No i znowu razem :), witam, witam. Pewnie chcielibyście wiedzieć co dzisiaj. Już spieszę z odpowiedzią na to jakże ważne pytanie. Dzisiaj będzie trochę o rysowaniu (dowiemy się miedzy innymi co to jest GDI, kontekst urządzenia, omówimy komunikat WM_PAINT. Jak zwykle zaczynamy, bo nie lubię ściemniania (ale kto lubi)?

GDI

Wielu z was pewnie wiele razy słyszało już o czymś takim jak GDI i zadawało sobie odwieczne pytanie co to jest za diabeł ?. Teraz właśnie powiem co to za diabeł i mam nadzieje, ze utrwali wam się to na tak długo, jak długo będzie istniał Windows :). GDI (czyli Graphics Device Interface), to w wolnym tłumaczeniu "Interfejs urządzenia graficznego", a na chłopski (nie obrażając kobiet :), rozum biorąc jest to cos co pozwala nam rysować w Windows. Mówiąc "rysowanie" mamy na myśli nie tylko to pojęte przez większość rysowanie na ekranie w oknie, czy gdzie tam wam się podoba. Rysowanie to równie dobrze może być drukowanie na drukarce. Zapamiętujemy od tego momentu raz na zawsze ! Czy rysujemy na oknie czy drukujemy na drukarce dla Windows jest to wszystko jedno. A dlaczego się tak dzieje ? Przy projektowaniu Windows panowie z Microsoft'u myśleli tak bardzo, ze aż niektórym robiło się momentami za gorąco, aż w końcu wymyślili, ze "ulepsza" Windows. Skutki tego ulepszania widać dzisiaj, gdy przychodzi nam złożyć do kupy komputer najnowszej generacji z PIII i innymi bajerami XXI - go wieku na pokładzie. W połowie przypadków na pewno nie uda się za pierwszym razem uruchomić tego jak należy. Ulepszenie Windows miało polegać na tym, aby nam programistom piszącym aplikacje dla Windows umilić i tak już skomplikowane życie. Po to stworzono właśnie cos takiego jak GDI. Jeśli dzisiaj piszemy banalna aplikacje rysującą na oknie trójwymiarowa animacje z renderingiem raytrace'owym, to w każdym przypadku, niezależnie od posiadanej karty graficznej wywołujemy zawsze te same funkcje, funkcje GDI. Tu właśnie tkwi cala siła a zarazem słabość tego interfejsu. Dzięki niemu nie musimy się martwic o to jaka mamy kartę grafiki. Karta taka, dzięki swoim sterownikom, przeznaczonym oczywiście dla odpowiedniej wersji Windows wykonuje pośrednio nasze polecenia właśnie za pomocą GDI. Jest to interfejs, czyli... hmm, jakby tu powiedzieć....o wiem, POMOST, tak to dobre słowo :), pomost pomiędzy nasza karta grafiki a nami - rysownikami. Taki interfejs udostępnia nam mnóstwo funkcji, którymi możemy dysponować wedle własnego uznania. Dzięki GDI funkcje te są zawsze te same, bowiem ich dostarcza system operacyjny, co jest chyba oczywiste, są to oczywiście funkcje API. A jak te funkcje są wykonywane przez nasz a kartę grafiki to już słodka tajemnica interfejsu GDI i sterownika karty graficznej, nas to w ogóle nie obchodzi, bo i nie powinno. Słabość tego rozwiązania to oczywiście jego powolność, ale tak było od zawsze i tak zawsze będzie. Im cos jest bardziej uniwersalne tym jest wolniejsze. Ale tak mówimy o tym GDI i mówimy, ale ktoś w końcu zada pytanie, co to jest to GDI ? Otóż GDI to (tu znowu brakuje słów :) biorąc najprościej jak można to zestaw funkcji, które umożliwiają:

- zarządzanie kontekstem urządzenia (jego tworzeniem, zwalnianiem, pozyskiwaniem o nim informacji i ustawianiem jego atrybutów),
- rysowanie (wyświetlanie tekstów, linii, prostokątów, bitmap itp.),
- określanie cech obiektów rysujących.

