

Programación orientada a eventos

En los computadores personales actualmente lo más típico es trabajar con un interfaz WIMP (Windows, Icons, Menus & Pointer), siendo Microsoft Windows el WIMP más extendido. En estos casos la principal interacción hombre-computador es a través de un GDI (Graphics Device Interface). Pero además hay que responder a los eventos de entrada típicos, como son el teclado y el ratón. Interactuar con un usuario cuando él lo solicite requiere una programación orientada a eventos, pues el comportamiento del programa ya no puede ser totalmente secuencial.

En esta práctica aprenderemos la gestión básica de la interacción hombre-computador en Windows utilizando los objetos gráficos y la gestión de eventos de entrada que nos proporciona las APIs (Application Programming Interface) Win32. Una vez que se aprende a programar sobre una API es muy fácil extrapolar estos conocimientos a otros sistemas, como IOs o Linux.

Objetivo:

Aprender a programar en respuesta a los eventos de un usuario.
Aprender la API Win32.

Además de aprender a mostrar la información al usuario mediante textos y gráficos, se debe aprender a capturar los eventos generados por el hombre a través de los dispositivos de entrada. Indirectamente se aprende a gestionar la resolución de la pantalla, los diferentes tipos de codificación de caracteres, configuración del teclado y comportamiento del ratón. También es necesario enfrentarse con el encapsulamiento de objetos como Ventanas y COMs.

Material:

Tutorial: Learn to Program for Windows in C++
[https://msdn.microsoft.com/es-es/library/windows/desktop/ff381399\(v=vs.85\).aspx](https://msdn.microsoft.com/es-es/library/windows/desktop/ff381399(v=vs.85).aspx)

Visual Studio C++ (cualquier versión).

Desarrollo:

Este tutorial se explicará en clase durante varias sesiones de laboratorio. Los alumnos deberán realizar el tutorial con la asistencia de los profesores de laboratorio. Finalmente se evaluará los conocimientos adquiridos por el alumno mediante una ampliación del tutorial.

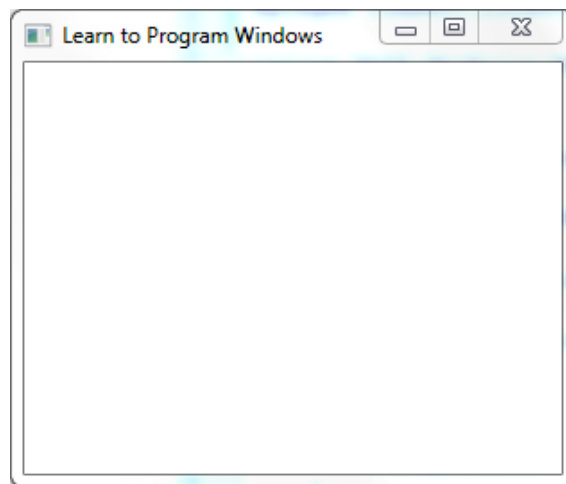
Tema	Descripción
<u>Introducción a la programación en C ++ de Windows</u>	En esta sección se describen algunas de las convenciones terminológicas y codificación básicas utilizadas en la programación de Windows.
<u>Módulo 1. Su primer programa de Windows</u>	En este módulo, se creará un simple programa de Windows que muestra una ventana en blanco.
<u>Módulo 2. El uso de COM en su programa de Windows</u>	Este módulo presenta el modelo de objetos componentes (COM), que subyace en muchas de las modernas API de Windows.
<u>Módulo 3. Gráficos de Windows</u>	Este módulo presenta la arquitectura de gráficos de Windows, con un enfoque en Direct2D.
<u>Módulo 4. Módulo de entrada de usuario</u>	Este módulo describe la entrada del ratón y el teclado.

Nota: A continuación, se proporciona una selección del material de estos tutoriales traducida de forma automática al español, por lo que puede haber errores de traducción.
¡Se agradecen anotaciones de corrección sobre la traducción!

Módulo 1. Primer programa Windows

¿Qué es una ventana?

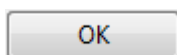
Obviamente, las ventanas son fundamentales para Windows. Son tan importantes que nombraron el sistema operativo. Pero, ¿qué es una ventana? Cuando piensas en una ventana, probablemente pienses en algo como esto:



Captura de pantalla de una ventana de aplicación

Este tipo de ventana se denomina *ventana de aplicación* o *ventana principal*. Normalmente tiene un marco con una barra de título, botones **Minimizar** y **Maximizar** y otros elementos de IU estándar. El marco se denomina *área no cliente* de la ventana, llamada así porque el sistema operativo administra esa parte de la ventana. El área dentro del marco es el *área del cliente*. Esta es la parte de la ventana que administra su programa.

Aquí hay otro tipo de ventana:



Captura de pantalla de una ventana de control

Si es nuevo en la programación de Windows, puede sorprenderle que los controles de la interfaz de usuario, como botones y cuadros de edición, sean ellos mismos ventanas. La principal diferencia entre un control de interfaz de usuario y una ventana de aplicación es que un control no existe por sí mismo. En cambio, el control se coloca en relación con la ventana de la aplicación. Cuando arrastra la ventana de la aplicación, el control se mueve con ella, como era de esperar. Además, el control y la ventana de la aplicación se pueden comunicar entre sí. (Por ejemplo, la ventana de la aplicación recibe notificaciones de clic desde un botón). Por lo tanto, cuando piense en la *ventana*, no piense simplemente en la *ventana de la aplicación*. En cambio, piense en una ventana como una construcción de programación que:

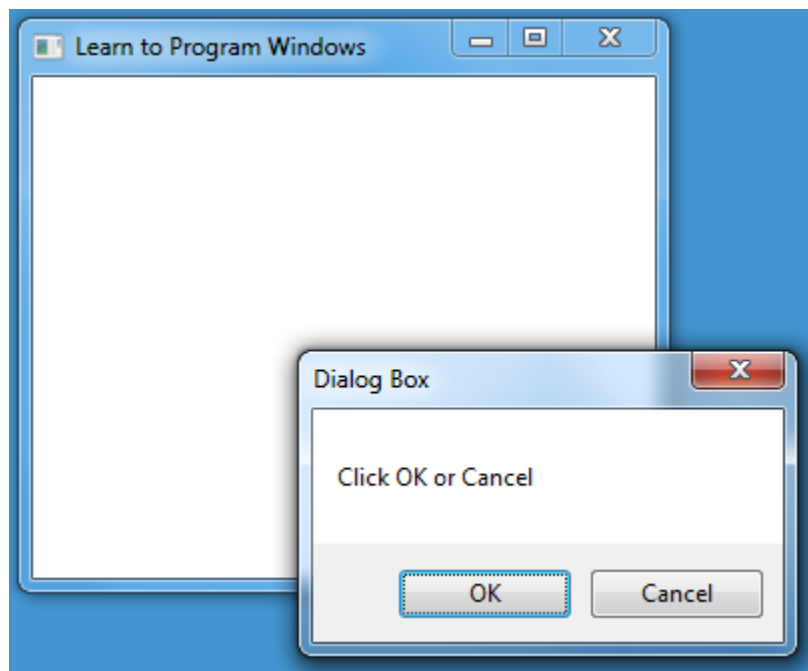
- Ocupa una cierta porción de la pantalla.
- Puede o no ser visible en un momento dado.
- Sabe cómo dibujarse a sí mismo.
- Responde a eventos del usuario o del sistema operativo.

Ventana padre y Ventana propietaria

En el caso de un control de UI, se dice que la ventana de control es el elemento *hija* de la ventana de la aplicación. La ventana de la aplicación es *padre* de la ventana de control. La ventana padre proporciona el sistema de coordenadas utilizado para ubicar una ventana hija. Tener una ventana principal afecta los aspectos de la apariencia de una ventana; por ejemplo, una ventana secundaria está recortada para que ninguna parte de la ventana secundaria pueda aparecer fuera de los límites de su ventana principal.

Otra relación es la relación entre una ventana de aplicación y una ventana de diálogo modal. Cuando una aplicación que muestra un cuadro de diálogo modal, la ventana de aplicación es el *propietario* de la ventana, y el diálogo es una *propiedad* de la ventana. Una ventana propiedad siempre aparece frente a su ventana de propietario. Se oculta cuando el propietario se minimiza y se destruye al mismo tiempo que el propietario.

La siguiente imagen muestra una aplicación que muestra un cuadro de diálogo con dos botones:



Captura de pantalla de una aplicación con un cuadro de diálogo

La ventana de la aplicación es propietaria de la ventana de diálogo. La ventana de diálogo es la ventana padre de ambas ventanas de botón. El siguiente diagrama muestra estas relaciones:

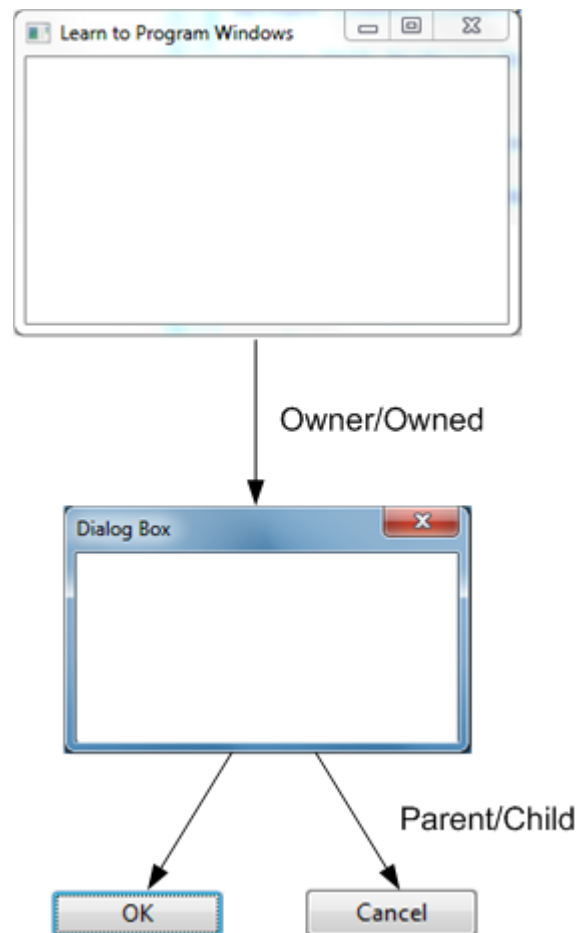


Ilustración que muestra las relaciones padre / hijo y propietario / propiedad

Windows Handles

Windows son objetos, tienen código y datos, pero no son clases de C++. En cambio, un programa hace referencia a una ventana utilizando un valor llamado *handle*. Un handle es un tipo opaco. Esencialmente, es solo un número que el sistema operativo usa para identificar un objeto. Puede imaginar a Windows como teniendo una gran tabla de todas las ventanas que se han creado. Utiliza esta tabla para buscar ventanas por sus handles. El tipo de datos para manejadores de ventanas es **HWND**, que generalmente se pronuncia "aitch-wind". Los identificadores de ventanas los devuelven las funciones que crean ventanas:

CreateWindow y **CreateWindowEx**.

Para realizar una operación en una ventana, típicamente llamará a alguna función que tome un valor **HWND** como parámetro. Por ejemplo, para cambiar la posición de una ventana en la pantalla, llame a la función **MoveWindow**:

```
BOOL MoveWindow (HWND hWnd, int X, int Y, int nWidth, int nHeight, BOOL  
bRepaint);
```

El primer parámetro es el identificador de la ventana que desea mover. Los otros parámetros especifican la nueva ubicación de la ventana y si la ventana debe volver a dibujarse. Tenga en cuenta que los handles no son punteros. Si *hWnd* es una variable que contiene un identificador, intentar desreferenciar el identificador al escribir **hWnd* es un error.

Coordenadas de pantalla y ventana

Las coordenadas se miden en píxeles (pueden ser independientes del dispositivo). Dependiendo de su tarea, puede medir coordenadas relativas a la pantalla, relativas a una ventana (incluido el marco) o relativas al área del cliente de una ventana. Por ejemplo, colocaría una ventana en la pantalla usando las coordenadas de la pantalla, pero dibujaría dentro de una ventana usando las coordenadas del cliente. En cada caso, el origen $(0, 0)$ siempre es la esquina superior izquierda de la región.

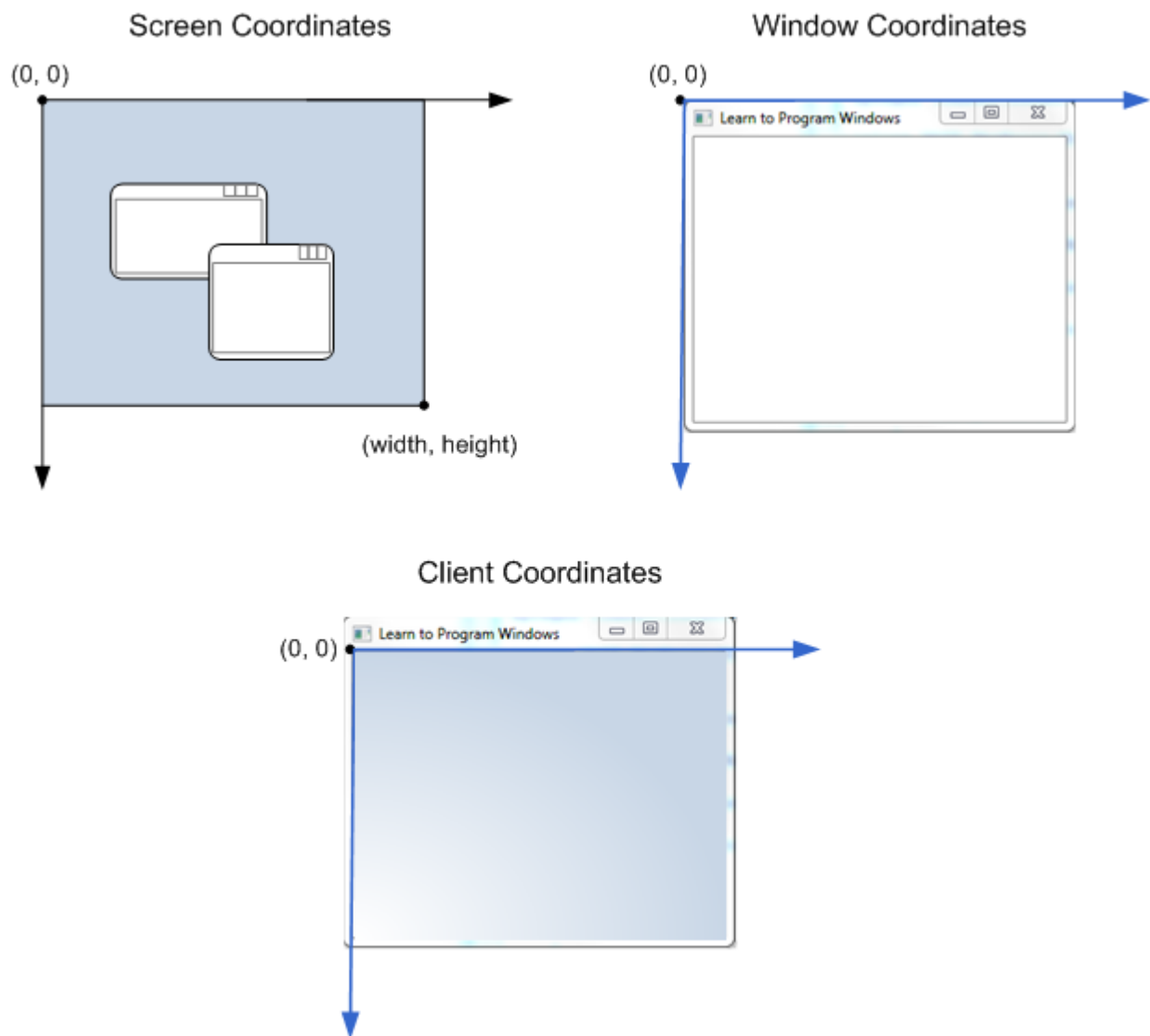


Ilustración que muestra la pantalla, la ventana y las coordenadas del cliente

WinMain: el punto de entrada de la aplicación

Todos los programas de Windows incluyen una función de punto de entrada que se llama **WinMain** o **wWinMain**.

```
int WINAPI wWinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PWSTR  
pCmdLine, int nCmdShow);
```

Los cuatro parámetros son:

- *hInstance* es un "handle a una instancia". El sistema operativo usa este valor para identificar el ejecutable (EXE) cuando se carga en la memoria. El identificador de instancia es necesario para determinadas funciones de Windows, por ejemplo, para cargar iconos o mapas de bits.
- *hPrevInstance* no tiene significado. Se usó en Windows de 16 bits, pero ahora siempre es cero.
- *pCmdLine* contiene los argumentos de línea de comandos como una cadena Unicode.
- *nCmdShow* es un indicador que indica si la ventana principal de la aplicación se minimizará, maximizará o mostrará normalmente.

La función devuelve un valor **int**. El sistema operativo no usa el valor de retorno, pero puede usar el valor de retorno para transmitir un código de estado a algún otro programa.

WINAPI es la convención de llamadas. Una *convención de llamadas* define como una función recibe parámetros de la persona que llama. Por ejemplo, define el orden en que los parámetros aparecen en la pila. Solo asegúrate de declarar tu función **wWinMain** como se muestra.

La función **WinMain** es idéntica a **wWinMain**, excepto que los argumentos de línea de comandos se pasan como una cadena ANSI. La versión Unicode es preferida. Puede usar la función ANSI **WinMain** incluso si compila su programa como Unicode. Para obtener una copia Unicode de los argumentos de la línea de comandos, llame a la función [GetCommandLine](#). Esta función devuelve todos los argumentos en una sola cadena. Si desea los argumentos como una matriz de estilo *argv*, pase esta cadena a [CommandLineToArgvW](#).

¿Cómo sabe el compilador invocar **wWinMain** en lugar de la función **principal** estándar? Lo que sucede en realidad es que la biblioteca de tiempo de ejecución de Microsoft C (CRT) proporciona una implementación de **main** que llama a **WinMain** o **wWinMain**.

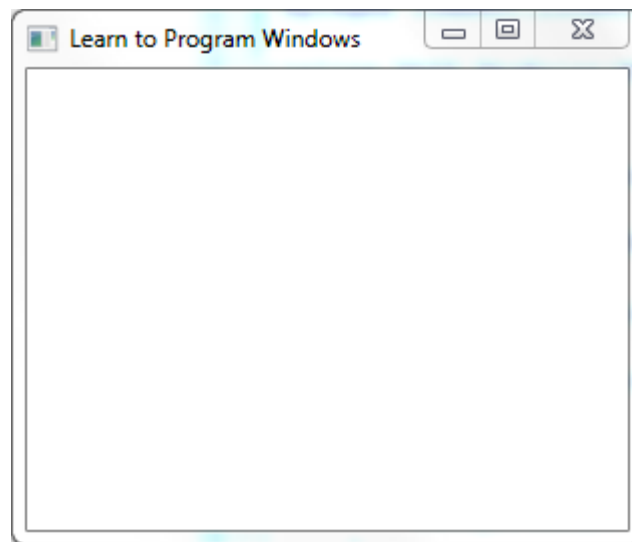
Nota: el CRT hace un trabajo adicional dentro de **main**. Por ejemplo, cualquier inicializador estático se llama antes de **wWinMain**. Aunque puede decirle al vinculador que use una función de punto de entrada diferente, use el valor predeterminado si se vincula al CRT. De lo contrario, se omitirá el código de inicialización CRT, con resultados impredecibles. (Por ejemplo, los objetos globales no se inicializarán correctamente).

Aquí hay una función **WinMain** vacía.

```
INT WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,  
PSTR lpCmdLine, INT nCmdShow)  
{  
    return 0;  
}
```

Ahora que tiene el punto de entrada y comprende algunas de las convenciones básicas de terminología y codificación, está listo para crear un programa de Ventana completo.

En el siguiente código escribiremos un programa mínimo de Windows. Todo lo que hace es crear y mostrar una ventana en blanco. Este primer programa contiene aproximadamente 50 líneas de código, sin contar líneas en blanco y comentarios. Será nuestro punto de partida; más adelante agregaremos gráficos, texto, entrada de usuario y otras características.



Captura de pantalla del programa de ejemplo

Se debe crear un nuevo proyecto del tipo: **Visual C++ > General > Proyecto vacío**. Después, en *Archivos de Origen* se debe **agregar un nuevo elemento .cpp** con el siguiente código:

```
#ifndef UNICODE
#define UNICODE
#endif

#include <windows.h>

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int nCmdShow)
{
    // Register the window class.
    const wchar_t CLASS_NAME[] = L"Sample Window Class";

    WNDCLASS wc = {};

    wc.lpfnWndProc = WindowProc;
    wc.hInstance = hInstance;
    wc.lpszClassName = CLASS_NAME;

    RegisterClass(&wc);

    // Create the window.

    HWND hwnd = CreateWindowEx(
```



```

    0, // Optional window styles.
    CLASS_NAME, // Window class
    L"Learn to Program Windows", // Window text
    WS_OVERLAPPEDWINDOW, // Window style

    // Size and position
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL, // Parent window
    NULL, // Menu
    hInstance, // Instance handle
    NULL // Additional application data
);

if (hwnd == NULL)
{
    return 0;
}

ShowWindow(hwnd, nCmdShow);

// Run the message loop.

MSG msg = {};
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

return 0;
}

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;

        case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hwnd, &ps);

            FillRect(hdc, &ps.rcPaint, (HBRUSH)(COLOR_WINDOW + 1));

            EndPaint(hwnd, &ps);
        }
        return 0;
    }
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}

```

Un resumen de lo que hace este código:

1. **wWinMain** es el punto de entrada del programa. Cuando se inicia el programa, registra cierta información sobre el comportamiento de la ventana de la aplicación. Uno de los elementos más importantes en este ejemplo es la dirección de una función, llamada

WindowProc. Esta función define el comportamiento de la ventana: su aspecto, cómo interactúa con el usuario, etc.

2. A continuación, el programa crea la ventana y recibe un identificador que identifica de forma única la ventana.
3. Si se crea la ventana, el programa entra en un **bucle while**. El programa permanece en este ciclo hasta que el usuario cierra la ventana y sale de la aplicación.

Observe que el programa no llama explícitamente a la función WindowProc, aunque dijimos que aquí es donde se define la mayor parte de la lógica de la aplicación. Windows se comunica con su programa pasándole una serie de *mensajes*. El código dentro del bucle **while** se encarga de ello. Cada vez que el programa llama a la función [DispatchMessage](#), provoca indirectamente que Windows invoque la función WindowProc, una vez para cada mensaje.

Clases de ventana

Una *clase de ventana* define un conjunto de comportamientos que varias ventanas pueden tener en común. Por ejemplo, en un grupo de botones, cada botón tiene un comportamiento similar cuando el usuario hace clic en el botón. Por supuesto, los botones no son completamente idénticos; cada botón muestra su propia cadena de texto y tiene sus propias coordenadas de pantalla. Los datos que son únicos para cada ventana se llaman *datos de instancia*.

