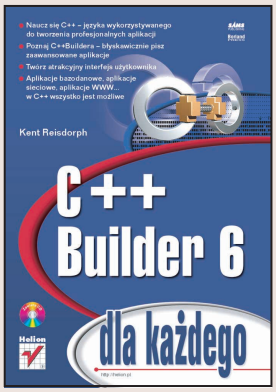
C++ Builder 6 dla każdego

Autor: Kent Reisdorph
Tłumaczenie: Tomasz M. Sadowski
ISBN: 83-7197-859-6
Tytuł oryginału: Sams Teach Yourself Borland C++ Builder 3 in 21 Days
Format: B5, stron: 812



C++Builder należy do systemów błyskawicznego projektowania aplikacji (ang. RAD - Rapid Application Development) i jak sama nazwa wskazuje, przeznaczony jest do tworzenia programów w języku C++. Wykorzystując to narzędzie, możesz efektywnie i szybko konstruować 32-bitowe programy pracujące w trybie graficznym bądź tekstowym pod kontrolą systemu Windows.

Książka ta poprowadzi Cię przez zagadnienia związane z programowaniem w systemie C++ Builder. Poznasz go począwszy od podstaw, poprzez bibliotekę VCL i jej komponenty, narzędzia systemu i metody programowania wizualnego, aż po techniki programowania grafiki, obsługi baz danych i aplikacji internetowych. Dzięki zdobytej wiedzy będziesz mógł błyskawicznie przejść od tradycyjnych metod programowania do projektowania i programowania wizualnego.

Czytając ją:

- Nauczysz się programować w języku C++
- Poznasz programowanie zorientowane obiektowo
- Zapoznasz się z elementami środowiska systemu C++Builder 6 oraz biblioteką VCL
- Zaznajomisz się ze sposobami błyskawicznego prototypowania, budowy i uruchamiania 32-bitowych aplikacji dla Windows 95 i Windows NT
- Nauczysz się tworzyć atrakcyjny interfejs użytkownika do swoich aplikacji
- Zapoznasz się z zaawansowanymi technikami programowania, jak obsługa wyjątków i komunikatów, wykorzystanie rejestru czy udostępnianie poleceń
- Poznasz metody programowania grafiki, obsługi baz danych, tworzenia aplikacji internetowych, własnych komponentów i bibliotek DLL

Autor książki, Kent Reisdorph, kieruje zespołem programistów w firmie TurboPower Software w Colorado Springs. Współpracuje on także z firmą Borland, biorąc udział w pracach grupy doradczej o nazwie TeamB jako niezależny programista i konsultant.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 13.

Programowanie operacji graficznych

Programowanie grafiki należy do milszych elementów rzemiosła programistycznego. Niniejszy rozdział stanowi wprowadzenie do programowania operacji graficznych systemie C++Builder. Większa część rozdziału poświęcona będzie omówieniu klas `TCanvas` i `TBitmap`. Na początek przedstawimy kilka elementarnych metod prezentacji grafiki w aplikacjach, tworzonych w systemie C++Builder, natomiast w dalszej części rozdziału powiemy kilka słów o interfejsie GDI (*Graphics Device Interface*) i jego składnikach. Niejako po drodze zapoznasz się z różnorodnymi metodami wyświetlania linii, figur geometrycznych i map bitowych. Ostatnią część rozdziału poświęcimy wykorzystaniu map bitowych przechowywanych w pamięci.

Grafika minimalnym kosztem

Programowanie grafiki bywa czasem łatwiejsze, niż przypuszczasz. Jeśli np. musisz wyświetlić na formularzu obrazek lub prostą figurę geometryczną, możesz użyć w tym celu odpowiedniego komponentu z biblioteki VCL. Zanim zatem przejdziemy do „poważnego" programowania, przyjrzyjmy się gotowym komponentom i możliwościom ich wykorzystania.

Aby wyświetlić na formularzu prostą figurę geometryczną, możesz wykorzystać komponent `Shape`. Narysowanie figury sprowadza się w takim przypadku do umieszczenia komponentu na formularzu i odpowiedniej zmiany właściwości `Brush`, `Pen` i `Shape`. Sposób ten pozwala na łatwe kreślenie kół, elips, kwadratów, prostokątów i prostokątów z zaokrąglonymi wierzchołkami. Kolor tła figury można modyfikować poprzez zmianę właściwości `Brush`, natomiast za pomocą właściwości `Pen` można zmieniać kolor i grubość linii tworzącej kontur figury.

Prezentację mapy bitowej na formularzu umożliwia komponent `Image`. Komponent ten jest idealnym narzędziem do realizacji wielu operacji graficznych, takich jak np. umieszczenie wzoru tła na formularzu. Wyświetlana przez komponent `Image` mapa bitowa

definiowana jest poprzez właściwość `Picture` typu `TPicture`. W fazie projektowania ustalenie, jaka bitmapa będzie przez komponent wyświetlana, dokonuje się za pomocą inspektora obiektów, natomiast w trakcie pracy programu można to zrobić np. tak:

```
Image1->Picture->Bitmap->LoadFromFile("bkgnd.bmp").
```

Właściwość `Stretch` kontroluje skalowanie mapy bitowej wyświetlanej przez komponent, natomiast za pomocą właściwości `AutoSize` możesz nakazać komponentowi dostosowanie swoich rozmiarów do wielkości bitmapy. Właściwość `Center` umożliwia wyśrodkowanie mapy bitowej w obrębie komponentu.

Na koniec tego punktu warto jeszcze wspomnieć o komponencie `PaintBox`, umożliwiającym rysowanie na ustalonym obszarze (wycinku) formularza. Jedyną godną uwagi właściwością komponentu `PaintBox` jest właściwość `Canvas`, będąca obiektem klasy `TCanvas`. Ponieważ to właśnie `TCanvas` jest klasą umożliwiającą wykonywanie większości operacji graficznych w aplikacjach, tworzonych w systemie C++Builder, jej omówieniu poświęcimy kilka kolejnych stron.

Konteksty urządzeń i klasa TCanvas

Obszar (ang. *canvas* — płótno), na którym rysowane są obiekty, określany jest w terminologii Windows mianem *kontekstu urządzenia* (ang. *device context*). Pojęcie to pojawiło się już w rozdziale 13. podczas omawiania operacji drukowania. Wykorzystując konteksty urządzeń, możesz realizować operacje graficzne m.in.:

- w obszarze roboczym lub ramce okienka,
- na pulpicie Windows,
- w pamięci,
- na drukarce i innych urządzeniach graficznych.

Konteksty urządzeń związane są również z innymi elementami systemu, takimi jak np. menu, ale zagadnienia te są dość złożone, toteż nie będziemy się nimi zajmować, ograniczając się do elementów wymienionych powyżej.

