

# Inicialización de DirectX 9

DirectX nos permite crear videojuegos, pero primero, necesitamos instalarlo y configurarlo correctamente para que el compilador pueda trabajar con él.

En este capítulo, se creará una aplicación base que puede ser usada como punto de partida para todos los ejemplos del libro y sus propias aplicaciones.

Podemos utilizar Microsoft Visual Studio .NET en su versión 2003 ó 2005 para compilar los ejemplos del libro.

<b>Los pasos previos</b>	<b>14</b>
Requerimientos	14
Creación de una aplicación base	14
Configurar el compilador	18
<b>El código de la aplicación</b>	<b>19</b>
La clase de soporte	27
<b>Resumen</b>	<b>35</b>
<b>Actividades</b>	<b>36</b>

## LOS PASOS PREVIOS

Antes de comenzar, debemos contar con el software necesario instalado en nuestra PC. A continuación, se detalla el software que se nos requerirá para realizar los ejemplos tratados en el presente libro.

### Requerimientos

En este texto, utilizaremos el lenguaje de programación **C++**. Lo primero que debemos instalar en la computadora es el compilador, en nuestro caso utilizaremos **MS Visual Studio 2003 ó 2005**.

Una vez hecho esto, instalaremos el **SDK de DirectX**. El **SDK (Software Development Kit)** o kit de desarrollo de software, que contiene las librerías, componentes, ejemplos, herramientas y documentación necesarios para poder desarrollar nuestras propias aplicaciones de **DirectX**.

### Sitios web

El **SDK de DirectX** se puede descargar, directamente, del sitio de **Microsoft: [www.microsoft.com/directx](http://www.microsoft.com/directx)**.

La versión que utilizamos en este libro es **DirectX 9.0c**. El **SDK** es de varios megabytes, por lo que puede tardar varios minutos en descargarse totalmente. En cuanto finaliza la descarga del **SDK**, lo instalamos en forma completa.

### Creación de una aplicación base

Para comenzar a desarrollar aplicaciones con **DirectX**, comenzaremos desarrollando una aplicación que se utilizará como base para los proyectos.

La aplicación base es la más importante; con ella, inicializaremos **DirectX** y dejaremos todo listo para poder crear diferentes aplicaciones. La aplicación base es **Win32** (podemos leer el **Apéndice B** si no se tienen conocimientos previos de programación en Win32). Los pasos que a continuación se detallan pueden ser usados tanto en la versión 2003 como en la versión 2005 de **MS Visual Studio**.

---

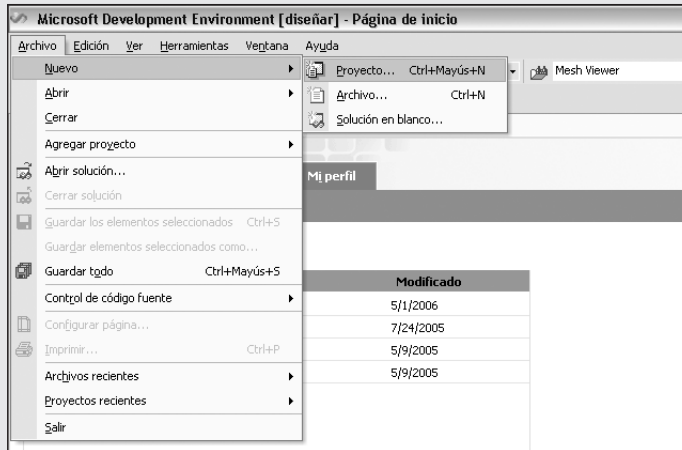
## INSTALAR MSDN Y EL SDK ES CONVENIENTE

Cuando instalemos el compilador, es importante que, también, se instale **MSDN** completo. Resultará de mucha ayuda cuando tengamos dudas sobre sintaxis, funciones, etcétera. También debemos instalar el **SDK de DirectX**. Recomendamos buscar las funciones de **DirectX** que se utilizan en los ejemplos de la ayuda y, de esta forma, aprender más sobre sus parámetros y forma de uso.

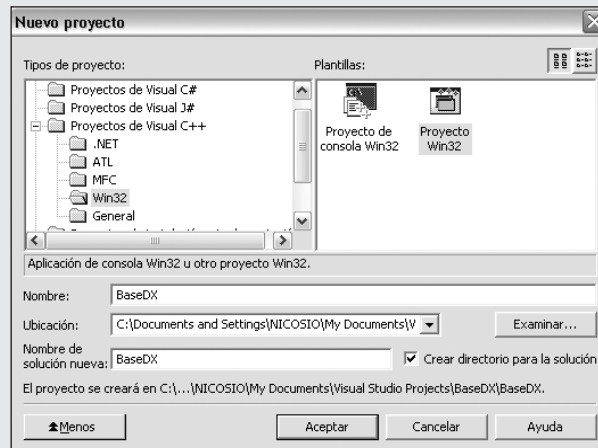
## ■ Creación de un proyecto

PASO A PASO

- 1 Inicie Visual Studio .Net y diríjase al menú de **Archivo**, seleccione **Nuevo** y, luego, **Proyecto**.



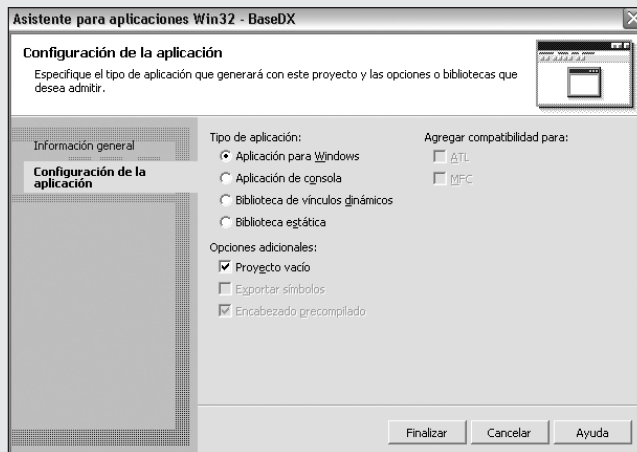
- 2 Aparecerá una ventana de diálogo. Ésta es la ventana del nuevo proyecto; allí colocaremos el tipo de proyecto que deseamos crear. En la sección de **Tipos de proyecto**, expanda **Proyectos de Visual C++** y seleccione **Win32**. En la sección de **Plantillas**, seleccione **Proyecto Win32**.