Funkcje GDI umożliwiają nam tworzenie grafiki raczej statycznej (właśnie okien, menu itp.) niż animacji. Do tego celu posłuży nam poznany w późniejszym terminie właśnie pakiet DirectX i OpenGL, które dają zacznie większe możliwości niż wspomniane funkcje GDI. Ale dobrze będzie znać choć podstawy, gdyż niektóre z funkcji możemy z powodzeniem wykorzystywać, niektóre nawet musimy przy użyciu tych bibliotek.

Kontekst urządzenia

Gdy chcemy cos narysować na ekranie komputera lub wydrukować cos na drukarce musimy najpierw dobrać się do naszego urządzenia wyjściowego. Do tego właśnie służy kontekst urządzenia. Wielu z was pewnie w tym momencie zadaje sobie pytanie co to takiego ten kontekst urządzenia. Owo "cos" to właściwie struktura danych przechowywana przez GDI w której zawarte są właściwości danego urządzenia. W przypadku ekranu kontekst urządzenia jest kojarzony z każdym oknem osobno, innymi słowy każde okno ma swój kontekst W strukturze takiej przechowywane są różne informacje o atrybutach graficznych takich jak bieżący kolor rysowania, grubość linii lub jej styl, rodzaj czcionki itp. Właściwie potrzebujemy nie samego kontekstu ale jego uchwytu. Mając uchwyt do kontekstu urządzenia możemy używać funkcji GDI. Większość (jeśli nie wszystkie) funkcje GDI potrzebują tego uchwytu aby moc poprawnie pracować. Gdy uzyskujemy uchwyt kontekstu urządzenia struktura powyższa jest wypełniana odpowiednimi wartościami, które oczywiście możemy pobierać i zmieniać wykorzystując do tego odpowiednie funkcje GDI. Po narysowaniu czegokolwiek powinniśmy zwolnić kontekst, którego używaliśmy. Gdy zwolnimy taki uchwyt to nie możemy go już dalej używać, wiec jeśli najdzie nas nagle ochota jeszcze raz cos narysować musimy go znowu pobrać, zrobić swoje, zwolnić i tak w kolko. Jak otrzymać uchwyt urządzenia ? Otóż już przechodzimy do sedna sprawy. Musimy oczywiście do tego celu wykorzystać funkcje GDI. Można to zrobić na kilka sposobów, ale podam ten, który potem przyda nam się najbardziej. Jeśli dobrze przeglądnęliście podręczniki pomocy to już pewnie wiecie co to za funkcja, natomiast tym, który się nie chciało pragnę powiedzieć o funkcji o jakże pięknie brzmiącej nazwie: **GetDC()**. Funkcja ta jako swój argument pobiera uchwyt okna, dla którego chcemy uzyskać jakże pożądany przez nas uchwyt kontekstu urządzenia. Uzyskujemy uchwyt kontekstu obszaru roboczego okna (roboczy obszar to cale okno bez belki tytułowe i ramek) i w tym momencie możemy już uruchomić niezgłębione pokłady naszej wyobraźni w celu wygenerowania pięknej animacji. Zanim jednak to nastąpi powiem jeszcze kilka słów o kontekście. Jak powiedziałem wcześniej po jego pobraniu i wykorzystaniu (brzmi trochę nieładnie, ale cóż...) powinniśmy go zwolnić, a służy do tego funkcja **ReleaseDC()**, która tym razem pobiera dwa argumenty, jeden z nich to uchwyt naszego kontekstu a drugi to uchwyt okna, z którym jest związany nasz kontekst. Funkcje te powinny zawsze być wywoływane parami. Jeśli kogoś interesuje to

