

4

WINDOWS GDI

Jeden obraz wart jest tysiąc słów.
znane przysłowie

Już na początku kursu WinAPI wyjaśniłem, że będziemy zajmować się programowaniem graficznego interfejsu użytkownika. Widoczne na ekranie monitora elementy GUI są zaś niczym innym jak odpowiednio spreparowanymi obrazkami. Windows potrafi zrobić całkiem sporo, jeżeli chodzi o manipulacje obrazem, jednak na co dzień nie pokazuje zwykle całej swojej mocy. Praca modułu graficznego ogranicza się do wyświetlania okien, przycisków, menu, list, pól tekstowych i innych kontroltek. A przecież potrafi on znacznie więcej.

Dlatego też w tym rozdziale zajmiemy się biblioteką graficzną, jaka jest wbudowana w Windows. Chodzi o tytułowe Windows GDI.

„Chwileczkę”, możesz powiedzieć, „Mieliśmy przecież zająć się inną biblioteką graficzną, jaką jest DirectX. Co z nią?...” Ha, jesteś uważny - to dobrze. Rzeczywiście, GDI można przyrównać do DirectX nie tylko dlatego, że oba podsystemy mają podobną rolę do spełnienia, ale też względu na zbliżone możliwości. W grach będziemy używać głównie DirectX, lecz niejednokrotnie przydatne będą zaawansowane operacje na grafice dwuwymiarowej, których brakuje temu interfejsowi. Programowe generowanie tekstur, wypisywanie formatowanego tekstu czy tworzenie własnego systemu GUI - to tylko niektóre kwestie, przy których bardzo pomocna, wręcz nieodzowna, staje się znajomość biblioteki Windows GDI.

Co jeszcze przemawia za uważnym przyjrzeniem się tej części WinAPI? Chociażby sam fakt, z czym mamy do czynienia. Skoro niedługo będziemy na codzień używać skomplikowanego i potężnego instrumentarium graficznego DirectX, warto byłoby zaznajomić się wpierw z jego młodszym bratem. Wiele koncepcji, terminów, pojęć, sposobów, a nawet nazw funkcji będzie powtarzało się w identycznej lub zbliżonej formie w obu bibliotekach. Jeśli więc teraz poznamy je podczas nauki stosunkowo prostego narzędzia, jakim jest GDI, łatwiej będzie nam przenieść je potem na grunt bardziej skomplikowanego DirectX'a.

Pomyślmy wreszcie, że oto będziemy przecież zajmować się prawdziwą Grafiką przez duże G :) Jaka to miła odmiana po godzinach spędzonych przed siemiężnym ekranem konsoli czy też na kontakcie z mało zachwycającym, standardowym GUI systemu Windows. Teraz będziemy mogli odetchnąć i zająć się bardziej przyjemnymi zagadnieniami, a przy tym dosłownie **zobaczyć** nasze programy w akcji!

Myślę więc, że nauka obsługi Windows GDI będzie dla ciebie całkiem przyjemnym (a przynajmniej znośnym ;D) zajęciem.

Słówko o grafice komputerowej

Dotychczas w zasadzie nie mogliśmy powiedzieć, że zajmujemy się grafiką komputerową. Teraz właśnie przyszedł czas na pierwsze spotkanie nią i dlatego musimy sobie od razu wyjaśnić kilka spraw z tym związanych.

Powiemy więc sobie o dwóch podstawowych rodzajach grafiki, pikselach i kolorach. Przypatrzymy się też różnym typom urządzeń graficznych

Rodzaje grafiki

Tradycyjny podział grafiki oznacza wyróżnienie jej dwóch rodzajów: grafiki **rastrowej** oraz **wektorowej**. Różnica pomiędzy nimi polega na innej interpretacji obrazu oraz jego zapisie w pamięci operacyjnej i w pliku dyskowym.

Grafika rastrowa

Tryb rastrowy jest naturalnym i często jedynym sposobem pracy większości urządzeń graficznych. Zalicza się do nich na przykład monitor i drukarka.

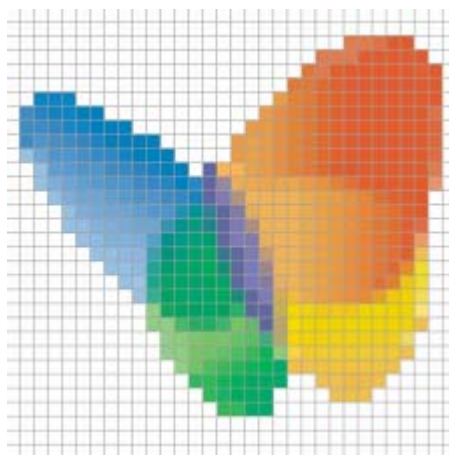
W **grafice rastrowej** (ang. *raster graphics*) obraz jest przedstawiany jako dwuwymiarowa tablica danych. Każdy jego mały fragment jest opisany przez określoną ilość informacji, czyli bitów. Od tego wzięła się też nazwa rysunków rastrowych - **bitmapy**.

To co oglądasz na ekranie monitora jest właśnie bitmapą.

Piksele

Najmniejszy element rastrowego obrazka nazywamy **pikselem**. Jest on najczęściej kwadratowy i wypełniony zawsze jednolitym kolorem. Oglądając ilustrację, zwykle nie widzimy jednak pojedynczych pikseli z bardzo prostego powodu: są one zbyt małe. Stają się widoczne dopiero przy dużych powiększeniach, tworząc niezbyt przyjemną dla oczu siatkę kwadratów.

Ażeby uniknąć takich niepożądanych efektów, bitmapa musi zawierać dostatecznie dużo pikseli. Inaczej mówiąc, musi ona posiadać odpowiednio dużą **rozdzielczość** (ang. *resolution*). Wielkość ta określa ilość pikseli w pionie i poziomie, tworzących siatkę obrazu; dla przykładu 300×200 oznacza, iż bitmapa ma szerokość 300 pikseli, a wysokość 200. Naturalnie, im wyższe są obie te wartości, tym lepsza jakość obrazu i wyższa jego „odporność na powiększanie”.



Rysunek 11. Obraz rastrowy w rozdzielczości 32×32

Ze wzrostem rozdzielczości związany jest wzrost liczby pikseli, a zatem zwiększenie liczby informacji opisujących. Sprawia to, że duże bitmapy zajmują wiele miejsca w komputerze - niekiedy są to nawet megabajty. Dlatego też wymyślono wiele formatów grafiki rastrowej, które oferują kompresję danych. Najczęściej odbywa się to kosztem jakości

obrazka, jest to więc kompresja stratna. Popularnym formatem wyposażonym w taką możliwość jest JPEG.

Zauważmy aczkolwiek, że takie sposoby zmniejszania rozmiaru obrazków dotyczą tylko ich przechowywania na dysku. Podczas obróbki bitmap muszą być one zapisane w pamięci operacyjnej w swej zwykłej postaci - oto przyczyna, dla której praca z grafiką wymaga dużej ilości RAMu.

Kolory

Na obiektywny rozmiar oraz subiektywną jakość rastrowej bitmapy wpływa jeszcze jeden ważny czynnik. Jest nim dokładność odzwierciedlenia rzeczywistych kolorów, ich odcieni oraz natężenia. Ta cecha obrazu jest najczęściej również proporcjonalna do jego wielkości: lepsze odzworowanie barw pociąga za sobą najczęściej większy rozmiar wynikowego pliku graficznego.

Dzieje się tak, gdyż bogatszy zbiór kolorów, udostępniający dużo odcieni barw, wymaga przechowywania większej ilości informacji dla pojedynczego piksela. Może się ona wahać od zaledwie jednego bitu do kilku bajtów.

Ważny jest również sposób, w jaki wartości zapisane dla każdego piksela przekładają się na rzeczywiste kolory, które możemy zobaczyć. Najprostszą drogą jest tutaj ustalenie pewnej stałej **palety barw** i przechowywanie w obrazie indeksów poszczególnych kolorów w tejże palecie jako zwyczajnych liczb. W zasadzie można powiedzieć, że wszystkie systemy zapisywania kolorów korzystają z tej metody. Jednak w każdym z nich wartość liczbową piksela może być także interpretowana na inny, bardziej swoisty sposób.

Przyjrzymy się teraz kilku takim systemom kodowania barw.

RGB

Akronim **RGB** pochodzi od nazw trzech kolorów podstawowych w tym systemie: **czerwonego** (ang. *red*), **zielonego** (ang. *green*) oraz **niebieskiego** (ang. *blue*).

Wszystkie inne barwy powstają poprzez odpowiednie zmieszanie tych trzech kolorów głównych.

Składowe barw podstawowych w wynikowym kolorze nazywamy **kanałami** (ang. *channels*). W systemie RGB mamy więc kanał czerwony, zielony i niebieski.

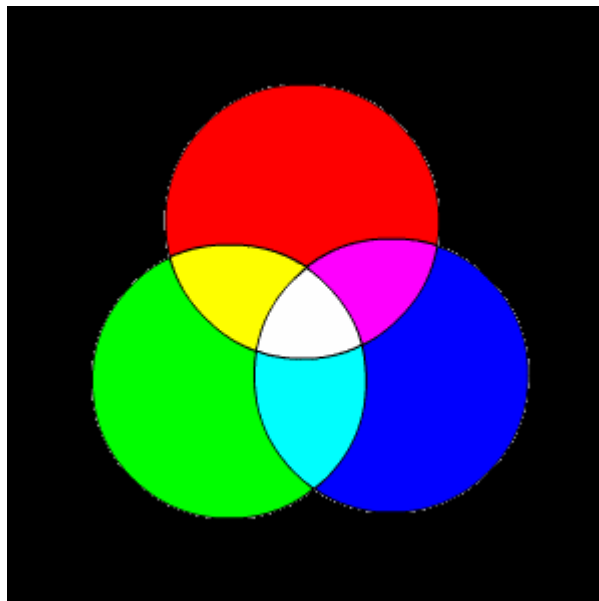
Trzeba jeszcze wiedzieć, jak odbywa się owo mieszanie. Otóż błędem jest sądzić, że działa ono na podobnej zasadzie jak łączenie farb na malarskiej paletce. RGB jest systemem używanym głównie do wyświetlania na ekranach monitorów i dlatego kryterium mieszania kolorów jest tu interferencja fal świetlnych, które padają na kineskop.

Spokojnie, nie oznacza to dla nas konieczności nauki praw fizyki decydujących o właściwościach światła (czyli optyki). Nie jest to nam potrzebne. Wystarczy tylko wiedzieć o dwóch stanach skrajnych:

- brak światła oznacza również brak koloru. Kolor czarny odpowiada więc takiej sytuacji, gdy wszystkie trzy składowe RGB są równe zeru
- największa intensywność światła w każdym z trójki kanałów oznacza natomiast kolor biały

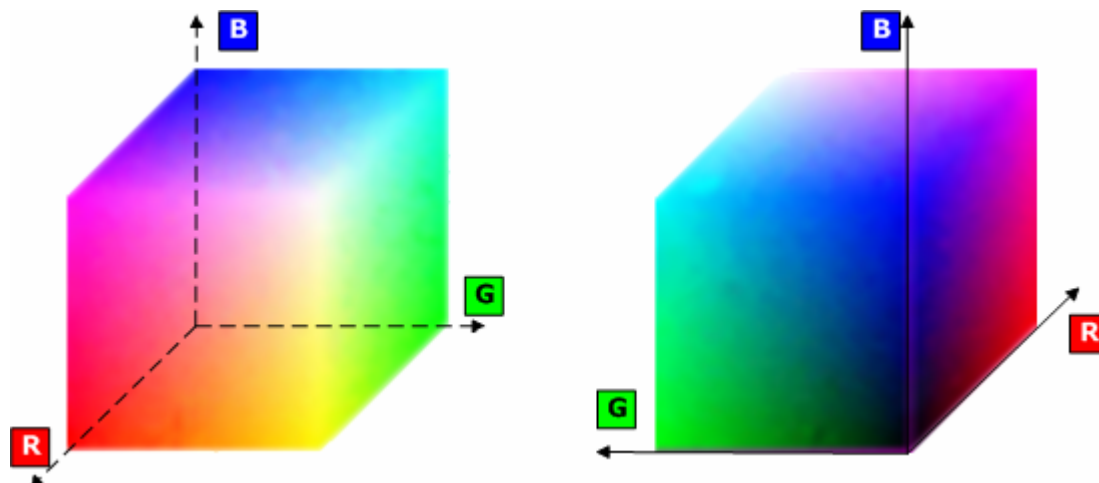
Z tego powodu system kolorów RGB nazywamy **addytywnym**. Większym wartościom składowych przyporządkowane są bowiem jaśniejsze kolory.

Wszystkie pozostałe barwy sytuują się gdzieś pomiędzy tymi dwoma krańcowymi kolorami. Jeśli każdej ze składowych odpowiada **taka sama intensywność koloru podstawowego**, wtedy mamy do czynienia z pewnym **odcieniem szarości** (ciemniejszym lub jaśniejszym). Inne kolory powstają przy różnych wartościach barw podstawowych w kanałach RGB.



Rysunek 12. Spektrum barw podstawowych RGB

Komputerowy zapis kolorów w systemie RGB odbywa się poprzez dobranie pewnych wartości liczbowych, które określają intensywność trzech składowych koloru. Zero oznacza zawsze brak danej składowej w finalnej barwie; drugi koniec skali zależy od dokładności odwzorowania kolorów, na jaką możemy sobie pozwolić. Im więcej wartości pośrednich zmieści się pomiędzy tymi skrajnymi, tym oczywiście większą liczbę kolorów będziemy mogli zapisać, a nasze obrazki będą miały lepszą jakość (i odpowiednio duży rozmiar).

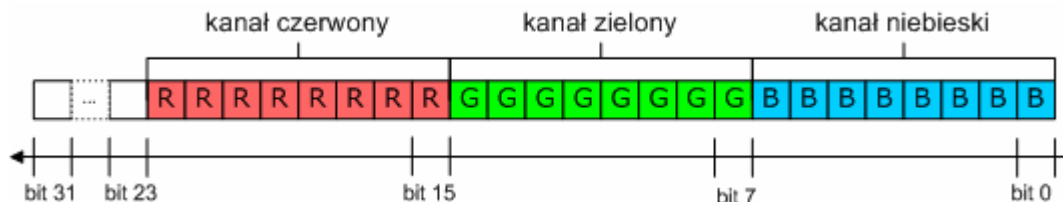


Rysunek 13. Przestrzeń barw RGB. Osie współrzędnych odpowiadają wartościom kolorów składowych

Obecnie większość aplikacji posługująca się systemem RGB (czyli większość aplikacji w ogóle :D) preferuje zapis każdego kanału w postaci liczby całkowitej bez znaku, pochodzącej z przedziału **od zera do 255**. Łącznie daje to więc 256^3 barw możliwych do reprezentacji - nieco ponad **16 milionów**, prawie tyle ile potrafi rozróżnić przeciętne ludzkie oko. Dlatego też tak bogaty zestaw kolorów nosi nazwę **True Color**, czyli 'rzeczywistych barw'.

Czy zakres $\langle 0; 255 \rangle$ nie wygląda znajomo?... Oczywiście, są to możliwości zapisu liczb w jednym bajcie - ośmiu bitach. Tryb *True Color* potrzebuje więc łącznie 3 bajtów (24 bitów) na zapisanie informacji o kolorze - taki kolor nazywamy więc **24-bitowym**.

W programowaniu nie ma jednak zmiennych zajmujących w pamięci dokładnie trójkę bajtów. Najmniejszym typem, który można by wykorzystać do przechowywania koloru, jest `DWORD` - liczba 32-bitowa. Tak też czynimy, zapisując do jej poszczególnych bajtów wartości kanałów RGB:



Schemat 46. Format XRGB zapisu koloru

Windows API posiada przygotowany typ dla zapisu kolorów: jest to `COLORREF`, będący niczym innym jak tylko kolejnym aliasem na 4-bajtowy numeryk.

O wiele bardziej przydatne są makra, ułatwiające pracę z takim sposobem reprezentacji koloru. Są one zadeklarowane w *windows.h*, a najważniejsze z nich to `RGB()`:

```
#define RGB(r, g, b) (COLORREF)((r) << 16 | ((g) << 8) | (b))
```

Tworzy ono identyfikator barwy z podanych mu składowych: czerwonej (*r*), zielonej (*g*) i niebieskiej (*b*). Jak widać, czyni to poprzez ich właściwe rozmieszczenie w dwusłowie, a następnie połączenie przy pomocy sumy bitowej `|`.

Odwrotnie do `RGB()` działają makra `GetRValue()`, `GetGValue()` i `GetBValue()`:

```
#define GetRValue(rgb) (BYTE)((rgb) >> 16)
#define GetGValue(rgb) (BYTE)((rgb) >> 8)
#define GetBValue(rgb) (BYTE)(rgb)
```

Wyławiają one kanały RGB, zwracając liczby określające intensywność barw podstawowych w podanym im kolorze. Robią to, dokonując operacji bitowych odwrotnych do tych z `RGB()`.

Uboższą wersją *True Color* jest *High Color* ('kolor wysokiej jakości'). Tryb ten używa tylko 16-bitów do zapisu informacji o kolorze, zatem mieści się w zmiennej typu `WORD`. Każdemu kanałowi jest tu przypisane po 5 bitów - z wyjątkiem składowej zielonej, która zajmuje 6 bitów. Jest tak dlatego, iż oko ludzkie jest przeciętnie najbardziej wyczulone na zmianę odcieni zieleni.

Format reprezentacji koloru, jaki przedstawiłem na ostatnim schemacie, nosi nazwę **XRGB**. Ta nazwa wskazuje na kolejność kanałów w gotowym dwusłowie. Litera X odnosi się natomiast do pierwszych ośmiu bitów, bowiem nie są one wykorzystywane do żadnych celów.

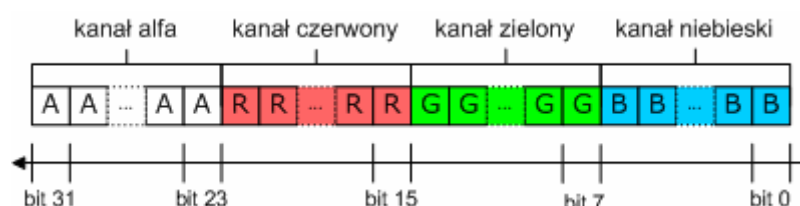
Takie marnotrawstwo trudno uznać za zadowalające rozwiązanie. Szybko więc wymyślono, co należy zapisywać w nadmiarowym bajcie. Stał się on w ten sposób **kanalem alfa** (ang. *alpha channel*), którego zadaniem jest przechowywanie informacji o **przezroczystości** danego piksela. Mówiąc ściślej, zawiera on wartość odwrotną do przezroczystości - coś w rodzaju „stopnia widoczności” punktu, zwanego po prostu **alfą**. Zero w kanale alfa oznacza, że ten fragment obrazu nie ma być w ogóle widoczny;

największa wartość 255 znaczy zaś, iż piksel ma całkowicie przykrywać te leżące pod nim.

Przydatność kanału alfa jest niemożliwa do zaobserwowania w przypadku pojedynczej bitmapy, ale otwiera bardzo ciekawe możliwości przy łączeniu dwóch obrazków w jeden. Działa wtedy mechanizm zwany **łączeniem** lub **mieszaniem alfa** (ang. *alpha blending*). Zmienia on kolory nakładających się pikseli tak, że ostatecznie mamy złudzenie częściowej lub całkowitej przezroczystości w wynikowym obrazie (z pikselami już bez kanału alfa). Bez *alpha blendingu* można uzyskać co najwyżej albo zupełne przykrycie, albo zupełne odkrycie spodnich pikseli.

Nie trzeba chyba dodawać, jak łączenie alfa jest przydatne, szczególnie w grach. Używając częściowej przezroczystości można chociażby stworzyć efektowny interfejs użytkownika, który nie zasłania całkowicie reszty ekranu gry.

Format koloru z określonym kanałem alfa określamy jako ARGB. Kolor można wówczas nazwać **32-bitowym**.



Schemat 47. Format ARGB zapisu koloru

CMY(K)

Drugim z najważniejszych systemów kodowania kolorów jest CMY. Skrót ponownie pochodzi od nazw barw podstawowych - tym razem są to kolory: **morski** (ang. *cyan*), **karmazynowy** (ang. *magenta*) oraz **żółty** (ang. *yellow*).

Na tym jednak nie kończą się różnice pomiędzy tym systemem a RGB. Osobną kwestią jest mianowicie sposób, w jaki składowe tych trzech podstawowych kolorów decydują o finalnej barwie; sposób ten jest odmienny niż w RGB. Jest uzasadnione, gdyż CMY został stworzony do współpracy przede wszystkim z drukarkami (oraz innymi urządzeniami, które tworzą swoją twórczość na papierze :D). Mieszanie barw składowych nie może więc już polegać na łączeniu fal świetlnych, ale barwników atramentu. Zasady tego łączenia są ci zapewne doskonale znane - przypomnijmy więc tylko, że:

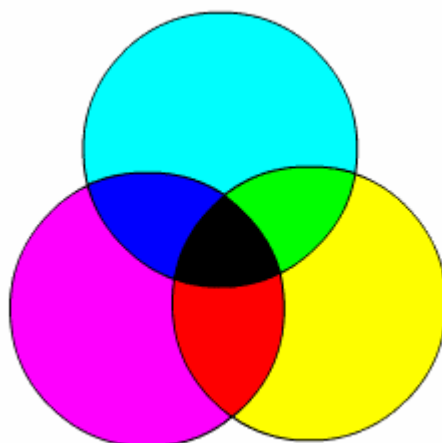
- brak barwnika jest brakiem koloru, a w przypadku systemu CMY oznacza to kolor „kartki”, czyli biały
- pełne nasycenie wszystkich trzech farb daje w wyniku kolor czarny

Mechanizm działa zatem dokładnie odwrotną¹³⁴ metodą niż ten z RGB.

W odróżnieniu od tego system CMY zwie się więc systemem **subtraktywnym**. Większej intensywności składowych odpowiadają tu ciemniejsze kolory wynikowe.

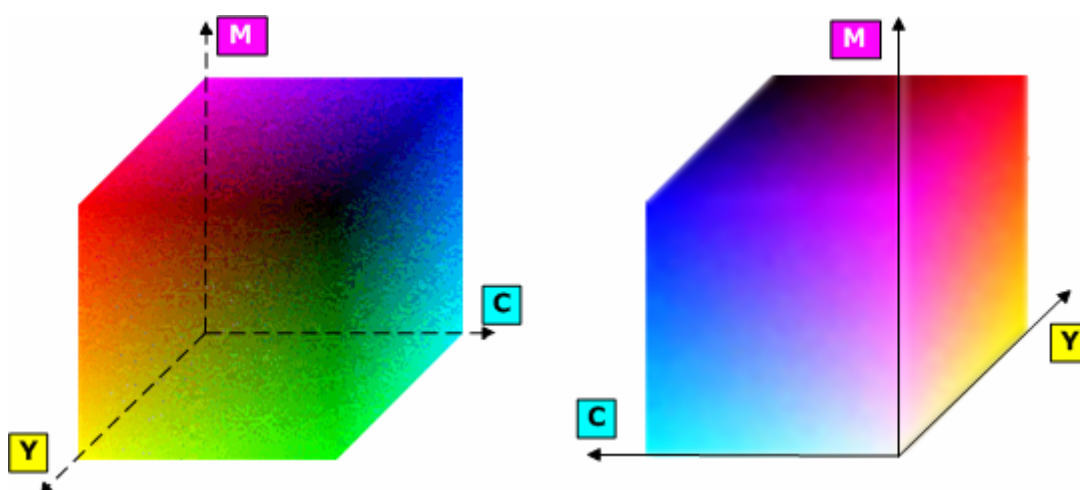
Kolory o takich samych wartościach składowych są w CMY również odcieniami szarości. Podobnie, różne intensywności barw podstawowych powodują powstanie pozostałych odcieni. Inaczej interpretowane są jedynie same wartości, zapisane w każdym z trzech kanałów - ich wzrost powoduje ściemnienie barwy, zaś spadek rozjaśnienie (przeciwnie niż to jest w systemie RGB).

¹³⁴ Lepiej powiedzieć - komplementarną.



Rysunek 14. Spektrum barw podstawowych CMY

Ewentualny zapis koloru w systemie CMY wygląda najczęściej tak samo, jak w RGB. Używana jest więc liczba 32-bitowa, z której efektywne wartości zawierają trzy dolne bajty.



Rysunek 15. Przestrzeń barw CMY

Nietrudno spostrzec, że sześciany barw CMY i RGB są do siebie bardzo podobne. Faktycznie, otrzymanie jednego z nich sprowadza się do obrotu drugiego o 90° i właściwego oznaczenia osi układu współrzędnych.

Standard CMY jest dobrym przykładem na to, iż dobra teoria może być daleka od rzeczywistej praktyki. Okazuje się, że zmieszanie maksymalnej intensywności trzech barwników podstawowych nie daje wcale koloru czarnego, lecz co najwyżej ciemnobrązowy. Aby rozwiązać ten problem, dodaje się jeszcze trochę czarnego atramentu. Tak oto powstał system CMYK - ostatnia litera pochodzi od nazwy koloru **czarnego** (ang. *black*).

W zasadzie więc to CMYK jest najszerszej używanym systemem kodowania kolorów dla drukarek. Do współpracy z tym formatem system Windows deleguje podobne makra, jakie przeznaczył do RGB. Najistotniejsze spośród nich to `CMYK()`:

```
#define CMYK(c, m, y, k) (COLORREF) (((c) << 24) | ((m) << 16) | ((y) << 8) | (k))
```

Układa ono wartości czterech składowych barwy poczynając od lewej strony dwusłowa COLORREF. Działanie odwrotne - wyłuskiwania wartości kanałów - jest zadaniem czterech makr: GetCValue(), GetMValue(), GetYValue() i GetKValue():

```
#define GetCValue(cmyk) (BYTE) ((cmyk) >> 24)
#define GetMValue(cmyk) (BYTE) ((cmyk) >> 16)
#define GetYValue(cmyk) (BYTE) ((cmyk) >> 8)
#define GetKValue(cmyk) (BYTE) (cmyk)
```

Robią one analogicznie to samo, co odpowiadające im makra z RGB - dokonują mianowicie odwrotnych operacji do tych z makra CMYK().

Jeszcze jednym używanym szeroko standardem zapisu barw jest HSB albo HSV. Skrót te pochodzą on cech koloru, które go wyznaczają: odcienia (ang. *hue*), nasycenia (ang. *saturation*) oraz jasności (ang. *brightness*), zwanej też walorem (ang. *value*). Odcień jest tym, co funkcjonuje potocznie pod nazwą 'koloru' - można go utożsamiać z długością fali świetlnej. Nasycenie określa, jak „czysta” jest barwa, tzn. ile koloru szarego, białego lub czarnego zawiera w sobie. Jasność (walor) odpowiada intensywności światła koloru.

Ze względu na to, że w systemie HSV (HSB) liczą się faktyczne własności koloru, a nie produkt mieszania barw podstawowych, jest on używany w wielu zaawansowanych programach do grafiki rastrowej. Poza tym jednak nie ma większego zastosowania, gdyż ostatecznie i tak musi zostać przeliczony na RGB, aby kolor mógł być wyświetlony na ekranie.

Grafika wektorowa

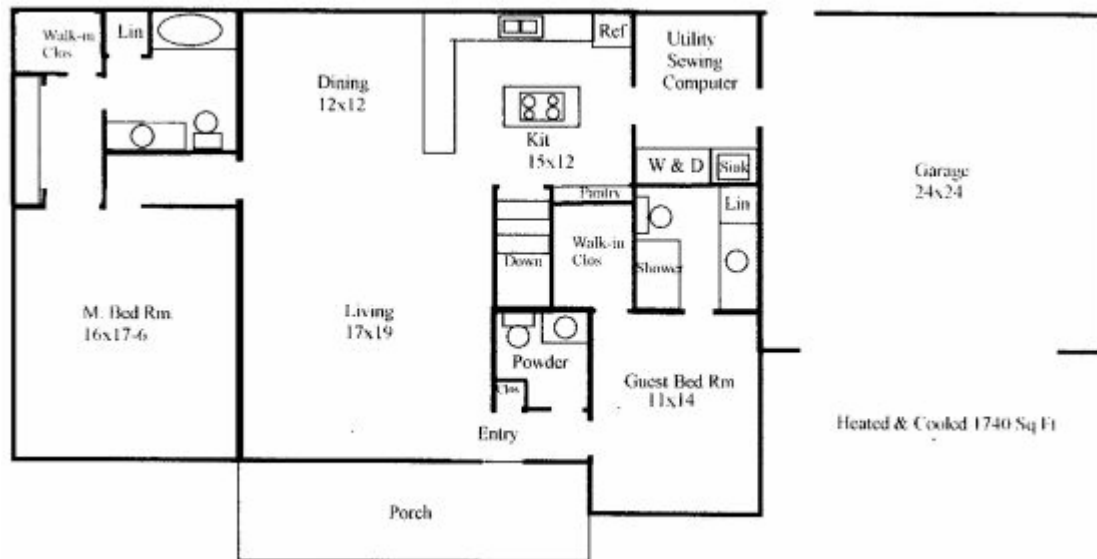
Zupełnie inne podejście do komputerowych rysunków prezentowane jest w **grafice wektorowej** (ang. *vector graphics*).

Nie ma tu pojęcia obrazu jako zbioru punktowych elementów - pikseli. Zamiast tego używany jest **geometryczny opis** tego, co można na nim zobaczyć. Obraz wektorowy składa się z linii prostych, otwartych i zamkniętych krzywych, figur geometrycznych i innych obiektów, które można opisać równaniami matematycznymi (na przykład tekstu).

Niekwestionowaną zaletą takiego potraktowania jest możliwość **dowolnego skalowania** rysunku wektorowego. Ponieważ zapisywana jest jedynie informacja o tym, jak wygenerować obraz, jego wygląd może zostać wyliczony przy każdym powiększeniu bez najmniejszej utraty ostrości.

Geometryczny sposób opisu ogranicza jednak zastosowanie grafiki wektorowej. Na pewno nie może być ona wykorzystywana do zapisu zdjęć, gdyż rzeczywisty świat jest zbyt skomplikowany, by móc go opisać matematycznie. Takie przedstawienie musiałoby zresztą wymagać konwersji rysunku rastrowego na wektorowy, a to nie jest możliwe ze względu na brak dostatecznych informacji w gotowej bitmapie.

Rysunki wektorowe nie powstają więc z fotograficznego odwzorowania rzeczywistości, lecz są tworzone przy pomocy odpowiednich narzędzi - programów. Są to często szkice techniczne, zawierające zgeometryzowane obiekty: prostokąty, linie różnej grubości, krzywe Béziera itp. Do takich zastosowań grafika wektorowa nadaje się wyśmienicie, bo zapewnia precyzję i ścisłość, której brak pikselowatym obrazom rastrowym. Do najważniejszych aplikacji do tworzenia rysunków wektorowych należą zapewne programy CAD, czyli narzędzia komputerowego wspomaganie projektowania.



Rysunek 16. Wektorowy szkic projektu budynku mieszkalnego
(rysunek pochodzi z [serwisu internetowego programu Embedded Vector Editor](#))

Grafika 3D

Bardzo ważnym typem grafiki wektorowej są **sceny trójwymiarowe** (ang. *3D scenes*). Są one bowiem jedynym sensownym kreowania przestrzennych światów.

Łatwo domyślić się, dlaczego tak jest. Gdyby w tym przypadku zastosować technikę znaną z grafiki rastrowej, czyli podział na elementarne punkty, gotowy „obraz” zajmował mnóstwo miejsca - nawet tysiące razy więcej niż duże bitmapy. Trudno też wymyślić jakiś sensowny sposób tworzenia takich trójwymiarowych bitmap, podobnie jak niemożliwie jest rysowanie przestrzennych szkiców na papierze.

W tym przypadku zwrócenie się w stronę geometrii było więc konieczne. Dało to zresztą całkiem zadowalające efekty.

Obecna technika modelowania trójwymiarowych scen zakłada ich podział na bryły złożone z prostych figur płaskich - zwykle trójkątów. Figury te tworzą zewnętrzną „powłokę” obiektów w scenie, widoczną dla oglądającego. Ich powierzchnie mogą być dodatkowo pokryte dwuwymiarowymi bitmapami - teksturami oraz dawać złudzenie odbijania i rozpraszania światła. Wszystkie te efekty są osiągnięte poprzez odpowiednie obliczenia matematyczne.



Screen 66. Programy do modelowania trójwymiarowego tworzone obiekty pokazują zazwyczaj w trzech płaskich widokach oraz w perspektywnie przestrzennej

Tworzenie obiektów 3D (czyli modelowanie) może się z kolei odbywać wieloma drogami. Teoretycznie najprostszym jest ręczne ustawianie w przestrzeni wierzchołków, składających się na trójkąty (a w konsekwencji na całe bryły). O wiele efektywniejsze jest ponowne zaprzęgnięcie do pracy geometrii analitycznej; za pomocą odpowiednich kalkulacji generowane są proste kształty, jak prostopadłościany, walce, kule czy stożki. Z ich połączenia są następnie tworzone zarysy właściwych brył, a przy pomocy pewnych

efektów (zwanymi modyfikatorami, ang. *modifiers*) osiągany jest ostateczny kształt obiektów.

Rasteryzacja

Z prezentacją grafiki wektorowej związany jest pewien kłopot. Otóż mało które urządzenie wyjściowe potrafi podołać temu zadaniu bezpośrednio¹³⁵; większość wymaga, aby rysunek wektorowy został wcześniej przełożony na bitmapę. Proces ten nazywamy **rasteryzacją** i zachodzi on za każdym razem, kiedy oglądamy obraz wektorowy na ekranie monitora lub drukujemy go na zwykłej drukarce.

Dla płaskiej grafiki rasteryzacja jest stosunkowo prostą czynnością, sprowadzającą się do przeprowadzenia obliczeń opisujących obiekty na rysunku. Znacznie więcej pracy wymaga przedstawienie sceny trójwymiarowej na płaszczyźnie ekranu - ten proces zwiemy **renderowaniem**. Obejmuje on wyliczenie kolorów modeli na podstawie nałożonych tekstur i oświetlenia, a następnie rzut przestrzeni 3D na powierzchnię płaską. Dopiero na końcu realizowane są formuły matematyczne opisujące bryły na scenie, która jest wreszcie rasteryzowana na ekranie monitora.

Renderowaniem scen trójwymiarowych, szczególnie w czasie rzeczywistym, zajmiemy się naturalnie jeszcze nie raz. Czynność ta jest przecież jednym z głównych zadań biblioteki DirectX, z którą mamy się wkrótce zaznajomić.

Wyjściowe urządzenia graficzne

Przypomnimy sobie teraz (a częściowo też wprowadzimy) niektóre ważne kwestie związane z dwoma najważniejszymi urządzeniami, służącymi do prezentacji grafiki: monitorem oraz drukarką. Oba te urządzenia służą do wyświetlania obrazu **rastrowego**.

Monitor

Chociaż serwery i komputery *mainframe* mogą obyć się bez monitora, osobiste pecety nie mogłyby działać bez tego komputerowego „telewizora”. Obraz wyświetlany na monitorze jest bowiem podstawowym sposobem, w jaki działające programy informują o swoim stanie. Od czasu rozpowszechnienia się interfejsów graficznych jest też czymś w typie pola manewrowego, po którym użytkownik rozstawia uruchomione aplikacje.

Typy monitorów

Generalnie, monitory dzielimy na dwie duże grupy: na **kineskopowe** oraz **ciekłokrystaliczne**. Różni je technologia wyświetlania obrazu, a nierzadko także jego jakość.

Monitory kineskopowe

Ten typ monitorów (zwanymi też **CRT**, od ang. *Catode Ray Tube* - kineskop katodowy) działa bardzo podobnie do odbiorników telewizyjnych. Powstawanie obrazu jest tu wynikiem odpowiedniego naładowania kineskopu pod wpływem strumienia elektronów. Ów strumień kilkadziesiąt razy na sekundę „przelatuje” przez cały kineskop, przemieszczając się wierszami - począwszy od lewego górnego rogu. Tak szybka zmiana wyświetlacza powoduje złudzenie jego stałości oraz płynnego ruchu.

Technologia monitorów kineskopowych liczy sobie prawie sto lat i przez ten czas była znacznie ulepszana. Obecnie jakość obrazu w monitorach CRT jest bardzo wysoka; co

¹³⁵ Do takich urządzeń należa bodaj wyłącznie specjalistyczne plotery.

więcej, nie przeszkadza ona w osiągnięciu wysokiej rozdzielczości i częstotliwości odświeżania.

Jakiś czas temu rozwiązano też problem, który pojawiał się przy wielogodzinnym pokazywaniu tego samego obrazu na ekranie monitora. Dawniej mogło spowodować wypalenie go na kineskopie, przez co zarys feralnego widoku zostawał na monitorze już na zawsze. Ta nieprzyjemna ewentualność była przyczyną powstania programów znanych jako **wygaszacze ekranu** (ang. *screen savers*, dosł. 'oszczędacze ekranu'), których zadaniem było wyświetlanie szybko zmieniających się pikseli w czasie bezczynności użytkownika komputera. Teraz ryzyko wypalenia już nie istnieje, ale wygaszacze pozostały - głównie jako cieszące oko spektakle obrazów i nawet dźwięków.



Fotografia 7 i 8. Monitory kineskopowe
(fotografie pochodzą z [serwisu internetowego firmy Philips](#))

Teoretyczną wadą monitorów kineskopowych jest emisja potencjalnie szkodliwego dla oczu promieniowania. Teoretyczną, gdyż obecne normy w tym zakresie (oznaczana jako TCO) są tak rygorystyczne, że spełniające je produkty nie są w zasadzie żadnych zagrożeniem dla naszych spojówek. Nie zmienia to jednak faktu, że długa praca przed ekranem męczy wzrok i przyczynia się do jego osłabienia. Powodem tego nie są jednak tajemnicze promienie spoza zakresu widzialnego, lecz zbyt duża ilość światła docierającego do oka. Temu można zaradzić tylko w jeden sposób: trzeba właściwie dostroić ustawienia swego sprzętu.

Najważniejszym spośród nich jest temperatura kolorów. Powinno się ją ustawić na jak najmniejszą wartość, zazwyczaj 6500 kelwinów. Następnie należy dopasować wygląd obrazu za pomocą kontroli ostrości, kontrastu i jasności.

Dawniej niedogodnością był również duży rozmiar i waga tych monitorów. Teraz jednak produkowane modele są lżejsze i węższe, dzięki czemu nie odbiegają zbyt wiele od monitorów ciekłokrystalicznych w konkurencji zajmowanego na biurku miejsca.

Monitory ciekłokrystaliczne

Drugi rodzaj monitorów oznaczany jest skrótem **LCD** (ang. *Liquid Crystal Display*), który wskazuje na wyświetlacz zbudowany z tzw. **ciekłych kryształów**. Ta dziwna substancja o wewnętrznie sprzecznej nazwie¹³⁶ posiada zdolność polaryzacji, co pozwala jej wyświetlać zaprogramowany obraz. Jest on kontrolowany przez pole elektryczne, zatem monitory LCD nie posiadają strzelby elektronowej; mogą więc być o wiele węższe niż ich kuzyni z kineskopami.

Powierzchnia wyświetlacza w monitorze ciekłokrystalicznym jest podzielona na pojedyncze piksele - wynika stąd, iż urządzenia te mają zaprogramowaną stałą

¹³⁶ Kryształy są przecież ciałami stałymi, a nie ciecżą...

rozdzielczość. Dodatkowo, każdy piksel składa się z trzech **subpikseli**, odpowiedzialnych za wyświetlanie barw składowych systemu RGB.

Subpiksele po okresie dłuższego użytkowania mogą się wypalić, przez nie będą wiernie odzwierciedlały kolorów. Uważa się, że niezauważalne dla ludzkiego oka jest wypalenie się od kilku do kilkunastu subpikseli. Niestety, wygaszacze ekranu nie mogą wiele zrobić w sprawie zapobiegania temu niepożądanemu zjawisku.



Fotografia 9 i 10. Monitory ciekłokrystaliczne
(fotografie pochodzą z [serwisu internetowego firmy Philips](#))

Wadą monitorów LCD jest niska jakość obrazu - niższa niż w modelach kineskopowych. Nie chodzi tu wcale o obecność uszkodzonych (sub)pikseli (choć to również się liczy), lecz o względne różnice w jasności poszczególnych obszarach ekranu. Dotyczy to szczególnie skrajnych i środkowych partii wyświetlacza.

Nieszczerólnie przychylna dla użytkownika jest też cena tych urządzeń. Monitory LCD są bowiem co najmniej dwukrotnie droższe od analogicznych (pod względem wielkości) modeli CRT. I nie zanosi się na szybką zmianę tego stanu rzeczy.

Na plus można aczkolwiek zaliczyć tym monitorom brak emisji jakiegokolwiek promieniowania (poza, oczywiście, światłem widzialnym). Mówiłem jednak, że nie wyprzedzają zbyt modeli CRT w tej dziedzinie, których drakońskie normy doprowadziły do spadku ilości emitowanych fal elektromagnetycznych niemal do zera.

Snobistyczną ciekawostką są monitory plazmowe, funkcjonujące na identycznej zasadzie jak tego rodzaju telewizory. Obraz powstaje w nich poprzez jonizację cząsteczek gazów szlachetnych (zwykle neonu, kryptonu i ksenonu), pod wpływem której gazy zaczynają emitować światło.

Jakość obrazu w monitorach i telewizorach plazmowych jest bardzo wysoka. Niestety, równie wysoka jest też cena - niemal dziesięć razy większa niż koszt monitorów CRT.

Parametry obrazu

Monitor monitorowi nierówny - nie tylko jeśli chodzi o zastosowaną technologię prezentacji obrazu. Ważnych jest mnóstwo empirycznie obserwowanych parametrów, z których najważniejszymi są: **rozdzielczość** obrazu, **głębia kolorów** oraz **częstotliwość odświeżania**.

Rozdzielczość

Najpopularniejszym (acz niezbyt precyzyjnym) sposobem opisanie modelu monitora jest podanie jego **przekątnej**. Liczba ta określa odległość przeciwległych wierzchołków kineskopu lub wyświetlacza LCD danego monitora. Zwróćmy więc uwagę, że nie mówi ona **nic** o wielkości rzeczywistego obrazu, jaki będziemy mogli obserwować. Tę zaś można sprawdzić tylko w praktyce - zazwyczaj jest ona widocznie mniejsza.

Zarówno przekątną kineskopu (wyświetlacza), jak i obrazu podajemy w calach. Jeden cal to ok. 2,54 cm, a symbolem tej jednostki jest znak ". Monitor CRT 17" ma zatem kineskop o przekątnej długości około 43 centymetrów.

Dłuższa przekątna oznacza więcej miejsca dla pikseli ekranu. Ilość punktów obrazu, jaką aktualnie wyświetla monitor, nazywamy jego **rozdzielczością**. Jest to wielkość analogiczna do rozdzielczości bitmapy i podajemy ją w tej samej postaci: dwóch liczb, określających liczbę pikseli w poziomie i pionie.

Typowe rozdzielczości monitorów to: 640×480, 800×600, 1024×768, 1152×864, 1280×960, 1600×1200, 1920×1440 oraz 2048×1536 pikseli. Kilka ostatnich wartości osiągają jednak tylko największe monitory; reszta może być stosowana dla popularnych wielkości ekranu. Zalecane rozdzielczości dla wybranych przekątnych monitorów przedstawia tabela:

<i>przekątna</i>	<i>rozdzielczość</i>
14 cali	640×480
15 cali	640×480 lub 800×600
17 cali	800×600 lub 1024×768
19 cali	1152×864 lub 1280×960
21 cali	1280×960 lub 1600×1200

Tabela 57. Optymalne rozdzielczości dla monitorów CRT różnych wielkości. Modele LCD mają stałą rozdzielczość, dla każdej z przekątnych jest ona równa większej wartości z tabeli

Rozdzielczość ekranu możesz ustawić we *Właściwościach ekranu* w Panelu Sterowania, a programowo pobrać za pomocą `GetSystemMetrics(SM_CXSCREEN/SM_CYSCREEN)`.

Można zauważyć, że stosunek szerokości do wysokości ekranu jest w przypadku wszystkich możliwych rozdzielczości ten sam i wynosi **4:3**. Iloraz ten nazywamy **aspektem obrazu** (ang. *image aspect*) monitora. Aspekt 4:3 jest używany szeroko także w telewizorach, natomiast filmy kinowe są kadrowane z aspektem 16:9. Ich odtwarzanie na domowych odbiornikach i monitorach powoduje więc pojawienie się czarnych pasków na górze i dole kadru.

Głębia kolorów

Nie mniej ważna niż wielkość obrazu jest wierność odwzorowania w nim barw. Decyduje o tym liczba dostępnych kolorów, czyli ich **głębia** (ang. *color depth*).

Głębia kolorów zależy od ilości bitów przypadających na jeden piksel. Liczba ta może wahać się od jednego do (na razie) 32 bitów. Najczęściej obsługiwane tryby barwne monitorów są zebrane w poniższej tabelce:

<i>ilość kolorów</i>	<i>liczba bitów</i>	<i>nazwa trybu</i>	<i>uwagi</i>
2	1	monochromatyczny	obraz czarno-biały
16	4	16 kolorów	tryb oparty na paletcie stałych barw, a nie na ich zapisie z użyciem składowych RGB
256	8	256 kolorów	
65 536	16	<i>High Color</i>	kolor zapisany z użyciem kanałów RGB
16 777 216	24	<i>True Color</i>	
16 777 216	32		Jest to taki sam tryb jak 24-bitowy <i>True Color</i> , a dodatkowy bajt sprawia, że dane pikseli są w pamięci obrazu wyrównywane do 4 bajtów.

Tabela 58. Tryby głębi kolorów obsługiwane przez współrzędne monitory

Dzisiaj każdy model monitora i karty graficznej bez problemu radzi sobie z wyświetlaniem milionów barw trybu *True Color*. Na starszych pecetach ustawia się aczkolwiek niższą głębię *High Color*, gdyż zużywa ona mniej cennego czasu procesora.

Częstotliwość odświeżania

Radosne zwiększanie rozdzielczości i głębi kolorów wyświetlanego obrazu, aż do sztywnych granic, jest całkowicie możliwe. Pomijając fakt, że większa ilość pikseli przy niezmiennej przekątnej monitora powoduje zmniejszenie czytelności małych elementów, zbyt wyśrubowane ustawienia mogą być przyczyną także innej formy dyskomfortu. Jest nią zbyt niska **częstotliwość odświeżania** (ang. *refresh rate*).

Parametr ten dotyczy wyłącznie monitorów kineskopowych. W modelach LCD obraz jest wyświetlany stale.

Wielkość ta mówi nam, jak wiele razy w ciągu sekundy monitor odrysowuje zawartość ekranu. Jak każdą częstotliwość podajemy ją w hercach (Hz). Uznaje się, że dla komfortu użytkownika komputera nie powinna ona zejść poniżej 60 Hz. W praktyce jest jednak o wiele większa, sięgająca co najmniej 85 Hz, a wszelka przesada w tej materii jest bardzo wskazana. Szybsze odświeżanie obrazu oznacza bowiem mniejsze zmęczenie dla oczu patrzącego.

