UNIVERSIDAD NACIONAL MAYOR DE SAN MARCOS

FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

Escuela Académico Profesional de Ingeniería de Sistemas

Aplicación MFC para Windows: Creación de un Protector de Pantalla para la UNMSM

Tesis para optar el Título Profesional de
INGENIERO DE SISTEMAS

PATRICIA ISABEL ESPICHAN BERETTA

Lima – Perú
2002

Agradecimientos

En primer lugar quisiera dedicar este trabajo a mi madre, que me enseñó que cada cosa que uno inicia en la vida debe terminarla.
A mi padre, que le daba tanta importancia a los estudios, y a quién heredé el gusto por la ciencia.
A mis hermanos, por involucrarse conmigo en cada proyecto loco que empiezo.
A todos mis amigos, que me apoyaron, me comprendieron y me dieron aliento para culminar este trabajo.
Y finalmente quiero dar gracias a Dios, por haberme permitido concluir el presente trabajo.

INDICE

INTRODUCCIÓN

Actualmente, ante el auge que tienen los medios de comunicación y el uso avanzado de la tecnología en el mercado, las empresas necesitan ser más competitivas y promocionarse aún más a través de su imagen y la elaboración de sus productos. Hoy en día no sólo las grandes casas americanas de software, sino las grandes empresas multinacionales promocionan su imagen a través de los diversos accesorios que se utilizan para las computadoras como almanaques, pad's para los mouses, protectores de pantalla (a través de disquetes, Cd's o de la Internet), etc.

Es a raíz del éxito que se ha tenido en estos últimos productos y de cómo son elaborados es que nació mi inquietud de cómo sería la construcción de cada uno de estos atractivos aplicativos de software y de cuáles son los elementos que los vuelven exitosos, es que me doy con la sorpresa acerca de la casi inexistente documentación sobre los mismos. En Internet, tenemos una cantidad enorme de estos aplicativos a disposición entera del público, con motivos sobre casi todos los temas y todos los personajes, pero en realidad casi nadie nos dice cómo construirlos. Lo que más llamó mi atención, fue descubrir que en este país de personajes tan creativos y con gran diversidad de empresas, ninguna de ellas poseía un protector propio de pantalla con el logo distintivo de su negocio.

Es por eso que este trabajo está destinado a construir un Protector de Pantalla para la Universidad Nacional Mayor de San Marcos, contribuyendo a que alguna empresa o a algún investigador interesado en este tipo de productos, se anime a desarrollar su propio protector de pantalla, mejorando aún más sus resultados.

1. FUNDAMENTOS TEÓRICOS

PROTECTORES DE PANTALLA

La Interfase de programación de aplicaciones (API) Microsoft Win32 soporta aplicaciones especiales llamadas Protectores de Pantalla. Los protectores de pantalla empiezan cuando el ratón y el teclado han estado ociosos por un período específico de tiempo. Ellos son usados por tres razones :

- Para proteger una pantalla de una quemadura del fósforo causada por imágenes estáticas.
- Para ocultar información delicada dejada sobre una pantalla.
- Como herramienta de Marketing para las empresas.

Acerca de los protectores de pantalla

La aplicación de Escritorio del Control de Panel del Microsoft Windows permite utilizar una lista de protectores de pantalla. Especifica cuanto tiempo debería pasar antes que el Protector de Pantalla empiece. Configura los protectores de pantalla y permite una vista previa de los mismos. Los protectores de Pantalla son cargados automáticamente cuando Windows empieza o cuando el usuario activa el protector de pantalla a través del panel de Control.

Una vez que el protector de pantalla es elegido, Windows monitorea los golpes de teclado y los movimientos del ratón y empieza el protector de pantalla después de un período de inactividad. Sin embargo, Windows no empieza un protector de Pantalla si cualquiera de las siguientes condiciones existen:

- La aplicación activa no es una aplicación basada en Windows
- Una ventana de entrenamiento está presente
- La aplicación activa recibe el mensaje WM_SYSCOMMAND con el parámetro wParam puesto al valor SC_SCREENSAVER, pero este no pasa el mensaje a la función DefWindowProc.

Los Protectores de Pantalla contienen funciones exportadas específicas, definiciones de recursos y declaraciones de variable. La librería de protector de pantalla contiene la función principal y el código requerido para empezar un protector de pantalla. Cuando un protector de pantalla empieza, el código de inicio en la librería del Protector de pantalla crea una ventana de toda la pantalla. La clase windows para esta ventana es declarada como sigue :

```
WNDCLASS cls;
cls.hCursor        = NULL;
cls.hIcon          = LoadIcon(hInst, MAKEINTATOM(ID_APP));
cls.lpszMenuName   = NULL;
cls.lpszClassName  = “WindowsScreenSaversClass”;
cls.hbrBackground  = GetStockObject(BLACK_BRUSH);
cls.hInstance      = hInst;
cls.style          = CS_VREDRAW | CS_HREDRAW | CS_SAVERBITS | CS_DBLCLKS;
cls.lpfnWndProc    = (WNDPROC) ScreenSaverProc;
cls.cbWndExtra     = 0;
cls.cbClsExtra     = 0;
```

Para crear un protector de pantalla, la mayoría de los desarrolladores crean un módulo de código fuente contendiendo 3 funciones requeridas y las enlazan con la librería de protector de pantalla. Un módulo de protector de pantalla es responsable solo por configurarse a sí mismo y para proveer efectos visuales.

Una de las 3 funciones requeridas en un módulo de Protector de pantalla está contenida dentro del cuerpo principal del programa. Esta función procesa específicos mensajes y pasa cualquier mensaje no procesable de regreso a la librería de Protector de Pantalla.

La segunda función requerida en un módulo de protector de pantalla es una pantalla de Configuración. Esta función debe mostrar una caja de diálogo que permite al usuario configurar el protector de pantalla. Windows muestra la caja de diálogo de configuración cuando el usuario selecciona el botón configurar

en la caja de diálogo de protector de pantalla de la Pantalla de Panel de Control.

La tercera función requerida en un módulo de protector de pantalla es guardar las especificaciones del cuadro de diálogo en el Registry del Windows. Esta función debe ser llamada por la aplicación de Protector de Pantalla. Sin embargo las aplicaciones que no requieren ventanas especiales o controles personalizados en la caja de dialogo de configuración pueden simplemente retornar TRUE. Las aplicaciones que requieren ventanas especiales o controles personalizados deberían usar esta función para registrarlas en el Windows.

En adición a crear un módulo que soporta esas tres funciones, un protector de pantalla debería suplir un ícono. Este ícono es visible solo cuando el protector de pantalla esté corriendo como una aplicación monousuario. El ícono debe ser identificado en el archivo de recursos del protector de pantalla.

Adicionalmente el Programa de Protector de Pantalla debe considerar la recepción de un parámetro de entrada que puede tomar cualquiera de estos tres valores que son enviados por el Windows desde el Panel de Control al momento de seleccionar el Protector de Pantalla:

-s Envío en modo de Protector de Pantalla (pantalla completa).

-p Envío en modo de Vista Previa.

-c Envío en modo para configuración.

Dependiendo de la versión de Windows, el envío de parámetros puede ser con los símbolos “-“ ó “/”.

Importancia:

Un protector de pantalla puede dejar de ser un elemento netamente decorativo en un computador, puede significar también una herramienta de marketing y publicidad para una empresa. Puede convertirse en una manera de difundir su imagen y hacerse más cercana a sus clientes.

Objetivo General:

Aprovechar e investigar las librerías MCF (Microsoft Clases Foundation) que se proporcionan con las herramientas de desarrollo Microsoft y que constituyen

una manera de desarrollar aplicaciones Windows de manera más simple y rápida. Además de ser portables a otros sistemas operativos como el UNIX.

Objetivo específico:

Proporcionar un código base que permita a futuros desarrolladores crear un Protector de Pantalla a partir del código fuente proporcionado en este trabajo.

Alcance:

Con este trabajo pretendo cubrir las áreas de programación orientada a objetos, aplicaciones Windows y librerías MCF, desarrollando un producto que interactúa directamente con el Sistema Windows.

2. CONCEPTOS DE ORIENTACION A OBJETOS

La programación orientada a objetos (OOP) es una forma de programación que utiliza objetos, ligados mediante mensajes, para la solución de problemas. Los mecanismos básicos de la programación orientada a objetos son: objetos, mensajes, métodos y clases.

Objetos

Un *objeto* es una entidad que tiene unos atributos particulares, los *datos*, y unas formas de operar sobre ellos, los *métodos* o *procedimientos.* Por lo tanto, un objeto contiene, por una parte, operaciones que definen su comportamiento, y por otra, variable manipuladas por esas operaciones que definen su estado. Por ejemplo, una ventana Windows contiene operaciones como *maximizar* y variables como *ancho* y *alto* de la ventana.

Mensajes

Cuando se ejecuta un programa orientado a objetos, los objetos, están recibiendo, interpretando y respondiente a *mensajes* de otros objetos. Por ejemplo, cuando hacemos clic en el botón de maximizar de una ventana, la ventana recibe un mensaje de notificación de que tiene que maximizarse. Cuando un objeto recibe un mensaje, debe conocer perfectamente lo que tiene que hacer, y cuando un objeto envía un mensaje, no necesita conocer cómo se desarrolla, sino simplemente que se está desarrollando.

En C++, *un mensaje* está asociado con el prototipo de una función miembro de tal forma que cuando se produce ese mensaje se ejecuta la correspondiente función miembro de la clase a la que pertenece el objeto. Esto es, el envío de un mensaje equivale en C++ a llamar a un función miembro.

Métodos

Un *método* (que en C++ se denomina *función miembro*) se implementa en una clase, y determina cómo tiene que actuar el objeto cuando recibe el *mensaje* asociado. En C++, un *método* se corresponde con la definición de la función miembro de una *clase*.

La estructura más interna de un *objeto* está oculta para otros usuarios y la única conexión que tiene con el exterior son los *mensajes*. Los datos que están dentro de un objeto, solamente pueden ser manipulados por los métodos asociados al propio objeto.

Clases

Una *clase* es un tipo de objetos definido por el usuario. Una *clase* equivale a la generalización de un tipo específico de objetos. Un *ejemplar* es la concreción de una clase (algunos autores utilizan el término *instancia*, traducción directa de instance). Por ejemplo, un *ejemplar* de la clase *C* sería un objeto *O* de esa clase.

Las *variables de la clase* tiene almacenados valores que son compartidos por todos los objetos de esa clase (en C++ se denominan *datos miembro static*), y cada objeto de una clase tiene sus propios valores, almacenados en las *variables asociada a cada objeto* de la clase en (en C++ se denomina *datos miembro*).

Un *objeto* de una determinada clase e crea en el momento que se declare una variable de dicha clase. Por ejemplo, la siguiente declaración en C++ crea un *objeto* de a clase *párrafo*: el objeto *parrafol*.

parrafo parrafol;
y para justificar el texto de ese párrafo, llamaremos a la función *justifi* de la forma siguiente:
parrafo1.justifi()

el prototipo de la función *justifi()* corresponde al mensaje, y la definición, al método.

Clases derivadas

Entre las características de la OOP, hay una que destaca: la *herencia*, ya que permite definir clases derivadas de otras clases ya existentes. Una *clase derivada* provee el mecanismo más simple para especificar una forma alternativa de acceso a una clase existente, o bien para definir una nueva clase que añada características a una clase existente. Esta nueva clase, denominada *clase derivada*, hereda todas las propiedades de la clase existente a la que nos referimos y que recibe la denominación de *clase base*. De una forma genérica, se puede decir que una clase derivada hereda los datos y funciones miembro de su clase base.

Por lo tanto una clase derivada pone en nuestras manos un mecanismo que evita tener que rescribir de nuevo un clase existente cuando se necesita ampliarla en cualquier sentido; esto es, la *herencia*, entre otras cosas permite la *reutilización del código*.

Constructor

Un *constructor* es una función miembro especial de una clase que es llamada automáticamente siempre que se declara un objeto de esa clase. Dado que los constructores son funciones miembro, admiten argumentos igual que éstas. Cuando en una clase no especificamos un constructor, el compilador crea uno por defecto sin argumentos.

Un *constructor* se distingue fácilmente porque tiene el *mismo nombre* que la clase a la que pertenece. Un *constructor* no es heredado, no puede retornar un valor (incluyendo void) y no puede ser declarado virtual o static.

Cuando una clase base tiene un *constructor* y una clase derivada también, al crear un objeto se llama primero el constructor de la clase, y cuando la ejecución de éste finaliza, entonces se llama al constructor de la clase derivada. Una clase derivada contiene todos los miembros de la clase base, y todos tienen que ser inicializados. Para ello, el constructor de la clase derivada tiene que llamar al constructor de la clase base.

Destructor

Un *destructor* es una función miembro especial de una clase que se utiliza para eliminar un objeto de esa clase. Liberándose la memoria que ocupa. Un *destructor* se distingue fácilmente porque tiene el mismo nombre que la clase a la que pertenece precedido por una tilde ~. Un *destructor* no es heredado, no tiene argumentos, no puede retornar un valor (incluyendo void) y no puede ser declarado static, pero sí puede ser declarado virtual. Utilizando destructores virtuales se puede destruir objetos sin conocer su tipo. Cuando en una clase no se especifica un destructor, el compilador crea uno por defecto.

Características de la OOP

Las características fundamentales de la OOP son: *abstracción, encapsulamiento, herencia y polimorfismo.*

Abstracción

Por medio de la abstracción conseguimos no detenernos en los detalles concretos de las cosas que no intereses en cada momento, sino generalizar y centrarse en los aspectos que permitan tener una visión global del tema. Precisamente la clave de la programación orientada a objetos está en abstraer los métodos y los datos comunes a un conjunto de objetos y almacenarlos en una *clase*. Desde este nivel de abstracción,, la introducción o eliminación de un objeto es una determinada aplicación supondrá un trabajo mínimo o nulo.

Encapsulamiento

Esta característica permite ver un objeto como una caja negra, en la que se ha metido de alguna manera toda la información relacionada con dicho objeto. Esto nos permitirá manipular los objetos como unidades básicas, permaneciendo oculta su estructura interna.

En C++ la abstracción y la encapsulación están representadas por la *clase* es una abstracción, porque en ella se definen las propiedades y los atributos genéricos de un determinado conjunto de objetos con características comunes y es una encapsulación, porque constituyen una caja negra que encierra tanto los datos de que constan los objetos como los métodos que permiten manipularlos.

Herencia

La herencia es el mecanismo para compartir automáticamente métodos y datos entre clases, subclases y objetos. Por ejemplo, si declaramos la clase *párrafo* como una subclase de *texto*, todos los métodos y variables asociadas con la clase *texto* son automáticamente heredados por la subclase *párrafo*. Si la clase *texto* contiene métodos inapropiados para la subclase *párrafo*, éstos pueden anularse, escribiendo nuevos métodos y almacenándolos como parte de la clase *párrafo*, o también pueden redefinirse para que respondan de forma distinta a como lo hacen en la clase base.

En C++, herencia equivale a derivación de clases; la clase padre (*texto*) recibe el nombre de *clase base* y la clase hija (*párrafo*) se denomina *clase derivada*.

Esta característica está fuertemente ligada a la reutilización del código en la OOP. Esto es, el código de cualquiera de las clases existentes pueden ser utilizado sin más que crear una clase derivada de ella.