istnieje taka funkcja jak **GetWindowDC()**, która tez zwraca nam kontekst urządzenia dla danego okna, ale tym razem możemy już malować na całym oknie, włącznie z belka tytułową itd. ale wymaga to obsługi dodatkowego komunikatu (jak kogoś interesuje, niech napisze), adres poniżej. Oczywiście zwalnianie tego uchwytu odbywa się za pomocą funkcji **ReleaseDC()**. Pamiętajmy wiec, mamy w sumie trzy funkcje:

```
HDC GetDC( HWND hWnd );
HDC GetWindowDC( HWND hWnd );
int ReleaseDC( HWND hWnd, HDC hDC );
```

Dobrze, skoro wiemy już (mniej więcej) co to GDI, kontekst urządzenia, możemy przystąpić do tak długo wyczekiwanego momentu jak narysowanie pierwszej linii w naszym oknie :). Zaczynajmy wiec !

Rysowanie

Funkcji GDI jest cale mnóstwo, my przyjrzymy się jednak tylko kilku. Biblioteki takie jak DirectX czy OpenGL dysponują własnymi funkcjami rysującymi, które są szybsze i bardziej uniwersalne wiec nie będziemy się wgłębiać w GDI, bo nie ma to większego sensu. Jeśli chcecie poznać więcej szczegółów to napiszcie o czym chcielibyście przeczytać a postaram się to zrobić w miarę możliwości.

Jeśli chcemy malować w Windows musimy wiedzieć jeszcze co nieco o układzie współrzędnych tam panującym. Standardowo os X to os pozioma liczona od lewej do prawej, natomiast os Y to os pionowa liczona z góry na dol. Nie ma oczywiście pikseli o współrzędnych ujemnych ! Malować można od zera do maksymalnego zakresu liczby int (w większości przypadków), jednak zazdroszczę temu kto potrafi wycisnąć ze swojego sprzętu taka rozdzielczość :). Orientacje układu współrzędnych można oczywiście zmieniać, ale dla nas nie ma to większego znaczenia bowiem jak wszystko inne, nasze ukochane biblioteki maja własne układy współrzędnych a co najważniejsze są to układy w trzech wymiarach !!! Dobrze, dobrze, nie wybiegajmy aż tak w przyszłość, zajmijmy się na razie starym poczciwym GDI. Załóżmy wiec, ze naszła nas ochota na narysowanie np. w oknie prostokąta. Co zrobić ? Nic prostszego: pobrać uchwyt kontekstu urządzenia, wywołać jedną funkcje GDI, zwolnić uchwyt i gotowe, prawda, ze proste ? No wiec jaka funkcja ? Nie może być chyba nic innego jak:

```
BOOL Rectangle( HDC hDC, int nLeft, int nTop, int nRight, int nBottom );
```

Dokładny opis jak zwykle, w help-ie :), my tylko napomknijmy, ze do narysowania prostokąta wystarcza nam jedynie dwa punkty, lewy górny i prawy dolny i w takiej tez kolejności podajemy współrzędne dla naszej funkcji. Po wywołaniu takiej funkcji powinniśmy otrzymać prostokąt o podanych współrzędnych (liczonych oczywiście w stosunku do współrzędnych okna ! a nie ekranu. Do innych tego typu funkcji możemy zaliczyć miedzy innymi:

```
BOOL RoundRect( HDC hDC, int nLeft, int nTop, int nRight, int nBottom, nWidth, nHeight );
BOOL Ellipse( HDC hDC, int nLeft, int nTop, int nRight, int nHeight );
BOOL Arc( HDC hDC, int nLeft, int nTop, int nRight, int nBottom, int nStartX, int nStartY,
int nEndX, int nEndY );
```

