

Introdução ao Win32 e C++

Artigo • 13/06/2023

O objetivo desta série Introdução é ensinar você a escrever um programa de área de trabalho em C++ usando APIs Win32 e COM.

No primeiro módulo, você aprenderá passo a passo como criar e mostrar uma janela. Os módulos posteriores apresentarão o COM (Component Object Model), os elementos gráficos e o texto e a entrada do usuário.

Para esta série, supõe-se que você tenha um bom conhecimento de trabalho sobre programação C++. Nenhuma experiência anterior com a programação do Windows é assumida. Se você for novo no C++, o material de aprendizagem estará disponível na [documentação da linguagem C++](#).

Nesta seção

 Expandir a tabela

Tópico	Descrição
Introdução à programação win32 no C++	Esta seção descreve algumas das convenções básicas de terminologia e codificação usadas na programação do Windows.
Módulo 1. Seu primeiro programa do Windows	Neste módulo, você criará um programa simples do Windows que mostra uma janela em branco.
Módulo 2. Usando COM em seu programa do Windows	Este módulo apresenta o COM (Component Object Model), que está por trás de muitas das APIs modernas do Windows.
Módulo 3. Gráficos do Windows	Este módulo apresenta a arquitetura gráfica do Windows, com foco em Direct2D.
Módulo 4. Entrada do usuário	Este módulo descreve a entrada de mouse e teclado.
Código de exemplo	Contém links para baixar o código de exemplo desta série.

Comentários

Esta página foi útil?

☐ Yes

☐ No

Introdução à programação win32 no C++

Artigo • 13/06/2023

Esta seção descreve algumas das convenções básicas de terminologia e codificação usadas na programação do Windows.

Nesta seção

- [Preparar seu ambiente de desenvolvimento](#)
- [Convenções de codificação do Windows](#)
- [Trabalhar com cadeias de caracteres](#)
- [O que é uma janela?](#)
- [WinMain: o ponto de entrada do aplicativo](#)

Tópicos relacionados

[Introdução ao Win32 e C++](#)

[Módulo 1. Seu primeiro programa do Windows](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Preparar seu ambiente de desenvolvimento

Artigo • 13/06/2023

Para escrever um programa do Windows em C ou C++, você deve instalar o Microsoft Windows Software Development Kit (SDK) ou o Microsoft Visual Studio. O SDK do Windows contém os cabeçalhos e bibliotecas necessários para compilar e vincular seu aplicativo. O SDK do Windows também contém ferramentas de linha de comando para criar aplicativos do Windows, incluindo o compilador e o vinculador do Visual C++. Embora você possa compilar e compilar programas do Windows com as ferramentas de linha de comando, é recomendável usar o Microsoft Visual Studio. Você pode baixar um download gratuito de Visual Studio Community ou avaliações gratuitas de outras versões do Visual Studio [aqui](#).

Cada versão do SDK do Windows tem como destino a versão mais recente do Windows, bem como várias versões anteriores. As notas sobre a versão listam as plataformas específicas com suporte, mas, a menos que você esteja mantendo um aplicativo para uma versão muito antiga do Windows, você deve instalar a versão mais recente do SDK do Windows. Você pode baixar o SDK do Windows mais recente [aqui](#).

O SDK do Windows dá suporte ao desenvolvimento de aplicativos de 32 e 64 bits. As APIs do Windows foram projetadas para que o mesmo código possa ser compilado para 32 bits ou 64 bits sem alterações.

⚠ Observação

O SDK do Windows não dá suporte ao desenvolvimento de driver de hardware e esta série não discutirá o desenvolvimento de driver. Para obter informações sobre como escrever um driver de hardware, consulte [Introdução com drivers do Windows](#).

Avançar

[Convenções de codificação do Windows](#)

Tópicos relacionados

- [Baixar o Visual Studio](#)

- [Baixar o SDK do Windows](#)
-

Comentários

Esta página foi útil?



Yes



No

[Obter ajuda no Microsoft Q&A](#)

Convenções de codificação do Windows

Artigo • 13/06/2023

Se você é novo na programação do Windows, pode ser desconcertante quando você vê um programa do Windows pela primeira vez. O código é preenchido com definições de tipo estranhas, como **DWORD_PTR** e **LPRECT**, e as variáveis têm nomes como *hWnd* e *pwsz* (chamado notação húngara). Vale a pena tirar um momento para aprender algumas das convenções de codificação do Windows.

A grande maioria das APIs do Windows consiste em funções ou interfaces COM (Component Object Model). Pouquíssimas APIs do Windows são fornecidas como classes C++. (Uma exceção notável é GDI+, uma das APIs gráficas 2D.)

Typedefs

Os cabeçalhos do Windows contêm muitos typedefs. Muitos deles são definidos no arquivo de cabeçalho WinDef.h. Aqui estão alguns que você encontrará com frequência.

Tipos de inteiro

Tipo de dados	Tamanho	Assinado?
BYTE	8 bits	Não assinado
DWORD	32 bits	Não assinado
INT32	32 bits	Com sinal
INT64	64 bits	Com sinal
LONG	32 bits	Com sinal
ONGLONG	64 bits	Com sinal
UINT32	32 bits	Não assinado
UINT64	64 bits	Não assinado
ULONG	32 bits	Não assinado
ULONGLONG	64 bits	Não assinado
WORD	16 bits	Não assinado

Como você pode ver, há uma certa quantidade de redundância nesses typedefs. Parte dessa sobreposição é simplesmente devido ao histórico das APIs do Windows. Os tipos listados aqui têm tamanho fixo e os tamanhos são os mesmos em aplicativos de 32 e 64 bits. Por exemplo, o tipo **DWORD** sempre tem 32 bits de largura.

Tipos boolianos

BOOL é um alias de tipo para **int**, diferente do **bool** do C++ e de outros tipos que representam um valor [booliano](#). O arquivo `WinDef.h` de cabeçalho também define dois valores para uso com **BOOL**.

C++

```
#define FALSE    0
#define TRUE     1
```

Apesar dessa definição de **TRUE**, no entanto, a maioria das funções que retornam um tipo **BOOL** pode retornar qualquer valor diferente de zero para indicar a verdade booliana. Portanto, você sempre deve escrever isso:

C++

```
// Right way.
if (SomeFunctionThatReturnsBoolean())
{
    ...
}

// or

if (SomeFunctionThatReturnsBoolean() != FALSE)
{
    ...
}
```

e não isso:

C++

```
if (result == TRUE) // Wrong!
{
    ...
}
```

BOOL é um tipo inteiro e não é intercambiável com o **bool** do C++.

Tipos de ponteiro

O Windows define muitos tipos de dados do *ponteiro para X* do formulário. Geralmente, eles têm o prefixo *P* ou *LP* no nome. Por exemplo, **LPRECT** é um ponteiro para um **RECT**, em que **RECT** é uma estrutura que descreve um retângulo. As declarações de variável a seguir são equivalentes.

C++

```
RECT* rect; // Pointer to a RECT structure.  
LPRECT rect; // The same  
PRECT rect; // Also the same.
```

Em arquiteturas de 16 bits (Windows de 16 bits) há dois tipos de ponteiros, *P* para "ponteiro" e *LP* significa "ponteiro longo". Ponteiros longos (também *chamados de ponteiros distantes*) eram necessários para abordar intervalos de memória fora do segmento atual. O prefixo *LP* foi preservado para facilitar a portabilidade do código de 16 bits para o Windows de 32 bits. Hoje não há distinção e esses tipos de ponteiro são todos equivalentes. Evite usar esses prefixos; ou se você precisar usar um, use *P*.

Tipos de precisão de ponteiro

Os tipos de dados a seguir são sempre o tamanho de um ponteiro, ou seja, 32 bits de largura em aplicativos de 32 bits e 64 bits de largura em aplicativos de 64 bits. O tamanho é determinado em tempo de compilação. Quando um aplicativo de 32 bits é executado no Windows de 64 bits, esses tipos de dados ainda têm 4 bytes de largura. (Um aplicativo de 64 bits não pode ser executado no Windows de 32 bits, portanto, a situação inversa não ocorre.)

- **DWORD_PTR**
- **INT_PTR**
- **LONG_PTR**
- **ULONG_PTR**
- **UINT_PTR**

Esses tipos são usados em situações em que um inteiro pode ser convertido em um ponteiro. Eles também são usados para definir variáveis para a aritmética de ponteiro e para definir contadores de loop que iteram sobre o intervalo completo de bytes em buffers de memória. Em geral, eles aparecem em locais onde um valor de 32 bits existente foi expandido para 64 bits no Windows de 64 bits.

Notação húngara

Notação húngara é a prática de adicionar prefixos aos nomes das variáveis para fornecer informações adicionais sobre a variável. (O inventor da notação, Charles Simonyi, era húngaro, daí seu nome).

Em sua forma original, a notação húngara fornece informações *semânticas* sobre uma variável, informando o uso pretendido. Por exemplo, *quero* dizer um índice, *cb* significa um tamanho em bytes ("contagem de bytes") e *rw* e *números* de linha média e coluna. Esses prefixos foram projetados para evitar o uso acidental de uma variável no contexto errado. Por exemplo, se você visse a expressão `rwPosition + cbTable`, saberia que um número de linha está sendo adicionado a um tamanho, o que é quase certamente um bug no código

Uma forma mais comum de notação húngara usa prefixos para fornecer informações de *tipo*, por exemplo, *dw* para **DWORD** e *w* para **WORD**.

ⓘ Observação

As **Diretrizes Principais do C++** [↗](#) desencorajam a notação de prefixo (por exemplo, notação húngara). Consulte **NL.5: evitar informações de tipo de codificação em nomes** [↗](#). Internamente, a equipe do Windows não o usa mais. Mas seu uso permanece em exemplos e documentação.

Avançar

[Trabalhar com cadeias de caracteres](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Trabalhar com cadeias de caracteres

Artigo • 07/08/2024

Este tópico explica como o Windows dá suporte a cadeias de caracteres Unicode para elementos da interface do usuário, nomes de arquivo e assim por diante (Unicode é a codificação de caracteres preferencial porque dá suporte a todos os conjuntos de caracteres e idiomas).

O Windows representa caracteres Unicode usando a codificação UTF-16, na qual cada caractere é codificado como um ou dois valores de 16 bits. Para distingui-los dos caracteres ANSI de 8 bits, os caracteres UTF-16 são chamados *de caracteres largos*. O compilador do Visual C++ dá suporte ao tipo de dados internos `wchar_t` para caracteres largos. O arquivo de cabeçalho `WinNT.h` também define o `typedef` a seguir.

C++


```
typedef wchar_t WCHAR;
```

Para declarar um literal de caractere largo ou um literal de cadeia de caracteres largos, coloque `L` antes do literal.

C++

```
wchar_t a = L'a';  
wchar_t *str = L"hello";
```

A tabela a seguir lista alguns outros `typedefs` relacionados a cadeia de caracteres:

 Expandir a tabela

Typedef	Definição
CHAR	<code>char</code>
PSTR ou LPSTR	<code>char*</code>
PCSTR ou LPCSTR	<code>const char*</code>
PWSTR ou LPWSTR	<code>wchar_t*</code>
PCWSTR ou LPCWSTR	<code>const wchar_t*</code>

Funções Unicode e ANSI

Quando a Microsoft introduziu o suporte a Unicode no Windows, ela facilitou a transição fornecendo dois conjuntos paralelos de APIs, um para cadeias de caracteres ANSI e outro para cadeias de caracteres Unicode. Por exemplo, há duas funções para definir o texto da barra de título de uma janela:

- **SetWindowTextA** recebe uma string ANSI.
- **SetWindowTextW** recebe uma string Unicode.

Internamente, a versão ANSI converte a cadeia de caracteres em Unicode. Os cabeçalhos do Windows também definem uma macro que resolve para a versão Unicode quando o símbolo de pré-processador `UNICODE` é definido ou para a versão ANSI, em caso contrário.

C++

```
#ifdef UNICODE
#define SetWindowText SetWindowTextW
#else
#define SetWindowText SetWindowTextA
#endif
```

A função é documentada sob o nome **SetWindowText** mesmo que esse seja realmente o nome da macro, e não o nome real da função.

Novos aplicativos sempre devem chamar as versões Unicode. Muitos idiomas do mundo exigem Unicode. Se você usar as cadeias de caracteres ANSI, será impossível localizar seu aplicativo. As versões ANSI também são menos eficientes, pois o sistema operacional deve converter as cadeias de caracteres ANSI em Unicode durante o tempo de execução. Dependendo de sua preferência, você pode chamar as funções Unicode explicitamente, como **SetWindowTextW** ou usar as macros. As APIs mais recentes do Windows normalmente têm apenas uma versão Unicode.

TCHARs

Em alguns casos, pode ser útil compilar o mesmo código para cadeias de caracteres ANSI ou Unicode, dependendo da plataforma de destino. Para esse fim, o SDK do Windows fornece macros que mapeiam cadeias de caracteres para Unicode ou ANSI, dependendo da plataforma.

 Expandir a tabela

Macro	Unicode	ANSI
TCHAR	wchar_t	char
TEXT("x") ou _T("x")	L"x"	"x"

Por exemplo, o código a seguir:

C++

```
SetWindowText(TEXT("My Application"));
```

resolve para uma das seguintes opções:

C++

```
SetWindowTextW(L"My Application"); // Unicode function with wide-character string.

SetWindowTextA("My Application"); // ANSI function.
```

As macros **TEXT** e **TCHAR** são menos úteis hoje, porque todos os aplicativos devem usar Unicode.

Os cabeçalhos das bibliotecas de tempo de execução do Microsoft C definem um conjunto semelhante de macros. Por exemplo, **_tcslen** resolve para **strlen** se **_UNICODE** é indefinido, caso contrário, ele resolve para **wcslen** que é a versão de caracteres largos de **strlen**.

C++

```
#ifdef _UNICODE
#define _tcslen    wcslen
#else
#define _tcslen    strlen
#endif
```

Tenha cuidado: alguns cabeçalhos usam o símbolo do pré-processador **UNICODE** e outros usam **_UNICODE** com um prefixo de sublinhado. Sempre defina os dois símbolos. O Visual C++ define ambos por padrão quando você cria um novo projeto.

Próximo

[O que é uma janela?](#)

Comentários

Esta página foi útil?



Yes



No

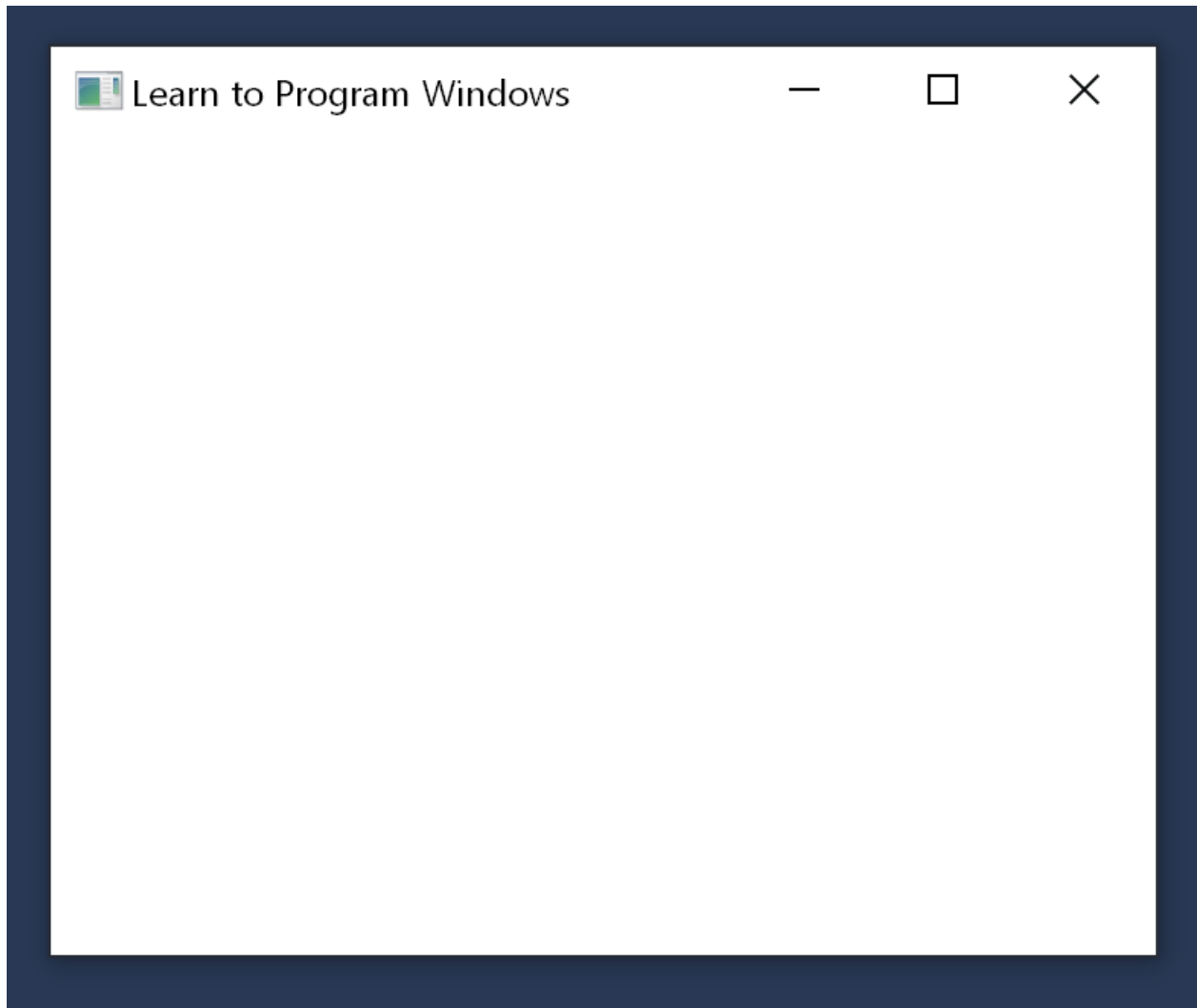
[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

O que é uma janela?

Artigo • 13/06/2023

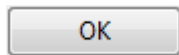
O que é uma janela?

Obviamente, as janelas são centrais para o Windows. Eles são tão importantes que nomearam o sistema operacional em homenagem a eles. Mas o que é uma janela? Quando você pensa em uma janela, você provavelmente pensa em algo assim:



Esse tipo de janela é chamado de *janela de aplicativo* ou *janela main*. Normalmente, ele tem um quadro com uma barra de título, botões **Minimizar** e **Maximizar** e outros elementos de interface do usuário padrão. O quadro é chamado de *área não cliente* da janela, assim chamado porque o sistema operacional gerencia essa parte da janela. A área dentro do quadro é a *área do cliente*. Essa é a parte da janela gerenciada pelo programa.

Aqui está outro tipo de janela:



Se você não estiver familiarizado com a programação do Windows, pode surpreendê-lo de que os controles de interface do usuário, como botões e caixas de edição, são janelas próprias. A principal diferença entre um controle de interface do usuário e uma janela do aplicativo é que um controle não existe por si só. Em vez disso, o controle é posicionado em relação à janela do aplicativo. Quando você arrasta a janela do aplicativo, o controle se move com ela, como você esperaria. Além disso, o controle e a janela do aplicativo podem se comunicar entre si. (Por exemplo, a janela do aplicativo recebe notificações de clique de um botão.)

Portanto, quando você pensar *na janela*, não pense simplesmente *na janela do aplicativo*. Em vez disso, pense em uma janela como uma construção de programação que:

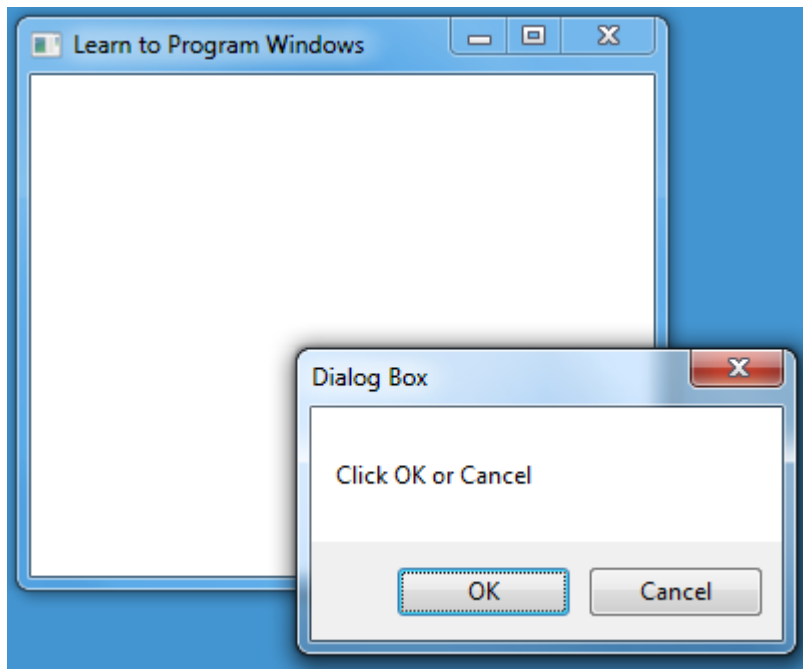
- Ocupa uma determinada parte da tela.
- Pode ou não estar visível em um determinado momento.
- Sabe como desenhar sozinho.
- Responde a eventos do usuário ou do sistema operacional.

Windows pai e Windows proprietário

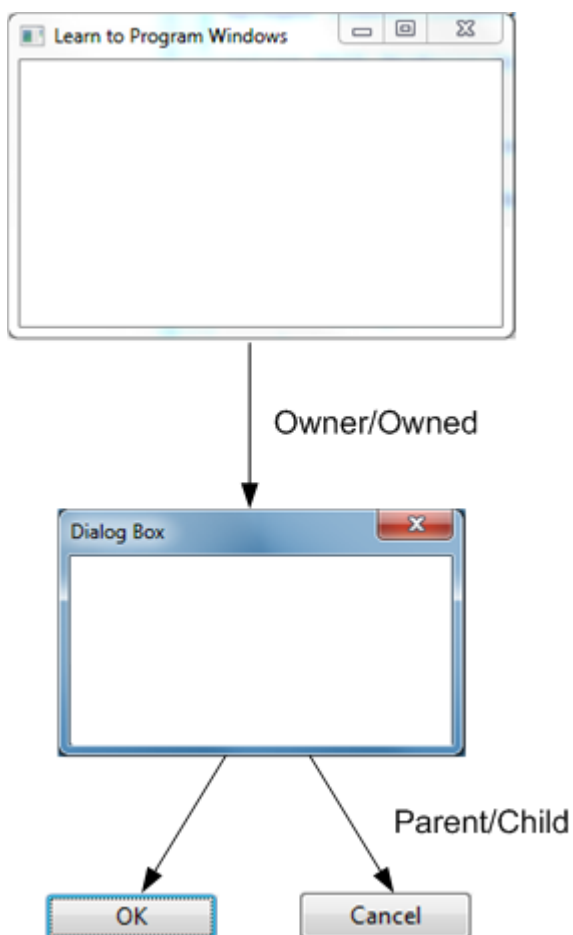
No caso de um controle de interface do usuário, a janela de controle é considerada o *filho* da janela do aplicativo. A janela do aplicativo é o *pai* da janela de controle. A janela pai fornece o sistema de coordenadas usado para posicionar uma janela filho. Ter uma janela pai afeta aspectos da aparência de uma janela; por exemplo, uma janela filho é recortada para que nenhuma parte da janela filho possa aparecer fora das bordas de sua janela pai.

Outra relação é a relação entre uma janela do aplicativo e uma janela de diálogo modal. Quando um aplicativo exibe uma caixa de diálogo modal, a janela do aplicativo é a janela *do proprietário* e a caixa de diálogo é uma janela *de propriedade*. Uma janela de propriedade sempre aparece na frente da janela do proprietário. Ele fica oculto quando o proprietário é minimizado e é destruído ao mesmo tempo que o proprietário.

A imagem a seguir mostra um aplicativo que exibe uma caixa de diálogo com dois botões:



A janela do aplicativo é proprietária da janela de diálogo e a janela de diálogo é o pai de ambas as janelas de botão. O diagrama a seguir mostra estas relações:



Identificadores de janela

O Windows são objetos — eles têm código e dados — mas não são classes C++. Em vez disso, um programa faz referência a uma janela usando um valor chamado *identificador*.

Um identificador é um tipo opaco. Essencialmente, é apenas um número que o sistema operacional usa para identificar um objeto. Você pode imaginar o Windows como tendo uma grande tabela de todas as janelas que foram criadas. Ele usa esta tabela para pesquisar janelas por seus identificadores. (Se é exatamente assim que funciona internamente não é importante.) O tipo de dados para identificadores de janela é **HWND**, que geralmente é pronunciado como "aitch-wind". Os identificadores de janela são retornados pelas funções que criam janelas: [CreateWindow](#) e [CreateWindowEx](#).

Para executar uma operação em uma janela, você normalmente chamará alguma função que usa um valor **HWND** como um parâmetro. Por exemplo, para reposicionar uma janela na tela, chame a função [MoveWindow](#) :

C++

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

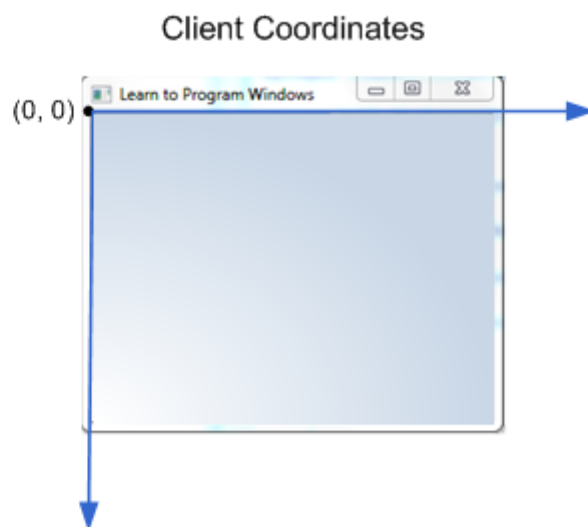
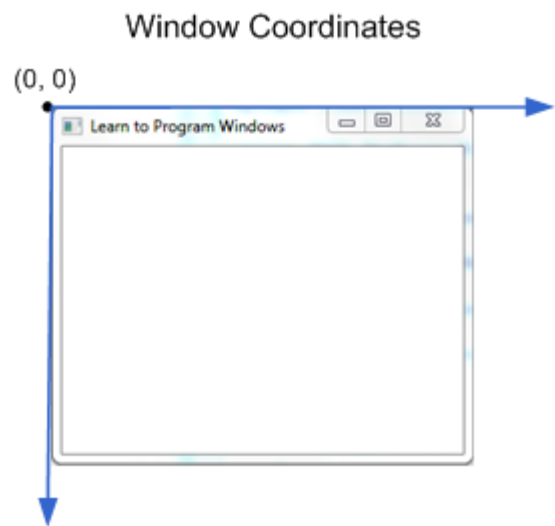
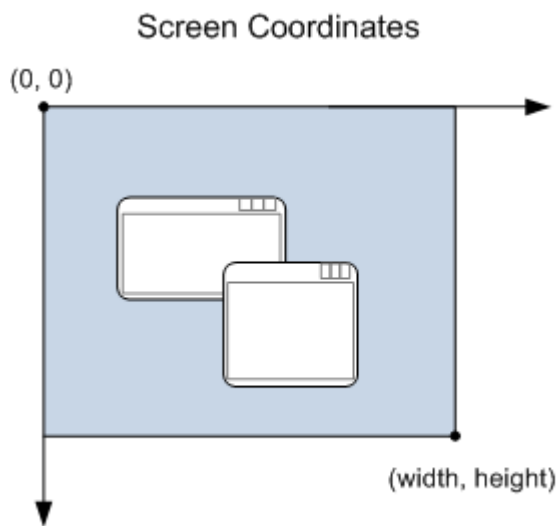
O primeiro parâmetro é o identificador para a janela que você deseja mover. Os outros parâmetros especificam o novo local da janela e se a janela deve ser redesenhada.

Tenha em mente que identificadores não são ponteiros. Se *hwnd* for uma variável que contém um identificador, tentar desreferenciar o identificador escrevendo `*hwnd` será um erro.

Coordenadas de tela e janela

As coordenadas são medidas em pixels independentes do dispositivo. Teremos mais a dizer sobre a parte *independente do dispositivo* de *pixels independentes do dispositivo* quando discutirmos os gráficos.

Dependendo da tarefa, você pode medir coordenadas em relação à tela, em relação a uma janela (incluindo o quadro) ou em relação à área do cliente de uma janela. Por exemplo, você posicionaria uma janela na tela usando coordenadas de tela, mas desenharia dentro de uma janela usando coordenadas do cliente. Em cada caso, a origem (0, 0) é sempre o canto superior esquerdo da região.



Avançar

[WinMain: o ponto de entrada do aplicativo](#)

Comentários

Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

O ponto de entrada do aplicativo

WinMain

Artigo • 11/03/2023

Cada programa do Windows inclui uma função de ponto de entrada chamada **WinMain** ou **wWinMain**. O código a seguir mostra a assinatura para **wWinMain**:

C++

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

Os quatro parâmetros **wWinMain** são os seguintes:

- *hInstance* é o *identificador de uma instância* ou identificador para um módulo. O sistema operacional usa esse valor para identificar o executável ou EXE quando ele é carregado na memória. Determinadas funções do Windows precisam do identificador de instância, por exemplo, para carregar ícones ou bitmaps.
- *hPrevInstance* não tem significado. Ele foi usado no Windows de 16 bits, mas agora é sempre zero.
- *pCmdLine* contém os argumentos de linha de comando como uma cadeia de caracteres Unicode.
- *nCmdShow* é um sinalizador que indica se a janela principal do aplicativo está minimizada, maximizada ou mostrada normalmente.

A função retorna um `int` valor. O sistema operacional não usa o valor retornado, mas você pode usar o valor para passar um código de status para outro programa.

Uma *convenção de chamada*, como `WINAPI`, define como uma função recebe parâmetros do chamador. Por exemplo, a convenção de chamada define a ordem em que os parâmetros aparecem na pilha. Declare sua função **wWinMain** conforme mostrado no exemplo anterior.

A função **WinMain** é a mesma que **wWinMain**, exceto que os argumentos de linha de comando são passados como uma cadeia de caracteres ANSI. A cadeia de caracteres Unicode é preferencial. Você pode usar a função **ANSI WinMain** mesmo se compilar seu programa como Unicode. Para obter uma cópia Unicode dos argumentos de linha de comando, chame a função [GetCommandLine](#). Essa função retorna todos os argumentos em uma única cadeia de caracteres. Se você quiser os argumentos como uma matriz de estilo *argv*, passe essa cadeia de caracteres para [CommandLineToArgvW](#).

Como o compilador sabe invocar **wWinMain** em vez da função **principal** padrão? O que realmente acontece é que a CRT (biblioteca de runtime do Microsoft C) fornece uma implementação de **main** que chama **WinMain** ou **wWinMain**.

O CRT faz mais algum trabalho dentro **do principal**. Por exemplo, ele chama todos os inicializadores estáticos antes **de wWinMain**. Embora você possa instruir o vinculador a usar uma função de ponto de entrada diferente, você deverá usar o padrão se vincular ao CRT. Caso contrário, o código de inicialização crt é ignorado, com resultados imprevisíveis, como objetos globais, não sendo inicializados corretamente.

O código a seguir mostra uma função **WinMain** vazia:

C++

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

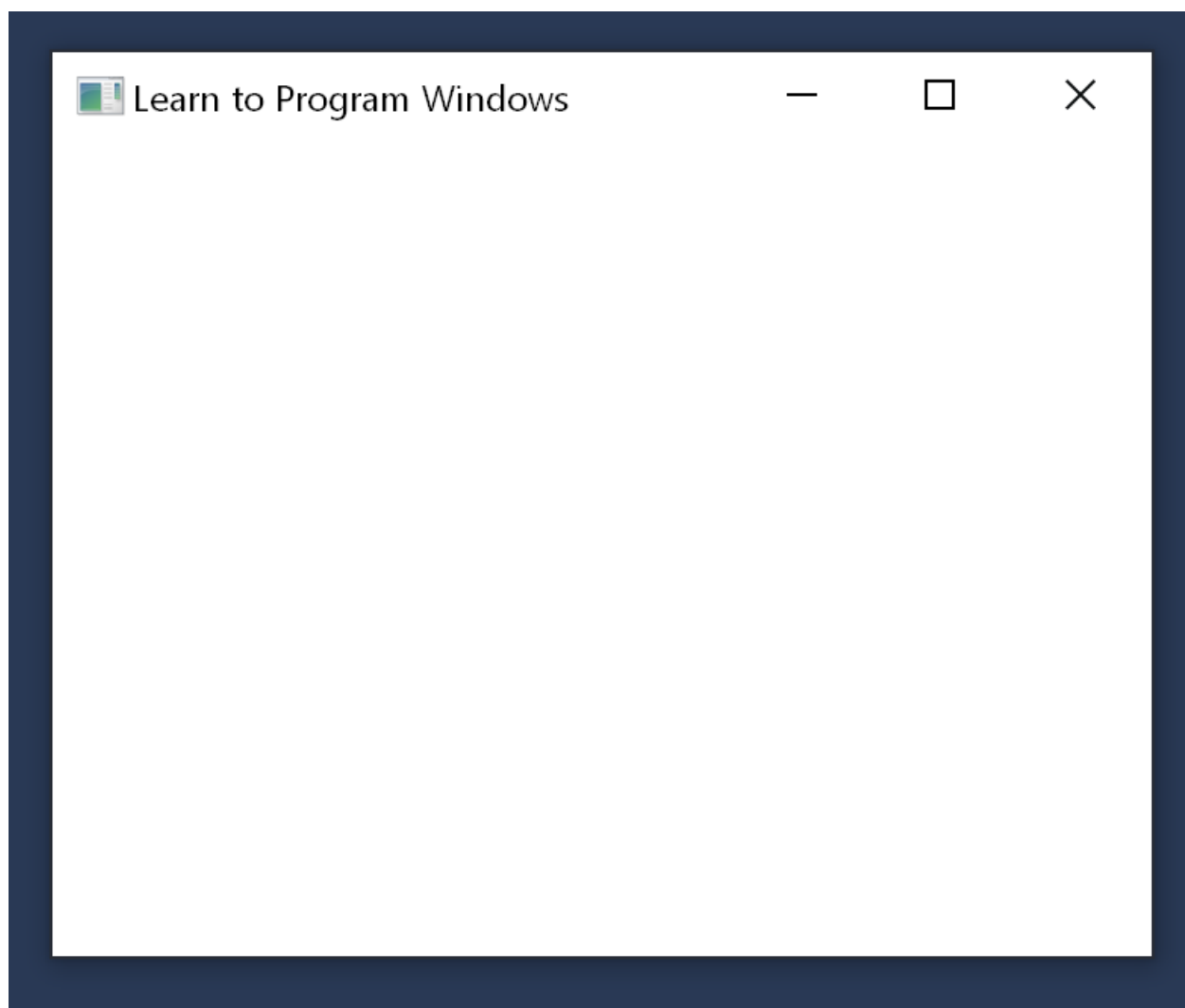
Agora que você tem o ponto de entrada e entende algumas das convenções básicas de terminologia e codificação, está pronto para [Criar seu primeiro programa do Windows](#).

Módulo 1. Seu primeiro programa do Windows

Artigo • 13/06/2023

Neste módulo, escreveremos um programa mínimo da área de trabalho do Windows. Tudo o que ele faz é criar e mostrar uma janela em branco. Este primeiro programa contém cerca de 50 linhas de código, sem contar linhas e comentários em branco. Será nosso ponto de partida; posteriormente, adicionaremos elementos gráficos, texto, entrada do usuário e outros recursos.

Se você estiver procurando mais detalhes sobre como criar um aplicativo de área de trabalho tradicional do Windows no Visual Studio, marcar passo a [passo: criar um aplicativo tradicional da Área de Trabalho do Windows \(C++\)](#).



Este é o código completo para o programa:

```
C++
```

```

#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 hPrevInstance, 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) > 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);

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

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

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

```

Você pode baixar o projeto completo do Visual Studio de [Windows Hello Exemplo Mundial](#).

Pode ser útil dar uma breve estrutura de tópicos sobre o que esse código faz. Tópicos posteriores examinarão o código em detalhes.

1. **wWinMain** é o ponto de entrada do programa. Quando o programa é iniciado, ele registra algumas informações sobre o comportamento da janela do aplicativo. Um dos itens mais importantes é o endereço de uma função, nomeada `WindowProc` neste exemplo. Essa função define o comportamento da janela — sua aparência, como ela interage com o usuário e assim por diante.
2. Em seguida, o programa cria a janela e recebe um identificador que identifica exclusivamente a janela.
3. Se a janela for criada com êxito, o programa inserirá um loop **while**. O programa permanece nesse loop até que o usuário feche a janela e saia do aplicativo.

