Rysowanie na formularzu

Widzieliśmy, że możliwe jest malowanie bezpośrednio na powierzchni formularza w odpowiedzi na zdarzenie myszy. Aby zobaczyć to zachowanie, po prostu utwórz nowy formularz za pomocą następującej procedury obsługi zdarzenia OnMouseDown:

```
procedure TForm1.FormMouseDown(Sender: TObject;

Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

begin

Canvas.Ellipse (X-10, Y-10, X+10, Y+10);

end;
```

Program wydaje się działać dość dobrze, ale tak nie jest. Każde kliknięcie tworzy nowe koło, ale jeśli zminimalizujesz formularz, wszystkie one znikną. Nawet jeśli pokryjesz część formularza innym oknem, kształty za tą inną formą znikną, a możesz skończyć z częściowo pomalowanymi kółkami. Ten bezpośredni rysunek nie jest automatycznie obsługiwany przez system Windows. Standardowym podejściem jest przechowywanie żądania malowania w zdarzeniu OnMouse-Down, a następnie odtwarzanie danych wyjściowych w zdarzeniu OnPaint. To zdarzenie jest w rzeczywistości wywoływane przez system za każdym razem, gdy forma wymaga odświeżenia. Musisz jednak wymusić jego aktywację, wywołując metody Invalidate or Repaint w module obsługi zdarzeń myszy. Innymi słowy, system Windows wie, kiedy formularz musi zostać przemalowany z powodu działania systemu (np. Umieszczenie innego okna przed formularzem), ale program musi powiadomić system, gdy malowanie jest wymagane z powodu wprowadzania danych przez użytkownika lub innych operacji programu .

Narzędzia do rysowania

Wszystkie operacje wyjściowe w Windows odbywają się przy użyciu obiektów klasy TCanvas. Operacje wyjściowe zwykle nie określają kolorów i podobnych elementów, ale korzystają z bieżących narzędzi do rysowania obszaru roboczego. Oto lista tych narzędzi do rysowania (lub obiektów GDI, z interfejsu urządzenia graficznego, który jest jedną z bibliotek systemu Windows):

• Właściwość Brush określa kolor zamkniętych powierzchni. Szczotka służy do wypełniania zamkniętych kształtów, takich jak okręgi lub prostokąty. Właściwości pędzla to Kolor, Styl i opcjonalnie Bitmapa.

• Właściwość Pen określa kolor i rozmiar linii oraz granic kształtów. Właściwości pisaka to Kolor, Szerokość i Styl, który zawiera kilka linii przerywanych i przerywanych (dostępne tylko, jeśli szerokość to 1 piksel). Inną istotną podwłasnością pisaka jest właściwość Tryb, która wskazuje, w jaki sposób kolor pióra modyfikuje kolor powierzchni rysunkowej. Domyślnie jest to po prostu użycie koloru pisaka (ze stylem pmCopy), ale możliwe jest również scalenie dwóch kolorów na wiele różnych sposobów i odwrócenie bieżącego koloru powierzchni rysunkowej.

• Właściwość Font określa czcionkę używaną do pisania tekstu w formularzu przy użyciu metody TextOut obszaru roboczego. Afont ma nazwę, rozmiar, styl, kolor i tak dalej.

Kolory

Pędzle, pióra i czcionki (jak również formularze i większość innych komponentów) mają właściwość Kolor. Jednak, aby odpowiednio zmienić kolor elementu, używając niestandardowych kolorów (takich

jak stałe kolorów w Delphi), powinieneś wiedzieć, jak system Windows traktuje kolor. Teoretycznie system Windows używa 24-bitowych kolorów RGB. Oznacza to, że możesz użyć 256 różnych wartości dla każdego z trzech podstawowych kolorów (czerwony, zielony i niebieski), uzyskując 16 milionów różnych odcieni. Jednak Ty lub Twoi użytkownicy mogą mieć kartę wideo, która nie może wyświetlać tak wielu kolorów, chociaż jest to coraz rzadziej. W tym przypadku system Windows używa techniki zwanej dithering, która zasadniczo polega na użyciu kilku pikseli dostępnych kolorów w celu zasymulowania żądanej; lub przybliża kolor, używając najbliższego dostępnego dopasowania. W przypadku koloru pędzla (i koloru tła formularza, który jest faktycznie oparty na pędzlu), system Windows wykorzystuje technikę ditheringu; dla koloru pióra lub czcionki używa on najbliższego dostępnego koloru. Jeśli chodzi o pisaki, można odczytać (ale nie zmienić) aktualną pozycję pióra za pomocą właściwości PenPos płótna. Pozycja pióra określa punkt początkowy następnego wiersza, który program narysuje, używając metody LineTo. Aby ją zmienić, można użyć metody MoveTo płótna. Inne właściwości płótna również wpływają na linie i kolory. Interesującymi przykładami są CopyMode i ScaleMode. Inną właściwością, którą można bezpośrednio manipulować, aby zmienić wyjście, jest tablica pikseli, za pomocą której można uzyskać dostęp (odczytać) lub zmienić (zapisać) kolor dowolnego pojedynczego punktu na powierzchni formularza. Jak zobaczymy w przykładzie BmpDraw, operacja na piksel jest bardzo powolna w GDI, w porównaniu do dostępu liniowego dostępnego przez właściwość ScanLines. Na koniec należy pamiętać, że wartości kolorów TCF Delphi nie zawsze odpowiadają zwykłym wartościom RGB natywnej reprezentacji Windows (COLORREF), ze względu na stałe kolorów Delphi. Zawsze możesz przekonwertować kolor Delphi na wartość RGB, używając funkcji ColorToRGB. Szczegóły dotyczące reprezentacji Delphi można znaleźć w wpisie pomocy typu TColor.

Rysowanie kształtów

Teraz chcę rozszerzyć przykład Mouse1 zbudowany na końcu rozdziału 6 i przekształcić go w aplikację Kształty. W tym nowym programie chcę korzystać z podejścia do przechowywania i rysowania z wieloma kształtami, obsługiwać atrybuty koloru i pióra oraz stanowić podstawę do dalszych rozszerzeń. Ponieważ musisz zapamiętać położenie i atrybuty każdego kształtu, możesz utworzyć obiekt dla każdego kształtu, który musisz przechowywać, i możesz przechowywać obiekty na liście. (Bardziej precyzyjnie, lista będzie przechowywać odniesienia do obiektów, które są przydzielane w oddzielnych obszarach pamięci.) Zdefiniowałem klasę bazową dla kształtów i dwóch dziedziczonych klas, które zawierają kod malowania dla dwóch typów kształtów I chcą obsługiwać, prostokąty i elipsy. Klasa bazowa ma kilka właściwości, które po prostu odczytują pola i zapisują odpowiednie wartości za pomocą prostych metod. Zauważ, że współrzędne można odczytać za pomocą właściwości Rect, ale należy je zmodyfikować za pomocą czterech właściwości pozycyjnych. Powodem jest to, że jeśli dodasz część do zapisu do właściwości Rect, możesz uzyskać dostęp do prostokąta jako całości, ale nie do jej określonych właściwości podrzędnych. Oto deklaracje trzech klas:

```
type

TBaseShape = class

private

FBrushColor: TColor;

FPenColor: TColor;

FPenSize: Integer;

procedure SetBrushColor(const Value: TColor);
```

```
procedure SetPenColor(const Value: TColor);
procedure SetPenSize(const Value: Integer);
procedure SetBottom(const Value: Integer);
procedure SetLeft(const Value: Integer);
procedure SetRight(const Value: Integer);
procedure SetTop(const Value: Integer);
protected
FRect: TRect;
public
procedure Paint (Canvas: TCanvas); virtual;
published
property PenSize: Integer read FPenSize write SetPenSize;
property PenColor: TColor read FPenColor write SetPenColor;
property BrushColor: TColor read FBrushColor write SetBrushColor;
property Left: Integer write SetLeft;
property Right: Integer write SetRight;
property Top: Integer write SetTop;
property Bottom: Integer write SetBottom;
property Rect: TRect read FRect;
end;
type
TEllShape = class (TBaseShape)
procedure Paint (Canvas: TCanvas); override;
end;
TRectShape = class (TBaseShape)
procedure Paint (Canvas: TCanvas); override;
end;
```

Większość kodu w metodach jest bardzo prosta. Jedyny odpowiedni kod znajduje się w trzech procedurach Paint:

```
procedure TO ReShape.Paint (Canvas: TCanvas);
begin
```

```
// set the attributes
Canvas.Pen.Color := fPenColor;
Canvas.Pen.Width := fPenSize;
Canvas.Brush.Color := fBrushColor;
end;
procedure TEllShape.Paint(Canvas: TCanvas);
begin
inherited Paint (Canvas);
Canvas.Ellipse (fRect.Left, fRect.Top,
fRect.Right, fRect.Bottom)
end;
procedure TRectShape.Paint(Canvas: TCanvas);
begin
inherited Paint (Canvas);
Canvas.Rectangle (fRect.Left, fRect.Top,
fRect.Right, fRect.Bottom)
end;
```

Cały ten kod jest przechowywany w drugorzędnej jednostce ShapesH (Shapes Hierarchy). Aby zapisać listę kształtów, formularz ma element danych TList obiektu o nazwie Shapes-List, który jest inicjowany w module obsługi zdarzeń OnCreate i zniszczony na końcu; destruktor zwalnia także wszystkie obiekty na liście (w odwrotnej kolejności, aby zbyt często odświeżać wewnętrzne dane listy):

```
procedure TShapesForm.FormCreate(Sender: TObject);
begin
ShapesList := TList.Create;
end;
procedure TShapesForm.FormDestroy(Sender: TObject);
var
I: Integer;
begin
// delete each object
for I := ShapesList.Count - 1 downto 0 do
```

```
TBaseShape (ShapesList [I]).Free;

ShapesList.Free;

end;
```

Program dodaje nowy obiekt do listy za każdym razem, gdy użytkownik rozpoczyna operację przeciągania. Ponieważ obiekt nie jest w pełni zdefiniowany, formularz zachowuje odniesienie do niego w polu CurrShape. Zauważ, że typ utworzonego obiektu zależy od stanu klawiszy myszy:

```
procedure TShapesForm.FormMouseDown(Sender: TObject;

Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

begin

if Button = mbLeft then

begin

// activate dragging

fDragging := True;

SetCapture (Handle);

// create the proper object

if ssShift in Shift then

CurrShape := TEllShape.Create

else

CurrShape := TRectShape.Create;

// set the style and colors

CurrShape.PenSize := Canvas.Pen.Width;

CurrShape.PenColor := Canvas.Pen.Color;

CurrShape.BrushColor := Canvas.Brush.Color;

// set the initial position

CurrShape.Left := X;

CurrShape.Top := Y;

CurrShape.Right := X;

CurrShape.Bottom := Y;

Canvas.DrawFocusRect (CurrShape.Rect);

// add to the list

ShapesList.Add (CurrShape);

end;
```

```
end;
```

Podczas operacji przeciągania narysujemy linię odpowiadającą kształtowi, tak jak to zrobiłem w przykładzie Mouse1:

```
procedure TShapesForm.FormMouseMove(Sender: TObject; Shift:

TShiftState;

X, Y: Integer);

var

ARect: TRect;

begin

// copy the mouse coordinates to the title

Caption := Format ('Shapes (x=%d, y=%d)', [X, Y]);

// dragging code

if fDragging then

begin

// remove and redraw the dragging rectangle

ARect := NormalizeRect (CurrShape.Rect);

Canvas.DrawFocusRect (ARect);

CurrShape.Right := X;

CurrShape.Bottom := Y;

ARect := NormalizeRect (CurrShape.Rect);

Canvas.DrawFocusRect (ARect);

end;

end;
```

Tym razem jednak dodałem również poprawkę do programu. W przykładzie Mouse1, jeśli przesuniesz mysz w kierunku lewego górnego rogu formularza podczas przeciągania, wywołanie DrawFocusRect nie wywoła efektu. Powodem jest to, że prostokąt przekazany jako parametr do DrawFocusRect musi mieć najwyższą wartość, która jest mniejsza niż wartość dolna, i to samo dotyczy wartości lewej i prawej. Innymi słowy, prostokąt, który rozciąga się po stronie negatywnej, nie działa poprawnie. Jednak na końcu maluje poprawnie, ponieważ funkcja rysowania Prostokąt nie ma tego problemu. Aby rozwiązać ten problem, napisałem prostą funkcję, która odwraca współrzędne prostokąt, aby odzwierciedlić żądania wywołania DrawFocusRect:

```
function NormalizeRect (ARect: TRect): TRect;

var

tmp: Integer;
```

```
begin
if ARect.Bottom < ARect.Top then
begin
tmp := ARect.Bottom;
ARect.Bottom := ARect.Top;
ARect.Top := tmp;
end;
if ARect.Right < ARect.Left then
begin
tmp := ARect.Right;
ARect.Right := ARect.Left;
ARect.Left := tmp;
end;
Result := ARect;
end;
```

Na koniec, procedura obsługi zdarzeń OnMouseUp ustawia ostateczny rozmiar obrazu i odświeża obraz formularza. Zamiast wywoływania metody Invalidate, która spowodowałaby przemalowanie wszystkich obrazów za pomocą migotania, program korzysta z funkcji InvalidateRect API:

```
procedure InvalidateRect (Wnd: HWnd;
Rect: PRect; Erase: Bool);
```

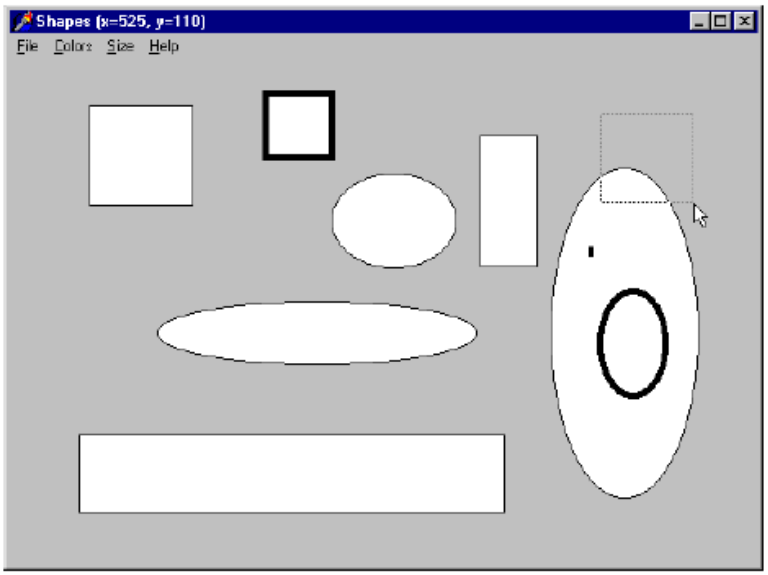
Te trzy parametry reprezentują uchwyt okna (czyli właściwość Handle formularza), prostokąt, który chcesz przemalować, oraz flagę wskazującą, czy chcesz wymazać obszar przed jego odświeżeniem. Ta funkcja wymaga znormalizowanego prostokąta. (Możesz spróbować zastąpić to połączenie jednym z Invalidate, aby zobaczyć różnicę, co jest bardziej oczywiste, gdy tworzysz wiele formularzy.) Oto pełny kod procedury obsługi OnMouseUp:

```
procedure TShapesForm.FormMouseUp(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
var
ARect: TRect;
begin
if fDragging then
begin
```

```
// end dragging

ReleaseCapture;

fDragging := False;

// set the final size

ARect := NormalizeRect (CurrShape.Rect);

Canvas.DrawFocusRect (ARect);

CurrShape.Right := X;

CurrShape.Bottom := Y;

// optimized invalidate code

ARect := NormalizeRect (CurrShape.Rect);

InvalidateRect (Handle, @ARect, False);

end;

end;
```

Po wybraniu dużego pisaka (niedługo przyjrzymy się kodowi), ramka zostanie pomalowana częściowo wewnątrz i częściowo poza ramką, aby pomieścić duże pióro. Aby to umożliwić, powinniśmy unieważnić prostokąt ramki, który jest nadmuchany o połowę mniejszy od bieżącego pisaka. Możesz to zrobić, wywołując funkcję InflateRect. Alternatywnie, w metodzie FormCreate ustawiłem styl pisaka w postaci Canvas na psInsideFrame. Powoduje to, że funkcja rysowania maluje pióro całkowicie w ramce kształtu.

W metodzie odpowiadającej zdarzeniu OnPaint wszystkie kształty aktualnie zapisane na liście są malowane, jak widać na rysunku.



Ponieważ kod malujący wpływa na właściwości płótna, musimy zapisać bieżące wartości i zresetować je na końcu. Powodem jest to, że jak później pokażę w tym tekście, właściwości płótna formularza są używane do śledzenia atrybutów wybranych przez użytkownika, który mógł je zmienić od czasu utworzenia ostatniego kształtu. Oto kod:

```
procedure TShapesForm.FormPaint(Sender: TObject);

var

I, OldPenW: Integer;

AShape: TBaseShape;

OldPenCol, OldBrushCol: TColor;

begin

// store the current Canvas attributes

OldPenCol := Canvas.Pen.Color;

OldPenW := Canvas.Pen.Width;

OldBrushCol := Canvas.Brush.Color;

// repaint each shape of the list

for I := 0 to ShapesList.Count - 1 do

begin

AShape := ShapesList.Items [I];

AShape.Paint (Canvas);

end;

// reset the current Canvas attributes

Canvas.Pen.Color := OldPenCol;

Canvas.Pen.Width := OldPenW;

Canvas.Brush.Color := OldBrushCol;

end;
```

Inne metody formularza są proste. Trzy polecenia menu pozwalają nam zmieniać kolory tła, obramowania kształtu (pióro) i obszaru wewnętrznego (pędzla). Te metody używają składnika ColorDialog i przechowują wynik we właściwościach obszaru roboczego formularza. To jest przykład:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);

begin

// select a new color for the pen

ColorDialog1.Color := Canvas.Pen.Color;

if ColorDialog1.Execute then
```

```
Canvas.Pen.Color := ColorDialog1.Color;

end;
```

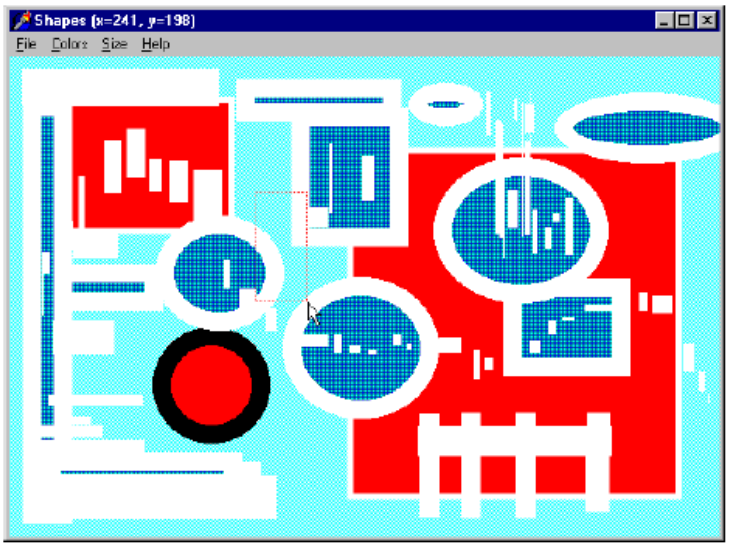
Nowe kolory wpłyną na kształty utworzone w przyszłości, ale nie na istniejące. To samo podejście jest stosowane dla szerokości linii (pióra), chociaż tym razem program sprawdza również, czy wartość stała się zbyt mała, wyłączając pozycję menu, jeśli ma:

```
procedure TShapesForm.DecreasePenSize1Click(Sender: TObject);

begin

Canvas.Pen.Width := Canvas.Pen.Width - 2;

if Canvas.Pen.Width < 3 then

DecreasePenSize1.Enabled := False;

end;
```

Aby zmienić kolory obramowania (pióra) lub powierzchni (pędzla) kształtu, użyłem standardowego okna dialogowego Kolor. Oto jedna z dwóch metod:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);

begin

ColorDialog1.Color := Canvas.Pen.Color;

if ColorDialog1.Execute then

Canvas.Pen.Color := ColorDialog1.Color;

end;
```

Na rysunku poniżej można zobaczyć inny przykład wyjścia programu Shapes, tym razem wykorzystując wiele kolorów dla kształtów i ich tła. Program prosi użytkownika o potwierdzenie niektórych operacji, takich jak wyjście z programu lub usunięcie wszystkich kształtów z listy (za pomocą polecenia Plik ➢ Nowy):



```
procedure TShapesForm.New1Click(Sender: TObject);

begin

if (ShapesList.Count > 0) and (MessageDlg (

'Are you sure you want to delete all the shapes?',

mtConfirmation, [mbYes, mbNo], 0) = idYes) then

begin

// delete each object

for I := ShapesList.Count - 1 downto 0 do

TBaseShape (ShapesList [I]).Free;

ShapesList.Clear;

Refresh;

end;

end;
```

Drukowanie kształtów

Poza malowaniem kształtów na płótnie, możemy je malować na płótnie drukarki, efektywnie je drukując! Ponieważ możliwe jest wykonanie tych samych metod na płótnie drukarki, jak na każdym innym płótnie, możesz pokusić się o dodanie do programu nowej metody drukowania kształtów. Jest to z pewnością łatwe, ale jeszcze lepszą opcją jest pisanie metody pojedynczego wyjścia do użycia zarówno na ekranie, jak i na drukarce. Jako przykład tego podejścia, stworzyłem nową wersję programu o nazwie ShapesPr. Interesujące jest to, że przeniosłem kod przykładu FormPaint do innej zdefiniowanej przeze mnie metody o nazwie CommonPaint. Ta nowa metoda ma dwa parametry, płótno i współczynnik skali (domyślnie 1):

```
procedure CommonPaint(Canvas: TCanvas; Scale: Integer = 1);
```

Metoda CommonPaint wyprowadza listę kształtów do płótna przekazanego jako parametry, używając odpowiedniego współczynnika skali

```
procedure TShapesForm.CommonPaint (

Canvas: TCanvas; Scale: Integer);

var

I, OldPenW: Integer;

AShape: TBaseShape;

OldPenCol, OldBrushCol: TColor;

begin

// store the current Canvas attributes

OldPenCol := Canvas.Pen.Color;
```

```
OldPenW := Canvas.Pen.Width;

OldBrushCol := Canvas.Brush.Color;

// repaint each shape of the list

for I := 0 to ShapesList.Count - 1 do

begin

AShape := ShapesList.Items [I];

AShape.Paint (Canvas, Scale);

end;

// reset the current Canvas attributes

Canvas.Pen.Color := OldPenCol;

Canvas.Pen.Width := OldPenW;

Canvas.Brush.Color := OldBrushCol;

end;
```

Po napisaniu tego kodu, metody FormPaint i Print1Click są łatwe do wdrożenia. Aby malować obraz na ekranie, możesz wywołać funkcję CommonPaint bez współczynnika skalowania (tak, aby użyć wartości domyślnej 1):

```
procedure TShapesForm.FormPaint(Sender: TObject);

begin

CommonPaint (Canvas);

end;
```

Aby namalować zawartość formularza na drukarce zamiast na formularzu, można odtworzyć wyniki na płótnie drukarki, używając odpowiedniego współczynnika skalowania. Zamiast wybierać skalę, zdecydowałem się ją obliczyć automatycznie. Chodzi o to, aby wydrukować kształty w formularzu tak duże, jak to możliwe, przez rozmiar obszaru roboczego formularza, tak aby zajmował całą stronę. Kod jest prawdopodobnie prostszy niż opis:

```
procedure TShapesForm.Print1Click(Sender: TObject);

var

Scale, Scale1: Integer;

begin

Scale := Printer.PageWidth div ClientWidth;

Scale1 := Printer.PageHeight div ClientHeight;

if Scale1 < Scale then

Scale := Scale1;
```

```
Printer.BeginDoc;

try

CommonPaint (Printer.Canvas, Scale);

Printer.EndDoc;

except

Printer.Abort;

raise;

end;

end;
```

Oczywiście, musisz pamiętać, aby wywołać konkretne polecenia, aby rozpocząć drukowanie (BeginDoc) i zatwierdzić wyjście (EndDoc) przed i po wywołaniu metody Common-Paint. Jeśli zostanie zgłoszony wyjątek, program wywoła przerwanie, aby zakończyć proces drukowania.

Komponenty graficzne Delphi

W przykładzie Kształty prawie nie ma żadnych komponentów, poza standardowym oknem dialogowym wyboru koloru. Alternatywnie mogliśmy użyć niektórych komponentów Delphi, które obsługują grafikę:

• Używasz komponentu PaintBox, gdy chcesz malować na określonym obszarze formularza, który może poruszać się po formularzu. Na przykład PaintBox jest przydatny do malowania w oknie dialogowym bez ryzyka mieszania obszaru dla wyjścia z obszarem dla elementów sterujących. PaintBox może zmieścić się w innych formach kontrolnych formularza, takich jak pasek narzędzi lub pasek stanu, i uniknąć pomyłki lub nakładania się danych wyjściowych. W przykładzie Kształty używanie tego komponentu nie miało sensu, ponieważ zawsze pracowaliśmy na całej powierzchni formularza.

• Używasz komponentu Kształt do malowania kształtów na ekranie, dokładnie tak, jak do tej pory. Można rzeczywiście używać komponentu Shape zamiast ręcznego, ale naprawdę chciałem pokazać, jak wykonać pewne bezpośrednie operacje wyjściowe. Takie podejście nie było bardziej złożone niż sugeruje Delphi. Używanie komponentu Shape może być przydatne do rozszerzenia przykładu, pozwalając użytkownikowi przeciągać kształty na ekranie, usuwać je i pracować nad nimi na wiele innych sposobów.

• Możesz użyć komponentu Obraz, aby wyświetlić istniejącą bitmapę, ewentualnie ładując ją z pliku lub nawet malować na bitmapie, co pokażę w dwóch następnych przykładach i omówię w następnej sekcji.

• Jeśli jest on zawarty w twojej wersji Delphi, możesz użyć formantu TeeChart do tworzenia wyników biznesowych, jak zobaczymy pod koniec tego tekstu.

• Można korzystać między innymi z obsługi graficznej udostępnianej przez przyciski bitmap i kontrolki przycisków prędkości.

• Możesz użyć komponentu Animate, aby grafika była bardziej ... dobrze animowana. Poza używaniem tego komponentu, możesz ręcznie tworzyć animacje, wyświetlając bitmapy w sekwencji lub przewijając je, ponieważ zobaczymy inne przykłady.

Jak widać, mamy długą drogę, aby objąć wsparcie graficzne Delphi ze wszystkich jego kątów.

Rysowanie w bitmapie

Wspomniałem już, że za pomocą komponentu Obraz można rysować obrazy bezpośrednio w bitmapie. Zamiast rysować na powierzchni okna, rysujesz na mapie bitowej w pamięci, a następnie kopiujesz bitmapę na powierzchnię okna. Zaletą jest to, że zamiast odświeżać obraz za każdym razem, gdy wystąpi zdarzenie OnPaint, komponent kopiuje bitmapę z powrotem do wideo. Z technicznego punktu widzenia obiekt TBitmap ma własne płótno. Rysując to płótno, możesz zmienić zawartość bitmapy. Alternatywnie możesz pracować na kanwie komponentu Obraz podłączonego do mapy bitowej, którą chcesz zmienić. Można rozważyć wybór tego podejścia zamiast typowego podejścia do malowania, jeśli spełniony jest dowolny z następujących warunków:

• Program musi obsługiwać rysowanie odręczne lub bardzo złożoną grafikę (np. Obrazy fraktalne).

• Program powinien bardzo szybko rysować wiele obrazów.

• Zużycie pamięci RAM nie jest problemem.

• Jesteś leniwym programistą.

Ostatni punkt jest interesujący, ponieważ malowanie na ogół wymaga więcej kodu niż rysowania, chociaż pozwala na większą elastyczność. Na przykład w programie graficznym, jeśli używasz malowania, musisz zapisać położenie i kolory każdego kształtu. Z drugiej strony możesz łatwo zmienić kolor istniejącego kształtu lub go przenieść. Operacje te są bardzo trudne w przypadku malowania i mogą spowodować utratę obszaru za obrazem. Jeśli pracujesz nad złożonym graficznie wniosku, prawdopodobnie powinieneś wybrać mieszankę dwóch podejść. W przypadku zwykłych programistów graficznych wybór między tymi dwoma podejściami wiąże się z typową decyzją dotyczącą prędkości i pamięci: malowanie wymaga mniej pamięci; przechowywanie bitmapy jest szybsze.

Rysowanie kształtów

Teraz spójrzmy na przykład komponentu Obraz, który będzie malować na bitmapie. Pomysł jest prosty. Zasadniczo napisałem uproszczoną wersję przykładu Shape, umieszczając komponent Obraz w jego formie i przekierowując wszystkie operacje wyjściowe na kanwę tego komponentu Obraz. W tym przykładzie, ShapeBmp, dodałem również kilka nowych pozycji menu, aby zapisać obraz do pliku i załadować istniejącą bitmapę. Aby to osiągnąć, dodałem do formularza kilka domyślnych komponentów dialogowych, OpenDialog i SaveDialog. Jedną z właściwości, które musiałem zmienić, był kolor tła formularza. W rzeczywistości, po wykonaniu pierwszej operacji graficznej na obrazie, tworzy ona bitmapę, która domyślnie ma białe tło. Jeśli formularz ma szare tło, za każdym razem, gdy okno jest przemalowywane, pojawia się migotanie. Z tego powodu wybrałem także białe tło dla formularza. Kod tego przykładu jest nadal dość prosty, biorąc pod uwagę liczbę operacji i poleceń menu. Część rysunkowa jest liniowa i bardzo zbliżona do myszy1, z tym wyjątkiem, że zdarzenia myszy odnoszą się teraz do obrazu zamiast do postaci; Użyłem funkcji NormalizeRect podczas przeciągania; a program używa płótna obrazu. Oto procedura obsługi zdarzeń OnMouseMove, która ponownie wprowadza rysowanie punktów podczas przesuwania myszy przy wciśniętym klawiszu Shift:

```
procedure TShapesForm.Image1MouseMove(Sender: TObject;

Shift: TShiftState; X, Y: Integer);
```

```
var
ARect: TRect;
begin
// display the position of the mouse in the caption
Caption := Format ('ShapeBmp (x=%d, y=%d)', [X, Y]);
if fDragging then
begin
// remove and redraw the dragging rectangle
ARect := NormalizeRect (fRect);
Canvas.DrawFocusRect (ARect);
fRect.Right := X;
fRect.Bottom := Y;
ARect := NormalizeRect (fRect);
Canvas.DrawFocusRect (ARect);
end
else
if ssShift in Shift then
// mark point in red
Image1.Canvas.Pixels [X, Y] := clRed;
end;
```

Zwróć uwagę, że tymczasowy prostokąt fokusu jest malowany bezpośrednio na formularzu, nad obrazem (a zatem nie jest przechowywany w mapie bitowej). Różnica polega na tym, że na końcu operacji przeciągania program maluje prostokąt na obrazie, przechowując go w bitmapie. Tym razem program nie wywołuje Invalidate i nie ma programu obsługi zdarzeń OnPaint:

```
procedure TShapesForm.Image1MouseUp(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
if fDragging then
begin
ReleaseCapture;
fDragging := False;
Image1.Canvas.Rectangle (fRect.Left, fRect.Top,
```

```
fRect.Right, fRect.Bottom);

end;

end;
```

Aby uniknąć nadmiernie złożonej obsługi plików, postanowiłem zaimplementować polecenia Plik ➢ Załaduj i Plik ➢ Zapisz jako i nie obsługuj polecenia Zapisz, które jest ogólnie bardziej złożone. Po prostu dodałem pole fChanged do formularza, aby wiedzieć, kiedy obraz się zmienił, i dodałem kod, który sprawdza tę wartość kilka razy (zanim poprosi użytkownika o potwierdzenie). Procedura obsługi zdarzenia OnClick elementu menu Plik ➢ Nowy wywołuje metodę FillArea, aby malować duży biały prostokąt na całej bitmapie. W tym kodzie możesz zobaczyć, w jaki sposób użyto pola Changed:

```
procedure TShapesForm.New1Click(Sender: TObject);

var

Area: TRect;

OldColor: TColor;

begin

if not fChanged or (MessageDlg (

'Are you sure you want to delete the current image?',

mtConfirmation, [mbYes, mbNo], 0) = idYes) then

begin

{repaint the surface, covering the whole area,

and resetting the old brush}

Area := Rect (0, 0, Image1.Picture.Width,

Image1.Picture.Height);

OldColor := Image1.Canvas.Brush.Color;

Image1.Canvas.Brush.Color := clWhite;

Image1.Canvas.FillRect (Area);

Image1.Canvas.Brush.Color := OldColor;

fChanged := False;

end;

end;
```

Oczywiście kod musi zapisać oryginalny kolor i przywrócić go później. Odwzorowanie kolorów jest również wymagane przez metodę Plik-Załaduj odpowiedź-odpowiedź. Po załadowaniu nowej bitmapy składnik Obraz tworzy nowe płótno z atrybutami domyślnymi. Z tego powodu program zapisuje kolory i rozmiar pióra i kopiuje je później na nowe płótno:

```
procedure TShapesForm.Load1Click(Sender: TObject);
```

```
var
PenCol, BrushCol: TColor;
PenSize: Integer;
begin
if not fChanged or (MessageDlg (
'Are you sure you want to delete the current image?',
mtConfirmation, [mbYes, mbNo], 0) = idYes) then
if OpenDialog1.Execute then
begin
PenCol := Image1.Canvas.Pen.Color;
BrushCol := Image1.Canvas.Brush.Color;
PenSize := Image1.Canvas.Pen.Width;
Image1.Picture.LoadFromFile (OpenDialog1.Filename);
Image1.Canvas.Pen.Color := PenCol;
Image1.Canvas.Brush.Color := BrushCol;
Image1.Canvas.Pen.Width := PenSize;
fChanged := False;
end;
end;
```

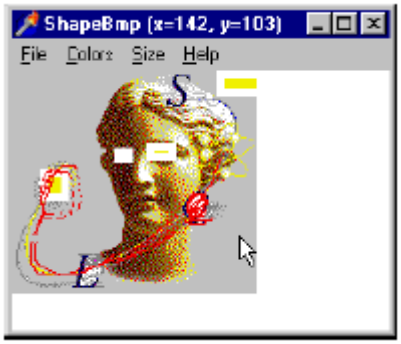
Zapisanie bieżącego obrazu jest znacznie prostsze:

```
procedure TShapesForm.Saveas1Click(Sender: TObject);
begin
if SaveDialog1.Execute then
begin
Image1.Picture.SaveToFile (
SaveDialog1.Filename);
fChanged := False;
end;
end;
```

Finally, here is the code of the OnCloseQuery event of the form, which uses the Changed field:

```
procedure TShapesForm.FormCloseQuery(Sender: TObject;

var CanClose: Boolean);

begin

if not fChanged or (MessageDlg (

'Are you sure you want to delete the current image?',

mtConfirmation, [mbYes, mbNo], 0) = idYes) then

CanClose := True

else

CanClose := False;

end;
```

ShapeBmp jest interesującym programem



, z ograniczoną obsługą plików roboczych. Prawdziwy problem polega na tym, że komponent Obraz tworzy mapę bitową o własnym rozmiarze. Gdy powiększysz rozmiar okna, komponent Obraz zostanie przeskalowany, ale nie będzie mapą bitową w pamięci. Dlatego nie można rysować na prawych i dolnych obszarach okna. Istnieje wiele możliwych rozwiązań: użyj właściwości Ograniczenia, aby ustawić maksymalny rozmiar formularza, użyj stałej krawędzi, zaznacz obszar rysunku na ekranie i tak dalej. Postanowiłem jednak opuścić program, ponieważ ma za zadanie zademonstrować, jak dobrze rysować mapę bitową.

Przeglądarka Obrazów

Program ShapeBmp może być używany jako przeglądarka obrazów, ponieważ można w nim załadować dowolną bitmapę. Ogólnie rzecz biorąc w formancie Image można załadować dowolny typ pliku graficznego, który został zarejestrowany w klasie Ticta VCL. Domyślne formaty plików to pliki bitmap (BMP), pliki ikon (ICO) lub metapliki Windows (WMF). Pliki bitmapowe i ikony są dobrze znanymi formatami. Metapliki Windows nie są jednak tak powszechne. Są zbiorem poleceń graficznych, podobnych do listy wywołań funkcji GDI, które muszą zostać wykonane w celu odbudowania obrazu. Metapliki są zwykle nazywane grafiką wektorową i są podobne do formatów plików graficznych używanych w bibliotekach graficznych. Delphi dostarcza również obsługę JPG dla TImage, a strony trzecie mają GIF i inne formaty plików.Aby utworzyć metaplik Windows, program powinien wywoływać funkcje GDI, przekierowując ich wyniki do pliku. W Delphi możesz korzystać z metod TManafileCanvas i Highanvel TCanvas. Później ten metaplik można odtwarzać lub wykonywać w celu wywołania odpowiednich funkcji, tworząc w ten sposób grafikę. Metapliki mają dwie główne zalety: ograniczoną

ilość pamięci, jakiej wymagają w porównaniu do innych formatów graficznych, oraz niezależność urządzenia od ich wydajności. Omówię obsługę metapliku Delphi później.

