.0f;
		return lado*altura/2.0f;
	}
	if(num_lados==4)
		return lado*lado;
}
```

7.3. INGENIERÍA INVERSA

Vamos a realizar la ingeniería inversa del proyecto construido en Visual C++. La ingeniería inversa consiste en realizar los diagramas de clases y de objetos correspondientes al código ya existente. Las herramientas CASE a menudo disponen de mecanismos automatizados para realizar esta tarea. En este capítulo realizaremos dicha ingeniería con la herramienta *Rational Rose*, pero se puede hacer de forma muy similar con otras herramientas, como BoUML.

Se inicia el Rational Rose con un proyecto vacío, y se procede a *Menu->Tools->Visual C++->Update Model from Code*, tal como se muestra en la figura:

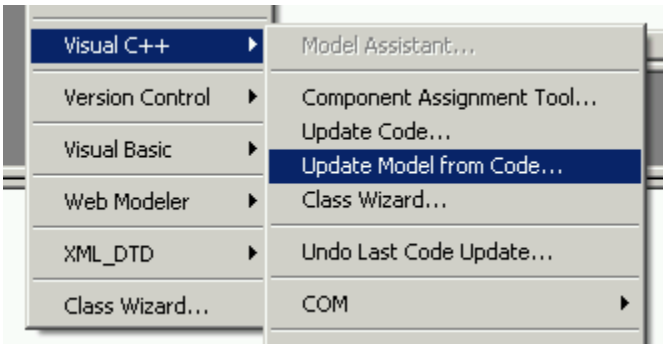


Figura 7-2. Pasos para realizar ingeniería inversa desde código existente

A continuación seleccionamos *Add a C++ Component->Select a Visual C++ Project->Existing* y navegamos hasta nuestro proyecto (se recomienda hacerlo sobre una copia de la carpeta del proyecto) y lo seleccionamos. El cuadro de dialogo debería aparecer como en la figura:

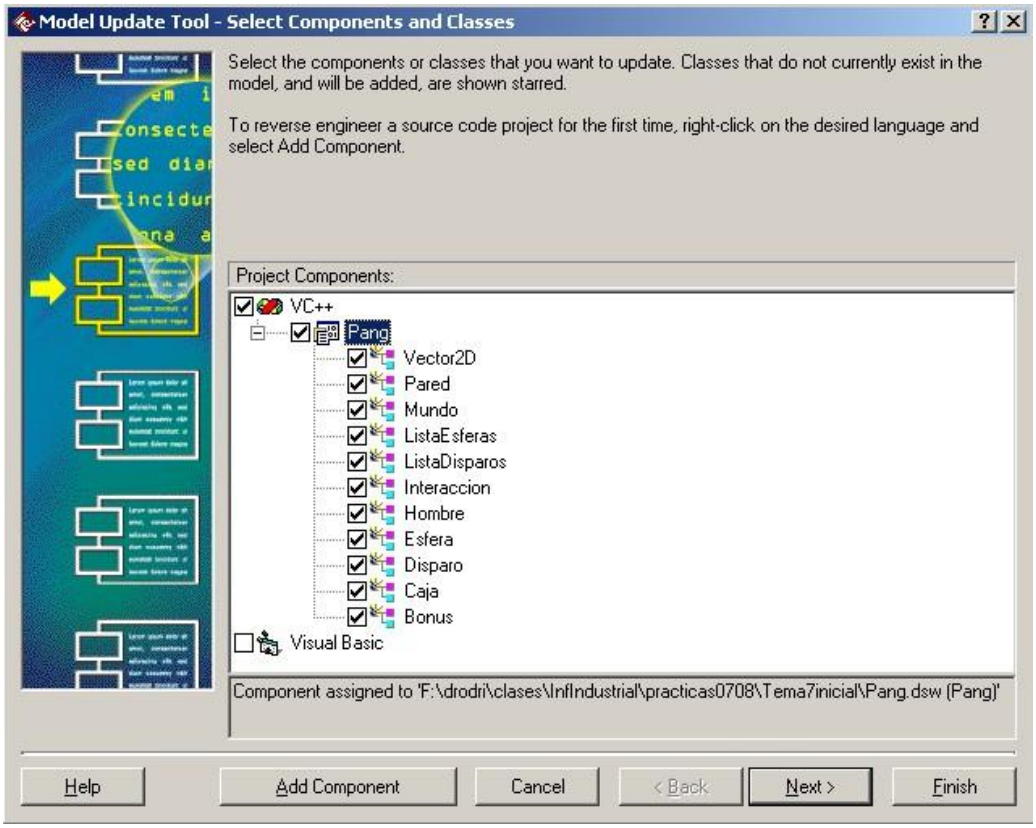


Figura 7-3. Selección de componentes a realizar ingeniería inversa

Finalmente pulsamos *Finish.*

Rational Rose ha creado una carpeta dentro del proyecto denominada *Visual C++ Reverse Engineered* en la que se puede navegar por las clases, atributos, métodos, asociaciones, etc. También ha creado un diagrama de clases básico en el que únicamente muestra una lista de las clases encontradas.

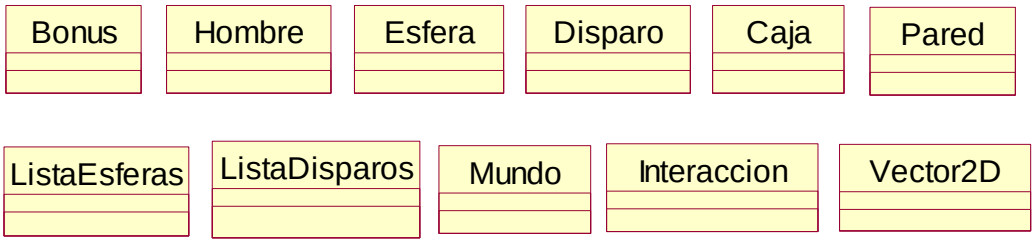


Figura 7-4. Diagrama de clases básico de la aplicación

Si echamos un vistazo al código del proyecto, veremos que *Rational Rose* ha introducido una serie de comentarios que le sirven de etiquetas de identificación de los distintos elementos

```
//##ModelId=49412D8B003E
class Hombre
{
      friend class Interaccion;
public:
      //##ModelId=49412D8B003F
      Vector2D GetPos();
      //##ModelId=49412D8B0040
      float GetAltura();
      //##ModelId=49412D8B004E
      void SetVel(float vx, float vy);
      //##ModelId=49412D8B0051
      void Mueve(float t);
      //##ModelId=49412D8B0053
      void Dibuja();
      //##ModelId=49412D8B005D
      Hombre();
      //##ModelId=49412D8B005E
      virtual ~Hombre();
…
```

7.4. GENERALIZACIÓN

La generalización consiste en crear una superclase más general dada una o varias clases existentes. Una forma común de descubrir la herencia es mediante la detección de miembros (atributos o funciones) repetidos en distintas clases. Esto suele indicar que dichas clases comparten algo en común, o lo que es lo mismo que existe una superclase común a todas ellas, en la que deberían residir esos atributos repetidos, extraídos mediante un proceso de factorización.

Procedemos a realizar un diagrama de clases que contenga más detallado en cuanto a algunos aspectos de nuestra aplicación. Para ello creamos un nuevo diagrama pinchando con el botón derecho sobre *Logical View->Pang->New->Class Diagram,* y lo llamamos con el nombre que queramos, por ejemplo DCD (diagrama de clases de diseño).

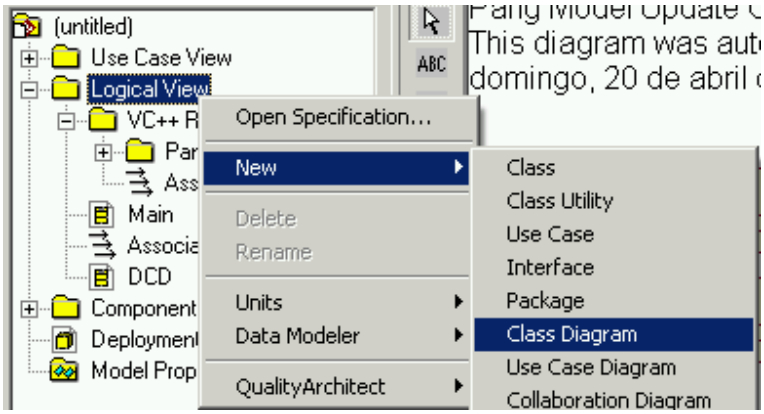


Figura 7-5. Nuevo diagrama de clases en Rational Rose

A continuación arrastramos nuestras clases `Disparo`, `Hombre`, `Esfera`, `Bonus` y `Vector2D` del árbol de navegación de la izquierda hasta el nuevo diagrama. Después de ajustar el aspecto del diagrama podría quedar con un aspecto similar al mostrado por la figura 7-6.

Nótese como *Rational Rose* ha interpretado que entre las distintas clases y la clase `Vector2D` existe una relación de agregación, lo que desde nuestro punto de vista no es totalmente adecuado. De hecho, otras herramientas como el *BoUML* detectan dicha relación como una composición, lo que parece más acertado. Otra cosa es que tal y como se explico en el capítulo 4, la clase `Vector2D` se puede entender perfectamente como un tipo de dato y representar como tal en el diagrama, para evitar un ruido visual innecesario.

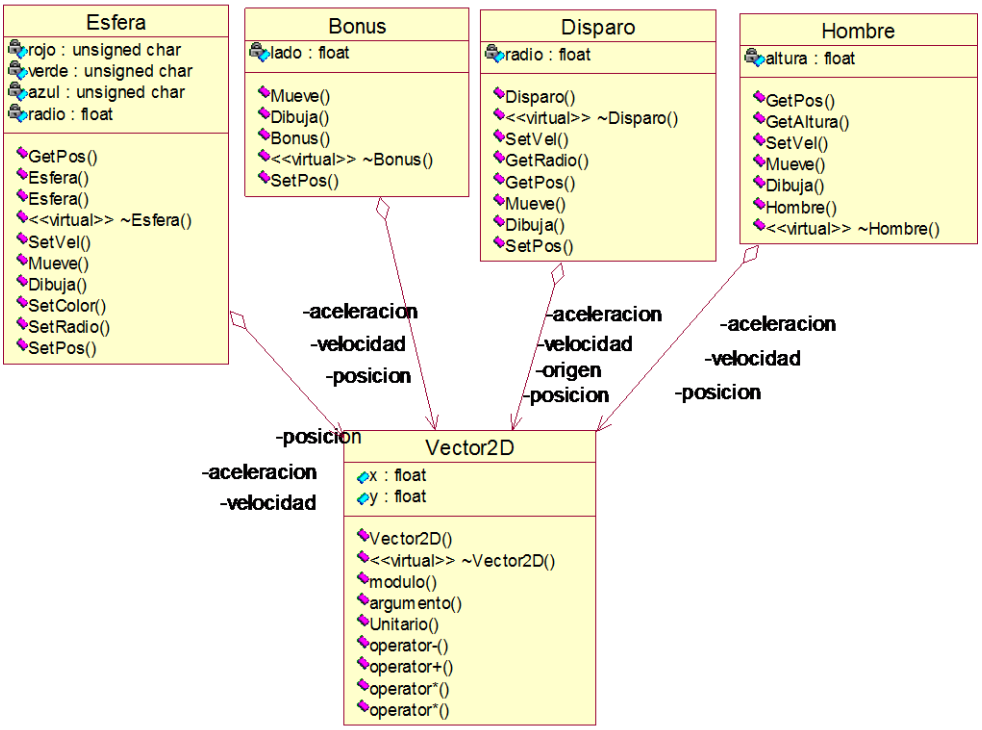


Figura 7-6. Diagrama de clases de los objetos móviles

En cualquier caso, lo relevante en este diagrama es darse cuenta de que existen muchas cosas comunes o repetidas. Todas las clases tienen tres atributos denominados `posicion`, `velocidad` y `aceleracion` de tipo `Vector2D`. Incluso tienen un método denominado `Mueve` que tiene una implementación idéntica en dichas clases:

```
void CLASE::Mueve(float t) //CLASE = Bonus, Esfera, Disparo, Hombre
{
      posicion=posicion+velocidad*t+aceleracion*(0.5f*t*t);
      velocidad=velocidad+aceleracion*t;
}
```

Se podría decir que existe un concepto, una clase que representa todo lo que es común entre las clases anteriores. La pregunta sería ¿Que tienen en común? ¿Cómo podría llamarse esa clase? Todos los elementos citados son cuerpos físicos que siguen un movimiento acelerado, a diferencia de otros cuerpos como las paredes, se mueven. Podríamos denominar a dicha clase `ObjetoMovil`. Vamos a representar esa clase en el diagrama.

Figura 7-7. Barra de herramientas de edición de diagramas

Primero añadimos la clase `ObjetoMovil` y a continuación trazamos las relaciones de **Generalización** desde la subclase hasta la superclase. Después añadimos a la clase base tres nuevos atributos (posición, velocidad, aceleración), pinchando sobre ella con el botón derecho, y añadiendo dichos atributos con notación UML *(posicion: Vector2D)*. De la misma forma añadimos un nuevo método a esta clase base, denominado *Mueve(t: float): void*

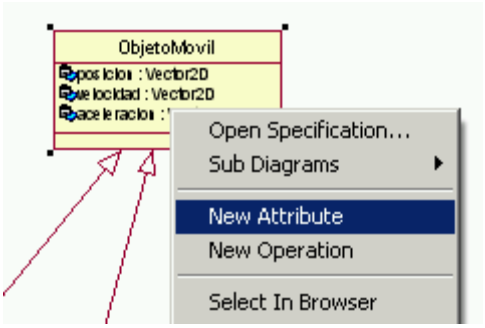


Figura 7-8. Adición de nuevos atributos a una clase

Hemos posicionado estos atributos en la clase base para que sean automáticamente heredados por las clases derivadas. Procedemos por lo tanto a **eliminar los atributos posición, velocidad y aceleración y el método Mueve()** de dichas clases derivadas:

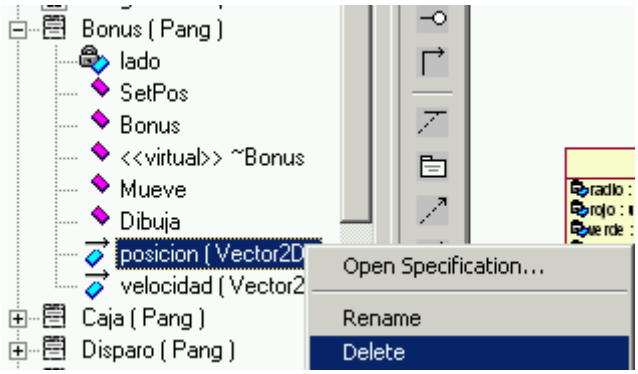


Figura 7-9. Eliminación de atributos

Los atributos posición, velocidad y aceleración, por defecto tienen visibilidad "privada", mostrada por *Rational Rose* por un candado (en UML por '-'). Si queremos que las clases que deriven de `ObjetoMovil` puedan acceder a dichos atributos tenemos que convertirlos en protegidos "`protected`", lo que se puede hacer pinchando sobre el icono que indica el nivel de acceso y seleccionando entre las alternativas el que se desee:

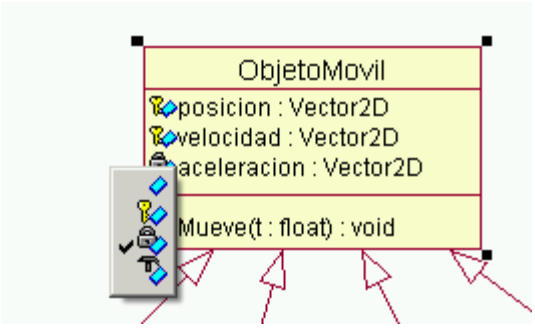


Figura 7-10. Cambio de visibilidad de privada a protegida

El diagrama final podría ser como se muestra en la figura 7-11, donde ya se ha omitido el dibujo de la clase `Vector2D`. Se podría decir que en este proceso hemos **factorizado los elementos comunes (posición, velocidad, aceleración y Mueve()) en una clase base más general (GENERALIZACION) que cada una de ellas**. Obsérvese como se cumplen las reglas "EsUn" y "100%" anteriormente descritas.

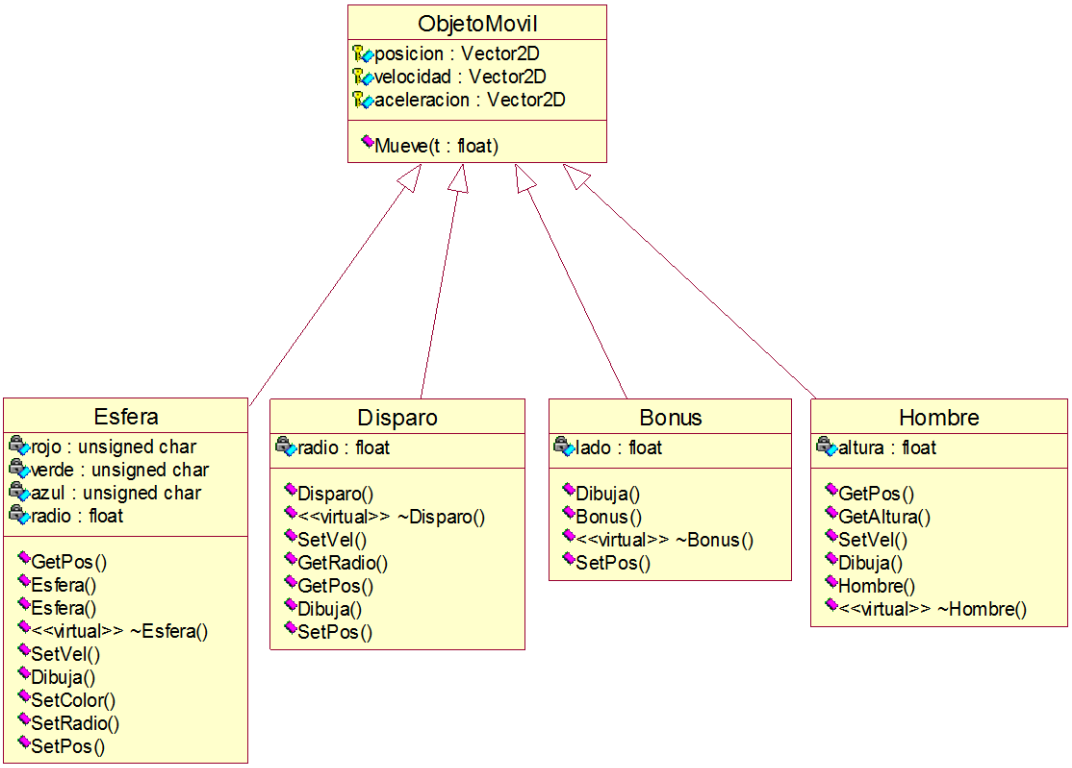


Figura 7-11. Diagrama de clases incluyendo la clase ObjetoMovil

7.5. INGENIERÍA DIRECTA

La ingeniería directa consiste en implementar el producto (el código) correspondiente a un diseño. Aunque las herramientas CASE también tienen a menudo automatizado el proceso de creación o actualización de código existente en función del diseño realizado en los diagramas de clases, este código suele contener a menudo ciertos errores y puede no representar fielmente lo deseado en los diagramas. Se va a proceder por lo tanto a realizar la labor de implementación directamente en la herramienta de desarrollo, lo que se deja como ejercicio al lector.

EJERCICIO: Implementar los cambios representados en el diagrama de clases anterior, siguiendo los siguientes pasos:

1. Para la clase base `ObjetoMovil`
   a. Crear una nueva clase denominada `ObjetoMovil`. **No olvidar** que debe ser guardada al ser creada en las subcarpetas "dominio".
   b. Añadir los atributos de posición, velocidad y aceleración a esta clase, de tipo `Vector2D`, con visibilidad `protected`
   c. Añadir e implementar el método `Mueve(float t)` a esta clase.

2. Para cada una de las subclases `Disparo, Bonus, Hombre, Esfera`:

a. Eliminar sus atributos de `posicion, velocidad y aceleracion` así como el método `Mueve(float t)`

b. Declararla como subclase de `ObjetoMovil`, lo que se realiza de la siguiente forma:

```
#include "ObjetoMovil.h"

class Hombre :public ObjetoMovil
{
public:
…
```

c. Compilar y eliminar posibles errores. Ejecutar para comprobar que sigue funcionando.

IMPORTANTE: Es importante resaltar que no se ha modificado para nada el código funcional de la aplicación, que sigue mostrando y ejecutándose exactamente igual. Lo que hemos realizado es un cambio estructural en el código, en el que ahora las ecuaciones de movimiento de todos los objetos móviles están implementadas en un solo lugar. Así, si es necesario cambiar las ecuaciones sólo es necesario hacerlo en un único sitio. Como ejemplo, impleméntese la función `ObjetoMovil::Mueve(float t)` de la siguiente forma, comprobando como todos los objetos han cambiado su movimiento

```
Vector2D ruido(   0.1f*(0.5f-rand()/((float)RAND_MAX))      ,0);
posicion=posicion+velocidad*t+aceleracion*0.5*t*t+ruido;
velocidad=velocidad+aceleracion*t+ruido;
```

7.6. ESPECIALIZACIÓN

Otra forma muy común de descubrir o generar herencia entre clases es la especialización, que consiste en dada una clase, generar una subclase de ésta que herede todo pero que implemente alguna funcionalidad extra que conceptualmente no debe ser añadida en la clase base.

Por ejemplo: Supóngase que queremos programar una `Esfera` que varíe su radio continuamente, primero creciendo y luego decreciendo. Asimismo, el color de la esfera va variando con su radio.

Esto se podría implementar de forma similar a como se ha expuesto anteriormente: se podría añadir un atributo a la clase `Esfera` de tipo `bool` denominado `pulsante`, e implementar la función `Mueve(float t)` de la clase `Esfera` con el siguiente aspecto:

```
#include "ObjetoMovil.h"
class Esfera  : public ObjetoMovil
{
public:
      …
      bool pulsante;//Añadimos esta variable para diferenciar
      float pulso;  //la velocidad de pulso
      void Mueve(float t); //añadimos este método que habiamos eliminado
};
```

Nótese que tenemos que implementar el método `Mueve(float t)`, porque no es suficiente el código existente en la clase base `ObjetoMovil`, ya que no deseamos hacer que los otros objetos presenten este comportamiento.

```
void Esfera::mover(float t)
{
      posicion=posicion+velocidad*t+aceleracion*0.5*t*t;
      velocidad=velocidad+aceleracion*t;

      if(pulsante) //PEOR SOLUCION
      {
            //código para cambiar el radio y el color
            //radio+=pulso*t;
      }
}
```

No obstante, esta forma no es correcta en POO. Si se quiere añadir un comportamiento distinto a uno o varios objetos de una clase, pero manteniendo el comportamiento ya existente para el resto, nos encontramos ante una herencia. En este caso, existe una subclase que podemos denominar `EsferaPulsante`, que hereda de la superclase `Esfera`. Modificamos nuestro diagrama para tener en cuenta esta nueva clase. A la nueva clase debemos añadirle los atributos pulso (la velocidad a la que pulsará la esfera), el radio máximo y el radio mínimo entre los que pulsará, y el método *Mueve(t: float)*. Asimismo, para que la clase `EsferaPulsante` pueda acceder a los atributos de `Esfera`, los convertimos en `protected`. Nótese que la `EsferaPulsante` cumple las reglas de "EsUn" y del "100%". Esta especialización se muestra en el diagrama siguiente.

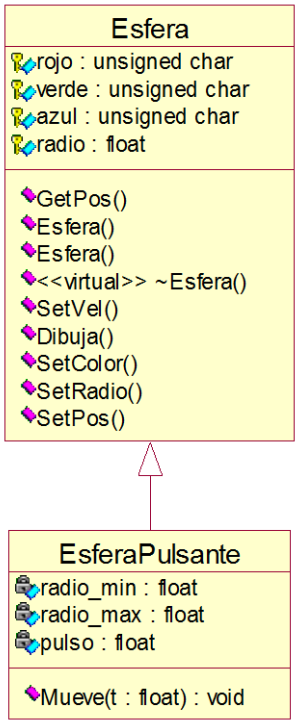


Figura 7-12. Especialización: la clase EsferaPulsante

EJERCICIO: Implementar el diagrama anterior, siguiendo los siguientes pasos:

1. Crear la clase `EsferaPulsante`, que hereda de la clase `Esfera`. Para ello utilizar el wizard *NewClass de VisualStudio.* **Recordar** que debe ser guardada al crearla en las subcarpetas "dominio".

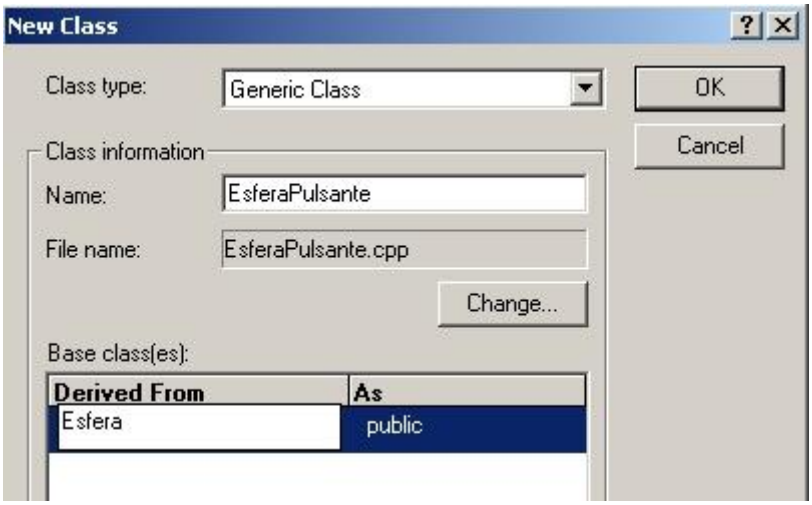


Figura 7-13. Especificar clases base en el ayudante de Visual C++

2. Añadir los atributos `pulso, radio_max y radio_min` (inicializarlos por defecto a 0.5, 0.5 y 2 por ejemplo). Inicializar también la aceleración por defecto a cero para las esferas pulsantes y las coordenadas de la posición inicial a (0, 5)
3. Cambiar el acceso de los atributos de `Esfera` de `private` a `protected`.
4. Añadir el método `Mueve(float t)` a la clase `EsferaPulsante`. IMPORTANTE: La sintaxis de la declaración debe coincidir exactamente con la de `ObjetoMovil` (incluido mayúsculas y minúsculas).

```
void EsferaPulsante::Mueve(float t)
{
	posicion=posicion+velocidad*t+aceleracion*(0.5f*t*t);
	velocidad=velocidad+aceleracion*t;

	if(radio>radio_max)
		pulso=-pulso;

	if(radio<radio_min)
		pulso=-pulso;

	radio+=pulso*t;

	rojo=radio*255;
	verde=255-radio*100;
	azul=100+radio*50;
}
```

5. Declarar un objeto de la clase `EsferaPulsante` en la clase `Mundo`

```
#include "EsferaPulsante.h"
class Mundo
{
private:
	EsferaPulsante esfera_pulsante;
```

6. Realizar las llamadas correspondientes a los métodos `Dibuja()` y `Mueve()` de la clase `EsferaPulsante`.

```
void Mundo::Dibuja()
{
…
       esfera_pulsante.Dibuja();
…
}

void Mundo::Mueve()
{
…
       esfera_pulsante.Mueve(0.025f);
…
}
```

El resultado es el que se muestra en la figura:

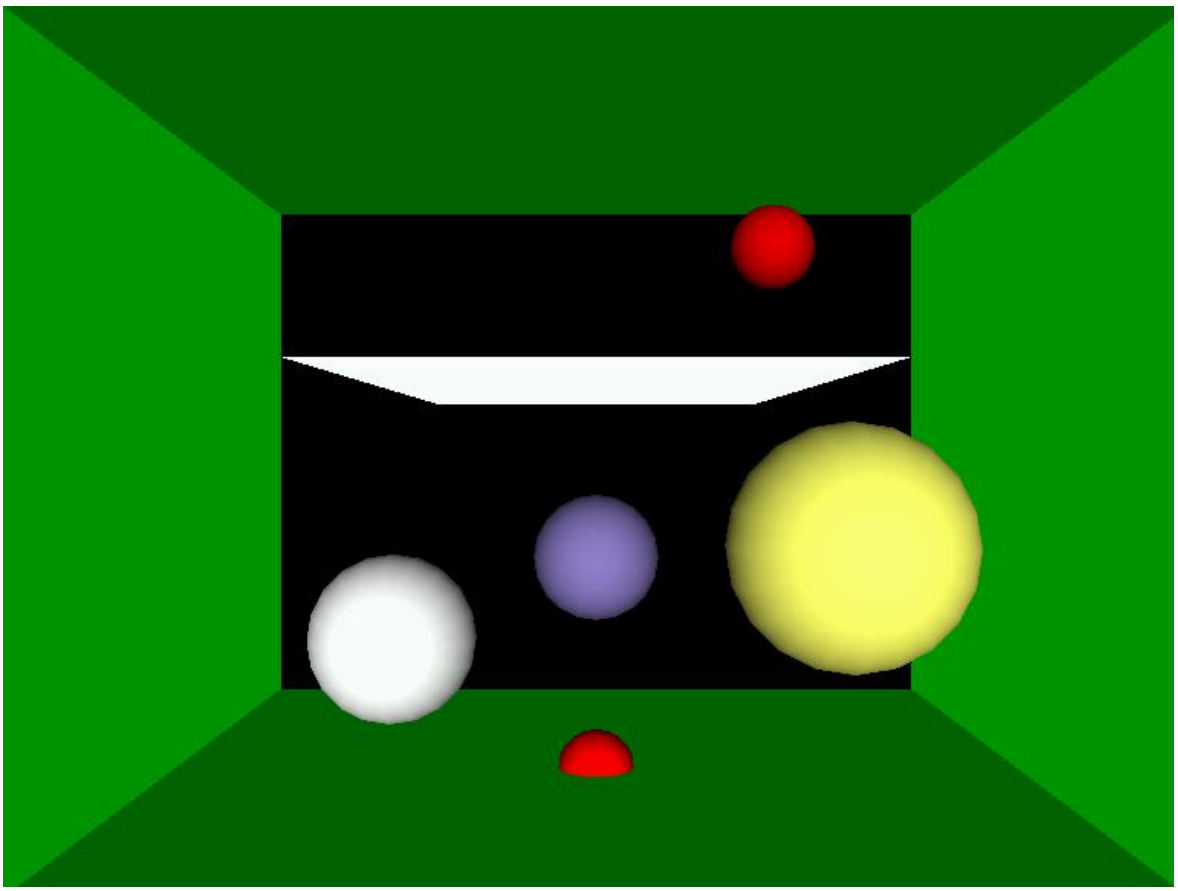


Figura 7-14. La esfera pulsante funcionando

Nótese como una clase puede ser subclase de otra y a la vez superclase de una tercera, componiendo una jerarquía de clases, tal y como se representa en la figura 7-15:

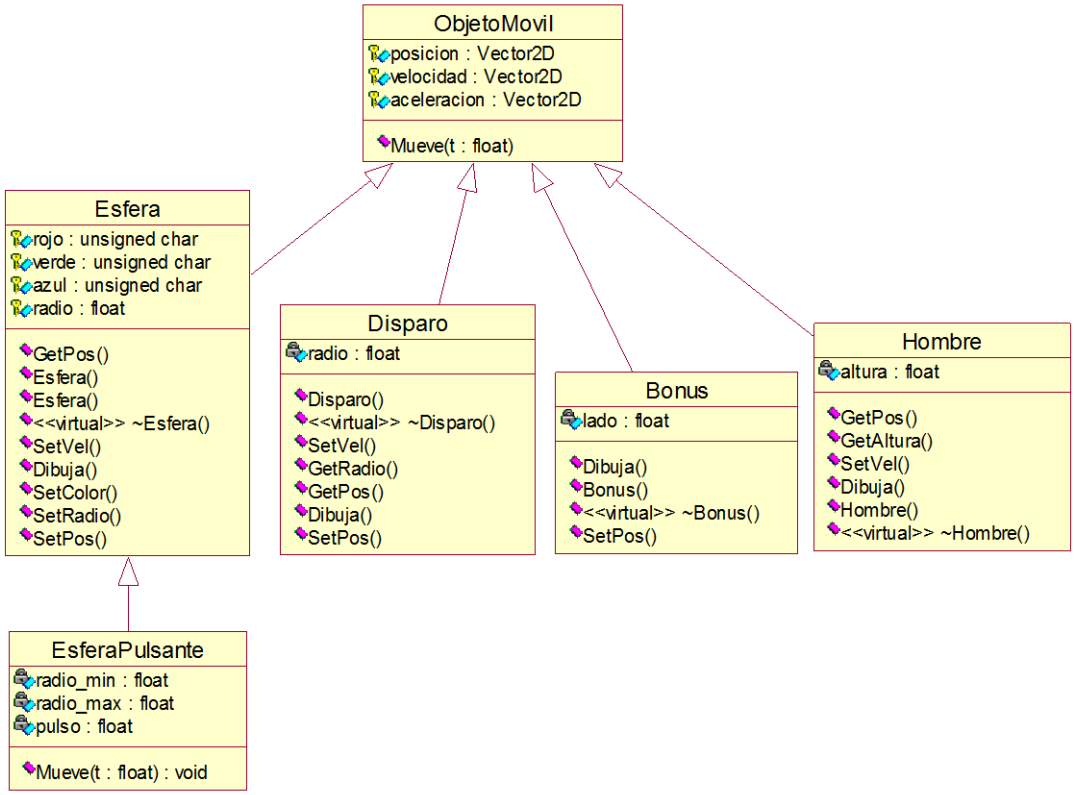


Figura 7-15. Jerarquía de clases

7.7. REUTILIZACIÓN DE CÓDIGO

Ya en el código anterior hemos reutilizado porciones de código importantes. Por ejemplo, hemos invocado la función `esfera_pulsante.Dibuja()`, sin necesidad de implementarla, porque ya la estamos heredando de la clase base.

Por otra parte nos hemos visto obligados a copiar y pegar código para que nuestra `EsferaPulsante` se moviera según las ecuaciones de `ObjetoMovil`. No obstante ahora tenemos duplicadas estas ecuaciones, lo que va en contra de la facilidad de mantenimiento de código. Esto se puede evitar sustituyendo las líneas siguientes de `EsferaPulsante::Mueve(float t)`

```
posicion=posicion+velocidad*t+aceleracion*0.5*t*t;
velocidad=velocidad+aceleracion*t;
```

por

```
Esfera::Mueve(t);
```

Hemos conseguido utilizar las ecuaciones de `ObjetoMovil` sin necesidad de copiarlas dentro de nuestra clase hija `EsferaPulsante`

Nótese como estamos invocando a una función que no está implementada en la clase Esfera, sino en la clase `ObjetoMovil`. No obstante, el compilador detecta esto automáticamente y utiliza la función adecuada que es `ObjetoMovil::Mueve(t)`. El

efecto hubiera sido el mismo, si en la función mover de `EsferaPulsante::Mueve(float t)` hubiéramos puesto la llamada.