Observe que o programa não chama explicitamente a `WindowProc` função, embora tenhamos dito que é aqui que a maior parte da lógica do aplicativo é definida. O

Windows se comunica com seu programa passando-lhe uma série de *mensagens*. O código dentro do loop **while** conduz esse processo. Sempre que o programa chama a função [DispatchMessage](#), ele indiretamente faz com que o Windows invoque a função WindowProc, uma vez para cada mensagem.

Nesta seção

- [Criando uma janela](#)
- [Mensagens de janela](#)
- [Gravando o procedimento de janela](#)
- [Pintando a janela](#)
- [Fechando a janela](#)
- [Gerenciando o estado do aplicativo](#)

Tópicos relacionados

[Saiba como Programar para Windows no C++](#)

[Amostra do Windows Hello World](#)

Comentários

Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

Criar uma janela

Artigo • 10/03/2023

Neste artigo, aprenda a criar e mostrar uma janela.

Classes de janela

Uma *classe de janela* define um conjunto de comportamentos que várias janelas podem ter em comum. Por exemplo, em um grupo de botões, cada botão tem um comportamento semelhante quando o usuário seleciona o botão. Claro, os botões não são completamente idênticos. Cada botão exibe sua própria cadeia de caracteres de texto e tem suas próprias coordenadas de tela. Os dados exclusivos para cada janela são chamados *de dados de instância*.

Cada janela deve ser associada a uma classe de janela, mesmo que seu programa crie apenas uma instância dessa classe. Uma classe de janela não é uma classe no sentido C++. Em vez disso, é uma estrutura de dados usada internamente pelo sistema operacional. As classes de janela são registradas com o sistema em tempo de execução. Para registrar uma nova classe de janela, preencha uma estrutura **WNDCLASS**:

C++

```
// 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;
```

Você deve definir os seguintes membros da estrutura:

- **lpfnWndProc** é um ponteiro para uma função definida pelo aplicativo chamada *procedimento de janela* ou *proc de janela*. O procedimento de janela define a maior parte do comportamento da janela. Por enquanto, esse valor é uma declaração de encaminhamento de uma função. Para obter mais informações, consulte [Escrevendo o procedimento Window](#).
- **hInstance** é o identificador para a instância do aplicativo. Obtenha esse valor do parâmetro *hInstance* de `wWinMain`.
- **lpszClassName** é uma cadeia de caracteres que identifica a classe de janela.

Os nomes de classe são locais para o processo atual, portanto, o nome só precisa ser exclusivo dentro do processo. No entanto, os controles padrão do Windows também têm classes. Se você usar qualquer um desses controles, deverá escolher nomes de classe que não entram em conflito com os nomes de classe de controle. Por exemplo, a classe de janela para o controle de botão é chamada *de Botão*.

A estrutura **WNDCLASS** tem outros membros que não são mostrados aqui. Você pode defini-los como zero, conforme mostrado neste exemplo, ou preenchê-los. Para obter mais informações, consulte [WNDCLASS](#).

Em seguida, passe o endereço da estrutura **WNDCLASS** para a função [RegisterClass](#) . Essa função registra a classe de janela com o sistema operacional.

C++

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

Criar a janela

Para criar uma nova instância de uma janela, chame a função [CreateWindowEx](#) :

C++

```
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;  
}
```

Para obter descrições detalhadas do parâmetro, consulte [CreateWindowEx](#). Aqui está um resumo rápido:

- O primeiro parâmetro permite especificar alguns comportamentos opcionais para a janela, por exemplo, janelas transparentes. Defina esse parâmetro como zero para os comportamentos padrão.
- `CLASS_NAME` é o nome da classe de janela. Esse nome define o tipo de janela a ser criada.
- O texto da janela é usado de maneiras diferentes por diferentes tipos de janelas. Se a janela tiver uma barra de título, o texto será exibido na barra de título.
- O estilo de janela é um conjunto de sinalizadores que definem parte da aparência de uma janela. A `WS_OVERLAPPEDWINDOW` constante é, na verdade, vários sinalizadores combinados com um bit a bit `OR`. Juntos, esses sinalizadores dão à janela uma barra de título, uma borda, um menu do sistema e botões **Minimizar** e **Maximizar**. Esse conjunto de sinalizadores é o estilo mais comum para uma janela de aplicativo de nível superior.
- Para posição e tamanho, a constante `CW_USEDEFAULT` significa usar valores padrão.
- O próximo parâmetro define uma janela pai ou uma janela de proprietário para a nova janela. Defina o pai se deseja criar uma janela filho. Para uma janela de nível superior, defina esse valor como `NULL`.
- Para uma janela de aplicativo, o próximo parâmetro define o menu da janela. Este exemplo não usa um menu, portanto, o valor é `NULL`.
- *hInstance* é o identificador de instância, descrito anteriormente. Consulte [WinMain: o ponto de entrada do aplicativo](#).
- O último parâmetro é um ponteiro para dados arbitrários do tipo `void*`. Você pode usar esse valor para passar uma estrutura de dados para o procedimento de janela. Para obter uma maneira possível de usar esse parâmetro, consulte [Gerenciando o estado do aplicativo](#).

CreateWindowEx retorna um identificador para a nova janela ou zero se a função falhar. Para mostrar a janela, ou seja, torne a janela visível, passe o identificador da janela para a função **ShowWindow** :

C++

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

O parâmetro *hwnd* é o identificador de janela retornado por **CreateWindowEx**. O parâmetro *nCmdShow* pode ser usado para minimizar ou maximizar uma janela. O sistema operacional passa esse valor para o programa por meio da função **wWinMain**.

Aqui está o código completo para criar a janela. Lembre-se de que **WindowProc** ainda é apenas uma declaração de encaminhamento de uma função.

C++

```
// 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);
```

Parabéns, você criou uma janela!

No momento, a janela não contém nenhum conteúdo ou interage com o usuário. Em um aplicativo de GUI real, a janela responderia a eventos do usuário e do sistema operacional. A próxima seção descreve como as mensagens de janela fornecem esse tipo de interatividade.

Confira também

Prossiga para [Mensagens de Janela](#) para continuar este módulo.

Mensagens de janela (Introdução ao Win32 e C++)

Artigo • 07/08/2024

Um aplicativo GUI deve responder aos eventos do usuário e do sistema operacional.

- **Os eventos do usuário** incluem todas as maneiras que alguém pode interagir com seu programa: cliques do mouse, pressionamentos de tecla, gestos na tela sensível ao toque e assim por diante.
- **Os eventos do sistema** operacional incluem qualquer coisa "fora" do programa que possa afetar o seu comportamento. Por exemplo, o usuário pode conectar um novo dispositivo de hardware ou o Windows pode entrar em um estado de baixo consumo de energia (suspensão ou hibernação).

Esses eventos podem ocorrer a qualquer momento enquanto o programa estiver em execução, em praticamente qualquer ordem. Como você estrutura um programa cujo fluxo de execução não pode ser previsto com antecedência?

Para resolver esse problema, o Windows usa um modelo de passagem de mensagens. O sistema operacional se comunica com a janela do aplicativo passando mensagens para ele. Uma mensagem é simplesmente um código numérico que designa um evento específico. Por exemplo, se o usuário pressionar o botão esquerdo do mouse, a janela receberá uma mensagem com o seguinte código de mensagem.

C++

```
#define WM_LBUTTONDOWN    0x0201
```

Algumas mensagens têm dados associados a elas. Por exemplo, a mensagem **WM_LBUTTONDOWN** inclui a coordenada x e a coordenada y do cursor do mouse.

Para passar uma mensagem para uma janela, o sistema operacional chama o procedimento de janela registrado para essa janela. (E agora você já sabe para que serve o procedimento de janela.)

O loop de mensagem

Um aplicativo receberá milhares de mensagens enquanto é executado. (Considere que cada pressionamento de tecla e cada clique no botão do mouse gera uma mensagem.) Além disso, um aplicativo pode ter várias janelas, cada uma com seu próprio

procedimento de janela. Como o programa recebe todas essas mensagens e as entrega no procedimento correto de janela? O aplicativo precisa de um loop para recuperar as mensagens e enviá-las para as janelas corretas.

Para cada thread que cria uma janela, o sistema operacional cria uma fila para mensagens de janela. Essa fila contém mensagens para todas as janelas criadas nesse thread. A fila em questão fica oculta em seu programa. Você não pode manipular a fila diretamente. No entanto, você pode efetuar pull de uma mensagem da fila chamando a função [GetMessage](#).

C++

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

Essa função remove a primeira mensagem do topo da fila. Se a fila estiver vazia, a função será bloqueada até que outra mensagem seja enfileirada. O fato de que [GetMessage](#) bloqueia não fará com que seu programa pare de responder. Se não houver mensagens, não há nada para o programa fazer. Se você precisar executar o processamento em segundo plano, poderá criar threads adicionais que continuarão a ser executados enquanto [GetMessage](#) aguarda outra mensagem. (Veja [Como evitar gargalos no procedimento de janela](#)).

O primeiro parâmetro de [GetMessage](#) é o endereço de uma estrutura [MSG](#). Se a função for bem-sucedida, ela preencherá a estrutura do [MSG](#) com informações sobre a mensagem. Isso inclui a janela de destino e o código da mensagem. Os outros três parâmetros permitem filtrar quais mensagens você recebe da fila. Em quase todos os casos, você definirá esses parâmetros como zero.

Apesar de a estrutura [MSG](#) conter informações sobre a mensagem, você raramente examinará essa estrutura diretamente. Em vez disso, você o passará diretamente para duas outras funções.

C++

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

A função [TranslateMessage](#) está relacionada à entrada do teclado. Ele traduz as teclas digitadas (tecla para baixo, tecla para cima) em caracteres. Você realmente não precisa saber como essa função funciona, basta lembrar de chamá-la antes de [DispatchMessage](#).

A função **DispatchMessage** informa ao sistema operacional para chamar o procedimento de janela, da janela que é o alvo da mensagem. Em outras palavras, o sistema operacional procura o identificador de janela em sua tabela de janelas, localiza o ponteiro de função associado à janela e invoca a função.

Por exemplo, suponha que o usuário pressione o botão esquerdo do mouse. Isso causa uma cadeia de eventos:

1. O sistema operacional coloca uma mensagem **WM_LBUTTONDOWN** na fila de mensagens.
2. Seu programa chama a função **GetMessage**.
3. **GetMessage** extrai a mensagem **WM_LBUTTONDOWN** da fila e preenche a estrutura **MSG**.
4. Seu programa chama as funções **TranslateMessage** e **DispatchMessage**.
5. Dentro de **DispatchMessage** o sistema operacional chama o procedimento de janela.
6. Seu procedimento de janela pode responder à mensagem ou ignorá-la.

Quando o procedimento de janela retorna, ele retorna para **DispatchMessage**. Isso retorna ao loop de mensagem para a próxima mensagem. Enquanto o programa estiver em execução, as mensagens continuarão a chegar na fila. Portanto, você deve ter um loop que extraia continuamente as mensagens da fila e as despache. Você pode pensar no loop como se estivesse fazendo o seguinte:

C++

```
// WARNING: Don't actually write your loop this way.

while (1)
{
    GetMessage(&msg, NULL, 0, 0);
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

Como está escrito, é lógico, esse loop nunca terminaria. É aí que entra o valor de devolução para a função **GetMessage** entrar. Normalmente, **GetMessage** retorna a um valor diferente de zero. Quando você quiser sair do aplicativo e sair do loop de mensagem, chame a função **PostQuitMessage**.

C++

```
PostQuitMessage(0);
```

A função [PostQuitMessage](#) coloca a mensagem [WM_QUIT](#) na fila de mensagem. [WM_QUIT](#) é uma mensagem especial e faz com que [GetMessage](#) retorne a zero, sinalizando o fim do loop de mensagem. Aqui está o loop de mensagem revisado.

C++

```
// Correct.

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

Desde que [GetMessage](#) retorne a um valor diferente de zero, a expressão no **loop while** é avaliada como verdadeira. Depois de chamar [PostQuitMessage](#) a expressão se torna falsa e o programa sai do loop. (Um resultado interessante desse comportamento é que seu procedimento de janela nunca recebe uma mensagem [WM_QUIT](#) portanto, você não precisa ter uma instrução case para essa mensagem no procedimento de janela.)

A próxima pergunta óbvia é quando chamar [PostQuitMessage](#). Voltaremos a essa questão no tópico [fechar a janela](#), mas primeiro temos que escrever nosso procedimento de janela.

Mensagens postadas versus mensagens enviadas

A seção anterior falou sobre as mensagens indo para uma fila. Às vezes, o sistema operacional chamará um procedimento de janela diretamente, ignorando a fila.

A terminologia para essa distinção pode ser confusa:

- *Postar* uma mensagem significa que a mensagem vai para a fila de mensagens e é despachada por meio do loop de mensagens ([GetMessage](#) e [DispatchMessage](#)).
- *Enviar* uma mensagem significa que a mensagem ignora a fila e o sistema operacional chama o procedimento de janela diretamente.

Por enquanto, a diferença não é muito importante. O procedimento de janela lida com todas as mensagens. Entretanto, algumas mensagens ignoram a fila e vão diretamente para o procedimento de janela. Entretanto, pode fazer diferença se o aplicativo se comunicar entre as janelas. Você pode encontrar uma discussão mais completa sobre esse problema no tópico [sobre mensagens e filas de mensagens](#).

Próximo

[Como gravar o procedimento de janela](#)

Comentários

Esta página foi útil?

 Yes

 No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Como gravar o procedimento de janela

Artigo • 07/08/2024

A função [DispatchMessage](#) chama o procedimento de janela da janela que é o destino da mensagem. O procedimento de janela tem a assinatura a seguir.

C++

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

Existem quatro parâmetros:

- *hwnd* é um identificador para a janela.
- *uMsg* é o código da mensagem, por exemplo, a mensagem [WM_SIZE](#) indica que a janela foi redimensionada.
- *wParam* e *lParam* contêm dados adicionais que pertencem à mensagem. O significado exato depende do código da mensagem.

LRESULT é um valor inteiro que seu programa retorna ao Windows. Ele contém a resposta do programa para uma mensagem específica. O significado desse valor depende do código da mensagem. **CALLBACK** é a convenção de chamada para a função.

Um procedimento de janela comum é simplesmente uma instrução switch grande que é alternado no código da mensagem. Adicionar ocorrências para cada mensagem que você deseja identificar.

C++

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

        // etc
}
```

Dados adicionais para a mensagem estão presentes nos parâmetros *lParam* e *wParam*. Os dois parâmetros são valores inteiros do tamanho de uma largura de ponteiro (32 bits ou 64 bits). O significado de cada depende do código de mensagem (*uMsg*). Para cada mensagem, você precisará procurar o código da mensagem e converter os parâmetros para o tipo de dados correto. Normalmente, os dados são um valor numérico ou um ponteiro para uma estrutura. Algumas mensagens não têm dados.

Por exemplo, a documentação da mensagem [WM_SIZE](#) indica que:

- *wParam* é um sinalizador que indica se a janela foi minimizada, maximizada ou redimensionada.
- *lParam* contém a nova largura e altura da janela como valores de 16 bits compactados em um número de 32 ou 64 bits. Você precisará realizar algumas mudanças de bits para obter esses valores. Felizmente, o arquivo de cabeçalho WinDef.h inclui macros auxiliares que fazem isso.

Um procedimento de janela comum lida com dezenas de mensagens, para que ele possa crescer bastante. Uma maneira de tornar seu código mais modular é colocando a lógica para tratar cada mensagem em uma função separada. No procedimento de janela, converta os parâmetros *wParam* e *lParam* para o tipo de dados correto e passe esses valores para a função. Por exemplo, para lidar com a mensagem [WM_SIZE](#), o procedimento de janela seria assim:

C++

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

            // Respond to the message:
            OnSize(hwnd, (UINT)wParam, width, height);
        }
        break;
    }
}

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

As macros [LOWORD](#) e [HIWORD](#) obtêm os valores de largura e altura de 16 bits de *lParam*. O procedimento de janela extrai a largura e a altura e, em seguida, passa esses valores para a `OnSize` função.

Manipulação de mensagens padrão

Se você não manipular uma mensagem específica em seu procedimento de janela, passe os parâmetros de mensagem diretamente para a função [DefWindowProc](#). Essa função executa a ação padrão para a mensagem, que varia de acordo com o tipo de mensagem.

C++

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

Evitando gargalos no procedimento de janela

Enquanto o procedimento de janela é executado, ele bloqueia todas as outras mensagens para janelas criadas no mesmo thread. Portanto, evite o processamento demorado dentro do procedimento de janela. Por exemplo, imagine que seu programa abriu uma conexão TCP e aguarda indefinidamente para que o servidor responda. Se você fizer isso dentro do procedimento de janela, sua interface do usuário não responderá até que a solicitação seja concluída. Durante esse tempo, a janela não pode processar a entrada do mouse ou teclado, repintar a si mesma ou até mesmo fechar.

Em vez disso, você deve mover o trabalho para outro thread, usando uma das instalações multitarefa que são incorporadas ao Windows:

- Criar uma thread nova.
- Use um pool de thread.
- Use chamadas de E/S assíncronas.
- Use chamadas de procedimento assíncronas.

Próximo

[Como pintar a janela](#)

Comentários

Esta página foi útil?

 Yes

 No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

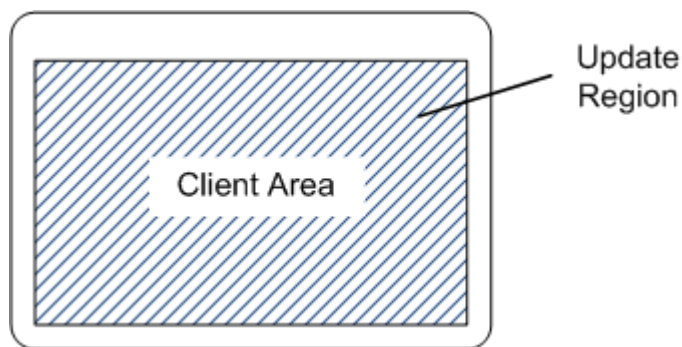
Pintando a janela

Artigo • 13/06/2023

Você criou sua janela. Agora você quer mostrar algo dentro dele. Na terminologia do Windows, isso é chamado de pintura da janela. Para misturar metáforas, uma janela é uma tela em branco, esperando você preenchê-la.

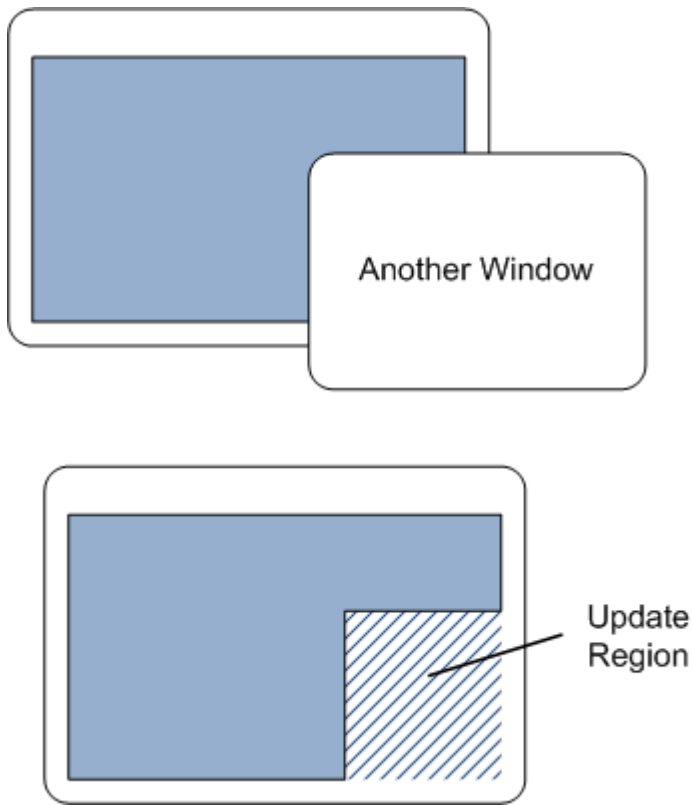
Às vezes, seu programa iniciará a pintura para atualizar a aparência da janela. Em outros momentos, o sistema operacional notificará você de que você deve repintar uma parte da janela. Quando isso ocorre, o sistema operacional envia à janela uma mensagem **WM_PAINT**. A parte da janela que deve ser pintada é chamada *de região de atualização*.

Na primeira vez que uma janela é mostrada, toda a área do cliente da janela deve ser pintada. Portanto, você sempre receberá pelo menos uma **mensagem WM_PAINT** ao mostrar uma janela.

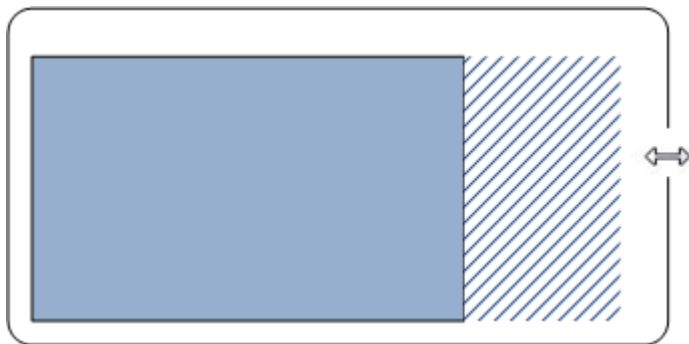


Você só é responsável por pintar a área do cliente. O quadro ao redor, incluindo a barra de título, é pintado automaticamente pelo sistema operacional. Depois de terminar de pintar a área do cliente, desmarque a região de atualização, que informa ao sistema operacional que ele não precisa enviar outro **WM_PAINT** mensagem até que algo seja alterado.

Agora, suponha que o usuário mova outra janela para que ela obscureça uma parte da janela. Quando a parte obscurecida fica visível novamente, essa parte é adicionada à região de atualização e sua janela recebe outra **mensagem WM_PAINT**.



A região de atualização também será alterada se o usuário esticar a janela. No diagrama a seguir, o usuário estende a janela para a direita. A área recém-exposta no lado direito da janela é adicionada à região de atualização:



Em nosso primeiro programa de exemplo, a rotina de pintura é muito simples. Ele apenas preenche toda a área do cliente com uma cor sólida. Ainda assim, este exemplo é suficiente para demonstrar alguns dos conceitos importantes.

C++

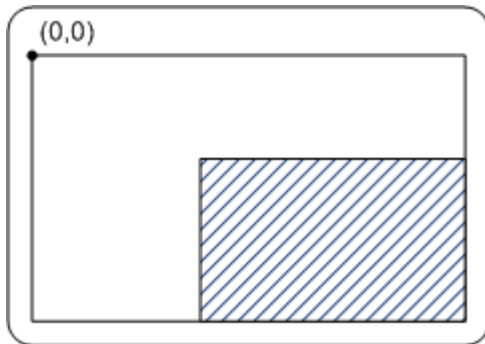
```
switch (uMsg)
{
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);

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

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

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

Inicie a operação de pintura chamando a função **BeginPaint**. Essa função preenche a estrutura **PAINTSTRUCT** com informações sobre a solicitação de repintar. A região de atualização atual é fornecida no membro **rcPaint** do **PAINTSTRUCT**. Essa região de atualização é definida em relação à área do cliente:



No código de pintura, você tem duas opções básicas:

- Pinte toda a área do cliente, independentemente do tamanho da região de atualização. Tudo o que estiver fora da região de atualização é recortado. Ou seja, o sistema operacional o ignora.
- Otimize pintando apenas a parte da janela dentro da região de atualização.

Se você sempre pintar toda a área do cliente, o código será mais simples. No entanto, se você tiver uma lógica de pintura complicada, poderá ser mais eficiente ignorar as áreas fora da região de atualização.

A linha de código a seguir preenche a região de atualização com uma única cor, usando a cor da tela de fundo da janela definida pelo sistema (**COLOR_WINDOW**). A cor real indicada por **COLOR_WINDOW** depende do esquema de cores atual do usuário.

C++

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

Os detalhes do **FillRect** não são importantes para este exemplo, mas o segundo parâmetro fornece as coordenadas do retângulo a ser preenchido. Nesse caso, passamos toda a região de atualização (o membro **rcPaint** do **PAINTSTRUCT**). Na primeira **WM_PAINT** mensagem, toda a área do cliente precisa ser pintada, portanto, **rcPaint** conterá toda a área do cliente. Nas mensagens **WM_PAINT** subsequentes, **rcPaint** pode conter um retângulo menor.

A função [FillRect](#) faz parte da GDI (Graphics Device Interface), que alimenta os elementos gráficos do Windows há muito tempo. No Windows 7, a Microsoft introduziu um novo mecanismo gráfico, chamado Direct2D, que dá suporte a operações gráficas de alto desempenho, como aceleração de hardware. Direct2D também está disponível para o Windows Vista por meio do [Platform Update para Windows Vista](#) e do Windows Server 2008 por meio da Atualização de Plataforma para Windows Server 2008. (A GDI ainda tem suporte total.)

Depois de terminar de pintar, chame a função [EndPaint](#) . Essa função limpa a região de atualização, que sinaliza para o Windows que a janela concluiu a pintura em si.

Avançar

[Fechando a janela](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Fechando a janela

Artigo • 13/06/2023

Quando o usuário fecha uma janela, essa ação dispara uma sequência de mensagens de janela.

O usuário pode fechar uma janela do aplicativo clicando no botão **Fechar** ou usando um atalho de teclado, como ALT+F4. Qualquer uma dessas ações faz com que a janela receba uma mensagem [WM_CLOSE](#). A mensagem **WM_CLOSE** oferece a oportunidade de solicitar ao usuário antes de fechar a janela. Se você realmente quiser fechar a janela, chame a função [DestroyWindow](#). Caso contrário, basta retornar zero da mensagem **WM_CLOSE** e o sistema operacional ignorará a mensagem e não destruirá a janela.

Aqui está um exemplo de como um programa pode lidar com [WM_CLOSE](#).

C++

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

Neste exemplo, a função [MessageBox](#) mostra uma caixa de diálogo modal que contém os botões **OK** e **Cancelar**. Se o usuário clicar em **OK**, o programa [chamará DestroyWindow](#). Caso contrário, se o usuário clicar em **Cancelar**, a chamada para **DestroyWindow** será ignorada e a janela permanecerá aberta. Em ambos os casos, retorne zero para indicar que você lidou com a mensagem.

Se você quiser fechar a janela sem avisar o usuário, basta chamar [DestroyWindow](#) sem a chamada para [MessageBox](#). No entanto, há um atalho nesse caso. Lembre-se [de que DefWindowProc](#) executa a ação padrão para qualquer mensagem de janela. No caso de [WM_CLOSE](#), **DefWindowProc** chama automaticamente **DestroyWindow**. Isso significa que, se você ignorar a mensagem **WM_CLOSE** na instrução **switch**, a janela será destruída por padrão.

Quando uma janela está prestes a ser destruída, ela recebe uma [mensagem WM_DESTROY](#). Essa mensagem é enviada depois que a janela é removida da tela, mas antes que a destruição ocorra (em particular, antes que qualquer janela filho seja destruída).

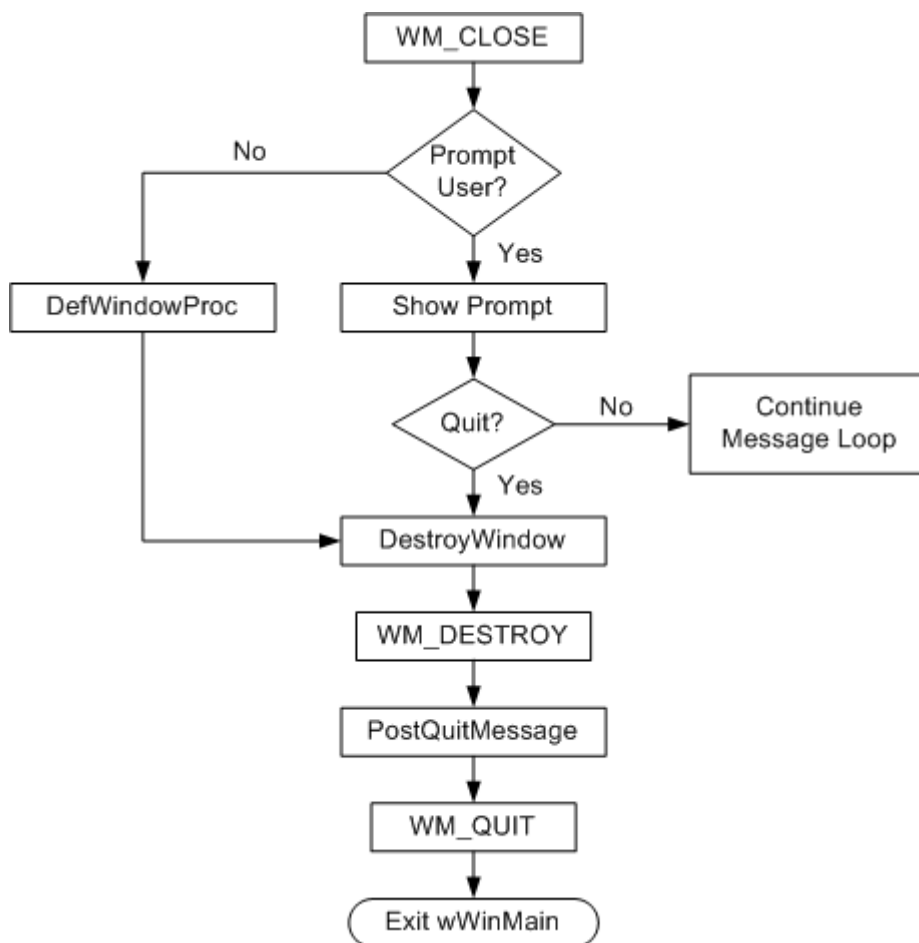
Na janela do aplicativo main, você normalmente responderá a **WM_DESTROY** chamando **PostQuitMessage**.

C++

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

Vimos na seção **Mensagens de Janela** que **PostQuitMessage** coloca uma mensagem **WM_QUIT** na fila de mensagens, fazendo com que o loop de mensagem termine.

Aqui está um fluxograma mostrando a maneira típica de processar **mensagens WM_CLOSE** e **WM_DESTROY** :



Avançar

[Gerenciando o estado do aplicativo](#)

Comentários

Esta página foi útil?



Yes



No

[Obter ajuda no Microsoft Q&A](#)

Gerenciando o estado do aplicativo

Artigo • 13/06/2023

Um procedimento de janela é apenas uma função que é invocada para cada mensagem, portanto, é inerentemente sem estado. Portanto, você precisa de uma maneira de acompanhar o estado do aplicativo de uma chamada de função para a próxima.

A abordagem mais simples é simplesmente colocar tudo em variáveis globais. Isso funciona bem o suficiente para programas pequenos, e muitos dos exemplos de SDK usam essa abordagem. Em um programa grande, no entanto, ele leva a uma proliferação de variáveis globais. Além disso, você pode ter várias janelas, cada uma com seu próprio procedimento de janela. Manter o controle de qual janela deve acessar quais variáveis se tornam confusas e propensas a erros.

A função [CreateWindowEx](#) fornece uma maneira de passar qualquer estrutura de dados para uma janela. Quando essa função é chamada, ela envia as duas seguintes mensagens para o procedimento de janela:

- [WM_NCCREATE](#)
- [WM_CREATE](#)

Essas mensagens são enviadas na ordem listada. (Essas não são as duas únicas mensagens enviadas durante [CreateWindowEx](#), mas podemos ignorar as outras para essa discussão.)

A [mensagem WM_NCCREATE](#) e [WM_CREATE](#) são enviadas antes que a janela fique visível. Isso os torna um bom lugar para inicializar sua interface do usuário, por exemplo, para determinar o layout inicial da janela.

O último parâmetro de [CreateWindowEx](#) é um ponteiro do tipo `void*`. Você pode passar qualquer valor de ponteiro desejado nesse parâmetro. Quando o procedimento de janela manipula o [WM_NCCREATE](#) ou [WM_CREATE](#) mensagem, ele pode extrair esse valor dos dados da mensagem.

Vamos ver como você usaria esse parâmetro para passar dados do aplicativo para sua janela. Primeiro, defina uma classe ou estrutura que contenha informações de estado.

C++

```
// Define a structure to hold some state information.  
  
struct StateInfo {  
    // ... (struct members not shown)  
};
```

Ao chamar [CreateWindowEx](#), passe um ponteiro para essa estrutura no parâmetro final `void*`.

C++

```
StateInfo *pState = new (std::nothrow) StateInfo;

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

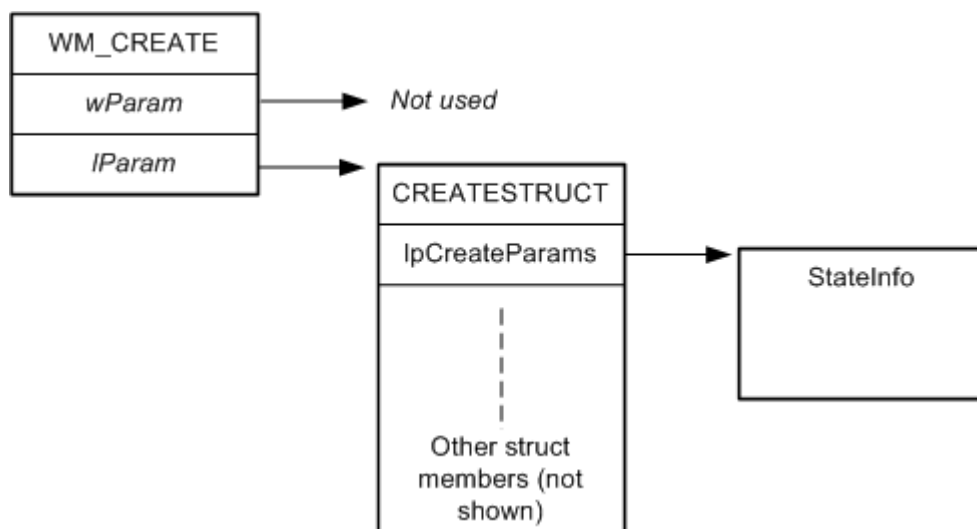
// Initialize the structure members (not shown).

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
    pState                 // Additional application data
);
```

Quando você recebe as mensagens [WM_NCCREATE](#) e [WM_CREATE](#), o parâmetro `lParam` de cada mensagem é um ponteiro para uma estrutura [CREATESTRUCT](#). A estrutura [CREATESTRUCT](#), por sua vez, contém o ponteiro que você passou para [CreateWindowEx](#).



Veja como extrair o ponteiro para sua estrutura de dados. Primeiro, obtenha a estrutura **CREATESTRUCT** convertendo o parâmetro *lParam*.

C++

```
CREATESTRUCT *pCreate = reinterpret_cast<CREATESTRUCT*>(lParam);
```

O membro **lpCreateParams** da estrutura **CREATESTRUCT** é o ponteiro nulo original especificado em **CreateWindowEx**. Obtenha um ponteiro para sua própria estrutura de dados convertendo **lpCreateParams**.

C++

```
pState = reinterpret_cast<StateInfo*>(pCreate->lpCreateParams);
```

Em seguida, chame a função **SetWindowLongPtr** e passe o ponteiro para sua estrutura de dados.

C++

```
SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pState);
```

A finalidade dessa última chamada de função é armazenar o ponteiro *StateInfo* nos dados da instância da janela. Depois de fazer isso, você sempre poderá obter o ponteiro de volta da janela chamando **GetWindowLongPtr**:

C++

```
LONG_PTR ptr = GetWindowLongPtr(hwnd, GWLP_USERDATA);  
StateInfo *pState = reinterpret_cast<StateInfo*>(ptr);
```

Cada janela tem seus próprios dados de instância, para que você possa criar várias janelas e dar a cada janela sua própria instância da estrutura de dados. Essa abordagem é especialmente útil se você definir uma classe de janelas e criar mais de uma janela dessa classe, por exemplo, se você criar uma classe de controle personalizada. É conveniente encapsular a chamada **GetWindowLongPtr** em uma função auxiliar pequena.

C++

```
inline StateInfo* GetAppState(HWND hwnd)  
{  
    LONG_PTR ptr = GetWindowLongPtr(hwnd, GWLP_USERDATA);  
    StateInfo *pState = reinterpret_cast<StateInfo*>(ptr);  
}
```

```
    return pState;
}
```

Agora você pode escrever o procedimento de janela da seguinte maneira.

C++

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    StateInfo *pState;
    if (uMsg == WM_CREATE)
    {
        CREATESTRUCT *pCreate = reinterpret_cast<CREATESTRUCT*>(lParam);
        pState = reinterpret_cast<StateInfo*>(pCreate->lpCreateParams);
        SetWindowLongPtr(hwnd, GWLP_USERDATA, (LONG_PTR)pState);
    }
    else
    {
        pState = GetAppState(hwnd);
    }

    switch (uMsg)
    {

        // Remainder of the window procedure not shown ...