Aby zbudować pełny program do przeglądania obrazów, ImageV, wokół komponentu Image, musimy tylko utworzyć formularz z obrazem, który wypełnia cały obszar klienta, proste menu i składnik OpenDialog:

object ViewerForm: TViewerForm

Caption = 'Image Viewer'

Menu = MainMenu1

object Image1: TImage

Align = alClient

end

object MainMenu1: TMainMenu

object File1: TMenuItem...

object Open1: TMenuItem...

object Exit1: TMenuItem...

object Options1: TMenuItem

object Stretch1: TMenuItem

object Center1: TMenuItem

object Help1: TMenuItem

object AboutImageViewer1: TMenuItem

end

object OpenDialog1: TOpenDialog

FileEditStyle = fsEdit

Filter = 'Bitmap (*.bmp)|*.bmp|

Icon (*.ico)|*.ico|Metafile (*.wmf)|*.wmf'

Options = [ofHideReadOnly, ofPathMustExist,

ofFileMustExist]

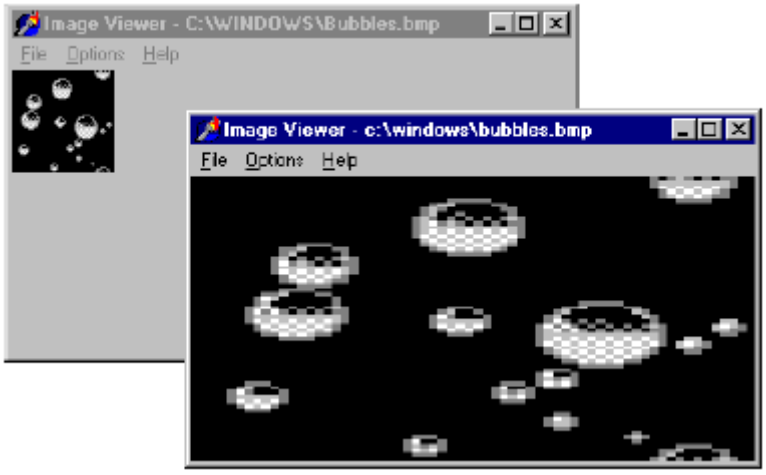
end

end

Zaskakujące, że ta aplikacja wymaga bardzo mało kodowania, przynajmniej w swojej pierwszej wersji podstawowej. Plik ➢ Wyjście i pomoc ➢Informacje są trywialne, a polecenie Plik ➢ Otwórz ma następujący kod:

```
procedure TViewerForm.Open1Click (Sender: TObject);
zaczynać
jeśli OpenDialog1.Execute następnie
zaczynać
Image1.Picture.LoadFromFile (OpenDialog1.FileName);
Caption: = 'Image Viewer -' + OpenDialog1.FileName;
koniec;
koniec;
```

Czwarte i piąte polecenia menu: Opcje ➢ Rozciągnij i Opcje ➢ Wyśrodkuj, po prostu przełącz właściwość Rozciągnij komponentu



lub Właściwości Centrum i dodaj znacznik wyboru. Oto obsługa zdarzeń OnClick pozycji menu Strech1:

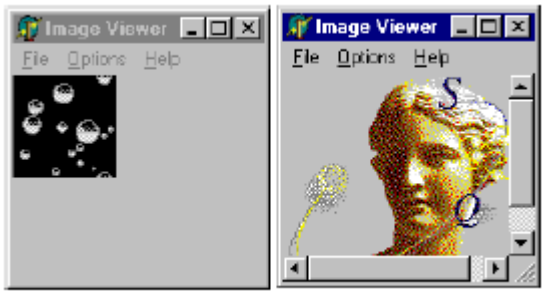
```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
Image1.Stretch := not Image1.Stretch;
Stretch1.Checked := Image1.Stretch;
end;
```

Należy pamiętać, że podczas rozciągania obrazu można zmienić jego stosunek szerokości do wysokości, prawdopodobnie zniekształcając kształt, a także, że nie wszystkie obrazy można prawidłowo rozciągnąć. Rozciąganie czarno-białych lub 256-bitowych bitmap nie zawsze działa poprawnie. Poza tym problemem, aplikacja ma kilka innych wad. Jeśli wybierzesz plik bez jednego ze standardowych rozszerzeń, komponent Obraz podniesie wartość wyjątek. Obsługa wyjątków zapewniona przez system zachowuje się tak, jak się spodziewaliśmy; zły plik obrazu nie został załadowany, a program można bezpiecznie kontynuować. Innym problemem jest to, że po załadowaniu dużego obrazu, przeglądarka

nie ma pasków przewijania. Możesz zmaksymalizować okno przeglądarki, ale to może nie wystarczyć. Komponenty obrazu nie obsługują automatycznie pasków przewijania, ale formularz może to zrobić. W dalszej części tego przykładu dodam paski przewijania w następnym akapicie.

Przewijanie obrazu

Zaletą sposobu, w jaki działa automatyczne przewijanie w Delphi, jest to, że jeśli rozmiar pojedynczego dużego komponentu w formularzu ulegnie zmianie, paski przewijania zostaną dodane lub usunięte automatycznie. Dobrym przykładem jest użycie komponentu obrazu. Jeśli właściwość Autosize tego komponentu jest ustawiona na True, a załadujesz do niej nowy obraz, komponent automatycznie sam się powiększy, a formularz doda lub usunie paski przewijania w razie potrzeby. Jeśli załadujesz dużą bitmapę w przykładzie ImageV, zauważysz, że część mapy bitowej pozostaje ukryta. Aby to naprawić, możesz ustawić właściwość Autosize komponentu Image na True i wyłączyć jego wyrównanie z obszarem klienta. Powinieneś również ustawić mały początkowy rozmiar obrazu. Nie trzeba wprowadzać żadnych zmian podczas ładowania nowej mapy bitowej, ponieważ rozmiar komponentu obrazu jest automatycznie ustawiany przez system. Na rysunku



widać, że paski przewijania są rzeczywiście dodawane do formularza. Na rysunku pokazano dwie różne kopie programu. Różnica między kopią programu po lewej stronie a kopią po prawej jest taka, że pierwszy ma obraz mniejszy niż obszar roboczy, więc nie dodano pasków przewijania. Po załadowaniu większego obrazu do programu automatycznie pojawią się dwa paski przewijania, jak w przykładzie po prawej stronie. Do wyłączenia pasków przewijania i zmiany wyrównania obrazu wymagane jest dodatkowe kodowanie, gdy wybrana jest komenda menu Rozciągnij i aby przywrócić je, gdy ta funkcja jest wyłączona. Ponownie, nie działamy bezpośrednio na same paski przewijania, ale po prostu zmieniamy wyrównanie panelu, używając jego właściwości Rozciągnij, i ręcznie obliczamy nowy rozmiar, używając rozmiaru aktualnie załadowanego obrazu. (Ten kod naśladuje efekt właściwości Autosize, która działa tylko wtedy, gdy ładowany jest nowy plik.)

```
procedure TViewerForm.Stretch1Click(Sender: TObject);

begin

Image1.Stretch := not Image1.Stretch;

Stretch1.Checked := Image1.Stretch;

if Image1.Stretch then

Image1.Align := alClient

else
```

```
begin

Image1.Align := alNone;

Image1.Height := Image1.Picture.Height;

Image1.Width := Image1.Picture.Width;

end;

end;
```

Gdy kontrola obrazu jest połączona z bitmapą, możesz wykonać kilka dodatkowych operacji, ale zanim je zbadamy, muszę wprowadzić formaty bitmapowe. W systemie Windows istnieją różne typy map bitowych. Mapy bitowe mogą być niezależne od urządzenia lub nie, termin używany do wskazania, czy mapa bitowa ma dodatkowe informacje dotyczące zarządzania paletami. Pliki BMP są zwykle niezależnymi od urządzenia bitmapami. Inna różnica dotyczy głębi kolorów - to znaczy liczby różnych kolorów, z których może korzystać bitmapa lub, innymi słowy, liczby bitów wymaganych do przechowywania każdego piksela. W 1-bitowej mapie bitowej każdy punkt może być czarny lub biały (bardziej precyzyjnie, 1-bitowe mapy bitowe mogą mieć paletę kolorów, co pozwala bitmapie reprezentować dowolne dwa kolory, a nie tylko czarno-białe). 8-bitowa mapa bitowa ma zwykle paletę towarzyszącą, aby wskazać, w jaki sposób 256 różnych kolorów odwzorowuje rzeczywiste kolory systemu, 24-bitowa mapa bitowa wskazuje bezpośrednio kolor systemu. Aby uczynić rzeczy bardziej skomplikowanymi, gdy system rysuje bitmapę na komputerze o innej zdolności kolorów, musi wykonać pewną konwersję. Wewnętrznie format bitmapy jest bardzo prosty, niezależnie od głębi kolorów. Wszystkie wartości tworzące linię są zapisywane sekwencyjnie w bloku pamięci. Jest to skuteczne w przenoszeniu danych z pamięci na ekran, ale nie jest to skuteczny sposób przechowywania informacji; Pliki BMP są na ogół bardzo duże i nie wykonują kompresji. Format BMP ma w rzeczywistości bardzo ograniczoną formę kompresji, znaną jako Run-Length Encoding (RLE), w której kolejne piksele o tym samym kolorze są zastępowane liczbą takich pikseli, po których następuje kolor. Może to zmniejszyć rozmiar obrazu, ale w niektórych przypadkach spowoduje jego wzrost. W przypadku skompresowanych obrazów w Delphi można użyć klasy TJpegImage i obsługi formatu JPEG oferowanego przez klasę TPicture. Właściwie wszystkie obrazki TP zarządzają zarejestrowaną listą klas graficznych.

Przykład BmpDraw wykorzystuje te informacje o wewnętrznej strukturze mapy bitowej i niektórych innych funkcjach technicznych, aby przenieść bezpośrednią obsługę bitmap na nowy poziom. Po pierwsze, rozszerza przykład ImageV, dodając element menu, którego można użyć do wyświetlenia głębi kolorów bieżącej mapy bitowej, używając odpowiedniej właściwości PixelFormat:

```
procedure TBitmapForm.ColorDepth1Click(Sender: TObject);

var

strDepth: String;

begin

case Image1.Picture.Bitmap.PixelFormat of

pfDevice: strDepth := 'Device';

pf1bit: strDepth := '1-bit';

pf4bit: strDepth := '4-bit';
```

pf8bit: strDepth := ‘8-bit’;

pf15bit: strDepth := ‘15-bit’;

pf16bit: strDepth := ‘16-bit’;

pf24bit: strDepth := ‘24-bit’;

pf32bit: strDepth := ‘32-bit’;

pfCustom: strDepth := ‘Custom’;

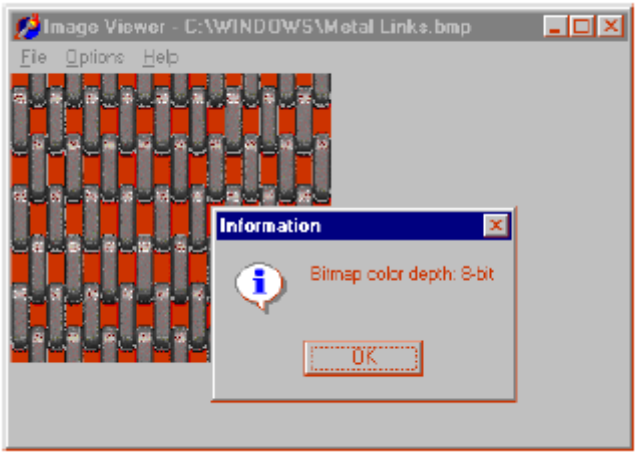
end;

MessageDlg (‘Bitmap color depth: ‘ + strDepth,

mtInformation, [mbOK], 0);

end;

Możesz spróbować załadować różne bitmapy i zobaczyć efekt tej metody, jak pokazano na rysunku



Bardziej interesujące jest zbadanie, jak uzyskać dostęp do obrazu pamięci przechowywanego przez obiekt bitmapowy. Najprostszym rozwiązaniem jest użycie właściwości Piksele, tak jak zrobiłem to w przykładzie ShapeBmp, aby narysować czerwone piksele podczas operacji przeciągania. W tym programie dodałem element menu, aby utworzyć całą nową mapę bitową piksel po pikselu, używając prostego obliczenia matematycznego do określenia koloru. (To samo podejście można wykorzystać na przykład do tworzenia obrazów fraktalnych). Oto kod metody, która po prostu skanuje bitmapę w obu kierunkach i określa kolor każdego piksela. Ponieważ wykonujemy wiele operacji na bitmapie, dla uproszczenia mogę przechowywać odniesienie do niej w lokalnej zmiennej Bmp:

procedure TBitmapForm.GenerateSlow1Click(Sender: TObject);

var

Bmp: TBitmap;

I, J, T: Integer;

begin

// get the image and modify it

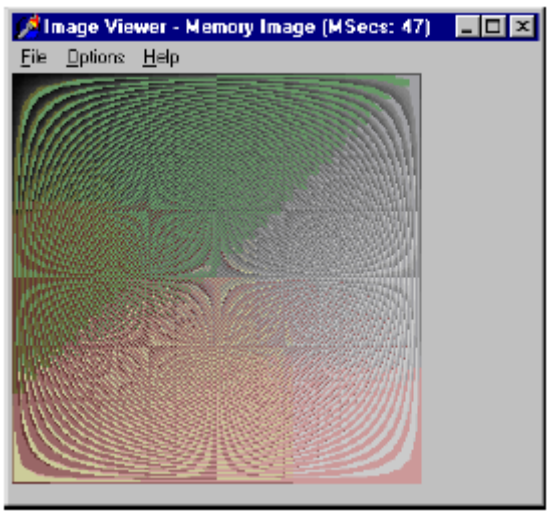
```
Bmp := Image1.Picture.Bitmap;

Bmp.PixelFormat := pf24bit;

Bmp.Width := 256;

Bmp.Height := 256;

T := GetTickCount;

// change every pixel

for I := 0 to Bmp.Height - 1 do

for J := 0 to Bmp.Width - 1 do

Bmp.Canvas.Pixels [I, J] := RGB (I*J mod 255, I, J);

Caption := 'Image Viewer - Memory Image (MSecs: ' +

IntToStr (GetTickCount - T) + ')';

end;
```

Zauważ, że program śledzi czas wymagany przez tę operację, który na moim komputerze zajmuje około sześciu sekund. Jak widać z nazwy funkcji, jest to powolna wersja kodu. Możemy znacznie przyspieszyć, uzyskując dostęp do mapy bitowej po jednym wierszu na raz. Ta mało znana funkcja jest dostępna za pośrednictwem właściwości ScanLine mapy bitowej, która zwraca wskaźnik do obszaru pamięci linii bitmapy. Biorąc ten wskaźnik i uzyskując bezpośredni dostęp do pamięci, program staje się znacznie szybszy. Jedynym problemem jest to, że musimy znać wewnętrzną reprezentację mapy bitowej. W przypadku 24-bitowej mapy bitowej każdy punkt jest reprezentowany przez trzy bajty określające ilość koloru niebieskiego, zielonego i czerwonego (odwrotność sekwencji RGB). Oto alternatywny kod z nieco innym wynikiem (ponieważ celowo zmodyfikowałem obliczenie koloru):

```
procedure TBitmapForm.GenerateFast1Click(Sender: TObject);

var

Bmp: TBitmap;

I, J, T: Integer;

Line: PByteArray;

begin

// get the image and modify it

Bmp := Image1.Picture.Bitmap;

Bmp.PixelFormat := pf24bit;

Bmp.Width := 256;

Bmp.Height := 256;

T := GetTickCount;

// change every pixel, line by line
```

```
for I := 0 to Bmp.Height - 1 do
begin
Line := PByteArray (Bmp.ScanLine [I]);
for J := 0 to Bmp.Width - 1 do
begin
Line [J*3] := J;
Line [J*3+1] := I*J mod 255;
Line [J*3+2] := I;
end;
end;
// refresh the video
Image1.Invalidate;
Caption := 'Image Viewer - Memory Image (MSecs: ' +
IntToStr (GetTickCount - T) + ')';
end;
```

Samo przesunięcie linii w pamięci nie powoduje aktualizacji ekranu, więc program wywołuje Invalidate na końcu. Wynik uzyskany dzięki tej drugiej metodzie



jest bardzo podobny, ale czas potrzebny na mój komputer wynosi około 60 milisekund. To mniej więcej jedna setna czasu innego podejścia! Ta technika jest tak szybka, że możemy jej użyć do przewijania linii bitmapy i nadal zapewniać szybki i płynny efekt. Operacja przewijania ma kilka opcji, więc po wybraniu odpowiednich elementów menu program po prostu wyświetla panel wewnątrz formularza. Ten panel ma pasek ścieżki, za pomocą którego można dostosować szybkość przewijania (zmniejszając jego płynność wraz ze wzrostem prędkości). Pozycja paska jest zapisana w lokalnym polu formularza:

```
procedure TBitmapForm.TrackBar1Change(Sender: TObject);
```

```
begin

nLines := TrackBar1.Position;

TrackBar1.Hint := IntToStr (TrackBar1.Position);

end;
```

W panelu znajdują się również dwa przyciski służące do uruchamiania i zatrzymywania operacji przewijania. Kod przycisku Go ma dwie pętle. Pętla zewnętrzna służy do powtarzania operacji przewijania, tyle razy, ile linii znajduje się w mapie bitowej. Pętla wewnętrzna wykonuje operację przewijania, kopiując każdą linię mapy bitowej do poprzedniej. Pierwsza linia jest tymczasowo przechowywana w bloku pamięci, a następnie kopiowana do ostatniej linii na końcu. Ten tymczasowy blok pamięci jest przechowywany w dynamicznie przydzielanym obszarze pamięci (AllocMem) wystarczająco dużym, aby pomieścić jedną linię. Informacje te uzyskuje się przez obliczenie różnicy adresów pamięci dwóch kolejnych linii. Rdzeń poruszającej się operacji jest realizowany za pomocą funkcji Delphi Move. Jego parametry to zmienna do przeniesienia, a nie adresy pamięci. Z tego powodu musisz odwołać wskaźniki. (Cóż, ta metoda jest naprawdę dobrym ćwiczeniem na wskaźnikach!) Na koniec zauważ, że tym razem nie możemy unieważnić całego obrazu po każdej operacji przewijania, ponieważ powoduje to zbyt duże migotanie na wyjściu. Przeciwnym rozwiązaniem jest unieważnienie każdej linii po jej przeniesieniu, ale powoduje to, że program jest zbyt wolny. Jako rozwiązanie pośrednie postanowiłem unieważnić blok wierszy naraz, zgodnie z wyrażeniem J mod nLines = 0. Po przeniesieniu określonej liczby linii program odświeża te linie:

```
Rect (0, PanelScroll.Height + H - nLines,

W, PanelScroll.Height + H);
```

Jak widać, liczba linii jest określona przez pozycję kontrolki TrackBar. Auser może nawet zmieniać prędkość, przesuwając kciuk podczas przewijania. Umożliwiamy również użytkownikowi naciśnięcie przycisku Anuluj podczas operacji. Jest to możliwe dzięki wywołaniu Application.ProcessMessages w zewnętrznej pętli for. Przycisk Anuluj zmienia flagę f Anuluj, która jest zaznaczona każda iteracja zewnętrznej pętli for:

```
procedure TBitmapForm.BtnCancelClick(Sender: TObject);

begin

fCancel := True;

end;
```

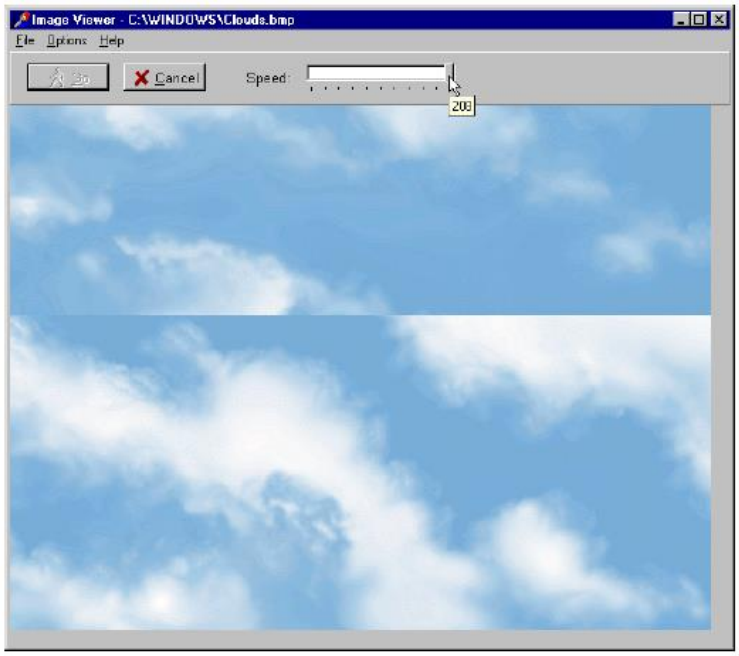
Tak więc po tym opisie znajduje się pełny kod programu obsługi zdarzenia OnClick przycisku Go:

```
procedure TBitmapForm.BtnGoClick(Sender: TObject);

var

W, H, I, J, LineBytes: Integer;

Line: PByteArray;

Bmp: TBitmap;

R: TRect;
```

```
begin
// set the user interface
fCancel := False;
BtnGo.Enabled := False;
BtnCancel.Enabled := True;
// get the bitmap of the image and resize it
Bmp := Image1.Picture.Bitmap;
W := Bmp.Width;
H := Bmp.Height;
// allocate enough memory for one line
LineBytes := Abs (Integer (Bmp.ScanLine [1]) -
Integer (Bmp.ScanLine [0]));
Line := AllocMem (LineBytes);
// scroll as many items as there are lines
for I := 0 to H - 1 do
begin
// exit the for loop if Cancel was pressed
if fCancel then
Break;
// copy the first line
Move ((Bmp.ScanLine [0])^, Line^, LineBytes);
// for every line
for J := 1 to H - 1 do
begin
// move line to the previous one
Move ((Bmp.ScanLine [J])^, (Bmp.ScanLine [J-1])^, LineBytes);
// every nLines update the output
if (J mod nLines = 0) then
begin
R := Rect (0, PanelScroll.Height + J-nLines,
W, PanelScroll.Height + J);
```

```
InvalidateRect (Handle, @R, False);
UpdateWindow (Handle);
end;
end;
// move the first line back to the end
Move (Line^, (Bmp.ScanLine [Bmp.Height - 1])^, LineBytes);
// update the final portion of the bitmap
R := Rect (0, PanelScroll.Height + H - nLines,
W, PanelScroll.Height + H);
InvalidateRect (Handle, @R, False);
UpdateWindow (Handle);
// let the program handle other messages
Application.ProcessMessages;
end;
// reset the UI
BtnGo.Enabled := True;
BtnCancel.Enabled := False;
end;
```

Możesz zobaczyć bitmapę podczas przewijania na rysunku.