Wykorzystanie kontekstów urządzeń na poziomie Windows API est dość skomplikowane. Na początek takiej operacji musisz uzyskać od systemu aktualny uchwyt kontekstu (ang. *device context handle*). W następnej kolejności należy wybrać odpowiedni obiekt (czyli narzędzie do rysowania), jak np. pióro (ang. *pen*), pędzel (ang. *brush*) czy czcionkę (ang. *font*), po czym nareszcie można przystąpić do rysowania. Po zakończeniu kreślenia musisz usunąć kontekst urządzenia, jednak przedtem konieczne jest usunięcie wszystkich wybranych uprzednio obiektów. Niedopełnienie tego obowiązku spowoduje „zjadanie" pamięci przez aplikację. Jak widać, stosowanie opisanego procesu jest co najmniej kłopotliwe.

Na szczęście biblioteka VCL daje programiście do dyspozycji klasę `TCanvas`, radykalnie ułatwiającą korzystanie z kontekstów urządzeń. Aby nie być gołosłownym, pokażmy,

jak wygląda kod wykreślający w obszarze danego okna czerwone koło o niebieskim konturze. Najpierw wersja wykorzystująca wywołania Windows API:

```
HDC hdc = GetDC(Handle);
HBRUSH hBrush = CreateSolidBrush(RGB(255, 0, 0));
HPEN hPen = CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
HBRUSH oldBrush = SelectObject(hdc, hBrush);
HPEN oldPen = SelectObject(hdc, hPen);
Ellipse(hdc, 20, 20, 120, 120);
SelectObject(hdc, oldBrush);
SelectObject(hdc, oldPen);
DeleteObject(hPen);
DeleteObject(hBrush);
ReleaseDC(Handle, hdc);
```

Powyższy kod wygląda jeszcze dość prosto, ale łatwo tutaj zapomnieć np. o usunięciu utworzonych uprzednio obiektów, czego rezultatem będzie pochłanianie zasobów systemowych przez naszą aplikację. Porównajmy powyższy fragment z jego VCL-owym odpowiednikiem:

```
Canvas->Brush->Color = clRed;
Canvas->Pen->Color = clBlue;
Canvas->Ellipse(20, 20, 120, 120);
```

Kod ten jest nie tylko czytelniejszy, ale także bardziej odporny na błędy — klasa `TCanvas` sama dba o usunięcie przydzielonych sobie zasobów, zwalniając z tego obowiązku programistę. Użycie klas biblioteki VCL jest więc nie tylko prostsze, ale i bezpieczniejsze niż wykorzystanie wywołań Windows API.

Klasa `TCanvas`, reprezentująca obszar, na którym tworzony jest rysunek (inaczej płótno), posiada sporo właściwości i metod. Kilka z nich omówimy dokładniej w dalszej części rozdziału. Podstawowe właściwości klasy `TCanvas` zestawiono w tabeli 13.1, natomiast metody — w tabeli 13.2.

Tabela 13.1. *Podstawowe właściwości klasy TCanvas*

Właściwość	Przeznaczenie
`Brush`	Określa kolor i wzór wypełnienia kreślonych figur
`ClipRect`	Określa prostokąt ograniczający obszar rysowania. Wszystkie rysowane figury będą obcinane na granicach tego obszaru. Właściwość ta może być wyłącznie odczytywana
`CopyMode`	Definiuje operacje bitowe wykorzystywane w trakcie rysowania (kopiowanie, inwersja bitów, suma modulo 2 itd.)
`Font`	Określa krój i parametry czcionki używanej do wyprowadzania tekstu
`Handle`	Zawiera uchwyt kontekstu urządzenia związanego z płótnem
`Pen`	Definiuje kolor i styl linii kreślonych na płótnie
`PenPos`	Zawiera współrzędne aktualnego położenia pióra kreślącego rysunek
`Pixels`	Oznacza tablicę pikseli tworzących płótno

Tabela 13.2. *Podstawowe metody klasy TCanvas*

Metoda	Przeznaczenie
`Arc`	Kreśli łuk okręgu, używając aktualnie wybranego pióra
`BrushCopy`	Wyświetla mapę bitową o przezroczystym tle
`CopyRect`	Kopiuje fragment innego płótna na płótno z którym pracujemy
`Draw`	Kopiuje fragment mapy bitowej na płótno
`Ellipse`	Kreśli elipsę, używając aktualnie wybranego pióra i pędzla (wzoru wypełnienia)
`FloodFill`	Wypełnia obszar płótna zawierający piksel o podanych współrzędnych i ograniczony podanym kolorem. Do wypełnienia takiego obszaru jest używany aktualnie wybrany pędzel
`LineTo`	Kreśli odcinek od bieżącego punktu do punktu o zadanych współrzędnych
`MoveTo`	Ustala współrzędne aktualnego położenia pióra
`Pie`	Kreśli wycinek koła
`Polygon`	Wykorzystując aktualnie wybrane pióro i pędzel, kreśli wielobok o współrzędnych wierzchołków zadanych w postaci tablicy punktów
`Polyline`	Wykorzystując aktualnie wybrane pióro, kreśli linię łamaną o współrzędnych wierzchołków zadanych tablicą punktów Linia nie jest automatycznie domykana
`Rectangle`	Kreśli prostokąt, używając aktualnie wybranego pióra i pędzla
`RoundRect`	Kreśli prostokąt o zaokrąglonych rogach, wypełniony wybranym wzorem
`StretchDraw`	Kopiuje wybrany obszar mapy bitowej na płótno, skalując jej zawartość tak, by wypełnić nią wybrany obszar docelowy
`TextExtent`	Zwraca szerokość i wysokość (w pikselach) łańcucha przekazanego jej jako parametr. Wielkość pola tekstu jest obliczana na podstawie bieżących ustawień czcionki
`TextHeight`	Zwraca wysokość (w pikselach) łańcucha przekazanego jej jako parametr. Wysokość tekstu jest obliczana na bazie bieżących ustawień czcionki
`TextOut`	Wypisuje na płótnie zadany tekst, wykorzystując aktualnie wybraną czcionkę
`TextRect`	Wypisuje tekst w obrębie prostokątnego obszaru ograniczającego

Wymienione wyżej właściwości i metody reprezentują zaledwie małą cząstkę (tak!) możliwości funkcjonalnych kontekstu urządzenia w Windows. Na całe szczęście przytoczony zestaw metodzapewnia obsługę mniej więcej 80 procent wszystkich czynności związanych z programowaniem operacji graficznych. Zanim jednak przejdziemy do szczegółów związanych z klasą `TCanvas`, wypada powiedzieć kilka słów o obiektach wykorzystywanych w programowaniu grafiki w Windows.

Obiekty GDI

W skład interfejsu GDI (ang. *Graphics Device Interface*) systemu Windows wchodzi cały szereg obiektów kontrolujących działanie kontekstów urządzeń. Do najczęściej używanych obiektów GDI należą pióra, pędzle i czcionki; z pozostałych obiektów warto tu wspomnieć o paletach, mapach bitowych i obszarach ograniczających. Na początek przyjrzymy się piórom, pędzlom i czcionkom, natomiast w następnej kolejności omówimy bardziej złożone obiekty.

