Recursos Gráficos do Delphi

No Capítulo 6, já apresentamos o objeto `Canvas`, o processo de pintura do Windows e o evento `OnPaint`. Neste capítulo, começaremos a partir desse ponto e continuaremos a abordar os recursos gráficos, seguindo várias direções diferentes. O código adicional discutido aqui está incluído no CD do *Dominando o Delphi 6*, nas subpastas da pasta bônus. Alguns exemplos citados, já vistos em outros capítulos do livro, estão nas pastas dos respectivos capítulos.

Começaremos com o desenvolvimento de um programa complexo para demonstrar como o modelo de pintura do Windows funciona. Em seguida, focalizaremos alguns componentes gráficos, como os botões gráficos e as grades. Ainda nesta parte do capítulo, também incluiremos alguma animação nos controles.

Por fim, discutiremos o uso de bitmaps, abordando alguns recursos avançados para rápida representação gráfica, meta-arquivos, o componente TeeChart (incluindo seu uso na Web) e mais alguns tópicos relacionados ao problema geral dos recursos gráficos.

Desenhando em um Formulário

No Capítulo 6, vimos que é possível pintar diretamente na superfície de um formulário, em resposta a um evento de mouse. Para ver esse comportamento, basta criar um novo formulário com o seguinte manipulador do evento `OnMouseDow`:

```
procedure TForm1.FormMouseDown(Sender: TObject;
   Button: TmouseButton; Shift: TShiftState; X, Y: Integer);
begin
   Canvas.Ellipse (X-10, Y-10, X+10, Y+10);
end;
```

O programa *parece* funcionar muito bem, mas não funciona. Todo clique de mouse produz um novo círculo, mas, se você minimizar o formulário, todos eles desaparecerão. Até mesmo se você cubrir apenas uma parte de seu formulário com outra janela, as figuras por trás desse outro formulário desaparecerão, e você poderá acabar com os círculos parcialmente pintados.

Conforme detalhamos no Capítulo 6, esse desenho direto não é suportado automaticamente pelo Windows. A estratégia padrão é armazenar o pedido de pintura no evento `OnMouseDown` e depois reproduzir a saída no evento `OnPaint`. Esse evento, na verdade, é chamado pelo sistema sempre que o formulário exige uma *repintura*. Entretanto, você precisará forçar essa ativação chamando os métodos

Invalidate ou Repaint no manipulador do evento do mouse. Em outras palavras, o Windows sabe quando o formulário precisa ser repintado devido a uma operação de sistema (como a colocação de outra janela na frente de seu formulário), mas seu programa deve notificar o sistema do momento em que a pintura é exigida por causa da entrada de usuário ou outras operações de programa.

As Ferramentas de Desenho

Todas as operações de saída no Windows ocorrem através do uso de objetos da classe TCanvas. As operações de saída normalmente não especificam cores e elementos semelhantes, mas usam as ferramentas de desenho atuais do canvas. Aqui está uma lista dessas ferramentas de desenho (ou *objetos GDI* — Graphics Device Interface —, uma das bibliotecas de sistema do Windows):

- A propriedade Brush determina a cor das superfícies englobadas. O pincel é usado para preencher figuras fechadas, como círculos ou retângulos. As propriedades de um pincel são Color, Style e, opcionalmente, Bitmap.
- A propriedade Pen determina a cor e o tamanho das linhas e das bordas das figuras. As propriedades de uma pena são Color, Width e Style, o que inclui várias linhas pontilhadas e tracejadas (disponíveis apenas se a propriedade Width for igual a 1 pixel). Outra subpropriedade relevante de Pen é Mode, que indica como a cor da pena modifica a cor da superfície de desenho. O padrão é simplesmente usar a cor da pena (com o estilo pmCopy), mas também é possível mesclar as duas cores de muitas maneiras diferentes e reverter a cor atual da superfície de desenho.
- A propriedade Font determina a fonte usada para escrever texto no formulário, usando o método TextOut do canvas. Uma fonte tem propriedades Name, Size, Style, Color etc.

DICA Os programadores experientes no Windows devem notar que um canvas do Delphi tecnicamente representa um contexto de dispositivo do Windows. Os métodos da classe TCanvas são semelhantes às funções de GDI da API do Windows. Você pode chamar métodos GDI extras usando a propriedade Handle do canvas, que é um handle de tipo HDC.

Cores

Pincéis, penas e fontes (assim como formulários e a maioria dos outros componentes) têm uma propriedade Color. Entretanto, para mudar a cor de um elemento corretamente, usando cores não-padrão (como as constantes de cor no Delphi), você deve saber como o Windows trata cores. Teoricamente, o Windows usa cores RGB de 24 bits. Isso significa que você pode usar 256 valores diferentes para cada uma das três cores básicas (vermelho, verde e azul), obtendo 16 milhões de tonalidades diferentes.

No entanto, você ou seus usuários podem ter uma placa de vídeo que não consegue apresentar tal variedade de cores, embora isso seja cada vez mais raro. Nesse caso, o Windows usa uma técnica chamada *dithering* (*pontilhamento*), que consiste basicamente no uso de um número de pixels das cores disponíveis para simular a solicitada; ou ela aproxima a cor, usando a combinação mais próxima disponível. Para a cor de um pincel (e para a cor de fundo de um formulário, que é na verdade baseada em um pincel), o Windows usa a técnica de pontilhamento; para a cor de uma pena ou fonte, ele usa a cor mais próxima disponível.

Em termos de penas, você pode ler (mas não alterar) a posição atual com a propriedade PenPos do canvas. A posição da pena determina o ponto inicial da próxima linha que o programa desenhará, usando o método LineTo. Para alterá-la, você pode usar o método MoveTo do canvas. Outras propriedades do canvas afetam linhas e cores também. Exemplos interessantes são CopyMode e ScaleMode. Ou-

tras propriedades possíveis de manipular diretamente para alterar a saída é o array `Pixels`, que você pode usar para acessar (ler) ou alterar (gravar) a cor de qualquer ponto específico na superfície do formulário. Conforme veremos no exemplo BmpDraw, as operações por pixel são muito lentas na GDI, comparadas com o acesso à linha disponível através da propriedade `ScanLines`.

Finalmente, lembre-se de que os valores de `TColor` do Delphi nem sempre correspondem aos valores RGB puros da representação nativa do Windows (`COLORREF`), devido às constantes de cor do Delphi. Você sempre pode converter uma cor do Delphi para o valor RGB, usando a função `ColorToRGB`. Você pode encontrar os detalhes da representação do Delphi na entrada *TColor type* da ajuda.

Desenhando Figuras Geométricas

Agora, queremos estender o exemplo MouseOne construído no Capítulo 9 e transformá-lo no aplicativo Shapes. Nesse novo programa, queremos usar a estratégia de *armazenar e desenhar* com várias figuras, manipular atributos de cor e pena e fornecer uma base para extensões futuras.

Como você precisa se lembrar da posição e dos atributos de cada figura, pode criar um objeto para cada figura que precisa armazenar e pode manter os objetos em uma lista (para sermos mais precisos, a lista armazenará referências aos objetos, que são alocados em áreas de memória separadas). Definimos uma classe-base para as figuras e duas classes herdadas que contêm o código de pintura para os dois tipos de figuras que queremos manipular — retângulos e elipses.

A classe-base tem algumas propriedades, que simplesmente lêem os campos e gravam os valores correspondentes com métodos simples. Note que as coordenadas podem ser lidas usando-se a propriedade `Rect`, mas devem ser modificadas usando-se as quatro propriedades posicionais. O motivo é que, se você incluir uma parte `write` na propriedade `Rect`, poderá acessar o retângulo como um todo, mas não suas subpropriedades específicas. Aqui estão as declarações das três classes:

```
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;
```

A maior parte do código dos métodos é muito simples. O único código relevante está nos três procedimentos `Paint`:

```
procedure TBaseShape.Paint (Canvas: TCanvas);
begin
  // configura os atributos
  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;
```

Todo esse código é armazenado na unidade ShapesH (Shapes Hierarchy). Para armazenar uma lista de figuras, o formulário tem um membro de dados de objeto `TList`, chamado `ShapesList`, que é inicializado no manipulador do evento `OnCreate` e destruído no final; o destrutor também libera todos os objetos da lista (em ordem inversa, para evitar a atualização dos dados da lista internos com muita freqüência):

```
procedure TShapesForm.FormCreate(Sender: TObject);
begin
  ShapesList := TList.Create;
end;

procedure TShapesForm.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  // exclui cada objeto
  for I := ShapesList.Count – 1 downto 0 do
       TBaseShape (ShapesList [I]).Free;
  ShapesList.Free;
end;
```

O programa inclui um novo objeto na lista sempre que o usuário inicia a operação de desenho. Como o objeto não está completamente definido, o formulário mantém uma referência a ele no campo CurrShape. Note que o tipo de objeto criado depende do status das teclas do mouse:

```
procedure TShapesForm.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if Button = mbLeft  then
  begin
    // ativa o arrasto
    fDragging := True;
    SetCapture (Handle);

    // cria o objeto correto
      if ssShift in Shift  then
      CurrShape := TEllShape.Create
    else
      CurrShape := TRectShape.Create;

        // configura o estilo e as cores
        CurrShape.PenSize := Canvas.Pen.Width;
        CurrShape.PenColor := Canvas.Pen.Color;
        CurrShape.BrushColor := Canvas.Brush.Color;

        // configura a posição inicial
        CurrShape.Left := X;
        CurrShape.Top := Y;
        CurrShape.Right := X;
        CurrShape.Bottom := Y;
        Canvas.DrawFocusRect (CurrShape.Rect);

        // inclui na lista
        ShapesList.Add (CurrShape);
      end;
    end;
```