- 3 Coloque el nombre del proyecto. Puede usar cualquier nombre que desee. Para este ejemplo, el nombre del proyecto es **BaseDX**. No olvide seleccionar la casilla **Crear directorio para la solución**. Si esta casilla no aparece, simplemente, haga clic en el botón **Más**, de **Nuevo Proyecto**. Para finalizar, presione el botón **Aceptar**.

4

Aparece una ventana de diálogo nueva. Es el **Asistente para aplicaciones Win32** cuya función es la de ayudar al programador a configurar las características de la aplicación. Haga clic en la sección **Configuración de la aplicación**. Verifique que esté seleccionado **Aplicación para Windows** en la sección **Tipo de aplicación**. Luego, verifique que la casilla **Proyecto vacío** en la sección de **Opciones adicionales** esté marcada. Después, simplemente, presione el botón **Finalizar**.



Con los pasos anteriores, se ha creado el proyecto con el cual vamos a trabajar. Éste necesitará tres archivos: **main.cpp**, **soporte.h** y **soporte.cpp**.

En el archivo **main.cpp**, se crea el programa principal, donde se coloca la lógica del programa; éste es el documento sobre el que se trabajará en los siguientes capítulos. Se crea una clase que brinda soporte a la aplicación. Su función es la de inicializar **DirectX** y crear la ventana de la aplicación. Esta clase se define en **soporte.h** y se implementa dentro de **soporte.cpp**.

Veamos los pasos que debemos seguir para incluir estos archivos dentro de nuestra aplicación. En primer lugar, incluiremos el documento **main.cpp**, luego continuaremos con el resto de los archivos mencionados.



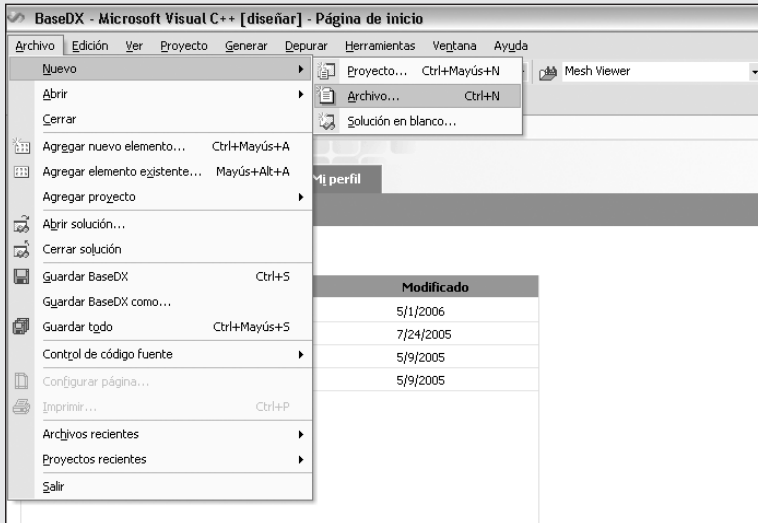
## VERIFIQUE LA VERSIÓN CORRECTA DE DIRECTX

Cuando descargue **DirectX**, verifique que es el **SDK** y no el **runtime**. El **SDK** nos permite crear aplicaciones, en cambio el **runtime** sólo nos permite ejecutar las aplicaciones. El **SDK** que utilizamos en este libro es la versión 9.0c. Descarguemos siempre del sitio oficial de Microsoft.

## ■ Colocar los archivos necesarios

## PASO A PASO

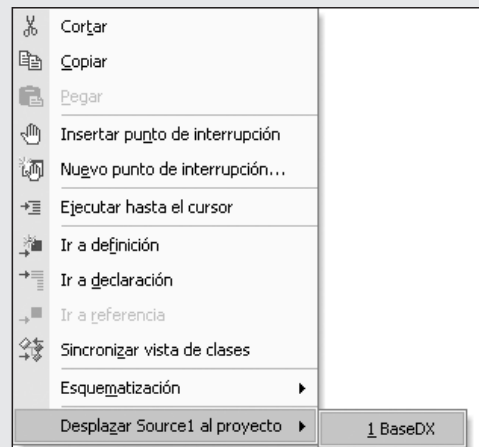
- 1 Vaya al menú de **Archivo**, seleccione **Nuevo** y, posteriormente, **Archivo**.



- 2 Aparecerá una nueva ventana de diálogo; en ella, seleccione dentro de la sección **Categorías** la carpeta **Visual C++**. En la sección **Plantillas** seleccione el icono **Archivo C++** y haga clic en el botón **Abrir**.

- 3 En Visual Studio aparecerá una página en blanco. Es necesario salvarla y, posteriormente, desplazarla al proyecto. Seleccione el menú de **Archivo** y, luego, **Guardar Source1 como...** Aparecerá el clásico diálogo para salvar. Seleccione la carpeta de su proyecto, ábrala. Adentro, aparecerá otra carpeta con el mismo nombre, ábrala también. Una vez adentro, dé como nombre del archivo **main.cpp**

- 4 Para desplazar el archivo, sobre el área de edición, haga clic derecho con el mouse. Aparecerá un menú; dentro de este menú seleccione **Desplazar main.cpp al proyecto** y, luego, haga clic donde aparece su nombre de proyecto.



- 5 Repita la misma operación para crear un archivo llamado **soporte.cpp**
- 6 Siguiendo los mismos pasos, cree un archivo llamado **soporte.h**, el único cambio que realizará es que, en el diálogo de **Nuevo archivo**, deberá seleccionar, en la sección **Plantillas**, el icono de **Archivo de encabezado**.