Pióra, pędzle i czcionki

Obiekty te nie są szczególnie skomplikowane. Ich krótki opis i sposób wykorzystania w klasie `TCanvas` został przedstawiony poniżej.

Pióra

Pióro (ang. *pen*) definiuje obiekt używany do kreślenia linii, przy czym pod pojęciem „linia" rozumiemy zarówno odcinek łączący dwa punkty, jak i krawędź (fragment obrysu) figury (prostokąta, elipsy, wieloboku itp.). Ustawienia pióra możemy kontrolować, korzystając z obiektu klasy `TPen`, dostępnego we właściwości `Pen` klasy `TCanvas`. Właściwości klasy `TPen` zostały opisane w tabeli 13.3; klasa ta nie posiada żadnych zdarzeń ani metod godnych wzmianki.

Tabela 13.3. *Właściwości klasy TPen*

Właściwość	Przeznaczenie
`Color`	Definiuje kolor linii
`Handle`	Określa uchwyt pióra (`HPEN`). Właściwość ta używana jest podczas bezpośrednich wywołań funkcji GDI
`Mode`	Określa rodzaj operacji bitowej wykonywanej podczas kreślenia linii (kopiowanie, inwersja bitów, suma modulo 2 itp.)
`Style`	Określa styl linii (ciągła, przerywana, kropkowa, niewidoczna itp.)
`Width`	Określa szerokość linii w pikselach

Wykorzystanie wymienionych właściwości nie wymaga w zasadzie komentarza. Za pomocą poniższych instrukcji na ekranie zostanie nakreślona linia przerywana w kolorze czerwonym:

```
Canvas->Pen->Color = clRed;
Canvas->Pen->Style = psDash;
Canvas->MoveTo(20, 20);
Canvas->LineTo(120, 120);
```

Aby sprawdzić działanie tych instrukcji, najprościej jest wstawić do formularza przycisk i wpisać powyższy kod w funkcji obsługi zdarzenia `OnClick`. Kliknięcie przycisku spowoduje wykonanie instrukcji, czyli wykreślenie linii w formularzu.

Opisana wyżej metoda może być również użyta dla innych przedstawionych w tym rozdziale przykładów. Zauważ jednak, że przysłonięcie okienka formularza i jego ponowne wyświetlenie powoduje usunięcie utworzonego rysunku. Dzieje się tak dlatego, iż wykreślony rysunek jest tymczasowy i znika w chwili ponownego wyrysowania okienka formularza. Aby „utrwalić" rysunek, powinieneś umieścić tworzące go instrukcje w funkcji obsługi zdarzenia `OnPaint` formularza. W takim przypadku każde żądanie ponownego wyrysowania formularza będzie powodowało odtworzenie rysunku.

Warto zwrócić uwagę, że linie kropkowane i przerywane mogą być kreślone wyłącznie piórem o szerokości 1 piksela. Z kolei styl `psClear`, definiujący linię niewidoczną, pozwala na kreślenie figur pozbawionych konturów.

Aby zbadać właściwości klasy `TPen`, możesz umieścić w formularzu komponent `Shape` i odpowiednio zmodyfikować jego właściwość `Pen`. Trik ten pozwala m.in. na łatwe zbadanie efektów zmiany właściwości `Mode`.

Pędzle

Pędzel (ang. *brush*) odpowiada za wypełnienie wnętrza wykreślonej figury. Każdy kreślony prostokąt, elipsa czy też wielobok będzie automatycznie wypełniany wzorem i kolorem zdefiniowanym przez bieżący pędzel. Warto tu podkreślić, że wypełnienie wcale nie musi być jednolite. Chociaż w wielu przypadkach tak właśnie jest, odpowiednie zdefiniowanie pędzla umożliwia wypełnienie wnętrza figury regularnym wzorem lub mapą bitową.

Do kontrolowania ustawień bieżącego pędzla służy właściwość `Brush` klasy `TCanvas`, będąca obiektem klasy `TBrush`. Podobnie jak w przypadku klasy `TPen`, z naszego punktu widzenia interesujące są jedynie właściwości klasy, wyszczególnione w tabeli 13.4; metody i zdarzenia tej klasy nie mają większego zastosowania.

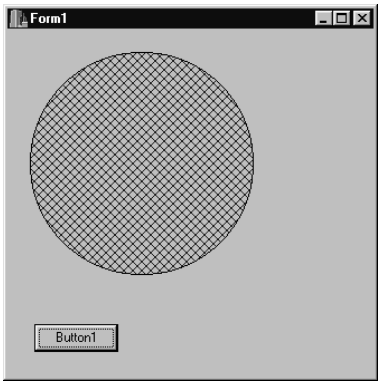
Tabela 13.4. *Właściwości klasy TBrush*

Właściwość	Przeznaczenie
`Bitmap`	Definiuje mapę bitową używaną jako wzór wypełnienia. W systemie Windows 95 maksymalny rozmiar mapy wynosi 8×8 pikseli
`Color`	Definiuje kolor wypełnienia
`Handle`	Określa uchwyt pędzla (`HBRUSH`). Właściwość ta używana jest podczas bezpośrednich wywołań funkcji GDI
`Style`	Określa styl pędzla. Predefiniowane style obejmują m.in. wypełnienie jednolite, brak wypełnienia oraz kilka użytecznych wzorów

Właściwość `Style` przyjmuje domyślnie wartość `bsSolid`, co oznacza wypełnienie jednolitym kolorem. Użycie wypełnienia wymaga zmiany właściwości `Style` na wartość określającą jeden z predefiniowanych wzorów (`bsHorizontal`, `bsVertical`, `bsDiagonal`, `bsCross` i `bsDiagCross`, czyli kreskowanie poziome, pionowe i ukośne oraz kratkowanie i kratkowanie ukośne). Za pomocą trzech przedstawionych poniżej instrukcji można wykreślić w formularzu koło wypełnione wzorem kratkowanym pod kątem 45 stopni. Wynik wykonania tych instrukcji przedstawia rysunek 13.1.

```
Canvas->Brush->Color = clBlue;
Canvas->Brush->Style = bsDiagCross;
Canvas->Ellipse(20, 20, 220, 220);
```

W przypadku wypełnienia figury wzorem, właściwość `Color` pędzla definiuje kolor rysowanych przez niego linii. Z niezbyt jasnych powodów VCL automatycznie używa w takiej sytuacji przezroczystego tła, tak więc tło wypełnienia będzie miało taki sam



Rysunek 13.1. *Koło wypełnione wzorem kratkowanym*

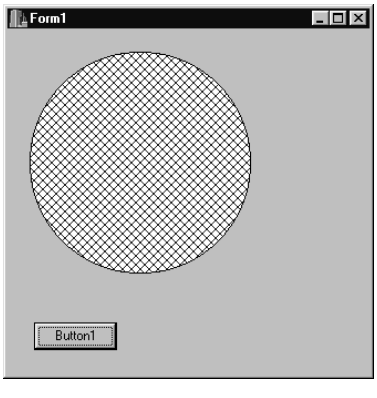
kolor, jak tło okienka, w którym kreślona jest wypełniana figura. Aby się o tym przekonać, rzuć okiem na rysunek 13.1 (zadanie to nieco utrudnia fakt, iż rysunek prezentowany jest w tonacji szarości, jednak uruchomienie programu powinno rozwiać Twoje wątpliwości). Jeśli chcesz jawnie wymusić kolor tła wypełnienia, musisz niestety obejść funkcję VCL przez bezpośrednie odwołanie do funkcji graficznych API. Aby np. wypełnić koło niebieską kratką na białym tle, powinieneś użyć następujących instrukcji:

```
Canvas->Brush->Color = clBlue;
Canvas->Brush->Style = bsDiagCross;
SetBkMode(Canvas->Handle, OPAQUE);
SetBkColor(Canvas->Handle, clWhite);
Canvas->Ellipse(20, 20, 220, 220);
```

Wykonanie tych instrukcji spowoduje pojawienie się koła wypełnionego niebieską kratką na białym tle, co zilustrowano na rysunku 13.2.