    }
    return TRUE;
}
```

Uma abordagem Object-Oriented

Podemos estender ainda mais essa abordagem. Já definimos uma estrutura de dados para armazenar informações de estado sobre a janela. Faz sentido fornecer essa estrutura de dados com funções membro (métodos) que operam nos dados. Isso naturalmente leva a um design em que a estrutura (ou classe) é responsável por todas as operações na janela. Em seguida, o procedimento de janela se tornaria parte da classe .

Em outras palavras, gostaríamos de ir a partir disso:

C++

```
// 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.
    }
}

```

Para isso:

C++

```

// pseudocode

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

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

```

O único problema é como conectar o `MyWindow::WindowProc` método . A função [RegisterClass](#) espera que o procedimento de janela seja um ponteiro de função. Você não pode passar um ponteiro para uma função membro (não estática) neste contexto. No entanto, você pode passar um ponteiro para uma função membro *estática* e, em seguida, delegar para a função membro. Aqui está um modelo de classe que mostra essa abordagem:

C++


```

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;
};

```

A `BaseWindow` classe é uma classe base abstrata, da qual as classes de janela específicas são derivadas. Por exemplo, aqui está a declaração de uma classe simples derivada de `BaseWindow`:

C++

```

class MainWindow : public BaseWindow<MainWindow>
{
public:
    PCWSTR  ClassName() const { return L"Sample Window Class"; }
    LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);
};

```

Para criar a janela, chame `BaseWindow::Create`:

C++

```

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;
}

```

O método pure-virtual `BaseWindow::HandleMessage` é usado para implementar o procedimento de janela. Por exemplo, a implementação a seguir é equivalente ao procedimento de janela mostrado no início do [Módulo 1](#).

C++

```

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;
}

```

Observe que o identificador de janela é armazenado em uma variável de membro (`m_hwnd`), portanto, não precisamos passá-lo como um parâmetro para `HandleMessage`.

Muitas das estruturas de programação existentes do Windows, como o Microsoft Foundation Classes (MFC) e a ATL (Biblioteca de Modelos Ativos), usam abordagens que são basicamente semelhantes à mostrada aqui. É claro que uma estrutura totalmente generalizada, como o MFC, é mais complexa do que este exemplo relativamente simplista.

Avançar

Tópicos relacionados

[Exemplo de BaseWindow](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Módulo 2. Usando COM em seu programa de Windows-Based

Artigo • 13/06/2023

O [módulo 1](#) desta série mostrou como criar uma janela e responder a mensagens de janela, como [WM_PAINT](#) e [WM_CLOSE](#). O módulo 2 apresenta o COM (Component Object Model).

COM é uma especificação para criar componentes de software reutilizáveis. Muitos dos recursos que você usará em um programa moderno baseado no Windows dependem do COM, como o seguinte:

- Gráficos (Direct2D)
- Texto (DirectWrite)
- O Shell do Windows
- O controle Faixa de Opções
- Animação da interface do usuário

(Algumas tecnologias nesta lista usam um subconjunto de COM e, portanto, não são "puras" COM.)

COM tem a reputação de ser difícil de aprender. E é verdade que escrever um novo módulo de software para dar suporte ao COM pode ser complicado. No entanto, se o programa for estritamente um *consumidor* de COM, você poderá achar que o COM é mais fácil de entender do que o esperado.

Este módulo mostra como chamar APIs baseadas em COM em seu programa. Ele também descreve alguns dos raciocínios por trás do design do COM. Se você entender por que o COM foi projetado como ele é, você pode programar com ele com mais eficiência. A segunda parte do módulo descreve algumas práticas de programação recomendadas para COM.

O COM foi introduzido em 1993 para dar suporte ao OLE (Object Linking and Embedding) 2.0. Pessoas às vezes acha que COM e OLE são a mesma coisa. Esse pode ser outro motivo para a percepção de que COM é difícil de aprender. O OLE 2.0 é criado com base no COM, mas você não precisa saber o OLE para entender o COM.

COM é um *padrão binário*, não um padrão de linguagem: define a interface binária entre um aplicativo e um componente de software. Como um padrão binário, o COM é neutro em linguagem, embora mapeia naturalmente para determinados constructos C++. Este módulo se concentrará em três objetivos principais do COM:

- Separando a implementação de um objeto de sua interface.
- Gerenciando o tempo de vida de um objeto.
- Descobrir os recursos de um objeto em tempo de execução.

Nesta seção

- [O que é uma interface COM?](#)
- [Inicializando a biblioteca COM](#)
- [Códigos de erro no COM](#)
- [Criando um objeto em COM](#)
- [Exemplo: a caixa de diálogo Abrir](#)
- [Gerenciando o tempo de vida de um objeto](#)
- [Solicitando um objeto para uma interface](#)
- [Alocação de memória em COM](#)
- [Práticas de codificação COM](#)
- [Tratamento de erros no COM](#)

Tópicos relacionados

[Saiba como Programar para Windows no C++](#)

Comentários

Esta página foi útil?

 Yes

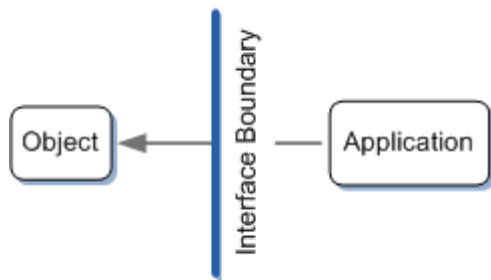
 No

[Obter ajuda no Microsoft Q&A](#)

O que é uma interface COM?

Artigo • 13/06/2023

Se você conhece C# ou Java, as interfaces devem ser um conceito familiar. Uma *interface* define um conjunto de métodos que um objeto pode dar suporte, sem ditar nada sobre a implementação. A interface marca um limite claro entre o código que chama um método e o código que implementa o método. Em termos de ciência da computação, o chamador é *dissociado* da implementação.



No C++, o equivalente mais próximo de uma interface é uma classe virtual pura, ou seja, uma classe que contém apenas métodos virtuais puros e nenhum outro membro. Aqui está um exemplo hipotético de uma interface:

C++

```
// The following is not actual COM.  
  
// Pseudo-C++:  
  
interface IDrawable  
{  
    void Draw();  
};
```

A ideia deste exemplo é que um conjunto de objetos em algumas bibliotecas gráficas seja desenhado. A `IDrawable` interface define as operações que qualquer objeto desenhável deve dar suporte. (Por convenção, os nomes de interface começam com "I".) Neste exemplo, a `IDrawable` interface define uma única operação: `Draw`.

Todas as interfaces são *abstratas*, portanto, um programa não pôde criar uma instância de um `IDrawable` objeto como tal. Por exemplo, o código a seguir não seria compilado.

C++

```
IDrawable draw;  
draw.Draw();
```

Em vez disso, a biblioteca de gráficos fornece objetos que *implementam* a `IDrawable` interface . Por exemplo, a biblioteca pode fornecer um objeto de forma para formas de desenho e um objeto bitmap para desenhar imagens. No C++, isso é feito herdando de uma classe base abstrata comum:

C++

```
class Shape : public IDrawable
{
public:
    virtual void Draw();    // Override Draw and provide implementation.
};

class Bitmap : public IDrawable
{
public:
    virtual void Draw();    // Override Draw and provide implementation.
};
```

As `Shape` classes e `Bitmap` definem dois tipos distintos de objeto desenhável. Cada classe herda de `IDrawable` e fornece sua própria implementação do `Draw` método . Naturalmente, as duas implementações podem ser consideravelmente diferentes. Por exemplo, o `Shape::Draw` método pode rasterizar um conjunto de linhas, enquanto `Bitmap::Draw` blit uma matriz de pixels.

Um programa que usa essa biblioteca gráfica manipularia `Shape` objetos e `Bitmap` por meio `IDrawable` de ponteiros, em vez de usar `Shape` ponteiros ou `Bitmap` diretamente.

C++

```
IDrawable *pDrawable = CreateTriangleShape();

if (pDrawable)
{
    pDrawable->Draw();
}
```

Aqui está um exemplo que faz loop sobre uma matriz de `IDrawable` ponteiros. A matriz pode conter uma variedade heterogênea de formas, bitmaps e outros objetos gráficos, desde que cada objeto na matriz herda `IDrawable`.

C++

```
void DrawSomeShapes(IDrawable **drawableArray, size_t count)
{
```



```
for (size_t i = 0; i < count; i++)
{
    drawableArray[i]->Draw();
}
```

Um ponto importante sobre COM é que o código de chamada nunca vê o tipo da classe derivada. Em outras palavras, você nunca declararia uma variável do tipo `Shape` ou `Bitmap` em seu código. Todas as operações em formas e bitmaps são executadas usando `IDrawable` ponteiros. Dessa forma, o COM mantém uma separação estrita entre interface e implementação. Os detalhes de implementação das `Shape` classes e `Bitmap` podem ser alterados, por exemplo, para corrigir bugs ou adicionar novos recursos, sem alterações no código de chamada.

Em uma implementação C++, as interfaces são declaradas usando uma classe ou estrutura.

❗ Observação

Os exemplos de código neste tópico destinam-se a transmitir conceitos gerais, não prática do mundo real. A definição de novas interfaces COM está além do escopo desta série, mas você não definiria uma interface diretamente em um arquivo de cabeçalho. Em vez disso, uma interface COM é definida usando uma linguagem chamada IDL (Interface Definition Language). O arquivo IDL é processado por um compilador de IDL, que gera um arquivo de cabeçalho C++.

C++

```
class IDrawable
{
public:
    virtual void Draw() = 0;
};
```

Quando você trabalha com COM, é importante lembrar que as interfaces não são objetos. São coleções de métodos que os objetos devem implementar. Vários objetos podem implementar a mesma interface, conforme mostrado com os `Shape` exemplos e `Bitmap`. Além disso, um objeto pode implementar várias interfaces. Por exemplo, a biblioteca de gráficos pode definir uma interface chamada `ISerializable` que dá suporte a salvar e carregar objetos gráficos. Agora considere as seguintes declarações de classe:

C++

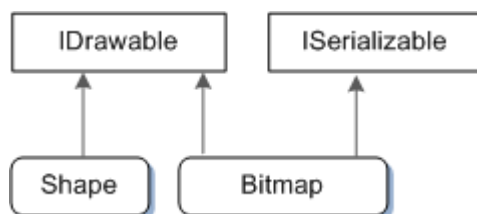
```
// An interface for serialization.
class ISerializable
{
public:
    virtual void Load(PCWSTR filename) = 0;    // Load from file.
    virtual void Save(PCWSTR filename) = 0;    // Save to file.
};

// Declarations of drawable object types.

class Shape : public IDrawable
{
    ...
};

class Bitmap : public IDrawable, public ISerializable
{
    ...
};
```

Neste exemplo, a `Bitmap` classe implementa `ISerializable`. O programa pode usar esse método para salvar ou carregar o bitmap. No entanto, a `Shape` classe não implementa `ISerializable`, portanto, ela não expõe essa funcionalidade. O diagrama a seguir mostra as relações de herança neste exemplo.



Esta seção examinou a base conceitual das interfaces, mas até agora não vimos o código COM real. Começaremos com a primeira coisa que qualquer aplicativo COM deve fazer: Inicializar a biblioteca COM.

Avançar

[Iniciando a biblioteca COM](#)

Comentários

Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

Inicialização da biblioteca COM

Artigo • 07/08/2024

Qualquer programa do Windows que usa COM deve inicializar a biblioteca COM chamando a função [CoInitializeEx](#). Cada thread que usa uma interface COM deve fazer uma chamada separada para essa função. **CoInitializeEx** tem a seguinte assinatura:


```
C++

HRESULT CoInitializeEx(LPVOID pvReserved, DWORD dwCoInit);
```

O primeiro parâmetro é reservado e deve ser **NULL**. O segundo parâmetro especifica o modelo de threading que seu programa usará. O COM dá suporte a dois modelos de threading diferentes, *apartment threaded* e *multithreaded*. Se você especificar o apartment threading estará fazendo as seguintes garantias:

- Você acessará cada objeto COM de um único thread e você não compartilhará ponteiros de interface COM entre vários threads.
- O thread terá um loop de mensagem. (Veja [Mensagens de janela](#) no Módulo 1).

Se qualquer uma dessas restrições não for verdadeira, use o modelo multithreaded. Para especificar o modelo de threading, defina um dos sinalizadores a seguir no parâmetro *dwCoInit*.

 Expandir a tabela

Sinalizador	Descrição
COINIT_APARTMENTTHREADED	Apartment threaded.
COINIT_MULTITHREADED	Multithreaded.

Você deve definir exatamente um desses sinalizadores. Geralmente, um thread que cria uma janela deve usar o sinalizador **COINIT_APARTMENTTHREADED** e outros threads devem usar **COINIT_MULTITHREADED**. No entanto, alguns componentes COM exigem um modelo de threading específico.

⚠ Observação

Na verdade, mesmo que você especifique o apartment threading, ainda é possível compartilhar interfaces entre threads, usando uma técnica chamada *marshaling*. O

Marshaling está além do escopo deste módulo. O ponto importante é que, com o apartment threading, você nunca deve simplesmente copiar um ponteiro de interface para outro thread. Para obter mais informações sobre os modelos de threading COM, consulte [Processos, Threads e Apartments](#)).

Além dos sinalizadores já mencionados, é uma boa ideia definir o sinalizador **COINIT_DISABLE_OLE1DDE** no parâmetro *dwCoInit*. Definir esse sinalizador evita sobrecarga associada ao OLE (Banco de dados de vinculação e incorporação de objeto) 1.0, uma tecnologia obsoleta.

Veja como você inicializaria o COM para apartment threading:

C++

```
HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |  
COINIT_DISABLE_OLE1DDE);
```

O tipo de retorno **HRESULT** contém um código de erro ou êxito. Veremos o tratamento de erros COM na próxima seção.

Cancelando a inicialização da biblioteca COM

Para cada chamada bem-sucedida para [CoInitializeEx](#), você deve chamar [CoUninitialize](#) antes que a thread encerre. Essa função não recebe parâmetros e não tem valor de retorno.

C++

```
CoUninitialize();
```

Próximo

[Códigos de erro em COM](#)

Comentários

Esta página foi útil?

 Yes

 No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Códigos de erro no COM

Artigo • 12/06/2023

Para indicar êxito ou falha, métodos e funções COM retornam um valor do tipo **HRESULT**. Um **HRESULT** é um inteiro de 32 bits. O bit de alta ordem do **HRESULT** sinaliza êxito ou falha. Zero (0) indica êxito e 1 indica falha.

Isso produz os seguintes intervalos numéricos:

- Códigos de êxito: 0x0–0x7FFFFFFF.
- Códigos de erro: 0x80000000–0xFFFFFFFF.

Um pequeno número de métodos COM não retorna um valor **HRESULT**. Por exemplo, os métodos [AddRef](#) e [Release](#) retornam valores longos sem sinal. Mas cada método COM que retorna um código de erro faz isso retornando um valor **HRESULT**.

Para marcar se um método COM é bem-sucedido, examine o bit de alta ordem do **HRESULT** retornado. Os cabeçalhos do SDK do Windows fornecem duas macros que facilitam isso: a macro [SUCCEEDED](#) e a macro [FAILED](#). A macro **SUCCEEDED** retornará **TRUE** se um **HRESULT** for um código de êxito e **FALSE** se for um código de erro. O exemplo a seguir verifica se [CoInitializeEx](#) foi bem-sucedido.

C++

```
HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
    COINIT_DISABLE_OLE1DDE);

if (SUCCEEDED(hr))
{
    // The function succeeded.
}
else
{
    // Handle the error.
}
```

Às vezes, é mais conveniente testar a condição inversa. A macro [FAILED](#) faz o oposto de [SUCCEEDED](#). Ele retorna **TRUE** para um código de erro e **FALSE** para um código de êxito.

C++

```
HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
    COINIT_DISABLE_OLE1DDE);
```

```
if (FAILED(hr))
{
    // Handle the error.
}
else
{
    // The function succeeded.
}
```

Posteriormente neste módulo, examinaremos alguns conselhos práticos sobre como estruturar seu código para lidar com erros COM. (Consulte [Tratamento de erros em COM.](#))

Avançar

[Criando um objeto em COM](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Criando um objeto em COM

Artigo • 13/06/2023

Depois que um thread inicializa a biblioteca COM, é seguro que o thread use interfaces COM. Para usar uma interface COM, seu programa primeiro cria uma instância de um objeto que implementa essa interface.

Em geral, há duas maneiras de criar um objeto COM:

- O módulo que implementa o objeto pode fornecer uma função especificamente projetada para criar instâncias desse objeto.
- Como alternativa, o COM fornece uma função de criação genérica chamada [CoCreateInstance](#).

Por exemplo, use o objeto hipotético `Shape` do tópico [O que é uma interface COM?](#).

Nesse exemplo, o `Shape` objeto implementa uma interface chamada `IDrawable`. A biblioteca de gráficos que implementa o `Shape` objeto pode exportar uma função com a assinatura a seguir.

C++

```
// Not an actual Windows function.  
  
HRESULT CreateShape(IDrawable** ppShape);
```

Dada essa função, você pode criar um novo `Shape` objeto da seguinte maneira.

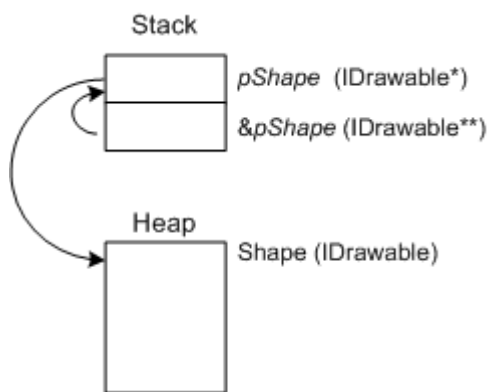
C++

```
IDrawable *pShape;  
  
HRESULT hr = CreateShape(&pShape);  
if (SUCCEEDED(hr))  
{  
    // Use the Shape object.  
}  
else  
{  
    // An error occurred.  
}
```

O parâmetro `ppShape` é do tipo pointer-to-pointer-to-`IDrawable`. Se você não viu esse padrão antes, o redirecionamento duplo pode ser intrigante.

Considere os requisitos da `CreateShape` função. A função deve devolver um `IDrawable` ponteiro ao chamador. Mas o valor retornado da função já é usado para o código de erro/êxito. Portanto, o ponteiro deve ser retornado por meio de um argumento para a função. O chamador passará uma variável do tipo `IDrawable*` para a função e a função substituirá essa variável por um novo `IDrawable` ponteiro. No C++, há apenas duas maneiras de uma função substituir um valor de parâmetro: passar por referência ou passar por endereço. O COM usa a última passagem por endereço. E o endereço de um ponteiro é um ponteiro para um ponteiro, portanto, o tipo de parâmetro deve ser `IDrawable**`.

Aqui está um diagrama para ajudar a visualizar o que está acontecendo.



A `CreateShape` função usa o endereço de `pShape (&pShape)` para gravar um novo valor de ponteiro em `pShape`.

CoCreateInstance: uma maneira genérica de criar objetos

A função `CoCreateInstance` fornece um mecanismo genérico para a criação de objetos. Para entender `CoCreateInstance`, tenha em mente que dois objetos COM podem implementar a mesma interface e um objeto pode implementar duas ou mais interfaces. Portanto, uma função genérica que cria objetos precisa de duas informações.

- Qual objeto criar.
- Qual interface obter do objeto.

Mas como podemos indicar essas informações quando chamamos a função? No COM, um objeto ou uma interface é identificado atribuindo-lhe um número de 128 bits, chamado guid (*identificador global exclusivo*). Os GUIDs são gerados de uma maneira que os torna efetivamente exclusivos. Os GUIDs são uma solução para o problema de como criar identificadores exclusivos sem uma autoridade de registro central. Às vezes, os GUIDs são chamados de UUIDs (*identificadores universalmente exclusivos*). Antes do

COM, eles eram usados no DCE/RPC (Distributed Computing Environment/Remote Procedure Call). Existem vários algoritmos para criar novos GUIDs. Nem todos esses algoritmos garantem estritamente a exclusividade, mas a probabilidade de criar acidentalmente o mesmo valor guid duas vezes é extremamente pequena—efetivamente zero. Os GUIDs podem ser usados para identificar qualquer tipo de entidade, não apenas objetos e interfaces. No entanto, esse é o único uso que nos preocupa neste módulo.

Por exemplo, a `Shapes` biblioteca pode declarar duas constantes GUID:

C++

```
extern const GUID CLSID_Shape;  
extern const GUID IID_IDrawable;
```

(Você pode supor que os valores numéricos reais de 128 bits para essas constantes sejam definidos em outro lugar.) A constante **CLSID_Shape** identifica o `Shape` objeto, enquanto a constante **IID_IDrawable** identifica a `IDrawable` interface. O prefixo "CLSID" significa *identificador de classe* e o *IID* do prefixo significa *identificador de interface*. Essas são convenções de nomenclatura padrão no COM.

Considerando esses valores, você criaria uma nova `Shape` instância da seguinte maneira:

C++

```
IDrawable *pShape;  
hr = CoCreateInstance(CLSID_Shape, NULL, CLSCTX_INPROC_SERVER,  
IID_IDrawable,  
    reinterpret_cast<void**>(&pShape));  
  
if (SUCCEEDED(hr))  
{  
    // Use the Shape object.  
}  
else  
{  
    // An error occurred.  
}
```

A função **CoCreateInstance** tem cinco parâmetros. O primeiro e o quarto parâmetros são o identificador de classe e o identificador de interface. Na verdade, esses parâmetros informam à função "Criar o objeto Shape e me dar um ponteiro para a interface IDrawable".

Defina o segundo parâmetro como **NULL**. (Para obter mais informações sobre o significado desse parâmetro, consulte o tópico [Agregação](#) na documentação com.) O terceiro parâmetro usa um conjunto de sinalizadores cuja finalidade main é especificar o *contexto de execução* para o objeto . O contexto de execução especifica se o objeto é executado no mesmo processo que o aplicativo; em um processo diferente no mesmo computador; ou em um computador remoto. A tabela a seguir mostra os valores mais comuns para esse parâmetro.

Sinalizador	Descrição
CLSCTX_INPROC_SERVER	Mesmo processo.
CLSCTX_LOCAL_SERVER	Processo diferente, mesmo computador.
CLSCTX_REMOTE_SERVER	Computador diferente.
CLSCTX_ALL	Use a opção mais eficiente à qual o objeto dá suporte. (A classificação, da mais eficiente para a menos eficiente, é: dentro do processo, fora do processo e entre computadores.)

A documentação de um componente específico pode informar a qual contexto de execução o objeto dá suporte. Caso contrário, use **CLSCTX_ALL**. Se você solicitar um contexto de execução ao qual o objeto não dá suporte, a função [CoCreateInstance](#) retornará o código de erro **REGDB_E_CLASSNOTREG**. Esse código de erro também pode indicar que o CLSID não corresponde a nenhum componente registrado no computador do usuário.

O quinto parâmetro para [CoCreateInstance](#) recebe um ponteiro para a interface . Como [CoCreateInstance](#) é um mecanismo genérico, esse parâmetro não pode ser fortemente tipado. Em vez disso, o tipo de dados é **void****, e o chamador deve forçar o endereço do ponteiro para um tipo **void**** . Essa é a finalidade do **reinterpret_cast** no exemplo anterior.

É crucial marcar o valor retornado de [CoCreateInstance](#). Se a função retornar um código de erro, o ponteiro da interface COM será inválido e tentar desreferenciar isso poderá causar uma falha no programa.

Internamente, a função [CoCreateInstance](#) usa várias técnicas para criar um objeto . No caso mais simples, ele pesquisa o identificador de classe no registro. A entrada do Registro aponta para uma DLL ou EXE que implementa o objeto . O [CoCreateInstance](#) também pode usar informações de um catálogo COM+ ou de um manifesto SxS (lado a lado). Independentemente disso, os detalhes são transparentes para o chamador. Para

obter mais informações sobre os detalhes internos do **CoCreateInstance**, consulte [Clientes e servidores COM](#).

O **Shapes** exemplo que temos usado é um pouco inventado, portanto, agora vamos recorrer a um exemplo real de COM em ação: exibindo a caixa de diálogo **Abrir** para o usuário selecionar um arquivo.

Avançar

[Exemplo: a caixa de diálogo Abrir](#)

Comentários

Esta página foi útil?

 **Yes**

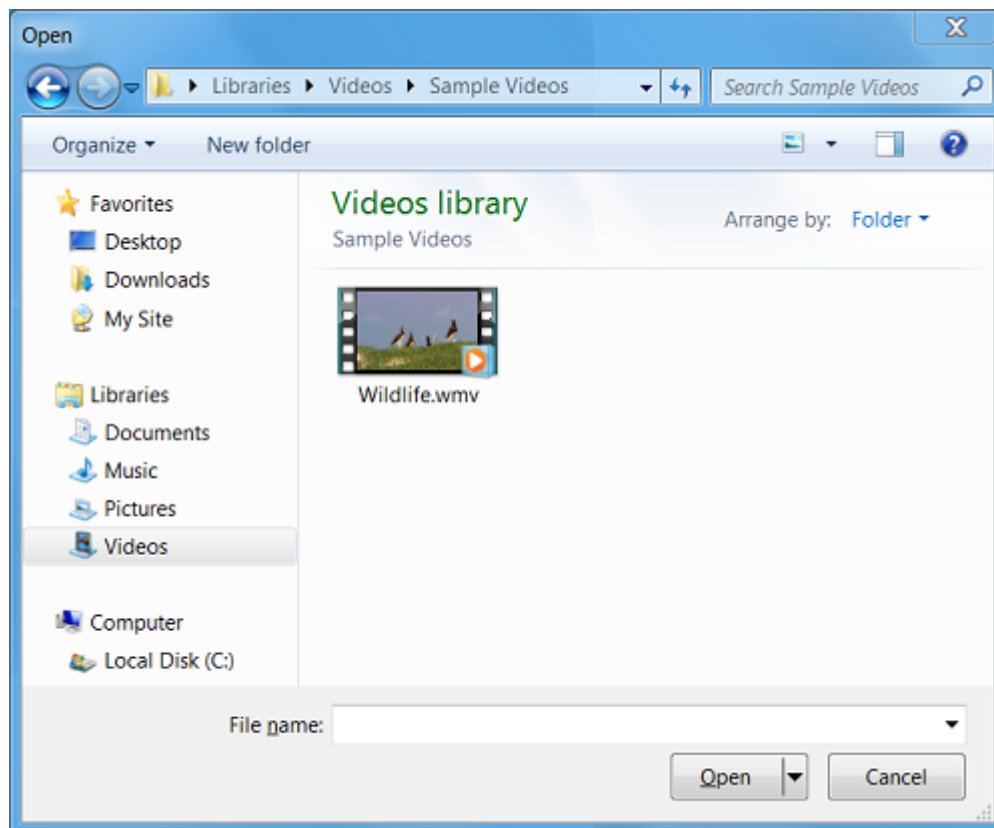
 **No**

[Obter ajuda no Microsoft Q&A](#)

Exemplo: a caixa de diálogo Abrir

Artigo • 12/06/2023

O `Shapes` exemplo que temos usado é um pouco inventado. Vamos recorrer a um objeto COM que você pode usar em um programa real do Windows: a caixa de diálogo **Abrir**.



Para mostrar a caixa de diálogo **Abrir**, um programa pode usar um objeto COM chamado objeto Common Item Dialog. A caixa de diálogo Item Comum implementa uma interface chamada `IFileOpenDialog`, que é declarada no arquivo de cabeçalho `Shobjidl.h`.

Aqui está um programa que exibe a caixa de diálogo **Abrir** para o usuário. Se o usuário selecionar um arquivo, o programa mostrará uma caixa de diálogo que contém o nome do arquivo.

C++

```
#include <windows.h>
#include <shobjidl.h>

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int
nCmdShow)
{
    HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
        COINIT_DISABLE_OLE1DDE);
```

```

if (SUCCEEDED(hr))
{
    IFileOpenDialog *pFileOpen;

    // Create the FileOpenDialog object.
    hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_ALL,
        IID_IFileOpenDialog, reinterpret_cast<void*>(&pFileOpen));

    if (SUCCEEDED(hr))
    {
        // Show the Open dialog box.
        hr = pFileOpen->Show(NULL);

        // Get the file name from the dialog box.
        if (SUCCEEDED(hr))
        {
            IShellItem *pItem;
            hr = pFileOpen->GetResult(&pItem);
            if (SUCCEEDED(hr))
            {
                PWSTR pszFilePath;
                hr = pItem->GetDisplayName(SIGDN_FILESYSPATH,
&pszFilePath);

                // Display the file name to the user.
                if (SUCCEEDED(hr))
                {
                    MessageBoxW(NULL, pszFilePath, L"File Path", MB_OK);
                    CoTaskMemFree(pszFilePath);
                }
                pItem->Release();
            }
            pItem->Release();
        }
        CoUninitialize();
    }
    return 0;
}

```

Esse código usa alguns conceitos que serão descritos posteriormente no módulo, portanto, não se preocupe se você não entender tudo aqui. Aqui está uma estrutura de tópicos básica do código:

1. Chame **ColnitializeEx** para inicializar a biblioteca COM.
2. Chame **CoCreateInstance** para criar o objeto Common Item Dialog e obter um ponteiro para a interface **IFileOpenDialog** do objeto.
3. Chame o método **Show** do objeto, que mostra a caixa de diálogo para o usuário. Esse método bloqueia até que o usuário ignore a caixa de diálogo.
4. Chame o método **GetResult** do objeto. Esse método retorna um ponteiro para um segundo objeto COM, chamado objeto *de item Shell* . O item Shell, que

- implementa a interface [IShellItem](#) , representa o arquivo selecionado pelo usuário.
5. Chame o método [GetDisplayName](#) do item shell. Esse método obtém o caminho do arquivo, na forma de uma cadeia de caracteres.
 6. Mostrar uma caixa de mensagem que exibe o caminho do arquivo.
 7. Chame [CoUninitialize](#) para não inicializar a biblioteca COM.

As etapas 1, 2 e 7 chamam funções definidas pela biblioteca COM. São funções COM genéricas. Etapas 3 a 5 chamam métodos definidos pelo objeto Common Item Dialog.

Este exemplo mostra as duas variedades de criação de objeto: a função [CoCreateInstance](#) genérica e um método ([GetResult](#)) específico do objeto Common Item Dialog.

Avançar

[Gerenciando o tempo de vida de um objeto](#)

Tópicos relacionados

[Abrir exemplo de caixa de diálogo](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Gerenciando o tempo de vida de um objeto

Artigo • 13/06/2023

Há uma regra para interfaces COM que ainda não mencionamos. Cada interface COM deve herdar, direta ou indiretamente, de uma interface chamada [IUnknown](#). Essa interface fornece alguns recursos de linha de base aos quais todos os objetos COM devem dar suporte.

A interface [IUnknown](#) define três métodos:

- [QueryInterface](#)
- [Addref](#)
- [Versão](#)

O método [QueryInterface](#) permite que um programa consulte os recursos do objeto em tempo de execução. Vamos dizer mais sobre isso no próximo tópico, [Solicitando um objeto para uma interface](#). Os métodos [AddRef](#) e [Release](#) são usados para controlar o tempo de vida de um objeto. Este é o assunto deste tópico.

Contagem de referências

Qualquer outra coisa que um programa possa fazer, em algum momento ele alocará e liberará recursos. Alocar um recurso é fácil. Saber quando liberar o recurso é difícil, especialmente se o tempo de vida do recurso se estende além do escopo atual. Esse problema não é exclusivo do COM. Qualquer programa que aloca memória de heap deve resolver o mesmo problema. Por exemplo, o C++ usa destruidores automáticos, enquanto C# e Java usam coleta de lixo. O COM usa uma abordagem chamada *contagem de referência*.

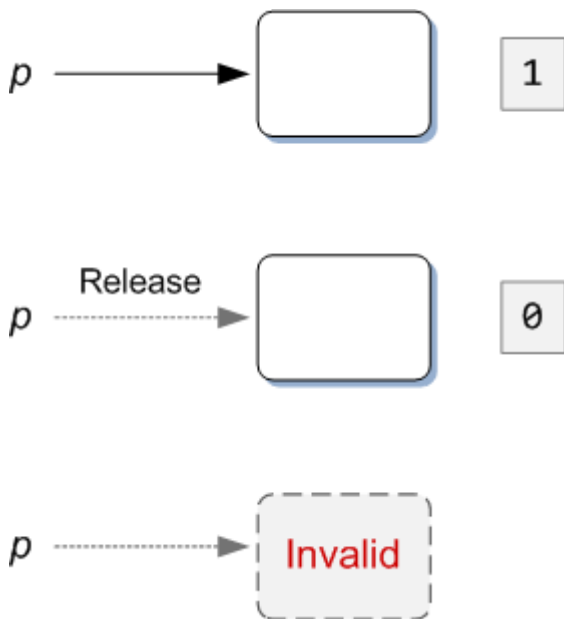
Cada objeto COM mantém uma contagem interna. Isso é conhecido como a contagem de referência. A contagem de referências rastreia quantas referências ao objeto estão ativas no momento. Quando o número de referências cai para zero, o objeto se exclui. A última parte vale a pena repetir: o objeto se exclui. O programa nunca exclui explicitamente o objeto .

Estas são as regras para contagem de referência:

- Quando o objeto é criado pela primeira vez, sua contagem de referência é 1. Neste ponto, o programa tem um único ponteiro para o objeto .

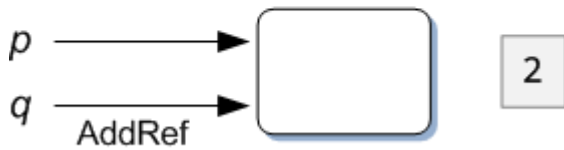
- O programa pode criar uma nova referência duplicando (copiando) o ponteiro. Ao copiar o ponteiro, você deve chamar o método **AddRef** do objeto . Esse método incrementa a contagem de referência por um.
- Quando terminar de usar um ponteiro para o objeto, você deverá chamar **Release**. O método **Release** diminui a contagem de referência por um. Ele também invalida o ponteiro. Não use o ponteiro novamente depois de chamar **Release**. (Se você tiver outros ponteiros para o mesmo objeto, poderá continuar a usar esses ponteiros.)
- Quando você chama **Release** com cada ponteiro, a contagem de referência de objeto do objeto atinge zero e o objeto se exclui.

O diagrama a seguir mostra um caso simples, mas típico.



O programa cria um objeto e armazena um ponteiro (*p*) no objeto . Neste ponto, a contagem de referência é 1. Quando o programa terminar de usar o ponteiro, ele **chamará Release**. A contagem de referência é decrementada para zero e o objeto se exclui. Agora *p* é inválido. É um erro usar *p* para qualquer outra chamada de método.

O próximo diagrama mostra um exemplo mais complexo.



Aqui, o programa cria um objeto e armazena o ponteiro p , como antes. Em seguida, o programa copia p para uma nova variável, q . Neste ponto, o programa deve chamar **AddRef** para incrementar a contagem de referência. A contagem de referência agora é 2 e há dois ponteiros válidos para o objeto. Agora suponha que o programa tenha terminado usando p . O programa chama **Release**, a contagem de referência vai para 1 e p não é mais válido. No entanto, q ainda é válido. Posteriormente, o programa termina usando q . Portanto, ele chama **Release** novamente. A contagem de referências vai para zero e o objeto se exclui.

Você pode se perguntar por que o programa copiaria p . Há dois motivos main: primeiro, talvez você queira armazenar o ponteiro em uma estrutura de dados, como uma lista. Em segundo lugar, talvez você queira manter o ponteiro além do escopo atual da variável original. Portanto, você a copiaria para uma nova variável com escopo mais amplo.

Uma vantagem da contagem de referência é que você pode compartilhar ponteiros em diferentes seções de código, sem os vários caminhos de código que coordenam para excluir o objeto. Em vez disso, cada caminho de código simplesmente chama **Release** quando esse caminho de código é feito usando o objeto. O objeto manipula a própria exclusão no momento correto.

Exemplo

Aqui está o código do [exemplo da caixa de diálogo Abrir](#) novamente.

C++

```
HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
    COINIT_DISABLE_OLE1DDE);

if (SUCCEEDED(hr))
{
    IFileOpenDialog *pFileOpen;

    hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_ALL,
        IID_IFileOpenDialog, reinterpret_cast<void*>(&pFileOpen));

    if (SUCCEEDED(hr))
    {
        hr = pFileOpen->Show(NULL);
        if (SUCCEEDED(hr))
        {
            IShellItem *pItem;
            hr = pFileOpen->GetResult(&pItem);
            if (SUCCEEDED(hr))
            {
                PWSTR pszFilePath;
                hr = pItem->GetDisplayName(SIGDN_FILESYSPATH, &pszFilePath);
                if (SUCCEEDED(hr))
                {
                    MessageBox(NULL, pszFilePath, L"File Path",
MB_OK);

                    CoTaskMemFree(pszFilePath);
                }
                pItem->Release();
            }
            pFileOpen->Release();
        }
        CoUninitialize();
    }
}
```

A contagem de referências ocorre em dois locais neste código. Primeiro, se o programa criar com êxito o objeto Common Item Dialog, ele deverá chamar **Release** no ponteiro *pFileOpen* .

C++