Durante a operação de arrasto, desenhamos a linha correspondente à figura, como fizemos no exemplo MouseOne:

```
procedure TShapesForm.FormMouseMove(Sender: TObject; Shift:
TShiftState;
   X, Y: Integer);
var
   ARect: TRect;
begin
   // copia as coordenadas do mouse no título
   Caption := Format ('Shapes (x=%d, y=%d)', [X, Y]);

   // código de arrasto
   if fDragging  then
   begin
     // remove e redesenha o retângulo de arrasto
     ARect := NormalizeRect (CurrShape.Rect);
     Canvas.DrawFocusRect (ARect);
     CurrShape.Right := X;
     CurrShape.Bottom := Y;
```

```
    ARect := NormalizeRect (CurrShape.Rect);
    Canvas.DrawFocusRect (ARect);
  end;
end;
```

Desta vez, entretanto, também incluímos uma correção no programa. No exemplo MouseOne, se você movesse o mouse em direção ao canto superior esquerdo do formulário, enquanto arrastava, a chamada a `DrawFocusRect` não produzia nenhum efeito. Isso porque o retângulo passado como parâmetro para `DrawFocusRect` deve ter um valor de `Top` menor do que o valor de `Bottom`, e o mesmo é verdade para os valores de `Left` e `Right`. Em outras palavras, um retângulo que se estende para o lado negativo não funciona de forma correta. Entretanto, no final ele é pintado corretamente, pois a função de desenho `Rectangle` não tem esse problema.

Para corrigir esse problema, escrevemos uma função simples que inverte as coordenadas de um retângulo para fazê-las refletir os pedidos da chamada a `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;
```

Finalmente, o manipulador do evento `OnMouseUp` configura o tamanho definitivo da imagem e atualiza a pintura do formulário. Em vez de chamar o método `Invalidate`, o que faria todas as imagens serem repintadas com muita cintilação, o programa usa a função da API `InvalidateRect`:

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

Os três parâmetros representam o handle da janela (isto é, a propriedade `Handle` do formulário), o retângulo que você deseja repintar e um flag indicando se você deseja ou não apagar a área antes de repintar. Essa função exige, mais uma vez, um retângulo *normalizado*. (Você pode tentar substituir essa chamada por uma chamada a `Invalidate`, para ver a diferença, que é mais evidente quando você cria muitos formulários.) Aqui está o código completo do manipulador do evento `OnMouseUp`:

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

```
    ReleaseCapture;
    fDragging := False;

    // configura o tamanho final
    ARect := NormalizeRect (CurrShape.Rect);
    Canvas.DrawFocusRect (ARect);
    CurrShape.Right := X;
    CurrShape.Bottom := Y;

    // código de invalidação otimizado
    ARect := NormalizeRect (CurrShape.Rect);
    InvalidateRect (Handle, @ARect, False);
   end;
end;
```

NOTA Quando você seleciona uma pena de desenho grande (veremos o código para isso em breve), a borda do quadro é pintada parcialmente dentro e parcialmente fora dele, para acomodar a pena grande. Para possibilitar isso, devemos invalidar um retângulo de moldura que é dilatado pela metade do tamanho da pena atual. Você pode fazer isso chamando a função `InflateRect`. Como alternativa, no método `FormCreate`, configuramos a propriedade `Style` de `Pen` do formulário `Canvas` como `psInsideFrame`. Isso faz com que a função de desenho pinte a pena completamente dentro do quadro da figura.

No método correspondente ao evento `OnPaint`, todas as figuras atualmente armazenadas na lista são pintadas, conforme você pode ver na Figura 1. Como o código de pintura afeta as propriedades de `Canvas`, precisamos armazenar os valores atuais e reinicializá-los no final. O motivo é que, conforme mostraremos posteriormente neste capítulo, as propriedades do canvas do formulário são usadas para controlar os atributos selecionados pelo usuário, que poderia tê-los mudado desde que a última figura foi criada. Aqui está o código:

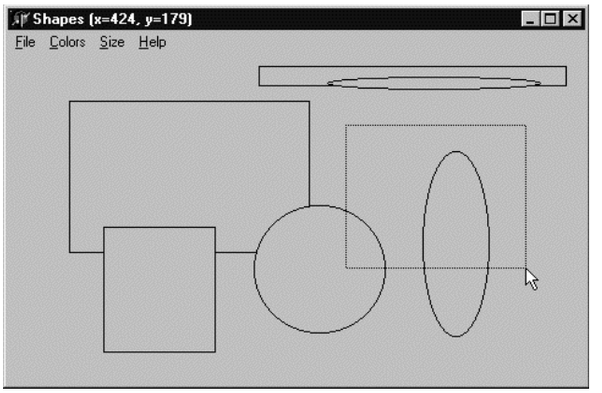
```
procedure TShapesForm.FormPaint(Sender: TObject);
var
  I, OldPenW: Integer;
  AShape: TBaseShape;
  OldPenCol, OldBrushCol: TColor;
begin
  // armazena os atributos de Canvas atual
  OldPenCol := Canvas.Pen.Color;
  OldPenW := Canvas.Pen.Width;
  OldBrushCol := Canvas.Brush.Color;

  // repinta cada figura da lista
  for I := 0 to ShapeList.Count - 1 do
  begin
    AShape := ShapeList.Items [I];
    AShape.Paint (Canvas);
  end;

  // reinicializa os atributos de Canvas atual
  Canvas.Pen.Color := OldPenCol;
  Canvas.Pen.Width := OldPenW;
  Canvas.Brush.Color := OldBrushCol;
end;
```

FIGURA 1

O exemplo Shapes pode ser usado para desenhar várias figuras, que ele armazena em uma lista.



Os outros métodos do formulário são simples. Três dos comandos de menu nos permitem usar as cores do fundo, as bordas da figura (a pena) e a área interna (o pincel). Esses métodos usam o componente ColorDialog e armazenam o resultado nas propriedades do canvas do formulário. Aqui está um exemplo:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  // seleciona uma nova cor para a pena
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute  then
   Canvas.Pen.Color := ColorDialog1.Color;
end;
```

As novas cores afetarão as figuras criadas no futuro, mas não as já existentes. A mesma estratégia é usada para a largura das linhas (a pena), embora desta vez o programa também verifique se o valor se tornou pequeno demais, desativando o item de menu, se isso acontecer:

```
procedure TShapesForm.DecreasePenSize1Click(Sender: TObject);
begin
   Canvas.Pen.Width := Canvas.Pen.Width - 2;
    if Canvas.Pen.Width < 3  then
   DecreasePenSize1.Enabled := False;
end;
```

Para alterar as cores da borda (a pena) ou da superfície (o pincel) da figura, usamos a caixa de diálogo Cor padrão. Aqui está um dos dois métodos:

```
procedure TShapesForm.PenColor1Click(Sender: TObject);
begin
  ColorDialog1.Color := Canvas.Pen.Color;
  if ColorDialog1.Execute  then
   Canvas.Pen.Color := ColorDialog1.Color;
end;
```

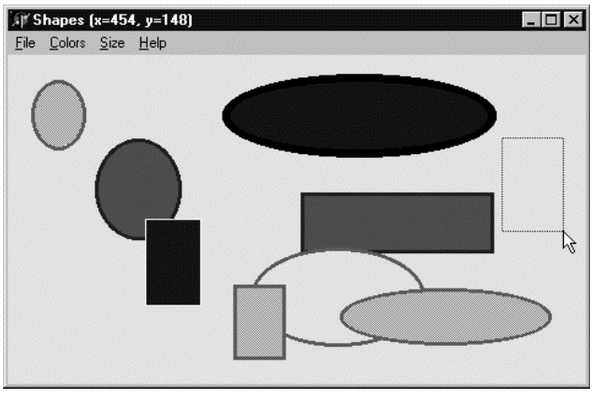
Na Figura 2, você pode ver outro exemplo da saída do programa Shapes, desta vez usando outras cores para as figuras e seus fundos. O programa pede para o usuário confirmar algumas operações, como a saída do programa ou a remoção de todas as figuras da lista (com o comando File | New):

```
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
    // exclui cada objeto
    for I := ShapesList.Count - 1 downto 0 do
        TBaseShape (ShapesList [I]).Free;
      ShapesList.Clear;
      Refresh;
   end;
end;
```

FIGURA 2

Alterar as cores e o tamanho da linha das figuras permite que você use o exemplo Shapes para produzir qualquer tipo de resultado.



Imprimindo Figuras

Além de pintar as figuras em um canvas de formulário, podemos pintá-las em um canvas de impressora, efetivamente imprimindo-as! Pelo fato de ser possível executar os mesmos métodos em um canvas de impressora como em qualquer outro canvas, você poderia ficar tentado a incluir no programa um novo método para imprimir as figuras. Com certeza isso é fácil, mas uma opção ainda melhor é escrever um único método de saída para usar para a tela e para a impressora.