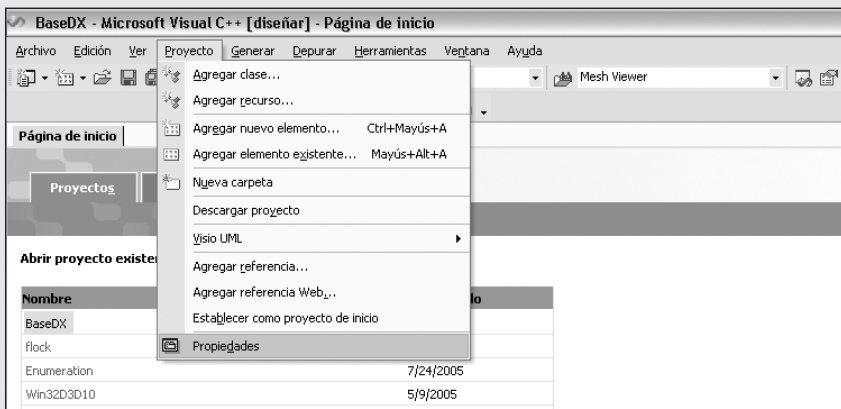
## Configurar el compilador

El proyecto ha sido creado. Ahora, es necesario configurar el compilador para que podamos usar DirectX correctamente. La configuración es muy sencilla.

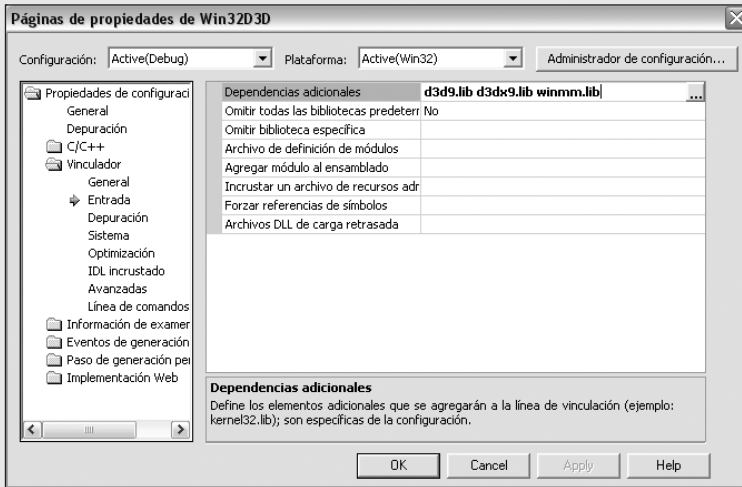
### ■ Configurar Visual Studio

PASO A PASO

- 1 Seleccione el menú **Proyecto** y, luego, **Propiedades de...**, ahí aparecerá el nombre de su proyecto.



- 2 Una nueva ventana de diálogo se abrirá y le permitirá colocar las propiedades de su proyecto. En el lado derecho, busque la carpeta **Vinculador** y, dentro de ésta, haga clic en la opción **Entrada**. Del lado derecho, en la propiedad de **Dependencias adicionales**, escriba: **d3d9.lib d3dx9.lib winmm.lib**.



- 3 Presione el botón **Aceptar** cuando finalice.

## EL CÓDIGO DE LA APLICACIÓN

El compilador ya está configurado y listo para poder compilar aplicaciones que utilicen DirectX 9, también hemos creado en él, nuestro proyecto. Ahora, solamente, queda por delante colocar el código necesario para que funcione nuestra aplicación con esta tecnología.

El código que veremos a continuación ha sido escrito de tal forma que sea fácil de entender y de aplicar. Conforme aumenten nuestros conocimientos de DirectX, es

### III EL EJE Y ES USADO PARA LA APERTURA DE CAMPO.

La apertura de campo está sobre el eje Y, tengamos esto en cuenta para que la aplicación presente el dibujo correctamente. Se puede calcular la apertura en X, tomando la proporción entre las resoluciones de la aplicación. Para convertir grados a radianes, debemos multiplicar el número de grados por el valor de PI y dividirlo entre 180.

posible llevar a cabo optimizaciones e inicializaciones más complejas. Si tenemos algunas dudas, siempre podemos leer los apéndices del libro en los cuales se explican temas complementarios.

Como se mencionó anteriormente, el primer archivo con el que trabajará es **main.cpp**. Éste es el documento principal de nuestra aplicación. Si lo deseamos, podemos abrir el programa de ejemplo de este capítulo.

En el inicio del archivo, encontramos el siguiente código:

```
// Incluimos las funciones de soporte
#include "soporte.h"

// Interfaz, representa el hardware
IDirect3DDevice9* Device=0;

CSoporte soporte; // Objeto de nuestra clase de soporte

// Aquí definimos las estructuras necesarias

// Fin de definición de las estructuras

// Prototipos necesarios
bool DibujaEscena(void);
void CalculaValores(void);
```

En primer lugar, tenemos un **include** para **soporte.h**. Este archivo que se incluye es el que contiene la definición de la clase de soporte, que se encargará de inicializar DirectX para crear la ventana de la aplicación.

El segundo elemento es un apuntador a una interfaz de tipo **IDirect3DDevice9**. Esta interfaz es muy importante, porque representa al dispositivo, por eso, generalmente, se lo llama **Device**, aunque podemos darle el nombre que nos guste. El dispositivo puede ser visto como la tarjeta de video con la que se va a trabajar. Cada vez que sea necesario comunicarnos con la tarjeta de video, utilizaremos esta interfaz.

Posteriormente, creamos un objeto llamado soporte, que es de la clase **CSoporte**, la cual se explicará en detalle posteriormente.

Después de crear este objeto, encontramos una sección delimitada por comentarios. En los próximos capítulos, se usará esta sección para definir estructuras, variables u objetos globales que pueda necesitar la aplicación. En este capítulo, no se utilizará, ya que, solamente, inicializaremos DirectX.