Zauważ, że przewijanie może odbywać się na dowolnym typie bitmapy, a nie tylko na 24-bitowych mapach bitowych generowanych przez ten program. Możesz w rzeczywistości załadować do programu inną bitmapę, a następnie ją przewinąć, tak jak to zrobiłem, aby utworzyć ilustrację.

Animowana mapa bitowa w przycisku

Przyciski bitmapowe są łatwe w użyciu i mogą tworzyć lepiej wyglądające aplikacje niż standardowe przyciski (komponent Button). Aby jeszcze bardziej poprawić efekt wizualny przycisku, możemy również pomyśleć o animowaniu przycisku. Zasadniczo istnieją dwa rodzaje animowanych przycisków - przyciski, które zmieniają nieznacznie swój glif, gdy są one naciskane, oraz przyciski z ruchomym obrazem, niezależnie od prądu operacja. Pokażę wam prosty przykład każdego rodzaju, Ognia i Świata. Dla każdego z tych przykładów zbadamy kilka nieco innych wersji.

Przycisk dwustanowy

Pierwszy przykład, program Fire, ma bardzo prostą formę, zawierającą tylko przycisk mapy bitowej. Ten przycisk jest połączony z Glifem reprezentującym działo. Wyobraź sobie taki przycisk jako część programu do gry. Po naciśnięciu przycisku glif zmienia się, pokazując działko strzelające. Po zwolnieniu przycisku domyślny glif jest ładowany ponownie. W międzyczasie program wyświetla komunikat, jeśli użytkownik rzeczywiście kliknął przycisk. Aby napisać ten program, musimy poradzić sobie z trzema zdarzeniami tego przycisku: OnMouse-Down, OnMouseUp i OnClick. Kod trzech metod jest niezwykle prosty:

```
procedure TForm1.BitBtnFireMouseDown(Sender: TObject;

Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

begin

// load firing cannon bitmap

if Button = mbLeft then

BitBtnFire.Glyph.LoadFromFile (‘fire2.bmp’);

end;

procedure TForm1.BitBtnFireMouseUp(Sender: TObject;

Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

begin

// load default cannon bitmap

if Button = mbLeft then

BitBtnFire.Glyph.LoadFromFile (‘fire.bmp’);

end;

procedure TForm1.BitBtnFireClick(Sender: TObject);

begin

PlaySound (‘Boom.wav’, 0, snd_Async);

MessageDlg (‘Boom!’, mtWarning, [mbOK], 0);
```

end;

Dodałem kilka funkcji dźwiękowych, odtwarzając plik WAV po naciśnięciu przycisku z wywołaniem funkcji PlaySound jednostki MmSystem. Przytrzymanie lewego przycisku myszy nad przyciskiem mapy bitowej powoduje naciśnięcie przycisku mapy bitowej. Jeśli odsuniesz kursor myszy od przycisku, trzymając wciśnięty przycisk myszy, przycisk mapy bitowej zostanie zwolniony, ale nie dostaniesz zdarzenia OnMouseUp, więc działo strzelające pozostaje tam. Jeśli później zwolnisz lewy przycisk myszy poza powierzchnią przycisku mapy bitowej, i tak otrzyma zdarzenie OnMouseUp. Powodem jest to, że wszystkie przyciski w systemie Windows przechwytują dane wejściowe myszy po ich naciśnięciu.

Wiele obrazów w mapie bitowej

Przykład Fire użył ręcznego podejścia. Załadowałem dwie mapy bitowe i zmieniłem wartość właściwości Glyph, gdy chciałem zmienić obraz. Komponent BitBtn może jednak również obsługiwać wiele bitmap automatycznie. Możesz przygotować pojedynczą bitmapę, która zawiera wiele obrazów (lub glifów) i ustawić tę liczbę jako wartość właściwości NumGlyphs. Wszystkie takie „sub-bitmapy” muszą mieć ten sam rozmiar, ponieważ ogólna mapa bitowa jest podzielona na równe części. Jeśli podasz więcej niż jeden glif w mapie bitowej, zostaną one użyte zgodnie z następującymi zasady:

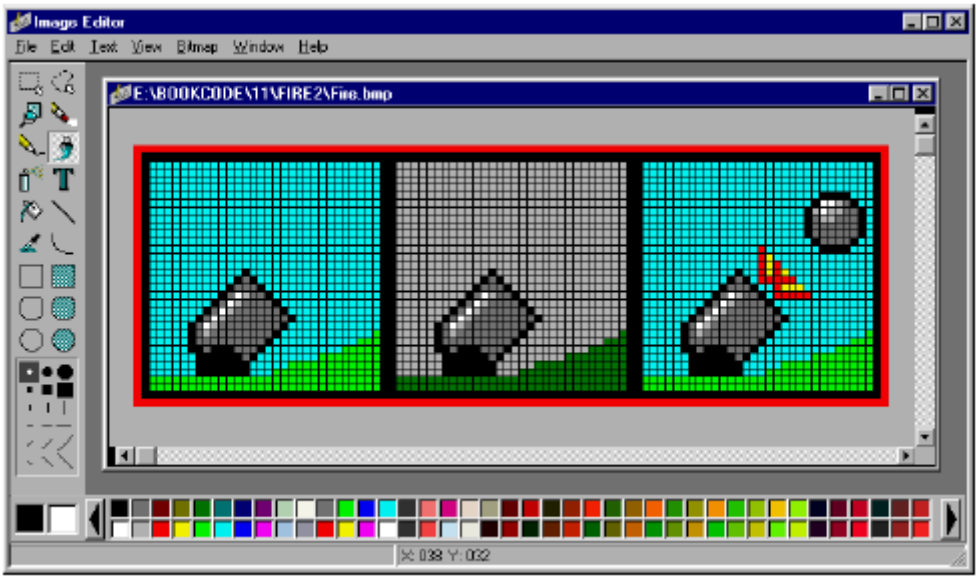
• Pierwsza mapa bitowa jest używana dla zwolnionego przycisku, domyślnej pozycji.

• Druga mapa bitowa jest używana dla wyłączonego przycisku.

• Trzecia mapa bitowa jest używana po kliknięciu przycisku.

• Czwarta mapa bitowa jest używana, gdy przycisk pozostaje w dół, tak jak w przypadku przycisków zachowujących się jak pola wyboru.

Zwykle udostępniasz pojedynczy glif, a pozostałe są automatycznie obliczane na podstawie prostych zmian graficznych. Łatwo jest jednak dostarczyć drugi, trzeci i czwarty niestandardowy obraz. Jeśli nie podasz wszystkich czterech map bitowych, brakujące zostaną obliczone automatycznie z pierwszego. W naszym przykładzie, nowa wersja Fire (o nazwie Fire2), potrzebujemy tylko pierwszego i trzeciego glifu bitmapy, ale musimy dodać drugą bitmapę. Aby zobaczyć, jak można użyć tego glifu (drugiego z bitmapy), dodałem pole wyboru, aby wyłączyć przycisk mapy bitowej. Aby zbudować nową wersję programu, przygotowałem bitmapę o wymiarach 32 × 96 pikseli



i użyłem jej do właściwości Glyph mapy bitowej. Delphi automatycznie ustawia właściwość NumGlyphs na 3, ponieważ bitmapa jest trzy razy większa niż jest wysoka. Pole wyboru, używane do włączania i wyłączania przycisku (abyśmy mogli zobaczyć glif odpowiadający wyłączonemu statusowi), ma następujące zdarzenie OnClick:

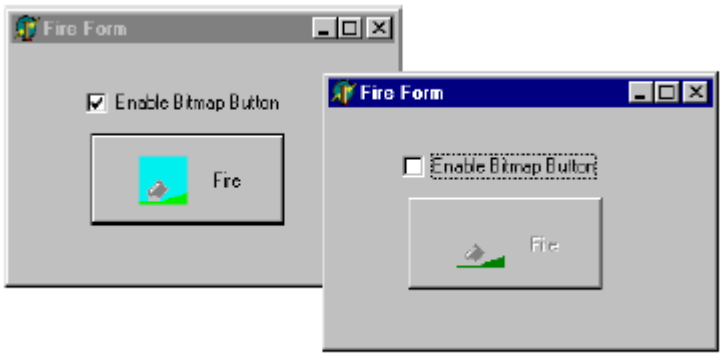
procedure TForm1.CheckBox1Click(Sender: TObject);

begin

BitBtnFire.Enabled := CheckBox1.Checked;

end;

Po uruchomieniu programu istnieją dwa sposoby zmiany mapy bitowej w przycisku. Możesz wyłączyć przycisk mapy bitowej za pomocą pola wyboru



lub możesz nacisnąć przycisk, aby zobaczyć ogień dział. W pierwszej wersji (przykład Ognia) obraz z działem wystrzeliwania pozostał na przycisku do momentu zamknięcia okna wiadomości. Teraz (w przykładzie Fire2) obraz jest wyświetlany tylko po naciśnięciu przycisku. Gdy tylko wyjdziesz poza powierzchnię przycisku lub zwolnisz przycisk po jego naciśnięciu (aktywacja okna wiadomości), wyświetlony zostanie pierwszy glif.

Obrotowy świat

Drugi przykład animacji, Świat, ma przycisk z ziemią, która powoli się obraca, pokazując różne kontynenty. Możesz zobaczyć niektóre próbki na rysunku 22.11, ale oczywiście powinieneś uruchomić program, aby zobaczyć jego wyjście. W poprzednim przykładzie obraz zmienił się po naciśnięciu przycisku. Teraz obraz zmienia się automatycznie. Dzieje się tak dzięki obecności komponentu Timera , który odbiera komunikat w stałych odstępach czasu. Oto podsumowanie właściwości komponentu:

object WorldForm: TWorldForm

Caption = ‘World’

OnCreate = FormCreate

object Label1: TLabel...

object WorldButton: TBitBtn

Caption = ‘&Start’

OnClick = WorldButtonClick

Glyph.Data = {W1.bmp}

Spacing = 15

end

object Timer1: TTimer

Enabled = False

Interval = 500

OnTimer = Timer1Timer

end

end

Składnik czasowy jest uruchamiany i zatrzymywany (włączony i wyłączany), gdy użytkownik naciśnie przycisk mapy bitowej z obrazem świata:

procedure TWorldForm.WorldButtonClick(Sender: TObject);

begin

if Timer1.Enabled then

begin

Timer1.Enabled := False;

WorldButton.Caption := ‘&Start’;

end

else

begin

Timer1.Enabled := True;

WorldButton.Caption := ‘&Stop’;

end;

end;

Jak widać na rysunku powyżej, etykieta nad przyciskiem wskazuje, które obrazy są wyświetlane. Za każdym razem, gdy odbierany jest komunikat zegara, zmienia się obraz i etykieta:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);

begin

Count := (Count mod 16) + 1;

Label1.Caption := 'Displaying image ' +

IntToStr (Count);

WorldButton.Glyph.LoadFromFile (

'w' + IntToStr (Count) + '.bmp');

end;
```

W tym kodzie Count jest polem formularza, który jest inicjalizowany na 1 w metodzie FormCreate. W każdym przedziale czasowym Count zwiększa moduł 16, a następnie przekształca go na ciąg znaków (poprzedzony literą w). Powód tego ograniczenia jest prosty - miałem 16 bitmap do wyświetlenia na ziemi. Nadawanie nazw plikom bitmapowym W1.BMP, W2.BMP itd. Ułatwia programowi dostęp do nich, tworząc łańcuchy z nazwą w czasie wykonywania. Operacja modulus zwraca resztę podziału między liczby całkowite. Oznacza to, że Count mod 16 niezmiennie zwraca wartość z zakresu 0–15. Dodając jeden do tej wartości zwracanej, uzyskujemy numer mapy bitowej, który mieści się w zakresie 1–16.

Lista bitmap, wykorzystanie zasobów i ControlCanvas

Program World działa, ale z kilku powodów jest bardzo powolny. Przede wszystkim w każdym przedziale czasowym musi odczytać plik z dysku i chociaż pamięć podręczna dysku może to przyspieszyć, z pewnością nie jest to najbardziej wydajne rozwiązanie. Oprócz odczytu pliku z dysku, program musi utworzyć i zniszczyć obiekty bitmapowe Windows, a to zajmuje trochę czasu. Drugi problem zależy od sposobu aktualizacji obrazu: po zmianie mapy bitowej przycisku komponent jest całkowicie wymazany i odmalowany. Powoduje to migotanie, jak widać po uruchomieniu programu. Aby rozwiązać pierwszy problem (i pokazać inne podejście do obsługi bitmap), stworzyłem drugą wersję tego przykładu, World2. W tym miejscu dodałem kontener TObjectList, przechowujący listę bitmap do formularza programu. Formularz ma również kilka dodatkowych pól:

```
type

TWorldForm = class(TForm)

...

private

Count, YPos, XPos: Integer;

BitmapsList: TObjectList;

ControlCanvas: TControlCanvas;

end;
```

Wszystkie mapy bitowe są ładowane, gdy program uruchamia się i ulega zniszczeniu po zakończeniu. W każdym przedziale czasowym program wyświetla jedną z map bitowych listy w przycisku mapy bitowej. Używając listy, unikamy ładowania pliku za każdym razem, gdy musimy wyświetlić bitmapę, ale nadal musimy mieć wszystkie pliki z obrazami w katalogu z plikiem wykonywalnym. Rozwiązaniem tego problemu jest przeniesienie bitmap z niezależnych plików do pliku zasobów aplikacji. Łatwiej to zrobić niż wyjaśnić. Aby użyć zasobów zamiast plików bitmap, musimy najpierw utworzyć ten plik. Najlepszym podejściem jest napisanie skryptu zasobów (pliku RC), zawierającego listę nazw plików bitmap i odpowiednich zasobów. Otwórz nowy plik tekstowy (w dowolnym edytorze)

i napisz następujący kod:

W1 BITMAP „W1.BMP”

W2 BITMAP „W2.BMP”

W3 BITMAP „W3.BMP”

// ... i tak dalej

Po przygotowaniu tego pliku RC (nazwałem go WorldBmp.RC), możesz skompilować go do pliku RES przy użyciu dołączonego kompilatora zasobów i aplikacji wiersza poleceń BRCC32, którą można znaleźć w katalogu BIN Delphi, a następnie włącz ją do projektu, dodając dyrektywę {$ R WORLDBMP.RES} w pliku kodu źródłowego projektu lub w jednej z jednostek. W Delphi 5 można jednak użyć prostszego podejścia. Możesz wziąć plik RC i po prostu dodać go do projektu za pomocą polecenia Dodaj Menedżera projektu lub po prostu przeciągając plik do projektu. Delphi 5 automatycznie aktywuje kompilator zasobów, a następnie wiąże plik zasobów z plikiem wykonywalnym. Operacje te są kontrolowane przez dyrektywę rozszerzonego włączania zasobów dodaną do kodu źródłowego projektu:

{$ R „WORLDBMP.res” WORLDBMP.RC ”}

Po prawidłowym zdefiniowaniu zasobów aplikacji musimy załadować bitmapy z zasobów. Dla obiektu TBitmap możemy użyć metody Load- FromResourceName, jeśli zasób ma identyfikator łańcucha lub metodę LoadFrom-ResourceID, jeśli ma identyfikator liczbowy. Pierwszym parametrem obu metod jest uchwyt do aplikacji, znany jako HInstance, dostępny w Delphi jako zmienna globalna.

Delphi definiuje drugą zmienną globalną MainInstance, która odnosi się do HInstance głównego pliku wykonywalnego. Jeśli nie znajdujesz się w bibliotece DLL, możesz używać jednego lub drugiego zamiennie

To jest kod metody FormCreate:

```
procedure TWorldForm.FormCreate(Sender: TObject);

var

I: Integer;

Bmp: TBitmap;

begin

Count := 1;

// load the bitmaps and add them to the list
```

```
BitmapsList := TList.Create;

for I := 1 to 16 do

begin

Bmp := TBitmap.Create;

Bmp.LoadFromResourceName (HInstance,

'W' + IntToStr (I));

BitmapsList.Add (Bmp);

end;

end;
```

Alternatywnie moglibyśmy użyć komponentu ImageList, ale w tym przykładzie postanowiłem zastosować podejście niskiego poziomu, aby pokazać wszystkie szczegóły. Jeden problem pozostaje do rozwiązania: uzyskanie płynnego przejścia od jednego obrazu świata do następnego. Program powinien malować bitmapy w płótnie za pomocą metody Draw. Niestety, płótno przycisku mapy bitowej nie jest bezpośrednio dostępne (i nie jest chronione przed zdarzeniami), więc zdecydowałem się użyć TControl-Canvas (zwykle wewnętrznego kanonu kontrolki, ale można go również skojarzyć z zewnętrzem) Aby użyć go do pomalowania przycisk, możemy przypisać przycisk do kontrolki płótno w metodzie FormCreate:

```
ControlCanvas: = TControlCanvas.Create;

ControlCanvas.Control: = WorldButton;

YPos: = (WorldButton.Height - Bmp.Height) div 2;

XPos: = WorldButton.Margin;
```

Poziome położenie przycisku, w którym znajduje się obraz (i gdzie powinniśmy malować), zależy od marginesu ikony przycisku mapy bitowej i wysokości mapy bitowej. Po prawidłowym ustawieniu płótna kontrolnego metoda Timer1Timer po prostu maluje nad nim - i nad przyciskiem:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);

begin

Count := (Count mod 16) + 1;

Label1.Caption := Format ('Displaying image %d', [Count]);

// draw the current bitmap in the control canvas

ControlCanvas.Draw (XPos, YPos,

BitmapsList.Items[Count-1] as TBitmap);

end;
```

Ostatnim problemem jest przesunięcie pozycji obrazu po naciśnięciu lub zwolnieniu lewego przycisku myszy (czyli w zdarzeniach OnMouseDown i OnMouseUp przycisku). Oprócz przesuwania obrazu o kilka pikseli, powinniśmy zaktualizować glif bitmapy, ponieważ Delphi automatycznie wyświetli go

podczas przerysowywania przycisku. W przeciwnym razie użytkownik zobaczy początkowy obraz do momentu upłynął interwał czasowy i komponent uruchomił zdarzenie OnTimer. (Może to trochę potrwać, jeśli go zatrzymasz!) Oto kod pierwszej z dwóch metod:

```
procedure TWorldForm.WorldButtonMouseDown(Sender: TObject;

Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

begin

if Button = mbLeft then

begin

// paint the current image over the button

WorldButton.Glyph.Assign (

BitmapsList.Items[Count-1] as TBitmap);

Inc (YPos, 2);

Inc (XPos, 2);

end;

end;
```

Kontrola animacji

Jest lepszy sposób na uzyskanie animacji niż wyświetlanie sekwencji bitmap w sekwencji. Użyj wspólnej kontroli Win32 Animate. Kontrola Animate opiera się na wykorzystaniu plików AVI (Audio Video Interleaved), serii bitmap podobnych do filmu. W rzeczywistości formant Animate może wyświetlać tylko te pliki AVI, które mają pojedynczy strumień wideo, są nieskompresowane lub skompresowane za pomocą kompresji RLE8 i nie mają zmian w palecie; a jeśli mają dźwięk, jest to ignorowane. W praktyce pliki odpowiadające temu wymaganiu to pliki utworzone z serii bitmap komputerowych, a nie z rzeczywistych filmów. Formant Animate może mieć dwa możliwe źródła animacji:

• Może być oparty na dowolnym pliku AVI, który spełnia wymagania wskazane w powyższej uwadze; aby użyć tego typu źródła, ustaw odpowiednią wartość dla właściwości FileName.