El concepto de herencia, conduce a una estructura jerarquizada de clases o estructura en árbol, lo cual no significa que en OOP todas las relaciones entre clases deban ajustarse a una estructura jerárquica.

La herencia puede ser también múltiple; esto es, una clase puede derivarse de dos o más clases base.

Poliformismo

Esta característica permite implementar múltiples formas de un mismo método, dependiente cada una de ellas de la clase sobre la que se realice la implementación. Esto hace que se pueda acceder a una variedad de métodos distintos (todos con el mismo nombre) utilizando exactamente el mismo medio de acceso.

En C++ el poliformismo se establece generalmente utilizando clases derivadas, funciones virtuales y punteros a objetos.

Funciones virtuales

Una función declarada virtual en una base y redefinida en cada una de las clases derivadas puede ser accedida mediante un puntero o referencia a la clase base. Eso permite que un mismo mensaje pueda ser enviado a objetos de diferentes clases, de forma que el objeto que responda al mensaje será en cada caso el referenciado por dicho puntero.

3. PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos (POO) se puede considerar como una extensión de las técnicas de programación estructurada. La *clase* es un elemento nuevo de modularidad, añadido al archivo y la función. POO es un estilo de programación en el que se modela objetos del mundo real, formas, estéreos, ventanas, documentos, con algunos objetos de software. Es posible hacer POO, incluso en los lenguajes de programación convencional como C, pero C++ proporciona soporte directo para la POO y lo hace más fácil y más natural de pensar y codificar en términos de objetos.

Los lenguajes de POO como C++ son extensamente ampliables. Además de los tipos de datos internos, *int, floats, char,* C++ posibilita la extensión del sistema de tipo de forma indefinida, especialmente cuando considera la posibilidad de cargar los operadores de C++. Cada clase nueva que cree, si es una clase base o clase derivada, es un tipo de datos nuevo, una clase derivada hereda los datos y comportamiento de su clase base, aún así también puede omitir y ampliar la funcionalidad de su clase base. Esto viene a ser la extensión de tipos de datos derivando tipos nuevos desde ellos.

La posibilidad de almacenar un puntero a la clase *B* en una variable de tipo *A** (donde *A* es una clase base de *B*) simplifica mucho el trabajo con colecciones de objetos en la misma jerarquía de clase. Se puede, almacenar CspRectangles y CSEllipses, y más tarde CshpTriangles, en una matriz de Cshapes. Y, utilizando un mecanismo de función virtual, se puede invocar la función miembro Draw del objeto almacenado ahí, este polimorfismo permite que los objetos funcionen juntos mientras que cada uno se comporta de forma propia. Se puede utilizar la POO en mayor o menor medida. MFC, por ejemplo, modela una aplicación de Windows y utiliza los objetos de clase de forma amplia para representar las ventanas, documentos, cuadros de diálogos, botones e incluso la aplicación en su totalidad.

El proceso de diseño orientado a objetos empieza normalmente con una especificación cuidadosa del problema. El paso siguiente es estudiar el problema para los elementos que puede ser buenos objetos. Conforme el diseño avanza, gradualmente desarrollo la estructura y comportamiento de estos objetos. Con el tiempo, implementa los objetos como clases C++ en una jerarquía.

MFC

Cuando el equipo Application Franework (AFX) en Microsoft empezó su trabajo en un marco de trabajo de aplicación orientado a objetos (un conjunto de clases que proporcionan el marco de trabajo de un programa utilizando las técnicas de POO), estaba muy orientado a objetos. Incluso los elementos de datos simples como *ints y chars* se alojaron en objetos.

Los desarrolladores de AFX pronto se dieron cuenta, sin embargo, de que se habían pasado de la raya. Su segundo marco de trabajo de la aplicación, desarrollado en un principio después de abandonar el trabajo de un año, se convirtió en la biblioteca MFC. Uno de los distintivos de MFC era que estaba lo suficientemente orientado a objetos, y no un poco más. El equipo AFX podría haber escrito un enfoque orientado a objetos en el que cada elemento de Windws estaría representado por un objeto. Por el contrario, eligieron sabiamente no volver a escribir Windows partiendo de cero, ni volver a hacer POO.

La biblioteca MFC que resultó representa unos cuantos conceptos primarios de Windows, ventanas, contextos de dispositivos y otros cuantos, como objetos, pero invoca las funciones API de Windows originales dentro de esos objetos. Por ejemplo, la clase *CWnd* representa una ventana. Su función miembro CWnd:ShowWindows, por ejemplo, invoca la función API ShowWindows en el sistema operativo Windows.

Mientras tanto, los desarrolladores de AFX se acostumbraron a llamarse a í mismo “OOPaholics reformado”. Habían vuelto de la excesiva POO para unirse a una filosofía que ponía la programación orientada a objetos en su sitio como una de las muchas herramientas de software.

Ésta es una filosofía que merece la pena considerar, es por eso el motivo de su utilización en el presente trabajo.

Distribución de aplicaciones que utilizan MFC

Si se vincula dinámicamente la aplicación con la biblioteca MFC, se deberá, como mínimo, redistribuir los archivos MFC71.dll y MSVCR71.dll. Todas las DLL de MFC utilizan la versión compartida de la biblioteca de tiempo de ejecución de C (CRT), por lo que se requiere MSVCR71.dll. Igualmente, deberá comprobar que el equipo de destino para una aplicación basada en MFC71.dll posea al menos la versión 4.0 de Internet Explorer, ya que MFC 7.1 utiliza los componentes incluidos en este último.

No es necesario redistribuir MFC71.dll con las aplicaciones MFC si se vinculan estáticamente con la DLL de MFC (es decir, a menos que haya especificado Utilizar MFC en una biblioteca estática en la ficha General del cuadro de diálogo Configuración del proyecto).

Si una aplicación utiliza las clases de base de datos de MFC, como CRecordSet y CRecordView, deberá redistribuir ODBC y cualquier controlador ODBC que utilice la aplicación.

Si redistribuye una DLL de MFC, asegúrese de redistribuir la versión comercial en lugar de la de depuración. Las versiones de depuración de las DLL no son redistribuibles. Las versiones de depuración de los archivos DLL de MFC tienen la letra "d" al final del nombre de archivo, como en MFC71d.dll.

Si modifica MFC de alguna forma, deberá cambiar el nombre de la DLL MFC modificada para que no entre en conflicto con la DLL MFC que otras aplicaciones MFC pueden haber instalado en el equipo de destino. No se recomienda volver a generar y cambiar el nombre de la DLL MFC.

Windows y la API Win32

Microsoft Visual C++ permite crear muchos tipos de programas. Pero el tipo principal, en el que se centrará este trabajo es la aplicación de un Protector de Pantalla escrita en lenguaje Visual C++ para Microsoft Windows, utilizando la Microsoft Foundation Class Library (MFC).

Las versiones de windows

Windows 95, Windows 98, Windows XP y Windows NT son todos variaciones de 32 bit del sistema operativo Microsoft Windows para los ordenadores personales Windows 3.1 y su sucesor, Windows 3.11, son versiones de 16 bit, el resto del mundo ha progresado. Actualmente encontramos sistemas operativos de 64 bits.

Los datos de un programa de computadora se basan fundamentalmente en el tamaño de una palabra de almacenamiento. El tamaño de una palabra determina cuanta memoria puede direccionar el sistema. Los sistemas basados en 32 bits tienen un tamaño de palabra de 32 bit que es doble del mundo de 16 bit, que utiliza Windows 3.1. Esto significa que los sistemas de 32 bit pueden direccionar mucha más memoria. En realidad, en los sistemas de 32 bit, la cantidad de memoria direccionable crece de forma exponencial a más de 4 GB. Además, al contrario que en los sistemas de 16 bit, en los que las direcciones de memoria se dividen en segmentos que resultan en código complicado, la memoria de 32 bit es plana. No hay segmentos, simplemente un gran especio

de direccionamiento. Esto no sólo facilita la programación, sino que hace programas más potentes.

API de Windows

Windows 95 y Windows 98 son bastante diferentes a Windows NT, pero todas estas versiones de 32 bit de Windows están descritas “en el mismo lenguaje”, por decirlo así. Todas se basan en la Application Programming, conocida como la API Win. Hay una versión de 16 bit de API Win, pero Windows 95, Windows 98 y Windows NT utilizan todos la versión de 32 bit, llamada API Wind32, o simplemente Wind32. (Para más claridad, la versión de 16 bit se llama a menudo API Wind16, o simplemente, Wind16). Este trabajo se basa en la Api Win32.

Win32, el SDK y Windows.h

La API es una colección de varios cientos de funciones, más un montón de constantes, macros, struct, tipos y otros elementos. Estos elementos de programación están descritos en el lenguaje de programación C, pero hoy en día puede invocar las funciones y utilizar otros elementos desde C++, Microsoft Visual Basic, lenguaje ensamblador, Fortran, Pascal y otros lenguajes de programación.

La mayor parte de la API está definida en un archivo llamado Windows.h. este archivo se incluye con la mayor parte de los entornos de programación para Windows, incluyendo Visual C++. Aunque también es posible obtenerla con Platdorm Software Development Kit (SDK) de Windows.

Conceptos clave de Windows

Las funciones Win32 API construidas en torno a un conjunto de conceptos subyacentes , incluyendo un entorno gráfico, múltiples ventanas que se superponen, menús, iconos, mensajes, archivos, recursos, multitarea y uso del ratón.

El concepto unificador, es la idea de una ventana. Muchas funciones de la API operan en ventanas, creándolas, dándoles tamaños, moviéndolas, etcétera. Otras funciones de la API son para dibujar en ventanas; abrir, leer, escribir, y cerrar archivos, comunicarse con el sistema operativo u otros programas; y mucho más.

Programación para diferentes plataformas Win32

El basar los diferentes sistemas operativos de 32 bit en una sola API significa que pueden programar todas las variantes de Windows de 32 bit de modo muy parecido. Aunque es verdad que la API contiene algunas funciones que pueden utilizar sólo con Windows NT y otras que sólo puede utilizarse con Windows 95 y 98, el corazón de la API es idéntico en todas estas plataformas. Así, que programando para Windows 95, se puede también programar para Windows 98, Windows NT, especialmente si se utiliza MFC.

Multitarea y multihilo

Windows, en particular en sus versiones de 32 bit, es un sistema multitarea. Un sistema operativo multitarea puede ejecutar programas al mismo tiempo.. Como en Windows 95, la multitarea en Windows es *preemptiva*. (Las versiones anteriores de Windows utilizaban un enfoque no preemptivo más simple y menos eficaz en el que los programas tenían que cooperar de forma explícita para compartir el procesador). En un entorno de multitarea preventivo, el sistema operativo reparte pequeñas porciones de tiempo a cada aplicación (o proceso) que se ejecuta. Como las porciones son lo suficientemente pequeñas, al usuario le da la impresión de que se ejecutan varios programas simultáneamente. En realidad, cada una “duerme” un breve espacio de tiempo hasta que le llega su turno otra vez.

Los sistemas Windows modernos también permiten el *multihilo* dentro de una aplicación, un concepto relacionado con la multitarea. La mayor parte de los programas tienen un solo hilo de ejecución. Es decir, hay un camino a través del código, y cada vez se ejecutan una sentencia conforme van apareciendo.

Pero un programa puede dividirse en dos o más hilos separados de ejecución que, como en la multitarea, parecen ejecutarse simultáneamente. En este sentido, un programa puede derivar hilos trabajadores que se encarguen de tareas independientes, como la impresión en segundo plano, mientras que el usuario continúa trabajando en el hilo principal.

La biblioteca MFC facilita en algo la tarea para poder trabajar con multitarea y multihilo.

Fundamentos De Programar En Windows

Sucesos

Microsoft Windows es un sistema *controlado por el usuario*, lo que significa que pasa gran parte del tiempo esperando que el usuario haga para que pueda responder. Tales sistemas se llaman también *controlados por sucesos*. Cuando el usuario pulsa una tecla,. Mueve el ratón o hace clic en un botón del ratón. El hardware de la máquina permite a Windows saber que ha ocurrido un suceso, qué tipo de suceso era, cuándo tuvo lugar y dónde ocurrió en relación con la pantalla (en un conjunto particular de coordenadas dentro de una ventana de aplicación particular, por ejemplo).

Lo sucesos que genera de una de estas tres formas. La primera es a través de dispositivos de entrada, como por ejemplo el teclado y el ratón. La segunda es a través de objetos visuales de la pantalla, como los menús, los botones de barra de herramienta, las barras de desplazamiento y los controles en los cuadros de diálogo. La tercera posibilidad procede del interior del propio Windows, como cuando se hace visible repentinamente una ventana que se había ocultado por otra ventana.

Mensajes

Un mensaje es una notificación a la aplicación de que ha sucedido algo y que por lo tanto debe realizarse alguna acción específica.

Los mensajes de Windows son constante definidas con macros en el archivo Windows.h (o en uno de los archivos que incluyen Windows.h). los nombres de constante de mensajes tiene la forma *WM_XXX;* por ejemplo, *WM_QUIT*, *WM_CHAR, WM_LBUTTONDOWN, WM_COMMAND.* el mensaje se encamina al programa adecuado basándose en la siguiente información:

Qué aplicación está actualmente activa

Qué ventana en esa aplicación está actualmente activa.

Dónde estaba el cursor en el suceso.

Windows coloca el mensaje dentro de una cola de mensajes de aplicaciones de destino, donde espera con cualquier otro mensaje pendiente hasta que la aplicación esté preparada para recuperarlo y procesarlo. En los sistemas Windows de 32 bit, cada aplicación tiene una cola de mensajes propia.

El bucle de mensajes:

Dentro de la aplicación, hay un *bucle de mensaje*. En el código, este bucle aparece del modo siguiente:

```
while (GetMessage (&msg));
{
translatemessage (&msg)
dispatchMessage (&msg)
        }
```

Este pequeño bucle continúa operando siempre que el mensaje que recupera de la cola de mensajes de aplicación no sea *WM_QUIT*. Este mensaje hace que el bucle termine porque *GetMessage* devuelve falso cuando se encuentra con *WM_QUIT,* y la aplicación termina. Pero mientras que el bucle continúe,

invoca la función Win32 API *GetMessage* para recuperar el mensaje. Si no hay mensaje en la cola. *GetMessage* se detiene ahí y espera uno. (esto, simplemente, significa que el usuario no está haciendo nada que el programa necesita saber.)

Cuando *GetMessage* no devuelve un mensaje, el bucle lo devuelve a la función *TranslateMessage* para ver si es un mensaje procedente del teclado que necesita un poco de trabajo extra. *TranslateMessage* convierte los mensajes de teclado en bruto en mensajes *WM_CHAR* diseñado para aportar información fácil de utilizar sobre el carácter que se ha tecleado.

TranslateMessage separa los comandos entregados por el teclado, como las combinaciones CTRL.+X, de los caracteres alfanuméricos tecleados y los símbolos imprimibles. *TranslateMessage* no hace nada con los mensajes que no sean de teclado.

Finalmente, la función *DispatchMessage* determina qué ventana en la aplicación (si hay más de una) debería recuperar el mensaje y mandar el mensaje. A continuación el bucle se inicia otra vez.