```
hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_ALL,
    IID_IFileOpenDialog, reinterpret_cast<void*>(&pFileOpen));

if (SUCCEEDED(hr))
```

```
{  
    // ...  
    pFileOpen->Release();  
}
```

Segundo, quando o método **GetResult** retorna um ponteiro para a interface **IShellItem**, o programa deve chamar **Release** no ponteiro *pItem*.

C++

```
hr = pFileOpen->GetResult(&pItem);  
  
if (SUCCEEDED(hr))  
{  
    // ...  
    pItem->Release();  
}
```

Observe que, em ambos os casos, a chamada **release** é a última coisa que acontece antes que o ponteiro saia do escopo. Observe também que **Release** é chamado somente depois que você testa o **HRESULT** para obter êxito. Por exemplo, se a chamada para **CoCreateInstance** falhar, o ponteiro *pFileOpen* não será válido. Portanto, seria um erro chamar **Release** no ponteiro.

Avançar

[Solicitando um objeto para uma interface](#)

Comentários

Esta página foi útil?

 Yes

 No

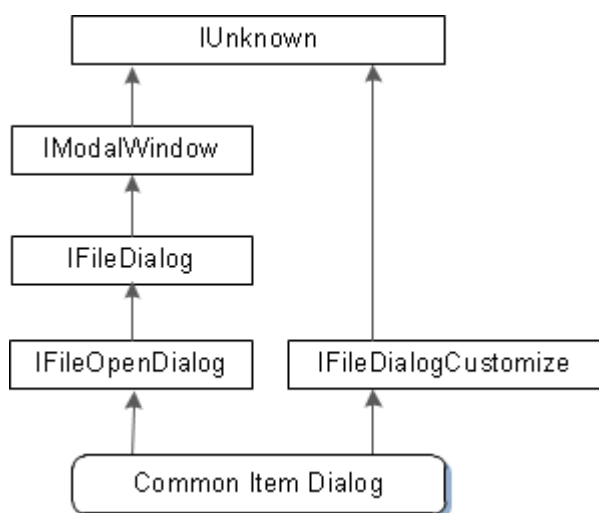
[Obter ajuda no Microsoft Q&A](#)

Solicitando um objeto para uma interface

Artigo • 13/06/2023

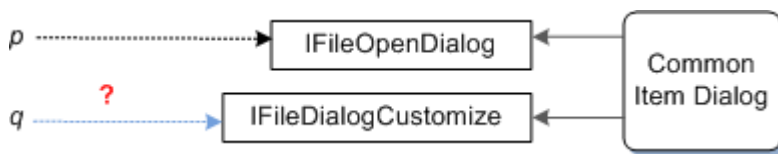
Vimos anteriormente que um objeto pode implementar mais de uma interface. O objeto Common Item Dialog é um exemplo real disso. Para dar suporte aos usos mais típicos, o objeto implementa a interface [IFileOpenDialog](#). Essa interface define métodos básicos para exibir a caixa de diálogo e obter informações sobre o arquivo selecionado. No entanto, para uso mais avançado, o objeto também implementa uma interface chamada [IFileDialogCustomize](#). Um programa pode usar essa interface para personalizar a aparência e o comportamento da caixa de diálogo, adicionando novos controles de interface do usuário.

Lembre-se de que cada interface COM deve herdar, direta ou indiretamente, da interface [IUnknown](#). O diagrama a seguir mostra a herança do objeto Common Item Dialog.



Como você pode ver no diagrama, o ancestral direto de [IFileOpenDialog](#) é a interface [IFileDialog](#), que por sua vez herda [IModalWindow](#). Conforme você sobe a cadeia de herança de [IFileOpenDialog](#) para [IModalWindow](#), as interfaces definem a funcionalidade de janela cada vez mais generalizada. Por fim, a interface [IModalWindow](#) herda [IUnknown](#). O objeto Common Item Dialog também implementa [IFileDialogCustomize](#), que existe em uma cadeia de herança separada.

Agora suponha que você tenha um ponteiro para a interface [IFileOpenDialog](#). Como você obteria um ponteiro para a interface [IFileDialogCustomize](#)?



Simplesmente converter o ponteiro [IFileOpenDialog](#) em um ponteiro [IFileDialogCustomize](#) não funcionará. Não há uma maneira confiável de "conversão cruzada" em uma hierarquia de herança, sem alguma forma de RTTI (informações de tipo de tempo de execução), que é um recurso altamente dependente de idioma.

A abordagem COM é *solicitar* que o objeto forneça um ponteiro [IFileDialogCustomize](#), usando a primeira interface como um canal para o objeto. Isso é feito chamando o método [IUnknown::QueryInterface](#) do primeiro ponteiro de interface. Você pode pensar em [QueryInterface](#) como uma versão independente da linguagem do `dynamic_cast` palavra-chave no C++.

O método [QueryInterface](#) tem a seguinte assinatura:

syntax

```
HRESULT QueryInterface(REFIID riid, void **ppvObject);
```

Com base no que você já sabe sobre [CoCreateInstance](#), talvez seja possível adivinhar como o [QueryInterface](#) funciona.

- O parâmetro *riid* é o GUID que identifica a interface que você está solicitando. O tipo de dados **REFIID** é um **typedef** para `const GUID&`. Observe que o CLSID (identificador de classe) não é necessário, pois o objeto já foi criado. Somente o identificador de interface é necessário.
- O parâmetro *ppvObject* recebe um ponteiro para a interface. O tipo de dados desse parâmetro é **void****, pelo mesmo motivo pelo qual [CoCreateInstance](#) usa esse tipo de dados: [QueryInterface](#) pode ser usado para consultar qualquer interface COM, portanto, o parâmetro não pode ser fortemente tipado.

Veja como você chamaria [QueryInterface](#) para obter um ponteiro [IFileDialogCustomize](#):

C++

```
hr = pFileOpen->QueryInterface(IID_IFileDialogCustomize,
    reinterpret_cast<void*>(&pCustom));
if (SUCCEEDED(hr))
{
    // Use the interface. (Not shown.)
    // ...
}
```

```
pCustom->Release();  
}  
else  
{  
    // Handle the error.  
}
```

Como sempre, marcar o valor retornado **HRESULT**, caso o método falhe. Se o método for bem-sucedido, você deverá chamar [Release](#) quando terminar de usar o ponteiro, conforme descrito em [Gerenciando o tempo de vida de um objeto](#).

Avançar

[Alocação de memória em COM](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Alocação de memória em COM

Artigo • 13/06/2023

Às vezes, um método aloca um buffer de memória no heap e retorna o endereço do buffer para o chamador. COM define um par de funções para alocar e liberar memória no heap.

- A função [CoTaskMemAlloc](#) aloca um bloco de memória.
- A função [CoTaskMemFree](#) libera um bloco de memória alocado com [CoTaskMemAlloc](#).

Vimos um exemplo desse padrão na caixa de [diálogo Abrir exemplo](#):

C++

```
PWSTR pszFilePath;  
hr = pItem->GetDisplayName(SIGDN_FILESYSPATH, &pszFilePath);  
if (SUCCEEDED(hr))  
{  
    // ...  
    CoTaskMemFree(pszFilePath);  
}
```

O método [GetDisplayName](#) aloca memória para uma cadeia de caracteres. Internamente, o método chama [CoTaskMemAlloc](#) para alocar a cadeia de caracteres. Quando o método retorna, *pszFilePath* aponta para o local de memória do novo buffer. O chamador é responsável por chamar [CoTaskMemFree](#) para liberar a memória.

Por que o COM define suas próprias funções de alocação de memória? Um dos motivos é fornecer uma camada de abstração sobre o alocador de heap. Caso contrário, alguns métodos podem chamar **malloc**, enquanto outros são chamados **de novos**. Em seguida, seu programa precisaria chamar **gratuitamente** em alguns casos e **excluir** em outros, e manter o controle de tudo isso rapidamente se tornaria impossível. As funções de alocação COM criam uma abordagem uniforme.

Outra consideração é o fato de que COM é um padrão *binário*, portanto, ele não está vinculado a uma linguagem de programação específica. Portanto, COM não pode confiar em nenhuma forma específica de linguagem de alocação de memória.

Avançar

[Práticas de codificação COM](#)

Comentários

Esta página foi útil?



Yes



No

[Obter ajuda no Microsoft Q&A](#)

Práticas de codificação COM

Artigo • 13/06/2023

Este tópico descreve maneiras de tornar seu código COM mais eficaz e robusto.

- [O operador __uuidof](#)
- [A macro IID_PPV_ARGS](#)
- [O padrão SafeRelease](#)
- [Ponteiros Inteligentes COM](#)

O operador __uuidof

Ao criar seu programa, você poderá obter erros de vinculador semelhantes aos seguintes:

```
unresolved external symbol "struct _GUID const IID_IDrawable"
```

Esse erro significa que uma constante GUID foi declarada com vinculação externa (**extern**) e o vinculador não pôde encontrar a definição da constante. O valor de uma constante GUID geralmente é exportado de um arquivo de biblioteca estática. Se você estiver usando Microsoft Visual C++, poderá evitar a necessidade de vincular uma biblioteca estática usando o operador **__uuidof**. Esse operador é uma extensão de idioma da Microsoft. Ele retorna um valor GUID de uma expressão. A expressão pode ser um nome de tipo de interface, um nome de classe ou um ponteiro de interface. Usando **__uuidof**, você pode criar o objeto Common Item Dialog da seguinte maneira:

C++

```
IFileOpenDialog *pFileOpen;  
hr = CoCreateInstance(__uuidof(FileOpenDialog), NULL, CLSCTX_ALL,  
    __uuidof(pFileOpen), reinterpret_cast<void**>(&pFileOpen));
```

O compilador extrai o valor guid do cabeçalho, portanto, nenhuma exportação de biblioteca é necessária.

❗ Observação

O valor guid está associado ao nome do tipo declarando `__declspec(uuid(...))` no cabeçalho. Para obter mais informações, consulte a documentação para **__declspec** na documentação do Visual C++.

A macro IID_PPV_ARGS

Vimos que [CoCreateInstance](#) e [QueryInterface](#) exigem coerção do parâmetro final para um tipo `void**`. Isso cria o potencial para uma incompatibilidade de tipo. Considere o fragmento de código a seguir:

C++

```
// Wrong!

IFileOpenDialog *pFileOpen;

hr = CoCreateInstance(
    __uuidof(FileOpenDialog),
    NULL,
    CLSCTX_ALL,
    __uuidof(IFileDialogCustomize),    // The IID does not match the
    pointer type!
    reinterpret_cast<void**>(&pFileOpen) // Coerce to void**.
);
```

Esse código solicita a interface [IFileDialogCustomize](#), mas passa um ponteiro [IFileOpenDialog](#). A expressão `reinterpret_cast` contorna o sistema de tipos C++, portanto, o compilador não capturará esse erro. Na melhor das hipóteses, se o objeto não implementar a interface solicitada, a chamada simplesmente falhará. Na pior das hipóteses, a função é bem-sucedida e você tem um ponteiro incompatível. Em outras palavras, o tipo de ponteiro não corresponde à vtable real na memória. Como você pode imaginar, nada de bom pode acontecer naquele momento.

⚠ Observação

Uma *vtable* (tabela de método virtual) é uma tabela de ponteiros de função. A vtable é como COM associa uma chamada de método à sua implementação em tempo de execução. Não coincidentemente, vtables são como a maioria dos compiladores C++ implementa métodos virtuais.

A macro [IID_PPV_ARGS](#) ajuda a evitar essa classe de erro. Para usar essa macro, substitua o seguinte código:

C++

```
__uuidof(IFileDialogCustomize), reinterpret_cast<void*>(&pFileOpen)
```

por este:

C++

```
IID_PPV_ARGS(&pFileOpen)
```

A macro insere `__uuidof(IFileOpenDialog)` automaticamente para o identificador de interface, portanto, é garantido que corresponda ao tipo de ponteiro. Aqui está o código modificado (e correto):

C++

```
// Right.
IFileOpenDialog *pFileOpen;
hr = CoCreateInstance(__uuidof(FileOpenDialog), NULL, CLSCTX_ALL,
    IID_PPV_ARGS(&pFileOpen));
```

Você pode usar a mesma macro com [QueryInterface](#):

C++

```
IFileDialogCustomize *pCustom;
hr = pFileOpen->QueryInterface(IID_PPV_ARGS(&pCustom));
```

O padrão SafeRelease

A contagem de referências é uma daquelas coisas na programação que é basicamente fácil, mas também é tediosa, o que facilita o erro. Os erros típicos incluem:

- Falha ao liberar um ponteiro de interface quando terminar de usá-lo. Essa classe de bug fará com que seu programa vazze memória e outros recursos, pois os objetos não são destruídos.
- Chamando [Release](#) com um ponteiro inválido. Por exemplo, esse erro poderá ocorrer se o objeto nunca tiver sido criado. Essa categoria de bug provavelmente fará com que seu programa falhe.
- Desreferenciando um ponteiro de interface após [Release](#) ser chamado. Esse bug pode causar falha no programa. Pior, isso pode fazer com que seu programa falhe aleatoriamente mais tarde, dificultando o rastreamento do erro original.

Uma maneira de evitar esses bugs é chamar **Release** por meio de uma função que libera com segurança o ponteiro. O código a seguir mostra uma função que faz isso:

C++

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

Essa função usa um ponteiro de interface COM como um parâmetro e faz o seguinte:

1. Verifica se o ponteiro é **NULL**.
2. Chama **Release** se o ponteiro não for **NULL**.
3. Define o ponteiro como **NULL**.

Aqui está um exemplo de como usar `SafeRelease`:

C++

```
void UseSafeRelease()
{
    IFileOpenDialog *pFileOpen = NULL;

    HRESULT hr = CoCreateInstance(__uuidof(FileOpenDialog), NULL,
        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));
    if (SUCCEEDED(hr))
    {
        // Use the object.
    }
    SafeRelease(&pFileOpen);
}
```

Se **CoCreateInstance** for bem-sucedido, a chamada para `SafeRelease` liberará o ponteiro. Se **CoCreateInstance** falhar, *pFileOpen* permanecerá **NULL**. A `SafeRelease` função verifica isso e ignora a chamada para **Release**.

Também é seguro chamar `SafeRelease` mais de uma vez no mesmo ponteiro, conforme mostrado aqui:

C++

```
// Redundant, but OK.
SafeRelease(&pFileOpen);
```

```
SafeRelease(&pFileOpen);
```

Ponteiros Inteligentes COM

A `SafeRelease` função é útil, mas exige que você se lembre de duas coisas:

- Inicialize cada ponteiro de interface para **NULL**.
- Chame `SafeRelease` antes que cada ponteiro saia do escopo.

Como programador C++, você provavelmente está pensando que não deveria ter que se lembrar de nenhuma dessas coisas. Afinal, é por isso que o C++ tem construtores e destruidores. Seria bom ter uma classe que encapsula o ponteiro de interface subjacente e inicializa e libera automaticamente o ponteiro. Em outras palavras, queremos algo assim:

C++

```
// Warning: This example is not complete.

template <class T>
class SmartPointer
{
    T* ptr;

public:
    SmartPointer(T *p) : ptr(p) { }
    ~SmartPointer()
    {
        if (ptr) { ptr->Release(); }
    }
};
```

A definição de classe mostrada aqui está incompleta e não é utilizável conforme mostrado. No mínimo, você precisaria definir um construtor de cópia, um operador de atribuição e uma maneira de acessar o ponteiro COM subjacente. Felizmente, você não precisa fazer nenhum deste trabalho, pois o Microsoft Visual Studio já fornece uma classe de ponteiro inteligente como parte da ATL (Biblioteca de Modelos Ativos).

A classe de ponteiro inteligente ATL é chamada **CComPtr**. (Há também uma classe **CComQIPtr**, que não é discutida aqui.) Aqui está o exemplo [Abrir Caixa de Diálogo](#) reescrita para usar **CComPtr**.

C++

```

#include <windows.h>
#include <shobjidl.h>
#include <atlbase.h> // Contains the declaration of CComPtr.

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int
nCmdShow)
{
    HRESULT hr = CoInitializeEx(NULL, COINIT_APARTMENTTHREADED |
        COINIT_DISABLE_OLE1DDE);
    if (SUCCEEDED(hr))
    {
        CComPtr<IFileOpenDialog> pFileOpen;

        // Create the FileOpenDialog object.
        hr = pFileOpen.CoCreateInstance(__uuidof(FileOpenDialog));
        if (SUCCEEDED(hr))
        {
            // Show the Open dialog box.
            hr = pFileOpen->Show(NULL);

            // Get the file name from the dialog box.
            if (SUCCEEDED(hr))
            {
                CComPtr<IShellItem> pItem;
                hr = pFileOpen->GetResult(&pItem);
                if (SUCCEEDED(hr))
                {
                    PWSTR pszFilePath;
                    hr = pItem->GetDisplayName(SIGDN_FILESYSPATH,
&pszFilePath);

                    // Display the file name to the user.
                    if (SUCCEEDED(hr))
                    {
                        MessageBox(NULL, pszFilePath, L"File Path", MB_OK);
                        CoTaskMemFree(pszFilePath);
                    }
                }

                // pItem goes out of scope.
            }

            // pFileOpen goes out of scope.
        }
        CoUninitialize();
    }
    return 0;
}

```

A main diferença entre esse código e o exemplo original é que essa versão não chama explicitamente **Release**. Quando a instância do **CComPtr** fica fora do escopo, o destruidor chama **Release** no ponteiro subjacente.

CComPtr é um modelo de classe. O argumento de modelo é o tipo de interface COM. Internamente, **CComPtr** contém um ponteiro desse tipo. **CComPtr** substitui **operator->()** e **operator&()** para que a classe atue como o ponteiro subjacente. Por exemplo, o código a seguir é equivalente a chamar o método **IFileOpenDialog::Show** diretamente:

C++

```
hr = pFileOpen->Show(NULL);
```

CComPtr também define um método **CComPtr::CoCreateInstance**, que chama a função COM **CoCreateInstance** com alguns valores de parâmetro padrão. O único parâmetro necessário é o identificador de classe, como mostra o próximo exemplo:

C++

```
hr = pFileOpen.CoCreateInstance(__uuidof(FileOpenDialog));
```

O método **CComPtr::CoCreateInstance** é fornecido puramente como uma conveniência; você ainda pode chamar a função **CoCreateInstance** COM, se preferir.

Avançar

[Tratamento de erros no COM](#)

Comentários

Esta página foi útil?

 Yes

 No


[Obter ajuda no Microsoft Q&A](#)

Tratamento de erros no COM

(Introdução ao Win32 e C++)

Artigo • 07/08/2024

O COM usa valores **HRESULT** para indicar o êxito ou a falha de uma chamada de método ou função. Vários cabeçalhos do SDK definem várias constantes **HRESULT**. Um conjunto comum de códigos em todo o sistema é definido em WinError.h. A tabela a seguir mostra alguns desses códigos de retorno em todo o sistema.

 Expandir a tabela

Constante	Valor numérico	Descrição
E_ACCESSDENIED	0x80070005	Acesso negado
E_FAIL	0x80004005	Erro não especificado.
E_INVALIDARG	0x80070057	Valor de parâmetro inválido.
E_OUTOFMEMORY	0x8007000E	Sem memória.
E_POINTER	0x80004003	NULL foi passado incorretamente para um valor de ponteiro.
E_UNEXPECTED	0x8000FFFF	Condição inesperada.
S_OK	0x0	Êxito.
S_FALSE	0x1	Êxito.

Todas as constantes com o prefixo "E_" são códigos de erro. As constantes **S_OK** e **S_FALSE** são ambas códigos de sucesso. Provavelmente 99% dos métodos COM retornam **S_OK** quando são bem-sucedidos; mas não deixe que esse fato o engane. Um método pode retornar outros códigos de êxito, portanto, sempre teste se há erros usando a macro **SUCCEEDED** ou **FAILED**. O código de exemplo a seguir mostra a maneira errada e a maneira certa de testar o sucesso de uma chamada de função.

C++

```
// Wrong.
HRESULT hr = SomeFunction();
if (hr != S_OK)
{
```

```

    printf("Error!\n"); // Bad. hr might be another success code.
}

// Right.
HRESULT hr = SomeFunction();
if (FAILED(hr))
{
    printf("Error!\n");
}

```

O código de sucesso **S_FALSE** merece menção. Alguns métodos usam **S_FALSE** para significar, grosso modo, uma condição negativa que não é uma falha. Também pode indicar um não operacional (no-op), o método foi bem-sucedido, mas não teve efeito. Por exemplo, a função **ColInitializeEx** retorna **S_FALSE** se você chamá-la uma segunda vez do mesmo thread. Se você precisar diferenciar entre **S_OK** e **S_FALSE** em seu código, deverá testar o valor diretamente, mas ainda usar **FAILED** ou **SUCCEEDED** para lidar com os casos restantes, conforme mostrado no exemplo de código a seguir.

C++

```

if (hr == S_FALSE)
{
    // Handle special case.
}
else if (SUCCEEDED(hr))
{
    // Handle general success case.
}
else
{
    // Handle errors.
    printf("Error!\n");
}

```

Alguns valores **HRESULT** são específicos para um determinado recurso ou subsistema do Windows. Por exemplo, a API de elementos gráficos Direct2D define o código de erro **D2DERR_UNSUPPORTED_PIXEL_FORMAT**, o que significa que o programa usou um formato de pixel sem suporte. A documentação do Windows geralmente fornece uma lista de códigos de erro específicos que um método pode retornar. No entanto, você não deve considerar essas listas como definitivas. Um método sempre pode retornar um valor **HRESULT** que não está listado na documentação. Novamente, use as macros **SUCCEEDED** e **FAILED**. Se você testar um código de erro específico, inclua também um caso padrão.

C++

```
if (hr == D2DERR_UNSUPPORTED_PIXEL_FORMAT)
{
    // Handle the specific case of an unsupported pixel format.
}
else if (FAILED(hr))
{
    // Handle other errors.
}
```

Padrões para tratamento de erros

Esta seção examina alguns padrões para lidar com erros COM de maneira estruturada. Cada padrão traz vantagens e desvantagens. Até certo ponto, a escolha é uma questão de gosto. Se você trabalhar em um projeto existente, talvez ele já tenha diretrizes de codificação que determinem um estilo específico. Independentemente do padrão que você adotar, o código robusto obedecerá às seguintes regras.

- Para cada método ou função que retorna um **HRESULT**, verifique o valor de devolução antes de continuar.
- Libere recursos depois que eles forem usados.
- Não tente acessar recursos inválidos ou não inicializados, como ponteiros **NULL**.
- Não tente usar um recurso depois de liberá-lo.

Com essas regras em mente, aqui estão quatro padrões para tratamento de erros.

- [Ifs aninhados](#)
- [Ifs em cascata](#)
- [Pule em falha](#)
- [Lance a falha](#)

Ifs aninhados

Após cada chamada que retorna um **HRESULT**, use uma instrução **if** para testar o êxito. Em seguida, coloque a próxima chamada de método dentro do escopo da instrução **if**. Mais instruções **if** podem ser aninhadas tão profundamente quanto necessário. Todos os exemplos de código anteriores neste módulo usaram esse padrão, mas aqui está novamente:

C++

```
HRESULT ShowDialog()
{
    IFileOpenDialog *pFileOpen;
```

```

HRESULT hr = CoCreateInstance(__uuidof(FileOpenDialog), NULL,
    CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));
if (SUCCEEDED(hr))
{
    hr = pFileOpen->Show(NULL);
    if (SUCCEEDED(hr))
    {
        IShellItem *pItem;
        hr = pFileOpen->GetResult(&pItem);
        if (SUCCEEDED(hr))
        {
            // Use pItem (not shown).
            pItem->Release();
        }
    }
    pFileOpen->Release();
}
return hr;
}

```

Vantagens

- As variáveis podem ser declaradas com escopo mínimo. Por exemplo, *pItem* não é declarado até que seja usado.
- Dentro de cada instrução **if**, certas invariáveis são verdadeiras: todas as chamadas anteriores foram bem-sucedidas e todos os recursos adquiridos ainda são válidos. No exemplo anterior, quando o programa atinge a instrução **if** ambas *pItem* e *pFileOpen* são reconhecidamente válidas.
- Fica evidente quando liberar ponteiros de interface e outros recursos. Você libera um recurso no final da instrução **if** que segue imediatamente a chamada que adquiriu o recurso.

Desvantagens

- Algumas pessoas acham o aninhamento profundo difícil de ler.
- O tratamento de erros é misturado com outras instruções de ramificação e loop. Isso pode tornar a lógica geral do programa mais difícil de seguir.

Ifs em cascata

Após cada chamada de método, use uma instrução **if** para testar o sucesso. Se o método for bem-sucedido, coloque a próxima chamada de método dentro do bloco **if**. Mas, em vez de aninhar mais instruções **if** coloque cada teste **SUCCEEDED** após o bloco **if**. Se algum método falhar, todos os testes **SUCCEEDED** restantes simplesmente falharão até que a parte inferior da função seja atingida.

C++

```
HRESULT ShowDialog()
{
    IFileOpenDialog *pFileOpen = NULL;
    IShellItem *pItem = NULL;

    HRESULT hr = CoCreateInstance(__uuidof(FileOpenDialog), NULL,
        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));

    if (SUCCEEDED(hr))
    {
        hr = pFileOpen->Show(NULL);
    }
    if (SUCCEEDED(hr))
    {
        hr = pFileOpen->GetResult(&pItem);
    }
    if (SUCCEEDED(hr))
    {
        // Use pItem (not shown).
    }

    // Clean up.
    SafeRelease(&pItem);
    SafeRelease(&pFileOpen);
    return hr;
}
```

Nesse padrão, você libera recursos no final da função. Se ocorrer um erro, alguns ponteiros poderão ser inválidos quando a função for encerrada. Chamar **Release** em um ponteiro inválido trará o programa (ou pior), portanto, você deve inicializar todos os ponteiros para **NULL** e verificar se eles são **NULL** antes de liberá-los. Este exemplo usa a função `SafeRelease` ponteiros inteligentes também são uma boa escolha.

Se você usar esse padrão, deverá ter cuidado com as construções de loop. Dentro de um loop, interrompa o loop se alguma chamada falhar.

Vantagens

- Esse padrão cria menos aninhamento do que o padrão "ifs aninhado".
- O fluxo de controle geral é mais fácil de ver.
- Os recursos são liberados em um ponto do código.

Desvantagens

- Todas as variáveis devem ser declaradas e inicializadas na parte superior da função.
- Se uma chamada falhar, a função fará várias verificações de erros desnecessárias, em vez de sair da função imediatamente.

- Como o fluxo de controle continua pela função após uma falha, é preciso ter cuidado para não acessar recursos inválidos em todo o corpo da função.
- Erros dentro de um loop exigem um caso especial.

Pule em falha

Após cada chamada de método, teste a falha (não o sucesso). Em caso de falha, pule para um rótulo próximo à parte inferior da função. Após o rótulo, mas antes de sair da função, libere os recursos.

C++

```
HRESULT ShowDialog()
{
    IFileOpenDialog *pFileOpen = NULL;
    IShellItem *pItem = NULL;

    HRESULT hr = CoCreateInstance(__uuidof(FileOpenDialog), NULL,
        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));
    if (FAILED(hr))
    {
        goto done;
    }

    hr = pFileOpen->Show(NULL);
    if (FAILED(hr))
    {
        goto done;
    }

    hr = pFileOpen->GetResult(&pItem);
    if (FAILED(hr))
    {
        goto done;
    }

    // Use pItem (not shown).

done:
    // Clean up.
    SafeRelease(&pItem);
    SafeRelease(&pFileOpen);
    return hr;
}
```

Vantagens

- O fluxo de controle geral é fácil de ver.

- Em todos os pontos do código após uma verificação **FAILED**, se você não tiver pulado para o rótulo, é garantido que todas as chamadas anteriores foram bem-sucedidas.
- Os recursos são liberados em um local no código.

Desvantagens

- Todas as variáveis devem ser declaradas e inicializadas na parte superior da função.
- Alguns programadores não gostam de usar **goto** em seu código. (No entanto, deve-se notar que esse uso de **goto** é altamente estruturado e o código nunca sai da chamada de função atual.)
- Instruções **goto** ignoram inicializadores.

Lance a falha

Em vez de pular para um rótulo, você pode lançar uma exceção quando um método falhar. Isso pode produzir um estilo mais idiomático de C++ se você estiver acostumado a escrever códigos à prova de exceções.

C++

```
#include <comdef.h> // Declares _com_error

inline void throw_if_fail(HRESULT hr)
{
    if (FAILED(hr))
    {
        throw _com_error(hr);
    }
}

void ShowDialog()
{
    try
    {
        CComPtr<IFileOpenDialog> pFileOpen;
        throw_if_fail(CoCreateInstance(__uuidof(FileOpenDialog), NULL,
            CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen)));

        throw_if_fail(pFileOpen->Show(NULL));

        CComPtr<IShellItem> pItem;
        throw_if_fail(pFileOpen->GetResult(&pItem));

        // Use pItem (not shown).
    }
    catch (_com_error err)
    {
        // Handle error.
    }
}
```