• Może używać specjalnej wewnętrznej animacji Windows, stanowiącej część wspólnej biblioteki kontrolnej; aby użyć tego typu źródła, wybierz jedną z możliwych wartości właściwości CommonAVI (która jest oparta na wyliczeniu).

Jeśli po prostu umieścisz formant Animate w formularzu, wybierz animację za pomocą jednej z opisanych powyżej metod, a na koniec ustaw jej właściwość Aktywna na Prawda, a zobaczysz animację wykonaną nawet w czasie projektowania. Domyślnie animacja działa nieprzerwanie, uruchamiając ją ponownie, gdy tylko zostanie wykonana. Możesz jednak regulować ten efekt za pomocą właściwości Powtórzenia. Wartość domyślna -1 powoduje nieskończone powtarzanie; użyj dowolnej innej wartości, aby określić liczbę powtórzeń. Możesz także określić początkową i końcową klatkę sekwencji, korzystając z właściwości Start-Frame i StopFrame. Te trzy właściwości (pozycja początkowa, pozycja końcowa i liczba powtórzeń) odpowiadają trzem parametrom metody Play, których często używasz z formantem Animate. Alternatywnie można ustawić właściwości, a następnie wywołać metodę Start. W czasie wykonywania możesz również uzyskać dostęp do całkowitej liczby ramek za pomocą właściwości FrameCount: możesz użyć tego do wykonania animacji od początku do końca. Na koniec,

dla lepszej kontroli, możesz użyć metody Seek, która wyświetla konkretną ramkę. Użyłem wszystkich tych metod w prostym programie demonstracyjnym, który może używać zarówno plików, jak i standardowych animacji systemu Windows. Program pozwala wybrać plik lub jedną z animacji za pomocą ListBox. Dodałem element do tego ListBox dla każdego elementu wyliczenia TCommonAVI i użyłem tej samej kolejności:

```
object ListBox1: TListBox

Items.Strings = (

'[Use an AVI file]'

'Find Folder'

'Find File'

'Find Computer'

'Copy Files'

'Copy File'

'Recycle File'

'Empty Recycle'

'Delete File')

OnClick = ListBox1Click

end
```

Dzięki tej strukturze, gdy użytkownik kliknie na ListBox, po prostu rzucając liczbę wybranych elementów do wyliczonego typu danych otrzyma właściwą wartość dla właściwości CommonAVI.

```
procedure TForm1.ListBox1Click(Sender: TObject);

begin

Animate1.CommonAVI := TCommonAVI (ListBox1.ItemIndex);

if (ListBox1.ItemIndex = 0) and

OpenDialog1.Execute then

Animate1.FileName := OpenDialog1.FileName

end;
```

Jak widać, gdy wybrany jest pierwszy element (wartość to caNone), program automatycznie ładuje plik AVI, używając komponentu OpenDialog. Najważniejszym elementem formularza jest formant Animate. Oto jego opis tekstowy:

```
object Animate1: TAnimate

AutoSize = False

Align = alClient

CommonAVI = aviFindFolder
```

OnOpen = Animate1Open

end

Jest dostosowany do obszaru klienta, dzięki czemu użytkownik może łatwo zmienić jego rozmiar w zależności od rzeczywistego rozmiaru klatek animacji. Jak widać, zdefiniowałem także obsługę zdarzenia dla tego komponentu OnOpen:

procedure TForm1.Animate1Open(Sender: TObject);

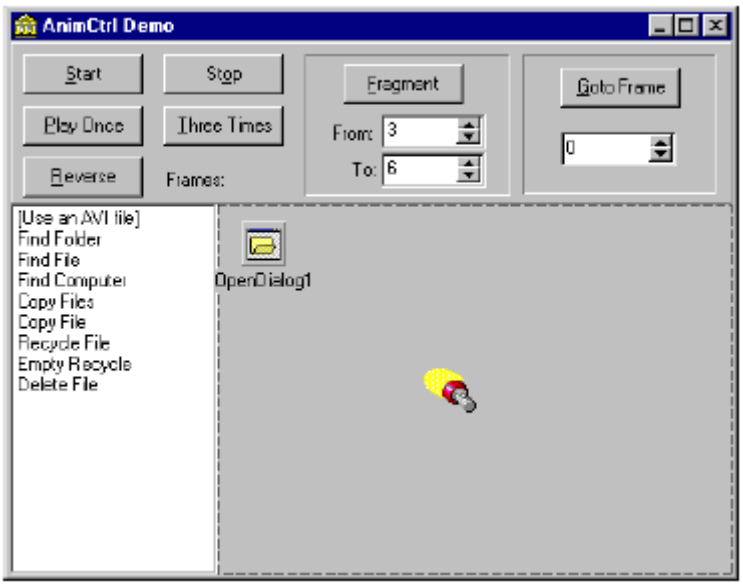
begin

LblFrames.Caption := 'Frames ' +

IntToStr (Animate1.FrameCount);

end;

Po otwarciu nowego pliku (lub wspólnej animacji) program po prostu wyświetla liczbę ramek na etykiecie. Ta etykieta jest hostowana wraz z kilkoma przyciskami i kilkoma kontrolkami SpinEdit w dużym panelu, działającym jako pasek narzędzi. Możesz je zobaczyć w formie czasu projektowania na rysunku



Przyciski Start i Stop są całkowicie trywialne, ale przycisk Odtwórz raz ma prosty kod:

procedure TForm1.BtnOnceClick(Sender: TObject);

begin

Animate1.Play (0, Animate1.FrameCount, 1);

end;

Sprawy zaczynają być ciekawsze dzięki kodowi używanemu do trzykrotnego odtwarzania animacji lub odtwarzania tylko jej fragmentu. Obie te metody są oparte na metodzie Play:

procedure TForm1.BtnTriceClick(Sender: TObject);

```
begin

Animate1.Play (0, Animate1.FrameCount, 3);

end;

procedure TForm1.BtnFragmentClick(Sender: TObject);

begin

Animate1.Play (SpinEdit1.Value, SpinEdit2.Value, -1);

end;
```

Ostatnie dwie procedury obsługi zdarzeń przycisków są oparte na metodzie Seek. Przycisk Goto po prostu przesuwa się do ramki wskazanej przez odpowiedni komponent SpinEdit, podczas gdy przyciski Odwróć do każdej klatki po kolei, zaczynając od ostatniego i zatrzymując się między nimi:

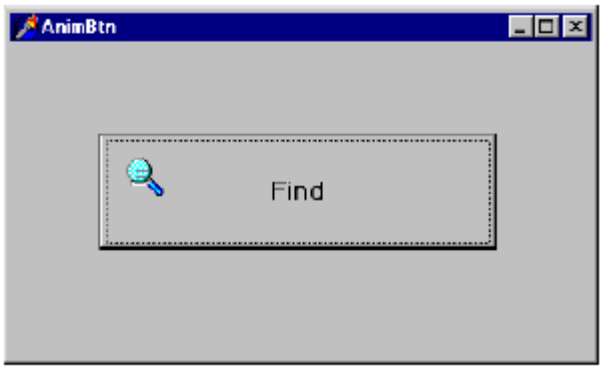
```
procedure TForm1.BtnGotoClick(Sender: TObject);

begin

Animate1.Seek (SpinEdit3.Value);

end;

procedure TForm1.BtnReverseClick(Sender: TObject);

var

Init: TDateTime;

I: Integer;

begin

for I := Animate1.FrameCount downto 1 do

begin

Animate1.Seek (I);

// wait 50 milliseconds

Init := Now;

while Now < Init + EncodeTime (0, 0, 0, 50) do

Application.ProcessMessages;

end;

end;
```

Animowana kontrola w przycisku

Teraz, gdy wiesz, jak działa formant Animate, możemy go użyć do zbudowania innego animowanego przycisku. Po prostu umieść kontrolkę Animate i duży przycisk (ewentualnie z dużą czcionką) w formularzu. Następnie napisz poniższy kod, aby przycisk stał się oknem nadrzędnym kontrolki Animate w czasie wykonywania i ustaw go poprawnie:

```
procedure TForm1.FormCreate(Sender: TObject);

var

hDiff: Integer;

begin

Animate1.Parent := Button1;

hDiff := Button1.Height - Animate1.Height;

Animate1.SetBounds (hDiff div 2, hDiff div 2,

Animate1.Width, Animate1.Height);

Animate1.Active := True;

end;
```

Przykład tego efektu można zobaczyć na rysunku .

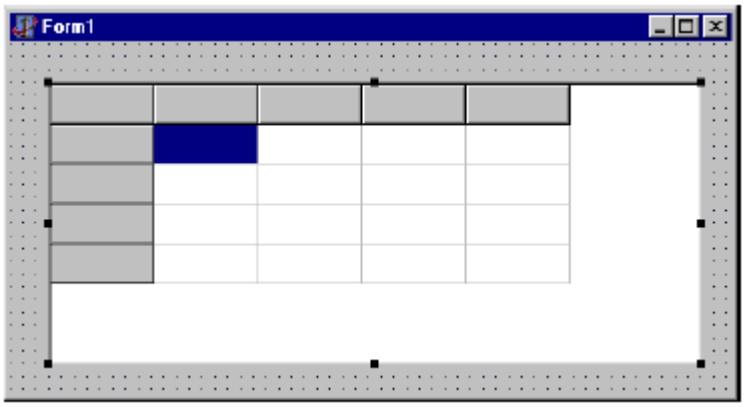


(Projekt ma nazwęAnimBtn.) Jest to najprostsze podejście do tworzenia animowanego przycisku, ale pozwala również na najmniejszą kontrolę.

Siatki graficzne

Siatki stanowią kolejną interesującą grupę komponentów graficznych Delphi. System oferuje różne składniki siatki: siatkę łańcuchów, jedną z obrazów, siatki związane z bazą danych i przykładową siatkę kolorów. Pierwsze dwa rodzaje siatek są szczególnie użyteczne, ponieważ umożliwiają reprezentowanie dużej ilości informacji i pozwalają użytkownikowi poruszać się po nim. Oczywiście siatki są niezwykle ważne w programowaniu baz danych i można je dostosowywać za pomocą grafiki. Komponenty DrawGrid i StringGrid są ściśle powiązane. W rzeczywistości Klasa TStringGrid jest podklasą TDrawGrid. Jaki jest pożytek z tych siatek? Zasadniczo można zapisać niektóre wartości, zarówno w łańcuchach związanych z StringGrid, jak iw innych strukturach danych, a następnie wyświetlić wybrane wartości, używając określonych kryteriów. Podczas gdy siatki ciągów mogą być używane prawie tak, jak są (ponieważ już oferują możliwości edycji), siatki ogólnych obiektów zwykle wymagają więcej kodowanie. Siatki w rzeczywistości definiują sposób organizacji informacji na potrzeby wyświetlania, a nie sposób ich przechowywania. Jedyną siatką przechowującą wyświetlane dane jest StringGrid. Wszystkie inne siatki (w tym komponenty DrawGrid i DBGrid) są tylko przeglądarkami danych, a nie kontenerami danych. DBGrid nie jest właścicielem danych, które wyświetla; pobiera dane z podłączonego źródła danych. Czasami jest to źródłem zamieszania.

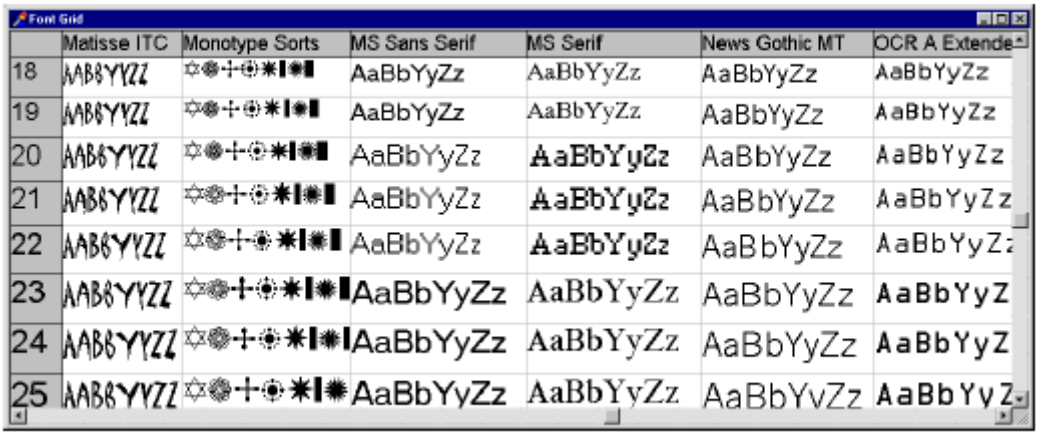
Podstawowa struktura siatki zawiera pewną liczbę stałych kolumn i wierszy, które wskazują obszar, w którym nie można przewijać siatki



Siatki należą do najbardziej złożonych komponentów dostępnych w Delphi, na co wskazuje duża liczba właściwości i metod, które zawierają. Istnieje wiele opcji i właściwości dla siatek, kontrolujących zarówno ich wygląd, jak i zachowanie. W swoim wyglądzie siatka może mieć linie o różnych rozmiarach lub może nie mieć linii. Możesz ustawić rozmiar każdej kolumny lub wiersza niezależnie od innych, ponieważ właściwości RowSize, ColWidth i RowHeight są tablicami. W przypadku zachowania siatki możesz pozwolić użytkownikowi zmienić rozmiar kolumn i wierszy (goColSizing i goRowSizing), przeciągnąć całe kolumny i wiersze do nowej pozycji (goRowMoving i goColumnMoving), wybrać automatyczną edycję i zezwolić na wybór zakresu. Ponieważ różne opcje umożliwiają użytkownikom wykonywanie wielu operacji na siatkach, istnieje również szereg zdarzeń związanych z siatkami, takich jak OnColumnMoved, OnDrawCell lub OnSetEditText. Najważniejszym wydarzeniem jest prawdopodobnie OnDrawCell. W odpowiedzi na to zdarzenie program musi namalować określoną komórkę siatki. Tylko siatki ciągów mogą automatycznie wyświetlać ich zawartość. W rzeczywistości DrawGrid nie obsługuje przechowywania danych. Jest to po prostu narzędzie do aranżacji części ekranu w celu wyświetlania informacji regularny format. Jest to proste narzędzie, ale także potężne. Metody takie jak Cell-Rect, która zwraca prostokąt odpowiadający obszarowi komórki, lub MouseTo- Cell, która zwraca komórkę w określonym miejscu, są radością z użycia. Obsługując zmienne wiersze i kolumny oraz przewijalne siatki, upraszczają złożone zadania i oszczędzają programiście żmudnych obliczeń. Do czego można użyć siatki? Tworzenie arkusza kalkulacyjnego jest prawdopodobnie pierwszym pomysłem, który przychodzi na myśl, ale prawdopodobnie jest to zbyt skomplikowany przykład. Zdecydowałem się użyć kontrolki StringGrid w programie, który pokazuje zainstalowane czcionki w systemie i formant DrawGrid w programie, który emuluje grę Mine-Sweeper.

Siatka czcionek

Jeśli umieścisz komponent StringGrid na formularzu i odpowiednio ustawisz jego opcje, masz pełny działający edytor ciągów ułożonych w siatkę, bez żadnego programowania. Aby uczynić ten przykład bardziej interesującym, postanowiłem narysować każdą komórkę siatki inną czcionką, zmieniając zarówno jej rozmiar, jak i krój pisma. Możesz zobaczyć wynik programu FontGrid na rysunku.