Manejadores de mensaje

Lo que le sucede a un mensaje de entrada dentro de la aplicación que lo recibe depende de lo el programador de aplicación, haya hecho. Hay dos posibilidades principales.

Se escribió un manejador, algún código que maneje un mensaje particular. Si existe un manejador para el mensaje, el código de manejador se ejecuta para procesar el mensaje.

No se escribió un manejador para ser mensaje. En su lugar, optó por devolver ese mensaje a Windows para cualquier que fuese el procesamiento predeterminado que Windows necesite hacer. En la programación tradicional

de Windows, esto se hace con una invocación a la función *DefWindowsProc* (“procedimiento de ventana predeterminado”).

De cualquier modo, alguien hace algo con el mensaje, ya sea la aplicación del manejador que ha proporcionado, o Windows.

Lo que sucede en el manejador para un mensaje depende de lo que necesite realizar. Pero hay muchas cosas típicas o estándar por hacer para los mensajes particulares. Por ejemplo, un manejador *WM_PAINT*, enviado cuando su ventana necesita volver a dibujar el contenido, lleva a cabo pasos para volver a dibujar las líneas visibles de texto, o los rectángulos y elipses que el usuario ha trazado, etc. un mensaje para el que escribirá manejadores frecuentemente es *WM_COMAND*, que utiliza para procesar comandos de menús y botones. Los manejadores *WK_COMMAND*, a menudo llamados, simplemente, *manejadores de comandos*, yacen en la parte más profunda de sus interacciones con el usuario. Un manejador de comandos puede visualizar un cuadro de diálogo, llevar a cabo un cálculo, seleccionar una opción o iniciar alguna otra acción.

Formas de invocación

Algunas veces el desarrollador puede ser el responsable de invocar los mensajes y en otras ocasiones lo hace directamente el Windows.
Es una situación que puede intercambiarse continuamente. La mayor parte del tiempo, el código está ahí esperando a ser invocado, mientras que Windows está por su parte realizandootras acciones propias del Sistema Operativo.

Windows es autocontenido. Si no se escribe un manejador para uno de los cientos de mensajes de Windows, Windows no es afectado. Si no se proporciona un manejador, Windows proporciona uno, llamado *DefWindowProc.* Pero si se proporciona un manejador. Windows invoca al desarrollado.

Dentro de su manejador, el desarrollador es responsable por las acciones a realizar. Es aquí donde se puede aprovechar las ventajas de muchos servicios que proporciona la API Win32, en la práctica, rara vez se proporcionará manejadores para más de unos cuantos de los cientos posibles de mensajes Windows.

En los programas Windows escritos en C (o en C++ sin MFC), para que Windows sepa que funciones invocar, se especifica el nombre del windproc (procedimiento de ventana) al sistema operativo al principio del programa. A continuación, se escribe una función con ese nombre. En el lenguaje de programación de Windows un modo de volver a llamarle para invocar su procedimiento. En el procedimiento de ventana, utiliza una sentencia swich de C/C++ para proporcionar código de manejador para los mensajes Windows que espera recibir y procedimiento de su ventana. En los programas MFC, no es necesario especificar el procedimiento de ventana, en su lugar, se escribe las funciones de manejador que MFC invoca por medio de un mapa de mensaje, cada vez que recibe un mensaje para su clase ó ventana, se ejecuta la función del manejador proporcionada.

Dibujar

Como el sistema operativo de Apple Macintosh y varios otros sistemas, Microsoft Windows tiene una *interfaz gráfica de usuario* (GUI). En lugar de interactuar sólo con un programa tecleando texto en respuesta a las solicitudes, el enfoque antiguo, como en las aplicaciones de consola que hemos estado escribiendo, por ejemplo, el usuario de Windows interactúa con le programa a través de una visualización visual. La visualización incluye elementos como menús, barras de herramientas, barras de desplazamiento y botones en cuadros de diálogo. Aunque es posible manejar la mayor parte de programa Windows con el teclado, la mayor parte están diseñados para que se manejen desplazando y haciendo clic en el ratón. Por ejemplo, el usuario hace clic en un título de menú. Provocando o elección del menú se despliegue. A continuación el usuario hace clic en un comando elección del menú.

Todos los menús, botones y otros controles que el usuario ve en la pantalla se dibujan ahí realmente. Un botón en un cuadro de diálogo, por ejemplo, es sólo un mapa de bits, una colección de pixeles en la que algunos están encendidos y otros están apagados. Es sólo una imagen aunque pueda parecer muy tridimensional. Por lo tanto se debe dibujar, mediante Windows o mediante el código que escriba.

Ventanas hijas y ventanas poseídas

En Windows, cada uno de estos controles y otros objetos de interfaz de usuario es también una ventana. Por ejemplo, un cuadro de diálogo es una ventana, y cada uno de los botones, cuadros de edición y otros controles dentro del cuadro de diálogo es una ventana, una ventana hija de alguna ventana padre. El padre de un control de cuadro de diálogo, por ejemplo, es la ventana de cuadro de diálogo. Las ventanas hijas están contenidas por las ventanas padre, la hija está completamente dentro de la ventana padre.

Una ventana también puede “poseer” otras ventanas, incluso si las ventanas no están completamente dentro del espacio del propietario. Por ejemplo, una ventana de cuadro de diálogo es propiedad de alguna otra ventana, como, por ejemplo, la ventana de marco principal de la aplicación o una hija de esa ventana.

A través de mensajes, Windows maneja muchos de sus propios mensajes. El dibujo que hace, también llamado a modo de sinónimo *pintura*, tiene lugar en los dos puntos siguientes en relación a un programa:

Primero, cuando el programa visualiza inicialmente una ventana, necesita pintar el interior de la ventana, o área de cliente, por primera vez. El área de cliente es el espacio rodeado por los bordes de ventana, barra de título y otros elementos de marco. El área de cliente es casi siempre responsabilidad del desarrollador, mientras que el marco alrededor de la misma es responsabilidad de Windows.

Segundo, siempre que se “estropee” parte de la ventana, o se *anule*, senecesita volver a pintarla. Esto puede ocurrir, por ejemplo, cuando alguna otra ventana abarca su ventana, y a continuación desaparece de modo que su ventana es completamente visible otra vez. Windows no guarda una imagen de lo que había en la venta, así que no puede volver a trazar todo sí misma. Por lo tanto notifica que necesita volver a pintar el contenido de la venta.

Los mensajes WM_PAINT

En cualquier de los casos descritos, Windows se le notifica enviando un mensaje *WM_PAINT*. Casi siempre se tiene que escribir un manejador para *WM_PAINT*. Algunas veces Windows puede pasar información que permitae volver a dibujarlo todo otra vez.

Windows prioriza los mensajes que se muestran en la cola de su aplicación. Y así sucede que a *WM_PAINT* se le da normalmente una prioridad baja. De este modo, una ventana podría permanecer sin pintarse durante un tiempo. La operación de gran prioridad monopoliza temporalmente el tiempo de procesamiento y el redibujado tiene que esperar su turno. Esto es, porque Windows considera que pintar es menos crucial que las operaciones de volver a calcular una hoja de cálculo, por ejemplo. Tales operaciones se deben hacer primero, y una ventana que no está pintada es sólo un mal menor que se soluciona rápidamente.

El contexto de dispositivo

Windows es responsable de volver a pintar el marco, mientras que el desarrollador es responsable de volver a pintar el interior, el área de cliente. Según el área de cliente, hay un objeto de Windows conocidos como un *contexto de dispositivo* (a menudo abreviado como “DC”). Un contexto de dispositivos es realmente sólo una estructura de datos mantenida por Windows. Contiene información sobre para qué área (ventana) es, el color de fondo actual o modelo de área, qué partes del área están actualmente invalidadas

(siendo necesario volver a pintarlas), etc. en realidad, el contexto de dispositivos contiene varios objetos más pequeños, sin pincel, una pluma y una fuente, que utiliza para dibujar en el área de cliente. El objeto más importante del contexto de dispositivo es un mapa de bits, esto es una superficie lógica sobre la que dibujar, y es el mapa de bit que Windows visualiza en una pantalla, una impresora o algún otro dispositivo de salida.

Puede que sea más útil pensar en el contexto de dispositivo como si fuera su estudio de pintura. Contiene varias herramientas de pintura y de dibujo así como un lienzo sobre el que pintar (el mapa de bit). Se puede cambiar estas herramientas y los otros atributos del contexto de dispositivo y adaptarlas a sus necesidades. Por ejemplo, se puede invocar una función que cambia el color de texto del negro predeterminado a, por ejemplo, rojo. O se puede sustituir una fuente diferente por la predeterminada. Pero no se puede de ningún modo dibujar sin un contexto. Eso es importante. El cualquiera de los manejadores que dibujan, se tendrá que obtener un contexto de dispositivos de alguna manera. Algunas veces hay contexto de dispositivo ya definido, y todo solucionado. Pero algunas veces hay que obtenerlo en el mismo programa.

Manejadores

El modo de acceder a un contexto de dispositivo es a través de un manejador que Windows devuelve. El tipo de variable para un manejador a un contexto de dispositivo es *HDC* (definido en Windef.h). El manejador le concede ciertos derechos y posibilidades dentro de su ventana. Proporcionar un manejador a algún objeto o recurso que usted necesite es una actividad normal de Windows. En realidad, Windows proporciona manejadores a muchos tipos de objetos: ventanas, pinceles, fuentes, etc. Cada tipo de objeto tiene un tiempo de manejador asociados: *HWND, HBRUSH y HFONT*, la siguiente tabla muestra los objetos Windows más comunes para los que se puede obtener manejadores.

Objetos de ventana	Manejador asociado
Tabla aceleradora	*HACCEL*
Mapa de bit	*HBITMAP*
Pincel	*HBRUSH*
Cursor	*HCURSOR*
Contexto de Dispositivo	*HDC*
Archivo	*HFILE*
Fuente	*HFONT*
Icono	*HICON*
Menú	*HMENU*
Paleta	*HPALETA*
Pluma	*HPEN*
Región	*HRGN*
Ventana	*HWND*

Funciones API Wind32

Teniendo un HDC (*manejador para un contexto de dispositivo*), sepuede invocar alrededor de unas cien funciones API Win32 para trazar en el área de su ventana.

La parte de Windows responsable de los contextos de dispositivos y de las funciones de dibujo se conoce como la Interfaz de dispositivos de gráficos (GDI). GDI es también un sistema de dibujo bidimensional. Dispone el contexto de dispositivos, las funciones de dibujo y varios sistemas de coordenadas para medir y localizar las imágenes que ha dibujado con él.

Ejemplos:

FunciónWin32	Lo que hace
TexOut	Dibuja una cadena de texto en un lugar específico en el área de cliente.
Rectangle	Dibuja un rectángulo en coordenadas

	específicas.
Ellipse	Dibuja una elipse en coordenadas específica.
GetBkColor	Obtiene el color de fondo actual.
MoveToEx (MoveTo en MFC)	Ajusta el color de fondo.
Linito	Dibuja una línea desde la posición actual (*consulte MoveToEx*) a una posición especifica.
Arc	Dibuja un arco elíptico, un segmento de una curva elíptica.
Polygon	Dibuja un polígono con dos o más vértices.
Pie	Dibuja una cuña en forma de pastel, como la cuña en un gráfico de pastel.

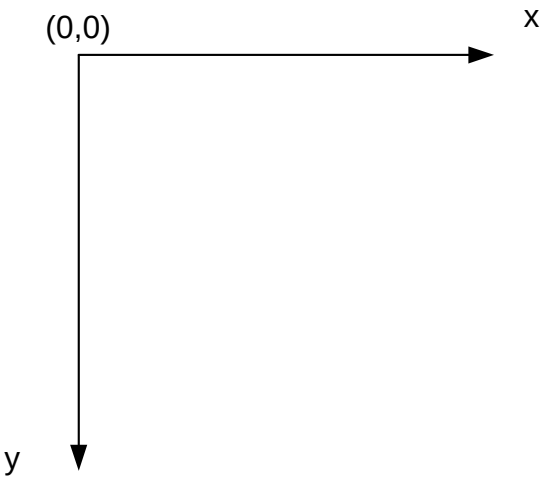
Coordenadas

Muchas de las funciones de dibujo enumeradas en la tabla anterior necesitan que se especifique información de ubicación cuando los invoca; un punto, un rectángulo, dos puntos, una matriz de puntos y demás.

Para especificar ubicaciones para dibujar y otras tareas, se necesita un *sistema de coordenadas*. Un sistema de coordenadas consiste en un origen (o punto de partida) y en un plano (como la superficie de dibujo plan en un área de cliente de ventan) dos *ejes*, uno horizontal y otro vertical. Las distancias se miden a lo largo del eje *x* (u horizontal), y el eje y (o vertical), desde el origen, el punto donde confluyen los ejes *x* e *y*. Las coordenadas del origen (0, 0), es decir (x, y). En algunos sistemas de coordenadas, los valores x e pueden ser negativos a un lado del origen y positivos al otro lado.

Windows es más que generoso proporcionando no sólo uno sino ocho sistemas de coordenadas, cada uno para un propósito diferente. El sistema de coordenadas predeterminado puede parecer no muy familiar, su origen está en el ángulo superior izquierdo del área de cliente de la ventana. En el sistema de

coordenadas predeterminado, los valores de **x** positivos si que aumentan hacia la derecha del origen, pero los valores positivos **y** aumentan hacia abajo desde el origen. No hay coordenadas negativas en este sistema, pero algunos de los sistemas si permiten valores negativos.



Coordenadas lógicas y coordenadas de dispositivo

El sistema de coordenadas es parte del contexto de dispositivo con el que se trabaja cuando se dibuja en una ventana, esta asociado sólo con el área del cliente de esa ventana. Esto significa que el origen está en la parte superior izquierda de la porción del área de cliente de su ventana. No hay coordenadas en este sistema para puntos fuera de la parte de la ventana que le pertenece. Algunas veces. Sin embargo, Windows debe pasarle información de ubicación relativa no al sistema de coordenadas de ventana sino a la pantalla completa. En casos como los mensajes relacionados con el ratón para un área que no sea de cliente de la ventana (barras de título, barras de desplazamiento, los bordes, etc), Windows pasa un punto en coordenadas de pantalla. Esto significará que las coordenadas son relativas al origen en el ángulo superior izquierdo de la pantalla completa, no el ángulo superior izquierdo del área del cliente.

Una ventana también se puede visualizar como un dispositivo, después de todo, tiene un contexto de dispositivo asociada a ella. En caso, un tipo de sistema de coordenadas en su ventana, se llama un *sistema de coordenadas de dispositivo*. Las coordenadas de dispositivo son ubicaciones físicas en la pantalla, relativas a un origen de ventana (o algunas veces a toda la pantalla), medidas en píxeles. Las coordenadas lógicas pueden estar en píxeles u otras unidades, como por ejemplo pulgadas. El contexto de dispositivo de ventana define las unidades utilizadas.

Modos de mapeado

La parte de un contexto de dispositivo que determina cuál de los ocho sistemas de coordenadas utiliza se llama *modo de mapeado*. Esto es sólo un código que indica que sistema utiliza el contexto de dispositivo.