```
}  
}
```

Observe que este exemplo usa a classe **CComPtr** para gerenciar ponteiros de interface. Geralmente, se o código gerar exceções, você deverá seguir o padrão RAII (Aquisição de recursos é inicialização). Ou seja, cada recurso deve ser gerenciado por um objeto cujo destruidor garante que o recurso seja liberado corretamente. Se uma exceção for lançada, o destruidor terá a garantia de ser invocado. Caso contrário, seu programa pode vaziar recursos.

Vantagens

- Compatível com o código existente que usa tratamento de exceção.
- Compatível com bibliotecas C++ que lançam exceções, como a STL (Biblioteca de Modelos Padrão).

Desvantagens

- Requer objetos C++ para gerenciar recursos como memória ou identificadores de arquivo.
- Requer uma boa compreensão de como escrever código à prova de exceções.

Próximo

[Módulo 3. Windows Graphics](#)

Comentários

Esta página foi útil?

 Yes

 No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Módulo 3. Gráficos do Windows

Artigo • 12/06/2023

O [módulo 1](#) desta série mostrou como criar uma janela em branco. O [módulo 2](#) fez um pequeno desvio por meio do COM (Component Object Model), que é a base para muitas das APIs modernas do Windows. Agora é hora de adicionar elementos gráficos à janela em branco que criamos no Módulo 1.

Este módulo começa com uma visão geral de alto nível da arquitetura gráfica do Windows. Em seguida, analisamos Direct2D, uma poderosa API de elementos gráficos que foi introduzida no Windows 7.

Nesta seção

- [Visão geral da arquitetura de gráficos do Windows](#)
- [O Gerenciador de Janelas da Área de Trabalho](#)
- [Modo retido versus modo imediato](#)
- [Seu primeiro programa de Direct2D](#)
- [Renderizar Alvos, Dispositivos e Recursos](#)
- [Desenhar com Direct2D](#)
- [Pixels de DPI e Device-Independent](#)
- [Usando Cor em Direct2D](#)
- [Aplicar transformações em Direct2D](#)
- [Apêndice: Transformações de Matriz](#)

Tópicos relacionados

[Saiba como Programar para Windows no C++](#)

Comentários

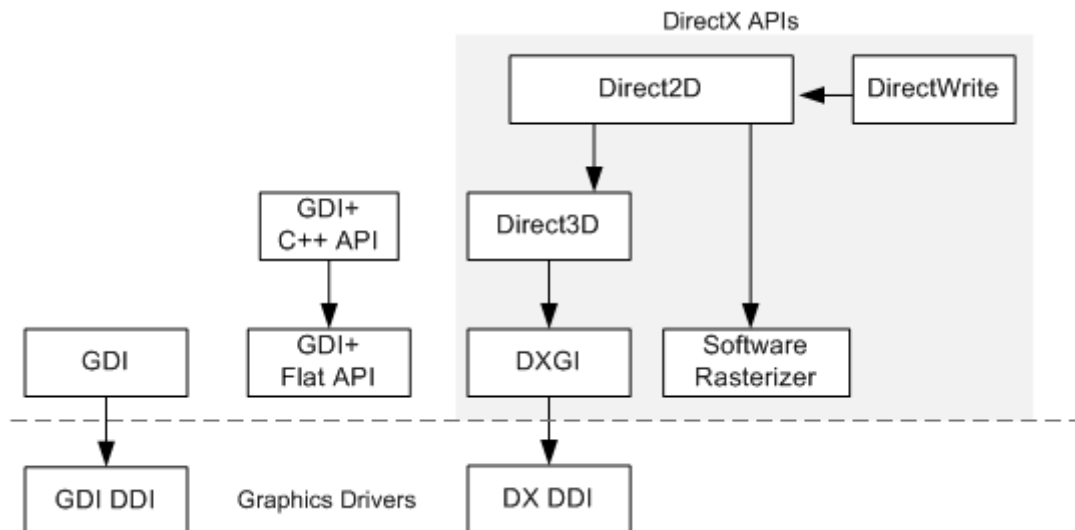
Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

Visão geral da arquitetura de gráficos do Windows

Artigo • 13/06/2023

O Windows fornece várias APIs C++/COM para elementos gráficos. Essas APIs são mostradas no diagrama a seguir.



- A GDI (Graphics Device Interface) é a interface gráfica original do Windows. A GDI foi gravada primeiro para Windows de 16 bits e, em seguida, atualizada para Windows de 32 bits e 64 bits.
- O GDI+ foi introduzido no Windows XP como sucessor do GDI. A biblioteca GDI+ é acessada por meio de um conjunto de classes C++ que encapsulam funções C simples. O .NET Framework também fornece uma versão gerenciada do GDI+ no namespace **System.Drawing**.
- O Direct3D dá suporte a elementos gráficos 3D.
- Direct2D é uma API moderna para elementos gráficos 2D, sucessora de GDI e GDI+.
- DirectWrite é um mecanismo de rasterização e layout de texto. Você pode usar gdi ou Direct2D para desenhar o texto rasterizado.
- A DXGI (DirectX Graphics Infrastructure) executa tarefas de baixo nível, como apresentar quadros para saída. A maioria dos aplicativos não usa DXGI diretamente. Em vez disso, ele serve como uma camada intermediária entre o driver gráfico e o Direct3D.

Direct2D e DirectWrite foram introduzidos no Windows 7. Eles também estão disponíveis para o Windows Vista e o Windows Server 2008 por meio de uma Atualização de Plataforma. Para obter mais informações, consulte [Atualização de plataforma para Windows Vista](#).

Direct2D é o foco deste módulo. Embora o GDI e o GDI+ continuem a ter suporte no Windows, Direct2D e DirectWrite são recomendados para novos programas. Em alguns casos, uma combinação de tecnologias pode ser mais prática. Para essas situações, Direct2D e DirectWrite são projetados para interoperar com a GDI.

As próximas seções descrevem alguns dos benefícios do Direct2D.

Aceleração de hardware

O termo *aceleração de hardware* refere-se a cálculos gráficos executados pela GPU (unidade de processamento gráfico), em vez da CPU. As GPUs modernas são altamente otimizadas para os tipos de computação usados na renderização de gráficos. Em geral, quanto mais desse trabalho for movido da CPU para a GPU, melhor.

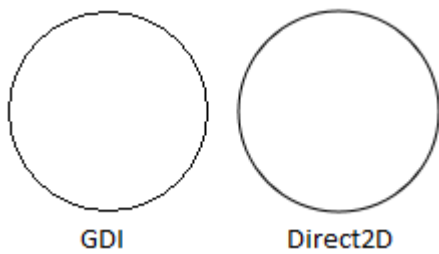
Embora a GDI dê suporte à aceleração de hardware para determinadas operações, muitas operações de GDI estão associadas à CPU. Direct2D está em camadas sobre Direct3D e aproveita ao máximo a aceleração de hardware fornecida pela GPU. Se a GPU não der suporte aos recursos necessários para Direct2D, Direct2D retornará à renderização de software. No geral, Direct2D supera GDI e GDI+ na maioria das situações.

Transparência e anti-aliasing

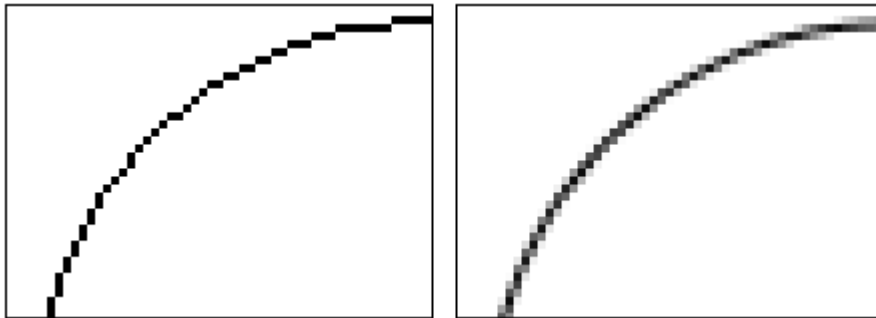
Direct2D dá suporte à combinação alfa totalmente acelerada por hardware (transparência).

A GDI tem suporte limitado para mesclagem alfa. A maioria das funções GDI não dá suporte à mesclagem alfa, embora a GDI dê suporte à mesclagem alfa durante uma operação `bitblt`. O GDI+ dá suporte à transparência, mas a mesclagem alfa é executada pela CPU, portanto, não se beneficia da aceleração de hardware.

A combinação alfa acelerada por hardware também permite o anti-aliasing. *Aliasing* é um artefato causado pela amostragem de uma função contínua. Por exemplo, quando uma linha curva é convertida em pixels, o alias pode causar uma aparência irregular. Qualquer técnica que reduza os artefatos causados pelo aliasing é considerada uma forma de anti-aliasing. Em gráficos, o anti-aliasing é feito pela combinação de bordas com a tela de fundo. Por exemplo, aqui está um círculo desenhado pela GDI e o mesmo círculo desenhado por Direct2D.



A próxima imagem mostra um detalhe de cada círculo.



O círculo desenhado pela GDI (esquerda) consiste em pixels pretos que se aproximam de uma curva. O círculo desenhado por Direct2D (à direita) usa mesclagem para criar uma curva mais suave.

A GDI não dá suporte à suavização quando desenha geometria (linhas e curvas). A GDI pode desenha texto anti-alias usando ClearType; mas, caso contrário, o texto GDI também é alias. O aliasing é particularmente perceptível para o texto, porque as linhas irregulares interrompem o design da fonte, tornando o texto menos legível. Embora o GDI+ dê suporte ao anti-aliasing, ele é aplicado pela CPU, portanto, o desempenho não é tão bom quanto Direct2D.

Gráficos vetoriais

Direct2D dá suporte a *elementos gráficos vetoriais*. Em gráficos vetoriais, fórmulas matemáticas são usadas para representar linhas e curvas. Essas fórmulas não dependem da resolução da tela, portanto, podem ser dimensionadas para dimensões arbitrárias. Elementos gráficos vetoriais são particularmente úteis quando uma imagem deve ser dimensionada para dar suporte a diferentes tamanhos de monitor ou resoluções de tela.

Avançar

[O Gerenciador de Janelas da Área de Trabalho](#)

Comentários

Esta página foi útil?

 Yes

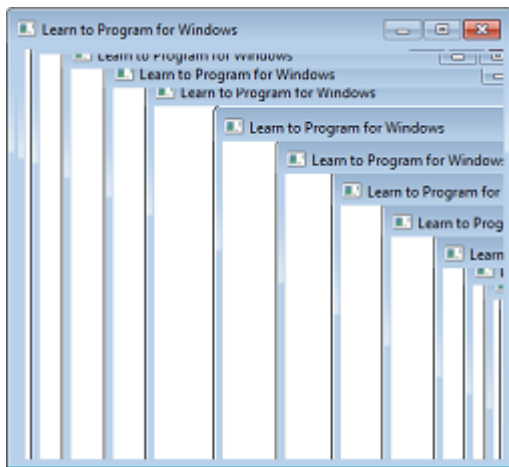
 No

[Obter ajuda no Microsoft Q&A](#)

O Gerenciador de Janelas da Área de Trabalho

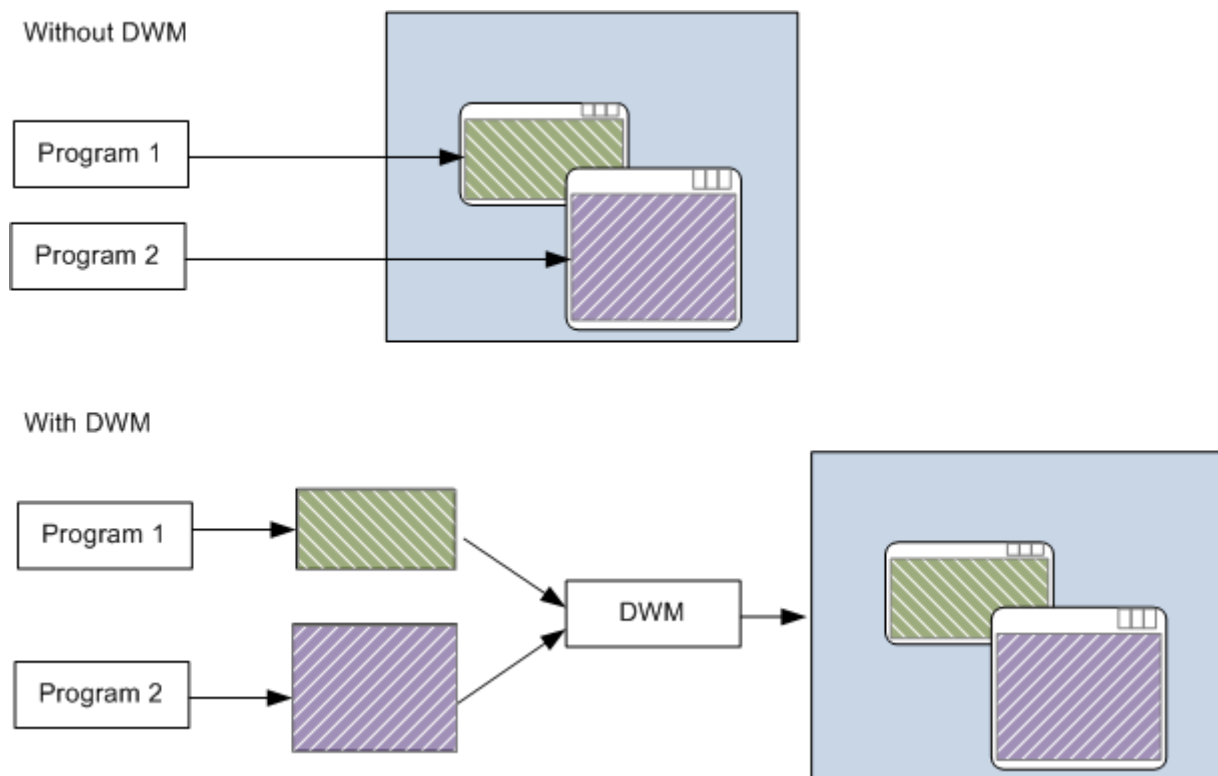
Artigo • 13/06/2023

Antes do Windows Vista, um programa do Windows desenhava diretamente para a tela. Em outras palavras, o programa gravaria diretamente no buffer de memória mostrado pelo vídeo cartão. Essa abordagem poderá causar artefatos visuais se uma janela não se repintar corretamente. Por exemplo, se o usuário arrastar uma janela sobre outra janela e a janela abaixo não se repintar rapidamente o suficiente, a janela superior mais pode deixar uma trilha:



A trilha é causada porque ambas as janelas pintam para a mesma área de memória. À medida que a janela mais alta é arrastada, a janela abaixo dela deve ser repintada. Se a repinização for muito lenta, isso causará os artefatos mostrados na imagem anterior.

O Windows Vista mudou fundamentalmente a forma como as janelas são desenhadas, introduzindo o DWM (Gerenciador de Janelas da Área de Trabalho). Quando o DWM está habilitado, uma janela não é mais desenhada diretamente para o buffer de exibição. Em vez disso, cada janela desenha para um buffer de memória fora da tela, também chamado de *superfície offscreen*. Em seguida, o DWM compõe essas superfícies na tela.



O DWM oferece várias vantagens em relação à arquitetura gráfica antiga.

- Menos mensagens de repintar. Quando uma janela é obstruída por outra janela, a janela obstruída não precisa se repintar.
- Artefatos reduzidos. Anteriormente, arrastar uma janela poderia criar artefatos visuais, conforme descrito.
- Efeitos visuais. Como o DWM é responsável por compor a tela, ele pode renderizar áreas translúcidas e desfocadas da janela.
- Dimensionamento automático para DPI alta. Embora o dimensionamento não seja a maneira ideal de lidar com alta DPI, é um fallback viável para aplicativos mais antigos que não foram projetados para configurações de DPI altas. (Retornaremos a este tópico mais tarde, na seção [DPI e Device-Independent Pixels](#).)
- Modos de exibição alternativos. O DWM pode usar as superfícies fora da tela de várias maneiras interessantes. Por exemplo, o DWM é a tecnologia por trás do Windows Flip 3D, miniaturas e transições animadas.

Observe, no entanto, que não há garantia de que o DWM esteja habilitado. Os elementos gráficos cartão podem não dar suporte aos requisitos do sistema DWM e os usuários podem desabilitar o DWM por meio do painel de controle **Propriedades do Sistema**. Isso significa que seu programa não deve depender do comportamento de repintamento do DWM. Teste o programa com o DWM desabilitado para garantir que ele seja repinto corretamente.

Avançar

Comentários

Esta página foi útil?



Yes



No

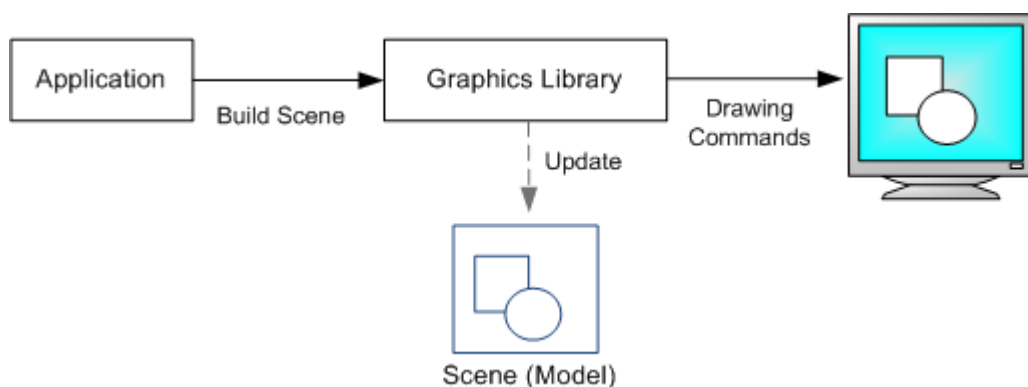
[Obter ajuda no Microsoft Q&A](#)

Modo retido versus modo imediato

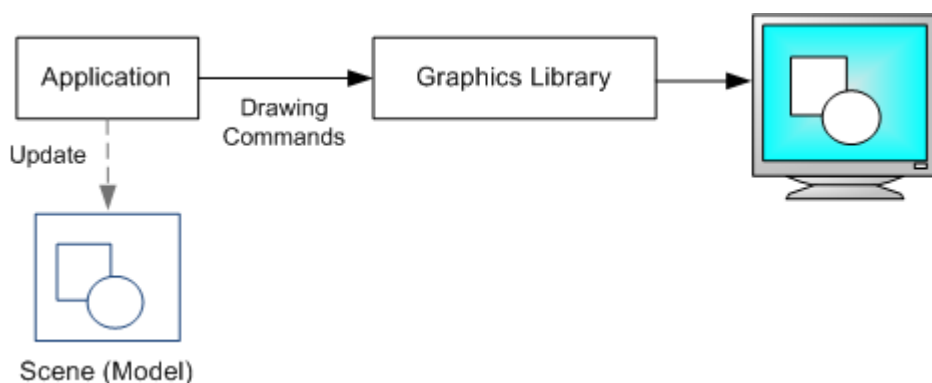
Artigo • 13/06/2023

As APIs gráficas podem ser divididas em APIs *de modo retido* e APIs *de modo imediato*. Direct2D é uma API de modo imediato. Windows Presentation Foundation (WPF) é um exemplo de uma API de modo retido.

Uma API de modo retido é declarativa. O aplicativo constrói uma cena a partir de primitivos gráficos, como formas e linhas. A biblioteca de gráficos armazena um modelo da cena na memória. Para desenhar um quadro, a biblioteca de gráficos transforma a cena em um conjunto de comandos de desenho. Entre quadros, a biblioteca de gráficos mantém a cena na memória. Para alterar o que é renderizado, o aplicativo emite um comando para atualizar a cena, por exemplo, para adicionar ou remover uma forma. Em seguida, a biblioteca é responsável por redesenhar a cena.



Uma API de modo imediato é um procedimento. Sempre que um novo quadro é desenhado, o aplicativo emite diretamente os comandos de desenho. A biblioteca de gráficos não armazena um modelo de cena entre quadros. Em vez disso, o aplicativo mantém o controle da cena.



As APIs de modo retido podem ser mais simples de usar, pois a API faz mais do trabalho para você, como inicialização, manutenção de estado e limpeza. Por outro lado, eles geralmente são menos flexíveis, porque a API impõe seu próprio modelo de cena. Além disso, uma API de modo retido pode ter requisitos de memória mais altos, pois precisa

fornecer um modelo de cena de uso geral. Com uma API de modo imediato, você pode implementar otimizações direcionadas.

Avançar

[Seu primeiro programa de Direct2D](#)

Comentários

Esta página foi útil?

 Yes

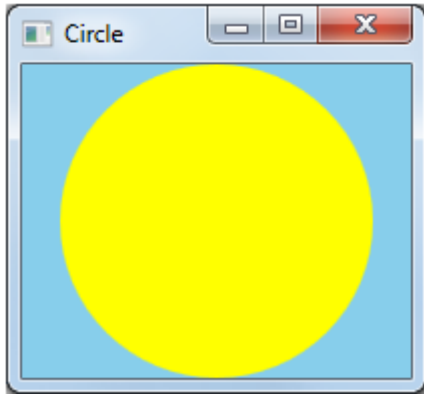
 No

[Obter ajuda no Microsoft Q&A](#)

Seu primeiro programa de Direct2D

Artigo • 13/06/2023

Vamos criar nosso primeiro programa de Direct2D. O programa não faz nada sofisticado — ele apenas desenha um círculo que preenche a área do cliente da janela. Mas este programa apresenta muitos conceitos essenciais Direct2D.



Aqui está a listagem de código para o programa Circle. O programa reutiliza a `BaseWindow` classe que foi definida no tópico [Gerenciando o estado do aplicativo](#). Tópicos posteriores examinarão o código em detalhes.

C++

```
#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);
}

```

Você pode baixar o projeto completo do Visual Studio em [Direct2D Exemplo de Círculo](#).

O namespace D2D1

O namespace **D2D1** contém funções auxiliares e classes. Elas não fazem parte estritamente da API de Direct2D — você pode programar Direct2D sem usá-las — mas ajudam a simplificar seu código. O namespace **D2D1** contém:

- Uma classe [ColorF](#) para construir valores de cor.
- Uma [Matrix3x2F](#) para construir matrizes de transformação.
- Um conjunto de funções para inicializar Direct2D estruturas.

Você verá exemplos do namespace **D2D1** ao longo deste módulo.

Avançar

[Renderizar Alvos, Dispositivos e Recursos](#)

Tópicos relacionados

[Exemplo de círculo Direct2D](#)

Comentários

Esta página foi útil?

 Yes

 No

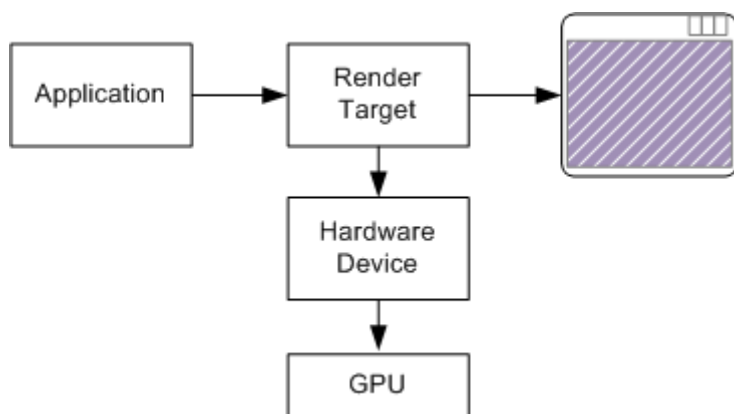
[Obter ajuda no Microsoft Q&A](#)

Renderizar Alvos, Dispositivos e Recursos

Artigo • 07/08/2024

Um *destino de renderização* é simplesmente o local onde seu programa será desenhado. Normalmente, o destino de renderização é uma janela (especificamente, a área do cliente da janela). Também pode ser um bitmap na memória que não é exibido. Um destino de renderização é representado pela interface [ID2D1RenderTarget](#).

Um *dispositivo* é uma abstração que representa o que de fato desenha os pixels. Um dispositivo de hardware usa a GPU para um desempenho mais rápido, enquanto um dispositivo de software usa a CPU. O aplicativo não cria o dispositivo. Em vez disso, o dispositivo é criado implicitamente quando o aplicativo cria o destino de renderização. Cada destino de renderização é associado a um dispositivo específico, seja hardware ou software.



Um *recurso* é um objeto que o programa usa para desenhar. Aqui estão alguns exemplos de recursos definidos em Direct2D:

- **Pincel.** Controla como as linhas e regiões são pintadas. Os tipos de pincel incluem pincéis de cor sólida e pincéis de gradiente.
- **Estilo de traço.** Controla a aparência de uma linha, por exemplo, tracejada ou sólida.
- **Geometria.** Representa uma coleção de linhas e curvas.
- **Malha.** Uma forma composta de triângulos. Os dados de malha podem ser consumidos diretamente pela GPU, ao contrário dos dados de geometria, que devem ser convertidos antes da renderização.

Os destinos de renderização também são considerados um tipo de recurso.

Alguns recursos se beneficiam da aceleração de hardware. Um recurso desse tipo está sempre associado a um dispositivo específico, seja hardware (GPU) ou software (CPU).

Esse tipo de recurso é chamado de *dependente do dispositivo*. Pincéis e malhas são exemplos de recursos dependentes do dispositivo. Se o dispositivo ficar indisponível, o recurso deverá ser recriado para um novo dispositivo.

Outros recursos são mantidos na memória da CPU, independentemente do dispositivo usado. Esses recursos são *independentes do dispositivo*, pois não estão associados a um dispositivo específico. Não é necessário recriar recursos independentes do dispositivo quando o dispositivo é alterado. Os estilos e geometrias de traçado são recursos independentes do dispositivo.

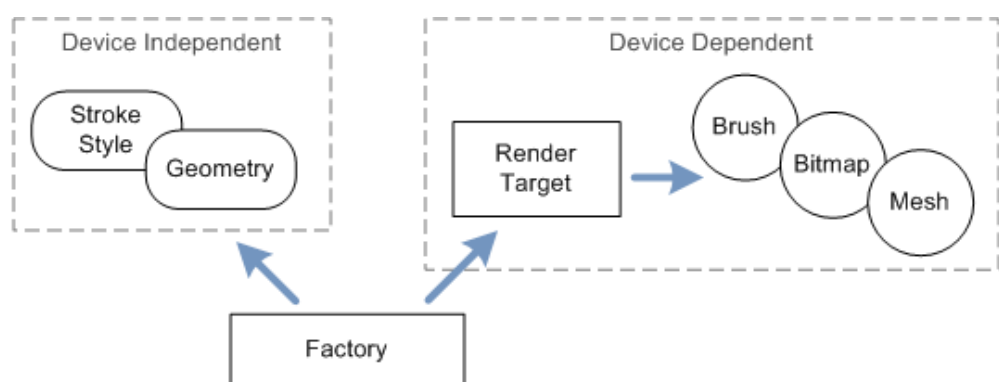
A documentação do Windows para cada recurso indica se o recurso é dependente ou independente do dispositivo. Cada tipo de recurso é representado por uma interface derivada de [ID2D1Resource](#). Por exemplo, os pincéis são representados pela interface [ID2D1Brush](#).

O objeto de fábrica Direct2D

A primeira etapa ao usar Direct2D é criar uma instância do objeto de fábrica Direct2D. Na programação de computadores, uma *fábrica* é um objeto que cria outros objetos. A fábrica Direct2D cria os seguintes tipos de objetos:

- Destinos de renderização.
- Recursos independentes do dispositivo, como estilos de traçado e geometrias.

Os recursos dependentes do dispositivo, como pincéis e bitmaps, são criados pelo objeto de destino de renderização.



Para criar o objeto de fábrica Direct2D, chame a função [D2D1CreateFactory](#).

C++

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

O primeiro parâmetro é um sinalizador que especifica as opções de criação. O sinalizador **D2D1_FACTORY_TYPE_SINGLE_THREADED** significa que você não chamará Direct2D de vários threads. Para dar suporte a chamadas de vários threads, especifique **D2D1_FACTORY_TYPE_MULTI_THREADED**. Se o programa usar um único thread para chamar Direct2D, a opção single-threaded será mais eficiente.

O segundo parâmetro para a função **D2D1CreateFactory** recebe um ponteiro para a interface **ID2D1Factory**.

Você deve criar o objeto de fábrica Direct2D antes da primeira mensagem **WM_PAINT**. O manipulador de mensagens **WM_CREATE** é um bom lugar para criar a fábrica:

C++

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

Criação de recursos Direct2D

O programa Circle usa os seguintes recursos dependentes do dispositivo:

- Um alvo de renderização que está associado à janela do aplicativo.
- Um pincel de cor sólida para pintar o círculo.

Cada um desses recursos é representado por uma interface COM:

- A interface **ID2D1HwndRenderTarget** representa o destino de renderização.
- A interface **ID2D1SolidColorBrush** representa o pincel.

O programa Circle armazena ponteiros para essas interfaces como variáveis de membro da `MainWindow` classe:

C++

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

O código a seguir cria esses dois recursos.

C++

```

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 criar um destino de renderização para uma janela, chame o método [ID2D1Factory::CreateHwndRenderTarget](#) na fábrica Direct2D.

- O primeiro parâmetro especifica opções comuns a qualquer tipo de destino de renderização. Aqui, passamos as opções padrão chamando a função auxiliar [D2D1::RenderTargetProperties](#).
- O segundo parâmetro especifica o identificador para a janela, mais o tamanho do destino de renderização em pixels.
- O terceiro parâmetro recebe um ponteiro [ID2D1HwndRenderTarget](#).

Para criar o pincel de cor sólida, chame o método [ID2D1RenderTarget::CreateSolidColorBrush](#) no destino de renderização. A cor é dada como um valor [D2D1_COLOR_F](#). Para obter mais informações sobre cores no Direct2D, consulte [Uso de cores no Direct2D](#).

Além disso, observe que, se o destino de renderização já existir, o método `CreateGraphicsResources` `S_OK` **retornará** sem fazer nada. O motivo desse design ficará evidente no próximo tópico.

Próximo

[Desenhar com Direct2D](#)

Comentários

Esta página foi útil?

 Yes

 No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Desenhar com Direct2D

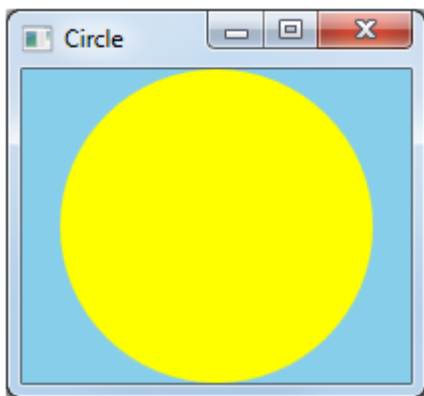
Artigo • 13/06/2023

Depois de criar seus recursos gráficos, você estará pronto para desenhar.

Desenhando uma elipse

O programa [Circle](#) executa uma lógica de desenho muito simples:

1. Preencha a tela de fundo com uma cor sólida.
2. Desenhe um círculo preenchido.



Como o destino de renderização é uma janela (em vez de um bitmap ou outra superfície fora da tela), o desenho é feito em resposta a mensagens [WM_PAINT](#). O código a seguir mostra o procedimento de janela para o programa Circle.

C++

```
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);
}
```

Aqui está o código que desenha o círculo.

C++

```

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);
    }
}

```

A interface **ID2D1RenderTarget** é usada para todas as operações de desenho. O método do `OnPaint` programa faz o seguinte:

1. O método **ID2D1RenderTarget::BeginDraw** sinaliza o início do desenho.
2. O método **ID2D1RenderTarget::Clear** preenche todo o destino de renderização com uma cor sólida. A cor é fornecida como uma estrutura **D2D1_COLOR_F**. Você pode usar a classe **D2D1::ColorF** para inicializar a estrutura. Para obter mais informações, consulte [Usando cor em Direct2D](#).
3. O método **ID2D1RenderTarget::FillEllipse** desenha uma elipse preenchida, usando o pincel especificado para o preenchimento. Uma elipse é especificada por um ponto central e os raios x e y. Se os raios x e y forem os mesmos, o resultado será um círculo.
4. O método **ID2D1RenderTarget::EndDraw** sinaliza a conclusão do desenho para esse quadro. Todas as operações de desenho devem ser colocadas entre chamadas para **BeginDraw** e **EndDraw**.

Todos os métodos **BeginDraw**, **Clear** e **FillEllipse** têm um tipo de retorno **void**. Se ocorrer um erro durante a execução de qualquer um desses métodos, o erro será sinalizado por meio do valor retornado do método **EndDraw**. O

`CreateGraphicsResources` método é mostrado no tópico [Criando recursos Direct2D](#). Esse método cria o destino de renderização e o pincel de cor sólida.

O dispositivo pode armazenar em buffer os comandos de desenho e adiar a execução deles até **que EndDraw** seja chamado. Você pode forçar o dispositivo a executar

quaisquer comandos de desenho pendentes chamando `ID2D1RenderTarget::Flush`. No entanto, a liberação pode reduzir o desempenho.

Manipulando a perda de dispositivo

Enquanto o programa está em execução, o dispositivo gráfico que você está usando pode ficar indisponível. Por exemplo, o dispositivo poderá ser perdido se a resolução de exibição for alterada ou se o usuário remover o adaptador de exibição. Se o dispositivo for perdido, o destino de renderização também se tornará inválido, juntamente com todos os recursos dependentes do dispositivo associados ao dispositivo. Direct2D sinaliza um dispositivo perdido retornando o código de erro `D2DERR_RECREATE_TARGET` do método `EndDraw`. Se você receber esse código de erro, deverá recriar o destino de renderização e todos os recursos dependentes do dispositivo.

Para descartar um recurso, basta liberar a interface para esse recurso.

C++

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

A criação de um recurso pode ser uma operação cara, portanto, não recrie seus recursos para cada `mensagem de WM_PAINT`. Crie um recurso uma vez e armazene o ponteiro do recurso em cache até que o recurso se torne inválido devido à perda de dispositivo ou até que você não precise mais desse recurso.

O loop de renderização do Direct2D

Independentemente do que você desenhar, seu programa deve executar um loop semelhante ao seguinte.

1. Criar recursos independentes do dispositivo.
2. Renderize a cena.
 - a. Verifique se existe um destino de renderização válido. Caso contrário, crie o destino de renderização e os recursos dependentes do dispositivo.
 - b. Chame `ID2D1RenderTarget::BeginDraw`.
 - c. Emita comandos de desenho.
 - d. Chame `ID2D1RenderTarget::EndDraw`.

e. Se **EndDraw** retornar **D2DERR_RECREATE_TARGET**, descarte o destino de renderização e os recursos dependentes do dispositivo.

3. Repita a etapa 2 sempre que precisar atualizar ou redesenhar a cena.

Se o destino de renderização for uma janela, a etapa 2 ocorrerá sempre que a janela receber uma **mensagem de WM_PAINT** .

O loop mostrado aqui lida com a perda de dispositivo descartando os recursos dependentes do dispositivo e recriando-os no início do próximo loop (etapa 2a).

Avançar

[DPI e pixels de Device-Independent](#)

Comentários

Esta página foi útil?



Yes



No

[Obter ajuda no Microsoft Q&A](#)

DPI e pixels independentes de dispositivo

Artigo • 25/05/2023

Para programar efetivamente com elementos gráficos do Windows, você deve entender dois conceitos relacionados:

- Pontos por polegada (DPI)
- DIPs (pixel independente de dispositivo).

Vamos começar com o DPI. Isso exigirá um pequeno desvio para tipografia. Na tipografia, o tamanho do tipo é medido em unidades chamadas *pontos*. Um ponto é igual a 1/72 de polegada.

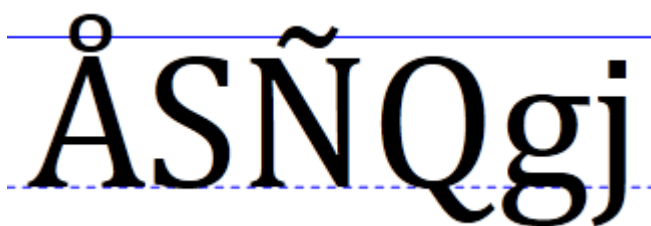
1 pt = 1/72 polegada

❗ Observação

Essa é a definição de ponto de publicação da área de trabalho. Historicamente, a medida exata de um ponto variou.

Por exemplo, uma fonte de 12 pontos foi projetada para caber dentro de uma linha de texto de 1/6" (12/72). Obviamente, isso não significa que todos os caracteres na fonte tenham exatamente 1/6" de altura. Na verdade, alguns caracteres podem ser mais altos que 1/6". Por exemplo, em muitas fontes, o caractere Å é mais alto que a altura nominal da fonte. Para exibir corretamente, a fonte precisa de algum espaço adicional entre o texto. Esse espaço é chamado de *à esquerda*.

A ilustração a seguir mostra uma fonte de 72 pontos. As linhas sólidas mostram uma caixa delimitadora de 1" de altura ao redor do texto. A linha tracejada é chamada *de linha de base*. A maioria dos caracteres em uma fonte está na linha de base. A altura da fonte inclui a parte acima da linha de base (a *ascensão*) e a parte abaixo da linha de base (a *descida*). Na fonte mostrada aqui, a ascensão é de 56 pontos e a descida é de 16 pontos.



The image shows a sample of the font 'ÅSÑQgj' in a large, black, serif typeface. The characters are positioned between two solid horizontal blue lines that define a 1-inch tall bounding box. A dashed horizontal blue line runs through the middle of the characters, representing the baseline. The characters 'Å', 'S', and 'Ñ' are tall, reaching the top solid line. The character 'Q' is shorter, reaching about halfway up. The characters 'g' and 'j' are lowercase and extend below the dashed baseline, reaching the bottom solid line.

No entanto, quando se trata de uma exibição de computador, medir o tamanho do texto é problemático, pois os pixels não têm o mesmo tamanho. O tamanho de um pixel depende de dois fatores: a resolução de exibição e o tamanho físico do monitor.

Portanto, polegadas físicas não são uma medida útil, porque não há nenhuma relação fixa entre polegadas físicas e pixels. Em vez disso, as fontes são medidas em unidades *lógicas*. Uma fonte de 72 pontos é definida como uma polegada lógica de altura. As polegadas lógicas são convertidas em pixels. Por muitos anos, o Windows usou a seguinte conversão: uma polegada lógica é igual a 96 pixels. Usando esse fator de dimensionamento, uma fonte de 72 pontos é renderizada com 96 pixels de altura. Uma fonte de 12 pontos tem 16 pixels de altura.

$12 \text{ pontos} = 12/72 \text{ polegada lógica} = 1/6 \text{ polegada lógica} = 96/6 \text{ pixels} = 16 \text{ pixels}$

Esse fator de dimensionamento é descrito como DPI (96 pontos por polegada). O termo dots deriva da impressão, onde os pontos físicos de tinta são colocados no papel. Para telas de computador, seria mais preciso dizer 96 pixels por polegada lógica, mas o termo DPI ficou preso.

Como os tamanhos reais de pixel variam, o texto legível em um monitor pode ser muito pequeno em outro monitor. Além disso, as pessoas têm preferências diferentes— algumas pessoas preferem texto maior. Por esse motivo, o Windows permite que o usuário altere a configuração de DPI. Por exemplo, se o usuário definir a exibição como 144 DPI, uma fonte de 72 pontos será de 144 pixels de altura. As configurações de DPI padrão são 100% (96 DPI), 125% (120 DPI) e 150% (144 DPI). O usuário também pode aplicar uma configuração personalizada. A partir do Windows 7, o DPI é uma configuração por usuário.

Dimensionamento de DWM

Se um programa não levar em conta o DPI, os seguintes defeitos poderão ser aparentes nas configurações de alto DPI:

- Elementos de interface do usuário recortados.
- Layout incorreto.
- Bitmaps e ícones pixelados.
- Coordenadas incorretas do mouse, que podem afetar o teste de clique, arrastar e soltar e assim por diante.

Para garantir que os programas mais antigos funcionem em configurações de alto DPI, o DWM implementa um fallback útil. Se um programa não estiver marcado como com reconhecimento de DPI, o DWM dimensionará toda a interface do usuário para corresponder à configuração do DPI. Por exemplo, em 144 DPI, a interface do usuário é dimensionada em 150%, incluindo texto, elementos gráficos, controles e tamanhos de

janela. Se o programa criar uma janela de 500 × 500, a janela será exibida como 750 × 750 pixels e o conteúdo da janela será dimensionado adequadamente.

Esse comportamento significa que os programas mais antigos "apenas funcionam" em configurações de alto DPI. No entanto, o dimensionamento também resulta em uma aparência um pouco desfocada, pois o dimensionamento é aplicado depois que a janela é desenhada.

Aplicativos com reconhecimento de DPI

Para evitar o dimensionamento do DWM, um programa pode se marcar como com reconhecimento de DPI. Isso informa ao DWM para não executar nenhum dimensionamento automático de DPI. Todos os novos aplicativos devem ser projetados para serem com reconhecimento de DPI, pois o reconhecimento de DPI melhora a aparência da interface do usuário em configurações de DPI mais altas.

Um programa se declara com reconhecimento de DPI por meio do manifesto do aplicativo. Um *manifesto* é simplesmente um arquivo XML que descreve uma DLL ou um aplicativo. O manifesto normalmente é inserido no arquivo executável, embora possa ser fornecido como um arquivo separado. Um manifesto contém informações como dependências de DLL, o nível de privilégio solicitado e para qual versão do Windows o programa foi projetado.

Para declarar que seu programa tem reconhecimento de DPI, inclua as informações a seguir no manifesto.

syntax

```
<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>
```

A listagem mostrada aqui é apenas um manifesto parcial, mas o vinculador do Visual Studio gera o restante do manifesto para você automaticamente. Para incluir um manifesto parcial em seu projeto, execute as etapas a seguir no Visual Studio.

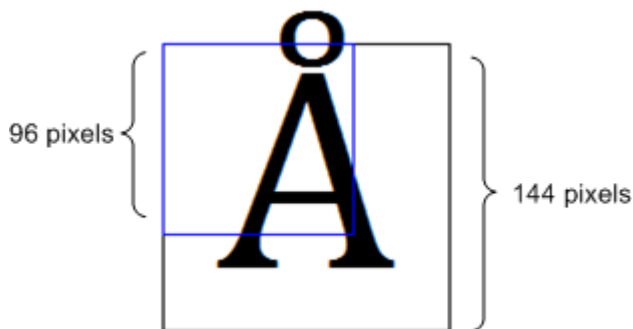
1. No menu **Projeto**, clique em **Propriedade**.

2. No painel esquerdo, expanda **Propriedades de Configuração**, expanda **Ferramenta de Manifesto** e clique em **Entrada e Saída**.
3. Na caixa de texto **Arquivos de Manifesto Adicionais**, digite o nome do arquivo de manifesto e clique em **OK**.

Ao marcar seu programa como com reconhecimento de DPI, você está dizendo ao DWM para não dimensionar a janela do aplicativo. Agora, se você criar uma janela de 500 × 500, a janela ocupará 500 × 500 pixels, independentemente da configuração de DPI do usuário.

GDI e DPI

O desenho GDI é medido em pixels. Isso significa que, se o programa estiver marcado como com reconhecimento de DPI e você pedir à GDI para desenhar um retângulo de 200 × 100, o retângulo resultante terá 200 pixels de largura e 100 pixels de altura na tela. No entanto, os tamanhos de fonte GDI são dimensionados para a configuração atual do DPI. Em outras palavras, se você criar uma fonte de 72 pontos, o tamanho da fonte será de 96 pixels a 96 DPI, mas 144 pixels a 144 DPI. Aqui está uma fonte de 72 pontos renderizada em 144 DPI usando GDI.



Se o aplicativo tiver reconhecimento de DPI e você usar a GDI para desenhar, dimensione todas as coordenadas de desenho para corresponder ao DPI.

Direct2D e DPI

Direct2D executa o dimensionamento automaticamente para corresponder à configuração de DPI. Em Direct2D, as coordenadas são medidas em unidades chamadas *DIPs* (*pixels independentes de dispositivo*). Um DIP é definido como 1/96 de polegada lógica. Em Direct2D, todas as operações de desenho são especificadas em DIPs e, em seguida, dimensionadas para a configuração de DPI atual.

Configuração de DPI	Tamanho do DIP
---------------------	----------------

Configuração de DPI	Tamanho do DIP
96	1 pixel
120	1,25 pixels
144	1,5 pixels

Por exemplo, se a configuração de DPI do usuário for 144 DPI e você solicitar que Direct2D desenhe um retângulo de 200 × 100, o retângulo terá 300 × 150 pixels físicos. Além disso, DirectWrite mede tamanhos de fonte em DIPs, em vez de pontos. Para criar uma fonte de 12 pontos, especifique 16 DIPs (12 pontos = 1/6 polegada lógica = 96/6 DIPs). Quando o texto é desenhado na tela, Direct2D converte os DIPs em pixels físicos. O benefício desse sistema é que as unidades de medida são consistentes para texto e desenho, independentemente da configuração atual do DPI.