Como exemplo dessa estratégia, construímos uma nova versão do programa, chamada ShapesPr. O ponto interessante é que movemos o código do exemplo `FormPaint` para outro método que definimos, chamado `CommonPaint`. Esse novo método possui dois parâmetros, o canvas e um fator de escala (cujo padrão é 1):

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

O método `CommonPaint` produz como saída a lista de figuras do canvas, passadas como parâmetros, usando o fator de escala correto:

```
procedure TShapesForm.CommonPaint (
  Canvas: TCanvas; Scale: Integer);
var
  I, OldPenW: Integer;
  AShape: TBaseShape;
  OldPenCol, OldBrushCol: TColor;
begin
  // armazena os atributos do Canvas atual
  OldPenCol := Canvas.Pen.Color;
```

```
  OldPenW := Canvas.Pen.Width;
   OldBrushCol := Canvas.Brush.Color;

   // repinta cada figura da lista
   for I := 0  to ShapesList.Count - 1 do
   begin
     AShape := ShapesList.Items [I];
     AShape.Paint (Canvas, Scale);
   end;

   // reinicializa os atributos do Canvas atual
   Canvas.Pen.Color := OldPenCol;
   Canvas.Pen.Width := OldPenW;
   Canvas.Brush.Color := OldBrushCol;
end;
```

Uma vez escrito esse código, os métodos `FormPaint` e `Print1Click` são simples de implementar. Para pintar a imagem na tela, você pode chamar `CommonPaint` sem um fator de escala (para que o valor padrão de 1 seja usado):

```
procedure TShapesForm.FormPaint(Sender: TObject);
begin
   CommonPaint (Canvas);
end;
```

Para pintar o conteúdo do formulário na impressora, em vez do formulário, você pode reproduzir a saída no canvas da impressora, usando um fator de escala correto. Em vez de escolher uma escala, decidimos calculá-la automaticamente. A idéia é imprimir as figuras no formulário com o maior tamanho possível, dimensionando a área cliente do formulário de modo que ela ocupe a página inteira. O código provavelmente é mais simples do que a descrição:

```
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;
```

É claro que você precisa se lembrar de chamar os comandos específicos para começar a imprimir (`BeginDoc`) e efetivar a saída (`EndDoc`), antes e depois de chamar o método `CommonPaint`. Se uma exceção for lançada, o programa chama `Abort` para terminar o processo de impressão de qualquer maneira.

Componentes Gráficos do Delphi

O exemplo Shapes usa quase nenhum componente, fora uma caixa de diálogo de seleção de cor padrão. Como alternativa, poderíamos ter usado alguns componentes do Delphi que suportam especificamente figuras:

- Você usa o componente PaintBox quando precisa pintar em uma certa área de um formulário que pode se mover no formulário. Por exemplo, PaintBox é útil para pintar em uma caixa de diálogo sem o risco de misturar a área da saída com a área dos controles. O componente PaintBox poderia se encaixar dentro de outros controles de um formulário, como uma barra de ferramentas ou uma barra de status, e evitar qualquer confusão ou sobreposição da saída. No exemplo Shapes, usar esse componente não fazia sentido, pois sempre trabalhamos na superfície inteira do formulário.
- Você usa o componente Shape para pintar figuras na tela, exatamente como fizemos até agora. Você poderia usar o componente Shape em lugar da saída manual, mas queremos mostrar como se realizam algumas operações de saída diretas. Esta estratégia não era muito mais complexa do que a que o Delphi sugere. Usar o componente Shape teria sido útil para estender o exemplo, permitindo ao usuário arrastar figuras na tela, removê-las e trabalhar com elas de várias outras maneiras.
- Você pode usar o componente Image para apresentar um bitmap existente, possivelmente carregando-o de um arquivo, ou mesmo para pintar sobre um bitmap, conforme demonstraremos nos próximos dois exemplos e que discutiremos na próxima seção.
- Se ele estiver incluído em sua versão de Delphi, você pode usar o controle TeeChart para gerar gráficos comerciais, conforme veremos no final deste capítulo.
- Você pode usar o suporte gráfico fornecido pelos controles botões de bitmap e speed buton, entre outros. Posteriormente neste capítulo, veremos como estender os recursos gráficos desses controles.
- Você pode usar o componente Animate para tornar as figuras animadas. Além de usar esse componente, você pode criar animações manualmente, apresentando bitmaps em seqüência ou rolando-os, conforme veremos em outros exemplos.

Como você pode ver, temos um longo caminho pela frente para abordar o suporte gráfico do Delphi de todos os seus ângulos.

Desenhando em um Bitmap

Já mencionamos que, usando um componente Image, você pode desenhar imagens diretamente em um bitmap. Em vez de desenhar na superfície de uma janela, você desenha em um bitmap na memória e depois copia o bitmap na superfície da janela. A vantagem é que, em vez de ter de repintar a imagem cada vez que um evento `OnPaint` ocorrer, o componente copia o bitmap para o vídeo.

Tecnicamente, um objeto `TBitmap` tem seu próprio canvas. Desenhando nesse canvas, você pode mudar o conteúdo do bitmap. Como alternativa, você pode trabalhar no canvas de um bitmap Image conectado ao bitmap que deseja alterar. Você poderia considerar a escolha dessa estratégia em lugar da estratégia de pintura típica se qualquer uma das seguintes condições fosse verdadeira:

- O programa precisa oferecer suporte a desenho à mão livre ou a imagens gráficas muito complexas (como imagens fractais).
- O programa deve ser muito rápido no desenho de várias imagens.
- O consumo de memória RAM não é problema.
- Você é um programador preguiçoso.

O último ponto é interessante, pois a pintura geralmente exige mais código do que o desenho, embora ela permita mais flexibilidade. Em um programa gráfico, por exemplo, se você usar pintura, terá de armazenar a localização e as cores de cada figura. Por outro lado, você pode mudar facilmente a cor de uma figura existente ou movê-la. Essas operações são muito difíceis com a estratégia de pintura, e podem fazer com que a área por trás de uma imagem seja perdida. Se você estiver trabalhando em um aplicativo gráfico complexo, provavelmente deverá escolher uma combinação das duas estratégias. Para os programadores gráficos casuais, a escolha entre as duas estratégias envolve uma decisão típica entre velocidade e memória: a pintura exige menos memória; o armazenamento do bitmap é mais rápido.

Desenhando Figuras

Agora, vamos ver um exemplo com o componente Image que pintará em um bitmap. A idéia é simples. Basicamente, escrevemos uma versão simplificada do exemplo Shape, colocando um componente Image em seu formulário e redirecionando toda as operações de saída para o canvas desse componente.

Neste exemplo, ShapeBmp, também incluímos alguns itens de menu novos, para salvar a imagem em um arquivo e para carregar um bitmap existente. Para fazer isso, incluímos no formulário dois componentes de diálogo padrão, OpenDialog e SaveDialog. Uma das propriedades que tivemos de mudar foi a cor de fundo do formulário. Na verdade, quando você executa a primeira operação gráfica na imagem, ela cria um bitmap que possui um fundo branco como padrão. Se o formulário tem um fundo cinza, sempre que a janela é repintada, alguma cintilação ocorre. Por esse motivo, escolhemos um fundo branco também para o formulário.

O código deste exemplo ainda é muito simples, considerando o número de operações e comandos de menu. A parte relativa ao desenho é linear e muito parecida com o exemplo MouseOne, exceto que os eventos de mouse agora estão relacionados à imagem; em vez do formulário, usamos a função `NormalizeRect` durante o arrasto, e o programa usa o canvas da imagem. Aqui está o manipulador do evento `OnMouseMove`, que reintroduz o desenho de pontos ao mover o mouse com a tecla Shift pressionada:

```
procedure TShapesForm.Image1MouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
  ARect: TRect;
begin
  // apresenta a posição do mouse no título
  Caption := Format ('ShapeBmp (x=%d, y=%d)', [X, Y]);
  if fDragging  then
  begin
    // remove e redesenha o retângulo de arrasto
    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
      // marca o ponto em vermelho
      Image1.Canvas.Pixels [X, Y] := clRed;
end;
```

Note que o retângulo de foco temporário é pintado diretamente no formulário, sobre a imagem (e, assim, não é armazenado no bitmap). A diferença é que no final da operação de desenho, o programa pinta o retângulo na imagem, armazenando-o no bitmap. Desta vez o programa não chama `Invalidate` e não tem manipulador do evento 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;
```

Para evitar um suporte a arquivos demasiadamente complexo, decidimos implementar os comandos File | Load e File | Save As e não manipular o comando Save, que em geral é mais complexo. Simplesmente incluímos um campo `fChanged` no formulário, para saber quando uma imagem mudou, e incluímos código que verifica esse valor várias vezes (antes de pedir confirmação para o usuário).

O manipulador do evento `OnClick` do item de menu File | New chama o método `FillArea` para pintar um grande retângulo branco sobre o bitmap inteiro. Neste código, você também pode ver como o campo `Changed` é usado:

```
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
       {repinta a superfície, cobrindo a área inteira,
       e reinicializando o pincel antigo}
       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;
```

É claro que o código tem de salvar a cor original e restaurá-la posteriormente. Um realinhamento das cores também é exigido pelo método de resposta do comando File | Load. Quando você carrega um novo bitmap, na verdade, o componente Image cria um novo canvas com os atributos padrão. Por esse motivo, o programa salva as cores e o tamanho da pena e os copia posteriormente no novo canvas:

```
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;
```

Salvar a imagem atual é muito mais simples:

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

Finalmente, aqui está o código do evento `OnCloseQuery` do formulário, que usa o campo `Changed`:

```
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 é um programa interessante (veja a Figura 3), com suporte a arquivos limitado, mas funcional. O problema em si é que o componente Image cria um bitmap de seu próprio tamanho. Quando você aumenta o tamanho da janela, o componente Image é redimensionado, mas não o bitmap presente na memória. Portanto, você não pode desenhar nas áreas direita e inferior da janela. Existem muitas soluções possíveis: usar a propriedade `Constraints` para configurar o tamanho máximo do formulário, usar uma borda fixa, marcar visualmente a *área de desenho* na tela etc. Entretanto, decidimos deixar o programa como está, porque ele realiza suficientemente bem o trabalho de demonstrar como se desenha em um bitmap.