El modo de mapeado predeterminado se llama *MM_TEXT*. Que no quiere decir que es el modo predeterminado cuando se quiere dibujar texto, lo que significa es que, como el texto en una página, su origen está en el ángulo superior izquierdo y sus valores x e y aumentan a la derecha y hacia abajo, respectivamente. Ocurre que *MM_TEXT* es muy conveniente para trazar texto en una ventaba. Pero puede que no sea tan conveniente cuando se trata de imprimir ese texto a través de un contexto de dispositivo que esté asociado no con una ventana sino con una impresora.

Otros modos de mapeado:

Una visión general de los modos de mapeado (y los sistemas de coordenadas lógicas) en Windows. Esta tabla es de Programming Windows 95 with MFC, de Jeff Prosise, (Microsoft Press 1996).

Modo de mapeado	Distancia correspondiente a una unidad lógica	Orientación de los ejes *x* e *y*
MM_TEXT	1 pixel	
MM_LOMETRIC	0.1 mm	
MM_HMETRIC	0.01 mm	
MM_LOPENGLISH	0.01 pulgadas	
MM_HIENGLISH	0.001 pulgadas	
MM_TWIPS'	1/14440 pulgadas (0.0007 pulgadas)	
MM_ISOTROPIC	Definido por el usuario (x e y se escalan de forma idéntica)	Definido por el usuario
MM_ANISOTROPIC	Definido por el usuario (x e y se escalan de forma idéntica)	Definido por el usuario

Nota: Un twip corresponde a 1/20 de un punto, lo que a su vez es alrededor de 1/72 de una pulgada. Así que un twip es (1/20 * 1/72), o 1/1440 pulgadas. Los puntos son un sistema de medida tradicional para las ubicaciones y tamaños tipográficos. Los modos MM_ISOTROPIC y MM_ANISOTROPIC le permite al programador, definir su propio sistema de coordenadas. La diferencia entre los dos (aunque no es algo sobre lo que debamos preocuparnos en este libro) es que los ejes x e y son idénticos en el modo *MM_ISOTROPIC*, pero pueden ser independientes en el modo *MM_ANISOTROPIC*.

Ciclo De Vida De Una Aplicación Windows

El usuario inicia la aplicación y Windows invoca una función llamada WinMain.

WinMain registra una "clase de ventana", información que identifica el tipo de ventana que utiliza la aplicación como su ventana principal. (Esta "clase de

ventana” no es la misma que la clase de ventana de C++, como CWnd. Pero rara vez deberá pensar en el tipo que registra WinMain. Registrar una clase de ventana significa definir sus características de modo que Windows pueda utilizar la clase para crear la ventana según su preferencia). Es en este punto donde utiliza WinMain para especificar el nombre de la función de procedimiento de ventana, que Windows invoca la continuación cada vez que necesita enviar un mensaje a su ventana.

WinMain invoca la función API para crear ventana principal de aplicación (utilizando la información de clase ventana).-

WinMain invoca a continuación la función API ShowWindow para visualizar la ventana.

WinMain invoca la función API UpdateWindow para hacer que la aplicación dibuje el contenido del área de cliente.

WinMain entra en un bucle de mensaje y se mantiene en ese bucle hasta que aparezca el mensaje WM_QUIT. En el bucle, invoca la función API GetMessage para obtener un mensaje de la cola de mensajes de aplicación invoca la función API TraslateMessage para hacer cualquier traducción necesaria para los mensajes relacionados con el teclado e invoca la función API DispatchMessage para distribuir el mensaje a la ventana apropiada en la aplicación (por medio de un procedimiento de ventana).

La ventana apropiada recibe el mensaje de Windows por medio de DispatchMessage, determina qué mensajes es, y ejecuta el código de manejador de mensaje apropiado. Si no tiene manejador para el mensaje, invoca la función API DefWindowProc para proporcionar el procesamiento predeterminado.

Cuando el bucle de mensaje de la aplicación se encuentra con un mensaje WM_QUIT, el bucle de mensaje sale, WinMain sale y Windows termina la aplicación.

Ésta es una descripción del proceso a nivel muy alto y con un lenguaje bastante llano. La descripción podría ser una aplicación escrita en C o una escrita en MFC. Éste es uno de los muchos medios que tiene MFC de sustituir tareas tediosas que, en teoría, el programador tendría que hacer para cada programa nuevo. Donde los programadores de lenguaje C para Windows tienen que escribir su propia función WinMain y su propio código para invocar el manejador de mensaje derecho, la mayoría de los programadores MFC sólo escriben los manejadores.

4. CLASES BASE DE MICROSOFT

Windows fue diseñado mucho antes que el popular lenguaje C++. Por este motivo hasta que apareció C++, prácticamente la totalidad de las aplicaciones para Windows se desarrollaban utilizando el lenguaje C y la librería de funciones de la API de Windows. No obstante, estas aplicaciones fueron construidas pensando en objetos, por tanto, un lenguaje orientado a objetos como C++ es lo más idóneo para construir una interfaz natural para desarrollar este tipo de aplicaciones.

La librería MFC (*Microsoft Foundation Class library*- librería de clases base de Microsoft), constituye verdaderamente una interfaz orientada a objetos para Windows, que permite desarrollar aplicaciones para Windows de una forma más intuitiva que la forma tradicional. Una aplicación desarrollada utilizando las MFC comparada con una versión de la misma en el sistema tradicional, contiene menos código, tiene una velocidad de ejecución comparable y también, permite llamadas a las funciones del lenguaje CC y de la API de Windows.

La MFC en ningún momento trata de reemplazar a las funciones de la API de Windows. Una función de Windows es cubierta por una función miembros de una clase sólo si hacerlo supone una clara ventaja. Esto significa que en algunas ocasiones tendremos que hacer llamadas a las funciones nativas de Windows.

La MFC están estrechamente ligadas con objetivos tales como ventanas (ventanas marco, ventanas de diálogo, botones, cajas de texto, etiquetas, etc), menús, contextos de dispositivos (pantalla, impresora, etc), y dispositivos gráficos (*bitmaps, fonts, pens, etc*), generalmente utilizados en el diseño de una aplicación Windows.

Jerarquía de la MFC

Igual que cualquier librería de clases de C++, la librería MFC encapsula su funcionalidad en clases. Las categorías de clases más importantes son:

Clase aplicación principal, CwinApp
Clase ventana, CWnd
Clase contexto de dispositivos, CDC
Clase interfaz de dispositivos gráfico (GDI), CGDIObject
Otras clases para almacenar puntos, definir rectángulos y dar soporte para menús, Cpoint, Csize, Crect y Cmenú.
Clases de propósito general.
Todas estas clases, excepto Cpoint, Csize, Crect y algunas de propósito general, se derivan de la clase Cobject. El código fuente para cada una de ellas se localiza en los subdirectorios *include* y *src* de directorio ... \m*fc*.

Notación húngara

Muchos programadores de Windows utilizan convenio para nombrar las variables, denominando *notación húngara* en honor al famoso programados de Microsoft Charles Simonyi. Sencillamente, consiste en hacer que el hombre de una variable empiece con una o más letras minúsculas que indiquen el tiempo de dato de la variable. Esta notación facilita el seguimiento de la aplicación y evita errores.

Por ejemplo el prefijo *pzs* de *pzzAppName* signifca puntero a una cadera de caracteres terminadas con el carácter nulo (*pointer a string* terminated by zero). Si el prefijo fuera m_psz entonces se trata de un dato *miembro* de una clase definido como un puntero a una cadena de caracteres terminada con el carácter nulo.

Los prefijos de las variables que se utilizarán en este trabajo son las siguientes:

Prefijo	Significado
c	Char
by	BYTE (unsigned char)
n	short o int
i	índice (int)
x, y	short (utilizadas como coordenadas *x o y*)
cx, cy	short (utilizadas como longitud *x o y*)
cb	short (longitud en bytes)
b	BOOL (int)
w	WORD (unsigned int)
l	LONG (long)
dw	DWORD (unsigned long)
fn	Función
s	cadena de carácter (string)
sz	cadena de carácter terminada con un byte ‘/0’ (string)
a	Array
p	Puntero
np	puntero (16 bits)
lp	puntero (32 bits)

Prefijo	Significado
h	handle (identificador de 16 bits)
hwns	handle a una ventana
m_	miembro de una clase
C	Clase
ID	Identificador (constante entera)

LA CLASE CObject

La clase CObject es la clase base abstracta para la librería MFC. La definición de esta clase puede verla en el futuro *afx.*

La clase CObject es la clase raíz para las clases de la librería MMFC y para cualquier clase que usted derive. Esta clase incluye un número de operaciones que son comunes a todas sus clases derivadas. Estas operaciones incluyen,

Creación y borrador de objetos
Soporte para serializaciín
Diagnóstico
Información de un objeto en tiempo de ejecución
Compatibilidad con clases de colección

Para utilizar esta funcionalidad simplemente se tiene que derivar la clase creada de la clase CObject. Por ejemplo:

```
Clase MiClase        : public CObject
{
        //declaración de la clase
};
```

la libreríaMMFC también declara un número de macros soportadas por la clase CObject y sus derivadas. Estas macros son las siguientes:

DECLARE_DYNAMIC e IMPLEMENT_DYNAMIC, permite determinar en tiempo de ejecución la clase exacta de un objeto y también su clase base. La macro DECLARE_DYNAMIC hay que ponerla dentro de la declaración de la clase y la macro IMPLEMENT_DYNAMIC hay que ponerla en el fichero *.cpp* para que sea evaluada sólo una vez. Por ejemplo, si escribe en el fichero. *h*,

```
Class CMClase       : public Cobject
{
DECLARE_DYNAMIC ( CMiClase )
Public:
CMiClase () { };
// sigue la declaración de la clase
        };
```

y en el fichero *.cpp* utiliza una línea recta similar a la siguiente,

```
IMPLEMENT_DYNAMIC (CMiClase; CObjet)
```

Ahora se puede acceder a la información de la clase CmiClase en tiempo de ejecución, utilizando la función miembro IsKindOf de la clase Cobject y la macro RUNTIME_CLASS. Por ejemplo,

```
If (MiObjeto.IsKind0f (RUNTIME_CLASS ( CMiClase ) ) )
```

La función IsKinOf devuelve TRUE si el objeto para el cual se invoca la función, pertenece a la clase especificada o una clase derivada de ella. Esta función opera correctamente en jerarquías de herencia simple.

RUNTIME_CLASS, devuelve un puntero a una estructura de tipo CRuntimeClass correspondiente a la clase específica. Los miembros de esta estructura contiene base, etc: la estructura CRuntimeClass esta definida en el fichero *afx.h*

```
CRuntimeClass* pClase = RUNTIME_CLASS ( CMiClase );
```

DECLARE_SERIAL y IMPLEMENT_SERIAL. La serialización es el proceso de escribir o de leer el contenido de un objeto desde un fichero, para ello, derivar una clase de CObject, poner la macro DECLARE_SERIAL dentro de la declaración de la clase y la macro IMPLEMENT_SERIAL en el fichero. *cpp* para que sea evaluada sólo una vez, definir un constructor sin argumentos y redefinir la función miembro Serialize. Por ejemplo.

```
// fichero .h
class CMiClase      :       public CObject
{
        DECLARE_SERIAL ( CMiClase, COject )
Public:
        CmiClase ( ) { } ; // constructor
Cstring m_nombre
UINT     m_número;
Void Serialize ( CArchive & fichero ) ;
// sigue la declaración de la clase
} ;
// fichero .cpp
IMPLEMENT_SERIAL ( CMiClase, Cobject, 0 )
```

El tercer argumento de esta macro es un número de esquema. En esencia es un número de versión ($n \geq 0$) para los objetos de la clase, que es chequeado cuando los objetos son leídos para asegurar que se corresponden con la clase en memoria.

Al redefinir la función miembro Serialize, primero se llama a la versión de Serialize de la clase base para asegurar que la parte del objeto heredada es serializada. A continuación se insertan o se extraen las variables miembros de nuestra clase. El operador de inserción es <<, y el operador de extracción>>.

```
Vid CMiClase : : Serialize ( CArchive & fichero )
```

```
{
//llamar a la versión de la clase base
Coject: : Serialize ( ichero ) ;

// insertar o extraer las variables de nuestra clase
if (      fichero. IsStoring ( )   )
fichero << m_nombre << m_numero;
else
fichero >> m_nombre >> m_numero;
}
```

Si el *fichero* se esta utilizando para escribir, la función IsStoring devuelve el valor TRUE.

Una vez que tenemos una clase serializable, podemos serializar objetos de esa clase a y desde un objeto de la clase CArchive. Un objejo CArchive siempre es asociado con un fichero.

Para crear un objeto CArchive y asociado con un fichero, hay que declarar un objeto Cfile que presente el fichero abierto y pasarlo como argumento al constructor de CArchive. Por ejemplo,

```
Cfile MiFichero;
MiFichero. Open ( “datos”, Cfile: : modewrite );
CArchive fichero (&MiFichero , Carchive : : store  );
```

El Segundo argumento que se pasa al constructor de CArchive es un valor que especifica si el fichero se va a utilizar para leer (load) o para escribir (store). Para saber cómo se está utilizando el fichero, hay que llamar a la función IsSotoring para el objeto CArchive, en nuestro caso para *fichero*.

Una vez que tenemos una clase serializable y un objeto CArchive, podemos serializar objetos de esa clase y desde el objeto CArchive. Por ejemplo,

```
CMiClase * Objetol = new CMiClase ( “Francisco” );
fichero << Objeto1;
fichero. Close ( );
MiFichero. Close ( );
```

Para deserializar un objeto, proceda según el ejemplo siguiente:

```
Cfile MiFichero;
MiFiochero.Open ( “datos”, Cfile: : modeRead );
CArchive fichero ( &MiFichero, CArchive : : load)

CMClase  *Objeto1;
fichero >> Objeto1;
fichero.Close ( ) ;
MiFichero.Close ( ) ;
```

La deserialización debe hacerse en el mismo orden que se hizo la serialización, lo que supone que el programa no debe perder de vista el orden en el que fueron serializados los objetos. Para evitar este inconveniente, lo mejor es colocar todos los objetos en una colección y serializar la colección como un único objeto.

Arquitectura de una aplicación

La librería MFC permite construir aplicaciones SDI (*Single Document interface*) y aplicaciones MDI (*Múltiple Document Interface*).

Una aplicación SDI, sólo permite tener abierta una ventana marco con la vista del documento que tiene abierto, y que también es único por cada ejemplar activo de la aplicación.

Una aplicación MMDI permite tener abiertas varias ventanas marco dentro de la ventana principal de la aplicación. Esto es, una aplicación MDI tiene una ventana marco principal dentro de la cual pueden abrirse varias ventanas marco hijas, de las cuales una solo estará activa, la que tiene la barra de título resaltada. Cada ventana fija contiene una vista de documento, lo que permite disponer de diferentes tipos ventanas; por ejemplo, ventanas de texto y ventanas de hojas de cálculo. Ninguna de las ventanas hija tiene menú, ya que comparten el menú de la ventana principal (ventana padre).

Los objetos fundamentales de una aplicación en ejecución son los siguientes:

Objeto aplicación (objeto de una clase derivada de CWinApp). Controla al resto de los objetos que forman parte de la aplicación y tiene como cometido inicializar, ejecutar y finalizar la aplicación. Sólo hay un objeto aplicación por cada aplicación Windows. También crea y gestiona las plantillas de documento para los distintos tipos de documentos que la aplicación soporte.

