



Win32 ve C++ ile Başlarken

Makale - 01/27/2022 - Okumak için 2 dakika

Bu Başlangıç serisinin amacı, Win32 ve COM API'lerini kullanarak C++'da bir masaüstü programının nasıl yazılacağını öğretmektir.

İlk modülde, bir pencerenin nasıl oluşturulacağını ve gösterileceğini adım adım öğreneceksiniz. Daha sonraki modüllerde Bileşen Nesne Modeli (COM), grafikler, metin ve kullanıcı girişi tanıtılacaktır.

Bu seri için, C++ programlama konusunda iyi bir çalışma bilgisine sahip olduğunuz varsayılmaktadır. Windows programlama ile ilgili önceden deneyiminiz olmadığı varsayılmaktadır. C++ [dilinde](#) yeniyseniz, [C++ dil belgelerinde](#) öğrenme materyalleri mevcuttur.

Bu bölümde

Konu	Açıklama
C++'da Win32 programlamaya giriş	Bu bölümde Windows programlamada kullanılan bazı temel terminoloji ve kodlama kuralları açıklanmaktadır.
Modül 1. İlk Windows Programınız	Bu modülde, boş bir pencere gösteren basit bir Windows programı oluşturacaksınız.
Modül 2. Windows Programınızda COM Kullanımı	Bu modülde, modern Windows API'lerinin çoğunun temelini oluşturan Bileşen Nesne Modeli (COM) tanıtılmaktadır.
Modül 3. Windows Grafikleri	Bu modül, Direct2D'ye odaklanarak Windows grafik mimarisini tanıtır.
Modül 4. Kullanıcı Girişi	Bu modülde fare ve klavye girişi açıklanmaktadır.
Örnek Kod	Bu seri için örnek kodu indirmek için bağlantılar içerir.

C++'da Win32 programlamaya giriş

Makale - 23.08.2019 - Okumak için 2 dakika

Bu bölümde Windows programlamada kullanılan bazı temel terminoloji ve kodlama kuralları açıklanmaktadır.

Bu bölümde

- ♦ [Geliştirme Ortamınızı Hazırlama Windows](#)
- ♦ [Kodlama Kuralları](#)
- ♦ [Dizelerle Çalışma](#)
- ♦ [Pencere Nedir?](#)
- ♦ [WinMain: Uygulama Giriş Noktası](#)

İlgili konular

[Win32 ve C++ ile Başlarken Modül 1.](#)

[İlk Windows Programınız](#)

Geliştirme Ortamınızı Hazırlayın

Makale - 08/19/2020 - Okumak için 2 dakika

C veya C++ dilinde bir Windows programı yazmak için Microsoft Windows Yazılım Geliştirme Kiti'ni (SDK) veya Microsoft Visual Studio'yu yüklemeniz gerekir. Windows SDK, uygulamanızı derlemek ve bağlamak için gerekli başlıkları ve kitaplıkları içerir. Windows SDK ayrıca Visual C++ derleyicisi ve bağlayıcısı dahil olmak üzere Windows uygulamaları oluşturmak için komut satırı araçları içerir. Windows programlarını komut satırı araçlarıyla derleyip oluşturabilmenize rağmen, Microsoft Visual Studio kullanmanızı öneririz. Visual Studio Community'nin ücretsiz indirmesini veya Visual Studio'nun diğer sürümlerinin ücretsiz deneme sürümlerini [buradan indirebilirsiniz](#)

Windows SDK'nın her sürümü, Windows'un en son sürümünün yanı sıra önceki birkaç sürümü de hedeflemektedir. Sürüm notları desteklenen belirli platformları listeler, ancak çok eski bir Windows sürümü için bir uygulama geliştirmiyorsanız, Windows SDK'nın en son sürümünü yüklemelisiniz. En son Windows SDK'sını [buradan](#) indirebilirsiniz.

Windows SDK hem 32 bit hem de 64 bit uygulamaların geliştirilmesini destekler. Windows API'leri, aynı kodun 32 bit veya 64 bit için değişiklik yapılmadan derlenebileceği şekilde tasarlanmıştır.

7 Not

Windows SDK donanım sürücüsü geliştirmeyi desteklemez ve bu seride sürücü geliştirme konusu ele alınmayacaktır. Donanım sürücüsü yazma hakkında bilgi için **Windows Sürücülerine Başlarken** bölümüne bakın.

Sonraki

[Windows Kodlama Kuralları](#)

İlgili konular

- ♦ [Visual Studio İndir](#)
- ♦ [Windows SDK İndir](#)

Windows Kodlama Kuralları

Makale - 19/11/2022 - Okumak için 3 dakika

Windows programlamada yeniyseniz, bir Windows programını ilk gördüğünüzde rahatsız edici olabilir. Kod, **DWORD_PTR** ve **LPRECT** gibi garip tip tanımlarıyla doludur ve değişkenlerin *hWnd* ve *pwsz* gibi isimleri vardır (Macar notasyonu olarak adlandırılır). Windows kodlama kurallarından bazılarını öğrenmek için bir dakikanızı ayırmaya değer.

Windows API'lerinin büyük çoğunluğu ya fonksiyonlardan ya da Bileşen Nesne Modeli (COM) arayüzlerinden oluşur. Çok az sayıda Windows API'si C++ sınıfı olarak sağlanır. (Dikkate değer bir istisna, 2 boyutlu grafik API'lerinden biri olan GDI+'dır).

Tip Tanımları

Windows başlıkları çok sayıda typedef içerir. Bunların çoğu WinDef.h başlık dosyasında tanımlanmıştır. İşte sıkça karşılaştığınız bazıları.

Tamsayı türleri


Veri türü	Boyut	İmzaladın mı?
BYTE	8 bit	İşaretsiz
DWORD	32 bit	İşaretsiz
INT32	32 bit	İmza
INT64	64 bit	İmza
UZUN	32 bit	İmza
LONGLONG	64 bit	İmza
UINT32	32 bit	İşaretsiz
UINT64	64 bit	İşaretsiz
ULONG	32 bit	İşaretsiz
ULONGLONG	64 bit	İşaretsiz
KELİME	16 bit	İşaretsiz

Gördüğünüz gibi, bu tip tanımlarında belirli bir miktar fazlalık vardır. Bu örtüşmenin bir kısmı Windows API'lerinin geçmişinden kaynaklanmaktadır. Burada listelenen tipler

sabit boyuttadır ve boyutlar hem 32-bit hem de 64-uygulamalarda aynıdır. Örneğin, **DWORD** tipi her zaman 32 bit genişliğindedir.

Boolean Tip

BOOL, **int** için bir tür takma adıdır, C++'ın **bool**'undan ve diğer türlerden farklıdır.

bir **Boolean**'ı temsil  değer. Başlık dosyası **WinD** ayrıca kullanım için iki değer tanımlar eder

BOOL ile.

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

TRUE'nin bu tanımına rağmen, **BOOL** tipi döndüren çoğu fonksiyon Boolean doğruluğunu belirtmek için sıfır olmayan herhangi bir değer döndürebilir. Bu nedenle, her zaman şunu yazmalısınız:

```
// Doğru yol.
if (SomeFunctionThatReturnsBoolean())
{
    ...
}

// veya

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

ve bu değil:

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

BOOL bir tamsayı türüdür ve C++'ın **bool**'u ile değiştirilemez.

Pointer Türleri

Windows, *pointer-to-X* biçiminde birçok veri türü tanımlar. Bunların adında genellikle *P-* veya *LP-* öneki bulunur. Örneğin, **LPRECT** bir **RECT** işaretçisidir, burada **RECT** bir dikdörtgeni tanımlayan bir yapıdır. Aşağıdaki değişken bildirimleri eşdeğerdir.

```
RECT* rect;    // Bir RECT yapısına işaretçi.  
LPRECT rect;   // Aynı  
PRECT rect;    // Aynı şekilde.
```

16 bit mimarilerde (16 bit Windows) 2 tür işaretçi vardır, *P* "işaretçi" ve *LP* "uzun işaretçi" anlamına gelir. Uzun işaretçiler (*uzak işaretçiler* olarak da adlandırılır) mevcut segmentin dışındaki bellek aralıklarını adreslemek için gereklidir. *LP* öneki, 16 bit kodun 32 bit Windows'a taşınmasını kolaylaştırmak için korunmuştur. Günümüzde herhangi bir ayırım yoktur ve bu işaretçi türlerinin hepsi eşdeğerdir. Bu örnekleri kullanmaktan kaçının; ya da birini kullanmanız gerekiyorsa, *P kullanın*.

Pointer Hassasiyet Türleri

Aşağıdaki veri türleri her zaman bir işaretçi boyutundadır; yani 32 bit uygulamalarda 32 bit genişliğinde, 64 bit uygulamalarda ise 64 bit genişliğindedir. Boyut derleme zamanında belirlenir. 32 bit bir uygulama 64 bit Windows üzerinde çalıştığında, bu veri türleri hala 4 bayt genişliğindedir. (64 bit bir uygulama 32 bit Windows üzerinde çalışamaz, bu nedenle tersi bir durum oluşmaz).

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

Bu tipler, bir tamsayının bir işaretçiye dönüştürülebileceği durumlarda kullanılır. Ayrıca işaretçi aritmetiği için değişkenler tanımlamak ve bellek tamponlarındaki tüm bayt aralığı üzerinde yineleme yapan döngü sayaçları tanımlamak için de kullanılırlar. Daha genel olarak, mevcut bir 32 bit değerin 64 bit Windows'ta 64 bite genişletildiği yerlerde görülürler.

Macarca Notasyon

Macar notasyonu, değişken hakkında ek bilgi vermek için değişkenlerin adlarına ön ekler

ekleme uygulamasıdır. (Bu notasyonun mucidi Charles Simonyi



Macarca, adı da buradan geliyor).

Orijinal biçiminde, Macar notasyonu bir değişken hakkında *anlamsal* bilgi verir ve size kullanım amacını söyler. Örneğin, *i* bir indeks, *cb* bayt cinsinden bir boyut ("bayt sayısı"), *rw* ve *col* ise satır ve sütun numaraları anlamına gelir. Bu ön ekler, bir değişkenin yanlış bağlamda yanlışlıkla kullanılmasını önlemek için tasarlanmıştır.

Örneğin, `rwPosition + cbTable` ifadesini görseydiniz, bir satır numarasının bir boyuta eklendiğini bilirdiniz, ki bu neredeyse kesinlikle koddaki bir hatadır

Macarca gösterimin daha yaygın bir biçimi, *tür* bilgisi vermek için örnekleri kullanır; örneğin, **DWORD** için *dw* ve **WORD** için *w*.

7 Not

C++ Çekirdek Yönergeleri  örnek gösteriminden (örneğin Macar gösterimi) kaçının. Bkz. **NL.5: Adlarda tür bilgisini kodlamaktan kaçının** . Dahili olarak, Windows ekibi artık bunu kullanmıyor. Ancak örneklerde ve belgelerde kullanımı devam etmektedir.

Sonraki

[Dizelerle Çalışma](#)

Dizelerle Çalışma

Makale - 23/08/2022 - Okumak için 2 dakika

Windows, kullanıcı arabirimi öğeleri, dosya adları ve benzerleri için Unicode dizelerini yerel olarak destekler. Unicode, tüm karakter kümelerini ve dilleri desteklediği için tercih edilen karakter kodlamasıdır. Windows, Unicode karakterlerini, her karakterin bir veya iki 16 bit değer olarak kodlandığı UTF-16 kodlamasını kullanarak temsil eder. UTF-16 karakterleri, 8 bit ANSI karakterlerinden ayırt etmek için *geniş karakterler olarak* adlandırılır. Visual C++ derleyicisi, geniş karakterler için yerleşik veri türü **wchar_t**'yi destekler. WinNT.h başlık dosyası ayrıca aşağıdaki **typedef**'i tanımlar.

C++

```
typedef wchar_t WCHAR;
```

MSDN örnek kodunda her iki sürümü de göreceksiniz. Geniş karakterli bir değişmez veya geniş karakterli bir dize değişmezi bildirmek için değişmezin önüne **L** koyun.

C++

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

İşte göreceğiniz string ile ilgili diğer bazı typedefler:

Typedef	Tanım
CHAR	char
PSTR veya LPSTR	char*
PCSTR veya LPCSTR	const char*
PWSTR veya LPWSTR	wchar_t*
PCWSTR veya LPCWSTR	const wchar_t*

Unicode ve ANSI İşlevleri

Microsoft, Windows'a Unicode desteği getirdiğinde, biri ANSI dizeleri diğeri Unicode için olmak üzere iki paralel API seti sağlayarak geçişi kolaylaştırdı

dizeleri. Örneğin, bir pencerenin başlık çubuğunun metnini ayarlamak için iki fonksiyon vardır:

- ♦ **SetWindowTextA** bir ANSI dizesi alır.
- ♦ **SetWindowTextW** bir Unicode dizesi alır.

Dahili olarak, ANSI sürümü dizeyi Unicode'a çevirir. Windows başlıkları ayrıca, önişlemci sembolü aşağıdaki durumlarda Unicode sürümüne çözümleyen bir makro tanımlar

UNICODE tanımlanmışsa veya aksi takdirde ANSI sürümü.

C++

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

MSDN'de fonksiyon **SetWindowText** adı altında belgelenmiştir, ancak bu gerçek fonksiyon adı değil makro adıdır.

Yeni uygulamalar her zaman Unicode sürümlerini çağırmalıdır. Birçok dünya dili Unicode gerektirir. ANSI dizeleri kullanırsanız, uygulamanızı yerelleştirmek imkansız olacaktır. ANSI sürümleri ayrıca daha az verimlidir, çünkü işletim sistemi ANSI dizelerini çalışma zamanında Unicode'a dönüştürmek zorundadır. Tercihinize bağlı olarak, **SetWindowTextW** gibi Unicode işlevlerini açıkça çağırabilir veya makroları kullanabilirsiniz. MSDN'deki örnek kod genellikle makroları çağırır, ancak iki form da tam olarak eşdeğerdir. Windows'taki çoğu yeni API'nin yalnızca Unicode sürümü vardır, karşılık gelen ANSI sürümü yoktur.

TCHAR'lar

Uygulamaların hem Windows NT'yi hem de Windows 95, Windows 98 ve Windows Me'yi desteklemesi gerektiği zamanlarda, hedef platforma bağlı olarak aynı kodu ANSI ya da Unicode dizeleri için derlemek faydalı oluyordu. Bu amaçla, Windows SDK, platforma bağlı olarak dizeleri Unicode veya ANSI ile eşleyen makrolar sağlar.

Makro	Unicode	ANSI
TCHAR	wchar_t	char
TEXT("x") veya _T("x")	L "x"	"x"

Örneğin, aşağıdaki kod:

C++

```
SetWindowText(TEXT("Uygulamam"));
```

aşağıdakilerden birine çözümlenir:

C++

```
SetWindowTextW(L "My Application"); // Geniş karakterli dize ile Unicode işlevi.
```

```
SetWindowTextA("Benim Uygulamam"); // ANSI fonksiyonu.
```

TEXT ve **TCHAR** makroları günümüzde daha az kullanışlıdır, çünkü tüm uygulamalar Unicode kullanmalıdır. Ancak, bunları eski kodlarda ve bazı MSDN kod örneklerinde görebilirsiniz.

Microsoft C çalışma zamanı kütüphanelerinin başlıkları benzer bir makro kümesi tanımlar. İçin Örneğin, **_tcslen** tanımlanmamışsa **strlen** olarak çözümlenir; aksi takdirde **wcslen**, **strlen**'in geniş karakterli sürümüdür.

C++

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

Dikkatli olun: Bazı başlıklar **UNICODE** önışlemci sembolünü kullanırken diğerle **_UNICODE** alt çizgi önekiyle birlikte. Her zaman her iki sembolü de tanımlayın. Visual C++, yeni bir proje oluşturduğunuzda varsayılan olarak her ikisini de ayarlar.

Sonraki

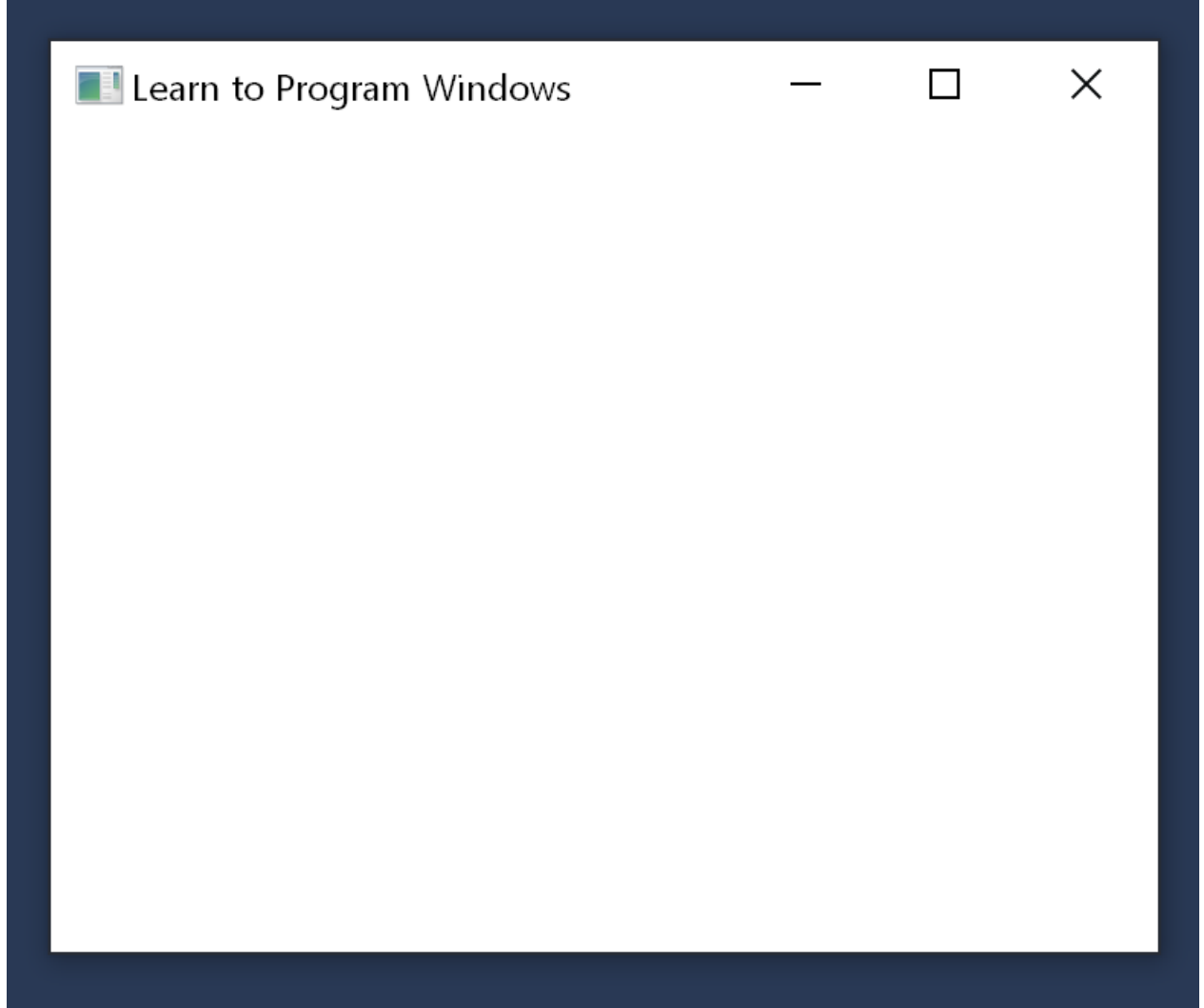
[Pencere Nedir?](#)

Pencere Nedir?

Makale - 04/27/2021 - Okumak için 3 dakika

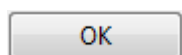
Pencere Nedir?

Açıkçası, pencereler Windows'un merkezinde yer almaktadır. O kadar önemlidirler ki işletim sistemine onların adını vermişlerdir. Ama pencere nedir? Bir pencere düşündüğünüzde muhtemelen aklınıza şöyle bir şey geliyordur:



Bu tür pencerelere *uygulama penceresi* veya *ana* pencere denir. Genellikle başlık çubuğu, **Simge Durumuna Küçült** ve **Büyüt** düğmeleri ve diğer standart kullanıcı arabirimi öğeleri içeren bir çerçeveye sahiptir. Çerçeve, pencerenin istemci *olmayan alanı* olarak *adlandırılır* ve işletim sistemi pencerenin bu bölümünü yönettiği için böyle adlandırılır. Çerçeve içindeki alan *istemci alanıdır*. Bu, pencerenin programınız tarafından yönetilen kısmıdır.

İşte başka bir pencere türü:



Windows programlamada yeniyseniz, düğmeler ve düzenleme kutuları gibi kullanıcı arayüzü denetimlerinin kendilerinin birer pencere olması sizi şaşırtabilir. Bir UI denetimi ile bir uygulama penceresi arasındaki en önemli fark, bir denetimin kendi başına var olmamasıdır. Bunun yerine, denetim uygulama penceresine göre konumlandırılır. Uygulama penceresini sürüklediğinizde, beklediğiniz gibi denetim de onunla birlikte hareket eder. Ayrıca, denetim ve uygulama penceresi birbirleriyle iletişim kurabilir. (Örneğin, uygulama penceresi bir düğmeden tıklama bildirimleri alır).

Bu nedenle, *pencereyi* düşündüğünüzde, sadece *uygulama penceresini düşünmeyin*. Bunun yerine, pencereyi bir programlama yapısı olarak düşünün:

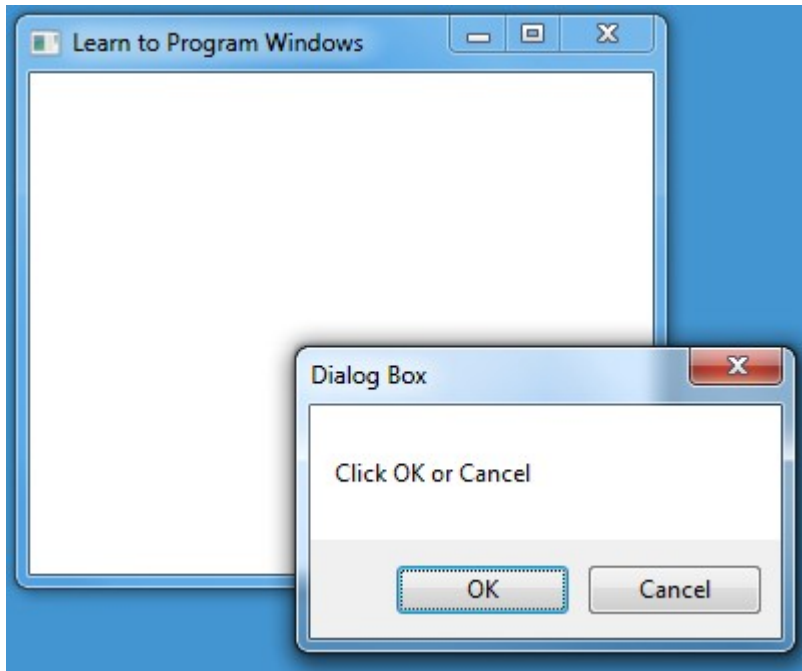
- ♦ Ekranın belirli bir bölümünü kaplar.
- ♦ Belirli bir anda görünür olabilir veya olmayabilir.
- ♦ Kendini nasıl çizeceğini bilir.
- ♦ Kullanıcıdan veya işletim sisteminden gelen olaylara yanıt verir.

Ebeveyn Pencereleeri ve Sahip Pencereleeri

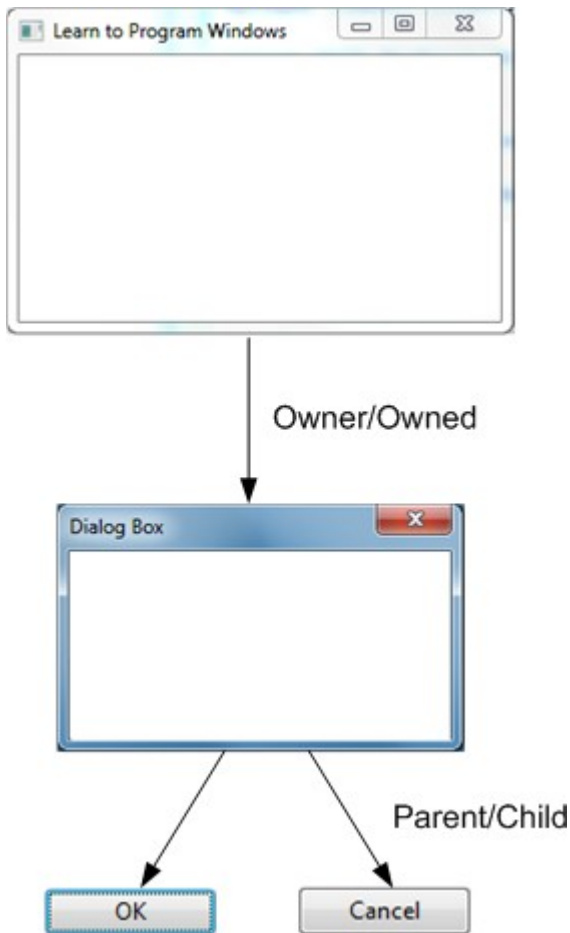
Bir UI kontrolü söz konusu olduğunda, kontrol penceresinin uygulama penceresinin *alt ögesi* olduğu söylenir. Uygulama penceresi, kontrol penceresinin *ebeveynidir*. Üst pencere, alt pencereyi konumlandırmak için kullanılan koordinat sistemini sağlar. Bir üst pencereye sahip olmak, bir pencerenin görünümünü etkiler; örneğin, bir alt pencere kırpılır, böylece alt pencerenin hiçbir parçası üst penceresinin sınırları dışında görünemez.

Bir başka ilişki de uygulama penceresi ile modal iletişim penceresi arasındaki ilişkidir. Bir uygulama modal bir iletişim kutusu görüntülediğinde, uygulama penceresi *sahip penceresidir* ve iletişim kutusu da *sahip olunan bir pencere*dir. Sahip olunan bir pencere her zaman sahibi olan pencerenin önünde görünür. Sahibi simge durumuna küçültüldüğünde gizlenir ve sahibiyile aynı anda yok edilir.

Aşağıdaki görüntüde iki düğmeli bir iletişim kutusu görüntüleyen bir uygulama gösterilmektedir:



Uygulama penceresi iletişim penceresinin sahibidir ve iletişim penceresi her iki düğme penceresinin de ebeveynidir. Aşağıdaki diyagram bu ilişkileri göstermektedir:



Pencere Kolları

Pencereler nesnelerdir - hem kod hem de veriye sahiptirler - ancak C++ sınıfları değildirler. Bunun yerine, bir program *tanıtıcı* adı verilen bir değer kullanarak bir pencereye başvurur. Tanıtıcı bir

opak tip. Esasen, işletim sisteminin bir nesneyi tanımlamak için kullandığı bir sayıdır. Windows'u, oluşturulmuş tüm pencerelerin büyük bir tablosuna sahip olarak hayal edebilirsiniz. Pencereleri tutamaçlarına göre aramak için bu tabloyu kullanır. (Dahili olarak tam olarak böyle çalışıp çalışmadığı önemli değildir.) Pencere tanıtıcıları için veri türü **HWND'dir ve** genellikle "aitch-wind" olarak telaffuz edilir. Pencere tanıtıcıları, pencereleri oluşturan fonksiyonlar tarafından döndürülür: **CreateWindow** ve **CreateWindowEx**.

Bir pencere üzerinde bir işlem gerçekleştirmek için, genellikle parametre olarak bir **HWND** değeri alan bir işlevi çağırırsınız. Örneğin, bir pencereyi ekranda yeniden konumlandırmak için **MoveWindow** işlevini çağırın:

C++

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

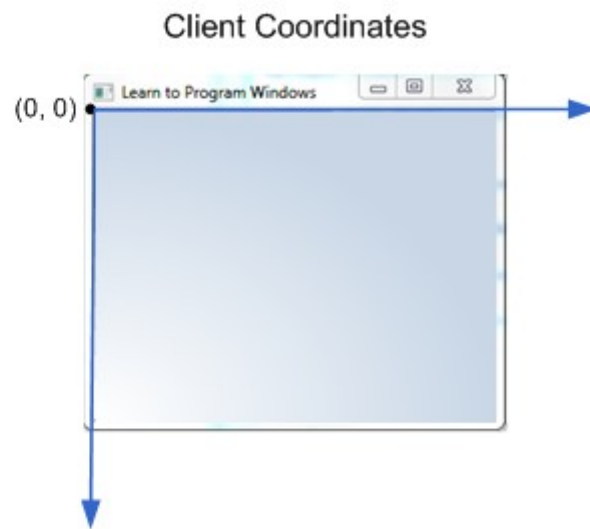
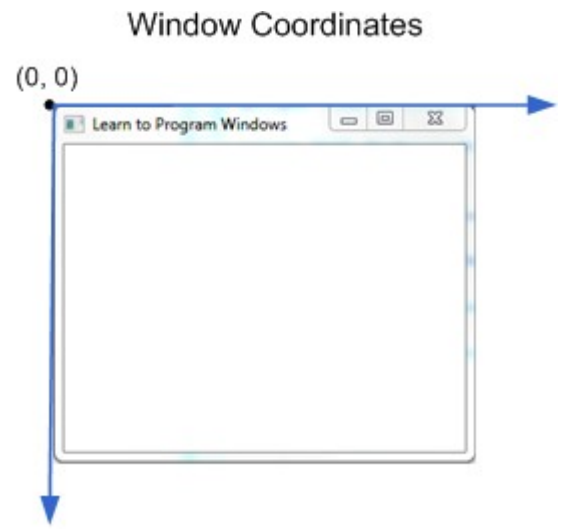
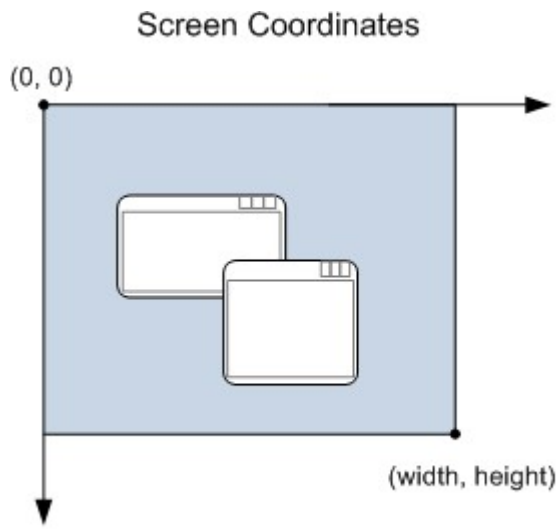
İlk parametre, taşımak istediğiniz pencerenin tutamacıdır. Diğer parametreler pencerenin yeni konumunu ve pencerenin yeniden çizilip çizilmeyeceğini belirtir.

Tutamaçların işaretçi olmadığını unutmayın. Eğer *hWnd* bir tanıtıcı içeren bir değişken ise, `*hWnd` yazarak tanıtıcıya referans vermeye çalışmak bir hatadır.

Ekran ve Pencere Koordinatları

Koordinatlar cihazdan bağımsız piksellerle ölçülür. *Cihazdan bağımsız piksellerin cihazdan bağımsız* kısmı hakkında grafikleri tartışırken daha fazla şey söyleyeceğiz.

Görevinize bağlı olarak, koordinatları ekrana göre, bir pencereye göre (çerçeve dahil) veya bir pencerenin istemci alanına göre ölçekbilirsiniz. Örneğin, ekran koordinatlarını kullanarak ekranda bir pencere konumlandırırsınız, ancak istemci koordinatlarını kullanarak bir pencerenin içine çizim yaparsınız. Her durumda, başlangıç noktası (0, 0) her zaman bölgenin sol üst köşesidir.



Sonraki

[WinMain: Uygulama Giriş Noktası](#)

WinMain uygulama giriş noktası

Makale - 03/10/2023 - Okumak için 2 dakika

Her Windows programı **WinMain** ya da **wWinMain**. Aşağıdaki kod **wWinMain** için imzayı göstermektedir:

C++

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

Dört **wWinMain** parametresi aşağıdaki gibidir:

- *hInstance* bir *örneğin tanıtıcısı* veya bir modülün tanıtıcısıdır. İşletim sistemi, belleğe yüklendiğinde yürütülebilir dosyayı veya EXE'yi tanımlamak için bu değeri kullanır.
Bazı Windows işlevleri, örneğin simgeleri veya bitmapleri yüklemek için örnek tanıtıcıya ihtiyaç duyar.
- *hPrevInstance*'in bir anlamı yoktur. 16-bit Windows'ta kullanılıyordu, ancak artık her zaman sıfırdır.
- *pCmdLine*, Unicode dizesi olarak komut satırı argümanlarını içerir. *nCmdShow*,
- ana uygulama penceresinin simge durumuna küçültüldüğünü, büyütüldüğünü veya normal şekilde gösterildiğini belirten bir bayraktır.

Fonksiyon bir değer döndürür . İşletim sistemi geri dönüş değerini kullanmaz, ancak değeri başka bir programa bir durum kodu iletmek için kullanabilirsiniz.

WINAPI gibi bir *çağırma kuralı*, bir işlevin çağırandan nasıl parametre alacağını tanımlar. Örneğin, çağrı kuralı parametrelerin yığında görünme sırasını tanımlar. **wWinMain** fonksiyonunuzu önceki örnekte gösterildiği gibi bildirdiğinizden emin olun.

WinMain işlevi, komut satırı bağımsız değişkenlerinin ANSI dizesi olarak geçirilmesi dışında **wWinMain** ile aynıdır. Unicode dizesi tercih edilir. Programınızı Unicode olarak derleseniz bile ANSI **WinMain işlevini** kullanabilirsiniz. Komut satırı bağımsız değişkenlerinin Unicode kopyasını almak için **GetCommandLine** işlevini çağırın. Bu fonksiyon tüm bağımsız değişkenleri tek bir dizide döndürür. *Argümanlar/argv tarzı* bir dizi olarak istiyorsanız, bu dizeyi **CommandLineToArgvW** ögesine aktarın.

Derleyici standart **main** işlevi yerine **wWinMain**'i çağıracağını nereden biliyor? Aslında olan şey, Microsoft C çalışma zamanı kütüphanesinin (CRT) **WinMain** veya **wWinMain**'i çağırarak bir **main** uygulaması sağlamasıdır.

CRT **main** içinde biraz daha fazla iş yapar. Örneğin, **wWinMain**'den önce tüm statik başlatıcıları çağırır. Bağlayıcıya farklı bir giriş noktası işlevi kullanmasını söyleyebilmenize rağmen, CRT'ye bağlarsanız varsayılanı kullanmalısınız. Aksi takdirde, CRT başlatma kodu atlanır ve global nesnelerin doğru şekilde başlatılmaması gibi öngörülemeyen sonuçlar ortaya çıkar.

Aşağıdaki kodda boş bir **WinMain** işlevi gösterilmektedir:

C++

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

Artık giriş noktasına sahip olduğunuza ve bazı temel terminoloji ve kodlama kurallarını anladığınıza göre, [ilk Windows programınızı oluşturmaya](#) hazırsınız.

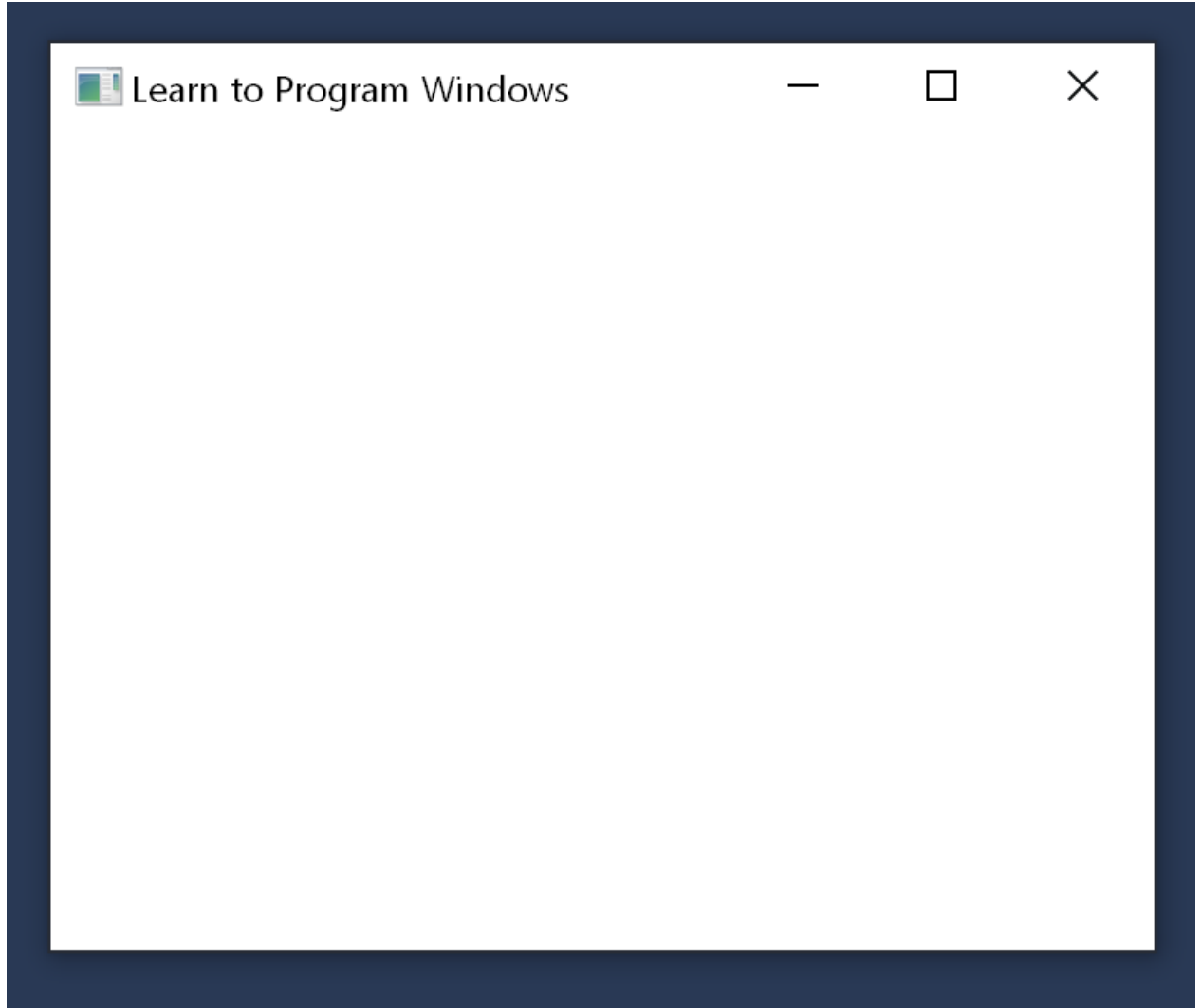
Modül 1. İlk Windows Programınız

Makale - 06/13/2022 - Okumak için 2 dakika

Bu modülde, minimal bir Windows masaüstü programı yazacağız. Tek yaptığı boş bir pencere oluşturmak ve göstermek. Bu ilk program, boş satırları ve yorumları saymazsak yaklaşık 50 satır kod içeriyor. Bu bizim başlangıç noktamız olacak; daha sonra grafik, metin, kullanıcı girişi ve diğer özellikleri ekleyeceğiz.

Visual Studio 'da geleneksel bir Windows masaüstü uygulamasının nasıl oluşturulacağı hakkında daha fazla ayrıntı arıyorsanız, [Walkthrough 'a](#) göz atın:

[Geleneksel Windows Masaüstü uygulaması oluşturma \(C++\)](#).



İşte programın tam kodu:

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)  
{
```

```
    // Pencere sınıfını kaydedin.
```

```
    const wchar_t SINIF_ADI[] = L "Örnek Pencere Sınıfı";
```

```
    WNDCLASS wc = { };
```

```
    wc.lpfnWndProc = WindowProc;
```

```
    wc.hInstance = hInstance;
```

```
    wc.lpszClassName = CLASS_NAME;
```

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

```
    // Pencereyi oluşturun.
```

```
    HWND hwnd = CreateWindowEx(
```

```
        0, // İsteğe bağlı pencere stilleri.
```

```
        SINIF_ADI, // Pencere sınıfı
```

```
        L "Windows Programlamayı Öğrenin", // Pencere
```

```
        metni WS_OVERLAPPEDWINDOW, // Pencere stili
```

```
        // Boyut ve konum
```

```
        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
```

```
        BOŞ, // Ana pencere
```

```
        BOŞ, // Menü
```

```
        hInstance, // Örnek tanıtıcı
```

```
        NULL // Ek uygulama verileri
```

```
    );
```

```
    eğer (hwnd == NULL)
```

```
    {
```

```
        0 döndür;
```

```
    }
```

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

```
    // Mesaj döngüsünü
```

```
    çalıştırın. MSG msg = { };
```

```
    while (GetMessage(&msg, NULL, 0, 0) > 0)
```

```
    {
```

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

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

```
    }
```

```
    0 döndür;
```

}

LRESULT CALLBACK [WindowProc](#)(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM

```

IParam)
{
    switch (uMsg)
    {
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;

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

                // Tüm boyama işlemi burada, BeginPaint ve EndPaint arasında

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

                EndPaint(hwnd, &ps);
            }
            0 döndür;

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

```

Visual Studio projesinin tamamını [Windows Hello World Sample](#) adresinden indirebilirsiniz.

Bu kodun ne işe yaradığına dair kısa bir özet vermek faydalı olabilir. Daha sonraki konular kodu ayrıntılı olarak inceleyecektir.