```
ObjetoMovil::Mueve(t);
```

No obstante, se prefiere la primera forma, ya que es más lógica e intuitiva y facilita el mantenimiento de código, sobre todo si en el futuro se implementara la función `Esfera::Mueve(float t)`

Queda mucho más código por reutilizar, en concreto todo el relacionado con las interacciones. Dado que una `EsferaPulsante` es una `Esfera`, se pueden gestionar todos los rebotes mediante el código ya existente:

```
void Mundo::Mueve()
{
      esfera_pulsante.Mueve(0.025f);
      Interaccion::Rebote(esfera_pulsante,plataforma);
      Interaccion::Rebote(esfera_pulsante,caja);

      for(int i=0;i<esferas.GetNumero();i++)
            Interaccion::Rebote(esfera_pulsante,*esferas[i]);

…
```

7.8. EJERCICIOS PROPUESTOS

1. Implementar un disparo especial, que vaya más rápido que uno normal, tenga un calibre mayor, un color más fuerte y dos estelas en vez de una. Este disparo se declara e inicializa directamente en el mundo, es decir, no es necesario que lo dispare el hombre.
2. Llevar a la clase base `ObjetoMovil` aquellas funciones comunes de los objetos derivados: `GetPos()`, `SetPos()`, `SetVel()`, etc. cuyas implementaciones son idénticas o al menos muy similares. En el caso del `Disparo`, realizar la sobrecarga de `SetPos()`, para establecer adecuadamente la posición del origen del disparo. También se recomienda utilizar la sobrecarga de `GetPos()` y definir `void ObjetoMovil::SetPos(Vector2D pos)` y `void Disparo::SetPos(Vector2D pos)`

8. POLIMORFISMO. MÁQUINAS DE ESTADO.

8.1. INTRODUCCIÓN

En el capítulo anterior se finalizó con una esfera pulsante que tenía unas interacciones parciales con el resto del sistema, pero no estaba integrada del todo en el juego. En este capítulo veremos como la potencia del polimorfismo sirve para integrar fácilmente la esfera pulsante en la clase `ListaEsferas` y el disparo especial en `ListaDisparos`

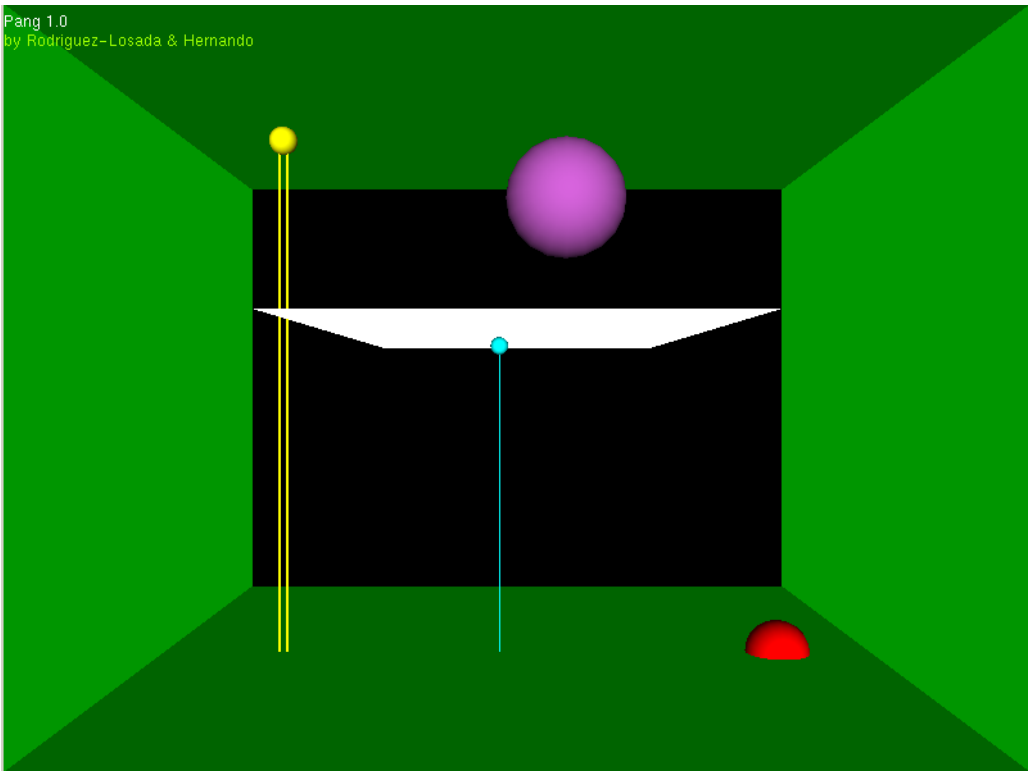


Figura 8-1. La esfera pulsante y el disparo especial

Se ha añadido por conveniencia la clase OpenGL que sirve para encapsular algunas funcionalidades avanzadas de OpenGL, como renderizar texto en la pantalla. Dicha clase

tiene un método estático que se puede utilizar **al final** de la función `Mundo::Dibuja()` de la siguiente forma:

```
OpenGL::Print("Pang 1.0",0,0,255,255,255);
OpenGL::Print("by Rodriguez-Losada & Hernando",0,20,155,255,0);
```

8.2. EL CONCEPTO DE POLIMORFISMO

Ha llegado el momento de introducir uno de los conceptos más importantes de la programación orientada a objetos: el polimorfismo.

Como se ha visto, C++ nos permite acceder a objetos de una clase derivada usando un puntero a la clase base. En eso consiste o se basa el polimorfismo. Hasta ahora sólo podemos acceder a datos y funciones que existen en la clase base, mientras que los datos y funciones propias de los objetos de clases derivadas son inaccesibles. Esto es debido a que el compilador decide en tiempo de compilación que métodos y atributos están disponibles en función del contenedor. Reiteramos:

> **Con el Polimorfismo se puede acceder a métodos de una clase derivada a través de un puntero a la clase base.**

Para ello hacemos uso de los denominados métodos **virtuales**. Un método virtual es un método de una clase base que puede ser redefinido en cada una de las clases derivadas de esta, y que una vez redefinido puede ser accedido por medio de un puntero o una referencia a la clase base, resolviéndose entonces la llamada en función del objeto referido en vez de en función de con qué se hace la referencia.

Si en una clase base definimos un método como virtual, y este método es superpuesto por una clase derivada, al invocarlo utilizando un puntero o una referencia de la clase base, ¡se ejecutará el método de la clase derivada!

Cuando una clase tiene algún método virtual –bien directamente, bien por herencia- se dice que dicha clase es **polimórfica**.

Para declarar un método como virtual se utiliza la siguiente sintaxis:

```
virtual <tipo> <nombre_función>(<lista_parámetros>) [{}];
```

Las principales características del mecanismo de virtualidad son las siguientes:

- Una vez que una función es declarada como virtual, lo seguirá siendo en las clases derivadas, es decir, la propiedad virtual se hereda.
- Si la función virtual no se define exactamente con el mismo tipo de valor de retorno y el mismo número y tipo de parámetros que en la clase base, no se considerará como la misma función, sino como una función superpuesta.
- El nivel de acceso no afecta a la virtualidad de las funciones. Es decir, una función virtual puede declararse como privada en las clases derivadas aun siendo pública en la clase base, pudiendo por tanto ejecutarse ese método privado desde fuera por medio de un puntero a la clase base.
- Una llamada a un método virtual se resuelve siempre en función del tipo del objeto referenciado.

- Una llamada a un método normal se resuelve siempre en función del tipo de la referencia o puntero utilizado.
- Una llamada a un método virtual especificando la clase, exige la utilización del operador de resolución de ámbito `::`, lo que suprime el mecanismo de virtualidad. Evidentemente este mecanismo solo podrá utilizarse para el método de la misma clase del contenedor o de clases base del mismo.
- Por su modo de funcionamiento interno (es decir, por el modo en que realmente trabaja el ordenador) las funciones virtuales son un poco menos eficientes que las funciones normales.

Vamos a ver algún ejemplo adicional que ayude a entender el polimorfismo y los métodos polimórficos. Para ello abrimos el proyecto ***Poligonos*** incluido entre los ficheros del capítulo.

Inicialmente tenemos el siguiente `main`:

```
#include <iostream.h>
#include "TrianguloEquilatero.h"
#include "Cuadrado.h"

int main(int argc, char* argv[])
{
      TrianguloEquilatero t;
      t.setLado(3.0f);
      float p=t.perimetro();
      float a=t.area();
      cout<<"Triangulo Perimetro:"<<p<<" Area:"<<a<<endl;

      Cuadrado c;
      c.setLado(4.5f);
      p=c.perimetro();
      a=c.area();
      cout<<"Cuadrado Perimetro:"<<p<<" Area:"<<a<<endl;

      PoligonoRegular *poligono;

      poligono=&c;
      cout<<"Perimetro:"<<poligono->perimetro();
      cout<<" Area:"<<poligono->area()<<endl;

      poligono=&t;
      cout<<"Perimetro:"<<poligono->perimetro();
      cout<<" Area:"<<poligono->area()<<endl;

      return 0;
}
```

Observamos que tanto el triángulo como el cuadrado son clases derivadas de polígono regular, por lo que es posible apuntarlas con un puntero del tipo de la clase base. Como tales además podemos preguntar por su perímetro, y por su área. Sin embargo, aunque el perímetro es calculado correctamente –dado que el código estaba definido en la clase base- observamos que el área del triangulo y del cuadrado no se calculan, puesto que el método que se ejecuta no es el suyo propio, sino el genérico definido en la clase base. Sin embargo, si hacemos que área sea una función polimórfica, entonces cada polígono utilizará el método específico en vez del genérico.

Para hacer que área sea una función polimórfica basta con anteponer virtual al prototipo de área:

```
class PoligonoRegular
{
public:
      void setLado(float l);
      float perimetro();
      virtual float area(){return 0.0f;} //VIRTUAL

protected:
      int num_lados;
      float lado;
};
```

Obsérvese entonces el efecto de haber convertido área en una función polimórfica. Ahora, aunque el puntero es de clase polígono, y parece que se ejecuta el método de la clase polígono, en verdad, se está ejecutando el método que corresponde en función del tipo de objeto que se apunta.

8.3. POLIMORFISMO EN EL PANG

Con la idea de reflejar las posibilidades que nos brinda el polimorfismo, vamos a intentar introducir las esferas pulsantes como parte del juego.

Observamos que para conseguir que la esfera pulsante colisionara con la caja y las paredes, hemos tenido que invocarlo expresamente para el objeto `esfera_pulsante`. Esto es porque se trata como un objeto especial. La intención es introducirlas dentro del conjunto `ListaEsferas`, para que así interaccione con las mismas cosas que interacciona una esfera normal.

Dado que son esferas, no tendría que haber mucho inconveniente en agregarlas a la lista de esferas que gestiona la evolución de las mismas. Modificamos por tanto la clase `Mundo`, eliminando el objeto `esfera_pulsante`, y todas las referencias al mismo y agregando en la función de inicialización:

```
void Mundo::Inicializa()
{
      …
      EsferaPulsante* e3=new EsferaPulsante;
      e3->SetColor(0,255,0);
      esferas.Agregar(e3);
}
```

Ejecutamos y vemos el resultado. Observamos entonces, que la esfera pulsante, ya no se comporta como tal. Esto es porque el método `Mueve()` que se está ejecutando no es el nuevo que hemos definido, sino el que tiene por ser una esfera normal. Es decir, que se ejecuta el método en función del puntero que apunta el objeto, en vez de en función que cual es el objeto apuntado.

Esto precisamente es lo que resuelve el polimorfismo (distintas formas de ejecución, para un mismo tipo de puntero) por medio de los métodos virtuales.

Definimos que el método `Mueve()` queremos que quede determinado por el objeto en vez de por el puntero utilizado. En la declaración de la clase `ObjetoMovil`, anteponemos al prototipo de `Mueve()` la palabra `virtual`, de forma que dicho método quedará sustituido por los que definamos posteriormente.

```
class ObjetoMovil
{
public:
      virtual void Mueve(float t);
      …
};
```

Compilamos y vemos el resultado. Nótese que el polimorfismo esta funcionando porque la clase `ListaEsferas` contiene unos punteros a `Esfera`.

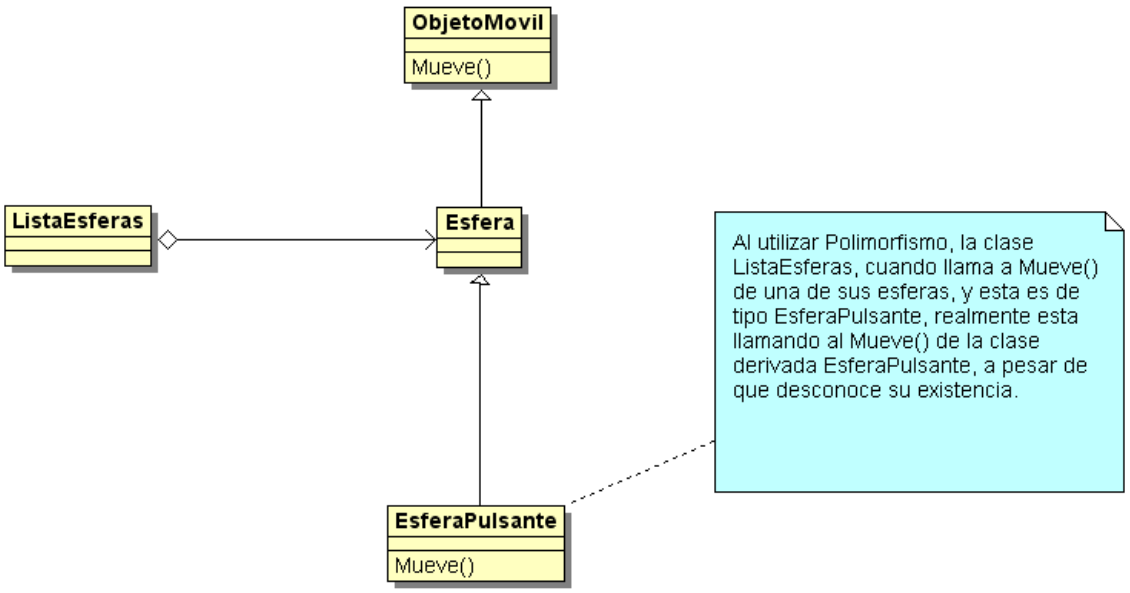


Figura 8-2. Polimorfismo en el Pang

Ya funciona. Gracias al polimorfismo se pueden lograr cosas muy sorprendentes y con muy poco esfuerzo. También es cierto que hay un cambio en el modo de diseñar las cosas y del planteamiento de como interaccionan los objetos entre sí. Por ejemplo, vamos a hacer ahora, que una esfera al ser impactada se divida en dos esferas más pequeñas, y que lo mismo suceda para la esfera pulsante, que al ser impactada se divida en dos esferas pulsantes. Lo primero que necesitamos es implementar el método de interacción correspondiente, que aun no lo habíamos desarrollado:

```
bool Interaccion::Colision(Disparo d, Esfera e)
{
      Pared aux; //Creamos una pared auxiliar
      Vector2D p1=d.posicion;
      Vector2D p2=d.origen;
      aux.SetPos(p1.x,p1.y,p2.x,p2.y); //Que coincida con el disparo.
      float dist=aux.Distancia(e.posicion); //para calcular su distancia
      if(dist<e.radio)
            return true;
      return false;
}
```

A continuación implementamos un método llamado `Dividir()` en la clase `Esfera` que lo que hace es dividir la esfera por la mitad, y crear un copia de dicha mitad, que devuelve como puntero para que pueda ser agregada a la lista de esferas. Además también se encarga de definir nuevas velocidades a ambas mitades. Si el radio de la esfera es suficientemente pequeño, entonces no se divide, sino que se devuelve inmediatamente un puntero nulo para indicarlo.

Vamos a crear un método `virtual` que cree copias de la misma esfera:

```
virtual Esfera* Dividir();
```

Y su implementación

```
Esfera* Esfera::Dividir()
{
	if(radio<1.0f)
		return 0; //no dividimos

	radio/=2.0f;//Dividimos el radio por 2
	//Creamos una esfera nueva, copiando la actual
	Esfera *aux=new Esfera(*this);
	//Les damos nuevas velocidades
	aux->SetVel(5,8);//a la nueva mitad
	SetVel(-5,8);//a la mitad original
	return aux;
}
```

Para probar este nuevo método, vamos a utilizar el disparo especial, que al no estar metido en el vector es más cómodo. Cuando el disparo impacte en una esfera, se inicializa su posición y velocidad de nuevo, para ver el efecto. Nuestra función `Mundo::Mueve()` quedará como sigue:

```
void Mundo::Mueve()
{
	for(int i=0;i<esferas.GetNumero();i++)
	{
		if(Interaccion::Colision(disparo_especial,*esferas[i]))
		{
			Esfera* e=esferas[i]->Dividir();
			if(e==0) //no division
				esferas.Eliminar(esferas[i]);
			else
				esferas.Agregar(e);
			disparo_especial.SetPos(0,0);
			disparo_especial.SetVel(0,10);
			break;
		}
	}
…
```

Nótese que también podríamos haber creado un objeto nuevo directamente, de la forma:

```
Esfera* aux=new Esfera(radio);
```

No obstante, la forma en la que lo hemos hecho, facilita posteriormente el uso del polimorfismo. Observamos que cuando se impacta a una esfera pulsante, no se divide en

dos esferas pulsantes, como cabria esperar, sino que la nueva esfera es normal. Para solucionar este aspecto, utilizaremos una vez más el polimorfismo:

Ejercicio: añadir el método de Dividir() en la clase EsferaPulsante, para que funcione polimórficamente y el impacto de un disparo en una EsferaPulsante (cuando su radio es suficientemente grande) se traduzca en dos esferas pulsantes, cuya velocidad de pulso se ve incrementada, pero el tamaño de oscilación no cambia. Es decir para destruir una esfera pulsante hay que esperar a que sea pequeña o de lo contrario, se duplicará, pero pulsará mas rápido, con lo que será más peligrosa.

8.4. MÁQUINA DE ESTADOS

Un elemento casi siempre presente en los programas interactivos –como es el caso del juego- es la máquina de estados que establece como se van logrando los distintos hitos del juego. Es decir, tenemos que controlar cuando comienza y termina el juego, si el usuario hace una pausa, si termina la partida o le matan las esferas. Para ello desarrollamos una clase siguiendo el patrón Controlador, que será la encargada de coordinar todo:

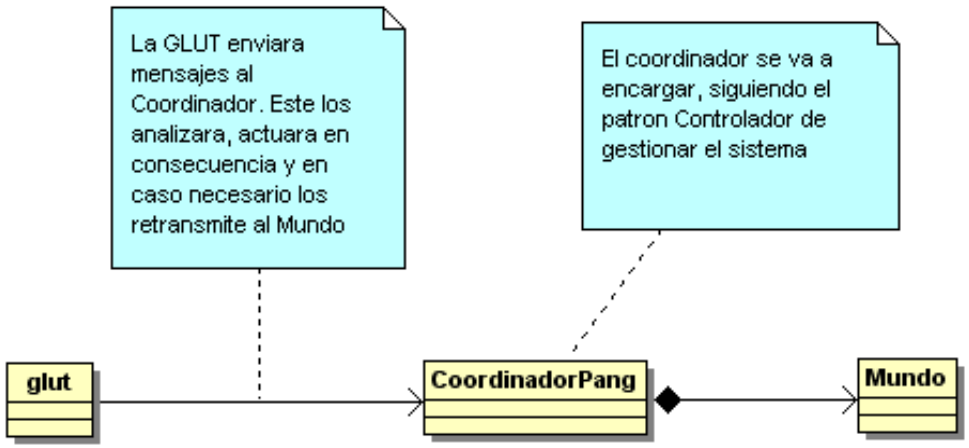


Figura 8-3. El coordinador del juego

A dicha clase la llamamos `CoordinadorPang`, y tendrá la siguiente cabecera:

```
#include "Mundo.h"

class CoordinadorPang
{
public:
	CoordinadorPang();
	virtual ~CoordinadorPang();

	void TeclaEspecial(unsigned char key);
	void Tecla(unsigned char key);
	void Mueve();
	void Dibuja();
```

```
protected:
      Mundo mundo;
};
```

Sustituimos por lo tanto la instancia a `Mundo` en el ***main.cpp***, para cambiarlo por el nuevo coordinador:

```
CoordinadorPang pang;

int main(int argc,char* argv[])
{
```

Y de momento dejamos las funciones vacías, excepto en la de dibujo que ponemos el mensaje de texto que antes teníamos en `Mundo::Dibuja()`

```
void CoordinadorPang::Dibuja()
{
      OpenGL::Print("Pang 1.0",0,0,255,255,255);
      OpenGL::Print("by Rodriguez-Losada & Hernando",0,15,155,255,0);
}
```

Vamos primero a realizar una máquina de estados (Figura 8-4) de tal forma que cuando el programa arranca, simplemente nos muestra un mensaje de información y nos pide que pulsemos una tecla. Cuando lo hagamos, ya se visualizará toda la escena.

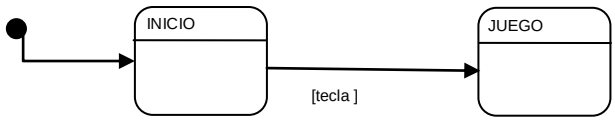


Figura 8-4. Máquina de estados

Para implementar esta máquina de estados, añadimos al coordinador una variable de tipo entero, cuyo valor representa el estado en el que nos encontramos

```
class CoordinadorPang
{
protected:
      enum Estado {INICIO, JUEGO};
      Estado estado;
```

El punto negro del diagrama representa el comienzo, que inmediatamente pasa al estado INICIO, por lo tanto iniciamos la variable de estado a este valor en el constructor:

```
CoordinadorPang::CoordinadorPang()
{
      estado=INICIO;
}
```

Lo que se representa en pantalla es diferente según el estado en el que nos encontremos, es decir:

```
void CoordinadorPang::Dibuja()
{
      if(estado==INICIO)
      {
            OpenGL::Print("Pang 1.0",0,0,255,255,255);
            OpenGL::Print("by Rguez-Losada & Hernando",0,15,155,255,0);
            OpenGL::Print("Pulse una tecla para empezar",0,30,5,255,255);
```

```
	}
	else if(estado==JUEGO)
		mundo.Dibuja();
}
```

Tenemos que implementar el hecho de que cuando se pulsa una tecla (si estamos en estado INICIO, se pasa a jugar (inicializando previamente la escena del mundo), y si estamos en estado JUEGO, entonces esa tecla tiene que ser procesada por el `Mundo`

```
void CoordinadorPang::Tecla(unsigned char key)
{
	if(estado==INICIO)
	{
		mundo.Inicializa();
		estado=JUEGO;
	}
	else if(estado==JUEGO)
	{
		mundo.Tecla(key);
	}
}
```

Lo mismo sucede con las teclas especiales y el movimiento. Si estamos en estado JUEGO, redireccionamos (coordinamos) los eventos hacia el objeto `mundo`

```
void CoordinadorPang::TeclaEspecial(unsigned char key)
{
	if(estado==JUEGO)
		mundo.TeclaEspecial(key);
}
void CoordinadorPang::Mueve()
{
	if(estado==JUEGO)
		mundo.Mueve();
}
```

Vamos a completar la máquina de estados con dos nuevos estados: el estado GAMEOVER en el que se entra si una de las esferas impacta (“mata”) al hombre, y el estado FIN, al que se llega si se consiguen destruir todas las esferas de la pantalla. En cualquiera de los dos estados, se congelará la imagen, se mostrará el mensaje correspondiente y pedirá pulsar la tecla “C” para continuar. Además, se ha añadido la posibilidad de elegir al INICIO que hacer: pulsar la tecla “E” para empezar o pulsar la tecla “S” para salir, terminando el programa.

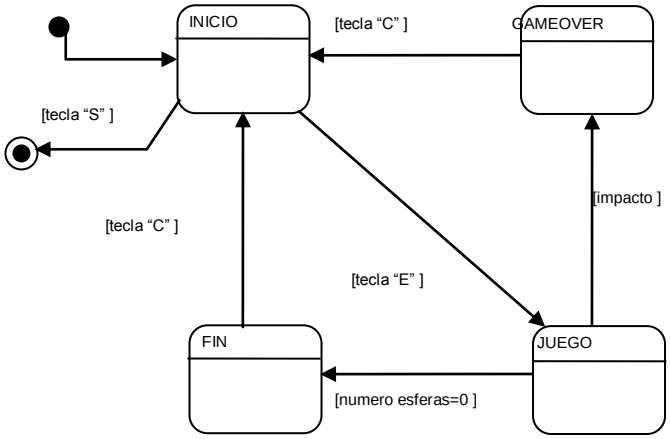


Figura 8-5. Máquina de estados añadiendo estados "GameOver" y "Fin"

La función que gestiona las teclas normales queda como sigue:

```
void CoordinadorPang::Tecla(unsigned char key)
{
	if(estado==INICIO)
	{
		if(key=='e')
		{
			mundo.Inicializa();
			estado=JUEGO;
		}
		if(key=='s')
			exit(0);
	}
	else if(estado==JUEGO)
	{
		mundo.Tecla(key);
	}
	else if(estado==GAMEOVER)
	{
		if(key=='c')
			estado=INICIO;
	}
	else if(estado==FIN)
	{
		if(key=='c')
			estado=INICIO;
	}
}
```

Nótese como se ha usado la estructura `if-else if` para tratar los estados. Se recomienda utilizar esta estructura en todas las funciones del coordinador. De esta forma, la función de dibujo quedaría:

```
void CoordinadorPang::Dibuja()
{
	if(estado==INICIO)
	{
		OpenGL::Print("Pang 1.0",0,0,255,255,255);
		OpenGL::Print("by Rguez-Losada & Hernando",0,15,155,255,0);
		OpenGL::Print("Pulse -E- tecla para empezar",0,30,5,255,255);
		OpenGL::Print("Pulse -S- tecla para salir",0,45,5,255,255);
	}
	else if(estado==JUEGO)
	{
		mundo.Dibuja();
	}
	else if(estado==GAMEOVER)
	{
		mundo.Dibuja();
		OpenGL::Print("GAMEOVER: Has perdido",0,0,255,255,255);
		OpenGL::Print("Pulsa -C- para continuar",0,15,155,255,0);
	}
	else if(estado==FIN)
	{
		mundo.Dibuja();
		OpenGL::Print("ENHORABUENA, Has triunfado!",0,0,255,255,255);
		OpenGL::Print("Pulsa -C- para continuar",0,15,155,255,0);
	}
}
```

Nos falta la parte en la que hay que analizar lo que pasa en el mundo durante el juego, para actuar en consecuencia. Saber si se han terminado las esferas es fácil, ya que conocemos su número. Sin embargo la situación de impacto de una esfera en el hombre no la podemos conocer fácilmente. De hecho lo que hacíamos hasta ahora es que cada vez que una esfera impactaba en el hombre, la hacíamos desaparecer. La forma más sencilla de hacerlo es añadir una variable booleana a la clase `Mundo` que indique esta situación, que inicializamos a `false` cuando se inicializa la pantalla, y que se cambia a `true` cuando hay un impacto:

```
void Mundo::Mueve()
{
	Esfera *aux=esferas.Colision(hombre);
	if(aux!=0)
		impacto=true;
```

Dado que lo lógico es que sea un atributo privado, será necesario implementar adicionalmente el método de interfaz `bool Mundo::Impacto()`. De esta forma, la función `Mueve()` del coordinador quedaría como sigue:

```
void CoordinadorPang::Mueve()
{
	if(estado==JUEGO)
```

```
	{
		mundo.Mueve();
		if(mundo.GetNumEsferas()==0)
		{
			estado=FIN;
		}
		if(mundo.Impacto())
		{
			estado=GAMEOVER;
		}
	}
}
```

Donde las funciones `Impacto()` y `GetNumEsferas()` de mundo deben de ser también implementadas. También es importante notar, que dado que es posible que la pantalla vuelva a ser inicializada desde una partida, es necesario borrar el posible contenido de la misma antes de añadir más objetos, así como de volver a establecer la posición inicial del hombre:

```
void Mundo::Inicializa()
{
	impacto=false;
	…
	hombre.SetPos(0,0);
	…
	esferas.DestruirContenido();
	disparos.DestruirContenido();
//aquí añadir cosas nuevas
```

Y con esto tenemos terminada nuestra máquina de estado básica.

Ejercicio: Implementar la siguiente máquina de estados, en la que la tecla "P" congela la imagen del juego, mostrando un mensaje de pausa, desde la que se puede reanudar el juego pulsando "C"

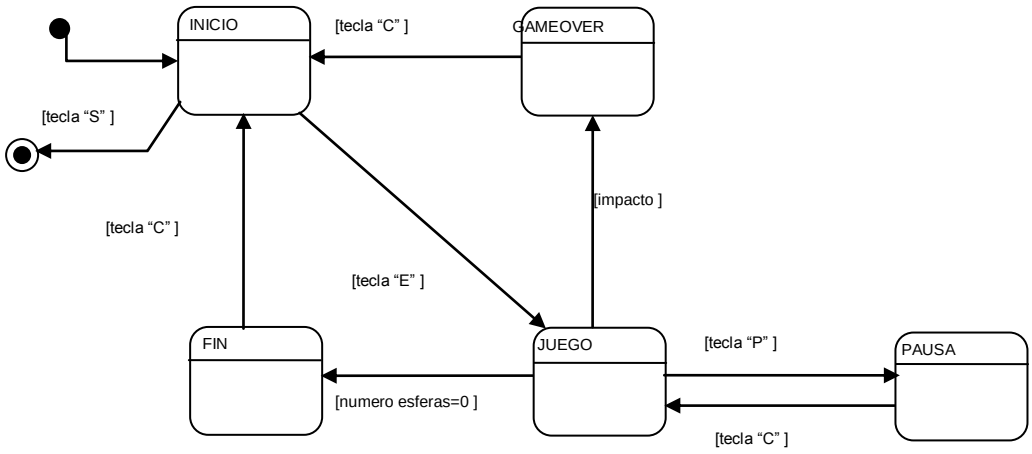


Figura 8-6. Máquina de estados con Pausa

Para terminar el capítulo, realizamos el paso de nivel, mediante una nueva función `CargarNivel()`, que añadimos a la clase `Mundo` y a la que nos llevamos gran parte del contenido de `Inicializa()`, así como una variable que indica el nivel actual.

```
void Mundo::Inicializa()
{
	impacto=false;

	x_ojo=0;
	y_ojo=7.5;
	z_ojo=30;

	bonus.SetPos(5.0f,5.0f);
	nivel=0;
	CargarNivel();
}
```

La definición del nivel puede establecer distintas esferas, cambiar la plataforma de color y posición, etc. Además hacemos que devuelva "`false`" si no ha sido capaz de cargar un nuevo nivel, indicando que hemos terminado el juego.

```
bool Mundo::CargarNivel()
{
	nivel++;
	hombre.SetPos(0,0);
	esferas.DestruirContenido();
	disparos.DestruirContenido();

	if(nivel==1)
	{

		plataforma.SetPos(-5.0f,9.0f,5.0f,9.0f);
		Esfera *e1=new Esfera(1.5f,2,4,5,15);
		e1->SetColor(0,0,255);
		esferas.Agregar(e1); //esfera
	}
	if(nivel==2)
	{
		plataforma.SetPos(-3.0f,6.0f,3.0f,6.0f);
		plataforma.SetColor(255,0,0);
		EsferaPulsante* e3=new EsferaPulsante;
		e3->SetPos(0,12);
		e3->SetVel(5,3);
		esferas.Agregar(e3);
	}
	if(nivel==3)
	{
		plataforma.SetPos(-10.0f,12.0f,4.0f,10.0f);
		plataforma.SetColor(255,0,255);
		for(int i=0;i<5;i++)
		{
			Esfera* aux=new Esfera(1.5,-5+i,12,i,5);
			aux->SetColor(i*40,0,255-i*40);
			esferas.Agregar(aux);
		}
	}
	if(nivel<=3)
		return true;
	return false;
}
```

Esta función puede ser invocada desde el Coordinador:

```
void CoordinadorPang::Mueve()
{
```

```
        if(estado==JUEGO)
        {
                mundo.Mueve();
                if(mundo.esferas.GetNumero()==0)
                {
                        if(!mundo.CargarNivel())
                                estado=FIN;
                }
                if(mundo.impacto)
                {
                        estado=GAMEOVER;
                }
        }
}
```

8.5. EJERCICIOS PROPUESTOS

1. Integrar el `DisparoEspecial`en `ListaDisparos` mediante el polimorfismo, utilizando una tecla alternativa para dispararlo, por ejemplo la “Z”
2. Implementar las colisiones entre los disparos y las esferas. Nótese que para eso se necesita:
   a. Implementar el acceso individual a las componentes de `ListaDisparos` mediante `GetNumero()` y el operador `[]`, de forma similar a como se hacía con la lista de esferas.
   b. Implementar las funciones que permiten eliminar un disparo de la lista de disparos
   c. Realizar un doble bucle `for()` que recorra la lista de esferas y la lista de disparos (se recomienda recorrer la lista de disparos para cada esfera). Cuando se realice un impacto, se borra el disparo correspondiente y se divide la esfera si es necesario. También es importante notar que si dentro del bucle `for()` que recorre las esferas se elimina una de ellas, los índices se verán afectados. La solución más conveniente es recorrer este bucle en orden inverso, es decir empezando por la última esfera del vector y finalizando en la primera.

Se puede considerar el juego como prácticamente terminado en este momento.

9. DISEÑO CON INTERFACES Y POLIMORFISMO

9.1. INTRODUCCIÓN

En capítulos anteriores se aprendió a descubrir las relaciones de Generalización e implementarlas mediante la Herencia de clases en C++, tanto en un proceso de factorización (hacia arriba) como en un proceso de especialización (hacia abajo). Más tarde se vio la potencia del polimorfismo, que puede decidir en tiempo de ejecución que método invoca en función del tipo de objeto al que apunta un puntero de una clase base que contiene la dirección de un objeto de una clase derivada.

Como ya se comentó en el capítulo anterior, el polimorfismo de hecho es una característica tan interesante que merece la pena ser tenido en cuenta desde el primer momento del diseño. Un buen uso de las relaciones de Generalización y del Polimorfismo genera una arquitectura y un código modular altamente reutilizable, robusto frente a cambios y ampliaciones futuras (Variaciones Protegidas) y con un mantenimiento más sencillo. De hecho el polimorfismo es la base para un buen número de Patrones de Diseño GOF.

En este capítulo se va a incidir en el concepto del polimorfismo, en el diseño de interfaces mediante clases abstractas que no representan objetos concretos sino que sirven como "conector" versátil al que se pueden conectar fácilmente una o varias clases concretas. También se describirá una solución al problema que aparece cuando el polimorfismo no es suficiente y se necesita conocer el tipo de cada objeto en tiempo de ejecución desde clases ajenas a la jerarquía de clases que soporta el polimorfismo. Por último, y como consecuencia, se presentará el uso de una Factoría para la creación de objetos.

9.2. CLASES ABSTRACTAS

Cuando una clase no puede tener instancias concretas de la misma, se denomina clase abstracta. En el ejemplo ya presentado en capítulos anteriores, la clase `PoligonoRegular` es una clase abstracta, ya que no existen objetos de la misma. Pueden existir triángulos, cuadrados, etc., pero no puede existir un objeto del tipo `PoligonoRegular` sin concretar su número de lados.

De hecho la prueba la tenemos en que hemos tenido que implementar su método `area()` y hemos convenido que devuelva cero ante la imposibilidad de devolver un valor con sentido. La forma de no tener que implementar un cuerpo concreto para un método de una clase abstracta es convertir ese método en **virtual puro**, lo que se consigue con la siguiente sintaxis:

```
virtual float area()=0;
```

Si una clase contiene métodos virtuales puros no es posible crear instancias de la misma, por ese motivo declarar objetos de la clase `PoligonoRegular` se convierte en un error de sintaxis. Sí que es posible sin embargo declarar punteros a objetos de esta clase, dado que se puede apuntar a objetos de clases derivadas que redefinan los métodos virtuales puros.