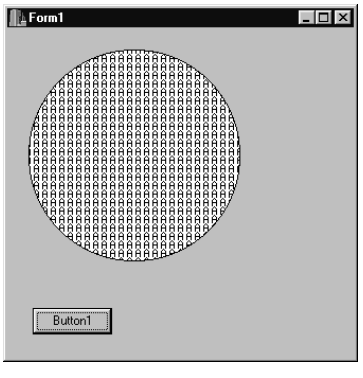
Inną ciekawą funkcją pędzlajest możliwość zdefiniowania wzoru wypełnienia za pomocą mapy bitowej. Zanim to zagadnienie zostanie omówione, przyjrzyjmy się przykładowi:

```
Canvas->Brush->Bitmap = new Graphics::TBitmap;
Canvas->Brush->Bitmap->LoadFromFile("tlo.bmp");
Canvas->Ellipse(20, 20, 220, 220);
delete Canvas->Brush->Bitmap;
```



Rysunek 13.2. *Tym razem wzór wypełnienia ma białe tło*

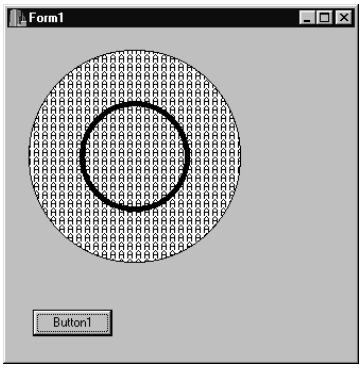
Pierwsza instrukcja tworzy obiekt klasy `TBitmap` i przypisuje go do właściwości `Bitmap` pędzla. Musisz pamiętać, że ta ostatnia nie posiada żadnej wartości domyślnej, toteż chcąc ją wykorzystać, musisz jawnie utworzyć obiekt klasy `TBitmap` i wykonać odpowiednie przypisanie. W drugim wierszu przykładu odczytujemy mapę bitową z pliku. Mapa taka musi mieć rozmiary 8×8 pikseli (możliwe jest użycie większej mapy, jednak zostanie ona obcięta do rozmiarów 8×8). Samo wykreślenie koła (elipsy) realizowane jest w wierszu trzecim, natomiast ostatnia instrukcja usuwa zawartość właściwości `Bitmap`. Ponieważ VCL nie zapewnia automatycznego usunięcia bloku pamięci, wskazywanego przez właściwość `Bitmap` pędzla, operację tę musisz wykonać jawnie, w przeciwnym przypadku bowiem program będzie nieodwracalnie zawłaszczał pamięć. Efekt wykonania naszego przykładu ilustruje rysunek 13.3.



Rysunek 13.3. *Użycie wzorca wypełnienia zdefiniowanego za pomocą mapy bitowej*

W niektórych przypadkach konieczne okazuje się użycie wypełnienia pustego (przezroczystego), nie przesłaniającego tła, na którym kreślona jest figura (rysunek 13.4). W takiej sytuacji musisz ustawić właściwość `Style` pędzla na `bsClear`. W ramach ilustracji dodajmy do rysunku, utworzonego w poprzednim przykładzie, jeszcze jedno koło, umieszczone w środku pierwszego i wypełnione z użyciem stylu `bsClear`:

```
Canvas->Pen->Width = 1;
Canvas->Brush->Bitmap = new Graphics::TBitmap;
Canvas->Brush->Bitmap->LoadFromFile("tlo.bmp");
Canvas->Ellipse(20, 20, 220, 220);
Canvas->Brush->Style = bsClear;
Canvas->Pen->Width = 5;
Canvas->Ellipse(70, 70, 170, 170);
delete Canvas->Brush->Bitmap;
```



Rysunek 13.4. *Koło o wypełnieniu przezroczystym*

Bezpośrednie odwołania do funkcji Windows API umożliwiają realizację bardziej wymyślnych trików, jednakże możliwości udostępniane przez VCL-ową klasę `TBrush` okazują się wystarczające w większości typowych zastosowań.

Czcionki

Pojęcie czcionki (ang. *font*) jest Ci doskonale znane — na kartach tej książki pojawiało się wielokrotnie. Czcionki używane przez klasę `TCanvas` nie różnią się niczym od czcionek wykorzystywanych na normalnych formularzach, natomiast właściwość `Font` tej klasy jest identyczna, jak w przypadku innych klas. Zmiana kroju lub parametrów czcionki używanej w operacjach graficznych może wyglądać np. tak:

```
Canvas->Font->Name = "Courier New CE";
Canvas->Font->Size = 14;
Canvas->Font->Style = Canvas->Font->Style << fsBold;
Canvas->TextOut(20, 20, "Malowanie po ekranie");
```

I to praktycznie wszystko. Wykorzystaniem czcionek zajmiemy się wkrótce, w podrozdziale pt. „Rysowanie tekstu”.

Mapy bitowe i palety

Pojęcia map bitowych i palet jeszcze kilka lat temu były związane ze sobą dość ściśle i z reguły występowały łącznie. Obiekty klasy `TBitmap` w systemie C++Builder pozwalają na proste korzystanie z map bitowych (ang. *bitmap*) i łatwą realizację związanych z nimi operacji, takich jak np. odczyt z dysku i wyświetlanie. Klasę `TBitmap` miałeś już okazję zastosować w programie Jumping Jack, o którym wspominano w rozdziale 9. Zastosowanie klasy `TBitmap` jest bardzo szerokie i w pewnym stopniu zostanie omówione w dalszej części rozdziału, przy okazji dyskusji rysowania map bitowych i użycia map przechowywanych w pamięci. Ze względu na sporą złożoność klasy `TBitmap` nie będziemy w tym miejscu zajmować się szczegółowo jej właściwościami i metodami.

Palety (ang. *palette*) są „urządzeniami” dość kłopotliwymi w użyciu i zrozumieniu. Na szczęście dzięki szybkiemu postępowi technologii w dziedzinie budowy układów kart graficznych i dzięki spadkowi cen, w typowych, najczęściej dziś spotykanych konfiguracjach komputerów palety są praktycznie niewykorzystywane.