1. **wWinMain** program giriş noktasıdır. Program başladığında, uygulama penceresinin davranışı hakkında bazı bilgileri kaydeder. En çok kullanılanlardan biri önemli öğeler, bu örnekte `WindowPr` adı verilen bir işlevin adresidir. Bu işlev pencerenin davranışını tanımlar; görünümü, kullanıcıyla nasıl etkileşime gireceği vb.
2. Ardından, program pencereyi oluşturur ve pencereyi benzersiz bir şekilde tanımlayan bir tanıtıcı alır.
3. Pencere başarıyla oluşturulursa, program bir **while** döngüsüne girer. Kullanıcı pencereyi kapatıp uygulamadan çıkana kadar program bu döngüde kalır.

Programın fonksiyonu açıkça çağırmadığına dikkat `wWinMain`, ancak Uygulama mantığının çoğunun burada tanımlandığını söylemiştik. Windows, programınıza bir dizi *mesaj* ileterek onunla iletişim kurar. **While** döngüsünün içindeki kod bu süreci yönlendirir. Program **DispatchMessage** fonksiyonunu her çağırdığında,

dolaylı olarak Windows'un WindowProc fonksiyonunu her mesaj için bir kez çağırmasına neden olur.

Bu bölümde

- ♦ [Pencere](#)
- ♦ [Oluşturma](#)
[Pencere Mesajları](#)
- ♦ [Pencere Prosedürünün Yazılması](#)
- ♦ [Pencerenin Boyanması](#)
- ♦ [Pencereyi Kapatma](#)
- ♦ [Uygulama Durumunu](#)
[Yönetme](#)

İlgili konular

[C++ ile Windows için Programlamayı](#)

[Öğrenin Windows Merhaba Dünya](#)

[Örneği](#)

Bir pencere oluřturun

Makale - 03/09/2023 - 4 dakika okumak için

Bu makalede, bir pencere oluřturmayı ve göstermeyi öğrenin.

Pencere sınıfları

Bir *pencere sınıfı*, birkaç pencerenin ortak sahip olabileceđi bir dizi davranıřı tanımlar. Örneđin, bir grup düğmede, kullanıcı düğmeyi seçtiğinde her düğmenin benzer bir davranıřı vardır. Elbette, düğmeler tamamen aynı deđildir. Her düğme kendi metin dizesini görüntüler ve kendi ekran koordinatlarına sahiptir. Her pencere için benzersiz olan verilere *örnek veriler* denir.

Programınız bu sınıfın yalnızca bir örneđini oluřtursa bile, her pencere bir pencere sınıfıyla ilişkilendirilmelidir. Pencere sınıfı C++ anlamında bir sınıf deđildir. Daha ziyade, işletim sistemi tarafından dahili olarak kullanılan bir veri yapısıdır. Pencere sınıfları çalışma zamanında sisteme kaydedilir. Yeni bir pencere sınıfı kaydetmek için, bir **WNDCLASS** yapısı doldurun:

C++

```
// Pencere sınıfını kaydedin.  
const wchar_t SINIF_ADI[] = L "Örnek Pencere Sınıfı";  
  
WNDCLASS wc = { };  
  
wc.lpfnWndProc = WindowProc;  
wc.hInstance = hInstance;  
wc.lpszClassName = CLASS_NAME;
```

Ařađıdaki yapı üyelerini ayarlamanız gerekir:

- ♦ **lpfnWndProc**, *pencere* olarak adlandırılan uygulama tanımlı bir işleve işaretdir *prosedürü* veya *pencere proc*. Pencere prosedürü, pencerenin davranıřının çođunu tanımlar. řimdilik, bu deđer bir fonksiyonun ileri bildirimidir. Daha fazla bilgi için [Pencere prosedürünün yazılması](#) bölümüne bakın.
- ♦ **hInstance**, uygulama örneđinin tanıtıcısıdır. Bu deđeri řuradan alın `wWinMain`'in *hInstance* parametresi.
- ♦ **lpszClassName**, pencere sınıfını tanımlayan bir dizedir.

Sınıf adları geçerli süreç için yereldir, bu nedenle adın yalnızca süreç içinde benzersiz olması gerekir. Ancak, standart Windows denetimlerinin de sınıfları vardır.

Eğer siz

Bu kontrollerden herhangi birini kullanmak için, kontrol sınıf adlarıyla çakışmayan sınıf adları seçmelisiniz. Örneğin, düğme denetimi için pencere sınıfının adı *Button'dır*.

WNDCLASS yapısının burada gösterilmeyen başka üyeleri de vardır. Bu örnekte gösterildiği gibi bunları sıfır olarak ayarlayabilir veya doldurabilirsiniz. Daha fazla bilgi için [WNDCLASS](#) bölümüne bakın.

Ardından, **WNDCLASS** yapısının adresini [RegisterClass](#) fonksiyonuna aktarın. Bu fonksiyon, pencere sınıfını işletim sistemine kaydeder.

C++

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

Pencere oluşturma

Yeni bir pencere örneği oluşturmak için [CreateWindowEx](#) işlevini çağırın:

C++

```
HWND hwnd = CreateWindowEx(  
    0,                                     // İsteğe bağlı pencere  
    SINIF_ADI,                             stilleri.  
    L "Learn to Program Windows",         // Pencere sınıfı  
    WS_OVERLAPPEDWINDOW,                 // Pencere metni  
  
    // Boyut ve konum  
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,  
  
    BOŞ,                                  // Ana pencere  
    BOŞ,                                  // Menü  
    hInstance,                            // Örnek tanıtıcı  
    NULL,                                  // Ek uygulama verileri  
);  
  
eğer (hwnd == NULL)  
,
```

Ayrıntılı parametre açıklamaları için [CreateWindowEx](#) bölümüne bakın. İşte hızlı bir özet:

- İlk parametre, pencere için bazı isteğe bağlı davranışlar (örneğin, saydam pencereler) belirlemenizi sağlar. Varsayılan davranışlar için bu parametreyi sıfır olarak ayarlayın.
- `SINIF_ADI` pencere sınıfının adıdır. Bu isim pencerenin türünü tanımlar

yaratmak için.

- Pencere metni farklı pencere türleri tarafından farklı şekillerde kullanılır. Pencerenin bir başlık çubuğu varsa, metin başlık çubuğunda görüntülenir.
- Pencere stili, bir pencerenin görünüm ve hissinin bir kısmını tanımlayan bir dizi bayraktır. **WS_OVERLAPPEDWINDOW** sabiti aslında bitwise OR ile birleştirilmiş birkaç bayraktır. Bu bayraklar birlikte pencereye bir başlık çubuğu, bir kenarlık, bir sistem menüsü ve **Simge Durumuna Küçült** ve **Büyüt** düğmeleri. Bu bayrak kümesi, üst düzey bir uygulama penceresi için en yaygın stildir.
- Konum ve boyut için **CW_USEDEFAULT** sabiti varsayılan değerlerin kullanılacağı anlamına gelir. Sonraki parametre, yeni pencere için bir üst pencere veya sahip penceresi ayarlar. Bir alt pencere oluşturmak için üst pencereyi ayarlayın. Üst düzey bir pencere için bu değeri **NULL**.
- Bir uygulama penceresi için, bir sonraki parametre pencerenin menüsünü tanımlar. Bu örnekte bir menü kullanılmadığından değer **NULL**'dur.
- *hInstance*, daha önce açıklanan örnek tanıtıcısıdır. [WinMain: Uygulama Giriş Noktası bölümüne](#) bakın.
- Son parametre, **void*** türünde rastgele verilere bir işaretçidir. Bunu kullanabilirsiniz değerini pencere yordamınıza bir veri yapısı iletmek için kullanabilirsiniz. Bu parametreyi kullanmanın olası bir yolu için [Uygulama Durumunu Yönetme](#) bölümüne bakın.

CreateWindowEx yeni pencereye bir tanıtıcı döndürür veya işlev başarısız olursa sıfır döndürür. Pencereyi göstermek, yani pencereyi görünür kılmak için, pencere tanıtıcısını **ShowWindow** fonksiyonuna aktarın:

C++

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

hwnd parametresi **CreateWindowEx** tarafından döndürülen pencere tanıtıcısıdır. *nCmdShow* parametresi bir pencereyi küçültmek veya büyütmek için kullanılabilir. İşletim sistemi bu değeri **wWinMain** fonksiyonu aracılığıyla programa aktarır.

İşte pencereyi oluşturmak için gereken kodun tamamı. **WindowProc**'un hala sadece bir fonksiyonun ileri bildirimi olduğunu unutmayın.

C++

```
// Pencere sınıfını kaydedin.
```

```
const wchar_t SINIF_ADI[] = L "Örnek Pencere Sınıfı";
```

```
WNDCLASS wc = { };
```

```
wc.lpfnWndProc = WindowProc;
```

```
wc.hInstance = hInstance;
```

```
RegisterClass(&wc);

// Pencereyi oluşturun.

HWND hwnd = CreateWindowEx( 0,
    SINIF_ADI,
    L "Learn to Program Windows",
    WS_OVERLAPPEDWINDOW,
    // İsteğe bağlı pencere stilleri.
    // Pencere sınıfı
    // Pencere metni

    // Boyut ve konum
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT,

    BOŞ,
    // Ana pencere
    BOŞ,
    // Menü
    hInstance,
    // Örnek tanıtıcı
    NULL
    // Ek uygulama verileri
);

eğer (hwnd == NULL)
{
    0 döndür;
```

Tebrikler, bir pencere yarattınız!

Şu anda, pencere herhangi bir içerik içermiyor veya kullanıcıyla etkileşime girmiyor. Gerçek bir GUI uygulamasında, pencere kullanıcıdan ve işletim sisteminden gelen olaylara yanıt verecektir. Bir sonraki bölümde pencere mesajlarının bu tür bir etkileşimi nasıl sağladığı açıklanmaktadır.

Ayrıca bakınız

Bu modüle devam etmek için [Pencere Mesajları](#) bölümüne geçin.

Pencere Mesajları (Win32 ve C++ ile Başlarken)

Makale - 04/27/2021 - Okumak için 5 dakika

Bir GUI uygulaması kullanıcıdan ve işletim sisteminden gelen olaylara yanıt vermelidir.

- **Kullanıcıdan gelen olaylar, bir kişinin** programınızla etkileşime girebileceği tüm yolları içerir: fare tıklamaları, tuş vuruşları, dokunmatik ekran hareketleri vb.
- **İşletim sisteminden gelen olaylar**, programın nasıl davrandığını etkileyebilecek programın "dışındaki" her şeyi içerir. Örneğin, kullanıcı yeni bir donanım aygıtı takabilir veya Windows daha düşük bir güç durumuna (uyku veya hazırda bekleme) girebilir.

Bu olaylar program çalışırken herhangi bir zamanda, neredeyse herhangi bir sırada meydana gelebilir. Yürütme akışı önceden tahmin edilemeyen bir programı nasıl yapılandırırsınız?

Bu sorunu çözmek için Windows bir mesaj geçirme modeli kullanır. İşletim sistemi, uygulama pencerenize mesajlar ileterek onunla iletişim kurar. Bir mesaj basitçe belirli bir olayı belirten sayısal bir koddur. Örneğin, kullanıcı sol fare düğmesine basarsa, pencere aşağıdaki mesaj koduna sahip bir mesaj alır.

C++

```
#define WM_LBUTTONDOWN 0x0201
```

Bazı mesajlar kendileriyle ilişkili verilere sahiptir. Örneğin, **WM_LBUTTONDOWN** mesajı fare imlecinin x-koordinatını ve y-koordinatını içerir.

Bir pencereye mesaj iletmek için, işletim sistemi o pencere için kayıtlı pencere prosedürünü çağırır. (Ve şimdi pencere prosedürünün ne için olduğunu biliyorsunuz).

Mesaj Döngüsü

Bir uygulama çalışırken binlerce mesaj alacaktır. (Her tuş vuruşunun ve fare düğmesi tıklamasının bir mesaj oluşturduğunu düşünün.) Ayrıca, bir uygulamanın her biri kendi pencere prosedürüne sahip birkaç penceresi olabilir. Program tüm bu mesajları nasıl alır ve doğru pencere prosedürüne nasıl iletir? Cevap

uygulamasının mesajları almak ve doğru pencerelere göndermek için bir döngüye ihtiyacı vardır.

Bir pencere oluşturan her iş parçacığı için işletim sistemi pencere mesajları için bir kuyruk oluşturur. Bu kuyruk, o iş parçacığında oluşturulan tüm pencereler için mesajları tutar. Kuyruğun kendisi programınızdan gizlenir. Kuyruğu doğrudan manipüle edemezsiniz. Ancak **GetMessage** fonksiyonunu çağırarak kuyruktan bir mesaj çekebilirsiniz.

C++

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

Bu fonksiyon kuyruğun başındaki ilk mesajı kaldırır. Kuyruk boşsa, başka bir mesaj kuyruğa alınana kadar fonksiyon bloke olur. **GetMessage**'in bloke olması programınızı yanıt veremez hale getirmez. Mesaj yoksa, programın yapacağı hiçbir şey yoktur. Arka planda işlem yapmanız gerekiyorsa, **GetMessage** başka bir mesaj beklerken çalışmaya devam eden ek iş parçacıkları oluşturabilirsiniz. ([Pencere Yordamınızda Darboğazları Önleme bölümüne](#) bakın).

GetMessage'in ilk parametresi bir **MSG** yapısının adresidir. Fonksiyon başarılı olursa, **MSG** yapısını mesajla ilgili bilgilerle doldurur. Buna hedef pencere ve mesaj kodu da dahildir. Diğer üç parametre kuyruktan hangi mesajları alacağınızı filtrelemenizi sağlar. Neredeyse tüm durumlarda, bu parametreleri sıfır olarak ayarlayacaksınız.

MSG yapısı mesaj hakkında bilgi içermesine rağmen, bu yapıyı neredeyse hiçbir zaman doğrudan incelemeyeceksiniz. Bunun yerine, doğrudan diğer iki fonksiyona aktaracaksınız.

C++

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

TranslateMessage işlevi klavye girişi ile ilgilidir. Tuş vuruşlarını (tuş aşağı, tuş yukarı) karakterlere çevirir. Bu fonksiyonun nasıl çalıştığını bilmek zorunda değilsiniz; sadece **DispatchMessage**'dan önce çağırmayı unutmayın. Merak ediyorsanız, MSDN belgelerine bağlantı size daha fazla bilgi verecektir.

DispatchMessage işlevi işletim sistemine mesajın hedefi olan pencerenin pencere yordamını çağırmasını söyler. Başka bir deyişle, işletim sistemi

pencere tanıtıcısını pencere tablosunda arar, pencereyle ilişkili işlem işaretçisini bulur ve işlevi çağırır.

Örneğin, kullanıcının sol fare düğmesine bastığını varsayalım. Bu bir olaylar zincirine neden olur:

1. İşletim sistemi mesaj kuyruğuna bir **WM_LBUTTONDOWN** mesajı koyar.
2. Programınız **GetMessage** işlevini çağırır.
3. **GetMessage** kuyruktan **WM_LBUTTONDOWN** mesajını çeker ve **MSG** yapısını doldurur.
4. Programınız **TranslateMessage** ve **DispatchMessage** işlevlerini çağırır.
5. **DispatchMessage** içinde, işletim sistemi pencere yordamınızı çağırır.
6. Pencere yordamınız mesaja yanıt verebilir ya da mesajı yok sayabilir.

Pencere yordamı geri döndüğünde, **DispatchMessage**'a geri döner. Bu da bir sonraki mesaj için mesaj döngüsüne geri döner. Programınız çalıştığı sürece, mesajlar kuyruğa gelmeye devam edecektir. Bu nedenle, mesajları sürekli olarak kuyruktan çeken ve gönderen bir döngüye sahip olmalısınız. Döngünün aşağıdakileri yaptığını düşünebilirsiniz:

C++

// UYARI: Döngünüzü aslında bu şekilde yazmayın.

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

Yazıldığı gibi, elbette bu döngü asla bitmeyecektir. **GetMessage** fonksiyonunun geri dönüş değeri burada devreye girer. Normalde **GetMessage** sıfır olmayan bir değer döndürür. Uygulamadan çıkmak ve mesaj döngüsünden çıkmak istediğinizde, **PostQuitMessage** fonksiyonunu çağırın.

C++

```
PostQuitMessage(0);
```

PostQuitMessage işlevi mesaj kuyruğuna bir **WM_QUIT** mesajı koyar. **WM_QUIT** özel bir mesajdır: **GetMessage**'in sıfır döndürmesine neden olarak mesaj döngüsünün sonunu işaret eder. İşte revize edilmiş mesaj döngüsü.

C++

// Doğru.

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

GetMessage sıfır olmayan bir değer döndürdüğü sürece, **while** döngüsündeki ifade true olarak değerlendirilir. **PostQuitMessage** ögesini çağırdıktan sonra, ifade yanlış olur ve program döngüden çıkar. (Bu davranışın ilginç bir sonucu, pencere yordamınızın asla bir **WM_QUIT** mesajı almamasıdır. Bu nedenle, pencere yordamınızda bu mesaj için bir case deyimine sahip olmanız gerekmez).

Bir sonraki bariz soru **PostQuitMessage**'in ne zaman çağrılacağıdır. Bu soruya **Pencereyi Kapatma** konusunda döneceğiz, ancak önce pencere prosedürümüzü yazmamız gerekiyor.

Gönderilen Mesajlara Karşı Gönderilen Mesajlar

Önceki bölümde bir kuyruğa giden mesajlardan bahsedilmişti. Bazen, işletim sistemi kuyruğu atlayarak doğrudan bir pencere yordamını çağırır.

Bu ayrıma ilişkin terminoloji kafa karıştırıcı olabilir:

- Mesaj *göndermek*, mesajın mesaj kuyruğuna girmesi ve mesaj döngüsü (**GetMessage** ve **DispatchMessage**) aracılığıyla gönderilmesi anlamına gelir. Mesaj
- *göndermek*, mesajın kuyruğu atlaması ve işletim sisteminin pencere prosedürünü doğrudan çağırması anlamına gelir.

Şimdilik aradaki fark çok önemli değildir. Pencere yordamı tüm mesajları işler. Ancak, bazı mesajlar kuyruğu atlar ve doğrudan pencere yordamınıza gider. Ancak, uygulamanız pencereler arasında iletişim kuruyorsa bu bir fark yaratabilir. Bu konuyla ilgili daha ayrıntılı bir tartışmayı [İletiler ve İleti Kuyrukları Hakkında başlıklı konuda bulabilirsiniz](#).

Sonraki

[Pencere Prosedürünün Yazılması](#)

Pencere Prosedürünün Yazılması

Makale - 04/27/2021 - Okumak için 3 dakika

DispatchMessage fonksiyonu, mesajın hedefi olan pencerenin pencere prosedürünü çağırır. Pencere yordamı aşağıdaki imzaya sahiptir.

C++

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

Dört parametre vardır:

- *hwnd* pencerenin tanıtıcısıdır.
- *uMsg* mesaj kodudur; örneğin, **WM_SIZE** mesajı pencerenin yeniden boyutlandırıldığını gösterir.
- *wParam* ve *lParam* mesajla ilgili ek veriler içerir. Tam anlamı mesaj koduna bağlıdır.

LRESULT, programınızın Windows'a döndürdüğü bir tamsayı değeridir. Programınızın belirli bir mesaja verdiği yanıtı içerir. Bu değerın anlamı mesaj koduna bağlıdır.

CALLBACK, fonksiyon için çağrı kuralını belirtir.

Tipik bir pencere prosedürü, mesaj kodunu değiştiren büyük bir switch deyimidir. İşlemek istediğiniz her mesaj için durumlar ekleyin.

C++

```
switch (uMsg)
{
    case WM_SIZE: // Pencere yeniden boyutlandırmayı ele al

        // vb
}
```

Mesaj için ek veriler *lParam* ve *wParam* parametrelerinde bulunur. Her iki parametre de bir işaretçi genişliği (32 bit veya 64 bit) boyutunda tamsayı değerlerdir. Her birinin anlamı mesaj koduna (*uMsg*) bağlıdır. Her mesaj için, MSDN'de mesaj koduna bakmanız ve parametreleri doğru veri türüne dönüştürmeniz gerekecektir. Genellikle veri ya sayısal bir değer ya da bir yapıya işaretçidir. Bazı mesajlarda herhangi bir veri bulunmaz.

Örneğin, **WM_SIZE** mesajına ilişkin belgelerde şu belirtilmektedir:

- *wParam*, pencerenin simge durumuna küçültüldüğünü, büyütüldüğünü veya yeniden boyutlandırıldığını gösteren bir bayraktır.
- *lParam*, 32 veya 64 bitlik bir sayıya paketlenmiş 16 bitlik değerler olarak pencerenin yeni genişliğini ve yüksekliğini içerir. Bu değerleri elde etmek için biraz bit kaydırma yapmanız gerekecektir. Neyse ki, WinDef.h başlık dosyası bunu yapan yardımcı makrolar içerir.

Tipik bir pencere yordamı düzinelerce mesajı işler, bu nedenle oldukça uzun olabilir. Kodunuzu daha modüler hale getirmenin bir yolu, her bir mesajı işleme mantığını ayrı bir fonksiyona koymaktır. Pencere yordamında, *wParam* ve *lParam* parametrelerini doğru veri türüne dönüştürün ve bu değerleri işleve aktarın. Örneğin, **WM_SIZE** mesajını işlemek için pencere yordamı aşağıdaki gibi görünecektir:

C++

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_SIZE:
        {
            int width = LOWORD(lParam);    // Düşük sıralı kelimeyi almak için
            makro. int height = HIWORD(lParam); // Yüksek sıralı kelimeyi almak
            için makro
            kelime.

            // Mesaja yanıt verin:
            OnSize(hwnd, (UINT)wParam, genişlik, yükseklik);
        }
        Mola;
    }
}

void OnSize(HWND hwnd, UINT flag, int width, int height)
{
    // Yeniden boyutlandırmayı ele al
```

LOWORD ve **HIWORD** makroları 16 bit genişlik ve yükseklik değerlerini *lParam*. (Bu tür ayrıntıları her mesaj kodu için MSDN belgelerinde bulabilirsiniz). Pencere yordamı genişlik ve yüksekliği çıkarır ve ardından bu değerler `OnSize` fonksiyon.

Varsayılan Mesaj İşleme

Pencere yordamınızda belirli bir mesajı işlemiyorsanız, mesaj parametrelerini doğrudan **DefWindowProc** işlevine iletin. Bu fonksiyon, mesaj için mesaj türüne göre değişen varsayılan eylemi gerçekleştirir.

C++

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

Pencere Prosedürünüzde Darboğazları Önleme

Pencere yordamınız yürütülürken, aynı iş parçacığında oluşturulan pencereler için diğer tüm mesajları engeller. Bu nedenle, pencere yordamınızın içinde uzun işlemler yapmaktan kaçının. Örneğin, programınızın bir TCP bağlantısı açtığını ve sunucunun yanıt vermesi için süresiz olarak beklediğini varsayalım. Bunu pencere prosedürü içinde yaparsanız, kullanıcı arayüzünüz istek tamamlanana kadar yanıt vermeyecektir. Bu süre zarfında, pencere fare veya klavye girdisini işleyemez, kendini yeniden boyayamaz ve hatta kapanamaz.

Bunun yerine, Windows'ta yerleşik olarak bulunan çoklu görev olanaklarından birini kullanarak işi başka bir iş parçacığına taşımalısınız:

- ♦ Yeni bir iş parçacığı
- ♦ oluşturun. Bir iş parçacığı havuzu kullanın.
- ♦ Eşzamansız G/Ç çağrıları kullanın.
- ♦ Eşzamansız yordam çağrıları kullanın.

Sonraki

[Pencerenin Boyanması](#)

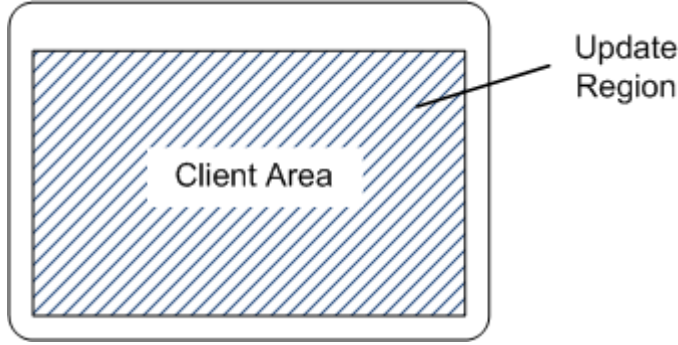
Pencerenin Boyanması

Makale - 08/19/2020 - Okumak için 3 dakika

Pencerenizi oluşturdunuz. Şimdi içinde bir şey göstermek istiyorsunuz. Windows terminolojisinde buna pencereyi boyamak denir. Mecazları karıştırmak gerekirse, pencere boş bir tuvaldir ve içeri doldurmanızı bekler.

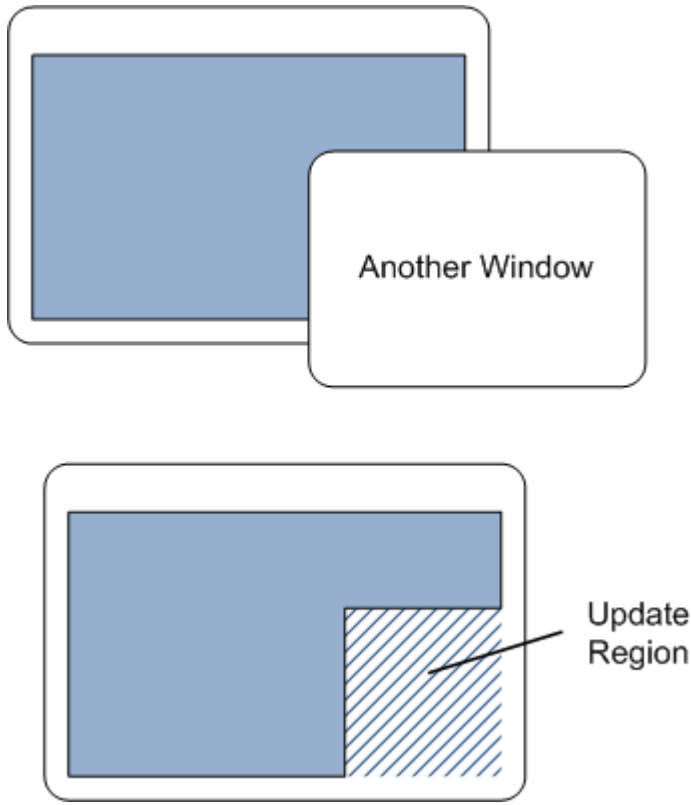
Bazen programınız pencerenin görünümünü güncellemek için boyama işlemini başlatır. Diğer zamanlarda, işletim sistemi pencerenin bir bölümünü yeniden boyamanız gerektiğini size bildirir. Bu gerçekleştiğinde, işletim sistemi pencereye bir **WM_PAINT** mesajı gönderir. Pencerenin boyanması gereken bölümüne *güncelleme bölgesi* adı verilir.

Bir pencere ilk kez gösterildiğinde, pencerenin tüm istemci alanı boyanmalıdır. Bu nedenle, bir pencere gösterdiğinizde her zaman en az bir **WM_PAINT** mesajı alırsınız.

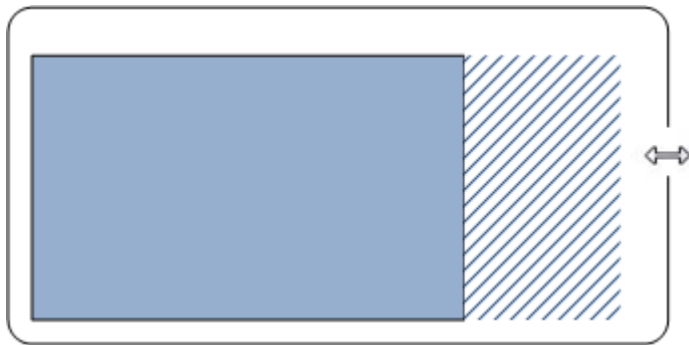


Yalnızca istemci alanını boyamaktan sorumlusunuz. Başlık çubuğu da dahil olmak üzere çevredeki çerçeve, işletim sistemi tarafından otomatik olarak boyanır. İstemci alanını boyamayı bitirdikten sonra, işletim sistemine bir şey değişene kadar başka bir **WM_PAINT** mesajı göndermesine gerek olmadığını söyleyen güncelleme bölgesini temizlersiniz.

Şimdi kullanıcının başka bir pencereyi hareket ettirdiğini ve böylece pencerenizin bir bölümünü gizlediğini varsayalım. Gizlenen kısım tekrar görünür hale geldiğinde, bu kısım güncelleme bölgesine eklenir ve pencereniz başka bir **WM_PAINT** mesajı alır.



Güncelleme bölgesi, kullanıcı pencereyi esnetirse de değişir. Aşağıdaki diyagramda, kullanıcı pencereyi sağa doğru uzatır. Pencerenin sağ tarafındaki yeni açık alan güncelleme bölgesine eklenir:



İlk örnek programımızda boyama rutini çok basittir. Sadece tüm istemci alanını düz bir renkle doldurur. Yine de, bu örnek bazı önemli kavramları göstermek için yeterlidir.

C++

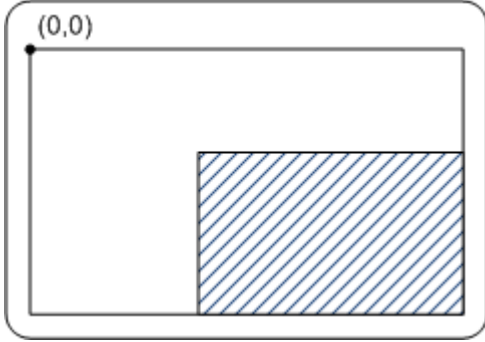
```
switch (uMsg)
{
    case WM_PAINT:
    {
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hwnd, &ps);

        // Tüm boyama işlemi burada, BeginPaint ve EndPaint arasında gerçekleşir.

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

```
    EndPaint(hwnd, &ps);  
}  
O döndür;  
}
```

BeginPaint fonksiyonunu çağırarak boyama işlemini başlatın. Bu fonksiyon **PAINTSTRUCT** yapısını yeniden boyama talebi ile ilgili bilgilerle doldurur. Geçerli güncelleme bölgesi **PAINTSTRUCT**'un **rcPaint** üyesinde verilir. Bu güncelleme bölgesi istemci alanına göre tanımlanır:



Boyama kodunuzda iki temel seçeneğiniz vardır:

- Güncelleme bölgesinin boyutundan bağımsız olarak tüm istemci alanını boyayın. Güncelleme bölgesinin dışında kalan her şey kırpılır. Yani, işletim sistemi bunu yok sayar.
- Pencerenin sadece güncelleme bölgesi içindeki kısmını boyayarak optimize edin.

Her zaman tüm istemci alanını boyarsanız, kod daha basit olacaktır. Ancak karmaşık boyama mantığınız varsa, güncelleme bölgesinin dışındaki alanları atlamak daha verimli olabilir.

Aşağıdaki kod satırı, sistem tarafından tanımlanan pencere arka plan rengini (**COLOR_WINDOW**) kullanarak güncelleme bölgesini tek bir renkle doldurur. **COLOR_WINDOW** tarafından gösterilen gerçek renk, kullanıcının geçerli renk şemasına bağlıdır.

C++

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

FillRect'in ayrıntıları bu örnek için önemli değildir, ancak ikinci parametre doldurulacak dikdörtgenin koordinatlarını verir. Bu durumda, güncelleme bölgesinin tamamını (**PAINTSTRUCT**'un **rcPaint** üyesi) aktarırız. İlk **WM_PAINT** mesajında, tüm istemci alanının boyanması gerekir, bu nedenle **rcPaint** tüm istemci alanını içerecektir. Sonraki **WM_PAINT** mesajlarında, **rcPaint** daha küçük bir dikdörtgen içerebilir.

FillRect işlevi, Windows grafiklerine çok uzun bir süredir güç veren Grafik Aygıt Arabiriminin (GDI) bir parçasıdır. Windows 7'de Microsoft, donanım hızlandırma gibi yüksek performanslı grafik işlemlerini destekleyen Direct2D adlı yeni bir grafik motoru tanıttı. Direct2D, Windows Vista için Platform Güncellemesi aracılığıyla Windows Vista için ve Windows Server 2008 için Platform Güncellemesi aracılığıyla Windows Server 2008 için de kullanılabilir. (GDI hala tam olarak desteklenmektedir.)

Boyama işlemini tamamladıktan sonra **EndPaint** fonksiyonunu çağırın. Bu fonksiyon güncelleme bölgesini temizler ve Windows'a pencerenin boyamayı tamamladığını bildirir.

Sonraki

[Pencereyi Kapatmak](#)

Pencereyi Kapatmak

Makale - 08/19/2020 - Okumak için 2 dakika

Kullanıcı bir pencereyi kapattığında, bu eylem bir dizi pencere mesajını tetikler.

Kullanıcı bir uygulama penceresini **Kapat** düğmesine tıklayarak veya ALT+F4 gibi bir klavye kısayolu kullanarak kapatabilir. Bu eylemlerden herhangi biri pencerenin bir **WM_CLOSE** mesajı almasına neden olur. **WM_CLOSE mesajı**, pencereyi kapatmadan önce kullanıcıyı uyarmanız için size bir fırsat verir. Eğer pencereyi gerçekten kapatmak istiyorsanız, **DestroyWindow** fonksiyonunu çağırın. Aksi takdirde, **WM_CLOSE mesajından** sıfır döndürün; işletim sistemi mesajı yok sayacak ve pencereyi yok etmeyecektir.

Aşağıda bir programın **WM_CLOSE**'u nasıl işleyebileceğine dair bir örnek verilmiştir.

C++

```
case WM_CLOSE:
    eğer (MessageBox(hwnd, L "Gerçekten çıkıldı mı?", L "Uygulamam",
MB_OKCANCEL) == IDOK)
    {
        DestroyWindow(hwnd);
    }
    // Else: Kullanıcı iptal etti. Hiçbir şey
    yapma. 0 döndür;
```

Bu örnekte, **MessageBox** işlevi, **OK** ve **Cancel** düğmelerini içeren modal bir iletişim kutusu gösterir. Kullanıcı **Tamam**'a tıklarsa, program **DestroyWindow**'u çağırır. Aksi takdirde, kullanıcı **Cancel düğmesini** tıklarınca, **DestroyWindow** çağırısı atlanır ve pencere açık kalır. Her iki durumda da, mesajı işlediğinizi belirtmek için sıfır döndürün.

Pencereyi kullanıcıya sormadan kapatmak istiyorsanız, **MessageBox**'a çağrı yapmadan **DestroyWindow**'u çağırabilirsiniz. Ancak, bu durumda bir kısayol vardır. **DefWindowProc**'un herhangi bir pencere mesajı için varsayılan eylemi yürüttüğünü hatırlayın. **WM_CLOSE** durumunda, **DefWindowProc** otomatik olarak **DestroyWindow**'u çağırır. Bu, **switch** deyiminizde **WM_CLOSE mesajını** yok sayarsanız, pencerenin varsayılan olarak yok edileceği anlamına gelir.

Bir pencere yok edilmek üzereyken, bir **WM_DESTROY** mesajı alır. Bu mesaj, pencere ekrandan kaldırıldıktan sonra, ancak yok etme işlemi gerçekleşmeden önce gönderilir (özellikle, herhangi bir alt pencere yok edilmeden önce).

Ana uygulama pencerenizde, **WM_DESTROY**'a genellikle şu çağrıyı yaparak yanıt verirsiniz **PostQuitMessage**.

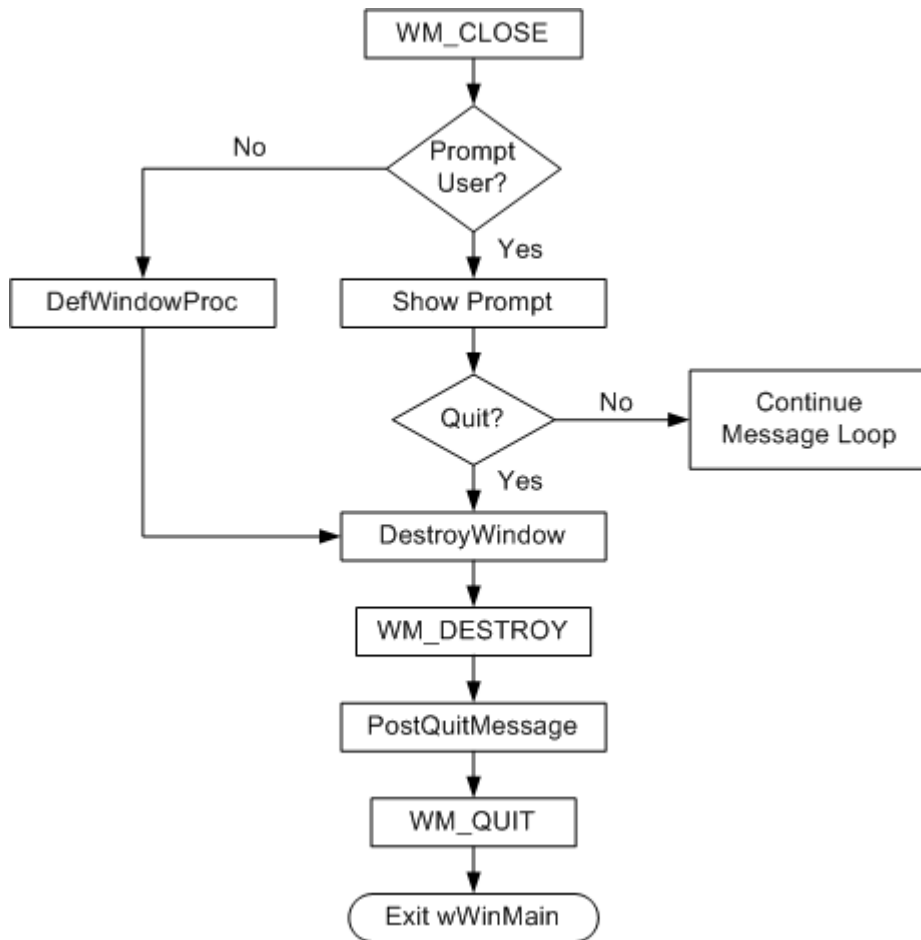
C++

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

[Pencere Mesajları](#) bölümünde [PostQuitMessage](#)'ın bir [WM_QUIT](#) mesaj kuyruğundaki mesaj, mesaj döngüsünün sona ermesine neden olur.

[WM_CLOSE](#) ve [WM_DESTROY](#) işlemlerinin tipik yolunu gösteren bir akış şeması aşağıda verilmiştir

Mesajlar:



Sonraki

[Uygulama Durumunu Yönetme](#)

Uygulama Durumunu Yönetme

Makale - 08/19/2020 - Okumak için 6 dakika

Bir pencere yordamı, her mesaj için çağrılan bir işlevdir, bu nedenle doğası gereği durumsuzdur. Bu nedenle, uygulamanızın durumunu bir işlev çağrısından diğerine takip etmek için bir yola ihtiyacınız vardır.

En basit yaklaşım, her şeyi global değişkenlere koymaktır. Bu, küçük programlar için yeterince iyi çalışır ve SDK örneklerinin çoğu bu yaklaşımı kullanır. Ancak büyük bir programda, global değişkenlerin çoğalmasına neden olur. Ayrıca, her biri kendi pencere prosedürüne sahip birkaç pencereniz olabilir. Hangi pencerenin hangi değişkenlere erişmesi gerektiğini takip etmek kafa karıştırıcı ve hataya açık hale gelir.

CreateWindowEx fonksiyonu, herhangi bir veri yapısını bir pencereye aktarmak için bir yol sağlar. Bu fonksiyon çağrıldığında, pencere yordamınıza aşağıdaki iki mesajı gönderir:

- ♦ **WM_NCCREATE**
- ♦ **WM_CREATE**

Bu mesajlar listelenen sırayla gönderilir. (**CreateWindowEx** sırasında gönderilen tek iki mesaj bunlar değildir, ancak bu tartışma için diğerlerini göz ardı edebiliriz).

WM_NCCREATE ve **WM_CREATE** mesajları pencere görünür hale gelmeden önce gönderilir. Bu, onları kullanıcı arayüzünüzü başlatmak için iyi bir yer yapar - örneğin, pencerenin ilk düzenini belirlemek için.

CreateWindowEx'in son parametresi **void*** türünde bir işaretçidir. Bu parametreye istediğiniz herhangi bir işaretçi değerini aktarabilirsiniz. Pencere yordamı **WM_NCCREATE** veya **WM_CREATE** mesajını işlediğinde, bu değeri mesaj verilerinden çıkarabilir.

Uygulama verilerini pencerenize aktarmak için bu parametreyi nasıl kullanacağınızı görelim. İlk olarak, durum bilgilerini tutan bir sınıf veya yapı tanımlayın.

C++

```
// Bazı durum bilgilerini tutmak için bir yapı tanımlayın.  
  
struct StateInfo {  
    // ... (struct üyeleri gösterilmemiştir)  
};
```



CreateWindowEx'i çağırdığınızda, bu yapıya bir işaretçi son **void*** parametre.