```
PoligonoRegular poligono; //Error de sintaxis, clase abstracta
PoligonoRegular* p; //correcto
p=new PoligonoRegular; //error de sintaxis, clase abstracta
Triangulo t;
p=&t; //p puede apuntar a objetos concretos
```

Cuando se hereda de una clase abstracta, **las clases concretas derivadas deben implementar los métodos virtuales puros**. Si alguno de los métodos virtuales puros quedase sin implementar, estaríamos ante otra clase abstracta derivada de la anterior y por tanto no sería posible instanciar objetos de la misma.

Las clases abstractas se representan en UML poniendo su nombre en cursiva, tal y como aparece en el diagrama. Los métodos virtuales puros se representan también en cursiva:

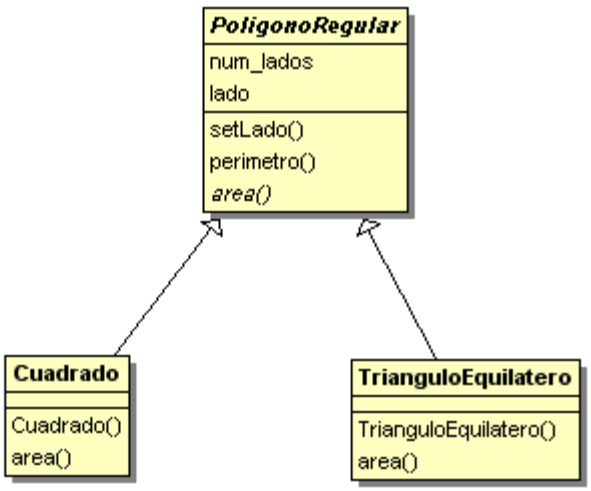


Figura 9-1. Representación de una clase abstracta en UML

9.3. DISEÑO CON INTERFACES Y POLIMORFISMO

Ya se ha esbozado en capítulos anteriores la posibilidad de que el jugador realice distintos tipos de disparo, en función de la tecla que pulse. Hasta ahora no se había tenido en cuenta la gestión de dichos elementos mediante una única lista de disparos, así como cuales son las relaciones con el resto de clases de la aplicación. Se implementó una

clase `Disparo` que representaba el disparo de un gancho que se quedaba pegado al techo o a la plataforma según con cual de ellos impactara. El disparo desaparecía cuando impactaba con una `Esfera`, ya fuera su extremo o cualquier parte del mismo.

9.3.1. Objetivos

Vamos a realizar ahora una versión más elaborada del juego, en la que se le permita al jugador realizar distintos tipos de disparos:

- Un gancho normal, que va relativamente lento y se representa con una estela simple de color cyan.
- Un gancho "especial" que va considerablemente más rápido y que se representa en amarillo, con un radio más grande y con una esfera
- Una lanza, que va aun más rápido que los anteriores, pero que no tiene estela, sino una longitud fija. Se representa en color magenta.

Las interacciones serán como sigue:

- Ambos ganchos se quedan pegados al techo o a la plataforma cuando impactan con ellos. Hasta que una esfera les golpee, permanecerán así pegados.
- Cuando la lanza impacta una superficie, desaparece, no se queda pegada a la misma.
- Una esfera impacta con un gancho si toca cualquier punto de su estela (incluido el extremo del disparo).
- Se debe de tener en cuenta que las lanzas no tienen estela, pero si una longitud fija. Si cualquier punto de la lanza toca una esfera, hace desaparecer la esfera, independientemente del tamaño de la misma.

Los distintos tipos de disparos se pueden ver en la figura anterior. Inicialmente se podrá realizar cada disparo con una tecla diferente. Más adelante en este capítulo, desarrollaremos una solución típica de los videojuegos en la que se habilita al jugador a utilizar los distintos disparos en función de otros criterios como podría ser la recolección de un bonus específico o un nivel de puntuación logrado.

Figura 9-2. Distintos tipos de disparos: Gancho, GanchoEspecial y Lanza

Se presupone que ya se dispone de una clase `ListaDisparos` que de modo similar a `ListaEsferas`, permite la gestión de un número variable de disparos. Dicha clase debería tener como declaración:

```
#define MAX_DISPAROS 10
#include "Disparo.h"
#include "Caja.h"

class ListaDisparos
{
public:
	ListaDisparos();
	virtual ~ListaDisparos();

	bool Agregar(Disparo* d);
	void DestruirContenido();
	void Eliminar(Disparo* d);
	void Eliminar(int index);

	void Mueve(float t);
	void Dibuja();

	void Colision(Pared p);
	void Colision(Caja c);

	int GetNumero(){return numero;}
	Disparo* operator[] (int index);

private:
	Disparo * lista[MAX_DISPAROS];
	int numero;
};
```

Como se observa, la clase contiene un vector de punteros a `Disparo`, lo que permite el uso del polimorfismo.

La clase `ListaDisparos` podría de hecho gestionar una forma un poco más elaborada de trabajar, permitiendo un número creciente de disparos posibles, de tal forma que inicialmente el jugador solo pueda realizar un disparo y según va sumando puntos, aguantando tiempo o recogiendo bonus, se le permite ir realizando más disparos simultáneos. Simplemente añadiendo una variable de tipo entero "numero de disparos posibles" a la misma con sus correspondientes métodos de acceso (garantizando que es siempre menor que `MAX_DISPAROS`), se podría conseguir este efecto. No obstante, esto no tiene mayor interés en este punto, y suponemos que el jugador puede realizar desde el principio un número simultáneo de disparos igual a `MAX_DISPAROS`.

9.3.2. Diseño con interfaces

Lo primero que se observa cuando analizamos los distintos tipos de disparos que tenemos es que a priori tienen pocos atributos en común: Los ganchos podrían tener un atributo de tipo `Vector2D` denominado `origen`. Sin embargo, no parece que este atributo tenga sentido para la `Lanza`. La `Lanza` tendrá como atributo una `longitud`.

De la misma forma el atributo `radio` que tienen los ganchos que representa el tamaño de la cabeza del mismo, no parece que tenga mucho sentido en la `Lanza`. Recuérdese que en una jerarquía de clase **se debe cumplir la regla del 100%** y no puede existir ningún atributo o método de una clase base que no se aplique a las clases derivadas.

¿Qué sucede entonces? ¿No existe nada en común entre los distintos disparos? Si que existe. Por ejemplo, todos los disparos tienen una representación gráfica y por tanto tendrán un método que se podría denominar en todos ellos `Dibuja()`. No obstante, el lector ya se habrá dado cuenta que el cuerpo del método `Dibuja()` no es igual para los distintos disparos. Entonces, ¿qué sentido tiene una clase base con un método `Dibuja()` que no se puede implementar? La respuesta es simple: Mediante una clase base Disparo, se consigue un buen diseño, una arquitectura software muy fácilmente extensible y preparada para modificaciones futuras.

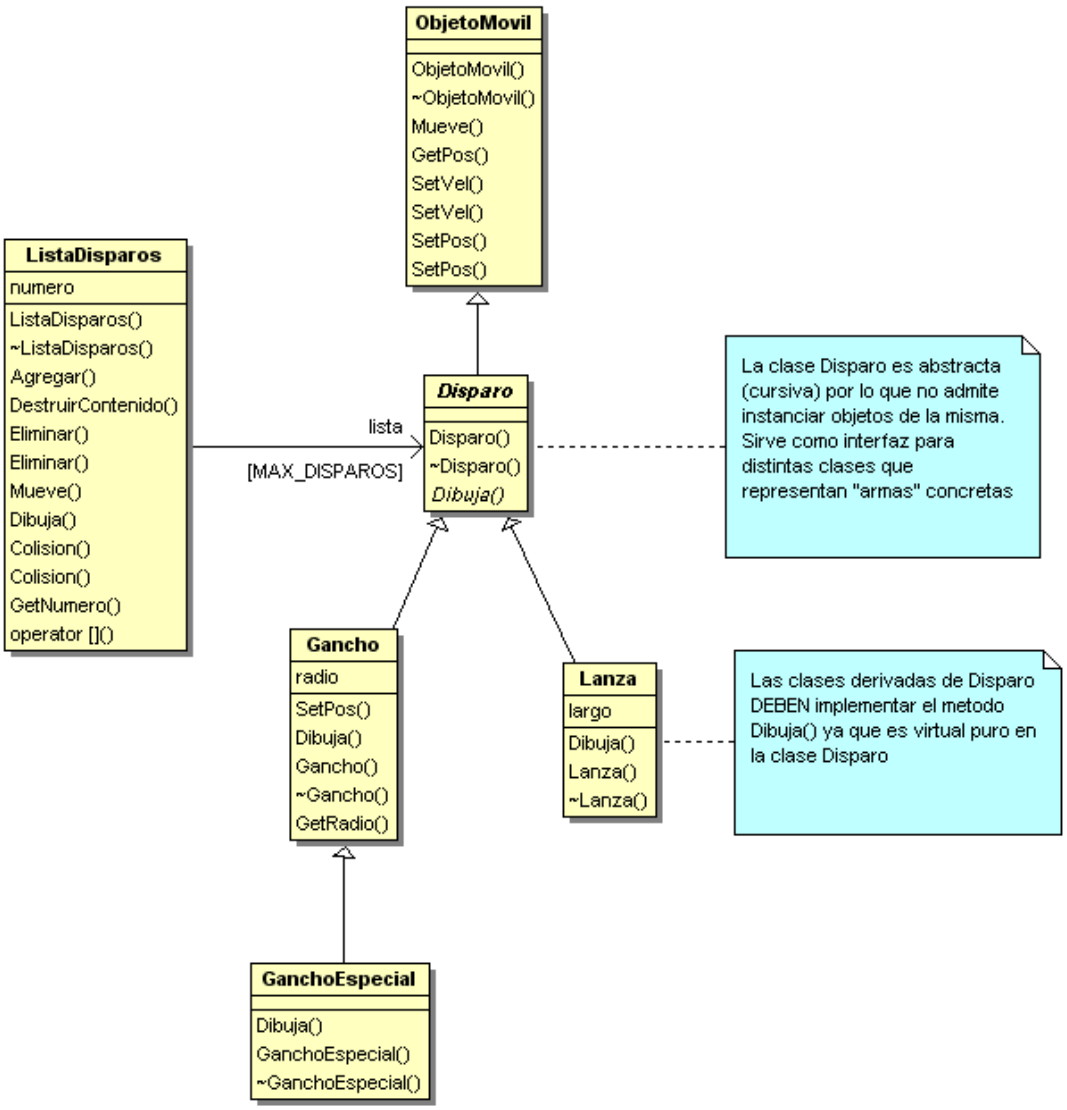


Figura 9-3. Diagrama de clases: La clase Disparo como Interfaz

Vamos a diseñar esta solución, que se representa en un diagrama de clases de diseño (DCD) en la figura 9-3.

9.3.3. La interfaz: Clase Disparo

Se parte de una clase base que denominaremos `Disparo`. Esta clase base será abstracta, es decir no se podrán instanciar objetos de dicha clase. Es una abstracción de lo que es un disparo concreto. Esta clase `Disparo` sigue heredando de `ObjetoMovil`, ya que suponemos que cualquier disparo seguirá teniendo una posición, una velocidad y una posible aceleración, y un movimiento según dichos parámetros. Esta clase es la que denominaremos de "**Interfaz**". No está pensada para representar un objeto físico concreto, sino únicamente para permitir que la clase `ListaDisparos` pueda manejar cualquier tipo de disparo concreto (ganchos, lanzas, etc.) a través de esta interfaz.

El método que convierte la clase en abstracta es el método `Dibuja()`, tal y como se muestra en la declaración de la clase `Disparo` una vez introducida esta modificación:

```
class Disparo  :public ObjetoMovil
{
public:
	Disparo();
	virtual ~Disparo();

	virtual void Dibuja()=0;
};
```

La implementación de la clase `Disparo` también tiene que ser vaciada. Se recuerda que el hecho de declarar el método `Dibuja()` como virtual puro **prohíbe** la implementación de `Disparo::Dibuja()` y **obliga** a las clases concretas que deriven de la clase `Disparo` a implementar dicho método.

Al haber eliminado tantas cosas de la clase `Disparo`, si intentamos compilar, saldrán multitud de errores. Hay que realizar todavía bastante trabajo.

9.3.4. Las clases concretas: Gancho, GanchoEspecial, Lanza

A continuación, derivaremos de la clase base `Disparo`, tantas clases como "disparos" concretos tengamos en el juego. **Importante: Al crear estas clases nuevas, recordar guardarlas en las subcarpetas "dominio".** La clase `Gancho`, representa el disparo tal y como lo habíamos manejado hasta el momento, con un atributo denominado `origen`, que es el punto del que parte, un radio que representa el tamaño de la cabeza del proyectil, y siendo dibujado con su estela. Su declaración quedaría entonces como sigue:

```
#include "Disparo.h"

class Gancho : public Disparo
{
public:
	Gancho();
	virtual ~Gancho();

	void SetPos(float x, float y);
	float GetRadio(){return radio;}
	void Dibuja();

protected:
	float radio;
	Vector2D origen;

};
```

La implementación de los métodos de la clase `Gancho` son básicamente los mismos que ya disponíamos anteriormente:

```
Gancho::Gancho()
{
	radio=0.25f;
	velocidad.y=8;
}
```

```
void Gancho::Dibuja()
{
	glColor3f(0.0f,1.0f,1.0f);

	glDisable(GL_LIGHTING);
	glBegin(GL_LINES);
		glVertex3f(origen.x,origen.y,0);
		glVertex3f(posicion.x,posicion.y,0);
	glEnd();
	glEnable(GL_LIGHTING);

	glPushMatrix();
	glTranslatef(posicion.x,posicion.y,0);

	glutSolidSphere(radio, 20, 20);
	glPopMatrix();
}

void Gancho::SetPos(float x, float y)
{
	Disparo::SetPos(x,y);
	origen=posicion;
}
```

La clase `GanchoEspecial` es muy sencilla, ya que se puede derivar directamente de la clase Gancho, cambiando únicamente su dibujo y sus parámetros por defecto en el constructor. Así la declaración de la clase quedaría como:

```
#include "Gancho.h"

class GanchoEspecial  :public Gancho
{
public:
	GanchoEspecial();
	virtual ~GanchoEspecial();

	void Dibuja();
};
```

La implementación de sus métodos sería:

```
GanchoEspecial::GanchoEspecial()
{
	radio=0.4f;
	velocidad.y=15;
}
```

Donde el dibujo utiliza otro color y traza dos estelas:

```
void GanchoEspecial::Dibuja()
{
	//amarillo
	glColor3f(1.0f,1.0f,0.0f);

	glDisable(GL_LIGHTING);
	glLineWidth(2.0f);

	glBegin(GL_LINES);
		glVertex3f(origen.x-0.1,origen.y,0);
		glVertex3f(posicion.x-0.1,posicion.y,0);
		glVertex3f(origen.x+0.1,origen.y,0);
		glVertex3f(posicion.x+0.1,posicion.y,0);
```

```
	glEnd();
	glEnable(GL_LIGHTING);
	glLineWidth(1.0f);

	glPushMatrix();
	glTranslatef(posicion.x,posicion.y,0);

	glutSolidSphere(radio, 20, 20);
	glPopMatrix();
}
```

Por otra parte, la clase `Lanza` es ligeramente diferente a las anteriores, teniendo como atributo principal una longitud, tal y como se muestra en la declaración de la misma:

```
#include "Disparo.h"

class Lanza : public Disparo
{
public:
	void Dibuja();
	Lanza();
	virtual ~Lanza();

protected:
	float largo;
};
```

La implementación de los métodos podría ser la siguiente:

```
Lanza::Lanza()
{
	largo=1.5;
	velocidad.y=20;
}

void Lanza::Dibuja()
{
	glColor3f(1.0f,0.0f,1.0f);

	glDisable(GL_LIGHTING);
	glLineWidth(2.0f);
	glBegin(GL_LINES);
		glVertex3f(posicion.x,posicion.y-largo,0);
		glVertex3f(posicion.x,posicion.y,0);
	glEnd();

	glEnable(GL_LIGHTING);
	glLineWidth(1.0f);
	glPushMatrix();
	glTranslatef(posicion.x,posicion.y,0);
	glRotatef(-90,1,0,0);
	glutSolidCone(largo/5, largo/2,20 ,1);
	glPopMatrix();
}
```

9.3.5. Polimorfismo

En el vector de la clase `ListaDisparos`, podemos introducir cualquier objeto que herede de la clase base Disparo, ya que el vector es un vector de punteros y un puntero a una clase base puede apuntar a un objeto de una clase derivada. Al declarar el método `Dibuja()` como virtual, el polimorfismo aparece cuando la clase `ListaDisparos` invoca el método `Dibuja()`, siendo ejecutado según el tipo. Para ver realmente esta funcionalidad, procedemos a crear los disparos en función de la pulsación de distintas teclas, y añadirlos a la `ListaDisparos`, tal y como se muestra en el código siguiente:

```
void Mundo::Tecla(unsigned char key)
{
	switch(key)
	{
		case ' ':
			{
				Disparo* d=new Gancho();
				Vector2D pos=hombre.GetPos();
				d->SetPos(pos.x,pos.y);
				disparos.Agregar(d);
				break;
			}
		case 'z':
			{
				Disparo* d=new GanchoEspecial();
				Vector2D pos=hombre.GetPos();
				d->SetPos(pos.x,pos.y);
				disparos.Agregar(d);
				break;
			}
		case 'x':
			{
				Disparo* d=new Lanza();
				Vector2D pos=hombre.GetPos();
				d->SetPos(pos.x,pos.y);
				disparos.Agregar(d);
				break;
			}
	…
}
```

Probamos en este punto a compilar. Todavía existirán unos cuantos errores, con las interacciones de los disparos. Para probar el programa, se puede poner entre comentarios las funciones respectivas de la clase `Interaccion`, así como las llamadas a dichas funciones.

El resultado obtenido es el siguiente:

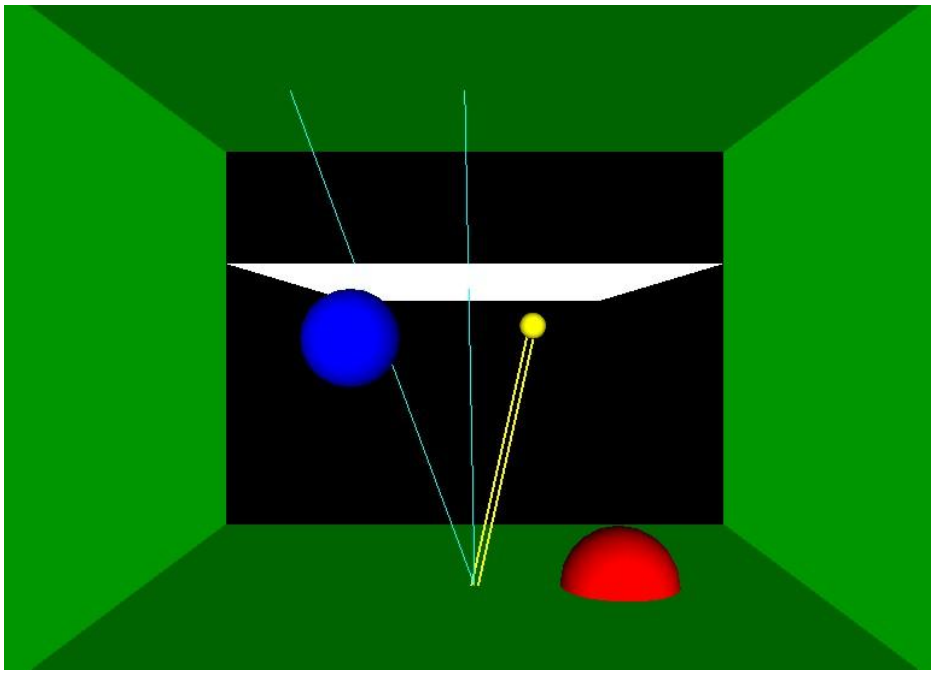


Figura 9-4. Los disparos saliendo del origen.

¿Por qué los disparos salen ahora del origen, a pesar de que hemos escrito el método `SetPos()` correspondiente? La respuesta está otra vez en el polimorfismo (en este caso en la falta de polimorfismo). El método `SetPos()` no es virtual, luego cuando se invoca se invoca el código existente en la clase base `ObjetoMovil`. Si queremos solucionar el problema basta con cambiar en dicha clase:

```
class ObjetoMovil
{
public:
      virtual void SetPos(float x,float y);
```

También se puede apreciar que la clase `DisparoEspecial` ya no es utilizada, con lo que puede ser eliminada del proyecto.

9.3.6. Las interacciones

Tal y como se ha planteado la clase `Interaccion`, esta realiza cálculos de colisión y rebote entre pares de objetos. Pero parece lógico que para poder calcular algo, dichos objetos tienen que ser concretos. En este caso, podremos calcular las colisiones de los ganchos y las lanzas con el resto de objetos. Nótese que como `GanchoEspecial` hereda de `Gancho` y únicamente difiere en su velocidad y dibujo, puede reaprovechar las interacciones de `Gancho`:

```
static bool Colision(Gancho d, Pared p);
static bool Colision(Gancho d, Caja c);
static bool Colision(Gancho d, Esfera e);
static bool Colision(Lanza d, Pared p);
static bool Colision(Lanza d, Caja c);
static bool Colision(Lanza d, Esfera e);
```

Donde las funciones del `Gancho` son exactamente iguales a las que disponíamos antes para el `Disparo`. Como la lanza tiene unos atributos distintos, las funciones de interacción de la lanza se suministran a continuación:

```
bool Interaccion::Colision(Lanza d, Pared p)
{
	Vector2D pos=d.GetPos();
	float dist=p.Distancia(pos);
	if(dist<0.25)
		return true;
	return false;
}
bool Interaccion::Colision(Lanza d, Caja c)
{
	//utilizo la funcion anterior
	return Colision(d,c.techo);
}
bool Interaccion::Colision(Lanza d, Esfera e)
{
	Pared aux; //Creamos una pared auxiliar
	Vector2D p1=d.posicion;
	Vector2D p2=p1;
	p1.y-=d.largo;

	aux.SetPos(p1.x,p1.y,p2.x,p2.y); //Que coincida con el disparo.
	float dist=aux.Distancia(e.posicion); // su distancia al centro
	if(dist<e.radio)
		return true;
	return false;
}
```

9.4. POLIMORFISMO VS. BAJO ACOPLAMIENTO: ALTERNATIVAS BASADAS EN EL TIPO

Aun queda un problema por solucionar. Echemos un vistazo a la función que controla la colisión de la lista de disparos con una pared:

```
void ListaDisparos::Colision(Pared p)
{
	for(int i=0;i<numero;i++)
	{
		if(Interaccion::Colision(*(lista[i]),p))
		{
			lista[i]->SetVel(0,0);
		}
	}
}
```

El problema es que la función correspondiente `Interaccion::Colision()` ya no existe para un disparo genérico sino para cada uno de los disparos (gancho, lanza) concretos.

Se debe de tener en cuenta los siguientes hechos:

- En la clase `ListaDisparos` se ha perdido la información del tipo concreto de los objetos (disparos) que contiene. La clase `ListaDisparos` solo conoce la clase `Disparo`, y esta no tiene suficiente información. Por tanto no hay ahí ninguna información que permita discernir la geometría de los diferentes disparos.
- El hecho de que se puedan dibujar de diferente forma a través del polimorfismo implica que dichos métodos están implementados en las clases pertenecientes a dicha jerarquía, pero no fuera de dicha jerarquía.

Podríamos decidir utilizar el polimorfismo para el cálculo de colisiones e implementar unos métodos virtuales puros en la clase `Disparo`:

```
//si decidieramos resolverlo polimorficamente
class Disparo  :public ObjetoMovil
{
      …
      virtual bool Colision(Pared p)=0;
      virtual bool Colision(Esfera e)=0;
};
```

De esta forma, cada clase derivada (`Lanza`, `Gancho`, etc.) debería implementar dichos métodos de colisión, ya teniendo en cuenta su geometría particular.

Aunque dicha solución es perfectamente factible desde el punto de vista de la implementación, el problema es que la simulación de las interacciones físicas y colisiones hemos decidido implementarlos en la clase `Interaccion`, siguiendo los principios del patrón **Indirección** y consiguiendo una **Alta Cohesión**. Dicho de otra forma: hemos agrupado todas las ecuaciones de colisiones dentro de la clase `Interaccion`, consiguiendo además que las diferentes clases (`Disparo`, `Esfera`, `Pared`, etc.) no tengan que conocerse entre ellas (no estén acopladas). Y obviamente queremos seguir respetando este principio y mantener en esta clase las colisiones, incluidas las de todos los disparos concretos, ya sean ganchos, lanzas o cualquier otro tipo que se nos pueda ocurrir en el futuro.

Dado que en la clase `ListaDisparos`, toda la información que tendremos disponible será la contenida en la clase `Disparo`, debemos añadir algo a esta última que nos permita conocer el tipo real de objeto que es.

Para ello seguiremos los siguientes pasos:

1. Declaramos una variable protegida de tipo entero en la clase abstracta de interfaz `Disparo`, denominada `tipo`.
2. Añadimos un método de acceso `GetTipo()` a la clase `Disparo` que permita consultar dicho tipo. Nótese que es importante NO implementar un método que permita cambiar dicha variable.
3. Añadimos `#define` en el fichero "***Disparo.h***" que sirvan como nemotécnicos para identificar el tipo de objeto.
4. Hacemos que cada constructor de cada clase asigne el valor correspondiente a la variable tipo.

La declaración de la clase `Disparo` queda entonces como sigue:

```
#define NINGUNO                    -1
#define GANCHO                     0
```

```
#define GANCHO_ESPECIAL             1
#define LANZA                       2

#include "comun/Vector2D.h"
#include "ObjetoMovil.h"

class Disparo  :public ObjetoMovil
{

public:
      Disparo();
      virtual ~Disparo();

      int GetTipo();
      virtual void Dibuja()=0;

protected:
      int tipo;
};
```

El cuerpo de la función que devuelve el tipo de objeto es obvio:

```
int Disparo::GetTipo()
{
      return tipo;
}
```

Por último es importante acordarse de dar un valor al tipo en los constructores:

```
Disparo::Disparo()
{
      tipo=NINGUNO;
}
Lanza::Lanza()
{
      tipo=LANZA;
      …
}
Gancho::Gancho()
{
      tipo=GANCHO;
      …
}
GanchoEspecial::GanchoEspecial()
{
      tipo=GANCHO_ESPECIAL;
      …
}
```

Con esto ya tenemos la estructura definida que nos permite conocer el tipo de objeto concreto a partir de un puntero o una referencia a un objeto de la clase base. Cuando en algún lugar de nuestro código como en la clase `ListaDisparos`, tenemos un puntero (o referencia) a un objeto de tipo `Disparo`, consultando a través de `GetTipo()` su tipo concreto, podemos conocer el mismo, y mediante un `cast`, declarar un nuevo puntero que permite acceder a los atributos y métodos particulares de la clase derivada, o en este caso invocar la sobrecarga adecuada del método `Interaccion::Colision`. Así, la gestión de colisiones con una pared en la lista de disparos queda:

```
void ListaDisparos::Colision(Pared p)
{
```

```
	for(int i=numero-1;i>=0;i--)
	{
		int tipo=lista[i]->GetTipo();
		if(tipo==GANCHO || tipo==GANCHO_ESPECIAL)
		{
			Gancho* g=(Gancho*)lista[i];
			if(Interaccion::Colision(*g,p))
			{
				lista[i]->SetVel(0,0);
			}
		}
		if(tipo==LANZA)
		{
			Lanza* g=(Lanza*)lista[i];
			if(Interaccion::Colision(*g,p))
			{
				Eliminar(i);
			}
		}
	}
}
```

Por supuesto la clase `Interaccion` queda acoplada mediante una dependencia a las clases derivadas, exactamente igual que estaba acoplada a las otras clases (`Hombre`, `Esfera`, etc.), tal y como se representa en el siguiente diagrama:

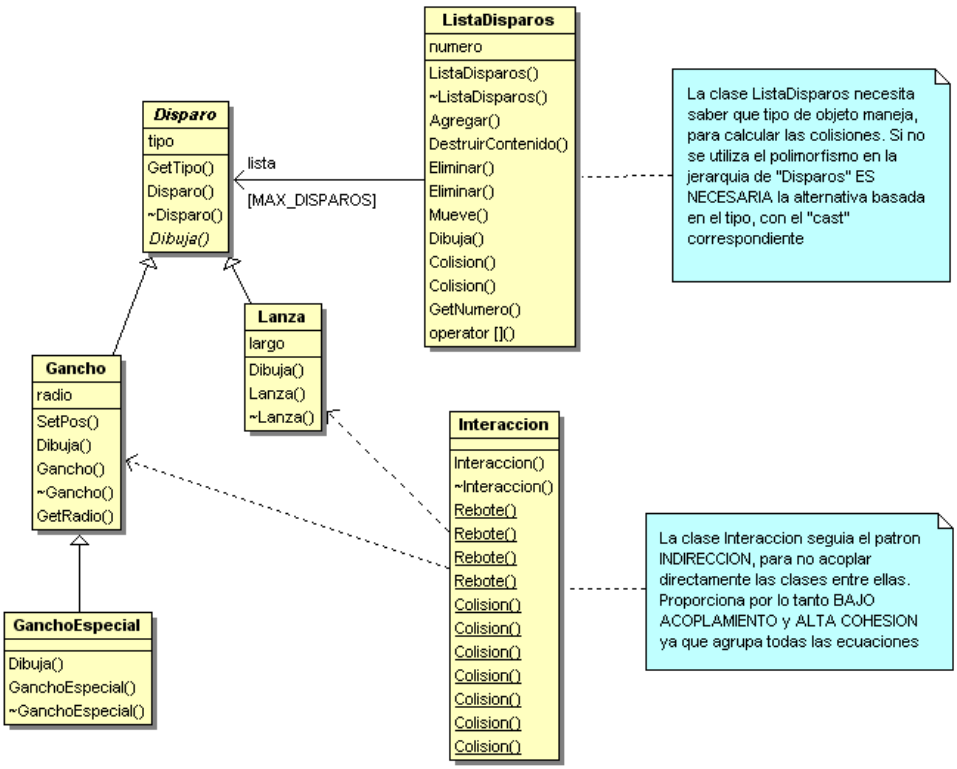


Figura 9-5. Manejo de las interacciones según el tipo de objeto

La función `ListaDisparos::Colision(Caja)` es muy similar.

Ejercicio: Modificar la función Mundo::Mueve(), de tal forma que las interacciones de las esferas y los disparos funcionen correctamente, incluyendo el hecho de que las lanzas siempre hacen desaparecer la esfera que impactan, independientemente de su radio.

9.5. CREACIÓN DE OBJETOS MEDIANTE UNA FACTORÍA

No tiene demasiado sentido que el jugador pueda recurrir desde el principio a realizar el tipo de disparo que quiera. Vamos a implementar el comportamiento normal, en el que el jugador inicialmente solo puede realizar el tipo de disparo más sencillo (gancho), y a medida que va alcanzando hitos o realizando acciones, pasa a disparar disparos más potentes como el gancho especial y las lanzas. En cualquier caso el disparo se realiza con la barra espaciadora, y es la propia aplicación la encargada de realizar el "mejor" disparo posible al que tiene derecho el jugador.

Para ello añadiremos una variable a la clase `Hombre` que represente el numero de bonus acumulados hasta este momento, así como las funciones respectivas que me permiten el acceso a dicho dato, incluyendo una función de conveniencia que incrementa dicho numero:

```
class Hombre  :public ObjetoMovil
{
      void IncrementaNumBonus();
      void SetNumBonus(int num);
      int GetNumBonus();

protected:
      int num_bonus;
};
```

Aparte de inicializar `num_bonus` a cero en el constructor de `Hombre`, también debemos establecer que el hombre pierde los bonus acumulados si es alcanzado por una pelota:

```
void Mundo::Mueve()
{
      …
      Esfera *aux=esferas.Colision(hombre);
      if(aux!=0)
      {
            hombre.SetNumBonus(0);
            impacto=true;
      }
```

Ahora ha llegado el momento de crear los objetos (`new`), en función de los bonus acumulados en el hombre. Aunque esta lógica podría ser implementada en la clase `Mundo`, en el método que responde al teclado, el patrón Factoría nos recomienda ubicar la lógica de creación de objetos en una clase destinada específicamente a ello. Por lo tanto añadimos a nuestro diseño y nuestro código dicha clase, que denominaremos `Factoria`.

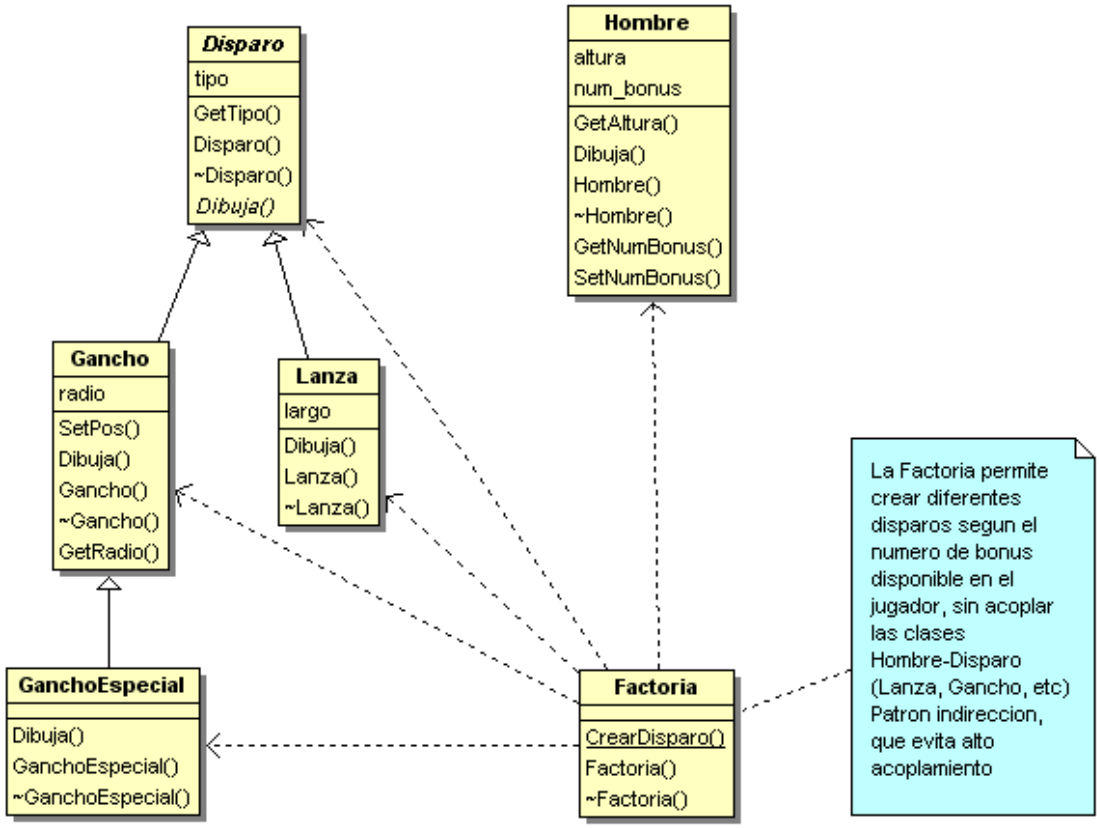


Figura 9-6. Uso de una Factoría para la creación condicional de objetos

A nuestra Factoría le añadimos un método estático que devuelve un puntero a `Disparo` y que admite como parámetro un objeto de la clase `Hombre`. Así nuestra clase Factoría queda como sigue:

```
#include "Hombre.h"
#include "Disparo.h"

class Factoria
{
public:

      Factoria();
      virtual ~Factoria();

      static Disparo* CrearDisparo(Hombre h);
};
```

El método de creación del disparo es simple, solo consulta los bonus del hombre y crea un nuevo disparo del tipo correspondiente. Nótese que aunque el puntero que se devuelve es del tipo de la clase base Disparo, realmente está apuntando a un objeto de la clase derivada (`Gancho`, `Lanza`, etc.). Por último se encarga también de inicializar las coordenadas del disparo en función de la posición del hombre.