Forma tego programu jest bardzo prosta. Wystarczy umieścić komponent siatki na formularzu, wyrównać go z obszarem klienta, ustawić kilka właściwości i opcji i pozwolić programowi wykonać resztę. Liczba kolumn i wierszy oraz ich rozmiar są w rzeczywistości obliczane w czasie wykonywania. Ważnymi właściwościami, które musisz ustawić, są Domyślny rysunek, który powinien być Fałszem, aby pozwolić nam namalować siatkę, jak nam się podoba, i Opcje:

object Form1: TForm1

Caption = ‘Font Grid’

OnCreate = FormCreate

object StringGrid1: TStringGrid

Align = alClient

DefaultColWidth = 200

DefaultDrawing = False

Options = [goFixedVertLine, goFixedHorzLine,

goVertLine, goHorzLine, goDrawFocusSelected,

goColSizing, goColMoving, goEditing]

OnDrawCell = StringGrid1DrawCell

end

end

Jak to zwykle bywa w Delphi, im prostsza jest forma, tym bardziej skomplikowany jest kod. Ten przykład jest zgodny z tą regułą, chociaż ma tylko dwie metody, jedną do inicjalizacji siatki przy starcie, a drugą do rysowania przedmiotów. Edycja w rzeczywistości nie została dostosowana i odbywa się za pomocą czcionki systemowej. Pierwsza z dwóch metod to FormCreate. Na początku ta metoda używa globalnego obiektu Screen, aby uzyskać dostęp do czcionek zainstalowanych w systemie. Siatka ma kolumnę dla każdej czcionki, jak również stałą kolumnę z liczbami reprezentującymi rozmiary czcionek. Nazwa każdej kolumny jest kopiowana z obiektu Ekran do pierwszego wiersza każdej kolumny (która ma indeks zerowy):

```
procedure TForm1.FormCreate(Sender: TObject);
var
I, J: Integer;
begin
{the number of columns equals the number of fonts plus
1 for the first fixed column, which has a size of 20}
StringGrid1.ColCount := Screen.Fonts.Count + 1;
StringGrid1.ColWidths [0] := 50;
for I := 1 to Screen.Fonts.Count do
begin
// write the name of the font in the first row
StringGrid1.Cells [I, 0] :=
Screen.Fonts.Strings [I-1];
{compute maximum required size of column, getting the width
of the text with the biggest size of the font in that column}
StringGrid1.Canvas.Font.Name :=
StringGrid1.Cells [I, 0];
StringGrid1.Canvas.Font.Size := 32;
StringGrid1.ColWidths [I] :=
StringGrid1.Canvas.TextWidth ('AaBbYyZz');
end;
...
```

W ostatniej części powyższego kodu program oblicza szerokość każdej kolumny. Osiąga się to poprzez ocenę przestrzeni zajmowanej przez niestandardowy ciąg tekstu AaBbYyZz, używając czcionki kolumny (zapisanej w pierwszym wierszu, Komórki [I, 0]) i największego rozmiaru czcionki używanego przez program (32). Aby obliczyć przestrzeń wymaganą przez tekst, można zastosować metody TextWidth i TextHeight do obszaru roboczego z wybraną odpowiednią czcionką. Wiersze natomiast mają zawsze wartość 26 i mają rosnącą wysokość, obliczoną z przybliżoną formułą: 15 + I x 2. W rzeczywistości obliczanie najwyższego tekstu oznacza sprawdzanie wysokości tekstu w każdej kolumnie, z pewnością zbyt złożonej operacji dla ten przykład. Przybliżona formuła działa dobrze, jak widać na rysunku powyżej i uruchamiając program. W pierwszej komórce każdego wiersza program zapisuje rozmiar czcionki, który odpowiada numerowi linii plus siedem. Ostatnia operacja polega na zapisaniu ciągu „AaBbYyZz" w każdej niemieszonej komórce siatki. Aby to osiągnąć, program używa zagnieżdżonej pętli for. Spodziewaj się często zagnieżdżonych pętli podczas pracy z siatkami. Oto druga część metody FormCreate:

```
// defines the number of columns

StringGrid1.RowCount := 26;

for I := 1 to 25 do

begin

// write the number in the first column

StringGrid1.Cells [0, I] := IntToStr (I+7);

// set an increasing height for the rows

StringGrid1.RowHeights [I] := 15 + I*2;

// insert default text in each column

for J := 1 to StringGrid1.ColCount do

StringGrid1.Cells [J, I] := 'AaBbYyZz'

end;

StringGrid1.RowHeights [0] := 25;

end;
```

Teraz możemy przestudiować drugą metodę, StringGrid1DrawCell, która odpowiada zdarzeniu OnDrawCell w sieci. Ta metoda ma wiele parametrów:

• Kol. I wiersz odnoszą się do aktualnie malowanej komórki.

• Rect to obszar komórki, który będziemy malować.

• Stan to stan komórki, zestaw trzech flag, które mogą być aktywne jednocześnie: gdSelected (komórka jest zaznaczona), gdFocused (komórka ma fokus wejściowy) i gdFixed (komórka znajduje się w stały obszar, który zwykle ma inny kolor tła). Ważne jest, aby znać stan komórki, ponieważ zwykle wpływa to na jej wyjście. Metoda DrawCell maluje tekst odpowiedniego elementu siatki, używając czcionki używanej przez kolumnę i rozmiaru użytego w wierszu. Oto listing tej metody:

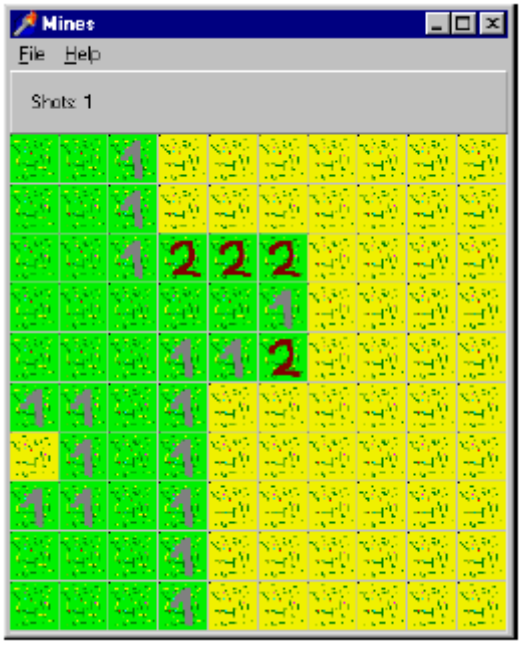
```
procedure TForm1.StringGrid1DrawCell (Sender: TObject;

Col, Row: Integer; Rect: TRect; State: TGridDrawState);

begin

// select a font, depending on the column

if (Col = 0) or (Row = 0) then

StringGrid1.Canvas.Font.Name := I

else

StringGrid1.Canvas.Font.Name :=

StringGrid1.Cells [Col, 0];

// select the size of the font, depending on the row
```

```
if Row = 0 then
StringGrid1.Canvas.Font.Size := 14
else
StringGrid1.Canvas.Font.Size := Row + 7;
// select the background color
if gdSelected in State then
StringGrid1.Canvas.Brush.Color := clHighlight
else if gdFixed in State then
StringGrid1.Canvas.Brush.Color := clBtnFace
else
StringGrid1.Canvas.Brush.Color := clWindow;
// output the text
StringGrid1.Canvas.TextRect (
Rect, Rect.Left, Rect.Top,
StringGrid1.Cells [Col, Row]);
// draw the focus
if gdFocused in State then
StringGrid1.Canvas.DrawFocusRect (Rect);
end;
```

Nazwa czcionki jest pobierana przez wiersz 0 tej samej kolumny. Rozmiar czcionki jest obliczany przez dodanie 7 do numeru wiersza. Ustalone kolumny używają pewnych wartości domyślnych. Po ustawieniu czcionki i jej rozmiaru program wybiera kolor tła komórki, w zależności od możliwych stanów: zaznaczony, stały lub normalny (to znaczy bez specjalnego stylu). Wartość flagi gdFocused stylu jest używana kilka linii później, aby narysować typowy prostokąt skupienia. Gdy wszystko jest skonfigurowane, program może wykonać pewne rzeczywiste wyjście, rysując tekst i, jeśli to konieczne, prostokąt skupienia, z dwoma ostatnimi instrukcjami metody StringGrid1DrawCell powyżej. Aby narysować tekst w komórce siatki, użyłem metody TextRect obszaru roboczego zamiast bardziej popularnej metody TextOut. Powodem jest to, że TextRect przycina wyjście do danego prostokąta, zapobiegając rysowaniu poza tym obszarem. Jest to szczególnie ważne w przypadku siatek, ponieważ wyjście komórki nie powinno przekraczać jej granic. Ponieważ malujemy na płótnie całej siatki, kiedy rysujemy komórkę, możemy w końcu uszkodzić także zawartość sąsiednich komórek. Jako ostatnią obserwację pamiętaj, że kiedy zdecydujesz się narysować zawartość komórki siatki, powinieneś nie tylko narysować domyślny obraz, ale także dostarczyć inny wynik dla wybranego elementu, poprawnie narysować fokus i tak dalej.

Saper w siatce

Komponent StringGrid używa tablicy Cells do przechowywania wartości elementów, a także posiada właściwość Objects do przechowywania danych niestandardowych dla każdej komórki. Zamiast tego komponent DrawGrid nie ma predefiniowanego magazynu. Z tego powodu następny przykład definiuje dwuwymiarową tablicę do przechowywania wartości komórek siatki - czyli pola gry. Przykład Mines to klon gry MineSweeper dołączonej do systemu Windows. Jeśli nigdy nie grałeś w tę grę, proponuję wypróbować ją i przeczytać jej zasady w pliku pomocy, ponieważ podam tylko podstawowy opis. Po uruchomieniu programu wyświetla puste pole (siatkę), w którym znajdują się ukryte miny. Klikając lewy przycisk myszy na komórce, sprawdzasz, czy w tej pozycji znajduje się kopalnia. Jeśli znajdziesz kopalnię, eksploduje, a gra się kończy. Przegrałeś. Jeśli w celi nie ma kopalni, program wskazuje liczbę min w ośmiu otaczających ją komórkach. Znając liczbę min w pobliżu celi, masz dobrą wskazówkę na kolejną turę. Aby pomóc ci dalej, gdy komórka ma zero min w okolicy, liczba min dla tych komórek jest automatycznie wyświetlana, a jeśli jeden z nich ma miny otaczające zero, proces jest powtarzany. Więc jeśli masz szczęście, jednym kliknięciem możesz odkryć dużą liczbę czystych komórek



Kiedy myślisz, że znalazłeś kopalnię, po prostu kliknij komórkę prawym przyciskiem myszy; to umieszcza tam flagę. Program nie mówi, czy twoje wnioskowanie jest poprawne; flaga jest tylko wskazówką dla przyszłych prób. Jeśli później zmienisz zdanie, możesz ponownie kliknąć komórkę prawym przyciskiem myszy, aby usunąć flagę. Kiedy znajdziesz wszystkie miny, wygrałeś i gra się kończy. Takie są zasady gry. Teraz musimy je zaimplementować, używając siatki rysunkowej jako punktu wyjścia. W tym przykładzie siatka jest stała i nie można jej zmienić rozmiaru ani zmodyfikować w żaden sposób w czasie wykonywania. W rzeczywistości ma kwadratowe komórki o wymiarach 30 × 30 pikseli, które będą używane do wyświetlania bitmap o tym samym rozmiarze. Kod tego programu jest złożony i nie jest łatwo znaleźć punkt wyjścia do jego opisania. Z tego powodu dodałem więcej komentarzy niż zwykle do kodu źródłowego (w plikach do pobrania), dzięki czemu można go przeglądać, aby zrozumieć, co robi. Niemniej jednak opiszę jego najważniejsze elementy. Przede wszystkim program dane są przechowywane w dwóch tablicach (zadeklarowanych jako prywatne pola formularza):

Display: array [0 .. NItems - 1, 0 .. NItems -1] of Boolean;

Map: array [0 .. NItems - 1, 0 .. NItems -1] of Char;

Pierwsza to tablica wartości logicznych, które wskazują, czy element powinien być wyświetlany, czy nie. Zauważ, że liczba wierszy i kolumn tej tablicy to NItems. Możesz dowolnie zmieniać tę stałą, ale powinieneś odpowiednio zmienić rozmiar siatki. Druga tablica, Mapa, utrzymuje pozycje min i flag oraz liczby okolicznych min. Używa kodów znaków zamiast właściwego typu danych wyliczeniowych, aby użyć cyfr 0–8 do wskazania liczby min wokół komórki. Oto lista kodów:

• M: Mine wskazuje pozycję miny , której użytkownik jeszcze nie znalazł.

• K: Znana mina wskazuje pozycję miny już znalezionej przez użytkownika i posiadającej flagę.

• W: Błędna mina wskazuje pozycję, w której użytkownik ustawił flagę, ale gdzie nie ma miny.

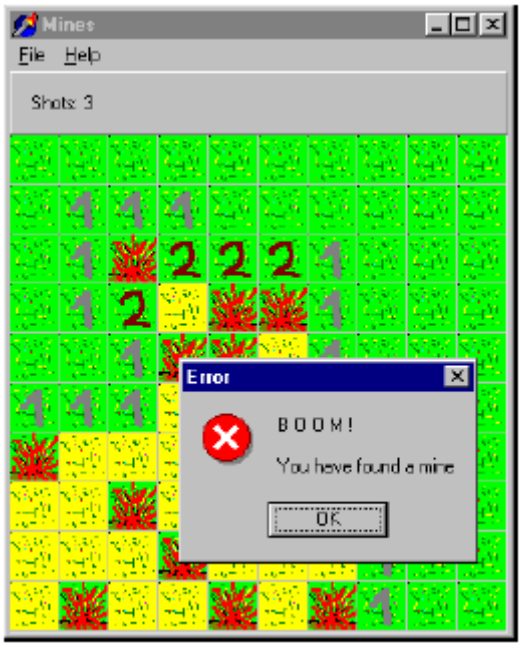
• 0 do 8: Liczba min wskazuje liczbę min w otaczających komórkach.

Pierwszą metodą do zbadania jest FormCreate, wykonywana podczas uruchamiania. Ta metoda inicjuje wiele pól klasy formularza, wypełnia dwie tablice wartościami domyślnymi (używając dwóch zagnieżdżonych pętli), a następnie ustawia miny w siatce. Dla liczby razy zdefiniowanej w stałej (tj. Liczby min) program dodaje nową kopalnię w losowej pozycji. Jeśli jednak była już kopalnia, kibel należy wykonać jeszcze raz, ponieważ ostateczna liczba min na mapie tablicy powinna być równa żądanemu numerowi. W przeciwnym razie program nigdy się nie zakończy, ponieważ sprawdza, kiedy liczba znalezionych min jest równa liczbie min dodanych do siatki. Oto kod pętli; może być wykonywany bardziej niż NMine razy, dzięki użyciu zmiennej całkowitej MinesToPlace, która jest zwiększana, gdy próbujemy umieścić kopalnię nad istniejącą kopalnią:

```
Randomize;

// place ‘NMines’ non-overlapping mines

MinesToPlace := NMines;

while MinesToPlace > 0 do

begin

X := Random (NItems);

Y := Random (NItems);

// if there isn’t a mine

if Map [X, Y] <> ‘M’ then

begin

// add a mine

Map [X, Y] := ‘M’;

Dec (MinesToPlace)

end;

end;
```

Ostatnia część kodu inicjującego oblicza liczbę otaczających min dla każdej komórki, która nie ma kopalni. Osiąga się to przez wywołanie procedury ComputeMines dla każdej komórki. Kod tej funkcji jest dość złożony, ponieważ musi uwzględniać szczególne przypadki min w pobliżu granicy siatki.

Efektem tego wywołania jest zapisanie w tablicy mapy znaku reprezentującego liczba kopalń otaczających każdą komórkę. Następną logiczną procedurą jest DrawGrid1MouseDown. Ta metoda najpierw oblicza komórkę, w której kliknięto mysz, z wywołaniem metody MouseToCell siatki. Następnie są trzy alternatywne części kodu: mały, gdy gra się skończy, a dwie pozostałe dla dwóch przycisków myszy. Po naciśnięciu lewego przycisku myszy program sprawdza, czy istnieje kopalnia (ukryta lub nie), a jeśli jest, wyświetla komunikat i kończy program eksplozją



Jeśli nie ma miny , program ustawia wartość wyświetlania komórki na True, a jeśli jest 0, uruchamia procedurę FloodZeros. Ta metoda wyświetla osiem elementów w pobliżu widocznej komórki o wartości 0, powtarzając tę operację w kółko, jeśli jedna z otaczających komórek ma również wartość 0. To wywołanie rekurencyjne jest złożone, ponieważ musisz zapewnić sposób na jego zakończenie . Jeśli w pobliżu znajdują się dwie komórki, z których każda ma wartość 0, każda z nich znajduje się w otaczającym obszarze drugiej, więc mogą one trwać wiecznie, aby poprosić drugą komórkę o wyświetlenie siebie i otaczających ją komórek. Ponownie, kod jest złożony i najlepszym sposobem na zbadanie go może być przejście przez debuger. Gdy użytkownik naciśnie prawy przycisk myszy, program zmieni status komórki. Działanie prawego przycisku myszy polega na przełączeniu flagi na ekranie, dzięki czemu użytkownik zawsze może usunąć istniejącą flagę, jeśli uważa, że wcześniejsza decyzja była błędna. Z tego powodu status komórki zawierającej kopalnię może zmienić się z M (ukryta kopalnia) na K (znana kopalnia) i odwrotnie; a status komórki bez kopalni może zmienić się z liczby na W (Błędna kopalnia) i odwrotnie. Po znalezieniu wszystkich kopalń program kończy się komunikatem gratulacyjnym. Bardzo ważny fragment kodu znajduje się na końcu metody odpowiedzi na zdarzenie OnMouseDown. Za każdym razem, gdy użytkownik kliknie komórkę i zmieni się jej zawartość, ta komórka powinna zostać przemalowana. Jeśli przemalujesz całą siatkę, program będzie wolniejszy. Z tego powodu użyłem funkcji API Windows InvalidateRect:

MyRect: = DrawGrid1.CellRect (Col, Row);

InvalidateRect (DrawGrid1.Handle, @MyRect, False);

Ostatnią ważną metodą jest DrawGrid1DrawCell. Użyliśmy już tej procedury malowania w ostatnim przykładzie, więc należy pamiętać, że jest ona wywoływana dla każdej komórki, która wymaga ponownego malowania. Zasadniczo metoda ta wyodrębnia kod odpowiadający komórce, który

pokazuje odpowiednią bitmapę załadowaną z pliku. Po raz kolejny przygotowałem bitmapę dla każdego z obrazów w nowym pliku zasobów, który jest zawarty w projekcie dzięki ulepszonemu Project Managerowi. Przypomnij sobie, że podczas korzystania z zasobów kod ma tendencję do bycia szybszym niż w przypadku używania oddzielnych plików, i znowu mamy do dyspozycji pojedynczy plik wykonywalny do dystrybucji. Mapy bitowe mają nazwy odpowiadające kodowi w siatce, ze znakiem („M") z przodu, ponieważ nazwa „0" byłaby nieprawidłowa. Bitmapy można załadować i narysować w komórce za pomocą tego kodu:

Bmp.LoadFromResourceName (HInstance, „M" + Kod);

DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);

Oczywiście ma to miejsce tylko wtedy, gdy komórka jest widoczna - to znaczy, jeśli Display jest True. W przeciwnym razie wyświetlana jest domyślna niezdefiniowana mapa bitowa. (Nazwa mapy bitowej to „UNDEF".). Ładowanie bitmap z zasobów za każdym razem wydaje się powolne, więc program mógł zapisać wszystkie bitmapy na liście w pamięci, tak jak wcześniej przykład World2. Jednak tym razem postanowiłem użyć innego, choć nieco mniej wydajnego podejścia: pamięci podręcznej. Ma to sens, ponieważ do przyspieszenia działania używamy już zasobów zamiast plików. Pamięć podręczna bitmap Mines jest niewielka, ponieważ ma tylko jeden element, ale jej obecność znacznie przyspiesza program. Program przechowuje ostatnio używaną bitmapę i jej kod; następnie, za każdym razem, gdy musi narysować nowy element, jeśli kod jest taki sam, używa buforowanej bitmapy. Oto nowa wersja powyższego kodu:

if not (Code = LastBmp) then

begin

Bmp.LoadFromResourceName (HInstance, 'M' + Code);

LastBmp := Code;

end;

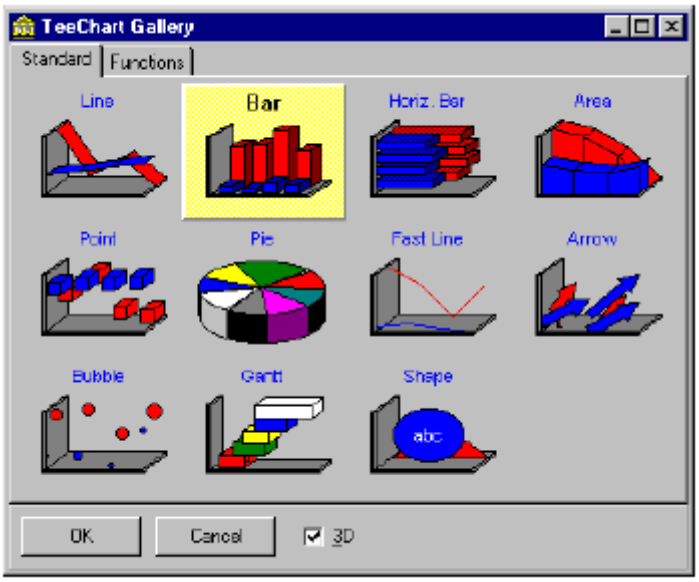
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);

Zwiększenie rozmiaru tej pamięci podręcznej z pewnością poprawi jej szybkość. Listę map bitowych można traktować jako dużą pamięć podręczną, ale jest to prawdopodobnie bezużyteczne, ponieważ niektóre mapy bitowe (o dużej liczbie) są rzadko używane. Jak widać, można wprowadzić pewne ulepszenia w celu przyspieszenia programu i wiele można zrobić, aby poprawić jego interfejs użytkownika. Jeśli zrozumiałeś tę wersję programu, myślę, że będziesz w stanie ją znacznie poprawić.