C++

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

eğer (pState == NULL)
{
    0 döndür;
}

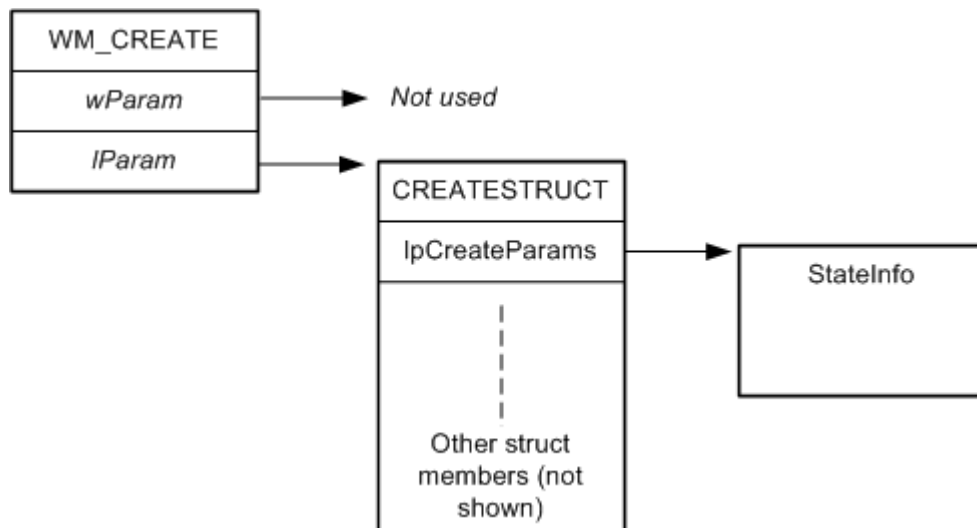
// Yapı üyelerini başlatın (gösterilmemiştir). HWND hwnd =

CreateWindowEx(
    0,                                // isteğe bağlı pencere stilleri.
    SINIF_ADI,                        // Pencere sınıfı
    L "Windows Programlamayı Öğrenin", // Pencere metni
    WS_OVERLAPPEDWINDOW,             // Pencere stili

    // Boyut ve konum
    CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

    BOŞ,                             // Ana pencere
    BOŞ,                             // Menü
    hInstance,                        // Örnek tanıtıcı
    pDevlet                           // Ek uygulama verileri
);
```

WM_NCCREATE ve **WM_CREATE** mesajlarını aldığınızda, her mesajın *lParam* parametresi **CREATESTRUCT** yapısına bir işaretçidir. **CREATESTRUCT** yapısı da **CreateWindowEx**'e aktardığınız işaretçi içerir.



Veri yapınızın işaretçisini şu şekilde elde edersiniz. İlk olarak, **CREATESTRUCT** yapısını *lParam* parametresini dökümleyerek oluşturur.

C++

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

CREATESTRUCT yapısının **lpCreateParams** üyesi, **CreateWindowEx** içinde belirttiğiniz orijinal void işaretçisidir. **lpCreateParams**'ı dökümleyerek kendi veri yapınıza bir işaretçi elde edin.

C++

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

Ardından, **SetWindowLongPtr** işlevini çağırın ve veri yapınızın işaretçisini iletin.

C++

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

Bu son fonksiyon çağrısının amacı, *StateInfo* işaretçisini pencerenin örnek verilerinde saklamaktır. Bunu yaptıktan sonra, **GetWindowLongPtr** çağrısını yaparak işaretçiyi her zaman pencereden geri alabilirsiniz:

C++

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

Her pencerenin kendi örnek verileri vardır, böylece birden fazla pencere oluşturabilir ve her pencereye veri yapısının kendi örneğini verebilirsiniz. Bu yaklaşım özellikle bir pencere sınıfı tanımladığınızda ve bu sınıftan birden fazla pencere oluşturduğunuzda (örneğin, özel bir kontrol sınıfı oluşturduğunuzda) kullanışlıdır. **GetWindowLongPtr** çağrısını küçük bir yardımcı fonksiyona sarmak uygundur.

C++

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

Şimdi pencere prosedürünüzü aşağıdaki gibi yazabilirsiniz.

C++

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

    switch (uMsg)
    {

        // Pencere prosedürünün geri kalanı gösterilmemiştir ...

    }
    TRUE döndür;
}
```

Nesne Yönelimli Bir Yaklaşım

Bu yaklaşımı daha da genişletebiliriz. Pencere hakkında durum bilgilerini tutmak için bir veri yapısı tanımladık. Bu veri yapısına veriler üzerinde çalışan üye fonksiyonlar (yöntemler) sağlamak mantıklıdır. Bu doğal olarak yapının (veya sınıfın) pencere üzerindeki tüm işlemlerden sorumlu olduğu bir tasarıma yol açar. Pencere prosedürü daha sonra sınıfın bir parçası haline gelecektir.

Başka bir deyişle, buradan devam etmek istiyoruz:

C++

// sözde kod

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

    /* pState'i HWND'den alın. */ switch
    (uMsg)
    {
        case WM_SIZE:
            HandleResize(pState, ...);
    }
}
```

Buna:

C++

// sözde kod

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

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

Tek sorun, metodun nasıl bağlanacağıdır `MyWindow::WindowProc` . Bu

RegisterClass işlevi, pencere yordamının bir işlev işaretçisi olmasını bekler. Bu bağlamda (statik olmayan) bir üye işleve işaretçi aktaramazsınız. Ancak, *statik* bir üye işleve bir işaretçi aktarabilir ve ardından üye işleve temsilci atayabilirsiniz. İşte bu yaklaşımı gösteren bir sınıf şablonu:

C++

```
şablon <class DERIVED_TYPE>
class BaseWindow
{
    Halka açık:
        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;
            }
            başka
            {
                pThis = (DERIVED_TYPE*)GetWindowLongPtr(hwnd, GWLP_USERDATA);
            }
            eğer (pThis)
            {
                return pThis->HandleMessage(uMsg, wParam, lParam);
            }
        }
    };
};
```

```

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

        HWND m_hwnd;
    };

```

Bu `BaseWindow` sınıfı, belirli pencere sınıflarının kendisinden türetildiği soyut bir temel sınıftır türetilmiştir. Örneğin, `BaseWindow`'dan türetilen basit bir sınıfın bildirimi aşağıda verilmiştir:

C++

```

class MainWindow : public BaseWindow<MainWindow>
{
    Halka açık:
        PCWSTR ClassName() const { return L "Örnek Pencere Sınıfı"; }
        LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);
};

```

Pencereyi oluşturmak için `BaseWindow::Create` ögesini çağırın:

C++

```

int WINAPI wWinMain(HINSTANCE hInstance, HINSTANCE, PWSTR pCmdLine, int
nCmdShow)
{
    MainWindow kazanır;

    if (!win.Create(L "Learn to Program Windows", WS_OVERLAPPEDWINDOW))
    {
        O döndür;
    }

    ShowWindow(win.Window(), nCmdShow);

    // Mesaj döngüsünü

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

    O döndür;
}

```

Pencereyi uygulamak için `BaseWindow::HandleMessage` pure-virtualmethod kullanılır prosedürüne eşdeğerdir. Örneğin, aşağıdaki uygulama pencereye eşdeğerdir

[Modül 1](#)'in başında gösterilen prosedür.

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);
            }
            0 döndür;

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

Pencere tanıtıcısının bir üye değişkende (*m_hwnd*) saklandığına dikkat edin, bu nedenle bunu `HandleMessage`'a parametre olarak aktarmamız gerekmez.

Microsoft Foundation Classes (MFC) ve Active Template Library (ATL) gibi mevcut Windows programlama çerçevelerinin çoğu, temelde burada gösterilene benzer yaklaşımlar kullanır. Elbette, MFC gibi tamamen genelleştirilmiş bir çerçeve, bu nispeten basit örnekten daha karmaşıktır.

Sonraki

[Modül 2: Windows Programınızda COM Kullanımı](#)

İlgili konular

[BaseWindow Örneği](#)

Modül 2. Windows Tabanlı Programınızda COM Kullanımı

Makale - 08/19/2020 - Okumak için 2 dakika

Bu serinin 1. [Modülü](#), bir pencerenin nasıl oluşturulacağını ve [WM_PAINT](#) ve [WM_CLOSE](#) gibi pencere mesajlarına nasıl yanıt verileceğini göstermiştir. Modül 2'de Bileşen Nesne Modeli (COM) tanıtılmaktadır.

COM, yeniden kullanılabilir yazılım bileşenleri oluşturmaya yönelik bir spesifikasyondur. Modern bir Windows tabanlı programda kullanacağınız özelliklerin çoğu COM'a dayanır, örneğin aşağıdakiler:

- ♦ Grafikler (Direct2D)
- ♦ Metin (DirectWrite)
- ♦ Windows Kabuğu
- ♦ Şerit kontrolü UI
- ♦ animasyonu

(Bu listedeki bazı teknolojiler COM'un bir alt kümesini kullanır ve bu nedenle "saf" COM değildir).

COM öğrenmesi zor bir yazılım olarak bilinir. Ve COM'u desteklemek için yeni bir yazılım modülü yazmanın zor olabileceği doğrudur. Ancak programınız sadece COM'un bir tüketicisi ise, COM'u anlamamanın beklediğinizden daha kolay olduğunu görebilirsiniz.

Bu modül, programınızda COM tabanlı API'lerin nasıl çağrılacağını gösterir. Ayrıca COM'un tasarımının arkasındaki bazı gerekçeler de açıklanmaktadır. COM'un neden olduğu gibi tasarlandığını anlarsanız, onunla daha etkili bir şekilde programlayabilirsiniz. Modülün ikinci bölümünde COM için önerilen bazı programlama uygulamaları açıklanmaktadır.

COM 1993 yılında Object Linking and Embedding (OLE) 2.0'ı desteklemek için tanıtılmıştır. İnsanlar bazen COM ve OLE'nin aynı şey olduğunu düşünürler. Bu, COM'un öğrenilmesinin zor olduğu algısının bir başka nedeni olabilir. OLE 2.0 COM üzerine inşa edilmiştir, ancak COM'u anlamak için OLE'yi bilmek zorunda değilsiniz.

COM bir dil standardı değil, *ikili bir standarttır*: Bir uygulama ile bir yazılım bileşeni arasındaki ikili arayüzü tanımlar. İkili bir standart olarak COM, belirli C++ yapılarıyla doğal olarak eşleşmesine rağmen dilden bağımsızdır. Bu modül COM'un üç ana hedefine odaklanacaktır:

- Bir nesnenin uygulamasını arayüzünden ayırma. Bir nesnenin yaşam süresini yönetme.
- Çalışma zamanında bir nesnenin yeteneklerini keşfetme.

Bu bölümde

- [COM Arayüzü Nedir? COM](#)
- [Kütüphanesini Başlatma](#)
- [COM'daki Hata Kodları](#)
- [COM'da Nesne Oluşturma](#)
- [Örnek: Aç İletişim Kutusu Bir Nesnenin](#)
- [Yaşam Süresini Yönetme COM'da Bir](#)
- [Nesneden Arayüz Bellek Tahsisi İsteme](#)
- [COM Kodlama](#)
- [Uygulamaları COM'da](#)
[Hata İşleme](#)

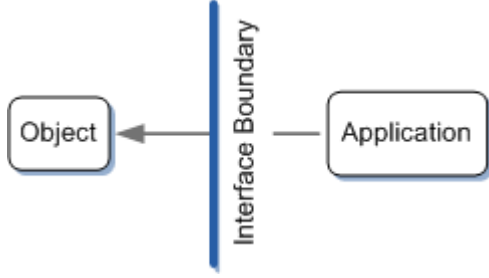
İlgili konular

[C++ ile Windows için Programlamayı Öğrenin](#)

COM Arayüzü Nedir?

Makale - 10/03/2019 - 3 dakika okumak için

C# veya Java biliyorsanız, arayüzler tanıdık bir kavram olmalıdır. Bir *arayüz*, bir nesnenin destekleyebileceği bir dizi yöntemi, uygulama hakkında herhangi bir şey dikte etmeden tanımlar. Arayüz, bir yöntemi çağıran kod ile yöntemi uygulayan kod arasında net bir sınır çizer. Bilgisayar bilimleri açısından, çağıran uygulamadan ayrıştırılmıştır.



C++'da, bir arayüze en yakın eşdeğer saf sanal sınıftır; yani, yalnızca saf sanal yöntemler içeren ve başka hiçbir üyesi olmayan bir sınıftır. İşte varsayımsal bir arayüz örneği:

C++

```
// Aşağıdakiler gerçek COM değildir.  
  
// Pseudo-C++:  
  
arayüz IDrawable  
{  
    void Draw();  
};
```

Bu örnekteki fikir, bazı grafik kütüphanelerindeki bir dizi nesnenin çizilebilir olduğudur.

`IDrawable` Arayüz, herhangi bir çizilebilir nesnenin desteklemesi gereken işlemleri tanımlar.

(Geleneksel olarak, arayüz adları "I" ile başlar.) Bu örnekte, the tek `IDrawable` arayüz işlem tanımlamaktadır: Çiz .

Tüm arayüzler *soyuttur*, bu nedenle bir program bir nesnenin örneğini oluşturamaz. Örneğin, aşağıdaki kod derlenmeyecektir.

`IDrawable`

C++

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

Bunun yerine, grafik kütüphanesi şu nesneleri sağlar `IDrawable` arayüz. Örneğin, kütüphane şekilleri çizmek için bir shape nesnesi ve görüntüleri çizmek için bir bitmap nesnesi sağlayabilir. C++'da bu, ortak bir soyut temel sınıftan miras alınarak yapılır:

C++

```
class Şekil : public IDrawable  
{  
    Halka açık:  
    virtual void Draw();           // Draw ögesini geçersiz kılın ve uygulama sağlayın.  
};  
  
class Bitmap : public IDrawable  
{  
    Halka açık:  
    virtual void Draw();           // Draw ögesini geçersiz kılın ve uygulama sağlayın.  
};
```

Bu `Şekil` ve `Bit` sınıfları iki farklı çizilebilir nesne türü tanımlar. Her sınıf 'den miras ve kendi uygulamasını sağlar yöntem. alır `IDrawable` `Çizmek` Doğal olarak, iki uygulama önemli ölçüde farklılık gösterebilir. Örneğin `Shape::Draw` yöntemi bir dizi çizgiyi rasterleştirebilirken `Bitmap::Çizim` bir blit olur piksel dizisi.

Bu grafik kütüphanesini kullanan bir program `Şekil` ve `Bit` nesneler aracılığı `IDrawable` işaretçilerini kullanmak `Şekil` ve `Bit` doğrudan işaretçiler. yla yerine ya

C++

```
IDrawable *pDrawable = CreateTriangleShape();  
  
eğer (pDrawable)  
{  
    pDrawable->Draw();  
}
```

İşte bir dizi üzerinde döngü yapan bir örnek `IDrawable` işaretçiler. Dizi şunları içerebilir dizideki her nesne `IDrawable`'ı miras aldığı sürece şekillerin, bitmaplerin ve diğer grafik nesnelerinin heterojen bir çeşitliliğini içerir.

C++

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

COM ile ilgili önemli bir nokta, çağırılan kodun türetilmiş sınıfın türünü asla görmemesidir.

Başka bir deyişle, asla şu türden bir değişken bildirmezsiniz: `Sekil` ve `Bit` senin içinde Kod. Şekiller ve bitmapler üzerindeki tüm işlemler `IDrawable` işaretçiler. içinde

Bu şekilde COM, arayüz ve uygulama arasında katı bir ayrımı korur. Bu da uygulama detayları `Sekil` ve `Bit` sınıflar değişebilir - örneğin, düzeltmek için

hataları giderir veya yeni özellikler ekler - çağrı kodunda hiçbir değişiklik yapmadan.

Bir C++ uygulamasında, arayüzler bir sınıf veya yapı kullanılarak bildirilir.

7 Not

Bu konudaki kod örnekleri gerçek dünya uygulamalarını değil, genel kavramları aktarmayı amaçlamaktadır. Yeni COM arayüzlerinin tanımlanması bu serinin kapsamı dışındadır, ancak bir arayüzü doğrudan bir başlık dosyasında tanımlamazsınız. Bunun yerine, bir COM arayüzü Arayüz Tanımlama Dili (IDL) adı verilen bir dil kullanılarak tanımlanır. IDL dosyası, bir C++ başlık dosyası oluşturan bir IDL derleyicisi tarafından işlenir

C++

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

COM ile çalışırken, arayüzlerin nesne olmadığını unutmamak önemlidir. Bunlar, nesnelerin uygulaması gereken yöntem koleksiyonlarıdır. Birkaç nesne şunları yapabilir ile gösterildiği gibi aynı arayüzü uygular. `Sekil` ve `Bit` Örnekler.

Dahası, bir nesne birkaç arayüzü uygulayabilir. Örneğin, grafik kütüphanesi kaydetme ve yüklemeyi destekleyen bir arayüz tanımlayabilir `ISerializable`. grafik nesneleri. Şimdi aşağıdaki sınıf bildirimlerini göz önünde bulundurun:

C++

```
// Serileştirme için bir arayüz. class
ISerializable
{
    Halka açık:

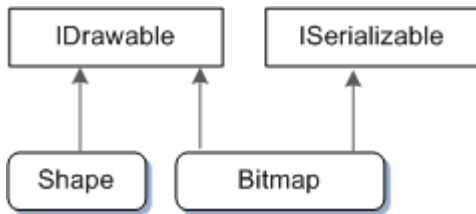
    virtual void Load(PCWSTR dosya adı) = 0;    // Dosyadan
    yükle. virtual void Kaydet(PCWSTR dosya adı) = 0;    //
    Dosyaya kaydet.
};

// Çizilebilir nesne türlerinin bildirimleri.

class Şekil : public IDrawable
{
    ...
};

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

Bu örnekte, aşağıdaki `Şekil` sınıfı `ISerializable`'i uygular. Program şunları kullanabilir bit eşlemeyi kaydetmek veya yüklemek için bu yöntemleri kullanabilirsiniz. Bununla birlikte, `Şekil` sınıfı `ISerializable`, bu nedenle bu işlevselliği ortaya çıkarmaz. Aşağıdaki diyagram bu örnekteki kalıtım ilişkilerini göstermektedir.



Bu bölümde arayüzlerin kavramsal temelini inceledik, ancak şu ana kadar gerçek COM kodunu görmedik. Her COM uygulamasının yapması gereken ilk şeyle başlayacağız: COM kütüphanesini başlatmak.

Sonraki

[COM Kitaplığını Başlatma](#)

COM Kitaplığını Başlatma

Makale - 08/19/2020 - Okumak için 2 dakika

COM kullanan herhangi bir Windows programı, **CoInitializeEx** işlevini çağırarak COM kitaplığını başlatmalıdır. Bir COM arayüzü kullanan her iş parçacığı bu fonksiyona ayrı bir çağrı yapmalıdır. **CoInitializeEx** aşağıdaki imzaya sahiptir:

C++

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

İlk parametre ayrılmıştır ve **NULL** olmalıdır. İkinci parametre, programınızın kullanacağı iş parçacığı modelini belirtir. COM, *daire iş parçacıklı* ve *çok iş parçacıklı* olmak üzere iki farklı iş parçacığı modelini destekler. Daire iş parçacığı belirtirseniz, aşağıdaki garantileri vermiş olursunuz:

- Her COM nesnesine tek bir iş parçacığından erişeceksiniz; COM arayüz işaretçilerini birden fazla iş parçacığı arasında paylaşmayacaksınız.
- İş parçacığı bir mesaj döngüsüne sahip olacaktır. (Modül 1'deki [Pencere Mesajları bölümüne](#) bakın.)

Bu kısıtlamalardan herhangi biri doğru değilse, çok iş parçacıklı modeli kullanın. İş parçacığı modelini belirtmek için, *dwCoInit* parametresinde aşağıdaki bayraklardan birini ayarlayın.

Bayrak	Açıklama
COINIT_APARTMENTTHREADED	Daire dışı.
COINIT_MULTITHREADED	Çoklu iş parçacıklı.

Bu bayraklardan tam olarak birini ayarlamanız gerekir. Genel olarak, pencere oluşturan bir iş parçacığı **COINIT_APARTMENTTHREADED** bayrağını, diğer iş parçacıkları ise **COINIT_MULTITHREADED** bayrağını kullanmalıdır. Ancak, bazı COM bileşenleri belirli bir iş parçacığı modeli gerektirir. MSDN belgeleri size böyle bir durumda ne yapmanız gerektiğini söyleyecektir.

7 Not

Aslında, apartman iş parçacığı belirlerseniz bile, *marshaling* adı verilen bir teknik kullanarak iş parçacıkları arasında arayüzleri paylaşmak hala mümkündür. Marshaling bu modülün kapsamı dışındadır. Önemli olan nokta, apartman iş parçacığı ile bir arayüz işaretçisini asla başka bir iş parçacığına kopyalamamanız gerektirir. Çünkü

COM iş parçacığı modelleri hakkında daha fazla bilgi için [Processes, Threads ve Apartments](#) bölümüne bakın.

Daha önce bahsedilen bayraklara ek olarak, *dwCoInit* parametresinde **COINIT_DISABLE_OLE1DDE** bayrağını ayarlamak iyi bir fikirdir. Bu bayrağın ayarlanması, eski bir teknoloji olan Object Linking and Embedding (OLE) 1.0 ile ilişkili bazı ek yükleri önler.

COM'u apartman iş parçacığı için şu şekilde başlatırsınız:

C++

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

HRESULT dönüş tipi bir hata veya başarı kodu içerir. Bir sonraki bölümde COM hata işleme konusuna bakacağız.

COM Kütüphanesinin Başlangıcını Kaldırma

CoInitializeEx'e yapılan her başarılı çağrı için, iş parçacığı çıkmadan önce **CoUninitialize**'i çağırmanız gerekir. Bu fonksiyon parametre almaz ve geri dönüş değeri yoktur.

C++

```
CoUninitialize();
```

Sonraki

[COM'daki Hata Kodları](#)

COM'daki Hata Kodları

Makale - 04/27/2021 - Okumak için 2 dakika

COM yöntemleri ve işlevleri, başarı veya başarısızlığı belirtmek için **HRESULT** türünde bir değer döndürür. Bir **HRESULT** 32 bitlik bir tamsayıdır. **HRESULT**'un yüksek sıralı biti başarı veya başarısızlığı belirtir. Sıfır (0) başarıyı, 1 ise başarısızlığı gösterir.

Bu, aşağıdaki sayısal aralıkları üretir:

- Başarı kodları: 0x0-0x7FFFFFFF.
- Hata kodları: 0x80000000-0xFFFFFFFF.

Az sayıda COM yöntemi **HRESULT** değeri döndürmez. Örneğin, **AddRef** ve **Release** yöntemleri işaretli uzun değerler döndürür. Ancak bir hata kodu döndüren her COM yöntemi bunu bir **HRESULT** değeri döndürerek yapar.

Bir COM yönteminin başarılı olup olmadığını kontrol etmek için, döndürülen **HRESULT**'un yüksek dereceli bitini inceleyin. Windows SDK başlıkları bunu kolaylaştıran iki makro sağlar: **SUCCEEDED** makrosu ve **FAILED** makrosu. **SUCCEEDED** makrosu, **HRESULT** bir başarı koduysa **TRUE**, bir hata koduysa **FALSE** döndürür. Aşağıdaki örnek **CoInitializeEx**'in başarılı olup olmadığını kontrol eder.

C++

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

eğer (SUCCEEDED(hr))
{
    // İşlev başarılı oldu.
}
başka
{
    // Hatayı ele alın.
}
```

Bazen ters koşulu test etmek daha uygundur. **FAILED** makrosu **SUCCEEDED**'in tersini yapar. Bir hata kodu için **TRUE** ve bir başarı kodu için **FALSE** döndürür.

C++

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

eğer (FAILED(hr))
{
    // Hatayı ele alın.
}
başka
```


Bu modülün ilerleyen bölümlerinde, COM hatalarını işlemek için kodunuzu nasıl yapılandıracağınıza ilişkin bazı pratik tavsiyelere bakacağız. (Bkz. [COM'da Hata İşleme](#).)

Sonraki

[COM'da Nesne Oluşturma](#)

COM'da Nesne Oluřturma

Makale - 23/08/2022 - Okumak için 5 dakika

Bir iř paracıęı COM kitaplıęını bařlattıktan sonra, iř paracıęının COM arabirimlerini kullanması gvenlidir. Bir COM arayzn kullanmak iin, programınız nce o arayz uygulayan bir nesnenin rneęini oluřturur.

Genel olarak, bir COM nesnesi oluřturmanın iki yolu vardır:

- Nesneyi uygulayan modl, bu nesnenin rneklerini oluřturmak iin zel olarak tasarlanmış bir iřlev saęlayabilir.
- Alternatif olarak COM, **CoCreateInstance** adında genel bir oluřturma iřlevi saęlar.

rneęin, varsayımsal **Arayz** ele **Sekil** **COM Nedir** konusundan nesne alalım. Bu rnekta **Sekil** nesnesi **IDrawable** adlı bir arayz uygular.

Ařaęıdaki imzayı uygulayan grafik **Sekil** nesnesi ile bir iřlev dıřa aktarabilir ktphanesi.

C++

```
// Gerek bir Windows iřlevi deęildir.  
  
HRESULT CreateShape(IDrawable** ppShape);
```

Bu iřlev gz nne alındıęında, ařaęıdaki gibi bir yeni nesne oluřturabilirsiniz.

C++

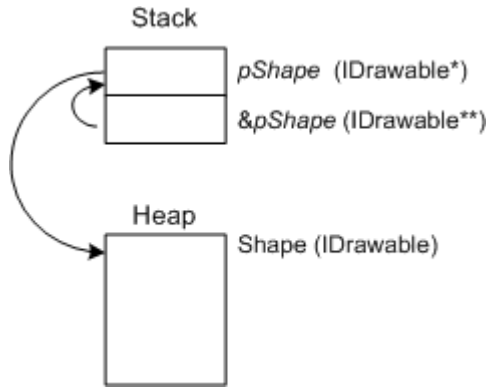
```
IDrawable *pShape;  
  
HRESULT hr = CreateShape(&pShape);  
if (SUCCEEDED(hr))  
{  
    // Shape nesnesini kullanın.  
}  
bařka  
{  
    // Bir hata oluřtu.  
}
```

ppShape parametresi pointer-to-pointer-to-IDrawable tipindedir. Bu kalıbı daha nce grmediyseniz, ift dolaylama kafa karıřtırıcı olabilir.

Fonksiyonun gerekliliklerini gz nne alalım. Fonksiyon bir **IDrawable** iřaretisini aęırana geri gnderir. Ancak iřlevin geri dnř deęeri zaten

hata/başarı kodu. Bu nedenle, işaretçi bir argüman aracılığıyla fonksiyon. Çağırان kişi şu tipte bir değişken `IDrawable*` fonksiyonuna ve aktaracaktır `IDrawable` işaretçi. C++'da sadece fonksiyonu bu değişkenin üzerine yeni bir `IDrawable` Bir fonksiyonun bir parametre değerinin üzerine yazması için iki yol vardır: referansa göre geçiş veya adrese göre geçiş. COM ikincisini, adrese göre geçişi kullanır. Ve bir işaretçinin adresi bir işaretçiye işaretçidir, bu nedenle parametre türü `IDrawable**` olmalıdır.

İşte neler olup bittiğini görselleştirmeye yardımcı olacak bir diyagram.



Bu `CreateShape` fonksiyonu yeni bir işaretçi yazmak için `pShape (&pShape)` adresini değerini `Shape (IDrawable)` kullanır `pShape`'e aktarır.

CoCreateInstance: Nesne Oluşturmanın Genel Bir Yolu

CoCreateInstance işlevi, nesne oluşturmak için genel bir mekanizma sağlar. **CoCreateInstance**'ı anlamak için, iki COM nesnesinin aynı arabirimi uygulayabileceğini ve bir nesnenin iki veya daha fazla arabirimi uygulayabileceğini unutmayın. Bu nedenle, nesne oluşturan genel bir işlev iki parça bilgiye ihtiyaç duyar.

- ♦ Hangi nesnenin oluşturulacağı.
- ♦ Nesneden hangi arayüzün alınacağı.

Ancak fonksiyonu çağırdığımızda bu bilgiyi nasıl belirteceğiz? COM'da, bir nesne ya da arayüz, *global olarak benzersiz olarak* adlandırılan 128 bitlik bir sayı atanarak tanımlanır *tanımlayıcı* (GUID). GUID'ler, onları etkin bir şekilde benzersiz kılacak şekilde oluşturulur. GUID'ler, merkezi bir kayıt otoritesi olmadan benzersiz tanımlayıcıların nasıl oluşturulacağı sorununa bir çözümdür. GUID'ler bazen *evrensel olarak benzersiz tanımlayıcılar* (UUID'ler) olarak adlandırılır. COM'dan önce, DCE/RPC'de (Dağıtılmış Bilgi İşlem Ortamı/Uzaktan Prosedür Çağırısı) kullanılıyorlardı. Yeni GUID'ler

oluřturmak iin eřitli algoritmalar mevcuttur. Bu algoritmaların hepsi benzersizlięi kesin olarak garanti etmez, ancak aynı GUID deęerinin yanlışlıkla iki kez oluřturulma olasılığı son derece dūřüktür - etkili bir řekilde sıfırdır.

GUID'ler sadece nesneleri ve arayüzleri değil, her türlü varlığı tanımlamak için kullanılabilir. Ancak, bu modülde bizi ilgilendiren tek kullanım budur.

Örneğin, `Sekille` kütüphane iki GUID sabiti bildirebilir:

C++

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

(Bu sabitler için gerçek 128 bitlik sayısal değerlerin tanımlandığını varsayabilirsiniz başka bir yerde). **CLSID_Shape** sabiti nesneyi tanımlar `Sekil`, **CLSID IID_IDrawable** şu öğeyi tanımlar `IDrawable` arayüz. "CLSID" öneki *sınıf* anlamına gelir *tanımlayıcısı* ve IID öneki *arayüz tanımlayıcısı* anlamına gelir. Bunlar COM'daki standart adlandırma kurallarıdır.

Bu değerler göz önüne alındığında, yeni bir `Sekil` örneği aşağıdaki gibidir:

C++

```
IDrawable *pShape;  
hr = CoCreateInstance(CLSID_Shape, NULL, CLSCTX_INPROC_SERVER,  
IID_IDrawable,  
    reinterpret_cast<void*>(&pShape));  
  
eğer (SUCCEEDED(hr))  
{  
    // Shape nesnesini kullanın.  
}  
başka  
{  
    // Bir hata oluştu.  
}
```

CoCreateInstance işlevinin beş parametresi vardır. Birinci ve dördüncü parametreler sınıf tanımlayıcısı ve arayüz tanımlayıcısıdır. Aslında bu parametreler fonksiyona "Shape nesnesini oluştur ve bana IDrawable arayüzüne bir işaretçi ver" der.

İkinci parametreyi **NULL** olarak ayarlayın. (Bu parametrenin anlamı hakkında daha fazla bilgi için COM belgelerindeki [Aggregation](#) konusuna bakın). Üçüncü parametre, temel amacı nesnenin *yürütme bağlamını* belirtmek olan bir dizi bayrak alır. Yürütme bağlamı, nesnenin uygulamayla aynı süreçte mi; aynı bilgisayarda farklı bir süreçte mi; yoksa uzak bir bilgisayarda mı çalışacağını belirtir. Aşağıdaki tabloda bu parametre için en yaygın değerler gösterilmektedir.

Bayrak	Açıklama
CLSCTX_INPROC_SERVER	Aynı süreç.
CLSCTX_LOCAL_SERVER	Farklı süreç, aynı bilgisayar.
CLSCTX_REMOTE_SERVER	Farklı bir bilgisayar.
CLSCTX_ALL	Nesnenin desteklediği en verimli seçeneği kullanın. (En verimli olandan en az verimli olana doğru sıralama şöyledir: süreç içi, süreç dışı ve bilgisayarlar arası).

Belirli bir bileşenin belgeleri size nesnenin hangi yürütme bağlamını desteklediğini söyleyebilir. Aksi takdirde, **CLSCTX_ALL** kullanın. Nesnenin desteklemediği bir yürütme bağlamı talep ederseniz, **CoCreateInstance** işlevi **REGDB_E_CLASSNOTREG** hata kodunu döndürür. Bu hata kodu, CLSID'nin kullanıcının bilgisayarında kayıtlı herhangi bir bileşene karşılık gelmediğini de gösterebilir.

CoCreateInstance'ın beşinci parametresi arayüze bir işaretçi alır. **CoCreateInstance** genel bir mekanizma olduğundan, bu parametre güçlü bir şekilde yazılamaz. Bunun yerine, veri türü **void****'dir ve çağıran, işaretçinin adresini bir **void**** türüne zorlamalıdır. Önceki örnekteki **reinterpret_cast**'in amacı da budur.

CoCreateInstance'ın geri dönüş değerini kontrol etmek çok önemlidir. İşlev bir hata kodu döndürürse, COM arabirim işaretçisi geçersizdir ve bu işaretçinin referansını kaldırmaya çalışmak programınızın çökmesine neden olabilir.

Dahili olarak, **CoCreateInstance** işlevi bir nesne oluşturmak için çeşitli teknikler kullanır. En basit durumda, kayıt defterinde sınıf tanımlayıcısını arar. Kayıt defteri girişi, nesneyi uygulayan bir DLL veya EXE'ye işaret eder. **CoCreateInstance** ayrıca COM+ kataloğundaki veya yan yana (SxS) bildirimdeki bilgileri de kullanabilir. Ne olursa olsun, ayrıntılar çağıran için şeffaftır. **CoCreateInstance**'ın dahili ayrıntıları hakkında daha fazla bilgi için, **COM İstemcileri ve Sunucuları** bölümüne bakın.

Kullandığımız örnek biraz yapmacıktı, bu yüzden şimdi COM'un iş başında olduğu gerçek dünyadan bir örnek: kullanıcının bir dosya seçmesi için **Aç** iletişim kutusunu görüntülemek.

Sonraki

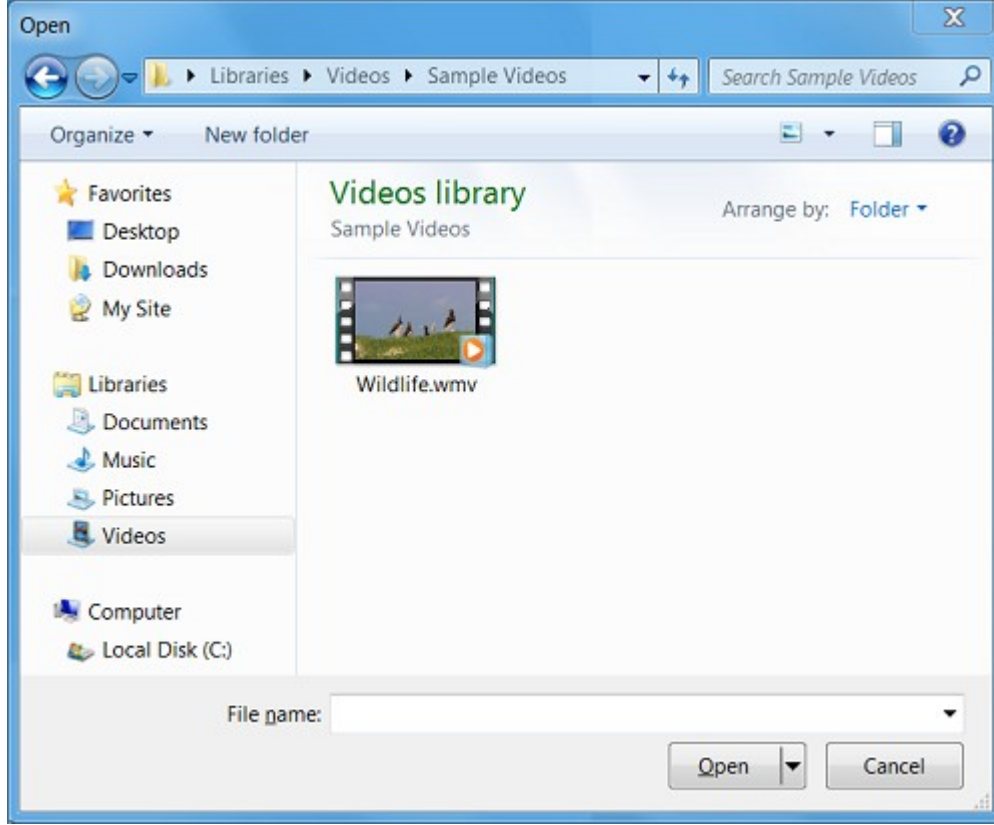
Örnek: Aç İletişim Kutusu

Örnek: Aç İletişim Kutusu

Makale - 08/19/2020 - Okumak için 2 dakika

Kullandığınız örnek biraz yapmacıktır. Şimdi bir

Gerçek bir Windows programında kullanabileceğiniz COM nesnesi: **Aç** iletişim kutusu.



Aç iletişim kutusunu göstermek için, bir program Common Item Dialog nesnesi adı verilen bir COM nesnesi kullanabilir. Ortak Öğe İletişim Kutusu, Shobjidl.h başlık dosyasında bildirilen **IFileOpenDialog** adlı bir arabirimi uygular.

İşte kullanıcıya **Aç** iletişim kutusunu görüntüleyen bir program. Kullanıcı bir dosya seçerse, program dosya adını içeren bir iletişim kutusu gösterir.