Por último, encontramos dos prototipos. Estas funciones serán utilizadas por la aplicación. La primera se especializa en dibujar la escena, y la segunda es usada para llevar a cabo todos los cálculos necesarios para dibujarla. Los cálculos son cambios de posición de los objetos, colisiones u otras aspectos que sea necesario calcular.

Continuemos con más código:

```
// Esta función inicializa la aplicación
bool InicializaAplicacion()
{

    // Inicia nuestro código de inicialización

    // Fin de nuestro código de inicialización

    // Crea una variable para la matriz
    D3DXMATRIX proj;

    // Crea la matriz de proyección
    D3DXMatrixPerspectiveFovLH(
        &proj,                // Regresa la matriz
        D3DX_PI*0.5,        // 90 grados
        (float)RESX/(float)RESY, // Relacion
        1.0,                // Plano cercano
        1000.0);            // Plano lejano

    // Coloca la matriz de proyección
    Device->SetTransform(D3DTS_PROJECTION, &proj);

    return true;
}
```

La función **InicializaAplicacion()** sirve para inicializar todos los elementos propios de la aplicación. Cuando se requiera inicializar algún elemento como objeto, textura, luz u otros, se hará en esta función.

Lo primero que se encuentra es una sección entre comentarios. En ella se colocarán los diferentes elementos por inicializar.

A continuación, se colocan las características de la matriz de proyección. Podemos imaginarnos que esta matriz contiene alguna de las características de una cámara

virtual que existe dentro de nuestra aplicación y por medio de la cual observamos el mundo que estamos creando.

Cuando se trabaja en DirectX con matrices, por lo general, se cumplen tres pasos:

- crear una variable de tipo matriz para guardar los resultados;
- calcular los valores de la matriz;
- colocar o hacer uso de la matriz.

Aquí vemos, precisamente, esos tres pasos en uso. En primer lugar, creamos una variable de tipo matriz. Esta variable tiene como tipo **D3DMATRIX**, la cual es una estructura que guarda una matriz de 4x4 en su interior. Como la vamos a usar para la matriz de proyección, el nombre que se le ha puesto es **proj**.

Ahora, necesitamos calcular la matriz de proyección. Esto lo hacemos por medio de la función **D3DXMatrixPerspectiveFovLH()**, la cual necesita una serie de parámetros. El primer parámetro necesario es dónde se guardará la matriz calculada, por eso, pasamos por referencia a **proj**.

La apertura de campo de la cámara se indica en el segundo parámetro; se mide de forma vertical, y el valor se da en radianes. Se coloca un valor de  $\pi/2$  radianes que equivale a 90 grados. El valor de  $\pi$  en DirectX puede ser encontrado en **D3DXPI**.

A continuación, se necesita indicar la relación. La relación es la proporción entre la resolución en X y la resolución en Y de la cámara, o la proporción entre la dimensión X y la dimensión Y del **viewport** en la aplicación. Colocar correctamente la relación permite que los objetos se dibujen y se vean naturales. Posteriormente, se verá el lugar donde se definen **RESX** y **RESY**. Se utilizan los **type cast** o flotantes para que el cálculo de la relación se lleve a cabo en forma adecuada.

El próximo parámetro indica la distancia de la cámara al plano cercano del volumen de visión, y el último, indica la distancia de la cámara al plano lejano del volumen de visión. Es decir, definimos entre estas dos distancias lo que vamos a ver. Si un objeto se encuentra más cercano a la cámara que la primera distancia, entonces no se dibuja y, si un objeto se encuentra más lejano que la segunda distancia, tampoco se dibuja. Una vez creada la matriz de proyección, es necesario colocarla para poder usarla. La matriz se coloca en la tarjeta de video, por lo que se usará Device. Para colocar la



## PROBLEMAS CON .DLL AL COMPILAR O EJECUTAR

Es necesario verificar primero que el SDK de DirectX esté correctamente instalado. Si el problema continúa, debemos ir al menú **Generar**, luego seleccionamos **Limpiar** solución. Cuando el compilador termine de limpiar, vamos nuevamente al menú **Generar** y seleccionamos **Volver a generar solución**. Esto debería solucionar el problema.

matriz, se usa la función **SetTransform()**. Esta función permite colocar los diversos tipos de matrices para poder utilizarlos. Tiene dos parámetros. El primero indica el tipo de matriz que se colocará, en este caso es la matriz de proyección y lo indicamos con **D3DTS\_PROJECTION**. El segundo parámetro es la matriz que se colocará, en este caso es **proj** pasada de forma indirecta. Si continuamos con el código, se encuentran otras funciones.

```
void FinalizaAplicacion()
{
    // Aquí colocamos el código necesario para finalizar nuestra aplicación
}

// Procedimiento de windows
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam)
{
    switch(msg)
    {
        case WM_DESTROY:
            ::PostQuitMessage(0);
            break;

        case WM_KEYDOWN:
            if(wParam==VK_ESCAPE)
                ::DestroyWindow(hwnd);
            break;
    }
    return ::DefWindowProc(hwnd,msg,wParam,lParam);
}
```

La función **FinalizaAplicacion()**, se utilizará para llevar a cabo finalizaciones, liberar memoria, cerrar archivos, liberar componentes, etcétera. También, se puede colocar ahí cualquier código que sea necesario ejecutar antes de terminar la aplicación. Después de esta función, se encuentra el procedimiento de Windows. Si no lo conocemos, podemos buscar en los apéndices. Básicamente, este procedimiento toma los mensajes de la aplicación y ejecuta determinado código que dependerá del mensaje obtenido. Si el mensaje es **WM\_DESTROY**, se procede a terminar la aplicación y, si el mensaje es que se oprimió la tecla de **ESCAPE**, entonces se destruye la ventana para que esto lleve a terminar la aplicación.