FIGURA 3

O exemplo ShapeBMP tem suporte a arquivos limitado, mas funcional: você pode carregar um bitmap existente, desenhar figuras sobre ele e salvá-lo no disco.

Um Visualizador de Imagens

O programa ShadeBmp pode ser usado como um visualizador de imagens, pois você pode carregar qualquer bitmap nele. Em geral, no controle Image, você pode carregar qualquer tipo de arquivo gráfico que tenha sido registrado com a classe `Tpicture` da VCL. Os formatos de arquivo padrão são arquivos de bitmap (BMP), arquivos de ícone (ICO) ou `meta-arquivos` do Windows (WMF). Os arquivos de bitmap e ícone são formatos bem conhecidos. Os meta-arquivos do Windows, entretanto, não são tão comuns. Eles são um conjunto de comandos gráficos, semelhantes a uma lista de chamadas de função de GDI que precisam ser executadas para reconstruir uma imagem. Os meta-arquivos normalmente são referidos como *imagens gráficas vetoriais* e são parecidos com os formatos de arquivo gráfico usados para bibliotecas de clip-art. O Delphi também vem com suporte de JPG para `TImage`, e outros fornecedores possuem GIF e outros formatos de arquivo tratados.

NOTA Para produzir um `meta-arquivo` do Windows, um programa deve chamar funções da GDI, redirecionando sua saída para o arquivo. No Delphi, você pode usar um método `TMetafileCanvas` e os métodos `TCanvas` de alto nível. Posteriormente, esse `meta-arquivo` pode ser *reproduzido* ou executado para chamar as funções correspondentes, produzindo assim uma figura. Os `meta-arquivos` têm duas vantagens principais: a quantidade de armazenamento limitada que eles exigem em comparação a outros formatos gráficos e a independência de dispositivo de sua saída. Vamos abordar o suporte a `meta-arquivo` do Delphi posteriormente neste capítulo.

Para construir um programa visualizador de imagens completo, ImageV, em torno do componente Image, precisamos apenas criar um formulário com uma imagem que preencha a área cliente inteira, um menu simples e um componente 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 (*. mf)|*. mf'
      Options = [ofHideReadOnly, ofPathMustExist,
        ofFileMustExist]
   end
end
```

De forma surpreendente, esse aplicativo exige muito pouco código, pelo menos em sua primeira versão básica. Os comandos File | Exit e Help | About são simples, e o comando File | Open tem o seguinte código:

```
procedure TViewerForm.Open1Click(Sender: TObject);
begin
   if OpenDialog1.Execute  then
   begin
      Image1.Picture.LoadFromFile (OpenDialog1.FileName);
      Caption := 'Image Viewer - ' + OpenDialog1.FileName;
   end;
end;
```

O quarto e o quinto comandos de menu, Options | Stretch e Options | Center, simplesmente ativam/desativam a propriedade `Stretch` (veja o resultado na Figura 4) e a propriedade `Center` do componente e incluem uma marca de seleção em si mesmos. Aqui está o manipulador do evento `OnClick` do item de menu `Stretch1`:

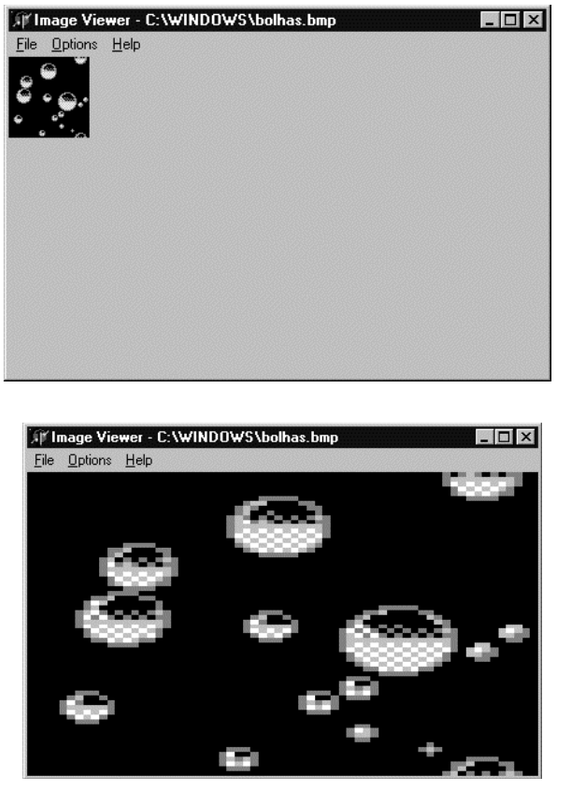
```
procedure TViewerForm.Stretch1Click(Sender: TObject);
begin
   Image1.Stretch := not Image1.Stretch;
   Stretch1.Checked := Image1.Stretch;
end;
```

Lembre-se de que, ao alongar uma imagem, você pode mudar sua relação largura-altura, possivelmente distorcendo a figura, e que nem todas as imagens podem ser alongadas de forma correta. O alongamento de bitmaps em preto-e-branco e de 256 cores nem sempre funciona corretamente.

Além desse problema, o aplicativo tem alguns outros inconvenientes. Se você selecionar um arquivo sem uma das extensões padrão, o componente Image lançará uma exceção. O manipulador de exceções fornecido pelo sistema se comporta como esperado; o arquivo de imagem errado não é carregado, e o programa pode continuar com segurança. Outro problema é que, se você carregar uma imagem grande, o visualizador não possuirá barras de rolagem. Você pode maximizar a janela do visualizador, mas isso poderá não ser suficiente. Os componentes Image não manipulam barras de rolagem automaticamente, mas o formulário pode fazer isso. A seguir, vamos estender ainda mais esse exemplo, para incluir barras de rolagem.

FIGURA 4

Dois exemplos do programa ImageV, que apresentam as versões normal e alongada do mesmo bitmap.



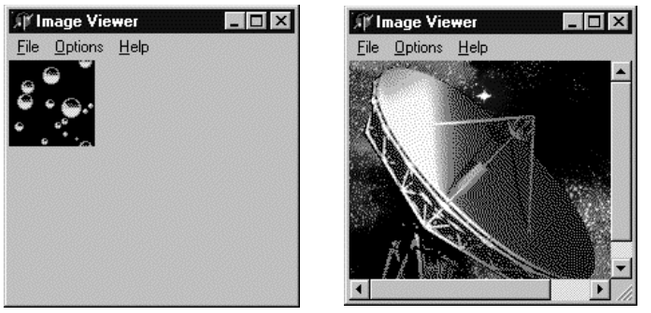
Rolando uma Imagem

Uma vantagem do modo como o rolamento automático funciona no Delphi é que, se o tamanho de um componente grande contido em um formulário muda, barras de rolagem são incluídas ou removidas automaticamente. Um bom exemplo é o uso do componente Image. Se a propriedade `AutoSize` desse componente for configurada para `True` e você carregar uma nova figura nele, o componente se dimensionará automaticamente, e o formulário incluirá ou removerá as barras de rolagem, conforme for necessário.

Se você carregar um bitmap grande no exemplo ImageV, notará que parte do bitmap permanece oculta. Para corrigir isso, você pode configurar a propriedade `AutoSize` do componente Image para `True` e desativar seu alinhamento com a área cliente. Você também deve configurar um tamanho inicial pequeno para a imagem. Você não precisa fazer quaisquer ajustes quando carrega um novo bitmap, pois o tamanho do componente Image é configurado automaticamente pelo sistema. Na Figura 5 você pode ver que barras de rolagem são realmente incluídas no formulário. A figura mostra duas execuções diferentes do programa. A diferença entre a execução do programa à esquerda e a que está à direita é que a primeira tem uma imagem menor do que sua área cliente; portanto, nenhuma barra de rolagem é inserida. Quando você carregar uma imagem maior no programa, duas barras de rolagem aparecerão automaticamente, como no exemplo da direita.

FIGURA 5

No exemplo ImageV2, barras de rolagem são incluídas automaticamente no formulário, quando o bitmap inteiro não cabe na área cliente do formulário apresentado.



Mais algum código é exigido para desativar as barras de rolagem e mudar o alinhamento da imagem quando o comando de menu Stretch é selecionado e para restaurá-las quando esse recurso for desativado. De novo, não atuamos diretamente sobre as barras de rolagem em si, mas simplesmente mudamos o alinhamento do painel, usando sua propriedade `Stretch`, e calculamos manualmente o novo tamanho, usando o tamanho da figura atualmente carregada. (Este código imita o efeito da propriedade `AutoSize`, que funciona apenas quando um novo arquivo é carregado.)