Cada ventana debe estar asociada a una clase de ventana, incluso si su programa solo crea una instancia de esa clase. Es importante entender que una clase ventana no es una "clase" en el sentido C++. Más bien, es una estructura de datos utilizada internamente por el sistema operativo. Las clases de ventana se registran en el sistema en tiempo de ejecución. Para registrar una nueva clase de ventana, comience rellenando una estructura **WNDCLASS**:

```
// Registra la clase de ventana.
const wchar_t CLASS_NAME [] = L "clase de ventana de muestra";

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;
wc.hInstance = hInstance;
wc.lpszClassName = CLASS_NAME;
```

Debe establecer los siguientes miembros de estructura:

- **lpfnWndProc** es un puntero a una función definida por la aplicación llamada el *procedimiento de ventana* o "ventana de proceso". El procedimiento de ventana define la mayor parte del comportamiento de la ventana. Examinaremos el procedimiento de ventana en detalle más adelante. Por ahora, solo trata esto como una referencia directa.
- **hInstance** es el manejador de la instancia de la aplicación. Obtenga este valor del parámetro *hInstance* de **wWinMain**.
- **lpszClassName** es una cadena que identifica la clase de ventana.

Los nombres de las clases son locales al proceso actual, por lo que el nombre solo debe ser exclusivo dentro del proceso. Sin embargo, los controles estándar de Windows también tienen clases. Si usa alguno de esos controles, debe elegir nombres de clases que no entren en conflicto.

con los nombres de clase de control. Por ejemplo, la clase de ventana para el control del botón se llama "Botón".

La estructura [WNDCLASS](#) tiene otros miembros que no se muestran aquí. Puede establecerlos en cero, como se muestra en este ejemplo, o completarlos. La documentación de MSDN describe la estructura en detalle.

A continuación, pase la dirección de la estructura [WNDCLASS](#) a la función [RegisterClass](#). Esta función registra la clase de ventana en el sistema operativo.

```
RegisterClass (& wc);
```

Creando la ventana

Para crear una nueva instancia de una ventana, llame a la función [CreateWindowEx](#) :

```
HWND hwnd = CreateWindowEx (
    0, // estilos de ventana opcionales.
    CLASS_NAME, // clase de ventana
    L "Aprenda a programar Windows", // texto de la ventana
    WS_OVERLAPPEDWINDOW, // Estilo de ventana

    // Tamaño y posición
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL, // ventana principal
    NULL, // Menú
    hInstance, // Identificador de instancia
    NULL // Datos de aplicación adicionales
);

if (hwnd == NULL)
{
    return 0;
}
```

Puede leer descripciones detalladas de los parámetros en MSDN, pero aquí hay un resumen rápido:

- El primer parámetro le permite especificar algunos comportamientos opcionales para la ventana (por ejemplo, ventanas transparentes). Establezca este parámetro en cero para los comportamientos predeterminados.
- CLASS_NAME es el nombre de la clase de ventana. Esto define el tipo de ventana que está creando.
- El texto de la ventana se usa de diferentes formas por diferentes tipos de ventanas. Si la ventana tiene una barra de título, el texto se muestra en la barra de título.
- El estilo de ventana es un conjunto de indicadores que definen parte del aspecto de una ventana. La constante WS_OVERLAPPEDWINDOW es en realidad varias banderas combinadas con un **OR** bit a bit. Juntas, estas banderas dan a la ventana una barra de título, un borde, un menú del sistema y los botones **Minimizar** y **Maximizar**. Este conjunto de indicadores es el estilo más común para una ventana de aplicación de nivel superior.
- Para posición y tamaño, la constante CW_USEDEFAULT significa usar valores predeterminados.

- El siguiente parámetro establece una ventana principal o una ventana de propietario para la nueva ventana. Establezca el elemento primario si está creando una ventana secundaria. Para una ventana de nivel superior, establezca esto en NULL.
- Para una ventana de aplicación, el siguiente parámetro define el menú para la ventana. Este ejemplo no usa un menú, por lo que el valor es NULL.
- *hInstance* es el identificador de instancia, descrito anteriormente.
- El último parámetro es un puntero a datos arbitrarios de tipo **void ***. Puede usar este valor para pasar una estructura de datos a su procedimiento de ventana.
- **CreateWindowEx** devuelve un identificador a la nueva ventana, o cero si la función falla. Para mostrar la ventana, es decir, hacer que la ventana sea visible, pase el control de la ventana a la función **ShowWindow** :

```
ShowWindow (hwnd, nCmdShow);
```

donde:

- El parámetro *hwnd* es el manejador de ventana devuelto por **CreateWindowEx** .
- El parámetro *nCmdShow* se puede usar para minimizar o maximizar una ventana. El sistema operativo pasa este valor al programa a través de la función **wWinMain**.

Aquí está el código completo para crear la ventana. Recuerde que WindowProc es solo una declaración de una función.

```
// Registra la clase de ventana.
const wchar_t CLASS_NAME [] = L "clase de ventana de muestra";

WNDCLASS wc = {};

wc.lpfnWndProc = WindowProc;
wc.hInstance = hInstance;
wc.lpszClassName = CLASS_NAME;

RegisterClass (& wc);

// Crea la ventana.

HWND hwnd = CreateWindowEx (
    0, // estilos de ventana opcionales.
    CLASS_NAME, // clase de ventana
    L "Aprenda a programar Windows", // texto de la ventana
    WS_OVERLAPPEDWINDOW, // Estilo de ventana

    // Tamaño y posición
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    NULL, // ventana principal
    NULL, // Menú
    hInstance, // Identificador de instancia
    NULL // Datos de aplicación adicionales
);

if (hwnd == NULL)
{
    return 0;
}
```

```
ShowWindow (hwnd, nCmdShow);
```

¡Felicitaciones, has creado una ventana! En este momento, la ventana no contiene ningún contenido ni interactúa con el usuario. En una aplicación GUI real, la ventana respondería a los eventos del usuario y del sistema operativo. La siguiente sección describe cómo los mensajes de ventana proporcionan este tipo de interactividad.

Mensajes en Windows

Una aplicación GUI debe responder a los eventos del usuario y del sistema operativo.

- **Los eventos del usuario** incluyen todas las formas en que alguien puede interactuar con su programa: clics del mouse, pulsaciones de tecla, gestos de pantalla táctil, etc.
- **Los eventos del sistema operativo** incluyen cualquier cosa "externa" al programa que pueda afectar la forma en que se comporta el programa. Por ejemplo, el usuario puede conectar un nuevo dispositivo de hardware, o Windows puede entrar en un estado de menor consumo (suspensión o hibernación).

Estos eventos pueden ocurrir en cualquier momento mientras el programa se está ejecutando, en casi cualquier orden. ¿Cómo se estructura un programa cuyo flujo de ejecución no puede predecirse por adelantado?

Para resolver este problema, Windows usa un modelo de paso de mensajes. El sistema operativo se comunica con la ventana de su aplicación al pasarle mensajes. Un mensaje es simplemente un código numérico que designa un evento en particular. Por ejemplo, si el usuario presiona el botón izquierdo del mouse, la ventana recibe un mensaje con el siguiente código de mensaje.

```
#define WM_LBUTTONDOWN 0x0201
```

Algunos mensajes tienen datos asociados a ellos. Por ejemplo, el mensaje **WM_LBUTTONDOWN** incluye la coordenada x y la coordenada y del cursor del mouse. Para pasar un mensaje a una ventana, el sistema operativo llama al procedimiento de ventana registrado para esa ventana. (Y ahora sabes para qué sirve el procedimiento de ventana).

El bucle de mensajes

Una aplicación recibirá miles de mensajes mientras se ejecuta. (Tenga en cuenta que cada clic de tecla y botón de mouse genera un mensaje.) Además, una aplicación puede tener varias ventanas, cada una con su propio procedimiento de ventana. ¿Cómo recibe el programa todos estos mensajes y los envía al procedimiento de ventana correcto? La aplicación necesita un bucle para obtener los mensajes y enviarlos a las ventanas correctas.

Para cada hilo que crea una ventana, el sistema operativo crea una cola para los mensajes de la ventana. Esta cola contiene mensajes para todas las ventanas que se crean en ese hilo. La cola en sí está oculta de su programa. No puede manipular la cola directamente, pero puede extraer un mensaje de la cola llamando a la función **GetMessage**.

```
MSG msg;  
GetMessage (&msg, NULL, 0, 0);
```

Esta función elimina el primer mensaje del jefe de la cola. Si la cola está vacía, la función se bloquea hasta que otro mensaje se ponga en cola (pero el programa no deja de responder). Si no hay mensajes, el programa no tiene nada que hacer.

El primer parámetro de **GetMessage** es la dirección de una estructura **MSG**. Si la función tiene éxito, rellena la estructura **MSG** con información sobre el mensaje, incluida la ventana de destino y el código del mensaje. Los otros tres parámetros le dan la capacidad de filtrar qué mensajes obtiene de la cola. En casi todos los casos, establecerá estos parámetros a cero.

Aunque la estructura **MSG** contiene información sobre el mensaje, casi nunca examinará esta estructura directamente. En cambio, lo pasará directamente a otras dos funciones.

```
TranslateMessage(&msg);  
DispatchMessage(&msg);
```

La función **TranslateMessage** está relacionada con la entrada del teclado; traduce las pulsaciones de teclas (tecla hacia abajo, tecla arriba) en los caracteres. Realmente no necesita saber cómo funciona esta función; solo recuerde llamarlo justo antes de **DispatchMessage**. El enlace a la documentación de MSDN le dará más información, si tiene curiosidad.

La función **DispatchMessage** le dice al sistema operativo que llame al procedimiento de ventana de la ventana que es el destino del mensaje. En otras palabras, el sistema operativo busca el identificador de ventana en su tabla de ventanas, encuentra el puntero de función asociado con la ventana e invoca la función.

Por ejemplo, supongamos que el usuario presiona el botón izquierdo del mouse. Esto causa una cadena de eventos:

1. El sistema operativo coloca un mensaje **WM_LBUTTONDOWN** en la cola de mensajes.
2. Su programa llama a la función **GetMessage**.
3. **GetMessage** extrae el mensaje **WM_LBUTTONDOWN** de la cola y rellena la estructura **MSG**.
4. Su programa llama a las funciones **TranslateMessage** y **DispatchMessage**.
5. Dentro de **DispatchMessage**, el sistema operativo llama a su procedimiento de ventana.
6. Su procedimiento de ventana puede responder al mensaje o ignorarlo.

Cuando el procedimiento de ventana vuelve, vuelve a **DispatchMessage**, que regresa al bucle de mensajes para el siguiente mensaje. Mientras su programa se esté ejecutando, los mensajes continuarán llegando a la cola. Por lo tanto, necesita un bucle que extraiga continuamente mensajes de la cola y los despache. Puedes pensar en el ciclo haciendo lo siguiente:

```
// ADVERTENCIA: No escribas tu loop de esta manera.  
  
while (1)  
{  
    GetMessage(&msg, NULL, 0, 0);  
    TranslateMessage(&msg);  
    DispatchMessage(&msg);  
}
```

Como está escrito, por supuesto, este ciclo nunca terminaría. Ahí es donde entra el valor de retorno de la función **GetMessage**. Normalmente, **GetMessage** devuelve un valor distinto de

cero. Siempre que desee salir de la aplicación y salir del bucle de mensajes, simplemente llame a la función [PostQuitMessage](#) .

```
PostQuitMessage (0);
```

La función [PostQuitMessage](#) coloca un mensaje [WM_QUIT](#) en la cola de mensajes. [WM_QUIT](#) es un mensaje especial: hace que [GetMessage](#) devuelva cero, señalando el final del ciclo de mensajes. Aquí está el ciclo de mensajes revisado.

```
// Correct.

MSG msg = { };
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Mientras [GetMessage](#) devuelve un valor distinto de cero, la expresión en el **while** se evalúa como verdadera. Después de llamar a [PostQuitMessage](#) , la expresión se convierte en falsa y el programa rompe el ciclo. (Una consecuencia interesante de este comportamiento es que su procedimiento de ventana nunca recibe un mensaje [WM_QUIT](#) , por lo que no necesita una declaración de caso para este mensaje en su procedimiento de ventana).

Mensajes publicados versus mensajes enviados

La sección anterior hablaba sobre los mensajes que van a una cola. En algunas situaciones, el sistema operativo llamará directamente a un procedimiento de ventana, evitando la cola. La terminología para esta distinción puede ser confusa:

- *Publicar* un mensaje significa que el mensaje va en la cola de mensajes y se envía a través del bucle de mensajes ([GetMessage](#) y [DispatchMessage](#)).
- *Enviar* un mensaje significa que el mensaje se salta la cola, y el sistema operativo llama directamente al procedimiento de ventana.

Por ahora, la distinción no es muy importante. El procedimiento de ventana maneja todos los mensajes, pero algunos mensajes omiten la cola e ir directamente al procedimiento de su ventana. Sin embargo, puede marcar la diferencia si su aplicación se comunica entre ventanas. Puede encontrar una discusión más detallada sobre este tema en el tema [Acerca de mensajes y colas de mensajes](#) .

El Procedimiento de Ventana (WindowProc)

La función [DispatchMessage](#) llama al procedimiento de ventana de la ventana que es el destino del mensaje. El procedimiento de ventana tiene la siguiente firma.

```
LRESULT CALLBACK WindowProc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```

Hay cuatro parámetros:

- *hwnd* es un handle para la ventana.
- *uMsg* es el código del mensaje; por ejemplo, el mensaje [WM_SIZE](#) indica que la ventana fue redimensionada.
- *wParam* y *lParam* contienen datos adicionales que pertenecen al mensaje. El significado exacto depende del código del mensaje.

LRESULT es un valor entero que su programa devuelve a Windows. Contiene la respuesta de su programa a un mensaje en particular. El significado de este valor depende del código del mensaje. **CALLBACK** es la convención de llamadas para la función.

Un procedimiento de ventana típico es simplemente una declaración de *switch* grande que maneja el código del mensaje. Agregue casos para cada mensaje que quiera manejar.

```
switch (uMsg)
{
case WM_SIZE: // Handle window resizing

// etc

}
```

Los datos adicionales para el mensaje se encuentran en los parámetros *lParam* y *wParam*. Ambos parámetros son valores enteros del tamaño de un ancho de puntero (32 bits o 64 bits). El significado de cada uno depende del código del mensaje (*uMsg*). Para cada mensaje, deberá buscar el código del mensaje en MSDN y convertir los parámetros al tipo de datos correcto. Por lo general, los datos son un valor numérico o un puntero a una estructura. Algunos mensajes no tienen datos.

Por ejemplo, la documentación del mensaje [WM_SIZE](#) indica que:

- *wParam* es una bandera que indica si la ventana fue minimizada, maximizada o redimensionada.
- *lParam* contiene el nuevo ancho y alto de la ventana como valores de 16 bits empaquetados en un número de 32 o 64 bits. Deberá realizar algunos cambios de bit para obtener estos valores. Afortunadamente, el archivo de encabezado WinDef.h incluye macros de ayuda que hacen esto.

Un procedimiento de ventana típico maneja docenas de mensajes, por lo que puede crecer bastante tiempo. Una forma de hacer que su código sea más modular es poner la lógica para manejar cada mensaje en una función separada. En el procedimiento de ventana, *coloque*

los parámetros *wParam* y *lParam* en el tipo de datos correcto y pase esos valores a la función. Por ejemplo, para manejar el mensaje **WM_SIZE**, el procedimiento de ventana se vería así:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_SIZE:
            {
                // Macro to get the low-order word.
                int width = LOWORD(lParam);
                // Macro to get the high-order word.
                int height = HIWORD(lParam);
                // Respond to the message:
                OnSize(hwnd, (UINT)wParam, width, height);
            }
            break;
    }
}

void OnSize(HWND hwnd, UINT flag, int width, int height);
{
    // Handle resizing
}
```

Las macros **LOWORD** e **HIWORD** obtienen los valores de ancho y alto de 16 bits de *lParam*. El procedimiento de ventana extrae el ancho y el alto, y luego pasa estos valores a la función *OnSize*.

Manejo de mensajes predeterminado

Si no maneja un mensaje particular en su procedimiento de ventana, pase los parámetros del mensaje directamente a la función **DefWindowProc**. Esta función realiza la acción predeterminada para el mensaje, que varía según el tipo de mensaje.

```
return DefWindowProc (hwnd, uMsg, wParam, lParam);
```

Evitar los cuellos de botella en su ventana Procedimiento

Mientras su procedimiento de ventana se ejecuta, bloquea cualquier otro mensaje para las ventanas creadas en el mismo subproceso. Por lo tanto, evite el procesamiento prolongado dentro de su procedimiento de ventana. Por ejemplo, supongamos que su programa abre una conexión TCP y espera indefinidamente para que el servidor responda. Si lo hace dentro del procedimiento de ventana, su IU no responderá hasta que la solicitud se complete. Durante ese tiempo, la ventana no puede procesar la entrada del mouse o del teclado, volver a pintarse o incluso cerrarse.

En su lugar, debe mover el trabajo a otro subproceso, utilizando una de las funciones de multitarea integradas en Windows:

- Crea un nuevo hilo.
- Use un grupo de hilos.
- Use llamadas de E / S asincrónicas.
- Use llamadas a procedimientos asincrónicos.

Pintar en la ventana

Has creado tu ventana. Ahora quieres mostrar algo dentro de él. En la terminología de Windows, esto se llama pintar la ventana. Para mezclar metáforas, una ventana es un lienzo en blanco, esperando que la llene.

A veces, su programa iniciará la pintura para actualizar la apariencia de la ventana. En otras ocasiones, el sistema operativo le notificará que debe repintar una parte de la ventana. Cuando esto ocurre, el sistema operativo envía a la ventana un mensaje **WM_PAINT**. La parte de la ventana que debe pintarse se llama *región de actualización*.

La primera vez que se muestra una ventana, se debe pintar toda el área del cliente de la ventana. Por lo tanto, siempre recibirá al menos un mensaje **WM_PAINT** cuando muestre una ventana.

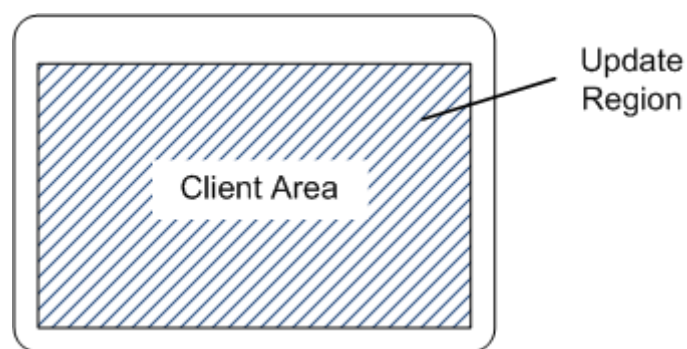


Ilustración que muestra la región de actualización de una ventana

Usted solo es responsable de pintar el área del cliente. El marco de trabajo circundante, incluida la barra de título, es pintado automáticamente por el sistema operativo. Después de que termine de pintar el área del cliente, borre la región de actualización, que le dice al sistema operativo que no necesita enviar otro mensaje **WM_PAINT** hasta que algo cambie.

Ahora suponga que el usuario mueve otra ventana para oscurecer una parte de su ventana. Cuando la parte oscurecida vuelve a ser visible, esa parte se agrega a la región de actualización y su ventana recibe otro mensaje **WM_PAINT**.

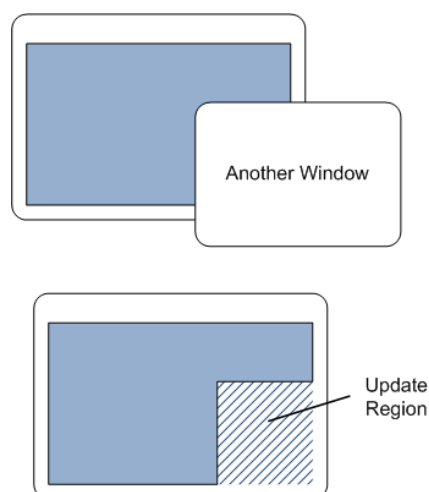


Ilustración que muestra cómo cambia la región de actualización cuando se superponen dos ventanas

La región de actualización también cambia si el usuario extiende la ventana. En el siguiente diagrama, el usuario extiende la ventana hacia la derecha. El área recién expuesta en el lado derecho de la ventana se agrega a la región de actualización:

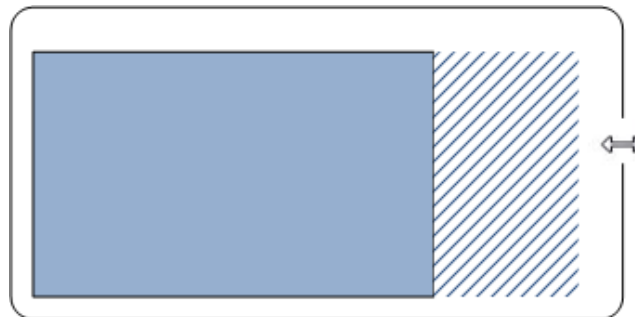


Ilustración que muestra cómo cambia la región de actualización cuando se cambia el tamaño de una ventana

En nuestro primer programa de ejemplo, la rutina de pintura es muy simple. Simplemente llena toda el área del cliente con un color sólido. Aun así, este ejemplo es suficiente para demostrar algunos de los conceptos importantes.

```
switch (uMsg)
{
case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);

        // All painting occurs here, between BeginPaint and EndPaint.

        FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));

        EndPaint(hwnd, &ps);
    }
    return 0;
}
```

Comience la operación de pintura llamando a la función **BeginPaint**. Esta función rellena la estructura **PAINTSTRUCT** con información sobre la solicitud de repintado. La región de actualización actual se da en el **rcPaint** miembro del **PAINTSTRUCT**. Esta región de actualización se define en relación con el área del cliente:

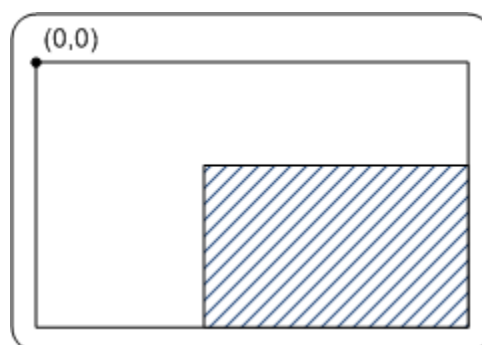


Ilustración que muestra el origen del área del cliente

En su código de pintura, tiene dos opciones básicas:

- Pinte toda el área del cliente, independientemente del tamaño de la región de actualización. Todo lo que quede fuera de la región de actualización queda recortado. Es decir, el sistema operativo lo ignora.
- Optimice pintando solo la parte de la ventana dentro de la región de actualización.

Si siempre pintas toda el área del cliente, el código será más simple. Sin embargo, si tiene una lógica de pintura complicada, puede ser más eficiente omitir las áreas fuera de la región de actualización.

La siguiente línea de código llena la región de actualización con un solo color, utilizando el color de fondo de la ventana definida por el sistema (COLOR_WINDOW). El color real indicado por COLOR_WINDOW depende del esquema de color actual del usuario.