Uma palavra de cuidado: as coordenadas do mouse e da janela ainda são fornecidas em pixels físicos, não em DIPs. Por exemplo, se você processar a mensagem `WM_LBUTTONDOWN`, a posição do mouse para baixo será fornecida em pixels físicos. Para desenhar um ponto nessa posição, você deve converter as coordenadas de pixel em DIPs.

Convertendo pixels físicos em DIPs

O valor base do DPI é definido como , que é definido como `USER_DEFAULT_SCREEN_DPI` 96. Para determinar o fator de dimensionamento, use o valor de DPI e divida por `USER_DEFAULT_SCREEN_DPI`.

A conversão de pixels físicos em DIPs usa a fórmula a seguir.

```
DIPs = pixels / (DPI / USER_DEFAULT_SCREEN_DPI)
```

Para obter a configuração de DPI, chame a função `GetDpiForWindow`. O DPI é retornado como um valor de ponto flutuante. Calcule o fator de dimensionamento para ambos os eixos.

C++

```
float g_DPIScale = 1.0f;

void InitializeDPIScale(HWND hwnd)
{
    float dpi = GetDpiForWindow(hwnd);
    g_DPIScale = dpi / USER_DEFAULT_SCREEN_DPI;
}
```

```

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

template <typename T>
float PixelsToDips(T y)
{
    return static_cast<float>(y) / g_DPIScale;
}

```

Aqui está uma maneira alternativa de obter a configuração de DPI se você não estiver usando Direct2D:

C++

```

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

```

ⓘ Observação

Recomendamos que, para um aplicativo da área de trabalho, você use **GetDpiForWindow**; e para um aplicativo Plataforma Universal do Windows (UWP), use **DisplayInformation::LogicalDpi**. Embora não o recomendemos, é possível definir o reconhecimento de DPI padrão programaticamente usando **SetProcessDpiAwarenessContext**. Depois que uma janela (um HWND) tiver sido criada em seu processo, não haverá mais suporte para a alteração do modo de reconhecimento de DPI. Se você estiver definindo o modo de reconhecimento de DPI padrão do processo programaticamente, deverá chamar a API correspondente antes que os HWNDs tenham sido criados. Para obter mais informações, consulte [Configurando o reconhecimento de DPI padrão para um processo](#).

Redimensionando o destino de renderização

Se o tamanho da janela for alterado, você deverá redimensionar o destino de renderização para corresponder. Na maioria dos casos, você também precisará atualizar o layout e repintar a janela. O código a seguir mostra essas etapas.

C++

```
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);
    }
}
```

A função [GetClientRect](#) obtém o novo tamanho da área do cliente, em pixels físicos (não em DIPs). O método [ID2D1HwndRenderTarget::Resize](#) atualiza o tamanho do destino de renderização, também especificado em pixels. A função [InvalidateRect](#) força uma repinta adicionando toda a área do cliente à região de atualização da janela. (Consulte [Pintando a janela](#), no Módulo 1.)

À medida que a janela cresce ou encolhe, você normalmente precisará recalcular a posição dos objetos que desenha. Por exemplo, no programa de círculo, o raio e o ponto central devem ser atualizados:

C++

```
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);
    }
}
```

O método [ID2D1RenderTarget::GetSize](#) retorna o tamanho do destino de renderização em DIPs (não pixels), que é a unidade apropriada para calcular o layout. Há um método intimamente relacionado, [ID2D1RenderTarget::GetPixelSize](#), que retorna o tamanho em pixels físicos. Para um destino de renderização **HWND**, esse valor corresponde ao tamanho retornado por [GetClientRect](#). Mas lembre-se de que o desenho é executado em DIPs, não em pixels.

Avançar









Usando cor em Direct2D

Usando Cor no Direct2D

Artigo • 13/06/2023

Direct2D usa o modelo de cores RGB, no qual as cores são formadas combinando valores diferentes de vermelho, verde e azul. Um quarto componente, alfa, mede a transparência de um pixel. Em Direct2D, cada um desses componentes é um valor de ponto flutuante com um intervalo de [0,0 1,0]. Para os três componentes de cor, o valor mede a intensidade da cor. Para o componente alfa, 0,0 significa completamente transparente e 1,0 significa completamente opaco. A tabela a seguir mostra as cores resultantes de várias combinações de 100% de intensidade.

Vermelho	Verde	Azul	Color
0	0	0	Preto
1	0	0	Vermelho
0	1	0	Verde
0	0	1	Azul
0	1	1	Ciano
1	0	1	Magenta
1	1	0	Amarelo
1	1	1	Branca

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

Valores de cor entre 0 e 1 resultam em diferentes tons dessas cores puras. Direct2D usa a estrutura `D2D1_COLOR_F` para representar cores. Por exemplo, o código a seguir especifica magenta.

C++

```
// Initialize a magenta color.  
  
D2D1_COLOR_F clr;  
clr.r = 1;  
clr.g = 0;
```

```
clr.b = 1;  
clr.a = 1; // Opaque.
```

Você também pode especificar uma cor usando a classe `D2D1::ColorF`, que deriva da estrutura `D2D1_COLOR_F`.

C++

```
// Equivalent to the previous example.  
  
D2D1::ColorF clr(1, 0, 1, 1);
```

Mesclagem Alfa

A mesclagem alfa cria áreas translúcidas mesclando a cor de primeiro plano com a cor da tela de fundo, usando a fórmula a seguir.

$$\text{color} = \text{af} * \text{Cf} + (1 - \text{af}) * \text{Cb}$$

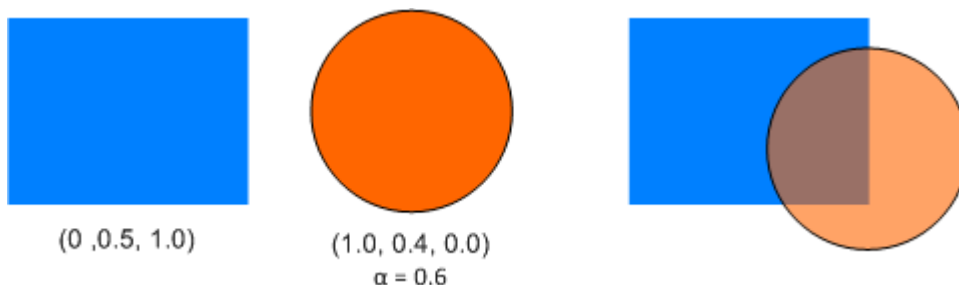
em que C_b é a cor da tela de fundo, C_f é a cor de primeiro plano e af é o valor alfa da cor de primeiro plano. Essa fórmula é aplicada em par a cada componente de cor. Por exemplo, suponha que a cor de primeiro plano seja ($R = 1,0$, $G = 0,4$, $B = 0,0$), com alfa = $0,6$ e a cor da tela de fundo seja ($R = 0,0$, $G = 0,5$, $B = 1,0$). A cor combinada alfa resultante é:

$$R = (1,0 * 0,6 + 0 * 0,4) = .6$$

$$G = (0,4 * 0,6 + 0,5 * 0,4) = .44$$

$$B = (0 * 0,6 + 1,0 * 0,4) = .40$$

A imagem a seguir mostra o resultado dessa operação de mesclagem.



Formatos de pixel

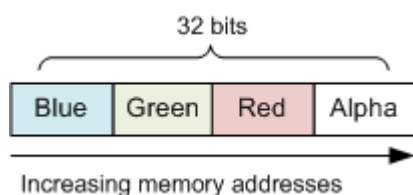
A estrutura `D2D1_COLOR_F` não descreve como um pixel é representado na memória. Na maioria dos casos, isso não importa. Direct2D manipula todos os detalhes internos da conversão de informações de cor em pixels. Mas talvez seja necessário saber o

formato de pixel se você estiver trabalhando diretamente com um bitmap na memória ou se combinar Direct2D com Direct3D ou GDI.

A [enumeração DXGI_FORMAT](#) define uma lista de formatos de pixel. A lista é bastante longa, mas apenas algumas delas são relevantes para Direct2D. (Os outros são usados pelo Direct3D).

Formato de pixel	Descrição
<code>DXGI_FORMAT_B8G8R8A8_UNORM</code>	Esse é o formato de pixel mais comum. Todos os componentes de pixel (vermelho, verde, azul e alfa) são inteiros sem sinal de 8 bits. Os componentes são organizados na ordem <i>BGRA</i> na memória. (Veja a ilustração a seguir.)
<code>DXGI_FORMAT_R8G8B8A8_UNORM</code>	Os componentes de pixel são inteiros sem sinal de 8 bits, na ordem <i>RGBA</i> . Em outras palavras, os componentes vermelho e azul são trocados, em relação a <code>DXGI_FORMAT_B8G8R8A8_UNORM</code> . Esse formato tem suporte apenas para dispositivos de hardware.
<code>DXGI_FORMAT_A8_UNORM</code>	Esse formato contém um componente alfa de 8 bits, sem componentes RGB. É útil para criar máscaras de opacidade. Para ler mais sobre como usar máscaras de opacidade no Direct2D, consulte Visão geral de destinos de renderização A8 compatíveis .

A ilustração a seguir mostra o layout de pixel BGRA.



Para obter o formato de pixel de um destino de renderização, chame `ID2D1RenderTarget::GetPixelFormat`. O formato de pixel pode não corresponder à resolução de exibição. Por exemplo, a exibição pode ser definida como cor de 16 bits, mesmo que o destino de renderização use a cor de 32 bits.

Modo Alfa

Um destino de renderização também tem um modo alfa, que define como os valores alfa são tratados.

Modo alfa	Descrição
-----------	-----------

Modo alfa	Descrição
D2D1_ALPHA_MODE_IGNORE	Nenhuma mesclagem alfa é executada. Os valores alfa são ignorados.
D2D1_ALPHA_MODE_STRAIGHT	Alfa reto. Os componentes de cor do pixel representam a intensidade da cor antes da mesclagem alfa.
D2D1_ALPHA_MODE_PREMULTIPLIED	Alfa pré-multiplicado. Os componentes de cor do pixel representam a intensidade de cor multiplicada pelo valor alfa. Esse formato é mais eficiente de renderizar do que alfa reto, pois o termo (af Cf) da fórmula de mistura alfa é pré-computado. No entanto, esse formato não é apropriado para armazenar em um arquivo de imagem.

Aqui está um exemplo da diferença entre alfa reto e alfa pré-multiplicado. Suponha que a cor desejada seja vermelho puro (100% de intensidade) com 50% de alfa. Como um tipo Direct2D, essa cor seria representada como (1, 0, 0, 0,5). Usando alfa reto e supondo componentes de cor de 8 bits, o componente vermelho do pixel é 0xFF. Usando alfa pré-multiplicado, o componente vermelho é dimensionado em 50% para igual a 0x80.

O tipo [de dados D2D1_COLOR_F](#) sempre representa cores usando alfa reto. Direct2D converte pixels em formato alfa pré-multiplicado, se necessário.

Se você souber que seu programa não executará nenhuma mesclagem alfa, crie o destino de renderização com o modo alfa **D2D1_ALPHA_MODE_IGNORE**. Esse modo pode melhorar o desempenho, pois Direct2D pode ignorar os cálculos alfa. Para obter mais informações, consulte [Aprimorando o desempenho de aplicativos de Direct2D](#).

Avançar

[Aplicar transformações em Direct2D](#)

Comentários

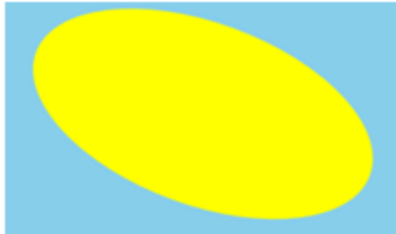
Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

Aplicar transformações em Direct2D

Artigo • 13/06/2023

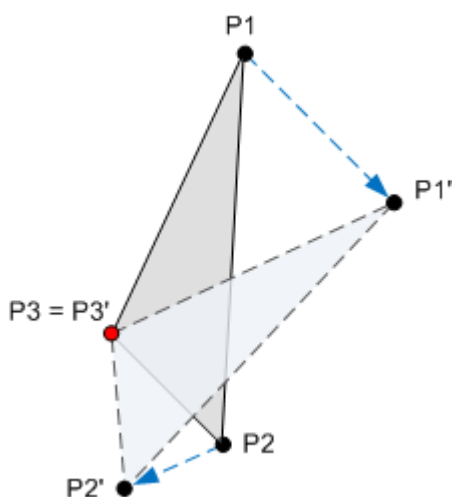
Em [Desenho com Direct2D](#), vimos que o método `ID2D1RenderTarget::FillEllipse` desenha uma elipse alinhada aos eixos x e y. Mas suponha que você queira desenhara uma elipse inclinada em um ângulo?



Usando transformações, você pode alterar uma forma das seguintes maneiras.

- Rotação em torno de um ponto.
- Dimensionamento.
- Tradução (deslocamento na direção X ou Y).
- Distorção (também conhecida como *tesoura*).

Uma transformação é uma operação matemática que mapeia um conjunto de pontos para um novo conjunto de pontos. Por exemplo, o diagrama a seguir mostra um triângulo girado ao redor do ponto P3. Depois que a rotação é aplicada, o ponto P1 é mapeado para P1', o ponto P2 é mapeado para P2' e o ponto P3 é mapeado para si mesmo.



As transformações são implementadas usando matrizes. No entanto, você não precisa entender a matemática das matrizes para usá-las. Se você quiser saber mais sobre a matemática, confira [Apêndice: Transformações de Matriz](#).

Para aplicar uma transformação em Direct2D, chame o método **ID2D1RenderTarget::SetTransform**. Esse método usa uma estrutura **D2D1_MATRIX_3X2_F** que define a transformação. Você pode inicializar essa estrutura chamando métodos na classe **D2D1::Matrix3x2F**. Essa classe contém métodos estáticos que retornam uma matriz para cada tipo de transformação:

- **Matrix3x2F::Rotation**
- **Matrix3x2F::Scale**
- **Matrix3x2F::Translation**
- **Matrix3x2F::Skew**

Por exemplo, o código a seguir aplica uma rotação de 20 graus ao redor do ponto (100, 100).

C++

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

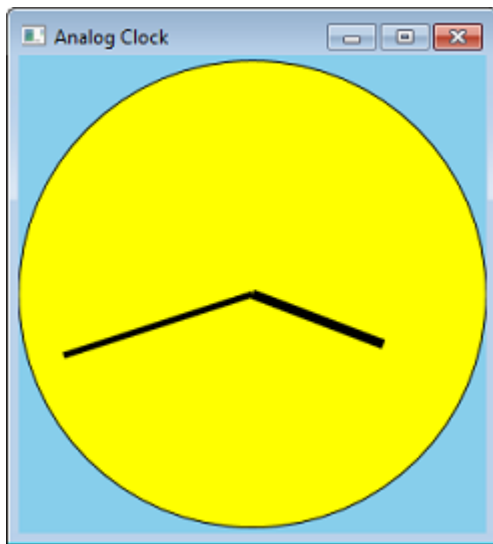
A transformação é aplicada a todas as operações de desenho posteriores até que você chame **SetTransform** novamente. Para remover a transformação atual, chame **SetTransform** com a matriz de identidade. Para criar a matriz de identidade, chame a função **Matrix3x2F::Identity**.

C++

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

Desenhando mãos do relógio

Vamos colocar transformações a serem usadas convertendo nosso programa Circle em um relógio analógico. Podemos fazer isso adicionando linhas para as mãos.



Em vez de calcular as coordenadas das linhas, podemos calcular o ângulo e aplicar uma transformação de rotação. O código a seguir mostra uma função que desenha uma mão de relógio. O parâmetro *fAngle* fornece o ângulo da mão, em graus.

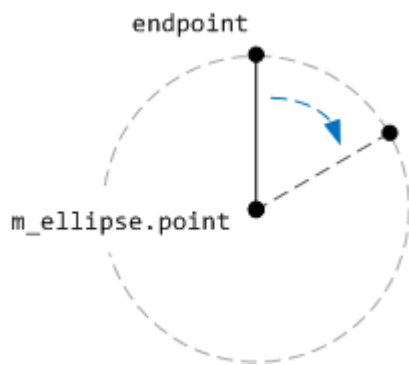
C++

```
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);
}
```

Esse código desenha uma linha vertical, começando do centro da face do relógio e terminando no *ponto de extremidadePoint*. A linha é girada ao redor do centro da elipse aplicando uma transformação de rotação. O ponto central da rotação é o centro da elipse que forma a face do relógio.



O código a seguir mostra como toda a face do relógio é desenhada.

C++

```
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() );
}
```

Você pode baixar o projeto completo do Visual Studio [Direct2D Exemplo de Relógio](#). (Só por diversão, a versão de download adiciona um gradiente radial à face do relógio.)

Combinando transformações

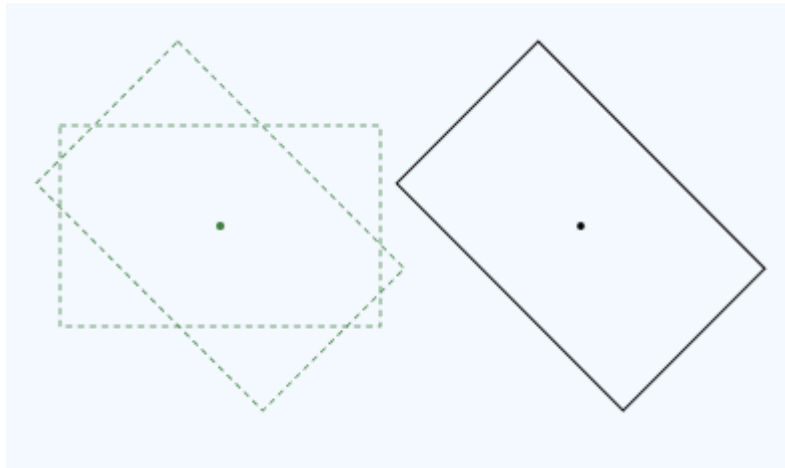
As quatro transformações básicas podem ser combinadas multiplicando duas ou mais matrizes. Por exemplo, o código a seguir combina uma rotação com uma tradução.

C++

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

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

A classe `Matrix3x2F` fornece `operator*()` para multiplicação de matriz. A ordem na qual você multiplica as matrizes é importante. Definir uma transformação ($M \times N$) significa "Aplicar M primeiro, seguido por N". Por exemplo, aqui está a rotação seguida pela tradução:

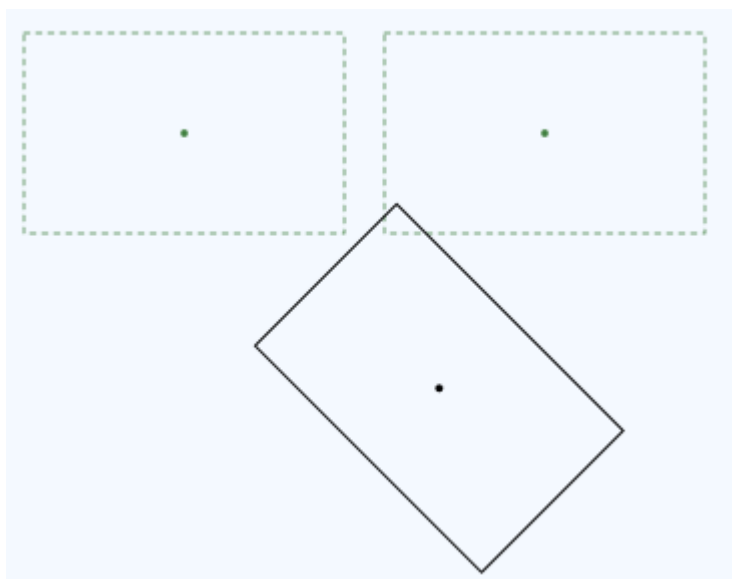


Este é o código para esta transformação:

C++

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

Agora compare essa transformação com uma transformação na ordem inversa, tradução seguida de rotação.



A rotação é executada ao redor do centro do retângulo original. Este é o código para essa transformação.

C++

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

Como você pode ver, as matrizes são as mesmas, mas a ordem das operações foi alterada. Isso acontece porque a multiplicação de matriz não é comutativa: $M \times N \neq N \times M$.

Avançar

[Apêndice: Transformações de Matriz](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Apêndice: Transformações de Matriz

Artigo • 13/06/2023

Este tópico fornece uma visão geral matemática das transformações de matriz para gráficos 2D. No entanto, você não precisa saber matemática de matriz para usar transformações em Direct2D. Leia este tópico se você estiver interessado em matemática; caso contrário, fique à vontade para ignorar este tópico.

- [Introdução às Matrizes](#)
 - [Operações de matriz](#)
- [Transformações de Affine](#)
 - [Transformação de Tradução](#)
 - [Transformação de dimensionamento](#)
 - [Rotação ao redor da origem](#)
 - [Rotação em torno de um ponto arbitrário](#)
 - [Transformação distorção](#)
- [Representando transformações em Direct2D](#)
- [Próximo](#)

Introdução às Matrizes

Uma matriz é uma matriz retangular de números reais. A *ordem* da matriz é o número de linhas e colunas. Por exemplo, se a matriz tiver 3 linhas e 2 colunas, a ordem será 3 × 2. As matrizes geralmente são mostradas com os elementos de matriz entre colchetes:

$$\begin{bmatrix} 2 & 0 \\ 1.5 & 1 \\ 4 & -0.3 \end{bmatrix}$$

Notação: uma matriz é designada por uma letra maiúscula. Os elementos são designados por letras minúsculas. Subscritos indicam o número de linha e coluna de um elemento. Por exemplo, a_{ij} é o elemento na i 'th linha e j 'th coluna da matriz A .

O diagrama a seguir mostra uma matriz $i \times j$, com os elementos individuais em cada célula da matriz.

	i × j			
Row 1	a_{1,1}	a_{1,2}	...	a_{1,j}
	a_{2,1}	a_{2,2}	...	a_{2,j}
	⋮	⋮		
Row i	a_{i,1}	a_{i,2}	...	a_{i,j}
	Column 1			Column j

Operações de matriz

Esta seção descreve as operações básicas definidas em matrizes.

Adição. A soma $A + B$ de duas matrizes é obtida adicionando os elementos correspondentes de A e B:

$$A + B = [a_{ij}] + [b_{ij}] = [a_{ij} + b_{ij}]$$

Multiplicação escalar. Essa operação multiplica uma matriz por um número real. Dado um número real k , o kA do produto escalar é obtido multiplicando cada elemento de A por k .

$$kA = k[a_{ij}] = [k \times a_{ij}]$$

Multiplicação de matriz. Considerando duas matrizes A e B com ordem $(m \times n)$ e $(n \times p)$, o produto $C = A \times B$ é uma matriz com ordem $(m \times p)$, definida da seguinte maneira:

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

ou, de forma equivalente:

$$c_{ij} = a_{i1} \times b_{1j} + a_{i2} \times b_{2j} + \dots + a_{in} \times b_{nj}$$

Ou seja, para calcular cada elemento c_{ij} , faça o seguinte:

1. Pegue a i^{a} linha de A e a coluna j^{th} de B.
2. Multiplique cada par de elementos na linha e coluna: a primeira entrada de linha pela primeira entrada de coluna, a segunda entrada de linha pela segunda entrada de coluna e assim por diante.
3. Somar o resultado.

Aqui está um exemplo de multiplicação de uma matriz (2×2) por uma matriz (2×3) .

$$\begin{vmatrix} -2 & 4 \\ 0 & -1 \end{vmatrix} \begin{vmatrix} 1 & 2 & 3 \\ 8 & -5 & 6 \end{vmatrix} = \begin{vmatrix} -2(1) + 4(8) & -2(2) + 4(-5) & -2(3) + 4(6) \\ 0(1) + (-1)(8) & 0(2) + (-1)(-5) & 0(3) + (-1)(6) \end{vmatrix} = \begin{vmatrix} 30 & -24 & 18 \\ -8 & 5 & -6 \end{vmatrix}$$

A multiplicação de matriz não é comutativa. Ou seja, $A \times B \neq B \times A$. Além disso, na definição a seguir, nem todos os pares de matrizes podem ser multiplicados. O número de colunas na matriz à esquerda deve ser igual ao número de linhas na matriz à direita. Caso contrário, o operador \times não será definido.

Identificar matriz. Uma matriz de identidade, designada como I , é uma matriz quadrada definida da seguinte maneira:

$I_{ij} = 1$ se $i = j$, ou 0 caso contrário.

Em outras palavras, uma matriz de identidade contém 1 para cada elemento em que o número da linha é igual ao número da coluna e zero para todos os outros elementos. Por exemplo, aqui está a matriz de identidade 3×3 .

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

As igualdades a seguir são para qualquer matriz M .

$$M \times I = M \quad I \times M = M$$

Transformações de Affine

Uma *transformação afim* é uma operação matemática que mapeia um espaço de coordenadas para outro. Em outras palavras, ele mapeia um conjunto de pontos para outro conjunto de pontos. As transformações de afim têm alguns recursos que as tornam úteis em elementos gráficos de computação.

- Transformações afim preservam a *colinearidade*. Se três ou mais pontos caírem em uma linha, eles ainda formarão uma linha após a transformação. Linhas retas permanecem retas.
- A composição de duas transformações afim é uma transformação afim.

As transformações de afim para o espaço 2D têm o seguinte formato.

$$M = \begin{vmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{vmatrix}$$

Se você aplicar a definição de multiplicação de matriz fornecida anteriormente, poderá mostrar que o produto de duas transformações de afim é outra transformação afim.

Para transformar um ponto 2D usando uma transformação affine, o ponto é representado como uma matriz de 1×3 .

$$P = \begin{bmatrix} x & y & 1 \end{bmatrix}$$

Os dois primeiros elementos contêm as coordenadas x e y do ponto. O 1 é colocado no terceiro elemento para que a matemática funcione corretamente. Para aplicar a transformação, multiplique as duas matrizes da seguinte maneira.

$$P' = P \times M$$

Isso se expande para o seguinte.

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ e & f & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

onde

$$x' = ax + cy + e \text{ e } y' = bx + dy + f$$

Para obter o ponto transformado, use os dois primeiros elementos da matriz P' .

$$p = (x', y') = (ax + cy + e, bx + dy + f)$$

ⓘ Observação

Uma matriz de $1 \times n$ é chamada de *vetor de linha*. Direct2D e Direct3D usam vetores de linha para representar pontos no espaço 2D ou 3D. Você pode obter um resultado equivalente usando um vetor de coluna ($n \times 1$) e transpondo a matriz de transformação. A maioria dos textos gráficos usa o formulário de vetor de coluna. Este tópico apresenta o formulário de vetor de linha para consistência com Direct2D e Direct3D.

As próximas seções derivam as transformações básicas.

Transformação de Tradução

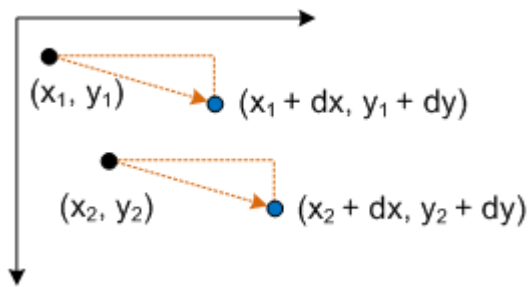
A matriz de transformação de tradução tem o seguinte formato.

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{bmatrix}$$

Conectar um ponto P a esta equação gera:

$$P' = (x^* + dx^*, y^* + dy^*)$$

que corresponde ao ponto (x, y) traduzido por dx no eixo X e dy no eixo Y.



Transformação de dimensionamento

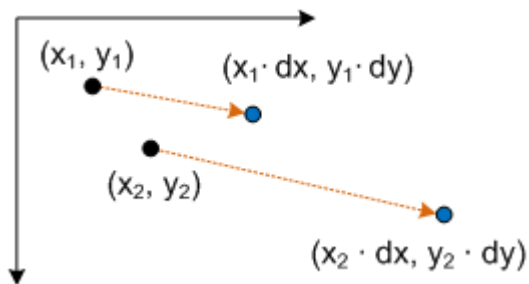
A matriz de transformação de dimensionamento tem a seguinte forma.

$$S = \begin{vmatrix} dx & 0 & 0 \\ 0 & dy & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Conectar um ponto P a esta equação gera:

$$P' = (x^* \cdot dx^*, y^* \cdot dy^*)$$

que corresponde ao ponto (x,y) dimensionado por dx e dy .



Rotação ao redor da origem

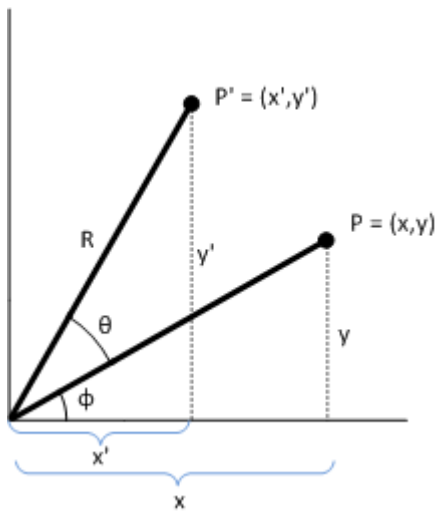
A matriz para girar um ponto ao redor da origem tem a seguinte forma.

$$R = \begin{vmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & -\cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

O ponto transformado é:

$$P' = (x^* \cos\theta - y^* \sin\theta, x^* \sin\theta + y^* \cos\theta)$$

Prova. Para mostrar que P' representa uma rotação, considere o diagrama a seguir.



Considerando:

$P = (x, y)$

O ponto original a ser transformado.

Φ

O ângulo formado pela linha (0,0) a P.

Θ

O ângulo pelo qual girar (x,y) sobre a origem.

$P' = (x', y')$

O ponto transformado.

R

O comprimento da linha (0,0) a P. Além disso, o raio do círculo de rotação.

⚠ Observação

Este diagrama usa o sistema de coordenadas padrão usado na geometria, em que o eixo y positivo aponta para cima. Direct2D usa o sistema de coordenadas do Windows, em que o eixo y positivo aponta para baixo.

O ângulo entre o eixo x e a linha (0,0) para P' é $\Phi + \Theta$. As seguintes identidades contêm:

$$x = R \cos\Phi \quad y = R \sin\Phi \quad x' = R \cos(\Phi + \Theta) \quad y' = R \sin(\Phi + \Theta)$$

Agora resolva para x' e y' em termos de Θ . Pelas fórmulas de adição trigonométrica:

$$x' = R(\cos\Phi\cos\Theta - \sin\Phi\sin\Theta) = R\cos\Phi\cos\Theta - R\sin\Phi\sin\Theta \quad y' = R(\sin\Phi\cos\Theta + \cos\Phi\sin\Theta) = R\sin\Phi\cos\Theta + R\cos\Phi\sin\Theta$$

Substituindo, obtemos:

$$x' = x\cos\Theta - y\sin\Theta \quad y' = x\sin\Theta + y\cos\Theta$$

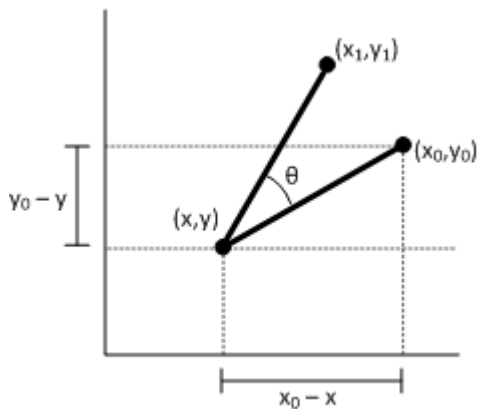
que corresponde ao ponto transformado P' mostrado anteriormente.

Rotação em torno de um ponto arbitrário

Para girar em torno de um ponto (x,y) diferente da origem, a matriz a seguir é usada.

$$R = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & -\cos\theta & 0 \\ x(1 - \cos\theta) + y(\sin\theta) & x(-\sin\theta) + y(1 - \cos\theta) & 1 \end{bmatrix}$$

Você pode derivar essa matriz usando o ponto (x,y) para ser a origem.



Let (x1, y1) be the point that results from rotating the point (x0, y0) around the point (x, y). Podemos derivar x1 da seguinte maneira.

$$x1 = (x0 - x)\cos\Theta - (y0 - y)\sin\Theta + x \quad x1 = x\cos\Theta - y0\sin\Theta + [(1 - \cos\Theta)x + y\sin\Theta]$$

Agora conecte essa equação de volta à matriz de transformação, usando a fórmula $x1 = ax0 + cy0 + e$ de antes. Use o mesmo procedimento para derivar y1.

Transformação distorção

A transformação de distorção é definida por quatro parâmetros:

- Θ : a quantidade a ser distorcida ao longo do eixo x, medida como um ângulo do eixo y.
- Φ : a quantidade a ser distorcida ao longo do eixo y, medida como um ângulo do eixo x.
- (px, py) : as coordenadas x e y do ponto sobre o qual a distorção é executada.

A transformação de distorção usa a matriz a seguir.

$$\begin{bmatrix} 1 & \tan\phi & 0 \\ \tan\theta & 1 & 0 \\ -p_y\tan\theta & -p_x\tan\phi & 1 \end{bmatrix}$$

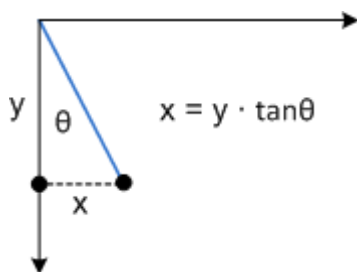
O ponto transformado é:

$$P' = (x^* + y^*\tan\theta - p_y\tan\theta, y^* + x^*\tan\phi) - p_y\tan\phi$$

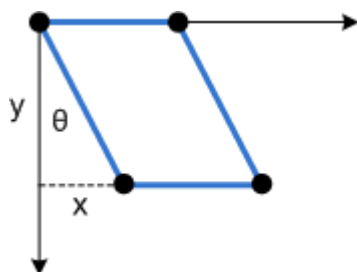
ou de forma equivalente:

$$P' = (x^* + (y^* - p_y)\tan\theta, y^* + (x^* - p_x)\tan\phi)$$

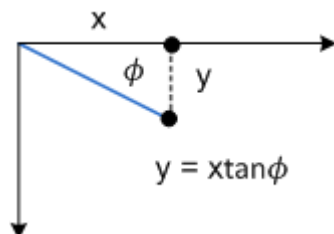
Para ver como essa transformação funciona, considere cada componente individualmente. O parâmetro θ move cada ponto na direção x em uma quantidade igual a $\tan\theta$. O diagrama a seguir mostra a relação entre θ e a distorção do eixo x.



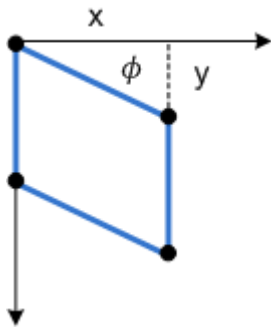
Aqui está a mesma distorção aplicada a um retângulo:



O parâmetro ϕ tem o mesmo efeito, mas ao longo do eixo y:



O próximo diagrama mostra a distorção do eixo y aplicada a um retângulo.



Por fim, os parâmetros px e py deslocam o ponto central para a distorção ao longo dos eixos x e y .

Representando transformações em Direct2D

Todas as transformações Direct2D são transformações afim. Direct2D não dá suporte a transformações não afim. As transformações são representadas pela estrutura `D2D1_MATRIX_3X2_F`. Essa estrutura define uma matriz de 3×2 . Como a terceira coluna de uma transformação afim é sempre a mesma $([0, 0, 1])$ e, como Direct2D não dá suporte a transformações não afim, não é necessário especificar a matriz inteira de 3×3 . Internamente, Direct2D usa três matrizes 3×3 para calcular as transformações.

Os membros do `D2D1_MATRIX_3X2_F` são nomeados de acordo com sua posição de índice: o membro `_11` é o elemento $(1,1)$, o membro `_12` é o elemento $(1,2)$ e assim por diante. Embora você possa inicializar os membros da estrutura diretamente, é recomendável usar a classe `D2D1::Matrix3x2F`. Essa classe herda `D2D1_MATRIX_3X2_F` e fornece métodos auxiliares para criar qualquer uma das transformações básicas de affine. A classe também define `operator*()` para compor duas ou mais transformações, conforme descrito em [Aplicando transformações em Direct2D](#).

Avançar

[Módulo 4. Entrada do usuário](#)

Comentários

Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

Módulo 4. Entrada do usuário

Artigo • 13/06/2023

Os módulos anteriores exploraram a criação de uma janela, o tratamento de mensagens de janela e o desenho básico com gráficos 2D. Neste módulo, examinaremos a entrada do mouse e do teclado. Ao final deste módulo, você poderá escrever um programa de desenho simples que usa o mouse e o teclado.

Nesta seção

- [Entrada por mouse](#)
- [Respondendo a cliques do mouse](#)
- [Movimento do mouse](#)
- [Operações diversas do mouse](#)
- [Entrada por teclado](#)
- [Tabelas de aceleradores](#)
- [Definindo a imagem do cursor](#)
- [Entrada do usuário: exemplo estendido](#)

Comentários

Esta página foi útil?