Planilla de documento (objeto de una clase derivada de las clases derivadas de CDocTemplate). Una planilla de documento es un mecanismo para integrar documento, vistas y ventanas. La plantilla de documento conecta un objeto documento con sus vistas asociadas y con las ventanas en las que las vistas visualizan los datos del documento, lo que asegura su utilización conjunta cuando el usuario abre o crea un determinado tipo de documento.

Una clase plantilla de documento concreta, crea y gestiona todos los documentos de un mismo tipo abiertos. Las aplicaciones que soportan más de un tipo de documento tienen varias plantillas de documento. Para crear plantillas de documento se utiliza la clase CsingleDocTemplate en aplicaciones SDI y clase CmultiDocTemplate en aplicaciones MDI.

El objeto aplicación mantiene una lista de todas las plantillas de documentos de la aplicación, de forma que cada nueva plantilla de documento se añade

automáticamente a esta lista. Por consiguiente, el objeto aplicación conoce en todo momento que tipo de ficheros están asociados con los documentos y con las vistas. La aplicación puede entonces registrar estos tipos de dicheros en el fichero *win.ini*, de esta forma un usuario puede hacer doble clic en un fichero del *Administrador de programas* y llamar a la aplicación para abrirla.

Documento (objeto de una clase derivada de CDocument). La clase documento específica los datos de la aplicación. Proporciona toda la funcionalidad necesaria para la apertura y administración de ficheros en el disco. Por lo tanto, se puede utilizar un objeto de una clase derivada de CDocument para soportar órdenes como *Fichero nuevo, Abrir fichero, Guardar fichero y Guardar como*. Para cada tipo de fichero que utilice una aplicación, hay que derivar una clase de CDocument.

Cuando se crea una aplicación, *AppWizard* crea una clase derivada de CDocument. Para crear más clases derivadas de CDocument hay que utilizar *ClassWizard*.

Ventana marco (objeto de una clase derivada de CFrameWnd, deriva de CWnd). Las ventanas marco proporcionan, a las vistas utilizadas para visualizar los datos, los controles estándar de una ventana; esto es, el marco, la barra de título, el menú de control y los botones de maximizar y minimizar la ventana. En una aplicación SSDI, la ventana marco de la vista del documento es también la ventana marco principal de la aplicación, en cambio, en una aplicación MDI, es una ventana fija de la ventana marco principal. En este caso son las ventanas hija las que contienen los objetos vistas de los documentos abiertos.

Para aplicaciones SDI, la ventana marco principal debe derivarse de CframeWnd. Si la aplicación es MDI, la ventana marco principal debe derivarse de CMDIFrameWnd y las ventanas hijas, ventanas marco de documento que soporte la aplicación se derivará una clase de CMDIhildWd.

Vista (objeto de una clase derivada de CView). Una vista es la ventana que hace de interfaz entre el usuario y los datos del objeto documento. Por lo tanto, determina cómo se visualizan los datos y cómo puede actuar el usuario sobre ellos. Un documento puede tener varias de los datos.

En una aplicación en ejecución, estos objetos responden de forma conjunta a las acciones del usuario, comunicándose entre sí por medio de mensajes. Un único objeto aplicación gestiona una o más plantillas de documento. Cada plantilla de documento crea y gestiona uno o más documentos (dependiente de si la aplicación es SDI o MDI). El usuario ve y manipula un documento a través de la vista contenida dentro de una ventana marco. La figura siguiente muestra las relaciones entre los objetos para una aplicación SDI.

La clase CWinApp

La clase CWinApp permite inicializar, ejecutar y finalizar una aplicación Windows. Cada aplicación que usted cree debe tener exactamente un objeto, global o static, de la clase CWinApp o de una clase derivada.

Para crear la ventana principal de la aplicación, básicamente, se deriva una clase de aplicación de CWinApp y se redefine la función miembro InitInstance. Por ejemplo:

```
class CAplicación   :       public CWnApp
{
        public;
                CAplicación ( ) ;
virtual BOOL InitInstance ( );
};
Caplicación MiAplicación;
```

La clase CWinApp, que está definida en el fichero *afxwind.h.* tiene un número de datos miembro que almacena valores que Windows pasa la función

predefinida WinMain. La función WinMain es el punto de entrada y de salida de una aplicación de Windows. Algunos de estos datos miembros públicos son:

m_pszAppName, especifica el nombre de la aplicación que es almacenado por el constructor de la clase. El nombre de la aplicación viene del parámetro pasado al constructor CWinApp. Si éste es el NULL, entonces el nombre se toma del recurso AFX_IDS_APP_TILLE, si éste se ha definido, y si no, se toma el nombre del fichero.*exe*.

m_hInstance, identifica al ejemplar de la aplicación actualmente en ejecución. Corresponde al parámetro hInstance que Windows pasa a WinMain. Este valor es retornado por la función AfxGetInstanceHandle.

m_hPrevInstance, identifica al ejemplar anterior de la aplicación. Correspondonde al parámetro hPrevInstance que Windows pasa a WinMain. Cuando el ejemplar actual de la aplicación es el primero que se ha puesto en ejecución, el valor de este miembro es NULL.

m_lpCmdLine, hace referencia a los argumentos que el usuario puede escribir en la línea de órdenes, cuando invoca a la aplicación. Corresponde al parámetro lpCmdLine que Windows pasa a WinMain.

m_nCmdShow, especifica cómo se visualizará la ventana principal de la aplicación cuando ésta se invoque (normal, minimizada, maximizada u oculta). Si la primera vez que se llama a la función Cwnd: :ShowWindw, el valor de m_nCmdShow es TRUE, se visualiza la ventana principal. Corresponde al parámetro nCmdShow que Windows pasa a WinMain.

m_pMainWnd, referencia la ventana principal de la aplicación.
La funcionalidad de esta clase está soportada por dos conjuntos de funciones miembro. El primer conjunto de funciones proporciona operaciones básicas tales como cargar un cursor o cargar un ícono. El segundo conjunto de

funciones inicializa, ejecuta y finaliza la aplicación. Estas funciones se pueden redefinir y algunas de ellas son las siguientes:

InitAplication. Inicializa la aplicación. Windows puede tener en ejecución varias copias o *ejemplares* de un aplicación. Esta función sólo se ejecuta para la primera copia.

InitInstance. Inicializa un ejemplar de la aplicación. Windows permite ejecutar más de una copia o ejemplar de la misma aplicación. WinMain llama a InitInstance cuando se arranca un nuevo ejemplar de aplicación.

La implementación estándar de InitInstance creada por *AppWizard* lleva a cabo las siguientes tareas:

- Carga las opciones especificas en el fichero. INI correspondiente a la aplicación, incluyendo la lista de los nombres de los ficheros usados más recientemente.
- Registra una o más plantillas de documento.
- Si la aplicación es MDI, crea la ventana marco principal.
- Procesa la línea de órdenes para abrir el documento especificado o, si no se especifico uno, abrir un nuevo documento vacío.

La función principal de IntInstance es crear las plantillas de documento que, a su vez, crean los documentos, las vistas y las ventanas marco.

Run. Proporciona un bucle de mensajes por defecto. Run toma y despacha los mensajes de la cola de mensajes de la aplicación, hasta que reciba un mensaje WM_QUIT. Cuando la cola de mensajes de la aplicación no contiene mensajes Run llama a la función OnIdle para permitir ejecutar otras tareas en segundo plano. Esta función se redefine raras veces.

int CWinApp: : Run ( )

```
{
/*      Toma y despacha mensajes hasta que reciba un mensaje WM_QUIT*
while ( 1 )
{
        LONG lIdleCount = 0;
//mientras no haya mensajes y OnIdle retorne TTRUE
while ( ! : : PeekMessage ( &m_msgCur, NULL, NULL, NULL,
PM_NOREMOVE) && OnIdle (1IdleCount++)
                {
                // hay un mensaje, o OnIdle ha retornado FALSE
}
//hay un mensaje, o OnIdle ha retornado FALSE
if ( ! : : GetMessage (&m_msgCur, NULL, NULL, NULL ) )
  break;
// procesar este mensaje
if ( !PreTranslatMessage ( &m_msgCur);
{
        : : TranslateMessage ( &m_msgCur);
: : DispatchMessage ( &m_msgCur);
}
}
return ExitInstance
}
```

PeekMessage devuelve TRUE si hay un mensaje en la cola de mensaje de la aplicación y FALSE cuando no haya mensajes en al cola de mensajes de ninguna aplicación funcionando en Windows.

La llamada a GetMessage, recoge un mensaje de la cola de mensajes. Si el mensaje es WM_QUIT, devuelve un cero y se sale del bucle de mensajes. Si no hay mensajes en la cola, cede el control a otras aplicaciones hasta que hay un mensaje.

La función PreTranslateMessage, por defecto traduce los mensajes de pulsaciones correspondientes a los acelerados (teclas aceleradoras). Redefina esta función cuando desee filtrat algún mensaje de venta antes de que sea despachado por las funciones Translate Message y DispatchMessage. En este caso, hay que llamar también a la implementación por defecto. La función PreTranslateMessage devuelve un valor distinto de cero en el caso de que ella a cabo el proceso total del mensaje; en otro caso, devuelve cero.

La función TranslateMessage traduce mensajes de pulsaciones a mensajes de caracteres. Por ejemplo si el mensaje de pulsación es WM_KEYDOWN y la pulsación produce un carácter, entonces Translate Message se coloca un mensaje de carácter en la cola de mensajes, por ejemplo WM_CHAR.

La función DispatchMessage llama al procedimiento de ventana adecuado, para procesar el mensaje.

OndIdle. Se redefine cuando se desea ejecutar tareas en segundo plano. Esta función se invoca cuando la cola de mensajes de la aplicación está vacía. El argumento lCount se pone a cero toda vez que se procesa un mensaje, y se incrementa cada vez que GetMessage encuentra la cola vacía. Este argumento determina relativamente cuanto tiempo ha estado la aplicación sin procesar un mensaje. La función OnIdle retorna por defecto un valor FALSE.

```
BOOL CwinApp: : OnIdle (LONG /*1Count*/ )
{
//proceso por defecto
//borrado de objetos temporales
   return FALSE;
}
```

La implementación por defecto de OnIdle actualiza el estado de los objetos de la interfaz del usuario, como son los botones de la barra de herramientas y los menús; también realiza un borrado de los objetos temporales creadas por el

entorno, durante el curso de sus operaciones. Esta función se redefine para efectuar tareas en segundo plano.

Cuando se redefine la función OnIdle, swdebe llamar explícitamente a la función CWinApp::OnIdle para realizar el proceso por defecto.

Si al redefinir la función OnIdle se hace que retorne un valor TRUE, dicha función se seguirá invocando mientras no haya mensajes en la cola de la aplicación.

No ejecutar tareas de larga duración porque mientras OnIdle no finalice, la aplicación no procesa entradas del usuario.

ExtInstance. Se llama cuando un ejemplar de la aplicación finaliza su ejecución. El valor que retorna esa función es el valor que devuelve WinMain. Un valor mayor que cero indica un error. La implementación por defecto de esta función escribe en el fichero. INI correspondiente a la aplicación, las opciones del entorno.

La clase Cwnd

La clase CWnd es una de las más importantes de la librería MFC. Proporciona la funcionalidad básica para todas las clases de ventana de las MFC. La definición de esta clase puede verla en el fichero *afxwin.h*.

Un objeto de la clase CWnd y una ventan Windows, no son lo mismo pero están estrechamente ligados. Un objeto CWnd es creado y destruido por el constructor y destructor, respectivamente, de CWnd. Una ventana Windows es una estructura de datos creada por la función miembro Create y destruida por el destructor de CWnd. La función miembro Destroy Window de CWnd destruye la ventana sin destruir el objeto. Si realiza alguna asignación de memoria en un objeto CWnd, redefina el destructor de CWnd para liberar la memoria. Lo expuesto es también aplicable a las clases derivadas de CWnd.

El procedimiento de ventana WndProc, queda oculto por el mecanismo de mapa de mensajes y por las funciones que manipulan cada uno de estos mensajes.

Dentro de las MFC, existen varias clases derivadas de CWnd para proporcionar tipos de ventana específicos, que podemos agrupar en ventanas propiamente dichas, las generadas a partir de CfeameWnd y Cdialog, y controles o ventanas filiales, las generadas a partir de Cstatic, Cbutton, Cedit, ClistBox, CcomboBox y CScrollBar. Estas ventanas heredan la funcionalidad de la clase CWnd, a la que añade su propia funcionalidad.

Para crear una ventana, se deriva una clase de CWnd o de una clase derivada, se añaden las variables miembro necesarias, y se implementa el mapa de mensajes y las funciones miembro correspondientes a cada uno de ellos. Por ejemplo,

```
//fichero .h
class CVentanaPpal :        public CframeWnd //clase derivada
{
public
CVentanaPpal(); //constructor
// Mensajes de notificación
afx_msg void Orden1_Clic();
//Macro para declarar el mapa de mensajes
DECLARE_MESSAGE_MAP()
};
//fichero .cpp – mapa de mensaje
BEJÍN_MESSAGE_MAP ( CventanaPpa1, CFrameWnd )
        ON_BN_CLICKED( IDB_Orden1, Orden1_Clic
END_MESSAGE_MAP()
// Definición de la función Orden 1_Clic
```

Una ventana se crea en dos pasos. Primero se llama al construir de CWnd o de una clase deriva para construir un objeto de esta clase y después, se llama a la función miembro Create para crear la ventana y ligarla al objeto. Por ejemplo,

```
BOOL CAplicación::InitInstance()
{
m_pMainWnd = new CventanaPpal(); //se llama al constructor
m_pMainWnd->ShowWindow ( m_nCmdShow );
return TRUE;
}
CEDit CajaTexto; //se llama al constructor por defecto de Cedit
CventanaPpal ::CventanaPpal () //constructor
}
//Crear la ventana principal
create (NULL, “Windows – C++”, WS_OVERLAPPEDWINDOW,
                    Crect (175, 100, 465, 300), NULL, NULL);
// Crear una caja de texto
Crect rect ( 225, 150, 415, 180 );
ScreenToClient ( &rect );
CajaTexto.Create ( WS_CHILD | WS_VISIBLE | WS_BORDER, rect,
This, IDE_CajaTexto);
}
```

La funcionalidad de esta clase está soportada por un número de datos miembro, entre los que cabe destacar m_hWnd que es un *andel* a la ventana ligada al objeto CWnd y varios conjuntos de funciones miembro. Estas funciones se clasifican de la forma siguiente:

- Constructores y destructores de la clase.
- Funciones de inicialización (Create, etc).
- Funciones para enviar mensajes (SendMessage y PostMessage).