```
FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
```

Los detalles de **FillRect** no son importantes para este ejemplo, pero el segundo parámetro proporciona las coordenadas del rectángulo a llenar. En este caso, se pasa en toda la región de actualización (la **rcPaint** miembro del **PAINTSTRUCT**). En el primer mensaje **WM_PAINT**, se debe pintar toda el área del cliente, por lo que **rcPaint** contendrá todo el área del cliente. En los siguientes mensajes **WM_PAINT**, **rcPaint** puede contener un rectángulo más pequeño.

La función **FillRect** es parte de la Interfaz de Dispositivo de Gráficos (GDI), que tiene gráficos de Windows activos durante mucho tiempo. En Windows 7, Microsoft presentó un nuevo motor de gráficos, llamado Direct2D, que admite operaciones de gráficos de alto rendimiento, como la aceleración de hardware. Cuando haya terminado de pintar, llame a la función **EndPaint**. Esta función borra la región de actualización, que indica a Windows que la ventana se ha terminado de pintar.

Cerrando la ventana

Cuando el usuario cierra una ventana, esa acción desencadena una secuencia de mensajes de la ventana. El usuario puede cerrar una ventana de la aplicación haciendo clic en el botón **Cerrar**, o usando un atajo de teclado como ALT + F4. Cualquiera de estas acciones hace que la ventana reciba un mensaje **WM_CLOSE**. El mensaje **WM_CLOSE** le da la oportunidad de avisar al usuario antes de cerrar la ventana. Si realmente desea cerrar la ventana, llame a la función **DestroyWindow**. De lo contrario, simplemente devuelva cero desde el mensaje **WM_CLOSE**, y el sistema operativo ignorará el mensaje y no destruirá la ventana.

Aquí hay un ejemplo de cómo un programa podría manejar **WM_CLOSE**.

```
case WM_CLOSE:
    if (MessageBox(hwnd, L"Really quit?", L"My application",
MB_OKCANCEL) == IDOK)
    {
        DestroyWindow(hwnd);
    }
    // Else: User canceled. Do nothing.
    return 0;
```

En este ejemplo, la función **MessageBox** muestra un diálogo modal que contiene los botones **Aceptar** y **Cancelar** . Si el usuario hace clic en **Aceptar** , el programa llama a **DestroyWindow**. De lo contrario, si el usuario hace clic en **Cancelar**, la llamada a **DestroyWindow** se omite y la ventana permanece abierta. En cualquier caso, devuelve cero para indicar que manejó el mensaje.

Si desea cerrar la ventana sin preguntar al usuario, simplemente puede llamar a **DestroyWindow** sin la llamada a **MessageBox**. Sin embargo, hay un atajo en este caso. Recuerde que **DefWindowProc** ejecuta la acción predeterminada para cualquier mensaje de ventana. En el caso de **WM_CLOSE** , **DefWindowProc** llama automáticamente **DestroyWindow**. Esto significa que si ignoras el mensaje **WM_CLOSE**, la ventana se destruye de manera predeterminada.

Cuando una ventana está a punto de destruirse, recibe un mensaje **WM_DESTROY** . Este mensaje se envía después de que la ventana se elimina de la pantalla, pero antes de que se produzca la destrucción (en particular, antes de que se destruyan las ventanas secundarias). En la ventana principal de su aplicación, normalmente responderá a **WM_DESTROY** llamando a **PostQuitMessage**.

```
caso WM_DESTROY:  
    PostQuitMessage (0);  
    return 0;
```

Vimos en la sección **Mensajes de la ventana** que **PostQuitMessage** coloca un mensaje **WM_QUIT** en la cola de mensajes, lo que hace que el ciclo de mensaje finalice. Aquí hay un diagrama de flujo que muestra la forma típica de procesar los mensajes **WM_CLOSE** y **WM_DESTROY** :

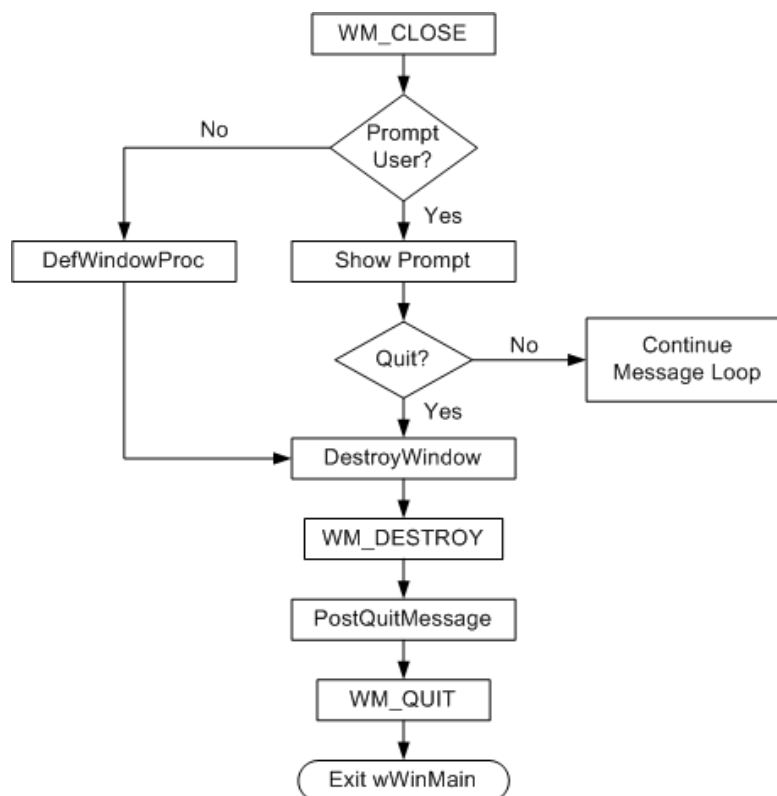


Diagrama de flujo que muestra cómo manejar los mensajes **WM_CLOSE y **WM_DESTROY****

Ejercicio 1. Montar la solución y el primer proyecto

1. Se debe montar un proyecto vacío (Visual C++>General>Proyecto vacío) y añadirle el fichero .cpp que se propone.
2. Depurar paso a paso para comprobar cómo se construye la ventana.
3. Poner breakpoints en el bucle de mensajes y en la respuesta a los eventos y analizar cómo funciona.
4. Incluir la opción de salir con un cuadro de diálogo que nos solicite confirmación.

Un enfoque orientado a objetos

Tiene sentido proporcionar esta estructura de datos con funciones miembro (métodos) que operan en los datos. Esto conduce naturalmente a un diseño donde la estructura (o clase) es responsable de todas las operaciones en la ventana. El procedimiento de ventana se convertiría en parte de la clase. En otras palabras, nos gustaría ir de esto:

```
// pseudocode
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    StateInfo *pState;

    /* Get pState from the HWND. */

    switch (uMsg)
    {
        case WM_SIZE:
            HandleResize(pState, ...);
            break;

        case WM_PAINT:
            HandlePaint(pState, ...);
            break;

        // And so forth.
    }
}
```

A esto:

```
// pseudocode
LRESULT MyWindow::WindowProc(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_SIZE:
            this->HandleResize(...);
            break;

        case WM_PAINT:
            this->HandlePaint(...);
            break;
    }
}
```

El único problema es cómo conectar el método `MyWindow::WindowProc`, que no es tan sencillo porque cuando se registra la ventana hay que pasar un puntero a una función estática.

La función **RegisterClass** espera que el procedimiento de ventana sea un puntero de función. No puede pasar un puntero a una función miembro (no estática) en este contexto. Sin embargo, puede pasar un puntero a una función miembro *estática* y luego delegar a la función miembro. Aquí hay una plantilla de clase que muestra este enfoque:

```

template <class DERIVED_TYPE>
class BaseWindow
{
public:
    static LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
    {
        DERIVED_TYPE *pThis = NULL;

        if (uMsg == WM_NCCREATE)
        {
            CREATESTRUCT* pCreate = (CREATESTRUCT*)lParam;
            pThis = (DERIVED_TYPE*)pCreate->lpCreateParams;
            SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pThis);

            pThis->m_hwnd = hwnd;
        }
        else
        {
            pThis = (DERIVED_TYPE*)GetWindowLongPtr(hwnd, GWLP_USERDATA);
        }
        if (pThis)
        {
            return pThis->HandleMessage(uMsg, wParam, lParam);
        }
        else
        {
            return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
    }

    BaseWindow() : m_hwnd(NULL) { }

    BOOL Create(
        PCWSTR lpWindowName,
        DWORD dwStyle,
        DWORD dwExStyle = 0,
        int x = CW_USEDEFAULT,
        int y = CW_USEDEFAULT,
        int nWidth = CW_USEDEFAULT,
        int nHeight = CW_USEDEFAULT,
        HWND hWndParent = 0,
        HMENU hMenu = 0
    )
    {
        WNDCLASS wc = {0};

        wc.lpfnWndProc = DERIVED_TYPE::WindowProc;
        wc.hInstance = GetModuleHandle(NULL);
        wc.lpszClassName = ClassName();

        RegisterClass(&wc);

        m_hwnd = CreateWindowEx(
            dwExStyle, ClassName(), lpWindowName, dwStyle, x, y,
            nWidth, nHeight, hWndParent, hMenu, GetModuleHandle(NULL), this
        );

        return (m_hwnd ? TRUE : FALSE);
    }

    HWND Window() const { return m_hwnd; }

protected:

    virtual PCWSTR ClassName() const = 0;
    virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam) = 0;

    HWND m_hwnd;
};

```


La clase BaseWindow es una clase base abstracta, de la que se derivan clases de ventana específicas. Por ejemplo, aquí está la declaración de una clase simple derivada de BaseWindow:

```
class MainWindow: public BaseWindow <MainWindow>
{
público:
    PCWSTR ClassName () const {return L "Clase de ventana de muestra"; }
    LRESULT HandleMessage (UINT uMsg, WPARAM wParam, LPARAM lParam);
};
```

Para crear la ventana, llame a BaseWindow :: Create:

```
int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int nCmdShow)
{
    MainWindow win;
    if (!win.Create(L"Learn to Program Windows", WS_OVERLAPPEDWINDOW))
    {
        return 0;
    }
    ShowWindow(win.Window(), nCmdShow);

    // Run the message loop.
    MSG msg = { };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return 0;
}
```

El método virtual puro BaseWindow :: HandleMessage se usa para implementar el procedimiento de ventana. Por ejemplo, la siguiente implementación es equivalente al procedimiento de ventana que se hizo anteriormente.

```
LRESULT MainWindow::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;

    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(m_hwnd, &ps);
            FillRect(hdc, &ps.rcPaint, (HBRUSH) (COLOR_WINDOW+1));
            EndPaint(m_hwnd, &ps);
        }
        return 0;

    default:
        return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
    }
    return TRUE;
}
```

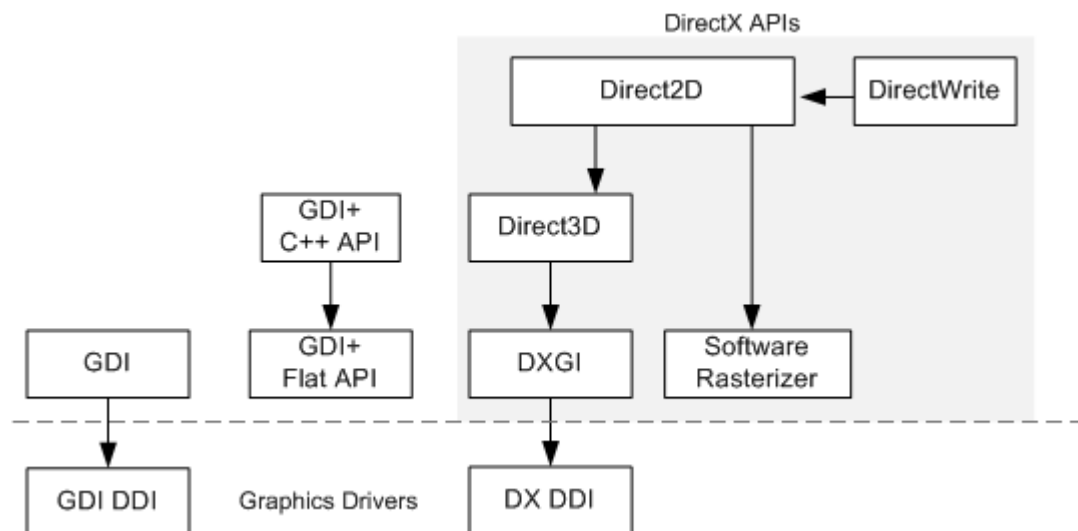
Ejercicio 2. Montar un proyecto con ventana OO

1. Se debe montar un proyecto vacío y añadirle los siguientes ficheros:
 - *Basewin.h*: Define la clase abstracta BaseWindows
 - *MainOO.cpp*: Construye una ventana que deriva de BaseWindows, la registra y ejecuta el bucle de mensajes.
2. Depurar paso a paso para comprobar cómo se construye el objeto ventana.
3. Poner puntos de ruptura en el procesado de eventos y comprobar cómo responde a éstos.

Muchos de los marcos de programación de Windows existentes, como Microsoft Foundation Classes (MFC) y Active Template Library (ATL), utilizan enfoques que son básicamente similares a la que se muestra aquí. Por supuesto, un marco completamente generalizado como MFC es más complejo que este ejemplo relativamente simplista.

Arquitectura de gráficos de Windows

Windows proporciona varias API C++ / COM para gráficos. Estas API se muestran en el siguiente diagrama.



Un diagrama que muestra las API de gráficos de Windows.

- Graphics Device Interface (GDI) es la interfaz gráfica original para Windows. GDI se escribió primero para Windows de 16 bits y luego se actualizó para Windows de 32 bits y 64 bits.
- GDI+ se introdujo en Windows XP como sucesor de GDI. Se accede a la biblioteca GDI+ a través de un conjunto de clases de C++ que envuelven las funciones C planas. .NET Framework también proporciona una versión administrada de GDI+ en el espacio de nombres **System.Drawing**.
- Direct3D admite gráficos tridimensionales.
- Direct2D es una API moderna para gráficos 2D, el sucesor de GDI y GDI+.
- DirectWrite es un motor de rasterización y diseño de texto. Puede usar GDI o Direct2D para dibujar el texto rasterizado.
- DirectX Graphics Infrastructure (DXGI) realiza tareas de bajo nivel, como la presentación de marcos para la salida. La mayoría de las aplicaciones no usan DXGI directamente. Por el contrario, sirve como una capa intermedia entre el controlador de gráficos y Direct3D.

Direct2D y DirectWrite se introdujeron en Windows 7. Direct2D es el foco de este módulo. Aunque tanto GDI como GDI+ siguen siendo compatibles con Windows, Direct2D y DirectWrite se recomiendan para los nuevos programas. En algunos casos, una combinación de tecnologías podría ser más práctica. Para estas situaciones, Direct2D y DirectWrite están diseñados para interoperar con GDI.

Las siguientes secciones describen algunos de los beneficios de Direct2D.

Aceleración de hardware

El término *aceleración de hardware* se refiere a cálculos gráficos realizados por la unidad de procesamiento de gráficos (GPU), en lugar de la CPU. Las GPU modernas están altamente

optimizadas para los tipos de cálculos utilizados en la representación de gráficos. En general, cuanto más de este trabajo se mueve de la CPU a la GPU, mejor.

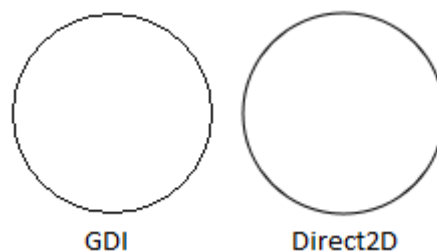
Mientras que GDI admite la aceleración de hardware para ciertas operaciones, muchas operaciones de GDI están ligadas a la CPU. Direct2D se superpone a Direct3D y aprovecha al máximo la aceleración de hardware proporcionada por la GPU. Si la GPU no admite las características necesarias para Direct2D, Direct2D recurre a la representación del software. En general, Direct2D supera a GDI y GDI + en la mayoría de las situaciones.

Transparencia y Anti-aliasing

Direct2D es compatible con la mezcla alfa acelerada por hardware (transparencia).

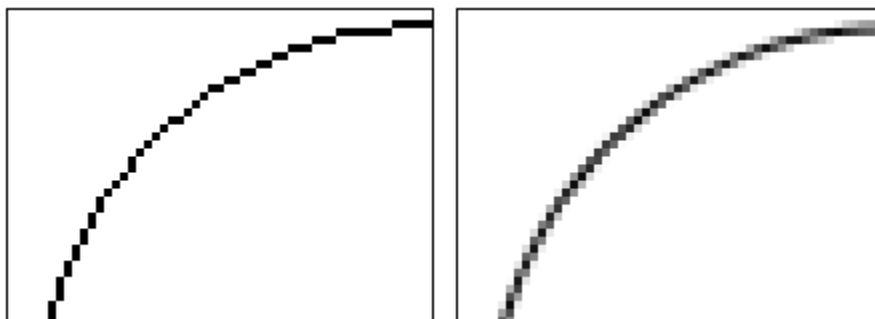
GDI tiene soporte limitado para alpha-blending. La mayoría de las funciones de GDI no son compatibles con la mezcla alfa, aunque GDI sí admite la fusión alfa durante una operación de bitblt. GDI + admite transparencia, pero la mezcla alfa es realizada por la CPU, por lo que no se beneficia de la aceleración de hardware.

La fusión alfa acelerada por hardware también permite el anti-aliasing. *Aliasing* es un artefacto causado por el muestreo de una función continua. Por ejemplo, cuando una línea curva se convierte en píxeles, el alias puede causar una apariencia dentada. [3] Cualquier técnica que reduce los artefactos causados por alias se considera una forma de anti-aliasing. En gráficos, el suavizado se realiza mezclando bordes con el fondo. Por ejemplo, aquí hay un círculo dibujado por GDI y el mismo círculo dibujado por Direct2D.



Una ilustración de técnicas anti-aliasing en Direct2D.

La siguiente imagen muestra un detalle de cada círculo.



Un detalle de la imagen anterior.

El círculo dibujado por GDI (izquierda) consiste en píxeles negros que se aproximan a una curva. El círculo dibujado por Direct2D (derecha) usa mezcla para crear una curva más suave. GDI no admite anti-aliasing cuando dibuja geometría (líneas y curvas). GDI puede dibujar texto anti-alias usando ClearType; pero de lo contrario, el texto de GDI también se alias. El alias es particularmente notable para el texto, porque las líneas irregulares interrumpen el diseño de la

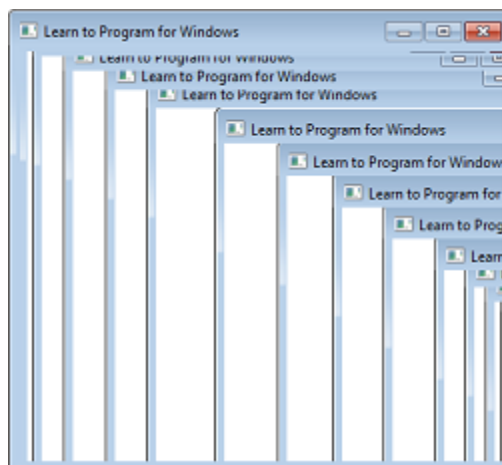
fuelle, lo que hace que el texto sea menos legible. Aunque GDI + admite el anti-aliasing, es aplicado por la CPU, por lo que el rendimiento no es tan bueno como Direct2D.

Gráficos vectoriales

Direct2D es compatible con *gráficos vectoriales*. En gráficos vectoriales, las fórmulas matemáticas se utilizan para representar líneas y curvas. Estas fórmulas no dependen de la resolución de pantalla, por lo que se pueden escalar a dimensiones arbitrarias. Los gráficos vectoriales son particularmente útiles cuando una imagen debe escalarse para admitir diferentes tamaños de monitores o resoluciones de pantalla.

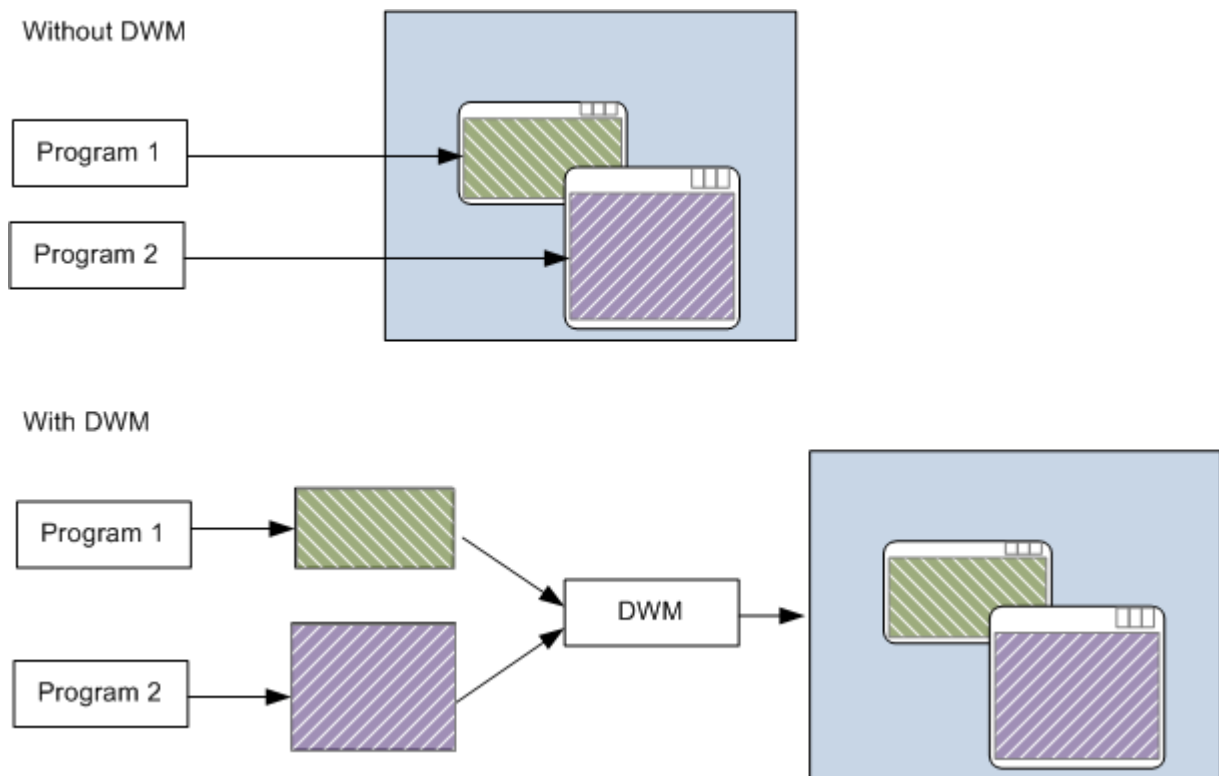
Administrador de la ventana del escritorio (DWM)

Antes de Windows Vista, un programa de Windows dibujaría directamente a la pantalla. En otras palabras, el programa escribiría directamente en el búfer de memoria que se muestra en la tarjeta de video. Este enfoque puede causar artefactos visuales si una ventana no se repinta correctamente. Por ejemplo, si el usuario arrastra una ventana sobre otra, y la ventana debajo no se repinta con la rapidez suficiente, la ventana más alta puede dejar un rastro:



Una captura de pantalla que muestra artefactos de repintado.

El rastro es causado porque ambas ventanas pintan en la misma área de memoria. A medida que se arrastra la ventana superior, se debe volver a pintar la ventana que se encuentra debajo. Si el repintado es demasiado lento, causa los artefactos que se muestran en la imagen anterior. Windows Vista cambió fundamentalmente la forma en que se dibujan las ventanas, al introducir Desktop Window Manager (DWM). Cuando el DWM está habilitado, una ventana ya no se dibuja directamente en el búfer de visualización. En su lugar, cada ventana se dibuja en un buffer de memoria fuera de pantalla, también llamado *superficie fuera de pantalla*. El DWM luego compone estas superficies en la pantalla.



Un diagrama que muestra cómo el DWM compone el escritorio.

El DWM ofrece varias ventajas sobre la arquitectura de gráficos anterior.

- Menos mensajes de repintado Cuando una ventana está obstruida por otra ventana, la ventana obstruida no necesita volver a pintarse.
- Artefactos reducidos Anteriormente, arrastrar una ventana podía crear artefactos visuales, como se describe.
- Efectos visuales. Debido a que DWM se encarga de componer la pantalla, puede mostrar áreas translúcidas y borrosas de la ventana.
- Escalado automático para alta DPI. A pesar de que la escala no es la forma ideal de manejar DPI alto, es una alternativa viable para aplicaciones antiguas que no fueron diseñadas para configuraciones de DPI altas. (Volveremos sobre este tema más adelante, en la sección [DPI y Píxeles independientes del dispositivo](#)).
- Vistas alternativas El DWM puede usar las superficies fuera de pantalla de varias maneras interesantes. Por ejemplo, el DWM es la tecnología detrás de Windows Flip 3D, miniaturas y transiciones animadas.

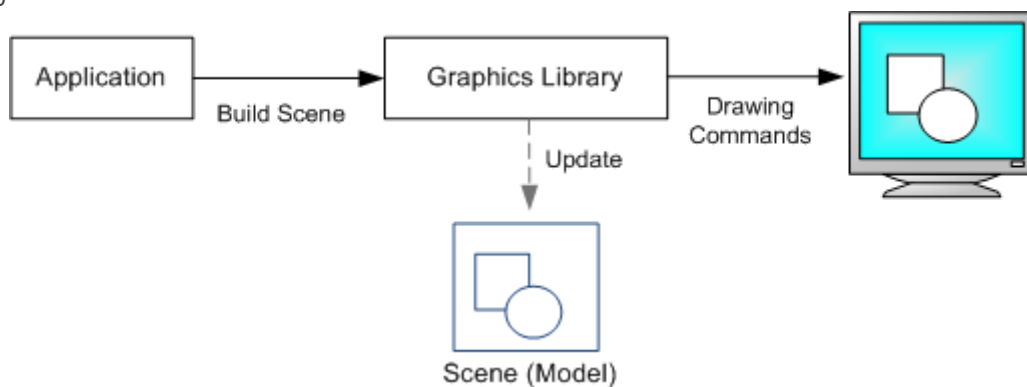
Sin embargo, tenga en cuenta que no se garantiza que DWM esté habilitado. Es posible que la tarjeta gráfica no admita los requisitos del sistema DWM y que los usuarios puedan deshabilitar el DWM a través del panel de control **Propiedades** del **sistema**. Eso significa que su programa no debe confiar en el comportamiento de repintado de DWM. Pruebe su programa con DWM desactivado para asegurarse de que se repinta correctamente.

Modo retenido versus modo inmediato

Las API de gráficos se pueden dividir en API de *modo retenido* y API de *modo inmediato*. P.e.:

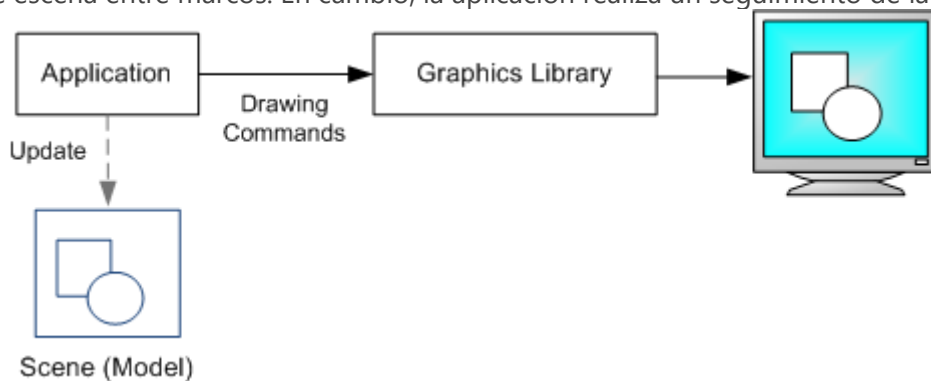
- Direct2D es una API de modo inmediato.
- Windows Presentation Foundation (WPF) es un ejemplo de una API de modo retenido.

Una API de modo retenido es declarativa. La aplicación construye una escena a partir de primitivas gráficas, como formas y líneas. La biblioteca de gráficos almacena un modelo de la escena en la memoria. Para dibujar un marco, la biblioteca de gráficos transforma la escena en un conjunto de comandos de dibujo. Entre fotogramas, la biblioteca de gráficos mantiene la escena en la memoria. Para cambiar lo que se representa, la aplicación emite un comando para actualizar la escena, por ejemplo, para agregar o eliminar una forma. La biblioteca es responsable de redibujar la escena.



Un diagrama que muestra gráficos de modo retenido.

Una API de modo inmediato es de procedimiento. Cada vez que se dibuja un nuevo marco, la aplicación emite directamente los comandos de dibujo. La biblioteca de gráficos no almacena un modelo de escena entre marcos. En cambio, la aplicación realiza un seguimiento de la escena.



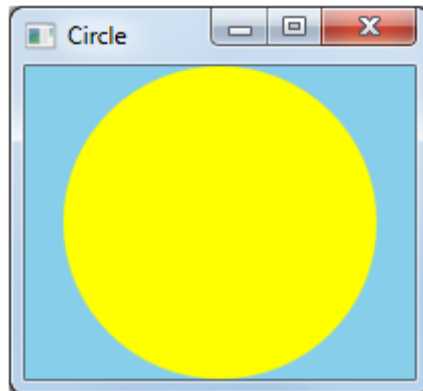
Un diagrama que muestra gráficos de modo inmediato.

Las API de modo retenido pueden ser más fáciles de usar, porque la API hace más trabajo por usted, como la inicialización, el mantenimiento del estado y la limpieza. Por otro lado, a menudo son menos flexibles, porque la API impone su propio modelo de escena. Además, una API de modo retenido puede tener requisitos de memoria más altos, ya que necesita proporcionar un modelo de escena de propósito general. Con una API de modo inmediato, puede implementar optimizaciones específicas.

Direct2D

Primer programa Direct2D

Creemos nuestro primer programa Direct2D. El programa no hace nada elegante, simplemente dibuja un círculo que llena el área del cliente de la ventana. Pero este programa presenta muchos conceptos esenciales de Direct2D.



Una captura de pantalla del programa Circle.

Aquí está el listado del código para el programa Circle. El programa reutiliza la clase `BaseWindow` que se definió en el tema [Administración del estado de la aplicación](#). Los temas posteriores examinarán el código en detalle.