```
Disparo* Factoria::CrearDisparo(Hombre h)
{
      Disparo* d=0;
```

```
	if(h.GetNumBonus()==1)
		d=new GanchoEspecial;
	else if(h.GetNumBonus()==2)
		d=new Lanza;
	else
		d=new Gancho;

	Vector2D pos=h.GetPos();
	d->SetPos(pos.x,pos.y);

	return d;
}
```

Los detalles de creación de los objetos han sido ocultados dentro de la Factoría, de tal forma que el método `Mundo::Tecla()` ha quedado muy simplificado. Nótese como se comprueba primero que el número de disparos máximo no ha sido superado.

```
void Mundo::Tecla(unsigned char key)
{
	switch(key)
	{
		case ' ': if(disparos.GetNumero()<MAX_DISPAROS)
				{
					Disparo* d=Factoria::CrearDisparo(hombre);
					disparos.Agregar(d);
				}
				break;
	}
}
```

Una forma fácil de probar la nueva funcionalidad consiste en asignar un número de bonus al hombre proporcional a cada nivel, de tal forma que en la primera pantalla se dispondrá del gancho, en la segunda pantalla del gancho especial y en la tercera pantalla de la lanza:

```
bool Mundo::CargarNivel()
{
	…
	if(nivel==1)
	{
		hombre.SetNumBonus(0);
	}
	if(nivel==2)
	{
		hombre.SetNumBonus(1);
	}
	if(nivel==3)
	{
		hombre.SetNumBonus(2);
```

Con todo esto tenemos el juego casi terminado. El remate final queda como ejercicio para el lector, siguiendo las líneas del siguiente apartado.

9.6. EJERCICIOS PROPUESTOS

- Implementar las interacciones con el bonus, de forma que recogerlo implica el incremento de los bonus del hombre, proporcionando nuevos disparos mejores. El bonus cae (solo uno en cada instante de tiempo) cuando una esfera se divide en dos.
- Si el hombre es impactado por una esfera, y tiene bonus, pierde los bonus, volviendo al disparo inicial, pero no pierde el juego.
- Añadir un método de creación de esferas a la Factoría, de tal forma que devuelva una esfera con un radio, color, posición, velocidad y aceleración aleatorios, dentro de unos rangos razonables. Para ello la caja será pasada como parámetro a dicho método. Además, con una probabilidad de 1/5 devolverá una `EsferaPulsante`, en lugar de una `Esfera`. Utilizar este método para crear un numero de esferas creciente según el nivel, en la función `CargarNivel()` para niveles superiores al 3 y hasta el 5

A. ANEXOS

B. TEXTURAS Y SONIDOS

Realmente no hay juego de computador moderno que no disponga de unos gráficos muy atractivos, así como de música y/o sonidos. En este anexo se realiza una pequeña introducción a las texturas con OpenGL, y se describe una función de Windows que permite añadir música a nuestro juego.

CARGAR LAS TEXTURAS

Para utilizar texturas en el dibujo de nuestro juego, es necesario conocer un poco más las funciones correspondientes de OpenGL. Para facilitar la tarea al lector, se ha dotado a la clase OpenGL de gran parte de la funcionalidad relacionada con las texturas, principalmente la carga en memoria de las mismas. Para ello se ha añadido un método denominado `CargaTextura()` que admite una cadena de texto indicando el nombre del archivo en el que reside la textura, que debe de ser un archivo BITMAP (extensión ***.bmp***) y cuyas dimensiones en píxeles deben de ser potencias de 2, por ejemplo 256x256 píxeles.

El programador debe referenciar las texturas por su nombre, aunque internamente OpenGL las identifica una vez cargadas en memoria como un entero. La función `CargaTextura()` se encarga de mirar si la textura ya ha sido cargada previamente, y devuelve el entero correspondiente si se encuentra. Si la textura no ha sido cargada, primero se carga y después se devuelve el entero. De esta forma, la carga de la textura es totalmente transparente para el usuario, la textura se carga la primera vez que se necesita.

```
#pragma warning (disable: 4786)

#include <string>
#include <vector>

class OpenGL
{
public:
      OpenGL();
      virtual ~OpenGL();
      static void Print(char *mensaje, int x, int y, unsigned char r=255,
                        unsigned char g=255, unsigned char b=255);
      static unsigned int CargaTextura(char* nombre);
protected:
      static std::vector<std::string> nombre_texturas;
      static std::vector<unsigned int> id_texturas;
```

```
};
```

El almacenamiento interno de los enteros y los nombres de las cadenas ha sido implementado haciendo uso de la Standard Template Library (STL), lo que facilita en gran medida la tarea. Aunque no se ha tratado dicha librería en este libro, el usuario no necesita en realidad conocer su funcionamiento, ya que son datos protegidos de la clase que el usuario no necesita conocer ni modificar y que solo la clase OpenGL necesita. La gran ventaja de la encapsulación.

Todos los miembros de la clase OpenGL tienen el modificador `static`.

Para cargar las texturas desde el fichero se necesita utilizar una librería auxiliar de OpenGL, denominada ***Glaux***. En el fichero de código fuente de la clase OpenGL, podemos poner el `include` correspondiente así como la directiva de preprocesador para enlazar con la librería. También hay que recordar la declaración de las variables `static` correspondientes:

```
#include "gl\glaux.h"
#pragma comment (lib, "glaux.lib")
std::vector<std::string> OpenGL::nombre_texturas;
std::vector<unsigned int> OpenGL::id_texturas;
```

La función `CargaTexturas()` primero busca el nombre de la textura en las ya cargadas anteriormente. Si la encuentra devuelve su identificador. En caso contrario invoca una serie de funciones de configuración, para posteriormente cargar la imagen con `auxDIBImageLoadA()`. Si la función tiene éxito, se carga en memoria dicha textura, se elimina la imagen de memoria (ya ha sido cargada por OpenGL) y se expanden las tablas de la clase OpenGL.

```
unsigned int OpenGL::CargaTextura(char* nombre)
{
      for(int i=0;i<id_texturas.size();i++)
            if(0==nombre_texturas[i].compare(nombre))
                  return id_texturas[i];

      unsigned int textura=-1;
      glGenTextures(1, &textura);
      … //codigo de texturas

      AUX_RGBImageRec * mitex=auxDIBImageLoadA(nombre);
      if(mitex!=NULL)
      {
            int ret=gluBuild2DMipmaps( GL_TEXTURE_2D, 3,
                              mitex->sizeX, mitex->sizeY,GL_RGB,
                              GL_UNSIGNED_BYTE,
                              mitex->data );
            delete mitex->data;
            delete mitex;
            id_texturas.push_back(textura);
            std::string cad(nombre);
            nombre_texturas.push_back(cad);
      }

      return textura;
}
```

Las texturas ocupan memoria que debe de ser liberada. Añadimos un método `static` a la clase `OpenGL` que se encargue de tal tarea.

```
void OpenGL::BorraTexturas()
{
	for(int i=0;i<id_texturas.size();i++)
	{
		glDeleteTextures(1,&id_texturas[i]);
	}
	id_texturas.clear();
	nombre_texturas.clear();
    glBindTexture(GL_TEXTURE_2D,NULL);
}
```

E invocamos ese método desde el destructor del coordinador:

```
CoordinadorPang::~CoordinadorPang()
{
	OpenGL::BorraTexturas();
}
```

DIBUJANDO CON TEXTURAS

Dado que a las texturas nos referiremos por el nombre del archivo, parece lógico añadir a las clases un atributo que sea dicho nombre. Añadimos dicho atributo y el método para establecerlo a las paredes:

```
class Pared
{
public:
	void SetFondo(char* f);

protected:
	char fondo[255];
};
```

Nótese que se podría considerar este atributo similar en concepto al color de la pared, y por tanto parece lógico que resida en la propia clase `Pared`. Dicho atributo se inicializa a cero en el constructor, por si acaso el programador decide no utilizarlo y utilizar colores normales como hasta ahora. La función que establece su valor simplemente realiza una copia de la cadena de caracteres:

```
Pared::Pared()
{
	fondo[0]=0;
}

void Pared::SetFondo(char *f)
{
	strcpy(fondo,f);
}
```

La función de dibujo puede analizar si la cadena de caracteres ha sido establecida y actuar en consecuencia, dibujando con colores planos o con texturas:

```
void Pared::Dibuja()
{
	if(fondo[0]==0)
	{
```

```
            … //codigo de dibujo con colores planos
      }
      else
      {
      unsigned int textura=OpenGL::CargaTextura(fondo);

      if(textura!=-1)
      {
            glEnable(GL_TEXTURE_2D);
            glBindTexture(GL_TEXTURE_2D,textura);

            glDisable(GL_LIGHTING);

            glColor3f(1,1,1);
            glBegin(GL_POLYGON);
                  glTexCoord2d(0,0);glVertex3d(limite1.x,limite1.y,10);
                  glTexCoord2d(1,0);glVertex3d(limite2.x,limite2.y,10);
                  glTexCoord2d(1,1);glVertex3d(limite2.x,limite2.y,-10);
                  glTexCoord2d(0,1);glVertex3d(limite1.x,limite1.y,-10);
            glEnd();

            glEnable(GL_LIGHTING);
            glDisable(GL_TEXTURE_2D);
      }
      }
}
```

Se observa que para dibujar con la textura, se tiene que obtener su identificador a través de la clase `OpenGL`, activar las texturas y establecer la textura actual y luego dibujar el polígono estableciendo para cada vértice la coordenada correspondiente de la textura, a través de la función `glTexCoord2d()`, a través de la cual también se puede establecer si queremos que la textura se repita como un patrón en todo nuestro polígono.

Con la caja podemos realizar algo parecido. En este caso podemos configurar el fondo de la misma (a modo de paisaje) y la textura de las paredes. También se podrían configurar distintas texturas para suelo, techo, etc. Lo que se deja como ejercicio para el lector. La definición de la clase sería:

```
class Caja
{
public:
      void SetFondoParedes(char* f);
      void SetFondo(char* fondo);

protected:
      char fondo[255];
};
```

El método que establece la textura de las paredes simplemente utiliza el método visto anteriormente para cada una de ellas:

```
void Caja::SetFondoParedes(char *f)
{
      pared_dcha.SetFondo(f);
      pared_izq.SetFondo(f);
```

```
	techo.SetFondo(f);
	suelo.SetFondo(f);
}
```

La función de dibujo de la caja, dibuja las paredes, el techo y suelo como anteriormente, y además dibuja el fondo de la misma. Es importante resaltar que dicho fondo no ha requerido ser parametrizado como una clase en todo el desarrollo del juego, ya que como se ve es algo puramente decorativo y no tiene otra funcionalidad. Por tanto es lógico que quede restringido a la función `Dibuja()` de la caja.

```
void Caja::Dibuja()
{
	suelo.Dibuja();
	techo.Dibuja();
	pared_izq.Dibuja();
	pared_dcha.Dibuja();

	if(fondo[0]==0)
		return;

	unsigned int textura=OpenGL::CargaTextura(fondo);

	if(textura!=-1)
	{
		glEnable(GL_TEXTURE_2D);
		glBindTexture(GL_TEXTURE_2D,textura);

		glDisable(GL_LIGHTING);
		glColor3f(1,1,1);
		glBegin(GL_POLYGON);
			glTexCoord2d(0.0,0.0);		glVertex3f(-10,0,-10);
			glTexCoord2d(1,0.0);		glVertex3f(10,0,-10);
			glTexCoord2d(1,1);		glVertex3f(10,15,-10);
			glTexCoord2d(0.0,1);		glVertex3f(-10,15,-10);
		glEnd();
		glEnable(GL_LIGHTING);
		glDisable(GL_TEXTURE_2D);
	}

}
```

Lo mismo sucede con el dibujo del `Hombre`. De hecho, como se dibuja siempre con la misma textura, ni siquiera es necesario incluir su nombre como miembro, aunque si planteáramos varios jugadores, si sería necesario:

```
void Hombre::Dibuja()
{
	glPushMatrix();
	glTranslatef(posicion.x,posicion.y,0);

	unsigned int textura=OpenGL::CargaTextura("hombre.bmp");
	if(textura!=-1)
```

```
	{
		glEnable(GL_TEXTURE_2D);
		glBindTexture(GL_TEXTURE_2D,textura);
				glDisable(GL_LIGHTING);
		glColor3f(1,1,1);
		glBegin(GL_POLYGON);
			glTexCoord2d(0.0,0.0);		glVertex3f(-1,0,0);
			glTexCoord2d(1,0.0);		glVertex3f(1,0,0);
			glTexCoord2d(1,1);		glVertex3f(1,altura,0);
			glTexCoord2d(0.0,1);		glVertex3f(-1,altura,0);
		glEnd();
		glEnable(GL_LIGHTING);
		glDisable(GL_TEXTURE_2D);
	}
	glPopMatrix();
}
```

El método `CargarNivel()` es un buen lugar para configurar que texturas se utilizan en cada elemento de la pantalla para cada nivel:

```
bool Mundo::CargarNivel()
{
	…
	if(nivel==1)
	{
		caja.SetFondo("alpes.bmp");
		caja.SetFondoParedes("ladrillos.bmp");
		plataforma.SetFondo("muro.bmp");
		…
	}
	if(nivel==2)
	{
		caja.SetFondo("tierra.bmp");
		caja.SetFondoParedes("muro.bmp");
		plataforma.SetFondo("ladrillos.bmp");
		…
	}
	…
}
```

TEXTURANDO PRIMITIVAS

Para ilustrar la aplicación de texturas a primitivas de dibujo como son las esferas, se ha introducido en la aplicación una pantalla de entrada con el nombre del juego, unas esferas que aparecen solas, rebotan y desaparecen, un fondo, y todo ello es orbitado por el punto de vista. Obviamente esta escena no tiene nada que ver con lo que hemos programado en el objeto `Mundo`. De hecho conviene dejar el objeto de la clase `Mundo` para gestión de la parte central del juego. Así implementamos una nueva clase denominada `EscenaEntrada`, que es utilizada por el coordinador de forma muy similar a la clase Mundo. Si quisiéramos también hacer una tabla con los mejores resultados (nombres, puntuaciones), también sería conveniente encapsular dicha funcionalidad en una clase denominada `GestorMarcadores` por ejemplo, a la que el coordinador enviara los mensajes oportunos en caso de encontrarse en el estado correspondiente.

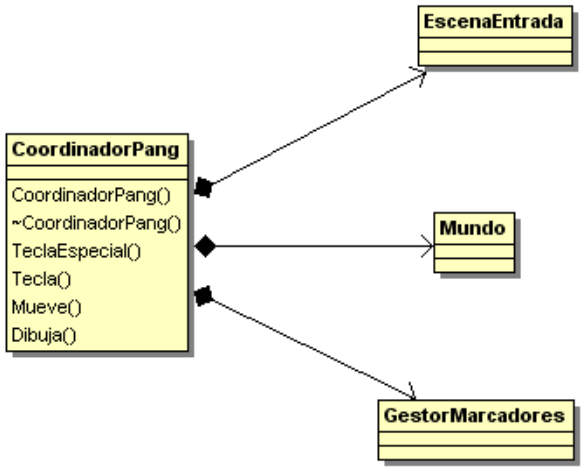


Figura B-1. Estructura de la aplicación para gestión completa del juego

Figura B-2. Escena de entrada en el juego

Lo importante en esta tarea es tener en cuenta que las funciones de GLUT (aquellas que empiezan por `glut`) no sirven para dibujar con texturas, sino que hay que utilizar funciones de menor nivel de la librería GLU. Para dibujar una esfera muy grande, de tal forma que incluso el punto de vista quede dentro, la función de dibujo de `EscenaEntrada` hace:

```
void EscenaEntrada::Dibuja()
{
      …

      textura=OpenGL::CargaTextura("agua.bmp");
      if(textura!=-1)
      {
            glEnable(GL_TEXTURE_2D);
            glBindTexture(GL_TEXTURE_2D,textura);
```

```
            glDisable(GL_LIGHTING);
            glColor3ub(255,255,255);