Se encuentra, también, la función **WinMain()**:

```
// WinMain
int WINAPI WinMain(HINSTANCE hinstance,HINSTANCE prevInstance,PSTR cmdLine,int
    showCmd)
{
    MSG msg;
    ::ZeroMemory(&msg,sizeof(MSG));

    // Mandamos a crear la ventana
    if(!soporte.CreaVentana(hinstance,D3DDEVTYPE_HAL,&Device))
    {
        MessageBox(NULL,"Error al crear la ventana","Error",NULL);
        return 0;
    }

    // Inicializamos la aplicación
    if(!InicializaAplicacion())
    {
        MessageBox(NULL,"Error al inicializar la aplicacion","Error",NULL);
        return 0;
    }

    while(msg.message!=WM_QUIT)
    {
        if(::PeekMessage(&msg,0,0,0,PM_REMOVE))
        {
            ::TranslateMessage(&msg);
            ::DispatchMessage(&msg);
        }
        else
        {
            // Llamamos a nuestra función de calculo
            CalculaValores();

            // Llamamos a nuestra función de dibujo
            DibujaEscena();
        }
    }
}
```

```
// Fuera del ciclo finalizamos la aplicación
FinalizaAplicacion();
// Liberamos el dispositivo

Device->Release();

return msg.wParam;
}
```

De manera similar a la función **main()** en C y C++, los programas de Win32 tienen un punto de inicio, la función **WinMain()**. Es decir, que la aplicación se empieza a ejecutar a partir de esta función y, cuando esta función finaliza, se termina la aplicación. En este momento, veremos sólo los puntos relacionados con DirectX.

Encontramos, dentro de esta función, una invocación a la función **CreaVentana()**, que es un método del objeto soporte. Esta función se encarga de crear la ventana de la aplicación y, a su vez, invoca la inicialización de DirectX.

El primer parámetro es la instancia de la aplicación. Luego, encontramos el tipo de dispositivo. Es posible indicar que el **render** de la aplicación se lleve a cabo por hardware o por software. Si deseamos que se utilice hardware, se coloca como parámetro **D3DDEVTYPE\_HAL**. Si deseamos que el render se lleve a cabo por medio de software, entonces, se coloca **D3DDEVTYPE\_REF** como parámetro.

El **render** por software es mucho más lento que por hardware y, generalmente, sólo se utiliza para hacer pruebas o cuando el dispositivo no tiene alguna característica en particular por hardware y queremos probar cómo funciona.

El último parámetro es el dispositivo. La función **CreaVentana()** regresa un valor, dependiendo de si la creación de la ventana y la inicialización de DirectX fue exitosa o no. Ésta es la razón por la que la colocamos adentro de un **if**. Si no se pudo inicializar o crear la ventana, se manda un mensaje de error al usuario y se finaliza la aplicación.

Ya que se tiene creada la ventana y DirectX es inicializado, entonces, se invoca la función **InicializaAplicacion()**. Recordemos que podemos colocar nuestro propio código dentro de la sección entre comentarios de la función. Si en nuestro propio código alguna inicialización falla, hacemos que la función retorne un **false**, en caso contrario debe retornar un **true**. De forma similar, se verifica si la función fue exitosa o no. Si hubo algún problema, se manda un mensaje al usuario y se finaliza la aplicación. Si todo estuvo bien, entonces, seguiremos con la ejecución del programa.

Continuando con el código, se encuentra un ciclo **while**. Este ciclo se repetirá de manera constante mientras la aplicación esté activa. Independientemente del código Win32 que se tiene, también se invoca la función **CalculaValores()** y, posteriormente, **DibujaEscena()**. Este ciclo se llama **ciclo de simulación**.

Si se recibe el mensaje **WM\_QUIT**, significa que la aplicación debe cerrarse, por lo que el ciclo falla y ya no se repite. Entonces, se invoca la función **FinalizaAplicación()** para llevar a cabo cualquier finalización necesaria.

Como último paso, se libera el dispositivo, y el programa retorna, finalizando con ello la ejecución de la aplicación.

```
void CalculaValores(void)
{

    // Aquí se coloca el código que calcula los valores de nuestra aplicación

}

bool DibujaEscena()
{
    // Verificamos que se tenga el dispositivo
    if(Device)
    {
        Device->Clear(0,0,D3DCLEAR_TARGET|D3DCLEAR_ZBUFFER,0xff606060,1.0f,0);

        // Indicamos que vamos a empezar a dibujar nuestra escena
        Device->BeginScene();

        // Aquí iniciamos nuestro código de dibujo

        // Aquí finaliza el código de dibujo

        // Indicamos que la escena ha finalizado
        Device->EndScene();

        // Se presenta lo dibujado
        Device->Present(0,0,0,0);
    }

    return true;
}
```

La función **CalculaValores()** se utiliza para llevar a cabo cualquier cálculo necesario, mientras la aplicación se ejecute. La función más interesante es **DibujaEscena()**, ya que ésta es la que se encarga de dibujar el mundo virtual que se desea crear.

El primer paso consiste en verificar que se tenga un dispositivo válido; si el dispositivo es válido, entonces, se procede a dibujar.

Usando el dispositivo, se invoca la función **Clear()**. Esta función es usada para limpiar diferentes elementos, de tal forma que el nuevo cuadro que se dibujará esté limpio y no tenga nada del cuadro anterior. Los dos primeros parámetros son avanzados, y los veremos en este capítulo. En el tercer parámetro, indicamos qué es lo que queremos limpiar. En este caso, es el **back buffer**, que es la superficie sobre la que dibujará y el buffer de profundidad. Después de esto, indicamos el color con el que se limpia el back buffer en formato **ARGB**, el color del ejemplo es un tono de gris. En el siguiente parámetro, se coloca el valor con el cual se limpia el buffer de profundidad; colocamos el valor 1.0 para indicar que el buffer queda limpio a su máxima distancia. El último parámetro es el valor con el cual se limpia el **stencil buffer** y constituye un tema más avanzado.

Después de limpiar, se puede dibujar. En DirectX es necesario indicar cuándo se inicia y cuándo se finaliza el dibujo. Por eso, se usan las funciones **BeginScene()** y **EndScene()**. Entre estas dos funciones, va el código que se necesite para dibujar la escena.