```
#include <windows.h>
#include <d2d1.h>
#pragma comment(lib, "d2d1")
#include "basewin.h"

template <class T> void SafeRelease(T **ppT)
{
    if (*ppT)
    {
        (*ppT)->Release();
        *ppT = NULL;
    }
}

class MainWindow : public BaseWindow<MainWindow>
{
    ID2D1Factory          *pFactory;
    ID2D1HwndRenderTarget *pRenderTarget;
    ID2D1SolidColorBrush  *pBrush;
    D2D1_ELLIPSE           ellipse;
    void CalculateLayout();
    HRESULT CreateGraphicsResources();
    void DiscardGraphicsResources();
    void OnPaint();
    void Resize();

public:
    MainWindow() : pFactory(NULL), pRenderTarget(NULL), pBrush(NULL)
    {
    }
    PCWSTR ClassName() const { return L"Circle Window Class"; }
    LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);
};

// Recalculate drawing layout when the size of the window changes.
void MainWindow::CalculateLayout()
```



```

{
    if (pRenderTarget != NULL)
    {
        D2D1_SIZE_F size = pRenderTarget->GetSize();
        const float x = size.width / 2;
        const float y = size.height / 2;
        const float radius = min(x, y);
        ellipse = D2D1::Ellipse(D2D1::Point2F(x, y), radius, radius);
    }
}

HRESULT MainWindow::CreateGraphicsResources()
{
    HRESULT hr = S_OK;
    if (pRenderTarget == NULL)
    {
        RECT rc;
        GetClientRect(m_hwnd, &rc);
        D2D1_SIZE_U size = D2D1::SizeU(rc.right, rc.bottom);

        hr = pFactory->CreateHwndRenderTarget(
            D2D1::RenderTargetProperties(),
            D2D1::HwndRenderTargetProperties(m_hwnd, size),
            &pRenderTarget);

        if (SUCCEEDED(hr))
        {
            const D2D1_COLOR_F color = D2D1::ColorF(1.0f, 1.0f, 0);
            hr = pRenderTarget->CreateSolidColorBrush(color, &pBrush);
            if (SUCCEEDED(hr))
            {
                CalculateLayout();
            }
        }
    }
    return hr;
}

void MainWindow::DiscardGraphicsResources()
{
    SafeRelease(&pRenderTarget);
    SafeRelease(&pBrush);
}

void MainWindow::OnPaint()
{
    HRESULT hr = CreateGraphicsResources();
    if (SUCCEEDED(hr))
    {
        PAINTSTRUCT ps;
        BeginPaint(m_hwnd, &ps);

        pRenderTarget->BeginDraw();
        pRenderTarget->Clear( D2D1::ColorF(D2D1::ColorF::SkyBlue) );
        pRenderTarget->FillEllipse(ellipse, pBrush);
        hr = pRenderTarget->EndDraw();

        if (FAILED(hr) || hr == D2DERR_RECREATE_TARGET)
        {
            DiscardGraphicsResources();
        }
        EndPaint(m_hwnd, &ps);
    }
}

void MainWindow::Resize()
{
    if (pRenderTarget != NULL)
    {
        RECT rc;
        GetClientRect(m_hwnd, &rc);
        D2D1_SIZE_U size = D2D1::SizeU(rc.right, rc.bottom);
        pRenderTarget->Resize(size);
        CalculateLayout();
        InvalidateRect(m_hwnd, NULL, FALSE);
    }
}

```

```

}

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR, int nCmdShow)
{
    MainWindow win;
    if (!win.Create(L"Circle", WS_OVERLAPPEDWINDOW))
    {
        return 0;
    }
    ShowWindow(win.Window(), nCmdShow);

    // Run the message loop.
    MSG msg = { };
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return 0;
}

LRESULT MainWindow::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
    case WM_CREATE:
        if (FAILED(D2D1CreateFactory(
            D2D1_FACTORY_TYPE_SINGLE_THREADED, &pFactory)))
        {
            return -1; // Fail CreateWindowEx.
        }
        return 0;

    case WM_DESTROY:
        DiscardGraphicsResources();
        SafeRelease(&pFactory);
        PostQuitMessage(0);
        return 0;

    case WM_PAINT:
        OnPaint();
        return 0;

    case WM_SIZE:
        Resize();
        return 0;
    }
    return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
}

```

El espacio de nombres D2D1

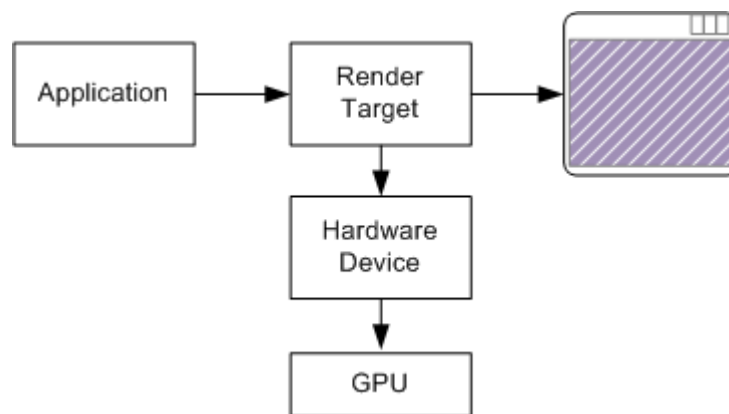
El espacio de nombres **D2D1** contiene funciones y clases de ayuda. Estos no son estrictamente parte de la API de Direct2D, puede programar Direct2D sin usarlos, pero ayudan a simplificar su código. El espacio de nombres **D2D1** contiene:

- Una clase **ColorF** para construir valores de color.
- Un **Matrix3x2F** para construir matrices de transformación.
- Un conjunto de funciones para inicializar estructuras Direct2D.

Render Targets, dispositivos y recursos

Un *target de renderizado* es simplemente la ubicación donde dibujará su programa. Normalmente, el target de renderizado es una ventana (específicamente, el área de cliente de la ventana). También podría ser un mapa de bits en la memoria que no se muestra. Un target de renderizado está representado por la interfaz [ID2D1RenderTarget](#).

Un *dispositivo* es una abstracción que representa lo que realmente dibuja los píxeles. Un dispositivo de hardware usa la GPU para un rendimiento más rápido, mientras que un dispositivo de software usa la CPU. La aplicación no crea el dispositivo. En cambio, el dispositivo se crea implícitamente cuando la aplicación crea el target de renderizado. Cada target de procesamiento está asociado a un dispositivo particular, ya sea hardware o software.



Un diagrama que muestra la relación entre un target de renderizado y un dispositivo.

Un *recurso* es un objeto que el programa usa para dibujar. Estos son algunos ejemplos de recursos definidos en Direct2D:

- **Brush:** Controla cómo se pintan las líneas y las regiones. Los tipos de pincel incluyen cepillos de color sólido y cepillos gradient.
- **Stroke style:** Controla la apariencia de una línea, por ejemplo, discontinua o sólida.
- **Geometry:** Representa una colección de líneas y curvas.
- **Mesh:** Una forma formada de triángulos. Los datos de malla pueden ser consumidos directamente por la GPU, a diferencia de los datos de geometría, que deben convertirse antes de la representación.

Los targets de renderizado también se consideran un tipo de recurso. Algunos recursos se benefician de la aceleración de hardware. Un recurso de este tipo siempre está asociado a un dispositivo en particular, ya sea hardware (GPU) o software (CPU). Este tipo de recurso se llama *dependiente del dispositivo*. Los pinceles y mallas son ejemplos de recursos dependientes del dispositivo. Si el dispositivo deja de estar disponible, el recurso se debe volver a crear para un nuevo dispositivo.

Otros recursos se guardan en la memoria de la CPU, independientemente del dispositivo que se use. Estos recursos son *independientes del dispositivo*, porque no están asociados con un dispositivo en particular. No es necesario volver a crear recursos independientes del dispositivo cuando cambia el dispositivo. Los estilos de trazo y las geometrías son recursos independientes del dispositivo.

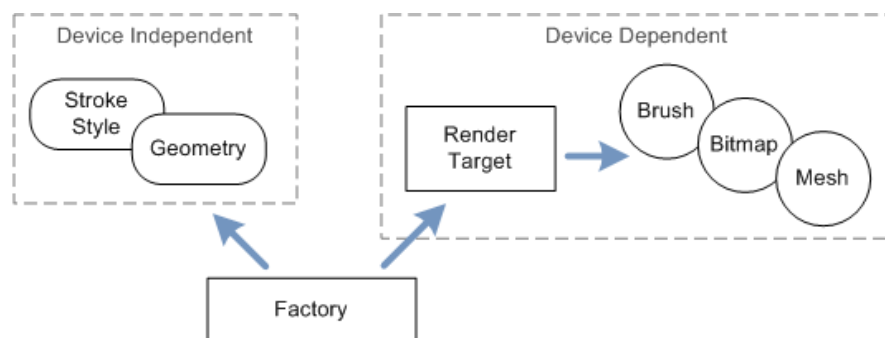
La documentación de MSDN para cada recurso indica si el recurso depende del dispositivo o si es independiente del dispositivo. Cada tipo de recurso está representado por una interfaz que se deriva de [ID2D1Resource](#) . Por ejemplo, los pinceles están representados por la interfaz [ID2D1Brush](#) .

El objeto factoría de Direct2D

El primer paso al usar Direct2D es crear una instancia del objeto de factoría Direct2D. En la programación de computadoras, una *factoría* es un objeto que crea otros objetos. La factoría de Direct2D crea los siguientes tipos de objetos:

- Targets de Renderizado
- Recursos independientes del dispositivo, como estilos de trazo y geometrías.

Los recursos dependientes del dispositivo, como pinceles y mapas de bits, son creados por el objeto de render target.



Un diagrama que muestra la factoría de Direct2D.

Para crear el objeto de factoría Direct2D, llame a la función [D2D1CreateFactory](#) .

```
ID2D1Factory * pFactory = NULL;
HRESULT hr = D2D1CreateFactory (D2D1_FACTORY_TYPE_SINGLE_THREADED, &
pFactory);
```

El primer parámetro es una bandera que especifica las opciones de creación. La bandera `D2D1_FACTORY_TYPE_SINGLE_THREADED` significa que no llamará a Direct2D desde múltiples hilos. Para admitir llamadas de múltiples hilos, especifique `D2D1_FACTORY_TYPE_MULTI_THREADED`. Si su programa usa un solo hilo para llamar a Direct2D, la opción de un único subproceso es más eficiente. El segundo parámetro de la función [D2D1CreateFactory](#) recibe un puntero a la interfaz [ID2D1Factory](#). Debe crear el objeto de factoría Direct2D antes del primer mensaje `WM_PAINT` . El **manejador de mensajes `WM_CREATE`** es un buen lugar para crear la factoría:

```
caso WM_CREATE:
    if (FAILED (D2D1CreateFactory (
        D2D1_FACTORY_TYPE_SINGLE_THREADED, & pFactory)))
    {
        return -1; // Falló CreateWindowEx.
    }
    return 0;
```

Crear recursos de Direct2D

El programa Circle usa los siguientes recursos dependientes del dispositivo:

- Un target de renderizado asociado a la ventana de la aplicación.
- Un pincel de color sólido para pintar el círculo.

Cada uno de estos recursos está representado por una interfaz COM:

- La interfaz **ID2D1HwndRenderTarget** representa el target de renderizado.
- La interfaz **ID2D1SolidColorBrush** representa el pincel.

El programa Circle almacena punteros a estas interfaces como variables miembro de la clase MainWindow:

```
ID2D1HwndRenderTarget * pRenderTarget;  
ID2D1SolidColorBrush * pBrush;
```

El siguiente código crea estos dos recursos.

```
HRESULT MainWindow::CreateGraphicsResources()  
{  
    HRESULT hr = S_OK;  
    if (pRenderTarget == NULL)  
    {  
        RECT rc;  
        GetClientRect(m_hwnd, &rc);  
  
        D2D1_SIZE_U size = D2D1::SizeU(rc.right, rc.bottom);  
  
        hr = pFactory->CreateHwndRenderTarget(  
            D2D1::RenderTargetProperties(),  
            D2D1::HwndRenderTargetProperties(m_hwnd, size),  
            &pRenderTarget);  
  
        if (SUCCEEDED(hr))  
        {  
            const D2D1_COLOR_F color = D2D1::ColorF(1.0f, 1.0f, 0);  
            hr = pRenderTarget->CreateSolidColorBrush(color, &pBrush);  
  
            if (SUCCEEDED(hr))  
            {  
                CalculateLayout();  
            }  
        }  
    }  
    return hr;  
}
```

Para crear un target de renderizado para una ventana, llame al método **ID2D1Factory :: CreateHwndRenderTarget** en la factoría de Direct2D.

- El primer parámetro especifica las opciones que son comunes a cualquier tipo de target de procesamiento. Aquí, pasamos las opciones predeterminadas llamando a la función auxiliar **D2D1 :: RenderTargetProperties**.

- El segundo parámetro especifica el identificador de la ventana más el tamaño del target de renderizado, en píxeles.
- El tercer parámetro recibe un puntero **ID2D1HwndRenderTarget** .

Para crear el pincel de color sólido, llame al método **ID2D1RenderTarget :: CreateSolidColorBrush** en el target de renderizado. El color se da como un valor **D2D1_COLOR_F**. Para obtener más información sobre los colores en Direct2D, consulte [Uso del color en Direct2D](#) .

Si el target de procesamiento ya existe, el método **CreateGraphicsResources** devuelve **S_OK** sin hacer nada. El motivo de este diseño quedará claro en el próximo tema.

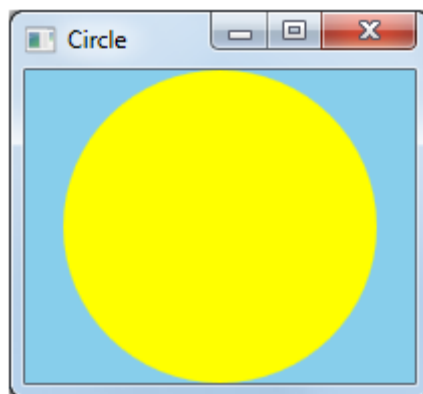
Dibujando con Direct2D

Después de crear sus recursos gráficos, está listo para dibujar.

Dibujando una elipse

El programa [Circle](#) realiza una lógica de dibujo muy simple:

1. Rellena el fondo con un color sólido.
2. Dibuja un círculo lleno



Una captura de pantalla del programa Circle.

Como el objetivo de renderizado es una ventana (a diferencia de un mapa de bits u otra superficie fuera de pantalla), el dibujo se realiza en respuesta a los mensajes **WM_PAINT** . El siguiente código muestra el procedimiento de ventana para el programa Circle.

```

LRESULT MainWindow::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM
lParam)
{
    switch (uMsg)
    {
        case WM_PAINT:
            OnPaint();
    }
}

```

```

        return 0;

        // Other messages not shown...

        return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
    }

```

Aquí está el código que dibuja el círculo.