Nawet jeśli będziecie musieli uwzględnić palety w swojej aplikacji, w większości przypadków będziecie mogli skorzystać z możliwości klasy `TBitmap`, która zazwyczaj potrafi obsługiwać palety w pełni automatycznie. Zamiast wdawać się w rozważania teoretyczne, spróbujemy zademonstrować rolę palety na przykładzie. W tym celu utwórz nową aplikację i wpisz w funkcji obsługi zdarzenia `OnPaint` formularza (lub `OnClick` przycisku) kod przytoczony poniżej. Pamiętaj, by w razie potrzeby podać właściwą ścieżkę dostępu do pliku obrazka *HANDSHAK.BMP* (powinien on znajdować się w katalogu *Borland Shared\Images\Splash\256Color*). A oto i sam przykład:

```
Graphics::TBitmap* bitmap = new Graphics::TBitmap;
//bitmap->IgnorePalette = true;
bitmap->LoadFromFile("handshak.bmp");
Canvas->Draw(0, 0, bitmap);
delete bitmap;
```

Po uruchomieniu programu w 256-kolorowym trybie graficznym powinieneś zobaczyć w formularzu ładny obrazek. Zwróć jednak uwagę, że druga z kolei instrukcja została zamieniona w komentarz; jeśli usuniesz znaki komentarza i uruchomisz program ponownie, okaże się, że kolory wyświetlonej mapy bitowej w niczym nie przypominają oryginału. Efekt ten jest wynikiem zignorowania informacji zawartych w palecie kolorów podczas wyświetlania mapy bitowej. Dopiero użycie tych informacji gwarantuje poprawne odwzorowanie kolorów, zdefiniowanych w pliku, na paletę systemową.

Mapy bitowe są bardzo istotnymi elementami programowania operacji graficznych, tym niemniej zrozumienie ich funkcjonowania nie jest sprawą łatwą i nie powinieneś się przejmować, jeśli pewne pojęcia wydadzą Ci się z początku niejasne. Więcej informacji na temat działania map bitowych i przykładów, wykorzystujących takie mapy, znajdziesz w dalszej części rozdziału.

Ograniczanie obszaru rysowania

W celu ograniczenia obszaru przeznaczonego do rysowania możesz zdefiniować odpowiedni *obszar ograniczający* (ang. *clipping region*). Niestety, właściwość `Region` klasy `TCanvas` daje się wyłącznie odczytywać, toteż definiując obszar ograniczający musisz odwołać się bezpośrednio do funkcji Windows API. Aby zademonstrować opisywaną technikę, wykorzystamy poprzedni przykład, nieco zmodyfikowany:

```
Graphics::TBitmap* bitmap = new Graphics::TBitmap;
bitmap->LoadFromFile("handshak.bmp");
HRGN hRgn = CreateRectRgn(50, 50, 200, 200);
SelectClipRgn(Canvas->Handle, hRgn);
Canvas->Draw(0, 0, bitmap);
delete bitmap;
```

Po uruchomieniu programu zawierającego (w metodzie obsługi odpowiedniego zdarzenia) powyższy kod powinieneś ujrzeć w okienku tylko fragment mapy bitowej. Wywołanie funkcji `SelectClipRgn()` tworzy i aktywuje na rysunku prostokątny obszar ograniczający miejsce dostępne dla operacji rysowania obiektów graficznych o współrzędnych wierzchołków (50, 50) i (200, 200). Co prawda, mapa bitowa jest nadal wyświetlana tam, gdzie poprzednio, jednak w wyniku nałożenia na nią obszaru ograniczającego, zewnętrzne jej fragmenty zostają obcięte, a widoczna pozostaje wyłącznie część środkowa.

Obszar ograniczający wcale nie musi mieć kształtu prostokąta. Aby uatrakcyjnić poprzedni przykład, zastąp wywołanie funkcji `CreateRectRgn()` wierszem o następującej treści:

```
HRGN hRgn = CreateEllipticRgn(30, 30, 170, 170);
```

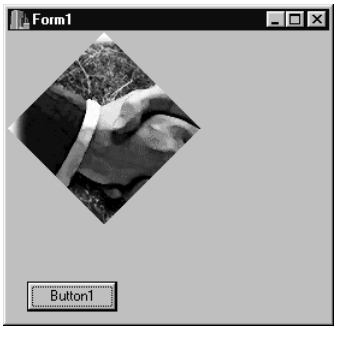
Po uruchomieniu programu wyświetlany w okienku fragment mapy bitowej powinien mieć kształt koła (rysunek 13.5).

Spróbujmy jeszcze czegoś innego. Zastąp definicję obszaru kolistego (eliptycznego) następującą parą instrukcji:

```
TPoint points[4] = {{80, 0}, {0, 80}, {80, 160}, {160, 80}};
HRGN hRgn = CreatePolygonRgn(points, 4, ALTERNATE);
```

Rysunek 13.5.
Efekt użycia eliptycznego obszaru ograniczającego

W tym przypadku definiowany obszar będzie miał kształt wieloboku, którego wierzchołki opisane są przez współrzędne zawarte w tablicy `points`. Sam obszar tworzony będzie na podstawie zawartych w tablicy danych przez funkcję `CreatePolygonRgn()`. Liczba wierzchołków może być dowolna, a określanie punktu zamykającego wielobok jest zbędne (domknięcie obrysu dokonywane jest automatycznie). Po uruchomieniu programu powinieneś uzyskać efekt zilustrowany na rysunku 13.6.



Rysunek 13.6.
Obszar ograniczający o kształcie wieloboku

Obszary ograniczające bywają bardzo przydatne w realizacji niektórych operacji graficznych. Co prawda nie używa się ich zbyt często, jednak czasami okazują się niezastąpione.

Podstawowe operacje graficzne

Z kilkoma procedurami kreślenia elementarnych figur geometrycznych miałeś już okazję się spotkać w poprzednich rozdziałach tej książki. Funkcje, z którymi do tej pory miałeś do czynienia, realizują kreślenie prostokątów i kwadratów (`Rectangle()`), kół i elips (`Ellipse()`) oraz linii (`MoveTo()`, `LineTo()`). Z pozostałych, najczęściej wykorzystywanych funkcji można jeszcze wymienić `Arc()` oraz `Pie()`, rysujące odpowiednio łuk okręgu oraz wycinek koła. Kreślenie prostych figur geometrycznych jest sprawą elementarną, toteż darujemy sobie tutaj szczegółowe omawianie odpowiednich metod klasy `TCanvas`, prezentując w zamian inne, nieco bardziej złożone (a czasem nawet kłopotliwe) operacje, z którymi możesz mieć do czynienia podczas projektowania aplikacji graficznych.

Rysowanie tekstu

Zagadnienie to wygląda na pozór banalnie, jednak z pisaniem tekstu wiąże się kilka trików i haczyków, których nieznajomość może poważnie skomplikować życie programisty. Na szczęście funkcje służące do rysowania tekstu mogą Ci znacznie ułatwić tę pracę.