Todo se dibuja en el back buffer, por lo que necesitamos mandar el dibujo al **front buffer** para que sea visto en la ventana o pantalla. Para esto, se usa la función **Present()**, pero sus parámetros no serán desarrollados en este momento.

## La clase de soporte

El archivo de **main.cpp** ha finalizado. A continuación, veremos de qué forma la clase de soporte funciona y se define. Primero, definiremos esta clase en el archivo **soporte.h**.

El código de soporte es el siguiente:

```
// Includes necesarios para Direct3D
#include <d3dx9.h>

// Defines para la aplicación
#define RESX 640      // Resolución en X
#define RESY 480      // Resolución en Y

#define VENTANA true  // Ventana o pantalla completa

// Prototipo del procedimiento de windows
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg,WPARAM wParam,LPARAM lParam);

// Creamos una clase de soporte para nuestra aplicacion
class CSoporte
```

```

{

public:

    HWND hwnd;    // Handle de la ventana

    // Esta función crea la ventana
    bool CreaVentana(
        HINSTANCE hInstance,           // Instancia de la aplicación
        D3DDEVTYPE tipoDispositivo,    // Tipo de dispositivo HAL o REF
        IDirect3DDevice9** dispositivo); // Dispositivo creado

    // Esta función inicializa Direct3D
    bool InicializaD3D(
        IDirect3DDevice9** dispositivo, // Dispositivo
        D3DDEVTYPE tipoDispositivo );   // Tipo de dispositivo HAL o REF

};

```

En primer lugar, encontramos el **include** para **d3dx9.h**, esto es necesario para que la aplicación pueda usar correctamente DirectX. En las siguientes líneas, hay dos directivas de preprocesador que utilizamos para definir **RESX** y **RESY** con los valores de 640 y 480. En **RESX**, colocamos el valor de la resolución en X y, en **RESY**, se coloca el valor para la resolución en Y. Con estos valores, se define la resolución de la aplicación. El código de inicialización nos permite crear la aplicación para que trabaje en una ventana o en pantalla completa. Se define **VENTANA** para esto. Si el valor de ventana es **true**, la aplicación se ejecuta dentro de una ventana. Si el valor es **false**, entonces, se ejecuta como pantalla completa.

Después de esto, se tiene el prototipo del procedimiento de Windows.

La definición de la clase soporte viene a continuación. Esta clase tiene un dato de tipo público, **HWND**, y nos permite tener un **handle** de la ventana. Luego, se tienen los prototipos de dos métodos. El primer método se llama **CreaVentana()**, y su función es la de crear la ventana de la aplicación. El segundo método es **InicializaD3D()** y sirve para llevar a cabo la inicialización de DirectX.

La función de **CreaVentana()** tiene tres parámetros. En el primero se pasa la instancia de la aplicación. El segundo es el tipo de dispositivo, ya sea por hardware o por software. El tercero es un apuntador al apuntador del dispositivo por crear.

En la función de **InicializaD3D()**, también tenemos tres parámetros. El primer parámetro es un apuntador al apuntador del dispositivo; en el segundo, se pasa el handle de la ventana y el tercero es el tipo de dispositivo.

Ya que contamos con la definición de la clase ventana, es necesario conocer cómo se implementa. La implementación de la clase se encuentra en el archivo **soporte.cpp**. Dentro de **soporte.cpp**, encontramos código de Win32 y de DirectX. Solamente, veremos el código propio de DirectX sin dar definiciones completas de Win32. Empecemos con el código de **soporte.cpp**:

```
#include "soporte.h"

bool CSoporte::CreaVentana(HINSTANCE hInstance,D3DDEVTYPE tipoDispositivo,
    IDirect3DDevice9** dispositivo)
{

    WNDCLASS wc; // Estructura de la clase ventana
    hwnd=0;      // Handle de la ventana

    // Rellenamos la estructura de la clase ventana
    wc.style      = CS_HREDRAW|CS_VREDRAW;
    wc.lpfWndProc = (WNDPROC)WndProc;      // Procedimiento para el manejo
                                           // de mensajes

    wc.cbClsExtra =0;
    wc.cbWndExtra =0;
    wc.hInstance  =hInstance;              // Coloca la instancia
    wc.hIcon      =LoadIcon(NULL,IDI_WINLOGO);
    wc.hCursor    =LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground =NULL;               // Sin fondo
    wc.lpszMenuName =NULL;                // Sin Menu
    wc.lpszClassName ="Aplicacion";       // Nombre de la clase ventana

    // Se registra la clase ventana y se verifica
    if(!RegisterClass(&wc))
    {
        MessageBox(NULL,"Error al registrar la clase ventana","Error",NULL);
        return false;
    }

    // Crea la ventana
    hwnd::CreateWindow("Aplicacion",      // Nombre de la clase ventana

        "NICOSIO DirectX",              // Titulo de la ventana
        WS_EX_TOPMOST|WS_OVERLAPPEDWINDOW, // Coloca la ventana hasta arriba
```

```

    0,0,                // Posicion de la ventana
    RESX,RESY,         // Tamano de la ventana
    NULL,              // No tiene parent
    NULL,              // No tiene menu
    hInstance,         // La instancia
    NULL);

// Verificamos que la ventana se creara correctamente
if(!hwnd)
{
    MessageBox(NULL,"Error al crear la ventana","Error",NULL);
    return false;
}

::ShowWindow(hwnd,SW_SHOW);    // Mostramos la ventana
::UpdateWindow(hwnd);         // La actualizamos

// Inicializa Direct3D

if(!InicializaD3D(dispositivo,tipoDispositivo))
{
    MessageBox(NULL,"Error al inicializar Direct3D","Error",NULL);
    return false;
}

return true;
}

```

En el inicio del archivo, se tiene un **include** a **soporte.h**; esto es necesario, ya que es preciso tener la definición de la clase para que el compilador sepa a quién pertenecen las funciones que se colocarán a continuación.