```

void MainWindow::OnPaint()
{
    HRESULT hr = CreateGraphicsResources();
    if (SUCCEEDED(hr))
    {
        PAINTSTRUCT ps;
        BeginPaint(m_hwnd, &ps);
        pRenderTarget->BeginDraw();
        pRenderTarget->Clear( D2D1::ColorF(D2D1::ColorF::SkyBlue) );
        pRenderTarget->FillEllipse(ellipse, pBrush);
        hr = pRenderTarget->EndDraw();

        if (FAILED(hr) || hr == D2DERR_RECREATE_TARGET)
        {
            DiscardGraphicsResources();
        }
        EndPaint(m_hwnd, &ps);
    }
}

```

La interfaz **ID2D1RenderTarget** se usa para todas las operaciones de dibujo. El método OnPaint del programa hace lo siguiente:

1. El método **ID2D1RenderTarget :: BeginDraw** señala el inicio del dibujo.
2. El método **ID2D1RenderTarget :: Clear** rellena todo el objetivo de renderizado con un color sólido. El color se da como una estructura **D2D1_COLOR_F**. Puede usar la clase **D2D1 :: ColorF** para inicializar la estructura. Para obtener más información, vea [Usar color en Direct2D](#).
3. El método **ID2D1RenderTarget :: FillEllipse** dibuja una elipse llena, utilizando el pincel especificado para el relleno. Una elipse se especifica mediante un punto central y los radios x e y. Si los radios x e y son los mismos, el resultado es un círculo.
4. El método **ID2D1RenderTarget :: EndDraw** señala la finalización del dibujo para este marco. Todas las operaciones de dibujo se deben realizar entre las llamadas a **BeginDraw** y **EndDraw**.

Los **métodos BeginDraw**, **Clear** y **FillEllipse** tienen un tipo de devolución **void**. Si se produce un error durante la ejecución de cualquiera de estos métodos, el error se señala a través del valor de **retorno del método EndDraw**. El método **CreateGraphicsResources** se muestra en el tema [Creación de recursos de Direct2D](#). Este método crea el target de renderizado y el pincel de color sólido.

El dispositivo puede almacenar en búfer los comandos de dibujo y diferir la ejecución hasta que se **llame a EndDraw**. Puede forzar al dispositivo a ejecutar cualquier comando de dibujo

pendiente llamando a **ID2D1RenderTarget :: Flush** . Flushing puede reducir el rendimiento, sin embargo.

Manejo de la pérdida del dispositivo

Mientras se ejecuta su programa, el dispositivo de gráficos que está utilizando podría no estar disponible. Por ejemplo, el dispositivo se puede perder si la resolución de la pantalla cambia o si el usuario retira el adaptador de pantalla. Si el dispositivo se pierde, el target de renderizado también se vuelve inválido, junto con los recursos dependientes del target que se asociaron con el dispositivo. Direct2D señala un dispositivo perdido al devolver el código de error D2DERR_RECREATE_TARGET del método **EndDraw** . Si recibe este código de error, debe volver a crear el target de procesamiento y todos los recursos dependientes del dispositivo. Para descartar un recurso, simplemente libere la interfaz para ese recurso.

```
void MainWindow::DiscardGraphicsResources()
{
    SafeRelease(&pRenderTarget);
    SafeRelease(&pBrush);
}
```

Crear un recurso puede ser una operación costosa, así que no vuelva a crear sus recursos para cada mensaje **WM_PAINT** . Cree un recurso una vez y guarde en caché el puntero del recurso hasta que el recurso no sea válido debido a la pérdida del dispositivo o hasta que ya no necesite ese recurso.

El bucle Render de Direct2D

Independientemente de lo que dibuje, su programa debe realizar un ciclo similar al siguiente.

1. Crear recursos independientes del dispositivo.
2. Renderiza la escena
3. Verifica si existe un target de renderizado válido. Si no, crea el target de renderizado y los recursos dependientes del dispositivo.
4. Llamar a **ID2D1RenderTarget :: StartDraw** .
5. Emita comandos de dibujo.
6. Llamar a **ID2D1RenderTarget :: EndDraw** .
7. Si **EndDraw** devuelve D2DERR_RECREATE_TARGET, descarte el target del renderizado y los recursos dependientes del dispositivo.
8. Repita el paso 2 cada vez que necesite actualizar o volver a dibujar la escena.

Si el target de renderizado es una ventana, el paso 2 ocurre cada vez que la ventana recibe un mensaje **WM_PAINT** . El bucle que se muestra aquí maneja la pérdida del dispositivo al descartar los recursos dependientes del dispositivo y recrearlos al comienzo del siguiente ciclo (paso 2a).

DPI y píxeles independientes del dispositivo

Para programar de manera efectiva con gráficos de Windows, debe comprender dos conceptos relacionados:

- Puntos por pulgada (DPI)
- Píxel independiente del dispositivo (DIPs).

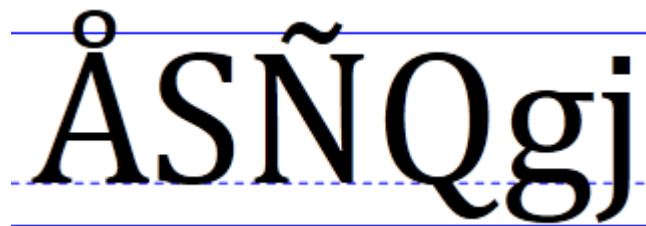
Comencemos con DPI. Esto requerirá un breve desvío hacia la tipografía. En la tipografía, el tamaño del tipo se mide en unidades llamadas *puntos*. Un punto equivale a 1/72 de pulgada.

1 pt = 1/72 de pulgada

Nota Esta es la definición de punto de publicación. Históricamente, la medida exacta de un punto ha variado.

Por ejemplo, una fuente de 12 puntos está diseñada para ajustarse a una línea de texto de 1/6 "(12/72). Obviamente, esto no significa que todos los caracteres de la fuente tengan exactamente 1/6" de alto. De hecho, algunos caracteres pueden ser más altos que 1/6 ". Por ejemplo, en muchas fuentes, el carácter Å es más alto que la altura nominal de la fuente. Para mostrar correctamente, la fuente necesita espacio adicional entre el texto, llamado el *líder*.

La siguiente ilustración muestra una fuente de 72 puntos. Las líneas continuas muestran un cuadro delimitador de 1 "de altura alrededor del texto. La línea punteada se denomina *línea de base*. La mayoría de los caracteres de una fuente se basan en la línea base. La altura de la fuente incluye la porción sobre la línea de base (el *ascenso*) y la porción debajo de la línea base (el *descenso*). En la fuente que se muestra aquí, el ascenso es de 56 puntos y el descenso es de 16 puntos.



Una ilustración que muestra una fuente de 72 puntos.

Sin embargo, cuando se trata de una pantalla de computadora, medir el tamaño del texto es problemático, porque los píxeles no son del mismo tamaño. El tamaño de un píxel depende de dos factores: la resolución de la pantalla y el tamaño físico del monitor. Por lo tanto, las pulgadas físicas no son una medida útil, porque no hay una relación fija entre las pulgadas físicas y los píxeles. En cambio, las fuentes se miden en unidades *lógicas*. Una fuente de 72 puntos se define como una pulgada lógica de alto. Las pulgadas lógicas se convierten a píxeles. Durante muchos años, Windows utilizó la siguiente conversión: una pulgada lógica equivale a 96 píxeles. Con este factor de escala, una fuente de 72 puntos se representa con 96 píxeles de alto. Una fuente de 12 puntos tiene 16 píxeles de alto.

12 puntos = 12/72 pulgada lógica = 1/6 pulgada lógica = 96/6 píxeles = 16 píxeles

Este factor de escala se describe como 96 puntos por pulgada (DPI). El término puntos se deriva de la impresión, donde los puntos físicos de tinta se colocan en el papel. Para las pantallas de computadora, sería más exacto decir 96 píxeles por pulgada lógica, pero el término DPI se ha atascado.

Debido a que los tamaños reales de los píxeles varían, el texto legible en un monitor puede ser demasiado pequeño en otro monitor. Además, las personas tienen diferentes preferencias:

algunas personas prefieren textos más grandes. Por esta razón, Windows permite al usuario cambiar la configuración de DPI. Por ejemplo, si el usuario configura la pantalla en 144 DPI, una fuente de 72 puntos tiene 144 píxeles de alto. La configuración estándar de DPI es 100% (96 DPI), 125% (120 DPI) y 150% (144 DPI). El usuario también puede aplicar una configuración personalizada. A partir de Windows 7, DPI es una configuración por usuario.

Escalado del DWM

Si un programa no tiene en cuenta los DPI, los siguientes defectos pueden aparecer en configuraciones de alta DPI:

- Elementos de UI recortados.
- Diseño incorrecto
- Bitmaps e iconos pixelados.
- Coordenadas incorrectas del mouse, que pueden afectar a arrastrar y soltar.

Para garantizar que los programas más antiguos funcionen con una configuración de PPP alta, DWM implementa una alternativa útil. Si un programa no está marcado como compatible con DPI, el DWM escalará toda la interfaz de usuario para que coincida con la configuración de DPI. Por ejemplo, en 144 DPI, la IU se escala en un 150%, incluidos texto, gráficos, controles y tamaños de ventana. Si el programa crea una ventana de 500 × 500, la ventana aparece realmente como 750 × 750 píxeles, y los contenidos de la ventana se escalan en consecuencia. Este comportamiento significa que los programas más antiguos "solo funcionan" en configuraciones de alta resolución de PPP. Sin embargo, la escala también da como resultado una apariencia algo borrosa, porque la escala se aplica después de dibujar la ventana.

Aplicaciones DPI-Aware

Para evitar la escala de DWM, un programa puede marcarse como compatible con DPI. Esto le dice al DWM que no realice ninguna escala automática de DPI. Todas las aplicaciones nuevas deben diseñarse para que sean compatibles con DPI, ya que el conocimiento de DPI mejora la apariencia de la IU en configuraciones de DPI más altas.

Un programa se declara compatible con DPI a través de su manifiesto de aplicación.

Un *manifiesto* es simplemente un archivo XML que describe una DLL o aplicación. El manifiesto normalmente está incrustado en el archivo ejecutable, aunque se puede proporcionar como un archivo separado. Un manifiesto contiene información como las dependencias de DLL, el nivel de privilegio solicitado y la versión de Windows para la que se diseñó el programa.

Para declarar que su programa es compatible con DPI, incluya la siguiente información en el manifiesto.

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0" xmlns:asmv3="urn:schemas-microsoft-com:asm.v3" >
  <asmv3:application>
    <asmv3:windowsSettings
xmlns="http://schemas.microsoft.com/SMI/2005/WindowsSettings">
      <dpiAware>true</dpiAware>
    </asmv3:windowsSettings>
  </asmv3:application>
</assembly>
```

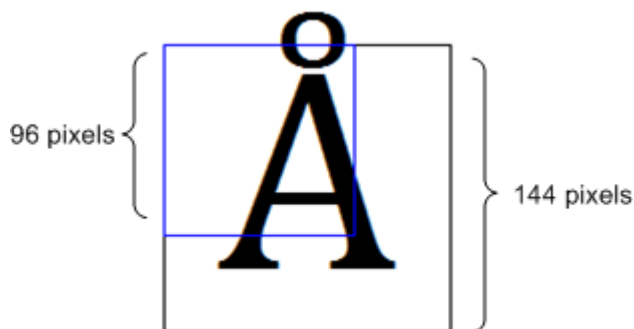
La lista que se muestra aquí es solo un manifiesto parcial, pero el vinculador de Visual Studio genera automáticamente el resto del manifiesto. Para incluir un manifiesto parcial en su proyecto, realice los siguientes pasos en Visual Studio.

1. En el menú **Proyecto** , haga clic en **Propiedad** .
2. En el panel izquierdo, expanda **Propiedades de configuración** , expanda **Herramienta de manifiesto** y luego haga clic en **Entrada y Salida** .
3. En el cuadro de texto **Archivos de manifiesto adicionales** , escriba el nombre del archivo de manifiesto y luego haga clic en **Aceptar** .

Al marcar su programa como compatible con DPI, le está diciendo al DWM que no escale la ventana de su aplicación. Ahora bien, si crea una ventana de 500×500 , la ventana ocupará 500×500 píxeles, independientemente de la configuración de DPI del usuario.

GDI y DPI

El dibujo de GDI se mide en píxeles. Eso significa que si su programa está marcado como DPI-aware, y le pide a GDI que dibuje un rectángulo de 200×100 , el rectángulo resultante tendrá 200 píxeles de ancho y 100 píxeles de alto en la pantalla. Sin embargo, los tamaños de letra GDI se cambian a la configuración actual de DPI. En otras palabras, si crea una fuente de 72 puntos, el tamaño de la fuente será de 96 píxeles a 96 ppp, pero 144 píxeles a 144 ppp. Aquí hay una fuente de 72 puntos representada en 144 DPI usando GDI.



Un diagrama que muestra la escala de fuente DPI en GDI.

Si su aplicación es compatible con DPI y usa GDI para dibujar, escale todas las coordenadas de su dibujo para que coincida con el DPI.

Direct2D y DPI

Direct2D realiza automáticamente la escala para que coincida con la configuración de DPI. En Direct2D, las coordenadas se miden en unidades llamadas *píxeles independientes del dispositivo* (DIP). Un DIP se define como 1/96 de una pulgada *lógica*. En Direct2D, todas las operaciones de dibujo se especifican en DIP y luego se ajustan a la configuración actual de DPI. Por ejemplo, si la configuración de PPP del usuario es 144 ppp, y le pide a Direct2D que dibuje un rectángulo de 200×100 , el rectángulo tendrá 300×150 píxeles físicos. Además, DirectWrite mide el tamaño de fuente en DIP, en lugar de puntos. Para crear una fuente de 12 puntos, especifique 16 DIP (12

puntos = 1/6 de pulgada lógica = 96/6 DIP). Cuando el texto se dibuja en la pantalla, Direct2D convierte los DIP en píxeles físicos. El beneficio de este sistema es que las unidades de medida son consistentes tanto para texto como para dibujo, independientemente de la configuración actual de DPI.

Configuración DPI	Tamaño DIP
96	1 pixel
120	1.25 pixeles
144	1.5 píxeles

Una advertencia: las coordenadas del mouse y la ventana todavía se dan en píxeles físicos, no en DIP. Por ejemplo, si procesa el mensaje [WM_LBUTTONDOWN](#), la posición del mouse se da en píxeles físicos. Para dibujar un punto en esa posición, debe convertir las coordenadas de píxel a DIP.

Conversión de píxeles físicos a DIP

La conversión de píxeles físicos a DIP usa la siguiente fórmula.

$$\text{DIP} = \text{píxeles} / (\text{DPI} / 96.0)$$

Para obtener la configuración de PPP, llame al método [ID2D1Factory :: GetDesktopDpi](#). El DPI se devuelve como dos valores de coma flotante, uno para el eje x y uno para el eje y. En teoría, estos valores pueden diferir. Calcule un factor de escala separado para cada eje.

```
float g_DPIScaleX = 1.0f;
float g_DPIScaleY = 1.0f;

void InitializeDPIScale(ID2D1Factory *pFactory)
{
    FLOAT dpiX, dpiY;
    pFactory->GetDesktopDpi(&dpiX, &dpiY);

    g_DPIScaleX = dpiX/96.0f;
    g_DPIScaleY = dpiY/96.0f;
}

template <typename T>
float PixelsToDipsX(T x)
{
    return static_cast<float>(x) / g_DPIScaleX;
}

template <typename T>
float PixelsToDipsY(T y)
{
    return static_cast<float>(y) / g_DPIScaleY;
}
```

Aquí hay una forma alternativa de obtener la configuración de DPI si no está usando Direct2D:

```
void InitializeDPIScale (HWND hwnd)
{
    HDC hdc = GetDC (hwnd);
    g_DPIScaleX = GetDeviceCaps (hdc, LOGPIXELSX) / 96.0f;
    g_DPIScaleY = GetDeviceCaps (hdc, LOGPIXELSY) / 96.0f;
    ReleaseDC (hwnd, hdc);
}
```

Cambiar el tamaño del target de renderizado

Si el tamaño de la ventana cambia, debe cambiar el tamaño del target de renderizado para que coincida. En la mayoría de los casos, también deberá actualizar el diseño y volver a pintar la ventana. El siguiente código muestra estos pasos.

```
void MainWindow::Resize()
{
    if (pRenderTarget != NULL)
    {
        RECT rc;
        GetClientRect(m_hwnd, &rc);

        D2D1_SIZE_U size = D2D1::SizeU(rc.right, rc.bottom);

        pRenderTarget->Resize(size);
        CalculateLayout();
        InvalidateRect(m_hwnd, NULL, FALSE);
    }
}
```

La función [GetClientRect](#) obtiene el nuevo tamaño del área del cliente, en píxeles físicos (no DIP). El método [ID2D1HwndRenderTarget::Resize](#) actualiza el tamaño del objetivo de procesamiento, también se especifica en píxeles. La función [InvalidateRect](#) fuerza un repintado al agregar todo el área del cliente a la región de actualización de la ventana. (Ver [Pintar la ventana](#), en el Módulo 1.)

A medida que la ventana crece o se reduce, normalmente necesitará volver a calcular la posición de los objetos que dibuja. Por ejemplo, en el programa circular, el radio y el punto central deben actualizarse:









```
void MainWindow::CalculateLayout()
{
    if (pRenderTarget != NULL)
    {
        D2D1_SIZE_F size = pRenderTarget->GetSize();
        const float x = size.width / 2;
        const float y = size.height / 2;
        const float radius = min(x, y);
        ellipse = D2D1::Ellipse(D2D1::Point2F(x, y), radius, radius);
    }
}
```

El método **ID2D1RenderTarget :: GetSize** devuelve el tamaño del target de procesamiento en DIP (no píxeles), que es la unidad adecuada para calcular el diseño. Existe un método estrechamente relacionado, **ID2D1RenderTarget :: GetPixelSize**, que devuelve el tamaño en píxeles físicos. Para un destino de renderizado **HWND**, este valor coincide con el tamaño devuelto por **GetClientRect**. Pero recuerde que el dibujo se realiza en DIP, no en píxeles.

Usando Color en Direct2D

Direct2D utiliza el modelo de color RGB, en el que los colores se forman combinando diferentes valores de rojo, verde y azul. Un cuarto componente, alfa, mide la transparencia de un píxel. En Direct2D, cada uno de estos componentes es un valor de coma flotante con un rango de [0.0-1.0]. Para los tres componentes de color, el valor mide la intensidad del color. Para el componente alfa, 0.0 significa completamente transparente, y 1.0 significa completamente opaco. La siguiente tabla muestra los colores que resultan de varias combinaciones de intensidad del 100%.

Red	Verde	Azul	Color
0	0	0	Negro
1	0	0	Red
0	1	0	Verde
0	0	1	Azul
0	1	1	Cyan
1	0	1	Magenta
1	1	0	Amarillo
1	1	1	Blanco

(0,0,0)		(0,1,1)	
(1,0,0)		(1,0,1)	
(0,1,0)		(1,1,0)	
(0,0,1)		(1,1,1)	

Una imagen que muestra colores RGB.

Los valores de color entre 0 y 1 dan como resultado diferentes tonos de estos colores puros. Direct2D usa la estructura **D2D1_COLOR_F** para representar colores. Por ejemplo, el siguiente código especifica magenta.

```
// Inicializa un color magenta.  
D2D1_COLOR_F clr;  
clr.r = 1;  
clr.g = 0;  
clr.b = 1;  
clr.a = 1; // Opaco.
```

También puede especificar un color utilizando la clase **D2D1 :: ColorF** , que se deriva de la estructura **D2D1_COLOR_F** .

```
// Equivalente al ejemplo anterior.  
D2D1 :: ColorF clr (1, 0, 1, 1);
```

Alpha Blending

La combinación Alpha crea áreas translúcidas al mezclar el color frontal con el color de fondo, usando la siguiente fórmula.

$$\text{color} = \alpha f C_f + (1-\alpha) C_b$$

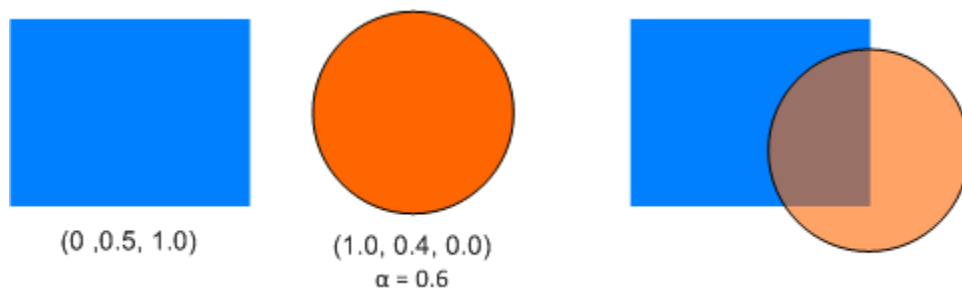
donde C_b es el color de fondo, C_f es el color de primer plano, y α es el valor alfa del color de primer plano. Esta fórmula se aplica por pares a cada componente de color. Por ejemplo, supongamos que el color de primer plano es ($R = 1.0$, $G = 0.4$, $B = 0.0$), con $\alpha = 0.6$, y el color de fondo es ($R = 0.0$, $G = 0.5$, $B = 1.0$). El color alfa-mezclado resultante es:

$$R = (1.0 \times 0.6 + 0 \times 0.4) = .6$$

$$G = (0.4 \times 0.6 + 0.5 \times 0.4) = .44$$

$$B = (0 \times 0.6 + 1.0 \times 0.4) = .40$$

La siguiente imagen muestra el resultado de esta operación de fusión.



Una imagen que muestra fusión alfa.

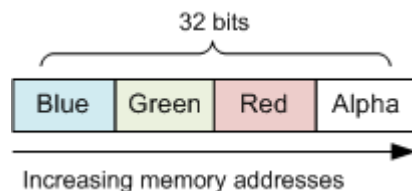
Formatos de píxel

La estructura **D2D1_COLOR_F** no describe cómo se representa un píxel en la memoria. En la mayoría de los casos, eso no importa. Direct2D maneja todos los detalles internos de la traducción de información de color en píxeles. Pero es posible que necesite conocer el formato de píxeles si está trabajando directamente con un mapa de bits en la memoria, o si combina Direct2D con Direct3D o GDI.

La enumeración **DXGI_FORMAT** define una lista de formatos de píxeles. La lista es bastante larga, pero solo algunas de ellas son relevantes para Direct2D. (Los otros son usados por Direct3D).

Formato de píxel	Descripción
DXGI_FORMAT_B8G8R8A8_UNORM	Este es el formato de píxel más común. Todos los componentes de píxeles (rojo, verde, azul y alfa) son enteros sin signo de 8 bits. Los componentes están organizados en orden BGRA en la memoria. (Vea la ilustración que sigue).
DXGI_FORMAT_R8G8B8A8_UNORM	Los componentes de píxeles son enteros sin signo de 8 bits, en orden RGBA . En otras palabras, los componentes rojo y azul se intercambian, en relación con DXGI_FORMAT_B8G8R8A8_UNORM. Este formato solo es compatible con dispositivos de hardware.
DXGI_FORMAT_A8_UNORM	Este formato contiene un componente alfa de 8 bits, sin componentes RGB. Es útil para crear máscaras de opacidad. Para obtener más información sobre el uso de máscaras de opacidad en Direct2D, consulte Descripción general de los objetivos compatibles con A8 .

La siguiente ilustración muestra el diseño de píxeles BGRA.



Un diagrama que muestra el diseño de píxeles BGRA.

Para obtener el formato de píxel de un objetivo de renderizado, llame a [ID2D1RenderTarget :: GetPixelFormat](#) . El formato de píxel puede no coincidir con la resolución de la pantalla. Por ejemplo, la pantalla puede configurarse en color de 16 bits, aunque el objetivo de renderizado utilice color de 32 bits.

Modo Alpha

Un objetivo de renderizado también tiene un modo alfa, que define cómo se tratan los valores alfa.

Modo alfa	Descripción
D2D1_ALPHA_MODE_IGNORE	No se realiza una mezcla alfa. Los valores alfa son ignorados.