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);
```

```
    eğer (SUCCEEDED(hr))
```

```
    {
```

```
        IFileOpenDialog *pFileOpen;
```

```
        // FileOpenDialog nesnesini oluşturun.
```

```
        hr = CoCreateInstance(CLSID_FileOpenDialog, NULL, CLSCTX_ALL,
```


Bu kod, modülde daha sonra açıklanacak bazı kavramları kullanır, bu nedenle burada her şeyi anlamazsanız endişelenmeyin. İşte kodun temel bir taslağı:

1. COM kütüphanesini başlatmak için **CoInitializeEx** ögesini çağırın.
2. Common Item Dialog nesnesini oluşturmak ve nesnenin **IFileOpenDialog** arayüzüne bir işaretçi almak için **CoCreateInstance** ögesini çağırın.
3. Nesnenin, iletişim kutusunu kullanıcıya gösteren **Show** yöntemini çağırın.
Bu yöntem, kullanıcı iletişim kutusunu kapatana kadar engellenir.
4. Nesnenin **GetResult** yöntemini çağırın. Bu yöntem, *Shell ögesi nesnesi* adı verilen ikinci bir COM nesnesine bir işaretçi döndürür. **IShellItem** arabirimini uygulayan Shell ögesi, kullanıcının seçtiği dosyayı temsil eder.
5. Shell ögesinin **GetDisplayName** yöntemini çağırın. Bu yöntem, dosya yolunu bir dize biçimindedir.
6. Dosya yolunu gösteren bir mesaj kutusu gösterin.

7. COM kütüphanesinin başlatmasını kaldırmak için [CoUninitialize](#) ögesini çağırın.

Adım 1, 2 ve 7, COM kütüphanesi tarafından tanımlanan işlevleri çağırır. Bunlar genel COM işlevleridir. Adım 3-5, Common Item Dialog nesnesi tarafından tanımlanan yöntemleri çağırır.

Bu örnek, nesne oluşturma'nın her iki çeşidini de göstermektedir: Genel [CoCreateInstance](#) işlevi ve Common Item Dialog nesnesine özgü bir yöntem ([GetResult](#)) içerir.

Sonraki

[Bir Nesnenin Yaşam Süresini Yönetme](#)

İlgili konular

[İletişim Kutusu Açma Örneği](#)

Bir Nesnenin Yaşam Süresini Yönetme

Makale - 01/28/2021 - 4 dakika okumak için

COM arayüzleri için henüz bahsetmediğimiz bir kural vardır. Her COM arayüzü, doğrudan veya dolaylı olarak **IUnknown** adlı bir arayüzden miras almalıdır. Bu arayüz, tüm COM nesnelerinin desteklemesi gereken bazı temel yetenekler sağlar.

IUnknown arayüzü üç yöntem tanımlar:

- ♦ **QueryInterface**
- ♦ **AddRef**
- ♦ **Serbest Bırakma**

QueryInterface yöntemi, bir programın çalışma zamanında nesnenin yeteneklerini sorgulamasını sağlar. Bir sonraki konu olan **Nesneden Arayüz İsteme** bölümünde bu konu hakkında daha fazla bilgi vereceğiz. **AddRef** ve **Release** yöntemleri bir nesnenin yaşam süresini kontrol etmek için kullanılır. Bu, bu konunun konusudur.

Referans Sayımı

Bir program başka ne yaparsa yapsın, bir noktada kaynakları tahsis edecek ve serbest bırakacaktır. Bir kaynak tahsis etmek kolaydır. Kaynağın ne zaman serbest bırakılacağını bilmek zordur, özellikle de kaynağın ömrü mevcut kapsamın ötesine uzanıyorsa. Bu sorun COM'a özgü değildir. Yığın bellek ayıran her program aynı sorunu çözmek zorundadır. Örneğin, C++ otomatik yıkıcıları kullanırken, C# ve Java çöp toplamayı kullanır.

COM, *referans sayma* adı verilen bir yaklaşım kullanır.

Her COM nesnesi dahili bir sayım tutar. Bu, referans sayısı olarak bilinir. Referans sayısı, o anda nesneye kaç referansın etkin olduğunu izler. Referans sayısı sıfıra düştüğünde, nesne kendini siler. Son kısım tekrarlamaya değer: Nesne kendini siler. Program nesneyi asla açıkça silmez.

İşte referans sayımı için kurallar:

- ♦ Nesne ilk oluşturulduğunda, referans sayısı 1'dir. Bu noktada, programın nesneye yönelik tek bir işaretçisi vardır.
- ♦ Program, işaretçiyi çoğaltarak (kopyalayarak) yeni bir referans oluşturabilir. İşaretçiyi kopyaladığınızda, nesnenin **AddRef** yöntemini çağırmanız gerekir. Bu yöntem, referans sayısını bir artırır.
- ♦ Nesneye yönelik bir işaretçiyi kullanmayı bitirdiğinizde **Release** ögesini çağırmanız

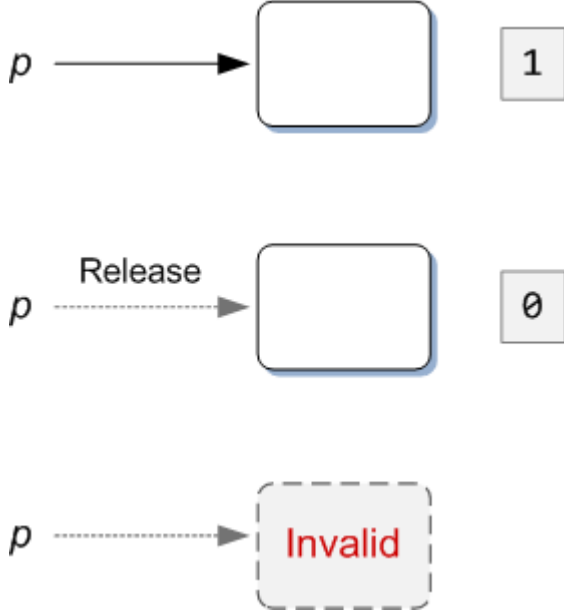
gerekir. Bu durumda

Release yöntemi referans sayısını bir azaltır. Aynı zamanda

işaretçi. **Release** çağrısını yaptıktan sonra işaretçi tekrar kullanmayın. (Aynı nesne için başka işaretçileriniz varsa, bu işaretçileri kullanmaya devam edebilirsiniz).

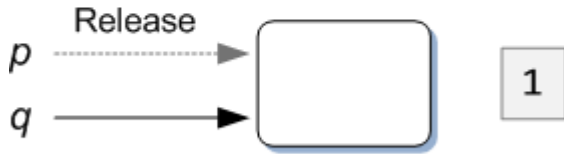
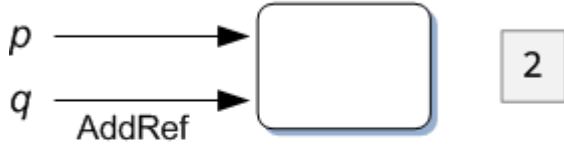
- Her işaretçi ile **Release** çağrısı yaptığınızda, nesnenin nesne referans sayısı sıfıra ulaşır ve nesne kendini siler.

Aşağıdaki diyagram basit ama tipik bir durumu göstermektedir.



Program bir nesne oluşturur ve nesneye bir işaretçi (p) saklar. Bu noktada, referans sayısı 1'dir. Program işaretçi kullanmayı bitirdiğinde **Release** çağrısını yapar. Referans sayısı sıfıra düşürülür ve nesne kendini siler. Artık p geçersizdir. Başka herhangi bir yöntem çağrısı için p 'yi kullanmak bir hatadır.

Bir sonraki diyagram daha karmaşık bir örneği göstermektedir.



Burada, program bir nesne yaratır ve daha önce olduğu gibi p işaretçisini saklar. Daha sonra, program p 'yi yeni bir değişken olan q 'ya kopyalar. Bu noktada, programın referans sayısını artırmak için **AddRef**'i çağırması gerekir. Referans sayısı artık 2'dir ve nesne için iki geçerli işaretçi vardır. Şimdi programın p 'yi kullanmayı bitirdiğini varsayalım. Program **Release**'i çağırır, referans sayısı 1'e düşer ve p artık geçerli değildir. Ancak q hala geçerlidir. Daha sonra, program q 'yu kullanmayı bitirir. Bu nedenle, **Release**'i tekrar çağırır. Referans sayısı sıfıra gider ve nesne kendini siler.

Programın neden p 'yi kopyaladığını merak edebilirsiniz. Bunun iki ana nedeni vardır: Birincisi, işaretçiyi liste gibi bir veri yapısında saklamak isteyebilirsiniz. İkincisi, işaretçiyi orijinal değişkenin mevcut kapsamının ötesinde tutmak isteyebilirsiniz. Bu nedenle, onu daha geniş kapsamı olan yeni bir değişkene kopyalarsınız.

Referans saymanın bir avantajı, çeşitli kod yolları nesneyi silmek için koordine olmadan kodun farklı bölümleri arasında işaretçileri paylaşabilmenizdir. Bunun yerine, her kod yolu yalnızca o kod yolu nesneyi kullanmayı bitirdiğinde **Release**'i çağırır. Nesne, doğru zamanda kendini silmeyi gerçekleştirir.

Örnek

İşte yine [Aç iletişim kutusu örneğindeki](#) kod.

C++

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

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

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

    eğer (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"Dosya Yolu",
MB_OK);

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

Referans sayma işlemi bu kodda iki yerde gerçekleşir. İlk olarak, program Common Item Dialog nesnesini başarıyla oluşturursa, *pFileOpen* işaretçisi üzerinde **Release** çağrısı yapılmalıdır.

C++

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

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

İkinci olarak, **GetResult** yöntemi **IShellItem** arayüzüne bir işaretçi döndürdüğünde, program *pItem* işaretçisi üzerinde **Release** çağrısı yapmalıdır.

C++

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

Her iki durumda da **Release** çağrısının, işaretçi kapsam dışına çıkmadan önce gerçekleşen son şey olduğuna dikkat edin. Ayrıca **Release çağrısının** yalnızca **HRESULT**'un başarılı olup olmadığını test ettikten sonra yapıldığına dikkat edin. Örneğin, **CoCreateInstance** çağrısı başarısız olursa, *pFileOpen* işaretçisi geçerli değildir. Bu nedenle, işaretçi üzerinde **Release çağrısı yapmak bir hata** olacaktır.

Sonraki

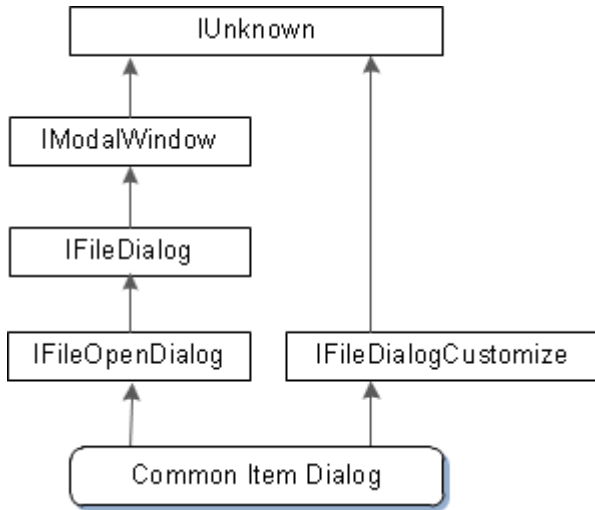
[Bir Nesneden Arayüz İsteme](#)

Bir Nesneden Arayüz İsteme

Makale - 08/19/2020 - Okumak için 2 dakika

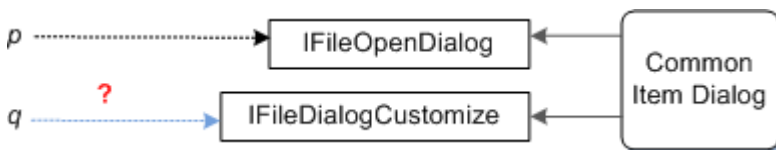
Daha önce bir nesnenin birden fazla arayüzü uygulayabileceğini görmüştük. Common Item Dialog nesnesi bunun gerçek dünyadaki bir örneğidir. En tipik kullanımları desteklemek için, nesne **IFileOpenDialog** arayüzünü uygular. Bu arayüz, iletişim kutusunu görüntülemek ve seçilen dosya hakkında bilgi almak için temel yöntemleri tanımlar. Ancak daha gelişmiş kullanım için, nesne **IFileDialogCustomize** adlı bir arayüzü de uygular. Bir program, yeni UI kontrolleri ekleyerek iletişim kutusunun görünümünü ve davranışını özelleştirmek için bu arayüzü kullanabilir.

Her COM arayüzünün doğrudan ya da dolaylı olarak **IUnknown** arayüzünden miras alması gerektiğini hatırlayın. Aşağıdaki diyagramda Common Item Dialog nesnesinin kalıtımı gösterilmektedir.



Diyagramdan da görebileceğiniz gibi, **IFileOpenDialog**'un doğrudan atası **IFileDialog** arayüzüdür ve bu arayüz de **IModalWindow**'u miras alır. **IFileOpenDialog**'dan **IModalWindow**'a miras zincirinde yukarı çıktıkça, arayüzler giderek genelleşen pencere işlevselliğini tanımlar. Son olarak, **IModalWindow** arayüzü **IUnknown**'ı miras alır. Common Item Dialog nesnesi de ayrı bir miras zincirinde bulunan **IFileDialogCustomize**'i uygular.

Şimdi **IFileOpenDialog** arayüzüne bir işaretçiniz olduğunu varsayalım. **IFileDialogCustomize** arayüzüne nasıl bir işaretçi elde edersiniz?



IFileOpenDialog işaretçisini basitçe bir **IFileDialogCustomize** işaretçisine dönüştürmek işe yaramayacaktır. Bir miras hiyerarşisi boyunca "çapraz döküm" yapmanın güvenilir bir

yolu yoktur.

dile bağılı bir özellik olan çalışma zamanı tür bilgisinin (RTTI) bir biçimi.

COM yaklaşımı, nesneye giden bir kanal olarak ilk arayüzü kullanarak nesneden size bir **IFileDialogCustomize** işaretçisi vermesini *istemektir*. Bu, ilk arayüz işaretçisinden **IUnknown::QueryInterface** yöntemi çağrılarak yapılır. **QueryInterface**'i, C++'daki **dynamic_cast** anahtar sözcüğünün dilden bağımsız bir versiyonu olarak düşünebilirsiniz.

QueryInterface yöntemi aşağıdaki imzaya sahiptir:

sözdizimi

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

CoCreateInstance hakkında zaten bildiklerinize dayanarak, **QueryInterface**'in nasıl çalıştığını tahmin edebilirsiniz.

- ♦ *riid* parametresi, sorduğunuz arayüzü tanımlayan GUID'dir. **REFIID** veri tipi `const GUID&` için bir **typedef**'tir. Nesne zaten oluşturulmuş olduğundan, sınıf tanımlayıcısının (CLSID) gerekli olmadığına dikkat edin. Yalnızca arayüz tanımlayıcısı gereklidir.
- ♦ *ppvObject* parametresi arayüze bir işaretçi alır. Bunun veri türü parametresinin **void**** olması, **CoCreateInstance**'in bu veri türünü kullanmasıyla aynı nedenden kaynaklanır: **QueryInterface** herhangi bir COM arayüzünü sorgulamak için kullanılabilir, bu nedenle parametre güçlü bir şekilde yazılamaz.

Bir **IFileDialogCustomize** işaretçisi almak için **QueryInterface**'i şu şekilde çağırırsınız:

C++

```
hr = pFileOpen->QueryInterface(IID_IFileDialogCustomize,
    reinterpret_cast<void**>(&pCustom));
eğer (SUCCEEDED(hr))
{
    // Arayüzü kullanın. (Gösterilmemiştir.)
    // ...

    pCustom->Release();
}
başka
{
    // Hatayı ele alın.
}
```

Her zaman olduđu gibi, yöntemin başarısız olması ihtimaline karşı **HRESULT** dönüş değerini kontrol edin. Yöntem başarılı olursa, işaretçiyi kullanmayı bitirdiğinizde, [Nesnenin Yaşam Süresini Yönetme](#) bölümünde açıklandığı gibi **Release** ögesini çağırmanız gerekir.

Sonraki

[COM'da Bellek Tahsisi](#)

COM'da Bellek Tahsisi

Makale - 04/27/2021 - Okumak için 2 dakika

Bazen bir yöntem heap üzerinde bir bellek tamponu ayırır ve tamponun adresini çağırana döndürür. COM, heap üzerinde bellek ayırmak ve serbest bırakmak için bir çift fonksiyon tanımlar.

- ♦ **CoTaskMemAlloc** işlevi bir bellek bloğu tahsis eder.
- ♦ **CoTaskMemFree** işlevi, aşağıdaki işlevlerle ayrılmış bir bellek bloğunu serbest bırakır **CoTaskMemAlloc**.

Bu modelin bir örneğini [Aç iletişim kutusu örneğinde](#) görmüştük:

C++

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

GetDisplayName yöntemi bir dize için bellek ayırır. Dahili olarak, yöntem dizeyi ayırmak için **CoTaskMemAlloc** ögesini çağırır. Yöntem geri döndüğünde, *pszFilePath* yeni tamponun bellek konumuna işaret eder. Çağırana, belleği boşaltmak için **CoTaskMemFree**'yi çağırmaktan sorumludur.

COM neden kendi bellek ayırma işlevlerini tanımlar? Bunun bir nedeni, heap ayırıcı üzerinde bir soyutlama katmanı sağlamaktır. Aksi takdirde, bazı yöntemler **malloc** çağırırken diğerleri **new** çağırabilirdi. O zaman programınızın bazı durumlarda **free**, bazı durumlarda **delete** çağırması gerekirdi ve tüm bunları takip etmek hızla imkansız hale gelirdi. COM ayırma fonksiyonları tek tip bir yaklaşım oluşturur.

Bir başka husus da COM'un *ikili* bir standart olması, dolayısıyla belirli bir programlama diline bağlı olmamasıdır. Bu nedenle, COM herhangi bir dile özgü bellek ayırma biçimine güvenemez.

Sonraki

[COM Kodlama Uygulamaları](#)

COM Kodlama Uygulamaları

Makale - 09/08/2020 - Okumak için 6 dakika

Bu konu COM kodunuzu daha etkili ve sağlam hale getirmenin yollarını açıklamaktadır.

- [uuidof İşleci](#)
- [IID_PPV_ARGS Makrosu](#)
- [SafeRelease Kalıbı COM](#)
- [Akıllı İşaretçiler](#)

Bu __uuidof Operatör

Programınızı derlediğinizde, aşağıdakine benzer bağlayıcı hataları alabilirsiniz:

```
cözümlenmemis harici sembol "struct GUID const
```

Bu hata, bir GUID sabitinin dış bağlantı (**extern**) ile bildirildiği ve bağlayıcının sabitin tanımını bulamadığı anlamına gelir. Bir GUID sabitinin değeri genellikle statik bir kütüphane dosyasından dışa aktarılır. Microsoft Visual C++ kullanıyorsanız, **uuidof** işlecini kullanarak statik bir kitaplık bağlama ihtiyacından kaçınabilirsiniz. Bu operatör bir Microsoft dil uzantısıdır. Bir ifadeden GUID değeri döndürür. İfade bir arayüz türü adı, bir sınıf adı veya bir arayüz işaretçisi olabilir. Kullanma

uuidof, Common Item Dialog nesnesini aşağıdaki gibi oluşturabilirsiniz:

C++

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

Derleyici GUID değerini başlıktan çıkarır, bu nedenle kütüphane aktarımı gerekmez.

7 Not

GUID değeri, `declspec(uuid(` ...)) başlık içinde. Daha fazla bilgi için **declspec** belgelerine bakın Visual C++ belgelerinde.

IID_PPV_ARGS Makrosu

Hem [CoCreateInstance](#) hem de [QueryInterface](#)'in son parametrenin bir **void**** türüne zorlanmasını gerektirdiğini gördük. Bu, bir tür uyumsuzluğu potansiyeli yaratır. Aşağıdaki kod parçasını düşünün:

C++

```
// Yanlış!

IFileOpenDialog *pFileOpen;

hr = CoCreateInstance(
    uuidof(FileOpenDialog),
    NULL,
    CLSCTX_ALL,
    uuidof(IFileDialogCustomize),           // IID işaretçi tipiyle
    eşleşmiyor!                             // void**'e zorlayın.
    reinterpret_cast<void**>(&pFileOpen)
);
```

Bu kod [IFileDialogCustomize](#) arayüzünü ister, ancak bir [IFileOpenDialog](#) işaretçisi geçirir. `reinterpret_cast` ifadesi C++ tür sistemini atlatır, bu nedenle derleyici bu hatayı yakalayamaz. En iyi durumda, nesne istenen arayüzü uygulamıyorsa, çağrı basitçe başarısız olur. En kötü durumda, işlev başarılı olur ve eşleşmeyen bir işaretçiniz olur. Başka bir deyişle, işaretçi türü bellekteki gerçek vtable ile eşleşmez. Tahmin edebileceğiniz gibi, bu noktada iyi bir şey olamaz.

7 Not

Bir *vtable* (sanal yöntem tablosu), işlev işaretçilerinden oluşan bir tablodur. Vtable, COM'un bir yöntem çağrısını çalışma zamanında uygulamasına nasıl bağladığını gösterir. Tesadüfi olmayan bir şekilde, vtable'lar çoğu C++ derleyicisinin sanal yöntemleri uygulama şeklidir.

[IID_PPV_ARGS](#) makrosu bu hata sınıfından kaçınmaya yardımcı olur. Bu makroyu kullanmak için aşağıdaki kodu değiştirin:

C++

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

bununla:

C++

```
IID_PPV_ARGS(&pFileOpen)
```

Makro, arayüz tanımlayıcısı `uuidof(IFileOpenDialog)` için otomatik olarak ekler, böylece işaretçi türüyle eşleşmesi garanti edilir. İşte değiştirilmiş (ve doğru) kod:

C++

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

Aynı makroyu [QueryInterface](#) ile de kullanabilirsiniz:

C++

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

SafeRelease Kalıbı

Referans sayma, programlamada temelde kolay olan, ancak aynı zamanda sıkıcı olan ve yanlış yapmayı kolaylaştıran şeylerden biridir. Tipik hatalar şunları içerir:

- Bir arayüz işaretçisini kullanmayı bitirdiğinizde serbest bırakmamak. Bu hata sınıfı, nesneler yok edilmediği için programınızın bellek ve diğer kaynakları sızdırmasına neden olur.
- Geçersiz bir işaretçi ile [Release](#) çağrısı. Örneğin, nesne hiç oluşturulmadıysa bu hata meydana gelebilir. Bu hata kategorisi muhtemelen programınızın çökmesine neden olacaktır.
- [Release](#) çağrıldıktan sonra bir arayüz işaretçisine dereferans verilmesi. Bu hata programınızın çökmesine neden olabilir. Daha da kötüsü, programınızın daha sonra rastgele bir zamanda çökmesine neden olabilir ve bu da orijinal hatanın izini sürmeyi zorlaştırır.

Bu hatalardan kaçınmanın bir yolu, [Release](#) işlevini işaretçiyi güvenli bir şekilde serbest bırakan bir işlev aracılığıyla çağırmaktır. Aşağıdaki kodda bunu yapan bir fonksiyon gösterilmektedir:

C++

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

Bu fonksiyon parametre olarak bir COM arayüz işaretçisi alır ve aşağıdakileri yapar:

1. İşaretçinin **NULL** olup olmadığını kontrol eder.
2. İşaretçi **NULL** değilse **Release** ögesini çağırır.
3. İşaretçiyi **NULL** olarak ayarlar.

İşte `SafeRelease`'in nasıl kullanılacağına dair bir örnek:

C++

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

    HRESULT hr = CoCreateInstance( uuidof(FileOpenDialog), NULL,
        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));
    eğer (SUCCEEDED(hr))
    {
        // Nesneyi kullanın.
    }
    SafeRelease(&pFileOpen);
}
```

CoCreateInstance başarılı olursa, **CoCreateInstance** `SafeRelease` işaretçiyi serbest bırakır. Eğer çağırısı başarısız olur, *pFileOpen* **NULL** olarak kalır. `SafeRelease` fonksiyonu

Bu ve **Release** çağrısını atlar. aşağıdakileri kontrol eder

Burada gösterildiği gibi, aynı işaretçi üzerinde birden fazla çağrı yapmak da güvenlidir:

C++

```
// Gereksiz, ama tamam.
SafeRelease(&pFileOpen);
SafeRelease(&pFileOpen);
```

COM Akıllı İşaretçiler

`SafeRelease` Fonksiyon kullanışıdır, ancak iki şeyi hatırlamanızı gerektirir:

- ♦ Her arayüz işaretçisini **NULL** olarak başlatın.

- Her işaretçi kapsam dışına çıkmadan önce çağrılır.

Bir C++ programcısı olarak, muhtemelen bunlardan herhangi birini hatırlamak zorunda olmamanız gerektiğini düşünüyorsunuz. Sonuçta, C++'ın kurucuları ve yıkıcıları olmasının nedeni budur. Temel arayüz işaretçisini saran ve işaretçiye otomatik olarak başlatan ve serbest bırakan bir sınıfa sahip olmak güzel olurdu. Başka bir deyişle, şöyle bir şey istiyoruz:

C++

// Uyarı: Bu örnek tamamlanmamıştır.

```
şablon <class T> class
SmartPointer
{
    T* ptr;

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

Burada gösterilen sınıf tanımı eksiktir ve gösterildiği gibi kullanılamaz. En azından, bir kopya kurucusu, bir atama operatörü ve temel COM işaretçisine erişmek için bir yol tanımlamanız gerekir. Neyse ki, Microsoft Visual Studio zaten Aktif Şablon Kitaplığı'nın (ATL) bir parçası olarak bir akıllı işaretçi sınıfı sağladığı için bu işlerin hiçbirini yapmanıza gerek yoktur.

ATL akıllı işaretçi sınıfı **CComPtr** olarak adlandırılır. (Burada ele alınmayan bir **CComQIPtr** sınıfı da vardır.) İşte **CComPtr** kullanmak üzere yeniden yazılmış [İletişim Kutusu Aç](#) örneği.

C++

```

#include <windows.h>
#include <shobjidl.h>
#include <atlbase.h> // CComPtr bildirimini içerir.

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

        // FileOpenDialog nesnesini oluşturun.
        hr = pFileOpen.CoCreateInstance( uuidof(FileOpenDialog)); if
        (SUCCEEDED(hr))
        {
            // Aç iletişim kutusunu gösterin.
            hr = pFileOpen->Show(NULL);

            // İletişim kutusundan dosya adını alın. if
            (SUCCEEDED(hr))
            {
                CComPtr<IShellItem> pItem;
                hr = pFileOpen->GetResult(&pItem);
                if (SUCCEEDED(hr))
                {
                    PWSTR pszFilePath;
                    hr = pItem->GetDisplayName(SIGDN_FILESYSPATH,
&pszFilePath);

                    // Dosya adını kullanıcıya gösterin. if
                    (SUCCEEDED(hr))
                    {
                        MessageBox(NULL, pszFilePath, L "Dosya Yolu", MB_OK);
                        CoTaskMemFree(pszFilePath);
                    }
                }

                // pItem kapsam dışına çıkar.
            }

            // pFileOpen kapsam dışına çıkar.
        }
        CoUninitialize();
    }
    0 döndür;
}

```

Bu kod ile orijinal örnek arasındaki temel fark, bu sürümün **Release**'i açıkça çağırmasıdır. **CComPtr** örneği kapsam dışına çıktığında, yıkıcı temel işaretçi üzerinde **Release** çağırısı yapar.

CComPtr bir sınıf şablonudur. Şablon argümanı COM arayüz tipidir. **CComPtr** dahili olarak bu tipte bir işaretçi tutar. **CComPtr**, **operator->()** ve **operator&()** işlevlerini geçersiz kılar, böylece sınıf temel işaretçi gibi davranır. Örneğin, aşağıdaki kod **IFileOpenDialog::Show** yöntemini doğrudan çağtırmaya eşdeğerdir:

C++

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

CComPtr ayrıca, COM **CoCreateInstance** işlevini bazı varsayılan parametre değerleriyle çağıran bir **CComPtr::CoCreateInstance** yöntemi tanımlar. Bir sonraki örnekte gösterildiği gibi, gerekli tek parametre sınıf tanımlayıcısıdır:

C++

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

CComPtr::CoCreateInstance yöntemi yalnızca bir kolaylık olarak sağlanmıştır; tercih ederseniz COM **CoCreateInstance** işlevini çağırmaya devam edebilirsiniz.

Sonraki

[COM'da Hata İşleme](#)

COM'da Hata İşleme (Win32 ve C++ ile Başlarken)

Makale - 04/27/2021 - Okumak için 7 dakika

COM, bir yöntem veya işlev çağrısının başarılı veya başarısız olduğunu belirtmek için **HRESULT** değerlerini kullanır. Çeşitli SDK başlıkları çeşitli **HRESULT** sabitleri tanımlar. WinError.h içinde sistem genelinde ortak bir kod kümesi tanımlanmıştır. Aşağıdaki tabloda bu sistem geneli dönüş kodlarından bazıları gösterilmektedir.

Sabit	Sayısal değer	Açıklama
E_ACCESSDENIED	0x80070005	Erişim reddedildi.
E_FAIL	0x80004005	Belirtilmemiş hata.
E_INVALIDARG	0x80070057	Geçersiz parametre değeri.
E_OUTOFMEMORY	0x8007000E	Hafıza dışı.
E_POINTER	0x80004003	NULL bir işaretçi değeri için yanlış aktarıldı.
E_UNEXPECTED	0x8000FFFF	Beklenmedik bir durum.
S_OK	0x0	Başarılar.
S_FALSE	0x1	Başarılar.

"E_" ön ekine sahip tüm sabitler hata kodlarıdır. **S_OK** ve **S_FALSE** sabitlerinin her ikisi de başarı kodlarıdır. Muhtemelen COM yöntemlerinin %99'u başarılı olduklarında **S_OK** döndürür; ancak bu gerçeğin sizi yanıltmasına izin vermeyin. Bir yöntem başka başarı kodları da döndürebilir, bu nedenle her zaman **SUCCEEDED** veya **FAILED** makrosunu kullanarak hataları test edin. Aşağıdaki örnek kod, bir fonksiyon çağrısının başarısını test etmenin yanlış ve doğru yollarını göstermektedir.

C++

```
// Yanlış.
HRESULT hr = SomeFunction();
if (hr != S_OK)
{
    printf("Hata!\n"); // Kötü. hr başka bir başarı kodu olabilir.
}

// Sağa.
```



```
HRESULT hr = SomeFunction();
if (FAILED(hr))
{
    printf("Hata!\n");
}
```

S_FALSE başarı kodundan bahsetmek gerekir. Bazı metotlar **S_FALSE**'i kabaca, başarısızlık olmayan olumsuz bir durumu ifade etmek için kullanır. Ayrıca bir "no-op" u da gösterebilir - yöntem başarılı olmuştur, ancak hiçbir etkisi olmamıştır. Örneğin, **CoInitializeEx** işlevi aynı iş parçacığından ikinci kez çağrılırsa **S_FALSE** döndürür. Kodunuzda **S_OK** ve **S_FALSE** arasında ayırım yapmanız gerekiyorsa, değeri doğrudan test etmeli, ancak aşağıdaki örnek kodda gösterildiği gibi kalan durumları işlemek için **FAILED** veya **SUCCEEDED** kullanmalısınız.

C++

```
if (hr == S_FALSE)
{
    // Özel durumları ele alın.
}
else if (SUCCEEDED(hr))
{
    // Genel başarı durumunu ele alın.
}
başka
{
    // Hataları ele alın.
    printf("Hata!\n");
}
```

Bazı **HRESULT** değerleri Windows'un belirli bir özelliğine veya alt sistemine özgüdür. Örneğin, Direct2D grafik API'si, programın desteklenmeyen bir piksel biçimi kullandığı anlamına gelen **D2DERR_UNSUPPORTED_PIXEL_FORMAT** hata kodunu tanımlar. MSDN belgeleri genellikle bir yöntemin döndürebileceği belirli hata kodlarının bir listesini verir. Ancak, bu listelerin kesin olduğunu düşünmemelisiniz. Bir yöntem her zaman belgelerde listelenmeyen bir **HRESULT** değeri döndürebilir. Yine, **SUCCEEDED** ve **FAILED** makrolarını kullanın. Belirli bir hata kodu için test yapıyorsanız, varsayılan bir durum da ekleyin.

C++

```
if (hr == D2DERR_UNSUPPORTED_PIXEL_FORMAT)
{
    // Desteklenmeyen bir piksel formatının özel durumunu ele alın.
}
else if (FAILED(hr))
{
    // Diğer hataları ele alın.
}
```

Hata İşleme Kalıpları

Bu bölümde, COM hatalarını yapılandırılmış bir şekilde ele almak için bazı kalıplar incelenmektedir. Her modelin avantajları ve dezavantajları vardır. Bir dereceye kadar, seçim bir zevk meselesidir. Mevcut bir proje üzerinde çalışıyorsanız, bu projede belirli bir stili yasaklayan kodlama yönergeleri zaten mevcut olabilir. Hangi kalıbı benimsediğinizden bağımsız olarak, sağlam kod aşağıdaki kurallara uyacaktır.

- ♦ Bir **HRESULT** döndüren her yöntem veya işlev için, devam etmeden önce dönüş değerini kontrol edin.
- ♦ Kaynakları kullandıktan sonra serbest bırakın.
- ♦ **NULL** işaretçiler gibi geçersiz veya başlatılmamış kaynaklara erişmeye
- ♦ çalışmayın. Bir kaynağı serbest bıraktıktan sonra kullanmaya çalışmayın.

Bu kuralları akılda tutarak, işte hataları ele almak için dört model.

- ♦ İç içe if'ler
- ♦ Basamaklı
- ♦ if'ler
- ♦ Başarısızlık durumunda atlama
- ♦ Başarısızlık durumunda atma

İç içe if'ler

HRESULT döndüren her çağrıdan sonra, başarıyı test etmek için bir **if** deyimi kullanın. Ardından, bir sonraki yöntem çağrısını **if** deyiminin kapsamı içine yerleştirin. Daha fazla **if deyimi** gerektiği kadar derine yerleştirilebilir. Bu modüldeki önceki kod örneklerinin hepsi bu kalıbı kullanmıştır, ancak burada tekrar yer almaktadır:

C++

```

HRESULT ShowDialog()
{
    IFileOpenDialog *pFileOpen;

    HRESULT hr = CoCreateInstance( uuidof(FileOpenDialog), NULL,
        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));
    eğer (SUCCEEDED(hr))
    {
        hr = pFileOpen->Show(NULL);
        if (SUCCEEDED(hr))
        {
            IShellItem *pItem;
            hr = pFileOpen->GetResult(&pItem);
            if (SUCCEEDED(hr))
            {
                // pItem kullanın
                (gösterilmemiştir). pItem-
                >Release();
            }
        }
        pFileOpen->Release();
    }
    dönüş sa a t i;
}

```

Avantajlar

- Değişkenler minimum kapsam ile bildirilebilir. Örneğin, *pItem* kullanılana kadar bildirilmez.
- Her **if** deyimi içinde belirli değişmezler doğrudur: Önceki tüm çağrılar başarılı olmuştur ve edinilen tüm kaynaklar hala geçerlidir. Önceki örnekte, program en içteki **if** **deyimine** ulaştığında, hem *pItem* hem de *pFileOpen* *öğelerinin* geçerli olduğu bilinmektedir.
- Arayüz işaretçilerinin ve diğer kaynakların ne zaman serbest bırakılacağı açıktır. Bir kaynağı, kaynağı edinen çağrının hemen ardından gelen **if** deyiminin sonunda serbest bırakırsınız.

Dezavantajlar

- Bazı insanlar derin yuvaları okumayı zor bulur.
- Hata işleme, diğer dallanma ve döngü ifadeleriyle karıştırılır. Bu, genel program mantığının takip edilmesini zorlaştırabilir.

Basamaklı if'ler

Her yöntem çağrısından sonra, başarıyı test etmek için bir **if** deyimi kullanın. Yöntem başarılı olursa, bir sonraki yöntem çağrısını **if** bloğunun içine yerleştirin. Ancak daha fazla **if** **deyimini** iç içe yerleştirmek yerine, sonraki her **SUCCEEDED** testini bir önceki **if**

bloğundan sonra yerleştirin. Herhangi bir yöntem başarısız olursa, fonksiyonun sonuna ulaşılanaya kadar kalan tüm **SUCCEEDED** testleri başarısız olur.

C++

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

    HRESULT hr = CoCreateInstance( uuidof(FileOpenDialog), NULL,

        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));

    eğer (SUCCEEDED(hr))
    {
        hr = pFileOpen->Show(NULL);
    }
    eğer (SUCCEEDED(hr))
    {
        hr = pFileOpen->GetResult(&pItem);
    }
    eğer (SUCCEEDED(hr))
    {
        // pItem kullanın (gösterilmemiştir).
    }

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

Bu modelde, kaynakları fonksiyonun en sonunda serbest bırakırsınız. Bir hata oluşursa, fonksiyondan çıkıldığında bazı işaretçiler geçersiz olabilir. Geçersiz bir işaretçi üzerinde **Release** çağrısı yapmak programı çökertir (ya da daha kötüsü), bu nedenle tüm işaretçileri **NULL** olarak başlatmalı ve serbest bırakmadan önce **NULL olup olmadıklarını kontrol** etmelisiniz. Bu örnekte fonksiyonu; akıllı işaretçiler de iyi bir seçimdir.

Bu kalıbı kullanırsanız, döngü yapıları konusunda dikkatli olmalısınız. Bir döngü içinde, herhangi bir çağrı başarısız olursa döngüden çıkın.

Avantajlar

- Bu model "iç içe ifs" modeline göre daha az iç içe geçme yaratır.
- Genel kontrol akışını görmek daha kolaydır.
- Kaynaklar kodun bir noktasında serbest bırakılır.

Dezavantajlar

- Tüm deęiřkenler fonksiyonun bařında bildirilmeli ve bařlatılmalıdır.
- Bir çağrı bařarısız olursa, iřlevden hemen çıkmak yerine iřlev birden fazla gereksiz hata denetimi yapar.
- Bir hatadan sonra kontrol akıřı fonksiyon boyunca devam ettięinden, fonksiyonun gövdesi boyunca geęersiz kaynaklara eriřmemeye dikkat etmelisiniz.
- Bir döngü içindeki hatalar özel bir durum gerektirir.

Bařarısızlıęa Atla

Her yöntem çağrısından sonra başarısızlığı test edin (başarılı değil). Başarısızlık durumunda, işlevin alt kısmına yakın bir etikete atlayın. Etiketten sonra, ancak işlevden çıkmadan önce kaynakları serbest bırakın.

C++

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

    HRESULT hr = CoCreateInstance( uuidof(FileOpenDialog), NULL,
        CLSCTX_INPROC_SERVER, IID_PPV_ARGS(&pFileOpen));
    eğer (FAILED(hr))
    {
        Tamamdır;
    }

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

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

    // pItem kullanın

    (gösterilmemiştir). done:
    // Temizle.
    SafeRelease(&pItem);
    SafeRelease(&pFileOpen);
    return hr;
}
```

Avantajlar

- Genel kontrol akışını görmek kolaydır.
- **FAILED** kontrolünden sonra kodun her noktasında, etikete atlamadıysanız, önceki tüm çağrıların başarılı olduğu garanti edilir.
- Kaynaklar kodun tek bir yerinde serbest bırakılır.

Dezavantajlar

- Tüm değişkenler fonksiyonun başında bildirilmeli ve başlatılmalıdır.

- ♦ Bazı programcılar kodlarında **goto** kullanmayı sevmezler. (Ancak, bu **goto** kullanımının son derece yapılandırılmış olduğuna dikkat edilmelidir; kod asla mevcut işlev çağrısının dışına atlamaz).
- ♦ **goto** deyimleri başlatıcıları atlar.

Başarısızlık üzerine atmak

Bir etikete atlamak yerine, bir yöntem başarısız olduğunda bir istisna fırlatabilirsiniz. İstisnalara karşı güvenli kod yazmaya alışkınsanız, bu daha deyimsel bir C++ tarzı oluşturabilir.

C++

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

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

void ShowDialog()
{
    dene
    {
        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));