Jeśli nasz monitor nie odświeża obrazu dostatecznie szybko, to możemy zauważyć jego migotanie - właściwy obraz przeplata się z czarnym ekranem. Dzieje się tak, gdyż to co widzimy na ekranie monitora CRT jest tak naprawdę stanem chwilowym, momentalnym rozbłyskiem elektronów płynących ze specjalnego działła. Ta „strzelba” przy każdym odświeżeniu obrazu wysyła ładunki do wszystkich pikseli, począwszy od lewego górnego i posuwając się wierszami aż do prawego dolnego rogu. Po wykonaniu tego pracochłonnego zadania działło wraca na wyjściową pozycję, gotowe do ponownego rozpoczęcia odświeżania. Moment pokonywania drogi po przekątnej, gdy nie są wysyłane żadne elektrony, nazywamy **powrotem pionowym** (ang. *vertical synchronisation*, w skrócie *VSynC*). To właśnie wtedy ekran monitora pozostaje czarny, co czasami można dostrzec na starych lub psujących się modelach.

Moment powrotu pionowego jest idealną chwilą na dokonanie całościowej zmiany obrazu prezentowanego na ekranie. Jeżeli bowiem dokonano by takiej zmiany w trakcie wysyłania strumienia elektronów, wówczas pokazywany na ekranie obrazek byłby podzielony na dwie części. Ten efekt nazywamy **rozdarciem** (ang. *tearing up*) i jest on wysoce niepożądany.

Zaawansowane biblioteki graficzne, takie jak DirectX, czekają więc z odświeżeniem obrazu aż do wystąpienia powrotu pionowego. Gwarantuje to, że efekt rozdarcia nigdy nie wystąpi.

Monitory ciekłokrystaliczne nie są określone przez swoją częstotliwość odświeżania, jako że takiego pojęcia w ogóle się do nich nie stosuje. Wyświetlacze LCD prezentują po prostu stały obraz, na żądanie zmieniając kolory potrzebnych pikseli.

Kombinacja rozdzielczości ekranu, głębi kolorów i częstotliwości odświeżania nazywana jest **trybem graficznym** (ang. *graphics mode*). Jego określenie zapisuje się często razem, posługując się czterema liczbami, np. 800×600×24@85. Ten tryb oznacza, że obraz jest wyświetlany w rozdzielczości 800×600, z 24-bitową głębią kolorów i odświeżany z częstotliwością 85 Hz.

Drukarka

Kiedy chcemy otrzymać edytowany dokument na papierze, używamy **drukarki** (ang. *printer*). Jest to drugie po monitorze, najważniejsze urządzenie wyjściowe.

Typy drukarek

Od lat wyróżnia się trzy typy drukarek, biorąc pod uwagę używaną technikę nakładania druku na papier. Te trzy rodzaje to drukarki **igłowe**, **atramentowe** i **laserowe**.

Drukarki igłowe

Jest to najstarszy i najprymitywniejszy, choć wciąż jeszcze popularny rodzaj drukarki. W modelach igłowych (ang. *needle printers*) litery powstają z drobnych porcji tuszu, nakładanych punktowo przez cienkie igły (stąd nazwa) i taśmę barwiącą. Wydruki są dokonywane zwykle na długich, perforowanych rolkach papieru, które w razie potrzeby można rozdzielić na pojedyncze arkusze.



Fotografia 11 i 12. Współczesne modele drukarek igłowych

Zaletą drukarki igłowej jest względna szybkość produkowania zadrukowanych arkuszy. Niebagatelnie ważna jest też bardzo tania eksploatacja takiej drukarki - sprawia to, że „igłówki” są popularne np. w sklepach, gdzie konieczne jest drukowanie dużej ilości rachunków i faktur.

Jakość wydruków pozostawia jednak wiele do życzenia - z pewnością jest zbyt niska dla zastosowań biurowych czy domowych. Poza tym praca drukarki igłowej wiąże się z głośnym i mało przyjemnym, piskliwym hałasem.

Drukarki atramentowe

Drukarki atramentowe (ang. *inkjet printers*), czyli popularne „plujki”, są obecnie najpopularniejszym rodzajem urządzeń drukujących. Znaleźć je można w wielu domach użytkowników komputerów.

Działanie drukarek atramentowych polega na rozpylaniu nad papierem bardzo drobnych kropelek tuszu. Kropelki te przylegają do kartki papieru, pokrywając ją i tworząc w ten sposób kształty tekstu oraz grafiki.

Pierwsze modele funkcjonowały w oparciu o jeden pojemnik z tuszem, lecz teraz standardem są cztery, zawierające podstawowe barwy systemu CMYK. Z ich połączenia można więc otrzymać dowolny kolor i dlatego drukarki atramentowe najczęściej dobrze oddają barwy widoczne na ekranie (choć zależy to oczywiście od klasy konkretnego modelu).

Ogólna jakość wydruków także jest zadowalająca, poza tym można ją często programowo ustawiać. Najlepsze rezultaty wymagają jednak dużych ilości atramentu i z tego powodu drukarki atramentowe nie są zbyt ekonomiczne w eksploatacji.

Bardzo powszechnym błędem jest określanie barwników do drukarek atramentowych mianem tonera. Jest to niepoprawne, gdyż tonery są tak naprawdę używane tylko przez drukarki laserowe. Atramentowe korzystają natomiast z tuszu lub po prostu atramentu.



Fotografia 13 i 14. Typowe modele drukarek atramentowych
(fotografie pochodzą z [serwisu internetowego firmy Hewlett-Packard](#))

Generalnie można aczkolwiek powiedzieć, że „atramentówki” są dobrym kompromisem między jakością wydruków a ich kosztami.

Drukarki laserowe

Trzeci rodzaj drukarek jest znany z bardzo ostrych wydruków czarno-białych oraz... wysokiej ceny.

Drukarki laserowe (ang. *laser printers*) zawierają w swym wnętrzu obrotowy mechanizm, który w trakcie drukowania jest naświetlany i elektryzowany przez laser. W miejscach, gdzie to się dokonuje, do bębna (bo tak nazywa się ten mechanizm) przylegają cząstki drobnego proszku (**tonera**). Osiadają one następnie na papierze, który w tym celu jest elektryzowany przeciwnym znakiem ładunku.



Fotografia 15 i 16. Przykładowe drukarki laserowe
(fotografie pochodzą z [serwisu internetowego firmy Hewlett-Packard](#))

Uzyskiwane w ten sposób obrazy charakteryzują się dużą ostrością i rozdzielczością. Nieco gorzej bywa z odzwierciedleniem kolorów, jako że technologia druku laserowego przez długi czas była przeznaczona tylko do wydruków monochromatycznych. W zasadzie jednak ogólną jakość drukowania można uznać za wysoką.

Za tę jakość trzeba niestety sporo zapłacić. Chodzi tu szczególnie o cenę samego urządzenia - co najmniej trzy razy większą niż cena przeciętnej drukarki atramentowej. Koszt zużywanego tonera jest natomiast nieco niższy od kosztu eksploatacji „plujki”.

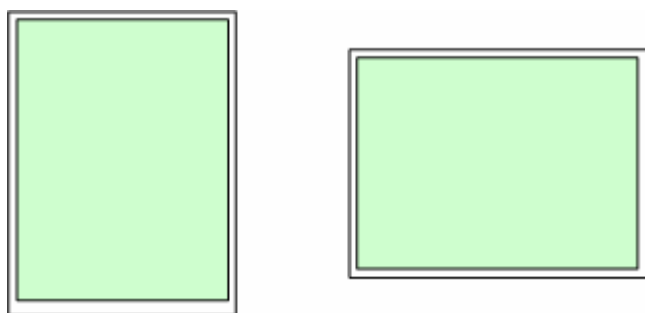
Przedstawione tu trzy rodzaje nie są naturalnie jedynymi typami drukarem. Pozostałe są jednak przeznaczone do specyficznych zastosowań. Ciekawym przykładem są choćby drukarki termosublimacyjne, w których obraz powstaje z cząsteczek barwnika doprowadzonych do stanu lotnego poprzez wysoką temperaturę. Cząsteczki te osiadają na papierze, tworząc nieprzeciętnie ostre wydruki, odpowiednie do prezentowania kolorowych fotografii. Nie trzeba chyba dodawać, że tego typu urządzenia są bardzo, bardzo drogie.

Parametry wydruku

Drukarka, podobnie jak monitor, produkuje obraz rastrowy¹³⁷. Jego parametry są więc podobne do tych określających wyświetlacz komputerowego „telewizora”. Przyjrzymy się im teraz.

Obszar wydruku

Bardzo niewiele drukarek potrafi zapełnić każdy kawałek podanej im kartki papieru, najczęściej w formacie A4. Ogromna większość ogranicza się do jej części, zwanej **obszarem drukowania** (ang. *printing area*). Zwykle nie jest on wiele mniejszy od wymiarów papieru.



Schemat 48. Przykładowy obszar drukowania

Należy jednak zwracać uwagę, aby marginesy naszych dokumentów znajdowały się w całości w tej strefie, bowiem w przeciwnym razie skończy się to „obcięciem” tekstu lub grafiki.

Rozdzielczość

Obraz wydrukowany, tak samo jak ten na ekranie, składa się z małych punktów - już nie pikseli, a **kropek** (ang. *dots*). Możemy więc także mówić o jego rozdzielczości.

Jej miarą nie jest jednak ilość punktów w pionowym i poziomym wymiarze kartki, gdyż takiej wielkości nie można porównywać między drukarkami operującymi na różnych formatach papieru. Zamiast tego mówi się, jak wiele kropek przypada na pewną małą jednostkę powierzchni - cal kwadratowy (ok. 6,5 cm²). Miarę tę oznaczamy literami **dpi** (ang. *dots per inch* - kropki na cal).

Rozdzielczość drukarki możemy podawać jedną lub dwoma liczbami. W tym drugim przypadku mówi się, ile kropek przypada na cal długości poziomej oraz pionowej. Przykładowo, 300×400 dpi oznacza, iż jeden cal kwadratowy wydruku jest prostokątem mającym 300 kropek długości i 400 wysokości.

Zwykle kropki są swym kształcie zbliżone raczej do kół i dlatego rozdzielczość w obu wymiarach jest taka sama. Wtedy też wystarcza tylko jedna liczba do jej opisu, tak więc

¹³⁷ Ścisłej to takie drukarki nazywamy mozaikowymi (gdyż plotery są formalnie także drukarkami), ale przyjęło się nieużywanie tego dodatkowego określenia.

zamiast mówić, że gęstość wydruku wynosi, dajmy na to, 600×600 dpi wystarczy powiedzieć, że jest ona równa 600 dpi. Taką miarę rozdzielczości stosuje się najczęściej.

Dzisiaj drukarki atramentowe mają rozdzielczość około 1200 dpi, zaś laserowe niemal dwa razy większą.

Kolory

W przypadku monitorów możemy mówić o wielu trybach wyświetlania, różniących się ilością potencjalnych kolorów. Dla drukarek sprawa wygląda inaczej.

Otóż nie stosuje się ogóle pojęcia głębi kolorów. Zamiast tego wydruk można określić jako:

- **monochromatyczny**, gdy jego punkty są albo czarne (zadrukowane), albo białe (niezadrukowane). W ten sposób funkcjonują drukarki igłowe
- wykonany w **skali szarości** przez drukarkę atramentową lub laserową. Większą lub mniejszą jasność punktów uzyskuje się poprzez zmienną ilość tuszu (tonera) pokrywającego kartkę
- **kolorowy**

W tym ostatnim przypadku możliwe są oczywiście rozbieżności między dokładnością odwzorowania barw w różnych drukarkach. Faktycznie jednak są one trudne do obiektywnego określenia, ponieważ wymagałyby sprecyzowania, jak małe porcje atramentu (tonera) mogą być mieszane ze sobą przez dany model drukarki. Mimo to wielu producentów chwali się milionami kolorów, jakie rzekomo mogą otrzymać ich urządzenia. Do takich doniesień trzeba więc podchodzić z dużą rezerwą.

Podstawy Windows GDI

W tym podrozdziale zajmiemy się nareszcie zasadniczym zagadnieniem. Przedstawię tutaj podstawowe wiadomości na temat biblioteki graficznej Windows GDI. Przydadzą się one w dalszej części rozdziału, gdy przejdziemy już do poszczególnych elementów tego przebogatego interfejsu.

Rozpocniemy tu od kluczowego pojęcia potoku grafiki.

Potok graficzny

Sekwencyjna natura komputerów jest przyczyną tego, że wiele związanych z nimi kwestii dzieli się na mniej lub bardziej oczywiste etapy. Nie inaczej jest też z wyświetlaniem obrazu przez biblioteki graficzne - tą kaskadę kolejnych szczebli nazywamy w ich przypadku **potokiem graficznym** (ang. *graphics pipeline*).

Taki potok obrazuje, w jaki sposób polecenia i funkcje rysujące, wywoływane przez program, przekładają się ostatecznie na rezultat widoczny na ekranie. Między punktem startu a końcem może się znajdować wiele stadiów pośrednich - przekształceń, transformacji, manipulacji. W sumie otrzymujemy taki, a nie inny obraz - obraz, który sami narysowaliśmy.

Znajomość potoku graficznego jest więc nieodzowna. Bez tego nie moglibyśmy świadomie korzystać z biblioteki graficznej. Nie moglibyśmy właściwie wykorzystać jej potencjału. Nie moglibyśmy wreszcie przedstawić na ekranie tego, co chcemy.

Potok graficzny jest też pewnym rodzajem abstrakcji, więc umożliwia niezależność biblioteki od sprzętu (ang. *device-independence*).

W Windows GDI potok graficzny także występuje, chociaż nie wszyscy zdają sobie sprawę z jego istnienia. Teoretycznie można by się nawet obyć bez wiedzy o tym, ale jest ona całkiem pożyteczna. Jeśli bowiem poznasz teraz prosty potok związany z GDI, łatwiej będzie ci później zaznajomić się ze znacznie bardziej skomplikowanym potokiem geometrii w DirectX.

Rzućmy zatem okiem na kolejne etapy przetwarzania obrazu w Windows GDI.

Tę sekcję możesz śmiało pominąć przy pierwszym czytaniu, jeżeli uznasz ją za zbyt trudną. Wróć jednak do niej po lekturze całego rozdziału.

Tryby grafiki

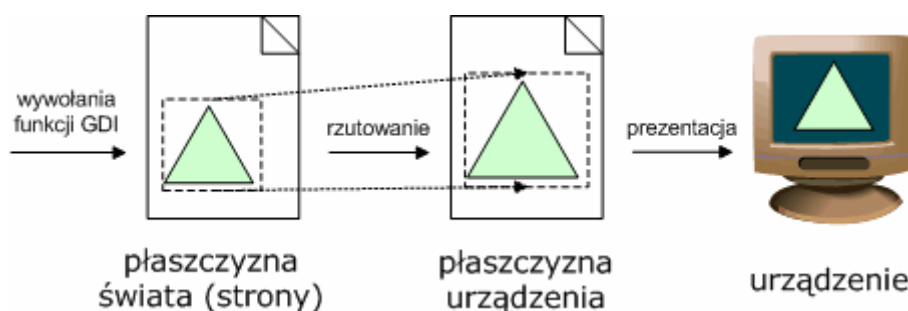
Jak wiele długożyjących produktów programistycznych, biblioteka Windows GDI podlegała ewolucji w trakcie swego istnienia. Zmiany nie omijały także jej istoty, czyli potoku graficznego.

Doprowadziły one w końcu do wyodrębnienia się dwóch **trybów grafiki** (ang. *graphics mode*). Mają one odrobinę różniące się od siebie potoki graficzne - a mówiąc dokładniej, jeden z nich jest uboższą wersją drugiego.

Tryb kompatybilny

Prostszym trybem grafiki jest **tryb kompatybilny** (ang. *compatible mode*) Windows GDI. Jego nazwa jest nieprzypadkowa, gdyż hipotetycznie został on zachowany wyłącznie celem zgodności z 16-bitowymi wersjami Windows. Z powodu swej prostoty jest on jednak szeroko wykorzystywany także i teraz; zwłaszcza, iż jest to **domyślny tryb** grafiki.

Potok graficzny w tym trybie można zilustrować poniższym schematem:



Schemat 49. Potok graficzny Windows GDI w trybie kompatybilnym

W tym akapicie omówimy go skrótowo, w kilku punktach. Każdym etapem zajmiemy się dokładnie w następnych akapitach, gdy poznamy także potok trybu zaawansowany. Wszystkie te stadia występują bowiem również i tam.

Rysowanie na płaszczyźnie świata (strony)

Wywoływanie funkcji Windows GDI nie przekłada się natychmiast na zmiany obrazu na ekranie monitora (lub innego urządzenia wejściowego). Wpierw modyfikowana jest **płaszczyzna świata** (ang. *world space*¹³⁸), w trybie kompatybilnym tożsama z **płaszczyzną strony** (ang. *page space*). Jest to pewien dodatkowy poziom abstrakcji, pozwalający na względną niezależność od rzeczywistego urządzenia. Dzięki temu GDI pozwala rysować zarówno na monitorze, jak choćby i na drukarce. To nadmiarowe stadium umożliwia też stosowanie dowolnych jednostek miary dla obrazu.

¹³⁸ W zasadzie jest to niby przestrzeń świata. Zdecydowałem się jednak na nazwę 'płaszczyzna', gdyż mówimy o rysunkach dwuwymiarowych. Nazwę 'przestrzeń' rezerwuję dla grafiki 3D.

Rzutowanie dla płaszczyzny urządzenia

Płaszczyzna świata (strony) jest teoretycznie niemal nieograniczona, więc nie możemy wyświetlić jej całej. Trzeba zdecydować się na pewien wycinek.

Dokładniej mówimy tu o dwóch wycinkach w kształcie prostokątów. Pierwszy znajduje się na płaszczyźnie świata (strony) i definiuje tę jej część, która zostanie pobrana do ostatecznego wyświetlenia. Poza tym - co zaraz sobie wyjaśnimy - precyzuje on też granice układu współrzędnych świata (strony).

Istnieje jeszcze drugi prostokąt, obecny już na **płaszczyźnie urządzenia** (ang. *device space*). Jest on miejscem, gdzie fragment płaszczyzny z poprzedniego stadium zostanie rzutowany i przygotowany do właściwego wyświetlenia.

Te dwa ważne prostokąty będziemy nazywać kadrem i wziernikiem, a powiemy sobie o nich więcej w kolejnych akapitach.

Prezentacja na fizycznym urządzeniu

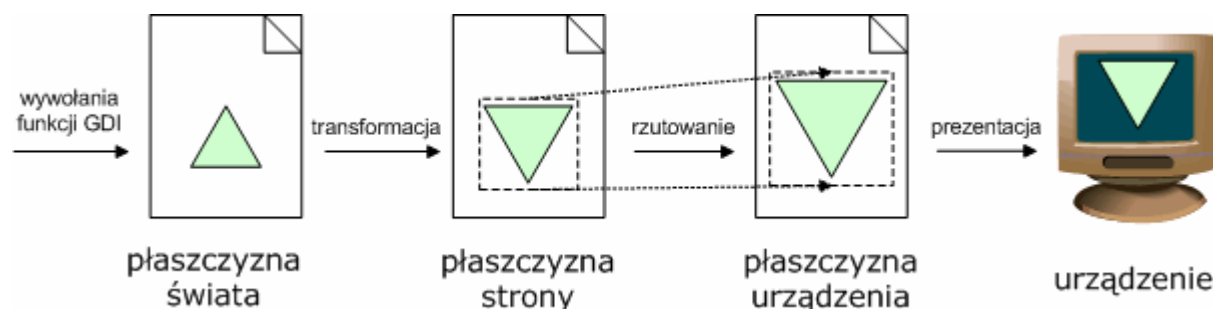
Z płaszczyzny urządzenia jest już krótka droga do... samego urządzenia. W tym momencie Windows GDI porzuca swoją niezależność od sprzętu i przystępuje do najbardziej widocznej dla nas pracy.

Biblioteka wysyła więc albo odpowiednie polecenia, albo też zasteryzowany obraz do sterownika urządzenia wyjściowego. Najczęściej tym urządzeniem jest monitor, zatem gotowy obraz trafia do karty graficznej. Ona przesyła sygnały do monitora, który ostatecznie wyświetla przygotowany rysunek.

Zwykle nie zajmuje on aczkolwiek całego dostępnego ekranu, lecz jest zawarty np. we wnętrzu jakiegoś okna. Windows GDI musi zatem ściśle współpracować z interfejsem użytkownika systemu Windows, jednak nas zbytnio to nie interesuje. Ważne jest, że wynik naszej współpracy z GDI zostaje definitywnie pokazany na urządzeniu wyjściowym. Sama biblioteka dba przy tym o jego odpowiednie przycięcie, gdyby nie mieścił się w wyznaczonym dla siebie obszarze (np. oknie).

Tryb zaawansowany

Oprócz trybu kompatybilnego, GDI daje możliwość dokonywania globalnych manipulacji obrazu podczas jego przejścia przez pierwsze etapy potoku graficznego. Tryb grafiki, który to umożliwia nazywamy **zaawansowanym** (ang. *advanced mode*); posiada on swój własny potok graficzny - bardzo podobny do poprzedniego:



Schemat 50. Potok graficzny w Windows GDI w trybie zaawansowanym

Występuje tu tylko jeden dodatkowy etap przekształcania, na który z grubsza rzucimy okiem.

Transformacja świata

W trybie zaawansowanym płaszczyzna świata jest oddzielna od płaszczyzny strony, gdyż ją poprzedza. Przejściu od tej pierwszej do drugiej mogą przy tym towarzyszyć uzupełniające transformacje.

Takimi transformacjami są zwykle geometryczne odwzorowania w rodzaju przesunięcia, obrotu i skalowania. Windows GDI używa macierzy 3×3 do reprezentacji tych działań na obrazach. Forma ta pozwala na ich łatwe łączenie z zachowaniem kolejności przekształceń.

Nie będziemy tutaj omawiać tego zagadnienia, ponieważ transformacje płaszczyzny świata stosuje się nadzwyczaj rzadko. Wprowadzenie w temat macierzy i ich rolę w geometrii grafiki odłożymy aż do czasu poznawania biblioteki DirectX. Tam już nie będziemy się mogli bez nich obyć, ale w Windows GDI jest to całkowicie dopuszczalne i poprawne.

Ustawianie trybu zaawansowego

W zasadzie jest to chyba nawet więcej niż dopuszczalne. Tryb zaawansowany nie jest bowiem domyślnym trybem GDI w tych systemach Windows, które go obsługują. Kwestią o tym decydującą jest zapewne zgodność z programami dla starszych wersji systemu. Zachowanie kompatybilności jest na razie koniecznością, ponieważ tryb zaawansowany jest obsługiwany dopiero w Windows NT, 2000 i XP.

Niemniej, chociaż nie będziemy korzystać z tego trybu w niniejszym rozdziale, warto wiedzieć jak można go przynajmniej włączyć. Nie jest to trudne, wystarczy posłużyć się funkcją `SetGraphicsMode()`:

```
SetGraphicsMode (hdcKontekst, GM_ADVANCED);
```

Jej drugi parametr wskazuje na wybrany tryb: `GM_ADVANCED` to żądany tryb zaawansowany, `GM_COMPATIBLE` spowoduje powrót do standardowego ustawienia kompatybilnego.

Pierwszym parametrem jest natomiast uchwyt do tzw. kontekstu urządzenia. To niezwykle ważne pojęcie Windows GDI i dlatego poświęcimy mu wiele miejsca - ale nieco później. Na razie zapamiętaj, że kontekst ten precyzuje miejsce, w którym będziemy rysować. Uchwytów do kontekstów urządzenia z powodzeniem używaliśmy w poprzednich rozdziałach, więc myślę, iż ta kwestia nie jest dla ciebie aż taką nowością.

Zatem tryb grafiki ustawiamy dla konkretnego kontekstu urządzenia, zwykle przynależnego naszej aplikacji. Nie jest to więc parametr właściwy całemu systemowi, a każda aplikacja może zdecydować, w jakim trybie chce spożytkować interfejs Windows GDI.

Ażeby jednak dobrze go wykorzystać, musimy dowiedzieć się nieco więcej o kolejnych stadiach potoku graficznego, co uczynimy zaraz. Później zajmiemy się również pojęciem kontekstu urządzenia.

Płaszczyzna świata (strony)

Pierwszym „miejscem”, gdzie wywołania GDI dają jakieś rezultaty, jest płaszczyzna świata - w trybie kompatybilnym zwana także płaszczyzną strony. Słowo „miejsce” piszę tu w cudzysłowie, ponieważ faktycznie chodzi o coś zupełnie abstrakcyjnego, znacznie bardziej „wirtualnego” niż choćby powierzchnia pulpitu Windows, którą możemy normalnie oglądać na ekranie swego monitora.

Płaszczyzna świata nie jest nieskończona, choćby dlatego że jej wymiary ograniczałby rozmiar zmiennych całkowitych. Jest ona skończona także z tego powodu, iż odnosi się do bardziej konkretnego zakresu na ekranie (np. wnętrza okna), strony w drukarce czy

jeszcze innego rejonu w innym urządzeniu wyjściowym. Nie możemy bowiem „wyjść” poza region, na którym pozwolono nam rysować.

Możemy jednak zmienić sposób, w jaki po tym regionie będziemy się orientować. Jest to możliwe poprzez ustanowienie na nim jakiegoś układu współrzędnych oraz zmianę kadru.

Mapowanie układu współrzędnych

Biblioteka GDI zachowuje się w tym względzie bardzo porządnie, bo pozwala programiście na daleko posuniętą swobodę w wyborze pasującego mu układu. System współrzędnych jest bowiem tutaj czymś więcej niż tylko dwoma przecinającymi się osiami.

W GDI mamy pojęcie **trybu mapowania** (ang. *mapping mode*) układu współrzędnych. Tryb ten precyzuje nie tylko orientację płaszczyzny (kierunek osi pionowej), ale też wielkość jednostek, na które tę płaszczyznę podzielimy. Nazywamy je **jednostkami logicznymi** (ang. *logical units*), w przeciwieństwie do jednostek urządzenia (ang. *device units*) - na przykład pikseli.

W Windows GDI możemy ustawić jeden z kilku predefiniowanych trybów mapowania.

Ustawianie trybu mapowania

Do tego celu posługujemy się funkcją `SetMapMode()`:

```
int SetMapMode(HDC hdc,
               int fnMapMode);
```

Ponieważ omawiany tryb jest znowu ustawieniem powiązanim z kontekstem urządzeń uchwyt do niego należy podać w pierwszym parametrze. W drugim wpisujemy natomiast jedną ze stałych, identyfikującą wybrany tryb mapowania:

stała	tryb mapowania	rozmiar jednostki logicznej	zwrot osi	uwagi
MM_HIMETRIC	metryczny gęsty	0,01 milimetra	x → y ↑	Te tryby mogą być przydatne podczas zaawansowanego przetwarzania obrazów.
MM_LOMETRIC	metryczny luźny	0,1 milimetra		
MM_HIENGLISH	angielski gęsty	0,001 cala (0,025 milimetra)		Tryby te są używane zwykle podczas drukowania.
MM_LOENGLISH	angielski luźny	0,01 cala (0,25 milimetra)		
MM_TWIPS	twips	1/20 punktu drukarskiego (1/1440 cala - 0,018 milimetra)		
MM_ANISOTROPIC	anizotropowy	ustalany przez programistę	dowolny	Umożliwia dowolne ustawienie parametrów układu współrzędnych.
MM_ISOTROPIC	izotropowy			Dbaj o to, aby pionowy i poziomy rozmiar jednostek był taki sam.
MM_TEXT	piksele urządzenia	jeden piksel urządzenia	x → y ↓	Jest to domyślny tryb mapowania.

Tabela 59. Tryby mapowania układu współrzędnych w Windows GDI

Możemy więc mierzyć nasze rysunki w milimetrach, calach, pikselach (domyślne ustawienie), jak również w naszych własnych jednostkach, ustalanych *ad hoc*. Zobaczmy, jak możemy je zdefiniować.

Kadr

Jeżeli wybraliśmy jako tryb mapowania ustawienie `MM_ISOTROPIC` lub `MM_ANISOTROPIC`, wówczas możemy sami ustalić jednostkę oraz zwrot układu współrzędnych danego nam fragmentu płaszczyzny świata. We wszystkich przypadkach możemy także określić położenie punktu początkowego (0, 0) wybranego układu.

O wszystkich tych sprawach decydujemy, modyfikując właściwości **kadru** na płaszczyźnie świata.

Kadr (ang. *window*¹³⁹) określa orientację osi, położenie początku, zakres jednostek oraz ewentualnie ich rozmiar w układzie współrzędnych płaszczyzny świata.

Ustawienie kadru pozwala więc opisać podarowany nam kawałek płaszczyzny zgodnie ze swoimi życzeniami. Zobaczmy zatem, jak można to zrobić.

Pozycja kadru

Położenie kadru możemy regulować. Możliwe jest rozmieszczenie go w każdym punkcie wielkiej płaszczyzny świata - do tego celu służą funkcja `SetWindowOrgEx()`:

```
BOOL SetWindowOrgEx(HDC hdc,
                    int X,
                    int Y,
                    LPPOINT lpPoint);
```

W parametrach `x` i `y` podajemy jej punkt (w jednostkach logicznych), w którym zostanie umieszczony lewy górny róg kadru. Punkt ten będzie rzutowany na piksel (0, 0) przy przekształcaniu płaszczyzny świata na płaszczyznę urządzenia.

Domyślnie kadr jest położony w logicznych koordynatach (0, 0), które przekładają się bezpośrednio na koordynaty urządzenia - też (0, 0). Jeżeli wywołamy `SetWindowOrgEx()`, zmienimy to.

W `lpPoint` funkcja zwróci nam poprzednie ustawienie kadru (chyba że podamy tu `NULL`).

Rozciągłość osi

Maksymalny rozstaw osi układu współrzędnych w kadrze ustawiamy za pomocą `SetWindowExtEx()`:

```
BOOL SetWindowExtEx(HDC hdc,
                    int nXExtent,
                    int nYExtent,
                    LPSIZE lpSize);
```

Wywołanie tej funkcji przynosi jakikolwiek efekt tylko wtedy, gdy tryb mapowania jest ustawiony na `MM_ISOTROPIC` lub `MM_ANISOTROPIC`. Wówczas parametry `nXExtent` i `nYExtent` określają wartość na osiach współrzędnych układu, jakie są osiągane na krawędziach kadru.

¹³⁹ *window* znaczy oczywiście 'okno'. Z powodu naturalnego konfliktu ze znacznie częściej używanym w WinAPI znaczeniem tego słowa zdecydowałem, że w tym kontekście lepiej będzie użyć innego terminu. Padło na kadr.

Pamiętajmy, że ustawianie większych rozciągłości osi nie powoduje wcale zwiększenia faktycznego obszaru, na którym będziemy rysować. Funkcja `SetWindowExtEx()` służy bowiem zdefiniowaniu jednostek logicznych, jakich będziemy używać przy rysowaniu na płaszczyźnie świata. Zatem:

Podawanie większych wartości do funkcji `SetWindowExtEx()` nie spowoduje rozszerzenia obszaru rysowania, lecz zmniejszenie rozmiaru jednostek logicznych.

Sam obszar rysowania jest dany nam odgórnie (np. jako wnętrze okna o ustalonym rozmiarze) i nie możemy go na siłę powiększyć lub zmniejszyć. Możemy aczkolwiek podzielić go na tyle jednostek, ile chcemy - do tego służy opisywana funkcja.

W parametrze `lpSize` zwraca ona bieżące wymiary kadru.

Gdy trybem mapowania jest `MM_ISOTROPIC`, wtedy najlepiej byłoby, jeśli jednostki logiczne były takie same w pionie i poziomie. Inaczej Windows GDI sam o to zadba, co niekoniecznie musi być dobre. Aby temu zapobiec, możemy pobrać pierwotne wymiary kadru poprzez `GetWindowExtEx()`, a następnie przeskalować je, mnożąc przez ten sam czynnik. Wtedy aspekt obrazu zostanie zachowany.

O [SetWindowOrgEx\(\)](#) i [SetWindowExtEx\(\)](#) możesz rzecz jasna poczytać w MSDN.

Płaszczyzna urządzenia

Przedostatnim etapem potoku graficznego - tuż przed wyświetleniem obrazu - jest **płaszczyzna urządzenia** (ang. *device space*).

Na tę płaszczyznę rzutowana jest poprzednia (świata lub też strony), dzięki czemu rysunek jest ostatecznie przygotowywany pod względem, nazwijmy to, „geometrycznym”. Obejmuje to między innymi dostosowanie do układu współrzędnych urządzenia.

Układ współrzędnych

Na tej płaszczyźnie na ma już jednostek logicznych - są tylko **jednostki urządzenia** (ang. *device units*). Interpretacja, czym one rzeczywiście są, zależy ściśle od sprzętu. Dla monitora będą to pojedyncze piksele, zaś dla drukarki - punkty na papierze, itp.

Z nową płaszczyzną związany jest też inny układ współrzędnych. Tutaj nie możemy już go zmieniać, właśnie ze względu na wspomnianą już zależność od sprzętu.

Ten układ nie jest trudny do opanowania, bo spotykałeś się z nim już niejednokrotnie.

W układzie współrzędnych płaszczyzny urządzenia punkt (0, 0) jest umieszczony w **lewym górnym rogu**, zaś oś X biegnie **w prawo**, a Y - **w dół**.

Jest to więc taki sam system, jaki stosujemy dla określania położenia okien i innych elementów interfejsu Windows. Nie jest to przypadkowe: przecież sam interfejs także jest rysowany po ekranie monitora.

Wziernik

Na płaszczyźnie urządzenia występuje również pojęcie prostokąta podobnego do kadru. Jest to **wziernik**.

Wziernik (ang. *viewport*) określa miejsce na płaszczyźnie urządzenia, gdzie pojawi się wygenerowany obraz.

Pokrywa się on początkowo z całym obszarem, na którym możemy rysować przy pomocy danego kontekstu urządzenia. W ogromnej większości przypadków nie ma też najmniejszej potrzeby zmiany tego.

Czasem konieczne jest może tylko jego przesunięcie. Dokonujemy tego poprzez funkcję `SetViewportOrgEx()`:

```
BOOL SetViewportOrgEx(HDC hdc,
                      int X,
                      int Y,
                      LPPOINT lpPoint);
```

W argumentach `x` i `y` podajemy jej koordynaty punktu (w jednostkach urządzenia), w którym zostanie umieszczony rzut początku układu współrzędnych płaszczyzny świata (strony). Jest to więc taki piksel, któremu zostanie przyporządkowany punkt (0, 0) w jednostkach logicznych. Pozostała część płaszczyzny świata będzie rzutowana w odniesieniu do tego właśnie punktu.

Kontekst urządzenia

O ile potok graficzny może nie wydawać się ważną sprawą (bo i o jego istnieniu niekoniecznie trzeba być uświadomionym), o tyle druga kluczowa koncepcja Windows GDI jest dla programisty absolutnie niezbędna. Mowa tu o kontekście urządzenia.

Z terminem tym spotkaliśmy się już parę razy - zawsze wtedy, gdy chcieliśmy coś narysować w oknie programu. Nie ma w tym nic niezwykłego, bowiem do tego właśnie służy ów kontekst.

Kontekst urządzenia (ang. *device context*) jest strukturą Windows GDI przechowującą informacje na temat urządzenia graficznego. Na rzecz kontekstu można wywoływać funkcje GDI, tworząc w ten sposób obrazy wyświetlane na tym urządzeniu.

Konkretny kontekst może więc być związany z monitorem, drukarką mozaikową, ploterem, planszą rzutnika slajdów czy nawet tablicą do internetowych konferencji. Jego użytkowanie w każdym z tych przypadków wygląda jednak bardzo podobnie i to jest jedną z głównych zalet biblioteki Windows GDI.

Kontekst urządzenia jest strukturą, a sądząc z pełnionych przez siebie zadań - strukturą bardzo skomplikowaną. Jej złożoność jest na szczęście problemem Windows, a nie naszym. Programista nie musi bowiem operować bezpośrednio na kontekście urządzenia; właściwie byłoby to zupełnie niewskazane, jako że nieuchronnie zatarłoby niezależność sprzętową, wpisaną w idee Windows GDI.

Zamiast samej struktury będziemy więc działać tylko przy pomocy **uchwyty** do niej. Naszą najważniejszą daną przy rysowaniu będzie w takim razie **uchwyt do kontekstu urządzenia** (ang. *handle to device context*). Ją też będziemy podawać, chcąc cokolwiek wyświetlić, zmienić parametry rysowania lub pobrać informacje o urządzeniu.

Dla wygody często utożsamia się kontekst urządzenia z jego uchwytem, ponieważ wspomniana wewnętrzna struktura Windows GDI nie jest używana przez programistę i liczy się tylko uchwyt do niej. Dlatego też jeśli dalej będę mówił o kontekście urządzenia, to prawie na pewno będę miał na myśli jego uchwyt (chyba że wyraźnie zaznaczę coś innego).

Uchwyt do kontekstu jest w gruncie rzeczy podobny do dziesiątków innych rodzajów uchwytów w Windows API. Jest to więc liczba 32-bitowa, dla której przewidziano osobny typ: `HDC`. Jak już może zdążyłeś zauważyć, wszystkie widziane przez ciebie dotąd funkcje GDI (np. `DrawText()` czy `MoveToEx()`) żądały przynajmniej jednego parametru tego typu. W ten sposób wiedzą one, gdzie dokładnie mają wykonać żądane operacje graficzne.

Gdyby kontekst urządzenia był klasą języka C++, to w zasadzie wszystkie funkcje Windows GDI działałyby jako metody tej klasy. Niestety, interfejsu GDI (jak i całego WinAPI) nie napisano w C++, więc sytuacja wygląda nieco gorzej. Niemniej, uchwyt do kontekstu urządzenia można w logiczny sposób utożsamiać ze wskaźnikiem `this`, jaki otrzymują przy wywołaniu metody obiektów. Takie „pseudoobiektywne” podejście, stawiające uchwyty w takiej roli jak obiekty OOP, jest zresztą charakterystyczne dla całego Windows API i sprawdza się całkiem dobrze. Jak zobaczysz niedługo, z pewnymi oporami można tak symulować nawet dziedziczenie i polimorfizm metod wirtualnych.

Pierwszą czynnością rysowania przy użyciu Windows GDI powinno być zatem uzyskanie skądś uchwytu do kontekstu urządzenia, gdyż bez niego nie zrobimy zgoła nic. Później można już dokonywać tych wszystkich wspaniałych rzeczy, o których traktuje większa część aktualnego rozdziału.

Rozpocznijmy więc od pobrania uchwytu kontekstu urządzenia.

Pobieranie uchwytu

Uchwyt do kontekstu urządzenia można zdobyć wieloma różnymi drogami.

Najlogiczniejsze wydawałoby się powiadomienie Windows, z którego urządzenia graficznego chcemy skorzystać, a system zwróciłby nam wtedy uchwyt do niego.

Faktycznie jest to możliwe i pokażę później, jak to zrobić.

W Windows częściej jednak będzie używali kontekstów pochodzących z innych źródeł, związanych z ekranem. Mam tu na myśli konteksty urządzeń powiązane z oknami Windows lub ich fragmentami (obszarami klienta). Właśnie tego rodzaju uchwytów graficznych używaliśmy dotąd, gdy w poprzednich rozdziałach rysowaliśmy cokolwiek w oknie naszych programów przykładowych.

Ostatnią możliwością jest samodzielne utworzenie kontekstu na podstawie innego, już istniejącego. Jest to bardzo przydatne przy operowaniu bitmapami, więc o tym także sobie powiemy.

Poznajmy zatem te trzy metody pobierania uchwytów kontekstu urządzenia.

Od okna

Chcąc narysować cokolwiek w oknie, musimy pobrać kontekst odnoszący do niego. Jest to wykonalne na kilka sposobów.

Podczas odrysowywania

Pierwszą możliwość wyraźnie podsuwa nam sam system operacyjny: jest to moment koniecznego odrysowania zawartości okna. Kontekst urządzenia związany z oknem możemy bowiem bez problemów pobrać podczas obsługi komunikatu `WM_PAINT`.

Wiemy już zresztą, jak to zrobić. Przy omawianiu tego komunikatu poznaliśmy mianowicie funkcję `BeginPaint()`, która do tego właśnie służy:

```
HDC hdcKontekst;  
PAINTSTRUCT ps;  
  
hdcKontekst = BeginPaint(hWnd, &ps);
```


Jej wywołanie zwraca w wyniku żądany kontekst; ponadto zawiera go także pole `hdc` struktury `PAINSTRUCT`, której wskaźnik podajemy do `BeginPaint()`.

Kontekst, jaki w ten sposób uzyskujemy, jest krótkożyjący i traci ważność po odrysowaniu zawartości. Jak wiemy, czynność tę kończymy poprzez przywołanie `EndPaint()`:

```
EndPaint (hWnd, &ps);
```

Po nim odświeżanie okna jest już zakończone, a kontekst pobrany na początku nie nadaje się do żadnego użytku. Wówczas kończymy więc obsługę `WM_PAINT` i nasza zabawa na tym się kończy.

Obszar klienta okna

Uchwyt kontekstu okna możemy pobrać nie tylko przy reagowaniu na komunikat `WM_PAINT`. Równie dobrze moglibyśmy uzyskać w dowolnej sytuacji - wystarczy posłużyć się funkcją `GetDC()`:

```
HDC hdcObszarKlienta = GetDC(hWnd);
```

Wymaga ona tylko podania uchwytu okna, a w zamian oddaje kontekst urządzenia, odnoszący się do jego **obszaru klienta**.

Po zakończeniu pracy z kontekstem należy go zwykle zwolnić, ponieważ zazwyczaj nie jest to twór trwały. Windows tworzy go tymczasowo, dla nas, i dlatego należy mu powiedzieć, kiedy już go nie potrzebujemy. Robimy to poprzez wywołanie `ReleaseDC()`:

```
ReleaseDC (hWnd, hdcObszarKlienta);
```

Jest to **zalecane w każdym przypadku**, bo całkowicie zapobiega ewentualnym wyciekom zasobów (kontekst urządzenia jest przecież zasobem systemowym).

Zwolnienie uchwytu nie jest aczkolwiek konieczne w przypadku, gdy jest to prywatny kontekst okna lub kontekst wspólny dla całej klasy okien. Te dwa przypadki zachodzą, kiedy przy rejestrowaniu klasy okna dołączymy (odpowiednio) `CS_OWNDC` lub `CS_CLASSDC` do jej stylu (pola `WNDCLASS[EX]::style`).

Ponieważ jednak w przypadku tego rodzaju sytuacji `ReleaseDC()` nie robi nic, stosowanie tego wywołania jest bardzo rozsądne niezależnie od okoliczności.

Całe okno

`GetDC()` pozwala nam bawić się z obszarem klienta okna - i tylko z nim. Pozostała jego część, czyli obszar pozakliencki, jest wtedy poza naszym zasięgiem. Jeśli jednak chcemy zająć się także i tym rejonem, potrzebujemy kontekstu dla **całego okna**. Pozyskujemy go funkcją `GetWindowDC()`:

```
HDC hdcOkno = GetWindowDC(hWnd);
```

Otrzymany tą drogą kontekst pokrywa obszar nie tylko wnętrza okna `hWnd`, ale też jego paska tytułu, menu czy brzegów. Posługiwanie się nim należy więc do sytuacji raczej specjalnych, gdyż te części okna są ważne dla systemu Windows.

Po zakończeniu pracy z kontekstem należy go zwolnić, a posługujemy się do tego poznaną przed chwilą funkcją `ReleaseDC()`:

```
ReleaseDC (hWnd, hdcOkno);
```

Dla kontekstu obejmującego całe okno trzeba ją wywołać **zawsze**, aby uniknąć niepożądanego zjawiska wycieku zasobów.

Do uzyskiwania kontekstu urządzenia związanego z oknem możliwe jest też użycie funkcji `GetDCEx()`. Jest ona bardzo elastyczna i zależnie od swych parametrów może zachowywać się jak `GetDC()`, `GetWindowDC()`, a nawet jak `BeginPaint()`. Oferuje też pewne dodatkowe możliwości (pomocne np. przy obsłudze `WM_NCPAINT`); na temat ich wszystkich możesz co nieco poczytać w dokumentacji [MSDN](#).

Od urządzenia

Drugim źródłem kontekstów urządzeń są... urządzenia :) W Windows możemy przy ich pomocy uzyskać dostęp do przyłączonego do komputera sprzętu graficznego.

Ekran

Bardzo proste jest pobranie kontekstu urządzenia pokrywającego **cały ekran**. Mówiąc 'ekran' nie mam na myśli pulpitu systemowego, lecz dosłownie ekran monitora: czyli to co widzimy patrząc w „telewizor”, tj. zarówno pulpit, jak i okna, które go ewentualnie przykrywają. Kontekst całego ekranu pozwala zatem na wtrącanie się w wygląd innych aplikacji, więc rysowanie po nim nie należy do dobrego tonu. Przy jego pomocy można jednak wykonywać inne, całkiem „kulturalne” i przydatne operacje. Jedną z nich za chwilę zaprezentuję.

Na razie dowiedzmy się, jak pozyskać taki specjalny kontekst. Jak mówiłem nie jest to trudne i ogranicza się do wywołania jednej prostej funkcji, w dodatku już nam znanej. Tą funkcją jest `GetDC()`:

```
HDC hdcEkran = GetDC(NULL);
```

Zamiast uchwytu okna podajemy jej wartość `NULL`, czyli zero. W zamian dostajemy kontekst dla całego ekranu¹⁴⁰.