Metody TextOut() i TextRect()

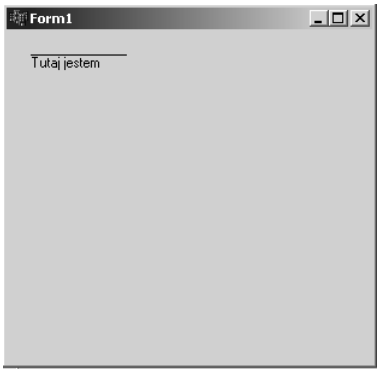
Metoda `TextOut()` jest najprostszym narzędziem do kreślenia napisów w okienku. W zasadzie nie wymaga ona specjalnego omawiania — argumentami jej wywołania są współrzędne X i Y rysowanego tekstu oraz, oczywiście, on sam. Wygląda to tak:

```
Canvas->TextOut(20, 20, "Komputerowe Graffiti");
```

Powyższa instrukcja wyświetla zadany łańcuch, poczynając od punktu o współrzędnych (`20`, `20`), przy czym wartości te określają położenie lewego górnego (a nie dolnego) wierzchołka obszaru zajętego przez tekst. Aby zilustrować wykorzystanie współrzędnych, wprowadź i wykonaj następujące instrukcje:

```
Canvas->TextOut(20, 20, "Tutaj jestem");
Canvas->MoveTo(20, 20);
Canvas->LineTo(100, 20);
```

Po napisaniu zadanego tekstu w punkcie o współrzędnych (`20`, `20`), w naszym programiku narysowany zostanie poziomy odcinek, rozpoczynający się w tym samym punkcie. Jak przedstawiono na rysunku 13.7, efektem wykonania przykładu będzie umieszczenie poziomej kreski *nad* wypisanym w okienku tekstem.



Rysunek 13.7. *Tekst utworzony za pomocą funkcji TextOut()*

Metoda `TextOut()` sprawdza się w sytuacjach, gdy wielkość obszaru zajmowanego przez tekst nie jest sprawą krytyczną. Jeśli tekst musi zmieścić się w określonym obszarze, lepiej jest wykorzystać metodę `TextRect()`, pozwalającą na zdefiniowanie prostokąta ograniczającego obszar napisu. W takiej sytuacji fragment tekstu, wykraczający poza zdefiniowany obszar, zostanie obcięty. Działanie metody `TextRect()` ilustruje poniższy przykład, w którym obszar, zajęty przez tekst, ograniczono do szerokości 100 pikseli:

```
Canvas->TextRect(Rect(20, 50, 120, 70), 20, 50, "Stoi na stacji lokomotywa...");
```

Metoda `TextOut()`, podobnie jak `TextRect()`, pozwala na tworzenie pojedynczych wierszy tekstu. Zbyt długie wiersze nie są automatycznie łamane.

Tekst, zawierający znaki tabulacji, rysuje się za pomocą funkcji Windows API `TabbedTextOut()`.

Tło napisu

Jeśli rzucisz okiem na rysunek 13.7, przekonasz się, że tekst napisany został na tle, które ma kolor taki, jak kolor tła okna. Kolor tła tekstu ustalany jest na podstawie bieżących ustawień pędzla (domyślnie specyfikujących tło w kolorze okna). Aby wpłynąć na kolor tła dla rysowanych tekstów, należy zmienić kolor zdefiniowany dla pędzla (można też uczynić tło tekstu przezroczystym).

Aby wykorzystać tło przezroczyste, należy posłużyć się własnością `Style` komponentu `TBrush` — ilustrują to poniższe instrukcje:

```
TBrushStyle oldStyle;
oldStyle = Canvas->Brush->Style;
Canvas->Brush->Style = bsClear;
Canvas->TextOut(20, 5, "Napis ćwiczebny");
Canvas->Brush->Style = oldStyle;
```

Po zachowaniu bieżącego ustawienia stylu pędzla możemy nadać mu wartość `bsClear`, definiującą tło przezroczyste. Po wyświetleniu napisu należy oczywiście przywrócić pierwotne ustawienie tła (stylu pędzla). Zachowanie oryginalnego ustawienia stylu przed jego zmianą i przywracanie go po napisaniu tekstu jest rutynową czynnością, którą powinieneś sobie szybko przyswoić. Użycie przezroczystego tła w kolejnych operacjach graficznych jest raczej mało prawdopodobne, toteż najlepiej przywrócić pierwotny styl pędzla niezależnie od dalszych jego modyfikacji.

Wykorzystanie przezroczystego tła ma jeszcze jedną zaletę. Wyobraź sobie, że chcesz wyświetlić tekst nałożony na mapę bitową. W takim przypadku z powodów estetycznych jednolite tło tekstu raczej nie jest stosowane, można natomiast wykorzystać tło przezroczyste. Opisywaną sytuację ilustruje poniższy przykład (plik *FACTORY.BMP* znajdziesz w katalogu *Borland Shared\Images\Splash\256Color*).

```
Graphics::TBitmap* bitmap = new Graphics::TBitmap;
bitmap->LoadFromFile("factory.bmp");
Canvas->Draw(0, 0, bitmap);
Canvas->Font->Name = "Arial CE Bold";
Canvas->Font->Size = 13;
TBrushStyle oldStyle;
oldStyle = Canvas->Brush->Style;
Canvas->Brush->Style = bsClear;
Canvas->TextOut(20, 5, "Tło przezroczyste");
Canvas->Brush->Style = oldStyle;
Canvas->Brush->Color = clWhite;
Canvas->TextOut(20, 30, "Tło nieprzezroczyste");
delete bitmap;
```

Pierwsza grupa instrukcji w powyższym przykładzie umieszcza w okienku mapę bitową, natomiast kolejne wiersze służą do utworzenia na jej tle tekstu, najpierw z wykorzystaniem tła przezroczystego, a następnie tła standardowego o kolorze tła okna. Efekt wykonania powyższego programu zilustrowano na rysunku 13.8. Nie trzeba chyba nikogo przekonywać, że tekst umieszczony na przezroczystym tle wygląda o wiele lepiej.

Rysunek 13.8. *Tekst nałożony na mapę bitową z użyciem tła przezroczystegoi nieprzezroczystego*

Inne zastosowanie przezroczystego tła podczas tworzenia tekstu opisano w rozdziale 13. (w punkcie pt. „Panele rysowane przez właściciela"). Być może pamiętasz, że trójwymiarowy efekt wyświetlania tekstu na pasku statusowym został osiągnięty przez narysowanie napisu w kolorze białym, a następnie — z lekkim przesunięciem — szarym. Uzyskanie zamierzonego wyglądu nie byłoby przy tym możliwe bez użycia przezroczystego tła. Wynika z tego, że omówiony mechanizm okazuje się czasem niezbędnym i jedynym sposobem uzyskania planowanego efektu wizualnego.