D2D1_ALPHA_MODE_STRAIGHT	Alfa recta. Los componentes de color del píxel representan la intensidad del color antes de la mezcla alfa.
D2D1_ALPHA_MODE_PREMULTIPLICADO	Premultiplicado alfa. Los componentes de color del píxel representan la intensidad del color multiplicada por el valor alfa. Este formato es más eficiente de representar que el alfa directo, porque el término ($\alpha \times C_f$) de la fórmula de fusión alfa se precalcula. Sin embargo, este formato no es apropiado para almacenar en un archivo de imagen.

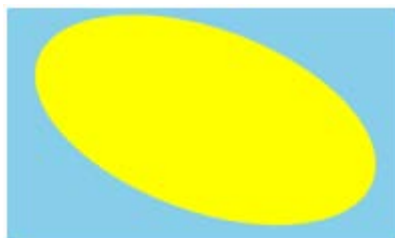
Aquí hay un ejemplo de la diferencia entre alfa recta y alfa premultiplicada. Supongamos que el color deseado es rojo puro (100% de intensidad) con 50% de alfa. Como un tipo Direct2D, este color se representaría como (1, 0, 0, 0.5). Usando alfa lineal, y asumiendo componentes de color de 8 bits, el componente rojo del píxel es 0xFF. Utilizando alfa premultiplicado, el componente rojo se escala en un 50% para igualar 0x80.

El tipo de datos **D2D1_COLOR_F** siempre representa colores usando alfa directa. Direct2D convierte píxeles a formato alfa premultiplicado si es necesario.

Si sabe que su programa no realizará ninguna mezcla alfa, cree el objetivo de renderizado con el modo alfa D2D1_ALPHA_MODE_IGNORE. Este modo puede mejorar el rendimiento, porque Direct2D puede omitir los cálculos alfa. Para obtener más información, vea [Mejorar el rendimiento de las aplicaciones de Direct2D](#).

Aplicación de transformaciones en Direct2D

En [Drawing with Direct2D](#), vimos que el método **ID2D1RenderTarget :: FillEllipse** dibuja una elipse que está alineada con los ejes xey. Pero supongamos que quieres dibujar una elipse inclinada en un ángulo.

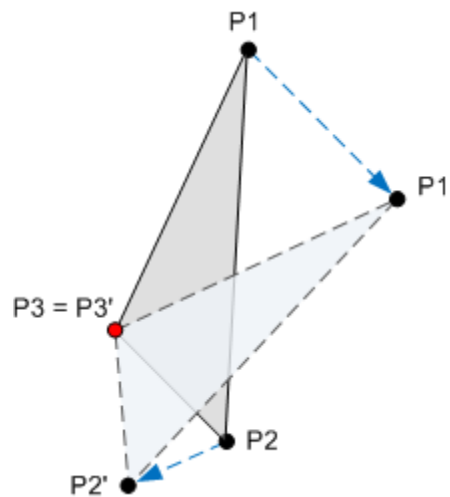


Una imagen que muestra una elipse inclinada.

Al usar transformadas, puede modificar una forma de las siguientes maneras.

- Rotación alrededor de un punto.
- Escalada.
- Traducción (desplazamiento en la dirección X o Y).
- Sesgo (también llamado *cortante*).

Una transformación es una operación matemática que mapea un conjunto de puntos a un nuevo conjunto de puntos. Por ejemplo, el siguiente diagrama muestra un triángulo girado alrededor del punto P3. Después de aplicar la rotación, el punto P1 se mapea a P1', el punto P2 se mapea a P2', y el punto P3 se mapea a sí mismo.



Un diagrama que muestra la rotación alrededor de un punto.

Las transformaciones se implementan usando matrices. Sin embargo, no es necesario que comprenda las matemáticas de las matrices para usarlas. Si desea obtener más información sobre las matemáticas, consulte el [Apéndice: Transformaciones de la matriz](#).

Para aplicar una transformación en Direct2D, llame al método **ID2D1RenderTarget::**

SetTransform. Este método toma una estructura **D2D1_MATRIX_3X2_F** que define la transformación. Puede inicializar esta estructura llamando a métodos en la clase **D2D1::**

Matrix3x2F. Esta clase contiene métodos estáticos que devuelven una matriz para cada tipo de transformación:

- **Matrix3x2F::Rotación**
- **Matrix3x2F::Escala**
- **Matrix3x2F::Traducción**
- **Matrix3x2F::sesgado**

Por ejemplo, el siguiente código aplica una rotación de 20 grados alrededor del punto (100, 100).

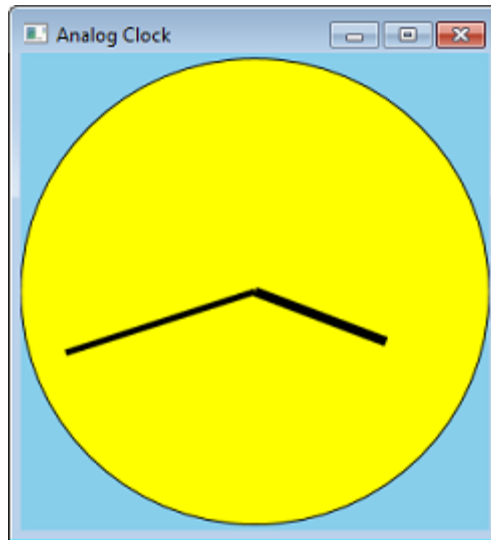
```
pRenderTarget->SetTransform(
    D2D1::Matrix3x2F::Rotation(20, D2D1::Point2F(100,100)));
```

La transformación se aplica a todas las operaciones de dibujo posteriores hasta que vuelva a llamar a **SetTransform**. Para eliminar la transformación actual, llame a **SetTransform** con la matriz de identidad, que devuelve la función **Matrix3x2F::Identity**.

```
pRenderTarget-> SetTransform (D2D1 :: Matrix3x2F :: Identity ());
```

Dibujando las agujas del reloj

Pongamos las transformaciones a usar convirtiendo nuestro programa Circle en un reloj analógico. Podemos hacer esto agregando líneas para las manos.



Una captura de pantalla del programa Reloj analógico.

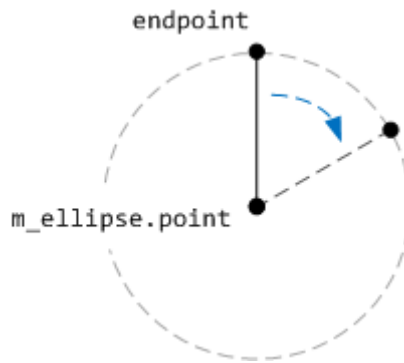
En lugar de calcular las coordenadas de las líneas, podemos simplemente calcular el ángulo y luego aplicar una transformación de rotación. El siguiente código muestra una función que dibuja una manecilla de reloj. El parámetro *fAngle* proporciona el ángulo de la mano, en grados.

```
void Scene::DrawClockHand(float fHandLength, float fAngle, float
fStrokeWidth)
{
    m_pRenderTarget->SetTransform(
        D2D1::Matrix3x2F::Rotation(fAngle, m_ellipse.point)
    );

    // endPoint defines one end of the hand.
    D2D_POINT_2F endPoint = D2D1::Point2F(
        m_ellipse.point.x,
        m_ellipse.point.y - (m_ellipse.radiusY * fHandLength)
    );

    // Draw a line from the center of the ellipse to endPoint.
    m_pRenderTarget->DrawLine(
        m_ellipse.point, endPoint, m_pStroke, fStrokeWidth);
}
```

Este código dibuja una línea vertical, comenzando desde el centro de la esfera del reloj y terminando en el punto *endPoint*. La línea gira alrededor del centro de la elipse aplicando una transformación de rotación. El punto central para la rotación es el centro de elipse que forma la esfera del reloj.



Un diagrama que muestra la rotación de la manecilla del reloj.

El siguiente código muestra cómo se dibuja toda la cara del reloj.

```
void Scene::RenderScene()
{
    m_pRenderTarget->Clear(D2D1::ColorF(D2D1::ColorF::SkyBlue));
    m_pRenderTarget->FillEllipse(m_ellipse, m_pFill);
    m_pRenderTarget->DrawEllipse(m_ellipse, m_pStroke);

    // Draw hands
    SYSTEMTIME time;
    GetLocalTime(&time);

    // 60 minutes = 30 degrees, 1 minute = 0.5 degree
    const float fHourAngle = (360.0f / 12) * (time.wHour) +
(time.wMinute * 0.5f);
    const float fMinuteAngle = (360.0f / 60) * (time.wMinute);

    DrawClockHand(0.6f, fHourAngle, 6);
    DrawClockHand(0.85f, fMinuteAngle, 4);

    // Restore the identity transformation.
    m_pRenderTarget->SetTransform( D2D1::Matrix3x2F::Identity() );
}
```

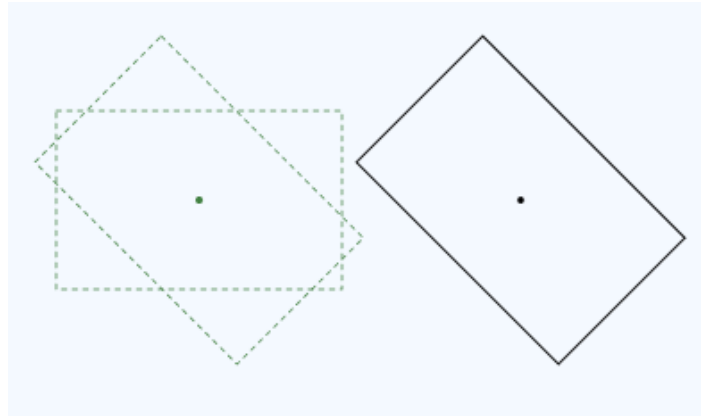
Combinando transformaciones

Las cuatro transformaciones básicas se pueden combinar multiplicando dos o más matrices. Por ejemplo, el siguiente código combina una rotación con una traducción.

```
const D2D1::Matrix3x2F rot = D2D1::Matrix3x2F::Rotation(20);
const D2D1::Matrix3x2F trans = D2D1::Matrix3x2F::Translation(40, 10);

pRenderTarget->SetTransform(rot * trans);
```

La clase **Matrix3x2F** proporciona el **operador *** () para la multiplicación de matrices. El orden en que multiplicas las matrices es importante. Establecer una transformación ($M \times N$) significa "Aplicar M primero, seguido de N." Por ejemplo, aquí está la rotación seguida de la traducción:

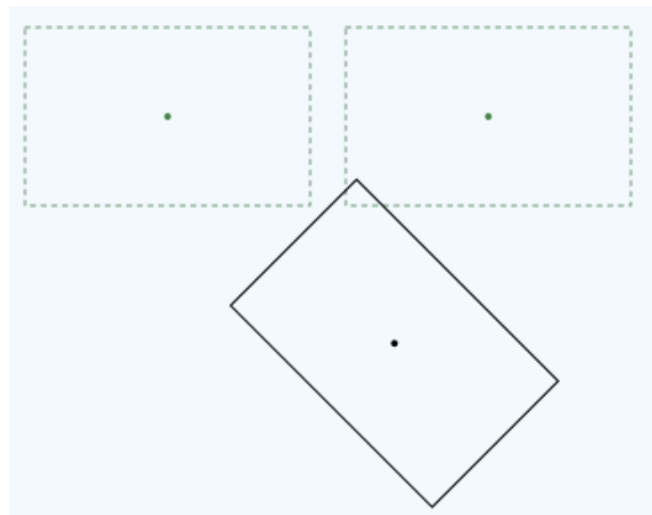


Un diagrama que muestra la rotación seguida de la traducción.

Aquí está el código para esta transformación:

```
const D2D1::Matrix3x2F rot = D2D1::Matrix3x2F::Rotation(45, center);
const D2D1::Matrix3x2F trans = D2D1::Matrix3x2F::Translation(x, 0);
pRenderTarget->SetTransform(rot * trans);
```

Ahora compare esa transformación con una transformación en orden inverso, traducción seguida de rotación.



Un diagrama que muestra la traducción seguida de rotación.

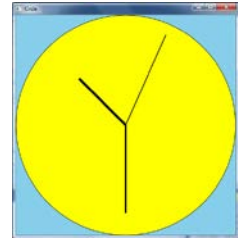
La rotación se realiza alrededor del centro del rectángulo original. Aquí está el código para esta transformación.

```
D2D1::Matrix3x2F rot = D2D1::Matrix3x2F::Rotation(45, center);
D2D1::Matrix3x2F trans = D2D1::Matrix3x2F::Translation(x, 0);
pRenderTarget->SetTransform(trans * rot);
```

Como puede ver, las matrices son las mismas, pero el orden de las operaciones ha cambiado. Esto sucede porque la multiplicación de la matriz no es conmutativa: $M \times N \neq N \times M$

Ejercicio 3. Dibujar el reloj en respuesta a eventos

Se deberá realizar un Reloj que se actualice con el tiempo de sistema. Además de la explicación del tutorial se necesita dominar TIMERS.



TIMERS de la librería Windows

- Identificador para el Timer:

```
UINT_PTR Timer; //Timer
```

- En el evento de creación debemos crear el timer:

```
case WM_CREATE:
    ...
    //Establezco un Timer T=1s;
    Timer=SetTimer(m_hwnd, 0, 1000, (TIMERPROC)NULL);
    return 0;
```

- En nuestra función de procesamiento de mensajes debemos atender el mensaje que genera. En nuestro caso debemos generar el evento de pintado.

```
case WM_TIMER:
    // process the 1-second timer
    OnTime();
    PostMessage(m_hwnd, WM_PAINT, NULL, NULL);
    return 0;
```

- Cuando ya no se necesite el timer es necesario liberarlo:

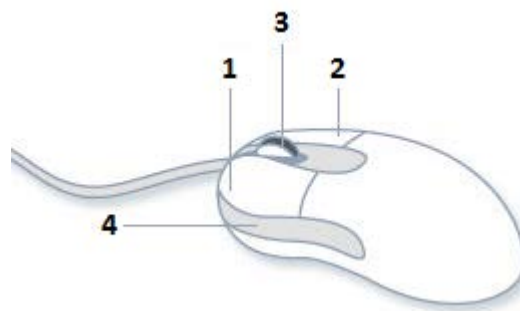
```
case WM_DESTROY:
    ...
    //Libero el Timer;
    KillTimer(m_hwnd, Timer);
    PostQuitMessage(0);
    return 0;
```

Entrada del usuario

Los módulos anteriores han explorado la creación de una ventana, manejo de mensajes de ventana y dibujo básico con gráficos 2D. En este módulo, observamos la entrada del mouse y el teclado. Al final de este módulo, podrá escribir un programa de dibujo simple que use el mouse y el teclado.

Entrada de mouse

Windows admite ratones con hasta cinco botones: izquierdo, medio y derecho, más dos botones adicionales llamados XBUTTON1 y XBUTTON2.



Una ilustración que muestra los botones izquierdo (1), derecho (2), central (3) y XBUTTON1 (4).

La mayoría de los ratones para Windows tienen al menos los botones izquierdo y derecho. El botón izquierdo del mouse se utiliza para señalar, seleccionar, arrastrar, etc. El botón derecho del mouse generalmente muestra un menú contextual. Algunos ratones tienen una rueda de desplazamiento ubicada entre los botones izquierdo y derecho. Dependiendo del mouse, también se puede hacer clic en la rueda de desplazamiento, por lo que es el botón central. Los botones XBUTTON1 y XBUTTON2 a menudo se encuentran a los lados del mouse, cerca de la base. Estos botones adicionales no están presentes en todos los ratones. Si está presente, los botones XBUTTON1 y XBUTTON2 a menudo se asignan a una función de aplicación, como navegación hacia adelante y hacia atrás en un navegador web.

A los usuarios zurdos a menudo les resulta más cómodo intercambiar las funciones de los botones izquierdo y derecho, usando el botón derecho como el puntero y el botón izquierdo para mostrar el menú contextual. Por esta razón, la documentación de ayuda de Windows usa los términos *botón primario* y *secundario*, que se refieren a la función lógica en lugar de a la ubicación física. En la configuración predeterminada (diestro), el botón izquierdo es el botón primario y el derecho es el secundario. Sin embargo, los términos *clic derecho* y *clic izquierdo* hacen referencia a acciones lógicas. *Hacer clic* con el botón *derecho* significa hacer clic en el botón primario, ya sea que ese botón esté físicamente en el lado derecho o izquierdo del mouse. Independientemente de cómo el usuario configure el mouse, Windows traduce automáticamente los mensajes del mouse para que sean coherentes. El usuario puede intercambiar los botones primario y secundario en el medio de usar su programa, y no afectará el comportamiento de su programa.

La documentación de MSDN usa los términos *botón derecho* y *botón izquierdo* para indicar el *botón primario* y *secundario*. Esta terminología es coherente con los nombres de los mensajes

de ventana para la entrada del mouse. Solo recuerda que los botones físicos izquierdo y derecho pueden intercambiarse.

Respondiendo a los clics del mouse

Si el usuario hace clic en un botón del mouse mientras el cursor está sobre el área del cliente de una ventana, la ventana recibe uno de los siguientes mensajes.

Mensaje	Sentido
<u>WM_LBUTTONDOWN</u>	Botón izquierdo abajo
<u>WM_LBUTTONUP</u>	Botón izquierdo arriba
<u>WM_MBUTTONDOWN</u>	Botón central abajo
<u>WM_MBUTTONUP</u>	Botón central arriba
<u>WM_RBUTTONDOWN</u>	Botón derecho hacia abajo
<u>WM_RBUTTONUP</u>	Botón derecho arriba
<u>WM_XBUTTONDOWN</u>	XBUTTONDOWN1 o XBUTTONDOWN2 abajo
<u>WM_XBUTTONUP</u>	XBUTTONDOWN1 o XBUTTONDOWN2 arriba

Recuerde que el área del cliente es la parte de la ventana que excluye el marco. Para obtener más información sobre las áreas de clientes, consulte [¿Qué es una ventana?](#)

Coordenadas del mouse

En todos estos mensajes, el parámetro *lParam* contiene las coordenadas x e y del puntero del mouse. Los 16 bits más bajos de *lParam* contienen la coordenada x, y los siguientes 16 bits contienen la coordenada y. Use las macros [GET_X_LPARAM](#) y [GET_Y_LPARAM](#) para descomprimir las coordenadas de *lParam*.

```
int xPos = GET_X_LPARAM (lParam);  
int yPos = GET_Y_LPARAM (lParam);
```

Estas macros se definen en el archivo de encabezado WindowsX.h.

En Windows de 64 bits, *lParam* tiene un valor de 64 bits. Los 32 bits superiores de *lParam* no se usan. La documentación de MSDN menciona la "palabra de orden baja" y la palabra de "alto orden" de *lParam*. En el caso de 64 bits, esto significa las palabras de orden bajo y alto de los 32 bits más bajos. Las macros extraen los valores correctos, por lo que si las usa, estará a salvo.

Las coordenadas del mouse se dan en píxeles, no en píxeles independientes del dispositivo (DIP), y se miden en relación con el área del cliente de la ventana. Las coordenadas son valores firmados. Las posiciones arriba y a la izquierda del área del cliente tienen coordenadas negativas, lo cual es importante si rastreas la posición del mouse fuera de la ventana. Veremos cómo hacerlo en un tema posterior, [Capturar movimiento del mouse fuera de la ventana](#) .

Banderas adicionales

El parámetro *wParam* contiene un **OR** de banderas bit a bit , que indica el estado de los otros botones del mouse más las teclas SHIFT y CTRL.

Bandera	Sentido
MK_CONTROL	La tecla CTRL está abajo.
MK_LBUTTON	El botón izquierdo del mouse está abajo.
MK_MBUTTON	El botón central del mouse está abajo.
MK_RBUTTON	El botón derecho del mouse está abajo.
MK_SHIFT	La tecla SHIFT está abajo.
MK_XBUTTON1	El botón XBUTTON1 está abajo.
MK_XBUTTON2	El botón XBUTTON2 está abajo.

La ausencia de una bandera significa que no se presionó el botón o la tecla correspondientes. Por ejemplo, para probar si la tecla CTRL está presionada:

```
if (wParam & MK_CONTROL) {...
```

Si necesita encontrar el estado de otras teclas además de CTRL y SHIFT, use la función [GetKeyState](#) , que se describe en [Entrada de teclado](#) .

Los mensajes de la ventana [WM_XBUTTONDOWN](#) y [WM_XBUTTONUP](#) se aplican tanto a XBUTTON1 como a XBUTTON2. El parámetro *wParam* indica en qué botón se hizo clic.

```
Botón UINT = GET_XBUTTON_WPARAM (wParam);
if (botón == XBUTTON1)
{
    // Se hizo clic en XBUTTON1.
}
más si (botón == XBUTTON2)
{
    // Se hizo clic en XBUTTON2.
}
```

Doble Clic

Una ventana no recibe notificaciones de doble clics de forma predeterminada. Para recibir clics dobles, configure el indicador `CS_DBLCLKS` en la estructura **WNDCLASS** cuando registra la clase de ventana.

```
WNDCLASS wc = {};  
wc.style = CS_DBLCLKS;  
  
/ * Establecer otros miembros de estructura. * /  
  
RegisterClass (& wc);
```

Si configura el indicador `CS_DBLCLKS` como se muestra, la ventana recibirá notificaciones de doble clic. Un doble clic se indica mediante un mensaje de ventana con "DBLCLK" en el nombre. Por ejemplo, un doble clic en el botón izquierdo del mouse produce la siguiente secuencia de mensajes:

WM_LBUTTONDOWN
WM_LBUTTONUP
WM_LBUTTONDBLCLK
WM_LBUTTONUP

En efecto, el segundo mensaje **WM_LBUTTONDOWN** que normalmente se generaría se convierte en un mensaje **WM_LBUTTONDBLCLK**. Los mensajes equivalentes se definen para los botones derecho, medio y XBUTTON.

Hasta que aparezca el mensaje de doble clic, no hay forma de saber que el primer clic del mouse es el inicio de un doble clic. Por lo tanto, una acción de doble clic debe continuar una acción que comienza con el primer clic del mouse. Por ejemplo, en el Shell de Windows, un solo clic selecciona una carpeta, mientras que un doble clic abre la carpeta.

Mensajes de mouse no cliente

Se define un conjunto separado de mensajes para eventos de mouse que ocurren dentro del área no cliente de la ventana. Estos mensajes tienen las letras "NC" en el nombre. Por ejemplo, **WM_NCLBUTTONDOWN** es el equivalente no cliente de **WM_LBUTTONDOWN**. Una aplicación típica no interceptará estos mensajes, porque la función **DefWindowProc** maneja estos mensajes correctamente. Sin embargo, pueden ser útiles para ciertas funciones avanzadas. Por ejemplo, podría usar estos mensajes para implementar un comportamiento personalizado en la barra de título. Si maneja estos mensajes, generalmente debe pasarlos a **DefWindowProc** después. De lo contrario, su aplicación romperá la funcionalidad estándar, como arrastrar o minimizar la ventana.

Movimiento del mouse

Cuando el mouse se mueve, Windows publica un mensaje **WM_MOUSEMOVE**. Por defecto, **WM_MOUSEMOVE** va a la ventana que contiene el cursor. Puede anular este comportamiento *capturando* el mouse, que se describe en la siguiente sección.