Skoro jednak nie powinniśmy po nim rysować, to cóż sensownego da się z nim zrobić? Otóż da się całkiem sporo; najciekawsze jest chyba stworzenie aplikacji **pobieracza kolorów** (ang. *color picker*). Oto, jak może ona wyglądać:

```
// ColorPicker - pobieracz kolorów

#include <string>
#include <sstream>
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <windowsx.h>

// dane okna
std::string g_strKlasaOkna = "od0dogk_ColorPicker_Window";
HWND g_hwndOkno = NULL;

// uchwyt do kontekstu ekranu
HDC g_hdcEkran = NULL;

// pobrany kolor
```

¹⁴⁰ Zamiast `GetDC()`, można też użyć `GetWindowDC()` (także podając jej `NULL`), ale wynik byłby inny w systemach wielomonitorowych. Tam `GetWindowDC()` zwróciłaby kontekst głównego monitora, zaś `GetDC()` w przedstawionej formie oddaje zawsze kontekst dla całego wirtualnego ekranu - niezależnie od tego, na ile rzeczywistych monitorów się on rozciąga.

```
COLORREF g_clKolor = RGB(255, 255, 255);    // początkowo biały

// ----- procedura zdarzeniowa okna -----

LRESULT CALLBACK WindowEventProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_LBUTTONDOWN:
            // łapiemy myszkę
            SetCapture (hWnd);

            // ustawiamy kursor w kształcie celownika
            SetCursor (LoadCursor(NULL, IDC_CROSS));
            return 0;

        case WM_MOUSEMOVE:
            // sprawdzamy, czy myszka jest złapana
            if (GetCapture() == hWnd)
            {
                // odcytujemy współrzędne kursora
                POINT ptKursor;
                ptKursor.x = GET_X_LPARAM(lParam);
                ptKursor.y = GET_Y_LPARAM(lParam);

                // przeliczamy je na koordynaty ekranowe
                ClientToScreen (hWnd, &ptKursor);

                // pobieramy kolor z miejsca kursora
                g_clKolor = GetPixel(g_hdcEkran,
                                     ptKursor.x, ptKursor.y);

                // wymuszamy odświeżenie okna programu,
                // aby pokazać pobrany kolor
                InvalidateRect (hWnd, NULL, TRUE);
            }
            return 0;

        case WM_LBUTTONUP:
            // uwalniamy mysz
            ReleaseCapture();

            // ustawiamy kursor strzałki
            SetCursor (LoadCursor(NULL, IDC_ARROW));
            return 0;

        case WM_PAINT:
        {
            // odrysowanie zawartości okna
            {
                PAINTSTRUCT ps;
                HDC hdcOkno;

                // zaczynamy
                hdcOkno = BeginPaint(hWnd, &ps);

                // pobieramy obszar klienta okna
                RECT rcObszarKlienta;
                GetClientRect (hWnd, &rcObszarKlienta);
            }
        }
    }
}
```

```

        // wypełniamy go pobranym kolorem
        // w tym celu najpierw tworzymy odpowiedni pędzel,
        // a potem wypełniamy prostokąt obszaru klienta
        // potem usuwamy pędzel
        HBRUSH hbrPedzel = CreateSolidBrush(g_clKolor);
        FillRect (hdcOkno, &rcObszarKlienta, hbrPedzel);
        DeleteObject (hbrPedzel);

        // kończymy rysowanie
        EndPaint (hWnd, &ps);
    }

    // pokazanie składowych koloru
    {
        // pobieramy te składowe i konwertujemy na napis
        std::stringstream Strumien;
        Strumien << "RGB: " << (int) GetRValue(g_clKolor)
            << ", " << (int) GetGValue(g_clKolor)
            << ", " << (int) GetBValue(g_clKolor);

        // ustawiamy ten napis jako tytuł okna programu
        SetWindowText (hWnd, Strumien.str().c_str());
    }

    return 0;
}

case WM_DESTROY:
    // zwalniamy kontekst ekranu
    ReleaseDC (NULL, g_hdcEkran);

    // kończymy program
    PostQuitMessage (0);
    return 0;
}

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

// -----funkcja WinMain() -----

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow)
{
    /* rejestrujemy klasę okna */

    WNDCLASSEX KlasaOkna;

    // wypełniamy strukturę WNDCLASSEX
    ZeroMemory (&KlasaOkna, sizeof(WNDCLASSEX));
    KlasaOkna.cbSize = sizeof(WNDCLASSEX);
    KlasaOkna.hInstance = hInstance;
    KlasaOkna.lpfnWndProc = WindowEventProc;
    KlasaOkna.lpszClassName = g_strKlasaOkna.c_str();
    KlasaOkna.hCursor = LoadCursor(NULL, IDC_ARROW);
    KlasaOkna.hIcon = LoadIcon(NULL, IDI_APPLICATION);

    // rejestrujemy klasę okna
    RegisterClassEx (&KlasaOkna);

```

```

/* tworzymy okno */

// tworzymy okno funkcja CreateWindowEx
g_hwndOkno = CreateWindowEx(WS_EX_TOOLWINDOW,
                            g_strKlasaOkna.c_str(),
                            NULL,
                            WS_OVERLAPPED | WS_BORDER
                            | WS_CAPTION | WS_SYSMENU,
                            0, 0,
                            125,
                            80,
                            NULL,
                            NULL,
                            hInstance,
                            NULL);

// pokazujemy nasze okno i je od razu odświeżamy
ShowWindow (g_hwndOkno, nCmdShow);
UpdateWindow (g_hwndOkno);

/* pobieramy kontekst urządzenia ekranu */
g_hdcEkran = GetDC(NULL);

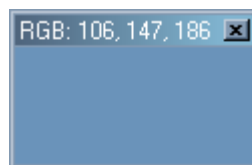
/* pętla komunikatów */

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

// zwracamy kod wyjścia
return static_cast<int>(msgKomunikat.wParam);
}

```

Program ten potrafi pobrać kolor z dowolnego piksela na ekranie, który zostanie mu wskazany przez użytkownika. W tym celu należy po prostu kliknąć lewym przyciskiem myszy i przeciągnąć kursor do wybranego punktu. Jego kolor ukaże się w oknie programu, poznamy również jego składowe RGB:



Screen 67. Okno pobieracza kolorów

Ta prosta aplikacja nie jest wcale tak nieużyteczna, jakby się mogło z początku wydawać. Gdyby ją odrobinę ulepszyć, stałaby się przydatnym narzędziem dla grafików i webmasterów. Często korzystają oni z takich właśnie programów.

Działanie naszego pobieracza jest dość proste. Najpierw, zaraz po utworzeniu własnego okna, pobiera on kontekst urządzenia całego ekranu w opisany wcześniej sposób:

```
g_hdcEkran = GetDC(NULL);
```

Używając `go`, może już pobierać kolory pikseli ekranu. Przy wciśnięciu lewego przycisku myszy przejmuje więc kontrolę nad urządzeniem wskazującym, ponieważ będzie chciał rejestrować pozycję kursora także poza własnym oknem. Po co? Po to, żeby wyłować kolor piksela w miejscu kursora, co też czyni poniższymi wierszami z kodu obsługi

`WM_MOUSEMOVE`:

```
POINT ptKursor;
ptKursor.x = GET_X_LPARAM(lParam);
ptKursor.y = GET_Y_LPARAM(lParam);

// przeliczamy pozycję kursora na koordynaty ekranowe
ClientToScreen (hWnd, &ptKursor);

// pobieramy kolor z miejsca kursora
g_clKolor = GetPixel(g_hdcEkran, ptKursor.x, ptKursor.y);
```

Najważniejsze zadanie spoczywa tu na funkcji `GetPixel()`. Nietrudno domyślić się tutaj jej działania, ale wyjaśnimy je sobie także w kolejnym podrozdziale.

Pokazaniem pobranego koloru oraz jego składowych RGB zajmuje się kod komunikatu `WM_PAINT`. Wypełnia on obszar klienta okna programy rzeczonym kolorem:

```
// pobieramy obszar klienta okna
RECT rcObszarKlienta;
GetClientRect (hWnd, &rcObszarKlienta);

// wypełniamy go pobranym kolorem
HBRUSH hbrPedzel = CreateSolidBrush(g_clKolor);
FillRect (hdcOkno, &rcObszarKlienta, hbrPedzel);
DeleteObject (hbrPedzel);
```

Dokonuje tego, tworząc tzw. pędzel i posługując się nim do wypełnienia prostokąta (`FillRect()`) definiującego wnętrze okna. O pędzlach również powiemy sobie więcej w następnym podrozdziale.

Dalej następuje jeszcze pobranie wartości kanałów RGB koloru i ich wyświetlenie na pasku tytułowym okna:

```
// pobieramy te składowe i konwertujemy na napis
std::stringstream Strumien;
Strumien << "RGB: " << (int) GetRValue(g_clKolor) << ", "
          << (int) GetGValue(g_clKolor) << ", "
          << (int) GetBValue(g_clKolor);

// ustawiamy ten napis jako tytuł okna programu
SetWindowText (hWnd, Strumien.str().c_str());
```

Widzimy tu praktyczne spożytkowanie makr `Get?Value()`.

Dowolne urządzenie

W ogólnym przypadku, uchwyt do kontekstu możemy utworzyć dla dowolnego urządzenia. Jest możliwe przy użyciu funkcji `CreateDC()`:

```
HDC CreateDC(LPCTSTR lpszDriver,
             LPCTSTR lpszDevice,
             LPCTSTR lpszOutput,
             CONST DEVMODE* lpInitData);
```


Funkcja ta jest bardziej skomplikowana niż to by się mogło wydawać z jej prototypu, dlatego też nie będziemy jej dokładnie omawiać. Wyjaśnimy sobie aczkolwiek znaczenie poszczególnych parametrów:

<i>typ</i>	<i>parametry</i>	<i>opis</i>
LPCTSTR	lpszDriver lpszDevice	Te dwa napisy określają łącznie urządzenie , którego kontekst chcemy pobrać. W <code>lpszDriver</code> wpisujemy nazwę sterownika - dokumentacja wspomina o dwóch możliwościach: "DISPLAY" dla monitora oraz "WINSPOOL" ("WINSPL16") dla drukarki. Możliwe jest też wpisanie tam nazwy konkretnego modelu (np. "HP Color LaserJet 1500L"), jednak podaje się ją zwykle w <code>lpszDevice</code> , zostawiając wówczas pierwszy parametr z wartością <code>NULL</code> . W ogóle niemal zawsze wykorzystuje się tylko jeden z tych dwóch parametrów, podając zero (<code>NULL</code>) w drugim.
	lpszOutput	Jest to parametr zachowany celem kompatybilności z 16-bitowymi wersjami Windows. Obecnie należy tu zawsze wpisywać <code>NULL</code> .
CONST DEVMODE*	lpInitData	Są to dodatkowe parametry dla urządzenia innego niż monitor (czyli zwykle dla drukarki). <code>DEVMODE</code> , na co pokazuje ten wskaźnik, jest skomplikowaną strukturą zawierającą te pomocnicze dane. Jeżeli w <code>lpszDriver</code> wpisujemy "DISPLAY", wtedy tutaj musimy podać wartość <code>NULL</code> .

Tabela 60. Parametry funkcji `CreateDC()`

Najprostsze użycie tej funkcji to pobranie za jej pośrednictwem kontekstu urządzenia ekranu:

```
HDC hdcEkran = CreateDC("DISPLAY", NULL, NULL, NULL);
```

Wynik jest wtedy identyczny z tym uzyskanym poprzez `GetDC(NULL)`.

Pobranie kontekstu drukarki jest o wiele trudniejsze, gdyż musimy wtedy podać pełną nazwę tego urządzenia. W systemie może być bowiem zainstalowanych wiele drukarek, niekoniecznie fizycznie istniejących, ale np. drukujących do pliku; poza tym fakty są także interpretowane jako drukarki.

Utworzenie kontekstu wymaga więc najpierw wyliczenia wszystkich tego typu urządzeń, a następnie wybrania jednego z nich. Inną metodą jest pozostawienie wyboru użytkownikowi poprzez wyświetlenie odpowiedniego okna dialogowego (za pomocą funkcji `PrintDlg[Ex]()`), a następnie pobranie otrzymanego wyniku.

Tak czy owak nie jest to proste, lecz jeśli interesują się szczegóły, to odsyłam do [MSDN](#) lub innych źródeł informacji o Windows GDI.

`CreateDC()` ma kuzyna w postaci funkcji [CreateIC\(\)](#). Funkcja ta przyjmuje identyczne parametry, ale jej wynik jest uchwyt do tzw. **kontekstu informacyjnego** (ang. *information context*). Różni się on od kontekstu urządzenia tym, iż nie można przy jego pomocy niczego narysować, a jedynie pobrać informacje o samym urządzeniu. Kontekst informacyjny może więc być tylko i wyłącznie przekazany do funkcji w rodzaju [GetDeviceCaps\(\)](#), [DeviceCapabilities\(\)](#) czy [DocumentProperties\(\)](#). Można się domyślić, że utworzenie kontekstu informacyjnego jest szybsze od stworzenia „pełnowymiarowego” kontekstu urządzenia. Rzeczywiście, tak właśnie jest.

Kontekst pamięciowy

Istnieje jeszcze jedna metoda pozyskania kontekstu urządzenia, niepodobna do żadnej wcześniej. Także jej produkt jest odmienny - to **kontekst pamięciowy** (ang. *memory context*).

Taki kontekst nie jest bezpośrednio związany z żadnym urządzeniem. Jest on raczej czymś w rodzaju bufora - pomocniczego obiektu, ułatwiającego (a często wręcz umożliwiającego) operacje na obrazach.

Pamięciowego kontekstu nie bierzemy znikąd. Możemy go utworzyć tylko nad podstawie już istniejącego, innego kontekstu. Wówczas ten nowy będzie z nim **kompatybilny**, co też sugeruje nazwa funkcji `CreateCompatibleDC()`:

```
// zakładamy, że w hdcKontekst mamy już jakiś kontekst urządzenia
HDC hdcKontekstPamieciowy = CreateCompatibleDC(hdcKontekst);
```

Funkcji tej podajemy uchwyt do posiadanego kontekstu, a w zamian dostajemy nowy kontekst pamięciowy, kompatybilny z podanym. Możemy też podać `NULL`, a wtedy otrzymamy kontekst zgodny z ekranem.

Ustawianie obszaru rysowania

Musimy koniecznie zwrócić uwagę na to, że kompatybilny kontekst pamięciowy **nie jest kopią** pierwotnego kontekstu. Jest on z nim jedynie zgodny, to znaczy może być używany do wymiany danych ze swą matrycą. Absolutnie jednak nie zawiera on kopii samego rysunku.

Nie mógłby zresztą jej zawierać, bo jego obszar rysowania jest początkowo znikomy: ma on postać monochromatycznej bitmapy o wymiarach 1×1. Taki obszar nie jest szczególnie przydatny, zatem należy go powiększyć. W tym celu można stworzyć nową bitmapę - o tak:

```
HBITMAP hbmpBitmapa = CreateCompatibleBitmap(hdcKontekst,
                                              nSzerokosc, nWysokosc);
```

Jest bardzo ważne, aby do `CreateCompatibleBitmap()` przekazać uchwyt do „starego” kontekstu, a nie do kontekstu pamięciowego.

Po utworzeniu nowej bitmapy (obszaru rysowania), wiążemy ją z naszym kontekstem pamięciowym. Jednocześnie zachowujemy tę poprzednią, jednopikselową:

```
HBITMAP hbmpStaraBitmapa = (HBITMAP) SelectObject(hdcKontekstPamieciowy,
                                                    hbmpBitmapa);
```

Teraz możemy już normalnie korzystać z kontekstu pamięciowego - zupełnie tak, jakby był on np. kontekstem wnętrza okna o wymiarach `nSzerokosc` i `nWysokosc`. Różnica polega na tym, że nie zobaczymy nigdzie efektów rysowania - chyba że świadomie je przekopiujemy do pierwotnego kontekstu. Jak to zrobić, dowiesz się przy omawianiu bitmap GDI.

Jest jeszcze jedno źródło kontekstu urządzenia - to **metaplik** (ang. *metafile*), czyli plik dyskowy zawierający zapis poleceń Windows GDI. Nie będziemy tutaj zajmować się metaplikami; możesz poczytać na ich temat w [MSDN](#), jeśli chcesz.

Atrybuty kontekstu

Co można powiedzieć o gotowym kontekście urządzenia? Na pewno to, że charakteryzuje go spora ilość różnych właściwości. Opiszę je teraz krótko. Większością z nich zajmiemy się dokładniej w dalszej części rozdziału.

Atrybuty potoku graficznego

O potoku graficznym traktuje bliżej poprzednia sekcja. Tutaj przedstawię tylko skrótowo trzy ustawienia przynależne każdemu kontekstowi urządzenia.

Tryb mapowania

<i>Domyślne ustawienie:</i>	MM_TEXT
<i>Funkcja ustawiająca atrybut:</i>	SetMapMode()
<i>Funkcja pobierająca atrybut:</i>	GetMapMode()

Tryb mapowania określa wielkość jednostek logicznych, czyli jednostek w płaszczyźnie świata (strony) dla danego kontekstu urządzenia. Domyślnie jednostkami te są identyczne z punktami fizycznego urządzenia, czyli na przykład z pikselami.

Kadr

	<i>pozycja</i>	<i>rozciągłość</i>
<i>Domyślne ustawienie:</i>	(0, 0)	(1, 1)
<i>Funkcja ustawiająca atrybut:</i>	SetWindowOrgEx()	SetWindowExtEx()
<i>Funkcja pobierająca atrybut:</i>	GetWindowOrgEx()	GetWindowExtEx()

Kadr jest specjalnym prostokątem na płaszczyźnie świata (strony), który podczas przekształcania obrazu na płaszczyznę urządzenia jest rzutowany do odpowiadającego mu wzziernika.

Wziernik

	<i>pozycja</i>	<i>rozciągłość</i>
<i>Domyślne ustawienie:</i>	(0, 0)	(1, 1)
<i>Funkcja ustawiająca atrybut:</i>	SetViewportOrgEx()	SetViewportExtEx()
<i>Funkcja pobierająca atrybut:</i>	GetViewportOrgEx()	GetViewportExtEx()

Wziernik jest prostokątem położonym na płaszczyźnie urządzenia. Określa on, gdzie pojawi się wygenerowany obraz.

Atrybuty pióra

Pióro jest obiektem kontrolującym styl i grubość linii używanej do rysowania prostych i krzywych. Jest on także odpowiedzialny za obramowanie figur zamkniętych. Szczegółowe informacje o piórze uzyskasz z następnego podrozdziału.

Obiekt pióra

<i>Domyślne ustawienie:</i>	BLACK_PEN
<i>Funkcja ustawiająca atrybut:</i>	SelectObject()
<i>Funkcja pobierająca atrybut:</i>	GetCurrentObject()

Pióra mogą istnieć niezależnie od kontekstu urządzenia, jednak każdy kontekst posiada dokładnie jedno pióro, którego używa do rysowania. Domyślnie jest to twór rysujący czarne kreski o grubości 1 piksela.

Aktualna pozycja

<i>Domyślne ustawienie:</i>	(0, 0)
	MoveToEx()
	LineTo()
<i>Funkcje ustawiające atrybut:</i>	PolylineTo()
	PolyBezierTo()
	ArcTo()
<i>Funkcja pobierająca atrybut:</i>	GetCurrentPositionEx()

Aktualna pozycja pióra jest czymś w rodzaju graficznego kursora. Wiele funkcji rysujących krzywe (cztery wymienione wyżej) rozpoczyna od tego właśnie miejsca. Mają one aczkolwiek swoje odpowiedniki, które nie polegają na pozycji pióra. Których użyjemy - jest to w dużej mierze kwestia gustu.

Tryb rysowania

<i>Domyślne ustawienie:</i>	R2_COPYPEN
<i>Funkcja ustawiająca atrybut:</i>	SetROP2 ()
<i>Funkcja pobierająca atrybut:</i>	GetROP2 ()

Tryb rysowania piórem mówi bibliotece GDI, jak ma się obchodzić z problemem przykrywania już narysowanych pikseli z tymi, które ma zamiar zakreślić pióro. Standardowo, nowe piksele całkowicie zasłaniają stare, lecz można tak poinstruować GDI, aby zamiast tego dokonywana była odpowiednia operacja maskowania na bitach koloru pióra oraz ekranu.

Atrybuty pędzla

Pędzel odpowiada za wypełnianie zamkniętych kształtów, takich jak figury geometryczne. W szczególnych przypadkach może też służyć do kreślenia wzorzystych obramowań. Obszerne informacje o pędzlach znajdują się w następnym podrozdziale.

Obiekt pędzla

<i>Domyślne ustawienie:</i>	WHITE_BRUSH
<i>Funkcja ustawiająca atrybut:</i>	SelectObject ()
<i>Funkcja pobierająca atrybut:</i>	GetCurrentObject ()

Każdy kontekst urządzenia posiada dokładnie jeden związany z nim pędzel. Jeżeli nie ustalimy inaczej, jest to obiekt wypełniający regiony jednolitym białym kolorem.

Punkt odniesienia pędzla

<i>Domyślne ustawienie:</i>	(0, 0)
<i>Funkcja ustawiająca atrybut:</i>	SetBrushOrgEx ()
<i>Funkcja pobierająca atrybut:</i>	GetBrushOrgEx ()

Punkt odniesienia jest stosowany tylko dla pędzli, które malują regiony sąsiadującymi kopiami bitmap lub predefiniowanymi deseniami. Dla takich pędzli punkt odniesienia kontroluje układanie się stworzonego w ten sposób wzoru na wypełnianych nim powierzchniach.

Atrybuty bitmap

Operacje na bitmapach są jednym z ważniejszych działań podejmowanych przy użyciu funkcji Windows GDI. Kontekst urządzenia posiada dwa atrybuty związane z bitmapami.

Bitmapa kontekstu urządzenia

<i>Domyślne ustawienie:</i>	monochromatyczna bitmapa 1×1 lub bitmapa o parametrach zależnych od macierzystego urządzenia kontekstu
<i>Funkcja ustawiająca atrybut:</i>	SelectObject ()
<i>Funkcja pobierająca atrybut:</i>	GetCurrentObject ()

Każdy kontekst urządzenia jest związany z pewną bitmapą. Funkcje rysujące zmieniają piksele tej właśnie bitmapy. Możliwa jest aczkolwiek całościowa zmiana obrazka na inny, na przykład na zawartość pliku graficznego. Takie postępowanie jest bardzo częste w przypadku kontekstów pamięciowych, używanych do prezentacji bitmap lub ich fragmentów na ekranie.

Tryb rozciągania

<i>Domyślne ustawienie:</i>	BLACKONWHITE
<i>Funkcja ustawiająca atrybut:</i>	SetStretchBltMode()
<i>Funkcja pobierająca atrybut:</i>	GetStretchBltMode()

Właściwość ta definiuje sposób rozciągania bitmap przy kopiowaniu ich za pomocą funkcji `StretchBlt()`. Ustawiając ten atrybut możemy w pewnym zakresie decydować, jak zmieniają się piksele kopiowanego obrazka przy zmianie jego wymiarów. Zazwyczaj jednak żadne ustawienie nie daje zbyt dobrych rezultatów.

Atrybuty tekstu

Dobra biblioteka graficzna powinna umożliwiać manipulację tekstem. Windows GDI nie jest tu wyjątkiem, a konteksty urządzenia zawierają kilka atrybutów związanych z tymi możliwościami.

Kolor tekstu

<i>Domyślne ustawienie:</i>	czarny (RGB(0, 0, 0))
<i>Funkcja ustawiająca atrybut:</i>	SetTextColor()
<i>Funkcja pobierająca atrybut:</i>	GetTextColor()

Tego ustawienia chyba nie trzeba wyjaśniać. Zauważmy tylko, że kolor tekstu jest niezależny od czcionki i jej stylu (patrz niżej).

Kolor tła

<i>Domyślne ustawienie:</i>	biały (RGB(255, 255, 255))
<i>Funkcja ustawiająca atrybut:</i>	SetBkColor()
<i>Funkcja pobierająca atrybut:</i>	GetBkColor()

Kolor tła jest kolorem wypełnienia najmniejszego prostokąta okalającego tekst wypisywany w kontekście urządzenia. Zwykle nie chcemy żadnego wypełnienia w tym rejonie, a to wymaga ustawienia trybu tła - o czym pisze poniżej.

Tryb tła

<i>Domyślne ustawienie:</i>	OPAQUE
<i>Funkcja ustawiająca atrybut:</i>	SetBkMode()
<i>Funkcja pobierająca atrybut:</i>	GetBkMode()

Tryb tła to ustawienie precyzujące, czy tło tekstu ma być rysowane (OPAQUE - domyślnie), czy też nie (TRANSPARENT). Jeśli wybierzemy drugą możliwość, to oczywiście nie zobaczymy koloru tła (poprzedni atrybut) wokół tekstu.

Zarówno kolor, jak i tryb tła są używane jeszcze w kilku innych sytuacjach niezwiązanych z tekstem, np. podczas wypełniania deseniowymi pędzlami. O większości tych sytuacji dowiesz się w stosownym czasie.

Czcionka

<i>Domyślne ustawienie:</i>	SYSTEM_FONT
<i>Funkcja ustawiająca atrybut:</i>	SelectObject()
<i>Funkcja pobierająca atrybut:</i>	GetCurrentObject()

W GDI, mając na myśli czcionkę, myślimy także o jej stylu czy dekoracji znaków. Zatem „Verdana” jest nie jest w tym rozumieniu czcionką, ale „Verdana, rozmiar 10 punktów, pogrubiona, kursywa, nachylenie 0°, ...” - jak najbardziej. Czcionki tworzymy, wybieramy i usuwamy podobnie jak pędzle i pióra. W danej chwili kontekst urządzenia jest związany z dokładnie jedną czcionką.

Odstęp między znakami

<i>Domyślne ustawienie:</i>	0
<i>Funkcja ustawiająca atrybut:</i>	SetTextCharacterExtra()
<i>Funkcja pobierająca atrybut:</i>	GetTextCharacterExtra()

Ten atrybut jest dokładnie tym, o czym mówi jego nazwa. Odstęp między pojedynczymi znakami wypisywanego tekstu jest tu podawany w jednostkach logicznych.

Atrybuty regionów

Regiony są zespołami zamkniętych figur, używanymi do ograniczania obszaru, który podlega rysowaniu. Przy ich pomocy można łatwiej wykonać pewne czynności graficzne, które inaczej byłyby skomplikowane.

Kontekst urządzenia ma około jeden atrybut, odnoszący się do regionów.

Region przycinania

<i>Domyślne ustawienie:</i>	cały obszar rysowania SelectObject() SetClipRgn()
<i>Funkcje ustawiające atrybut:</i>	IntersectClipRect() OffsetClipRgn() ExcludeClipRect() SelectClipPath()
<i>Funkcje pobierające atrybut:</i>	GetClipRgn() GetClipBox()

Region przycinania definiuje obszar rysunku, które zostanie wyświetlony. Standardowo pokazywana jest cała bitmapa związana z kontekstem urządzenia, lecz ustawianie regionu przycinania pozwala wpłynąć na to zachowanie.

Tryb wypełniania wielokątów

<i>Domyślne ustawienie:</i>	ALTERNATE
<i>Funkcja ustawiająca atrybut:</i>	SetPolyFillMode()
<i>Funkcja pobierająca atrybut:</i>	GetPolyFillMode()

Określa tryb wypełniania wielokątów o przecinających się krawędziach (zarówno tych pochodzących od regionów, jak i rysowanych za pomocą funkcji `Polygon()` i `PolyPolygon()`). Domyślny tryb `ALTERNATE` sprawia, że wypełnienie otrzymają tylko te fragmenty prostokąta wielokąta, które leżą między jego nieparzystymi wierzchołkami; druga możliwa opcja `WINDING` powoduje bezwarunkowe wypełnienie całej figury.

Zapisywanie i odtwarzanie atrybutów

Mnogość atrybutów kontekstów urządzenia sprawia, że ich ustawianie, pobieranie i modyfikacja są czynnościami bardzo częstymi. Nierzadko też kilka parametrów kontekstów ustawia się po sobie. Potem zaś często zachodzi potrzeba powrotu do stanu początkowego i wydawałoby się, że nie można tego zrobić inaczej niż przez zapisywanie pierwotnych wartości atrybutów w wydzielonych zmiennych, a potem ich mozolne przywracanie. Nic bardziej mylnego!

GDI udostępnia prosty mechanizm uwalniający programistę od tej uciążliwej czynności. Są nim dwie funkcje: `SaveDC()` i `RestoreDC()`. Pokażę teraz, jak należy je stosować.

Tworzenie i przywracanie stanów kontekstu

Zacniemy od omówienia podstawowego sposobu użycia zapisu stanów kontekstu, wynikającego bezpośrednio ze składni wspomnianych funkcji.

Funkcje `SaveDC()` i `RestoreDC()`

Pierwsza z przedstawianych funkcji to `SaveDC()`. Oto jej prototyp:

```
int SaveDC(HDC hdc);
```

Jak widzimy, funkcja ta żąda uchwytu do kontekstu urządzenia. W zamian wykona ona coś w rodzaju fotografii jego bieżącego stanu. Zapisze po prostu wartości wszystkich jego atrybutów w wewnętrznej strukturze danych Windows, abyśmy w razie potrzeby mogli je przywrócić.

Wynikiem wywołania funkcji `SaveDC()` jest liczba całkowita, działająca jako jednoznaczny **identyfikator** zapisanego stanu kontekstu. Zmieniając atrybuty kontekstu, a następnie wywołując kilkakrotnie `SaveDC()` uzyskaliśmy wiele takich identyfikatorów i każdy byłby poprawny. Wynika stąd, że Windows GDI potrafi przechowywać **wiele stanów kontekstu** urządzenia i, jeśli tylko zapisujemy ich identyfikatory, możemy w każdej chwili wrócić do **dowolnego wcześniejszego**.

Jak to zrobić? Należy wywołać drugą funkcję, `RestoreDC()`:

```
BOOL RestoreDC(HDC hdc,  
               int nSavedDC);
```

Łatwo wydedukować, że podajemy jej uchwyt do naszego kontekstu urządzenia oraz identyfikator zapisanego statusu. `RestoreDC()` przywraca kontekst do podanego stanu, ustawiając jego atrybuty na zachowane wcześniej wartości.

Użycie mechanizmu stanów kontekstu

Podejrzewam, że właściwy sposób użycia tych funkcji nasuwa ci się sam, ale może dla zupełnej pewności zaprezentuję go.

Mając kontekst urządzenia, dajmy na to `hdcKontekst`, możemy zapisać jego bieżący stan:

```
int idStan = SaveDC(hdcKontekst);
```

Teraz możemy w spokoju wykonywać założone czynności. Dla przykładu, narysujemy czerwony prostokąt o wymiarach 50×50 pikseli otoczony grubą czarną kreską i umieszczony w punkcie (20, 20). W tym celu musimy między innymi ustawić nowe obiekty pióra i pędzla dla kontekstu urządzenia.

O piórach, pędzlach i rysowaniu figur geometrycznych dowiesz się (prawie) wszystkiego z następnego podrozdziału.

Ponieważ aktualne pióro i pędzel zostały zapisane przez `SaveDC()`, nie musimy się o nie martwić. Możemy normalnie zastąpić je nowymi obiektami:

```
// utworzenie pióra rysującego grubą czarną kreskę i ustawienie go  
HPEN hpenPioro = CreatePen(PS_SOLID, 5, RGB(0, 0, 0));  
SelectObject(hdcKontekst, hpenPioro);  
  
// utworzenie pędzla wypełniającego czerwienią i ustawienie go  
HBRUSH hbrPedzel = CreateSolidBrush(RGB(255, 0, 0));  
SelectObject(hdcKontekst, hbrBrush);
```

Następnie rysujemy prostokąt:

```
Rectangle(hdcKontekst, 20, 20, 70, 70); // 70 = 20 + 50
```


Potem możemy już przywrócić poprzedni stan kontekstu, czyli poprzednie pióro i pędzel. Robimy to oczywiście poprzez `RestoreDC()`:

```
RestoreDC (hdcKontekst, idStan);
```

Na koniec nie zapomnijmy jeszcze o zwolnieniu obiektów pióra i pędzla; możemy i musimy to zrobić. Możemy - bo po przywróceniu stanu kontekstu nie są one z nim związane. Musimy - bo sami je stworzyliśmy i bez ich zwolnienia doszłoby do wycieku zasobów. Wywołujemy zatem odpowiednią funkcję:

```
DeleteObject (hpenPioro);
DeleteObject (hbrPedzel);
```

Ogólny schemat postępowania w tego rodzaju sytuacjach wygląda więc następująco:

```
int identyfikator_zapisu = SaveDC(kontekst);
zmiana_stanu_kontekstu_i_rysowanie
RestoreDC (kontekst, identyfikator_zapisu);
[zwolnienie_utworzonych_obiektów]
```

Ostatni etap *zwolnienia* wystąpi wówczas, gdy *zmiana_stanu_kontekstu* pociągała za sobą utworzenie jakichś obiektów, jak pióra czy pędzle. W innym przypadku nie jest konieczna, bo i nie ma czego zwalniać.

Prostszy sposób

Możliwości zapisu stanu kontekstu urządzenia używamy zwykle zgodnie z podanym wyżej schematem. Dlatego też Windows GDI ułatwił nawet jeszcze bardziej wykorzystanie go.

Stos ustawień

Otóż niekoniecznie musimy zapisywać identyfikator zapisu, jaki zwraca `SaveDC()`.

Funkcja ta układa bowiem kolejne stanu kontekstu w stos, przykrywając starsze zapisy nowszymi. Wszystkie one są jednak dostępne na zwykłych zasadach, jakimi kieruje się stos, tzn. pobieranie zachowanych stanów powinno się odbywać w kolejności odwrotnej do ich zapisywania.

Jeżeli więc wywołamy `SaveDC()` np. trzy razy po sobie, to następujące dalej przywołanie `RestoreDC()` przywróci najnowszy zapis; kolejne wywołanie - starszy status; trzecie zaś - najstarszy. Rzadko jednak będziemy potrzebowali aż tylu możliwych stanów, gdyż w zupełności wystarczy jeden.

Ażeby więc zachować bieżące ustawienia kontekstu bez zbędnych ceregieli, wywołujemy jednokrotnie `SaveDC()`, ignorując zwracaną przezeń:

```
SaveDC (hdcKontekst);
```

Kiedy zaś pragniemy przywrócić zapisany stan, posługujemy się `RestoreDC()` w nieco inny sposób. Nie możemy już podać jej identyfikatora zapisu, bo go nie mamy. Zamiast tego w drugim parametrze wpisujemy **liczbę ujemną** określającą, który status, licząc od góry stosu, chcemy ustawić. **-1** przywróci więc pierwszy (szczytowy) stan; **-2** - ten leżący bezpośrednio pod nim; **-3** - jeszcze głębszy, itd. Podajemy więc, ile do którego ostatnich wywołań `SaveDC()` chcielibyśmy się cofnąć.

Najczęściej chodzi nam wszakże o wywołanie najpóźniejsze, zatem przywrócenie zapisanego wówczas stanu to użycie poniższej linijki:

```
RestoreDC (hdcKontekst, -1);
```

Eliminujemy zatem konieczność posiadania dodatkowej zmiennej oraz potencjalne ryzyko pomyłki, jeżeli mechanizm zachowywania stanu stosujemy wobec kilku kontekstów urządzenia.

Przykład

Przykładem wykorzystania funkcji `SaveDC()` i `RestoreDC()` może być procedura rysująca jakiś kształt, który wymaga tymczasowej zmiany pióra, pędzla lub innego obiektu związanego z kontekstem urządzenia. Oto jest funkcja, która kreśli koło o podanym kolorze kreski i wypełnienia (a także pozycji i promieniu):

```
void RysujKolo(HDC hdcKontekst,
               POINT ptPozycja, unsigned uPromien,
               COLORREF clObwod, COLORREF clWnetrze)
{
    // zachowanie bieżącego stanu kontekstu
    SaveDC (hdcKontekst);

    /* ustawienie odpowiedniego pióra i pędzla */

    // stworzenie nowego pióra i ustawienie go w kontekście
    HPEN hpenPioro = CreatePen(PS_SOLID, 1, clObwod);
    SelectObject (hdcKontekst, hpenPioro);

    // stworzenie nowego pędzla i wybranie go
    HBRUSH hbrPedzel = CreateSolidBrush(clWnetrze);
    SelectObject (hdcKontekst, hbrPedzel);

    /* wykreślenie koła */
    unsigned uSrednica = uPromien * 2;
    Ellipse (hdcKontekst,
             ptPozycja.x, ptPozycja.y,
             ptPozycja.x + uSrednica, ptPozycja.y + uSrednica);

    /* czynności końcowe */

    // przywrócenie początkowego stanu kontekstu urządzenia
    RestoreDC (hdcKontekst);

    // usunięcie utworzonego pióra i pędzla
    DeleteObject (hpenPioro);
    DeleteObject (hbrPedzel);
}
```

Dzięki zapisowi stanów nie musimy osobno troszczyć się o zachowanie oryginalnego pióra i pędzla. Im więcej parametrów kontekstu zmieniamy, tym bardziej będziemy to doceniać.

Wszystkie potrzebne wiadomości na temat użytych tu funkcji GDI odnoszących się piór i pędzli oraz rysujących figury geometryczne uzyskasz w następnym podrozdziale. Nawet teraz nie powinieneś mieć jednak zbyt dużych kłopotów z wywnioskowaniem ich działania na podstawie składni wywołań.

Zwalnianie kontekstu

Jak niemal wszystko w programowaniu, także kontekst urządzenia wymaga poprawnego posprzątania, gdy już nie jest potrzebny. Po użyciu kontekst należy więc **zwolnić** (ang. *release*).

Sposoby zwalniania kontekstów

W jaki sposób trzeba uczynić? Właściwa droga zależy od źródła pozyskania danego kontekstu, przy czym mamy trzy możliwe metody. Zawsze należy wybierać właściwą w danym przypadku!

Tymczasowy kontekst w obsłudze `WM_PAINT`

Najczęstszym powodem pobrania kontekstu urządzenia jest obsługa komunikatu `WM_PAINT`. Uzyskany wtedy kontekst jest związany z obszarem klienta okna, które ulega oświeżaniu.

Nie jest to obiekt długo żyjący. Powstaje w wyniku wywołania funkcji `BeginPaint()`, która mówi systemowi, że oto rozpoczyna się odrysowywanie okna. Kontekst istnieje tylko podczas trwania tej czynności, a jak wiemy, kończymy ją funkcją `EndPaint()`. Ona też zwalnia kontekst urządzenia:

```
case WM_PAINT:
{
    HDC hdcKontekst;
    PAINTSTRUCT ps;

    hdcKontekst = BeginPaint(hWnd, &ps);
    // tutaj rysujemy, odświeżając okno
    EndPaint (hWnd, &ps);

    // !! W tym miejscu hdcKontekst jest już niepoprawnym uchwytem
    //    do kontekstu urządzenia !!
}
```

Musimy pamiętać, że za wywołaniem `EndPaint()` kontekst pobrany z `BeginPaint()` jest już **nieważny**. Łatwo to przeoczyć, ponieważ uchwytu do tego kontekstu nie podajemy do `EndPaint()` bezpośrednio. Jest on jednak polem struktury `PAINTSTRUCT`, której wskaźnik przekazujemy w drugim parametrze funkcji.

Zapamiętajmy zatem (a raczej przypomnijmy sobie), że:

Kontekst urządzenia uzyskany w `BeginPaint()` jest zwalniany poprzez funkcję `EndPaint()`. Te dwie funkcje wyznaczają także **proces odświeżania** okna w reakcji na komunikat `WM_PAINT`. Wspomniany kontekst możemy wykorzystywać **wyłącznie** w ramach tego procesu.

Konteksty związane z oknem

Poza kodem obsługi `WM_PAINT` także możemy pobrać kontekst urządzenia związany z oknem. Jak powiedzieliśmy wcześniej, funkcja `GetDC()` pobiera kontekst pokrywający obszar klienta okna, natomiast `GetWindowDC()` - całe okno.

W ogromnej większości przypadków oba te konteksty wymagają zwolnienia. Odpowiada za to funkcja `ReleaseDC()`:

```
ReleaseDC (hWnd, hdcKontekst);
```

Musimy jej przekazać dwa parametry: pierwszym jest uchwyt okna, od którego pobraliśmy kontekst; drugim - uchwyt samego kontekstu. Funkcja zwolni wówczas zasoby stworzone wraz z kontekstem urządzenia, łącznie z nim samym.

Teoretycznie istnieją dwie sytuacje, w których zwolnienie kontekstu pobranego przez `GetDC()` (lecz nie przez `GetWindowDC()`!) nie jest niezbędne. Sytuacje te zachodzą, gdy klasa okna, od którego pobieramy kontekst, ma ustawiony styl `CS_OWNDC` lub `CS_CLASSDC`. Wówczas bowiem kontekst urządzenia, jaki pobieramy, istnieje przez cały czas istnienia okna i nie jest specjalnie tworzony dla nas. Jednak nawet w tych szczególnych przypadkach użycie `ReleaseDC()` nie jest błędem. Funkcja ta zignoruje po prostu każdy prywatny (`CS_OWNDC`) lub klasowy (`CS_CLASSDC`) kontekst okna, który jej przekazemy i nie robi z nim absolutnie nic.

Możemy więc zapamiętać ogólną i prostą zasadę:

Konteksty urządzenia **związane z oknem**, czyli pobrane przez `GetDC()`, `GetDCEx()` i `GetWindowDC()`, powinny być po użyciu **bezwzględnie zwalniane** funkcją `ReleaseDC()`.

W przypadku kontekstu ekranu pobranego przez `GetDC(NULL)` zwolnienie odbywa się także poprzez `ReleaseDC()`, ale z uchwyt okna (pierwszym parametrem) ustawionym na zero - czyli właśnie `NULL`.

Pozostałe rodzaje kontekstów

A co z innymi rodzajami kontekstów - tymi stworzonymi przez `CreateDC()` czy `CreateCompatibleDC()`?... Otóż ich usunięcie przebiega chyba najprościej, bo ogranicza się przekazania ich - i tylko ich - do funkcji `DeleteDC()`:

```
DeleteDC (hdcKontekst);
```

Ta bez zbędnych pytań usunie podany jej kontekst, zwalniając wszystkie powiązane z nim zasoby.

Jako ostatnią zasadę pamiętajmy zatem, że:

Niezwiązane z oknami konteksty urządzeń należy zwalniać poprzez `DeleteDC()`.

O zwalnianiu obiektów powiązanych

Trzeba nam wiedzieć, że z każdym kontekstem urządzenia w Windows GDI jest związanych kilka obiektów pomocniczych. Wspomniałem o nich podczas pobieżnej prezentacji atrybutów kontekstu, ale powtórzę listę ich wszystkich.

Tak więc kontekst urządzenia jest przez cały czas swego istnienia powiązany jest z jednym i tylko jednym obiektem:

- pióra
- pędzla
- czcionki
- bitmapy
- regionu przycinania

Każdy z tych obiektów możemy aczkolwiek podmienić¹⁴¹, posługując się na przykład funkcją `SelectObject()`. Więcej informacji na temat uzyskasz w następnych podrozdziałach.

¹⁴¹ Pewnym wyjątkiem jest bitmapa, których podmiana jest możliwa tylko dla kontekstu pamięciowego. Z pozostałymi typami kontekstów bitmapy są związane na stałe.

Teraz skupimy się tylko na tym, jak zmiana powyższych obiektów wpływa na sposób zwalniania kontekstu urządzenia. Wpływ ten jest bowiem bardzo znaczący.

Porzucony obiekt nie ginie

Konieczne musimy zdawać sobie sprawę, że ową piątkę obiektów **kontekst posiada zawsze**, nawet tuż po swoim powstaniu. Jeżeli więc zmienimy któryś, to nie wypełnimy żadnej luki, ale zamienimy miejscami dwa obiekty. Do kontekstu trafi ten wybrany przez nas, my natomiast otrzymamy poprzedni obiekt. Będziemy odtąd odpowiedzialni za jego zwolnienie.

Dlatego też kategorycznie niepoprawnym postępowaniem jest np. wybranie dla kontekstu nowego pióra w ten oto sposób:

```
HPEN hpenNiebieskiePioro = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
SelectObject(hdcKontekst, hpenNiebieskiePioro); // ŻŁE!!! (zazwyczaj)
```

Błędem jest tu zignorowanie wartości zwracanej przez `SelectObject()`. Jest nią tutaj uchwyt do starego pióra kontekstu, który powinien być zachowany; samo pióro przechodzi teraz pod naszą kuratelę i dlatego musimy znać jego uchwyt, aby móc je zwolnić, jeżeli nie jest już potrzebne.

Prawidłowe wybranie nowego pióra powinno więc wyglądać tak:

```
HPEN hpenStarePioro = (HPEN) SelectObject(hdcKontekst,
                                          hpenNiebieskiePioro);
```

Możliwe też (a nawet bardzo częste), że stare pióro nie jest nam potrzebne już w tej chwili - bo przecież wybieramy nowe. W takim przypadku możemy od razu pozbyć się kłopotu, wykorzystując poniższą - dość zaskakującą, ale całkowicie poprawną - konstrukcję:

```
DeleteObject (SelectObject(hdcKontekst, hpenNiebieskiePioro));
```

Przekazujemy tu po prostu wynik funkcji `SelectObject()`, czyli uchwyt do starego pióra, do funkcji `DeleteObject()`, która je natychmiast usunie. Nie potrzebujemy wtedy pośrednictwa dodatkowej zmiennej.

Obiekty powiązane znikają razem ze swoim kontekstem

Wielu programistów inaczej podchodzi do przedstawionej tu kwestii. Zamiast pozbywać się starego obiektu wziętego z kontekstu urządzenia (jak to czyni ostatni wiersz kodu w poprzednim punkcie), zostawiają to samej bibliotece GDI. Zachowują więc te oryginalne obiekty, a gdy przychodzi czas usunięcia kontekstu, umieszczają je w nim z powrotem. Następnie zwalniają kontekst, a potem (lub ewentualnie przed zwolnieniem kontekstu) usuwają swoje własne obiekty, stworzone na początku.

Na przykładzie naszych operacji z piórem wyglądałoby to tak:

```
// stworzenie własnego pióra i wybranie go (z zachowaniem oryginalnego)
HPEN hpenNiebieskiePioro = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
HPEN hpenStarePioro = (HPEN) SelectObject(hdcKontekst,
                                          hpenNiebieskiePioro);

// (rysowanie...)

// przywrócenie starego pióra i usunięcie kontekstu
SelectObject(hdcKontekst, hpenStarePioro);
DeleteDC(hdcKontekst); // lub ReleaseDC(), ewentualnie EndPaint()...

// usunięcie własnego pióra
```

```
DeleteObject (hpenNiebieskiePioro);
```

Nie usuwamy tutaj jawnie oryginalnego pióra kontekstu urządzenia, lecz pozwalamy to zrobić bibliotece GDI. I ona to robi - w chwili usunięcia kontekstu, ponieważ:

Wszystkie pięć **obiektów powiązanych z kontekstem urządzenia** jest **niszczonych** w momencie jego **zwalniania**.

Który sposób jest lepszy: natychmiastowe niszczenie nieużywanych obiektów czy też przywracanie ich do kontekstu i zwalnianie tylko tych własnych? Ciężko odpowiedzieć na to pytanie. I w jednym, i w drugim przypadku musimy zwolnić któryś z obiektów - nasz lub oryginalny obiekt kontekstu, więc nie wygląda na to, iż między obiema metodami była jakaś istotna różnica.

Można jeszcze argumentować, że sposób pokazany przed chwilą jest bardziej przejrzysty, ponieważ dokładnie widać czas życia naszych obiektów. Poza tym zwalniamy tutaj tylko twory, które sami wykreowaliśmy. Tworzą się więc „bloki” kodu, ograniczone funkcjami `CreatePen()` i podobnymi oraz funkcją `DeleteObject()`; wewnątrz tych bloków istnieją nasze obiekty. Nieco łatwiej więc śledzić czas ich życia, a przy większej liczbie zapobiegać wyciekom zasobów.

No tak, ale większa ilość zmienianych obiektów implikuje konieczność istnienia coraz większej liczby zmiennych. Jeżeli na przykład oprócz pióra podmienialibyśmy także pędzel i czcionkę, to potrzebowalibyśmy w sumie sześciu uchwytów, chociaż tworzylibyśmy tylko trzy obiekty.

Ale i na to jest rada. Przypomnijmy sobie mechanizm zapisu stanów kontekstu z poprzedniego paragrafu - jest on dokładnie tym, czego potrzebujemy. Zachowując początkowy stan kontekstu i przywracając go tuż przed usunięciem, unikamy konieczności deklarowania dodatkowych zmiennych. Zaprezentowany uprzednio kod może więc wyglądać tak:

```
// zapisanie stanu kontekstu
SaveDC (hdcKontekst);

// stworzenie własnego pióra i wybranie go (z zachowaniem oryginalnego)
HPEN hpenNiebieskiePioro = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
SelectObject(hdcKontekst, hpenNiebieskiePioro); // teraz tak możemy

// (rysowanie...)

// przywrócenie zapisanego stanu i usunięcie kontekstu
RestoreDC (hdcKontekst, -1);
DeleteDC (hdcKontekst);

// usunięcie własnego pióra
DeleteObject (hpenNiebieskiePioro);
```

Jak widać, nie potrzebujemy już zmiennej `hpenStarePioro`. Możemy też bezpiecznie zignorować rezultat funkcji `SelectObject()`, bo i tak jest on zapisany razem z fotografią statusu dokonaną przez `SaveDC()`. Pamiętajmy aczkolwiek, że w innym przypadku byłoby to niepoprawne - zwracałem na to uwagę wcześniej.