```
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;
```

Bitmaps ao Máximo

Quando o controle Image está conectado a um bitmap, existem algumas operações adicionais que você deve realizar, mas, antes de examiná-las, temos que apresentar os formatos de bitmap. Existem diferentes tipos de bitmaps no Windows. Os bitmaps podem ser *independentes de dispositivo* ou não, um termo usado para indicar se o bitmap tem informações de gerenciamento de paleta extras. Os arquivos BMP normalmente são bitmaps independentes de dispositivo.

Outra diferença se relaciona com a profundidade de cores — isto é, o número de diferentes cores que o bitmap pode usar ou, em outras palavras, o número de bits exigidos para armazenar cada pixel. Em um bitmap de 1 bit, cada ponto pode ser branco ou preto (para sermos mais precisos, os bitmaps de 1 bit podem ter uma paleta de cores, permitindo que ele represente quaisquer duas cores e não apenas preto e branco). Um bitmap de 8 bits normalmente possui uma paleta acompanhante para indicar como as 256 diferentes cores são mapeadas nas cores de sistema reais, um bitmap de 24 bits indica a cor de sistema diretamente. Para tornar as coisas mais complexas, quando o sistema desenha um bitmap em um computador com uma capacidade de cores diferente, ele precisa realizar alguma conversão.

Internamente, o formato do bitmap é muito simples, qualquer que seja a profundidade de cores. Todos os valores que constituem uma linha são armazenados em um bloco de memória. Isso é efi-

ciente para mover os dados da memória para a tela, mas não é um modo tão eficiente para armazenar informações; os arquivos BMP geralmente são muito grandes e eles não têm compactação.

NOTA Na verdade, o formato BMP tem uma forma muito limitada de compactação, conhecida como Run-Length Encoding (RLE), em que os pixels subseqüentes com a mesma cor são substituídos pelo número de tais pixels seguidos da cor. Isso pode reduzir o tamanho da imagem, mas, em alguns casos, a fará crescer. Para imagens compactadas no Delphi, você pode usar a classe `TJpegImage` e o suporte ao formato JPEG oferecido pela classe `TPicture`. Na verdade, tudo que `TPicture` faz é gerenciar uma lista registrada de classes gráficas.

O exemplo BmpDraw usa essa informação sobre a estrutura interna de um bitmap e alguns outros recursos técnicos para levar a manipulação direta de bitmaps para um novo nível. Primeiro, ele estende o exemplo ImageV através da inclusão de um item de menu que você pode usar para apresentar a profundidade de cores do bitmap atual, usando a propriedade `PixelFormat` correspondente:

```
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;
```

Você pode tentar carregar bitmaps diferentes e ver o efeito desse método, como ilustra a Figura 6.

O mais interessante é estudar como se acessa a imagem de memória mantida pelo objeto bitmap. Uma solução simples é usar a propriedade `Pixels`, conforme fizemos no exemplo ShapeBmp, para desenhar os pixels vermelhos durante a operação de arrasto. Nesse programa, incluímos um item de menu para criar um bitmap inteiramente novo, pixel por pixel, usando um cálculo matemático simples para determinar a cor. (A mesma estratégia pode ser usada, por exemplo, para construir imagens fractais.)

Aqui está o código do método, que simplesmente varre o bitmap nas duas direções e define a cor de cada pixel. Como estamos realizando muitas operações sobre o bitmap, podemos armazenar uma referência a ele na variável local `Bmp`, por simplicidade:

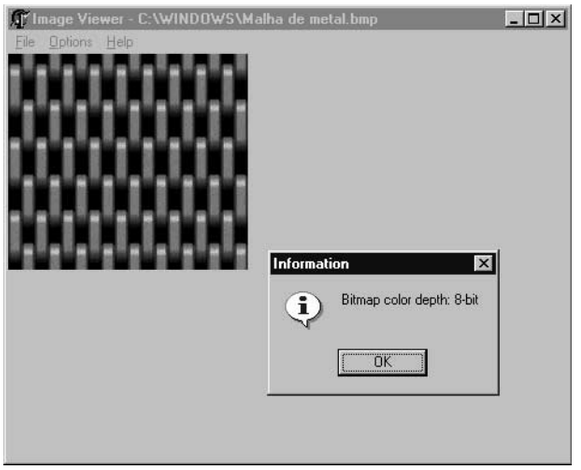
```
procedure TBitmapForm.GenerateSlow1Click(Sender: TObject);
var
   Bmp: TBitmap;
   I, J, T: Integer;
begin
   // obtém a imagem e a modifica
   Bmp := Image1.Picture.Bitmap;
   Bmp.PixelFormat := pf24bit;
   Bmp.Width := 256;
```

```
   Bmp.Height := 256;

   T := GetTickCount;
   // altera cada 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;
```

FIGURA 6

A profundidade de cores de um bitmap padrão do Windows, conforme apresentada pelo exemplo BmpDRAW.



Note que o programa controla o tempo exigido por essa operação, que, em meu computador, levou cerca de seis segundos. Conforme se vê a partir do nome da função, essa é uma versão lenta do código.

Podemos acelerá-la consideravelmente acessando o bitmap uma linha inteira por vez. Esse recurso pouco conhecido está disponível através da propriedade `ScanLine` do bitmap, que retorna um ponteiro para a área de memória da linha de bitmap. Pegando esse ponteiro e acessando a memória diretamente, tornamos o programa muito mais rápido. O único problema é que precisamos conhecer a representação interna do bitmap. No caso de um bitmap de 24 bits, todo ponto é representado por três bytes que definem a intensidade de azul, verde e vermelho (o inverso da seqüência RGB). Aqui está o código alternativo, com uma saída ligeiramente diferente (pois modificamos deliberadamente o cálculo da cor):

```
procedure TBitmapForm.GenerateFast1Click(Sender: TObject);
var
   Bmp: TBitmap;
   I, J, T: Integer;
   Line: PByteArray;
begin
   // obtém a imagem e a modifica
   Bmp := Image1.Picture.Bitmap;
   Bmp.PixelFormat := pf24bit;
   Bmp.Width := 256;
   Bmp.Height := 256;
```

```
T := GetTickCount;
// muda cada pixel, linha por linha
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;
// atualiza o vídeo
Image1.Invalidate;
Caption := 'Image Viewer - Memory Image (MSecs: ' +
   IntToStr (GetTickCount - T) + ')';
 end;
```

Apenas mover uma linha na memória não faz uma tela ser atualizada; portanto, o programa chama `Invalidate` no final. A saída produzida por este segundo método (veja a Figura 7) é muito parecida, mas o tempo que ela levou em meu computador foi de cerca de 60 milissegundos. Isso é quase um centésimo do tempo da outra estratégia! Essa técnica é tão rápida que podemos usá-la para rolar as linhas do bitmap e ainda produzir um efeito rápido e suave. A operação de rolagem tem algumas opções; portanto, quando você seleciona os itens de menu correspondentes, o programa simplesmente mostra um painel dentro do formulário. Esse painel tem uma barra de controle que você pode usar para ajustar a velocidade da operação de rolagem (reduzindo sua suavidade à medida que a velocidade aumenta). A posição da barra de controle é salva em um campo local do formulário:

```
procedure TBitmapForm.TrackBar1Change(Sender: TObject);
begin
   nLines := TrackBar1.Position;
   TrackBar1.Hint := IntToStr (TrackBar1.Position);
end;
```

FIGURA 7

O desenho que você vê na tela é gerado pelo exemplo BmpDraw em uma fração de segundo (conforme informado em seu título).

No painel existem também dois botões, usados para iniciar e interromper a operação de rolagem. O código do botão Go tem dois laços `for`. O laço externo é usado para repetir a operação de rolagem,

tantas vezes quantas forem as linhas no bitmap. O laço interno realiza a operação de rolagem copiando cada linha do bitmap na anterior. A primeira linha é armazenada temporariamente em um bloco de memória e depois copiada na última linha no final. Esse bloco de memória temporário é mantido em uma área de memória alocada dinamicamente (`AllocMem`), grande o suficiente para conter uma linha. Essa informação é obtida pelo cálculo da diferença nos endereços de memória de duas linhas consecutivas.

O centro da operação de movimentação é efetuado através do uso da função `Move` do Delphi. Seu parâmetro é a variável a ser movida, e não os endereços de memória. Por isso, você tem de retirar a referência aos ponteiros. (Bem, esse método é realmente um bom exercício sobre ponteiros!) Finalmente, note que, desta vez, não podemos invalidar a imagem inteira após cada operação de rolagem, pois isso produz muito tremido na saída. A solução oposta é invalidar cada linha depois de ela ter sido movida, mas isso torna o programa lento demais. Como uma solução intermediária, decidimos invalidar um bloco de linhas por vez, conforme determinado pela expressão `J mod nLines = 0`. Quando determinado número de linhas tiver sido movido, o programa atualiza essas linhas:

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

Conforme você pode ver, o número de linhas é determinado pela posição do controle TrackBar.

Um usuário pode até mudar a velocidade movendo o controle speed durante a operação de rolagem. Também permitimos que o usuário pressione o botão Cancel durante a operação. Isso se tornou possível pela chamada a `Application.ProcessMessages` no laço `for` interno. O botão Cancel altera o flag `fCancel`, que é testado em cada iteração do laço `for` externo:

```
procedure TBitmapForm.BtnCancelClick(Sender: TObject);
begin
   fCancel := True;
end;
```

Assim, após toda essa descrição, aqui está o código completo do manipulador do evento `OnClick` do botão Go:

```
procedure TBitmapForm.BtnGoClick(Sender: TObject);
var
   W, H, I, J, LineBytes: Integer;
   Line: PByteArray;
   Bmp: TBitmap;
   R: TRect;
begin
   // configura a interface com o usuário
   fCancel := False;
   BtnGo.Enabled := False;
   BtnCancel.Enabled := True;

   // obtém o bitmap da imagem e o redimensiona
   Bmp := Image1.Picture.Bitmap;
   W := Bmp.Width;
   H := Bmp.Height;

   // aloca memória suficiente para uma linha
   LineBytes := Abs (Integer (Bmp.ScanLine [1]) -
     Integer (Bmp.ScanLine [0]));
   Line := AllocMem (LineBytes);
```

```
   // rola tantos itens quantas forem as linhas
   for I := 0  to H - 1 do
   begin
       // sai do laço for se o botão Cancel foi pressionado
       if fCancel  then
        Break;

       // copia a primeira linha
       Move ((Bmp.ScanLine [0])^, Line^, LineBytes);

       // para toda linha
       for J := 1  to H - 1 do
       begin
          // move a linha para a anterior
          Move ((Bmp.ScanLine [J])^, (Bmp.ScanLine [J-1])^, LineBytes);
         // cada nLines atualiza a saída
         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 a primeira linha para o final
       Move (Line^, (Bmp.ScanLine [Bmp.Height - 1])^, LineBytes);
       // atualiza a parte final do bitmap
       R := Rect (0, PanelScroll.Height + H - nLines,
         W, PanelScroll.Height + H);
       InvalidateRect (Handle, @R, False);
       UpdateWindow (Handle);

       //  permite que o programa manipule outras mensagens
       Application.ProcessMessages;
   end;

   // reinicializa a UI
   BtnGo.Enabled := True;
   BtnCancel.Enabled := False;
end;
```

Você pode ver um bitmap durante a operação de rolagem na Figura 8. Note que a rolagem pode ocorrer em qualquer tipo de bitmap, e não apenas nos bitmaps de 24 bits gerados por este programa. Na verdade, você pode carregar outro bitmap no programa e depois rolá-lo, como fizemos para criar a ilustração.

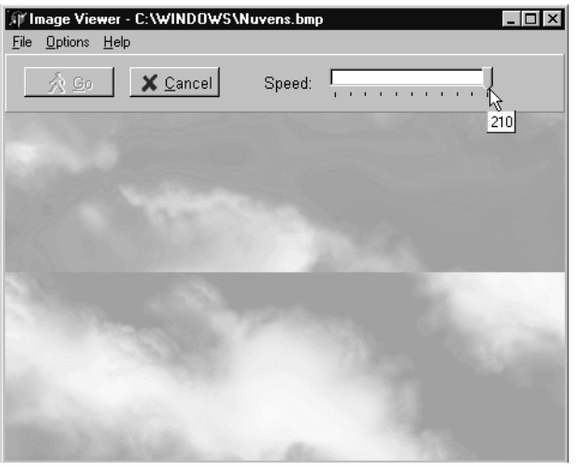
Um Bitmap Animado em um Botão

Os botões de bitmap são fáceis de usar e podem produzir aplicativos de melhor aparência do que os botões padrão (o componente Button). Para aprimorar ainda mais o efeito visual de um botão, também podemos considerar a *animação* do botão. Existem basicamente dois tipos de botões animados — botões que mudam sua imagem ligeiramente quando são pressionados e botões que possuem uma

imagem que se move, independentemente da operação. Vamos mostrar um exemplo simples de cada tipo, Fire e World. Para cada um dos exemplos, exploraremos duas versões ligeiramente diferentes.

FIGURA 8

O exemplo BmpDraw permite a rolagem rápida de um bitmap.



Um Botão de Dois Estados

O primeiro exemplo, o programa Fire, tem um formulário simples, contendo apenas um botão de bitmap. Esse botão está conectado a um elemento `Glyph` que representa um canhão. Imagine tal botão como parte de um programa de jogo. Quando o botão é pressionado, a figura muda para mostrar um canhão atirando. Assim que o botão é solto, a imagem padrão é novamente carregada. Nesse ínterim, o programa apresenta uma mensagem, caso o usuário tenha realmente dado um clique no botão.

Para escrever este programa, precisamos manipular três eventos do botão: `OnMouseDown`, `OnMouseUp` e `OnClick`. O código dos três métodos é extremamente simples:

```
procedure TForm1.BitBtnFireMouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
   // carrega o bitmap do canhão atirando
   if Button = mbLeft  then
     BitBtnFire.Glyph.LoadFromFile ('fire2.bmp');
end;

procedure TForm1.BitBtnFireMouseUp(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
   // carrega o bitmap do canhão padrão
   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;
```

Incluímos alguns recursos sonoros, reproduzindo um arquivo WAV quando o botão é pressionado, com uma chamada para a função `PlaySound` da unidade MmSystem. Quando você mantém o botão esquerdo do mouse pressionado sobre o botão de bitmap, o botão é pressionado. Se, então, você move o cursor do mouse para fora do botão, enquanto mantém o botão do mouse pressionado, o botão de bitmap é liberado, mas ele não recebe um evento `OnMouseUp`; portanto, o canhão atirando permanece lá. Se, posteriormente, você soltar o botão esquerdo do mouse fora da superfície do botão de bitmap, ele receberá o evento `OnMouseUp`. O motivo é que todos os botões no Windows capturam a entrada de mouse quando são pressionados.

Muitas Imagens em um Bitmap

O exemplo Fire usou uma estratégia manual. Carregamos dois bitmaps e alteramos o valor da propriedade `Glyph` quando queríamos mudar a imagem. O componente BitBtn, entretanto, também pode manipular vários bitmaps automaticamente. Você pode preparar um único bitmap que contenha várias imagens (ou grifos) e configurar esse número como o valor da propriedade `NumGlyphs`. Todos esses "sub-bitmaps" devem ter o mesmo tamanho, pois o bitmap global é dividido em partes iguais.

Se você fornecer mais de uma imagem no bitmap, elas serão usadas de acordo com as regras a seguir:

- O primeiro bitmap é usado para o botão solto, a posição padrão.
- O segundo bitmap é usado para o botão desativado.
- O terceiro bitmap é usado quando o botão recebe um clique de mouse.
- O quarto bitmap é usado quando o botão permanece pressionado, como nos botões que se comportam como caixas de seleção.

Normalmente, você fornece uma única imagem, e as outras são automaticamente calculadas a partir dela, com alterações gráficas simples. Entretanto, é fácil fornecer uma segunda, uma terceira e uma quarta figura personalizada. Se você não fornecer todos os quatro bitmaps, os ausentes serão calculados automaticamente a partir do primeiro.

Em nosso exemplo, a nova versão de Fire (chamada Fire2), precisamos apenas da primeira e da terceira imagens do bitmap, mas somos obrigados a incluir o segundo bitmap. Para ver como essa imagem (a segunda do bitmap) pode ser usada, incluímos uma caixa de seleção para desativar o botão de bitmap. Para construir a nova versão do programa, preparamos um bitmap de 32x96 pixels (veja a Figura 9) e o usamos para a propriedade `Glyph` do bitmap. O Delphi configura automaticamente a propriedade `NumGlyphs` em 3, pois o bitmap é três vezes mais largo do que sua altura.

FIGURA 9

O bitmap com três imagens do exemplo Fire2, conforme visto no Delphi Image Editor.

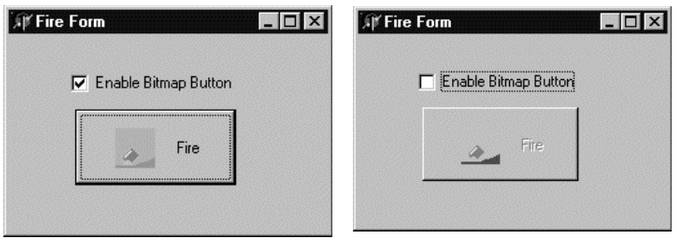
A caixa de seleção, usada para ativar e desativar o botão (para que possamos ver a imagem correspondente ao status desativado), tem o seguinte evento `OnClick`:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
   BitBtnFire.Enabled := CheckBox1.Checked;
end;
```

Quando você executa o programa, existem duas maneiras de mudar o bitmap no botão. Você pode desativar o botão de bitmap usando a caixa de seleção (veja a Figura 10) ou pode pressionar o botão para ver o canhão atirar. Na primeira versão (o exemplo Fire), a imagem com o canhão atirando permanecia no botão até que a caixa de mensagem fosse fechada. Agora (no exemplo Fire2), a imagem é mostrada apenas enquanto o botão está pressionado. Assim que você sai da superfície do botão, a primeira imagem é apresentada.