Encontramos la función **CreVentana()**, que, prácticamente, es código de Win32. Tenemos la estructura de la clase ventana para la cual creamos una variable llamada **wc**. Se inicializa el handler a 0 por seguridad. Posteriormente, se rellena la estructura y se registra. Si al registrarse hay algún problema, salimos de la función regresando el valor de **false**; esto hará que la aplicación finalice.

Si el registro de la ventana es exitoso, se continúa con la ejecución del programa. Se procede a crear la ventana con las características definidas en la clase ventana recién registrada. La función **CreateWindow()** regresa el handle a la ventana creada. Es necesario, entonces, verificar si la creación fue correcta. Si hubo algún problema al

crear la ventana **hwnd**, tendría el valor de **NULL**. Si esto ocurre, salimos de la función regresando un **false** para que finalice la aplicación.

La siguiente función es **InicializaD3D()**. Veamos los pasos para inicializar DirectX. La forma de inicialización que se presenta en este texto es muy sencilla, pero es suficiente para empezar a aprender sobre DirectX. Conforme se incrementen nuestros conocimientos, podemos crear nuestra propia función de inicialización.

```
bool CSoporte::InicializaD3D(IDirect3DDevice9** dispositivo,D3DDEVTYPE
    tipoDispositivo)
{
    // Variable para guardar el resultado de llamadas a funciones
    HRESULT hr=0;

    // Interfaz a Direct3D, permite encontrar información sobre el hardware
    IDirect3D9* d3d9=0;
    d3d9=Direct3DCreate9(D3D_SDK_VERSION); // Siempre este parámetro

    // Verificamos que la creación sea correcta
    if(!d3d9)
    {
        MessageBox(NULL,"No se pudo crear el objeto IDirect3D9","Error",NULL);
        return false;
    }

    D3DCAPS9 caps; // Esta estructura enumera las capacidades del hardware

    d3d9->GetDeviceCaps(D3DADAPTER_DEFAULT, // Adaptador del cual vamos a leer
        sus capacidades
        tipoDispositivo, // Dispositivo de software o hardware
        &caps); // Estructura donde se guarda

    int vp=0;

    // Verificamos si hay soporte por hardware
    if(caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT)
        vp=D3DCREATE_HARDWARE_VERTEXPROCESSING; // Indicamos proceso por hardware
    else

        vp=D3DCREATE_SOFTWARE_VERTEXPROCESSING; // Indicamos proceso por software
    // Esta estructura se usa para especificar algunas características
```

```

// del device que vamos a crear
D3DPRESENT_PARAMETERS d3dpp;
d3dpp.BackBufferWidth=RESX;           // Ancho del back buffer
d3dpp.BackBufferHeight=RESY;          // Alto del back buffer
d3dpp.BackBufferFormat=D3DFMT_A8R8G8B8; // Pixel format del back buffer
d3dpp.BackBufferCount=1;              // Cantidad de back buffers a usar
d3dpp.MultiSampleType=D3DMULTISAMPLE_NONE; // Tipo de multisampleo
d3dpp.MultiSampleQuality=0;           // Calidad de multisampleo
d3dpp.SwapEffect=D3DSWAPEFFECT_DISCARD; // Como se hace el flipping
d3dpp.hDeviceWindow=hwnd;             // Handle a la ventana
d3dpp.Windowed=VENTANA;               // ventana o pantalla completa
d3dpp.EnableAutoDepthStencil=true;    // Direct3D se encarga del
                                        depth/stencil buffer
d3dpp.AutoDepthStencilFormat=D3DFMT_D24S8; // Formato del depth/stencil
                                        buffer
d3dpp.Flags=0;                        // Caracteristicas adicionales
d3dpp.FullScreen_RefreshRateInHz=D3DPRESENT_RATE_DEFAULT; // Refresh de la
                                        pantalla
d3dpp.PresentationInterval=D3DPRESENT_INTERVAL_IMMEDIATE; // Intervalo
para la presentacion de imagen

hr=d3d9->CreateDevice(
    D3DADAPTER_DEFAULT, // Adaptador del dispositivo a crear -
                        // Adaptador primario
    tipoDispositivo,    // Tipo de dispositivo
    hwnd,               // Handle de la ventana
    vp,                 // Procesamiento de vertices
    &d3dpp,              // Parametros colocados en la estructura
    dispositivo);      // Dispositivo creado

// Verificamos que se haya creado correctamente
if(FAILED(hr))
{

    // Colocamos un buffer de profundidad de 16 bits
    d3dpp.AutoDepthStencilFormat = D3DFMT_D16;

    // Intentamos crear nuevamente el dispositivo
    hr=d3d9->CreateDevice(
        D3DADAPTER_DEFAULT,

```

```

        tipoDispositivo,
        hwnd,
        vp,
        &d3dpp,
        dispositivo);

    // Verificamos si no ha fallado nuevamente
    if(FAILED(hr))
    {
        // Liberamos la interface
        d3d9->Release();
        MessageBox(NULL,"Error al crear el dispositivo","Error",NULL);
        return false;
    }
}

// Ya no necesitamos la interfaz, la liberamos
d3d9->Release();

return true;
}

```

Dentro de la función, encontramos una variable de tipo **HRESULT**. Muchas funciones de DirectX regresan valores de tipo **HRESULT**, y se puede consultar dicho valor para saber si la función fue exitosa o tuvo algún problema al ejecutarse.

Lo primero que necesitamos es obtener una interfaz de tipo **IDirect3D9\***. Esta interfaz permitirá obtener información sobre el dispositivo y llevar a cabo el proceso de inicialización. La interfaz se guarda en la variable **d3d9**, la cual es inicializada a cero por seguridad.

La interfaz se obtiene por medio de la función **Direct3DCreate9()**. Esta función tiene un solo parámetro, el cual siempre va a ser **D3D\_SDK\_VERSION**. Con esto, obtenemos una interfaz de la misma versión que el SDK que se tiene instalado. Es necesario verificar que la interfaz se obtenga correctamente. Si no fuera así, se manda un mensaje de error, y salimos de la función regresando el valor de **false** para que finalice la aplicación.