Niewybrane obiekty także należy zwolnić

Gdy usuwamy kontekst urządzenia, tak jak w przykładzie powyżej, zajmie się on wszystkimi posiadanymi obiektami, czyli także je zniszczy. Pozostałe obiekty, nienależące już do niego, nie ucierpią jednak w ogóle. U nas takim obiektem jest `hpenNiebieskiePioro`; musimy zatem usunąć je samodzielnie:

```
DeleteObject (hpenNiebieskiePioro);
```

Jest to logiczne, bo w końcu sami to pióro stworzyliśmy.

Zachowaj zatem w swojej pamięci, iż:

Należy **zwalniać wszystkie obiekty**, które **nie były powiązane z kontekstem urządzenia w chwili jego niszczenia**. W przeciwnym wypadku dojdzie do niebezpiecznego wycieku zasobów systemowych.

GDI w skrócie

Poznaliśmy już podstawowe pojęcia Windows GDI, czyli kontekst urządzenia oraz potok graficzny. Zanim jednak przejdziemy do szczegółowego omówienia poszczególnych części tej obszernej biblioteki, spójrzmy na nią z ogólniejszego punktu widzenia.

Podstawowym założeniem GDI, którym wionie niemal z każdego jej kąta, jest **uniwersalność**. Korzystając z tej biblioteki nie odwołujemy się bezpośrednio do sprzętu, takiego jak monitor czy drukarka. Zamiast tego, wykorzystujemy pewien poziom abstrakcji, jakim jest kontekst urządzenia. Ogromną większość funkcji graficznych wykonujemy w stosunku do takich właśnie kontekstów. Dzięki temu nie musimy interesować się tym, w jaki sposób wyniki naszej pracy są ostatecznie prezentowane. W tym jest bowiem rola biblioteki Windows GDI, która odpowiednio współpracuje ze sterownikami sprzętu.

Drugim ważnym aspektem GDI jest jej **elastyczność** oraz bardzo duże możliwości. Istnieje naprawdę niewiele operacji na grafice dwuwymiarowej, które nie zostały zaimplementowane w tej bibliotece. Co więcej, te które zostały w niej zawarte (a jest ich mnóstwo), zakodowano w sposób niezwykle łatwo poddający się zamierzeniom programisty. W GDI prawie nic nie jest niezmiennie, zmodyfikować możemy wszystko: styl i kolor rysowanych linii, sposób wypełniania zamkniętych powierzchni, czcionkę używaną do pisania tekstu, kolor i wielkość znaków, ich nachylenie do poziomu czy choćby obszar, w którym dokonuje się rysowanie (może on mieć najróżniejsze kształty), a także mnóstwo innych parametrów. Bardzo rzadko będziemy więc napotykać na istotne ograniczenia w możliwościach biblioteki Windows GDI.

„Skoro tak”, odpowiesz, „to gdzie tu jest haczyk?... Przecież nie może być aż **tak** pięknie!” Chciałbym powiedzieć, że jesteś pesymistą, ale niestety muszę stwierdzić, że jesteś raczej realistą. Faktycznie jest pewien kruczek, i to z gatunku tych kruczków, które programiści gier „lubią” najbardziej. Tak jest, zgadza się - to wydajność. Windows GDI nie jest szczególnie szybkie, jeżeli chodzi o wyświetlanie dynamicznie zmieniających się obrazów. Została ona bowiem głównie do prezentacji statycznych rysunków, nie radzi sobie z ruchem czy animacją.

Zupełnie źle jednak nie jest. Nie ma przeciwwskazań, ażeby skorzystać z GDI do napisania np. gry logicznej, karcianej, strategicznej turówki czy nawet zręcznościowego Ponga albo wręcz prostej gry platformowej. Nie należy jednak oczekiwać, że biblioteka ta sprawi się wystarczająco dobrze w najeżonej efektami graficznymi kosmicznej strzelance, a już na pewno polegnie, jeżeli spróbowałoby się symulować przy jej pomocy rendering grafiki 3D (co zresztą nie ma zbytniego sensu w obliczu istnienia wyspecjalizowanych API). Niezależność od sprzętu, uniwersalność i elastyczność odbijają się wtedy czkawką, stają się niepotrzebnych balastem na drodze do osiągnięcia dużej szybkości działania.

Już we wstępie do tego rozdziału napisałem jednak, że nie powinniśmy zarzucać sensu nauki biblioteki GDI. Nie wykorzystamy jej wprowadzić jako głównego „silnika” naszych gier, ale wypełnimy lukę, jaką niewątpliwie zostawia DirectX. Pozwólmy GDI wykazać się tym, w czym jest niezrównana: w generowaniu nieruchomych obrazów dwuwymiarowych. DirectX nie jest biegły w tej sztuce, więc GDI może pełnić nieodzownie

ważną rolę pomocniczą. Dotyczy to w szczególności tekstur - jednego z fundamentów grafiki 3D.

Kontynuujemy zatem poznawanie Windows GDI, przypatrując się teraz głębiej strukturze tego narzędzia.

Składniki interfejsu GDI

GDI jest bardzo dużą biblioteką, zawierającą wiele funkcji i struktur danych. Naturalnie więc dzielą się one pomiędzy składniki, jakie można wyodrębnić w interfejsie GDI.

Konteksty urządzeń

Kontekst urządzenia jest podstawą biblioteki GDI. Uchwyt do niego jest najważniejszą daną, jaką musi posiadać program, aby móc korzystać z funkcji rysujących. Bez kontekstu urządzenia nie można właściwie w żaden sposób korzystać z Windows GDI.

Jak pokazałem w poprzedniej sekcji, możliwe jest uzyskanie kontekstu z bardzo wielu źródeł. Do najważniejszych należą okna Windows oraz wyjściowe urządzenia graficzne, jak monitor i drukarka.

Z każdym kontekstem związanych jest pięć obiektów, niezbędnych dla jego funkcjonowania. Korzystanie z GDI sprowadza się do kontrolowania tych obiektów, a także do rysowania w kontekście urządzenia za pomocą prymitywów.

Prymitywy

Typy grafik, jakie można rysować w ramach kontekstu urządzenia w Windows GDI, nazywamy dość dziwną nazwą **prymitywów** (ang. *primitives*). Dzielą się one na kilka kategorii:

- **figury geometryczne** - są to czysto wektorowe prymitywy, opisane przez równania matematyczne. Można wśród nich wyróżnić jeszcze dwie podgrupy:
 - ✓ **krzywe otwarte** - należą do nich linie proste, łamane, łuki okręgów oraz krzywe Béziera. Wszystkie one są kreślone za pomocą pióra aktualnie wybranego w kontekście urządzenia
 - ✓ **krzywe zamknięte** to prostokąty (z ostrymi lub zaokrąglonymi rogami) oraz elipsy, a także szczególne przypadki tych figury, czyli kwadraty i koła. GDI rysuje ich obramowania przy pomocy wybranego pióra, natomiast wnętrza są wypełniane pędzlem należącym do kontekstu urządzenia
- **bitmapy** - to prostokątne tablice bitów, zawierające dane o kolorowych pikselach urządzenia. Są one używane do prezentacji skomplikowanych obrazów rastrowych, jakich nie dałoby się zapisać w postaci wektorowej. Bitmapa jest ponadto czymś w rodzaju płótna, na którym pojawiają się efekty działania funkcji rysunkowych; każdy kontekst urządzenia ma na własność taką właśnie bitmapę, której zawartość jest najczęściej prezentowana na ekranie
- **tekst** jest najbardziej skomplikowanym prymitywem, ale też najważniejszym (jako że analfabeci są nadal w mniejszości ;D). GDI zapewnia tutaj pełne wsparcie dla czcionek systemowych, w tym także dla czcionek TrueType, które mogą być dowolnie skalowane. Możliwe jest również stosowanie dowolnego formatowania tekstu (jak pogrubienie, pochylenie, kursywa czy kolor) oraz rysowanie znaków przy użyciu piór i pędzli

Poznanie biblioteki GDI to w gruncie rzeczy nauka posługiwania się tymi trzema rodzajami prymitywów. Toteż omówimy sobie dokładnie każdy z nich już w kolejnym podrozdziale.

Pozostałe składniki

Oprócz wymienionych wyżej prymitywów, w Windows GDI możemy też znaleźć kilka innych aspektów grafiki: Oto i one:

- **regiony** - nazywamy nimi dowolne kombinacje figur zamkniętych. Regionów będziemy używać do kreślenia obramowań, wypełniania ich, a także do przycinania rysunków
- **ścieżki** (ang. *paths*) to połączenia kilku krzywych otwartych. Kontekst urządzenia może w danej chwili przechowywać tylko jedną ścieżkę, dla której możliwe do wykonania są operacje wykreślenia piórem, wypełnienia pędzlem oraz konwersji na region.
Nie będziemy się bliżej zajmować ścieżkami, więc jeżeli chciałbyś samodzielnie dowiedzieć się czegoś o nich, zajrzyj do [MSDN](#)
- **metapliki** (ang. *metafiles*) są zapisem poleceń dla Windows GDI w postaci pliku dyskowego. Pliki takie mają rozszerzenie *.wmf* (zwykłe metapliki) lub *.emf* (rozszerzone metapliki) i mogą być odczytywane i zapisywane przez bibliotekę GDI.
Jeżeli chcesz dowiedzieć się więcej na ich temat, zajrzyj do [opisu w MSDN](#). W tym kursie nie będziemy bowiem zajmować się metaplikami
- **palety kolorów** (ang. *color palettes*) są już mocno przestarzałą częścią GDI, reliktem z czasów, gdy monitory mogły wyświetlać najwyżej 256 kolorów. Paleta określała wówczas zestaw 236 dowolnych barw, z jakich mogły korzystać obrazy. Obecnie, w czasach niepodzielnego panowania trybu *True Color* palety są już zupełnie nieprzydatne

Z powyższych zagadnień obszernie omówimy tylko pierwsze z nich, czyli regiony. Po informacje na temat reszty możesz udać się do dokumentacji MSDN, jeżeli tego potrzebujesz.

Obiekty

Oprócz właściwego rysowania, praca z Windows GDI polega także na manipulowaniu różnego rodzaju obiektami pomocniczymi - mniej lub bardziej istotnymi. Oto ich wyszczególnienie:

- **konteksty urządzenia** - o nich powiedziałem już tak dużo, że chyba nie mam już nic do dodania :) Jak echo powtórzę tylko, że to najważniejsze i kluczowe obiekty Windows GDI
- **pióra** definiują styl, kolor, grubość i inne cechy linii, którą GDI używa do zakreślania obwodów figur zamkniętych, wyrysowywania krzywych otwartych oraz wyznaczania ścieżek.
Prawie wszystko na temat piór powiemy sobie przy opisywaniu prymitywów figur geometrycznych, jako że w ich towarzystwie są one najpowszechniej stosowane
- **pędzle** określają sposób wypełniania zamkniętych powierzchni. Możliwe jest ich pokrywanie jednolitym kolorem, jednym z ustalonych deseni, jak również wybraną bitmapą.
O pędzlach napiszę więcej również przy okazji omawiania figur geometrycznych
- **bitmapy** przechowują rastrową postać obrazków. Każdy kontekst urządzenia jest związany z dokładnie jedną bitmapą, na której wygląd wpływają wywoływane funkcje GDI.
Więcej wiadomości o bitmapach znajduje się w poświęconej im sekcji następnego podrozdziału
- **czcionki** regulują krój pisma w wypisywanym tekście. Określają one nie tylko nazwę fizycznie istniejącej na dysku czcionki (np. Arial czy Times New Roman), ale też jej dodatkowe atrybuty, jak pogrubienie czy podkreślenie.
Tworzenie obiektów czcionek i ich wykorzystanie będzie tematem sekcji poświęconej tekstowi w następnym podrozdziale
- **regiony** mogą wpływać na zmianę obszaru rysowania, a także służyć do wykonywania innych czynności graficznych. Są to zespoły połączonych, zamkniętych figur geometrycznych, takich jak prostokąty i elipsy.
Regionom, a zwłaszcza ich roli w przycinaniu, jest poświęcony osobny podrozdział

Każdy z tych typów obiektów poznamy bliżej przy omawianiu prymitywów, na rysowanie których mają one wpływ. Regionom poświęcimy osobną część naszej uwagi, natomiast o kontekstach urządzeń mamy już całkowicie wystarczające wiadomości i więcej już ich nam nie trzeba ;)

Funkcje

Ogromną część Windows GDI stanowi kilkaset (!) różnorodnych funkcji graficznych i pokrewnych. To oczywiste, że fizycznie niemożliwe jest dogłębne omówienie ani nawet wyliczenie ich wszystkich; jest to zresztą niepotrzebne, skoro ich dokładne opisy znajdują się w dokumentacji MSDN.

W tej sekcji wyróżnię więc tylko kilka(naście) kategorii, na które można podzielić funkcje Windows GDI. Podam też nazwy najważniejszych procedur, które spełniają konkretne zadania - tak, abyś samodzielnie poszukać informacji o nich, czy to w dalszej części tego rozdziału, czy to w dokumentacji.

Jeżeli masz dobre IDE, jak np. Visual C++ .NET, to poniższe opisy funkcji mogą być całkiem wystarczające do ich użytkowania. Kiedy bowiem napiszesz nazwę którejś z nich w oknie edytora kodu i otworzysz nawias, otrzymasz listę jej parametrów, z nazwami i typami każdego z nich. To często wystarcza do wydedukowania prawidłowego działania funkcji.

Obejrzyjmy więc ten skromny katalog funkcji GDI.

Zarządzanie kontekstem urządzenia

Wokół kontekstów urządzenia wszystko tu się kręci, zatem zaczniemy od funkcji umożliwiających zarządzanie tymi obiektami.

Tworzenie kontekstu

Za tworzenie kontekstu urządzenia i zwracanie uchwytu do niego odpowiadają takie oto funkcje:

- dla kontekstów pochodzących od okien:
 - ✓ `BeginPaint()` pobiera uchwyt do tymczasowego kontekstu urządzenia, istniejącego na czas obsługi komunikatu `WM_PAINT`. Kontekst ten jest zwalniany przez wywołanie `EndPaint()`
 - ✓ `GetDC()` zwraca kontekst urządzenia wnętrza okna. Jeżeli w stylu klasy okna nie są podane flagi: `CS_OWNDC` lub `CS_CLASSDC`, nietrwały kontekst urządzenia jest specjalnie tworzony i po użyciu powinien być zwolniony za pomocą `ReleaseDC()`
 - ✓ `GetWindowDC()` podaje nam kontekst urządzenia związany z całym oknem, a więc także z jego obszarem pozaklienckim. Musi on być zawsze zwolniony przez `ReleaseDC()`
 - ✓ `GetDCEx()` zachowuje się jak jedna z trzech poprzednich funkcji (zależnie od podanej kombinacji flag), a także udostępnia pewne dodatkowe możliwości
- dla kontekstów tworzonych dla dowolnych urządzeń
 - ✓ `CreateDC()` tworzy kontekst dla podanego urządzenia
 - ✓ `CreateIC()` tworzy kontekst informacyjny, który może być użyty wyłącznie do pobrania informacji o urządzeniu, lecz nie do rysowania
- `CreateCompatibleDC()` kreuje pamięciowy kontekst urządzenia, kompatybilny z podanym

Pobieranie informacji o urządzeniu

Mając już konteksty urządzenia (zwykły lub informacyjny), możemy pokusić się o pobranie jakichś informacji o związanym z nim sprzęcie. Czynią to poniższe funkcje:

- `GetDeviceCaps()` służy do uzyskania specyficznych informacji o urządzeniu
- `DeviceCapabilities()` podaje dane na temat drukarki, jeżeli posiadamy jej kontekst urządzenia

Kontrola atrybutów kontekstu

Za zmianę i pozyskanie wartości kilkunastu atrybutów kontekstu urządzenia służy kilka poniższych funkcji:

- za zarządzanie obiektami kontekstu odpowiadają dwie funkcje:
 - ✓ `SelectObject()` ustawia nowe pióro, pędzel, bitmapę, czcionkę lub region przycinania, zwracając jednocześnie uchwyt do starego obiektu
 - ✓ `GetCurrentObject()` zwraca uchwyt do obiektu określonego rodzaju, który jest aktualnie powiązany z podanym kontekstem urządzenia
 - ✓ `GetStockObject()` pobiera jeden z przechowywanych wewnętrznie obiektów GDI
- za ustawianie parametrów potoku graficznego odpowiadają następujące funkcje:
 - ✓ `SetMapMode()` ustawia tryb mapowania, czyli wielkość jednostek logicznej oraz kierunek osi układu współrzędnych płaszczyzny świata (strony)
 - ✓ kadr na płaszczyźnie świata (strony) kontrolują funkcje:
 - ✖ `SetWindowOrgEx()` i `GetWindowOrgEx()` odpowiedzialne są za ustawianie i pobieranie pozycji kadru
 - ✖ `SetWindowExtEx()` i `GetWindowExtEx()` zarządzają rozciągłością osi kadru
 - ✓ wzornik na płaszczyźnie urządzenia jest pod opieką funkcji:
 - ✖ `SetViewportOrgEx()` i `GetViewportOrgEx()`, które dbają o jego położenie
 - ✖ `SetViewportExtEx()` i `GetViewportExtEx()`, zarządzających jego rozciągłością
- z piórem w kontekście urządzenia radzą sobie poniższe funkcje:
 - ✓ pozycja pióra na bitmapie kontekstu urządzenia to domena funkcji:
 - ✖ `MoveToEx()` - przesuwa ona pióro do podanej pozycji
 - ✖ `GetCurrentPositionEx()` - zwraca aktualną pozycję pióra
 - ✓ `SetROP2()` i `GetROP2()` kontrolują tryb rysowania piórem
- `SetBrushOrgEx()` i `GetBrushOrgEx()` zarządzają punktem zaczepienia pędzla
- `SetStretchBltMode()` i `GetStretchBltMode()` modyfikują zachowanie funkcji `StretchBlt()` przy kopiowaniu bitmapy do podanego kontekstu urządzenia
- ustawienia związane z tekstem są zasługą funkcji:
 - ✓ `SetTextColor()` i `GetTextColor()`, które ustawiają i pobierają kolor tekstu
 - ✓ `SetBkColor()` i `GetBkColor()`, kontrolujących kolor tła tekstu
 - ✓ `SetBkMode()` i `GetBkMode()`, zmieniających tryb wypełnienia tła tekstu (przezroczysty lub nie)
 - ✓ `SetTextCharacterExtra()` i `GetTextCharacterExtra()`, zawiadujących odstępami między znakami tekstu
- region przycinania pozostaje pod władzą funkcji:
 - ✓ z których niektóre go modyfikują:
 - ✖ `SelectClipRgn()` ustawia region przycinania równie dobrze jak `SelectObject()`
 - ✖ `ExtSelectClipRgn()` umożliwia kombinację nowego regionu przycinania z już obowiązującym
 - ✖ `IntersectClipRect()` dodaje do regionu podany prostokąt
 - ✖ `ExcludeClipRect()` wyklucza z regionu dany prostokąt
 - ✖ `OffsetClipRgn()` przesuwa region przycinania o podany wektor
 - ✖ `SelectClipPath()` łączy aktualną ścieżkę kontekstu urządzenia z jego regionem przycinania
 - ✓ a niektóre służą do pobierania regionu przycinania:

- * `GetClipRgn()` pobiera ów region
- * `GetRandomRgn()` pobiera kopię regionu przycinania
- * `GetClipBox()` pobiera koordynaty najmniejszego prostokąta otaczającego region przycinania

Zapisywanie i odtwarzanie kontekstu urządzenia

W tej kategorii są tylko dwie znane funkcje:

- `SaveDC()` zapisuje stan kontekstu urządzenia, tj. wszystkich jego atrybutów
- `RestoreDC()` odtwarza zapisany wcześniej stan

Zwalnianie kontekstu urządzenia

Utworzony kontekst trzeba prędzej czy później zwolnić (raczej prędzej niż później). Mamy wtedy do wyboru takie oto funkcje:

- `EndPaint()`, która kończy odświeżanie zawartości okna rozpoczęte przez `BeginPaint()` w reakcji na komunikat `WM_PAINT`
- `ReleaseDC()` zwalnia kontekst urządzenia związany z oknem (stworzony przez `GetDC()`, `GetDCEx()` lub `GetWindowDC()`)
- `DeleteDC()` usuwa wszystkie pozostałe rodzaje kontekstów (w szczególności te utworzone przez `CreateDC()`, `CreateIC()` i `CreateComaptibleDC()`)

Tworzenie obiektów GDI

Zanim pokażę funkcję odpowiedzialną za tworzenie poszczególnych typów obiektów GDI, podejść do zagadnienia wpierw „od tyłu”. Przedtem jak stworzymy jakikolwiek obiekt, musimy bowiem wiedzieć, w jaki sposób go potem zniszczyć. Służy do tego funkcja:

- `DeleteObject()`, która usuwa obiekt pióra, pędzla, bitmapy, czcionki, regionu lub palety

Dalej patrzymy już na funkcje kreujące obiekty.

Tworzenie piór

Do utworzenia pióra można w GDI wykorzystać funkcje:

- `CreatePen()` tworzy pióro o podanym stylu, grubości i kolorze linii
- `CreatePenIndirect()` ma te same możliwości co `CreatePen()`, ale przyjmuje jeden parametr (w postaci struktury `LOGPEN`) zamiast trzech
- `ExtCreatePen()` pozwala stworzyć pióro kreślące linie pokryte deseniem lub bitmapą, czyli mające właściwości pędzla

Tworzenie pędzli

Za tworzenie pędzli odpowiadają poniższe funkcje:

- `CreateSolidBrush()` tworzy pędzel malujący jednolitym kolorem
- `CreateHatchBrush()` stwarza pędzel posługujący się dwukolorowym deseniem
- `CreatePatternBrush()` kreuje pędzel wypełniający figury kopiami bitmapy
- `CreateBrushIndirect()` tworzy pędzel na podstawie podanej struktury `LOGBRUSH`

Tworzenie bitmap

Do stworzenia obiektu bitmapy można wykorzystać funkcje:

- `CreateBitmap()`, tworzącą bitmapę o podanych wymiarach, głębi kolorów i ewentualnie zawartości
- `CreateBitmapIndirect()`, działającą tak jak `CreateBitmap()`, lecz przyjmującą jako parametr strukturę typu `BITMAP`
- `CreateCompatibleBitmap()`, stwarzającą bitmapę kompatybilną z danym kontekstem urządzenia i mającą podane wymiary
- `LoadImage()` (zastępującą starszą, lecz nadal działającą funkcję `LoadBitmap()`), która potrafi wczytać bitmapę z pliku dyskowego

Tworzenie obiektów czcionek

Aby utworzyć obiekt czcionki, należy użyć jednej z tych oto funkcji:

- `CreateFont()` tworzy obiekt czcionki o podanym kroju i stylu pisma
- `CreateFontIndirect()` działa jak `CreateFont()`, lecz przyjmuje parametry w formie struktury `LOGFONT`
- `CreateFontIndirectEx()` tworzy czcionkę na podstawie przekazanej struktury `ENUMLOGFONTEXDV`

Tworzenie regionów

Funkcji tworzących regiony także mamy kilka:

- są wśród nich funkcje stwarzające proste regiony:
 - ✓ na przykład prostokątne:
 - * `CreateRectRgn()` tworzy prostokątny region
 - * `CreateRectRgnIndirect()` działa jak `CreateRectRgn()`, lecz przyjmuje jeden parametr typu `RECT`
 - * `CreateRoundRectRgn()` tworzy region w kształcie prostokąta z zaokrąglonymi rogami
 - ✓ a także takie w kształcie wielokątów:
 - * `CreatePolygonRgn()` kreuje region w formie wielokąta
 - * `CreatePolyPolygonRgn()` stwarza region złożony z kilku wielokątów
 - ✓ ewentualnie także w formie elips:
 - * `CreateEllipticRgn()` tworzy region w kształcie elipsy
 - * `CreateEllipticRgnIndirect()` działa jak `CreateEllipticRgn()`, ale żąda struktury typu `RECT`
- `CombineRgn()` łączy dwa regiony w jeden, posługując się podanym trybem kombinacji

Rysowanie prymitywów

Popatrzmy teraz na listę funkcji GDI rysujących prymitywy.

Figury geometryczne

Za kreślenie figur geometrycznych są odpowiedzialne poniższe funkcje:

- punkty i linie rysują takie oto funkcje:
 - ✓ za zaznaczanie punktów odpowiadają procedury:
 - * `SetPixel()` zaznacza podanym kolorem punkt na powierzchni urządzenia
 - * `SetPixelV()` używa do tego najbliższej aproksymacji podanego koloru
 - * `GetPixel()` pobiera kolor podanego punktu
 - ✓ linie proste rysują funkcje:
 - * `LineTo()` kresli prostą od aktualnej pozycji pióra do podanego punktu
 - * `PolylineTo()` kreśli łamaną od bieżącej pozycji pióra przez podane punkty
 - * `Polyline()` rysuje łamaną zaczynając od podanego punktu przez kolejne podane
 - * `PolyPolyline()` wykreśla kilka łamanych naraz
 - ✓ krzywe otwarte są domeną takich funkcji:
 - * `AngleArc()` rysuje prostą od bieżącego położenia pióra do podanej punktu, a następnie łuk - wycinek obwodu elipsy
 - * `ArcTo()` rysuje wycinek obwodu elipsy, poczynając od pozycji pióra
 - * `Arc()` kreśli łuk w dowolnym miejscu

- ✖ `PolyBezierTo()` rysuje krzywą Béziera począwszy od aktualnego miejsca pióra
- ✖ `PolyBezier()` kreśli krzywą Béziera w dowolnym miejscu
- figury zamknięte są rysowane przez te oto funkcje:
 - ✓ wielokątami zajmują się:
 - ✖ `Rectangle()` - rysuje prostokąt
 - ✖ `FillRect()` wypełnia prostokąt podanym pędzlem
 - ✖ `FrameRect()` kreśli obramowanie prostokąta przy pomocy pędzla
 - ✖ `InvertRect()` odwraca kolory (za pomocą bitowej negacji) w podanym prostokącie
 - ✓ krzywe zamknięte to zadania funkcji:
 - ✖ `Ellipse()` rysuje elipsę (także koło)
 - ✖ `Pie()` rysuje wycinek elipsy (koła)
 - ✖ `Chord()` rysuje odcinek elipsy (koła)
 - ✓ `RoundRect()` rysuje prostokąt z zaokrąglonymi rogami

Bitmapy

Za wyświetlanie bitmap (a raczej zawartości innych kontekstów urządzeń) odpowiadają funkcje:

- `BitBlt()`, która dokonuje dosłownego przekopiowania pikseli z jednego kontekstu urządzenia do drugiego, używając podanego sposobu łączenia kolorów
- `TransparentBlt()`, dokonująca kopiowania z możliwością wybrania koloru przezroczystego
- `StretchBlt()`, potrafiąca kopiować obrazy z ich jednoczesnym skalowaniem

Tekst

Wypisywanie tekstu w kontekście urządzenia to zadanie dla poniższych funkcji:

- oto funkcje piszące tekst w określonej pozycji:
 - ✓ `TextOut()` dokonuje prostego wypisania tekstu w podanym miejscu
 - ✓ `ExtTextOut()` potrafi jeszcze dokonać np. przycinania do prostokąta
 - ✓ `TabbedTextOut()` pozwala na wyrównywanie tekstu do podanych tabulatorów
- są też funkcje rozmieszczające tekst w podanym prostokącie:
 - ✓ `DrawText()` rysuje tekst wyrównany do krawędzi lub środka danego prostokąta
 - ✓ `DrawTextEx()` umożliwia jeszcze określenie marginesów poprzez strukturę `DRAWTEXT_PARAMS`

Regiony

Mamy jeszcze kilka funkcji związanych z regionami GDI.

Rysowanie z użyciem regionów

Regiony mogą służyć do rysowania za pośrednictwem tych funkcji:

- `PaintRgn()` wypełnia region pędzlem wybranym w kontekście urządzenia
- `FillRgn()` wypełnia region podanym pędzlem
- `FrameRgn()` kreśli obramowanie wokół regionu przy użyciu podanego pędzla
- `InvertRgn()` odwraca kolory (jak `InvertRect()`) w obszarze regionu

Regiony i okna

Do łączenia regionów i okien służą funkcje:

- `SetWindowRgn()` ustawia nowy region, wyznaczający kształt okna
- `GetWindowRgn()` pobiera region okna

Zakończamy już ten przydługi wstęp do opisu biblioteki Windows GDI. W następnym podrozdziale zajmiemy się już konkretnymi, czyli rysowaniem prymitywów. Wreszcie więc ujrzymy cokolwiek na naszych ekranach :)

Prymitywy

To, czym się teraz będę zajmować, dla wielu programistów (głównie niezbyt zaawansowanych) jest niemal tożsame z **całą** biblioteką GDI. Jak wiemy, nie jest to prawda, jednak nie da się ukryć, że prymitywy graficzne są jej najważniejszą częścią. To przecież naturalne, że narzędzie graficzne oceniamy przede wszystkim po tym, co możemy przy jego pomocy rysować. Bogate możliwości wyświetlania kształtów graficznych są więc niezwykle ważne.

Window GDI jest pod tym względem bardzo rozwiniętą biblioteką, mogącą zarówno kreślić zgeometryzowane kształty figur, jak również rastrowe bitmapy czy wreszcie napisy tekstowe. Dla każdego z tych prymitywów istnieje poza tym wiele opcji regulujących ich prezencję.

Niniejszy podrozdział poświęcimy tym strategicznym elementom GDI, jakimi są prymitywy. Omówimy tu osobno figury geometryczne, bitmapy oraz tekst.

Figury geometryczne

Najbardziej wektorowy charakter ze wszystkich prymitywów w Windows GDI zachowują figury geometryczne. Są one całkowicie niewrażliwe na skalowanie czy przesunięcie, zatem mogą być rysowane w dowolnym rozmiarach i w dowolnych miejscach.

W matematyce figury na płaszczyźnie są opisane odpowiednimi równaniami, ale nie musimy ich znać, aby rysować takie kształty. Biblioteka GDI zawiera sporo funkcji wyręczających nas w tym zadaniu - wiele z nich poznamy w tej sekcji.

Zanim jednak to się stanie, musimy sobie powiedzieć co nieco o dwóch ważnych obiektach, które wiążą się z kwestią rysowania figur geometrycznych w Windows GDI. Tymi obiektami są pióra i pędzle.

Pióro

GDI pozwala na rysowanie linii prostych oraz krzywych. Takie linie mogą mieć określone atrybuty, jak na przykład kolor. Decyduje o nich obiekt kontekstu urządzenia zwany **piórem**.

Pióro (ang. *pen*) kontroluje właściwości rysowanych linii: ich **grubość**, **kolor** (ewentualnie deseń) oraz **styl**.

Pióra są w GDI reprezentowane poprzez uchwyty typu `HPEN`.

Chcąc zatem rysować różne typy linii, musimy odpowiednio zmodyfikować właściwości pióra. Jest ich niewiele, więc w GDI najczęściej będziemy po prostu tworzyć nowe, swoje własne pióro i wybierać je dla danego kontekstu urządzenia. Po tym wszystkie linie będą kreślone przy użyciu tego właśnie nowowybranego pióra.

Korzystanie z piór

Typowa kolejność kroków przy korzystaniu z własnego pióra sprowadza się zatem do:

- stworzenia pióra
- wybrania go w używanym kontekście urządzenia

- narysowania figur
- odłożenia pióra z kontekstu, czyli wybranie w nim poprzednio ustawionego pióra
- usunięcia pióra

Wyjaśnimy sobie tutaj każdy z tych kroków, oczywiście z wyjątkiem samego rysowania figur, gdyż to jest tematem prawie całej pozostałej części sekcji.

Tworzenie pióra

Do utworzenia nowego pióra możemy wykorzystać funkcję `CreatePen()` i tak też będziemy czynić najczęściej. Oto prototyp tej funkcji:

```
HPEN CreatePen(int fnPenStyle,
               int nWidth,
               COLORREF crColor);
```

Umożliwia ona stworzenie pióra kreślącego linie o podanej grubości, kolorze i stylu. Te cechy pióra wyznaczają trzy parametry funkcji:

typ	parametr	opis
int	fnPenStyle	Ten parametr określa styl pióra , tj. rysowanych przy jego pomocy linii. Możliwe wartości ujmuje następująca tabela.
	nWidth	Tutaj podajemy grubość linii pióra w jednostkach logicznych. Jeżeli chcemy użyć innego stylu pióra niż domyślny jednolity (<code>PS_SOLID</code> , ewentualnie także <code>PS_INSIDEFRAME</code>), to najlepiej podać tu <code>0</code> , gdyż jeśli szerokość linii przekroczy jedną jednostkę urządzenia (zwykle piksel), inny styl niż <code>PS_SOLID/PS_INSIDEFRAME</code> nie będzie mógł być zastosowany i zostanie wybrany <code>PS_SOLID</code> . W przypadku podania zera grubość linii wyniesie natomiast jeden piksel i wszystko będzie w porządku, niezależnie od wartości <code>fnPenStyle</code> .
COLORREF	crColor	W tym parametrze określamy kolor linii rysowanych przez pióro.

Tabela 61. Parametry funkcji `CreatePen()`

Czym jest styl pióra?... To po prostu pewien sposób na określenie ciągłości linii. GDI udostępnia kilka takich stylów, przedstawia je poniższa tabela:

flaga	styl	linia
<code>PS_NULL</code>	brak linii	
<code>PS_SOLID</code>	linia ciągła	
<code>PS_DASH</code>	linia przerywana (kreski)	
<code>PS_DOT</code>	linia kropkowana	
<code>PS_DASHDOT</code>	kreska-kropka	
<code>PS_DASHDOTDOT</code>	kreska-kropka-kropka	
<code>PS_INSIDEFRAME</code>	linia ciągła	

Tabela 62. Style zwykłych piór w Windows GDI

Ostatni styl `PS_INSIDEFRAME` wygląda jak pierwszy, ale jest między nimi pewna różnica. Uwidacznia się ona przy obrysowywaniu regionów: `PS_INSIDEFRAME` generuje ramkę zawierającą się w całości wewnątrz regionu, zaś `PS_SOLID` - na zewnątrz.

Popatrzmy teraz na przykłady wykorzystania funkcji `CreatePen()` do tworzenia piór:

```
// tworzy pióro rysujące grubą czarną kreską
```

```
HPEN hpenCzarnyFlamaster = CreatePen(PS_SOLID, 5, 0x0);

// pióro kreślone najcieńszą możliwą, czerwoną linię
HPEN hpenCienkaCzerwonaLinia = CreatePen(PS_SOLID, 0, RGB(255, 0, 0));

// bardzo gruba linia w kolorze zielonym
HPEN hpenPasZieleni = CreatePen(PS_SOLID, 10, RGB(0, 255, 0));

// wykropkowana linia w kolorze magenty
HPEN hpenKarmazynoweKropki = CreatePen(PS_DOT, 0, RGB(255, 0, 255));

// normalna niebieska kreska
HPEN hpenAtrament = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
```

Jak widać, utworzenie własnego pióra jest bardzo proste.

Identycznie do `CreatePen()` działa funkcja `CreatePenIndirect()`. Zamiast trzech parametrów przyjmuje ona jedną strukturę `LOGPEN`, zawierając pola odpowiadające tym parametrom. Funkcja ta może być użyteczna, jeżeli np. chcemy zapisać nasze pióra w pliku na dysku.

Wiązanie pióra z kontekstem urządzenia

Samo istnienie pióra nic nam jeszcze nie daje. Musimy je bowiem związać z kontekstem urządzenia.

Możemy to zrobić przy pomocy funkcji `SelectObject()`. Trzeba jednak pamiętać, aby zająć się odpowiednio starym piórem, oryginalnie wybranym w kontekście urządzenia. Należy bowiem zadbać o jego zwolnienie; jest to, jak wiemy, możliwe na trzy sposoby:

- poprzez natychmiastowe usunięcie starego pióra funkcją `DeleteObject()`
- poprzez zachowanie uchwytu do starego pióra w osobnej zmiennej i przywrócenie go do kontekstu przed jego usunięciem
- przez zapisanie stanu kontekstu (`SaveDC()`) i przywrócenie go (`RestoreDC()`) tuż przed kresem jego życia

Wszystkie te trzy drogi były prezentowane w paragrafie poświęconym zwalnianiu kontekstu urządzenia. Tutaj więc pokażę tylko najczęściej stosowaną - zachowanie starego w dodatkowej zmiennej:

```
HPEN hpenZoltePioro = CreatePen(PS_SOLID, 2, RGB(255, 255, 0));
HPEN hpenStarePioro = (HPEN) SelectObject(hdcKontekst, hpenZoltePioro);
```

Po wybraniu pióra możemy już rysować przy jego pomocy dowolne figury geometryczne.

Odkładanie i zwalnianie pióra

Gdy zakończymy już pracę z piórem, powinniśmy je zwolnić. Jak już powiedziałem, zwolnienie może odbywać się wraz z niszczeniem kontekstu urządzenia lub też (częściej) być prowokowane jawnie funkcją `DeleteObject()`.

Ten drugi przypadek ukazuje ten kod. Jest to „druga połowa” listingu z poprzedniego punktu:

```
SelectObject(hdcKontekst, hpenStarePioro);
DeleteDC(hdcKontekst);           // tu jest usuwane pióro hpenStarePioro
DeleteObject(hpenZoltePioro);    // tu jest usuwane pióro hpenZoltePioro
```

Możliwe jest rzecz jasna, abyś stosował inne drogi (np. z `SaveDC()` i `RestoreDC()`), jeżeli uznasz je za wygodniejsze. Ważne jest jednak, by żadne pióro nie zostało „zgubione”, a

wszystkie obiekty zwolnione. W przeciwnym wypadku dojdzie do niebezpiecznego wycieku zasobów.

Elastyczne pióro

Częsta wymiana obiektu pióra w kontekście urządzenia może nie być zbyt efektywna. Jednocześnie zmiana koloru rysowanej linii jest czynnością bardzo często, a ta, jak wiemy, wymaga zmiany obiektu pióra... A może niekoniecznie?

Począwszy od Windows 2000 możliwe jest wykorzystanie tzw. **elastycznego pióra**. Jest to pióro rysujące cienką linię ciągłą. Kolor tej linii może być zmieniany bez konieczności wybierania nowego obiektu.

Zobaczmy zatem, jak wykorzystać elastyczne pióro.

Wybieranie elastycznego pióra

Na początek musimy poinformować GDI, że będziemy korzystać z tego specjalnego rodzaju pióra. Uchwyty do niego można pozyskać, wywołując funkcję `GetStockObject()` z parametrem `DC_PEN`; wybranie elastycznego pióra w kontekście urządzenia najlepiej zrealizować więc w jednym wywołaniu:

```
DeleteObject (SelectObject(hdcKontekst, GetStockObject(DC_PEN)));
```

Robimy tu jednocześnie aż trzy rzeczy: pobieramy uchwyt do elastycznego pióra (`GetStockObject()`), ustawiamy je w kontekście urządzenia (`SelectObject()`), a na koniec usuwamy obiekt starego pióra (`DeleteObject()`).

Ten - wydawałoby się, nieco zakręcony sposób - jest najwłaściwszy. Elastyczne pióro nie wymaga bowiem jawnego usunięcia¹⁴², więc jeśli zachowalibyśmy uchwyt do starego pióra, a po zakończeniu pracy z powrotem umieścili je w kontekście, tylko dodalibyśmy sobie większego zachodu. Lepiej będzie, jeżeli od razu pozbędziemy się niepotrzebnego, oryginalnego pióra; później nie będziemy już musieli martwić się o żadne inne. Jeżeli jednak kontekst urządzenia miał wcześniej ustawione nasze własne pióro, wtedy trzeba naturalnie rozważyć, czy chcemy je teraz usunąć. Zwykle nie chcemy.

Zmiana koloru pióra

Kiedy wybraliśmy już elastyczne pióro w kontekście urządzenia, możemy kontrolować jego kolor. Robimy to poprzez funkcję `SetDCPenColor()`. Poniższa linijka kodu ustawia przykładowo kolor pióra na morski:

```
SetDCPenColor ((hdcKontekst, RGB(0, 255, 255)));
```

Zmieniając kolory elastycznego pióra możemy łatwo narysować np. kwadrat o bokach w czterech barwach:

```
void Kwadrat4Kolorowy(HDC hdcKontekst, POINT ptPozycja, unsigned uBok,
                     COLORREF aKolory[4])
{
    // zapisanie ustawień kontekstu
    SaveDC (hdcKontekst);

    /* narysowanie kwadratu */

    // ustawienie elastycznego pióra
    SelectObject (hdcKontekst, GetStockObject(DC_PEN));
```

¹⁴² Podobnie zresztą jak każdy obiekt uzyskany funkcją `GetStockObject()`.

```

// pomocnicza tablica stałych, opisujących kierunki boków
// (kwadrat rysujemy od lewego górnego rogu zgodnie ze wsk. zegara)
const POINT BOKI[4] = { {1, 0}, {0, 1}, {-1, 0}, {0, -1} };

// ustawimy się w podanej pozycji (lewy górny róg kwadratu)
MoveToEx (hdcKontekst, ptPozycja.x, ptPozycja.y, NULL);

// rysujemy kolejne boki
POINT ptPozycjaPiora;
for (unsigned i = 0; i < 4; ++i)
{
    // ustawiamy kolor pióra na kolejny z podanych
    SetDCPenColor (hdcKontekst, aKolory[i]);

    // pobieramy aktualną pozycję pióra
    GetCurrentPositionEx (hdcKontekst, &ptPozycjaPiora);

    // kreślimy linię w odpowiednim kierunku; uzyskujemy go,
    // mnożąc długość boków przez współrzędne wektorów zapisane
    // w tablicy BOKI. W ten sposób dowiadujemy się, o ile
    // powinniśmy się przesunąć
    LineTo (hdcKontekst,
            ptPozycjaPiora.x + BOKI[i].x * uBok,
            ptPozycjaPiora.y + BOKI[i].y * uBok);
}

/* przywracamy oryginalny stan kontekstu */
RestoreDC (hdcKontekst, -1);
}

```

SetDCPenColor() zapobiega tutaj konieczności utworzenia, przechowywania i zniszczenia czterech piór w kontekście z urządzenia. Piór, które różnią się tylko kolorem; w takich sytuacjach znacznie lepiej jest użyć elastycznego pióra.

Właściwości pióra w kontekście urządzenia

Z obecnością pióra w kontekście urządzenia związane są dwie jego właściwości. Wpływają one na rysowane figury. Tymi atrybutami kontekstu są aktualna pozycja pióra oraz tryb rysowania.

Aktualna pozycja

Nie dziwiłbym się, jeżeli przynajmniej niektórzy z obecnych tu czytelników mieli styczność z tak zwanym 'językiem programowania' o nazwie LOGO. Jeżeli nawet nie zaczynali od niego swoich koderskich doświadczeń, to jest całkiem prawdopodobne, że byli nim męczeni w szkole.

Nie będę jednak rozwodził się tutaj nad kwestią, jak koszmarnym programem (bo językiem nie mogę tego nazwać...) jest LOGO, bo zajęłoby to resztę miejsca przeznaczonego na ten kurs ;D Chcę tylko wspomnieć o czymś takim jak żółw. Tak więc żółw w LOGO to taki rodzaj kursora, który mógł być sterowany instrukcyjnie i rysował przeróżne figury geometryczne. Zajmował on pewną pozycję na ekranie i można go było skierować w inną; po drodze zostawiał za sobą ślad w postaci linii.

Jak to się ma do GDI?... Otóż, żółw z LOGO ma sporo wspólnego z piórami w Windows GDI - nie wszystko wprawdzie, ale przynajmniej jedna jego właściwość jest dla nas teraz istotna. Tą właściwością jest **pozycja**.

Pióro w GDI także zajmuje określoną pozycję na bitmapie kontekstu urządzenia. Niektóre funkcje rysujące, takie jak LineTo(), korzystają z niej, rozpoczynając z tego miejsca

rysowanie figur. Po ich wykreśleniu pozycja pióra ulega zmianie: zatrzymuje się ono na przeciwległym końcu narysowanej krzywej. W ten sposób sterujemy piórem podobnie jak żółwiem.

Przesunięcie pióra może się też odbywać bez zostawiania jakichkolwiek „śladów”. Służy do tego funkcja `MoveToEx()`:

```
BOOL MoveToEx(HDC hdc,
              int X,
              int Y,
              LPPOINT lpPoint);
```

Byłbym bardzo zawiedziony, gdybyś z jej prototypu nie wydedukował znaczenia parametrów... Myślę więc, że poradziłeś sobie z tym :) Powiem jedynie, że wskaźnik do struktury `POINT`, jakiego funkcja żąda w ostatnim parametrze, może być ustawiony na `NULL`. W takiej sytuacji nie otrzymamy poprzednich współrzędnych pióra - zazwyczaj i tak nie są nam one potrzebne.

Jeżeli jednak zdarzyłaby się taka okoliczność, można się jeszcze salwować funkcją `GetCurrentPositionEx()`:

```
BOOL GetCurrentPositionEx(HDC hdc, LPPOINT lpPoint);
```

Przy okazji rysowania czterokolorowego kwadratu, mogłeś zobaczyć, że bywa ona przydatna.

Tryb rysowania

Rysując linie proste, krzywe, obramowania figur i inne kształty, pióro domyślnie zastępuje już istniejące piksele w kontekście urządzenia. Te standardowe zachowanie możemy zmienić - służy do tego funkcja `SetROP2()`:

```
int SetROP2(HDC hdc,
            int fnDrawMode);
```

ROP jest tu skrótem od *raster operation*, czyli 'operacji rastrowej', zaś 2 oznacza ilość argumentów tej operacji. Owymi dwiema danymi są tutaj:

- istniejący w kontekście urządzenia kolor, zwany kolorem ekranowym (oznaczę go tutaj `clScreen`)
- kolor pióra (`clPen`)

Operacja rastrowa definiuje sposób połączenia tych dwóch kolorów wejściowych w jeden kolor wynikowy. Barwa ta pozostanie na pikselu w kontekście urządzenia.

Windows GDI oferuje kilka możliwych operacji rastrowych. Wszystkie one są działaniami na bitach kanałów RGB, więc w tabeli są one zapisane przy użyciu obecnych w C++ operatorów bitowych¹⁴³:

flaga operacji	kolor wynikowy
<code>R2_BLACK</code>	czarny
<code>R2_NOTMERGEPEN</code>	$\sim(\text{clPen} \mid \text{clScreen})$
<code>R2_MASKNOTPEN</code>	$\text{clScreen} \& \sim\text{clPen}$
<code>R2_NOTCOPYPEN</code>	$\sim\text{clPen}$

¹⁴³ Dla przypomnienia: `&` to koniunkcja bitowa (daje 1 tylko dla dwóch jedynek), `|` to alternatywa (daje 0 tylko dla dwóch zer), `^` jest różnicą symetryczną (daje 1 dla różnych bitów), zaś `~` to negacja, zmieniająca 1 w 0 i odwrotnie.

flaga operacji	kolor wynikowy
R2_MASKPENN	clPen & ~clScreen
R2_NOT	~clScreen
R2_XORPEN	clPen ^ clScreen
R2_NOTMASKPEN	~(clPen & clScreen)
R2_MASKPEN	clPen & clScreen
R2_NOTXORPEN	~(clPen ^ clScreen)
R2_NOP	clScreen
R2_MERGEINVERT	clScreen ~clPen
R2_COPYPEN	clPen
R2_MERGEINVERT	clPen ~clScreen
R2_MERGEINVERT	clPen clScreen
R2_WHITE	biały

Tabela 63. Stałe binarnych operacji rastrowych w Windows GDI

Poszczególne operacje możesz wypróbować empirycznie, jeśli chcesz. Rzadko jednak będziesz musiał korzystać z innego trybu rysowania niż domyślny R2_COPYPEN.