- Funciones para manipular el texto de una ventana, incluido el título su lo tiene (SetWindowText,GetWindowTex, GetWindowTextLenght, etc).
- Funciones para manipular los menús asociados con una ventana (ver la clase CMenu).
- Función GetDlgCtrlID para obtener el ID de una ventana hija.
- Funciones para manipular el tamaño y la posición de una ventana (CloseWindow, IsIconic, IsZoomed, MoveWindow, etc),
- Conversión de coordinadas de pantalla a coordenadas del área de trabajo y viceversa ScreenToClient y ClientScreen).
- Funciones para actualizar y pintar ventanas (ShowWindow, GetWindowDC, UpdateWindow, IsWindow Visible, etc).
- Funciones para manipular temporizadores (SetTimer y KillTimer).
- Funciones de estado de una ventana (GetCapture, SetCapture, GeFocus, SetFocus, etc).
- Funciones relativas a controles de una caja de diálogo (GetDlgItem, GetDlgItemText, SetDlgItemText, CheckRadioButton, etc).
- Funciones relativas a las barras de desplazamiento de una ventana (Show-ScrollBar, SetScrollRange, GetScrollPos, etc.).
- Funciones de acceso a ventanas (GetParent,IsChild, etc)
- Funciones de alerta (MessageBox –visualizar un mensaje- y FlashWindow).
- Funciones relativas al portapapeles
- Funciones relativas al punto de inserción, no al cursor (ShowCaret, etc).
- Funciones para manipular mensajes (OnCommand, OnClose, etc).
- Funciones para manipular mensajes generados al actuar sobre un área de la ventana diferente al área de trabajo (OnNcLButtonDown, OnNcMoueseMove, OnButtonDblClk).
- Funciones para manipular mensajes del sistema (OnSysChar, etc).
- Funciones para manipular mensajes generados por la entrada del usuario (OnChar, OnKeyDown, OnKeyUp, etc).
- Funciones para manipular mensajes de inicialización (OnInitMenu y OnInitMenuPopup).

- Funciones para manipular mensajes del portapapeles (OnDrawClipboard, OnDestroyClipboard, etc).
- Funciones para manipular mensajes de controles (OnDeleteItem, etc).
- Función OnMDIActivate, llamada cuando una ventan de documento MMDI se activa o desactiva.
- Funciones miembro protegidas, para inicialización (CreateEx) y operaciones varias (PreTranslateMessage, WindowsProc, getCurrent, GetCurrentMessage, etc).

La clase CBitmap

La clase CBitmap encapsula un mapa de bits, objeto GDI, y provee de funciones miembro para manipular el mapa de bits. Para utilizar un objeto CBitmap, hay que construir el objeto, instalar el *handle* del mapa de bits en el objeto con una de las funciones miembro de inicialización, y después de haber seguido estos pasos se puede llamar a las distintas funciones miembro del objeto.

5. Documentación del Programa

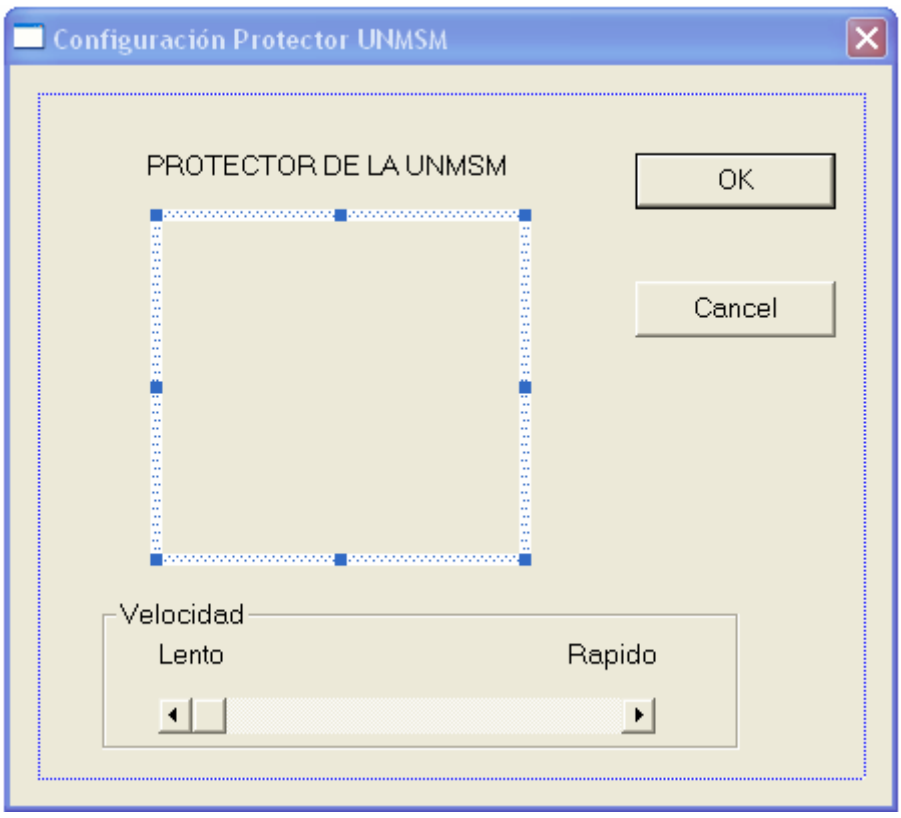
Este programa genera un ejecutable llamado ProtectorUNMSM.scr, simulando la animación del escudo de la Universidad, cargando tres diferentes mapas de bits, cada uno de ellos con vistas desde diferentes perspectivas del escudo.

Archivos Cabecera :

protector.h:	Archivo con la definición de la clase principal del Programa
protectorwnd.h:	Archivo con la definición del mapa de mensajes a ser controlados por el Protector.
dibujownd.h:	Archivo con la definición del programa que pinta los mapas de bits del Protector
protectorVista.h:	Archivo con la definición de la caja de diálogo donde se podrá configurar el Protector.
stdafx.h:	Incluir archive para archivos estándar del Sistema.

Archivos de Recursos:

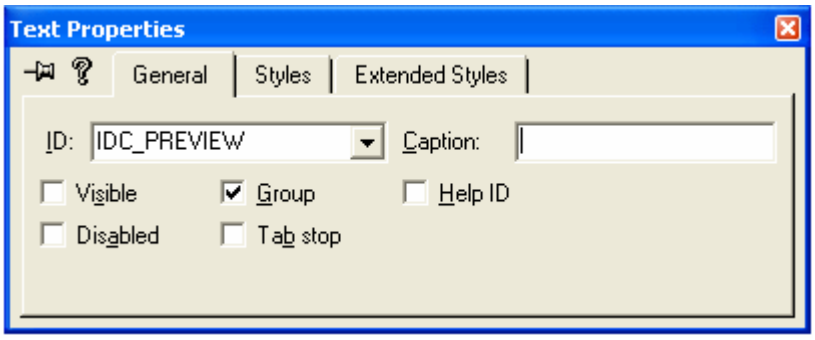
Cuadro de Diálogo : IDD_PROTECTOR_DIALOG_VISTA



Que contiene los siguientes controles:

- Barra Scrooll para configurar la velocidad del TIMER con que se hará los cambios de Mapa de Bits:

- Caja de Texto donde se muestra previamente los mapa de bits del Protector:



Icono de la aplicación:

Mapa de Bits:

IDB_BITMAP1; Nombre del archivo: Escudo.bmp

IDB_BITMAP2; Nombre del archivo: Escudo2.bmp

IDB_BITMAP3; Nombre del archivo: Escudo3.bmp

Archivos Fuentes :

Clase Principal : CProtectorApp

En el evento InitInstance de esta clase, preguntamos el parámetro de entrada, para decidir el modo de inicialización del Programa.

```
BOOL CProtectorApp::InitInstance()
{
	// Standard initialization
	// If you are not using these features and wish to reduce the size
	//  of your final executable, you should remove from the following
	//  the specific initialization routines you do not need.

	Enable3dControls();
	SetRegistryKey(_T("MFC Screen Protectors Inc."));

	if (__argc == 1 || MatchOption(__argv[1], _T('c'))) // Configuración
		DoConfig();
	else if (MatchOption(__argv[1], _T('p')))   // Modo Preview
	{
		CWnd* pParent = CWnd::FromHandle((HWND)atol(__argv[2]));
		ASSERT(pParent != NULL);
		CDibujoWnd* pWnd = new CDibujoWnd();
		CRect rect;
		pParent->GetClientRect(&rect);
		pWnd->Create(NULL, WS_VISIBLE|WS_CHILD, rect, pParent, NULL);
		m_pMainWnd = pWnd;
		return TRUE;
	}
	else if (MatchOption(__argv[1], _T('s')))   // Modo ScreenSaver
	{
		CProtectorWnd* pWnd = new CProtectorWnd;
		pWnd->Create();
		m_pMainWnd = pWnd;
		return TRUE;
	}

	return FALSE;
}
```

Si el parámetro es ‘c’ se llama a la caja de diálogo para configurar el protector. Sino encuentra el valor predeterminado para la velocidad, toma por defecto el valor más alto.

```
void CProtectorApp::DoConfig()
{
	CProtectorVista dlg;

	dlg.m_nSpeed = GetProfileInt(szConfig, _T("Speed"), 100);

	m_pMainWnd = &dlg;
	if (dlg.DoModal() == IDOK)
	{
		WriteProfileInt(szConfig, _T("Speed"), dlg.m_nSpeed);

	}
}
```

Si el parámetro es ‘p’ se llama al protector para que se ejecute dentro de la ventana que está activa.

.

Si el parámetro es ‘s’ se llama al protector para que se ejecute en toda la pantalla como ventana principal.

Clase : CProtectorVista

Cuando una aplicación recibe un mensaje WM_PAINT mientras la ventana está minimizada, dibuja el icono en el área de cliente de la aplicación. La función IsIconic devuelve TRUE si se minimiza la aplicación de Microsoft.
Sino simplemente llama a la función OnPaint()

```
void CProtectorVista::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
}
```

Cuando el programa se ejecuta en modo Configuración, muestra el Protector dentro de su caja de texto (IDC_PREVIEW) y el control para poder configurar la velocidad de cambio.

```
BOOL CProtectorVista::OnInitDialog()
{
	CDialog::OnInitDialog();

	CRect rect;
	GetDlgItem(IDC_PREVIEW)->GetWindowRect(&rect);
	ScreenToClient(&rect);
	m_wndPreview.Create(NULL, WS_VISIBLE|WS_CHILD, rect, this, NULL);
	m_wndPreview.SetSpeed(m_nSpeed);
	m_scrollSpeed.SetScrollRange(1, 100);
	m_scrollSpeed.SetScrollPos(m_nSpeed);

	CenterWindow();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	CString strAboutMenu;
	strAboutMenu.LoadString(IDS_ABOUTBOX);
	if (!strAboutMenu.IsEmpty())
	{
		pSysMenu->AppendMenu(MF_SEPARATOR);
		pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
	}

	// TODO: Add extra initialization here

	return TRUE;  // return TRUE  unless you set the focus to a control
}
```

Clase CProtectorWnd

Esta clase es la que controla si hay algún movimiento del cursor ó del teclado para detener y enviar el mensaje de finalización al Protector.

Al crearse, llama a la clase CDibujoWnd que es quién muestra los diferentes mapas de bits y que al intercambiarse dan la sensación de animación.

```
BOOL CProtectorWnd::Create()
{
	CRect rect(0, 0, ::GetSystemMetrics(SM_CXSCREEN),
		::GetSystemMetrics(SM_CYSCREEN));

	return CDibujoWnd::Create(WS_EX_TOPMOST, WS_VISIBLE|WS_POPUP, rect, NULL,
		0, NULL);
}
```

Clase CDibujoWnd

Inicializa y controla el timer, que para esta aplicación solo estamos utilizando uno, cada cambio de tiempo corre en uno el mapa de bits a cargarse y vuelve a llamar a la función draw. Que es la que dibuja el mapa de bits.

```
void CDibujoWnd::OnTimer(UINT nIDEvent)
{
//En este método se captura el evento respectivo del temporizador, que se está
anunciando, y se le asigna una acción determinada.
if (nIDEvent == 1)
	{
		m_numbitmap = m_numbitmap + 1;
		CClientDC dc(this);
		Draw(dc);
		}
	else
		CWnd::OnTimer(nIDEvent);
}
```

Obtiene la velocidad con la que está configurado el timer (previamente capturado en la venta de Diálogo de la clase **CProtectorVista**) y lo inicializa :

```
void CDibujoWnd::SetSpeed(int nSpeed)
{
	// "SetTimer(Número del temporizador, Intervalo, CERO)".
	KillTimer(1);  // borrado de la cuenta del temporizador, se pone a 0
	// inicializa el timer; cada determinados milisegundos aumenta la cuenta
	VERIFY(SetTimer(1, 50+500-nSpeed*5, NULL) != 0);
}
```

y además obtiene las coordenadas de la ventana en la que se mostrará el gráfico.

```
void CDibujoWnd::OnSize(UINT nType, int cx, int cy)
{
	CWnd::OnSize(nType, cx, cy);

	m_nScale  = cx;
	m_nHeight = cy;
}
```

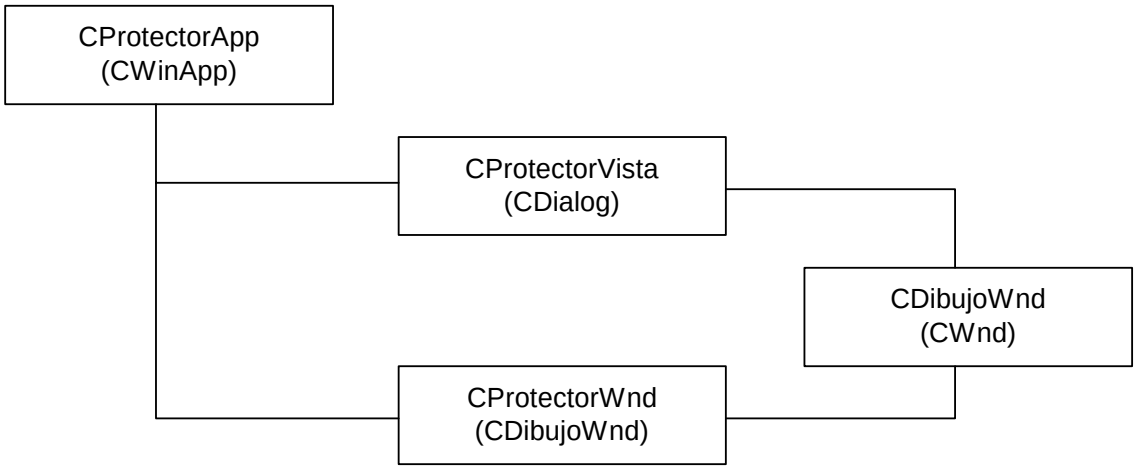
Finalmente es quién controla y carga el mapa de bits a mostrarse en la Pantalla.

```
switch (m_numbitmap)
	{
		case 1 : pBitmapActual->LoadBitmap( IDB_BITMAP1 );
				break;
		case 2 : pBitmapActual->LoadBitmap( IDB_BITMAP2 );
				break;
		case 3 : pBitmapActual->LoadBitmap( IDB_BITMAP1 );
				break;
		case 4 : pBitmapActual->LoadBitmap( IDB_BITMAP3 );
				break;
		default :
				pBitmapActual->LoadBitmap( IDB_BITMAP3 );
				break;
	}
```