Komunikat WM_PAINT

Jedno co można powiedzieć to nie ma rysowania w Windows bez komunikatu **WM_PAINT** (no prawie, ale to już inna bajka :). Wszystko co chcemy narysować, trzeba wywołać bezpośrednio (lub pośrednio, za pomocą funkcji) w obsłudze tego komunikatu. Windows wysyła komunikat **WM_PAINT** do aplikacji za każdym razem, gdy okno wymaga odmalowania. Dzieje się tak oczywiście przy starcie aplikacji, po zmianie rozmiarów jej okna, po przykryciu jej przez inne okno i przywróceniu jej z powrotem, po zminimalizowaniu okna i jego ponownym zmaksymalizowaniu itp. Jeśli chcemy wywołać na siłę komunikat **WM_PAINT** to możemy to zrobić oczywiście na kilka sposobów. Pierwszy to wysłanie do aplikacji naszego komunikatu jawnie (**SendMessage()** lub **PostMessage()**), jednak nie używajmy tego brzydkiego nie eleganckiego sposobu. Lepszym będzie wywołanie metody **InvalidateRect()**. Co robi taka metoda ? Już wyjaśniam. Otóż: powoduje ona dodanie pewnego prostokąta do regionu okna, wymagającego uaktualnienia. Co do czego ? Region uaktualnienia mówi o tym jaka cześć okna ma zostać odrysowana. Wbrew pozorom nie musi to być cały obszar klienta, co nas powinno bardzo cieszyć. Po wywołaniu obsługi komunikatu **WM_PAINT** zostanie odmalowany taki prostokąt, jaki mamy aktualnie zawarty w obszarze uaktualniania, przeważnie całe okno. Funkcja **InvalidateRect()** powoduje wywołanie komunikatu **WM_PAINT** a co za tym idzie odmalowanie obszaru unieważnionego. Po namalowaniu czegokolwiek powinniśmy zatwierdzić obszar funkcją **ValidateRect()**.

Bitmapy

Działanie całego systemu Windows nie może się obejść bez czegoś takiego jak bitmapa. W Windows jest to pojecie trochę blednie używane, ponieważ z definicji bitmapa jest jest obrazem biało-czarnym (jeden bit to jeden piksel), ale przyjęło się tak a nie inaczej i już zostało. W zasadzie kolorowe bitmapy to pixmapy a nie bitmapy, ale nie bądźmy drobiazgowi, skoro już mamy jaki taki standard w nazewnictwie to niech już tak zostanie :). Oprócz funkcji rysujących możemy oczywiście użyć do malowania w Windows bitmap. Niech ktoś spróbuje namalować swoja podobiznę funkcjami typu "line" czy "rectangle" :). Bitmapy używamy zawsze w ten sam sposób, ale możemy ją pobierać do naszych celów w dwojaki sposób. Pierwszy to taki,

ze dołączamy nasza bitmapę do naszego pliku zasobów, czego ja jednak nie pochwalam, no bo rośnie nam niepotrzebnie plik exe, jak spróbujemy sobie włożyć do naszego programu bitmapę 800x600x32 bity to nasz exec urośnie prawie jak programy microsoftu a na tym nam chyba nie zależy. Ja wolę lądować je z dysku, z określonego katalogu, ale każdy może robić jak mu wygodnie.