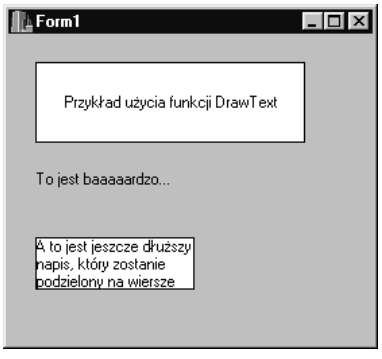
Funkcja DrawText()

Dostępna w zestawie Windows API funkcja `DrawText()` oferuje programiście znacznie większe możliwości wykreślania tekstu w okienku, aniżeli omówiona wcześniej metoda `TextOut()`. Z niewiadomych przyczyn klasa `TCanvas` nie implementuje metody będącej odpowiednikiem `DrawText()`, toteż aby wykorzystać wszystkie zalety tej ostatniej, musisz odwołać się bezpośrednio do API systemu Windows. Nie jest to specjalnie trudne, ale wiąże się z kilkoma niedogodnościami, z którymi, niestety, trzeba się pogodzić. Zanim jednak przejdziemy do szczegółowego omówienia możliwości funkcji `DrawText()`, zaprezentujmy mały przykład:

```
RECT rect = Rect(20, 20, 220, 80);
Canvas->Brush->Color = clWhite;
Canvas->Rectangle(20, 20, 220, 80);
DrawText(Canvas->Handle, "Przykład użycia funkcji DrawText",
  -1, &rect, DT_SINGLELINE | DT_VCENTER | DT_CENTER);
```

Wynik wykonania powyższych instrukcji (wraz z instrukcjami z dwóch kolejnych przykładów) przedstawiony został na rysunku 13.9.

Pierwsza, instrukcja z przedstawionego wyżej przykładutworzy strukturę typu `RECT`, i inicjalizuje ją wartością zwracaną przez funkcję `Rect()` Windows API. Zdefiniowany w ten sposób prostokąt jest następnie rysowany, co pozwala na wizualizację obszaru, w którym zostanie umieszczony tekst. Ta ostatnia czynność realizowana jest za pomocą wywołania funkcji `DrawText()`, której obecnie poświęcimy kilka słów.



Rysunek 13.9. *Kilka zastosowań funkcji DrawText()*

Pierwszy parametr funkcji `DrawText()` określa kontekst urządzenia, na którym zostanie narysowany tekst. W przypadku obiektu klasy `TCanvas` uchwyt odpowiadającego mu kontekstu określa właściwość `Handle`, toteż właśnie jej zawartość przekazujemy jako pierwszy parametr wywołania `DrawText()`. Drugim parametrem jest wyświetlany łańcuch, natomiast liczba wyświetlanych znaków określana jest przez trzeci parametr (nadanie mu wartości –1 powoduje wyświetlenie wszystkich znaków łańcucha). Czwarty parametr funkcji `DrawText()` jest wskaźnikiem struktury typu `RECT`; przekazywanie tego parametru przez wskaźnik jest konieczne, gdyż w niektórych przypadkach może on zostać zmodyfikowany w wyniku wywołania funkcji.

Istotę programu stanowi ostatni parametr funkcji, przeznaczony do przekazywania znaczników kontrolujących sposób rysowania tekstu. W cytowanym wyżej przykładzie użyliśmy opcji `DT_SINGLELINE`, `DT_VCENTER` i `DT_CENTER`, nakazując w ten sposób systemowi narysowanie pojedynczego wiersza tekstu wyśrodkowanego w pionie i w poziomie. Łączna liczba opcji, których można używać dla funkcji `DrawText()` wynosi około 20; nie będziemy omawiać ich wszystkich, odsyłając zainteresowanych Czytelników do systemu pomocy Win32 API.

Przytoczony wyżej przykład ilustruje najbardziej typowe zastosowanie funkcji `DrawText()`, sprowadzające się do rysownaia odpowiednio wyśrodkowanego tekstu. Możliwość taka jest niezwykle przydatna podczas tworzenia komponentów rysowanych przez właściciela, szczególnie przy tworzeniu list zwykłych, rozwijanych i menu, w których nader często wykorzystywane jest środkowanie tekstu w jednej lub obu osiach. Być może wszystko to brzmi w tej chwili dość gołosłownie, ale gdy tylko zaczniesz posługiwać się komponentami rysowanymi przez właściciela, z pewnością docenisz zalety funkcji `DrawText()`.

Innym godnym uwagi znacznikiem używanym przez funkcję `DrawText()` jest `DT_END_ELLIPSIS`. Jeśli wyświetlany tekst jest zbyt długi, by mógł się w całości zmieścić w zadanym prostokącie, jego końcówka jest obcinana i zastępowana znakiem wielokropka. Przykładowo, wykonanie poniższych instrukcji:

```
rect = Rect(20, 100, 120, 150);
DrawText(Canvas->Handle, "To jest baaaaardzo długi napis",
  -1, &rect, DT_END_ELLIPSIS | DT_MODIFYSTRING);
```

spowoduje wyświetlenie tekstu

```
To jest baaaaardzo...
```

Jeśli przewidujesz, że tekst może nie zmieścić się w przeznaczonym dla niego polu, możesz użyć znacznika DT_END_ELLIPSIS, aby uniknąć obcięcia tekstu w połowie wyrazu.

Kolejnym bardzo użytecznym znacznikiem jest DT_CALCRECT, którego użycie nakazuje funkcji DrawText() określenie wielkości prostokątnego obszaru potrzebnego do wyświetlenia całości tekstu. W przypadku użycia znacznika DT_CALCRECT wywołanie naszej funkcji nie spowoduje wyświetlenia tekstu, a jedynie modyfikację współrzędnych, przekazanych w czwartym parametrze. Efektem takiego wywołania będzie modyfikacja pól bottom i right struktury typu RECT przekazywanej jako parametr. Rozwiązanie takie pozwala na narysowanie tekstu z podziałem na wiersze, czego przykład został zademonstrowany poniżej. Działanie przedstawionych instrukcji sprowadza się do ustalenia wysokości pola tekstu (szerokość jest ustalona), wyrysowania prostokąta o współrzędnych ustalonych przez funkcję DrawText() i narysowania tekstu wewnątrz prostokąta. A oto i sam kod:

```
rect = Rect(20, 150, 150, 200);
String S = "A to jest jeszcze dłuższy napis, który zostanie podzielony na wiersze";
DrawText(Canvas->Handle, S.c_str(),
  -1, &rect, DT_CALCRECT | DT_WORDBREAK);
Canvas->Brush->Style = bsSolid;
Canvas->Rectangle(rect.left, rect.top, rect.right, rect.bottom);
Canvas->Brush->Style = bsClear;
DrawText(Canvas->Handle, S.c_str(), -1, &rect, DT_WORDBREAK);
```

Zwróć uwagę na sposób przekazywania tekstu do funkcji DrawText(): konieczność użycia formy S.c_str() wynika z faktu, iż drugim parametrem funkcji musi być wskaźnik tablicy znaków (obiekt typu char*), natomiast w naszym przykładzie tekst przechowywany jest w obiekcie typu AnsiString. Konwersja zawartości obiektu na wskaźnik znakowy char* dokonywana jest właśnie za pomocą metody c_str() klasy AnsiString.