FIGURA 10

Os botões de bitmap ativado e desativado do exemplo Fire2, em duas execuções diferentes do aplicativo.



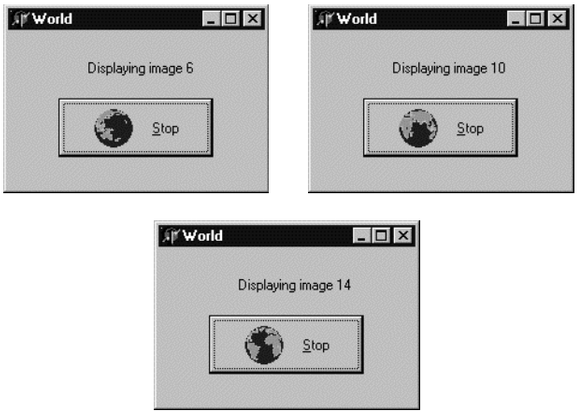
O Mundo Girando

O segundo exemplo de animação, World, tem um botão que apresenta a Terra, que gira lentamente, mostrando os vários continentes. Você pode ver alguns exemplos na Figura 11, mas, é claro, deve executar o programa para ver sua saída. No exemplo anterior, a imagem mudava quando o botão era pressionado. Agora, a imagem muda sozinha, automaticamente. Isso ocorre graças à presença de um componente Timer, que recebe uma mensagem em intervalos fixos de tempo.

Aqui está um resumo das propriedades do componente:

```
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
```

FIGURA 11
Alguns exemplos do programa World em funcionamento.



O componente temporizador é iniciado e parado (ativado e desativado) quando o usuário pressiona o botão de bitmap que possui a imagem do mundo:

```
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;
```

Conforme você pode ver na Figura 11, um rótulo sobre o botão indica qual das imagens está sendo apresentada. Sempre que a mensagem do temporizador é recebida, a imagem e o rótulo mudam:

```
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;
```

Nesse código, `Count` é um campo do formulário que é inicializado como `1` no método `FormCreate`. A cada intervalo do temporizador, `Count` é aumentado como módulo `16` e depois convertido para uma string (precedida pela letra *w*). A razão para esse limite é simples — tínhamos `16` bitmaps da Terra para apresentar. Chamar os arquivos de bitmap de `W1.BMP`, `W2.BMP` etc. facilita para o programa acessá-los, construindo as strings com o nome em tempo de execução.

NOTA A operação de módulo retorna o resto da divisão entre inteiros. Isso significa que `Count mod 16` retorna invariavelmente um valor no intervalo de 0 a 15. Somando um a esse valor de retorno, obtemos o número do bitmap, que está no intervalo de 1 a 16.

Uma Lista de Bitmaps, o Uso de Recursos e um ControlCanvas

O programa World funciona, mas é muito lento por dois motivos. Primeiramente, a cada intervalo do temporizador, ele precisa ler um arquivo do disco e, embora uma cache de disco possa tornar isso mais rápido, certamente essa não é a solução mais eficiente. Além de ler o arquivo do disco, o programa tem de criar e destruir objetos de bitmap do Windows e isso requer algum tempo. O segundo problema depende de como a imagem é atualizada: quando você muda o bitmap do botão, o componente é completamente apagado e repintado. Isso causa algum tremido, conforme você pode ver executando o programa.

Para resolver o primeiro problema (e para mostrar a você uma estratégia diferente para manipular bitmaps), criamos uma segunda versão do exemplo, World2. Aqui, incluímos um contêiner `TObjectList` do Delphi 5, armazenando uma lista de bitmaps, no formulário do programa. O formulário também tem mais alguns campos:

```
type
  TWorldForm = class(TForm)
     ...
  private
     Count, YPos, XPos: Integer;
     BitmapsList: TObjectList;
     ControlCanvas: TControlCanvas;
end;
```

Todos os bitmaps são carregados quando o programa começa e destruídos quando ele termina. A cada intervalo do temporizador, o programa mostra um dos bitmaps da lista no botão de bitmap. Usando uma lista, evitamos o carregamento de um arquivo cada vez que precisarmos apresentar um bitmap, mas ainda precisamos ter todos os arquivos com as imagens no diretório que possui o arquivo executável. Uma solução para esse problema é mover os bitmaps dos arquivos independentes para o arquivo de recursos do aplicativo. Isso é mais fácil fazer do que explicar.

Para usar os recursos em lugar dos arquivos de bitmap, precisamos primeiro criar esse arquivo. A melhor estratégia é escrever um script de recursos (um arquivo RC), listando os nomes dos arquivos de bitmap e dos recursos correspondentes. Abra um novo arquivo de texto (em qualquer editor) e escreva o seguinte código:

```
W1  BITMAP  "W1.BMP"
W2  BITMAP  "W2.BMP"
W3  BITMAP  "W3.BMP"
// ... etc.
```

Uma vez que você tenha preparado esse arquivo RC (o chamamos de `WorldBmp.RC`), é possível compilá-lo em um arquivo RES, usando o compilador de recursos incluído e o aplicativo de linha de comando BRCC32, que pode ser encontrado no diretório BIN do Delphi, e depois incluí-lo no projeto através da inserção da diretiva `{$R WORLDBMP.RES}` no código-fonte do projeto ou em uma das unidades.

Entretanto, no Delphi 5, você pode usar uma estratégia mais simples. Você pode pegar o arquivo RC e simplesmente incluí-lo no projeto, usando o comando Add to Project do menu Project ou simplesmente arrastando o arquivo para o projeto. O Delphi 5 ativará automaticamente o compilador de recursos e depois ligará o arquivo de recursos ao arquivo executável. Essas operações são controladas por uma diretiva de inclusão de recurso estendida, incluída no código-fonte do projeto:

```
{$R 'WORLDBMP.res' WORLDBMP.RC'}
```

Uma vez que tenhamos definido corretamente os recursos do aplicativo, precisamos carregar os bitmaps dos recursos. Para um objeto TBitmap, podemos usar o método LoadFromResourceName, caso o recurso tenha um identificador de string, ou o método LoadFromResourceID, caso ele tenha um identificador numérico. O primeiro parâmetro dos dois métodos é um handle para o aplicativo, conhecido como HInstance, disponível no Delphi como uma variável global.

DICA O Delphi define uma segunda variável global, MainInstance, que se refere a HInstance do arquivo executável principal. A não ser que você esteja dentro de uma DLL, pode usar uma ou outra indistintamente.

Este é o código do método FormCreate:

```
procedure TWorldForm.FormCreate(Sender: TObject);
var
   I: Integer;
   Bmp: TBitmap;
begin
   Count := 1;
   // carrega os bitmaps e os inclui na lista
   BitmapsList := TList.Create;
   for I := 1  to 16 do
   begin
      Bmp := TBitmap.Create;
      Bmp.LoadFromResourceName (HInstance,
      'W' + IntToStr (I));
      BitmapsList.Add (Bmp);
   end;
end;
```

NOTA Como alternativa, poderíamos ter usado o componente ImageList, mas, para este exemplo, decidimos usar uma estratégia de baixo nível para mostrar a você todos os detalhes envolvidos.

Permanece um problema para ser resolvido: obter uma transição suave de uma imagem do mundo para a seguinte. O programa deve pintar os bitmaps em um canvas usando o método Draw. Infelizmente, o canvas do botão de bitmap não está diretamente disponível (e sem evento protegido); portanto, decidimos usar um componente TControlCanvas (em geral o canvas interno de um controle, mas um que você também possa associar externamente). Para usá-lo para pintar sobre um botão, podemos atribuir o botão ao controle canvas no método FormCreate:

```
ControlCanvas := TControlCanvas.Create;
ControlCanvas.Control := WorldButton;
YPos := (WorldButton.Height – Bmp.Height) div 2;
XPos := WorldButton.Margin;
```

A posição horizontal do botão onde a imagem está localizada (e onde devemos pintar) depende do componente Margin do ícone do botão de bitmap e da altura do bitmap. Uma vez que o canvas do controle esteja corretamente definido, o método Timer1Timer simplesmente pinta sobre ele — e sobre o botão:

```
procedure TWorldForm.Timer1Timer(Sender: TObject);
begin
   Count := (Count mod 16) + 1;
   Label1.Caption := Format ('Displaying image %d', [Count]);
```

```
   // desenha o bitmap atual no canvas do controle
 ControlCanvas.Draw (XPos, YPos,
   BitmapsList.Items[Count-1] as TBitmap);
end;
```

O último problema é mudar a posição da imagem quando o botão esquerdo do mouse for pressionado ou solto sobre ele (isto é, nos eventos `OnMouseDown` e `OnMouseUp` do botão). Além de mover a imagem por alguns pixels, devemos atualizar a imagem do bitmap, pois o Delphi a apresentará automaticamente enquanto redesenha o botão. Caso contrário, um usuário veria a imagem inicial até que o intervalo do temporizador tenha decorrido e o componente ativado o evento `OnTimer`. (Isso poderia levar algum tempo, caso você o tivesse parado!) Aqui está o código do primeiro dos dois métodos:

```
procedure TWorldForm.WorldButtonMouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
   if Button = mbLeft  then
   begin
      // pinta a imagem atual sobre o botão
      WorldButton.Glyph.Assign (
          BitmapsList.Items[Count-1] as TBitmap);
      Inc (YPos, 2);
      Inc (XPos, 2);
   end;
end;
```

O Controle Animado

Há uma maneira melhor de obter animação do que apresentar uma série de bitmaps em seqüência. Use o controle comum Animate do Win32. O controle Animate é baseado no uso de arquivos AVI (Audio Video Interleaved), uma série de bitmaps semelhante a um filme.