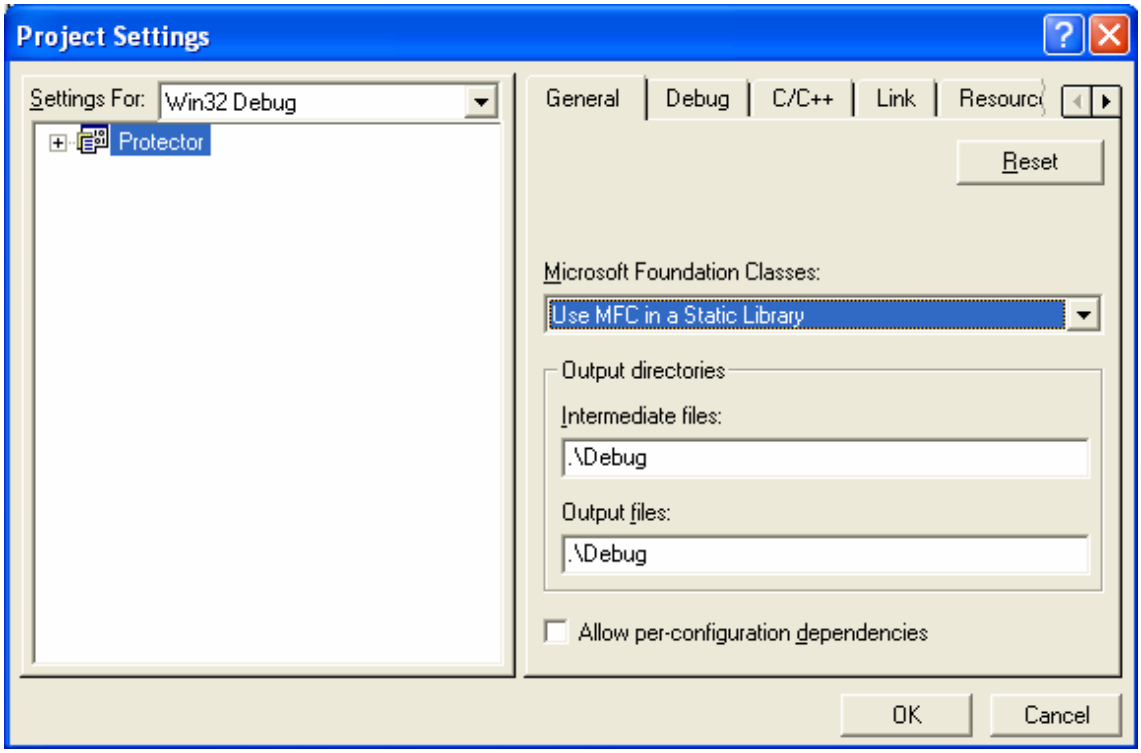
Y lo carga.en la Pantalla:

```
// Seleccionar el mapa de bits para el DC en memoria
  pBitmapAnterior = m_pMemDC->SelectObject( pBitmapActual );
 // Guardar el handle del mapa de bits anterior
 m_hBitmapAnterior = (HBITMAP)pBitmapAnterior->GetSafeHandle();

 // Dimensiones del mapa de bits fuente
 BITMAP bm; // estructura de datos BITMAP
 pBitmapActual->GetObject( sizeof(bm), &bm );
 m_nAnchoF = bm.bmWidth ;
 m_nAltoF = bm.bmHeight ;
    CRect rect(0,0,m_nAnchoF,m_nAltoF);
 dc.DPtoLP( &rect ); // tamaño del mapa de bits en unidades lógicas

 // Dimensiones del mapa de bits destino
 m_nAnchoD = rect.Width();
 m_nAltoD  = rect.Height();

    m_nposx   = (m_nScale - m_nAnchoD ) / 2;
    m_nposy   = (m_nHeight - m_nAltoD ) / 2 ;

    dc.SetMapMode(MM_TEXT);
    dc.BitBlt(
           m_nposx,m_nposy,
           m_nAnchoD ,
           m_nAltoD ,
           m_pMemDC,
           0,0,
           SRCCOPY );
```

Jeraquía de las Clases de este programa:



No olvidar que para hacer exportable el ejecutable de este programa, sin necesidad de que la PC destino tenga instalado las .dll de MFC, se debe colocar en los Settings del Proyecto la siguiente opción:



6. CONCLUSIONES:

- Hay muchas maneras de implementar un Protector de Pantalla, sin embargo hay que tener un conocimiento intermedio de cómo trabaja el Sistema Operativo sobre el cual se va a implementar, ya que este tipo de software tiene una comunicación directa con el Sistema Operativo.

- El corazón del sistema windows es la API Win32, que está construida en C. Es por eso, que el lenguaje C++ ofrece grandes posibilidades para implementar los protectores de pantallas, y estas ventajas se acrecientan con el uso de tecnología orientada a objetos y MFC.

- Construir un protector de pantalla es sencillo, solo es necesario conocer los mensajes que envía el Sistema Operativo, capturarlos y manipularlos.

- Se ha conseguido esta forma de animación, intercambiando mapa de bits del mismo objeto visualizados desde diferentes ángulos. Lo cual facilita el uso de esta aplicación para cualquier otro desarrollador que desee reutilizarla, solo bastaría reemplazar los mapas de bits por lo que se desee visualizar.

- Programar en Visual C++ es relativamente sencillo y rápido, utilizando las Líbrerías MFC que ya tienen desarrollado aplicaciones para Windows. .

- La tecnología orientada a objetos ofrece mayores ventajas al utilizar las características propias de objetos tales como Herencia, Polimorfismo y Funciones Virtuales de librerías de clases de objetos para Windows ya definidas como por ejemplo MFC.

7. BIBLIOGRAFIA

1. AprendaVisual C** 6.0 Ya. Chuck Sohar. McGraw-Hill/Interamericana de España1999.
2. Microsoft Visual C++6. Kate Gregory.Prentice Hall, Madrid 1999.
3. Programación en Windows 2000. Francisco Charte. Ediciones Anaya Multimedia s.a. 2000
4. Microsoft Visual C++. Aplicaciones para Windows. Fco. Javier Ceballos. Alfaomega Grupo Editor Mexico1997.

5. Microsoft Visual C++. Programación Avanzada en Win32. Fco Javier Ceballos.RA-MA Editorial. Madrid 1999

ANEXO 1: CONFIGURACION EN EL SISTEMA WINDOWS XP

Copiar el programa ***ProtectorUNMSM*.scr** provisto en la ruta “EjecutableFinal” del CD que acompaña este trabajo a la unidad de disco donde se guarda el Sistema Operativo de la PC destino, en la siguiente ruta:

\WINDOWS\SYSTEM32

En el ícono Pantalla, elegir la pestaña Protector de Pantalla y elegir el Protector llamado **ProtectorUNMSM**

Para modificar la velocidad del movimiento del Escudo, presionar el botón Configuración, se visualizará la siguiente pantalla:

Y con la barra de desplazamiento llamada “Velocidad” configurar la velocidad de cambio de los Mapa de Bits.