        // pItem kullanın (gösterilmemiştir).
    }
    catch (_com_error err)
    {
        // Hatayı ele al.
    }
}
```

Bu örneğin arayüz işaretçilerini yönetmek için **CComPtr** sınıfını kullandığına dikkat edin. Genel olarak, kodunuz istisnalar atıyorsa RAI (Resource Acquisition is Initialization) modelini izlemelisiniz. Yani, her kaynak, yıkıcısı kaynağın doğru şekilde serbest bırakılmasını garanti eden bir nesne tarafından yönetilmelidir. Eğer bir

istisnası atıldığında, yıkıcının çağrılacağı garanti edilir. Aksi takdirde, programınız kaynak sızdırabilir.

Avantajlar

- ♦ İstisna işleme kullanan mevcut kodla uyumludur.
- ♦ Standart Şablon Kütüphanesi (STL) gibi istisnalar atan C++ kütüphaneleri ile uyumludur.

Dezavantajlar

- ♦ Bellek veya dosya tutamaçları gibi kaynakları yönetmek için C++ nesneleri
- ♦ gerektirir. İstisnalara karşı güvenli kod yazma konusunda iyi bir anlayış gerektirir.

Sonraki

[Modül 3. Windows Grafikleri](#)



Modül 3. Windows Grafikleri

Makale - 23.08.2019 - Okumak için 2 dakika

Bu serinin [1. Modülü](#) boş bir pencerenin nasıl oluşturulacağını göstermiştir. Modül [2](#), modern Windows API'lerinin birçoğunun temelini oluşturan Bileşen Nesne Modeli (COM) üzerinde küçük bir gezinti yaptı. Şimdi Modül 1'de oluşturduğumuz boş pencereye grafik ekleme zamanı.

Bu modül, Windows grafik mimarisine üst düzey bir genel bakış ile başlar. Daha sonra Windows 7'de tanıtılan güçlü bir grafik API'si olan Direct2D'ye bakacağız.

Bu bölümde

- [Windows Grafik Mimarisine Genel Bakış Masaüstü](#)
- [Pencere Yöneticisi](#)
- [Tutulan Moda Karşı Anlık Mod İlk](#)
- [Direct2D Programınız](#)
- [Direct2D ile Render Hedefleri, Aygıtlar ve](#)
- [Kaynaklar Çizimi](#)
- [Direct2D'de Renk Kullanarak DPI ve](#)
- [Cihazdan Bağımsız Pikseller](#)
- [Direct2D'de Dönüşümleri](#)
- [Uygulama Ek: Matris Dönüşümleri](#)

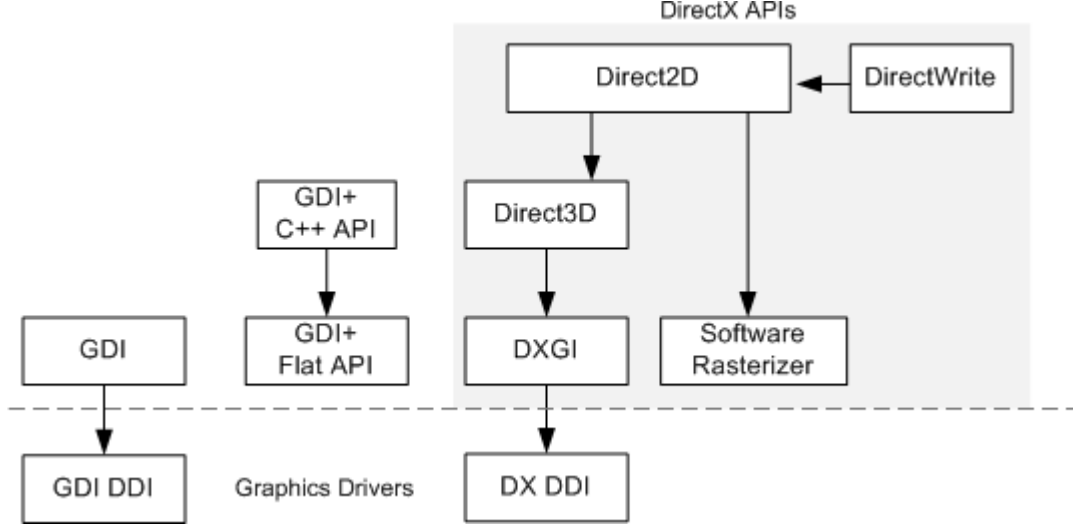
İlgili konular

[C++ ile Windows için Programlamayı Öğrenin](#)

Windows Grafik Mimarisine Genel Bakış

Makale - 23/08/2022 - Okumak için 3 dakika

Windows grafikler için çeşitli C++/COM API'leri sağlar. Bu API'ler aşağıdaki şemada gösterilmiştir.



- Graphics Device Interface (GDI) Windows için orijinal grafik arayüzüdür. GDI ilk olarak 16-bit Windows için yazılmış ve daha sonra 32-bit ve 64-bit Windows için güncellenmiştir.
- GDI+, Windows XP'de GDI'nin halefi olarak tanıtılmıştır. GDI+ kütüphanesi düz C fonksiyonlarını saran bir dizi C++ sınıfı aracılığıyla erişilir. NET Framework ayrıca **System.Drawing** ad alanında GDI+'ın yönetilen bir sürümünü de sağlar.
- Direct3D 3 boyutlu grafikleri destekler.
- Direct2D, GDI ve GDI+'ın halefi olan 2 boyutlu grafikler için modern bir API'dir.
- DirectWrite bir metin düzeni ve rasterleştirme motorudur. Rasterleştirilmiş metni çizmek için GDI veya Direct2D kullanabilirsiniz.
- DirectX Grafik Altyapısı (DXGI), çıktı için çerçevelerin sunulması gibi düşük seviyeli görevleri yerine getirir. Çoğu uygulama DXGI'yi doğrudan kullanmaz. Bunun yerine, grafik sürücüsü ile Direct3D arasında bir ara katman görevi görür.

Direct2D ve DirectWrite Windows 7'de tanıtılmıştır. Bunlar ayrıca Platform Güncellemesi aracılığıyla Windows Vista ve Windows Server 2008 için de mevcuttur. Daha fazla bilgi için [Windows Vista için Platform Güncellemesi](#) bölümüne bakın.

Direct2D bu modülün odak noktasıdır. Hem GDI hem de GDI+ Windows'ta desteklenmeye devam ederken, yeni programlar için Direct2D ve DirectWrite önerilir.

Bazı durumlarda, teknolojilerin bir karışımı daha pratik olabilir. Bu durumlar için Direct2D ve DirectWrite, GDI ile birlikte çalışacak şekilde tasarlanmıştır.

Sonraki bölümlerde Direct2D'nin bazı faydaları açıklanmaktadır.

Donanım Hızlandırma

Donanım hızlandırma terimi, CPU yerine grafik işleme birimi (GPU) tarafından gerçekleştirilen grafik hesaplamalarını ifade eder. Modern GPU'lar grafiklerin işlenmesinde kullanılan hesaplama türleri için son derece optimize edilmiştir. Genel olarak, bu iş CPU'dan GPU'ya ne kadar çok taşınırsa o kadar iyidir.

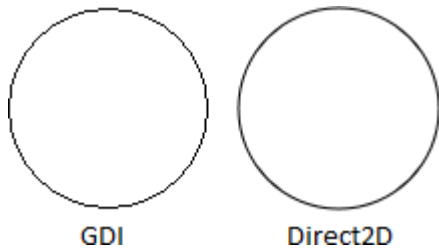
GDI belirli işlemler için donanım hızlandırmayı desteklese de, birçok GDI işlemi CPU'ya bağlıdır. Direct2D, Direct3D'nin üzerine yerleştirilmiştir ve GPU tarafından sağlanan donanım hızlandırmasından tam olarak yararlanır. GPU, Direct2D için gereken özellikleri desteklemiyorsa, Direct2D yazılım oluşturmaya geri döner. Genel olarak, Direct2D çoğu durumda GDI ve GDI+'dan daha iyi performans gösterir.

Şeffaflık ve Kenar Yumuşatma

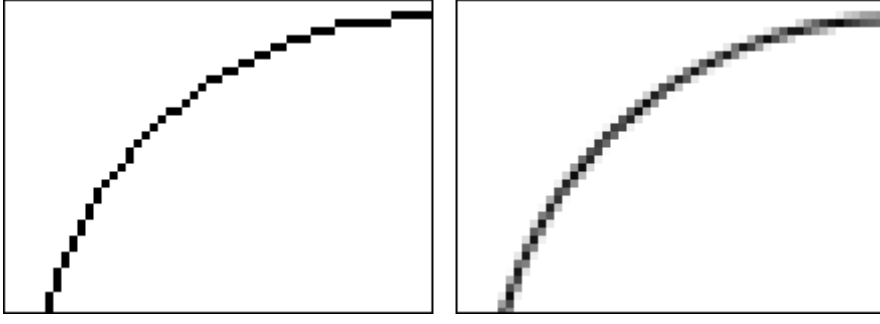
Direct2D tamamen donanım hızlandırmalı alfa harmanlamayı (şeffaflık) destekler.

GDI alfa harmanlama için sınırlı desteğe sahiptir. Çoğu GDI işlevi alfa karıştırmayı desteklemez, ancak GDI bitblt işlemi sırasında alfa karıştırmayı destekler. GDI+ şeffaflığı destekler, ancak alfa harmanlama CPU tarafından gerçekleştirilir, bu nedenle donanım hızlandırmadan faydalanmaz.

Donanım hızlandırmalı alfa harmanlama aynı zamanda kenar yumuşatmayı da mümkün kılar. Örtüşme, sürekli bir fonksiyonun örneklenmesinden kaynaklanan bir *yapaylıktır*. Örneğin, eğri bir çizgi piksellere dönüştürüldüğünde, örtüşme pürüzlü bir görünüme neden olabilir. Örtüşmenin neden olduğu yapaylıkları azaltan herhangi bir teknik, bir tür örtüşme önleme olarak kabul edilir. Grafiklerde kenar yumuşatma, kenarları arka planla harmanlayarak yapılır. Örneğin, burada GDI tarafından çizilen bir daire ve Direct2D tarafından çizilen aynı daire görülmektedir.



Bir sonraki resim her bir dairenin detayını göstermektedir.



GDI tarafından çizilen daire (solda) bir eğriye yaklaşan siyah piksellerden oluşur. Direct2D tarafından çizilen daire (sağda) daha yumuşak bir eğri oluşturmak için harmanlama kullanır.

GDI, geometri (çizgiler ve eğriler) çizerken kenar yumuşatmayı desteklemez. GDI, ClearType kullanarak kenar yumuşatmalı metin çizebilir; ancak bunun dışında GDI metni de kenar yumuşatmalıdır. Kenar yumuşatma özellikle metin için belirgindir, çünkü pürüzlü çizgiler yazı tipi tasarımını bozar ve metni daha az okunabilir hale getirir. GDI+ kenar yumuşatmayı desteklese de, CPU tarafından uygulanır, bu nedenle performans Direct2D kadar iyi değildir.

Vektör Grafikleri

Direct2D *vektör grafiklerini* destekler. Vektör grafiklerinde, çizgileri ve eğrileri temsil etmek için matematiksel formüller kullanılır. Bu formüller ekran çözünürlüğüne bağlı değildir, bu nedenle rastgele boyutlara ölçeklendirilebilirler. Vektör grafikleri, bir görüntünün farklı monitör boyutlarını veya ekran çözünürlüklerini desteklemek için ölçeklendirilmesi gerektiğinde özellikle kullanışlıdır.

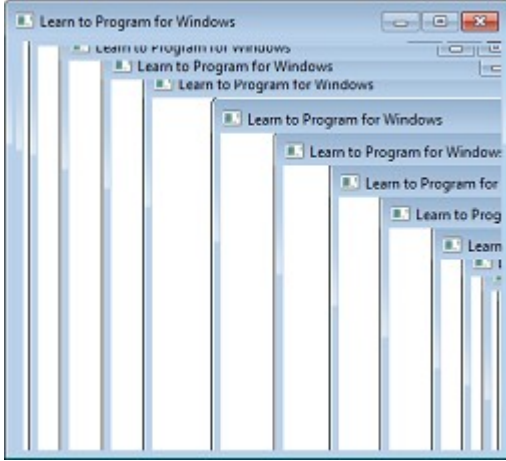
Sonraki

[Masaüstü Pencere Yöneticisi](#)

Masaüstü Pencere Yöneticisi

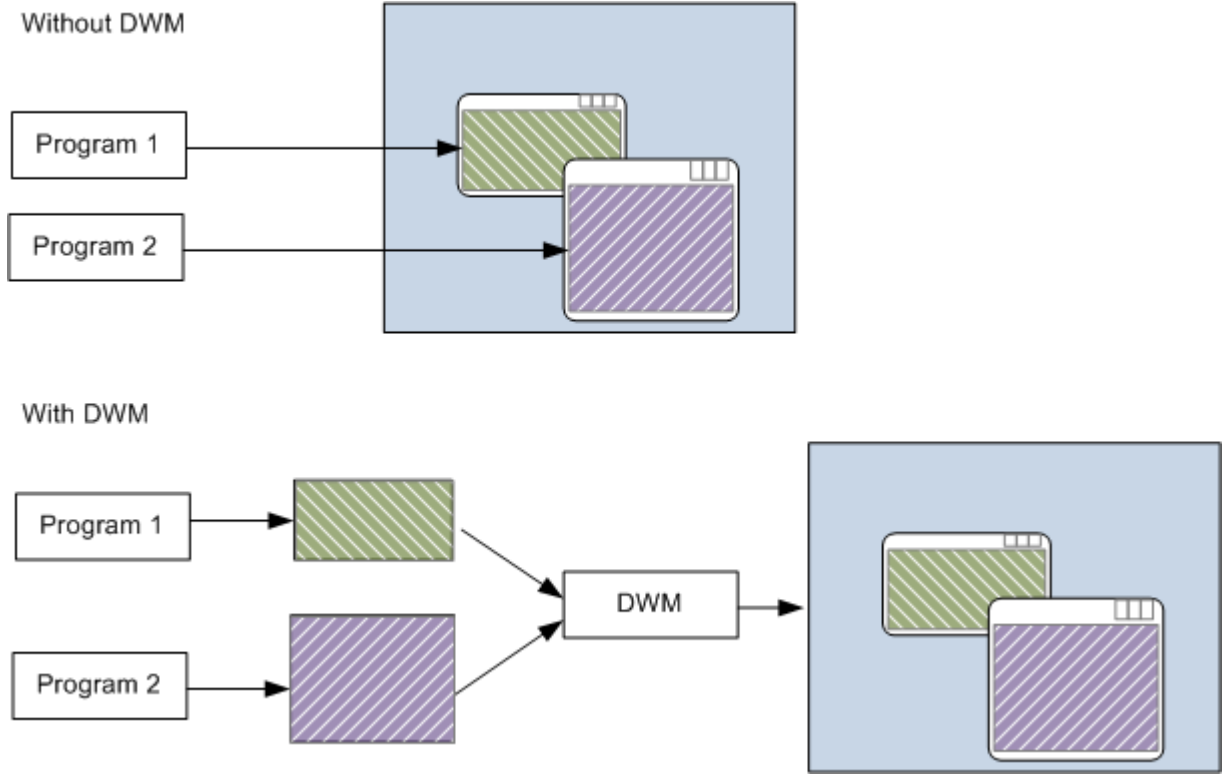
Makale - 04/27/2021 - Okumak için 2 dakika

Windows Vista'dan önce, bir Windows programı doğrudan ekrana çizim yapardı. Başka bir deyişle, program doğrudan video kartı tarafından gösterilen bellek tamponuna yazardı. Bu yaklaşım, bir pencere kendini doğru şekilde yeniden boyamazsa görsel yapaylıklara neden olabilir. Örneğin, kullanıcı bir pencereyi başka bir pencerenin üzerine sürüklerse ve alttaki pencere kendini yeterince hızlı yeniden boyamazsa, en üstteki pencere bir iz bırakabilir:



İzin nedeni her iki pencerenin de aynı bellek alanına boyanmasıdır. En üstteki pencere sürüklendiğinde, altındaki pencerenin yeniden boyanması gerekir. Yeniden boyama çok yavaşsa, önceki resimde gösterilen yapaylıklara neden olur.

Windows Vista, Masaüstü Pencere Yöneticisi'ni (DWM) tanıtarak pencerelerin çizilme şeklini temelden değiştirdi. DWM etkinleştirildiğinde, bir pencere artık doğrudan ekran tamponuna çizilmez. Bunun yerine, her pencere ekran dışı *yüzey olarak* da adlandırılan bir ekran dışı bellek tamponuna çizilir. DWM daha sonra bu yüzeyleri ekrana birleştirir.



DWM, eski grafik mimarisine göre çeşitli avantajlar sağlamaktadır.

- Daha az yeniden boyama mesajı. Bir pencere başka bir pencere tarafından engellendiğinde, engellenen pencerenin kendini yeniden boyaması gerekmez.
- Artifaktlar azaltıldı. Önceden, bir pencereyi sürüklemek açıklandığı gibi görsel yapaylıklar oluşturabiliyordu.
- Görsel efektler. DWM ekranın birleştirilmesinden sorumlu olduğu için, pencerenin yarı saydam ve bulanık alanlarını oluşturabilir.
- Yüksek DPI için otomatik ölçeklendirme. Ölçekleme, yüksek DPI ile başa çıkmak için ideal bir yol olmasa da, yüksek DPI ayarları için tasarlanmamış eski uygulamalar için uygun bir yedektir. (Bu konuya daha sonra [DPI ve Cihazdan Bağımsız Pikseller](#) bölümünde döneceğiz).
- Alternatif görünüm. DWM ekran dışı yüzeyleri çeşitli ilginç şekillerde kullanabilir. Örneğin, DWM Windows Flip 3D, küçük resimler ve animasyonlu geçişlerin arkasındaki teknolojidir.

Ancak DWM'nin etkinleştirileceğinin garanti edilmediğini unutmayın. Grafik kartı DWM sistem gereksinimlerini desteklemeyebilir ve kullanıcılar **Sistem Özellikleri** kontrol paneli aracılığıyla DWM'yi devre dışı bırakabilir. Bu, programınızın DWM'nin yeniden boyama davranışına güvenmemesi gerektiği anlamına gelir. Doğru şekilde yeniden boyandığından emin olmak için programınızı DWM devre dışıyken test edin.

Sonraki

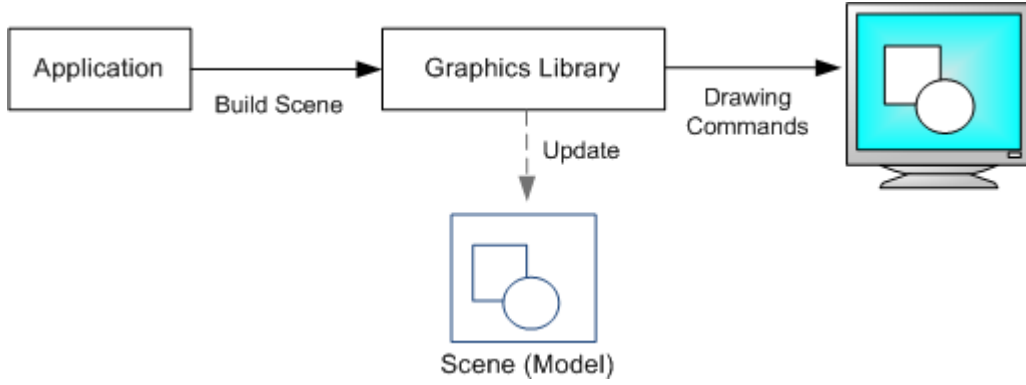
[Bekletme Moduna Karşı Anlık Mod](#)

Bekletme Moduna Karşı Anlık Mod

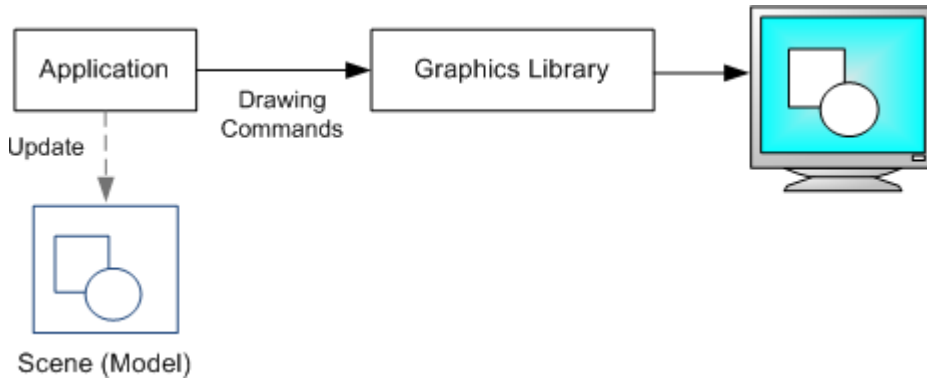
Makale - 23.08.2019 - Okumak için 2 dakika

Grafik API'leri *tutulan modlu API'ler* ve *anında modlu API'ler* olarak ikiye ayrılabilir. Direct2D bir anlık mod API'sidir. Windows Presentation Foundation (WPF), tutulan mod API'sine bir örnektir.

Tutulan mod API'si bildirimseldir. Uygulama, şekiller ve çizgiler gibi grafik ilkellerinden bir sahne oluşturur. Grafik kütüphanesi sahnenin bir modelini bellekte saklar. Bir kare çizmek için, grafik kütüphanesi sahneyi bir dizi çizim komutuna dönüştürür. Kareler arasında, grafik kütüphanesi sahneyi bellekte tutar. Oluşturulanı değiştirmek için, uygulama sahneyi güncellemek üzere bir komut verir; örneğin, bir şekil eklemek veya kaldırmak için. Ardından kütüphane sahnenin yeniden çizilmesinden sorumludur.



Anında mod API'si prosedürelidir. Her yeni kare çizildiğinde, uygulama çizim komutlarını doğrudan verir. Grafik kütüphanesi kareler arasında bir sahne modeli saklamaz. Bunun yerine, uygulama sahneyi takip eder.



Tutulan mod API'lerinin kullanımı daha basit olabilir, çünkü API başlatma, durum bakımı ve temizleme gibi işlerin çoğunu sizin için yapar. Öte yandan, API kendi sahne modelini dayattığı için genellikle daha az esnektirler. Ayrıca, genel amaçlı bir sahne modeli sağlaması gerektiğinden, tutulan modlu bir API daha yüksek bellek gereksinimlerine sahip olabilir. Anında mod API ile hedeflenen optimizasyonları uygulayabilirsiniz.

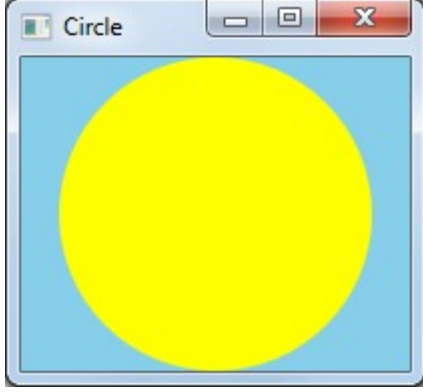
Sonraki

İlk Direct2D Programınız

İlk Direct2D Programınız

Makale - 08/19/2020 - Okumak için 2 dakika

İlk Direct2D programımızı oluşturalım. Program süslü bir şey yapmıyor - sadece pencerenin istemci alanını dolduran bir daire çiziyor. Ancak bu program birçok temel Direct2D kavramını tanıtmaktadır.



İşte Circle programı için kod listesi. Program şu kodları yeniden kullanır `BaseWindow`

[Uygulama Durumunu Yönetme](#) konusunda tanımlanan sınıf. Daha sonraki konular kodu ayrıntılı olarak inceleyecektir.

C++

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

#include "basewin.h"

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

class MainWindow : public BaseWindow<MainWindow>
{
    ID2D1Factory          *pFactory;
    ID2D1HwndRenderTarget *pRenderTarget;
    ID2D1SolidColorBrush  *pFırça;
    D2D1_ELLIPSE           elips;

    geçersiz CalculateLayout();
    HRESULT CreateGraphicsResources();
    void DiscardGraphicsResources();

    geçersiz OnPaint();
    geçersiz
```

```

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 kazanır;

    if (!win.Create(L "Circle", WS_OVERLAPPEDWINDOW))
    {
        O döndür;
    }

    ShowWindow(win.Window(), nCmdShow);
}

```

```

// Mesaj döngüsünü çalıştırın.

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

O döndür;
}

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.
        }
        O döndür;

    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);
}

```

Visual Studio projesinin tamamını [Direct2D Circle Sample](#) adresinden indirebilirsiniz.

D2D1 Ad Alanı

D2D1 isim alanı yardımcı fonksiyonlar ve sınıflar içerir. Bunlar kesinlikle Direct2D API'sinin bir parçası değildir - Direct2D'yi bunları kullanmadan programlayabilirsiniz - ancak kodunuzu basitleştirmeye yardımcı olurlar. **D2D1** isim alanı şunları içerir:

- Renk deęerleri oluřturmak iin bir **ColorF** sınıfı.
- Dnřm matrisleri oluřturmak iin bir **Matrix3x2F**.
- Direct2D yapılarını bařlatmak iin bir dizi fonksiyon.

Bu modl boyunca **D2D1** ad alanının rneklerini greceksiniz.

Sonraki

[Hedefleri, Cihazları ve Kaynakları İřleme](#)

İlgili konular

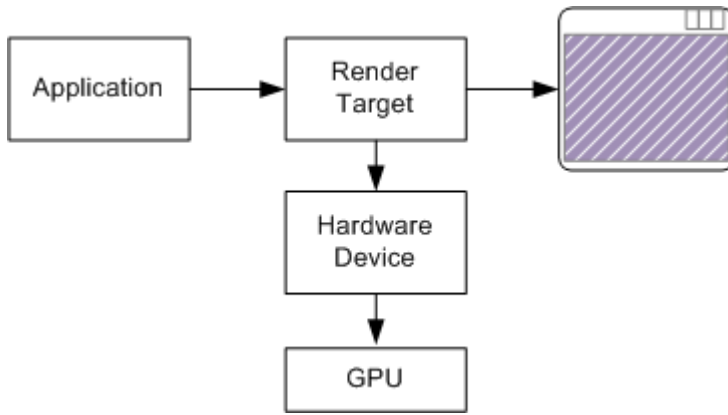
[Direct2D Daire rneęi](#)

Hedefleri, Cihazları ve Kaynakları İşleme

Makale - 08/19/2020 - Okumak için 3 dakika

Bir *render hedefi* basitçe programınızın çizim yapacağı konumdur. Tipik olarak, render hedefi bir penceredir (özellikle pencerenin istemci alanı). Bellekte görüntülenmeyen bir bitmap de olabilir. Bir render hedefi **ID2D1RenderTarget** arayüzü tarafından temsil edilir.

Aygıt, pikselleri gerçekte çizen şeyi temsil eden bir soyutlamadır. Bir donanım cihazı daha hızlı performans için GPU'yu kullanırken, bir yazılım cihazı CPU'yu kullanır. Uygulama aygıtı oluşturmaz. Bunun yerine, uygulama render hedefini oluşturduğunda cihaz dolaylı olarak oluşturulur. Her render hedefi, donanım veya yazılım olmak üzere belirli bir cihazla ilişkilendirilir.



Kaynak, programın çizim için kullandığı bir nesnedir. Direct2D'de tanımlanan bazı kaynak örnekleri aşağıda verilmiştir:

- **Fırça**. Çizgilerin ve bölgelerin nasıl boyanacağını kontrol eder. Fırça türleri arasında düz renkli fırçalar ve degrade fırçalar bulunur.
- **Kontur stili**. Bir çizginin görünümünü kontrol eder; örneğin, kesikli veya düz.
- **Geometri**. Çizgi ve eğrilerden oluşan bir koleksiyonu temsil eder.
- **Mesh**. Üçgenlerden oluşan bir şekil. Mesh verileri, render işleminden önce dönüştürülmesi gereken geometri verilerinin aksine doğrudan GPU tarafından tüketilebilir.

Render hedefleri de bir kaynak türü olarak kabul edilir.

Bazı kaynaklar donanım hızlandırmadan yararlanır. Bu tür bir kaynak her zaman donanım (GPU) veya yazılım (CPU) olmak üzere belirli bir cihazla ilişkilendirilir. Bu tür kaynaklara cihaza *bağımlı* kaynak denir. Fırçalar ve kafesler cihaza bağımlı kaynaklara örnektir. Cihaz kullanılamaz hale gelirse, kaynak yeni bir cihaz için yeniden oluşturulmalıdır.

Diğer kaynaklar, hangi cihazın kullanıldığına bakılmaksızın CPU belleğinde tutulur. Bu kaynaklar cihazdan *bağımsızdır*, çünkü belirli bir cihazla ilişkili değildirler. Cihaz değiştiğinde cihazdan bağımsız kaynakların yeniden oluşturulması gerekmez. Kontur stilleri ve geometriler cihazdan bağımsız kaynaklardır.

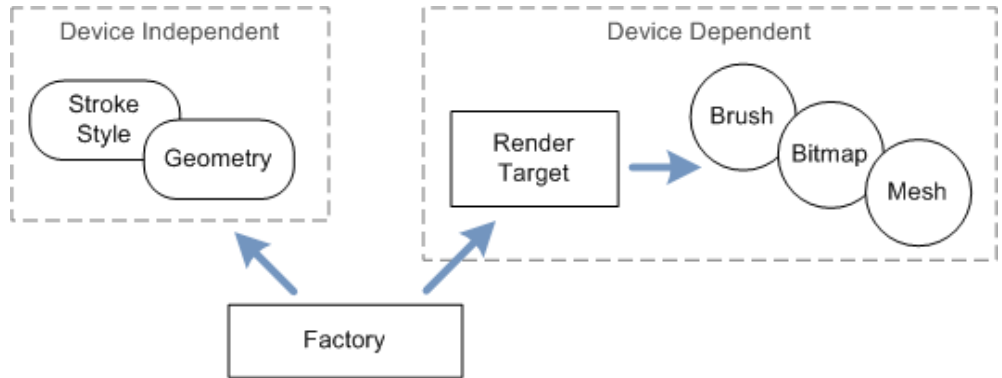
Her kaynak için MSDN belgeleri, kaynağın cihaza bağımlı mı yoksa cihazdan bağımsız mı olduğunu belirtir. Her kaynak türü **ID2D1Resource**'tan türetilen bir arayüzle temsil edilir. Örneğin, fırçalar **ID2D1Brush** arayüzü ile temsil edilir.

Direct2D Fabrika Nesnesi

Direct2D kullanırken ilk adım Direct2D factory nesnesinin bir örneğini oluşturmaktır. Bilgisayar programlamada *fabrika*, başka nesneler oluşturan bir nesnedir. Direct2D fabrikası aşağıdaki nesne türlerini oluşturur:

- Hedefleri oluşturun.
- Kontur stilleri ve geometriler gibi cihazdan bağımsız kaynaklar.

Fırçalar ve bitmap'ler gibi cihaza bağlı kaynaklar, render hedef nesnesi tarafından oluşturulur.



Direct2D fabrika nesnesini oluşturmak için **D2D1CreateFactory** işlevini çağırın.

C++

```
ID2D1Factory *pFactory = NULL;

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

İlk parametre, oluşturma seçeneklerini belirten bir bayraktır.

D2D1_FACTORY_TYPE_SINGLE_THREADED bayrağı, Direct2D'yi birden fazla iş parçacığından çağırmayacağınız anlamına gelir. Birden fazla iş parçacığından yapılan çağrılarını desteklemek için

D2D1_FACTORY_TYPE_MULTI_THREADED. Programınız Direct2D'ye çağrı yapmak için tek bir iş parçacığı kullanıyorsa, tek iş parçacıklı seçenek daha verimlidir.

D2D1CreateFactory işlevinin ikinci parametresi, aşağıdaki öğeye bir işaretçi alır **ID2D1Factory** arayüzü.

Direct2D fabrika nesnesini ilk **WM_PAINT** mesajından önce oluşturmalsınız. Bu **WM_CREATE** mesaj işleyicisi fabrikayı oluşturmak için iyi bir yerdir:

C++

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

Direct2D Kaynakları Oluşturma

Circle programı aşağıdaki cihaza bağlı kaynakları kullanır:

- Uygulama penceresiyle ilişkilendirilmiş bir render hedefi.
- Daireyi boyamak için düz renkli bir fırça.

Bu kaynakların her biri bir COM arayüzü ile temsil edilir:

- **ID2D1HwndRenderTarget** arayüzü render hedefini temsil eder.
- **ID2D1SolidColorBrush** arayüzü fırçayı temsil eder.

Circle programı bu arayüzlerin işaretçilerini sınıfın üye değişkenleri olarak saklar:

MainWindow

C++

```
ID2D1HwndRenderTarget *pRenderTarget;
ID2D1SolidColorBrush *pFırça;
```

Aşağıdaki kod bu iki kaynağı oluşturur.

C++

```
HRESULT MainWindow::CreateGraphicsResources()
{
    HRESULT hr = S_OK;

    if (pRenderTarget == NULL)
    {
        RECT rc;
        GetClientRect(hwnd, &rc);
        pRenderTarget = new ID2D1HwndRenderTarget(
            hr, hwnd, rc, D2D1_RENDER_TARGET_PROPERTIES{
                D2D1_RT_SINGLE_THREADED, D2D1_RENDER_TARGET_TYPE_DEFAULT,
                D2D1_RENDER_OPTIONS_DEFAULT, D2D1_COLOR_SPACE_DEFAULT,
                D2D1_ALPHA_MODE_PREMULTIPLIED
            }
        );
    }

    pFırça = new ID2D1SolidColorBrush(
        hr, D2D1_COLOR_F(0.5f, 0.5f, 0.5f, 1.0f)
    );
}
```


Bir pencere için **render** hedefi oluşturmak üzere **ID2D1Factory::CreateHwndRenderTarget** Direct2D fabrikasındaki yöntem.

- İlk parametre, her tür render hedefi için ortak olan seçenekleri belirtir. Burada, **D2D1::RenderTargetProperties** yardımcı fonksiyonunu çağırarak varsayılan seçenekleri aktarıyoruz.
- İkinci parametre, pencerenin tutamacını ve render hedefinin boyutunu piksel cinsinden belirtir.
- Üçüncü parametre bir **ID2D1HwndRenderTarget** işaretçisi alır.

Düz renk fırçasını oluşturmak için, render hedefinde **ID2D1RenderTarget::CreateSolidColorBrush** yöntemini çağırın. Renk, **D2D1_COLOR_F** değeri olarak verilir. Direct2D'de renkler hakkında daha fazla bilgi için [Direct2D'de Renk Kullanımı](#) bölümüne bakın.

Ayrıca, render hedefi zaten mevcutsa **CreateGraphicsResource** yöntem hiçbir şey yapmadan **S_OK** döndürür. Bu tasarımın nedeni bir sonraki konuda açıklığa kavuşacaktır.

Sonraki

[Direct2D ile Çizim](#)

Direct2D ile Çizim

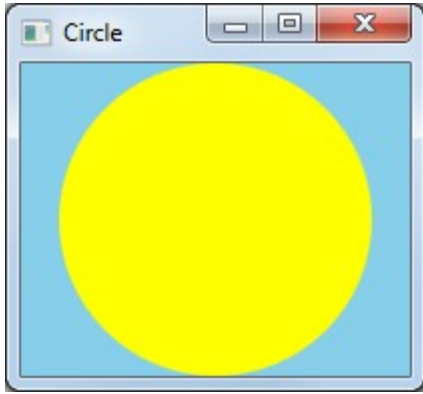
Makale - 08/19/2020 - Okumak için 3 dakika

Grafik kaynaklarınızı oluşturduktan sonra çizim yapmaya hazırsınız demektir.

Elips Çizme

[Circle](#) programı çok basit bir çizim mantığı yürütür:

1. Arka planı düz bir renkle doldurun.
2. Dolu bir daire çizin.



Oluşturma hedefi bir pencere olduğu için (bir bitmap veya başka bir ekran dışı yüzeyin aksine), çizim **WM_PAINT** mesajlarına yanıt olarak yapılır. Aşağıdaki kod Circle programı için pencere prosedürünü göstermektedir.

C++

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

        // Gösterilmeyen diğer mesajlar...
    }
    return DefWindowProc(m_hwnd, uMsg, wParam, lParam);
}
```

İşte daireyi çizen kod.

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);
    }
}

```

ID2D1RenderTarget arayüzü tüm çizim işlemleri için kullanılır. Programın yöntemi **OnPaint** aşağıdakileri yapar:

1. **ID2D1RenderTarget::BeginDraw** yöntemi çizimin başladığını bildirir.
2. **ID2D1RenderTarget::Clear** yöntemi, render hedefinin tamamını düz bir renkle doldurur. Renk bir **D2D1_COLOR_F** yapısı olarak verilir. Yapıyı başlatmak için **D2D1::ColorF** sınıfını kullanabilirsiniz. Daha fazla bilgi için [Direct2D'de Renk Kullanımı bölümüne](#) bakın.
3. **ID2D1RenderTarget::FillEllipse** yöntemi, dolgu için belirtilen fırçayı kullanarak içi dolu bir elips çizer. Elips, bir merkez noktası ve x- ve y- yarıçaplarıyla belirtilir. Eğer x ve y yarıçapları aynıysa, sonuç bir daire olur.
4. **ID2D1RenderTarget::EndDraw** yöntemi, bu çerçeve için çizimin tamamlandığını bildirir. Tüm çizim işlemleri **BeginDraw** ve **EndDraw** çağrıları arasına yerleştirilmelidir.

BeginDraw, **Clear** ve **FillEllipse** yöntemlerinin tümü **void** dönüş türüne sahiptir. Bu yöntemlerden herhangi birinin yürütülmesi sırasında bir hata oluşursa, hata **EndDraw** yönteminin dönüş değeri. Yöntem **CreateGraphicsResources** gösterilmiştir [Direct2D Kaynakları Oluşturma](#) konusunu inceleyin. Bu yöntem, render hedefini ve düz renkli fırçayı oluşturur.

Aygıt çizim komutlarını arabelleğe alabilir ve **EndDraw** çağrılana kadar bunları yürütmeyi erteleyebilir. **ID2D1RenderTarget::Flush** çağrısını yaparak cihazı bekleyen çizim komutlarını yürütmeye zorlayabilirsiniz. Ancak yıkama işlemi performansı düşürebilir.

Cihaz Kaybının Ele Alınması

Programınız çalışırken, kullandığınız grafik aygıtı kullanılamaz hale gelebilir. Örneğin, ekran çözünürlüğü değişirse veya kullanıcı ekran bağdaştırıcısını kaldırırsa aygıt kaybolabilir. Aygıt kaybolursa, aygıtla ilişkilendirilmiş tüm aygıtla bağlı kaynaklarla birlikte render hedefi de geçersiz hale gelir. Direct2D, **EndDraw** yönteminden **D2DERR_RECREATE_TARGET** hata kodunu döndürerek aygıtın kaybolduğunu bildirir. Bu hata kodunu alırsanız, render hedefini ve cihaza bağlı tüm kaynakları yeniden oluşturmanız gerekir.

Bir kaynağı atmak için, o kaynağın arayüzünü serbest bırakmanız yeterlidir.

C++

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

Kaynak oluşturmak pahalı bir işlem olabilir, bu nedenle her **WM_PAINT** mesajı için kaynaklarınızı yeniden oluşturmayın. Bir kaynağı bir kez oluşturun ve kaynak aygıt kaybı nedeniyle geçersiz hale gelene kadar veya artık o kaynağa ihtiyacınız kalmayana kadar kaynak işaretçisini önbelleğe alın.

Direct2D Render Döngüsü

Ne çizdiğinizden bağımsız olarak, programınız aşağıdakine benzer bir döngü gerçekleştirmelidir.

1. Cihazdan bağımsız kaynaklar oluşturun.
2. Sahneyi oluşturun.
 - a. Geçerli bir render hedefinin var olup olmadığını kontrol edin. Değilse, render hedefini ve cihaza bağlı kaynakları oluşturun.
 - b. **ID2D1RenderTarget::BeginDraw** ögesini çağırın.
 - c. Çizim komutlarını yayınlayın.
 - d. **ID2D1RenderTarget::EndDraw** ögesini çağırın.
 - e. **EndDraw**, **D2DERR_RECREATE_TARGET** döndürürse, oluşturma hedefini ve cihaza bağlı kaynakları atın.
3. Sahneyi güncellemeniz veya yeniden çizmeniz gerektiğinde 2. adımı tekrarlayın.

Eğer render hedefi bir pencere ise, pencere bir render hedefi aldığı anda 2. adım gerçekleşir.

WM_PAINT mesaji.

Burada gösterilen döngü, cihaza bağlı kaynakları atarak ve bir sonraki döngünün başlangıcında (adım 2a) yeniden oluşturarak cihaz kaybını ele alır.

Sonraki

[DPI ve Cihazdan Bağımsız Pikseller](#)

DPI ve cihazdan bağımsız pikseller

Makale - 05/27/2022 - Okumak için 8 dakika

Windows grafikleriyle etkili bir şekilde programlama yapabilmek için birbiriyle ilişkili iki kavramı anlamamız gerekir:

- İnç başına nokta sayısı (DPI)
- Cihazdan bağımsız piksel (DIP'ler).

DPI ile başlayalım. Bunun için tipografiye kısa bir giriş yapmamız gerekecek. Tipografide yazı boyutu *nokta* adı verilen birimlerle ölçülür. Bir nokta bir inçin 1/72'sine eşittir.

1 pt = 1/72 inç

7 Not

Bu, noktanın masaüstü yayıncılıktaki tanımıdır. Tarihsel olarak, bir noktanın tam ölçüsü değişmiştir.

Örneğin, 12 puntoluk bir yazı tipi 1/6" (12/72) metin satırına sığacak şekilde tasarlanmıştır. Elbette bu, yazı tipindeki her karakterin tam olarak 1/6" boyunda olduğu anlamına gelmez. Aslında, bazı karakterler 1/6 inçten daha uzun olabilir. Örneğin, birçok yazı tipinde Å karakteri yazı tipinin nominal yüksekliğinden daha uzundur. Yazı tipinin doğru görüntülenebilmesi için metin arasında bir miktar ek boşluğa ihtiyacı vardır. Bu boşluğa satır *başı* denir.

Aşağıdaki resimde 72 puntoluk bir yazı tipi gösterilmektedir. Düz çizgiler metnin etrafındaki 1 inç yüksekliğindeki sınırlayıcı kutuyu göstermektedir. Kesikli çizgi *taban* çizgisi olarak adlandırılır. Bir yazı tipindeki karakterlerin çoğu taban çizgisine dayanır. Yazı tipinin yüksekliği, taban çizgisinin üstündeki kısmı (*çıkış*) ve taban çizgisinin altındaki kısmı (*iniş*) içerir. Burada gösterilen yazı tipinde, çıkış 56 nokta ve iniş 16 noktadır.

Ancak söz konusu bilgisayar ekranı olduğunda, metin boyutunu ölçmek sorunludur çünkü piksellerin hepsi aynı boyutta değildir. Bir pikselin boyutu iki faktöre bağlıdır: ekran çözünürlüğü ve monitörün fiziksel boyutu. Bu nedenle, fiziksel inçler kullanışlı bir ölçü değildir, çünkü fiziksel inçler ve pikseller arasında sabit bir ilişki yoktur. Bunun

yerine, yazı tipleri *mantıksal* birimlerle ölçülür. 72 noktalı bir yazı tipi bir olarak tanımlanır

mantıksal inç uzunluğundadır. Mantıksal inçler daha sonra piksellere dönüştürülür. Windows uzun yıllar boyunca aşağıdaki dönüşümü kullanmıştır: Bir mantıksal inç 96 piksele eşittir. Bu ölçekleme faktörü kullanıldığında, 72 puntoluk bir font 96 piksel boyunda gösterilir. 12 puntoluk bir yazı tipi 16 piksel uzunluğundadır.

$12 \text{ nokta} = 12/72 \text{ mantıksal inç} = 1/6 \text{ mantıksal inç} = 96/6 \text{ piksel} = 16 \text{ piksel}$

Bu ölçeklendirme faktörü inç başına 96 nokta (DPI) olarak tanımlanır. Nokta terimi, fiziksel mürekkep noktalarının kağıda döküldüğü baskıdan gelmektedir. Bilgisayar ekranları için, mantıksal inç başına 96 piksel demek daha doğru olacaktır, ancak DPI terimi yapışmıştır.

Gerçek piksel boyutları farklılık gösterdiğinden, bir monitörde okunabilen metin başka bir monitörde çok küçük olabilir. Ayrıca, insanların farklı tercihleri vardır - bazı insanlar daha büyük metinleri tercih eder. Bu nedenle, Windows kullanıcının DPI ayarını değiştirmesine olanak tanır. Örneğin, kullanıcı ekranı 144 DPI olarak ayarlarsa, 72 noktalı bir yazı tipi 144 piksel uzunluğunda olur. Standart DPI ayarları %100 (96 DPI), %125 (120 DPI) ve %150'dir (144 DPI). Kullanıcı ayrıca özel bir ayar da uygulayabilir. Windows 7'den itibaren, DPI kullanıcı başına bir ayardır.

DWM ölçeklendirme

Bir program DPI'ı hesaba katmazsa, yüksek DPI ayarlarında aşağıdaki kusurlar görülebilir:

- Kırılmış kullanıcı
- arayüzü öğeleri. Yanlış düzen.
- Pikseli bitmapler ve simgeler.
- İsabet testi, sürükle ve bırak ve benzerlerini etkileyebilecek yanlış fare koordinatları.

Eski programların yüksek DPI ayarlarında çalışmasını sağlamak için DWM kullanışlı bir geri dönüş uygular. Bir program DPI farkında olarak işaretlenmemişse, DWM tüm kullanıcı arayüzünü DPI ayarına uyacak şekilde ölçeklendirir. Örneğin, 144 DPI'da kullanıcı arayüzü metin, grafikler, kontroller ve pencere boyutları dahil olmak üzere %150 oranında ölçeklendirilir. Program 500 × 500 pencere oluşturursa, pencere aslında 750 × 750 piksel olarak görünür ve pencerenin içeriği buna göre ölçeklendirilir.

Bu davranış, eski programların yüksek DPI ayarlarında "sadece çalıştığı" anlamına gelir. Bununla birlikte, ölçeklendirme pencere çizildikten sonra uygulandığından, ölçeklendirme biraz bulanık bir görünüme de neden olur.

DPI farkındalığı olan uygulamalar

DWM ölçeklendirmesinden kaçınmak için, bir program kendisini DPI farkında olarak işaretleyebilir. Bu, DWM'ye herhangi bir otomatik DPI ölçeklendirmesi yapmamasını söyler. Tüm yeni uygulamalar şu şekilde tasarlanmalıdır

DPI farkındalığı, daha yüksek DPI ayarlarında kullanıcı arayüzünün görünümünü iyileştirir.

Bir program, uygulama bildirim aracılığıyla kendisini DPI-aware olarak beyan eder.

Bildirim, bir DLL veya uygulamayı tanımlayan basit bir XML dosyasıdır. Bildirim genellikle yürütülebilir dosyaya gömülüdür, ancak ayrı bir dosya olarak da sağlanabilir. Bildirim DLL bağımlılıkları, istenen ayrıcalık düzeyi ve programın hangi Windows sürümü için tasarlandığı gibi bilgileri içerir.

Programınızın DPI-aware olduğunu bildirmek için manifestoya aşağıdaki bilgileri ekleyin.

sözdizimi

```
<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>
```

Burada gösterilen liste yalnızca kısmi bir bildirimdir, ancak Visual Studio bağlayıcısı bildirimin geri kalanını sizin için otomatik olarak oluşturur. Projenize kısmi bir bildirim eklemek için Visual Studio 'da aşağıdaki adımları gerçekleştirin.

1. **Proje** menüsünde, **Özellik** ögesine tıklayın.
2. Sol bölmede **Yapılandırma Özellikleri**'ni genişletin, **Manifesto Aracı**'ni genişletin ve ardından **Giriş ve Çıkış**'ı tıklatın.
3. **Ek Manifesto Dosyaları** metin kutusuna manifesto dosyasının adını yazın ve ardından **Tamam**'ı tıklatın.

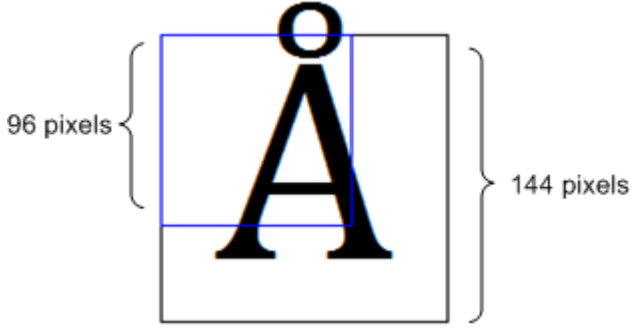
Programınızı DPI-aware olarak işaretleyerek, DWM'ye uygulama pencerenizi ölçeklendirmemesini söylemiş olursunuz. Şimdi 500 × 500 boyutunda bir pencere oluşturursanız, pencere 500

× 500 piksel, kullanıcının DPI ayarından bağımsız olarak.

GDI ve DPI

GDI çizimi piksel cinsinden ölçülür. Bu, programınız DPI-aware olarak işaretlenmişse ve GDI'dan 200 × 100 dikdörtgen çizmesini isterseniz, ortaya çıkan dikdörtgenin ekranda 200 piksel genişliğinde ve 100 piksel yüksekliğinde olacağı anlamına gelir. Ancak, GDI yazı tipi boyutları geçerli DPI ayarına göre ölçeklendirilir. Başka bir deyişle, 72 noktalı bir yazı tipi oluşturursanız, yazı tipinin boyutu 96

96 DPI'da piksel, ancak 144 DPI'da 144 piksel. İşte GDI kullanılarak 144 DPI'da işlenen 72 puntoluk bir yazı tipi.



Uygulamanız DPI farkındaysa ve çizim için GDI kullanıyorsanız, tüm çizim koordinatlarınızı DPI ile eşleştirecek şekilde ölçeklendirin.

Direct2D ve DPI

Direct2D, DPI ayarına uyacak şekilde ölçeklendirmeyi otomatik olarak gerçekleştirir. Direct2D'de koordinatlar *aygıttan bağımsız piksel* (DIP) adı verilen birimlerle ölçülür. Bir DIP, *mantıksal* bir inçin 1/96'sı olarak tanımlanır. Direct2D'de tüm çizim işlemleri DIP cinsinden belirtilir ve ardından geçerli DPI ayarına göre ölçeklendirilir.

DPI ayarı	DIP boyutu
	961 piksel
120	1,25 piksel
144	1,5 piksel

Örneğin, kullanıcının DPI ayarı 144 DPI ise ve Direct2D'den 200 × 100 dikdörtgen çizmesini isterseniz, dikdörtgen 300 × 150 fiziksel piksel olacaktır. Ayrıca, DirectWrite yazı tipi boyutlarını nokta yerine DIP cinsinden ölçer. 12 noktalı bir yazı tipi oluşturmak için 16 DIP belirtin (12 nokta = 1/6 mantıksal inç = 96/6 DIP). Metin ekrana çizildiğinde, Direct2D DIP'leri fiziksel piksellere dönüştürür. Bu sistemin yararı, geçerli DPI ayarından bağımsız olarak, ölçü birimlerinin hem metin hem de çizim için tutarlı olmasıdır.

Bir uyarı: Fare ve pencere koordinatları hala fiziksel piksel cinsinden verilir, DIP cinsinden değil. Örneğin, [WM_LBUTTONDOWN](#) mesajını işlerseniz, fare aşağı konumu fiziksel piksel cinsinden verilir. Bu konumda bir nokta çizmek için piksel koordinatlarını DIP'lere dönüştürmeniz gerekir.

Fiziksel pikselleri DIP'lere dönüştürme

Fiziksel piksellerden DIP'lere dönüşüm için aşağıdaki formül kullanılır.

$$\text{DIPs} = \text{pikseller} / (\text{DPI}/96.0)$$

DPI ayarını almak için [GetDpiForWindow](#) işlevini çağırın. DPI kayan noktalı bir değer olarak döndürülür. Her iki eksen için ölçekleme faktörünü hesaplayın.

C++

```
float g_DPIScale = 1.0f;

void InitializeDPIScale(HWND hwnd)
{
    float dpi = GetDpiForWindow(hwnd); g_DPIScale
    = dpi/96.0f;
}

şablon <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;
}
```

Direct2D kullanmıyorsanız DPI ayarını elde etmenin alternatif bir yolunu burada bulabilirsiniz:

C++

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

7 Not

Bir masaüstü uygulaması için [GetDpiForWindow](#); Evrensel Windows Platformu (UWP) uygulaması için ise [DisplayInformation::LogicalDpi](#) kullanmanızı öneririz. Tavsiye etmememize rağmen, varsayılan DPI farkındalığını ayarlamak mümkündür

SetProcessDpiAwarenessContext kullanarak programlı olarak değiştirebilirsiniz. İşleminizde bir pencere (bir HWND) oluşturulduktan sonra, DPI farkındalık modunun değiştirilmesi artık desteklenmemektedir. Süreç varsayılan DPI farkındalık modunu programlı olarak ayarlıyorsanız, herhangi bir HWND oluşturulmadan önce ilgili API'yi çağırmanız gerekir. Daha fazla bilgi için [Bir işlem için varsayılan DPI farkındalığını ayarlama bölümüne](#) bakın.

Oluşturma hedefini yeniden boyutlandırma

Pencerenin boyutu değişirse, render hedefini eşleştirecek şekilde yeniden boyutlandırmanız gerekir. Çoğu durumda, düzeni güncellemeniz ve pencereyi yeniden boyamanız da gerekecektir. Aşağıdaki kod bu adımları göstermektedir.

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);
    }
}
```

GetClientRect işlevi, istemci alanının yeni boyutunu fiziksel piksel cinsinden (DIP değil) alır. **ID2D1HwndRenderTarget::Resize** yöntemi, yine piksel cinsinden belirtilen render hedefinin boyutunu günceller. **InvalidateRect** işlevi, tüm istemci alanını pencerenin güncelleme bölgesine ekleyerek yeniden boyamayı zorlar. (Modül 1'deki [Pencerenin Boyanması bölümüne](#) bakın).