Spróbuj teraz umieścić powyższy fragment kodu w funkcji obsługi zdarzenia OnPaint() formularza i dokonać kilku doświadczeń z tak utworzonym programem, zmieniając zawartość (czyli długość) pisanego tekstu. Jak się przekonasz, wielkość prostokątnego pola, zawierającego napis, będzie za każdym razem dokładnie dostosowywana do aktualnej długości napisu. Wynik wykonania naszego przykładu znajdziesz na rysunku 13.9.

Jeśli możliwości funkcji DrawText() okażą się dla Ciebie niewystarczające, możesz odwołać się do funkcji DrawTextEx(), której omówienie znajdziesz w pliku pomocy opisującym funkcje Win32 API.

Rysowanie tekstu z użyciem funkcji DrawText() jest nieco wolniejsze, niż w przypadku użycia metody TextOut(). Jeśli zależy Ci na czasie wykonania programu, powinieneś odwoływać się raczej do metody TextOut(); co prawda będziesz musiał zaprogramować nieco więcej operacji samodzielnie, jednak z punktu widzenia optymalizacji algorytmu rozwiązanie takie powinno być korzystniejsze. W większości typowych przypadków różnice czasowe pomiędzy obiema funkcjami są do pominięcia.

Funkcja `DrawText()` jest bez wątpienia narzędziem uniwersalnym, a tym samym — bardzo użytecznym, toteż prawdopodobnie będziesz ją często wykorzystywał w swoich programach.

Rysowanie map bitowych

Temat wygląda groźnie, ale na szczęście — jak miałeś już okazję kilkakrotnie się przekonać — rysowanie map bitowych nie jest wcale zadaniem trudnym. Klasa `TCanvas` oferuje Ci szereg związanych z tym zagadnieniem funkcji, z których najczęściej używaną jest metoda `Draw()`. Jej działanie sprowadza się do umieszczenia zawartości mapy bitowej (reprezentowanej przez obiekt klasy `TBitmap`, pochodnej od `TGraphic`) w zadanym miejscu płótna. Na poprzednich stronach widziałeś już kilka przykładów tej operacji, ale jeden więcej nie zaszkodzi:

```
Graphics::TBitmap* bitmap = new Graphics::TBitmap;
bitmap->LoadFromFile("c:\\windows\\forest.bmp");
Canvas->Draw(0, 0, bitmap);
delete bitmap;
```

Cytowane wyżej instrukcje tworzą obiekt klasy `TBitmap`, umieszczają w nim mapę bitową odczytaną z pliku *FOREST.BMP* i wyświetla w okienku aplikacji (ściślej — w jego lewym górnym rogu).

O ile metoda `Draw()` używana jest do rysowania map bitowych bez dodatkowych modyfikacji, użycie metody `StretchDraw()` umożliwia przeskalowanie wyświetlanej mapy do zadanych wymiarów. Jeśli prostokąt definiujący obszar docelowy jest większy od oryginalnego rozmiaru mapy, ta ostatnia zostanie powiększona (rozciągnięta); jeśli jest on mniejszy — mapa zostanie pomniejszona tak, by dopasować się do zadanej wielkości obszaru docelowego. Oto przykład:

```
Graphics::TBitmap* bitmap = new Graphics::TBitmap;
bitmap->LoadFromFile("c:\\windows\\forest.bmp");
TRect rect = Rect(0, 0, 100, 150);
Canvas->StretchDraw(rect, bitmap);
delete bitmap;
```

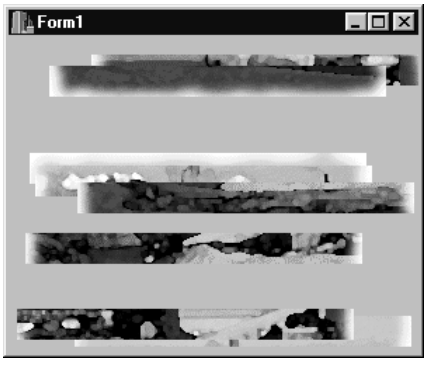
Skalując mapę bitową metoda `StretchDraw()` nie zachowuje oryginalnych proporcji. W razie potrzeby musisz zadbać o to sam.

Kolejną funkcją obsługującą wyświetlanie map bitowych jest `CopyRect()`, umożliwiająca zdefiniowanie zarówno obszaru docelowego, jak i źródłowego, tj. fragmentu mapy, który zostanie skopiowany. Pozwala to np. na podział wyświetlanego obrazka na mniejsze prostokątne fragmenty, czego odzwierciedlenie znajdziemy w poniższym przykładzie:

```
Graphics::TBitmap* bitmap = new Graphics::TBitmap;
bitmap->LoadFromFile("factory.bmp");
const int strips = 8;
int stripSize = bitmap->Height/strips;
for(int i = 0; i < strips; i++)
```

```
{
  TRect src = Rect(0, i*stripSize, bitmap->Width, (i*stripSize+stripSize));
  int x = random(Width - bitmap->Width);
  int y = random(Height - stripSize);
  TRect dst = Rect(x, y, x + bitmap->Width, y + stripSize);
  Canvas->CopyRect(dst, bitmap->Canvas, src);
}
delete bitmap;
```

Przedstawione wyżej instrukcje odczytują mapę bitową z pliku i dzielą ją na podłużne pasy, które następnie zostają rozmieszczone w losowo wybranych miejscach okienka. Przykładowy wynik wykonania programu pokazano na rysunku 13.10. Jeśli umieścisz powyższy blok instrukcji w funkcji obsługi zdarzenia `OnPaint()` formularza, zauważysz, że układ pasków zmienia się każdorazowo po przesłonięciu okienka pracującej aplikacji i jego ponownym odsłonięciu. Wynika to z faktu, że cała procedura wykonywana jest każdorazowo w chwili powstania konieczności ponownego wyrysowania zawartości okienka.



Rysunek 13.10. *Fragmenty mapy bitowej rozłożone w przypadkowych miejscach okienka za pomocą funkcji CopyRect()*

Ostatnią godną wzmianki metodą rysowania mapy bitowej jest `BrushCopy()`. Metoda ta umożliwia zdefiniowanie prostokątnego obszaru źródłowego i docelowego kopiowanej mapy bitowej oraz wybranie koloru, który będzie traktowany jako przezroczysty. Co prawda, zawarty w pliku pomocy opis metody `BrushCopy()` zaleca użycie zamiast niej komponentu typu `ImageList`, jednak w wielu przypadkach okazuje się to przesadą — `BrushCopy()` spełnia swoje zadanie równie dobrze, a jej wykorzystanie jest znacznie łatwiejsze. O metodzie tej powinieneś pamiętać zwłaszcza podczas operowania na mapach bitowych używających przezroczystego tła. W części rozdziału, poświęconej bitmapom typu DIB, przyjrzymy się dokładniej funkcjonalności klasy `TBitmap`, pozwalającej na stosowanie bitmap przezroczystych.