ANEXO 2: CODIGOS FUENTES

```
// Protector.h : main header file for the Protector application
//

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorApp:
// See Protector.cpp for the implementation of this class
//

class CProtectorApp : public CWinApp
{
public:
      CProtectorApp();
// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CProtectorApp)
      public:
      virtual BOOL InitInstance();
      //}}AFX_VIRTUAL

// Implementation


      //{{AFX_MSG(CProtectorApp)
            // NOTE - the ClassWizard will add and remove member
functions here.
            //    DO NOT EDIT what you see in these blocks of generated
code !
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()

      void DoConfig();
};


/////////////////////////////////////////////////////////////////////////
///////
//
```

```
// ProtectorVista.h : header file
//

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorVista dialog

class CProtectorVista : public CDialog
{
// Construction
public:
      CProtectorVista(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
      //{{AFX_DATA(CProtectorVista)
      enum { IDD = IDD_PROTECTOR_DIALOG_VISTA };
            // NOTE: the ClassWizard will add data members here
      CScrollBar  m_scrollSpeed;
      int     m_nSpeed;
      //}}AFX_DATA
      COLORREF m_color;


// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CProtectorVista)
      protected:
      virtual void DoDataExchange(CDataExchange* pDX);     // DDX/DDV
support
      //}}AFX_VIRTUAL

// Implementation
protected:
      HICON m_hIcon;
      CDibujoWnd m_wndPreview; // don't autodelete


      // Generated message map functions
      //{{AFX_MSG(CProtectorVista)
      virtual BOOL OnInitDialog();
      afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
      afx_msg void OnPaint();
      afx_msg HCURSOR OnQueryDragIcon();
      afx_msg  void  OnHScroll(UINT  nSBCode,  UINT  nPos,  CScrollBar*
pScrollBar);
      afx_msg void OnPreview();
      virtual void OnOK();
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
//   Microsoft   Visual   C++   will   insert   additional   declarations
immediately before the previous line.
```

```
// Protectorwnd.h : header file
//

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorWnd window

class CProtectorWnd : public CDibujoWnd
{
// Construction
public:
      CProtectorWnd();

// Attributes
public:
      CPoint m_ptLast;

// Operations
public:
      BOOL Create();

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CProtectorWnd)
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CProtectorWnd();

      // Generated message map functions
protected:
      //{{AFX_MSG(CProtectorWnd)
      afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
      afx_msg void OnDestroy();
      afx_msg  BOOL  OnSetCursor(CWnd*  pWnd,  UINT  nHitTest,  UINT
message);
      afx_msg BOOL OnNcActivate(BOOL bActive);
      afx_msg  void  OnActivate(UINT  nState,  CWnd*  pWndOther,  BOOL
bMinimized);
      afx_msg void OnActivateApp(BOOL bActive, HTASK hTask);
      afx_msg void OnMouseMove(UINT nFlags, CPoint point);
      afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
      afx_msg void OnMButtonDown(UINT nFlags, CPoint point);
      afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
      afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
      afx_msg  void  OnSysKeyDown(UINT  nChar,  UINT  nRepCnt,  UINT
nFlags);
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
///////
```

```
// DibujoWnd.h : header file
//

/////////////////////////////////////////////////////////////////////////
///////
// CDibujoWnd window

class CDibujoWnd : public CWnd
{
// Construction
public:
      CDibujoWnd(BOOL bAutoDelete = TRUE);

// Attributes
public:
      CRgn m_rgnLast;
      int m_nWidth;
      int m_nScale;
      int m_nHeight;
      int m_numbitmap;
      int m_nposx;
      int m_nposy;
      LOGBRUSH m_logbrush;
      LOGBRUSH m_logbrushBlack;
      static LPCTSTR m_lpszClassName;
      void SetSpeed(int nSpeed);
      void SetLineStyle(int nStyle);

//***********************
//private:
      CDC* m_pMemDC;    // puntero a un DC en memoria
      HBITMAP m_hBitmapAnterior;
      int m_nAnchoF, m_nAltoF , m_nAnchoD , m_nAltoD;
//***********************

// Operations
public:
      void Draw(CDC& dc);

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CDibujoWnd)
      public:
      virtual BOOL Create(DWORD dwExStyle, DWORD dwStyle, const RECT&
rect, CWnd* pParentWnd, UINT nID, CCreateContext* pContext = NULL);
      protected:
      virtual void PostNcDestroy();
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~CDibujoWnd();

protected:
      BOOL m_bAutoDelete;

      // Generated message map functions
protected:
      //{{AFX_MSG(CDibujoWnd)
```

```
	afx_msg void OnTimer(UINT nIDEvent);
	afx_msg void OnPaint();
	afx_msg void OnSize(UINT nType, int cx, int cy);
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
///////
```

```
// Protector.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "Protector.h"
#include "DibujoWnd.h"
#include "Protectorwnd.h"
#include "ProtectorVista.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorApp

BEGIN_MESSAGE_MAP(CProtectorApp, CWinApp)
      //{{AFX_MSG_MAP(CProtectorApp)
            //  NOTE  -  the  ClassWizard  will  add  and  remove  mapping
macros here.
            //    DO NOT EDIT what you see in these blocks of generated
code!
      //}}AFX_MSG
      ON_COMMAND(ID_HELP, CWinApp::OnHelp)
END_MESSAGE_MAP()

TCHAR szConfig[]=_T("Config");

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorApp construction

CProtectorApp::CProtectorApp()
{
      // TODO: add construction code here,
      // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////
///////
// The one and only CProtectorApp object

CProtectorApp theApp;

BOOL MatchOption(LPTSTR lpsz, TCHAR ch)
{
      if (lpsz[0] == '-' || lpsz[0] == '/')
            lpsz++;

      if (lpsz[0] == ch)
        return TRUE;

      return FALSE;
}

/////////////////////////////////////////////////////////////////////////
///////
```

```
// CProtectorApp initialization

BOOL CProtectorApp::InitInstance()
{
	// Standard initialization
	// If you are not using these features and wish to reduce the
size
	//	of your final executable, you should remove from the
following
	//	the specific initialization routines you do not need.

	Enable3dControls();
	SetRegistryKey(_T("MFC Screen Protectors Inc."));

	if (__argc == 1 || MatchOption(__argv[1], _T('c'))) //
Configuración
		DoConfig();
	else if (MatchOption(__argv[1], _T('p')))// Modo Preview
	{
		CWnd* pParent = CWnd::FromHandle((HWND)atol(__argv[2]));
		ASSERT(pParent != NULL);
		CDibujoWnd* pWnd = new CDibujoWnd();
		CRect rect;
		pParent->GetClientRect(&rect);
		pWnd->Create(NULL, WS_VISIBLE|WS_CHILD, rect, pParent,
NULL);
		m_pMainWnd = pWnd;
		return TRUE;
	}
	else if (MatchOption(__argv[1], _T('s')))// Modo ScreenSaver
	{
		CProtectorWnd* pWnd = new CProtectorWnd;
		pWnd->Create();
		m_pMainWnd = pWnd;
		return TRUE;
	}

	return FALSE;
}

void CProtectorApp::DoConfig()
{
	CProtectorVista dlg;

	dlg.m_nSpeed = GetProfileInt(szConfig, _T("Speed"), 100);

	m_pMainWnd = &dlg;
	if (dlg.DoModal() == IDOK)
	{
		WriteProfileInt(szConfig, _T("Speed"), dlg.m_nSpeed);

	}
}
```

```
// ProtectorVista.cpp : implementation file
//

#include "stdafx.h"
#include "Protector.h"
#include "DibujoWnd.h"
#include "ProtectorVista.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorVista dialog

CProtectorVista::CProtectorVista(CWnd* pParent /*=NULL*/)
	: CDialog(CProtectorVista::IDD, pParent), m_wndPreview(FALSE)
{
	//{{AFX_DATA_INIT(CProtectorVista)
		// NOTE: the ClassWizard will add member initialization
here
	m_nSpeed = 0;
	//}}AFX_DATA_INIT
	// Note that LoadIcon does not require a subsequent DestroyIcon
in Win32
	m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}



/////////////////////////////////////////////////////////////////////////
///////
// CProtectorVista message handlers

void CProtectorVista::OnPreview()
{
	// TODO: Add your control notification handler code here

}

/////////////////////////////////////////////////////////////////////////
///////
// CAboutDlg message handlers

/////////////////////////////////////////////////////////////////////////
///////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
	CAboutDlg();

// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
```

```
	//}}AFX_DATA

// Implementation
protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support
	//{{AFX_MSG(CAboutDlg)
	virtual BOOL OnInitDialog();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
///////
// CAboutDlg message handlers

BOOL CAboutDlg::OnInitDialog()
{
	CDialog::OnInitDialog();
	CenterWindow();

	// TODO: Add extra about dlg initialization here

	return TRUE;  // return TRUE  unless you set the focus to a
control
}


/////////////////////////////////////////////////////////////////////////////
///////


void CProtectorVista::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	DDX_Control(pDX, IDC_SCROLLBAR_SPEED, m_scrollSpeed);
	DDX_Scroll(pDX, IDC_SCROLLBAR_SPEED, m_nSpeed);
	//{{AFX_DATA_MAP(CProtectorVista)
	//}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CProtectorVista, CDialog)
	//{{AFX_MSG_MAP(CProtectorVista)
	ON_WM_SYSCOMMAND()
	ON_WM_PAINT()
	ON_WM_QUERYDRAGICON()
	ON_WM_HSCROLL()
	ON_BN_CLICKED(IDC_PREVIEW, OnPreview)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorVista message handlers

BOOL CProtectorVista::OnInitDialog()
{
	CDialog::OnInitDialog();

	CRect rect;
	GetDlgItem(IDC_PREVIEW)->GetWindowRect(&rect);
	ScreenToClient(&rect);
	m_wndPreview.Create(NULL,   WS_VISIBLE|WS_CHILD,   rect,   this,
NULL);
	m_wndPreview.SetSpeed(m_nSpeed);
	m_scrollSpeed.SetScrollRange(1, 100);
	m_scrollSpeed.SetScrollPos(m_nSpeed);

	CenterWindow();

	// Add "About..." menu item to system menu.

	// IDM_ABOUTBOX must be in the system command range.
	ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
	ASSERT(IDM_ABOUTBOX < 0xF000);

	CMenu* pSysMenu = GetSystemMenu(FALSE);
	CString strAboutMenu;
	strAboutMenu.LoadString(IDS_ABOUTBOX);
	if (!strAboutMenu.IsEmpty())
	{
		pSysMenu->AppendMenu(MF_SEPARATOR);
		pSysMenu->AppendMenu(MF_STRING,              IDM_ABOUTBOX,
strAboutMenu);
	}

	// TODO: Add extra initialization here

	return TRUE;   // return TRUE   unless you set the focus to a
control
}

void CProtectorVista::OnSysCommand(UINT nID, LPARAM lParam)
{
	if ((nID & 0xFFF0) == IDM_ABOUTBOX)
	{
		CAboutDlg dlgAbout;
		dlgAbout.DoModal();
	}
```

```
	else
	{
		CDialog::OnSysCommand(nID, lParam);
	}
}

// If you add a minimize button to your dialog, you will need the code
below
//  to draw the icon.  For MFC applications using the document/view
model,
//  this is automatically done for you by the framework.

void CProtectorVista::OnPaint()
{
	if (IsIconic())
	{
		CPaintDC dc(this); // device context for painting

		SendMessage(WM_ICONERASEBKGND,  (WPARAM)  dc.GetSafeHdc(),
0);

		// Center icon in client rectangle
		int cxIcon = GetSystemMetrics(SM_CXICON);
		int cyIcon = GetSystemMetrics(SM_CYICON);
		CRect rect;
		GetClientRect(&rect);
		int x = (rect.Width() - cxIcon + 1) / 2;
		int y = (rect.Height() - cyIcon + 1) / 2;

		// Draw the icon
		dc.DrawIcon(x, y, m_hIcon);
	}
	else
	{
		CDialog::OnPaint();
	}
}

// The system calls this to obtain the cursor to display while the
user drags
//  the minimized window.
HCURSOR CProtectorVista::OnQueryDragIcon()
{
	return (HCURSOR) m_hIcon;
}

void CProtectorVista::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
	ASSERT(pScrollBar != NULL);
	int nCurPos = pScrollBar->GetScrollPos();
	switch (nSBCode)
	{
	case SB_LEFT: //    Scroll to far left.
		nCurPos = 1;
		break;
	case SB_LINELEFT: //    Scroll left.
		nCurPos--;
		break;
```

```
	case SB_LINERIGHT: //    Scroll right.
		nCurPos++;
		break;
	case SB_PAGELEFT: //    Scroll one page left.
		nCurPos -= 10;
		break;
	case SB_PAGERIGHT: //    Scroll one page right.
		nCurPos += 10;
		break;
	case SB_RIGHT: //    Scroll to far right.
		nCurPos = 100;
		break;
	case SB_THUMBPOSITION: //     Scroll to absolute position. The
current position is specified by the nPos parameter.
	case SB_THUMBTRACK: //    Drag scroll box to specified position.
The current position is specified by the nPos parameter.
		nCurPos = nPos;
	}
	if (nCurPos < 1)
		nCurPos = 1;
	if (nCurPos > 100)
		nCurPos = 100;
	pScrollBar->SetScrollPos(nCurPos);

	m_wndPreview.SetSpeed(m_scrollSpeed.GetScrollPos());

	CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}


void CProtectorVista::OnOK()
{
	// TODO: Add extra validation here

	CDialog::OnOK();
}
```

```
// Protectorwnd.cpp : implementation file
//

#include "stdafx.h"
#include "Protector.h"
#include "DibujoWnd.h"
#include "Protectorwnd.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
// CProtectorWnd

CProtectorWnd::CProtectorWnd()
{
      m_ptLast = CPoint(-1, -1);
}

CProtectorWnd::~CProtectorWnd()
{
}


BEGIN_MESSAGE_MAP(CProtectorWnd, CDibujoWnd)
      //{{AFX_MSG_MAP(CProtectorWnd)
      ON_WM_SYSCOMMAND()
      ON_WM_DESTROY()
      ON_WM_SETCURSOR()
      ON_WM_NCACTIVATE()
      ON_WM_ACTIVATE()
      ON_WM_ACTIVATEAPP()
      ON_WM_MOUSEMOVE()
      ON_WM_LBUTTONDOWN()
      ON_WM_MBUTTONDOWN()
      ON_WM_RBUTTONDOWN()
      ON_WM_KEYDOWN()
      ON_WM_SYSKEYDOWN()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////
///////
// CProtectorWnd message handlers

BOOL CProtectorWnd::Create()
{
      CRect rect(0, 0, ::GetSystemMetrics(SM_CXSCREEN),
            ::GetSystemMetrics(SM_CYSCREEN));

      return   CDibujoWnd::Create(WS_EX_TOPMOST,   WS_VISIBLE|WS_POPUP,
rect, NULL,
            0, NULL);
}
```

```
void CProtectorWnd::OnSysCommand(UINT nID, LPARAM lParam)
{
	if ((nID == SC_SCREENSAVE) || (nID == SC_CLOSE))
		return;
	CDibujoWnd::OnSysCommand(nID, lParam);
}

void CProtectorWnd::OnDestroy()
{
	PostQuitMessage(0);
	CDibujoWnd::OnDestroy();
}

BOOL CProtectorWnd::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT
message)
{
	SetCursor(NULL);
	return TRUE;
}

BOOL CProtectorWnd::OnNcActivate(BOOL bActive)
{
	if (!bActive)
		return FALSE;
	return CDibujoWnd::OnNcActivate(bActive);
}

void CProtectorWnd::OnActivate(UINT nState, CWnd* pWndOther, BOOL
bMinimized)
{
	if (nState == WA_INACTIVE)
		PostMessage(WM_CLOSE);
	CDibujoWnd::OnActivate(nState, pWndOther, bMinimized);
}

void CProtectorWnd::OnActivateApp(BOOL bActive, HTASK hTask)
{
	if (!bActive)
		PostMessage(WM_CLOSE);
	CDibujoWnd::OnActivateApp(bActive, hTask);
}

void CProtectorWnd::OnMouseMove(UINT nFlags, CPoint point)
{
	if (m_ptLast == CPoint(-1,-1))
		m_ptLast = point;
	else if (m_ptLast != point)
		PostMessage(WM_CLOSE);
	CDibujoWnd::OnMouseMove(nFlags, point);
}

void CProtectorWnd::OnLButtonDown(UINT nFlags, CPoint point)
{
	PostMessage(WM_CLOSE);
	CDibujoWnd::OnLButtonDown(nFlags, point);
}

void CProtectorWnd::OnMButtonDown(UINT nFlags, CPoint point)
{
```

```
	PostMessage(WM_CLOSE);
	CDibujoWnd::OnMButtonDown(nFlags, point);
}

void CProtectorWnd::OnRButtonDown(UINT nFlags, CPoint point)
{
	PostMessage(WM_CLOSE);
	CDibujoWnd::OnRButtonDown(nFlags, point);
}

void CProtectorWnd::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
	PostMessage(WM_CLOSE);
	CDibujoWnd::OnKeyDown(nChar, nRepCnt, nFlags);
}

void  CProtectorWnd::OnSysKeyDown(UINT  nChar,  UINT  nRepCnt,  UINT
nFlags)
{
	PostMessage(WM_CLOSE);
	CDibujoWnd::OnSysKeyDown(nChar, nRepCnt, nFlags);
}
```

```
// DibujoWnd.cpp : implementation file
//

#include "stdafx.h"
#include "Protector.h"
#include "DibujoWnd.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

LPCTSTR CDibujoWnd::m_lpszClassName = NULL;

/////////////////////////////////////////////////////////////////////////////
///////
// CDibujoWnd

CDibujoWnd::CDibujoWnd(BOOL bAutoDelete)
{

//****************************
      m_pMemDC = 0;
      m_hBitmapAnterior = 0;

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

      m_bAutoDelete = bAutoDelete;
      m_rgnLast.CreateRectRgn(0,0,0,0);
      m_logbrushBlack.lbColor = RGB(0, 0, 0);
}

CDibujoWnd::~CDibujoWnd()
{
// ********************************

      // Eliminar el mapa de bits
      if (m_hBitmapAnterior )
      {
            CBitmap *pbm = CBitmap::FromHandle( m_hBitmapAnterior );
            delete m_pMemDC->SelectObject( pbm );
      }
      // Eliminar el DC de memoria
      delete m_pMemDC;

// ********************************
}


BEGIN_MESSAGE_MAP(CDibujoWnd, CWnd)
      //{{AFX_MSG_MAP(CDibujoWnd)
      ON_WM_TIMER()
      ON_WM_PAINT()
      ON_WM_SIZE()
      ON_WM_CREATE()
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

```
void CDibujoWnd::Draw(CDC& dc)
{
      // Controla que mapa de bits se visualizará
      if (m_numbitmap > 4  || m_numbitmap < 0 )
            m_numbitmap = 1 ;

      // Borrar cualquier mapa de bits actual
      if ( m_hBitmapAnterior )
      {
        CBitmap *pbm = CBitmap::FromHandle( m_hBitmapAnterior );
        delete m_pMemDC->SelectObject( pbm );

      }
      // Eliminar el DC de memoria
      delete m_pMemDC;

      //  Crear  un  DC  en  memoria  compatible  con  el  DC  del  área  de
trabajo
      m_pMemDC = new CDC;
      m_pMemDC->CreateCompatibleDC( &dc );
      // Cargar un mapa de bits
    CBitmap *pBitmapAnterior, *pBitmapActual = new CBitmap;

      switch (m_numbitmap)
      {
            case 1 : pBitmapActual->LoadBitmap( IDB_BITMAP1 );
                        break;
            case 2 : pBitmapActual->LoadBitmap( IDB_BITMAP2 );
                        break;
            case 3 : pBitmapActual->LoadBitmap( IDB_BITMAP1 );
                        break;
            case 4 : pBitmapActual->LoadBitmap( IDB_BITMAP3 );
                        break;
            default :
                        pBitmapActual->LoadBitmap( IDB_BITMAP3 );
                        break;
      }


    // Seleccionar el mapa de bits para el DC en memoria
     pBitmapAnterior = m_pMemDC->SelectObject( pBitmapActual );
    // Guardar el handle del mapa de bits anterior
    m_hBitmapAnterior = (HBITMAP)pBitmapAnterior->GetSafeHandle();

    // Dimensiones del mapa de bits fuente
    BITMAP bm; // estructura de datos BITMAP
    pBitmapActual->GetObject( sizeof(bm), &bm );
    m_nAnchoF = bm.bmWidth ;
    m_nAltoF = bm.bmHeight ;
      CRect rect(0,0,m_nAnchoF,m_nAltoF);
    dc.DPtoLP( &rect ); // tamaño del mapa de bits en unidades lógicas

    // Dimensiones del mapa de bits destino
    m_nAnchoD = rect.Width();
    m_nAltoD  = rect.Height();

      m_nposx   = (m_nScale - m_nAnchoD ) / 2;
      m_nposy   = (m_nHeight - m_nAltoD ) / 2 ;
```

```
	dc.SetMapMode(MM_TEXT);
	dc.BitBlt(
		m_nposx,m_nposy,
		m_nAnchoD ,
		m_nAltoD ,
		m_pMemDC,
		0,0,
		SRCCOPY );


}

void CDibujoWnd::SetSpeed(int nSpeed)
{
	// "SetTimer(Número del temporizador, Intervalo, CERO)".
	KillTimer(1);	// borrado de la cuenta del temporizador, se
pone a 0
	// inicializa el timer; cada determinados milisegundos aumenta
la cuenta
	VERIFY(SetTimer(1, 50+500-nSpeed*5, NULL) != 0);
}



/////////////////////////////////////////////////////////////////////////
///////
// CDibujoWnd message handlers

void CDibujoWnd::OnTimer(UINT nIDEvent)
{
//En este método se captura el evento respectivo del temporizador, que
se está anunciando, y se le asigna una acción determinada.
if (nIDEvent == 1)
	{
		m_numbitmap = m_numbitmap + 1;
		CClientDC dc(this);
		Draw(dc);
		}
	else
		CWnd::OnTimer(nIDEvent);
}

void CDibujoWnd::OnPaint()
{
	CPaintDC dc(this); // device context for painting
	m_rgnLast.DeleteObject();
	m_rgnLast.CreateRectRgn(0,0,0,0);
	CBrush brush(RGB(0,0,0));
	CRect rect;
	GetClientRect(rect);
	dc.FillRect(&rect, &brush);
	Draw(dc);
	// Do not call CWnd::OnPaint() for painting messages
}

void CDibujoWnd::OnSize(UINT nType, int cx, int cy)
{
	CWnd::OnSize(nType, cx, cy);
```

```
	m_nScale  = cx;
	m_nHeight = cy;
}

int CDibujoWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CWnd::OnCreate(lpCreateStruct) == -1)
		return -1;

	int nSpeed = AfxGetApp()->GetProfileInt("Config", "Speed", 1);
	if (nSpeed < 0)
		nSpeed = 0;
	SetSpeed(nSpeed);

	return 0;
}

BOOL CDibujoWnd::Create(DWORD dwExStyle, DWORD dwStyle, const RECT&
rect,
	CWnd* pParentWnd, UINT nID, CCreateContext* pContext)
{
	// Register a class with no cursor
	if (m_lpszClassName == NULL)
	{
		m_lpszClassName                                          =
AfxRegisterWndClass(CS_HREDRAW|CS_VREDRAW,
			::LoadCursor(AfxGetResourceHandle(),
			MAKEINTRESOURCE(IDC_NULLCURSOR)));
	}

	// TODO: Add your specialized code here and/or call the base
class
	return CreateEx(dwExStyle, m_lpszClassName, _T(""), dwStyle,
		rect.left, rect.top, rect.right - rect.left, rect.bottom -
rect.top,
		pParentWnd->GetSafeHwnd(), NULL, NULL );
}

void CDibujoWnd::PostNcDestroy()
{
	if (m_bAutoDelete)
		delete this;
}
```