 Yes

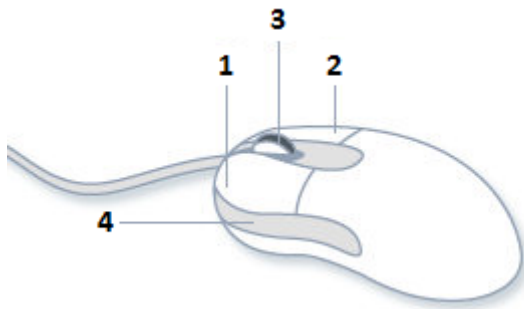
 No

[Obter ajuda no Microsoft Q&A](#)

Entrada do mouse (introdução a Win32 e C++)

Artigo • 07/08/2024

O Windows oferece suporte a mouses com até cinco botões: esquerdo, central e direito, além de dois botões adicionais chamados XBUTTON1 e XBUTTON2.



A maioria dos mouses para o Windows tem pelo menos os botões esquerdo e direito. O botão esquerdo do mouse é usado para apontar, selecionar, arrastar e assim por diante. O botão direito do mouse normalmente exibe um menu de contexto. Alguns mouses têm uma roda de rolagem localizada entre os botões esquerdo e direito. Dependendo do mouse, a roda de rolagem também pode ser clicável, tornando-a o botão do meio.

Os botões XBUTTON1 e XBUTTON2 geralmente estão localizados nas laterais do mouse, perto da base. Esses botões extras não estão presentes em todos os mouses. Se estiverem presentes, os botões XBUTTON1 e XBUTTON2 geralmente são mapeados para uma função do aplicativo, como a navegação para frente e para trás em um navegador da Web.

Os usuários canhotos geralmente acham mais confortável trocar as funções dos botões esquerdo e direito - usando o botão direito como ponteiro e o botão esquerdo para mostrar o menu de contexto. Por esse motivo, a documentação de ajuda do Windows usa os termos *botão primário* e *botão secundário*, que se referem à função lógica em vez do posicionamento físico. Na configuração padrão (destro), o botão esquerdo é o botão principal e o direito é o botão secundário. No entanto, os termos *clique com o botão direito* e *clique com o botão esquerdo* referem-se a ações lógicas. *Clicar com o botão esquerdo* significa clicar no botão principal, esteja esse botão fisicamente no lado direito ou esquerdo do mouse.

Independentemente de como o usuário configura o mouse, o Windows traduz automaticamente as mensagens do mouse para que sejam consistentes. O usuário pode trocar os botões primário e secundário no meio do uso do programa e isso não afetará o comportamento do programa.

Os termos *botão esquerdo* e *botão direito* às vezes são usados em vez de *botões primários* e *secundários*, respectivamente. Essa terminologia é consistente com os nomes das mensagens da janela para entrada do mouse. Lembre-se de que os botões físicos esquerdo e direito podem ser trocados.

Próximo

[Respondendo aos cliques do mouse](#)

Comentários

Esta página foi útil?



Yes




No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Resposta aos cliques do mouse

Artigo • 07/08/2024

Se o usuário clicar em um botão do mouse enquanto o cursor estiver sobre a área do cliente de uma janela, a janela receberá uma das seguintes mensagens.

 Expandir a tabela

Mensagem	Significado
WM_LBUTTONDOWN	Botão esquerdo para baixo
WM_LBUTTONUP	Botão esquerdo para cima
WM_MBUTTONDOWN	Botão do meio para baixo
WM_MBUTTONUP	Botão do meio para cima
WM_RBUTTONDOWN	Botão direito para baixo
WM_RBUTTONUP	Botão direito para cima
WM_XBUTTONDOWN	XBUTTONDOWN1 ou XBUTTONDOWN2 para baixo
WM_XBUTTONUP	XBUTTONDOWN1 ou XBUTTONDOWN2 para cima

Lembre-se de que a área do cliente é a parte da janela que exclui o quadro. Para obter mais informações sobre áreas do cliente, consulte [O que é uma janela?](#)

Coordenadas do mouse

Em todas essas mensagens, o parâmetro *lParam* contém as coordenadas x e y do ponteiro do mouse. Os 16 bits mais baixos de *lParam* contêm a coordenada x e os próximos 16 bits contêm a coordenada y. Use as macros [GET_X_LPARAM](#) e [GET_Y_LPARAM](#) para descompactar as coordenadas do *lParam*.

C++

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

Essas macros são definidas no arquivo de cabeçalho WindowsX.h.

No Windows de 64 bits *lParam* é o valor de 64 bits. Os 32 bits superiores de *lParam* não são usados. Onde a documentação do Windows menciona a "palavra de baixa ordem" e a "palavra de alta ordem" de *lParam*, o caso de 64 bits significa as palavras de baixa e alta ordem dos 32 bits inferiores. As macros extraem os valores corretos, portanto, se você usá-las, estará seguro.

As coordenadas do mouse são fornecidas em pixels, não em pixels independentes do dispositivo (DIPs), e são medidas em relação à área da janela do cliente. As coordenadas são valores assinados. As posições acima e à esquerda da área do cliente têm coordenadas negativas, o que é importante se você rastrear a posição do mouse fora da janela. Veremos como fazer isso em um tópico posterior, [Capturando o movimento do mouse fora da janela](#).

Sinalizadores adicionais

O parâmetro *wParam* contém um OR **bit a bit** de sinalizadores, indicando o estado dos outros botões do mouse mais as teclas SHIFT e CTRL.

 Expandir a tabela

Sinalizador	Significado
MK_CONTROL	A tecla CTRL está pressionada.
MK_LBUTTON	O botão esquerdo do mouse está pressionado.
MK_MBUTTON	O botão do meio do mouse está pressionado.
MK_RBUTTON	O botão direito do mouse está pressionado.
MK_SHIFT	A tecla SHIFT está pressionada.
MK_XBUTTON1	O botão XBUTTON1 está para baixo.
MK_XBUTTON2	O botão XBUTTON2 está para baixo.

A ausência de um sinalizador significa que o botão ou tecla correspondente não foi pressionado. Por exemplo, para testar se a tecla CTRL está para baixo:

C++

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

Se você precisar encontrar o estado de outras teclas além de CTRL e SHIFT, use a função [GetKeyState](#) que é descrita em [entrada por teclado](#).

As mensagens da janela [WM_XBUTTONDOWN](#) e [WM_XBUTTONUP](#) se aplicam tanto ao XBUTTON1 quanto ao XBUTTON2. O parâmetro *wParam* indica qual botão foi clicado.

C++

```
UINT button = GET_XBUTTON_WPARAM(wParam);
if (button == XBUTTON1)
{
    // XBUTTON1 was clicked.
}
else if (button == XBUTTON2)
{
    // XBUTTON2 was clicked.
}
```

Clique duplo

Uma janela não recebe notificações de clique duplo por padrão. Para receber cliques duplos, defina o sinalizador **CS_DBLCLKS** na estrutura [WNDCLASS](#) ao registrar a classe de janela.

C++

```
WNDCLASS wc = { };
wc.style = CS_DBLCLKS;

/* Set other structure members. */

RegisterClass(&wc);
```

Se você definir o sinalizador **CS_DBLCLKS** conforme mostrado, a janela receberá notificações de clique duplo. Um clique duplo é indicado por uma mensagem de janela com "DBLCLK" no nome. Por exemplo, um clique duplo no botão esquerdo do mouse produz a seguinte sequência de mensagens:

[WM_LBUTTONDOWN](#)

[WM_LBUTTONUP](#)

[WM_LBUTTONDBLCLK](#)

[WM_LBUTTONUP](#)

Na verdade, a segunda mensagem `WM_LBUTTONDOWN` que normalmente seria gerada, se torna uma mensagem `WM_LBUTTONDBLCLK`. Mensagens equivalentes são definidas para botões direito, meio e `XBUTTON`.

Até que você receba a mensagem de clique duplo, não há como saber que o primeiro clique do mouse é o início de um clique duplo. Portanto, uma ação de clique duplo deve continuar uma ação que começa com o primeiro clique do mouse. Por exemplo, no Shell do Windows, um único clique seleciona uma pasta, enquanto um clique duplo abre a pasta.

Mensagens de mouse não cliente

Um conjunto separado de mensagens é definido para eventos de mouse que ocorrem dentro da área não cliente da janela. Essas mensagens têm as letras "NC" no nome. Por exemplo, `WM_NCLBUTTONDOWN` é o equivalente não cliente de `WM_LBUTTONDOWN`. Um aplicativo típico não interceptará essas mensagens, pois a função `DefWindowProc` lida com essas mensagens corretamente. No entanto, eles podem ser úteis para certas funções avançadas. Por exemplo, você pode usar essas mensagens para implementar um comportamento personalizado na barra de título. Se você manipular essas mensagens, geralmente deverá passá-las para `DefWindowProc` posteriormente. Caso contrário, seu aplicativo interromperá a funcionalidade padrão, como arrastar ou minimizar a janela.

Próximo

[Movimento do mouse](#)

Comentários

Esta página foi útil?

 Yes

 No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Movimento do mouse

Artigo • 12/06/2023

Quando o mouse se move, o Windows posta uma [mensagem WM_MOUSEMOVE](#). Por padrão, `WM_MOUSEMOVE` vai para a janela que contém o cursor. Você pode substituir esse comportamento *capturando* o mouse, que é descrito na próxima seção.

A mensagem [WM_MOUSEMOVE](#) contém os mesmos parâmetros que as mensagens para cliques do mouse. Os 16 bits mais baixos de `lParam` contêm a coordenada x e os próximos 16 bits contêm a coordenada y. Use as macros [GET_X_LPARAM](#) e [GET_Y_LPARAM](#) para desempacotar as coordenadas do `lParam`. O parâmetro `wParam` contém um **OR** bit a bit de sinalizadores, indicando o estado dos outros botões do mouse mais as teclas SHIFT e CTRL. O código a seguir obtém as coordenadas do mouse do `lParam`.

C++

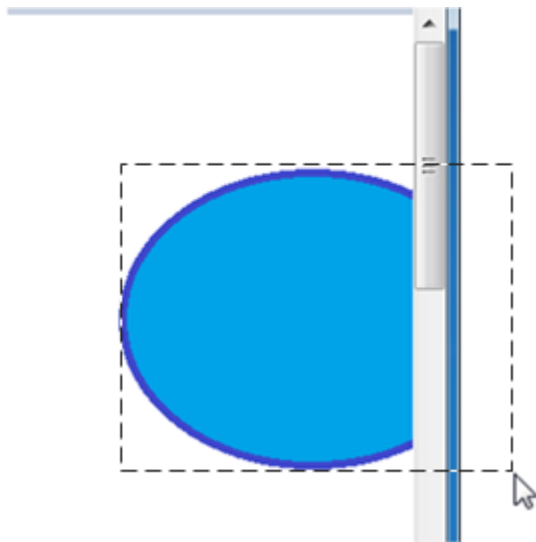
```
int xPos = GET_X_LPARAM(lParam);  
int yPos = GET_Y_LPARAM(lParam);
```

Lembre-se de que essas coordenadas estão em pixels, não em DIPs (pixels independentes de dispositivo). Posteriormente neste tópico, examinaremos o código que converte entre as duas unidades.

Uma janela também poderá receber uma mensagem [WM_MOUSEMOVE](#) se a posição do cursor for alterada em relação à janela. Por exemplo, se o cursor estiver posicionado sobre uma janela e o usuário ocultar a janela, a janela receberá `WM_MOUSEMOVE` mensagens mesmo que o mouse não tenha se movido. Uma consequência desse comportamento é que as coordenadas do mouse podem não mudar entre [mensagens WM_MOUSEMOVE](#).

Capturando o movimento do mouse fora da janela

Por padrão, uma janela para de receber [mensagens WM_MOUSEMOVE](#) se o mouse passar pela borda da área do cliente. Mas, para algumas operações, talvez seja necessário rastrear a posição do mouse além desse ponto. Por exemplo, um programa de desenho pode permitir que o usuário arraste o retângulo de seleção para além da borda da janela, conforme mostrado no diagrama a seguir.



Para receber mensagens de movimentação do mouse além da borda da janela, chame a função [SetCapture](#) . Depois que essa função for chamada, a janela continuará recebendo [WM_MOUSEMOVE](#) mensagens, desde que o usuário mantenha pelo menos um botão do mouse para baixo, mesmo que o mouse se mova para fora da janela. A janela de captura deve ser a janela em primeiro plano e apenas uma janela pode ser a janela de captura por vez. Para liberar a captura do mouse, chame a função [ReleaseCapture](#) .

Normalmente, você usaria [SetCapture](#) e [ReleaseCapture](#) da seguinte maneira.

1. Quando o usuário pressionar o botão esquerdo do mouse, chame [SetCapture](#) para começar a capturar o mouse.
2. Responder a mensagens de movimentação do mouse.
3. Quando o usuário liberar o botão esquerdo do mouse, chame [ReleaseCapture](#).

Exemplo: círculos de desenho

Vamos estender o programa Círculo do [Módulo 3](#) , permitindo que o usuário desenhe um círculo com o mouse. Comece com o programa [Direct2D Circle Sample](#).

Modificaremos o código neste exemplo para adicionar um desenho simples. Primeiro, adicione uma nova variável de membro à `MainWindow` classe .

C++

```
D2D1_POINT_2F ptMouse;
```

Essa variável armazena a posição do mouse para baixo enquanto o usuário arrasta o mouse. `MainWindow` No construtor, inicialize as variáveis *de ellipse* e *ptMouse*.

C++

```

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

```

Remova o corpo do `MainWindow::CalculateLayout` método; ele não é necessário para este exemplo.

C++

```

void CalculateLayout() { }

```

Em seguida, declare manipuladores de mensagens para o botão esquerdo para baixo, botão esquerdo para cima e mensagens de movimentação do mouse.

C++

```

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

```

As coordenadas do mouse são fornecidas em pixels físicos, mas Direct2D espera DIPs (pixels independentes de dispositivo). Para manipular as configurações de alto DPI corretamente, você deve converter as coordenadas de pixel em DIPs. Para obter mais discussões sobre DPI, consulte [DPI e pixels de Device-Independent](#). O código a seguir mostra uma classe auxiliar que converte pixels em DIPs.

C++

```

class DPIScale
{
    static float scale;

public:
    static void Initialize(HWND hwnd)
    {
        float dpi = GetDpiForWindow(hwnd);
        scale = dpi/96.0f;
    }

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

```

```
float DPIScale::scale = 1.0f;
```

Chame **DPIScale::Initialize** no manipulador **de WM_CREATE** depois de criar o objeto de fábrica Direct2D.

C++

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

Para obter as coordenadas do mouse em DIPs das mensagens do mouse, faça o seguinte:

1. Use as macros **GET_X_LPARAM** e **GET_Y_LPARAM** para obter as coordenadas de pixel. Essas macros são definidas no WindowsX.h, portanto, lembre-se de incluir esse cabeçalho em seu projeto.
2. Chame `DPIScale::PixelsToDips` para converter pixels em DIPs.

Agora, adicione os manipuladores de mensagens ao procedimento de janela.

C++

```
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;
```

Por fim, implemente os próprios manipuladores de mensagens.

Botão esquerdo para baixo

Para a mensagem de botão esquerdo para baixo, faça o seguinte:

1. Chame [SetCapture](#) para começar a capturar o mouse.
2. Armazene a posição do clique do mouse na variável *ptMouse* . Essa posição define o canto superior esquerdo da caixa delimitadora para a elipse.
3. Redefina a estrutura de elipse.
4. Chame [InvalidateRect](#). Essa função força a janela a ser repintada.

C++

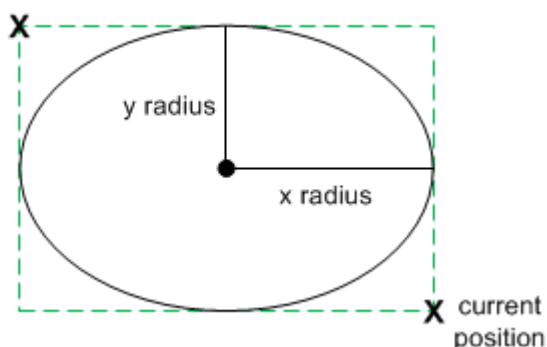
```
void MainWindow::OnLButtonDown(int pixelX, int pixelY, DWORD flags)
{
    SetCapture(m_hwnd);
    ellipse.point = ptMouse = DPIScale::PixelsToDips(pixelX, pixelY);
    ellipse.radiusX = ellipse.radiusY = 1.0f;
    InvalidateRect(m_hwnd, NULL, FALSE);
}
```

Mover o mouse

Para a mensagem de movimentação do mouse, marcar se o botão esquerdo do mouse está para baixo. Se for, recalcule a elipse e repinte a janela. Em Direct2D, uma elipse é definida pelo ponto central e pelos raios x e y. Queremos desenhar uma elipse que se ajuste à caixa delimitadora definida pelo ponto do mouse para baixo (*ptMouse*) e a posição atual do cursor (*x, y*), portanto, um pouco de aritmética é necessário para localizar a largura, altura e posição da elipse.

O código a seguir recalcula a elipse e chama [InvalidateRect](#) para repintar a janela.

mouse down



C++

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

Botão esquerdo para cima

Para a mensagem de botão esquerdo, basta chamar [ReleaseCapture](#) para liberar a captura do mouse.

C++

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

Avançar

- [Outras operações do mouse](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Operações diversas do mouse

Artigo • 03/06/2023

As seções anteriores discutiram cliques do mouse e movimento do mouse. Aqui estão algumas outras operações que podem ser executadas com o mouse.

Arrastando elementos da interface do usuário

Se a interface do usuário der suporte ao arrasto de elementos da interface do usuário, há uma outra função que você deve chamar no manipulador de mensagens do mouse para baixo: **DragDetect**. A função **DragDetect** retornará **TRUE** se o usuário iniciar um gesto do mouse que deve ser interpretado como arrastando. O código a seguir mostra como usar essa função.

C++

```
case WM_LBUTTONDOWN:
{
    POINT pt = { GET_X_LPARAM(lParam), GET_Y_LPARAM(lParam) };
    if (DragDetect(m_hwnd, pt))
    {
        // Start dragging.
    }
}
return 0;
```

Aqui está a ideia: quando um programa dá suporte a arrastar e soltar, você não quer que cada clique do mouse seja interpretado como um arraste. Caso contrário, o usuário pode arrastar algo acidentalmente quando ele ou ela simplesmente pretendia clicar nele (por exemplo, para selecioná-lo). Mas se um mouse for particularmente sensível, pode ser difícil manter o mouse perfeitamente parado enquanto clica. Portanto, o Windows define um limite de arrastar de alguns pixels. Quando o usuário pressiona o botão do mouse, ele não é considerado um arraste, a menos que o mouse cruze esse limite. A função **DragDetect** testa se esse limite foi atingido. Se a função retornar **TRUE**, você poderá interpretar o clique do mouse como um arraste. Caso contrário, não.

ⓘ Observação

Se **DragDetect** retornar **FALSE**, o Windows suprimirá a mensagem **WM_LBUTTONUP** quando o usuário liberar o botão do mouse. Portanto, não chame **DragDetect**, a menos que seu programa esteja atualmente em um modo que dê suporte à arrastar. (Por exemplo, se um elemento de interface do usuário

arrastável já estiver selecionado.) No final deste módulo, veremos um exemplo de código mais longo que usa a função **DragDetect** .

Limitando o cursor

Às vezes, talvez você queira restringir o cursor à área do cliente ou a uma parte da área do cliente. A função **ClipCursor** restringe o movimento do cursor a um retângulo especificado. Esse retângulo é fornecido em coordenadas de tela, em vez de coordenadas do cliente, portanto, o ponto (0, 0) significa o canto superior esquerdo da tela. Para converter coordenadas do cliente em coordenadas de tela, chame a função **ClientToScreen**.

O código a seguir limita o cursor à área do cliente da janela.

C++

```
// 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 usa uma estrutura **RECT** , mas **ClientToScreen** usa uma estrutura **POINT** . Um retângulo é definido por seus pontos superior esquerdo e inferior direito. Você pode limitar o cursor a qualquer área retangular, incluindo áreas fora da janela, mas limitar o cursor à área do cliente é uma maneira típica de usar a função. Limitar o cursor a uma região totalmente fora da janela seria incomum, e os usuários provavelmente perceberiam isso como um bug.

Para remover a restrição, chame **ClipCursor** com o valor **NULL**.

C++

```
ClipCursor(NULL);
```

Eventos de acompanhamento do mouse: focalizar e sair

Duas outras mensagens do mouse são desabilitadas por padrão, mas podem ser úteis para alguns aplicativos:

- **WM_MOUSEHOVER**: o cursor passou o mouse sobre a área do cliente por um período fixo de tempo.
- **WM_MOUSELEAVE**: o cursor deixou a área do cliente.

Para habilitar essas mensagens, chame a função **TrackMouseEvent**.

C++

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

A estrutura **TRACKMOUSEEVENT** contém os parâmetros da função. O membro **dwFlags** da estrutura contém sinalizadores de bits que especificam em quais mensagens de acompanhamento você está interessado. Você pode optar por obter **WM_MOUSEHOVER** e **WM_MOUSELEAVE**, conforme mostrado aqui, ou apenas um dos dois. O membro **dwHoverTime** especifica por quanto tempo o mouse precisa passar o mouse antes que o sistema gere uma mensagem de foco. Esse valor é fornecido em milissegundos. A constante **HOVER_DEFAULT** significa usar o padrão do sistema.

Depois de receber uma das mensagens solicitadas, a função **TrackMouseEvent** será redefinida. Você deve chamá-lo novamente para obter outra mensagem de acompanhamento. No entanto, você deve aguardar até a próxima mensagem de movimentação do mouse antes de chamar **TrackMouseEvent** novamente. Caso contrário, sua janela poderá ser inundada com mensagens de rastreamento. Por exemplo, se o mouse estiver passando o mouse, o sistema continuará gerando um fluxo de mensagens **WM_MOUSEHOVER** enquanto o mouse estiver parado. Na verdade, você não quer outra mensagem **WM_MOUSEHOVER** até que o mouse se mova para outro local e passe o mouse novamente.

Aqui está uma classe auxiliar pequena que você pode usar para gerenciar eventos de rastreamento de mouse.

C++

```

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;
    }
};

```

O exemplo a seguir mostra como usar essa classe no procedimento de janela.

C++

```

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);
}

```

Os eventos de rastreamento do mouse exigem processamento adicional pelo sistema, portanto, deixe-os desabilitados se você não precisar deles.

Para integridade, aqui está uma função que consulta o sistema para o tempo limite de foco padrão.

```

C++

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

```

Botão de rolagem do mouse

A função a seguir verifica se uma roda do mouse está presente.

```

C++

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

```

Se o usuário girar a roda do mouse, a janela com foco receberá uma **mensagem WM_MOUSEWHEEL**. O parâmetro *wParam* dessa mensagem contém um valor inteiro chamado *delta* que mede o quão longe a roda foi girada. O delta usa unidades arbitrárias, em que 120 unidades são definidas como a rotação necessária para executar uma "ação". É claro que a definição de uma ação depende do seu programa. Por

exemplo, se a roda do mouse for usada para rolar texto, cada 120 unidades de rotação rolará uma linha de texto.

O sinal do delta indica a direção da rotação:

- Positivo: gire para frente, longe do usuário.
- Negativo: gire para trás, em direção ao usuário.

O valor do delta é colocado em *wParam* junto com alguns sinalizadores adicionais. Use a macro [GET_WHEEL_DELTA_WPARAM](#) para obter o valor do delta.

C++

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

Se a roda do mouse tiver uma alta resolução, o valor absoluto do delta poderá ser menor que 120. Nesse caso, se fizer sentido a ação ocorrer em incrementos menores, você poderá fazer isso. Por exemplo, o texto pode rolar por incrementos de menos de uma linha. Caso contrário, acumule o delta total até que a roda gire o suficiente para executar a ação. Armazene o delta não utilizado em uma variável e, quando 120 unidades forem acumuladas (positivas ou negativas), execute a ação.

Avançar

[Entrada por teclado](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Entrada do teclado (Introdução a Win32 e C++)

Artigo • 13/06/2023

O teclado é usado para vários tipos distintos de entrada, incluindo:

- Entrada de caractere. Texto que o usuário digita em um documento ou caixa de edição.
- Atalhos de teclado. Traços de chave que invocam funções de aplicativo; por exemplo, CTRL + O para abrir um arquivo.
- Comandos do sistema. Traços de chave que invocam funções do sistema; por exemplo, ALT + TAB para alternar janelas.

Ao pensar na entrada do teclado, é importante lembrar que um traço de tecla não é o mesmo que um caractere. Por exemplo, pressionar a tecla A pode resultar em qualquer um dos caracteres a seguir.

- um
- Um
- á (se o teclado der suporte à combinação de diacríticos)

Além disso, se a tecla ALT for mantida pressionada, pressionar a tecla A produzirá ALT+A, que o sistema não tratará como um caractere, mas sim como um comando do sistema.

Códigos de chave

Quando você pressiona uma tecla, o hardware gera um *código de verificação*. Os códigos de verificação variam de um teclado para o outro e há códigos de verificação separados para eventos de tecla para cima e para baixo. Você quase nunca se importará com códigos de verificação. O driver de teclado converte códigos de verificação em *códigos de tecla virtual*. Os códigos de chave virtual são independentes do dispositivo. Pressionar a tecla A em qualquer teclado gera o mesmo código de tecla virtual.

Em geral, os códigos de chave virtual não correspondem aos códigos ASCII ou a qualquer outro padrão de codificação de caracteres. Isso é óbvio se você pensar sobre isso, porque a mesma chave pode gerar caracteres diferentes (a, A, á) e algumas teclas, como teclas de função, não correspondem a nenhum caractere.

Dito isto, os seguintes códigos de chave virtual são mapeados para equivalentes ASCII:

- 0 a 9 chaves = ASCII '0' – '9' (0x30 – 0x39)
- Chaves A a Z = ASCII 'A' – 'Z' (0x41 – 0x5A)

Em alguns aspectos, esse mapeamento é lamentável, porque você nunca deve pensar em códigos de chave virtual como caracteres, pelos motivos discutidos.

O arquivo de cabeçalho WinUser.h define constantes para a maioria dos códigos de chave virtual. Por exemplo, o código de tecla virtual para a tecla SETA PARA A ESQUERDA é `VK_LEFT` (0x25). Para obter a lista completa de códigos de chave virtual, consulte [Códigos de chave virtual](#). Nenhuma constante é definida para os códigos de chave virtual que correspondem aos valores ASCII. Por exemplo, o código de chave virtual para a chave A é 0x41, mas não há nenhuma constante chamada `VK_A`. Em vez disso, basta usar o valor numérico.

mensagens Key-Down e Key-Up

Quando você pressiona uma tecla, a janela que tem o foco do teclado recebe uma das mensagens a seguir.

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

A mensagem [WM_SYSKEYDOWN](#) indica uma *chave do sistema*, que é um traço de chave que invoca um comando do sistema. Há dois tipos de chave do sistema:

- ALT + qualquer tecla
- F10

A tecla F10 ativa a barra de menus de uma janela. Várias combinações de teclas ALT invocam comandos do sistema. Por exemplo, ALT + TAB alterna para uma nova janela. Além disso, se uma janela tiver um menu, a tecla ALT poderá ser usada para ativar itens de menu. Algumas combinações de teclas ALT não fazem nada.

Todos os outros traços de chave são considerados chaves não sistema e produzem a mensagem [WM_KEYDOWN](#). Isso inclui as teclas de função diferentes de F10.

Quando você libera uma chave, o sistema envia uma mensagem de chave correspondente:

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

Se você segurar uma tecla por tempo suficiente para iniciar o recurso de repetição do teclado, o sistema enviará várias mensagens de tecla para baixo, seguidas por uma

única mensagem de tecla.

Em todas as quatro mensagens de teclado discutidas até agora, o parâmetro *wParam* contém o código de tecla virtual da tecla. O parâmetro *lParam* contém algumas informações diversas empacotadas em 32 bits. Normalmente, você não precisa das informações no *lParam*. Um sinalizador que pode ser útil é o bit 30, o sinalizador "estado de chave anterior", que é definido como 1 para mensagens de tecla repetidas.

Como o nome indica, os traços de chave do sistema são destinados principalmente para uso pelo sistema operacional. Se você interceptar a mensagem **WM_SYSKEYDOWN**, chame **DefWindowProc** posteriormente. Caso contrário, você bloqueará o sistema operacional de manipular o comando.

Mensagens de caractere

Os traços principais são convertidos em caracteres pela função **TranslateMessage**, que vimos pela primeira vez no **Módulo 1**. Essa função examina mensagens de tecla para baixo e as converte em caracteres. Para cada caractere produzido, a função **TranslateMessage** coloca uma mensagem **WM_CHAR** ou **WM_SYSCHAR** na fila de mensagens da janela. O parâmetro *wParam* da mensagem contém o caractere UTF-16.

Como você pode imaginar, **WM_CHAR** mensagens são geradas a partir de mensagens **WM_KEYDOWN**, enquanto **WM_SYSCHAR** mensagens são geradas a partir de mensagens **WM_SYSKEYDOWN**. Por exemplo, suponha que o usuário pressione a tecla SHIFT seguida pela tecla A. Supondo um layout de teclado padrão, você obterá a seguinte sequência de mensagens:

WM_KEYDOWN: SHIFT
WM_KEYDOWN: A
WM_CHAR: 'A'

Por outro lado, a combinação ALT + P geraria:

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

(O código de chave virtual para a chave ALT é nomeado VK_MENU por motivos históricos.)

A mensagem **WM_SYSCHAR** indica um caractere do sistema. Assim como [acontece com WM_SYSKEYDOWN](#), você geralmente deve passar essa mensagem diretamente para **DefWindowProc**. Caso contrário, você poderá interferir nos comandos padrão do sistema. Em particular, não trate **WM_SYSCHAR** como texto que o usuário digitou.

A [mensagem WM_CHAR](#) é o que você normalmente considera como entrada de caractere. O tipo de dados para o caractere é **wchar_t**, representando um caractere Unicode UTF-16. A entrada de caracteres pode incluir caracteres fora do intervalo ASCII, especialmente com layouts de teclado que são comumente usados fora do Estados Unidos. Você pode experimentar layouts de teclado diferentes instalando um teclado regional e, em seguida, usando o recurso Teclado Virtual.

Os usuários também podem instalar um IME (Editor de Método de Entrada) para inserir scripts complexos, como caracteres japoneses, com um teclado padrão. Por exemplo, usando um IME japonês para inserir o caractere katakana 力 (ka), você pode receber as seguintes mensagens:

WM_KEYDOWN: VK_PROCESSKEY (a chave PROCESS do IME)

WM_KEYUP: 0x4B

WM_KEYDOWN: VK_PROCESSKEY

WM_KEYUP: 0x41

WM_KEYDOWN: VK_PROCESSKEY

WM_CHAR: 力

WM_KEYUP: VK_RETURN

Algumas combinações de teclas CTRL são convertidas em caracteres de controle ASCII. Por exemplo, CTRL+A é convertido para o caractere ASCII ctrl-A (SOH) (valor ASCII 0x01). Para entrada de texto, você geralmente deve filtrar os caracteres de controle. Além disso, evite usar **WM_CHAR** para implementar atalhos de teclado. Em vez disso, use **WM_KEYDOWN** mensagens; ou melhor ainda, use uma tabela de aceleradores. Tabelas de acelerador são descritas no próximo tópico, [Tabelas aceleradoras](#).

O código a seguir exibe as mensagens de teclado main no depurador. Tente reproduzir com diferentes combinações de pressionamento de tecla e veja quais mensagens são geradas.

❗ Observação

Certifique-se de incluir `wchar.h` ou então `swprintf_s` será indefinido.