Pencere büyüdükçe veya küçüldükçe, genellikle çizdiğiniz nesnelerin konumunu yeniden hesaplamamız gerekecektir. Örneğin, daire programında yarıçap ve merkez noktası güncellenmelidir:

C++

```
void MainWindow::CalculateLayout()
{
    if (pRenderTarget != NULL)
    {
        D2D1_SIZE_F size = pRenderTarget->GetSize(); const
```

ID2D1RenderTarget::GetSize yöntemi, render hedefinin boyutunu DIP (piksel değil) cinsinden döndürür; bu, yerleşimi hesaplamak için uygun birimdir. Fiziksel piksel cinsinden boyut döndüren **ID2D1RenderTarget::GetPixelSize** adlı yakından ilişkili bir yöntem vardır. Bir **HWND** render hedefi için bu değer **GetClientRect** tarafından döndürülen boyutla eşleşir. Ancak çizimin piksellerle değil DIP'lerle yapıldığını unutmayın.

Sonraki









[Direct2D'de renk kullanımı](#)

Direct2D'de Renk Kullanımı

Makale - 08/19/2020 - 4 dakika okumak için

Direct2D, renklerin farklı kırmızı, yeşil ve mavi değerlerinin bir araya getirilmesiyle oluşturulduğu RGB renk modelini kullanır. Dördüncü bir bileşen olan alpha, bir pikselin saydamlığını ölçer. Direct2D'de bu bileşenlerin her biri [0.0 1.0] aralığında kayan noktalı bir değerdir. Üç renk bileşeni için değer, rengin yoğunluğunu ölçer. Alfa bileşeni için, 0.0 tamamen saydam ve 1.0 tamamen opak anlamına gelir. Aşağıdaki tabloda %100 yoğunluğun çeşitli kombinasyonlarından kaynaklanan renkler gösterilmektedir.

Kırmızı	Yeşil	Mavi	Renk
0	0	0	Siyah
1	0	0	Kırmızı
0	1	0	Yeşil
0	0	1	Mavi
0	1	1	Cyan
1	0	1	Magenta
1	1	0	Sarı
1	1	1	Beyaz

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

0 ile 1 arasındaki renk değerleri, bu saf renklerin farklı tonlarıyla sonuçlanır. Direct2D, renkleri temsil etmek için **D2D1_COLOR_F** yapısını kullanır. Örneğin, aşağıdaki kod macentayı belirtir.

C++

```
// Macenta rengini başlatın.
```

```
D2D1_COLOR_F clr;  
clr.r = 1;  
clr.g = 0;
```

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

D2D1::ColorF sınıfını kullanarak da bir renk belirtebilirsiniz.

D2D1_COLOR_F yapısı.

C++

```
// Önceki örneğe eşdeğerdir.
```

```
D2D1::ColorF clr(1, 0, 1, 1);
```

Alfa Karıştırma

Alfa karıştırma, aşağıdaki formülü kullanarak ön plan rengini arka plan rengiyle karıştırarak yarı saydam alanlar oluşturur.

$$\text{renk} = af * C_f + (1 - af) * C_b$$

Burada C_b arka plan rengi, C_f ön plan rengi ve af ön plan renginin alfa değeridir. Bu formül her renk bileşenine çift olarak uygulanır. Örneğin, ön plan renginin (**R = 1.0, G = 0.4, B = 0.0**), alfa = 0.6 ve arka plan renginin (**R = 0.0, G = 0.5, B = 1.0**) olduğunu varsayalım. Elde edilen alfa karışımı renk şöyledir:

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

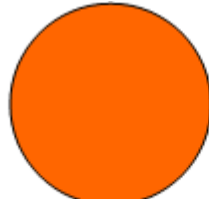
$$G = (0.4 * 0.6 + 0.5 * 0.4) = .44$$

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

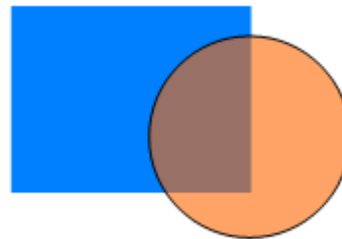
Aşağıdaki görüntü bu karıştırma işleminin sonucunu göstermektedir.



(0, 0.5, 1.0)



(1.0, 0.4, 0.0)
 $\alpha = 0.6$



Piksel Biçimleri

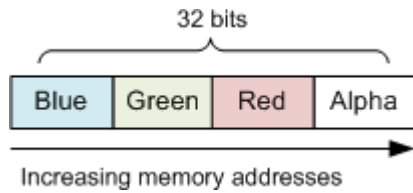
D2D1_COLOR_F yapısı bir pikselin bellekte nasıl temsil edildiğini tanımlamaz. Çoğu durumda, bu önemli değildir. Direct2D, renk bilgisini piksellere çevirmenin tüm dahili ayrıntılarını ele alır. Ancak aşağıdaki durumlarda piksel biçimini bilmeniz gerekebilir

doğrudan bellekteki bir bitmap ile çalışıyorsanız veya Direct2D'yi Direct3D veya GDI ile birleştiriyorsanız.

DXGI_FORMAT numaralandırması piksel formatlarının bir listesini tanımlar. Liste oldukça uzundur, ancak bunlardan yalnızca birkaçı Direct2D ile ilgilidir. (Diğerleri Direct3D tarafından kullanılır).

Piksel biçimi	Açıklama
DXGI_FORMAT_B8G8R8A8_UNORM	Bu en yaygın piksel formatıdır. Tüm piksel bileşenler (kırmızı, yeşil, mavi ve alfa) 8 bitlik işaretli tamsayılardır. Bileşenler bellekte <i>BGRA</i> sırasına göre düzenlenir. (Aşağıdaki resme bakın.)
DXGI_FORMAT_R8G8B8A8_UNORM tamsayıdır	Piksel bileşenleri <i>RGBA</i> cinsinden 8 bitlik işaretli düzenine göre değiştirilir. Başka bir deyişle, kırmızı ve mavi bileşenler DXGI_FORMAT_B8G8R8A8_UNORM 'a göre yer değiştirir. Bu biçim yalnızca donanım aygıtları için desteklenir.
DXGI_FORMAT_A8_UNORM	Bu format, 8 bitlik bir alfa bileşeni içerir ve RGB bileşenleri. Opaklık maskeleri oluşturmak için kullanışlıdır. Direct2D'de opaklık maskeleri kullanma hakkında daha fazla bilgi edinmek için Uyumlu A8 Render Hedeflerine Genel Bakış bölümüne bakın.

Aşağıdaki çizimde BGRA piksel düzeni gösterilmektedir.



Bir **render** hedefinin piksel biçimini almak için **ID2D1RenderTarget::GetPixelFormat** ögesini çağırın. Piksel biçimi ekran çözünürlüğüyle eşleşmeyebilir. Örneğin, render hedefi 32 bit renk kullanmasına rağmen ekran 16 bit renge ayarlanmış olabilir.

Alfa Modu

Bir render hedefinin, alfa değerlerinin nasıl ele alınacağını tanımlayan bir alfa modu da vardır.

Alfa modu	Açıklama
D2D1_ALPHA_MODE_IGNORE	Alfa harmanlama gerçekleştirilmez. Alfa değerleri yok sayılır.

D2D1_ALPHA_MODE_STRAIGHT

Düz alfa. Pikselin renk bileşenleri
alfa harmanlamadan önceki renk yoğunluğunu temsil eder.

Alfa modu	Açıklama
D2D1_ALPHA_MODE_PREMULTIPLIED	Önceden çarpılmış alfa. Pikselin renk bileşenleri renk yoğunluğunun alfa değeriyle çarpımını temsil eder. Bu format, alfa harmanlama formülündeki terim (af Cf) önceden hesaplandığı için düz alfaya göre daha verimli işlenir. Ancak bu format bir görüntü dosyasında saklamak için uygun değildir.

İşte düz alfa ile önceden çarpılmış alfa arasındaki farka bir örnek. İstenen rengin %50 alfa ile saf kırmızı (%100 yoğunluk) olduğunu varsayalım. Direct2D türü olarak bu renk (1, 0, 0, 0,5) şeklinde gösterilir. Düz alfa kullanıldığında ve 8 bit renk bileşenleri varsayıldığında, pikselin kırmızı bileşeni 0xFF olur. Önceden çarpılmış alfa kullanıldığında, kırmızı bileşen 0x80'e eşit olacak şekilde %50 ölçeklenir.

D2D1_COLOR_F veri tipi renkleri her zaman düz alfa kullanarak temsil eder. Direct2D, gerekirse pikselleri önceden çarpılmış alfa biçimine dönüştürür.

Programınızın herhangi bir alfa harmanlaması yapmayacağını biliyorsanız, render hedefini **D2D1_ALPHA_MODE_IGNORE** alfa moduyla oluşturun. Bu mod performansı artırabilir, çünkü Direct2D alfa hesaplamalarını atlayabilir. Daha fazla bilgi için [Direct2D Uygulamalarının Performansını İyileştirme](#) bölümüne bakın.

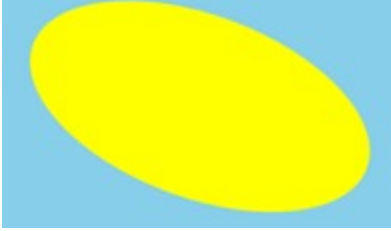
Sonraki

[Direct2D'de Dönüşümleri Uygulama](#)

Direct2D'de Dönüşümleri Uygulama

Makale - 08/19/2020 - Okumak için 3 dakika

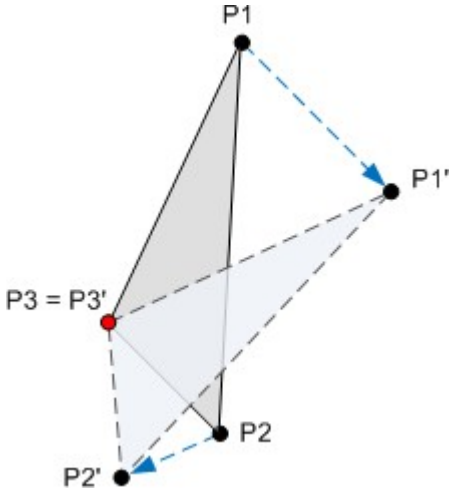
Direct2D ile Çizim bölümünde, [ID2D1RenderTarget::FillEllipse](#) yönteminin x ve y eksenlerine hizalanmış bir elips çizdiğini gördük. Ancak bir açıyla eğilmiş bir elips çizmek istediğinizi varsayalım.



Dönüşümleri kullanarak bir şekli aşağıdaki şekillerde değiştirebilirsiniz.

- Bir nokta etrafında
- döndürme. Ölçekleme.
- Öteleme (X veya Y yönünde yer değiştirme).
- Eğiklik (*kayma* olarak da bilinir).

Dönüşüm, bir nokta kümesini yeni bir nokta kümesine eşleyen matematiksel bir işlemdir. Örneğin, aşağıdaki diyagram P3 noktası etrafında döndürülmüş bir üçgeni göstermektedir. Döndürme uygulandıktan sonra, P1 noktası P1' ile, P2 noktası P2' ile ve P3 noktası da kendisiyle eşlenir.



Dönüşümler matrisler kullanılarak gerçekleştirilir. Ancak, bunları kullanmak için matrislerin matematiğini anlamak zorunda değilsiniz. Matematik hakkında daha fazla bilgi edinmek istiyorsanız, bkz: [Matris Dönüşümleri](#).

Direct2D'de bir dönüşüm uygulamak için [ID2D1RenderTarget::SetTransform](#) yöntemini çağırın. Bu yöntem, dönüşümü tanımlayan bir [D2D1_MATRIX_3X2_F](#) yapısı alır. Sen

D2D1::Matrix3x2F sınıfındaki yöntemleri çağırarak bu yapıyı başlatabilir. Bu sınıf, her bir dönüşüm türü için bir matris döndüren statik yöntemler içerir:

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

Örneğin, aşağıdaki kod (100, 100) noktası etrafında 20 derecelik bir döndürme uygular.

C++

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

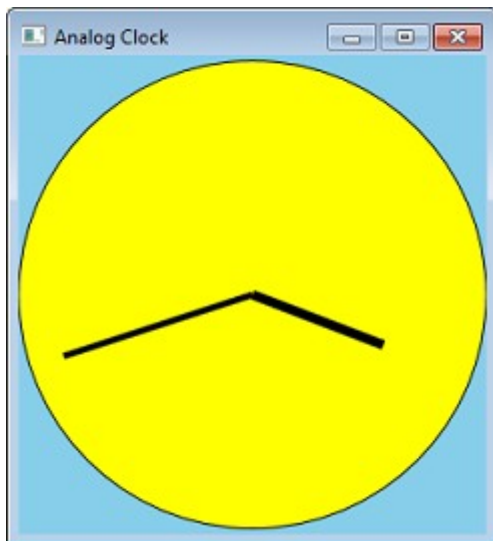
Dönüşüm, siz **SetTransform** ögesini tekrar çağırana kadar sonraki tüm çizim işlemlerine uygulanır. Geçerli dönüşümü kaldırmak için **SetTransform** ögesini kimlik matrisiyle birlikte çağırın. Özdeşlik matrisini oluşturmak için **Matrix3x2F::Identity** fonksiyonunu çağırın.

C++

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

Saat İbresi Çizimi

Daire programımızı analog bir saate dönüştürerek dönüşümleri kullanalım. Bunu akrep ve yelkovan için çizgiler ekleyerek yapabiliriz.



Çizgiler için koordinatları hesaplamak yerine, açıyı hesaplayabilir ve ardından bir döndürme dönüşümü uygulayabiliriz. Aşağıdaki kodda bir saat çizen bir fonksiyon

gösterilmektedir

el. *fAngle* parametresi elin açısını derece cinsinden verir.

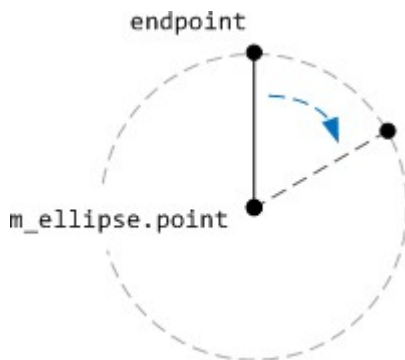
C++

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

    // endPoint elin bir ucunu tanımlar. D2D_POINT_2F
    endPoint = D2D1::Point2F(
        m_ellipse.point.x,
        m_ellipse.point.y - (m_ellipse.radiusY * fHandLength)
    );

    // Elipsin merkezinden bitiş noktasına bir çizgi çizin.
    m_pRenderTarget->DrawLine(
        m_ellipse.point, endPoint, m_pStroke, fStrokeWidth);
}
```

Bu kod, saat yüzünün merkezinden başlayıp *endPoint* noktasında biten dikey bir çizgi çizer. Çizgi, bir döndürme dönüşümü uygulanarak elipsin merkezi etrafında döndürülür. Döndürme için merkez noktası, saat kadranını oluşturan elipsin merkezidir.



Aşağıdaki kod tüm saat yüzünün nasıl çizildiğini göstermektedir.

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);

    // Elleri çizin
    SİSTEM ZAMANI zaman;
    GetLocalTime(&time);
```

```

// 60 dakika = 30 derece, 1 dakika = 0,5 derece
const float fHourAngle = (360.0f / 12) * (time.wHour) + (time.wMinute *
0.5f);
const float fMinuteAngle =(360.0f / 60) * (time.wMinute);

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

// Özdeşlik dönüşümünü geri yükleyin.
m_pRenderTarget->SetTransform( D2D1::Matrix3x2F::Identity() );
}

```

Visual Studio projesinin tamamını [Direct2D Clock Sample](#) adresinden indirebilirsiniz. (Sadece eğlence için, indirme sürümü saat yüzüne radyal bir gradyan ekler).

Dönüşümleri Birleştirme

Dört temel dönüşüm, iki veya daha fazla matrisin çarpılmasıyla birleştirilebilir. Örneğin, aşağıdaki kod bir döndürme ile bir ötelemeyi birleştirir.

C++

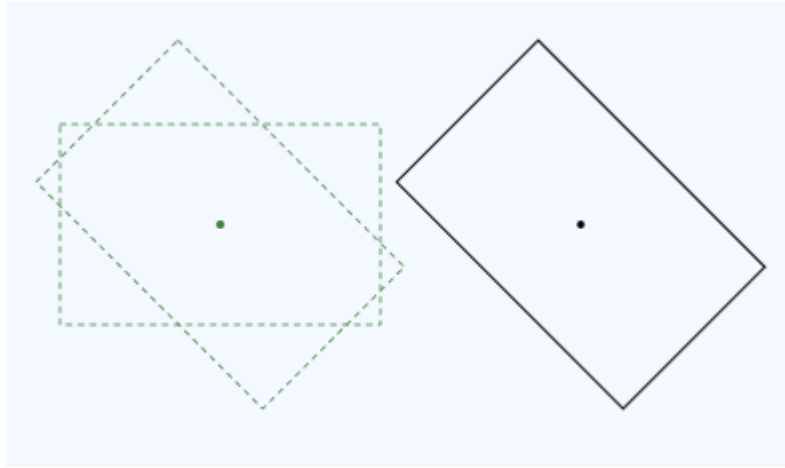
```

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

pRenderTarget->SetTransform(rot * trans);

```

Matrix3x2F sınıfı matris çarpımı için **operator*()** sağlar. Matrisleri hangi sırayla çarptığınız önemlidir. Bir dönüşüm ($M \times N$) ayarlamak "Önce M'yi, ardından N'yi uygulayın" anlamına gelir. Örneğin, burada döndürme ve ardından öteleme vardır:

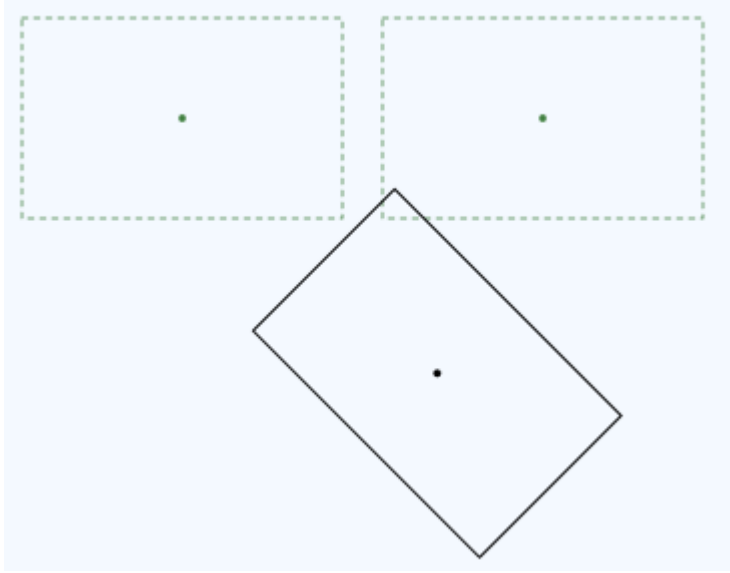


İşte bu dönüşüm için kod:

C++

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

Şimdi bu dönüşümü ters sırada bir dönüşümle karşılaştırın, öteleme ve ardından döndürme.



Döndürme işlemi orijinal dikdörtgenin merkezi etrafında gerçekleştirilir. İşte bu dönüşüm için kod.

C++

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

Gördüğünüz gibi, matrisler aynıdır, ancak işlem sırası değişmiştir. Bunun nedeni matris çarpımının değişmeli olmamasıdır: $M \times N \neq N \times M$.

Sonraki

[Ekler: Matris Dönüşümleri](#)

Ekler: Matris Dönüşümleri

Makale - 01/28/2021 - Okumak için 7 dakika

Bu konu, 2 boyutlu grafikler için matris dönüşümlerine matematiksel bir genel bakış sağlar. Ancak, Direct2D'de dönüşümleri kullanmak için matris matematiğini bilmenize gerek yoktur. Matematikle ilgileniyorsanız bu konuyu okuyun; aksi takdirde, bu konuyu atlamaktan çekinmeyin.

- ♦ [Matrislere Giriş Matris](#)
 - [İşlemleri](#)
- ♦ [Afin Dönüşümler Çeviri](#)
 - [Dönüşümü](#)
 - [Ölçekleme Dönüşümü](#)
 - [Başlangıç Noktası Etrafında Dönme](#)
 - [Keyfi Bir Nokta Etrafında Döndürme](#)
 - [Çarpık Dönüşüm](#)
- ♦ [Direct2D'de Dönüşümleri Temsil](#)
- ♦ [Etme Sonraki](#)

Matrislere Giriş

Bir matris, gerçek sayılardan oluşan dikdörtgen bir dizidir. Matrisin *sırası* satır ve sütun sayısıdır. Örneğin, matrisin 3 satırı ve 2 sütunu varsa, sıra $3 \times$

2. Matrisler genellikle köşeli parantez içine alınmış matris elemanları ile gösterilir:

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

Gösterim: Bir matris büyük harfle gösterilir. Elemanlar küçük harflerle gösterilir. Alt simgeler bir elemanın satır ve sütun numarasını gösterir. Örneğin, a_{ij} , A matrisinin i'inci satır ve j'inci sütunundaki elemandır.

Aşağıdaki diyagram, matrisin her bir hücreindeki ayrı elemanlarla birlikte bir $i \times j$ matrisini göstermektedir.

	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

Matris İşlemleri

Bu bölümde matrisler üzerinde tanımlanan temel işlemler açıklanmaktadır.

Toplama işlemi. İki matrisin $A + B$ toplamı, A ve B 'nin karşılık gelen elemanlarının toplanmasıyla elde edilir:

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

Skaler çarpma. Bu işlem bir matrisi bir reel sayı ile çarpar. Bir k reel sayısı verildiğinde, kA skaler çarpımı A 'nın her elemanının k ile çarpılmasıyla elde edilir. $kA = k[a_{ij}] = [k \times a_{ij}]$

Matris çarpımı. Sırası $(m \times n)$ ve $(n \times p)$ olan iki A ve B matrisi verildiğinde, $C = A \times B$ çarpımı aşağıdaki gibi tanımlanan $(m \times p)$ sıralı bir matristir:

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

ya da eşdeğer olarak:

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

Yani, her bir c_{ij} elemanını hesaplamak için aşağıdakileri yapın:

1. A 'nın i 'inci satırını ve B 'nin j 'inci sütununu alın.
2. Satır ve sütundaki her bir öğe çiftini çarpın: ilk satır girişini ilk sütun girişiyle, ikinci satır girişini ikinci sütun girişiyle ve bu şekilde devam edin.
3. Sonucu topla.

Burada bir (2×2) matrisin bir (2×3) matris ile çarpılmasına bir örnek verilmiştir.

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

Matris çarpımı değişmeli değildir. Yani, $A \times B \neq B \times A$. Ayrıca, tanımdan her matris çiftinin çarpılamayacağı sonucu çıkar. Sol taraftaki matrisin sütun sayısı sağ taraftaki matrisin satır sayısına eşit olmalıdır. Aksi takdirde \times operatörü tanımlanmaz.

Kimlik matrisi. I olarak adlandırılan bir kimlik matrisi, aşağıdaki gibi tanımlanan bir kare matristir: $I_{ij}^* = \delta_{ij}^* = \delta_{ji}^*$ ise 1, aksi takdirde 0'dır.

Başka bir deyişle, bir özdeşlik matrisi, satır numarasının sütun numarasına eşit olduğu her eleman için 1 ve diğer tüm elemanlar için sıfır içerir. Örneğin, burada 3×3 özdeşlik matrisi verilmiştir.

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

Aşağıdaki eşitlikler herhangi bir M matrisi için geçerlidir. $M \times I = M$ $I \times M = M$

Afin Dönüşümler

Afin dönüşüm, bir koordinat uzayını diğerine eşleyen matematiksel bir işlemdir. Başka bir deyişle, bir nokta kümesini başka bir nokta kümesine eşler. Afin dönüşümler, bilgisayar grafiklerinde kullanışlı olmalarını sağlayan bazı özelliklere sahiptir.

- Afin dönüşümler eş *doğrusallığı* korur. Üç veya daha fazla nokta bir doğru üzerinde yer alıyorsa, dönüşümden sonra da bir doğru oluştururlar. Düz çizgiler düz kalır.
- İki afin dönüşümün bileşimi bir afin dönüşümdür.

2-B uzay için afin dönüşümler aşağıdaki forma sahiptir.

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

Daha önce verilen matris çarpımı tanımını uygularsanız, iki afin dönüşümün çarpımının başka bir afin dönüşüm olduğunu gösterebilirsiniz. Bir afin dönüşümü kullanarak 2 boyutlu bir noktayı dönüştürmek için, nokta 1×3 matris olarak temsil edilir.

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

İlk iki eleman noktanın x ve y koordinatlarını içerir. Matematiğin doğru çalışması için 1 üçüncü elemana yerleştirilir. Dönüşümü uygulamak için iki matrisi aşağıdaki gibi çarpın.

$$P' = P \times M$$

Bu, aşağıdakilere kadar genişler.

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

nerede

$$x' = ax + cy + e \quad y' = bx + dy + f$$

Dönüştürülmüş noktayı elde etmek için P' matrisinin ilk iki elemanını

$$\text{alın. } p = (x', y') = (ax + cy + e, bx + dy + f)$$

7 Not

$1 \times n$ matrisine *satır vektörü* denir. Direct2D ve Direct3D'nin her ikisi de 2D veya 3D uzaydaki noktaları temsil etmek için satır vektörleri kullanır. Bir sütun vektörü ($n \times 1$) kullanarak ve dönüşüm matrisini transpoze ederek eşdeğer bir sonuç elde edebilirsiniz. Çoğu grafik metni sütun vektör formunu kullanır. Bu konu, Direct2D ve Direct3D ile tutarlılık için satır vektör formunu sunar.

Sonraki birkaç bölüm temel dönüşümleri türetmektedir.

Çeviri Dönüşümü

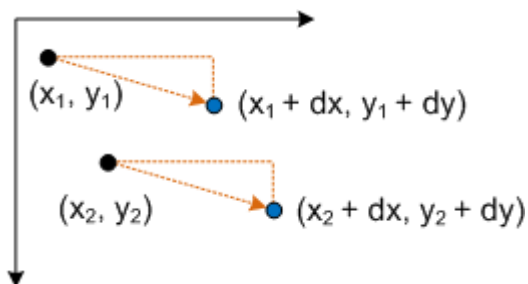
Öteleme dönüşüm matrisi aşağıdaki forma sahiptir.

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

Bir P noktasını bu denkleme yerleştirirsek elde ederiz:

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

Bu da X ekseninde dx ve Y ekseninde dy ile ötelenen (x, y) noktasına karşılık gelir.



Ölçekleme Dönüşümü

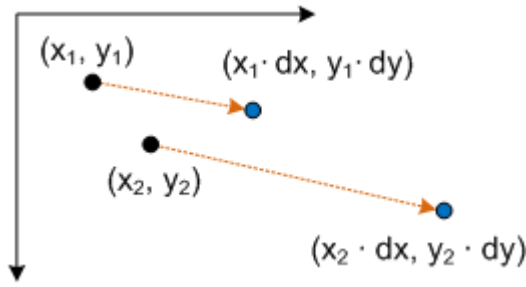
Ölçekleme dönüşüm matrisi aşağıdaki forma sahiptir.

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

Bir P noktasını bu denkleme yerleştirirsek elde ederiz:

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

dx ve dy ile ölçeklendirilmiş (x,y) noktasına karşılık gelir.



Başlangıç Noktası Etrafında Dönme

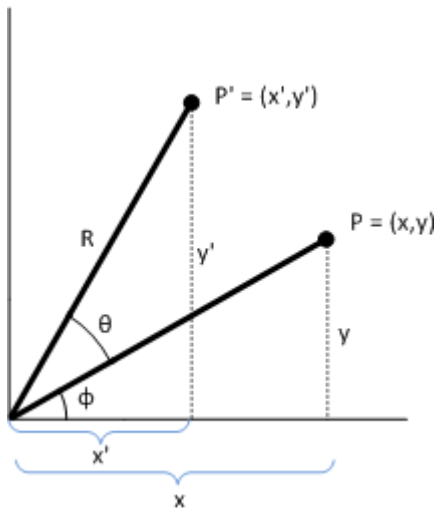
Bir noktayı orijin etrafında döndürmek için kullanılan matris aşağıdaki forma sahiptir.

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

Dönüşen nokta şu:

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

Kanıt. P' 'nin bir dönüşü temsil ettiğini göstermek için aşağıdaki diyagramı göz önünde bulundurun.



Verildi:

$$P = (x,y)$$

Dönüştürülecek orijinal nokta.

$$\Phi$$

(0,0) doğrusunun P ile oluşturduğu açı.

$$\Theta$$

(x,y)'nin orijin etrafında döndürüleceği açı. P' =

$$(x',y')$$

Dönüştürülmüş nokta.

$$R$$

P'ye giden (0,0) doğrusunun uzunluğu. Ayrıca dönme çemberinin yarıçapı.

7 Not

Bu diyagram, geometride kullanılan ve pozitif y ekseninin yukarıyı gösterdiği standart koordinat sistemini kullanır. Direct2D, pozitif y ekseninin aşağıyı gösterdiği Windows koordinat sistemini kullanır.

X eksenini ile (0,0) doğrusu arasındaki P' açısı $\Phi + \Theta$ 'dır. Aşağıdaki özdeşlikler geçerlidir:

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

Şimdi x' ve y' değerlerini Θ cinsinden çözün. Trigonometrik toplama formülleri ile:

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

Yerine koyarsak, elde ederiz:

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

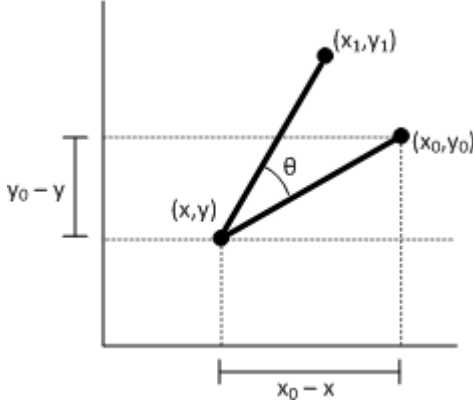
Bu da daha önce gösterilen dönüştürülmüş P' noktasına karşılık gelir.

Keyfi Bir Nokta Etrafında Dönme

Orijin dışındaki bir nokta (x,y) etrafında döndürmek için aşağıdaki matris kullanılır.

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

Bu matrisi (x,y) noktasını orijin olarak alarak türetebilirsiniz.



(x1, y1), (x0, y0) noktasının (x,y) noktası etrafında döndürülmesi sonucu elde edilen nokta olsun. x1'i aşağıdaki gibi türetebiliriz.

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

Şimdi $x1 = ax0 + cy0$ formülünü kullanarak bu denklemi dönüşüm matrisine geri ekleyin + e değerini elde edin. Aynı prosedürü y1'i türetmek için de kullanın.

Eğik Dönüşüm

Çarpıklık dönüşümü dört parametre ile tanımlanır:

- Θ : Y ekseninden bir açı olarak ölçülen x eksenine boyunca eğilme miktarı. Φ : Y eksenine boyunca eğilme miktarı, x ekseninden bir açı olarak ölçülür.
- (p_x, p_y) : Eğiltme işleminin gerçekleştirildiği noktanın x ve y koordinatları.

Çarpık dönüşüm aşağıdaki matrisi kullanır.

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

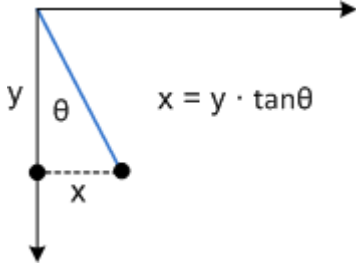
Dönüşen nokta şu:

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

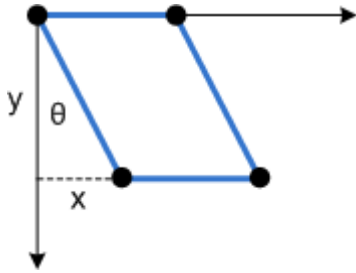
ya da eşdeğer olarak:

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

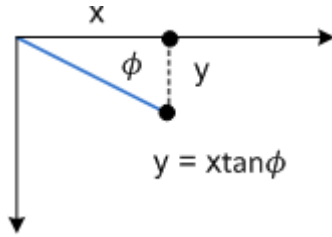
Bu dönüşümün nasıl çalıştığını görmek için her bir bileşeni ayrı ayrı ele alın. Θ parametresi x yönündeki her noktayı $\tan\Theta$ 'ya eşit bir miktarda hareket ettirir. Aşağıdaki diyagram Θ ile x eksen eğriliği arasındaki ilişkiyi göstermektedir.



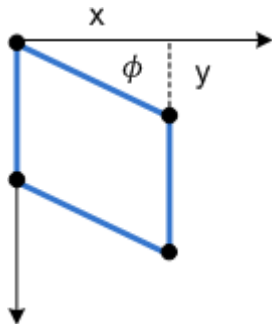
İşte bir dikdörtgene uygulanan aynı eğrilik:



Φ parametresi de aynı etkiye sahiptir, ancak y eksenini boyuncadır:



Bir sonraki diyagramda bir dikdörtgene uygulanan y eksen eğriliği gösterilmektedir.



Son olarak, p_x ve p_y parametreleri çarpıklık için merkez noktasını x ve y eksenleri boyunca kaydırır.

Direct2D'de Dönüşümleri Temsil Etme

Tüm Direct2D dönüşümleri afin dönüşümlerdir. Direct2D afin olmayan dönüşümleri desteklemez. Dönüşümler **D2D1_MATRIX_3X2_F** yapısı ile temsil edilir. Bu

yapısı 3×2 matris tanımlar. Bir afin dönüşümün üçüncü sütunu her zaman aynı olduğundan $([0, 0, 1])$ ve Direct2D afin olmayan dönüşümleri desteklemediğinden, 3×3 matrisin tamamını belirtmeye gerek yoktur. Direct2D dahili olarak 3 Dönüşümleri hesaplamak için 3×3 matrisler.

D2D1_MATRIX_3X2_F'nin üyeleri indeks konumlarına göre adlandırılır: **_11 üyesi** (1,1) elemanıdır, **_12 üyesi** (1,2) elemanıdır ve bu böyle devam eder. Yapı üyelerini doğrudan başlatabilmenize rağmen, **D2D1::Matrix3x2F** sınıfını kullanmanız önerilir. Bu sınıf **D2D1_MATRIX_3X2_F**'yi miras alır ve temel afin dönüşümlerden herhangi birini oluşturmak için yardımcı yöntemler sağlar. Sınıf ayrıca [Direct2D'de Dönüşümleri Uygulama](#) bölümünde açıklandığı gibi iki veya daha fazla dönüşüm oluşturmak için **operator* ()** tanımlar.

Sonraki

[Modül 4. Kullanıcı Girişi](#)

Modül 4. Kullanıcı Giriş

Makale - 23.08.2019 - Okumak için 2 dakika

Önceki modüllerde pencere oluşturma, pencere mesajlarını işleme ve 2D grafiklerle temel çizim konuları incelenmişti. Bu modülde, fare ve klavye girişine bakıyoruz. Bu modülün sonunda, fare ve klavyeyi kullanan basit bir çizim programı yazabileceksiniz.

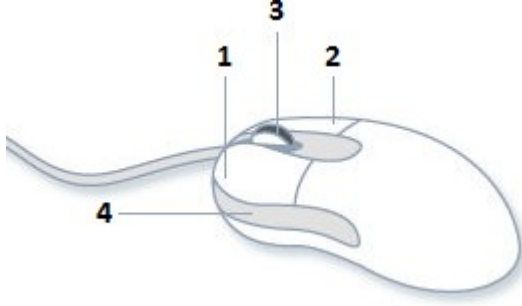
Bu bölümde

- ♦ [Fare Giriş](#)
- ♦ [Fare Tıklamalarına Yanıt Verme](#)
- ♦ [Fare Hareketi](#)
- ♦ [Çeşitli Fare İşlemleri Klavye Giriş](#)
- ♦ [Hızlandırıcı Tabloları](#)
- ♦ [İmleç Görüntüsünün Ayarlanması](#)
- ♦ [Kullanıcı Giriş: Genişletilmiş Örnek](#)

Fare Giriři (Win32 ve C++ ile Bařlarken)

Makale - 03/25/2022 - Okumak iin 2 dakika

Windows beř adede kadar dğmesi olan fareleri destekler: sol, orta ve saė, ayrıca XBUTTON1 ve XBUTTON2 olarak adlandırılan iki ek dğme.