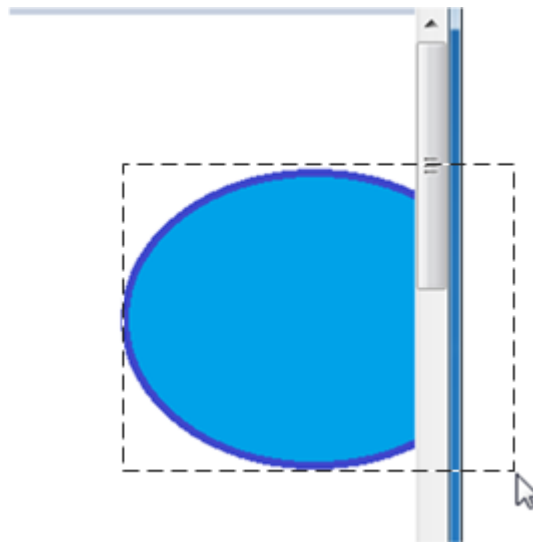
El mensaje **WM_MOUSEMOVE** contiene los mismos parámetros que los mensajes para clics del mouse. Los 16 bits más bajos de *lParam* contienen la coordenada x, y los siguientes 16 bits contienen la coordenada y. Use las macros **GET_X_LPARAM** y **GET_Y_LPARAM** para descomprimir las coordenadas de *lParam* . El parámetro *wParam* contiene un **OR** de banderas bit a bit , que indica el estado de los otros botones del mouse más las teclas SHIFT y CTRL. El siguiente código obtiene las coordenadas del mouse de *lParam* .

```
int xPos = GET_X_LPARAM (lParam);  
int yPos = GET_Y_LPARAM (lParam);
```

Recuerde que estas coordenadas están en píxeles, no en píxeles independientes del dispositivo (DIP). Más adelante en este tema, veremos el código que se convierte entre las dos unidades. Una ventana también puede recibir un mensaje **WM_MOUSEMOVE** si la posición del cursor cambia con respecto a la ventana. Por ejemplo, si el cursor se coloca sobre una ventana y el usuario oculta la ventana, la ventana recibe mensajes **WM_MOUSEMOVE** incluso si el mouse no se movió. Una consecuencia de este comportamiento es que las coordenadas del mouse podrían no cambiar entre los mensajes **WM_MOUSEMOVE** .

Captura del movimiento del mouse fuera de la ventana

De forma predeterminada, una ventana deja de recibir mensajes **WM_MOUSEMOVE** si el mouse se mueve más allá del borde del área del cliente. Pero para algunas operaciones, es posible que deba seguir la posición del mouse más allá de este punto. Por ejemplo, un programa de dibujo podría permitir al usuario arrastrar el rectángulo de selección más allá del borde de la ventana, como se muestra en el siguiente diagrama.



Una ilustración de la captura del mouse.

Para recibir mensajes de movimiento del mouse más allá del borde de la ventana, llame a la función **SetCapture** . Después de llamar a esta función, la ventana continuará recibiendo mensajes **WM_MOUSEMOVE** mientras el usuario mantenga presionado al menos un botón del mouse, incluso si el mouse se mueve fuera de la ventana. La ventana de captura debe ser la ventana de primer plano, y solo una ventana puede ser la ventana de captura a la vez. Para liberar la captura del mouse, llame a la función **ReleaseCapture** . Normalmente usaría **SetCapture** y **ReleaseCapture** de la siguiente manera.

1. Cuando el usuario presiona el botón izquierdo del mouse, llame a [SetCapture](#) para comenzar a capturar el mouse.
2. Responda a los mensajes de movimiento del mouse.
3. Cuando el usuario suelta el botón izquierdo del mouse, llame a [ReleaseCapture](#) .

Ejemplo: dibujar círculos

Extiendamos el programa Circle del [Módulo 3](#) permitiendo al usuario dibujar un círculo con el mouse. Comience con el programa [Direct2D Circle Sample](#) . Modificaremos el código en esta muestra para agregar un dibujo simple. Primero, agregue una nueva variable miembro a la clase MainWindow.

```
D2D1_POINT_2F ptMouse;
```

Esta variable almacena la posición del mouse hacia abajo mientras el usuario arrastra el mouse. En el constructor MainWindow, inicialice las variables *ellipse* y *ptMouse* .

```
MainWindow (): pFactory (NULL), pRenderTarget (NULL), pBrush (NULL),  
    ellipse (D2D1 :: Ellipse (D2D1 :: Point2F (), 0, 0)),  
    ptMouse (D2D1 :: Point2F ())  
{  
}
```

Elimina el cuerpo del método MainWindow :: CalculateLayout; no es requerido para este ejemplo.

```
void CalculateLayout () {}
```

A continuación, declare los manejadores de mensaje para los mensajes de botón izquierdo hacia abajo, botón izquierdo hacia arriba y movimiento de mouse.

```
void OnLButtonDown (int pixelX, int pixelY, DWORD flags);  
void OnLButtonUp ();  
void OnMouseMove (int pixelX, int pixelY, DWORD flags);
```

Las coordenadas del mouse se dan en píxeles físicos, pero Direct2D espera píxeles independientes del dispositivo (DIP). Para manejar las configuraciones de alta resolución de ppp correctamente, debe traducir las coordenadas de píxel a DIP. Para obtener más información sobre DPI, vea [DPI y píxeles independientes del dispositivo](#) . El siguiente código muestra una clase de ayuda que convierte píxeles en DIP.

```
class DPIScale  
{  
    static float scaleX;  
    static float scaleY;  
  
public:  
    static void Initialize(ID2D1Factory *pFactory)  
    {  
        FLOAT dpiX, dpiY;  
        pFactory->GetDesktopDpi(&dpiX, &dpiY);  
        scaleX = dpiX/96.0f;  
        scaleY = dpiY/96.0f;  
    }  
}
```

```

    }

    template <typename T>
    static D2D1_POINT_2F PixelsToDips(T x, T y)
    {
        return D2D1::Point2F(static_cast<float>(x) / scaleX,
                               static_cast<float>(y) / scaleY);
    }
};

float DPIScale::scaleX = 1.0f;
float DPIScale::scaleY = 1.0f;

```

Llame a `DPIScale::Initialize` en su controlador **WM_CREATE**, después de crear el objeto de fábrica `Direct2D`.

```

case WM_CREATE:
    if (FAILED(D2D1CreateFactory(
        D2D1_FACTORY_TYPE_SINGLE_THREADED, &pFactory)))
    {
        return -1; // Fail CreateWindowEx.
    }
    DPIScale::Initialize(pFactory);
    return 0;

```

Para obtener las coordenadas del mouse en DIP de los mensajes del mouse, haga lo siguiente:

1. Use las macros **GET_X_LPARAM** y **GET_Y_LPARAM** para obtener las coordenadas de píxeles. Estas macros están definidas en `WindowsX.h`, así que recuerde incluir ese encabezado en su proyecto.
2. Llame a `DPIScale::PixelsToDipsX` y `DPIScale::PixelsToDipsY` para convertir píxeles a DIP.

Ahora agregue los manejadores de mensajes a su procedimiento de ventana.

```

case WM_LBUTTONDOWN:
    OnLButtonDown(GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam), (DWORD)wParam);
    return 0;

case WM_LBUTTONUP:
    OnLButtonUp();
    return 0;

case WM_MOUSEMOVE:
    OnMouseMove(GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam), (DWORD)wParam);
    return 0;

```

Finalmente, implemente los manejadores de mensajes.

Para el mensaje de botón izquierdo, haga lo siguiente:

1. Llame a **SetCapture** para comenzar a capturar el mouse.
2. Guarde la posición del clic del mouse en la variable *ptMouse*. Esta posición define la esquina superior izquierda del cuadro delimitador para la elipse.
3. Restablece la estructura de elipse.
4. Llamar a **InvalidateRect**. Esta función fuerza a la ventana a ser repintada.

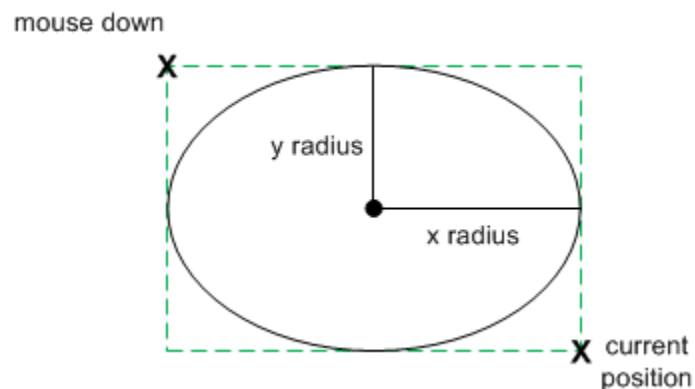
```

void MainWindow::OnLButtonDown(int pixelX, int pixelY, DWORD flags)
{
    SetCapture(m_hwnd);
    ellipse.point = ptMouse = DIPS::PixelsToDips(pixelX, pixelY);
    ellipse.radiusX = ellipse.radiusY = 1.0f;
    InvalidateRect(m_hwnd, NULL, FALSE);
}

```

Para el mensaje de movimiento del mouse, verifique si el botón izquierdo del mouse está abajo. Si es así, vuelva a calcular la elipse y vuelva a pintar la ventana. En Direct2D, una elipse se define por el punto central y los radios xyy. Queremos dibujar una elipse que se ajuste al cuadro delimitador definido por el punto del mouse-down (*ptMouse*) y la posición actual del cursor (*x, y*), por lo que se necesita un poco de aritmética para encontrar el ancho, alto y posición del elipse.

El siguiente código vuelve a calcular la elipse y luego llama a [InvalidateRect](#) para volver a pintar la ventana.



```

void MainWindow::OnMouseMove(int pixelX, int pixelY, DWORD flags)
{
    if (flags & MK_LBUTTON)
    {
        const D2D1_POINT_2F dips = DIPS::PixelsToDips(pixelX, pixelY);

        const float width = (dips.x - ptMouse.x) / 2;
        const float height = (dips.y - ptMouse.y) / 2;
        const float x1 = ptMouse.x + width;
        const float y1 = ptMouse.y + height;

        ellipse = D2D1::Ellipse(D2D1::Point2F(x1, y1), width, height);

        InvalidateRect(m_hwnd, NULL, FALSE);
    }
}

```

Para el mensaje de levantar botón izquierdo, simplemente llame a [ReleaseCapture](#) para liberar la captura del mouse.

```

void MainWindow::OnLButtonUp ()
{
    ReleaseCapture ();
}

```

Otras operaciones con el mouse

Las secciones anteriores han discutido los clics del mouse y el movimiento del mouse. Estas son algunas otras operaciones que se pueden realizar con el mouse.

Arrastrando elementos de UI

Si la interfaz de usuario permite arrastrar de elementos de interfaz de usuario, hay otra función que debe llamar en su controlador de mensajes del ratón hacia abajo: **DragDetect**. La función **DragDetect** devuelve TRUE si el usuario inicia un gesto del mouse que debe interpretarse como arrastre. El siguiente código muestra cómo usar esta función.

```
caso WM_LBUTTONDOWN:
{
    POINT pt = {GET_X_LPARAM (lParam), GET_Y_LPARAM (lParam)};
    if (DragDetect (m_hwnd, pt))
    {
        // Comience a arrastrar.
    }
}
return 0;
```

Aquí está la idea: cuando un programa admite arrastrar y soltar, no quiere que cada clic del mouse se interprete como un arrastre. De lo contrario, el usuario puede arrastrar accidentalmente algo cuando simplemente quiere hacer clic en él (por ejemplo, para seleccionarlo). Pero si un mouse es particularmente sensible, puede ser difícil mantener el mouse perfectamente quieto al hacer clic. Por lo tanto, Windows define un umbral de arrastre de unos pocos píxeles. Cuando el usuario presiona el botón del mouse, no se considera un arrastre a menos que el mouse cruce este umbral. La función **DragDetect** prueba si se alcanza este umbral. Si la función devuelve TRUE, puede interpretar el clic del mouse como un arrastre. De lo contrario, no.

Nota Si **DragDetect** devuelve FALSE, Windows suprime el mensaje **WM_LBUTTONUP** cuando el usuario suelta el botón del mouse. Por lo tanto, no llame a **DragDetect** a menos que su programa esté actualmente en un modo que admita el arrastre. (Por ejemplo, si ya se ha seleccionado un elemento de IU arrastrable). Al final de este módulo, veremos un ejemplo de código más largo que usa la función **DragDetect**.

Confinando el cursor

En ocasiones, es posible que desee restringir el cursor al área del cliente o a una parte del área del cliente. La función **ClipCursor** restringe el movimiento del cursor a un rectángulo especificado. Este rectángulo se da en coordenadas de pantalla, en lugar de coordenadas del cliente, por lo que el punto (0, 0) significa la esquina superior izquierda de la pantalla. Para traducir las coordenadas del cliente en coordenadas de pantalla, llame a la función **ClientToScreen**.

El siguiente código limita el cursor al área del cliente de la ventana.

```
// Get the window client area.
RECT rc;
GetClientRect(m_hwnd, &rc);

// Convert the client area to screen coordinates.
```

```
POINT pt = { rc.left, rc.top };
POINT pt2 = { rc.right, rc.bottom };
ClientToScreen(m_hwnd, &pt);
ClientToScreen(m_hwnd, &pt2);
SetRect(&rc, pt.x, pt.y, pt2.x, pt2.y);

// Confine the cursor.
ClipCursor(&rc);
```

ClipCursor toma una **RECT** estructura, pero **ClientToScreen** toma una **PUNTO** estructura. Un rectángulo se define por sus puntos superior izquierdo e inferior derecho. Puede limitar el cursor a cualquier área rectangular, incluidas las áreas fuera de la ventana, pero limitar el cursor al área del cliente es una forma típica de usar la función. Limitar el cursor a una región completamente fuera de su ventana sería inusual, y los usuarios probablemente lo percibirían como un error. Para eliminar la restricción, llame a **ClipCursor** con el valor NULL.

```
ClipCursor (NULL);
```

Eventos de seguimiento de mouse: hover y leave

Otros dos mensajes del mouse están deshabilitados de manera predeterminada, pero pueden ser útiles para algunas aplicaciones:

- **WM_MOUSEHOVER** : el cursor ha estado sobre el área del cliente durante un período de tiempo fijo.
- **WM_MOUSELEAVE** : el cursor ha salido del área del cliente.

Para habilitar estos mensajes, llame a la función **TrackMouseEvent** .

```
TRACKMOUSEEVENT tme;
tme.cbSize = sizeof(tme);
tme.hwndTrack = hwnd;
tme.dwFlags = TME_HOVER | TME_LEAVE;
tme.dwHoverTime = HOVER_DEFAULT;
TrackMouseEvent(&tme);
```

La estructura **TRACKMOUSEEVENT** contiene los parámetros para la función.

El miembro **dwFlags** de la estructura contiene indicadores de bits que especifican qué mensajes de seguimiento le interesan. Puede optar por obtener **WM_MOUSEHOVER** y **WM_MOUSELEAVE**, como se muestra aquí, o solo uno de los dos. El miembro **dwHoverTime** especifica cuánto tiempo debe pasar el mouse antes de que el sistema genere un mensaje emergente. Este valor se da en milisegundos. La constante **HOVER_DEFAULT** significa usar el sistema predeterminado.

Después de recibir uno de los mensajes que solicitó, se restablece la función

TrackMouseEvent. Debe volver a llamar para obtener otro mensaje de seguimiento. Sin embargo, debe esperar hasta el siguiente mensaje de mover el mouse antes de **volver a** llamar a **TrackMouseEvent** . De lo contrario, su ventana podría estar inundada de mensajes de seguimiento. Por ejemplo, si el mouse está en el aire, el sistema continuará generando una secuencia de mensajes **WM_MOUSEHOVER** mientras el mouse está parado. En realidad, no

desea otro mensaje **WM_MOUSEHOVER** hasta que el mouse se mueva a otro punto y se desplace nuevamente.

Aquí hay una pequeña clase de ayuda que puede usar para administrar eventos de rastreo de mouse.

```
class MouseTrackEvents
{
    bool m_bMouseTracking;

public:
    MouseTrackEvents() : m_bMouseTracking(false)
    {
    }

    void OnMouseMove(HWND hwnd)
    {
        if (!m_bMouseTracking)
        {
            // Enable mouse tracking.
            TRACKMOUSEEVENT tme;
            tme.cbSize = sizeof(tme);
            tme.hwndTrack = hwnd;
            tme.dwFlags = TME_HOVER | TME_LEAVE;
            tme.dwHoverTime = HOVER_DEFAULT;
            TrackMouseEvent(&tme);
            m_bMouseTracking = true;
        }
    }

    void Reset(HWND hwnd)
    {
        m_bMouseTracking = false;
    }
};
```

El siguiente ejemplo muestra cómo usar esta clase en su procedimiento de ventana.

```
LRESULT MainWindow::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_MOUSEMOVE:
            mouseTrack.OnMouseMove(m_hwnd); // Start tracking.

            // TODO: Handle the mouse-move message.

            return 0;

        case WM_MOUSELEAVE:

            // TODO: Handle the mouse-leave message.

            mouseTrack.Reset(m_hwnd);
            return 0;

        case WM_MOUSEHOVER:
```

```

        // TODO: Handle the mouse-hover message.

        mouseTrack.Reset(m_hwnd);
        return 0;

    }
    return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
}

```

Los eventos de rastreo de mouse requieren un procesamiento adicional por parte del sistema, por lo tanto déjelos deshabilitados si no los necesita. Para completar, aquí hay una función que consulta al sistema por el tiempo de espera excedido predeterminado.

```

UINT GetMouseHoverTime()
{
    UINT msec;
    if (SystemParametersInfo(SPI_GETMOUSEHOVERTIME, 0, &msec, 0))
    {
        return msec;
    }
    else
    {
        return 0;
    }
}

```

Rueda de ratón

La siguiente función comprueba si hay una rueda del mouse presente.

```

BOOL IsMouseWheelPresent ()
{
    return (GetSystemMetrics (SM_MOUSEWHEELPRESENT) != 0);
}

```

Si el usuario gira la rueda del mouse, la ventana con foco recibe un mensaje **WM_MOUSEWHEEL**. El parámetro *lParam* de este mensaje contiene un valor entero llamado *delta* que mide cuánto giró la rueda. El delta usa unidades arbitrarias, donde 120 unidades se definen como la rotación necesaria para realizar una "acción". Por supuesto, la definición de una acción depende de tu programa. Por ejemplo, si la rueda del mouse se usa para desplazar el texto, cada 120 unidades de rotación desplazarán una línea de texto. El signo del delta indica la dirección de rotación:

- Positivo: gire hacia adelante, lejos del usuario.
- Negativo: gire hacia atrás, hacia el usuario.

El valor del delta se coloca en *lParam* junto con algunos indicadores adicionales. Use la macro [GET_WHEEL_DELTA_WPARAM](#) para obtener el valor del delta.

```
int delta = GET_WHEEL_DELTA_WPARAM (wParam);
```

Si la rueda del mouse tiene una resolución alta, el valor absoluto del delta puede ser menor que 120. En ese caso, si tiene sentido que la acción ocurra en incrementos más pequeños, puede hacerlo. Por ejemplo, el texto podría desplazarse por incrementos de menos de una línea. De lo contrario, acumula el delta total hasta que la rueda gire lo suficiente como para realizar la acción. Almacene el delta no utilizado en una variable, y cuando se acumulen 120 unidades (positivas o negativas), realice la acción.

Configuración de la imagen del cursor

El *cursor* es la imagen pequeña que muestra la ubicación del mouse u otro dispositivo señalador. Muchas aplicaciones cambian la imagen del cursor para proporcionar comentarios al usuario. Aunque no es necesario, agrega un poco de brillo a su aplicación.

Windows proporciona un conjunto de imágenes de cursor estándar, denominadas *cursores del sistema*. Estos incluyen la flecha, la mano, la viga en I, el reloj de arena (que ahora es un círculo giratorio) y otros. Esta sección describe cómo usar los cursores del sistema. Para tareas más avanzadas, como crear cursores personalizados, vea [Cursores](#).

Puede asociar un cursor a una clase de ventana configurando el miembro **hCursor** de la estructura [WNDCLASS](#) o [WNDCLASSEX](#). De lo contrario, el cursor predeterminado es la flecha. Cuando el mouse se mueve sobre una ventana, la ventana recibe un mensaje [WM_SETCURSOR](#) (a menos que otra ventana haya capturado el mouse). En este punto, ocurre uno de los siguientes eventos:

- La aplicación establece el cursor y el procedimiento de ventana devuelve TRUE.
- La aplicación no hace nada y pasa [WM_SETCURSOR](#) a [DefWindowProc](#).

Para establecer el cursor, un programa hace lo siguiente:

1. Llama a [LoadCursor](#) para cargar el cursor en la memoria. Esta función devuelve un mango al cursor.
2. Llama a [SetCursor](#) y pasa en el controlador del cursor.

De lo contrario, si la aplicación pasa [WM_SETCURSOR](#) a [DefWindowProc](#), la función **DefWindowProc** usa el siguiente algoritmo para establecer la imagen del cursor:

1. Si la ventana tiene un elemento primario, reenvía el mensaje [WM_SETCURSOR](#) al padre para que lo [administre](#).
2. De lo contrario, si la ventana tiene un cursor de clase, ajuste el cursor al cursor de clase.
3. Si no hay un cursor de clase, ajuste el cursor al cursor de flecha.

La función [LoadCursor](#) puede cargar un cursor personalizado desde un recurso o uno de los cursores del sistema. El siguiente ejemplo muestra cómo establecer el cursor en el cursor de la mano del sistema.

```
hCursor = LoadCursor(NULL, cursor);  
SetCursor (hCursor);
```

Si cambia el cursor, la imagen del cursor se restablece en el siguiente movimiento del mouse, a menos que intercepte el mensaje **WM_SETCURSOR** y establezca el cursor nuevamente. El siguiente código muestra cómo manejar **WM_SETCURSOR**.

```
caso WM_SETCURSOR:  
    if (LOWORD (lParam) == HTCLIENT)  
    {  
        SetCursor (hCursor);  
        devuelve VERDADERO;  
    }  
    descanso;
```

Este código primero verifica los 16 bits más bajos de *lParam*. Si `LOWORD(lParam)` es igual a `HTCLIENT`, significa que el cursor está sobre el área del cliente de la ventana. De lo contrario, el cursor está sobre el área no cliente. Normalmente, solo debe establecer el cursor para el área del cliente y dejar que Windows configure el cursor para el área no cliente.

Ejercicio 4. Utilizar el ratón para dibujar elipses

1. Se deben dibujar elipses en cualquier zona de la pantalla mediante pulsación y arrastre del ratón.
2. Se debe mantener el reloj en $\frac{1}{8}$ del área cliente, rescaldándose automáticamente si cambia el tamaño de la ventana.



Entrada de teclado

El teclado se usa para varios tipos distintos de entrada, que incluyen:

- Entrada de caracteres Texto que el usuario escribe en un documento o cuadro de edición.
- Atajos de teclado: Key strokes que invocan funciones de aplicación; por ejemplo, CTRL + O para abrir un archivo.
- Comandos del sistema: pulsaciones de teclas que invocan funciones del sistema; por ejemplo, ALT + TAB para cambiar de ventana.