Korzystanie z TeeChart

TeeChart to oparty na VCL komponent do tworzenia wykresów zbudowany przez Davida Bernedę i licencjonowany firmie Borland do włączenia do wersji Delphi dla programistów i klientów / serwerów. Składnik TeeChart jest bardzo złożony: Delphi zawiera plik pomocy i inne materiały referencyjne dla tego komponentu, więc nie będę tracić czasu na wyświetlanie wszystkich jego funkcji. Zbuduj kilka przykładów. TeeChart występuje w trzech wersjach: komponent autonomiczny (na dodatkowej stronie palety komponentów), wersja obsługująca dane (na stronie Data Controls) oraz wersja raportu (na stronie QuickReport). Delphi Client / Server zawiera również kontrolę DecisionChart na stronie kostki decyzyjnej palety. Komponent TeeChart zapewnia podstawową strukturę dla wykresów, poprzez złożoną strukturę wykresów i klas serii oraz wizualny kontener dla wykresów (rzeczywista kontrola). Rzeczywiste wykresy są obiektami klasy TChartSeries lub klas pochodnych. Po umieszczeniu

komponentu TeeChart na formularzu powinieneś utworzyć jedną lub więcej serii. Aby to osiągnąć, możesz otworzyć Edytor składników wykresu: wybierz komponent, kliknij prawym przyciskiem myszy, aby wyświetlić menu lokalne projektanta formularzy, i wybierz polecenie Edytuj wykres. Teraz naciśnij przycisk Dodaj i wybierz wykres (lub serię), który chcesz dodać z wielu dostępnych



Jak tylko utworzysz nową serię, nowy obiekt podklasy TChartSeries zostanie dodany do twojego formularza. Jest to to samo zachowanie, co komponent MainMenu, który dodaje obiekty klasy TMenuItem do formularza. Następnie możesz edytować właściwości obiektu TSeries w Edytorze składników wykresu lub możesz wybrać obiekt TChartSeries w Inspektorze obiektów (za pomocą pola kombi Selektor obiektów) i edytować jego wiele właściwości. Różne podklasy TChartSeries - czyli różne rodzaje wykresów - mają różne właściwości i metody (choć niektóre z nich są wspólne dla więcej niż jednej podklasy). Pamiętaj, że wykres może mieć wiele serii: jeśli wszystkie są tego samego typu prawdopodobnie będą lepiej zintegrowane, jak w przypadku wielu prętów. W każdym razie możesz mieć złożony układ z wykresami różnych typów widocznymi jednocześnie. Czasami jest to niezwykle potężna opcja.

Budowanie pierwszego przykładu

Aby zbudować ten przykład, umieściłem komponent TeeChart w formularzu, a następnie po prostu dodałem cztery serie 3D Bar - czyli cztery obiekty klasy TBarSeries. Następnie ustawiam niektóre właściwości globalne, takie jak tytuł wykresu i tak dalej. Oto podsumowanie tych informacji, zaczerpnięte z tekstowego opisu formularza:

object Chart1: TChart

AnimatedZoom = True

Title.Text.Strings = (

‘Simple TeeChart Demo for Mastering Delphi’)

BevelOuter = bvLowered

object Series1: TBarSeries

SeriesColor = clRed

Marks.Visible = False

end

object Series2: TBarSeries

SeriesColor = clGreen

Marks.Visible = False

end

object Series3: TBarSeries

SeriesColor = clYellow

Marks.Visible = False

end

object Series4: TBarSeries

SeriesColor = clBlue

Marks.Visible = False

end

end

Następnie dodałem do formularza siatkę łańcuchów i przycisk oznaczony Aktualizuj. Ten przycisk służy do kopiowania wartości liczbowych siatki ciągów na wykres. Siatka jest oparta na macierzy 5 × 4, a także linii i kolumnie tytułów. Oto jego opis tekstowy:

object StringGrid1: TStringGrid

ColCount = 6

DefaultColWidth = 50

Options = [goFixedVertLine, goFixedHorzLine,

goVertLine, goHorzLine, goEditing]

ScrollBars = ssNone

OnGetEditMask = StringGrid1GetEditMask

end

Wartość 5 dla właściwości RowCount jest wartością domyślną i nie pojawia się w opisie tekstowym. (To samo dotyczy wartości 1 dla właściwości FixedCols i FixedRows.) Ważnym elementem tej siatki ciągów jest maska edycji używana przez wszystkie jej komórki. Jest to ustawiane za pomocą zdarzenia OnGetEditMask:

procedure TForm1.StringGrid1GetEditMask(Sender: TObject;

ACol, ARow: Longint; var Value: string);

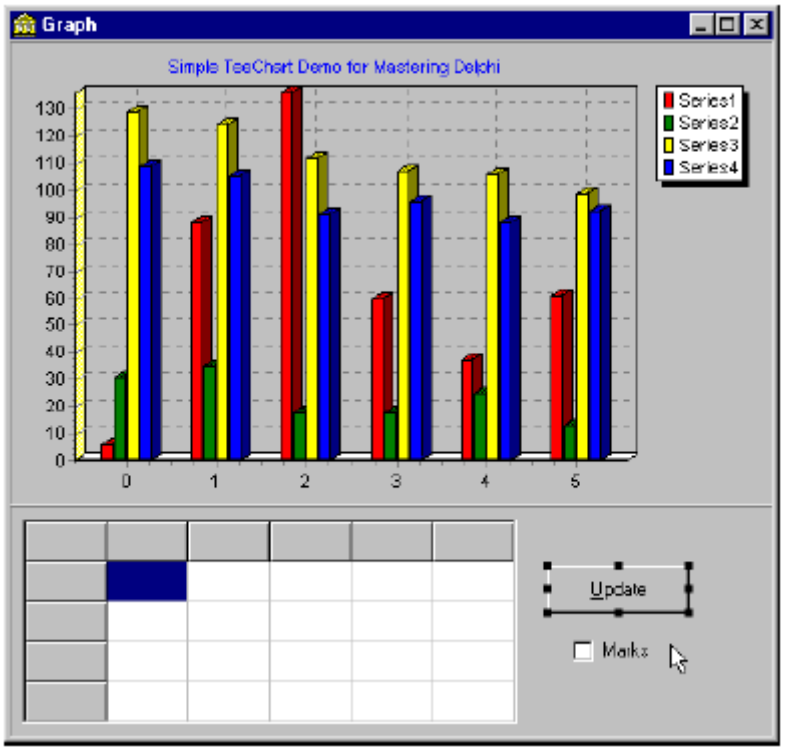
begin

// edit mask for the grid cells

Value := ‘09;0’;

end;

W rzeczywistości jest jeszcze jeden składnik, pole wyboru używane do przełączania widoczności znaków serii. (Znaki są małymi żółtymi znacznikami opisującymi każdą wartość; musisz je uruchomić, aby je zobaczyć.) Formularz można zobaczyć w czasie projektowania na rysunku. W tym przypadku seria jest wypełniona losowymi wartościami; jest to przyjemna cecha komponentu, ponieważ pozwala na podgląd wydruku bez wprowadzania rzeczywistych danych.



Dodawanie danych do wykresu

Teraz po prostu zainicjujemy dane siatki ciągów i skopiujemy ją do serii wykresu. Odbywa się to w programie obsługi zdarzenia OnCreate formularza. Ta metoda wypełnia stałe elementy siatki i nazwy serii, a następnie wypełnia część danych siatki ciągów, a na koniec wywołuje procedurę obsługi zdarzenia OnClick przycisku Aktualizuj, aby zaktualizować wykres:

procedure TForm1.FormCreate(Sender: TObject);

var

I, J: Integer;

begin

with StringGrid1 do

begin

{fills the fixed column and row,

```
and the chart series names}

for I := 1 to 5 do

Cells [I, 0] := Format ('Group%d', [I]);

for J := 1 to 4 do

begin

Cells [0, J] := Format ('Series%d', [J]);

Chart1.Series [J-1].Title := Format ('Series%d', [J]);

end;

// fills the grid with random values

Randomize;

for I := 1 to 5 do

for J := 1 to 4 do

Cells [I, J] := IntToStr (Random (100));

end; // with

// update the chart

UpdateButtonClick (Self);

end;
```

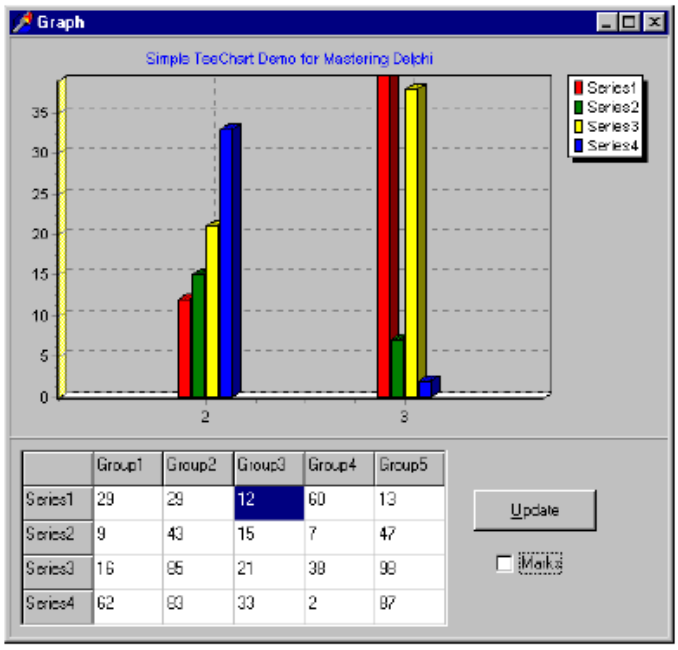
Możemy uzyskać dostęp do serii za pomocą nazwy komponentu (jako Seria1) lub za pomocą właściwości tablicy serii na wykresie, jak na wykresie 1.Series [J-1]. W tym wyrażeniu zauważ, że rzeczywiste dane w siatce łańcuchowej zaczynają się od wiersza i kolumny pierwszej - pierwszy wiersz i kolumna, wskazane przez indeks zerowy, są używane dla stałych elementów - podczas gdy tablica szeregów wykresów jest oparta na zero. Inny przykład aktualizacji każdej serii jest obecny w procedurze obsługi zdarzenia OnClick dla pola wyboru; ta metoda przełącza widoczność znaków:

```
procedure TForm1.ChBoxMarksClick(Sender: TObject);

var

I: Integer;

begin

for I := 1 to 4 do

Chart1.Series [I-1].Marks.Visible :=

ChBoxMarks.Checked;

end;
```

Ale naprawdę interesujący kod znajduje się w metodzie UpdateButtonClick, która aktualizuje wykres. Aby to osiągnąć, program najpierw usuwa istniejące dane każdego wykresu, a następnie dodaje nowe dane (lub punkty danych, aby użyć terminu żargonowego):

```
procedure TForm1.UpdateButtonClick(Sender: TObject);

var

I, J: Integer;

begin

for I := 1 to 4 do

begin

Chart1.Series [I-1].Clear;

for J := 1 to 5 do

Chart1.Series [I-1].Add (

StrToInt (StringGrid1.Cells [J, I]),

'', Chart1.Series [I-1].SeriesColor);

end;

end;
```

Parametry metody Add (używane, gdy nie chcesz określać wartości X, ale tylko wartość Y) to rzeczywista wartość, etykieta i kolor. W tym przykładzie etykieta nie jest używana, więc po prostu ją pominąłem. Mogłem użyć wartości domyślnej, clTeeColor, aby uzyskać odpowiedni kolor serii. Możesz użyć określonych kolorów, aby wskazać różne zakresy danych. Po utworzeniu wykresu TeeChart umożliwia wiele opcji wyświetlania. Możesz łatwo powiększyć widok (wystarczy wskazać obszar lewym przyciskiem myszy), pomniejszyć widok (używając myszy w przeciwnym kierunku, przeciągając w kierunku lewego górnego rogu) i użyć prawego przycisku myszy, aby przesunąć widok. Możesz zobaczyć przykład powiększenia na rysunku



Dynamiczne tworzenie serii

Przykład Graph1 pokazuje niektóre możliwości komponentu TeeChart, ale jest oparty na pojedynczym, ustalonym typie wykresu. Mogłem go poprawić, pozwalając na pewne dostosowanie kształtu pionowych prętów; zamiast tego wybrałem bardziej ogólne podejście, pozwalające użytkownikowi wybrać różne rodzaje serii (wykresy). Komponent TeeChart początkowo ma takie same atrybuty, jak w poprzednim przykładzie. Ale formularz ma teraz cztery pola kombi, po jednym dla każdego wiersza siatki ciągów. Każde pole kombi ma cztery wartości (Linia, Pasek, Powierzchnia i Punkt), odpowiadające czterem typom serii, które chcę obsługiwać. Aby obsłużyć te pola kombi w bardziej elastyczny sposób w kodzie, dodałem tablicę tych formantów do prywatnych pól formularza:

```
private

Combos: array [0..3] of TComboBox;
```

Ta tablica jest wypełniona rzeczywistym składnikiem w metodzie FormCreate, która również wybiera początkowy element każdego z nich. Oto nowy kod FormCreate:

```
// fill the Combos array

Combos [0] := ComboBox1;

Combos [1] := ComboBox2;

Combos [2] := ComboBox3;

Combos [3] := ComboBox4;

// show the initial chart type

for I := 0 to 3 do

Combos [I].ItemIndex := 1;
```

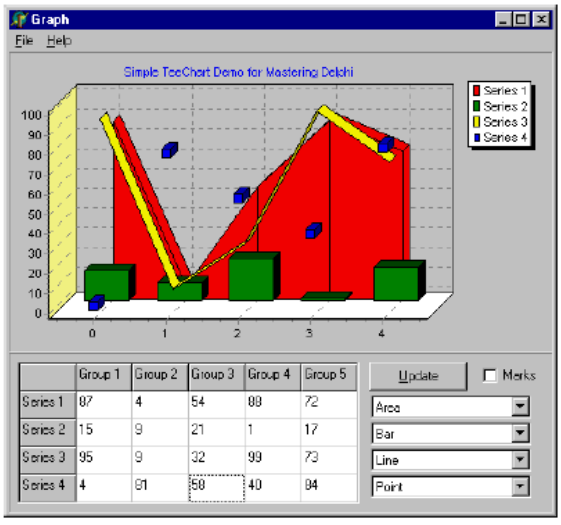
Wszystkie te pola kombi współdzielą tę samą procedurę obsługi zdarzenia OnClick, która niszczy każdą bieżącą serię wykresu, tworzy nowe zgodnie z wymaganiami, a następnie aktualizuje ich właściwości i dane:

```
procedure TForm1.ComboChange(Sender: TObject);

var

I: Integer;

SeriesClass: TChartSeriesClass;

NewSeries: TChartSeries;

begin

// destroy the existing series (in reverse order)

for I := 3 downto 0 do

Chart1.Series [I].Free;

// create the new series

for I := 0 to 3 do

begin
```

```
case Combos [I].ItemIndex of

0: SeriesClass := TLineSeries;

1: SeriesClass := TBarSeries;

2: SeriesClass := TAreaSeries;

else // 3: and default

SeriesClass := TPointSeries;

end;

NewSeries := SeriesClass.Create (self);

NewSeries.ParentChart := Chart1;

NewSeries.Title :=

Format ('Series %d', [I + 1]);

end;

// update the marks and update the data

ChBoxMarksClick (self);

UpdateButtonClick (self);

Modified := True;

end;
```

Centralną częścią tego kodu jest instrukcja case, która przechowuje nową klasę w zmiennej referencyjnej klasy SeriesClass, używanej do tworzenia nowych obiektów serii i ustawiania ParentChart i Title każdego z nich. Mogłem również użyć wywołania metody AddSeries wykresu w każdej gałęzi przypadku, a następnie ustawić Tytuł za pomocą innej pętli for. W rzeczywistości wywołanie takie jak Chart1.AddSeries (TBarSeries.Create (self)); tworzy obiekty serii i jednocześnie ustawia swój wykres nadrzędny. Zauważ, że ta nowa wersja programu pozwala na zmianę typu wykresu dla każdej serii niezależnie. Możesz zobaczyć przykład wynikowego efektu na rysunku

---



Wreszcie przykład Graph2 obsługuje zapisywanie bieżących danych, które wyświetla na pliku i ładuje istniejące pliki. Program ma zmodyfikowaną zmienną boolowską, używaną do śledzenia, czy użytkownik zmienił jakiekolwiek dane, i monituje użytkownika o potwierdzenie zamknięcia formularza, gdy dane uległy zmianie. Obsługa plików jest oparta na strumieniach i nie jest szczególnie skomplikowany, ponieważ liczba elementów do zapisania jest stała (wszystkie pliki mają ten sam rozmiar). Oto dwie metody związane z elementami menu Otwórz i Zapisz:

```
procedure TForm1.Open1Click(Sender: TObject);

var

LoadStream: TFileStream;

I, J, Value: Integer;

begin

if OpenDialog1.Execute then

begin

CurrentFile := OpenDialog1.Filename;

Caption := 'Graph [' + CurrentFile + ']';

// load from the current file

LoadStream := TFileStream.Create (

CurrentFile, fmOpenRead);

try

// read the value of each grid element

for I := 1 to 5 do

for J := 1 to 4 do

begin

LoadStream.Read (Value, sizeof (Integer));

StringGrid1.Cells [I, J] := IntToStr(Value);

end;

// load the status of the checkbox and the combo boxes

LoadStream.Read (Value, sizeof (Integer));

ChBoxMarks.Checked := Boolean(Value);

for I := 0 to 3 do

begin

LoadStream.Read (Value, sizeof (Integer));

Combos [I].ItemIndex := Value;
```

```
end;
finally
LoadStream.Free;
end;
// fire udpate events
ChBoxMarksClick (Self);
ComboChange (Self);
UpdateButtonClick (Self);
Modified := False;
end;
end;
procedure TForm1.Save1Click(Sender: TObject);
var
SaveStream: TFileStream;
I, J, Value: Integer;
begin
if Modified then
if CurrentFile = '' then // call save as
SaveAs1Click (Self)
else
begin
// save to the current file
SaveStream := TFileStream.Create (
CurrentFile, fmOpenWrite or fmCreate);
try
// write the value of each grid element
for I := 1 to 5 do
for J := 1 to 4 do
begin
Value := StrToIntDef (Trim (
StringGrid1.Cells [I, J]), 0);
```

```
SaveStream.Write (Value, sizeof (Integer));
end;
// save check box and combo boxes
Value := Integer (ChBoxMarks.Checked);
SaveStream.Write (Value, sizeof (Integer));
for I := 0 to 3 do
begin
Value := Combos [I].ItemIndex;
SaveStream.Write (Value, sizeof (Integer));
end;
Modified := False;
finally
SaveStream.Free;
end;
end;
end;
```