Wiec co zrobić jeśli mamy już bitmapę ze zdjęciem naszej dziewczyny ;) i pragniemy umieścić ja w naszym oknie ? Pierwsze co musimy zrobić to załadować nasza bitmapę do pamięci. Robi to funkcja **LoadBitmap()**. Ale zanim załadujemy bitmapę do pamięci musimy tam stworzyć kontekst urządzenia, kompatybilny z naszym oknem. Służy do tego funkcja **CreateCompatibleDC()**. Funkcja ta pobiera uchwyt aktualnego kontekstu urządzenia i tworzy w pamięci taki sam kontekst z identycznymi atrybutami jak nasze urządzenie wyświetlające. Po co nam to ? Otóż do wyświetlania naszej bitmapy będziemy używać funkcji **BitBlt()** (bit block transfer), która kopiuje (w zasadzie transferuje, ale mniejsza o to :) bajty z pamięci do pamięci. Funkcja ta przenosi bajty z jednego kontekstu urządzenia do drugiego, wykonując po drodze dodatkowe operacje (jeśli sobie tego zażyczymy oczywiście). Tak wiec aby móc wyświetlać bitmapę musimy posiadać kontekst, którego będziemy kopiować i kontekst na który będziemy kopiować (ten mamy z funkcji **GetDC()** lub **GetWindowDC()**). Aby nie było żadnych niespodzianek (he, he, nadzieja matka głupich :) tworzymy właśnie kompatybilny kontekst urządzenia z naszym posiadanym wyżej wymieniona funkcja. Jeśli już mamy uchwyt do kontekstu kompatybilnego, możemy przystąpić do ładowania bitmapy z pliku. Wywołajmy wiec długo oczekiwana funkcje **LoadBitmap()**.

Po jej wywołaniu otrzymamy uchwyt do naszej bitmapy (typ **HBITMAP**) i jest to nic innego jak nadal numer (jak każdy uchwyt, jeśli dobrze pamiętamy). I tu w zależności od tego czy mamy bitmapę w pliku zasobów czy na dysku robimy tak:

```
hBitmap = (HBITMAP)LoadImage( GetModuleHandle( NULL ), name, IMAGE_BITMAP, 0, 0,
LR_CREATEDIBSECTION );
```

natomiast jeśli chcemy ja załadować z dysku to tak:

```
hBitmap = (HBITMAP)LoadImage( NULL, name, IMAGE_BITMAP, 0, 0, LR_LOADFROMFILE |
LR_CREATEDIBSECTION );
```

Jeśli już mamy uchwyt bitmapy to możemy przystąpić do jej wyświetlenia. Zanim jednak ujrzymy nasza długo wyczekiwana kobietę na ekranie monitora musimy jeszcze wyselekcjonować jakiego obiektu pragniemy użyć z kontekstu kompatybilnego, który wcześniej utworzyliśmy w pamięci. Robimy to funkcja **SelectObject()**. Pierwszy argument to adres kompatybilnego kontekstu, utworzonego przez nas, drugi argument to uchwyt bitmapy, która jest umieszczona w tym kontekście. Jeśli już mamy wyselekcjowaną nasza bitmapę to nie pozostaje nam nic innego jak zrobić:

`BitBlt( HDC, INT, INT, INT, INT, HDC, INT, INT, DWORD )` - dokładny opis można znaleźć w help-ie oczywiście. Pierwszy **HDC** to ten na który kopiujemy, czyli kontekst naszego okna, drugi to tez z którego kopiujemy (kompatybilny w naszym przypadku). Pierwsze cztery liczby to współrzędne obrazka, jakie będzie zajmował po skopiowaniu go do okna (możemy go np. rozciągnąć, ścisnąć, obrócić itp.), drugie to lewy górny róg obrazu źródłowego. Nader ciekawy jest ostatni parametr, który określa tzw. kod operacji ROP (raster operation code1). Mówi on o tym w jaki sposób piksele obrazu źródłowego zostaną połączone z pikselami obrazu docelowego. I tu mamy multum możliwości. Możemy po prostu skopiować nasze piksele w miejsce starych (najczęściej), wykonać jakieś operacje (dodawanie, mnożenie itp.) ściemnić obraz lub rozjaśnić i kilka innych, po szczegóły oczywiście należy się udać do nieocenionego helpa.

No to skoro już wiecie z czym to się wszystko je możecie spróbować napisać prościutką animacyjke, a może jeszcze cos więcej. Oczywiście nie będzie to jeszcze żaden 3D ale zawsze to coś. W przyszłym odcinku powiemy sobie cos nie cos o timerze no i powoli zaczniemy zagłębiać się w istotę grafiki 3D.