Windows farelerinin oėunda en azından sol ve saė dğmeler bulunur. Sol fare dğmesi iřaretleme, seėme, srkleme ve benzeri iřlemler iin kullanılır. Saė fare dğmesi genellikle bir ierik mens grntler. Bazı farelerde sol ve saė dğmeler arasında bulunan bir kaydırma tekerleėi vardır. Fareye baėlı olarak, kaydırma tekerleėi de tıklanabilir olabilir, bu da onu orta dğme yapar.

XBUTTON1 ve XBUTTON2 dğmeleri genellikle farenin yanlarında, tabana yakın bir yerde bulunur. Bu ekstra dğmeler tm farelerde mevcut deėildir. Varsa, XBUTTON1 ve XBUTTON2 dğmeleri genellikle bir Web tarayıcısında ileri ve geri gezinme gibi bir uygulama iřleviyle eřleřtirilir.

Solak kullanıcılar genellikle sol ve saė dğmelerin iřlevlerini deėiřtirmeyi daha rahat bulurlar; saė dğmeyi iřareti olarak, sol dğmeyi ise ierik mensn gstermek iin kullanırlar. Bu nedenle, Windows yardım belgelerinde fiziksel yerleřimden ziyade mantıksal iřlevi ifade eden *birincil dğme* ve *ikincil dğme* terimleri kullanılır. Varsayılan (saė el) ayarda, sol dğme birincil dğme ve saė dğme ikincil dğmedir. Ancak, *saė tıklama* ve *sol tıklama* terimleri mantıksal eylemleri ifade eder. *Sol tıklama*, dğmenin fiziksel olarak farenin saė veya sol tarafında olmasına bakılmaksızın birincil dğmenin tıklanması anlamına gelir.

Kullanıcının fareyi nasıl yapılandırdıėına bakılmaksızın, Windows fare mesajlarını tutarlı olacak řekilde otomatik olarak evirir. Kullanıcı, programınızı kullanırken birincil ve ikincil dğmeleri deėiřtirebilir ve bu, programınızın nasıl davrandıėını etkilemez.

MSDN belgeleri, *birincil* ve *ikincil düğme* anlamında *sol* düğme ve *sağ düğme* terimlerini kullanır. Bu terminoloji, fare girişi için pencere mesajlarının adlarıyla tutarlıdır. Sadece fiziksel sol ve sağ düğmelerin yer değiştirebileceğini unutmayın.

Sonraki

[Fare Tıklamalarına Yanıt Verme](#)

Fare Tıklamalarına Yanıt Verme

Makale - 08/19/2020 - Okumak için 3 dakika

Kullanıcı, imleç bir pencerenin istemci alanı üzerindeyken bir fare düğmesine tıklarsa, pencere aşağıdaki mesajlardan birini alır.

Mesaj	Anlamı
WM_LBUTTONDOWN	Sol düğme aşağı
WM_LBUTTONUP	Sol düğme yukarı
WM_MBUTTONDOWN	Orta düğme aşağı
WM_MBUTTONUP	Orta düğme yukarı
WM_RBUTTONDOWN	Sağ düğme aşağı
WM_RBUTTONUP	Sağ düğme yukarı
WM_XBUTTONDOWN	XBUTTONDOWN1 veya XBUTTONDOWN2 aşağı
WM_XBUTTONUP	XBUTTONDOWN1 veya XBUTTONDOWN2 yukarı

İstemci alanının pencerenin çerçeve dışında kalan kısmı olduğunu hatırlayın. İstemci alanları hakkında daha fazla bilgi için [Pencere Nedir?](#)

Fare Koordinatları

Tüm bu mesajlarda, *lParam* parametresi fare işaretçisinin x- ve y-koordinatlarını içerir. *lParam*'ın en düşük 16 biti x-koordinatını, sonraki 16 biti ise y-koordinatını içerir. Koordinatları *lParam*'dan açmak için **GET_X_LPARAM** ve **GET_Y_LPARAM** makrolarını kullanın.

C++

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

Bu makrolar WindowsX.h başlık dosyasında tanımlanmıştır.

64 bit Windows'ta, *lParam* 64 bit değerdir. *lParam*'ın üst 32 biti kullanılmaz. MSDN belgelerinde "düşük sıralı kelime" ve "yüksek sıralı kelime"den bahsedilmektedir.

lParam. 64 bitlik durumda bu, alt 32 bitin düşük ve yüksek sıralı sözcükleri anlamına gelir. Makrolar doğru değerleri çıkarır, bu nedenle bunları kullanırsanız güvende olursunuz.

Fare koordinatları aygıttan bağımsız piksel (DIP) olarak değil piksel olarak verilir ve pencerenin istemci alanına göre ölçülür. Koordinatlar işaretli değerlerdir.

İstemci alanının üstündeki ve solundaki konumlar negatif koordinatlara sahiptir, bu da fare konumunu pencerenin dışında izliyorsanız önemlidir. Bunu nasıl yapacağımızı daha sonraki bir konu olan [Pencere Dışındaki Fare Hareketlerini Yakalama](#) bölümünde göreceğiz.

Ek Bayraklar

wParam parametresi, diğer fare düğmelerinin yanı sıra SHIFT ve CTRL tuşlarının durumunu gösteren bayrakların bitset VEYA'sını içerir.

Bayrak	Anlamı
MK_CONTROL	CTRL tuşu kapalı.
MK_LBUTTON	Sol fare düğmesi aşağıdadır.
MK_MBUTTON	Farenin orta düğmesi aşağıdadır.
MK_RBUTTON	Sağ fare düğmesi aşağıdadır.
MK_SHIFT	SHIFT tuşu aşağıdadır.
MK_XBUTTON1	XBUTTON1 düğmesi aşağıdadır.
MK_XBUTTON2	XBUTTON2 düğmesi aşağıdadır.

Bir bayrağın olmaması, ilgili düğmeye veya tuşa basılmadığı anlamına gelir. Örneğin, CTRL tuşunun basılı olup olmadığını test etmek için:

```
C++
```

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

CTRL ve SHIFT dışında diğer tuşların durumunu bulmanız gerekiyorsa, [GetKeyState](#) işlevi, [Klavye Girişi](#) bölümünde açıklanmıştır.

[WM_XBUTTONDOWN](#) ve [WM_XBUTTONUP](#) pencere mesajları hem XBUTTON1 hem de XBUTTON2 için geçerlidir. *wParam* parametresi hangi düğmeye tıklandığını gösterir.

C++

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

Çift Tıklama

Bir pencere varsayılan olarak çift tıklama bildirimleri almaz. Çift tıklamaları almak için, pencere sınıfını kaydettiğinizde **WNDCLASS** yapısında **CS_DBLCLKS** bayrağını ayarlayın.

C++

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

/* Diğer yapı üyelerini ayarlayın. */

RegisterClass(&wc);
```

CS_DBLCLKS bayrağını gösterildiği gibi ayarlarsanız, pencere çift tıklama bildirimleri alacaktır. Çift tıklama, adında "DBLCLK" bulunan bir pencere mesajıyla belirtilir. Örneğin, sol fare düğmesine çift tıklama aşağıdaki mesaj dizisini üretir:

WM_LBUTTONDOWN

WM_LBUTTONUP

WM_LBUTTONDBLCLK

WM_LBUTTONUP

Gerçekte, normalde oluşturulacak ikinci **WM_LBUTTONDOWN** mesajı bir **WM_LBUTTONDBLCLK** mesajı haline gelir. Eşdeğer mesajlar sağ, orta ve XBUTTON düğmeleri için tanımlanmıştır.

Çift tıklama mesajını alana kadar, ilk fare tıklamasının bir çift tıklamanın başlangıcı olduğunu söylemenin bir yolu yoktur. Bu nedenle, bir çift tıklama eylemi ilk fare tıklamasıyla başlayan bir eylemi devam ettirmelidir. Örneğin, Windows Kabuğunda, tek bir tıklama bir klasörü seçerken, çift tıklama klasörü açar.

İstemci Olmayan Fare Mesajları

Pencerenin istemci olmayan alanında meydana gelen fare olayları için ayrı bir mesaj seti tanımlanmıştır. Bu mesajların adında "NC" harfleri bulunur. Örneğin, **WM_NCLBUTTONDOWN**, **WM_LBUTTONDOWN**'un istemci olmayan eşdeğeridir. **DefWindowProc** işlevi bu mesajları doğru şekilde işlediğinden, tipik bir uygulama bu mesajlara müdahale etmeyecektir. Ancak, bazı gelişmiş işlevler için yararlı olabilirler. Örneğin, başlık çubuğunda özel davranış uygulamak için bu mesajları kullanabilirsiniz. Bu mesajları işlerseniz, genellikle bunları daha sonra **DefWindowProc işlevine** aktarmanız gerekir. Aksi takdirde, uygulamanız pencereyi sürükleme veya simge durumuna küçültme gibi standart işlevleri bozacaktır.

Sonraki

[Fare Hareketi](#)

Fare hareketi

Makale - 02/02/2023 - 4 dakika okumak için

Fare hareket ettiğinde, Windows bir **WM_MOUSEMOVE** mesajı gönderir. Varsayılan olarak, **WM_MOUSEMOVE** imlecin bulunduğu pencereye gider. Bir sonraki bölümde açıklanan fareyi *yakalayıp* bu davranışı geçersiz kılabilirsiniz.

WM_MOUSEMOVE mesajı, fare tıklamaları için olan mesajlarla aynı parametreleri içerir. *lParam*'ın en düşük 16 biti x-koordinatını, sonraki 16 biti ise y-koordinatını içerir. Koordinatları *lParam*'dan açmak için **GET_X_LPARAM** ve **GET_Y_LPARAM** makrolarını kullanın. *wParam* parametresi, diğer fare düğmelerinin yanı sıra SHIFT ve CTRL tuşlarının durumunu gösteren bir bitlik VEYA bayrakları içerir. Aşağıdaki kod fare koordinatlarını *lParam*'dan alır.

C++

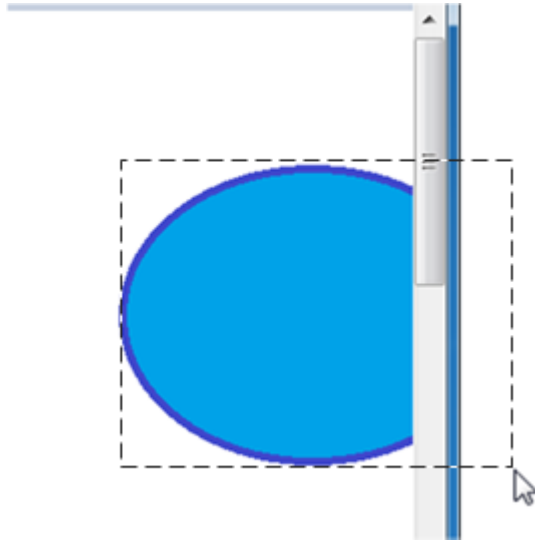
```
int xPos = GET_X_LPARAM(lParam);  
int yPos = GET_Y_LPARAM(lParam);
```

Bu koordinatların cihazdan bağımsız piksel (DIP) değil piksel cinsinden olduğunu unutmayın. Bu konunun ilerleyen bölümlerinde, iki birim arasında dönüşüm yapan kodlara bakacağız.

Bir pencere, imlecin konumu pencereye göre değişirse de bir **WM_MOUSEMOVE** mesajı alabilir. Örneğin, imleç bir pencerenin üzerine konumlandırılmışsa ve kullanıcı pencereyi gizlerse, fare hareket etmemiş olsa bile pencere **WM_MOUSEMOVE** mesajları alır. Bu davranışın bir sonucu, fare koordinatlarının **WM_MOUSEMOVE** mesajları arasında değişmeyebilmesidir.

Pencere dışındaki fare hareketini yakalama

Varsayılan olarak, fare istemci alanının kenarını geçerse bir pencere **WM_MOUSEMOVE** mesajlarını almayı durdurur. Ancak bazı işlemler için, fare konumunu bu noktanın ötesinde izlemeniz gerekebilir. Örneğin, bir çizim programı, aşağıdaki şemada gösterildiği gibi, kullanıcının seçim dikdörtgenini pencerenin kenarının ötesine sürüklemesini sağlayabilir.



Pencerenin kenarından geçen fare hareketi mesajlarını almak için **SetCapture** fonksiyonunu çağırın. Bu fonksiyon çağırıldıktan sonra, kullanıcı en az bir fare düğmesini basılı tuttuğu sürece, fare pencerenin dışına çıksa bile pencere **WM_MOUSEMOVE** mesajlarını almaya devam eder. Yakalama penceresi ön plan penceresi olmalıdır ve aynı anda yalnızca bir pencere yakalama penceresi olabilir. Fare yakalamayı serbest bırakmak için **ReleaseCapture** fonksiyonunu çağırın.

SetCapture ve **ReleaseCapture**'ı genellikle aşağıdaki şekilde kullanırsınız.

1. Kullanıcı sol fare düğmesine bastığında, fareyi yakalamaya başlamak için **SetCapture** ögesini çağırın.
2. Fare hareket mesajlarına yanıt verin.
3. Kullanıcı sol fare düğmesini bıraktığında **ReleaseCapture** ögesini çağırın.

Örnek: daire çizme

Kullanıcının fare ile bir daire çizmesini sağlayarak **Modül 3**'teki Daire programını genişletelim. **Direct2D Daire Örnek** programı ile başlayın. Basit çizim eklemek için bu örnekteki kodu değiştireceğiz. İlk olarak, programa yeni bir üye değişken ekleyin

`MainWindow` Sınıf.

C++

```
D2D1_POINT_2F ptMouse;
```

Bu değişken, kullanıcı fareyi sürüklerken farenin aşağı konumunu saklar. İçinde yapıldığı `MainWindow`, `elips` ve `ptMouse` değişkenlerini başlatın.

C++

```

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

```

Örneğin gövdesini `MainWindow::CalculateLayo` yöntemi; bu yöntem için gerekli değildir

C++

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

Ardından, sol düğme aşağı, sol düğme yukarı ve fare hareket mesajları için mesaj işleyicileri bildirin.

C++

```

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

```

Fare koordinatları fiziksel piksel olarak verilir, ancak Direct2D aygıtı bağımsız pikseller (DIP'ler) bekler. Yüksek DPI ayarlarını doğru şekilde işlemek için piksel koordinatlarını DIP'lere çevirmeniz gerekir. DPI hakkında daha fazla bilgi için [DPI ve Aygıt Bağımsız Pikseller bölümüne](#) bakın. Aşağıdaki kod, pikselleri DIP'lere dönüştüren bir yardımcı sınıfı göstermektedir.

C++

```

sınıf DPIScale
{
    statik float ölçek;

    Halka açık:
    statik void Initialize(HWND hwnd)
    {
        float dpi = GetDpiForWindow(hwnd);
        scale = dpi/96.0f;
    }

    şablon <tipadi T>
    statik D2D1_POINT_2F PixelsToDips(T x, T y)
    {
        return D2D1::Point2F(static_cast<float>(x) / scale,
            static_cast<float>(y) / scale);
    }
};

```

Direct2D fabrika nesnesini oluşturduktan sonra **WM_CREATE** işleyicinizde **DPIScale::Initialize** ögesini çağırın.

C++

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

Fare koordinatlarını fare mesajlarından DIP cinsinden almak için aşağıdakileri yapın:

1. Piksel koordinatlarını almak için **GET_X_LPARAM** ve **GET_Y_LPARAM** makrolarını kullanın. Bu makrolar WindowsX.h dosyasında tanımlanmıştır, bu nedenle bu başlığı projenize eklemeyi unutmayın.
2. Pikselleri DIP'ye dönüştürmek için **DIPToDIP** arayın.

Şimdi mesaj işleyicilerini pencere prosedürünüze ekleyin.

C++

```
case WM_LBUTTONDOWN:
    OnLButtonDown(GET_X_LPARAM(IParam), GET_Y_LPARAM(IParam),
    (DWORD)wParam);
    O döndür;

case WM_LBUTTONUP:
    OnLButtonUp();
    return 0;

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

Son olarak, mesaj işleyicilerini kendiniz uygulayın.

Sol düğme aşağı

Sol tuş aşağı mesajı için aşağıdakileri yapın:

1. Fareyi yakalamaya başlamak için **SetCapture** ögesini çağırın.
2. Fare tıklamasının konumunu *ptMouse* değişkeninde saklayın. Bu konum, elips için sınırlayıcı kutunun sol üst köşesini tanımlar.
3. Elips yapısını sıfırlayın.
4. **InvalidateRect** ögesini çağırın. Bu fonksiyon pencereyi yeniden boyanmaya zorlar.

C++

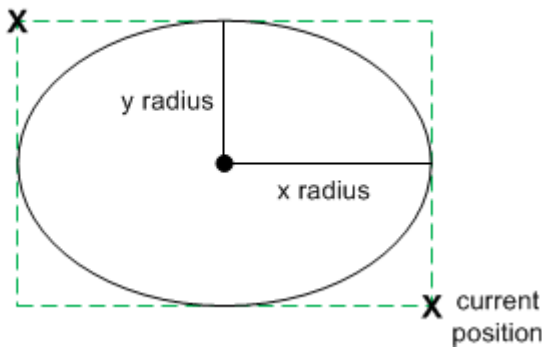
```
void MainWindow::OnLButtonDown(int pixelX, int pixelY, DWORD flags)
{
    SetCapture(m_hwnd);
    ellipse.point = ptMouse = DIPS::PixelsToDips(pixelX, pixelY);
    ellipse.radiusX = ellipse.radiusY = 1.0f;
    InvalidateRect(m_hwnd, NULL, FALSE);
}
```

Fare hareketi

Fare hareket mesajı için, sol fare düğmesinin kapalı olup olmadığını kontrol edin. Eğer öyleyse, elipsi yeniden hesaplayın ve pencereyi yeniden boyayın. Direct2D'de bir elips, merkez noktası ve x ve y yarıçapları ile tanımlanır. Farenin aşağı noktası (*ptMouse*) ve geçerli imleç konumu (*x*, *y*) tarafından tanımlanan sınırlayıcı kutuya uyan bir elips çizmek istiyoruz, bu nedenle elipsin genişliğini, yüksekliğini ve konumunu bulmak için biraz aritmetik gerekir.

Aşağıdaki kod elipsi yeniden hesaplar ve ardından pencereyi yeniden boyamak için **InvalidateRect**'i çağırır.

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 + yükseklik;

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

InvalidateRect(m_hwnd, NULL, FALSE);
    }
}
```

Sol düğme yukarı

Sol düğme yukarı mesajı için, fare yakalamayı serbest bırakmak üzere [ReleaseCapture](#)'ı çağırmanız yeterlidir.

C++

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

Sonraki

- [Diğer Fare İşlemleri](#)

Çeşitli Fare İşlemleri

Makale - 19/11/2022 - Okumak için 5 dakika

Önceki bölümlerde fare tıklamaları ve fare hareketleri ele alınmıştı. İşte fare ile gerçekleştirilebilecek diğer bazı işlemler.

UI Öğelerini Sürükleme

Kullanıcı arayüzünüz kullanıcı arayüzü öğelerinin sürüklenmesini destekliyorsa, fare aşağı mesajı işleyicinizde çağırmanız gereken bir işlev daha vardır: **DragDetect**. **DragDetect** işlevi, kullanıcı sürükleme olarak yorumlanması gereken bir fare hareketi başlatırsa **TRUE** değerini döndürür. Aşağıdaki kodda bu fonksiyonun nasıl kullanılacağı gösterilmektedir.

C++

```
case WM_LBUTTONDOWN:
{
    POINT pt = { GET_X_LPARAM(IParam), GET_Y_LPARAM(IParam) }; if
    (DragDetect(m_hwnd, pt))
    {
        // Sürüklemeye başlayın.
    }
}
O döndür;
```

Fikir şu: Bir program sürükle ve bırak özelliğini desteklediğinde, her fare tıklamasının sürükleme olarak yorumlanmasını istemezsiniz. Aksi takdirde, kullanıcı sadece üzerine tıklamak istediği bir şeyi (örneğin seçmek için) yanlışlıkla sürükleyebilir. Ancak bir fare özellikle hassassa, tıklarken fareyi tamamen sabit tutmak zor olabilir.

Bu nedenle, Windows birkaç piksellik bir sürükleme eşiği tanımlar. Kullanıcı fare düğmesine bastığında, fare bu eşiği geçmediği sürece sürükleme olarak kabul edilmez. **DragDetect** fonksiyonu bu eşiğe ulaşıp ulaşılmadığını test eder. İşlev **TRUE** değerini döndürürse, fare tıklamasını sürükleme olarak yorumlayabilirsiniz. Aksi takdirde, yorumlamayın.

7 Not

DragDetect **FALSE** döndürürse, kullanıcı fare düğmesini bıraktığında Windows **WM_LBUTTONUP** mesajını bastırır. Bu nedenle, programınız o anda sürüklemeyi destekleyen bir modda değilse **DragDetect**'i çağırmayın. (Örneğin, sürüklenebilir bir UI öğesi zaten seçiliyse.) Bu modülün sonunda, **DragDetect** işlevini kullanan daha uzun bir kod örneği göreceğiz.

İmleci Sınırlama

Bazen imleci istemci alanıyla veya istemci alanının bir bölümüyle sınırlamak isteyebilirsiniz. **ClipCursor** işlevi imlecin hareketini belirtilen bir dikdörtgenle sınırlar. Bu dikdörtgen istemci koordinatları yerine ekran koordinatlarında verilir, bu nedenle (0, 0) noktası ekranın sol üst köşesi anlamına gelir. İstemci koordinatlarını ekran koordinatlarına çevirmek için **ClientToScreen** fonksiyonunu çağırın.

Aşağıdaki kod imleci pencerenin istemci alanıyla sınırlar.

C++

```
// Pencere istemci alanını alın.  
RECT rc;  
GetClientRect(m_hwnd, &rc);  
  
// İstemci alanını ekran koordinatlarına dönüştürün.  
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);  
  
// İmleci sınırlandırın.  
ClipCursor(&rc);
```

ClipCursor bir **RECT** yapısı alır, ancak **ClientToScreen** bir **POINT** yapısı alır. Bir dikdörtgen, sol üst ve sağ alt noktaları ile tanımlanır. İmleci pencerenin dışındaki alanlar da dahil olmak üzere herhangi bir dikdörtgen alanla sınırlayabilirsiniz, ancak imleci istemci alanıyla sınırlamak işlevi kullanmanın tipik bir yoludur. İmleci pencerenizin tamamen dışındaki bir bölgeyle sınırlandırmak alışılmadık bir durumdur ve kullanıcılar muhtemelen bunu bir hata olarak algılayacaktır.

Kısıtlamayı kaldırmak için **ClipCursor** öğesini **NULL** değeriyle çağırın.

C++

```
ClipCursor(NULL);
```

Fare İzleme Olayları: Üzerine Gelme ve Ayrılma

Diğer iki fare mesajı varsayılan olarak devre dışıdır, ancak bazı uygulamalar için yararlı olabilir:

- ♦ **WM_MOUSEHOVER**: İmleç sabit bir süre boyunca istemci alanı üzerinde gezinmiştir.
- ♦ **WM_MOUSELEAVE**: İmleç istemci alanını terk etti.

Bu mesajları etkinleştirmek için **TrackMouseEvent** işlevini çağırın.

C++

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

TRACKMOUSEEVENT yapısı fonksiyon için parametreleri içerir. Yapının **dwFlags** üyesi, hangi izleme mesajlarıyla ilgilendiğinizi belirten bit bayraklarını içerir. Burada gösterildiği gibi hem **WM_MOUSEHOVER** hem de **WM_MOUSELEAVE** almayı ya da sadece ikisinden birini almayı seçebilirsiniz. **dwHoverTime** üyesi, sistem bir gezinme mesajı oluşturmadan önce farenin ne kadar süre gezinmesi gerektiğini belirtir. Bu değer milisaniye cinsinden verilir. **HOVER_DEFAULT** sabiti, sistem varsayılanının kullanılacağı anlamına gelir.

İstediğiniz mesajlardan birini aldıktan sonra **TrackMouseEvent** işlevi sıfırlanır. Başka bir izleme mesajı almak için tekrar çağırmanız gerekir. Ancak, **TrackMouseEvent** işlevini tekrar çağırmadan önce bir sonraki fare hareketi mesajına kadar beklemeniz gerekir. Aksi takdirde, pencereniz izleme mesajlarıyla dolup taşabilir. Örneğin, fare geziniyorsa, fare sabitken sistem bir **WM_MOUSEHOVER** mesajı akışı oluşturmaya devam edecektir. Fare başka bir noktaya hareket edip tekrar üzerine gelene kadar başka bir **WM_MOUSEHOVER** mesajı istemezsiniz.

İşte fare izleme olaylarını yönetmek için kullanabileceğiniz küçük bir yardımcı sınıf.

C++

```
class MouseTrackEvents  
{  
    bool m_bMouseTracking;  
  
    Halka açık:  
    MouseTrackEvents() : m_bMouseTracking(false)  
    {  
    }  
  
    void OnMouseMove(HWND hwnd)  
    {
```

Bir sonraki örnekte bu sınıfın pencere prosedürünüzde nasıl kullanılacağı gösterilmektedir.

C++

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

            // TODO: Fare hareket mesajını işleyin. return

            0;

        case WM_MOUSELEAVE:

            // TODO: Fareden ayrılma mesajını işleyin.

            mouseTrack.Reset(m_hwnd);
            0 döndür;

        case WM_MOUSEHOVER:

            // TODO: Fare üzerine gelme mesajını işleyin.

            mouseTrack.Reset(m_hwnd);
            0 döndür;

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

Fare izleme olayları sistem tarafından ek işlem gerektirir, bu nedenle ihtiyacınız yoksa bunları devre dışı bırakın.

Eksiksiz olması için, burada varsayılan fareyle üzerine gelme zaman aşımı için sistemi sorgulayan bir işlev bulunmaktadır.

C++

```
UINT GetMouseHoverTime()
{
    UINT msn;
    eğer (SystemParametersInfo(SPI_GETMOUSEHOVERTIME, 0, &msec, 0))
    {
        msn'yi döndür;
    }
    başka
    {
        0 döndür;
    }
}
```

Fare Tekerleği

Aşağıdaki fonksiyon, bir fare tekerleğinin mevcut olup olmadığını kontrol eder.

C++

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

Kullanıcı fare tekerleğini döndürürse, odağın bulunduğu pencere bir **WM_MOUSEWHEEL** iletisi alır. Bu mesajın *wParam* parametresi, tekerleğin ne kadar döndürüldüğünü ölçen *delta* adı verilen bir tamsayı değeri içerir. Delta keyfi birimler kullanır, burada 120 birim bir "eylem" gerçekleştirmek için gereken dönüş olarak tanımlanır. Elbette, bir eylemin tanımı programınıza bağlıdır. Örneğin, fare tekerleği metni kaydırmak için kullanılıyorsa, her 120 birimlik dönüş bir metin satırını kaydırır.

Deltanın işareti dönüş yönünü gösterir:

- Pozitif: Kullanıcıdan uzağa, öne doğru döndürün.
- Negatif: Geriye, kullanıcıya doğru döndürün.

Deltanın değeri bazı ek bayraklarla birlikte *wParam*'a yerleştirilir. Kullanın **GET_WHEEL_DELTA_WPARAM** makrosu delta değerini almak için.

C++

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

Fare tekerleği yüksek bir çözünürlüğe sahipse, deltanın mutlak değeri 120'den az olabilir. Bu durumda, eylemin daha küçük artışlarla gerçekleşmesi mantıklıysa, bunu yapabilirsiniz. Örneğin, metin bir satırdan daha az artışlarla kaydırılabilir. Aksi takdirde, tekerlek eylemi gerçekleştirmek için yeterince dönene kadar toplam deltayı biriktirin. Kullanılmayan deltayı bir değişkende saklayın ve 120 birim biriktiğinde (pozitif veya negatif) eylemi gerçekleştirin.

Sonraki

[Klavye Girişi](#)

Klavye Giriş (Win32 ve C++ ile Başlarken)

Makale - 18.10.2022 - Okumak için 7 dakika

Klavye, aşağıdakiler de dahil olmak üzere birkaç farklı giriş türü için

- kullanılır: Karakter girişi. Kullanıcının bir belgeye veya düzenleme kutusuna yazdığı metin.
- Klavye kısayolları. Uygulama işlevlerini çağıran tuş vuruşları; örneğin, bir dosyayı açmak için CTRL + O.
- Sistem komutları. Sistem işlevlerini çağıran tuş vuruşları; örneğin, pencereler arasında geçiş yapmak için ALT + TAB.

Klavye girişi hakkında düşünürken, bir tuş vuruşunun bir karakterle aynı olmadığını hatırlamak önemlidir. Örneğin, A tuşuna basmak aşağıdaki karakterlerden herhangi biriyle sonuçlanabilir.

- a
- A
- á (klavye aksanları birleştirmeyi destekliyorsa)

Ayrıca, ALT tuşu basılı tutulursa, A tuşuna basmak ALT+A sonucunu doğurur ve sistem bunu bir karakter olarak değil, bir sistem komutu olarak değerlendirir.

Anahtar Kodlar

Bir tuşa bastığınızda, donanım bir *tarama kodu* oluşturur. Tarama kodları bir klavyeden diğerine değişir ve tuş yukarı ve tuş aşağı olayları için ayrı tarama kodları vardır. Tarama kodlarını neredeyse hiç önemsemeyeceksiniz. Klavye sürücüsü tarama kodlarını *sanal tuş kodlarına* çevirir. Sanal tuş kodları cihazdan bağımsızdır. Herhangi bir klavyede A tuşuna basmak aynı sanal tuş kodunu üretir.

Genel olarak, sanal anahtar kodları ASCII kodlarına veya başka bir karakter kodlama standardına karşılık gelmez. Eğer düşünürseniz bu çok açıktır, çünkü aynı tuş farklı karakterler üretebilir (a, A, á) ve fonksiyon tuşları gibi bazı tuşlar herhangi bir karaktere karşılık gelmez.

Bununla birlikte, aşağıdaki sanal tuş kodları ASCII eşdeğerleriyle

- eşleşir: 0 - 9 arası tuşlar = ASCII '0' - '9' (0x30 - 0x39)
- A'dan Z'ye tuşlar = ASCII 'A' - 'Z' (0x41 - 0x5A)

Bazı açılardan bu eşleme talihsizdir, çünkü tartışılan nedenlerden dolayı sanal tuş kodlarını asla karakter olarak düşünmemelisiniz.

WinUser.h başlık dosyası, sanal tuş kodlarının çoğu için sabitler tanımlar. Örneğin, SOL OK tuşu için sanal tuş kodu **VK_LEFT**'tir (0x25). Sanal anahtar kodlarının tam listesi için [Sanal Anahtar Kodları](#) bölümüne bakın. ASCII değerleriyle eşleşen sanal tuş kodları için hiçbir sabit tanımlanmamıştır. Örneğin, A tuşu için sanal anahtar kodu 0x41'dir, ancak **VK_A** adında bir sabit yoktur. Bunun yerine, sadece sayısal değeri kullanın.

Key-Down ve Key-Up Mesajları

Bir tuşa bastığınızda, klavye odağına sahip olan pencere aşağıdaki mesajlardan birini alır.

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

[WM_SYSKEYDOWN](#) mesajı, bir *sistem* komutunu çağıran bir tuş vuruşu olan bir sistem tuşunu belirtir. İki tür sistem tuşu vardır:

- ♦ ALT +
- ♦ herhangi bir tuş F10

F10 tuşu bir pencerenin menü çubuğunu etkinleştirir. Çeşitli ALT tuşu kombinasyonları sistem komutlarını çağırır. Örneğin, ALT + TAB yeni bir pencereye geçiş yapar. Ayrıca, bir pencerede menü varsa, ALT tuşu menü öğelerini etkinleştirmek için kullanılabilir. Bazı ALT tuş kombinasyonları hiçbir şey yapmaz.

Diğer tüm tuş vuruşları sistem dışı tuşlar olarak kabul edilir ve [WM_KEYDOWN](#) Mesaj. Buna F10 dışındaki fonksiyon tuşları da dahildir.

Bir tuşu bıraktığınızda, sistem ilgili bir tuş-yukarı mesajı gönderir:

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

Klavyenin tekrarlama özelliğini başlatmak için bir tuşu yeterince uzun süre basılı tutarsanız, sistem birden fazla tuş aşağı mesajı ve ardından tek bir tuş yukarı mesajı gönderir.

Şimdiye kadar tartışılan klavye mesajlarının dördünde de *wParam* parametresi tuşun sanal anahtar kodunu içerir. *lParam* parametresi 32 bit içine paketlenmiş bazı çeşitli bilgiler içerir. Genellikle *lParam*'daki bilgilere ihtiyacınız olmaz. Yararlı olabilecek bir bayrak, tekrarlanan tuş indirme mesajları için 1'e ayarlanan "önceki tuş durumu"

bayrağı olan bit 30'dur.

Adından da anlaşılacağı gibi, sistem tuş vuruşları öncelikle işletim sistemi tarafından kullanılmak üzere tasarlanmıştır. **WM_SYSKEYDOWN** mesajını keserseniz, daha sonra **DefWindowProc** ögesini çağırın. Aksi takdirde, işletim sisteminin komutu işlemlerini engellemiş olursunuz.

Karakter Mesajları

Tuş vuruşları, ilk olarak **Modül 1**'de gördüğümüz **TranslateMessage** fonksiyonu tarafından karakterlere dönüştürülür. Bu fonksiyon tuşa basma mesajlarını inceler ve bunları karakterlere çevirir. Üretilen her karakter için, **TranslateMessage** fonksiyonu pencerenin mesaj kuyruğuna bir **WM_CHAR** veya **WM_SYSCHAR** mesajı koyar. Mesajın *wParam* parametresi UTF-16 karakterini içerir.

Tahmin edebileceğiniz gibi, **WM_CHAR** mesajları **WM_KEYDOWN** mesajlarından üretilirken, **WM_SYSCHAR** mesajları **WM_SYSKEYDOWN** mesajlarından üretilir. Örneğin, kullanıcının SHIFT tuşuna ve ardından A tuşuna bastığını varsayalım. Standart bir klavye düzeni olduğunu varsayarsak, aşağıdaki mesaj dizisini alırsınız:

WM_KEYDOWN: SHIFT

WM_KEYDOWN: A

WM_CHAR: 'A'

Öte yandan, ALT + P kombinasyonu üretecektir:

WM_SYSKEYDOWN: VK_MENU

WM_SYSKEYDOWN: 0x50

WM_SYSCHAR: 'p'

WM_SYSKEYUP: 0x50

WM_KEYUP: VK_MENU

(ALT tuşu için sanal tuş kodu tarihsel nedenlerden dolayı VK_MENU olarak adlandırılmıştır).

WM_SYSCHAR mesajı bir sistem karakterini belirtir. **WM_SYSKEYDOWN**'da olduğu gibi, bu mesajı genellikle doğrudan **DefWindowProc**'a aktarmalısınız. Aksi takdirde, standart sistem komutlarıyla etkileşime girebilirsiniz. Özellikle, **WM_SYSCHAR**'ı kullanıcının yazdığı metin olarak değerlendirmeyin.

WM_CHAR mesajı, normalde karakter girişi olarak düşündüğünüz şeydir. Karakterin veri türü **wchar_t**'dir ve UTF-16 Unicode karakterini temsil eder. Karakter girişi, özellikle Amerika Birleşik Devletleri dışında yaygın olarak kullanılan klavye düzenlerinde ASCII aralığı dışındaki karakterleri içerebilir. Bölgesel bir klavye yükleyerek ve ardından Ekran Klavyesi özelliğini kullanarak farklı klavye düzenlerini deneyebilirsiniz.

Kullanıcılar ayrıca Japonca karakterler gibi karmaşık komut dosyalarını standart bir klavye ile girmek için bir Giriş Yöntemi Düzenleyicisi (IME) yükleyebilirler. Örneğin, katakana karakteri 力 (ka) girmek için bir Japonca IME kullanarak aşağıdaki mesajları alabilirsiniz:

WM_KEYDOWN: VK_PROCESSKEY (IME İŞLEM tuşu)

WM_KEYUP: 0x4B **WM_KEYDOWN:**

VK_PROCESSKEY **WM_KEYUP:**

0x41 **WM_KEYDOWN:**

VK_PROCESSKEY **WM_CHAR:** 力

WM_KEYUP: VK_RETURN

Bazı CTRL tuş kombinasyonları ASCII kontrol karakterlerine çevrilir. Örneğin, CTRL+A ASCII ctrl-A (SOH) karakterine çevrilir (ASCII değeri 0x01). Metin girişi için genellikle kontrol karakterlerini filtrelemeniz gerekir. Ayrıca, klavye kısayollarını uygulamak için **WM_CHAR** kullanmaktan kaçının. Bunun yerine, **WM_KEYDOWN** mesajlarını kullanın; ya da daha iyisi, bir hızlandırıcı tablo kullanın. Hızlandırıcı tablolar bir sonraki konu olan Hızlandırıcı [Tablolar](#)'da açıklanmaktadır.

Aşağıdaki kod hata ayıklayıcıda ana klavye mesajlarını görüntüler. Farklı tuş kombinasyonlarıyla oynamayı deneyin ve hangi mesajların üretildiğini görün.

7 Not

wchar.h dosyasını dahil ettiğinizden emin olun, aksi takdirde swprintf_s tanımsız olacaktır.

C++

```
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);
            Mola;

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

        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);
        Mola;

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

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

    /* Diğer mesajları işleyin (gösterilmemiştir) */

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

```

Çeşitli Klavye Mesajları

Diğer bazı klavye mesajları çoğu uygulama tarafından güvenle göz ardı edilebilir.

- **WM_DEADCHAR** mesajı, aksan gibi bir birleştirme tuşu için gönderilir. Örneğin, İspanyolca klavyede aksan (') ve ardından E yazıldığında é karakteri oluşur. Aksan karakteri için **WM_DEADCHAR** mesajı gönderilir.
- **WM_UNICHAR** mesajı artık kullanılmamaktadır. ANSI programlarının Unicode karakter girişi almasını sağlar.
- **WM_IME_CHAR** karakteri, bir IME bir tuş vuruşu dizisini karakterlere çevirdiğinde gönderilir. Normal **WM_CHAR** mesajına ek olarak gönderilir.

Klavye Durumu

Klavye mesajları olay odaklıdır. Yani, bir tuşa basmak gibi ilginç bir şey olduğunda bir mesaj alırsınız ve mesaj size az önce ne olduğunu söyler. Ancak **GetKeyState** fonksiyonunu çağırarak istediğiniz zaman bir tuşun durumunu da test edebilirsiniz.

Örneğin, sol fare tıklaması + ALT tuşu kombinasyonunu nasıl tespit edeceğinizi düşünün. Tuş vuruşu mesajlarını dinleyerek ve bir bayrak depolayarak ALT tuşunun durumunu takip edebilirsiniz, ancak **GetKeyState** sizi bu zahmetten kurtarır. **WM_LBUTTONDOWN** mesajını aldığınızda, **GetKeyState**'i aşağıdaki gibi çağırmanız yeterlidir:

C++

```
if (GetKeyState(VK_MENU) & 0x8000)
{
    // ALT tuşu aşağıdadır.
}
```

GetKeyState mesajı girdi olarak bir sanal anahtar kodu alır ve bir dizi bit bayrağı (aslında sadece iki bayrak) döndürür. 0x8000 değeri, tuşun o anda basılı olup olmadığını test eden bit bayrağını içerir.

Çoğu klavyede sol ve sağ olmak üzere iki ALT tuşu vardır. Önceki örnek, bunlardan herhangi birine basılıp basılmadığını test eder. ALT, SHIFT veya CTRL tuşlarının sol ve sağ örneklerini ayırt etmek için **GetKeyState**'i de kullanabilirsiniz. Örneğin, aşağıdaki kod sağ ALT tuşunun basılı olup olmadığını test eder.

C++