Pędzel

Jeśli chodzi o rysowanie figur geometrycznych w Windows, to wraz z piórem w parze idzie tu zawsze **pędzel**.

Pędzel (ang. *brush*) decyduje o **sposobie wypełniania** zamkniętych powierzchni.

Typem uchwytu do pędzla jest HBRUSH.

Wypełnienie pędzlem jest stosowane dla wszystkich figur zamkniętych, jakie rysujemy w kontekście urządzenia. Należą do nich na przykład prostokąty i okręgi. Innym, bardzo ważnym zastosowaniem pędzla jest też pokrywanie jakimś wzorem lub kolorem wnętrza okna. Pędzel był pierwszym obiektem GDI, z jakim w ogóle mieliśmy do czynienia. Spotkaliśmy go bowiem już przy rejestrowaniu klasy okna, gdzie pole `WNDCLASS[EX]::hbrBackground` musiało zawierać nic innego, jak tylko uchwyt do pędzla malującego obszar klienta okna.

Obecnie jednak skoncentrujemy się głównie na zastosowaniu pędzli w działaniach rysunkowych biblioteki Windows GDI. Powiemy więc sobie, jak się je tworzy i korzysta z nich.

Korzystanie z pędzli

Z pędzli korzystamy identycznie jak z piór. Także tutaj występuje więc ich tworzenie, wybieranie, odkładanie i zwalnianie.

Tworzenie pędzla

Windows GDI oferuje pędzle malujące powierzchnie aż na trzy sposoby. Mogą być one wypełniane:

- jednolitym kolorem (ang. *solid color*)
- dwukolorowym deseniem (ang. *hatch*)
- kafelkowaną bitmapą (ang. *pattern*)

W związku z tym mamy trzy podstawowe funkcje tworzące obiekty pędzli.

Pierwszą z nich jest `CreateSolidBrush()`, najprostsza z nich wszystkich:

```
HBRUSH CreateSolidBrush(COLORREF crColor);
```


Wynikiem jest działania jest pędzel oferujący wypełnienie całkowicie jednolitym kolorem. Wartość tej barwy podajemy oczywiście w jednym parametrze funkcji, `crColor`.

Innym rodzajem wypełnienia jest desień. Pędzel, który będzie je stosował, należy stworzyć funkcją `CreateHatchBrush()`:

```
HBRUSH CreateHatchBrush(int fnStyle,
                        COLORREF clrref);
```

Parametr `fnStyle` określa w niej rodzaj deseni, jakim będzie się posługiwał nasz pędzel. Możliwych jest sześć rodzajów, przedstawia je poniższa tabelka:

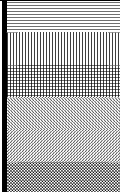
<i>stała</i>	<i>nazwa deseni</i>	<i>desień</i>
HS_HORIZONTAL	poziomy	
HS_VERTICAL	pionowy	
HS_CROSS	siatka	
HS_FDIAGONAL	ukośny w dół	
HS_BDIAGONAL	ukośny w górę	
HS_DIAGCROSS	kratka	

Tabela 64. Wzory deseni dla pędzli Windows GDI

Nie możemy niestety uzupełniać tego zestawu o własne desenie. Powyższa szóstka jest chyba jednak wystarczająca do większości potrzeb.

Można zauważyć, że desień jest rysowany dwoma kolorami. Pierwszy jest nazywany **kolorem pierwszoplanowym** (ang. *foreground color*) - w tabeli jest to kolor czarny. Drugi kolor jest **drugoplanowym** albo **kolorem tła** (ang. *background color*). Obie te barwy możemy ustalić, chociaż tylko kolor pierwszoplanowy jest własnością samego pędzla - podajemy go bowiem w parametrze `clrref`. Kolor tła jest natomiast atrybutem kontekstu urządzenia i możemy go zmienić funkcją `SetBkColor()`:

```
COLORREF SetBkColor(HDC hdc, COLORREF crColor);
```

Jak się później przekonamy, wpływa on nie tylko na wypełnianie deseniowymi pędzlami, ale też na wypisywanie tekstu.

Ostatni rodzaj pędzla potrafi zamalowywać powierzchnie sąsiadującymi kopiami bitmap; nazywamy to często kafelkowaniem. Utworzenie pędzla tego typu wymaga wywołania funkcji `CreatePatternBrush()`:

```
BRUSH CreatePatternBrush(HBITMAP hbitmap);
```

Ta zaś wymaga tylko jednego parametru: jest to uchwyt do obiektu bitmapy, która będzie służyć za „kafelek”. O tworzeniu bitmap będziemy wiele mówić w poświęconej im sekcji, na razie więc powiem tylko, że obrazek do kafelkowania może być wzięty zarówno z pliku, jak i stworzony programowo. Nie ma też ograniczeń, co do rodzaju bitmapy: dozwolony jest obrazek rastrowy typu DIB (bitmapa niezależna od urządzenia), jak i normalna bitmapa, zapisywana w plikach z rozszerzeniem *.bmp*.

Jest jednak wymagane, aby podana **bitmapa istniała przez cały czas** istnienia pędzla. Nie powinniśmy bowiem usuwać żadnej bitmapy, która jest związana z jakimkolwiek pędzlem (podobnie jak nie możemy usuwać pędzla, pióra czy innego obiektu związanego z kontekstem urządzenia).

Wybieranie pędzla

Jeśli chcemy wykorzystać pędzel, musimy wybrać go w swoim kontekście urządzenia. Operacja ta wygląda dokładnie identycznie jak wiązanie pióra. W jej przypadku także musimy więc pamiętać o tym, aby zapisać stary pędzel kontekstu:

```
HBRUSH hbrZielonaSiatka = CreateHatchBrush(HS_DIAGCROSS, 0x0000ff00);
HBRUSH hbrStaryPedzel = (HBRUSH) SelectObject(hdcKontekst,
                                                hbrZielonaSiatka);
```

Jak tego dokonamy, nie ma znaczenia. Tutaj pokazuję najczęstszą metodę z deklaracją osobnej zmiennej, ale równie dobrze możemy użyć mechanizmu zapisywania stanu kontekstu.

Zwalnianie pędzla

Po użyciu pędzel należy odłożyć, a następnie usunąć. Ponownie wygląda to w zasadzie tak samo, jak dla piór:

```
SelectObject ((hdcKontekst, hbrStaryPedzel);
DeleteDC ((hdcKontekst); // usunięcie pędzla hbrStaryPedzel
DeleteObject (hbrZielonaSiatka); // usunięcie pędzla hbrZielonaSiatka
```

Ważne, aby zarówno stary, jak i nasz własny pędzel zostały usunięte. Sposób, w jaki to się dokona nie jest już tak istotny dla biblioteki GDI, więc mamy tutaj pełną swobodę wyboru.

Wiedźmy też, że gdy usuwamy pędzel kafelkujący bitmapowy wzór, to zwolnieniu podlega sam obiekt pędzla i tylko on. Całkowicie bez szwanku wychodzi z tego bitmapa. Tak więc jeśli nie potrzebujemy jej już dłużej, powinniśmy usunąć wykorzystywaną bitmapę poprzez oddzielne wywołanie `DeleteObject()`.

Elastyczny pędzel

Odpowiednikiem elastycznego pióra w obiektach pędzli jest **elastyczny pędzel**. Potrafi on malować kształty jednolitym kolorem, którego odcień można swobodnie i wygodnie zmieniać. Jest to więc użyteczne, gdy musimy wyrysować wiele figur o zmieniających się kolorach.

Elastyczny pędzel, tak samo jak i pióro, są dostępne od Windows 2000 wzwyż.

Wybór elastycznego pędzla

Uzyskanie uchwytu do elastycznego pędzla oznacza wywołanie funkcji `GetStockObject()` z parametrem `DC_BRUSH`. Jeżeli zaś chcemy ustawić ten pędzel w kontekście urządzenia, to oczywiście podajemy go do `SelectObject()`:

```
SelectObject ((hdcKontekst, GetStockObject(DC_BRUSH));
```

Nic nie stoi też na przeszkodzie, aby jednocześnie usunąć poprzedni pędzel kontekstu. Musimy tylko (tak samo jak przy elastycznym piórze) objąć przywołanie `SelectObject()` funkcją `DeleteObject()`.

Zmiana koloru pędzla

Kolor elastycznego pędzla leży pod kontrolą funkcji `SetDCBrushColor()`. Używamy jej identycznie jak `SetDCPenColor()`, tzn. podajemy uchwyt kontekstu urządzenia oraz nowy kolor, np. tak:

```
// zmiana koloru elastycznego pędzla na szary
SetDCBrushColor ((hdcKontekst, RGB(128, 128, 128));
```

Ilustracją dla elastycznego pędzla niech będzie poniższy, całkiem efektowny program. Nie robi on niczego konkretnego, ale prezentuje prosty fajerwerk graficzny:

```

// RandomRects - inwazja losowych prostokątów

#include <string>
#include <ctime>
#define _WIN32_WINNT 0x500          // aby działał elastyczny pędzel
#define WIN32_LEAN_AND_MEAN
#include <windows.h>

// nazwa klasy okna
std::string g_strKlasaOkna = "od0dogk_Window";

// dane okna
HDC g_hdcOkno;          // uchwyt kontekstu urządzenia okna

// pomocnicza funkcja zwracająca liczbę losową z podanego zakresu -----

int Random(int nMin, int nMax)
{ return rand() % (nMax - nMin + 1) + nMin; }

// ----- procedura zdarzeniowa okna -----

LRESULT CALLBACK WindowEventProc(HWND hWnd, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
    switch (uMsg)
    {
        case WM_TIMER:
        {
            // pobieramy obszar klienta
            RECT rcObszarKlienta;
            GetClientRect (hWnd, &rcObszarKlienta);

            // generujemy współrzędne nowego prostokąta;
            // prawą i dolną krawędź dobieramy tak,
            // aby zawsze była położona między lewą/górną krawędzią
            // prostok. i prawą/dolną krawędzią obszaru klienta okna
            RECT rcProstokat;
            rcProstokat.left = Random(0, rcObszarKlienta.right);
            rcProstokat.right = Random(rcProstokat.left,
                                     rcObszarKlienta.right);
            rcProstokat.top = Random(0, rcObszarKlienta.bottom);
            rcProstokat.bottom = Random(rcProstokat.top,
                                       rcObszarKlienta.bottom);

            // ustawiamy losowy kolor pędzla
            SetDCBrushColor (g_hdcOkno,
                           RGB(Random(0, 255),
                                Random(0, 255),
                                Random(0, 255)));

            // rysujemy prostokąt
            Rectangle (g_hdcOkno, rcProstokat.left, rcProstokat.top,
                     rcProstokat.right, rcProstokat.bottom);

            return 0;
        }

        case WM_DESTROY:
    }
}

```

```

        // kończymy program
        PostQuitMessage (0);
        return 0;
    }

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

// -----funkcja WinMain() -----
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int nCmdShow)
{
    /* rejestrujemy klasę okna */

    WNDCLASSEX KlasaOkna;

    // wypełniamy strukturę WNDCLASSEX
    ZeroMemory (&KlasaOkna, sizeof(WNDCLASSEX));
    KlasaOkna.cbSize = sizeof(WNDCLASSEX);
    KlasaOkna.hInstance = hInstance;
    KlasaOkna.lpfnWndProc = WindowEventProc;
    KlasaOkna.lpszClassName = g_strKlasaOkna.c_str();
    KlasaOkna.hCursor = LoadCursor(NULL, IDC_ARROW);
    KlasaOkna.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    KlasaOkna.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
    KlasaOkna.style = CS_OWNDC // własny kontekst urządzenia okna
        | CS_HREDRAW | CS_VREDRAW;

    // rejestrujemy klasę okna
    RegisterClassEx (&KlasaOkna);

    /* tworzymy okno */

    HWND hOkno;
    // (darujemy sobie wywołanie CreateWindowEx())

    // pokazujemy nasze okno
    ShowWindow (hOkno, nCmdShow);

    /* przygotowujemy się do rysowania prostokątów */

    // pobieramy uchwyt do kontekstu urządzenia obszaru klienta okna
    g_hdcOkno = GetDC(hOkno);

    // ustawiamy mu elastyczny pędzel
    DeleteObject (SelectObject(g_hdcOkno, GetStockObject(DC_BRUSH)));

    // tworzymy stoper, aby generował zdarzenie WM_TIMER
    // co ćwierć sekundy
    SetTimer (hOkno, 1, 250 /* milisekund */, NULL);

    // inicjujemy generator liczb pseudolosowych
    srand (static_cast<unsigned>(time(NULL)));

    /* pętla komunikatów */

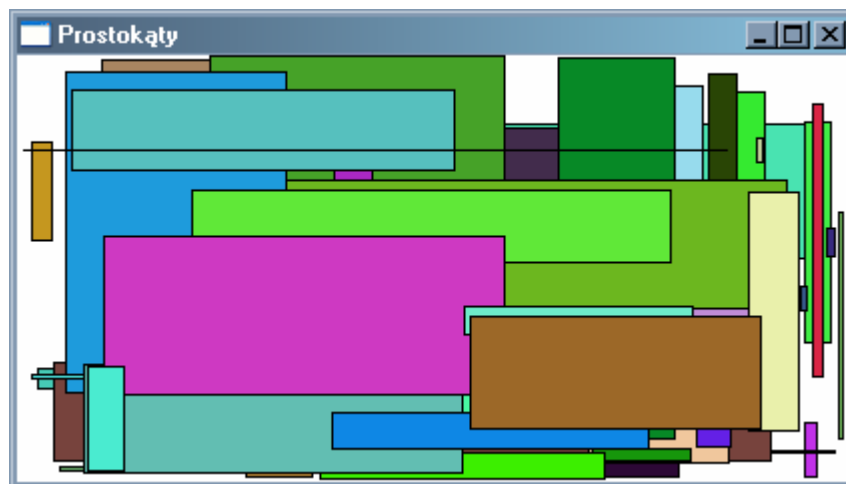
    MSG msgKomunikat;

```

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

// zwracamy kod wyjścia
return static_cast<int>(msgKomunikat.wParam);
}
```

Gdy uruchomimy ten program, zobaczymy prostokąty o losowych wymiarach, wypełnione losowym kolorem i pojawiające się w losowych miejscach. Jednym słowem: całkowity chaos :)



Screen 68. Efekt losowych prostokątów

Właśnie ze względu na tę przypadkowość, elastyczny pędzel sprawdza się tu dobrze. Gdybyśmy bowiem dla każdego prostokąta generowali odrębny obiekt pędzla z innym kolorem, mogłoby to nawet w widoczny sposób zaważyć na wydajności.

W tym programie przykładowym użyłem stopera (ang. *timer*), o którym jeszcze nie mówiliśmy i omówimy w jednym z przyszłych rozdziałów. Jeżeli chcesz, możesz o tym poczytać w [MSDN](#), bo jest w gruncie rzeczy bardzo łatwe zagadnienie. Stoper wysyła po prostu komunikat `WM_TIMER` do okna programu w określonych odstępach czasu - u nas jest to 250 milisekund, czyli ćwierć sekundy. Aplikacja może zaś reagować na to zdarzenie i wykonywać jakieś cykliczne akcje, jak np. mało sensowne rysowanie prostokątów :)

Wypełnianie obszaru rysunku

W powyższym przykładzie użyłem prostokątów wypełnionych kolorami. W zasadzie jednak nie trzeba nawet rysować żadnych figur, aby móc skorzystać z pędzla. Potrafi on bowiem wykonać czynność znaną dobrze z programów graficznych - **wypełnienie obszaru** (ang. *flood fill*, dosłownie 'wypełnienie powodziowe', co można też tłumaczyć jako 'wylanie farby').

Jak działa wypełnienie obszaru, możesz się przekonać, uruchamiając choćby program Paint, wybierając narzędzie *Wypełnienie kolorem* i klikając w dowolne miejsce obrazka. Zobaczysz zwykle, że spory obszar rysunku został pokryty nowym kolorem. Technicznie rzecz biorąc, Wypełnianie kolorem działa w ten sposób, iż zastępuje barwy wszystkich pikseli, które sąsiadują z tym klikniętym i mają taką samą barwę jak on. Tych pikseli

może być bardzo dużo, a wtedy nowy kolor „rozlewa” się na znacznym fragmencie bitmapy. Stąd też wzięła się angielska nazwa tego rodzaju wypełniania.

Biblioteka Windows GDI jest mądrzejsza niż Paint, bo oferuje nie tyle wypełnianie jakimś kolorem, ale każdym możliwym rodzajem pędzla (w tym także jednolicie kolorującym). Ponadto udostępnia też dodatkowy sposób wyznaczania obszaru, na który zostanie „wylana farba”.

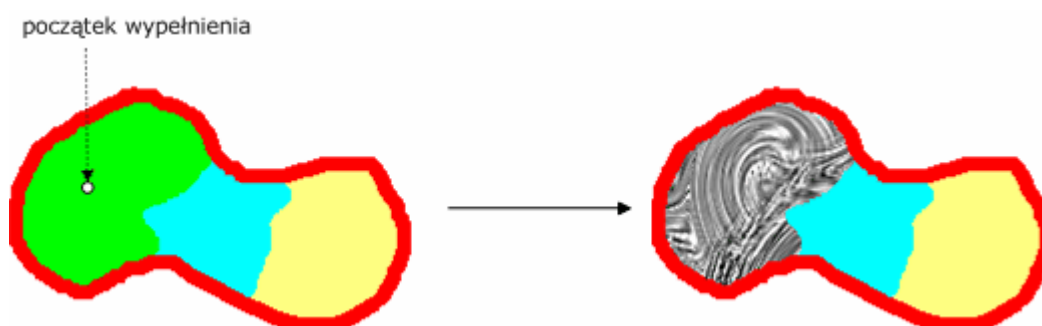
Za czynność wypełniania odpowiada funkcja `ExtFloodFill()`:

```
BOOL ExtFloodFill(HDC hdc,
                  int nXStart,
                  int nYStart,
                  COLORREF crColor,
                  UINT fuFillType);
```

Jeśli chodzi o jej parametry, to ich znaczenie jest raczej nietrudne do zrozumienia. `hdc` to kontekst urządzenia, w którym będzie grasować wypełnienie. Liczby `nXStart` i `nYStart` są współrzędnymi punktu, z którego efekt weźmie swój początek; w programach graficznych jest to ten punkt, w który klikamy myszą. Kolor `crColor` może mieć dwa znaczenia, w zależności od typu wypełniania, podanego w `fuFillType`.

`ExtFloodFill()` obsługuje dwa typy wypełniania pędzlem, które rozróżnia za pomocą wartości parametru `fuFillType`:

- `FLOODFILLSURFACE` jest znanym nam sposobem, polegającym na wypełnianiu wszystkich pikseli, które sąsiadują z tym „klikniętym” i mają ten sam kolor. Polega to po prostu na zastąpieniu pewnego obszaru, wypełnionego danym kolorem, obszarem pokrytym wzorem pędzla:



Rysunek 17. Wypełnienie funkcją `ExtFloodFill()` w trybie `FLOODFILLSURFACE`

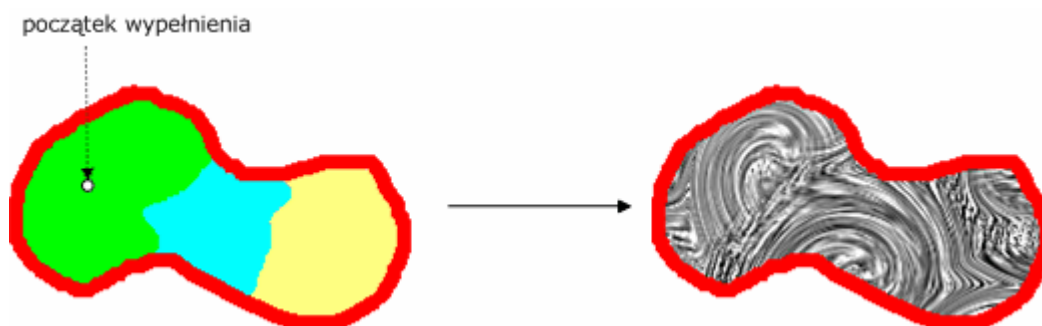
Przy taki trybie wypełniania, wartość parametru `crColor` musi zgadzać się w z kolorem w punkcji o podanych współrzędnych `nXStart` i `nYStart`. Najlepiej więc pobrać go stamtąd i napisać np. taką funkcję:

```
BOOL Fill(HDC hdc, int nX, int nY)
{
    return ExtFloodFill(hdc, nX, nY,
                        GetPixel(hdc, nX, nY), FLOODFILLSURFACE);
}
```

Funkcją `GetPixel()` pobieramy kolor piksela o podanych współrzędnych

- `FLOODFILLBORDER` to drugi sposób wypełniania. W tym ustawieniu działa ona właściwie odwrotnie niż poprzednim: kiedy tam kolor podany w `crColor` był swego rodzaju warunkiem kontynuowania wypełniania, tutaj jest on kryterium jego

zakończenia. Mówiąc prościej, w tym trybie funkcja `ExtFloodFill()` radośnie maluje pędzlem cały rysunek aż do napotkania obramowania w kolorze `crColor`:



Rysunek 18. Wypełnienie funkcją `ExtFloodFill()` w trybie `FLOODFILLBORDER`

Wtedy kończy ona swe działanie. Jego wynikiem jest wypełnienie pędzlem całego obszaru, znajdującego się wewnątrz obramowania w kolorze `crColor`. Na rysunku powyżej jest to kolor czerwony

Używając `ExtFloodFill()` musimy dbać o to, aby warunki działania tej funkcji były spełnione. Znaczy to na przykład, że przy stosowaniu trybu `FLOODFILLSURFACE` kolor piksela o współrzędnych `nXStart` i `nYStart` powinien rzeczywiście być równy `crColor`. Dla `FLOODFILLBORDER` musi być z kolei odwrotnie: piksel ten nie może mieć koloru podanego w czwartym parametrze, bo w tej sytuacji funkcja nie będzie miała czego wypełniać.

Pamiętajmy też, że funkcja poszukuje **dokładnego dopasowania** koloru podanego w `crColor`. Jeśli więc podamy jej barwę `RGB(34, 56, 178)`, to `RGB(34, 57, 178)` będzie uznana za całkowicie inny kolor, mimo że optyczna różnica między nimi jest zasadzie żadna. Może to powodować powstawanie niewypełnionych „dziur” lub przeciwnie, wypełnionych „plam”, których wcale nie chcieliśmy pokrywać pędzlem.

Kwestia ta w programach graficznych nieco lepszych od Painta nosi nazwę **tolerancji koloru** i jest zwykle ustawialna. Nie tylko można podać jej liczbową wartość (zazwyczaj od 0 do 255 - im więcej, tym większa tolerancja zmiany koloru), ale także sposób, w jaki kolory są dopasowywane do tego poszukiwanego. Możliwe jest nie tylko porównywanie kanałów RGB, czyli ilości czerwieni, zieleni i błękitu, ale też konfrontacja odcienia, jasności, optycznego wrażenia koloru czy też wartości kanałów barw podstawowych w innych systemach, np. CMYK.

Niestety, w przypadku `ExtFloodFill()` tolerancja wynosi zawsze 0 i nie da się jej ustawić. Można co najwyżej napisać własną funkcję...

Punkty i linie

Wykreślanie figur zamkniętych zaczniemy od punktów, linii prostych i krzywych. Nie są to zamknięte kształty, więc za ich wygląd odpowiada wyłącznie pióro. Współpracę pióra i pędzla zobaczymy dopiero przy takich figurach jak prostokąty czy elipsy.

Na razie jednak nie zajmujemy się nimi, lecz poprzestaniemy na rysowaniu punktów, odcinków, łamanych oraz linii krzywych.

Zaznaczanie punktu

Pomyślmy sobie: po co nam w zasadzie taka rozbudowana biblioteka graficzna jak Windows GDI? Przecież większość jej czynności sprowadza się do stawiania kolorowych

pikseli na bitmapach. Mając funkcję od tego, moglibyśmy teoretycznie obyć się bez całej reszty procedur i interfejsów!...

Cóż, teoria teorią, ale praktyka uczy, że lepiej nie wyważać otwartych drzwi. Jest prawie pewne, że wszystkie funkcje rysujące GDI zostały napisane tak szybko, jak to tylko było możliwe. Istnieje więc bardzo niewielkie prawdopodobieństwo, że moglibyśmy napisać lepsze wersje algorytmów kreślących np. linie czy prostokąty. Takie operacje są zresztą nierzadko optymalizowane sprzętowo, a z kartą graficzną naprawdę nie radzę się ścierać :)

Poza tym, czy aby na pewno chcielibyśmy pisać wszystko sami?... Oczywiście, że nie! W końcu po to ktoś wymyśla, projektuje i implementuje takie biblioteki jak Windows GDI, aby ułatwić innym programistom wykonywanie powtarzalnych czynności. Rysowanie skomplikowanych figur geometrycznych czy grafiki w ogóle jest taką właśnie czynnością. Mamy więc interfejs GDI, który radzi sobie z nim; zapewniam cię, że nauka posługiwania się nim jest przynajmniej o kilka rzędów wielkości łatwiejsza od samodzielnego pisania podobnego kodu.

A jednak czasami trudno obyć się bez bezpośredniej modyfikacji pikseli. To wszakże nie problem, jako że GDI udostępnia i taką działalność graficzną. Zobaczmy więc, jak się ona odbywa.

Ustawienie piksela na określony kolor

A zatem - aby ustawić piksel o znanej współrzędnej na żądany kolor, wywołujemy funkcję o wiele mówiącej nazwie `SetPixel()`:

```
COLORREF SetPixel(HDC hdc,
                  int X,
                  int Y,
                  COLORREF crColor);
```

Cóż można o niej powiedzieć? Właściwie sposób jej użycia jednoznacznie wynika z prototypu. Napiszę więc tylko, że wartością zwracaną przez funkcję jest kolor, na który piksel został ustawiony. Barwa ta może się różnić od parametru `crColor`, jeśli kontekst urządzenia nie obsługuje pełnego spektrum 24-bitowych kolorów RGB.

Nieco szybciej

Zazwyczaj nie potrzebujemy informacji, jaką zwraca `SetPixel()`. W takim wypadku możemy użyć jej wydajniejszej wersji, `SetPixelV()`:

```
BOOL SetPixelV(HDC hdc,
               int X,
               int Y,
               COLORREF crColor);
```

Różnica jest zwykle niewielka, ale zawsze lepiej używać wydajniejszego kodu - szczególnie jeśli nie wiąże się to z żadnymi niedogodnościami.

Przykład szumu

Spytasz może: „Jak szybkie jest takie ustawianie pikseli, jeżeli musielibyśmy zapełnić nimi np. cały ekran?” Myślę, że dobrze jest przekonać się o tym samemu. Przedstawię prosty program symulujący zachowanie się... telewizora odłączonego od anteny :) Oto najważniejszy fragment kodu tego programu:

```
// DeadTV - efekt zepsutego telewizora ;)

// zmienność efektu
// (liczba pikseli zmienianych z każdym przebiegiem)
```



```
const unsigned ZMIENNOSC = 2500;

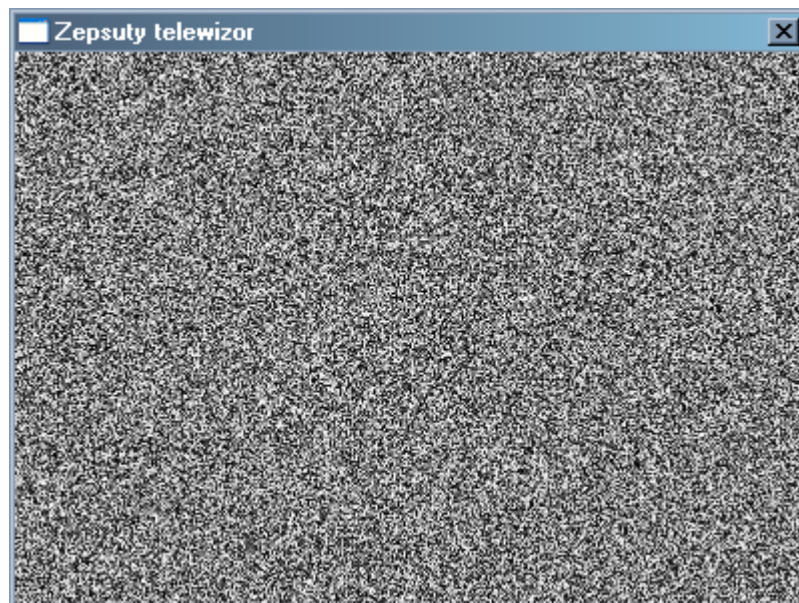
// ----- funkcja wykonywana w każdym przebiegu efektu -----
void Pracuj()
{
    POINT ptPiksel;
    BYTE byOdcien;

    // wybieramy pewną ilość pikseli i zmieniamy ich kolory
    for (unsigned i = 0; i < ZMIENNOSC; ++i)
    {
        // losujemy współrzędne zmienianego piksela
        ptPiksel.x = rand() % g_rcObszarKlienta.right;
        ptPiksel.y = rand() % g_rcObszarKlienta.bottom;

        // losujemy odcień szarości
        byOdcien = rand() % 256;

        // zmieniamy piksel
        SetPixelV (g_hdcOkno, ptPiksel.x, ptPiksel.y,
                  RGB(byOdcien, byOdcien, byOdcien));
    }
}
```

Jego ciągle wykonywanie się powoduje powstanie efektu charakterystycznego szumu:



Screen 69. Efekt jednorodnego szumu

Im większa częstość zmiany pikseli, tym bardziej program upodabnia się prawdziwego „śniegu” w telewizorze. A ponieważ w tym przykładzie zastosowałem nie stoper, lecz pętlę komunikatów z `PeekMessage()`, efekt wykonuje się tak szybko, na ile pozwalają mu możliwości komputera.

Nie wiem oczywiście, jak wygląda on na twoim sprzęcie, lecz bardzo prawdopodobne, że dosyć niewiele brakuje mu do doskonałości. Biblioteka GDI wydajnościowo sprawdza tu się zatem całkiem dobrze, a musisz wiedzieć, że ustawianie kolorów pojedynczych pikseli jest newralgicznym punktem każdego interfejsu graficznego.

Linie proste

Teraz przyszedł na najprostsze krzywe, czyli krzywe, które są... proste :) Zobaczmy tutaj, jak można wykreślać odcinki oraz łamane. Figury te są rysowane przy pomocy pióra wybranego w kontekście urządzenia, zatem zmiana ich koloru czy szerokości linii odbywa się poprzez podmianę obiektu pióra. Jak to zrobić, pisałem w odpowiednim paragrafie poświęconym piórom.

Kreślenie odcinka

Za najprostsze narysowanie odcinka od jednego punktu do drugiego odpowiada w GDI funkcja `LineTo()`:

```
BOOL LineTo(HDC hdc,
            int X,
            int Y);
```

„Hmm... Tu jest chyba za mało parametrów!...” Faktycznie, można tu podać współrzędne tylko jednego punktu - **punktu końcowego**. Po narysowaniu odcinka pióro zostaje w nim umieszczone. Swój początek linia bierze natomiast z aktualnej pozycji pióra. Jeżeli więc chcielibyśmy rozpocząć rysowanie odcinka od ustalonego miejsca, wpierw musimy jeszcze skorzystać z funkcji `MoveToEx()`.

Możemy aczkolwiek napisać sobie taką funkcję, która potrafi kreślić linie o określonym początku i końca. Prawdopodobnie będzie ona wyglądała tak:

```
BOOL Line(HDC hdcKontekst, POINT ptStart, POINT ptKoniec)
{
    POINT ptPoprzedniaPozycja;

    // przesuwamy pióro do ptStart, zapisując jego poprzednią pozycję
    if (!MoveToEx(hdcKontekst, ptStart.x, ptStart.y,
                  &ptPoprzedniaPozycja))
        return FALSE;

    // kreślimy linię do ptKoniec
    if (!LineTo(hdcKontekst, ptKoniec.x, ptKoniec.y))
        return FALSE;

    // przywracamy oryginalną pozycję pióra
    MoveToEx(hdcKontekst, ptPoprzedniaPozycja.x, ptPoprzedniaPozycja.y,
             NULL);

    // zwracamy TRUE
    return TRUE;
}
```

Pamiętajmy, że wymaga ona łącznie trzech operacji, więc jeśli jest to możliwe, lepiej stosuj wbudowaną funkcję `LineTo()`.

Rysowanie łamanej

Co się stanie, gdy parokrotnie wywołamy funkcję `LineTo()`?... Nic strasznego: zwyczajnie wyrysujemy kilka odcinków połączonych ze sobą tak, że koniec jednego jest jednocześnie początkiem następnego. Innymi słowy, narysujemy łamaną.

W GDI można oczywiście stosować i ten sposób, ale mamy przecież funkcje `Polyline()` i `PolylineTo()`:

```
HDC Polyline[To](HDC hdc,
```

```
CONST POINT* lppt,  
int cPoints);
```

Wskaźnik, którego obie funkcje żądają w drugim parametrze, jest tablicą elementów typu `POINT` - czyli współrzędnych punktów. Liczbę elementów tej tablicy określamy w ostatnim parametrze, `cPoints`.

Do funkcji `Polyline()` i `PolylineTo()` musimy tą drogą przekazać co najmniej dwa punkty. Funkcje korzystają z nich, rysując linię łamaną: rozpoczynają od pierwszego punktu, ustawiając w nim pióro; dalej prowadzą linię do kolejnych miejsc opisanych koordynatami elementów tablicy, aż narysują wszystkie odcinki. Nietrudno wyliczyć, że będzie to łącznie `cPoints - 1` kresek.

Teraz pewnie zapytasz, czym różnią się obydwie funkcje. I bardzo słusznie, go w ten sposób dojdziemy do ogólniejszej, a ważnej i przydatnej zasady. Otóż `PolylineTo()` zachowuje się dokładnie tak samo jak ciąg poleceń:

```
MoveToEx (hdc, lppt[0].x, lppt[0].y, NULL);  
LineTo (hdc, lppt[1].x, lppt[1].y, NULL);  
LineTo (hdc, lppt[2].x, lppt[2].y, NULL);  
...  
LineTo (hdc, lppt[cPoints - 1].x, lppt[cPoints - 1].y, NULL);
```

Natomiast `Polyline()` można przedstawić jako:

```
POINT ptPos;  
MoveToEx (hdc, lppt[0].x, lppt[0].y, &ptPos);  
  
LineTo (hdc, lppt[1].x, lppt[1].y, NULL);  
LineTo (hdc, lppt[2].x, lppt[2].y, NULL);  
...  
LineTo (hdc, lppt[cPoints - 1].x, lppt[cPoints - 1].y, NULL);  
  
MoveToEx (hdc, ptPos.x, ptPos.y, NULL);
```

Jestem przekonany, że dostrzegasz różnicę - zwłaszcza, jeżeli przypomnisz sobie zaprezentowaną wcześniej funkcję `Line()`.

Powiedzmy jednak jasno, o co chodzi. Mianowicie, `PolylineTo()` kończy rysowanie, zostawiając pióro w pozycji ostatniego punktu - ma zatem wpływ na jeden z atrybutów kontekstu urządzenia. Z kolei `Polyline()` jest kulturalniejsza: po zakończonym rysowaniu przywraca pióro do pierwotnej pozycji tak, że wydaje nam się, iż nie zostało ono w ogóle poruszone.

Oczywiście nie należy mówić, że druga z funkcji jest „grzeczniejsza” niż pierwsza. Dla nas ważne jest, że obie takie funkcje istnieją i zawsze można wybierać wariant bardziej pasujący w danej chwili.

Co więcej, wyłaniającą się tu zasadę można uogólnić dla wszystkich funkcji rysujących krzywe otwarte. Brzmi ona:

Funkcje o nazwach zakończonych na `To` **zmieniają pozycję pióra**, ustawiając je w **miejscu, gdzie skończyły** rysowanie.

Podwójne wersje procedur istnieją również dla łuków elips oraz krzywych Béziera. Poznamy je wszystkie za moment.

Do wykreślania łamanych można jeszcze wykorzystać funkcję `PolyPolyline()`. Potrafi narysować kilka łamanych za jednym „zamachem” wirtualnego pióra. Trudno wprawdzie znaleźć ku temu jakieś konkretne zastosowanie nawet przy dużej dozie entuzjazmu, ale

przyjrzenie się [opisowi tej funkcji w MSDN](#) może być ciekawe. Jest on bowiem ideowo bardzo podobny do metod interfejsu DirectX, renderujących trójwymiarowe prymitywy. Tam nieustannie używa się tablic punktów oraz innych tablic liczb, opisujących te pierwsze - tak jak w `PolyPolyline()`.

Krzywe otwarte

Chcąc narysować krzywą, moglibyśmy spróbować jej przybliżeniu odpowiednio krótkimi odcinkami łamanej. To oczywiste, że nie odcinków tych nie możemy skracać w nieskończoność, ale przecież nie ma takiej potrzeby, ponieważ rozdzielczość każdego urządzenia, a tym bardziej monitora, jest skończona.

Windows GDI posiada aczkolwiek bardziej wyspecjalizowane funkcje rysujące krzywe otwarte. Radzą sobie one z łukami elips oraz z krzywymi Béziera.

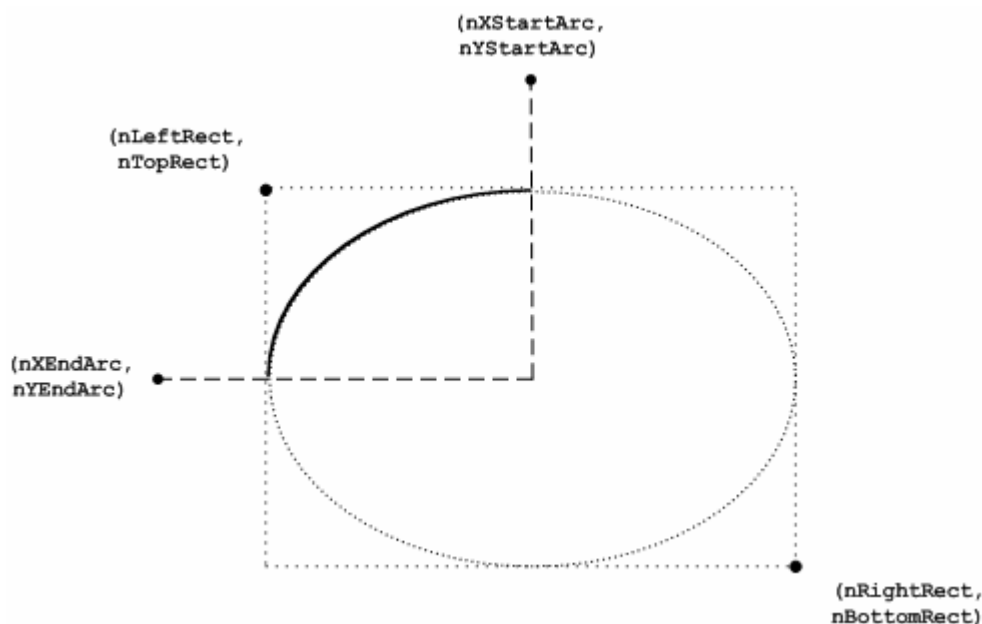
Łuki

Są dwie funkcje, które rysują łuki elips: `Arc()` i `ArcTo()`. Różnica między nimi sprowadza się tylko do tego, iż `ArcTo()` uaktualnia pozycję pióra, ustawiając je na końcu łuku; `Arc()` tego nie robi.

Prototyp obu funkcji wygląda tak:

```
BOOL Arc[To] (HDC hdc,
              int nLeftRect,
              int nTopRect,
              int nRightRect,
              int nBottomRect,
              int nXStartArc,
              int nYStartArc,
              int nXEndArc,
              int nYEndArc);
```

Niestety, oczy cię nie mylą: oto skromne *dziewięć* parametrów. Byłoby trudno dociec ich znaczenia z samego tylko nagłówka funkcji, ale pewnie i opis słowny niewiele tu pomoże. Najlepszy będzie rysunek, który podpowie znaczenie wszystkich argumentów:



Rysunek 19. Znaczenie parametrów funkcji `Arc()` i `ArcTo()`

Zastosowane tu podejście jest dość ciekawe: łuk definiowany jest przez całą elipsę, do której należy, oraz tzw. punkt początkowy (o współrzędnych `nXStartArc` i `nYStartArc`) i końcowy (`nXEndArc` i `nYEndArc`). Wraz z środkiem elipsy wyznaczają one dwa odcinki, a miejscach, gdzie te odcinki przecinają się z tą elipsą, rozpoczyna się i kończy łuk. Jak można zauważyć, jest on kreślony w kierunku **przeciwnym do ruchu wskazówek zegara** (ang. *counterclockwise*).

Co do samej elipsy, to jest ona wyznaczona przez otaczający ją najmniejszy prostokąt. Kolejne parametry - `nLeftRect`, `nTopRect`, `nRightRect` i `nBottomRect` - są odpowiednikami pól struktury `RECT`. Jak przekładają się one na kształt owalu, widać na rysunku; o wykreślaniu elipsy będziemy zresztą mówić w akapicie poświęconym figurom zamkniętym.

Zauważmy, że punkty początkowe i końcowe łuku nie muszą koniecznie leżeć na samej elipsie - Windows nie wymaga aż takiej precyzji. Mogą być one umieszczone w dowolnej odległości na zewnątrz elipsy.

Nie da się jednak ukryć, że ten sposób określania kawałka obwodu owalu jest kłopotliwy. Wygodniej byłoby podawać go jako zakres kątów. Nic aczkolwiek nie stoi na przeszkodzie, aby napisać procedurę, która będzie spełniała to żądanie:

```
#include <cmath>

void EllipticArc(HDC hdcKontekst,
                 RECT rcElipsa, float fStart, float fKoniec)
{
    // obliczamy środek elipsy
    POINT ptSrodek;
    ptSrodek.x = rcElipsa.left + (rcElipsa.right - rcElipsa.left) / 2;
    ptSrodek.y = rcElipsa.top + (rcElipsa.bottom - rcElipsa.top) / 2;

    // wyliczamy długość promienia wodzącego, który będzie odległością
    // punktu początkowego i końcowego od środka elipsy
    // promień ten musi być dłuższy niż każdy z "boków" elipsy
    int nPromien = max(rcElipsa.right - rcElipsa.left,
                      rcElipsa.bottom - rcElipsa.top);

    // wyznaczamy punkty początkowe i końcowe łuku
    // za pomocą funkcji trygonometrycznych
    POINT ptPoczatek = { ptSrodek.x + nPromien * cos(fStart),
                        ptSrodek.y - nPromien * sin(fStart) };
    POINT ptKoniec = { ptSrodek.x + nPromien * cos(fKoniec),
                      ptSrodek.y - nPromien * sin(fKoniec) };

    // wreszcie kreślimy łuk
    Arc (hdcKontekst,
         rcElipsa.left, rcElipsa.top, rcElipsa.right, rcElipsa.bottom,
         ptPoczatek.x, ptPoczatek.y, ptKoniec.x, ptKoniec.y);
}
```

Używając tej funkcji należy tylko pamiętać, że względu na pewne własności funkcji sinus i cosinus, kąt 0 odnosi się do najbardziej wysuniętej w prawo części elipsy. No i nie zapominajmy, że kąty podajemy zawsze **w radianach**.

Krzywe Béziera

Ten rodzaj krzywych nie jest tak powszechnie znany jak inne, ale w grafice komputerowej mają one bardzo duże znaczenie.

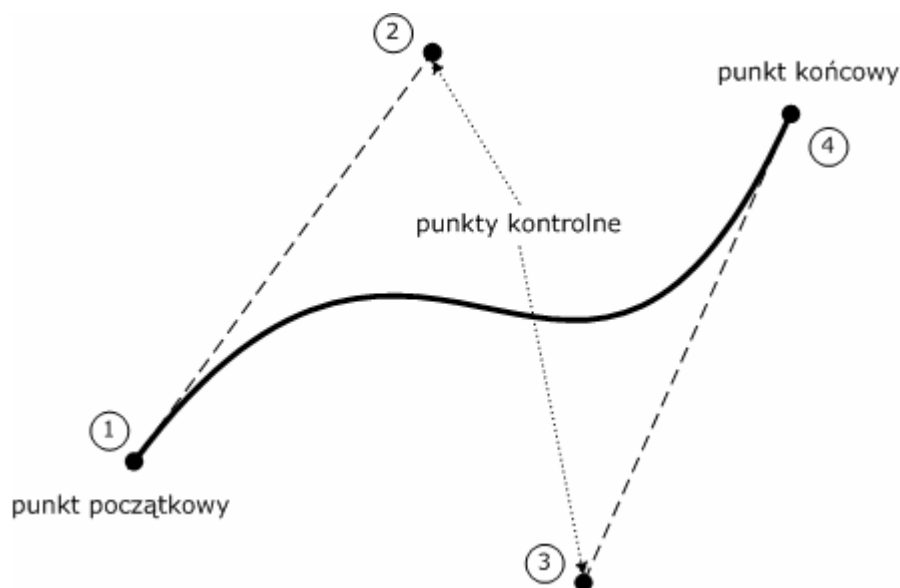
Krzywe Béziera wzięły swoją nazwę od nazwiska Pierre'a Béziera, francuskiego matematyka pracującego w firmie Renault. W latach 60. ubiegłego wieku opracował on ten ciekawy rodzaj krzywych, które stały mu się potem pomocne w projektowaniu

karoserii samochodowych. Później znalazły zastosowanie także w grafie komputerowej, na przykład do opisywania kształtów czcionek proporcjonalnych.

Krzywa Béziera jest wyznaczona przez co najmniej trzy punkty:

- jeden punkt początkowy
- jeden punkt końcowy
- przynajmniej jeden punkt kontrolny

Punkty kontrolne definiują krzywiznę figury: odcinki łączące je punktem początkowym i końcowym potrafią „wyciągać” lub „ściskać” krzywą. Ważnym faktem jest, że krzywa Béziera **nigdy nie przechodzi przez punkty kontrolne**.



Rysunek 20. Krzywa Béziera z dwoma punktami kontrolnymi

Windows GDI rysuje tylko krzywe wyznaczone przez swoje dwa punkty kontrolne. Łącznie zatem potrzebuje do tego czterech punktów, wraz z początkowym i końcowym. Takie krzywe Béziera nazywamy **krzywymi trzeciego stopnia** ze względu na stopień wielomianów w opisujących je równaniach.

Równania opisujące taką krzywą wyglądają mniej więcej tak:

$$x(t) = t^3 \cdot x_p + 3t^2(1-t)x_{k_1} + 3t(1-t)^2x_{k_2} + (1-t)^3x_k$$

$$y(t) = t^3 \cdot y_p + 3t^2(1-t)y_{k_1} + 3t(1-t)^2y_{k_2} + (1-t)^3y_k$$

W tych wzorach współrzędne (x_p, y_p) i (x_k, y_k) oznaczają punkty: początkowy i końcowy, zaś (x_{k_1}, y_{k_1}) i (x_{k_2}, y_{k_2}) są punktami kontrolnymi. Parametr t powinien przebiegać po wartościach od 0 do 1 ($t \in \langle 0; 1 \rangle$); wartość 0 da w wyniku punkt początkowy, 1 - końcowy, a liczby pośrednie pozwalają wyliczyć inne punkty należące do krzywej.

Można zauważyć, że równania dla osi X i Y różnią się tylko tym, że wykorzystujemy w nich inne współrzędne punktów krzywej. To duża zaleta, bo łatwo możemy dodać trzeci wzór, opisujący współrzędną Z, i kreślić krzywe Béziera w przestrzeni trójwymiarowej.

Dla dowolnej liczby punktów kontrolnych równanie krzywej jest trochę bardziej skomplikowane:

$$\vec{B}(t) = \sum_{k=0}^n \binom{n}{k} \cdot t^k (1-t)^{n-k} \cdot \vec{p}_k$$

Opisuje ono krzywą Béziera wyznaczoną przez $(n+1)$ punktów, czyli przez jeden początkowy (\vec{p}_0), jeden końcowy (\vec{p}_n) i $(n-1)$ punktów kontrolnych ($\vec{p}_1, \dots, \vec{p}_{n-1}$).

Koordinaty punktów zapisałem jako wektory, więc równanie działa niezależnie od liczby wymiarów¹⁴⁴: aby uzyskać wzory dla każdej osi układu współrzędnych, wystarczy zamiast \vec{p}_k wstawić odpowiednie współrzędne punktów (x_k , y_k i ewentualnie z_k).

Do narysowania krzywej można się posłużyć funkcjami `PolyBezier()` oraz `PolyBezierTo()`. Różnica między nimi polega oczywiście na tym, że `PolyBezierTo()` uaktualnia pozycję pióra. Oprócz tego funkcja ta traktuje aktualną pozycję pióra jako punkt początkowy krzywej. Oto prototyp obydwu funkcji:

```
BOOL PolyBezier[To](HDC hdc,
                    CONST POINT* lppt,
                    DWORD cCount);
```

Żądają one tablicy punktów wyznaczających krzywą - podajemy ją w parametrze `lppt`, zaś ilość punktów `cCount`. Ilość ta powinna wynosić co najmniej 4 w funkcji `PolyBezier()` i 3 w `PolyBezierTo()`, aby funkcja mogła wyrysować co najmniej jedną figurę.

`PolyBezier[To]()` potrafi bowiem rysować więcej takich krzywych. Są one wtedy połączone ze sobą tak, że punkt końcowy jednej z nich jest jednocześnie punktem początkowym drugiej. Na każdą następną krzywą potrzeba więc już tylko 3 punktów - dwóch kontrolnych i końcowych. Chcąc np. wykreślić cztery połączone krzywe Béziera trzeciego stopnia, musimy użyć 13 punktów dla funkcji `PolyBezier()` lub 12 dla `PolyBezierTo()`.

Połączenie między tak narysowanymi krzywymi może być „kanciaste”. Jeżeli chcemy, aby było gładkie, to musimy pamiętać, żeby dwa punkty sąsiadujące z punktem połączenia były współliniowe. Tzn. chodzi o to, aby drugi punkt kontrolny pierwszej krzywej, punkt połączenia oraz pierwszy punkt kontrolny drugiej krzywej leżały na jednej prostej. W tym celu możemy wyliczyć pozycję pierwszego punktu kontrolnego drugiej krzywej posługując się równaniem:

$$\vec{p}_{k+1} = \vec{p}_k + (\vec{p}_k - \vec{p}_{k-1})$$

\vec{p}_k reprezentuje w nim współrzędne punktu wspólnego dla obu krzywych, zaś indeksy przy wektorach odnoszą się do numerów elementów tablicy przekazywanej do funkcji `PolyBezier[To]()`.

Krzywe Béziera są bardzo użyteczne, gdyż za ich pomocą można narysować niemal każdą możliwą krzywiznę. Figury te nie są jednak odpowiednie do rysowania łuków elips czy kół, jako że żadna krzywa Béziera nie będzie dokładnie wyznaczała okręgu.

¹⁴⁴ Aczkolwiek krzywe Béziera istnieją w przestrzeni co najwyżej trójwymiarowej.

Jeśli sposób, w jaki punkty kontrolne określają kształt krzywej Béziera sprawia ci kłopot, możesz pobawić się w ich rysowanie w programie Paint. Narzędzie *Krzywa*, w które jest on wyposażony, to nic innego jak właśnie krzywa Béziera. Przyjrzyj się także przykładowemu programowi *Bezier* - nie tylko w działaniu, ale i od strony kodu.

Figury zamknięte

Przyszedł czas na rozpoczęcie rysowania figur zamkniętych. Są one zwykle tym, co rozumiemy pod pojęciem 'figura geometryczna'. Najprostszymi figurami są prostokąty i elipsy.

Windows GDI rysuje takie figury z użyciem zarówno pióra, jak i pędzla. Pióro służy mu do zaznaczenia obwodu figury, natomiast pędzel jest używany do wypełnienia wnętrza. Możliwe jest aczkolwiek pozostawienie wnętrza w nienaruszonym stanie - należy po prostu wybrać odpowiedni pędzel:

```
HBRUSH hbrStary = (HBRUSH) SelectObject(hdcKontekst,  
                                         GetStockObject(NULL_BRUSH));
```

W niniejszym paragrafie zajmiemy się więc głównie funkcjami Windows GDI, rysującymi kształty o obramowaniu kreślonym piórem oraz wnętrzu wypełnionym pędzlem. Podzielimy je sobie na funkcje odnoszące się do wielokątów i do elips lub ich fragmentów.

Wielokąty

Z geometrycznego punktu widzenia wielokąt jest to łamana zamknięta, czyli taka, której początek pokrywa się z końcem. Wynika stąd, że w GDI moglibyśmy rysować wielokąty za pomocą funkcji `Polyline[To]()`. Takie figury miałyby wtedy nienaruszone wnętrza, nietknięte żadnym pędzlem.

Jeżeli jednak chcemy zastosować wypełnienie, wtedy odpowiedniejszą funkcją jest `Polygon()`. Omówimy ją za chwilę.

Najpierw bowiem zajmiemy się szczególnym, chyba najważniejszym rodzajem wielokątów - prostokątami.

Prostokąt

Z prostokątami znamy się już dosyć długo, od kiedy poznaliśmy strukturę `RECT` i dowiedzieliśmy się, że przy jej pomocy Windows określa te figury na ekranie. Przypomnę tylko, że pola tej struktury - `left`, `top`, `right` i `bottom` - są współrzędnymi czterech krawędzi prostokąta. Jeśli zgrupujemy je w pary, to otrzymamy też pozycję lewego górnego oraz prawego dolnego wierzchołka prostokąta.

Także w GDI prostokąty są reprezentowane w ten sposób. Weźmy na przykład funkcję `Rectangle()`:

```
BOOL Rectangle(HDC hdc,  
               int nLeftRect,  
               int nTopRect,  
               int nRightRect,  
               int nBottomRect);
```

Jej cztery parametry dokładnie odpowiadają polom struktury `RECT`. Sama funkcja rysuje prostokąt o podanej charakterystyce za pomocą aktualnego pióra, zaś wypełnia go przy pomocy bieżącego pędzla.

`Rectangle()` przyjmuje łącznie cztery parametry, choć mógłaby dwa - wtedy opis prostokąta byłby zapisany w strukturze `RECT`. Nic jednak nie stoi na przeszkodzie, aby

napisać sobie pasujące nam funkcje. Przy okazji także tą, która rysować będzie prostokąt o podanej pozycji i wymiarach:

```
BOOL Rectangle(HDC hdc, RECT rc)
{ return Rectangle(hdc, rc.left, rc.top, rc.right, rc.bottom); }

BOOL Rectangle(HDC hdc, POINT pos, SIZE size)
{ return Rectangle(hdc, pos.x, pos.y,
                  pos.x + size.cx, pos.y + size.cy); }
```

Być może znalazłyby się one w samej bibliotece Windows GDI, gdyby tylko język C, w którym została ona napisana, dopuszczał przeciążanie funkcji.

Stosując `Rectangle()` musimy pamiętać, że jeżeli nasze pióro nie ma stylu `PS_INSIDEFRAME`, a jego linia jest grubsza niż 1 piksel, wtedy część obramowania może „wystawać” poza zakres określony przez parametry funkcji.

Inną funkcją rysującą prostokąty jest `FillRect()`:

```
BOOL FillRect(HDC hdc,
              CONST RECT* lprc,
              HBRUSH hbr);
```

Ona z kolei potrzebuje wskaźnika do struktury `RECT`. Oprócz tego różni się ona tym, że w ogóle **nie bierze pod uwagę** obecnych w kontekście obiektów pióra i pędzla. Ignoruje pióro - gdyż go zwyczajnie nie używa, nie rysuje obramowania figury; pędzel natomiast podajemy jako dodatkowy parametr funkcji. Funkcja wypełnia nim wskazany obszar i na tym kończy się jej rola.

Niemal identyczny prototyp posiada funkcja `FrameRect()`:

```
int FrameRect(HDC hdc,
              CONST RECT* lprc,
              HBRUSH hbr);
```

Funkcja żąda tu uchwytu do pędzla, ale nie stosuje go do wypełniania. Otóż, ona maluje nim nie wnętrze, lecz **obramowanie** prostokąta. Innymi słowy, pędzel działa tu trochę jak pióro. Nie możemy jednak liczyć na jakieś oszałamiające efekty deseniowego obramowania prostokątów, gdyż rysowana krawędź ma grubość tylko 1 jednostki logicznej. Najlepiej więc stosować tutaj wyłącznie pędzle o jednolitym kolorze.

Operację zupełnie innego rodzaju przeprowadza funkcja `InvertRect()`:

```
BOOL InvertRect(HDC hdc,
                CONST RECT* lprc);
```

W zasadzie trudno powiedzieć, aby wykonywała ona jakiegokolwiek rysowanie. Funkcja ta bierze podany jej prostokąt, odczytuje zawarte w nim piksele, a następnie **odwraca ich kolory** i zapisuje z powrotem. Odwrócenie oznacza tu operacji negacji bitowej (NOT, w C++ operator `~`) na liczbowej reprezentacji kanałów RGB tych kolorów. Przykładowo, jeżeli mieliśmy prostokąt, gdzie jakiś fragment był koloru białego, inny żółtego, a jeszcze inny niebieskiego, to po zastosowaniu `InvertRect()` kolory zmieniają się na (odpowiednio): czarny, niebieski i żółty.

`InvertRect()` przypomina więc zrobienie negatywu fotografii.

Chcąc zobaczyć przykłady rysowania prostokątów, wróć do programu `RandomRects`, prezentowanego podczas omawiania pędzli.

Dowolny wielokąt

Do narysowania wielokąta o liczbie boków mniejszej lub większej od 4 możesz posłużyć się funkcją `Polygon()`:

```
BOOL Polygon(HDC hdc,
             CONST POINT* lpPoints,
             int nCount);
```

Funkcja ta wygląda podobnie jak `Polyline()` - też potrzebuje do szczęścia tablicy punktów, którą podajemy w drugim parametrze, jej wielkość zaś w trzecim. W tym przypadku są to jednak wierzchołki wielokąta, które są łączone linią wyrysowaną przez pióro. Nie musimy aczkolwiek podawać dwa razy pierwszego wierzchołka, bo figura zostanie zamknięta automatycznie. Następnie jest ona wypełniana przez aktualny pędzel.

Przy wypełnianiu wielokątów, których boki się przecinają, liczy się także tryb wypełniania, ustawiany funkcją `SetPolyFillMode()`.

Przykładowo, narysowanie pewnego prostokątnego trójkąta oznacza wywołanie:

```
POINT aTrojkat[] = { { 100, 100 }, { 100, 200 }, { 200, 200 } };
Polygon (hdc, aTrojkat, 3);
```

Przy pomocy `Polygon()` można też napisać funkcje rysujące bardziej konkretne rodzaje wielokątów. Oto np. procedura rysowania trójkąta foremnego:

```
BOOL RegularTriangle(HDC hdc, POINT ptPozycja, unsigned uBok)
{
    /* tworzymy tablicę punktów */

    POINT aTrojkat[3];

    // obliczamy wysokość
    unsigned uWysokosc = static_cast<unsigned>(uBok * sqrtf(3) / 2);

    // pierwszy wierzchołek - lewy dolny
    aTrojkat[0].x = ptPozycja.x;
    aTrojkat[0].y = ptPozycja.y + uWysokosc;

    // drugi wierzchołek - prawy dolny
    aTrojkat[1].x = ptPozycja.x + uBok;
    aTrojkat[1].y = aTrojkat[0].y;

    // trzeci wierzchołek - górny
    aTrojkat[2].x = ptPozycja.x + uBok / 2;
    aTrojkat[2].y = ptPozycja.y;

    /* rysujemy trójkąt */
    return Polygon(hdc, aTrojkat, 3);
}
```

`Polygon()` może też służyć do przybliżonego rysowania zamkniętych krzywych wypełnionych, podobnie jak `Polyline()` potrafi przybliżać krzywe otwarte.

Elipsy i koła

Elipsa to drugi ważny rodzaj figury geometrycznej. GDI potrafi rysować go w całości, jak również kreślić i wypełniać wycinki oraz odcinki elips. Przypatrzmy się więc funkcjom, które to potrafią.

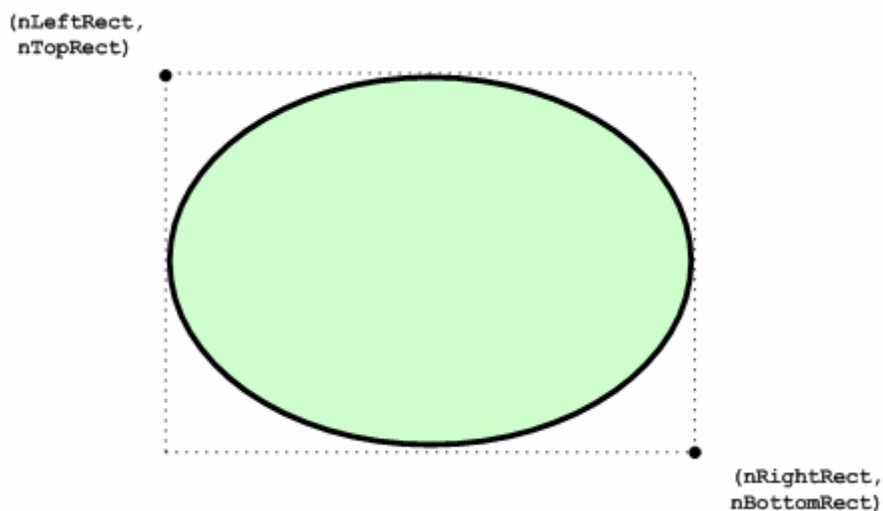
Elipsa

W potocznym rozumieniu elipsa to takie „ściśnięte koło”. Nie jest to wcale złe określenie. Przypomnijmy sobie, że koło może być wpisane w kwadrat. Wobec tego elipsa może być wpisana w prostokąt. Nawet więcej - w jeden i w **dokładnie jeden prostokąt**. Łatwy z tego wynika wniosek: elipsę możemy w wygodny sposób określić, podając charakterystykę najmniejszego prostokąta, który może ją otoczyć.

Tak też czynimy, wywołując funkcję `Ellipse()`:

```
BOOL Ellipse(HDC hdc,
             int nLeftRect,
             int nTopRect,
             int nRightRect,
             int nBottomRect);
```

Ma ona identyczny prototyp, co `Rectangle()`. Tutaj współrzędne prostokąta oznaczają jednak nie sam wielokąt, ale granice, w których jest wrysowywana elipsa:



Rysunek 21. Znaczenie parametrów funkcji `Ellipse()`

Długości dwóch promieni elipsy będą wynosiły $(nRightRect - nLeftRect) / 2$ oraz $(nBottomRect - nTopRect) / 2$. Jeśli chcemy narysować okrąg, to musimy naturalnie zadbać, aaby były one równe. Przekazany do funkcji okrąg musi więc być kwadratem.

Wycinek elipsy

Wycinkiem elipsy nazywamy jej część ograniczoną dwoma promieniami, biegnącymi od środka do krawędzi figury. Wygląda ona trochę jak kawałek tortu - tym smaczniejszy, im bardziej elipsa jest okrągła :)

Do rysowania wycinka elipsy służy w Windows GDI funkcja o nazwie `Pie()`, zatem porównaniu do tortu nie jest wcale takie odległe¹⁴⁵. Oto prototyp tej funkcji:

```
BOOL Pie(HDC hdc,
```

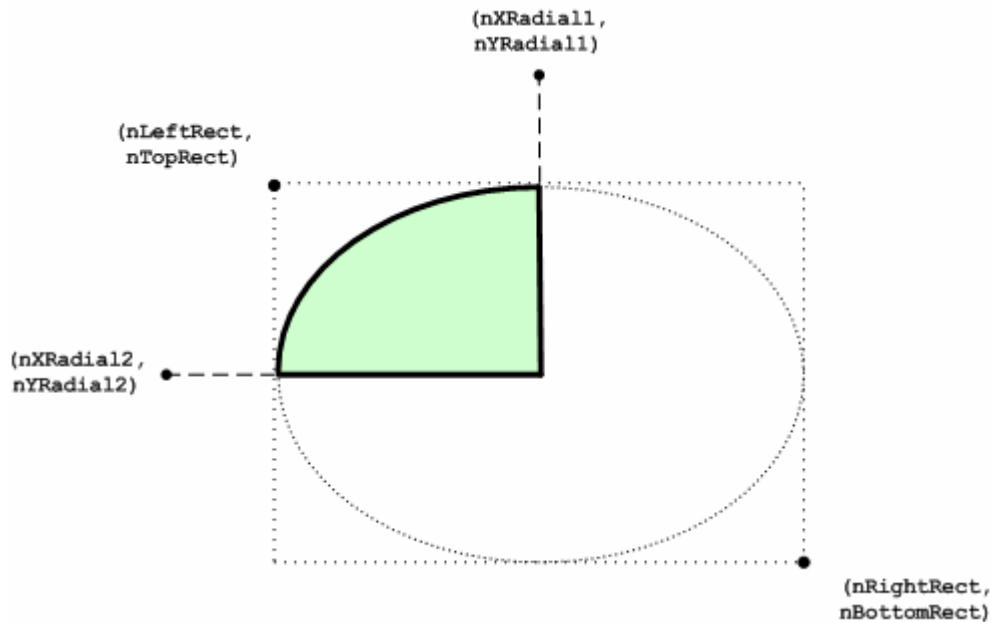
¹⁴⁵ *pie* po angielsku oznacza 'szarlotkę'.

```

int nLeftRect,
int nTopRect,
int nRightRect,
int nBottomRect,
int nXRadial1,
int nYRadial1,
int nXRadial2,
int nYRadial2);

```

Wygląda on podobnie do nagłówka `Arc[To]()` i, jak się pewnie domyślasz, nie jest to przypadek. `Pie()` określa wycinek elipsy w bardzo podobny sposób, jak wspomiane funkcje definiują łuk. Spójrz zresztą na poniższy rysunek:



Rysunek 22. Znaczenie parametrów funkcji `Pie()`

Kierunek zakreślania wycinka elipsy jest tu, tak samo jest w `Arc[To]()`, przeciwny do ruchu wskazówek. Punkty wyznaczające fragment elipsy również nie muszą na niej leżeć.

Odcinek elipsy

Ostatnią figurą z gatunku elips i okolic jest odcinek elipsy. Jest to figura geometryczna, będącą częścią wspólną elipsy i połpłaszczyzny. Mówiąc jaśniej, jeżeli przetniemy naszą elipsę linią prostą, to dwie figury, jakie przy okazji powstaną, będą niczym innym jak właśnie odcinkami elipsy.

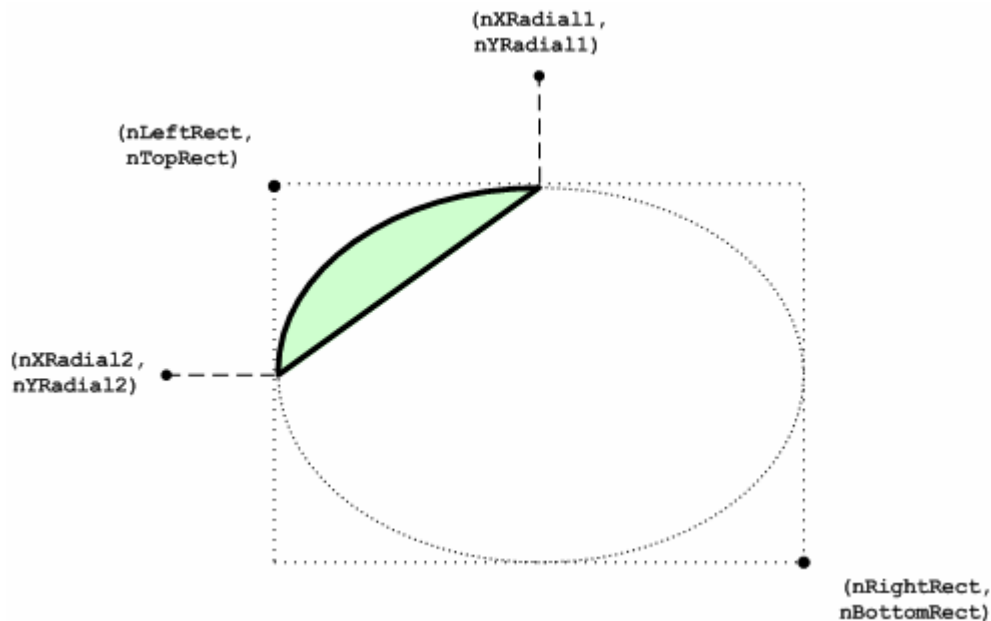
Za rysowanie tego rodzaju kształtów odpowiada w Windows GDI funkcja o nazwie `Chord()` ('ciąciwa'). Popatrzmy na jej, znajomy już pewnie, prototyp:

```

BOOL Chord(HDC hdc,
int nLeftRect,
int nTopRect,
int nRightRect,
int nBottomRect,
int nXRadial1,
int nYRadial1,
int nXRadial2,
int nYRadial2);

```

Znowu mamy dziewięć znanych argumentów, określających całą elipsę oraz wycinek jej okręgu. Ich znaczenie tradycyjnie przestudiujemy na rysunku:



Rysunek 23. Znaczenie parametrów funkcji `Chord()`

Zasady znane z `Arc[To]()` stosują się także i tutaj.

Zaokrąglony prostokąt

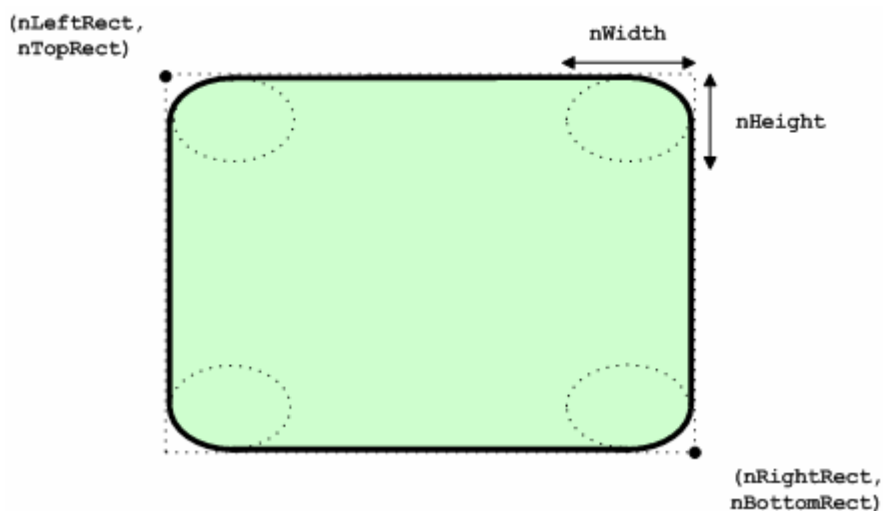
Patrząc na tytuł tego akapitu możesz być trochę zdezorientowany. Jak prostokąt, który nawet z nazwy ma proste kąty, może być zaokrąglony?... Cóż, to pewna nieścisłość, bo faktycznie chodzi o **prostokąt z zaokrąglonymi rogami**, ale powszechnie przyjęło się, by o tej figurze mówić właśnie 'zaokrąglony prostokąt'. Zapewne wynika to z tłumaczenia angielskiego terminu *rounded rectangle*.

W GDI tego typu figurę można narysować, stosując przeznaczoną do tego funkcję `RoundRect()`:

```
BOOL Rectangle(HDC hdc,
               int nLeftRect,
               int nTopRect,
               int nRightRect,
               int nBottomRect,
               int nWidth,
               int nHeight);
```

Jej pierwszych pięć parametrów z pewnością wygląda znajomo, bo pochodzi z funkcji `Rectangle()`. Ostatnie dwa muszą więc mieć wpływ na nową cechę prostokąta, czyli zaokrąglenie jego rogów - i rzeczywiście tak jest. Parametry `nWidth` i `nHeight` określają bowiem szerokość i wysokość elipsy, która wyznacza krągłość rogów. Innymi słowy, jest to pozioma i pionowa średnica tejże elipsy lub jeszcze inaczej - wymiary prostokąta okalającego tę elipsę.

No dobrze, ale czym właściwie jest ta elipsa?... Odpowiedzią na to pytanie będzie kolejny rysunek, obrazujący znaczenie wszystkich parametrów funkcji `RoundRect()`. Oto i ten szkic:



Rysunek 24. Znaczenie parametrów funkcji `RoundRect()`

Na rysunku widać, że zaokrąglenie rogów jest precyzowane poprzez małą elipsę. Parametry `nWidth` i `nHeight` są jej wymiarami - w praktyce równymi, gdyż symetryczne naroża wyglądają zwykle najlepiej.

Bitmapy

Duża część pracy z grafiką związana jest z obróbką gotowych bitmap. Używając wykonanych wcześniej obrazków nie musimy dbać o ręczne generowanie każdego szczegółu wyglądu obrazka. Wiele elementów graficznych, pochodzących zwłaszcza z realnego świata, nie da się efektywnie zamodelować przy pomocy obiektów wektorowych. Bitmapy stają się wówczas niezbędne.

Windows GDI posiada kilka sposobów manipulacji bitmapami jako całością. Umożliwia między innymi wczytywanie ich z plików, kopiowanie do wybranego kontekstu urządzenia, skalowanie czy wyświetlanie z przezroczystością. Jest to możliwe przy pomocy obiektów bitmap, którymi zarządzamy poprzez uchwyty typu `HBITMAP`.

Zarządzanie bitmapą

Z bitmapami obchodzimy się podobnie jak z innymi obiektami GDI: tworzymy je, podpinamy do kontekstu urządzenia, wykonujemy pożądane operacje, a wreszcie zwalniamy, odzyskując zasoby. Popatrzymy teraz na każdy z tych etapów.

Tworzenie obiektu bitmapy

Obiekt bitmapy możemy stworzyć, opierając się na pliku, na kontekście urządzenia lub też samodzielnie podać jej wszystkie parametry. Poznamy tutaj każdy z tych trzech sposobów.

Odczytanie z pliku

Znamy już funkcję, która potrafi wczytywać obrazki z plików. Jest to `LoadImage()`:

```
HANDLE LoadImage(HINSTANCE hInstance,
                  LPCTSTR lpszName,
                  UINT uType,
                  int cxDesired,
                  int cyDesired,
                  UINT fuLoad);
```

Zgodnie z obietnicą, omówimy ją sobie dokładnie w tym momencie. Zajmijmy się więc jej parametrami - pomocą będzie tu tradycyjna tabelka:

<i>typ</i>	<i>parametr</i>	<i>opis</i>
HINSTANCE	hInstance	Jeżeli chcemy wczytać obrazek z zasobów zawartych w pliku EXE, podajemy tutaj uchwyt instancji programu. Jeżeli natomiast zależy nam na załadowaniu obrazku z pliku na dysku (względnie skorzystanie z obrazu systemowego), wpisujemy tu <code>NULL</code> . My będziemy na razie tak właśnie robić, jako że jeszcze nie umiemy posługiwać się zasobami Windows.
LPCTSTR	lpszName	Tutaj podajemy jedną z trzech informacji: <ul style="list-style-type: none"> ➤ identyfikator zasobu, jeżeli w <code>hInstance</code> podaliśmy wartość inną niż <code>NULL</code> ➤ stałą określającą obrazek systemowy, jeżeli chcemy takowy wczytać ➤ nazwę pliku, skąd chcemy wczytać obrazek Nas będzie naturalnie interesować ostatnia opcja.
UINT	uType	Podajemy tu typ wczytywanego obrazka. Kiedy chcieliśmy pobrać ikonę lub kursor, był to <code>IMAGE_ICON</code> lub <code>IMAGE_CURSOR</code> . Teraz chcemy wczytywać bitmapy, więc wpisujemy tu <code>IMAGE_BITMAP</code> .
<code>int</code>	cxDesired cyDesired	Gdybyśmy zajmowali się ładowaniem ikony lub kursora, wpisalibyśmy tu jego pożądane rozmiary . Ponieważ jednak chodzi nam o bitmapę, możemy zignorować te parametry i wpisać w nich zera - nie są one brane pod uwagę, jeżeli <code>uType</code> ma wartość <code>IMAGE_BITMAP</code> .
UINT	fuLoad	To zaś są flagi wczytywania obrazka, czyli dodatkowe opcje.

Tabela 65. Parametry funkcji `LoadImage()`

Jeżeli chodzi o ostatni parametr, to dozwolone są między innymi takie oto flagi:

<i>flaga</i>	<i>opis</i>
LR_CREATEDIBSECTION	Zachowuje oryginalne kolory bitmapy i nie dostosowuje ich do głębi kolorów ekranu. Tę flagę trzeba podać, jeżeli chcemy wczytać bitmapę z zapisanym kanałem alfa (z 32-bitowym formatem koloru).
LR_MONOCHROME	Redukuje liczbę kolorów bitmapy do monochromatyczności, czyli czerni i bieli.
LR_LOADFROMFILE	Flaga ta określa, iż chcemy wczytać bitmapę z pliku.

Tabela 66. Flagi bitowe funkcji `LoadImage()`

Nas najbardziej interesuje `LR_LOADFROMFILE`. Przy jej pomocy możemy bowiem bez problemu odczytać obrazek rastrowy zapisany w pliku, a następnie używać go w operacjach graficznych. `LoadImage()` obsługuje formaty BMP oraz DIB.

Przykładowe użycie tej funkcji może być następujące:

```
HBITMAP hbmpBitmapa = (HBITMAP) LoadImage(NULL, "obrazek.bmp", 0, 0,
                                             LR_LOADFROMFILE);
```

Rzutowanie na `HBITMAP` jest tu konieczne, bo `LoadImage()` ogólny uchwyt typu `HANDLE`. Jest on dla uchwytów tym, czy `void*` dla wskaźników. Musimy zatem zastosować konwersję, aby przypisać wartość do zmiennej `hbmpBitmapa`.

Nazwa pliku, jaką podajemy do funkcji `LoadImage()`, jest relatywna do katalogu programu. Najlepiej więc wpisywać pełną ścieżkę do pliku graficznego.

W taki oto sposób stajemy się posiadaczami uchwytu do obiektu bitmapy wczytanej z pliku na dysku.

Dopasowanie do kontekstu

Czasem warto jest zacząć od zera. Jeżeli nie chcemy opierać się na istniejącym pliku graficznym przy tworzeniu obiektu bitmapy, to oczywiście nie musimy tego robić. Powinniśmy jednak wiedzieć, z jakim kontekstem urządzenia ma współpracować nasza, z początku pusta, bitmapa. Dzięki temu będziemy mogli ją w przyszłości podpiąć pod ten lub kompatybilny z nim kontekst lub chociażby mieć pewność zgodności kolorów przy wyświetlaniu bitmapy w tym kontekście.

Kiedy już wiemy, gdzie będziemy pracować, pozostaje nam wywołać funkcję `CreateCompatibleBitmap()`:

```
HBITMAP CreateCompatibleBitmap(HDC hdc,
                               int nWidth,
                               int nHeight);
```

Podajemy jej uchwyt kontekstu urządzenia, który ma być kompatybilny z tworzoną bitmapą. Musimy tutaj koniecznie pamiętać, aby był to kontekst **inny niż pamięciowy**. Najlepiej niech to będzie taki kontekst, w którym chcielibyśmy tak stworzoną bitmapę wyświetlić - a więc związany np. z ekranem lub obszarem klienta okna. Oprócz tego dostarczamy też wymiary nowej bitmapy.

Tak utworzona bitmapa jest pusta, więc wydaje się to mało przydatne rozwiązanie. Jednak, jak się niedługo przekonamy, możliwe jest rysowanie po takiej bitmapie przy użyciu pamięciowego kontekstu urządzenia. Zatem nie jest to wcale takie nieużyteczne, jakby się mogło wydawać.

Dowolny format

Dla porządku podam jeszcze prototyp funkcji `CreateBitmap()`:

```
HBITMAP CreateBitmap(int nWidth,
                     int nHeight,
                     UINT cPlanes,
                     UINT cBitsPerPel,
                     CONST VOID* lpvBits);
```

Funkcja ta służy do stworzenia bitmapy od podstaw, tj. z podaniem jej wszystkich parametrów. Oto i one:

<i>typ</i>	<i>parametry</i>	<i>opis</i>
<code>int</code>	<code>nWidth</code> <code>nHeight</code>	Wpisujemy tu wymiary bitmapy w pikselach.
<code>UINT</code>	<code>cPlanes</code>	Ten parametr to liczba tzw. płatów koloru (ang. <i>color planes</i>). Wartość ta wywodzi z zamierzonych czasów kart graficznych w rodzaju EGA i została zachowana wyłącznie celem kompatybilności wstecz. Obecnie wpisujemy tu zawsze 1 .
	<code>cBitsPerPel</code>	Głębina koloru , czyli ilość bitów przypadających na jeden piksel. Zwykle jest to 8 , 16 lub 24 . Można też wpisać 32 - wówczas będziemy mieli bitmapę z miejscem na kanał alfa.
<code>CONST VOID*</code>	<code>lpvBits</code>	Można tu podać zawartość bitmapy w postaci ciągu bitów. Ciąg ten powinien zawierać reprezentacje kolejnych

<i>typ</i>	<i>parametry</i>	<i>opis</i>
		pikseli bitmapy, poczynając od jej lewego górnego rogu i posuwając się rzędami. Jeżeli nie chcemy inicjować nowej bitmapy żadną zawartością, wtedy w tym parametrze należy wpisać wartość <code>NULL</code> .

Tabela 67. Parametry funkcji `CreateBitmap()`

`CreateBitmap()` używamy zwykle wtedy, gdy chcemy utworzyć obiekt bitmapy mając już jej pamięciową reprezentację w postaci tablicy. Taka tablica może np. poprzez własny algorytm wczytujący (ang. *loader*) obrazek w jakimś szczególnym formacie pliku. Inne przypadki korzystania z tej funkcji co raczej rzadkie.

Pamięciowy kontekst urządzenia

Wczytanie bitmapy to dopiero pierwszy krok do jej wykorzystania. Tak naprawdę aby zrobić z nią cokolwiek konkretnego, musimy ją związać z kontekstem urządzenia.

Ale co to znaczy - związać bitmapę z kontekstem urządzenia? Przecież obrazek to nie jest pióro ani pędzel, w jaki sposób miałby on pomagać w rysowaniu, którym zajmuje się kontekst?...

A jednak pomaga on, a właściwie to je nawet umożliwia. Żeby to zrozumieć musimy sobie uświadomić, że to, na czym rysujemy poprzez kontekst urządzenia, to tak naprawdę nic innego jak właśnie **bitmapa**. Ekran jest jedną wielką bitmapą, podobnie jak wszystkie jego części (np. okno), od których możemy uzyskać konteksty urządzeń. Bitmapę związaną z kontekstem urządzenia nazywamy **plótnem** (ang. *canvas*).

Powiązanie bitmapy z kontekstem urządzenia jest więc zamianą płótna. To trochę tak, jakbyśmy zdjęli jeden obraz ze sztalugi malarskiej i położyli inny, który w szczególności może być pustym płótnem. Po zamianie wszystkie następne czynności rysunkowe będą skutkowały malowaniem po nowej bitmapie kontekstu urządzenia.

Jednak nie wszystkim kontekstom urządzenia możemy swobodnie zabierać płótna.

Właściwie to większości z nich nie możemy tego zrobić, ponieważ bitmapy, do których się one odnoszą, należą do systemu operacyjnego, a ten zdecydował, iż będą one na stałe przybite do swoich sztalug - kontekstów. Dzieje się tak, bo konteksty fizycznie odpowiadają np. fragmentom ekranu monitora, a tej przynależności nie możemy zmienić - przecież nie odbierzemy pecetowi monitora, prawda? :)

Ale jak w takim razie uzyskać niezależny kontekst urządzenia, który moglibyśmy związać z naszą bitmapą?... Otóż musimy go sobie stworzyć i wiemy już, jak to zrobić. Środkiem do osiągnięcia celu jest bowiem **pamięciowy kontekst urządzenia** (ang. *memory device context*).

Aby móc wykonywać na bitmapie operacje graficzne, powinniśmy powiązać ją z pamięciowym kontekstem urządzenia.

Powinniśmy tworzyć go dla każdej bitmapy, którą zamierzamy kopiować, wyświetlać na ekranie czy też po której chcemy rysować. Jest to związane ze specyfiką niektórych operacji w Windows GDI, które działają tylko w odniesieniu do kontekstów urządzeń, a nie bitmap jako takich. Poznamy je całkiem niedługo.

Najpierw zobaczymy, jak poprawnie stworzyć pamięciowy kontekst urządzenia i podpiąć pod niego bitmapę.

Utworzenie kontekstu

Pamięciowego kontekstu urządzenia nie tworzy się od podstaw, lecz tylko przy pomocy innego, już istniejącego kontekstu. Nowy kontekst będzie z nim kompatybilny, tzn. możliwe będzie przeprowadzanie operacji graficznych między nim, a starym kontekstem.

Do utworzenia pamięciowego kontekstu urządzenia posługujemy się funkcją `CreateCompatibleDC()`:

```
HDC CreateCompatibleDC(HDC hdc);
```

Podajemy jej oczywiście uchwyt do kontekstu urządzenia, z którym nasz nowy kontekst ma być kompatybilny. Może to być `hdc` pozyskany na przykład od okna czy też całego ekranu - w tym drugim przypadku możliwe jest podanie `NULL` jako parametru.

Utworzenie pamięciowego kontekstu wygląda więc mniej więcej tak:

```
HDC hdcPamiec = CreateCompatibleDC(hdcKontekst);
```

Istniejący uchwyt `hdcKontekst` musi się odnosić do kontekstu urządzenia, które potrafi wykonywać działania na grafice rastrowej - czyli np. do monitora.

Powiązanie bitmapy z kontekstem

Ostatnim etapem sztuki jest powiązanie naszej bitmapy (załóżmy na razie, że wczytanej z pliku) z nowostworzonym, pamięciowym kontekstem urządzenia. Jest to bardzo proste, należy jedynie wywołać funkcję `SelectObject()` w znany nam doskonale sposób:

```
HBITMAP hbmpStaraBitmapa = (HBITMAP) SelectObject (hdcPamiec,  
                                                    hbmpBitmapa);
```

W zmiennej `hbmpStaraBitmapa` zapisujemy uchwyt do starej bitmapy - w przypadku pamięciowego kontekstu jest to zawsze **monochromatyczny obrazek o wymiarach 1×1** piksela. Zapisujemy jego uchwyt, ponieważ tak samo jak pióro czy pędzel bitmapa nie może być pozostawiona sama sobie. Wtedy bowiem nastąpiłby wyciek pamięci. Oczywiście możliwe jest zastosowanie innej metody postępowania z nieużywanymi obiektami, takiej jak natychmiastowe usunięcie bitmapy (opakowanie wywołania `SelectObject()` w `DeleteObject()`) lub skorzystanie z zapisu stanów kontekstu przez `SaveDC()`.

W sumie, przygotowanie bitmapy do pracy z GDI wygląda następująco:

```
// 1. wczytanie bitmapy  
HBITMAP hbmpBitmapa = (HBITMAP) LoadImage(NULL, "bitmapa.bmp", 0, 0,  
                                           LR_LOADFROMFILE);  
  
// 2. stworzenie pamięciowego kontekstu urządzenia  
// (tutaj będzie on kompatybilny z ekranem)  
HDC hdcPamiec = CreateCompatibleDC(NULL);  
  
// 3. wybranie wczytanej bitmapy w kontekście pamięciowym  
HBITMAP hbmpStara = (HBITMAP) SelectObject(hdcPamiec, hbmpBitmapa);
```

Po wykonaniu tych czynności możemy stosować takie funkcje jak `BitBlt()` czy `StretchBlt()`, aby np. wyświetlić zawartość bitmapy `hbmpBitmapa` wewnątrz obszaru klienta okna. O tych funkcjach powiemy sobie wszystko w następnym paragrafie.

Muszę jeszcze wspomnieć o sytuacji, gdy naszej bitmapy nie chcemy wczytać z pliku. Możemy mianowicie stworzyć sobie pustą bitmapę, związać ją z pamięciowym

kontekstem urządzenia, wykonywać na niej wybrane operacje graficzne i wyświetlić dopiero wtedy, gdy będzie już gotowa.

W takim wypadku musimy sobie stworzyć nowy obiekt bitmapy przy pomocy funkcji `CreateCompatibleBitmap()`. Do funkcji tej podajemy m.in. uchwyt kontekstu urządzenia, z którym bitmapa będzie kompatybilna. Z tego kontekstu obrazek pobierze ustawienia głębi kolorów, czyli ilość bitów przypadającą na jeden piksel.

Poprawne zastosowanie wspomnianej funkcji do stworzenia pustej bitmapy oraz powiązanie jej z pamięciowym kontekstem urządzenia wygląda tak:

```
// (zakładamy, że w hdcKontekst mamy pewien kontekst, np. od okna)

// tworzymy kontekst pamięciowy
HDC hdcPamiec = CreateCompatibleDC(hdcKontekst);

// tworzymy pustą bitmapę dla tego kontekstu, o wymiarach 100x100
HBITMAP hbmpBitmapa = CreateCompatibleBitmap(hdcKontekst, 100, 100);

// wiążemy nową bitmapę z kontekstem pamięciowym, zachowując
// jednocześnie uchwyt do starej (czarno-biała, 1x1 piksel)
HBITMAP hbmpStara = (HBITMAP) SelectObject(hdcPamiec, hbmpBitmapa);
```

Konieczne zwróćmy uwagę na linijkę tworzącą bitmapę:

```
HBITMAP hbmpBitmapa = CreateCompatibleBitmap(hdcKontekst, 100, 100);
```

Widać, że do funkcji `CreateCompatibleBitmap()` **nie podajemy uchwytu do kontekstu pamięciowego**. Zamiast tego przekazujemy jej **oryginalny kontekst** `hdcKontekst`, a więc ten, który posłużył nam do stworzenia pamięciowego `hdcPamiec`. Dlaczego właśnie tak? Przypomnij sobie, co przed chwilą mówiłem na temat początkowej bitmapy w kontekście pamięciowym. Jest to czarno-biały obrazek wielkości 1 piksela. Rozmiar nie jest tu akurat ważny, ale głębia kolorów - jak najbardziej. W pierwotnej bitmapie pamięciowego kontekstu obejmuje ona tylko dwa kolory, jest 1-bitowa. Jeżeli więc posłużymy się tym kontekstem do utworzenia kompatybilnej bitmapy, jej głębia kolorów będzie z tym zgodna - *ergo*: nowa bitmapa również będzie monochromatyczna. Nie wydaje mi się, aby o to właśnie nam chodziło. Chcielibyśmy raczej, by nasz obrazek mógł zawierać tyle kolorów, ile potrafi wyświetlić ekran monitora. Dlatego też do `CreateCompatibleBitmap()` powinniśmy podać uchwyt kontekstu odnoszącego się do monitora właśnie, nie zaś do kontekstu pamięciowego. Zapamiętaj zatem, że:

Tworząc **pustą bitmapę dla pamięciowego kontekstu urządzenia**, do funkcji `CreateCompatibleBitmap()` musisz **przekazać oryginalny hdc** - ten, na podstawie którego stworzyłeś kontekst pamięciowy. W przeciwnym wypadku powstała bitmapa będzie monochromatyczna.

Zwalnianie bitmapy

Bitmapą należy się odpowiednio zająć, gdy już nie jest nam potrzebna. Kolejność i charakter czynności następujących w tym procesie jest w zasadzie odwrotna do tych, jakie podejmujemy podczas przygotowywania bitmapy do pracy. Musimy więc najpierw pozbyć się pamięciowego kontekstu urządzenia (jeżeli takowy stwarzaliśmy, zwykle tak), a następnie usunąć też sam obiekt bitmapy.

Usuwanie kontekstu pamięciowego

Przez usunięciem kontekstu pamięciowego postępujemy podobnie, jak przez zwalnianie każdego innego kontekstu. Przywracamy więc jego obiekty do stanu początkowego - w

tym przypadku jedynym takim obiektem jest monochromatyczne płótno 1×1. Wybieramy je w kontekście pamięciowym:

```
SelectObject (hdcPamiec, hbmStara);
```

Naturalnie, jeżeli nie zachowaliśmy uchwytu do starej bitmapy, lecz usunęliśmy ją od razu, nie będziemy mieli co przywracać, zatem ten etap pominiemy. Możemy od razu przejść do usunięcia samego kontekstu.

Zwolnienie kontekstu pamięciowego jest proste i oznacza tylko wywołanie funkcji `DeleteDC()`:

```
DeleteDC (hdcPamiec);
```

Razem z kontekstem zostaje też usunięte jego płótno, czyli najczęściej ta mała monochromatyczna bitmapa. Możliwe jest aczkolwiek, że ta bitmapa została usunięta już wcześniej, przy wybieraniu dla kontekstu nowego płótna. Wtedy razem z usunięciem pamięciowego `hdc` ginie też jego bitmapa. W takiej sytuacji nie jest konieczne jej oddzielne zwalnianie, opisane w następnym punkcie.

Usuwanie obiektu bitmapy

Na sam koniec pozbywamy się właściwego obiektu bitmapy. Wywołujemy `DeleteObject()`, usuwając go z pamięci:

```
DeleteObject (hbmBitmapa);
```

Nie jest to konieczne, jeżeli nasz bitmapa zginęła razem z kontekstem pamięciowym. Mówiłem jednak już kilka razy, że dla przejrzystości kodu lepiej jest, aby obiekty tworzone przez nas były przez nas usuwane, a te pochodzące od GDI - zwalniane przez samą bibliotekę GDI.

Posługiwanie się bitmapą

Między stworzeniem a zwolnieniem obiektu wypadłoby wykonać na nim jakieś sensowne czynności. Tym właśnie zajmiemy się w niniejszym paragrafie: zobaczymy, cóż takiego możemy zrobić z posiadaną bitmapą.

Wyświetlanie bitmapy

Chyba najlogiczniejszą czynnością, którą możemy wykonać przy użyciu bitmapy, jest jej wyświetlenie. Oznacza to prezentację zawartości obrazka w wybranym kontekście urządzenia, związanym z fizycznym urządzeniem - zwykle monitorem.

Na tej, w gruncie rzeczy prostej, czynności opiera się mnóstwo aplikacji, z grami na czele. Pokazywanie dwuwymiarowych obrazków (tzw. *sprite'ów*) jest w nich bowiem podstawowym sposobem tworzenia oprawy graficznej.

Zobaczmy więc, jak realizować to ważne zadanie w Windows GDI. Poznamy zaraz trzy sposoby (czy może raczej tryby) prezentacji bitmapy w kontekście urządzenia rastrowego.

Dosłowne kopiowanie

Wyświetlenie bitmapy w innym kontekście urządzenia niż pamięciowy wymaga pewnej formy przekopiowania pikseli. Mówimy, że należy zastosować **transfer bloku bitów** (ang. *bit-block transfer*), co oznaczamy angielskim skrótem *bitblt* (czytaj [*bit blit*])

Taką też nazwę ma funkcja Windows GDI, która wykonuje transfer - `BitBlt()`. Oto jej prototyp:

```

BOOL BitBlt(HDC hdcDest,
            int  nXDest,
            int  nYDest,
            int  nWidth,
            int  nHeight,
            HDC  hdcSrc,
            int  nXSrc,
            int  nYSrc,
            DWORD dwRop);

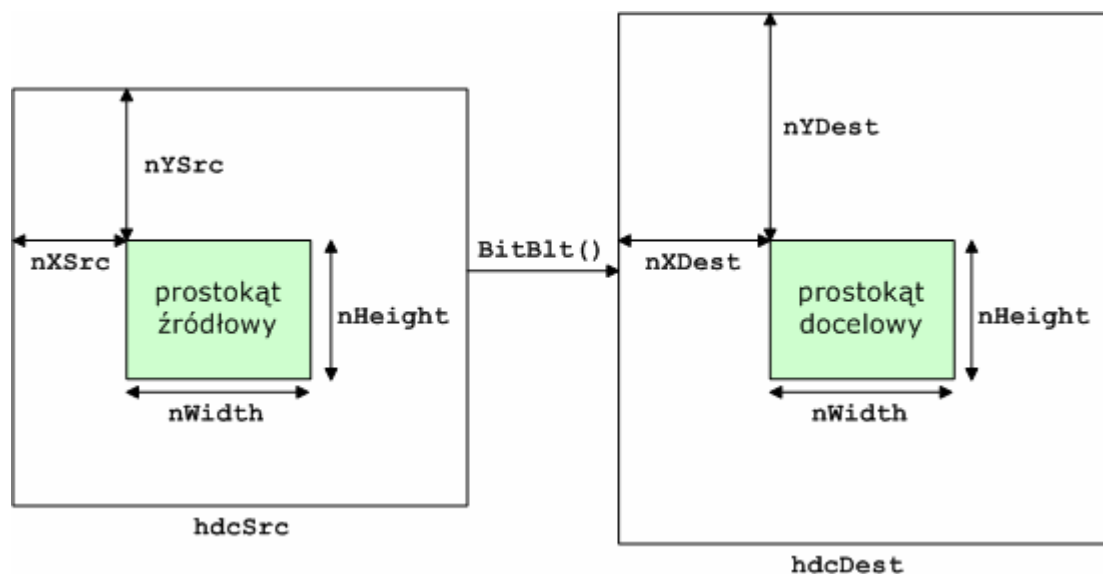
```

Wydaje się może, iż to skomplikowana funkcja, ale w rzeczywistości wcale tak nie jest. Przekazujemy jej raczej niezbędne dane - przede wszystkim uchwyt do dwóch kontekstów urządzenia. Pierwszy z nich, `hdcDest`, określa cel transferu bitów; jest to ten kontekst, w którym pojawi się wyświetlana przez nas bitmapa. Z kolei zatem drugi uchwyt, `hdcSrc`, musi być źródłem kopiowanych pikseli.

Następne parametry - `nWidth` i `nHeight` - są rozmiarami kopiowanego prostokąta. Mówią one po prostu, jak duży jest kopiowany fragment płótna. Jego wielkość jest w `BitBlt()` identyczna zarówno dla kontekstu źródłowego i docelowego.

Ostatnie dwie pary parametrów są współrzędnymi lewego górnego wierzchołka kopiowanego prostokąta. `nXDest` i `nYDest` są docelowymi koordynatami w kontekście `hdcDest`, zaś `nXSrc` i `nYSrc` to współrzędne źródłowego kawałka bitmapy z kontekstu `hdcSrc`.

Rolę każdego z tych parametrów najłatwiej prześledzić na rysunku:



Rysunek 25. Kopiowanie (fragmentu) bitmapy poprzez funkcję `BitBlt()`

Wszystko jasne? To świetnie, bo teraz będzie najcięższy orzech do zgryzienia :) Ale spokojnie, nie będzie aż tak źle. Parametr `dwRop`, bo o nim mowa, nie jest wcale taki trudny do zrozumienia.

Określa on operację rastrową przeprowadzaną podczas łączenia prostokąta źródłowego z docelowym. Z takimi operacjami spoktaliśmy się już przy okazji piór i funkcji `SetROP2()`. Tutaj także mamy do wyboru kilkanaście znaczników, których działanie przedstawia poniższa tabela. Użyto w nich trzech oznaczeń dla argumentów operacji (dlatego nazywamy ją **ternarną**):

- `clSrc` - kolor ze źródłowego kontekstu urządzenia
- `clDest` - kolor z docelowego kontekstu urządzenia

- `clDestBrush` - kolor pędzla docelowego kontekstu urządzenia

<i>flaga operacji</i>	<i>kolor wynikowy</i>
BLACKNESS	czarny
DSTINVERT	$\sim clDest$
MERGECOPY	$clSrc \& clDestBrush$
MERGEPAINT	$\sim clSrc \mid clDest$
NOTSRCCOPY	$\sim clSrc$
NOTSRCERASE	$\sim (clSrc \mid clDest)$
PATCOPY	$clDestBrush$
PATINVERT	$clDestBrush \wedge clDest$
PATPAINT	$(clDestBrush \mid \sim clSrc) \mid clDest$
SRCAND	$clSrc \& clDest$
SRCCOPY	$clSrc$
SRCERASE	$clSrc \& \sim clDest$
SRCINVERT	$clSrc \wedge clDest$
SRCPAINT	$clSrc \mid clDest$
WHITENESS	biały

Tabela 68. Stałe ternarych operacji rastrowych w Windows GDI

Zdecydowanie najczęściej używa się `SRCCOPY`, jako że zazwyczaj chodzi nam o dosłowne przekopiowanie bitmapy z `hdcSrc` do `hdcDest`. Inne znaczniki mogą być przydatne np. wtedy, gdy chcemy wyświetlić bitmapę z nieregularnym kształtem, którego tło ma być przezroczyste.

Na koniec omawiania tej funkcji zobaczmy konkretny przykład jej wykorzystania - czyli wyświetlenie bitmapy wczytanej z pliku w kontekście urządzenia:

```
// zakładamy, że posiadamy kontekst hdcKontekst, np. od okna

// wczytujemy bitmapę
HBITMAP hbmpBitmapa = (HBITMAP) LoadImage(NULL, "bitmapa.bmp", 0, 0,
                                           LR_LOADFROMFILE);

// tworzymy dla niej kontekst pamięciowy i wiążemy ją z nim
HDC hdcPamiec = CreateCompatibleDC(hdcKontekst)
HBITMAP hbmpStara = (HBITMAP) SelectObject(hdcPamiec, hbmpBitmapa);

// pobieramy wymiary bitmapy (potrzebne do jej skopiowania);
// będą one zawarte w polach bmWidth i bmHeight poniższej struktury
BITMAP Bitmapa;
GetObject (hbmpBitmapa, sizeof(BITMAP), &Bitmapa);

// dokonujemy transferu pikseli, czyli wyświetlamy bitmapę
// w punkcie (nX, nY) kontekstu hdcKontekst
BitBlt (hdcKontekst, nX, nY, Bitmapa.bmWidth, Bitmapa.bmHeight,
        hdcPamiec, 0, 0, SRCCOPY);

// zwalniamy kontekst pamięciowy, przywracając mu wpierw starą bitmapę
SelectObject (hdcPamiec, hbmpStara);
DeleteDC (hdcPamiec);

// zwalniamy obiekt bitmapy
DeleteObject (hbmpBitmapa);
```

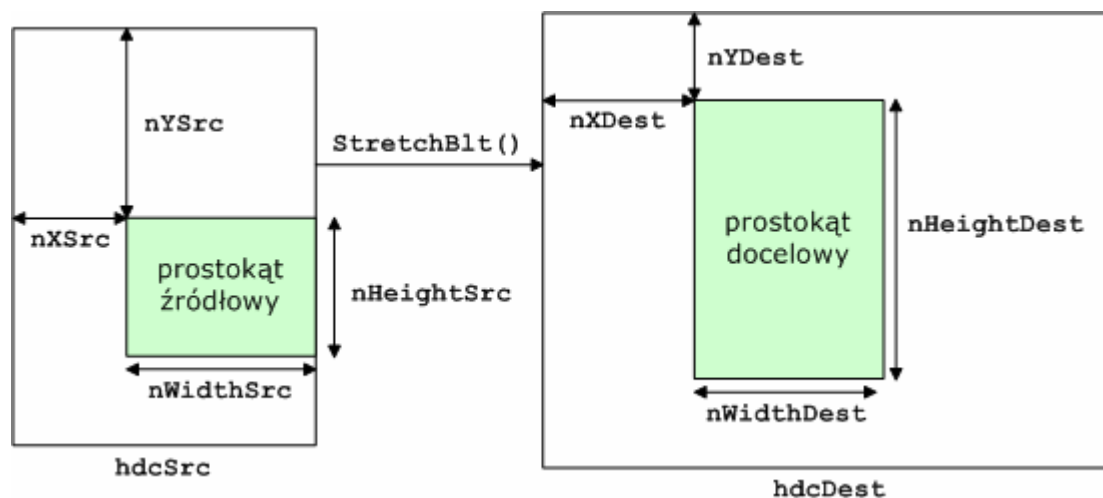
Przy okazji możesz tu zobaczyć, w jaki sposób pobiera się wymiary bitmapy o znanym uchwycie.

Rozciąganie obrazka

`BitBlt()` gwarantuje, że prostokąt wzięty z kontekstu źródłowego będzie w niezmienionej postaci zmiksowany¹⁴⁶ z tej samej wielkości prostokątem w kontekście docelowym. Windows pozwala też na transfer fragmentu bitmapy z jego jednoczesnym skalowaniem - dokonuje tego funkcja `StretchBlt()` (czytaj [*strecz blit*]):

```
BOOL StretchBlt(HDC hdcDest,
               int nXOriginDest,
               int nYOriginDest,
               int nWidthDest,
               int nHeightDest,
               HDC hdcSrc,
               int nXOriginSrc,
               int nYOriginSrc,
               int nWidthSrc,
               int nHeightSrc,
               DWORD dwRop);
```

Jej prototyp jest podobny do `BitBlt()`, ale posiada on dwie pary parametrów określających wymiary kopiowanego prostokąta. `nWidthSrc` i `nHeightSrc` określają więc jego rozmiar w kontekście źródłowym (`hdcSrc`), zaś `nWidthDest` i `nHeightDest` - w kontekście docelowym (`hdcDest`). Pozostałe współrzędne mają takie samo znaczenie jak w `BitBlt()`, tzn. określają pozycję kopiowanego i docelowego prostokąta. W tym przypadku dobry rysunek może być nawet bardziej pomocny niż wcześniej. Popatrz zatem na następujący szkic:



Rysunek 26. Kopiowanie (fragmentu) bitmapy poprzez funkcję `StretchBlt()`

Widzimy na nim, że bitmapa pobrana z kontekstu źródłowego ulega przeskalowaniu - zmieniają się jej rozmiary. Nie byłoby w tym nic złego, gdyby nie to, że mamy przecież do czynienia z obrazkami rastrowymi. Takie rysunku „nie wiedzą”, jak się zachować w sytuacji, gdy zmieniana jest ich wielkość: mają bowiem zapisany określony zestaw pikseli, który jest właściwie interpretowany tylko w swej oryginalnej rozdzielczości. Kiedy chcemy zmienić rozmiary obrazka rastrowego, mogą pojawić się problemy.

Dlatego z funkcji `StretchBlt()` należy korzystać rozsądnie. Rzadko, jeżeli w ogóle, powinno się zwiększać rozmiary bitmapy - zazwyczaj bowiem powoduje to powstanie niezbyt miłej dla oka „pikselozoy”. Zmniejszanie obrazka jest bezpieczniejsze, o ile pamiętamy o zachowaniu jego oryginalnego aspektu - znaczy to, że stosunek szerokości

¹⁴⁶ Zgodnie z podaną w `dwRop` operacją rastrową.

do wysokości w źródłowym i docelowym prostokącie powinien być taki sam, jeżeli chcemy otrzymać zadowalający rezultat.

Sposób, w jaki `StretchBlt()` zmienia rozmiary obrazów, można kontrolować funkcją `SetStretchBltMode()`. Poczytaj o niej w [MSDN](#).

Nie mogę też nie wspomnieć o jeszcze jednej, mało przyjemnej cesze funkcji `StretchBlt()`. Otóż jest ona w większości wypadków żałośnie wolna, o wiele wolniejsza niż `BitBlt()`. Jest to spowodowane tym, iż skalowanie rastrowego obrazka wymaga sporo zasobów obliczeniowych komputera - niestety, jak już mówiłem, rezultaty i tak nie są zbyt dobre.

Oczywiście szybkość `StretchBlt()` nie ma zbyt wielkiego znaczenia w aplikacjach użytkowych. Jeśli jednak chcielibyśmy przy pomocy GDI napisać jakąkolwiek grę czy prezentację multimedialną (co jest jak najbardziej możliwe), wtedy `StretchBlt()` może być głównym winowajcą niezadowalającej szybkości działania.

Chcąc zobaczyć przykład wykorzystania funkcji `StretchBlt()`, zerknij na program `Magnifier` dołączony do kursu. Jest to ekranowa lupa, dokonująca powiększenia wybranego kursorem fragmentu pulpitu.

Przezroczyste wyświetlanie

Zarówno `BitBlt()`, jak i `StretchBlt()` mają pewien duży mankament: obie funkcje potrafią wyświetlać wyłącznie prostokątne fragmenty bitmap. To niewystarczające, jeżeli chcemy prezentować obrazki o nieregularnych kształtach. Przykładowo, chcąc wyświetlić obrazek piłki, nie otrzymamy okrągłego zestawu pikseli, lecz prostokąt obejmujący także oryginalne tło bitmapy. Najczęściej nie pasuje ono do tła okna i wtedy zaczynają się problemy.

Z początku radzono sobie z nimi w dość pokrętny sposób. Przygotowywano bowiem po dwie bitmapy tego samego rozmiaru dla każdego *sprite'a* u nieregularnych kształtach:

- pierwszym był właściwy obrazek. Musiał on koniecznie posiadać czarne tło, gdyż w procesie wyświetlania wszystkie czarne piksele były traktowane jako przezroczyste
- drugą bitmapą była tzw. maska. Był to monochromatyczny zestaw pikseli: czarne punkty znajdowały się w miejscach odpowiadających właściwemu obrazkowi (czyli np. piłce), natomiast białymi pikselami wypełniano tło (które na właściwym obrazku było czarne)

Mając tak spreparowany obrazek oraz jego maskę, wyświetlanie częściowo przezroczystego kształtu odbywało się dwuetapowo:

1. Najpierw wykonywano `BitBlt()` dla maski obrazu, posługując się znacznikiem `SRCAND`. Powodowało to zaczerpnienie na ekranie wszystkich pikseli, które w masce były czarne. W miejscu, gdzie miał pojawić się *sprite*, powstawała „czarna dziura”.
2. Następnie przywoływano `BitBlt()` dla właściwego obrazu, tym razem z operacją rastrową `SRCPAINT`. Wówczas czarne piksele tła obrazka nie zmieniały istniejących pikseli na docelowym kontekście urządzenia. Interesująca nas część bitmapy zostawała natomiast przekopiowana w miejsce „czarnej dziury”, zasłaniając ją całkowicie.

Widać, że ten sposób jest co najmniej mocno kombinowany. Na szczęście od czasu wydania Windows 98 nie jesteśmy zmuszeni do wykonywania takich dziwacznych operacji. GDI wzbogaciło się bowiem o niezwykle przydatną funkcję `TransparentBlt()` (czytaj [*transparent blit*]):

```
BOOL TransparentBlt(HDC hdcDest,  
                   int nXOriginDest,
```



```
int nYOriginDest,
int nWidthDest,
int nHeightDest,
HDC hdcSrc,
int nXOriginSrc,
int nYOriginSrc,
int nWidthSrc,
int nHeightSrc,
COLORREF crTransparent);
```

Aby z niej skorzystać, w ustawieniach linkera musisz dodać bibliotekę *msimg32.lib* do listy linkowanych modułów.

W Visual Studio .NET otwórz zakładkę *Solution Explorer*, kliknij prawym przyciskiem myszy na nazwę swojego projektu i wybierz *Properties* z menu podręcznego. Przejdź do zakładki *Linker|Input* i wpisz nazwę biblioteki w polu *Additional Dependencies* (oddzielając ją średnikiem od ewentualnych innych nazw).

Jej prototyp jest podobny do `StretchBlt()`, skąd wynika, że funkcja ta również obsługuje skalowanie obrazka. W większości przypadków nie jest to jednak potrzebne. O wiele ważniejszy jest tutaj ostatni parametr, `crTransparent`. Zastępuje on kod operacji rastrowej, ponieważ w `TransparentBlt()` jest to zawsze `SRCCOPY`. Nie jest to jednak dokładnie to samo kopiowanie, co w `Bit/StretchBlt()`. Ów ostatni parametr pozwala nam podać kolor, który w źródłowym kontekście urządzenia będzie traktowany jako przezroczysty. Innymi słowy, jeżeli `TransparentBlt()` spotka piksel tego koloru w źródłowym prostokącie, to nie **przekopiuje go** - zupełnie tak, jakby był on właśnie **przezroczysty**.

Oto więc mamy sposób na proste wyświetlanie obrazów o nieprostokątnych kształtach. Wystarczy zaznaczyć w ich bitmapach piksele tła tym samym kolorem, a przy prezentacji podać jego wartość do `TransparentBlt()`. Tą drogą otrzymamy na ekranie wyłącznie pożądaną kształt *sprite'a*.

Możemy nawet uwolnić się od konieczności pamiętania koloru, który ma być przezroczysty. Z dużą dozą prawdopodobieństwa można bowiem przyjąć, że tym kolorem jest barwa pierwszego piksela bitmapy - czyli tego o współrzędnych (0, 0). Możliwe jest wówczas napisanie prostej funkcji, dokonującej prezentacji częściowo przezroczystego obrazu:

```
BOOL ShowSprite(HDC hdcDest, int nXDest, int nYDest,
                HDC hdcSprite, int nWidth, int nHeight)
{
    return TransparentBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,
                          hdcSprite, 0, 0, nWidth, nHeight,
                          GetPixel(hdcSprite, 0, 0));
}
```

`TransparentBlt()` nie oferuje nic więcej poza wykluczeniem jednego koloru z kopiowania. Jest to zwykle wystarczające, choć nie zawsze. Chcąc uzyskać bardziej wyrafinowane efekty, musimy sięgnąć po inne środki...

Ciekawostka: łączenie alfa

Generalnie Windows GDI jest przygotowana do pracy z bitmapami o 24-bitowym formacie koloru. Biblioteka posiada jednak ograniczone wsparcie dla kanału alfa w postaci np. funkcji `AlphaBlend()`. Oprócz tego GDI potrafi też wczytywać bitmapy z 32-bitowym formatem pikseli - czyni to funkcja `LoadImage()`, której podamy flagę

`LR_CREATEDIBSECTION`. Ewentualnie można się też posłużyć funkcją `CreateBitmap()`, jeżeli napisaliśmy własny *loader* bitmap z zachowanym kanałem alfa - wtedy w

parametrze `cBitsPerPel` należy podać wartość `32`, a w `lpvBits` wskaźnik do odczytanej samodzielnie tablicy bitów.

Zajmijmy się jednak samą kwestią wyświetlania bitmap z kanałem alfa. W GDI istnieje przeznaczona do tego funkcja `AlphaBlend()`. Ma ona prototyp podobny w swej postaci do znanej z operacji na bitmapach 24-bitowych funkcji `TransparentBlt()` - jest ona bowiem jakby lepszą wersją tej funkcji. Oto i jej deklaracja:

```
BOOL AlphaBlend(HDC hdcDest,
                int nXOriginDest,
                int nYOriginDest,
                int nWidthDest,
                int nHeightDest,
                HDC hdcSrc,
                int nXOriginSrc,
                int nYOriginSrc,
                int nWidthSrc,
                int nHeightSrc,
                BLENDFUNCTION blendFunction);
```

Również podajemy tutaj pozycję i wymiary źródłowego i docelowego prostokąta. Wynika stąd, że `AlphaBlend()` obsługuje także skalowanie kopiowanego obrazka w docelowym kontekście urządzenia.

Oprócz znanych parametrów mamy jeszcze jeden, będą strukturą typu `BLENDFUNCTION`:

```
struct BLENDFUNCTION
{
    BYTE BlendOp;
    BYTE BlendFlags;
    BYTE SourceConstantAlpha;
    BYTE AlphaFormat;
};
```

Mimo że widzimy tu cztery pola, swoboda wypełniania tej struktury jest praktycznie żadna, jako że trzy z nich muszą mieć jedynie słuszne wartości domyślne, a czwarte (`SourceConstantAlpha`) daje sensowny efekt *alpha blendingu* pikseli też tylko przy jednej określonej wartości.

Wszystkie te poprawne wartości przedstawia tabela:

pole	wartość
BlendOp	AC_SRC_OVER
BlendFlags	0
SourceConstantAlpha	255
AlphaFormat	AC_SRC_ALPHA

Tabela 69. Właściwe wartości pól struktury `BLENDFUNCTION`

Poprawne wywołanie funkcji `AlphaBlend()` wygląda więc na przykład tak:

```
const BLENDFUNCTION BF = { AC_SRC_OVER, 0, 255, AC_SRC_ALPHA };
AlphaBlend (hdcEkran, 0, 0, 100, 100,
            hdcPamiec, 0, 0, 100, 100, BF);
```

Niestety, efekty zastosowania tej funkcji nie są zwykle zadowalające - wynikowy obrazek ma zazwyczaj zbyt duży kontrast. Procedura używa bowiem do *blendingu* tradycyjnego wzoru:

$$\tilde{C} = \alpha \cdot \tilde{C}_s + (1 - \alpha) \tilde{C}_d$$

gdzie α to oczywiście wartość kanału alfa obrazka źródłowego, a \tilde{C} , \tilde{C}_s i \tilde{C}_d to kolory: wynikowy, źródłowy oraz istniejący na obrazku docelowym. Wszystkie kanały RGB oraz alfa muszą tu być **znormalizowane**, tzn. mieścić się w przedziale wartości od 0 do 1.

Chcąc **znormalizować bajtową reprezentację** koloru, powinniśmy wartość każdego kanału **podzielić zmiennoprzecinkowo przez 255**.

Jak widać, równanie interpoluje kolory w sposób liniowy, więc efekty mogą być nieco „poszarpane”. Dlatego też lepiej użyć przybliżenia kwadratowego:

$$\tilde{C} = \sqrt{\alpha \cdot \tilde{C}_s^2 + (1 - \alpha) \cdot \tilde{C}_d^2}.$$

Jeżeli zaś chcemy zastosować taki *blending* w praktyce, to piszemy np. taką funkcję:

```
// makro wyłuskujące wartość kanału alfa z piksela ARGB
#define GetAValue(rgba) (BYTE)((rgba) >> 24)

// struktura zawierająca znormalizowane wartości ARGB
struct ARGB { float a, r, g, b; };

// --- funkcja wykonująca łączenie alfa z interpolacją kwadratową -----

void AlphaBlending(HDC hdcSrc, POINT ptSrc,
                  HDC hdcDest, POINT ptDest, SIZE cSize)
{
    COLORREF clSrc, clDest, clResult;
    ARGB rgbaSrc, rgbaDest, rgbaResult = { 255, 0, 0, 0 };

    // kopiujemy piksel po pikselu
    for (unsigned i = 0; i <= cSize.cx; ++i)
        for (unsigned j = 0; j <= cSize.cy; ++j)
        {
            /* pobieramy kolory i normalizujemy je */

            // obrazek źródłowy
            clSrc = GetPixel(hdcSrc, ptSrc.x + i, ptSrc.y + j);
            rgbaSrc.a = GetAValue(clSrc) / 255.0f;
            rgbaSrc.r = GetRValue(clSrc) / 255.0f;
            rgbaSrc.g = GetGValue(clSrc) / 255.0f;
            rgbaSrc.b = GetBValue(clSrc) / 255.0f;

            // obrazek docelowy
            clDest = GetPixel(hdcDest, ptDest.x + i, ptDest.y + j);
            rgbaDest.a = GetAValue(clDest) / 255.0f;
            rgbaDest.r = GetRValue(clDest) / 255.0f;
            rgbaDest.g = GetGValue(clDest) / 255.0f;
            rgbaDest.b = GetBValue(clDest) / 255.0f;

            /* wyliczamy kolor wynikowy */

            // kanał czerwony
            rgbaResult.r = sqrtf(rgbaSrc.a * rgbaSrc.r * rgbaSrc.r
                               + (1 - rgbaSrc.a)
                               * rgbaDest.r * rgbaDest.r);

            // kanał zielony
```

```

        argbResult.g = sqrtf(argbSrc.a * argbSrc.g * argbSrc.g
                             + (1 - argbSrc.a)
                             * argbDest.g * argbDest.g);

        // kanał czerwony
        argbResult.b = sqrtf(argbSrc.a * argbSrc.b * argbSrc.b
                             + (1 - argbSrc.a)
                             * argbDest.b * argbDest.b);

        // przeliczamy na format 0..255
        clResult = RGB(ArgbResult.r * 255,
                       ArgbResult.g * 255,
                       ArgbResult.b * 255);

        /* ustawiamy kolor piksela w obrazku docelowym */
        SetPixelV(hdcDest, clResult);
    }
}

```

Ponieważ jednak są to działania na pojedynczych pikselach, nie należy oczekiwać wielkiej szybkości. Zawsze efektywniejsze będą sprzętowe wspomaganie łączenia alfa w nowoczesnych kartach graficznych, których jednak nie obsługuje GDI. Aby stosować wydajny *alpha blending*, trzeba użyć lepszej biblioteki graficznej, jak np. DirectX.

Rysowanie po bitmapie

Prezentacja gotowej bitmapy nie jest jedyną czynnością, jaką możemy wykonać na obrazie rastrowym w Windows GDI. Zupełnie poprawne jest przecież zwyczajne rysowanie po powierzchni tejże bitmapy przy pomocy wszystkich znanych funkcji interfejsu graficznego.

Dlaczego tak można? Przypomnijmy sobie, że podczas przygotowywania bitmapy tworzymy dla niej osobny (pamięciowy) kontekst urządzenia. Następnie wiążemy ją z tym kontekstem, wobec czego nasza bitmapa **staje się dla niego płótnem**. Zaś płótno, jak wiemy, jest tym miejscem, gdzie wysiłki rysunkowe poczynione w kontekście urządzenia dają widoczny skutek. Wynika stąd, że:

Rysowanie w pamięciowym kontekście urządzenia powoduje modyfikację bitmapy, którą z nim związaliśmy.

Nie widzimy rzecz jasna bezpośrednich efektów funkcji graficznych wywoływanych dla pamięciowego kontekstu. Jest tak, bo kontekst ten z samej nazwy nie może tego zapewnić: jest on tylko pomocniczym tworem rezydującym w pamięci operacyjnej, nie odnosi się do żadnego fizycznego urządzenia.

Tym niemniej potrafilibyśmy zobaczyć efekty swej pracy - wystarczy tylko skorzystać z poznanych w poprzednim akapicie technik transferu bitów między kontekstami. Stosując `BitBlt()`, `StretchBlt()` czy `TransparentBlt()` możemy zaprezentować użytkownikowi dynamicznie wygenerowaną bitmapę w identyczny sposób, w jaki pokazujemy obrazek wczytany z pliku i pozostawiony bez zmian. Daje to spore możliwości tworzenia elementów grafiki w czasie działania programu, a następnie ich wielokrotnego wykorzystywania.

Tekst

Geometria geometrią, bitmapy też są ważne, ale żadna biblioteka graficzna nie może obyć się bez choćby prostych możliwości wyświetlania tekstu. Potencjał Windows GDI jest w tym względzie więcej niż duży: interfejs ten pozwala nie tylko na wielorakie

wypisywanie łańcuchów znaków, ale też na szeroko zakrojoną zmianę jego rozmiaru czy wyglądu. Obsługuje bowiem formatowanie za pomocą **czcionek** (ang. *fonts*).

W tej sekcji przyjrzymy się obu tym kwestiom wykorzystania tekstu w GDI. Zobaczymy więc, jak prezentować napisy w kontekście urządzenia oraz w jaki sposób pracować z czcionkami.

Wypisywanie tekstu

Pisanie tekstu na ekranie jest czynnością tak starą, jak samo programowanie. Pierwszy program, jaki był zaprezentowany w tym kursie, robił nic innego jak właśnie wypisywanie tekstu przy pomocy strumienia wyjścia. Podobnie, pierwsza aplikacja dla Windows również pokazywała nam komunikat, tyle że korzystała z funkcji WinAPI - `MessageBox()`. Można więc powiedzieć, że znowu zataczamy koło i wracamy do zagadnienia omówionego już wielokrotnie. Ale są to tylko pozory: używanie `std::cout` czy `MessageBox()` nie może się bowiem równać z mechanizmami GDI służącymi do wyświetlania tekstu. Dla nich napis jest bowiem kolejnym prymitywem, na którym można wykonywać wszelkiego typu operacje graficzne. Zmiana położenia, koloru czy wreszcie czcionki jest tu całkiem naturalna, podczas gdy w stosowanych przez nas dotąd narzędziach tekstowych - zupełnie niemożliwa.

Windows GDI pozwala zatem na znacznie bardziej elastyczne posługiwanie się tekstem. Poznawanie możliwości biblioteki w tym zakresie musimy jednak zacząć od podstaw. Zobaczymy najpierw, w jaki sposób wypisuje się tekst w domyślnych ustawieniach, a dopiero potem zajmiemy się zmianą jego parametrów - z czcionką na czele.

Proste wyświetlanie

Być może pamiętasz funkcję `TextOut()`, której użyłem kiedyś jako przykładu podczas omawiania odświeżania okna. Jeśli nie, nic straconego - teraz właśnie przypomnimy ją sobie i opiszemy dokładniej.

Istnieją też funkcje: `TabbedTextOut()` oraz `ExtTextOut()`. Pierwsza z nich pozwala wypisać tekst z uwzględnieniem pozycji zdefiniowanych tabulatorów, zaś druga potrafi m.in. zmienić odległości między znakami napisu.

Funkcja `TextOut()`

Rozpocniemy oczywiście od prototypu:

```
BOOL TextOut(HDC hdc,
             int nXStart,
             int nYStart,
             LPCTSTR lpString,
             int cbString);
```

Kolejne parametry nie powinni ci chyba sprawić kłopotu. `hdc` to kontekst urządzenia, w którym zostanie wypisany wypisany tekst. Napis podajemy w parametrze `lpString` - zwróć uwagę, że nie musi to być łańcuch znaków w stylu C (zakończony zerem), ponieważ funkcja chce jeszcze jego długości (liczby znaków) w parametrze `cbString`. Dlatego też jeżeli używamy łańcuchów `std::string`, to nie ma znaczenia, czy skorzystamy z ich metod `c_str()` czy `data()`. W innych funkcjach WinAPI trzeba zawsze stosować tę pierwszą.

Pozycja tekstu

Dwa pozostałe parametry, `nXStart` i `nYStart`, określają pozycję tekstu (punkt referencyjny) w bitmapie kontekstu urządzenia. Interpretacja tych wartości może być

różna; domyślnie oznaczają one współrzędne lewego górnego rogu najmniejszej obwiedni tekstu. Można to aczkolwiek zmienić przy pomocy funkcji `SetTextAlign()`:

```
UINT SetTextAlign(HDC hdc, UINT fMode);
```

Dopuszcza ona kilka rodzajów flag, określających odpowiednie położenie punktu referencyjnego w stosunku do prostokąta otaczającego tekst. Jeden ich rodzaj manipuluje tymże punktem w poziomie, drugi w pionie; trzeci rodzaj mówi jeszcze, czy aktualna pozycja pióra ma się przesunąć w punkt referencyjny (`nXStart`, `nYStart`) po wywołaniu `TextOut()`. Do funkcji możemy podać co najwyżej jedną flagę każdego rodzaju.

Wszystkie te flagi funkcji `SetTextAlign()` podaje poniższa tabela (podkreśleniem zaznaczyłem wartości domyślne):

rodzaj flag	flaga	znaczenie
pozycja pozioma punktu referencyjnego	<u>TA_LEFT</u>	punkt referencyjny po lewej stronie tekstu
	TA_CENTER	punkt referencyjny na środku tekstu
	TA_RIGHT	punkt referencyjny po prawej stronie tekstu
pozycja pionowa punktu referencyjnego	<u>TA_TOP</u>	punkt referencyjny na górze tekstu
	TA_BASELINE	punkt referencyjny na linii bazowej ¹⁴⁷ tekstu
	TA_BOTTOM	punkt referencyjny na dole tekstu
aktualizacja położenia pióra	TA_UPDATECP	pozycja jest brana pod uwagę i uaktualniana
	<u>TA_NOUPDATECP</u>	pozycja pióra nie jest brana pod uwagę

Tabela 70. Flagi bitowe funkcji `SetAlignText()`

Flaga `TA_UPDATECP` sprawia, że funkcja `TextOut()` ignoruje parametry `nXStart` i `nYStart`, a zamiast tego wypisuje tekst w punkcie referencyjnym bieżącej pozycji pióra. Może też jego położenie: przy `TA_LEFT` ustawi je po prawej stronie tekstu, a przy `TA_RIGHT` - po lewej.

Przykład wykorzystania `TextOut()`

Funkcję `TextOut()` wykorzystywaliśmy już parę razy, ale nie zaszkodzi przypomnieć jej zastosowania:

```
std::string strTekst = "Hello world!";

// wypisanie tekstu w lewym górnym rogu np. okna
TextOut(hdc, 0, 0, strTekst.c_str(), strTekst.length());

// pokazanie tekstu w prawym dolnym rogu ekranu
HDC hdcEkran = GetDC(NULL);
SetTextAlign(hdcEkran, TA_RIGHT | TA_BOTTOM);
TextOut(hdcEkran,
        GetSystemMetrics(SM_CXSCREEN), GetSystemMetrics(SM_CYSCREEN),
        strTekst.c_str(), strTekst.length());
ReleaseDC(NULL, hdcEkran);
```

Szczególnie drugi przykład jest interesujący. Użyte w nim wywołania funkcji `GetSystemMetrics()` zwracają wymiary ekranu, czyli rozdzielczość monitora.

Bardziej wyrafinowany sposób

Nieco większą kontrolę nad wypisywaniem tekstu oferuje funkcja `DrawText()`:

¹⁴⁷ Linia bazowa jest linią, poniżej której leżą „ogonki” od liter ‘p’, ‘y’, itd. Można ją utożsamiać z pionowym środkiem tekstu, choć trochę obniżonym.

```
int DrawText(HDC hdc,
             LPCTSTR lpString,
             int nCount,
             LPRECT lpRect,
             UINT uFormat);
```

Od razu spostrzeżemy, że w niej nie ma parametrów odpowiedzialnych bezpośrednio za pozycję wypisywanego tekstu. Zamiast tego mamy prostokąt `lpRect`, który, najogólniej mówiąc, otacza tekst i pozwala na jego wyrównywanie do swoich krawędzi. Za to wyrównywanie, a także za kilka innych opcji, odpowiada parametr `uFormat`. Może on przyjmować zestaw paru flag bitowych. Nie podam ich wszystkich tutaj, ale opiszę kilka kwestii z nimi związanych.

Co do znaczenia pozostałych parametrów nie mam nic odkrywczego do powiedzenia. `lpString` to tekst, który wypisujemy, `nCount` jest jego długością, a `hdc` oznacza docelowy kontekst urządzenia.

Bardziej zaawansowana wersja tej funkcji nosi nazwę `DrawTextEx()`. Potrafi ona nie tylko ustawiać tabulatory, ale też określać marginesy. Znajdźmy jeszcze parę podobnych funkcji, a będziemy bez problemu napisać własny edytor tekstu ;-)

Wymiary prostokąta okalającego

W najprostszym przypadku możemy przyjąć, że pola `left` i `top` prostokąta `lpRect` oznaczają lewy górny róg obramowania napisu. Mogłyby one być odpowiednikami parametrów `nXStart` i `nYStart` funkcji `TextOut()`.

Mogłyby - ale nie do końca. `DrawText()` bierze bowiem pod uwagę cały podany jej prostokąt, sprawdzając, czy tekst zmieści się w nim całkowicie. Jeżeli tak nie będzie, jego „wystająca” część zostanie przycięta, zatem liczba faktycznie wypisanych znaków (wynik funkcji) będzie mniejsza od długości napisu.

Możemy zmienić to domyślne zachowanie: wystarczy podać flagę `DT_NOCLIP` w parametrze `uFormat`. Przycinanie nie będzie wówczas dokonywane, a ponadto sama czynność rysowania tekstu przebiegnie szybciej.

Alternatywnie, możemy zapytać funkcję `DrawText()` o prawidłowe wymiary prostokąta. Aby to uczynić, należy podać jej flagę `DT_CALCRECT`. Takie wywołanie jest odrobinę mylące, ponieważ obecność tej flagi powoduje, że nie jest dokonywane żadne wypisywanie tekstu. Funkcja oblicza po prostu wymiary prostokąta dla tekstu i zapisuje w strukturze o wskaźniku `lpRect`.

Dopiero następne wywołanie funkcji powinno dokonać wyrysowania tekstu - w nim nie dołączamy już flagi `DT_CALCRECT` do ostatniego parametru.

Wyrównanie tekstu

„A w zasadzie to po co mi ten cały prostokąt `lpRect`?...” Słuszna uwaga. Ów prostokąt jest jednak bardzo przydatny w momencie, gdy chcemy wyrównać tekst do krawędzi jakiegoś prostokątnego zakresu rysunku. Może to być chociażby prostokąt obszaru klienta okna albo też figura narysowana przed chwilą za pomocą funkcji `Rectangle()`.

O czymkolwiek byśmy nie mówili, wyrównanie tekstu do brzegów (lub środka) prostokąta jest nadzwyczaj proste. Kontroluje je zbiór sześciu flag - po trzy na rozmieszczenie w pionie i poziomie. Przedstawia je ta oto tabelka (podkreślenie oznacza flagę domyślną):

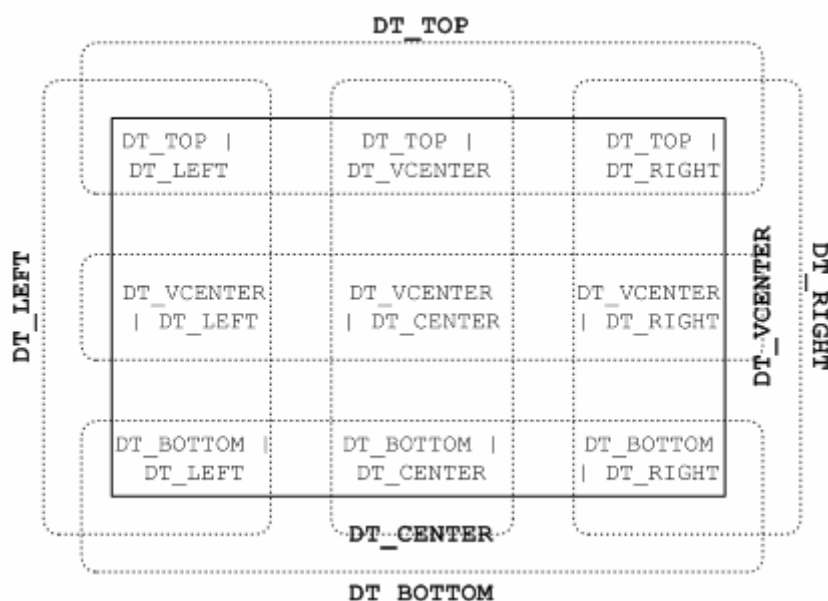
<i>kierunek</i>	<i>flaga</i>	<i>wyrównanie</i>
<i>poziomy</i>	<u>DT_LEFT</u>	do lewej krawędzi
	DT_CENTER	do poziomego środka
	DT_RIGHT	do prawej krawędzi

kierunek	flaga	wyrównanie
pionowy	DT_TOP	do górnej krawędzi
	DT_VCENTER	do pionowego środka
	DT_BOTTOM	do dolnej krawędzi

Tabela 71. Flagi wyrównania tekstu funkcji DrawText()

Musimy aczkolwiek wiedzieć, że zastosowanie wyrównania pionowego innego niż DT_TOP wymaga podania jeszcze flagi DT_SINGLELINE. W takim wypadku wypisywany tekst nie może być podzielony na wiersze.

Ilustracją skutków zastosowania każdej z 9 możliwych kombinacji tych flag jest poniższy rysunek:



Rysunek 27. Wyrównanie tekstu w prostokącie funkcji DrawText()

Można na nim łatwo zauważyć, że funkcja DrawText() potrafi wyrównywać tekst podobnie, jak czynią to zaawansowane edytory tekstu w komórkach tabel.

Dzielenie na wiersze

Przewaga DrawText() ujawnia się także w tym, iż funkcja ta potrafi dzielić wyświetlany tekst na wiersze. Jeżeli nie podamy jej flagi DT_SINGLELINE, to złamie ona nasz napis na znakach powrotu karetki ('\r', kod 0x0D) oraz końca linii ('\n', kod 0x0A). Niepodanie flagi DT_SINGLELINE wyklucza aczkolwiek użycie wyrównania w pionie innego niż DT_TOP.

DrawText() potrafi jednak więcej. Nie tylko interpretuje odpowiednio ustalone przez nas miejsca łamania wierszy, ale też potrafi samodzielnie zająć się podziałem na linijki. Podając jej znacznik DT_WORDBREAK sprawimy, że zostanie od podzielony tak, aby żaden wyraz nie przekraczał prawej krawędzi prostokąta lpRect. Jeżeli miałoby się tak stać, kłopotliwe słowo zostanie w całości przeniesione do nowej linijki. Miejmy na uwadze, że powoduje to zwykle rozrost tekstu w pionie. Przy obecnej fladze DT_WORDBREAK nadal możemy też wstawić ręczny podział linijki: należy wtedy użyć kombinacji dwóch wspomnianych wcześniej znaków podziału (czyli sekwencji "\r\n").

Nie trzeba chyba dodawać, że flagi `DT_SINGLELINE` i `DT_WORDBREAK` wzajemnie się wykluczają i nie mogą wystąpić jednocześnie.

Ustawienia tekstu

Teraz wiemy już całkiem sporo na temat metod wypisywania tekstu i ta wiedza nam chyba wystarczy. Zajmijmy się więc zmianą wyglądu wyświetlanych liter. Na początek poznamy te ustawienia tekstu, które nie wymagają użycia obiektów czcionek. Jest to: kolor tekstu, ustawienia tła oraz odstępy międzyznakowe.

Kolor

W większości aplikacji Windows posługujących się tekstem jego kolor wybieramy często w tym samym oknie, co czcionkę. W GDI kolor tekstu jest jedną kwestią zupełnie odrębną od obiektu czcionki. Kontroluje go bowiem ustawienie kontekstu urządzenia, a te można zmieniać poprzez funkcję `SetTextColor()`:

```
COLORREF SetTextColor(HDC hdc, COLORREF crColor);
```

W parametrze `crColor` podajemy rzecz jasną nową barwę tekstu. Wpłynie ona na wszystkie następujące dalej w kodzie wywołania funkcji `TextOut()`, `DrawText()`, itp. Stary kolor otrzymujemy jako wynik wywołania `SetTextColor()`.

Tło

Jeżeli próbowałeś wypisać jakiś tekst w oknie o domyślnym kolorze (`COLOR_WINDOW`), na pewno zorientowałeś się, że jest on otoczony białym prostokątem. Biał jest domyślnym **kolorem tła** w Windows GDI, który możemy oczywiście zmienić. Służy do tego funkcja `SetBkColor()`:

```
COLORREF SetBkColor(HDC hdc, COLORREF crColor);
```

Działa ona analogicznie jak `SetTextColor()`. W opisanej przed chwilą wywołałibyśmy ją zapewne w takiej formie:

```
SetBkColor (hdcOkno, GetSysColor(COLOR_WINDOW));
```

W ten sposób ustawilibyśmy kolor tła tekstu na zgodny z kolorem okien Windows - białe tło pod napisem zniknęłoby¹⁴⁸.

Lepiej jednak nie polegać na dopasowywaniu kolorów, szczególnie że zupełnie nie sprawdza się przy pisaniu po niejednorodnej powierzchni. W takim wypadku rozsądniejsze jest uczynienie tła całkowicie przezroczystym. Pozwala na to funkcja `SetBkMode()`:

```
int SetBkMode(HDC hdc, int iBkMode);
```

Możemy jej podać jedną z dwóch stałych. Domyślna `OPAQUE` czyni tło widocznym i powoduje jego zamalowanie przy pomocy koloru tła. Druga możliwość to `TRANSPARENT` - w tym ustawieniu tło nie jest wyświetlane i zwykle jest to bardziej pasująca nam ewentualność.

Ustawienia tła modyfikowane przez `SetBkColor()` i `SetBkMode()` mają wpływ nie tylko na tekst, lecz także na wypełnianie przerw pomiędzy liniami w niektórych stylach piór oraz na tło pędzli deseniowych.

¹⁴⁸ Do pełni szczęścia musielibyśmy jeszcze odrysowywać okno w reakcji na komunikat `WM_SYSCOLORCHANGE`.

Odstępy między znakami

Ostatnim ustawieniem tekstu są odstępy między poszczególnymi znakami prezentowanych napisów. Standardowo Windows GDI nie wstawia żadnego dodatkowego odstępu, poza tym zdefiniowanym w aktualnej czcionce. Możemy to jednak zmienić za pomocą funkcji `SetTextCharacterExtra()`:

```
int SetTextCharacterExtra(HDC hdc, int nCharExtra);
```

Zważmy, że pomimo typu drugiego parametru (`int`), nadmiarowy odstęp pomiędzy znakami może być tylko większy niż domyślne zero. Jeżeli podamy tutaj liczbę ujemną (chcąc zapewne ścisnąć litery napisu), Windows zastosuje bezwzględną wartość przekazanego argumentu. Dziwne, ale prawdziwe.

Czcionki

Zaawansowane opcje formatowania tekstu są związane przede wszystkim z czcionkami. Windows GDI zawiera mnóstwo możliwości kontroli tego aspektu tekstu; zajmiemy się nimi w tym rozdziale.

Typy czcionek

Zanim przejdziemy do praktycznego wykorzystywania różnych czcionek, powinniśmy dokładnie wiedzieć, o czym mówimy. Odpowiedzmy więc sobie na pytanie: Czym jest czcionka?

Czcionka (ang. *font*) jest elektroniczną postacią pisma, czyli zestawem obrazków (**glifów**) reprezentujących poszczególne znaki.



Czcionki (zwane też krojami pisma) decydują więc o kształcie liter i cyfr, zatem w największym stopniu wpływają na ich wygląd. Wybór odpowiedniej czcionki jest bardzo ważny dla czytelności dokumentu czy nawet zwykłego komunikatu na ekranie.

Jeżeli chodzi o podział komputerowych czcionek, to można wśród nich wyróżnić:

- czcionki zawierające zestawy znaków alfanumerycznych, wśród których mamy:
 - ✓ czcionki imitujące druk, czyli naśladujące tradycyjne czcionki zecerskie. Wśród nich możemy jeszcze rozgraniczyć:
 - * czcionki szeryfowe (franc. *serif*) - w tych krojach skrajne punkty liter są zakończone małymi, prostopadłymi liniami - tzw. **szeryfami**. W dłuższym tekście szeryfy wyznaczają bazową linię pismę, co niektórzy uważają za ułatwienie w czytaniu. Przykładami znanych czcionek szeryfowych są: Times New Roman, Garamond, Courier New, Sylfaen, Batang
 - * czcionki bezszeryfowe (franc. *sans serif*) nie posiadają szeryfów. Linie znaków kończą się w nich swobodnie. Do najbardziej znanych czcionek tego rodzaju należą: Tahoma, Verdana, Trebuchet, Arial
 - ✓ czcionki naśladujące pismo odręczne. Litery w tych czcionkach są wyposażone w różne „zawijasy”, imitujące naturalne pismo ręczne. Często też występują w nich tzw. ligatury, czyli często występujące połączenia dwóch znaków, zapisane inaczej niż dwa osobne glify (np. 'ff', 'ffi', 'ffi', a w polskim 'ł'). Przykładami takich czcionek są: Inkburner, *Airstream*, Galileo, *Dynamaxion Script*, *Monotype Corsiva*, *Aristoreerat*
 - ✓ czcionki zawierają litery i cyfry o specjalnych, celowo wykonanych kształtach. Mogą one naśladować litery ze znaków popularnych marek (np. Coca Cola czy Disney) lub też być całkowicie oryginalnymi dziełami

Takimi czcionkami są np.: first order, CNN, **PEPSI**, Δϵ\Tϑ∇VΤϑ, NASALIZATION, WALT DISNEY SCRIPT, KERNING

- czcionki bez znaków alfanumerycznych, wśród których mamy:
 - ✓ czcionki zawierające różne, często używane symbole, jak choćby litery greckie czy symbole matematyczne.
Najbardziej znaną czcionką tego typu jest Symbol ($\Sigma\psi\mu\beta\omicron\lambda$)
 - ✓ czcionki, które posiadają zbiór różnych obrazków, używanych najczęściej w celach zdobniczych.

Do najpopularniejszych czcionek z tej grupy należą trzy kroje Wingdings
 oraz Webdings


Powyższy podział jest w wielu miejscach tożsamy z powszechnym terminem **rodziny czcionki** (ang. *font family*).

Z punktu widzenia intensywnego użytkownika czcionek ważny jest jednak inny podział, związany z podatnością znaków na skalowanie. Chodzi tu o wyróżnienie krojów proporcjonalnych i nieproporcjonalnych.

Proporcjonalne

Czcionki **proporcjonalne** charakteryzują się tym, że ich znaki są opisane jako rysunki wektorowe. Nie zawierają więc map bitowych dla poszczególnych glifów, lecz figury geometryczne i równania matematyczne, które je definiują.

Zalety takiego podejścia wydają się oczywiste: podobnie jak grafika wektorowa, czcionki proporcjonalne mogą być skalowane do dowolnych rozmiarów bez widocznej utraty jakości. Bez problemu można nimi pisać tekst zarówno w rozmiarze 10, jak i 72 punktów. Co więcej, wyglądają one identycznie w wydruku mozaikowym, jak i na ekranie monitora.

Wśród czcionek proporcjonalnych najpopularniejsze są czcionki **TrueType** (opisywane krzywami Béziera) oraz Type 1.

Pewnym rodzajem fontów proporcjonalnych są czcionki kreskowe (ang. *stroke fonts*). Ich znaki składają się wyłącznie z linii prostych, poprawdzonych między skończoną liczbą punktów. Takie czcionki teoretycznie mogą być także skalowane do dowolnych rozmiarów, jednak przy małych wielkościach ich czytelność jest niewielka, a przy większych widać „kanciatość” krawędzi znaków. Czcionki kreskowe są używane głównie przez drukarki wektorowe, tj. plotery.

Nieproporcjonalne

Historycznie pierwszym rodzajem czcionek są fonty **nieproporcjonalne**. Stanowiły one zestawy kilkudziesięciu małych bitmap (stąd inna nazwa - czcionki bitmapowe), odpowiadających danemu stylowi pisma i jego wielkości. Takie czcionki występowały często w wielu kopiach, gdyż musiały zapewniać oddzielne glify dla każdego rozmiaru znaków.

Można oczywiście uzyskać wielkości liter nieuwzględnione przez twórcę czcionki rastrowej (kolejna nazwa tych fontów :D), ale będzie to operacja rozciągania bitmapy, co jak wiemy, zwykle nie daje zadowalających rezultatów. Dlatego też czcionki nieproporcjonalne wychodzą z użycia, znajdując wykorzystanie chyba tylko w konsolach tekstowych.

Praca z czcionkami GDI

W Windows GDI czcionkami posługujemy się dokładnie tak, jak piórami czy pędzlami. Najpierw więc tworzymy obiekt czcionki, wybieramy go w kontekście urządzenia, wykonujemy działania graficzne (tutaj: wypisywanie tekstu), a następnie zwalniamy obiekt.

Obiektom czcionek odpowiadają uchwytty typu `HFONT`.

W tym akapicie przyjrzymy się każdej z tych czynności, koncentrując największą uwagę na tworzeniu fontu.

Tworzenie obiektu czcionki

Utworzenie obiektu czcionki (tzw. **czcionki logicznej**, ang. *logical font*) zajmie nam raczej dużo miejsca. Będzie tak choćby ze względu na prototyp funkcji `CreateFont()`, która służy do wykonania tego zadania:

```
HFONT CreateFont(int nHeight,
                 int nWidth,
                 int nEscapement,
                 int nOrientation,
                 int fnWeight,
                 DWORD fdwItalic,
                 DWORD fdwUnderline,
                 DWORD fdwStrikeOut,
                 DWORD fdwCharset,
                 DWORD fdwOutputPrecision,
                 DWORD fdwClipPrecision,
                 DWORD fdwQuality,
                 DWORD fdwPitchAndFamily,
                 LPCTSTR lpszFace);
```

Możesz przecierać oczy, możesz wyglądać za okno, możesz się uszczypnąć lub zrobić cokolwiek innego, ale ten prototyp nie zniknie, bo on wcale nie jest sennym koszmarem :D Naprawdę funkcja ta ma aż **czternaście parametrów** - to prawdopodobnie jedna z rekordzistek w Windows API pod tym względem.

Cóż więc począć z taką gigantyczną funkcją?... Mogę cię tylko pocieszyć tym, iż zdecydowana większość parametrów da się ustawić na sensowne wartości domyślne, prawidłowe w większości przypadków. W praktyce więc najlepiej będzie opakować funkcję `CreateFont()` w coś bardziej dla nas przyjaznego: własną funkcję czy nawet klasę.

Abyś jednak mógł to uczynić, powinieneś przynajmniej spojrzeć na znaczenie wszystkich parametrów oryginalnej funkcji. W tym przypadku stosowana tabelka będzie wyjątkowo pomocna:

typ	parametry	opis
<code>int</code>	<code>nHeight</code>	<p>Podajemy tu wysokość znaków w tworzonej czcionce logicznej. Miara ta nie jest jednak wyrażona w punktach typograficznych, ale w jednostkach logicznych - przy najczęstszym trybie mapowania <code>MM_TEXT</code> oznacza to wysokość w pikselach.</p> <p><code>nHeight</code> jest liczbą typu <code>int</code>, ponieważ może być zarówno dodatnią, jak i ujemną wartością:</p> <ul style="list-style-type: none"> ➤ wartość dodatnia oznacza, że podajemy wysokość tzw. komórki znaku (ang. <i>character cell</i>); jest to prostokąt, w którym twórca czcionki zmieścił wszystkie znaki kroju ➤ wartość ujemna wskazuje, że bezwzględna liczba oznacza wysokość rzeczywistego znaku, zwykle mniejszą niż wysokość jego komórki ➤ zero oznacza przyjęcie przez Windows

typ	parametry	opis
		<p>domyślnej wysokości czcionki</p> <p>Ponieważ wielkość w jednostkach logicznych czy nawet w pikselach nie jest czasami tym, o co nam chodzi przy używaniu czcionki, cytuję za MSDN formułę pozwalającą przeliczać żadaną wysokość znaku w punktach na piksele (wymagany tryb mapowania <code>MM_TEXT</code>):</p> <pre>nHeight = -MulDiv(nPunkty, GetDeviceCaps(hdc, LOGPIXELSY), 72);</pre> <p>Zauważmy jednak, że jeśli chcemy posługiwać fontem w celach bardziej graficznych, wtedy wysokość podana w pikselach nie jest wcale złym rozwiązaniem.</p>
int	nWidth	<p>Możemy tutaj podać przeciętną szerokość znaków tworzonego stylu pisma. W większości przypadków nie należy tutaj zdawać się na własną intuicję lub przypadek, ponieważ źle dobrana wartość zaburza aspekt znaków (chyba że jest to celowe). Dlatego też najlepiej wpisać tu <code>0</code>, zostawiając kwestię szerokości znaków samemu systemowi GDI.</p>
int	nEscapement	<p><code>nEscapement</code> określa kąt nachylenia między bazową linią pisma a dodatnią osią X. Podajemy go, uwaga, w dziesiątych częściach stopnia (!) - oznacza to np., iż wartość <code>900</code> spowoduje pisanie tekstu w kierunku pionowym w górę.</p>
int	nOrientation	<p><code>nOrientation</code> jest podobny do poprzedniego parametru z tym, że określa kąt nachylenia poszczególnych znaków. Używa przy tym tej samej miary, czyli 1/10 stopnia.</p> <p>W kompatybilnym trybie grafiki (czyli zdecydowanie najczęściej, bo domyślnie) wartości <code>nEscapement</code> i <code>nOrientation</code> powinny być równe.</p> <p>Zwykle zarówno w <code>nEscapement</code>, jak i w <code>nOrientation</code> podajemy <code>0</code>, co oznacza pisanie tekstu w poziomie.</p>
int	fnWeight	<p>W tym parametrze podajemy pogrubienie czcionki. Dozwolone są tu wartości od <code>0</code> do <code>1000</code>, przy czym większe liczby oznaczają grubsze pismo.</p> <p>Najczęściej stosuje się tu jednak wartość <code>400</code> (lub stałe <code>FW_NORMAL</code>/<code>FW_REGULAR</code>), oznaczającą normalną czcionkę, lub <code>700</code> (względnie <code>FW_BOLD</code>), odpowiadającą zwykłemu pogrubieniu. Powód jest prosty: w przypadku czcionek TrueType pogrubienie nie może być aplikowane dowolnie, gdyż każdy jego stopień wymaga dodatkowego fontu. Zatem podawanie wartości innych niż <code>400</code> lub <code>700</code> będzie zaokrąglane do najbliższych możliwych do zrealizowania.</p> <p>Podanie zera powoduje stworzenia czcionki o normalnej grubości pisma.</p>

<i>typ</i>	<i>parametry</i>	<i>opis</i>
DWORD	fdwItalic fdwUnderline fdwStrikeOut	Oto są trzy wartości <code>BOOL</code> owskie, określające dodatkowe efekty dla czcionki. Jest to odpowiednio: kursywa (italik), podkreślenie i przekreślenie . Podanie w tych parametrach <code>TRUE</code> powoduje zastosowanie efektu, <code>FALSE</code> da przeciwny efekt.
DWORD	fdwCharset	Definiuje zestaw znaków (ang. <i>charset</i>), z jakiego chcemy korzystać. Najczęściej wykorzystywanymi wartościami są tu: <ul style="list-style-type: none"> ➤ <code>ANSI_CHARSET</code> - oryginalny zestaw znaków ANSI ➤ <code>OEM_CHARSET</code> - zestaw zależny od systemu operacyjnego ➤ <code>SYMBOL_CHARSET</code> - zestaw symboli ➤ <code>DEFAULT_CHARSET</code> - domyślny zestaw znaków, zwykle ANSI <p>Chcąc używać polskich liter diakrytycznych, należy skorzystać z zestawu <code>EASTEUROPE_CHARSET</code>.</p>
DWORD	fdwOutputPrecision	Ten parametr określa, jak bardzo serio funkcja <code>CreateFont()</code> ma traktować podane jej w <code>nHeight</code> , <code>nWidth</code> , <code>nEscapement</code> , <code>nOrientation</code> oraz <code>fdwPitchAndFamily</code> dane. Wiadomo, że ścisłe dopasowanie się do tych parametrów może albo być niemożliwe, albo powodować duże zniekształcenia wyglądu znaków.
DWORD	fdwClipPrecision	W praktyce ten parametr określa, czy chcemy korzystać z czcionek TrueType, czy też nie. Ponieważ trudno nie chcieć z nich korzystać, więc w tym polu wpisuje się zwykle <code>OUT TT PRECIS</code> .
DWORD	fdwQuality	Tutaj podajemy funkcji, w jaki sposób tworzona czcionka ma ulegać przycinaniu (np. w funkcji <code>DrawText()</code>). Zwykle nie ma to szczególnego znaczenie i dlatego stosujemy tutaj stałą <code>CLIP_DEFAULT_PRECIS</code> .
DWORD	fdwQuality	<code>fdwQuality</code> specyfikuje pożądaną jakość czcionki . Można tutaj określić, czy na przykład chcemy skorzystać z mechanizmu wygładzania krawędzi (ang. <i>antialiasing</i>) - wtedy podajemy stałą <code>ANTIALIASED_QUALITY</code> .
DWORD	fdwQuality	To pole ma również znaczenie przy możliwym skalowaniu czcionek rastrowych. Podając tu <code>DRAFT_QUALITY</code> pozwalamy na tę operację, co aczkolwiek nie musi wyglądać zbyt dobrze. <code>PROOF_QUALITY</code> zapobiega takiemu skalowaniu, więc tekst pisany fontem bitmapowym może być mniejszy niż założony.
DWORD	fdwQuality	Ponieważ jednak obecnie mamy do czynienia głównie z czcionkami TrueType, parametr ten nie ma zbyt wielkiego znaczenia. Zazwyczaj podajemy w nim <code>DEFAULT_QUALITY</code> . Wartość leży w połowie drogi między ścisłym dopasowywaniem się do parametrów funkcji (<code>DRAFT_QUALITY</code>), a w miarę dobrym wyglądem tekstu (<code>PROOF_QUALITY</code>) przy użyciu czcionek

<i>typ</i>	<i>parametry</i>	<i>opis</i>
		rastrowych. Dla fontów proporcjonalnych wartość w <code>fdwQuality</code> zdaje się w ogóle nie mieć żadnego znaczenia.
DWORD	<code>fdwPitchAndFamily</code>	<p>To pole to modelowy przykład oszczędności w przekazywaniu informacji... a może tylko rozpaczliwa próba uczynienia prototypu funkcji <code>CreateFont()</code> mniej odstręczającym?...</p> <p>Niezależnie od tego, jak jest naprawdę, pole <code>fdwPitchAndFamily</code> stało się zbiorem dwóch informacji. Łączymy je za pomocą alternatywy bitowej, czyli operatora <code> </code>.</p> <p>Pierwszą daną jest tzw. skok (ang. <i>pitch</i>) czcionki. Określa on, czy szerokość znaków czcionki jest stała (<code>FIXED_PITCH</code>) - jak to jest np. w foncie Courier New - czy też zmienna (<code>VARIABLE_PITCH</code>) - w większości czcionek. Najczęściej stosujemy tu trzecią wartość <code>DEFAULT_PITCH</code>, co oznacza słuszny brak zainteresowania tym problemem.</p> <p>Druga wartość precyzuje rodzinę czcionki (ang. <i>font family</i>). Rozsądną domyślną wartością, z jakiej zwykle korzystamy, jest tu <code>FF_DONTCARE</code>.</p>
LPCTSTR	<code>lpszFace</code>	<p>Dopiero na ostatku mamy ten parametr, który wydawałby się najważniejszy. W <code>lpszFace</code> podajemy bowiem nazwę czcionki, której logiczny obiekt chcemy stworzyć.</p> <p>Możliwe jest jednak pominięcie tego parametru i podanie w nim <code>NULL</code>. <code>CreateFont()</code> wykorzysta wtedy wartości nadane pozostałym parametrom i wykorzysta pierwszą napotkaną w systemie czcionkę, która im odpowiada.</p>

Tabela 72. Parametry funkcji `CreateFont()`

Wyjaśnienie ostatniego parametru funkcji - `lpszFace` - tłumaczy ogólną ilość parametrów `CreateFont()`. Większość z nich jest bowiem przygotowana na okoliczność nieobecności w systemie czcionki o nazwie podanej na końcu. W takiej sytuacji dokonana zostanie próba wybrania alternatywnego fontu, najbardziej pasującego do pokaźnej liczby danych przekazanych funkcji.

Takie zachowanie rzadko jest pożądane, bowiem nawet najlepsze dopasowanie wykonane przez komputer nie będzie równało się z oceną estetyczną dokonaną przez grafikę czy choćby programistę. Dlatego też lepiej jest zadbać o to, aby `CreateFont()` na pewno znalazła czcionkę, której nazwę podajemy jej w `lpszFace`. Można to uczynić dwojako:

- korzystając tylko z tych krojów pisma, które są standardowo dostępne w Windows. Mamy wtedy gwarancję, że na każdym systemie użytkownika będą one obecne.
Do standardowych fontów należą: Arial, Courier New, Times New Roman (wszystkie z wariantami Bold i Italic), Symbol, Fixedsys, System, Terminal, Courier, MS Serif, MS Sans Serif i Small Fonts. Tylko pierwsze cztery są czcionkami TrueType
- dołączając do programu każdą użytą, niestandardową czcionkę i dbając o to, aby trafiła ona do katalogu *Fonts* w Windows. Zwykle oznacza to konieczność zapewnienia aplikacji programu instalacyjnego

Niezależnie od tego, który sposób wybierzemy, możemy zignorować znaczenie większości parametrów `CreateFont()`. A ponieważ twórcy Windows API wykazali się zdolnościami profetycznymi i przewidzieli, że tak postąpimy, przygotowali dla nas ułatwienie.

Tym udogodnieniem jest funkcja `CreateFontIndirect()` i struktura `LOGFONT`, na którą wskaźnik jako jedyny parametr przyjmuje owa funkcja. Pola struktury odpowiadają natomiast parametrom `CreateFont()`. W połączeniu z faktem, że większość wspomnianych wartości domyślnych dla parametrów wyraża się zerami, otrzymujemy prosty sposób tworzenia czcionek. Wystarczy bowiem:

- zadeklarować i wyzerować (`ZeroMemory()`) strukturę typu `LOGFONT`
- wypełnić tych kilka pól, które nas interesują
- wywołać funkcję `CreateFontIndirect()`, podając jej adres struktury

Opierając się na tym, możemy łatwo napisać prostszą wersję funkcji do tworzenia logicznych fontów:

```

HFONT CreateLogFont(HDC hdcKontekst,
                    const std::string& strNazwa, unsigned uWysPunkty,
                    bool bPogrubienie = false,
                    bool bKursywa = false,
                    bool bPodkreslenie = false,
                    bool bPrzekreslenie = false)
{
    if (strNazwa.empty() || strNazwa.size() > 31 || uWysPunkty == 0)
        return NULL;

    /* tworzymy czcionkę */

    // deklarujemy i zerujemy strukturę LOGFONT
    LOGFONT Font;
    ZeroMemory(&Font, sizeof(LOGFONT));

    // wypełniamy strukturę LOGFONT
    CopyMemory(Font.lfFaceName, strNazwa.c_str(), strNazwa.size() + 1);
    Font.lfCharSet = DEFAULT_CHARSET;
    Font.lfHeight = -MulDiv(uWysPunkty,
                           GetDeviceCaps(hdcKontekst, LOGPIXELSY), 72);
    Font.lfWeight = (bPogrubienie ? FW_BOLD : FW_NORMAL);
    Font.lfItalic = bKursywa;
    Font.lfUnderline = bPodkreslenie;
    Font.lfStrikeOut = bPrzekreslenie;

    // wywołujemy funkcję CreateFontIndirect()
    return CreateFontIndirect(&Font);
}

```

Patrząc na jej prototyp stwierdzimy, że obsługuje ona tylko podstawowe efekty tekstowe. Są one jednak w wielu przypadkach wystarczające. Chcąc osiągnąć bardziej skomplikowane ustawienia, musimy sami pobawić się z funkcją `CreateFontIndirect()`.

Wybieranie czcionki dla kontekstu urządzenia

Przy wiązaniu gotowej czcionki logicznej z kontekstem urządzenia powtarza się ten sam schemat, który przerabialiśmy już dla piór, pędzli i płócien.

Wybieramy więc font w kontekście urządzenia, zachowując jednocześnie starą czcionkę:

```

HFONT hfntVerdana = CreateLogFont(hdcKontekst, "Verdana", 10);

```

```
HFONT hfntStara = (HFONT) SelectObject(hdcKontekst, hfntVerdana);
```

Wszelkie inne sposoby - vide usunięcie poprzedniej czcionki lub zachowanie stanu kontekstu - są naturalnie również poprawne.

Zwalnianie obiektu czcionki

Zwalnianie czcionki po użyciu nie przynosi żadnych niespodzianek. Ponownie musimy zatroszczyć się o to, aby zarówno nasza, jak i oryginalna czcionka kontekstu została usunięta podczas zwalniania kontekstu urządzenia.

Muszę tu wyjaśnić możliwe nieporozumienie. Otóż „usuwanie czcionki”, o jakim mówi ten punkt, nie ma nic wspólnego z fizyczną eksterminacją pliku *.ttf* lub *.fon*, gdzie fizyczne czcionki rezydują na dysku. Usunięcie czcionki jest tu tylko usunięciem jej reprezentacji w postaci obiektu Windows GDI - rzeczywisty krój pisma pozostaje nienaruszony. Podobna uwaga może też dotyczyć bitmap GDI.

Kod usuwający czcionkę stworzoną i wybraną w poprzednim punkcie może więc wyglądać tak:

```
SelectObject ((hdcKontekst, hfntStara);  
DeleteObject (hfntVerdana);
```

Jeżeli po nim nastąpi usunięcie kontekstu, to oczywiście czcionka z uchwytem w *hfntStara* także zostanie zwolniona.

Przypomnieniem tej wielokrotnie przypominanej drogi zakończymy ten obszerny podrozdział. Omówiłem w nim wszystkie podstawowe prymitywy Windows GDI, których możesz użyć do prezentacji wszelkiego rodzaju grafiki w systemie Windows. Zajęliśmy się więc figurami geometrycznymi, bitmapami rastrowymi oraz tekstem i czcionkami.

Na prymitywach nie kończy się wszakże biblioteka GDI. Większość z pozostałych jej możliwości nie jest jednak na tyle ważna i interesująca, by poświęcać im miejsce w tym kursie - który nie jest, bądź co bądź, kursem samego tylko Windows API czy GDI. Istnieje aczkolwiek jeden mechanizm wart bliższego poznania - to regiony. Przybliżymy je sobie w następnym podrozdziale.

Regiony i przycinanie

Regiony są elementem GDI kontrolującym obszar rysowania oraz umożliwiającym wykonywanie niektórych operacji graficznych.

Region jest zespołem figur zamkniętych: prostokątów, wielokątów i elips.

W Windows GDI regionom odpowiadają uchwyty typu *HRGN*. W niniejszym podrozdziale zobaczymy, jak można tworzyć regiony i do czego się one przydają.

Tworzenie regionów

Proces tworzenia regionu zależy od tego, jak bardzo ma on być skomplikowany. Dla prostych obiektów ogranicza się to do wywołania jednej funkcji, bardziej złożone regiony wymagają nieco więcej pracy związanej z łączeniem regionów elementarnych.

Tutaj zobaczymy zarówno kreowanie najprostszych, jak i nieco bardziej pokrętnych regionów.

Proste regiony

Najprostszy region składa się z pojedynczej figury zamkniętej. Jego utworzenie oznacza wywołanie jednej funkcji, bardzo podobnej do tej, która rysuje ową figurę w kontekście urządzenia.

Prostokątny region

Typowym przedstawicielem regionów jest wariant prostokątny. Za jego utworzenie odpowiada funkcja `CreateRectRgn()`:

```
HRGN CreateRectRgn(int nLeftRect,
                  int nTopRect,
                  int nRightRect,
                  int nBottomRect);
```

Inną możliwością jest też `CreateRectRgnIndirect()`:

```
HRGN CreateRectRgnIndirect(CONST RECT* lprc);
```

Jak widać, obie funkcje przyjmują te same dane, z tym że jedna pobiera je jako cztery parametry, a druga w postaci struktury `RECT`.

Wynikiem jest oczywiście uchwyt do regionu w kształcie prostokąta o podanych wymiarach.

Eliptyczny region

Niemal identycznie wyglądają funkcje tworzące regiony eliptyczne:

```
HRGN CreateEllipticRgn(int nLeftRect,
                      int nTopRect,
                      int nRightRect,
                      int nBottomRect);
```

```
HRGN CreateEllipticRgnIndirect(CONST RECT* lprc);
```

Podajemy do nich prostokąt okalający elipsę, której kształt przyjmie region. Jest ten sam mechanizm, jaki mogliśmy zaobserwować w funkcji `Ellipse()`; przypomnij sobie działanie tej funkcji, jeżeli go nie pamiętasz.

Wielokątny region

Region w kształcie dowolnego wielokątu jest także możliwy do stworzenia. Wystarczy posłużyć się funkcją `CreatePolygonRgn()`:

```
HRGN CreatePolygonRgn(CONST POINT* lppt,
                    int cPoints,
                    int fnPolyFillMode);
```

Jej parametry są niemal identyczne jak w funkcji `Polygon()`. Ostatni argument `fnPolyFillMode` określa tryb wypełniania wielokątów, jaki będzie użyty do stworzenia regionu. Obowiązują tu te same dwie stałe, jak w przypadku analogicznego trybu - atrybutu kontekstu urządzenia. Tak więc wartość `ALTERNATE` powoduje, że części wielokąta powstałe z przecięcia się jego boków będą wypełniane na przemian, zaś `WINDING` gwarantuje, że region będzie składał się z całkowicie zamkniętej figury, bez żadnych „dziur”.

Łączenie regionów

Tworzenie prostych regionów nie byłoby niczym nadzwyczajnym, gdyby nie to, że możemy je ze sobą łączyć (ang. *combine*). To łączenie regionów jest najważniejszą i najpotężniejszą ich cechą.

Do łączenia używamy funkcji `CombineRgn()`:

```
int CombineRgn(HRGN hrgnDest,
               HRGN hrgnSrc1,
               HRGN hrgnSrc2,
               int fnCombineMode);
```

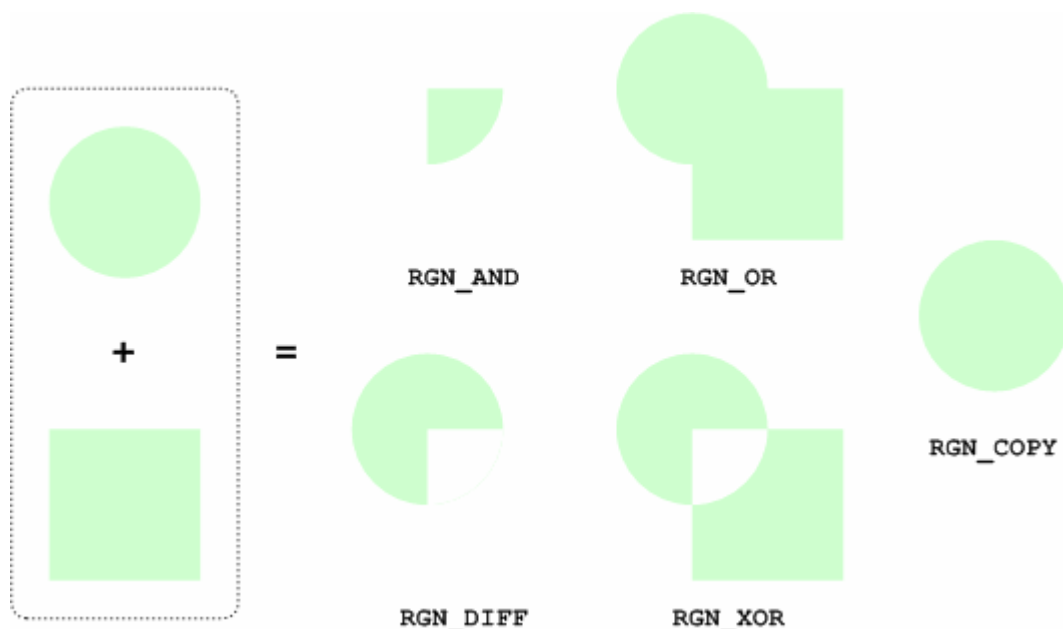
Tworzy ona kombinację regionów `hrgnSrc1` i `hrgnSrc2`, zapisując ją w regionie `hrgnDest`. Region ten musi więc istnieć, ale sposób jego utworzenia nie ma znaczenia, gdyż i tak zostanie on zastąpiony przez połączenie `hrgnSrc1` i `hrgnSrc2`.

W jaki sposób regiony są łączone? O tym decyduje czwarty parametr funkcji - `fnCombineMode`. Określa on operację na regionach `hrgnSrc1` i `hrgnSrc2`, w wyniku której powstanie docelowy region `hrgnDest`. Możliwe działania ujmuje tabela:

<i>stała</i>	<i>nazwa operacji</i>	<i>wynik operacji</i>
<code>RGN_AND</code>	iloczyn (część wspólna)	$\text{hrgnSrc1} \cap \text{hrgnSrc2}$
<code>RGN_OR</code>	suma	$\text{hrgnSrc1} \cup \text{hrgnSrc2}$
<code>RGN_DIFF</code>	różnica	$\text{hrgnSrc1} - \text{hrgnSrc2}$
<code>RGN_XOR</code>	różnica symetryczna	$\text{hrgnSrc1} \oplus \text{hrgnSrc2}$
<code>RGN_COPY</code>	kopia	<code>hrgnSrc1</code>

Tabela 73. Stałe operacji łączenia regionów funkcji `CombineRgn()`

Potencjalne operacje najlepiej jednak prześledzić na rysunku, np. takim:



Rysunek 28. Łączenie regionów różnymi operacjami przy pomocy funkcji `CombineRgn()`

Widzimy na nim, że skomplikowane regiony możemy łatwo składać z prostszych, a ponadto mamy pełną kontrolę nad procesem ich łączenia.

Zajmijmy się jeszcze wartością zwracaną przez `CombineRgn()`. Oto są możliwe jej rezultaty:

<i>stała</i>	<i>znaczenie</i>
<code>NULLREGION</code>	powstały region jest pusty
<code>SIMPLEREGION</code>	wynikowy region ma kształt prostokąta
<code>COMPLEXREGION</code>	powstał region o skomplikowanym kształcie
<code>ERROR</code>	wystąpił błąd

Tabela 74. Możliwe wyniki zwracane przez funkcję `CombineRgn()`

Informują one nie tylko o powodzeniu lub niepowodzeniu operacji kombinowania, ale też dają pewne pojęcie na temat jej rezultatu.

Wykorzystanie regionów

Skoro wiemy już, jak tworzyć regiony, dowiedzmy się, do czego możemy je wykorzystać. W tej sekcji zobaczysz zastosowania regionów w rysowaniu, przycinaniu oraz zmianie kształtu okien.

Rysowanie z pomocą regionu

Jako zbiory figur zamkniętych, regiony mogą być w rysowaniu. Możliwe jest ich wykorzystanie na kilka sposobów.

Wypełnianie pędzlem

Najprostszą czynnością jest wypełnienie obszaru regionu pędzlem, czyli pozostawienie przezeń pewnego rodzaju śladu na bitmapie kontekstu urządzenia.

Operację tę można przeprowadzić na przykład za pomocą funkcji `PaintRgn()`:

```
BOOL PaintRgn(HDC hdc, HRGN hrgn);
```

Używa ona pędzla aktualnie wybranego w kontekście `hdc` to wypełnienia obszaru, jaki wyznacza region o podanym uchwycie `hrgn`. Przyjmuje też, że współrzędne regionu są wyrażone w jednostkach logicznych.

Podobną funkcją jest `FillRgn()`:

```
BOOL FillRgn(HDC hdc,
             HRGN hrgn,
             HBRUSH hbr);
```

Widać w niej podobieństwo do `FillRect()`, lecz ma ona nieco większe możliwości, ponieważ używa regionów, nie zaś prostokątów. `FillRgn()` także wypełnia region pewnym pędzlem, z tym że pozwala na podanie w trzecim parametrze. Pędzel ten nie musi być więc wybrany w kontekście urządzenia `hdc`. Ponownie, funkcja uznaje, że koordynaty regionu są zapisane w postaci jednostek logicznych.

Obrysowywanie

Regiony mają też funkcję będącą odpowiednikiem `FrameRect()` - jest to `FrameRgn()`:

```
BOOL FrameRgn(HDC hdc,
              HRGN hrgn,
              HBRUSH hbr,
              int nWidth,
```

```
int nHeight);
```

Dokonuje ona obrysowywania krawędzi regionu, używając do tego pędzla podanego w parametrze `hbr`. Dwa ostatnie argumenty, `nWidth` i `nHeight`, oznaczają natomiast szerokość i wysokość linii obramowania.

Oto przykład, jak można narysować obramowanie regionu składającego się z dwóch prostokątnych elips:

```
/* utworzenie regionu */

// regiony elementarne
HRGN hrgnElipsa1 = CreateEllipticRgn(0, 30, 90, 60);
HRGN hrgnElipsa2 = CreateEllipticRgn(30, 0, 60, 90);

// kombinacja regionów
HRGN hrgnRegion = CreateRectRgn(0, 0, 0, 0);
CombineRgn (hrgnRegion, hrgnElipsa1, hrgnElipsa2, RGN_OR);

// usunięcie regionów elementarnych
DeleteObject (hrgnElipsa1);
DeleteObject (hrgnElipsa2);

/* obramowanie */

// stworzenie pędzla malującego na zielono
HBRUSH hbrPedzel = CreateSolidBrush(RGB(0, 255, 0));

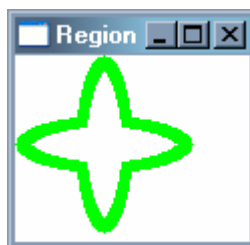
// wykonanie obramowania
FrameRgn (hdcKontekst, hrgnRegion, hbrPedzel, 5, 5);

/* porządki */

// usunięcie pędzla
DeleteObject (hbrPedzel);

// usunięcie regionu
DeleteObject (hrgnRegion);
```

Wynikiem wykonania powyższego kodu (przy założeniu, że `hdcKontekst` jest uchwyttem kontekstu wnętrza okna) będzie poniższy obrazek:



Screen 70. Obramowanie regionu

Inwersja kolorów

Ostatnią operacją z gatunku rysunkowych jest inwersja pikseli kontekstu urządzenia we wnętrzu regionu. Wykonuje ją funkcja `InvertRgn()`:

```
BOOL InvertRgn(HDC hdc, HRGN hrgn);
```


Inwersja oznacza negację bitową wartości koloru, tak samo jak w funkcji `InvertRect()`. Kolor biały staje się więc czarnym, zielony - karmazynowym, itd.

Użycie regionu do przycinania

Innym zastosowaniem regionów w GDI jest ich użycie do **przycinania**.

Przycinanie (ang. *clipping*) zapewnia, że zostaną wyświetlone tylko te piksele płótna kontekstu urządzenia, które leżą **wewnątrz ustalonego regionu** - nazywamy go **regionem przycinania** (ang. *clipping region*).

Region przycinania jest własnością kontekstu urządzenia, podobnie jak pióro, pędzel, płótno i czcionka. Możemy go pobierać i ustawiać, aby kontrolować wyniki pokazywane na ekranie.

Początkowo region przycinania obejmuje oczywiście cały obszar rysowania kontekstu urządzenia.

Ustawienie nowego regionu przebiega jednak nieco inaczej niż podobne postępowanie z innymi obiektami GDI. Wygląda bowiem np. tak:

```
SelectObject (hdcKontekst, hrgnRegion);
```

Zignorowanie rezultatu funkcji `SelectObject()` jest tu jak najbardziej możliwe, gdyż nie jest nim wcale uchwyt do starego regionu. Kontekst urządzenia nie wykorzystuje bowiem obiektu `hrgnRegion` w sposób bezpośredni, lecz tworzy jego kopię. Ta kopia jest w całkowitej władzy kontekstu urządzenia - kontekst sam ją zwalnia, gdy nie jest już potrzebna, a dzieje się to choćby wtedy, kiedy ustawiamy inny region przycinania. Dlatego też możemy bezpiecznie zignorować wartość zwróconą przez `SelectObject()`. Faktycznie nie jest to żaden uchwyt, lecz jedna ze stałych, będących rezultatami `CombineRgn()`.

Innym sposobem ustawienia regionu przycinania jest funkcja `SelectClipRgn()`:

```
SelectClipRgn (hdcKontekst, hrgnRegion);
```

Do niej także stosuje się reguła kopiowania regionu i możliwego zignorowania wartości zwracanej.

A co ze zwalnianiem regionów?... Kod dla tego zadania jest skromniejszy, ponieważ nie mamy żadnego „oryginalnego regionu przycinania”, który należałoby przywrócić kontekstowi urządzenia przed jego zwolnieniem. Naszym zadaniem jest tylko usunięcie tego regionu, który sami stworzyliśmy:

```
DeleteObject (hrgnRegion);
```

Można to zrobić zarówno przed, jak i po usunięciu kontekstu urządzenia.

Zmiana kształtu okna

Jedną z bardziej interesujących opcji wykorzystania regionu jest użycie go zmiany kształtu okna. Mówiąc „kształt”, mam na myśli ten obszar okna, który jest rysowany i zasłania okna leżące niżej w porządku Z.

Ustawianie regionu okna

Chcąc zmienić kształt okna, ustawiamy jego region przy pomocy `SetWindowRgn()`:

```
int SetWindowRgn(HWND hWnd,
                 HRGN hRgn,
                 BOOL bRedraw;
```

Trzeci parametr funkcji informuje Windows, czy po ustawieniu regionu ma dokonać odrysowania okna. Prawie zawsze chcemy tego, toteż ustawiamy ten parametr na `TRUE`.

Musimy wiedzieć, że po wywołaniu `SetWindowRgn()` region podany w `hRgn` staje się własnością systemu Windows. Zatem:

Nie powinniśmy nic robić z regionem, którego uchwyt przekazaliśmy do funkcji `SetWindowRgn()`.

Nie musi nawet dbać o jego usunięcie, zajmie się tym system operacyjny podczas niszczenia okna.

Koordynaty regionu okna są liczone względem **położenia okna**, nie zaś względem jego obszaru klienta. Jest tak, bo region obejmuje swoim zasięgiem nie tylko wnętrze okna, ale też jego obszar pozakliencki.

Pobieranie regionu okna

Dla porządku zerknijmy jeszcze na funkcję pobierającą region okna, `GetWindowRgn()`:

```
int GetWindowRgn(HWND hWnd, HRGN hRgn);
```

Zapisuje ona kopię regionu, zastępując nią region o podanym uchwycie `hRgn`. Tak więc żeby wywołać tę funkcję, musimy już posiadać jakiś region - możemy go stworzyć podobnie jak region docelowy dla `CombineRgn()`, tzn. tak:

```
HRGN hrgnRegion = CreateRectRgn(0, 0, 0, 0);
```

Sposób nie ma żadnego znaczenia, ponieważ region i tak zostanie zastąpiony po wywołaniu `GetWindowRgn()`.

Na tym zakończymy nasze spotkanie z regionami. Pozostałe funkcje, które ich dotyczą, związane są głównie z obiektami ścieżek, a tych zdecydowałem się nie omawiać. Naturalnie, jeżeli ten temat interesuje cię bardziej, możesz zawsze zajrzeć do stosownych źródeł, na przykład MSDN.

Podsumowanie

Biblioteka Windows GDI jest naprawdę ogromna. Nawet ten, rozpuchnięty do granic możliwości rozdział nie opisuje wszystkich jej elementów. Pozwala jednak poznać te zagadnienia, które są chyba niezbędne do stosowania interfejsu GDI w swoich programach.

Czego zatem zdołałeś się dowiedzieć?...

Najpierw zaprezentowałem ci podstawowe kwestie związane ze współczesną grafiką komputerową. Poznaliśmy więc jej rodzaje, systemy zapisu kolorów oraz typy najważniejszych urządzeń graficznych.

Dalej przedstawiłem fundamenty biblioteki Windows GDI: potok graficzny oraz kontekst urządzenia. W tym podrozdziale zrobiliśmy też krótką wycieczkę po wszystkich elementach interfejsu.

Potem już na poważnie zajęliśmy się samym GDI. Omówiłem po kolei trzy rodzaje prymitywów graficznych: figury geometryczne, bitmapy i tekst. Mogłeś się dowiedzieć, w jaki sposób korzystać z tego bogactwa narzędzi do tworzenia aplikacji wyposażonych w grafikę.

Na koniec przyswoiłeś sobie umiejętność korzystania z regionów w celu rysowania oraz przycinania.

Na tym etapie twoja nauka GDI może się rzecz jasna skończyć i nie będzie to wielką stratą. Chcąc jednak tworzyć bardziej zaawansowane programy dla Windows, będziesz prędzej czy później zmuszony poznać ten interfejs dokładnie. Ponieważ wykracza to poza zagadnienie programowania gier, nie znajdzie sobie miejsca w tym kursie...

Pytania i zadania

Wielki rozdział wymaga równie wielkiej pracy domowej, prawda? ;) Oto więc są zestawy pytań i ćwiczeń do wykonania dla ciebie.

Pytania

1. Wymień dwa rodzaje grafiki komputerowej. Czym charakteryzuje się każdy z nich?
2. Z jakich barw podstawowych korzysta system RGB, a z jakich CMYK?
3. Za co odpowiada kanał alfa w systemie RGB?
4. Ile kolorów wyświetla monitor w trybie *True Color*? Ilu bitów używa wtedy do zapisu pojedynczego piksela w pamięci graficznej?
5. Czym jest rasteryzacja i dlaczego jest ona konieczna?
6. Wymień dwa typy monitorów komputerowych.
7. O czym mówi rozdzielczość obrazu?
8. Co składa się na tryb graficzny, w jakim pracuje monitor?
9. Podaj trzy najpopularniejsze typy drukarek mozaikowych.
10. W jakich dwóch trybach rysowania pracuje Windows GDI?
11. Co określa tryb mapowania?
12. Skąd możemy wziąć kontekst urządzenia przeznaczony do pracy z oknem?
13. Jakie rodzaje prymitywów graficznych oferuje GDI?
14. Jakimi obiektami posługuje się GDI?
15. Co charakteryzuje pióro? Jak je tworzymy?
16. Jakimi sposobami można stworzyć pędzel?
17. Jak działa wypełnianie pędzlem?
18. Jakie rodzaje krzywych otwartych i figur zamkniętych możemy rysować przy pomocy funkcji Windows GDI?
19. Co, oprócz obiektu bitmapy, jest potrzebne do prezentacji obrazka rastrowego w wybranym kontekście urządzenia?
20. Jak można wyświetlić *sprite* bez tła otaczającego jego bitmapę?
21. Podaj dwie funkcje służące do wypisywania tekstu.
22. W jaki sposób zmieniamy kolor tekstu, wyświetlanego w kontekście urządzenia?
23. Jakie dwie funkcje służą do tworzenia czcionek logicznych? Dlaczego żądają tak dużo informacji, które niekoniecznie muszą być przez nie wykorzystywane?
24. Czym są regiony i do czego można je wykorzystać?

Ćwiczenia

1. Zmodyfikuj szkicownik z poprzedniego rozdziału (przykład `Scribble`) tak, aby:
 - a) klawiszami 1-9 z klawiatury alfanumerycznej można było wybierać kolor rysowanych linii.

- b) klawiszami strzałek *w lewo* i *w prawo* możliwy był wybór stylu linii
- c) klawiszami strzałek *w górę* i *w dół* możliwe było określenie grubości linii
- 2. Stwórz program, który po kliknięciu we wnętrzu okna zmienia jego kolor na losowy.
- 3. (**Trudniejsze**) Pobaw się w symulację znanego z LOGO żółwia w Windows. Stwórz klasę `CTurtle`, która będzie reprezentowała ów zacy rodzaj kursora. Określ jej pola i metody. Projektując klasę, pamiętaj, że:
 - a) żółw porusza się zawsze naprzód w ustalonym kierunku. Kształt żółwia na ekranie (zwykle trójkąt) odpowiada temu kierunkowi
 - b) kierunek można zmieniać, każąc żółwiowi obrócić się w jego lewą lub prawą stronę o określoną liczbę stopni
 - c) możliwe jest także polecenie żółwiowi, aby poszedł do określonego punktu. Po jego wykonaniu żółw zachowuje obrany kierunek ruchu
 - d) żółw może poruszać się, rysując lub nie rysując linii, w zależności od aktualnego ustawienia (domyślnie linie są rysowane)Pomyśl też o jakiej formie interakcji z obiektem żółwia. Proponuję jeden z dwóch sposobów:
 - a) wykorzystanie klawiatury. Niech klawisze strzałek *w lewo* i *w prawo* powodują obrót w tych kierunkach w jakimś sensownym tempie. *Strzałka w górę* niech skutkuje ruchem żółwia naprzód, a *Spacja* włączeniem lub wyłączeniem rysowania. Kliknięcie myszą powinno umiejscawiać żółwia w klikniętym punkcie
 - b) (**Trudne**) dodanie do programu pola tekstowego, przeznaczonego na wpisywanie poleceń sterujących (zatwierdzanych klawiszem *Enter*):
 - 1) `fwd odległość` - ruch naprzód o podaną odległość
 - 2) `rotl kąt` - obrót w lewo o podany kąt (w stopniach)
 - 3) `rotr kąt` - obrót w prawo o podany kąt (w stopniach)
 - 4) `draw on/off` - włączenie/wyłączenie rysowania
 - 5) `goto x,y` - wysłanie żółwia w określone miejsce
- 2. (**Bardzo trudne**) Rozbuduj przykład `Bezier` tak, aby możliwe było dodawanie i usuwanie punktów kontrolnych krzywej. Niech kliknięcie prawym przyciskiem myszy powoduje te działania: dodanie punktu, jeżeli kliknięto w wolne miejsce, lub usunięcie punktu, jeżeli kliknięto w już istniejący.
- 3. (**Trudniejsze**) Napisz funkcje `Pentagon()` i `Hexagon()`, rysujące pięciokąt i sześciokąt foremny.
- 4. Poszerz funkcjonalność procedury `EllipticArc()`, zaprezentowanej przy omawianiu łuków elips. Niech nowa funkcja potrafi rysować zarówno łuki, jak też wycinki i odcinki elipsy. Zastanów się, jaki mechanizm rozróżniania tych trzech czynności będzie najlepszy.
- 5. Stwórz klasę ułatwiającą posługiwanie się bitmapami w Windows GDI. Klasa ta powinna ukrywać wszystkie szczegóły związane z wczytywaniem i zwalnianiem bitmap, a na zewnątrz powinna udostępniać:
 - a) metodę pozwalającą wczytać bitmapę z pliku
 - b) metodę umożliwiającą stworzenie pustej bitmapy o podanych wymiarach
 - c) uchwyt do pamięciowego kontekstu urządzenia, aby możliwe było używanie go w dowolnych operacjach graficznych
 - d) wymiary bitmapy
- 4. Napisz program wypisujący tekst w środku obszaru klienta okna o zmiennym rozmiarze.
- 5. (**Trudne**) Stwórz aplikację pokazującą ten sam napis przy użyciu 5 czcionek wybranych losowo spośród wszystkich obecnych w systemie.
- 6. Utwórz okno w kształcie koła, w którym narysujesz koncentryczne, żółto-czerwone kręgi, przypominające tarczę strzelecką.
(**Trudne**) Zapewnij możliwość przesuwania okna poprzez przeciągania za jego obszar klienta (czy raczej to, co z niego zostanie...).

7. (**Bardzo trudne**) Wypisz na Pulpicie tekst, który będzie można przesuwąć, przeciągając go myszą.