Al pensar en la entrada de teclado, es importante recordar que pulsación de tecla no es lo mismo que un carácter. Por ejemplo, presionar la tecla A podría resultar en cualquiera de los siguientes caracteres.

- a
- A
- á (si el teclado admite la combinación de signos diacríticos)

Además, si la tecla ALT se mantiene presionada, presionar la tecla A produce ALT + A, que el sistema no trata como un carácter, sino como un comando del sistema.

Códigos clave

Cuando presiona una tecla, el hardware genera un *código de exploración*. Los códigos de escaneo varían de un teclado al siguiente, y hay códigos de escaneo separados para eventos de activación y desactivación de teclas. Casi nunca te importará los códigos de escaneo. El controlador de teclado traduce los códigos de escaneo en *códigos de tecla virtual*. Los códigos de clave virtual son independientes del dispositivo. Presionar la tecla A en cualquier teclado genera el mismo código de tecla virtual.

En general, los códigos de clave virtual no se corresponden con los códigos ASCII o cualquier otro estándar de codificación de caracteres. Esto es obvio si lo piensas, porque la misma clave puede generar diferentes caracteres (a, A, á), y algunas teclas, como las teclas de función, no se corresponden con ningún carácter.

Dicho esto, los siguientes códigos de clave virtual se asignan a equivalentes ASCII:

- 0 a 9 teclas = ASCII '0' - '9' (0x30 - 0x39)
- Teclas de la A a la Z = ASCII 'A' - 'Z' (0x41 - 0x5A)

En algunos aspectos, este mapeo es desafortunado, ya que nunca se debe pensar en los códigos de clave virtual como caracteres, por las razones discutidas.

El archivo de encabezado WinUser.h define constantes para la mayoría de los códigos de clave virtual. Por ejemplo, el código de la tecla virtual para la tecla FLECHA IZQUIERDA es VK_LEFT (0x25). Para obtener la lista completa de códigos de clave virtual, vea [Códigos de clave virtual](#). No se definen constantes para los códigos de clave virtual que coinciden con los valores ASCII. Por ejemplo, el código de la tecla virtual para la tecla A es 0x41, pero no hay una constante llamada VK_A. En cambio, solo usa el valor numérico.

Mensajes Key-Down y Key-Up

Cuando presiona una tecla, la ventana que tiene el foco del teclado recibe uno de los siguientes mensajes.

- [WM_SYSKEYDOWN](#)
- [WM_KEYDOWN](#)

El mensaje [WM_SYSKEYDOWN](#) indica una *clave del sistema*, que es un trazo clave que invoca un comando del sistema. Hay dos tipos de claves del sistema:

- ALT + cualquier tecla
- F10

La tecla F10 activa la barra de menú de una ventana. Varias combinaciones de teclas ALT invocan comandos del sistema. Por ejemplo, ALT + TAB cambia a una nueva ventana. Además, si una ventana tiene un menú, la tecla ALT se puede usar para activar los elementos del menú. Algunas combinaciones de teclas ALT no hacen nada.

Todas las demás pulsaciones de teclas se consideran claves que no son del sistema y producen el mensaje [WM_KEYDOWN](#). Esto incluye las teclas de función que no sean F10.

Cuando suelta una tecla, el sistema envía un mensaje de activación de teclado correspondiente:

- [WM_KEYUP](#)
- [WM_SYSKEYUP](#)

Si mantiene presionada una tecla durante el tiempo suficiente para iniciar la función de repetición del teclado, el sistema envía múltiples mensajes de tecla pulsada, seguidos por un mensaje de activación único.

En los cuatro mensajes de teclado discutidos hasta ahora, el parámetro *wParam* contiene el código de clave virtual de la clave. El parámetro *lParam* contiene información miscelánea empaquetada en 32 bits. Por lo general, no necesita la información en *lParam*. Una bandera que podría ser útil es el bit 30, el indicador de "estado de tecla anterior", que se establece en 1 para mensajes de tecla repetidos.

Como su nombre lo indica, las pulsaciones de teclas del sistema están destinadas principalmente para su uso por parte del sistema operativo. Si interceptas el mensaje [WM_SYSKEYDOWN](#), llama [DefWindowProc](#) después. De lo contrario, bloqueará el sistema operativo para que no maneje el comando.

Mensajes de caracteres

Las pulsaciones de teclas se convierten en caracteres mediante la función [TranslateMessage](#), que vimos por primera vez en el [Módulo 1](#). Esta función examina los mensajes de pulsación y los traduce en caracteres. Para cada carácter que se produce, la función [TranslateMessage](#) coloca un mensaje [WM_CHAR](#) o [WM_SYSCHAR](#) en la cola de mensajes de la ventana.

El parámetro *wParam* del mensaje contiene el carácter UTF-16.

Como se puede adivinar, los mensajes [WM_CHAR](#) se generan a partir de mensajes [WM_KEYDOWN](#), mientras que los mensajes [WM_SYSCHAR](#) se generan a partir de mensajes [WM_SYSKEYDOWN](#). Por ejemplo, supongamos que el usuario presiona la tecla MAYÚS seguido

de la tecla A. Suponiendo un diseño de teclado estándar, obtendría la siguiente secuencia de mensajes:

```
WM_KEYDOWN: SHIFT
WM_KEYDOWN: A
WM_CHAR: 'A'
```

Por otro lado, la combinación ALT + P generaría:

```
WM_SYSKEYDOWN: VK_MENU
WM_SYSKEYDOWN: 0x50
WM_SYSCHAR: 'p'
WM_SYSKEYUP: 0x50
WM_KEYUP: VK_MENU
```

(El código de clave virtual para la clave ALT se llama VK_MENU por razones históricas).

El mensaje **WM_SYSCHAR** indica un carácter de sistema. Al igual que con **WM_SYSKEYDOWN**, generalmente debe pasar este mensaje directamente a **DefWindowProc**. De lo contrario, puede interferir con los comandos del sistema estándar. En particular, no trate **WM_SYSCHAR** como texto que el usuario haya tipeado.

El mensaje **WM_CHAR** es lo que normalmente piensas como entrada de caracteres. El tipo de datos para el personaje es **wchar_t**, que representa un carácter UTF-16 Unicode. La entrada de caracteres puede incluir caracteres fuera del rango ASCII, especialmente con diseños de teclado que se usan comúnmente fuera de los Estados Unidos. Puede probar diferentes diseños de teclado instalando un teclado regional y luego usando la función Teclado en pantalla.

Los usuarios también pueden instalar un Editor de métodos de entrada (IME) para ingresar scripts complejos, como caracteres japoneses, con un teclado estándar. Por ejemplo, al usar un IME japonés para ingresar el carácter katakana 力 (ka), es posible que reciba los siguientes mensajes:

```
WM_KEYDOWN: VK_PROCESSKEY (la tecla IME PROCESS)
WM_KEYUP: 0x4B
WM_KEYDOWN: VK_PROCESSKEY
WM_KEYUP: 0x41
WM_KEYDOWN: VK_PROCESSKEY
WM_CHAR: poder
WM_KEYUP: VK_RETURN
```

Algunas combinaciones de teclas CTRL se traducen en caracteres de control ASCII. Por ejemplo, CTRL + A se traduce al carácter ASCII ctrl-A (SOH) (valor ASCII 0x01). Para la entrada de texto, generalmente debe filtrar los caracteres de control. Además, evite usar **WM_CHAR** para implementar atajos de teclado. En cambio, use los mensajes **WM_KEYDOWN**; o mejor aún, usa una tabla de acelerador. Las tablas de acelerador se describen en el siguiente tema, [Tablas de aceleración](#).

El siguiente código muestra los mensajes principales del teclado en el depurador. Intente jugar con diferentes combinaciones de teclas y vea qué mensajes se generan.

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    wchar_t msg[32];
    switch (uMsg)
    {
        case WM_SYSKEYDOWN:
```

```

        swprintf_s(msg, L"WM_SYSKEYDOWN: 0x%x\n", wParam);
        OutputDebugString(msg);
        break;

    case WM_SYSCHAR:
        swprintf_s(msg, L"WM_SYSCHAR: %c\n", (wchar_t)wParam);
        OutputDebugString(msg);
        break;

    case WM_SYSKEYUP:
        swprintf_s(msg, L"WM_SYSKEYUP: 0x%x\n", wParam);
        OutputDebugString(msg);
        break;

    case WM_KEYDOWN:
        swprintf_s(msg, L"WM_KEYDOWN: 0x%x\n", wParam);
        OutputDebugString(msg);
        break;

    case WM_KEYUP:
        swprintf_s(msg, L"WM_KEYUP: 0x%x\n", wParam);
        OutputDebugString(msg);
        break;

    case WM_CHAR:
        swprintf_s(msg, L"WM_CHAR: %c\n", (wchar_t)wParam);
        OutputDebugString(msg);
        break;

    /* Handle other messages (not shown) */

}
return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
}

```

Mensajes de teclado diversos

Algunos otros mensajes de teclado pueden ser ignorados por la mayoría de las aplicaciones.

- El mensaje **WM_DEADCHAR** se envía para una combinación de teclas, como un signo diacrítico. Por ejemplo, en un teclado en español, escribir acento (') seguido de E produce el carácter é. El **WM_DEADCHAR** se envía para el carácter de acento.
- El mensaje **WM_UNICHAR** está obsoleto. Permite a los programas ANSI recibir entrada de caracteres Unicode.
- El carácter **WM_IME_CHAR** se envía cuando un IME traduce una secuencia de pulsaciones de teclas en caracteres. Se envía además del mensaje habitual **WM_CHAR**.

Estado del teclado

Los mensajes del teclado están controlados por eventos. Es decir, recibes un mensaje cuando sucede algo interesante, como presionar una tecla, y el mensaje te dice lo que acaba de suceder. Pero también puede probar el estado de una clave en cualquier momento, llamando a la función **GetKeyState**.

Por ejemplo, considere cómo detectaría la combinación de clic con el botón izquierdo del mouse + tecla ALT. Puede rastrear el estado de la tecla ALT escuchando los mensajes de pulsación de tecla y almacenando una marca, pero **GetKeyState** le ahorra el problema. Cuando reciba el mensaje **WM_LBUTTONDOWN** , simplemente llame a **GetKeyState** de la siguiente manera:

```
if (GetKeyState (VK_MENU) & 0x8000))
{
    // La tecla ALT está abajo.
}
```

El mensaje **GetKeyState** toma un código de clave virtual como entrada y devuelve un conjunto de indicadores de bits (en realidad solo dos indicadores). El valor 0x8000 contiene el indicador de bit que comprueba si la tecla está presionada actualmente.

La mayoría de los teclados tienen dos teclas ALT, izquierda y derecha. El ejemplo anterior prueba si alguno de ellos está presionado. También puede usar **GetKeyState** para distinguir entre las instancias izquierda y derecha de las teclas ALT, MAYÚS o CTRL. Por ejemplo, el siguiente código prueba si se presiona la tecla ALT derecha.

```
if (GetKeyState (VK_RMENU) & 0x8000))
{
    // La tecla ALT derecha está abajo.
}
```

La función **GetKeyState** es interesante porque informa un estado de teclado *virtual* . Este estado virtual se basa en el contenido de su cola de mensajes y se actualiza a medida que elimina los mensajes de la cola. A medida que su programa procesa los mensajes de la ventana, **GetKeyState** le brinda una instantánea del teclado en el momento en que cada mensaje se puso en cola. Por ejemplo, si el último mensaje en la cola era **WM_LBUTTONDOWN** , **GetKeyState** informa el estado del teclado en el momento en que el usuario hizo clic con el botón del mouse.

Debido a que **GetKeyState** se basa en su cola de mensajes, también ignora la entrada de teclado que se envió a otro programa. Si el usuario cambia a otro programa, **GetKeyState** ignorará cualquier pulsación de tecla que se envíe a ese programa. Si realmente quiere saber el estado físico inmediato del teclado, hay una función para eso: **GetAsyncKeyState** . Sin embargo, para la mayoría del código UI, la función correcta es **GetKeyState** .

Atajos de Teclado

Las aplicaciones a menudo definen atajos de teclado, como CTRL + O para el comando Abrir archivo. Podría implementar atajos de teclado mediante el manejo de mensajes individuales **WM_KEYDOWN** , pero las tablas de aceleradores ofrecen una mejor solución que:

- Requiere menos codificación.
- Consolida todos sus accesos directos en un solo archivo de datos.
- Admite la localización en otros idiomas.
- Permite que los atajos y los comandos de menú usen la misma lógica de aplicación.

Una *tabla de acelerador* es un recurso de datos que asigna combinaciones de teclado, como CTRL + O, a los comandos de la aplicación. Antes de ver cómo usar una tabla de acelerador,

necesitaremos una introducción rápida a los recursos. Un *recurso* es un blob de datos que está integrado en una aplicación binaria (EXE o DLL). Los recursos almacenan los datos que necesita la aplicación, como menús, cursores, íconos, imágenes, cadenas de texto o cualquier información personalizada de la aplicación. La aplicación carga los datos de recursos del binario en tiempo de ejecución. Para incluir recursos en un archivo binario, haga lo siguiente:

1. Cree un archivo de definición de recursos (.rc). Este archivo define los tipos de recursos y sus identificadores. El archivo de definición de recursos puede incluir referencias a otros archivos. Por ejemplo, un recurso icono se declara en el archivo .rc, pero la imagen del icono se almacena en un archivo separado.
2. Utilice el compilador de recursos de Microsoft Windows (RC) para compilar el archivo de definición de recursos en un archivo de recursos compilados (.res). El compilador RC se proporciona con Visual Studio y también con el SDK de Windows.
3. Enlace el archivo de recursos compilados al archivo binario.

Estos pasos son más o menos equivalentes al proceso de compilación / enlace para archivos de código. Visual Studio proporciona un conjunto de editores de recursos que facilitan la creación y modificación de recursos. (Estas herramientas no están disponibles en las ediciones Express de Visual Studio.) Pero un archivo .rc es simplemente un archivo de texto y la sintaxis está documentada en MSDN, por lo que es posible crear un archivo .rc utilizando cualquier editor de texto. Para obtener más información, consulte [Acerca de los archivos de recursos](#).

Definición de una tabla de aceleración

Una tabla de aceleradores es una tabla de atajos de teclado. Cada atajo está definido por:

- Un identificador numérico. Este número identifica el comando de la aplicación que será invocado por el atajo.
- El carácter ASCII o el código de la tecla virtual del acceso directo.
- Teclas modificadoras opcionales: ALT, SHIFT o CTRL.

La tabla de aceleradores tiene un identificador numérico que identifica la tabla en la lista de recursos de la aplicación. Vamos a crear una tabla de acelerador para un programa de dibujo simple. Este programa tendrá dos modos, modo dibujar y modo de selección. En el modo de dibujo, el usuario puede dibujar formas. En el modo de selección, el usuario puede seleccionar formas. Para este programa, nos gustaría definir los siguientes atajos de teclado.

Atajo	Mando
CTRL + M	Alternar entre modos.
F1	Cambiar al modo de dibujo.
F2	Cambiar al modo de selección.

Primero, defina los identificadores numéricos para la tabla y para los comandos de la aplicación. Estos valores son arbitrarios. Puede asignar constantes simbólicas para los identificadores definiéndolos en un archivo de encabezado. Por ejemplo:

```
#define IDR_ACCEL1 101
#define ID_TOGGLE_MODE 40002
#define ID_DRAW_MODE 40003
#define ID_SELECT_MODE 40004
```

En este ejemplo, el valor `IDR_ACCEL1` identifica la tabla del acelerador, y las siguientes tres constantes definen los comandos de la aplicación. Por convención, un archivo de cabecera que define las constantes de recursos a menudo se denomina `resource.h`. La siguiente lista muestra el archivo de definición de recursos.

```
#include "resource.h"
IDR_ACCEL1 ACCELERATORS
{
    0x4D,    ID_TOGGLE_MODE, VIRTKEY, CONTROL    // ctrl-M
    0x70,    ID_DRAW_MODE,  VIRTKEY             // F1
    0x71,    ID_SELECT_MODE, VIRTKEY             // F2
}
```

Los accesos directos del acelerador están definidos dentro de las llaves. Cada atajo contiene las siguientes entradas.

- El código de tecla virtual o el carácter ASCII que invoca el acceso directo.
- El comando de la aplicación. Tenga en cuenta que las constantes simbólicas se utilizan en el ejemplo. El archivo de definición de recurso incluye `resource.h`, donde se definen estas constantes.
- La palabra clave **VIRTKEY** significa que la primera entrada es un código de tecla virtual. La otra opción es usar caracteres ASCII.
- Modificadores opcionales: `ALT`, `CONTROL` o `SHIFT`.

Si usa caracteres ASCII para accesos directos, entonces un carácter en minúsculas será un atajo diferente que un carácter en mayúscula. (Por ejemplo, escribir "a" podría invocar un comando diferente de escribir "A"). Eso podría confundir a los usuarios, por lo que generalmente es mejor usar códigos de tecla virtual, en lugar de caracteres ASCII, para accesos directos.

Cargando la tabla de aceleración

El recurso para la tabla de aceleradores debe cargarse antes de que el programa pueda usarlo. Para cargar una tabla de acelerador, llame a la función [LoadAccelerators](#) .

```
HACCEL hAccel = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDR_ACCEL1));
```

Llame a esta función antes de ingresar al ciclo de mensajes. El primer parámetro es el manejador del módulo (este parámetro se pasa a su función [WinMain](#)) . El segundo parámetro es el identificador del recurso. La función devuelve un identificador al recurso. Recuerde que un identificador es un tipo opaco que hace referencia a un objeto administrado por el sistema. Si la función falla, devuelve `NULL`.

Puede liberar una tabla de aceleradores llamando a [DestroyAcceleratorTable](#) . Sin embargo, el sistema libera automáticamente la tabla cuando el programa sale, por lo que solo necesita llamar a esta función si está reemplazando una tabla por otra. Hay un ejemplo interesante de esto en el tema [Creación de aceleradores modificables por el usuario](#) .

Traducir pulsaciones de teclas en comandos

Una tabla de acelerador funciona traduciendo pulsaciones de tecla en mensajes **WM_COMMAND**. El parámetro *wParam* de **WM_COMMAND** contiene el identificador numérico del comando. Por ejemplo, utilizando la tabla que se muestra anteriormente, la **pulsación** de tecla CTRL + M se traduce en un mensaje **WM_COMMAND** con el valor ID_TOGGLE_MODE. Para que esto suceda, cambie su ciclo de mensajes a lo siguiente:

```
MSG msg;
while (GetMessage(&msg, NULL, 0, 0))
{
    if (!TranslateAccelerator(win.Window(), hAccel, &msg))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

Este código agrega una llamada a la función **TranslateAccelerator** dentro del ciclo del mensaje. La función **TranslateAccelerator** examina cada mensaje de ventana, en busca de mensajes de pulsaciones. Si el usuario presiona una de las combinaciones de teclas que figuran en la tabla del acelerador, **TranslateAccelerator** envía un mensaje **WM_COMMAND** a la ventana. La función envía **WM_COMMAND** invocando directamente el procedimiento de ventana. Cuando **TranslateAccelerator** traduce con éxito una pulsación de tecla, la función devuelve un valor distinto de cero, lo que significa que debe omitir el procesamiento normal del mensaje. De lo contrario, **TranslateAccelerator** devuelve cero. En ese caso, pase el mensaje de ventana a **TranslateMessage** y **DispatchMessage**, como es normal. Así es como el programa de dibujo podría manejar el mensaje **WM_COMMAND**:

```
case WM_COMMAND:
    switch (LOWORD(wParam))
    {
        case ID_DRAW_MODE:
            SetMode(DrawMode);
            break;
        case ID_SELECT_MODE:
            SetMode(SelectMode);
            break;
        case ID_TOGGLE_MODE:
            if (mode == DrawMode)
            {
                SetMode(SelectMode);
            }
            else
            {
                SetMode(DrawMode);
            }
            break;
    }
    return 0;
```

Este código asume que **SetMode** es una función definida por la aplicación para cambiar entre los dos modos. Los detalles de cómo manejaría cada comando obviamente dependen de su programa.

Módulo 4. Teclado y Ratón

- Como ejercicio de teclado queremos que:
 - a. La tecla ESPACIO sirva para *parar/continuar* el reloj. Capturar la pulsación de la tecla ESPACIO y pasar el reloj a un modo propio *stop/run*. En este modo la aplicación no debe hacer caso al TIMER.
 - b. La tecla ESC debe cerrar la aplicación enviando el mensaje WM_DESTROY.

Se recomienda generar dos **Modos de la Aplicación**:

```
enum class ClockMode { RunMode, StopMode };
```

- Como ejercicio de procesamiento de eventos de ratón se deben:
 - a. dibujar,
 - b. modificar
 - c. y arrastrar elipses.

Se recomienda definir diferentes **Modos del Interfaz de Usuario**:

```
enum class EditionMode { SelectMode, DrawMode, DragMode };
```

Para mostrar potencialidades al usuario se debe cambiar el cursor del ratón según el modo en el que nos encontremos. Se deben implementar las siguientes funcionalidades:

- Al hacer *Clic izquierdo* y arrastrar debemos crear y dibujar la elipse mostrando el cursor *IDC_CROSS*.
- Para mostrar potencialidades, debemos tener una función que nos diga si estamos encima de la elipse y consultarla en el evento *WM_MOUSEMOVE*.

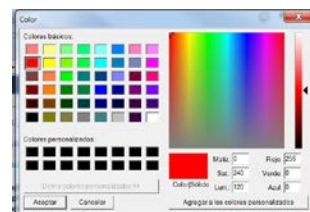
```
BOOL HitTest(D2D1_ELLIPSE ellipse, float x, float y)
```

 - a. Dentro de la elipse el cursor debe ser la *IDC_HAND*.
 - b. Fuera de la elipse el cursor debe ser la *IDC_ARROW*.
- Al hacer *Clic derecho* sobre la elipse se debe seleccionar el color de relleno. La forma más sencilla de hacer esto es con un cuadro de diálogo *ChosseColor*. Este código permite crear el cuadro de diálogo que devolverá un color seleccionado, utilizando la API de Windows.

```
CHOOSECOLOR cc;           // common dialog box structure
static COLORREF acrCustClr[16]; // array of custom colors
static DWORD rgbCurrent;    // initial color selection
```

```
// Initialize CHOOSECOLOR
ZeroMemory(&cc, sizeof(cc));
cc.lStructSize = sizeof(cc);
cc.hwndOwner = m_hwnd;
cc.lpCustColors = (LPDWORD)acrCustClr;
cc.rgbResult = rgbCurrent;
cc.Flags = CC_FULLOPEN | CC_RGBINIT;
```

```
if (ChooseColor(&cc) == TRUE)
{
    //En cc.rgbResult tenemos el color seleccionado
    //Utilizarlo para configurar nuestra brocha
    //Es necesario transformarlo al formato de color de D2D
}
```



- Además de procesar eventos, otras veces es necesario reconocer y programar gestos. **Programar el gesto de arrastrar un objeto gráfico, p.e la elipse.** Es necesario mostrar el resultado del arrastre en directo para que el usuario pueda decidir cuándo soltar la elipse.