```

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(hwnd, uMsg, wParam, lParam);
}

```

Mensagens de teclado diversas

Algumas outras mensagens de teclado podem ser ignoradas com segurança pela maioria dos aplicativos.

- A **mensagem WM_DEADCHAR** é enviada para uma chave de combinação, como um diacrítico. Por exemplo, em um teclado de idioma espanhol, digitar ênfase (')

seguido por E produz o caractere é. O `WM_DEADCHAR` é enviado para o personagem de destaque.

- A mensagem `WM_UNICHAR` está obsoleta. Ele permite que programas ANSI recebam entrada de caractere Unicode.
- O caractere `WM_IME_CHAR` é enviado quando um IME converte uma sequência de pressionamento de teclas em caracteres. Ele é enviado além da mensagem `WM_CHAR` usual.

Estado do teclado

As mensagens de teclado são controladas por eventos. Ou seja, você recebe uma mensagem quando algo interessante acontece, como uma tecla, e a mensagem informa o que acabou de acontecer. Mas você também pode testar o estado de uma chave a qualquer momento, chamando a função `GetKeyState`.

Por exemplo, considere como você detectaria a combinação de clique no mouse esquerdo + tecla ALT. Você pode acompanhar o estado da chave ALT escutando mensagens de traço de chave e armazenando um sinalizador, mas `GetKeyState` salva o problema. Ao receber a mensagem `WM_LBUTTONDOWN`, basta chamar `GetKeyState` da seguinte maneira:

C++

```
if (GetKeyState(VK_MENU) & 0x8000)
{
    // ALT key is down.
}
```

A mensagem `GetKeyState` usa um código de chave virtual como entrada e retorna um conjunto de sinalizadores de bits (na verdade, apenas dois sinalizadores). O valor `0x8000` contém o sinalizador de bit que testa se a tecla está pressionada no momento.

A maioria dos teclados tem duas teclas ALT, esquerda e direita. O exemplo anterior testa se um deles é pressionado. Você também pode usar `GetKeyState` para distinguir entre as instâncias esquerda e direita das teclas ALT, SHIFT ou CTRL. Por exemplo, o código a seguir testa se a tecla ALT correta é pressionada.

C++

```
if (GetKeyState(VK_RMENU) & 0x8000)
{
    // Right ALT key is down.
}
```

A função **GetKeyState** é interessante porque relata um estado de teclado *virtual* . Esse estado virtual é baseado no conteúdo da fila de mensagens e é atualizado à medida que você remove mensagens da fila. À medida que seu programa processa mensagens de janela, **GetKeyState** fornece uma instantâneo do teclado no momento em que cada mensagem foi enfileirada. Por exemplo, se a última mensagem na fila foi **WM_LBUTTONDOWN**, **GetKeyState** relata o estado do teclado no momento em que o usuário clicou no botão do mouse.

Como **GetKeyState** é baseado na fila de mensagens, ele também ignora a entrada de teclado que foi enviada para outro programa. Se o usuário alternar para outro programa, todas as teclas que forem enviadas para esse programa serão ignoradas por **GetKeyState**. Se você realmente quiser saber o estado físico imediato do teclado, há uma função para isso: **GetAsyncKeyState**. No entanto, para a maioria dos códigos de interface do usuário, a função correta é **GetKeyState**.

Avançar

[Tabelas de aceleradores](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Tabelas de aceleradores

Artigo • 13/06/2023

Os aplicativos geralmente definem atalhos de teclado, como CTRL+O para o comando Abrir Arquivo. Você pode implementar atalhos de teclado manipulando mensagens [WM_KEYDOWN](#) individuais, mas as tabelas de aceleradores fornecem uma solução melhor que:

- Requer menos codificação.
- Consolida todos os atalhos em um arquivo de dados.
- Dá suporte à localização em outros idiomas.
- Permite que atalhos e comandos de menu usem a mesma lógica de aplicativo.

Uma *tabela aceleradora* é um recurso de dados que mapeia combinações de teclado, como CTRL+O, para comandos de aplicativo. Antes de vermos como usar uma tabela de aceleradores, precisaremos de uma introdução rápida aos recursos. Um *recurso* é um blob de dados integrado a um binário de aplicativo (EXE ou DLL). Os recursos armazenam dados necessários para o aplicativo, como menus, cursores, ícones, imagens, cadeias de caracteres de texto ou dados de aplicativo personalizados. O aplicativo carrega os dados do recurso do binário em tempo de execução. Para incluir recursos em um binário, faça o seguinte:

1. Crie um arquivo de definição de recurso (.rc). Esse arquivo define os tipos de recursos e seus identificadores. O arquivo de definição de recurso pode incluir referências a outros arquivos. Por exemplo, um recurso de ícone é declarado no arquivo .rc, mas a imagem de ícone é armazenada em um arquivo separado.
2. Use o RC (Compilador de Recursos do Microsoft Windows) para compilar o arquivo de definição de recurso em um arquivo de recurso compilado (.res). O compilador RC é fornecido com o Visual Studio e também o SDK do Windows.
3. Vincule o arquivo de recurso compilado ao arquivo binário.

Essas etapas são aproximadamente equivalentes ao processo de compilação/link para arquivos de código. O Visual Studio fornece um conjunto de editores de recursos que facilitam a criação e a modificação de recursos. (Essas ferramentas não estão disponíveis nas edições Express do Visual Studio.) Mas um arquivo .rc é simplesmente um arquivo de texto e a sintaxe está documentada no MSDN, portanto, é possível criar um arquivo .rc usando qualquer editor de texto. Para obter mais informações, consulte [Sobre arquivos de recurso](#).

Definindo uma tabela aceleradora

Uma tabela de aceleradores é uma tabela de atalhos de teclado. Cada atalho é definido por:

- Um identificador numérico. Esse número identifica o comando do aplicativo que será invocado pelo atalho.
- O caractere ASCII ou o código de chave virtual do atalho.
- Teclas modificadoras opcionais: ALT, SHIFT ou CTRL.

A tabela de aceleradores em si tem um identificador numérico, que identifica a tabela na lista de recursos do aplicativo. Vamos criar uma tabela de aceleradores para um programa de desenho simples. Este programa terá dois modos, modo de desenho e modo de seleção. No modo de desenho, o usuário pode desenhar formas. No modo de seleção, o usuário pode selecionar formas. Para este programa, gostaríamos de definir os seguintes atalhos de teclado.

Atalho	Comando
CTRL+M	Alternar entre modos.
F1	Alterne para o modo de desenho.
F2	Alterne para o modo de seleção.

Primeiro, defina identificadores numéricos para a tabela e para os comandos do aplicativo. Esses valores são arbitrários. Você pode atribuir constantes simbólicas para os identificadores definindo-as em um arquivo de cabeçalho. Por exemplo:

C++

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

Neste exemplo, o valor `IDR_ACCEL1` identifica a tabela de aceleradores e as próximas três constantes definem os comandos do aplicativo. Por convenção, um arquivo de cabeçalho que define constantes de recurso geralmente é nomeado `resource.h`. A próxima listagem mostra o arquivo de definição de recurso.

C++

```
#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
}
```

Os atalhos do acelerador são definidos dentro das chaves. Cada atalho contém as entradas a seguir.

- O código de chave virtual ou caractere ASCII que invoca o atalho.
- O comando do aplicativo. Observe que constantes simbólicas são usadas no exemplo. O arquivo de definição de recurso inclui `resource.h`, em que essas constantes são definidas.
- O palavra-chave **VIRTKEY** significa que a primeira entrada é um código de chave virtual. A outra opção é usar caracteres ASCII.
- Modificadores opcionais: ALT, CONTROL ou SHIFT.

Se você usar caracteres ASCII para atalhos, um caractere minúsculo será um atalho diferente de um caractere maiúsculo. (Por exemplo, digitar 'a' pode invocar um comando diferente de digitar 'A'.) Isso pode confundir os usuários, portanto, geralmente é melhor usar códigos de chave virtual, em vez de caracteres ASCII, para atalhos.

Carregando a tabela aceleradora

O recurso para a tabela de aceleradores deve ser carregado antes que o programa possa usá-lo. Para carregar uma tabela de aceleradores, chame a função [LoadAccelerators](#).

C++

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

Chame essa função antes de inserir o loop de mensagem. O primeiro parâmetro é o identificador do módulo. (Esse parâmetro é passado para sua função [WinMain](#). Para obter detalhes, consulte [WinMain: o ponto de entrada do aplicativo](#).) O segundo parâmetro é o identificador de recurso. A função retorna um identificador para o recurso. Lembre-se de que um identificador é um tipo opaco que se refere a um objeto gerenciado pelo sistema. Se a função falhar, ela retornará **NULL**.

Você pode liberar uma tabela de aceleradores chamando [DestroyAcceleratorTable](#). No entanto, o sistema libera automaticamente a tabela quando o programa é encerrado, portanto, você só precisa chamar essa função se estiver substituindo uma tabela por

outra. Há um exemplo interessante disso no tópico [Criando aceleradores editáveis do usuário](#).

Traduzindo traços de teclas em comandos

Uma tabela de aceleradores funciona traduzindo traços de tecla em mensagens **WM_COMMAND**. O parâmetro *wParam* de **WM_COMMAND** contém o identificador numérico do comando. Por exemplo, usando a tabela mostrada anteriormente, o traço de tecla CTRL+M é convertido em uma mensagem **WM_COMMAND** com o valor **ID_TOGGLE_MODE**. Para fazer isso acontecer, altere o loop de mensagem para o seguinte:

C++

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

Esse código adiciona uma chamada à função **TranslateAccelerator** dentro do loop de mensagem. A função **TranslateAccelerator** examina cada mensagem de janela, procurando mensagens de tecla para baixo. Se o usuário pressionar uma das combinações de teclas listadas na tabela de aceleradores, **TranslateAccelerator** enviará uma mensagem de **WM_COMMAND** para a janela. A função envia **WM_COMMAND** invocando diretamente o procedimento de janela. Quando **TranslateAccelerator** converte com êxito um traço de chave, a função retorna um valor diferente de zero, o que significa que você deve ignorar o processamento normal da mensagem. Caso contrário, **TranslateAccelerator** retornará zero. Nesse caso, passe a mensagem da janela para **TranslateMessage** e **DispatchMessage**, normalmente.

Veja como o programa de desenho pode lidar com a mensagem **WM_COMMAND**:

C++

```
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;
```

Esse código pressupõe que `SetMode` seja uma função definida pelo aplicativo para alternar entre os dois modos. Os detalhes de como você lidaria com cada comando obviamente dependem do seu programa.

Avançar

[Definindo a imagem do cursor](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Definindo a imagem do cursor

Artigo • 08/06/2023

O *cursor* é a imagem pequena que mostra a localização do mouse ou de outro dispositivo apontador. Muitos aplicativos alteram a imagem do cursor para fornecer comentários ao usuário. Embora não seja necessário, ele adiciona um pouco de polimento ao seu aplicativo.

O Windows fornece um conjunto de imagens de cursor padrão, chamadas *cursores do sistema*. Isso inclui a seta, a mão, o feixe de I, a ampulheta (que agora é um círculo giratório) e outros. Esta seção descreve como usar os cursores do sistema. Para tarefas mais avançadas, como criar cursores personalizados, consulte [Cursores](#).

Você pode associar um cursor a uma classe de janela definindo o membro **hCursor** da estrutura **WNDCLASS** ou **WNDCLASSEX**. Caso contrário, o cursor padrão é a seta. Quando o mouse se move sobre uma janela, a janela recebe uma mensagem **WM_SETCURSOR** (a menos que outra janela tenha capturado o mouse). Neste ponto, ocorre um dos seguintes eventos:

- O aplicativo define o cursor e o procedimento de janela retorna **TRUE**.
- O aplicativo não faz nada e passa **WM_SETCURSOR** para **DefWindowProc**.

Para definir o cursor, um programa faz o seguinte:

1. Chama **LoadCursor** para carregar o cursor na memória. Essa função retorna um identificador para o cursor.
2. Chama **SetCursor** e passa o identificador do cursor.

Caso contrário, se o aplicativo passar **WM_SETCURSOR** para **DefWindowProc**, a função **DefWindowProc** usará o seguinte algoritmo para definir a imagem do cursor:

1. Se a janela tiver um pai, encaminhe a mensagem **WM_SETCURSOR** para o pai a ser manipulado.
2. Caso contrário, se a janela tiver um cursor de classe, defina o cursor como o cursor de classe.
3. Se não houver cursor de classe, defina o cursor como o cursor de seta.

A função **LoadCursor** pode carregar um cursor personalizado de um recurso ou um dos cursores do sistema. O exemplo a seguir mostra como definir o cursor para o cursor de seleção de link predefinido do sistema.

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

Se você alterar o cursor, a imagem do cursor será redefinida no próximo movimento do mouse, a menos que você intercepte a mensagem **WM_SETCURSOR** e defina o cursor novamente. O código a seguir mostra como lidar com **WM_SETCURSOR**.

C++

```
case WM_SETCURSOR:  
    if (LOWORD(lParam) == HTCLIENT)  
    {  
        SetCursor(hCursor);  
        return TRUE;  
    }  
    break;
```

Esse código primeiro verifica os 16 bits inferiores de *lParam*. Se `LOWORD(lParam)` for igual a **HTCLIENT**, significa que o cursor está sobre a área do cliente da janela. Caso contrário, o cursor estará sobre a área não cliente. Normalmente, você só deve definir o cursor para a área do cliente e permitir que o Windows defina o cursor para a área não cliente.

Avançar

[Entrada do usuário: exemplo estendido](#)

Comentários

Esta página foi útil?

 Yes

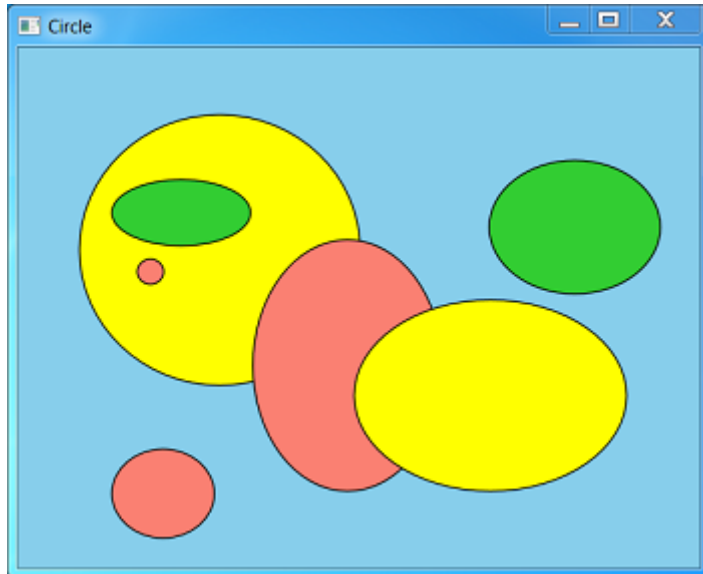
 No

[Obter ajuda no Microsoft Q&A](#)

Entrada do usuário: exemplo estendido

Artigo • 07/08/2024

Vamos combinar tudo o que aprendemos sobre a entrada do usuário para criar um programa de desenho simples. Aqui está uma captura de tela do programa:



O usuário pode desenhar elipses em várias cores diferentes e selecionar, mover ou excluir elipses. Para manter a interface do usuário simples, o programa não permite que o usuário selecione as cores da elipse. Em vez disso, o programa percorre automaticamente uma lista predefinida de cores. O programa não suporta nenhuma forma além de elipses. Obviamente, este programa não ganhará nenhum prêmio de software gráfico. No entanto, ainda é um exemplo útil de aprendizado. Você pode baixar o código fonte completo da [amostra de desenho simples](#). Esta seção abordará apenas alguns destaques.

As elipses são representadas no programa por uma estrutura que contém os dados da elipse (`D2D1_ELLIPSE`) e a cor (`D2D1_COLOR_F`). A estrutura também define dois métodos: um método para desenhar a elipse e um método para executar o teste de clique.

C++

```
struct MyEllipse
{
    D2D1_ELLIPSE    ellipse;
    D2D1_COLOR_F    color;

    void Draw(ID2D1RenderTarget *pRT, ID2D1SolidColorBrush *pBrush)
    {
        pBrush->SetColor(color);
        pRT->FillEllipse(ellipse, pBrush);
    }
};
```

```

        pBrush->SetColor(D2D1::ColorF(D2D1::ColorF::Black));
        pRT->DrawEllipse(ellipse, pBrush, 1.0f);
    }

    BOOL HitTest(float x, float y)
    {
        const float a = ellipse.radiusX;
        const float b = ellipse.radiusY;
        const float x1 = x - ellipse.point.x;
        const float y1 = y - ellipse.point.y;
        const float d = ((x1 * x1) / (a * a)) + ((y1 * y1) / (b * b));
        return d <= 1.0f;
    }
};

```

O programa usa o mesmo pincel de cor sólida para desenhar o preenchimento e o contorno de cada elipse, alterando a cor sempre que necessário. Em Direct2D, alterar a cor de um pincel de cor sólida é uma operação eficiente. Portanto, o objeto pincel de cor sólida dá suporte a um método [SetColor](#).

As elipses são armazenadas em um container de lista **STL**:

C++

```
list<shared_ptr<MyEllipse>> ellipses;
```

❗ Observação

shared_ptr é uma classe de ponteiro inteligente que foi adicionada ao C++ em TR1 e formalizada em C++0x.. O Visual Studio 2010 adiciona suporte para **shared_ptr** e outros recursos do C++0x. Para obter mais informações, consulte o artigo da MSDN Magazine [Explorando novos recursos do C++ e do MFC no Visual Studio 2010](#).

O programa tem três modos:

- Modo de desenho. O usuário pode desenhar novas elipses.
- Modo de seleção. O usuário pode selecionar uma elipse.
- Modo de arrastar. O usuário pode arrastar uma elipse selecionada.

O usuário pode alternar entre o modo de desenho e o modo de seleção usando os mesmos atalhos do teclado descritos em [tabelas de aceleradores](#). No modo de seleção, o programa alterna para o modo de arrastar se o usuário clicar em uma elipse. Ele volta

para o modo de seleção quando o usuário solta o botão do mouse. A seleção atual é armazenada como um iterador na lista de elipses. O método auxiliar retorna um ponteiro para a elipse selecionada ou o valor `nullptr` se não houver seleção.

`MainWindow::Selection` retorna um ponteiro para a elipse selecionada ou o valor `nullptr` se não houver seleção.


C++

```
list<shared_ptr<MyEllipse>>::iterator selection;

shared_ptr<MyEllipse> Selection()
{
    if (selection == ellipses.end())
    {
        return nullptr;
    }
    else
    {
        return (*selection);
    }
}

void ClearSelection() { selection = ellipses.end(); }
```

A tabela a seguir resume os efeitos da entrada do mouse em cada um dos três modos.

 Expandir a tabela

Entrada por mouse	Modo de Desenho	Modo de Seleção	Modo de arrastar
Botão esquerdo para baixo	Defina a captura do mouse e comece a desenhar uma nova elipse.	Libere a seleção atual e execute um teste de clique. Se uma elipse for tocada, capture o cursor, selecione a elipse e mude para o modo de arrastar.	Nenhuma ação.
Movimento do mouse	Se o botão esquerdo estiver pressionado, redimensione a elipse.	Nenhuma ação.	Mova a elipse selecionada.
Botão esquerdo para cima	Pare de desenhar a elipse.	Nenhuma ação.	Mudar para o modo de seleção.

O método a seguir na classe `MainWindow` lida com mensagens `WM_LBUTTONDOWN`.

C++

```
void MainWindow::OnLButtonDown(int pixelX, int pixelY, DWORD flags)
{
    const float dipX = DPIScale::PixelsToDipsX(pixelX);
    const float dipY = DPIScale::PixelsToDipsY(pixelY);

    if (mode == DrawMode)
    {
        POINT pt = { pixelX, pixelY };

        if (DragDetect(m_hwnd, pt))
        {
            SetCapture(m_hwnd);

            // Start a new ellipse.
            InsertEllipse(dipX, dipY);
        }
    }
    else
    {
        ClearSelection();

        if (HitTest(dipX, dipY))
        {
            SetCapture(m_hwnd);

            ptMouse = Selection()->ellipse.point;
            ptMouse.x -= dipX;
            ptMouse.y -= dipY;

            SetMode(DragMode);
        }
    }
    InvalidateRect(m_hwnd, NULL, FALSE);
}
```

As coordenadas do mouse são passadas para esse método em pixels e, em seguida, são convertidas em DIPs. É importante não confundir essas duas unidades. Por exemplo, a função **DragDetect** usa pixels, porém o desenho e o teste de clique usam DIPs. A regra geral é que as funções relacionadas a janelas ou entrada do mouse usam pixels, enquanto Direct2D e DirectWrite usam DIPs. Sempre teste seu programa em uma configuração de alta DPI e lembre-se de marcar seu programa como compatível com DPI. Para obter mais informações, consulte [DPI e Pixels independentes do dispositivo](#).

Aqui está o código que lida com as mensagens **WM_MOUSEMOVE**.

C++

```
void MainWindow::OnMouseMove(int pixelX, int pixelY, DWORD flags)
{
```

```

const float dipX = DPIScale::PixelsToDipsX(pixelX);
const float dipY = DPIScale::PixelsToDipsY(pixelY);

if ((flags & MK_LBUTTON) && Selection())
{
    if (mode == DrawMode)
    {
        // Resize the ellipse.
        const float width = (dipX - ptMouse.x) / 2;
        const float height = (dipY - ptMouse.y) / 2;
        const float x1 = ptMouse.x + width;
        const float y1 = ptMouse.y + height;

        Selection()->ellipse = D2D1::Ellipse(D2D1::Point2F(x1, y1),
width, height);
    }
    else if (mode == DragMode)
    {
        // Move the ellipse.
        Selection()->ellipse.point.x = dipX + ptMouse.x;
        Selection()->ellipse.point.y = dipY + ptMouse.y;
    }
    InvalidateRect(m_hwnd, NULL, FALSE);
}
}

```

A lógica para redimensionar uma elipse foi descrita anteriormente, na seção [Exemplo: desenhando círculos](#). Observe também a chamada para [InvalidateRect](#). Isso garante que a janela seja repintada. O código a seguir lida com mensagens [WM_LBUTTONUP](#).

C++

```

void MainWindow::OnLButtonUp()
{
    if ((mode == DrawMode) && Selection())
    {
        ClearSelection();
        InvalidateRect(m_hwnd, NULL, FALSE);
    }
    else if (mode == DragMode)
    {
        SetMode(SelectMode);
    }
    ReleaseCapture();
}

```

Como você pode ver, todos os manipuladores de mensagens para entrada do mouse têm código de ramificação, dependendo do modo atual. Esse é um design aceitável para esse programa relativamente simples. No entanto, ele pode rapidamente tornar-se muito complexo se novos modos forem adicionados. Para um programa maior, uma

arquitetura (MVC) pode ter um design melhor. Nesse tipo de arquitetura, o *controlador* que lida com a entrada do usuário, é separado do *modelo* que gerencia os dados do aplicativo.

Quando o programa alterna os modos, o cursor muda para fornecer comentários ao usuário.

C++

```
void MainWindow::SetMode(Mode m)
{
    mode = m;

    // Update the cursor
    LPWSTR cursor;
    switch (mode)
    {
        case DrawMode:
            cursor = IDC_CROSS;
            break;

        case SelectMode:
            cursor = IDC_HAND;
            break;

        case DragMode:
            cursor = IDC_SIZEALL;
            break;
    }

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

E, finalmente, lembre-se de definir o cursor quando a janela receber uma mensagem **WM_SETCURSOR**:

C++

```
case WM_SETCURSOR:
    if (LOWORD(lParam) == HTCLIENT)
    {
        SetCursor(hCursor);
        return TRUE;
    }
    break;
```

Resumo

Neste módulo, você aprendeu a lidar com a entrada do mouse e do teclado, como definir atalhos de teclado, e como atualizar a imagem do cursor para refletir o estado atual do programa.

Comentários

Esta página foi útil?



Yes



No

[Fornecer comentários sobre o produto](#)  | [Obter ajuda no Microsoft Q&A](#)

Introdução ao Win32: código de exemplo

Artigo • 13/06/2023

Esta seção contém links para o código de exemplo para a série [Introdução ao Win32 e C++](#).

Nesta seção

Tópico	Descrição
Amostra do Windows Hello World	Este aplicativo de exemplo mostra como criar um programa mínimo do Windows.
Exemplo de BaseWindow	Este aplicativo de exemplo mostra como passar dados de estado do aplicativo na mensagem WM_NCCREATE .
Abrir exemplo de caixa de diálogo	Este aplicativo de exemplo mostra como inicializar a biblioteca COM (Component Object Model) e usar uma API baseada em COM em um programa do Windows.
Exemplo de círculo Direct2D	Este aplicativo de exemplo mostra como desenhar um círculo usando Direct2D.
Exemplo de relógio Direct2D	Este aplicativo de exemplo mostra como usar transformações em Direct2D para desenhar as mãos de um relógio.
Amostra de Círculo de Desenho	Este aplicativo de exemplo mostra como usar a entrada do mouse para desenhar um círculo.
Exemplo de desenho simples	Este aplicativo de exemplo é um programa de desenho muito simples que mostra como usar a entrada do mouse, a entrada do teclado e as tabelas de acelerador.

Tópicos relacionados

[Introdução ao Win32 e C++](#)

Comentários

Esta página foi útil?



Yes



No

[Obter ajuda no Microsoft Q&A](#)

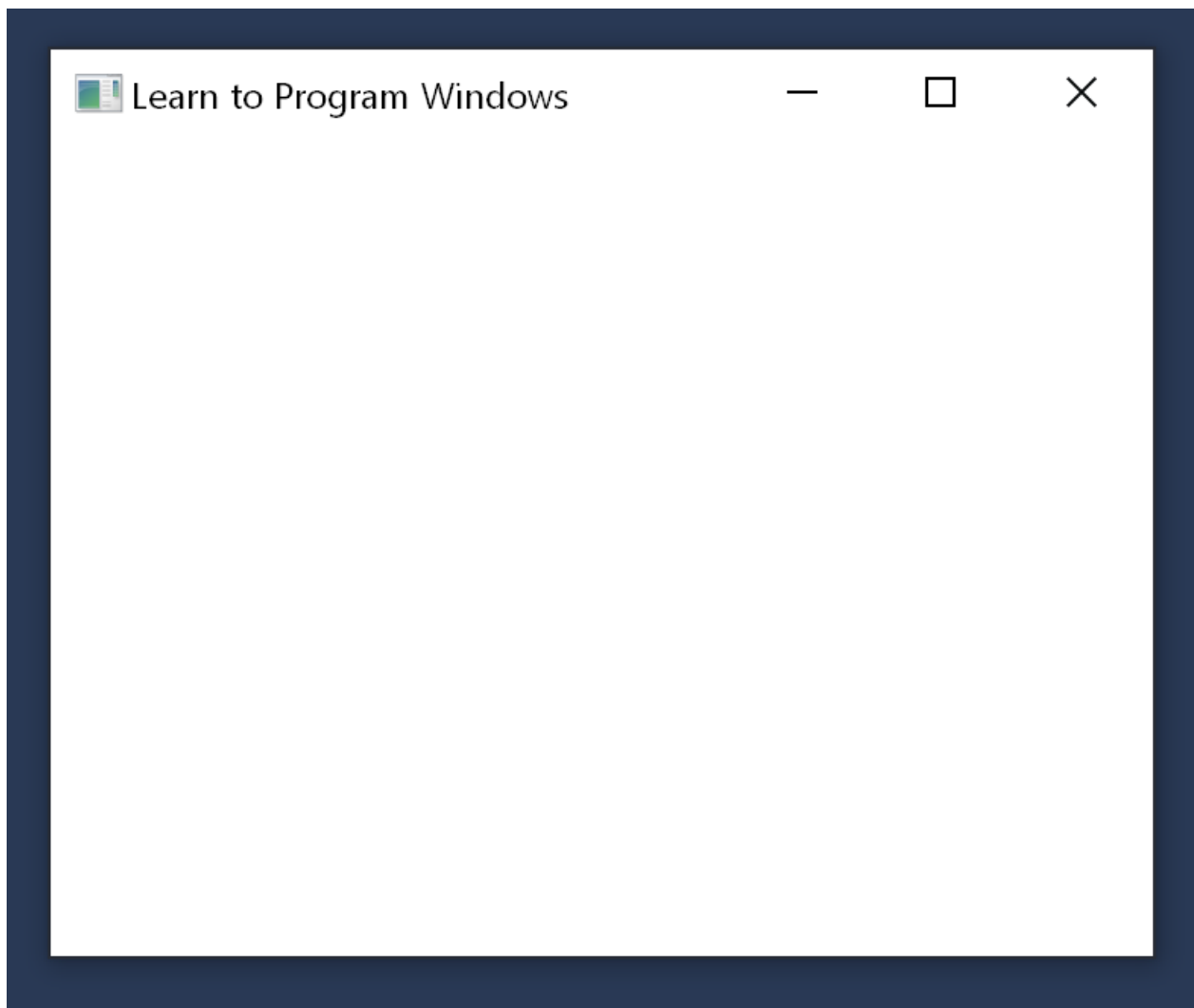
Amostra do Windows Hello World

Artigo • 08/06/2023

Este aplicativo de exemplo mostra como criar um programa mínimo do Windows.

Descrição

O aplicativo de exemplo Windows Hello World cria e mostra uma janela vazia, conforme mostrado na captura de tela a seguir. Este exemplo é discutido no [Módulo 1. Seu primeiro programa do Windows](#).



Baixando o exemplo

Este exemplo está disponível [aqui](#).

Para baixá-lo, vá para a raiz do repositório de exemplo no GitHub ([microsoft/Windows-classic-samples](#)) e clique no botão **Clonar ou baixar** para baixar o arquivo zip de todos os exemplos em seu computador. Em seguida, descompacte a pasta.

Para abrir o exemplo no Visual Studio, selecione **Arquivo/Abrir/Projeto/Solução** e navegue até o local em que você descompactou a pasta e **Windows-classic-samples-main/Samples/ Win7Samples / begin / LearnWin32 / HelloWorld / cpp**. Abra o arquivo **HelloWorld.sln**.

Depois que o exemplo for carregado, você precisará atualizá-lo para trabalhar com Windows 10. No menu **Projeto** no Visual Studio, selecione **Propriedades**. Atualize a **versão do SDK do Windows** para um SDK do Windows 10, como 10.0.17763.0 ou superior. Em seguida, altere o **Conjunto de Ferramentas de Plataforma** para o Visual Studio 2017 ou superior. Agora você pode executar o exemplo pressionando F5!

Tópicos relacionados

- [Aprenda a programar para Windows: código de exemplo](#)
- [Módulo 1. Seu primeiro programa do Windows](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

Exemplo de BaseWindow

Artigo • 13/06/2023

Este aplicativo de exemplo mostra como passar dados de estado do aplicativo na mensagem [WM_NCCREATE](#).

Descrição

O aplicativo de exemplo BaseWindow é uma variação do [exemplo Windows Hello World](#). Ele usa a mensagem [WM_NCCREATE](#) para passar dados do aplicativo para o procedimento de janela. Este exemplo é discutido no tópico [Gerenciando o estado do aplicativo](#).

Baixando o exemplo

Este exemplo está disponível [aqui](#).

Tópicos relacionados

- [Aprenda a programar para Windows: código de exemplo](#)
- [Gerenciando o estado do aplicativo](#)
- [Módulo 1. Seu primeiro programa do Windows](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

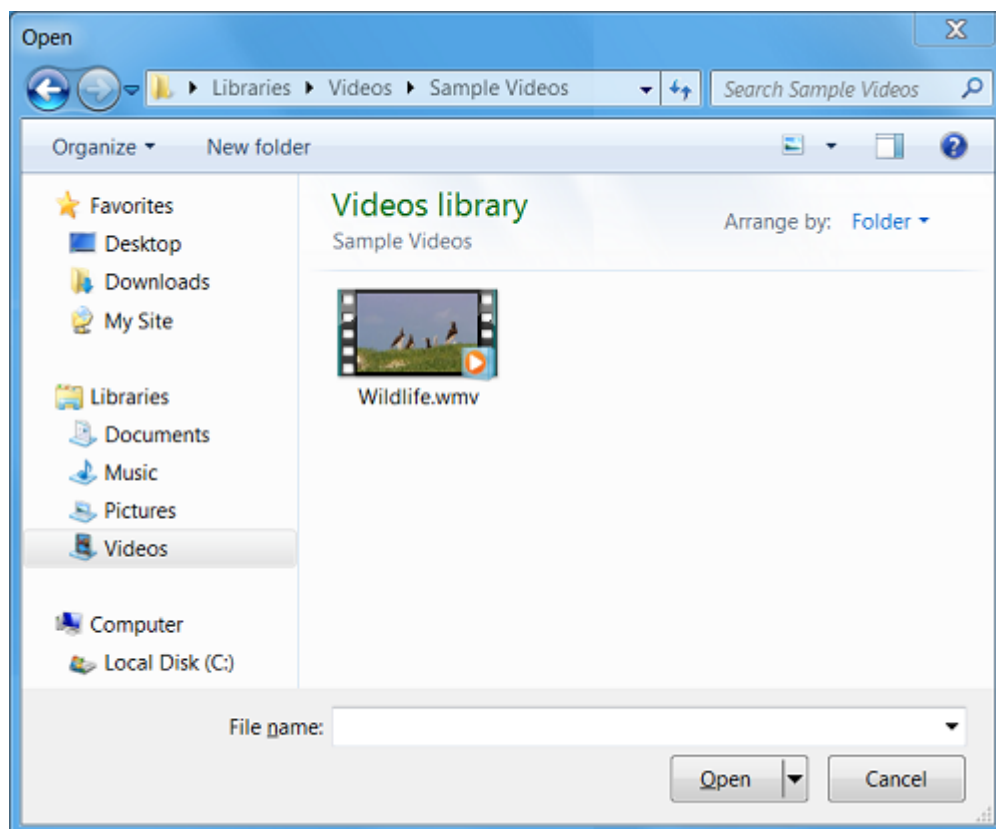
Abrir exemplo de caixa de diálogo

Artigo • 13/06/2023

Este aplicativo de exemplo mostra como inicializar a biblioteca COM (Component Object Model) e usar uma API baseada em COM em um programa do Windows.

Descrição

O aplicativo de exemplo Abrir Caixa de Diálogo exibe a caixa de diálogo **Abrir**, conforme mostrado na captura de tela a seguir. O exemplo demonstra como chamar um objeto COM em um programa do Windows. Este exemplo é discutido no [Módulo 2: Usando COM em seu programa do Windows](#).



Baixando o exemplo

Este exemplo está disponível [aqui](#).

Tópicos relacionados

- [Exemplo: a caixa de diálogo Abrir](#)
- [Aprenda a programar para Windows: código de exemplo](#)
- [Módulo 2: Usando COM em seu programa do Windows](#)

Comentários

Esta página foi útil?

 Yes

 No

[Obter ajuda no Microsoft Q&A](#)

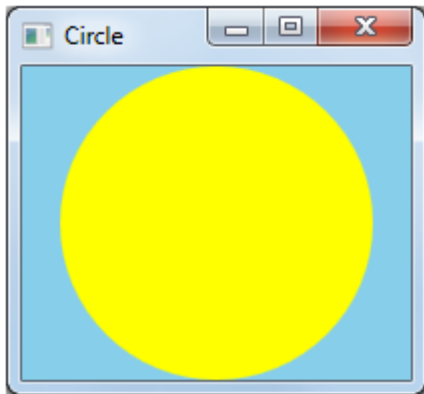
Exemplo de círculo Direct2D

Artigo • 12/06/2023

Este aplicativo de exemplo mostra como desenhar um círculo usando Direct2D.

Descrição

O aplicativo de exemplo Direct2D Circle desenha um círculo, conforme mostrado na captura de tela a seguir. Este exemplo é discutido no [Módulo 3: Gráficos do Windows](#).



Baixando o exemplo

Este exemplo está disponível [aqui](#).

Tópicos relacionados

- [Aprenda a programar para Windows: código de exemplo](#)
- [Primeiro programa de Direct2D](#)
- [Módulo 3: Gráficos do Windows](#)

Comentários

Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

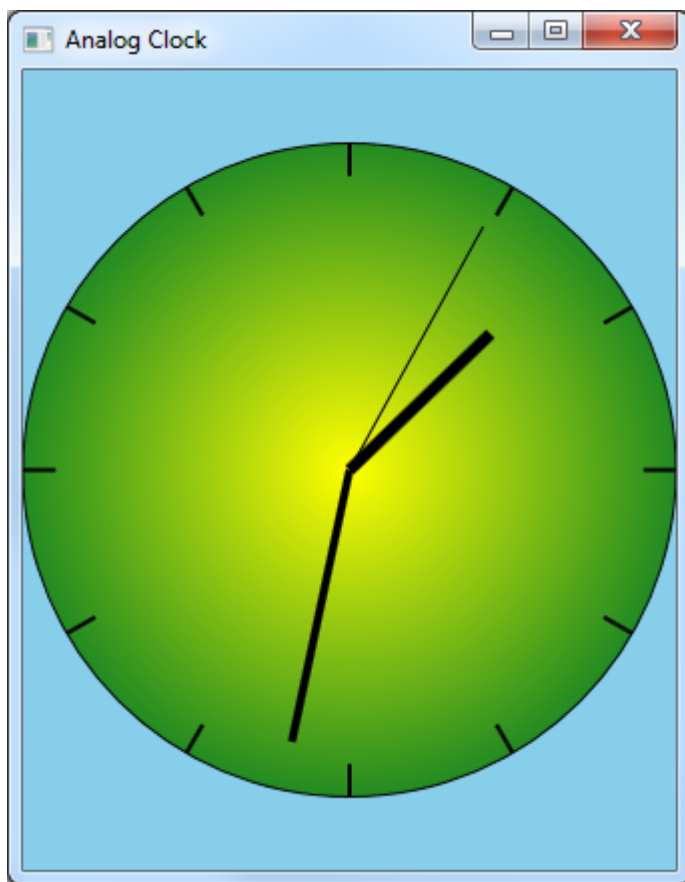
Exemplo de relógio Direct2D

Artigo • 13/06/2023

Este aplicativo de exemplo mostra como usar transformações em Direct2D para desenhar as mãos de um relógio.

Descrição

O aplicativo de exemplo Direct2D Clock desenha um relógio analógico, conforme mostrado na captura de tela a seguir. Este exemplo é discutido em [Aplicando transformações em Direct2D](#).



Baixando o exemplo

Este exemplo está disponível [aqui](#).

Tópicos relacionados

[Aprenda a programar para Windows: código de exemplo](#)

[Aplicar transformações em Direct2D](#)

Comentários

Esta página foi útil?



Yes



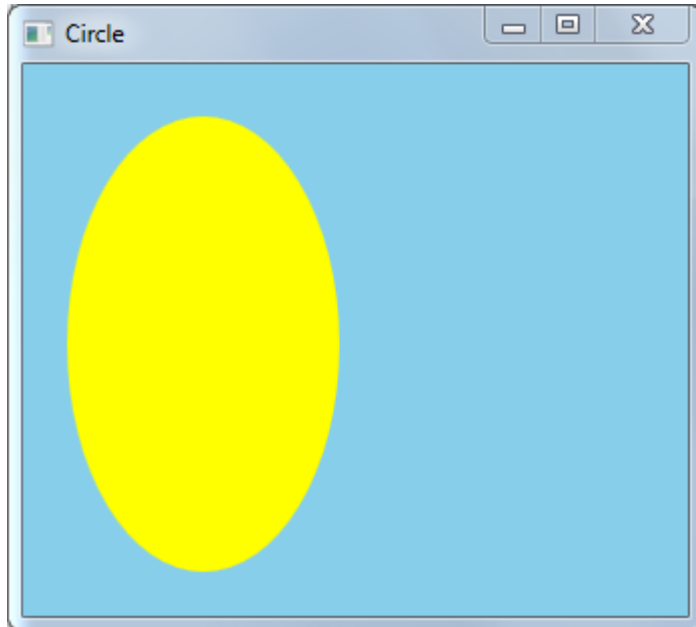
No

[Obter ajuda no Microsoft Q&A](#)

Amostra de Círculo de Desenho

Artigo • 12/06/2023

Este aplicativo de exemplo mostra como usar a entrada do mouse para desenhar um círculo.



Baixando o exemplo

Este exemplo está disponível [aqui](#).

Tópicos relacionados

- [Aprenda a programar para Windows: código de exemplo](#)
- [Módulo 4. Entrada do usuário](#)

Comentários

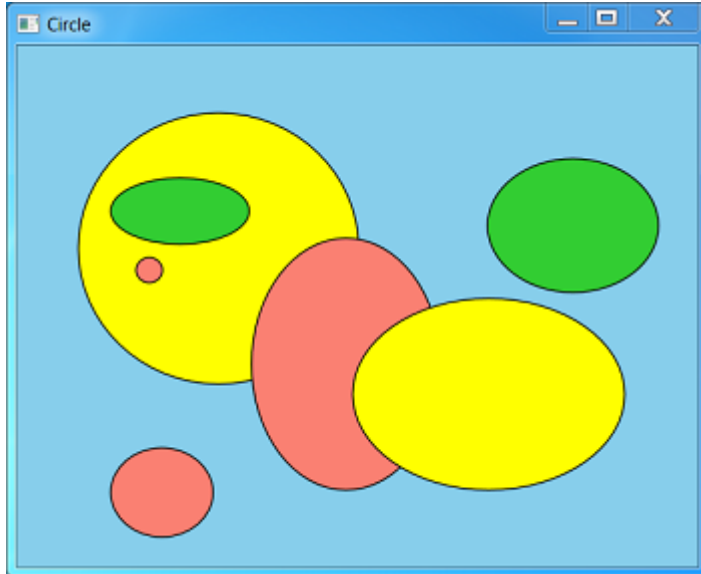
Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)

Exemplo de desenho simples

Artigo • 13/06/2023

Este aplicativo de exemplo é um programa de desenho muito simples que mostra como usar a entrada do mouse, a entrada do teclado e as tabelas de acelerador.



Baixando o exemplo

Este exemplo está disponível [aqui](#).

Tópicos relacionados

- [Aprenda a programar para Windows: código de exemplo](#)
- [Módulo 4. Entrada do usuário](#)
- [Entrada do usuário: exemplo estendido](#)

Comentários

Esta página foi útil?

[Obter ajuda no Microsoft Q&A](#)