Wykres bazy danych w sieci

Możemy zastosować podejście do zwracania złożonego i dynamicznego wykresu zbudowanego ze składnika TDBChart. Używanie tego komponentu w pamięci jest nieco bardziej skomplikowane niż ustawienie wszystkich jego właściwości w czasie projektowania, ponieważ będziesz musiał ustawić właściwości w kodzie Pascala. (Nie można użyć komponentu wizualnego, takiego jak DBChart, w module WWW lub innym module danych). W aplikacji WebChart ISAPI użyłem tabeli Country.DB do stworzenia wykresu kołowego z obszarem i populacją krajów amerykańskich. Dwa wykresy są generowane przez dwie różne akcje, wskazane przez ścieżki / populację i / obszar. Ponieważ większość kodu jest używana więcej niż jeden raz, zebrałem go w zdarzeniach OnCreate i OnAfterDispatch modułu WebModule. Moduł danych ma obiekt tabeli, który jest prawidłowo inicjowany w czasie projektowania i trzy pola prywatne:

```
private
Chart: TDBChart;
Series: TPieSeries;
Image: TImage;
```

Obiekty odpowiadające tym polom są tworzone wraz z modułem WWW (i używane przez kolejne wywołania):

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
begin
```

```
// open the database table
Table1.Open;
// create the chart
Chart := TDBChart.Create (nil);
Chart.Width := 600;
Chart.Height := 400;
Chart.AxisVisible := False;
Chart.Legend.Visible := False;
Chart.BottomAxis.Title.Caption := 'Name';
// create the pie series
Series := TPieSeries.Create (Chart);
Series.ParentChart := Chart;
Series.DataSource := Table1;
Series.XLabelsSource := 'Name';
Series.OtherSlice.Style := poBelowPercent;
Series.OtherSlice.Text := 'Others';
Series.OtherSlice.Value := 2;
Chart.AddSeries (Series);
// create the memory bitmap
Image := TImage.Create (nil);
Image.Width := Chart.Width;
Image.Height := Chart.Height;
end;
```

Następnym krokiem jest wykonanie procedury obsługi określonej akcji, która ustawia serię wykresów kołowych na konkretne pole danych i aktualizuje kilka podpisów:

```
procedure TWebModule1.WebModule1ActionPopulationAction(
Sender: TObject; Request: TWebRequest;
Response: TWebResponse; var Handled: Boolean);
begin
// set specific values
Chart.Title.Text.Clear;
```

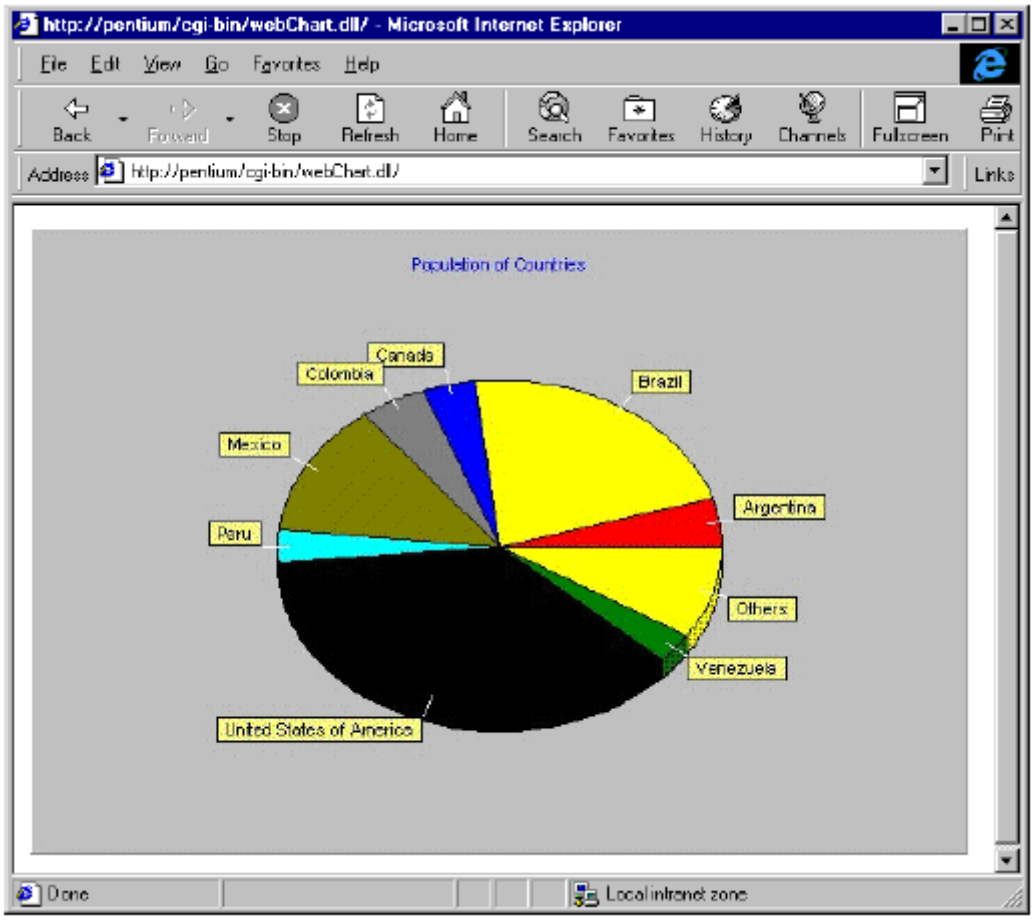
```
Chart.Title.Text.Add ('Population of Countries');

Chart.LeftAxis.Title.Caption := 'Population';

Series.Title := 'Population';

Series.PieValues.ValueSource := 'Population';

end;
```

Tworzy to odpowiednią pamięć DBChart w pamięci. Ostatnim krokiem, znów wspólnym dla obu akcji, jest zapisanie wykresu w obrazie bitmapowym, a następnie sformatowanie go jako JPEG w strumieniu, aby później powrócić z aplikacji po stronie serwera. Kod jest w rzeczywistości podobny do tego z poprzedniego przykładu:

```
procedure TWebModule1.WebModule1AfterDispatch(

Sender: TObject; Request: TWebRequest;

Response: TWebResponse; var Handled: Boolean);

var

Jpeg: TJpegImage;

MemStr: TMemoryStream;

begin

// paint the chart on the memory bitmap

Chart.Draw (Image.Canvas, Image.BoundsRect);

// create the jpeg and copy the image to it

Jpeg := TJpegImage.Create;

try

Jpeg.Assign (Image.Picture.Bitmap);

MemStr := TMemoryStream.Create;

// save to a stream and return it

Jpeg.SaveToStream (MemStr);

MemStr.Position := 0;

Response.ContentType := 'image/jpeg';

Response.ContentStream := MemStr;

Response.SendResponse;

finally

Jpeg.Free;

end;
```

end;

Wynik, widoczny na rysunku , jest z pewnością interesujący. Opcjonalnie można rozszerzyć tę aplikację, podłączając ją do tabeli HTML zawierającej dane bazy danych. Wystarczy napisać program z główną akcją zwracającą tabelę HTML i odwołanie do osadzonej grafiki, która zostanie zwrócona przez drugą aktywację biblioteki DLL ISAPI inną ścieżką.



Używanie metaplików

Formaty bitmapowe opisane wcześniej w tej części przechowują status każdego piksela bitmapy, chociaż zazwyczaj kompresują informacje. Całkowicie inny typ formatu graficznego jest reprezentowany przez formaty wektorowe. W tym przypadku plik przechowuje informacje wymagane do odtworzenia obrazu, takie jak punkt początkowy i końcowy każdej linii lub matematyka definiująca krzywą. Istnieje wiele różnych formatów plików zorientowanych na wektor, ale jedynym obsługiwanym przez system operacyjny Windows jest Windows Metafile Format (WMF). Format ten został rozszerzony w Win32 do formatu Extended Metafile Format (EMF), który przechowuje dodatkowe informacje związane z trybami mapowania i układem współrzędnych. Metaplik AWindows jest w zasadzie serią wywołań funkcji podstawowych GDI. Po zapisaniu sekwencji połączeń możesz je odtworzyć, odtwarzając grafika. Delphi obsługuje metapliki Windows poprzez klasy TMetafile i TMetaFileCanvas, więc bardzo łatwo jest zbudować przykład. Klasa TMetafile służy do obsługi samego pliku, metod ładowania i zapisywania plików oraz właściwości określających kluczowe funkcje pliku. Jednym z nich jest właściwość Enhanced, która określa typ formatu metapliku. Zauważ, że gdy system Windows odczytuje plik, właściwość Enhanced jest ustawiana w zależności od rozszerzenia pliku - WMF dla metaplików Windows 3.1 i EMF dla Rozszerzone metapliki Win32. Aby wygenerować metaplik,

możesz użyć obiektu klasy TMetafileCanvas, połączonego z plikiem poprzez jego konstruktory, jak pokazano w następującym fragmencie kodu:

```
Wmf := TMetafile.Create;

WmfCanvas := TMetafileCanvas.CreateWithComment(

Wmf, 0, 'Marco', 'Demo metafile');
```

Po utworzeniu dwóch obiektów można malować obiekt płótna zwykłymi wywołaniami, a na końcu zapisać połączony metaplik w pliku fizycznym. Po utworzeniu metapliku (nowego, który właśnie utworzyłeś lub jednego, który zbudowałeś w innym programie) możesz pokazać go w komponencie Image lub po prostu wywołać metody Draw lub StretchDraw dowolnego płótna. W przykładzie WmfDemo napisałem prosty kod, aby pokazać podstawy tego podejścia. Obsługa zdarzenia OnCreate formularza tworzy rozszerzony metaplik, pojedynczy obiekt, który jest używany zarówno do operacji odczytu, jak i zapisu:

```
procedure TForm1.FormCreate(Sender: TObject);

begin

Wmf := TMetafile.Create;

Wmf.Enhanced := True;

Randomize;

end;
```

Forma programu ma dwa przyciski i komponenty PaintBox oraz pole wyboru. Pierwszy przycisk tworzy metaplik, generując serię częściowo losowych linii. Wynik jest wyświetlany zarówno w pierwszym PaintBoxie, jak i zapisany w stałym pliku:

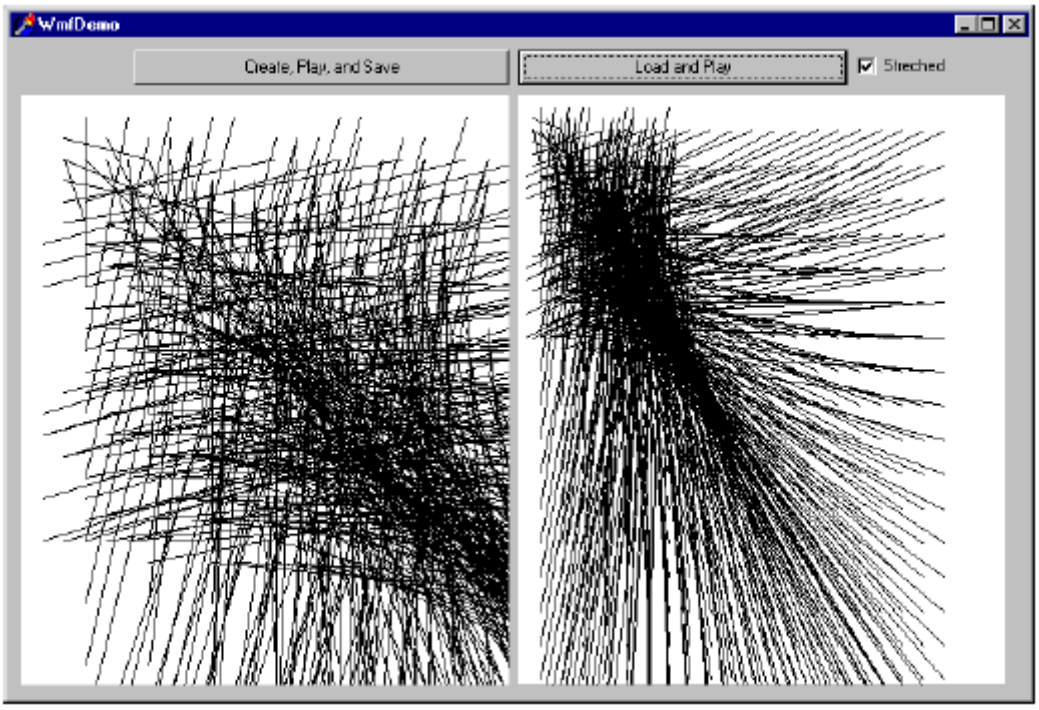
```
procedure TForm1.BtnCreateClick(Sender: TObject);

var

WmfCanvas: TMetafileCanvas;

X, Y: Integer;

begin

// create the virtual canvas

WmfCanvas := TMetafileCanvas.CreateWithComment(

Wmf, 0, 'Marco', 'Demo metafile');

try

// clear the background

WmfCanvas.Brush.Color := clWhite;

WmfCanvas.FillRect (WmfCanvas.ClipRect);

// draws 400 lines
```

```
for X := 1 to 20 do
for Y := 1 to 20 do
begin
WmfCanvas.MoveTo (15 * (X + Random (3)), 15 * (Y + Random (3)));
WmfCanvas.LineTo (45 * Y, 45 * X);
end;
finally
// end the drawing operation
WmfCanvas.Free;
end;
// show the current drawing and save it
PaintBox1.Canvas.Draw (0, 0, Wmf);
Wmf.SaveToFile (ExtractFilePath (
Application.ExeName) + 'test.emf');
end;
```

Ponowne ładowanie i malowanie metapliku jest jeszcze prostsze:

```
procedure TForm1.BtnLoadClick(Sender: TObject);
begin
// load the metafile
Wmf.LoadFromFile (ExtractFilePath (
Application.ExeName) + 'test.emf');
// draw or stretch it
if cbStretched.Checked then
PaintBox2.Canvas.StretchDraw (PaintBox2.Canvas.ClipRect, Wmf)
else
PaintBox2.Canvas.Draw (0, 0, Wmf);
end;
```

Zauważ, że możesz odtworzyć dokładnie ten sam rysunek, ale także zmodyfikować go za pomocą wywołania StretchDraw. (Rezultat tej operacji jest widoczny na rysunku) Ta operacja różni się od rozciągania mapy bitowej, która zwykle degraduje lub modyfikuje obraz, ponieważ tutaj skalujemy poprzez zmianę mapowania współrzędnych. Oznacza to, że podczas drukowania metapliku można go powiększyć, aby wypełnić całą stronę z dość dobrym efektem, co jest bardzo trudne do wykonania z mapą bitową.



Obracanie tekstu

W tym rozdziale omówiliśmy wiele różnych przykładów użycia bitmap i stworzyliśmy grafikę wielu typów. Jednak najważniejszym typem grafiki, z którym zwykle mamy do czynienia w aplikacjach Delphi, jest tekst. W rzeczywistości, nawet gdy pokazuje etykietę lub tekst pola edycji, Windows nadal maluje go tak samo, jak każdy inny element graficzny. Przedstawiłem przykład malowania czcionek wcześniej w tym rozdziale w przykładzie FontGrid. Teraz wracam do tego tematu z nieco bardziej niezwykłym podejściem. Podczas malowania tekstu w systemie Windows nie ma możliwości wskazania kierunku czcionki: system Windows rysuje tekst tylko poziomo. Aby być dokładnym, Windows rysuje tekst w kierunku obsługiwanym przez jego czcionkę, która domyślnie jest pozioma. Na przykład możemy zmienić tekst wyświetlany przez składniki na formularzu, modyfikując czcionkę samego formularza, tak jak to zrobiłem w przykładzie SideText. Właściwie nie można modyfikować czcionki, ale można utworzyć nową, podobną do istniejącej czcionki:

```
procedure TForm1.FormCreate(Sender: TObject);

var

ALogFont: TLogFont;

hFont: THandle;

begin

ALogFont.lfHeight := Font.Height;

ALogFont.lfWidth := 0;

ALogFont.lfEscapement := -450;

ALogFont.lfOrientation := -450;

ALogFont.lfWeight := fw_DemiBold;
```

```
ALogFont.lfItalic := 0; // false

ALogFont.lfUnderline := 0; // false

ALogFont.lfStrikeOut := 0; // false

ALogFont.lfCharSet := Ansi_CharSet;

ALogFont.lfOutPrecision := Out_Default_Precis;

ALogFont.lfClipPrecision := Clip_Default_Precis;

ALogFont.lfQuality := Default_Quality;

ALogFont.lfPitchAndFamily := Default_Pitch;

StrCopy (ALogFont.lfFaceName, PChar (Font.Name));

hFont := CreateFontIndirect (ALogFont);

Font.Handle := hFont;

end;
```

Ten kod wywołał pożądany efekt na etykiecie formularza przykładu, ale jeśli dodasz do niego inne komponenty, tekst będzie na ogół drukowany poza widoczną częścią komponentu. Innymi słowy, musisz dostarczyć tego typu wsparcie w ramach komponentów, jeśli chcesz, aby wszystko wyświetlało się poprawnie. Jednak w przypadku etykiet można uniknąć pisania nowego komponentu; zamiast tego po prostu zmień czcionkę powiązaną z kanwą formularza (nie na cały formularz) i użyj standardowych metod rysowania tekstu. Przykład SideText zmienia czcionkę Canvas w metodzie OnPaint, która jest podobna do OnCreate:

```
procedure TForm1.FormPaint(Sender: TObject);

var

ALogFont: TLogFont;

hFont: THandle;

begin

ALogFont.lfHeight := Font.Height;

ALogFont.lfEscapement := 900;

ALogFont.lfOrientation := 900;

...

hFont := CreateFontIndirect (ALogFont);

Canvas.Font.Handle := hFont;

Canvas.TextOut (0, ClientHeight, 'Hello');

end;
```

Orientacja czcionki jest również modyfikowana przez trzecią procedurę obsługi zdarzeń, powiązaną z zegarem. Jego efektem jest obracanie postaci w czasie, a jej kod jest bardzo podobny do powyższej procedury, z wyjątkiem kodu określającego wychylenie czcionki (kąt obrotu czcionki):

ALogFont.lfEscapement: = - (GetTickCount div 10) mod 3600;

Dzięki tym trzem różnym technikom obracania czcionek (podpis etykiety, malowany tekst, obracający się z czasem tekst) forma programu SideText w czasie wykonywania wygląda jak na rysunku