            GLUquadricObj * qobj=gluNewQuadric();
            gluQuadricTexture  (qobj , GL_TRUE );
            gluSphere(qobj,42,15,15);
            gluDeleteQuadric(qobj);

            glEnable(GL_LIGHTING);
            glDisable(GL_TEXTURE_2D);
      }
}
```

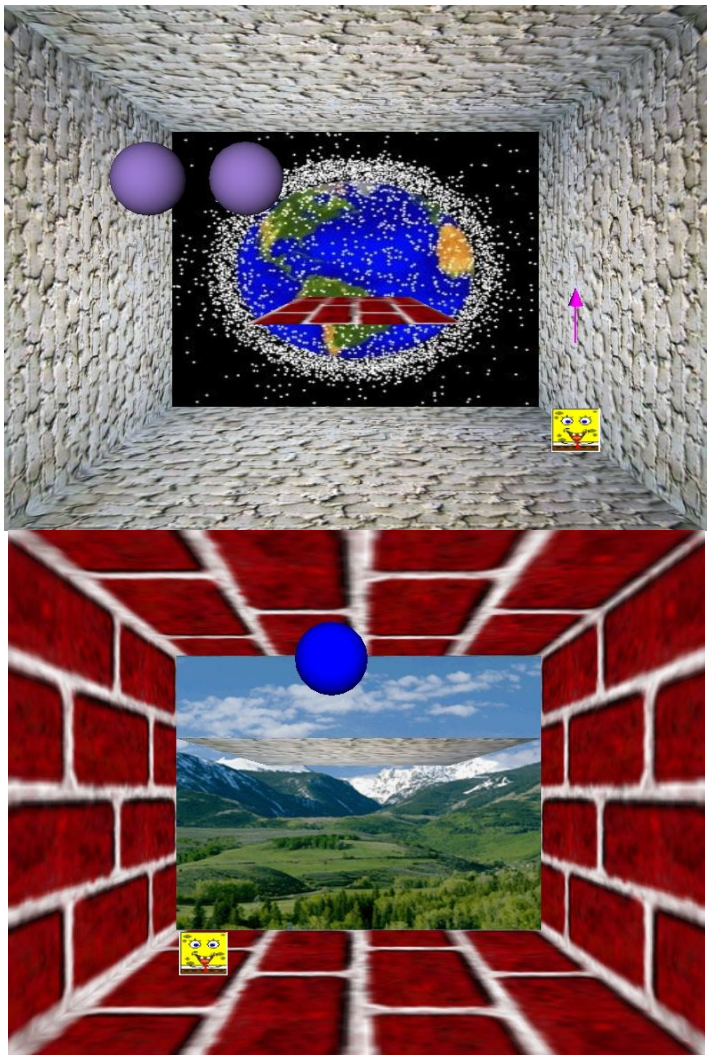


Figura B-3. Distintas texturas en los niveles

SONIDOS

Existe una forma sencilla, aunque muy limitada, de reproducir sonidos y música en nuestro juego, se trata de la función `PlaySound()` de Windows, que viene incluida en Visual Studio. Esta función permite la reproducción de música desde archivos en el disco en formato "***.wav***". Hay que tener en cuenta que esta función no permite reproducir

sonidos simultáneamente, si intentamos reproducir un sonido mientras suena una música, el sonido se parará.

En un uso sencillo, se podría decir que la música o sonidos terminan y empiezan asociados a eventos como el comienzo del juego, el fin o la victoria. Parece razonable entonces que dichos sonidos sean reproducidos por la clase coordinadora. Así la función que gestiona el teclado quedaría:

```
#include <windows.h>

void CoordinadorPang::Tecla(unsigned char key)
{
	if(estado==INICIO)
	{
		if(key=='e')
		{
			mundo.Inicializa();
			PlaySound("TestMusic.wav",NULL,
				SND_FILENAME |SND_ASYNC |SND_LOOP );
			estado=JUEGO;
		}
		if(key=='s')
			exit(0);
	}
}
```

Y de forma similar la función `Mueve()`:

```
void CoordinadorPang::Mueve()
{
	…
		mundo.Mueve();
		if(mundo.GetNumEsferas()==0)
		{
			if(!mundo.CargarNivel())
			{
			PlaySound("Aplauso.wav",NULL,SND_FILENAME |SND_ASYNC );
			estado=FIN;
			}
		}
		if(mundo.Impacto())
		{
			estado=GAMEOVER;
			PlaySound("Golpe.wav",NULL,SND_FILENAME |SND_ASYNC );
		}
	}
}
```

UBICACIÓN DE LOS ARCHIVOS

Tanto en el caso de texturas como de sonidos, hemos ubicado los archivos en la carpeta ***bin*** del proyecto, que es donde se encuentra el ejecutable, ya que estos archivos son necesarios para la ejecución de la aplicación.

No obstante, si se parte de los proyectos de capítulos previos, es necesario modificar la configuración del proyecto tal y como se indica en el anexo siguiente, para indicar

adecuadamente que el directorio de trabajo, ya que el proyecto utilizado no incluía la configuración del "*Working directory*" o directorio de trabajo.

C. CONFIGURACIÓN DE UN PROYECTO VISUAL C++ 6.0

Para que el Visual Studio funcione correctamente con la estructura de carpetas (***bin, include, lib, src***) utilizada en este libro, es necesaria su configuración. Dicha configuración se realiza toda en un cuadro de dialogo denominado "*Project Settings*", que se encuentra en *Menu->Project->Settings*.

La configuración activa durante todo el desarrollo ha sido "*Debug*", por si fuera necesaria la depuración del código. El lector no debería haberlo notado, ya que nuestra aplicación no requiere excesivo cómputo. Para aplicaciones que requieran gran cantidad de procesamiento, la diferencia entre modo *Release y Debug* puede ser considerable.

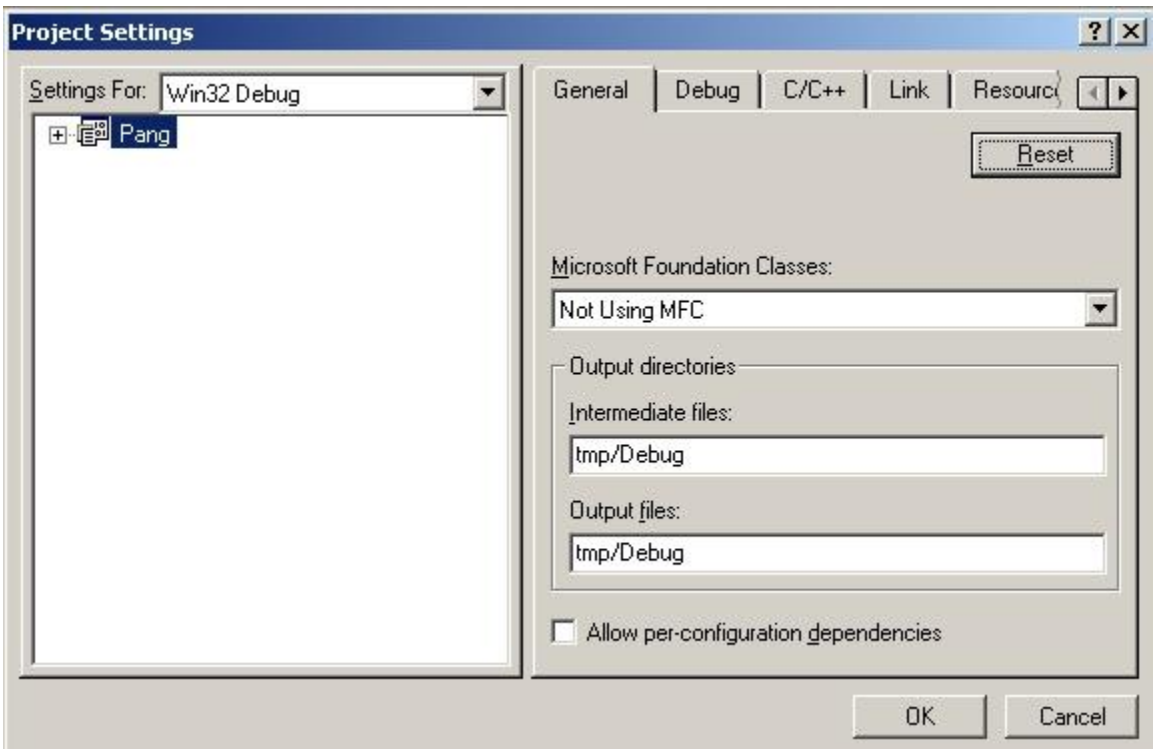


Figura C-1. Configuración general.

El directorio de ejecución tiene que ser también configurado si deseamos que los archivos necesarios para la aplicación como los de texturas y sonidos sean encontrados. Para ello en "*Working directory*" especificamos el directorio "***bin***":

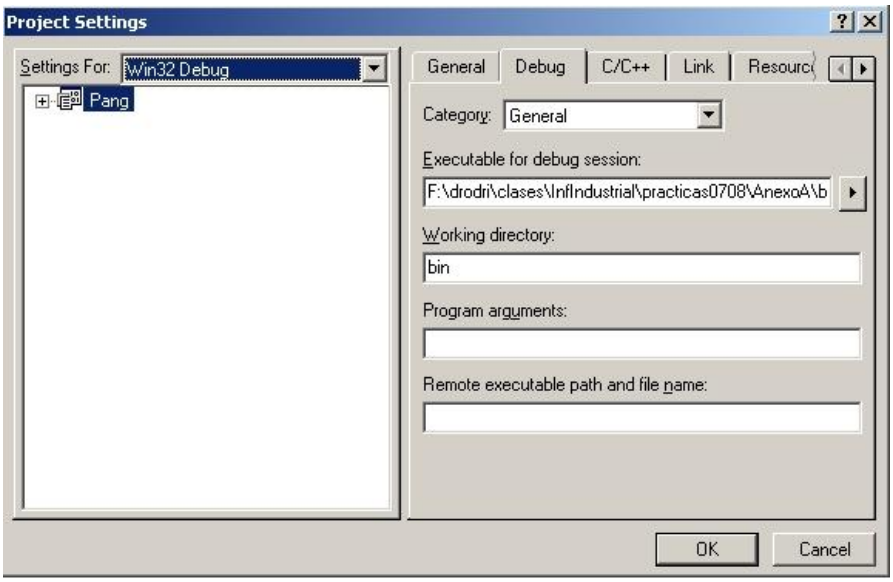


Figura C-2. Configuración del directorio de ejecución.

En la pestaña de configuración general se puede establecer la ubicación de ficheros temporales (como objetos ***.obj***) del proyecto, para cada uno de los modos (*Release y Debug).* En las casillas correspondientes ponemos el camino relativo desde el directorio de trabajo (que es en el que se encuentra el archivo ***Pang.dsw***)

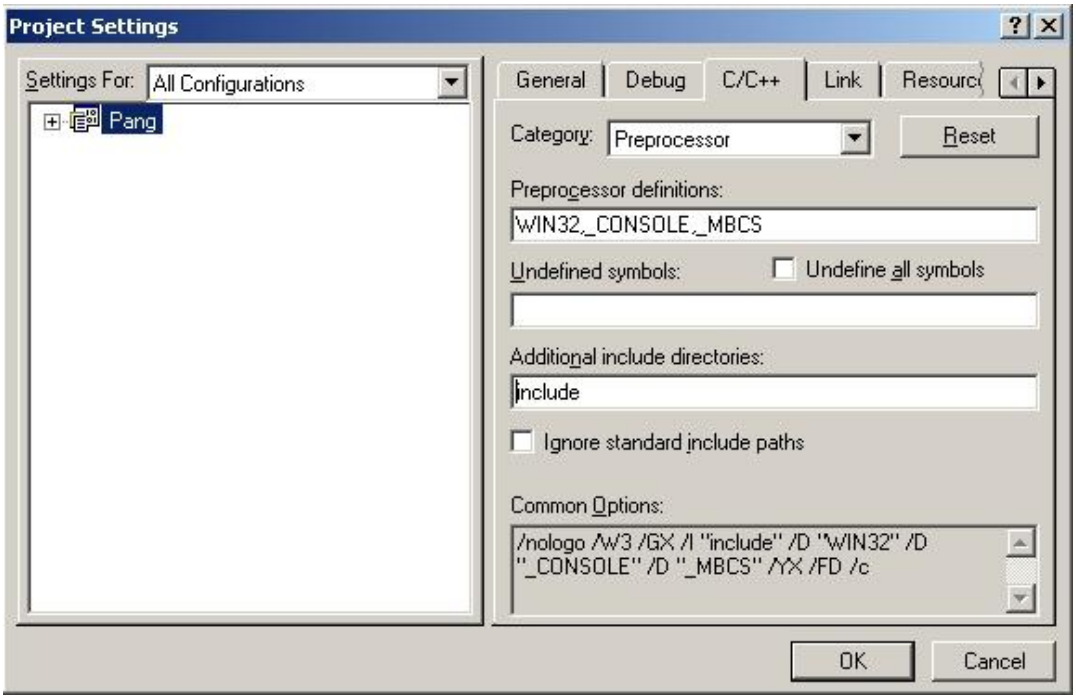


Figura C-3. Configuración del directorio include.

El directorio ***include*** puede ser establecido en la pestaña C/C++, categoría "*Preprocessor*". La ruta local desde el directorio del proyecto es simplemente "***include***".

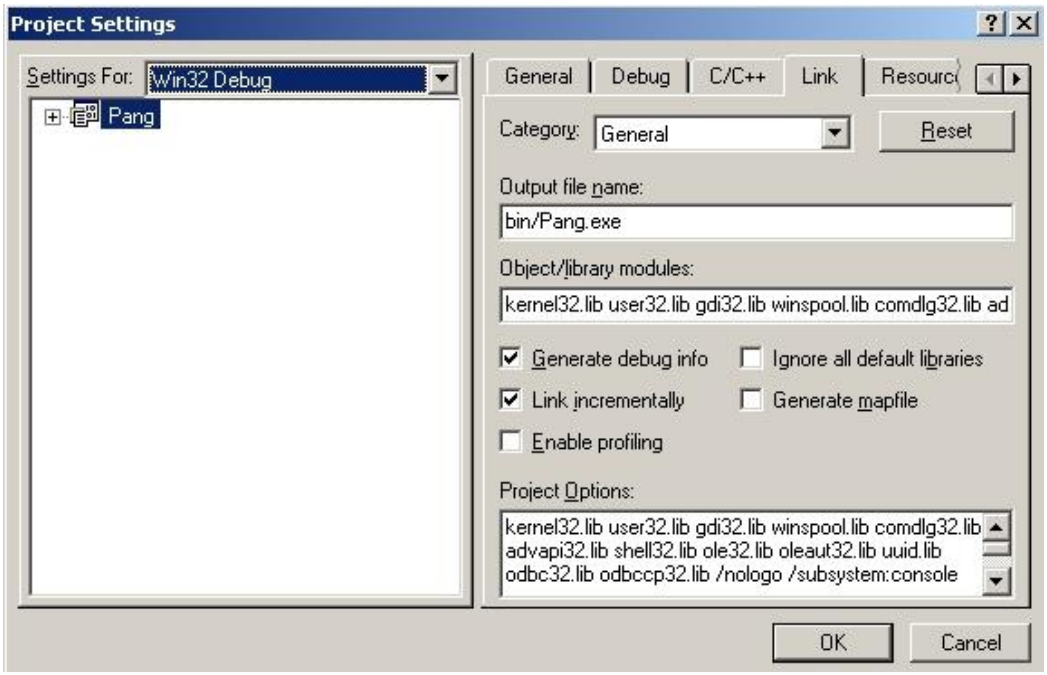


Figura C-4. Configuración de la ubicación del ejecutable.

La ubicación del ejecutable se puede especificar en la pestaña "*Link*", categoría general. Basta con añadir al nombre del archivo de salida, la ruta local al mismo "***bin/Pang.exe***"

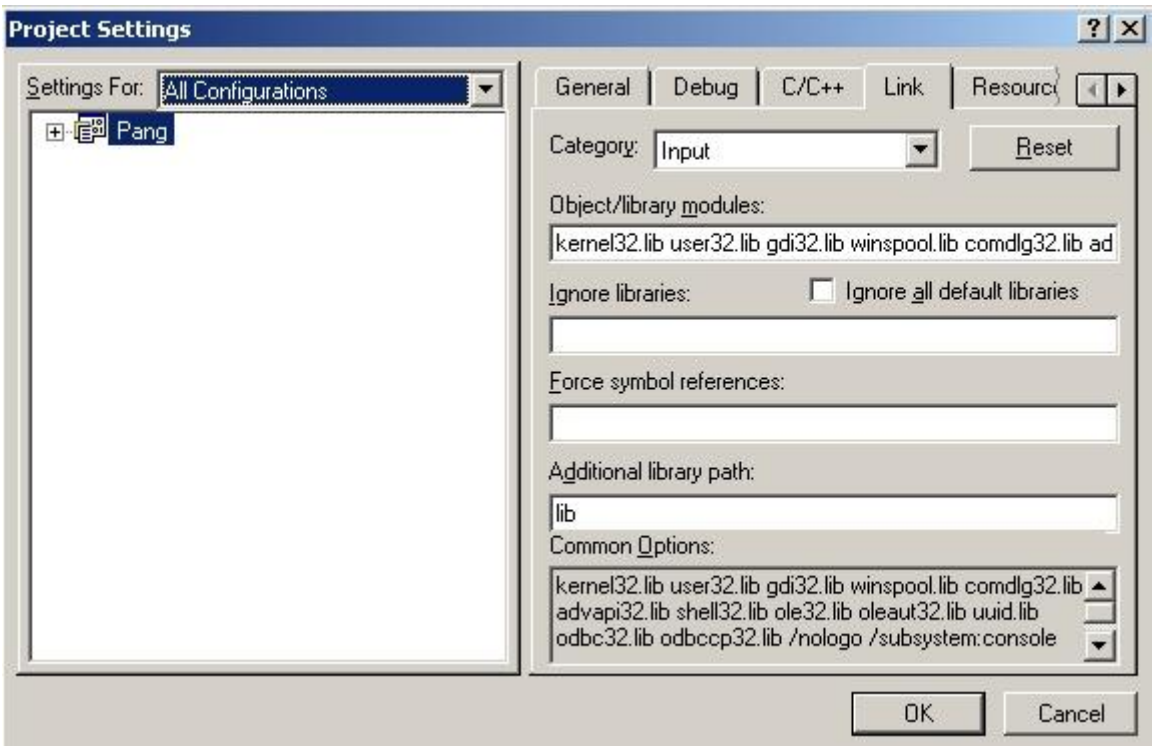


Figura C-5. Configuración del directorio de librerías.

La ubicación de las librerías estáticas del proyecto se puede especificar en la misma pestaña "*Link*", en la categoría "*input*", donde añadimos como ruta adicional de librerías la carpeta "*lib*".

En este caso no ha sido necesario especificar las librerías con las que se debe enlazar en la casilla "*Object/library modules*", en las que en la mayoría de los casos se debe escribir el nombre concreto de las librerías, que en este caso serían ***opengl32.lib, glu32.lib y glut32.lib***. Lo que pasa es que el fichero ***glut.h***, contiene unas directivas del preprocesador que sirven para indicar que se debe enlazar con esos ficheros. Esas directivas son:

```
#pragma comment (lib, "opengl32.lib")  /* link with OpenGL lib */
#pragma comment (lib, "glu32.lib")  /* link with OpenGL Utility lib */
#pragma comment (lib, "glut32.lib") /* link with Win32 GLUT lib */
```