El próximo paso será verificar que se tenga procesamiento de vértices por hardware. Cuando se obtiene esto, el hardware de la tarjeta de video puede llevar a cabo los cálculos necesarios de las transformaciones y de la iluminación sobre el vértice. Este tipo de procesamiento es mucho más rápido que el realizado por software.

Para poder verificarlo, es necesario obtener las capacidades del hardware. Necesitaremos una variable del tipo **D3DCAPS9**, llamada **caps**. En el interior de esta estructura, se guardarán las capacidades del hardware y, leyendo el contenido de un campo en particular, se podrá saber si el hardware tiene dicha capacidad o no.

La función que se utiliza para obtener las capacidades es **GetDeviceCaps()**. Esta función tiene tres parámetros. El primer parámetro es el adaptador o tarjeta de video del que nos interesa obtener sus capacidades. En este caso, utilizamos **D3DADAPTER\_DEFAULT**, con lo cual indicamos que es el adaptador primario. En el segundo parámetro, se indica si el tipo de dispositivo será por hardware o por software. Por último, el tercer parámetro es la variable de la estructura donde se guardará la información. Se crea una variable de trabajo de tipo entero llamada **vp**. Esta variable se usa únicamente para facilitar el código del programa y, en su interior, se guarda el **flag** que indica el tipo de procesamiento de vértices.

Ahora verificamos el campo **DevCaps** dentro de la estructura de capacidades para ver si tiene **D3DDEVCAPS\_HWTRANSFORMANDLIGHT**, es decir, el procesamiento por hardware para el vértice. Si lo tiene, **vp** obtiene el valor de **D3DCREATE\_HARDWARE\_VERTEXPROCESSING**; de lo contrario, se coloca el flag para procesamiento por software.

A continuación, se crea una variable llamada **d3dpp**, que es del tipo **D3DPRESENT\_PARAMETERS**. Esta estructura guarda los parámetros de presentación, y colocamos, en ella, algunas de las características del dispositivo que se creará.

Se procede a rellenar la estructura. Veamos sus campos básicos. En primer término, se colocamos el ancho y el alto del back buffer. Es conveniente que el back buffer tenga el mismo tamaño que la resolución que usamos para la aplicación. Luego, viene el formato del píxel, donde indicamos cómo se utilizará el color en el píxel. En este caso, son ocho bits para alpha, ocho para rojo, ocho para verde y ocho para azul.

Se indica, también, la cantidad de back buffers por utilizar: para esta aplicación es uno. Los demás campos son avanzados para este momento, por lo que no los explicaremos. Notemos que, también, se pasa el handle a la ventana y si se quiere que la aplicación sea en ventana o en pantalla completa.

Con la estructura de parámetros de presentación llena, ya se puede crear el dispositivo. Para crearlo, se usa la función **CreateDevice()**. Esta función necesita de seis parámetros. El primer parámetro es el dispositivo sobre el cual se creará el dispositivo, en este caso es el primario. Luego, se indica el tipo de dispositivo. El tercer parámetro es el handle a la ventana de la aplicación. Posteriormente, se indica el tipo de procesamiento de vértices, según lo encontrado en las capacidades del hardware. En el quinto parámetro, se pasa la estructura de parámetros de presentación. Para finalizar, el apuntador con el que vamos a referenciar al dispositivo.

Una vez invocada la función, es necesario verificar que el dispositivo se cree correctamente. En este ejemplo, se utilizan características bastante comunes para crearlo. En caso de que no se lograra crear, se intentará, nuevamente, con un dispositivo aún más sencillo.

Si la creación vuelve a fallar, entonces, se libera la interfaz y se retorna con **false** para que la aplicación finalice. Aun si el dispositivo es creado correctamente, procedemos a liberar la interfaz antes de terminar la función, porque ya no es necesaria. Continuaremos trabajando con el dispositivo creado y no con la interfaz.

Si todo se inicializa correctamente, entonces, la función regresa con el valor de **true**. Con esto, finaliza el código del programa de inicialización. Ahora, compilamos la aplicación y la ejecutamos. Aparecerá una ventana que muestra un fondo gris. Aunque en este momento no se vea nada, realmente, es una aplicación de DirectX, y hay un mundo 3D en ella, pero vacío. En los próximos capítulos, aprenderemos cómo colocar objetos en ese mundo 3D y trabajar con ellos.

---

## RESUMEN

Para poder crear aplicaciones de DirectX, es necesario inicializarlo correctamente. La aplicación base que se creó en este capítulo sirve como punto de partida para nuestras propias aplicaciones. La función `InicializaD3D()` se encarga de inicializar DirectX. El código de dibujo debe estar adentro de la función `DibujaEscena()`. Es necesario no olvidarse de finalizar correctamente la aplicación por medio de la función `FinalizaAplicacion()`.



### TEST DE AUTOEVALUACIÓN

**1** ¿Cuáles son los pasos para crear un proyecto en MS Visual C++?

---

**2** ¿Cómo se adicionan archivos al proyecto?

---

**3** ¿De qué forma se colocan dependencias adicionales en un proyecto?

---

**4** ¿Cuáles son los pasos necesarios para usar una matriz en DirectX?

---

**5** ¿Cómo se convierte de grados a radianes?

---

**6** ¿En qué lugar se coloca la resolución de la aplicación?

---

**7** ¿Qué funciones indican el inicio y el fin del código de dibujo?

---

**8** ¿Con qué función se obtienen las capacidades del hardware?

---

**9** ¿Cómo se verifica que se tenga soporte por hardware para los vértices?

---

**10** ¿Dónde se especifican las características del dispositivo por crear?

---

**11** ¿Qué función sirve para crear el dispositivo?

---

**12** ¿Qué se puede hacer si el dispositivo no se crea correctamente?

---