```
if (GetKeyState(VK_RMENU) & 0x8000)
{
    // Sağ ALT tuşu aşağıdadır.
}
```

GetKeyState işlevi ilginçtir çünkü *sanal* bir klavye durumu bildirir. Bu sanal durum, mesaj kuyruğunuzun içeriğine dayanır ve siz mesajları kuyruktan çıkardıkça güncellenir. Programınız pencere mesajlarını işlerken, **GetKeyState** size her mesajın kuyruğa alındığı andaki klavyenin anlık görüntüsünü verir. Örneğin, kuyruktaki son mesaj **WM_LBUTTONDOWN** ise, **GetKeyState** kullanıcının fare düğmesini tıklattığı andaki klavye durumunu bildirir.

GetKeyState mesaj kuyruğunuzu temel aldığından, başka bir programa gönderilen klavye girdisini de yok sayar. Kullanıcı başka bir programa geçerse, bu programa gönderilen tüm tuş basışları **GetKeyState** tarafından yok sayılır. Klavyenin anlık fiziksel durumunu gerçekten bilmek istiyorsanız, bunun için bir fonksiyon vardır: **GetAsyncKeyState**. Ancak çoğu kullanıcı arayüzü kodu için doğru fonksiyon **GetKeyState**'tir.

Sonraki

[Hızlandırıcı Masalar](#)

Hızlandırıcı Masalar

Makale - 04/27/2021 - Okumak için 5 dakika

Uygulamalar genellikle Dosya Aç komutu için CTRL+O gibi klavye kısayolları tanımlar. [WM_KEYDOWN](#) mesajlarını tek tek işleyerek klavye kısayollarını uygulayabilirsiniz, ancak hızlandırıcı tabloları bu konuda daha iyi bir çözüm sağlar:

- Daha az kodlama gerektirir.
- Tüm kısayollarınızı tek bir veri dosyasında
- birleştirir. Diğer dillere yerelleştirmeyi destekler.
- Kısayolların ve menü komutlarının aynı uygulama mantığını kullanmasını sağlar.

Hızlandırıcı tablosu, CTRL+O gibi klavye kombinasyonlarını uygulama komutlarıyla eşleyen bir veri kaynağıdır. Bir hızlandırıcı tablosunu nasıl kullanacağımızı görmeden önce, kaynaklara hızlı bir giriş yapmamız gerekecek. *Kaynak*, bir uygulama ikili dosyasında (EXE veya DLL) yerleşik olarak bulunan bir veri bloğudur. Kaynaklar, menüler, imleçler, simgeler, resimler, metin dizeleri veya herhangi bir özel uygulama verisi gibi uygulama tarafından ihtiyaç duyulan verileri depolar. Uygulama, kaynak verilerini çalışma zamanında ikiliden yükler. Kaynakları bir ikiliye dahil etmek için aşağıdakileri yapın:

1. Bir kaynak tanımı (.rc) dosyası oluşturun. Bu dosya kaynak türlerini ve bunların tanımlayıcılarını tanımlar. Kaynak tanım dosyası diğer dosyalara referanslar içerebilir. Örneğin, bir simge kaynağı .rc dosyasında bildirilir, ancak simge görüntüsü ayrı bir dosyada saklanır.
2. Kaynak tanım dosyasını derlenmiş bir kaynak (.res) dosyasına derlemek için Microsoft Windows Kaynak Derleyicisini (RC) kullanın. RC derleyicisi Visual Studio ve ayrıca Windows SDK ile birlikte sağlanır.
3. Derlenmiş kaynak dosyasını ikili dosyaya bağlayın.

Bu adımlar kabaca kod dosyaları için derleme/bağlama işlemine eşdeğerdir. Visual Studio, kaynak oluşturmaya ve değiştirmeyi kolaylaştıran bir dizi kaynak düzenleyicisi sağlar. (Bu araçlar Visual Studio'nun Express sürümlerinde mevcut değildir.) Ancak bir .rc dosyası basitçe bir metin dosyasıdır ve sözdizimi MSDN'de belgelenmiştir, bu nedenle herhangi bir metin düzenleyici kullanarak bir .rc dosyası oluşturmak mümkündür. Daha fazla bilgi için [Kaynak Dosyaları Hakkında](#) bölümüne bakın.

Hızlandırıcı Tablosu Tanımlama

Hızlandırıcı tablosu, klavye kısayollarından oluşan bir tablodur. Her kısayol şu şekilde tanımlanır:

- Sayısal bir tanımlayıcı. Bu sayı, kısayol tarafından çağrılacak uygulama komutunu tanımlar.

- Kısayolun ASCII karakteri veya sanal anahtar kodu. İsteğe
- bağlı değiştirici tuşlar: ALT, SHIFT veya CTRL.

Hızlandırıcı tablonun kendisi, uygulama kaynakları listesinde tabloyu tanımlayan sayısal bir tanımlayıcıya sahiptir. Basit bir çizim programı için bir hızlandırıcı tablo oluşturalım. Bu programın iki modu olacaktır, çizim modu ve seçim modu. Çizim modunda, kullanıcı şekiller çizebilir. Seçim modunda, kullanıcı şekilleri seçebilir. Bu program için aşağıdaki klavye kısayollarını tanımlamak istiyoruz.

Kısayol	Komuta
CTRL+M	Modlar arasında geçiş yapın.
F1	Çizim moduna geçin.
F2	Seçim moduna geçin.

İlk olarak, tablo ve uygulama komutları için sayısal tanımlayıcılar tanımlayın. Bu değerler isteğe bağlıdır. Bir başlık dosyasında tanımlayarak tanımlayıcılar için sembolik sabitler atayabilirsiniz. Örneğin:

C++

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

Bu örnekte, değer hızlandırıcı tablosunu tanımlar ve sonraki üç sabitleri uygulama komutlarını tanımlar. Geleneksel olarak, kaynak sabitlerini tanımlayan bir başlık dosyası genellikle resource.h olarak adlandırılır.

C++

```
#include "resource.h"

IDR_ACCEL1 HIZLANDIRICILAR
{
    0x4D, ID_TOGGLE_MODE, VIRTKEY, KONTROL // ctrl-M
    0x70, ID_DRAW_MODE, VIRTKEY // F1
    0x71, ID_SELECT_MODE, VIRTKEY // F2
}
```

Hızlandırıcı kısayolları küme parantezleri içinde tanımlanır. Her kısayol aşağıdaki girdileri içerir.

- Kısayolu çağıran sanal tuş kodu veya ASCII karakteri.
- Uygulama komutu. Örnekte sembolik sabitlerin kullanıldığına dikkat edin. Kaynak tanımlama dosyası, bu sabitlerin tanımlandığı resource.h dosyasını içerir. **VIRTKEY**
- anahtar sözcüğü, ilk girişin bir sanal anahtar kodu olduğu anlamına gelir. Diğer seçenek ASCII karakterleri kullanmaktır.
- İsteğe bağlı değiştiriciler: ALT, CONTROL veya SHIFT.

Kısayollar için ASCII karakterleri kullanırsanız, küçük harfli bir karakter büyük harfli bir karakterden farklı bir kısayol olacaktır. (Örneğin, 'a' yazmak 'A' yazmaktan farklı bir komut çağırabilir.) Bu kullanıcıların kafasını karıştırabilir, bu nedenle kısayollar için ASCII karakterleri yerine sanal tuş kodlarını kullanmak genellikle daha iyidir.

Hızlandırıcı Tablosunun Yüklmesi

Programın kullanılabilmesi için önce hızlandırıcı tablosunun kaynağının yüklenmesi gerekir. Bir hızlandırıcı tablosunu yüklemek için [LoadAccelerators](#) fonksiyonunu çağırın.

C++

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

Mesaj döngüsüne girmeden önce bu fonksiyonu çağırın. İlk parametre modülün tanıtıcısıdır. (Bu parametre [WinMain](#) işlevinize aktarılır. Ayrıntılar için [WinMain: Uygulama Giriş Noktası bölümüne](#) bakın). İkinci parametre kaynak tanımlayıcısıdır. İşlev, kaynağa bir tanıtıcı döndürür. Tanıtıcının, sistem tarafından yönetilen bir nesneye başvuran opak bir tür olduğunu hatırlayın. İşlev başarısız olursa **NULL** döndürür.

[DestroyAcceleratorTable](#) fonksiyonunu çağırarak bir hızlandırıcı tablosunu serbest bırakabilirsiniz. Ancak, programdan çıkıldığında sistem tabloyu otomatik olarak serbest bırakır, bu nedenle bu işlevi yalnızca bir tabloyu başka bir tabloyla değiştiriyorsanız çağırmanız gerekir. [Kullanıcı Tarafından Düzenlenebilir Hızlandırıcılar Oluşturma](#) konusunda bunun ilginç bir örneği bulunmaktadır.

Tuş Vuruşlarını Komutlara Çevirme

Bir hızlandırıcı tablo, tuş vuruşlarını [WM_COMMAND](#) mesajlarına çevirerek çalışır. **WM_COMMAND**'ın *wParam* parametresi komutun sayısal tanımlayıcısını içerir. Örneğin, daha önce gösterilen tabloyu kullanarak, CTRL+M tuş vuruşu

ID_TOGGLE_MODE değerine sahip bir **WM_COMMAND** mesajına çevrilir. Bunu gerçekleştirmek için mesaj döngüsünü aşağıdaki şekilde değiştirin:

C++

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

Bu kod, mesaj döngüsünün içine **TranslateAccelerator** fonksiyonuna bir çağrı ekler. **TranslateAccelerator işlevi**, her pencere mesajını inceleyerek tuş aşağı mesajlarını arar. Kullanıcı hızlandırıcı tablosunda listelenen tuş kombinasyonlarından birine basarsa, **TranslateAccelerator** pencereye bir **WM_COMMAND** mesajı gönderir. Fonksiyon, pencere prosedürünü doğrudan çağırarak **WM_COMMAND** gönderir. **TranslateAccelerator** bir tuş vuruşunu başarıyla çevirdiğinde, fonksiyon sıfır olmayan bir değer döndürür, bu da mesaj için normal işlemeyi atlamanız gerektiği anlamına gelir. Aksi takdirde, **TranslateAccelerator** sıfır değerini döndürür. Bu durumda, pencere mesajını normal şekilde **TranslateMessage** ve **DispatchMessage**'a iletin.

Çizim programının **WM_COMMAND** mesajını nasıl ele alabileceği aşağıda açıklanmıştır:

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);
            }
            başka
            {
                SetMode(DrawMode);
            }

            Mola;
    }
}
```

Bu kod, uygulama tarafında `setMod` geçiş yapmak için tanımlanmış bir işlev olduğunu varsayar

iki mod arasında geçiş yapabilirsiniz. Her bir komutu nasıl ele alacağınıza ilişkin ayrıntılar elbette programınıza bağlıdır.

Sonraki

[İmleç Görüntüsünü Ayarlama](#)

İmleç Görüntüsünü Ayarlama

Makale - 08/19/2020 - Okumak için 2 dakika

İmleç, farenin veya diğer işaretleme aygıtının konumunu gösteren küçük görüntüdür. Birçok uygulama kullanıcıya geri bildirim vermek için imleç görüntüsünü değiştirir. Gerekli olmasa da, başvuruza hoş bir cila katar.

Windows, *sistem imleçleri* adı verilen bir dizi standart imleç görüntüsü sağlar. Bunlar arasında ok, el, I-kirişi, kum saati (artık dönen bir daire olan) ve diğerleri bulunur. Bu bölümde sistem imleçlerinin nasıl kullanılacağı açıklanmaktadır. Özel [imleçler](#) oluşturmak gibi daha gelişmiş görevler için bkz.

'nin **hCursor** üyesini ayarlayarak bir imleci bir pencere sınıfıyla ilişkilendirebilirsiniz. **WNDCLASS** veya **WNDCLASSEX** yapısı. Aksi takdirde, varsayılan imleç oktur. Fare bir pencerenin üzerine geldiğinde, pencere bir **WM_SETCURSOR** mesajı alır (başka bir pencere fareyi yakalamadığı sürece). Bu noktada, aşağıdaki olaylardan biri gerçekleşir:

- Uygulama imleci ayarlar ve pencere prosedürü **TRUE** değerini döndürür.
- Uygulama hiçbir şey yapmaz ve **WM_SETCURSOR** ögesini **DefWindowProc** ögesine geçirir.

İmleci ayarlamak için bir program aşağıdakileri yapar:

1. İmleci belleğe yüklemek için **LoadCursor** işlevini çağırır. Bu fonksiyon imlece bir tanıtıcı döndürür.
2. **SetCursor** ögesini çağırır ve imleç tanıtıcısını iletir.

Aksi takdirde, uygulama **WM_SETCURSOR** ögesini **DefWindowProc** ögesine geçirirse **DefWindowProc** işlevi imleç görüntüsünü ayarlamak için aşağıdaki algoritmayı kullanır:

1. Pencerenin bir üst ögesi varsa, **WM_SETCURSOR** mesajını işlemek üzere üst ögeye iletin.
2. Aksi takdirde, pencerede bir sınıf imleci varsa, imleci sınıf imlecine ayarlayın.
3. Sınıf imleci yoksa, imleci ok imlecine ayarlayın.

LoadCursor işlevi, bir kaynaktan özel bir imleci ya da sistem imleçlerinden birini yükleyebilir. Aşağıdaki örnekte imlecin sistem el imlecine nasıl ayarlanacağı gösterilmektedir.

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

İmleci değiştirirseniz, **WM_SETCURSOR** mesajını yakalayıp imleci yeniden ayarlamadığınız sürece, imleç görüntüsü bir sonraki fare hareketinde sıfırlanır. Aşağıdaki kod **WM_SETCURSOR**'un nasıl işleneceğini gösterir.

C++

```
case WM_SETCURSOR:  
    if (LOWORD(IParam) == HTCLIENT)  
    {  
        SetCursor(hCursor);  
        TRUE döndürür;  
    }  
Mola;
```

Bu kod ilk olarak *IParam*'ın alt 16 bitini kontrol eder. **HTCLIENT** değerine eşitse imlecin pencerenin istemci alanı üzerinde olduğu anlamına gelir. Aksi takdirde, imleç istemci olmayan alanın üzerindedir. Tipik olarak, imleci yalnızca istemci alanı için ayarlamalı ve Windows'un istemci olmayan alan için imleci ayarlamasına izin vermelisiniz.

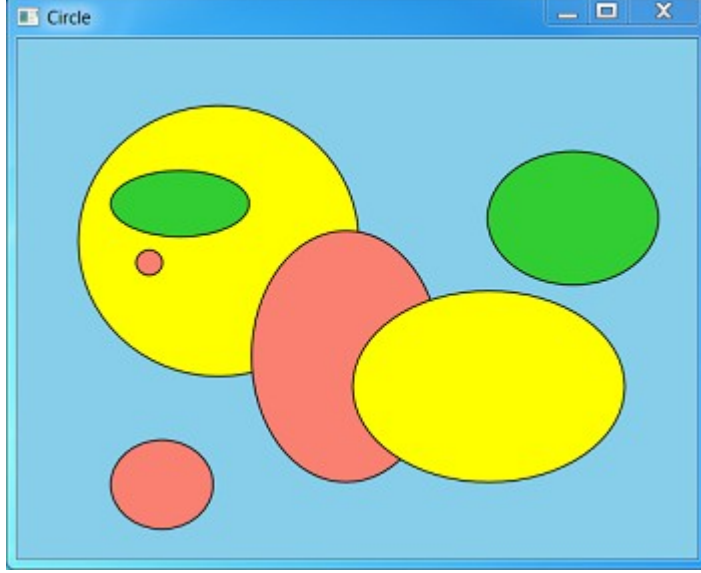
Sonraki

[Kullanıcı Girişi: Genişletilmiş Örnek](#)

Kullanıcı giriři: genişletilmiş örnek

Makale - 04/28/2022 - Okumak için 5 dakika

Basit bir çizim programı oluşturmak için kullanıcı girdisi hakkında öğrendiğimiz her şeyi birleştirelim. İşte programın bir ekran görüntüsü:



Kullanıcı birkaç farklı renkte elips çizebilir ve elipsleri seçebilir, taşıyabilir veya silebilir. Kullanıcı arayüzünü basit tutmak için, program kullanıcının elips renklerini seçmesine izin vermez.

Bunun yerine, program otomatik olarak önceden tanımlanmış bir renk listesi arasında geçiş yapar. Program elipsler dışında herhangi bir şekli desteklemiyor. Açıkçası, bu program grafik yazılımı için herhangi bir ödöl kazanmayacaktır. Ancak, yine de öğrenmek için yararlı bir örnektir. Kaynak kodunun tamamını [Simple Drawing Sample](#) adresinden indirebilirsiniz. Bu bölüm sadece bazı önemli noktaları kapsayacaktır.

Elipsler programda elips verilerini ([D2D1_ELLIPSE](#)) ve rengi ([D2D1_COLOR_F](#)) içeren bir yapı ile temsil edilir. Yapı ayrıca iki yöntem tanımlar: elipsi çizmek için bir yöntem ve isabet testi gerçekleştirmek için bir yöntem.

C++

```

struct MyEllipse
{
    D2D1_ELLIPSE    elips;
    D2D1_COLOR_F    Renkli;

    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;
    }
};

```

Program, her elipsin dolgusunu ve dış çizgisini çizmek için aynı düz renk fırçayı kullanır ve gerektiğinde rengi değiştirir. Direct2D'de, düz renkli bir fırçanın rengini değiştirmek verimli bir işlemdir. Bu nedenle, düz renkli fırça nesnesi bir **SetColor** yöntemini destekler.

Elipsler bir STL **liste** konteynerinde saklanır:

C++

```
list<shared_ptr<MyEllipse>>
```

```
üç nokta;
```

7 Not

shared_ptr, TR1'de C++'a eklenen ve C++0x'te resmileştirilen bir smart-pointer sınıfıdır. Visual Studio 2010, **shared_ptr** ve diğer C++0x özellikleri için destek ekler. Daha fazla bilgi için *MSDN Magazine*'de **Visual Studio 2010'da Yeni C++ ve MFC Özelliklerini Keşfetme** bölümüne bakın. (Bu kaynak bazı dillerde ve ülkelerde mevcut olmayabilir).

Programın üç modu vardır:

- ◆ Çizim modu. Kullanıcı yeni elipsler çizebilir. Seçim
- ◆

modu. Kullanıcı bir elips seçebilir. Sürükleme modu.

Kullanıcı seçili bir elipsi sürükleyebilir.

Kullanıcı, [Hızlandırıcı Tablolar](#)'da açıklanan klavye kısayollarını kullanarak çizim modu ile seçim modu arasında geçiş yapabilir. Kullanıcı bir elipse tıklarsa, program seçim modundan sürükleme moduna geçer. Kullanıcı fare düğmesini bıraktığında seçim moduna geri döner. Geçerli seçim bir yineleyici olarak saklanır

elipsler listesine ekler. Yardımcı yöntem `MainWindow::Secim` 'e bir işaretçi döndürür. seçilen elips veya seçim yoksa **nullptr** değeri.

C++

```
list<shared_ptr<MyEllipse>>::iterator seçim;

shared_ptr<MyEllipse> Selection()
{
    eğer (seçim == ellipses.end())
    {
        nullptr döndürür;
    }
    başka
    {
        return (*seçim);
    }
}

geçersiz ClearSelection() { seçim = ellipses.end(); }
```

Aşağıdaki tablo, üç modun her birinde fare girişinin etkilerini özetlemektedir.

Fare Giriş	Çizim Modu	Seçim Modu	Sürükle Mod
Sol elçizmeye başlarsanızimleci yakalayın, elipsi seçin yapın. aşağı sürükleme moduna geçin	Fare yakalamayı ayarla Düğme yok	Geçerli seçimi bırakın ve bir isabet testi gerçekleştirin. Eğer vebir yeni el	eylemi ve
FareSol düğme ise hareket et aşağı, yeniden boyutlandırın elips.	Eylem	yok.	Hareket ve seçilmiş elips.
düğme yukarı	SolHayır eylemini çizmeyi durdurun. elips.		Geçiş yap seçim Mod.

MainWindow

Sınıftaki aşağıdaki yöntem **WM_LBUTTONDOWN** mesajlarını işler.

C++

```

void MainWindow::OnLButtonDown(int pixelX, int pixelY, DWORD flags)
{
    const float dipX = DIPS::PixelsToDipsX(pixelX);
    const float dipY = DIPS::PixelsToDipsY(pixelY);

    if (mode == DrawMode)
    {
        NOKTA pt = { pikselX, pikselY };

        eğer (DragDetect(m_hwnd, pt))
        {
            SetCapture(m_hwnd);

            // Yeni bir elips başlatın.
            InsertEllipse(dipX, dipY);
        }
    }
    başka
    {
        ClearSelection();

        eğer (HitTest(dipX, dipY))
        {
            SetCapture(m_hwnd);

            ptMouse = Selection()->ellipse.point;
            ptMouse.x -= dipX;
            ptMouse.y -= dipY;

            SetMode(DragMode);
        }
    }
    InvalidateRect(m_hwnd, NULL, FALSE);
}

```

Fare koordinatları bu yöntemle piksel olarak aktarılır ve daha sonra DIP'lere dönüştürülür. Bu iki birimi karıştırmamak önemlidir. Örneğin, **DragDetect** işlevi pikselleri kullanır, ancak çizim ve isabet testi DIP'leri kullanır. Genel kural, pencereler veya fare girişi ile ilgili işlevlerin pikselleri, Direct2D ve DirectWrite işlevlerinin ise DIP'leri kullanmasıdır. Programınızı her zaman yüksek DPI ayarında test edin ve programınızı DPI farkında olarak işaretlemeyi unutmayın. Daha fazla bilgi için DPI [ve Aygıttan Bağımsız Pikseller bölümüne](#) bakın.

İşte **WM_MOUSEMOVE** mesajlarını işleyen kod.

C++

```

void MainWindow::OnMouseMove(int pixelX, int pixelY, DWORD flags)
{
    const float dipX = DIPS::PixelsToDipsX(pixelX);
    const float dipY = DIPS::PixelsToDipsY(pixelY);

    eğer ((flags & MK_LBUTTON) && Selection())
    {
        if (mode == DrawMode)
        {
            // Elipsi yeniden boyutlandırın.

            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 + yükseklik;

            Selection()->ellipse = D2D1::Ellipse(D2D1::Point2F(x1, y1),
width, height);
        }
        else if (mode == DragMode)
        {
            // Elipsi hareket ettirin.
            Selection()->ellipse.point.x = dipX + ptMouse.x;
            Selection()->ellipse.point.y = dipY + ptMouse.y;
        }
        InvalidateRect(m_hwnd, NULL, FALSE);
    }
}

```

Bir elipsi yeniden boyutlandırma mantığı daha önce [Örnek](#) bölümünde açıklanmıştı: [Daire Çizme bölümünde açıklanmıştı](#). Ayrıca [InvalidateRect](#) çağrısına da dikkat edin. Bu, pencerenin yeniden boyandığından emin olunmasını sağlar. Aşağıdaki kod [WM_LBUTTONDOWN](#) mesajlarını işler.

```

C++

void MainWindow::OnLButtonUp()
{
    if ((mode == DrawMode) && Selection())
    {
        ClearSelection();
        InvalidateRect(m_hwnd, NULL, FALSE);
    }
    else if (mode == DragMode)
    {
        SetMode(SelectMode);
    }
    ReleaseCapture();
}

```

Gördüğünüz gibi, fare girişi için mesaj işleyicilerinin tümü, geçerli moda bağlı olarak

dallanma koduna sahiptir. Bu oldukça basit bir program için kabul edilebilir bir tasarımıdır. Ancak, yeni modlar eklenirse hızla çok karmaşık hale gelebilir. Daha büyük bir program için model-view-controller (MVC) mimarisi daha iyi bir tasarım olabilir. Bu tür bir mimaride, kullanıcı girdisini işleyen *denetleyici*, uygulama verilerini yöneten *modelden* ayrılır.

Program mod değiştirdiğinde, kullanıcıya geri bildirim vermek için imleç değişir.

C++

```

void MainWindow::SetMode(Mode m)
{
    mod = m;

    // İmleci güncelle
    LPWSTR imleç;
    anahtar (mod)
    {
        case DrawMode:
            cursor = IDC_CROSS;
            break;

        case SelectMode:
            cursor = IDC_HAND;
            break;

        case DragMode:
            cursor = IDC_SIZEALL;
            break;
    }

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

```

Ve son olarak, pencere bir **WM_SETCURSOR** aldığıında imleci ayarlamayı unutmayın
Mesaj:

C++

```

case WM_SETCURSOR:
    if (LOWORD(lParam) == HTCLIENT)
    {
        SetCursor(hCursor);
        TRUE döndürür;
    }
    Mola;

```

Özet

Bu modülde, fare ve klavye girişinin nasıl işleneceğini; klavye kısayollarının nasıl tanımlanacağını ve programın mevcut durumunu yansıtmak için imleç görüntüsünün nasıl güncelleneceğini öğrendiniz.

Win32 ile Başlarken: Örnek Kod

Makale - 08/19/2020 - Okumak için 2 dakika

Bu bölüm, [Win32 ve C++ ile Başlarken](#) serisi için örnek kod bağlantıları içerir.

Bu bölümde

Konu	Açıklama
Windows Hello	Bu örnek uygulama, minimal bir Windows programının nasıl oluşturulacağını göstermektedir. Dünya Örneği
BaseWindow	Bu örnek uygulama, uygulama durumu verilerinin Sample WM_NCCREATE mesajı.
Diyalogu Aç	Bu örnek uygulama, Bileşen Nesne Modeli Kutusu Örneğinin nasıl başlatılacağını gösterir (COM) kitaplığını kullanabilir ve bir Windows programında COM tabanlı bir API kullanabilirsiniz.
Direct2D Çember	Bu örnek uygulama Direct2D kullanarak bir dairenin nasıl çizileceğini gösterir. Örnek uygulama
Direct2D Saat	Bu örnek uygulama, Örnek çizmek için Direct2D'de dönüşümlerin nasıl kullanılacağını gösterir Bir saatin akrep ve yelkovanı gibi.
Daire Çiz	Bu örnek uygulama, bir daire çizmek için fare girişinin nasıl kullanılacağını gösterir. Örnek uygulama
Basit Örnek	Bu örnek uygulama, nasıl çizim yapılacağını gösteren çok basit bir çizim programıdır fare girişi, klavye girişi ve hızlandırıcı tabloları kullanın.

İlgili konular

[Win32 ve C++ ile Başlarken](#)

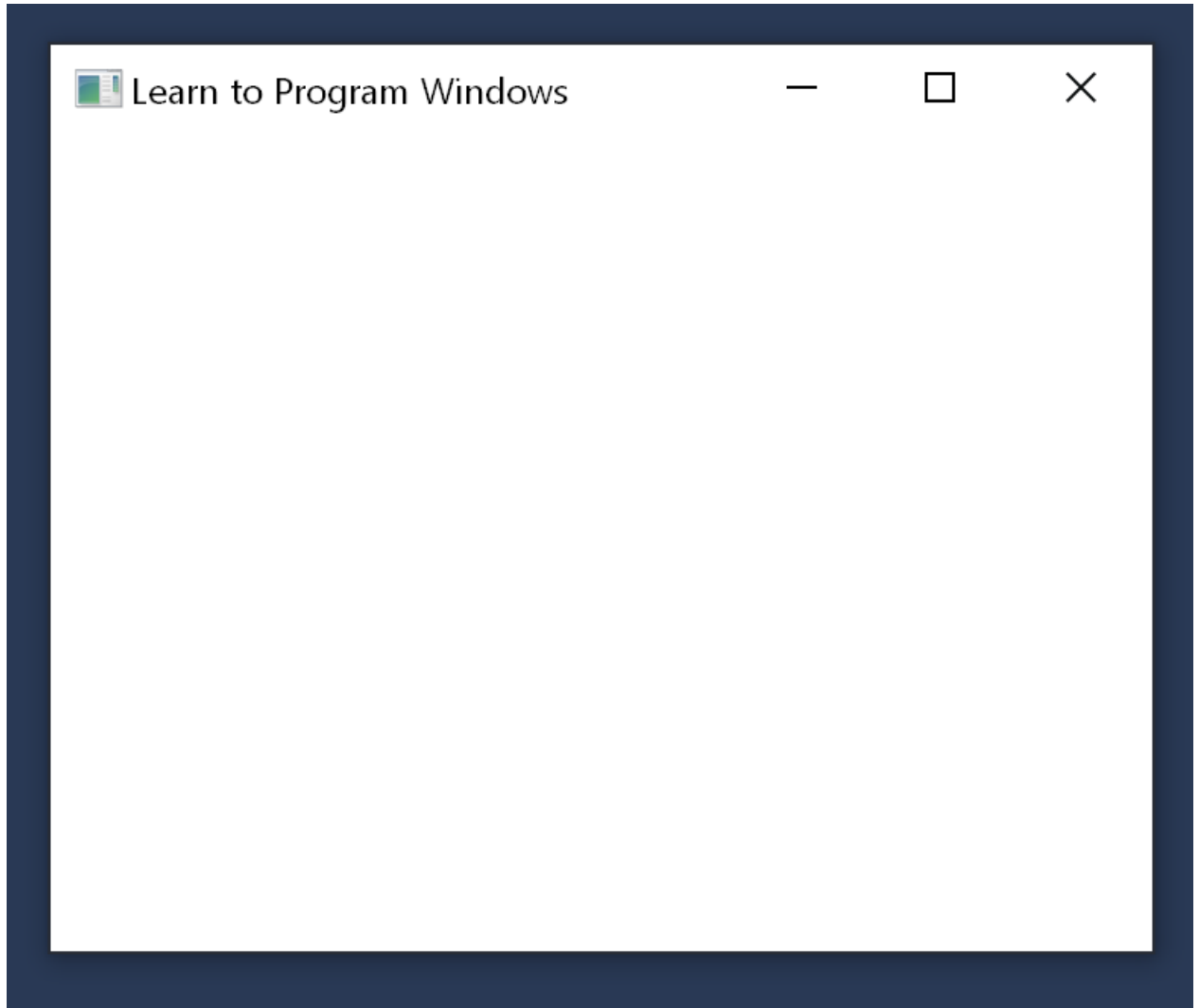
Windows Hello Dünya Örneđi

Makale - 10/09/2019 - Okumak için 2 dakika

Bu örnek uygulama, minimal bir Windows programının nasıl oluşturulacağını göstermektedir.

Açıklama

Windows Hello World örnek uygulaması, aşağıdaki ekran görüntüsünde gösterildiğı gibi boş bir pencere oluşturur ve gösterir. Bu örnek [Modül 1](#)'de ele alınmıştır. [İlk Windows Programınız](#).



Numunenin İndirilmesi

Bu örneđe [buradan](#) ulaşabilirsiniz .

İndirmek için GitHub'daki örnek deposunun kök dizinine gidin ([microsoft/Windows-classic-samples](#)) ve tüm örneklerin zip dosyasını bilgisayarınıza **indirmek için Klonla**

veya indir düğmesine tıklayın. Ardından klasörü açın.

Örneđi Visual Studio'da açmak için **Dosya / Aç / Proje / Çözüm'**ü seçin ve klasörü ve **Windows-classic-samples-master / Samples / Win7Samples / begin / LearnWin32 / HelloWorld / cpp dosyasını** açtığınız konuma gidin. **HelloWorld.sln** dosyasını açın.

Örnek yüklendikten sonra, Windows 10 ile çalışması için güncellemeniz gerekecektir. Visual Studio'daki **Proje** menüsünden **Özellikler'i** seçin. **Windows SDK Sürümünü** 10.0.17763.0 veya daha iyisi gibi bir Windows 10 SDK'sı olarak güncelleyin. Ardından **Platform Araç Seti'ni** Visual Studio 2017 veya daha iyisi olarak değiştirin. Şimdi F5 tuşuna basarak örneđi çalıştırabilirsiniz!

İlgili konular

- [Windows için Programlamayı Öğrenin: Örnek Kod](#)
- [Modül 1. İlk Windows Programınız](#)

BaseWindow Örneđi

Makale - 08/19/2020 - Okumak için 2 dakika

Bu örnek uygulama, [WM_NCCREATE](#) içinde uygulama durumu verilerinin nasıl aktarılacağını gösterir Mesaj.

Açıklama

BaseWindow örnek uygulaması, [Windows Merhaba Dünya Örneđi](#)'nin bir varyasyonudur. Uygulama verilerini pencere yordamına aktarmak için [WM_NCCREATE](#) mesajını kullanır. Bu örnek [Uygulama Durumunu Yönetme](#) konusunda ele alınmıştır.

Numunenin İndirilmesi

Bu örneđe [buradan](#) ulaşabilirsiniz.

İlgili konular

- [Windows için Programlamayı Öğrenin: Örnek Kod](#)
- [Uygulama Durumunu Yönetme](#)
- [Modül 1. İlk Windows Programınız](#)

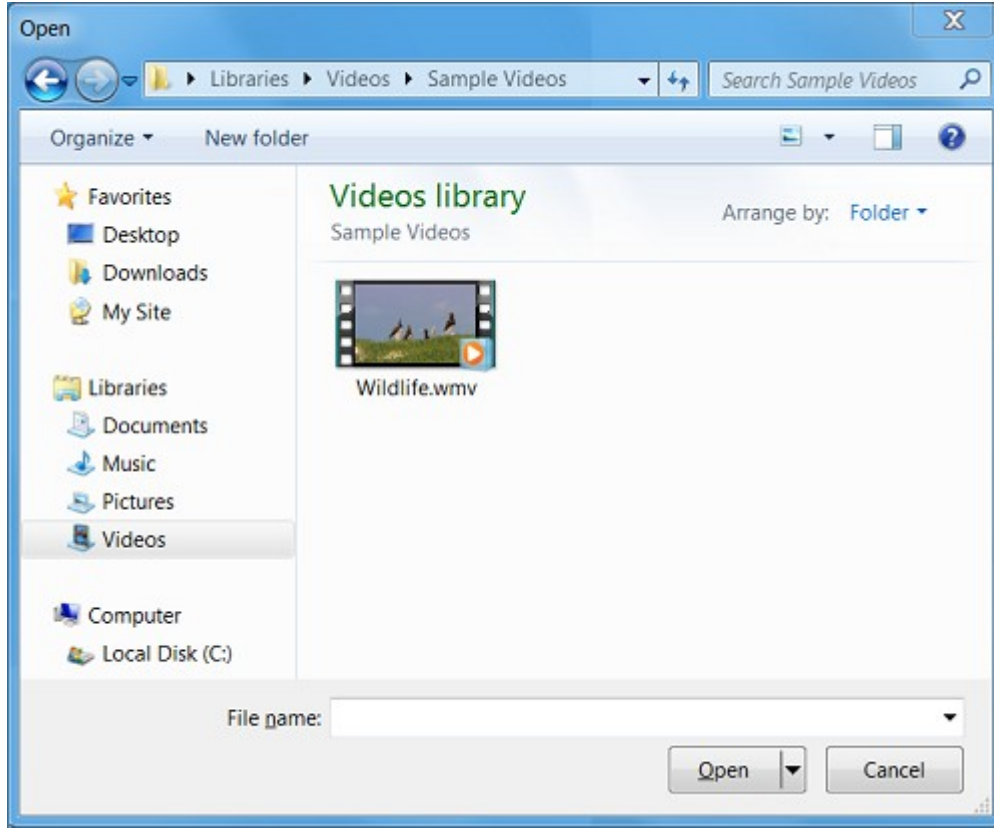
İletişim Kutusu Açma Örneği

Makale - 10/03/2019 - 2 dakika okumak için

Bu örnek uygulama, Bileşen Nesne Modeli (COM) kitaplığının nasıl başlatılacağını ve bir Windows programında COM tabanlı bir API'nin nasıl kullanılacağını gösterir.

Açıklama

İletişim Kutusunu Aç örnek uygulaması, aşağıdaki ekran görüntüsünde gösterildiği gibi **Aç** iletişim kutusunu görüntüler. Örnek, bir Windows programında bir COM nesnesinin nasıl çağrılacağını gösterir. Bu örnek [Modül 2: Windows Programınızda COM Kullanımı bölümünde](#) ele alınmaktadır.



Numunenin İndirilmesi

Bu örneğe [buradan](#) ulaşabilirsiniz .

İlgili konular

- [Örnek: Aç İletişim Kutusu](#)
- [Windows için Programlamayı Öğrenin: Örnek Kod](#)
- [Modül 2: Windows Programınızda COM Kullanımı](#)

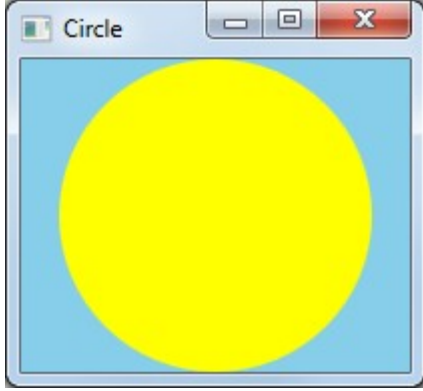
Direct2D Daire Örneđi

Makale - 10/03/2019 - 2 dakika okumak için

Bu örnek uygulama Direct2D kullanarak bir dairenin nasıl çizileceđini göstermektedir.

Açıklama

Direct2D Circle örnek uygulaması, aşağıdaki ekran görüntüsünde gösterildiđi gibi bir daire çizer. Bu örnek [Modül 3: Windows Grafikleri](#) bölümünde ele alınmıştır.



Numunenin İndirilmesi

Bu örneđe [buradan](#) ulaşabilirsiniz .

İlgili konular

- [Windows için Programlamayı Öğrenin: Örnek](#)
- [Kod İlk Direct2D Programı](#)
- [Modül 3: Windows Grafikleri](#)

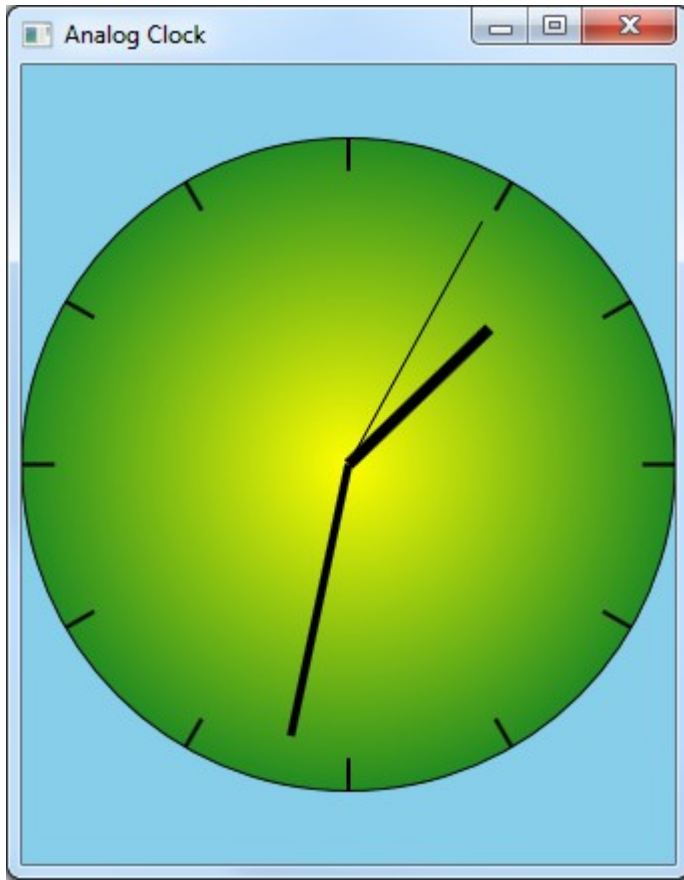
Direct2D Saat Örneđi

Makale - 10/03/2019 - 2 dakika okumak için

Bu örnek uygulama, bir saatin ibrelerini çizmek için Direct2D'de dönüşümlerin nasıl kullanılacağını gösterir.

Açıklama

Direct2D Clock örnek uygulaması, aşağıdaki ekran görüntüsünde gösterildiđi gibi bir analog saat çizer. Bu örnek, [Direct2D'de Dönüşümleri Uygulama](#) bölümünde ele alınmıştır.



Numunenin İndirilmesi

Bu örneđe [buradan](#) ulaşabilirsiniz .

İlgili konular

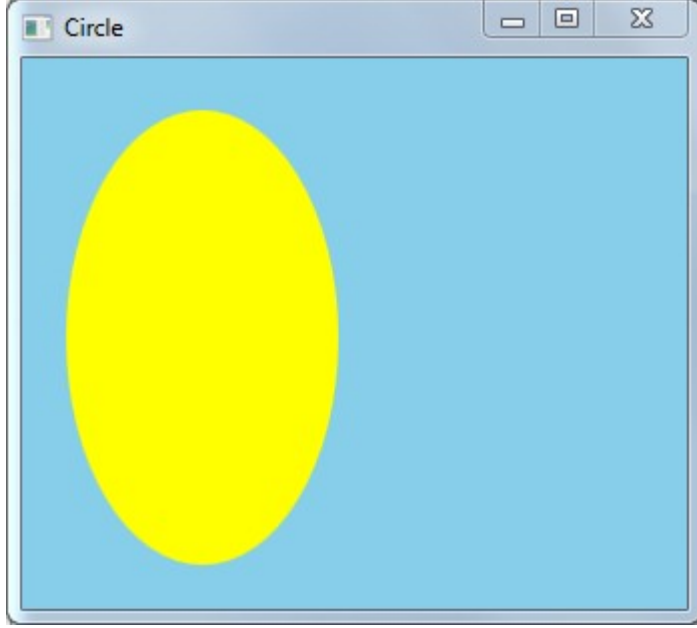
[Windows için Programlamayı Öğrenin: Direct2D'de](#)

[Dönüşümleri Uygulayan Örnek Kod](#)

Daire Örneđi Çizin

Makale - 03/30/2020 - Okumak için 2 dakika

Bu örnek uygulama, bir daire çizmek için fare girişinin nasıl kullanılacağını gösterir.



Numunenin İndirilmesi

Bu örneđe [buradan](#) ulaşabilirsiniz .

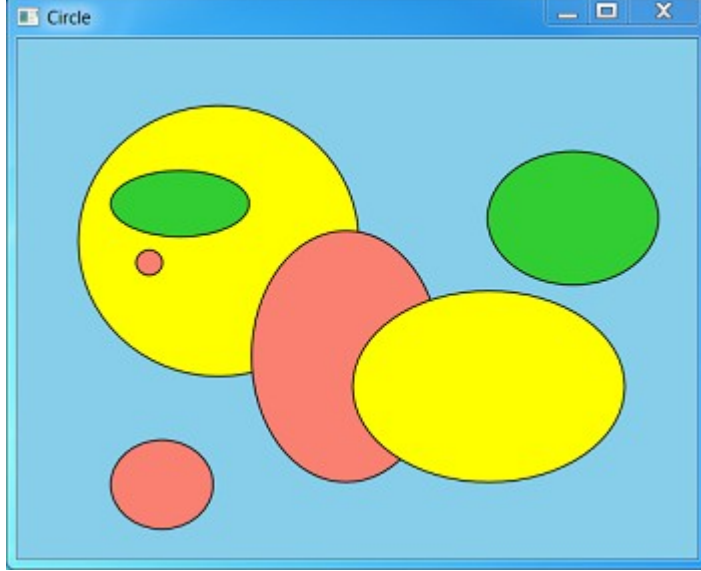
İlgili konular

- [Windows için Programlamayı Öğrenin: Örnek Kod](#)
- [Modül 4. Kullanıcı Giriş](#)

Basit Çizim Örneđi

Makale - 10/03/2019 - 2 dakika okumak için

Bu örnek uygulama, fare giriři, klavye giriři ve hızlandırıcı tablolarının nasıl kullanılacağını gösteren çok basit bir çizim programıdır.



Numunenin İndirilmesi

Bu örneđe [buradan](#) ulaşabilirsiniz .

İlgili konular

- [Windows için Programlamayı Öğrenin: Örnek Kod](#)
- [Modül 4. Kullanıcı Giriři](#)
- [Kullanıcı Giriři: Geniřletilmiş Örnek](#)