NOTA Na verdade, o controle Animate pode apresentar apenas os arquivos AVI que tenham um único fluxo de vídeo, sejam descompactados ou compactados com compactação `RLE8` e não tenham mudanças de paleta; e se eles tiverem som, ele será ignorado. Na prática, os arquivos correspondentes a esse requisito são aqueles constituídos de uma série de bitmaps de computador e não aqueles baseados em um filme real.

O controle Animate pode ter duas fontes possíveis para sua animação:

- Ele pode ser baseado em qualquer arquivo AVI que atenda aos requisitos indicados na nota anterior; para usar esse tipo de fonte, configure um valor correto para a propriedade `FileName`.
- Ele pode usar uma animação interna especial do Windows, parte da biblioteca de controle comum; para usar esse tipo de fonte, escolha um dos valores possíveis da propriedade `CommonAVI` (que é baseada em uma enumeração).

Se você simplesmente colocar um controle Animate em um formulário, escolha uma animação usando um dos métodos que acabamos de descrever e, finalmente, configure sua propriedade `Active` para `True`. Você começará a ver a animação até em tempo de projeto. Por definição, a animação ocorre continuamente, reiniciando assim que acaba. Entretanto, você pode regular esse efeito usando a propriedade `Repetitions`. O valor padrão -1 causa uma repetição infinita; use qualquer outro valor para especificar um número de repetições.

Você também pode especificar o quadro inicial e final da seqüência, com as propriedades StartFrame e StopFrame. Essas três propriedades (posição inicial, posição final e número de repetições) correspondem aos três parâmetros do método Play, que você usará freqüentemente com um controle Animate. Como alternativa, você pode configurar as propriedades e depois chamar o método Start. Em tempo de execução, você também pode acessar o número total de quadros usando a propriedade FrameCount. Você pode usar isso para executar a animação do início ao fim. Finalmente, para obter um controle mais apurado, você pode usar o método Seek, que apresenta um quadro específico.

Usamos todos esses métodos em um programa demonstrativo simples (AnimCtrl), que pode usar os dois arquivos e as animações padrão do Windows. O programa permite que você escolha um arquivo ou uma das animações usando um componente ListBox. Incluímos nesse componente ListBox um item para cada elemento da enumeração TCommonAVI e usamos a mesma ordem:

```
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
```

Graças a essa estrutura, quando o usuário dá um clique no componente ListBox, apenas converter o número dos itens selecionados para o tipo de dados enumerado fornecerá o valor correto para a propriedade 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;
```

Conforme você pode ver, quando o primeiro item é selecionado (o valor é caNone), o programa carrega automaticamente um arquivo AVI, usando um componente OpenDialog. O componente mais importante do formulário é o controle Animate. Aqui está sua descrição textual:

```
object Animate1: TAnimate
   AutoSize = False
   Align = alClient
   CommonAVI = aviFindFolder
   OnOpen = Animate1Open
end
```

Ele é alinhado com a área cliente, de modo que um usuário pode redimensioná-lo facilmente, dependendo do tamanho real dos quadros da animação. Conforme você pode ver, também definimos um manipulador para um evento desse componente, OnOpen:

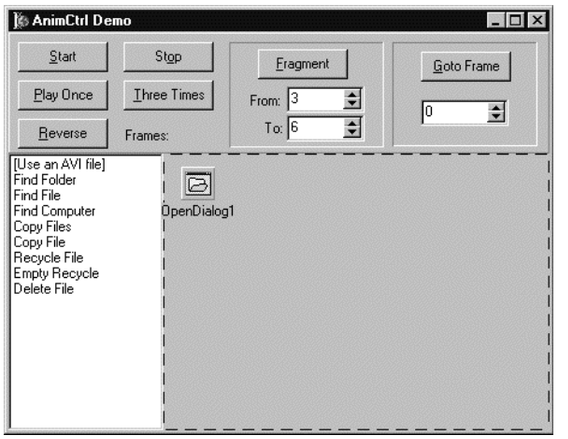
```
procedure TForm1.Animate1Open(Sender: TObject);
begin
```

```
   LblFrames.Caption := 'Frames ' +
      IntToStr (Animate1.FrameCount);
end;
```

Quando um novo arquivo (ou animação comum) é aberto, o programa simplesmente produz como saída o número de seus quadros em um rótulo. Esse rótulo fica junto a vários botões e alguns controles SpinEdit em um painel grande, atuando como uma barra de ferramentas. Você pode vê-los no formulário em tempo de projeto da Figura 12.

FIGURA 12

O formulário do exemplo AnimCtrl em tempo de projeto.



Os botões Start e Stop são totalmente triviais, mas o botão Play Once tem algum código:

```
procedure TForm1.BtnOnceClick(Sender: TObject);
begin
   Animate1.Play (0, Animate1.FrameCount, 1);
end;
```

As coisas começam a ficar mais interessantes com o código usado para reproduzir a animação três vezes ou para reproduzir apenas um trecho dela. Esses dois métodos têm por base o método `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;
```

Os dois últimos manipuladores de evento de botão são baseados no método `Seek`. O botão Goto simplesmente vai para o quadro indicado pelo componente SpinEdit correspondente, enquanto os botões Reverse vão para cada quadro por sua vez, partindo do último e fazendo uma pausa entre cada um deles:

```
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);
      // espera 50 milissegundos
      Init := Now;
      while Now < Init + EncodeTime (0, 0, 0, 50) do
         Application.ProcessMessages;
   end;
end;
```

O Controle Animate em um Botão

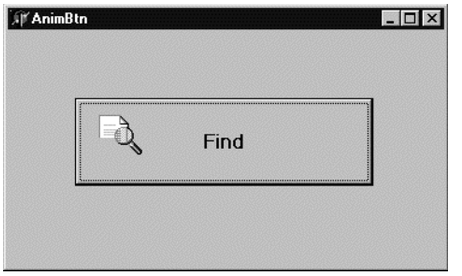
Agora que você sabe como o controle Animate funciona, podemos usá-lo para construir outro botão animado. Basta colocar um controle Animate e um botão grande (possivelmente com uma fonte grande também) em um formulário. Em seguida, escreva o seguinte código para transformar o botão na janela progenitora do controle Animate em tempo de execução e posicioná-lo corretamente:

```
procedure TForm1.FormCreate(Sender: TObject);
var
   Diff: Integer;
begin
   Animate1.Parent := Button1;
   Diff := Button1.Height - Animate1.Height;
   Animate1.SetBounds ( hDiff div 2,  hDiff div 2,
     Animate1.Width, Animate1.Height);
   Animate1.Active := True;
end;
```

Você pode ver um exemplo desse efeito na Figura 13 (o projeto tem o nome AnimBtn). Essa é mesmo a estratégia mais simples para produzir um botão animado, mas ela também permite o mínimo de controle.

FIGURA 13

O efeito do controle Animate dentro de um botão, como ilustrado pelo programa AnimBtn.



Grades Gráficas

As grades representam outro grupo interessante de componentes gráficos do Delphi. O sistema oferece diferentes componentes gráficos: uma grade de strings, uma de imagens, grades relacionadas a

banco de dados e uma grade de amostra de cores. Os dois primeiros tipos de grades são particularmente úteis, pois eles permitem que você represente muitas informações e também que o usuário navegue nelas. É claro que as grades são extremamente importantes na programação de banco de dados, e elas podem ser personalizadas com figuras, conforme vimos no Capítulo 13 do *Dominando o Delphi 6*.

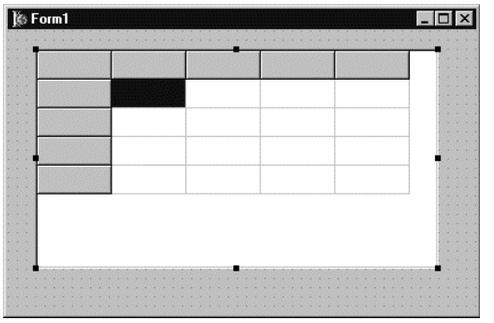
Os componentes DrawGrid e StringGrid são intimamente relacionados. Na verdade, a classe `TStringGrid` é uma subclasse de `TDrawGrid`. Para que usar essas grades? Basicamente, você pode armazenar alguns valores, ou nas strings relacionadas ao componente StringGrid ou em outras estruturas de dados, e depois apresentar valores selecionados, usando critérios específicos. Embora as grades de strings possam ser usadas quase como estão (pois elas já fornecem recursos de edição), as grades de objetos genéricos normalmente exigem mais codificação.

Na verdade, as grades definem o modo como as informações são organizadas para exibição e não da maneira como são armazenadas. A única grade que armazena os dados que apresenta é StringGrid. Todas as outras grades (incluindo os componentes DrawGrid e DBGrid) são apenas visualizadores de dados, e não contêineres de dados. O componente DBGrid não possui os dados que apresenta; ele busca os dados a partir da fonte de dados conectada. Às vezes isso causa confusão.

A estrutura básica de uma grade inclui várias colunas e linhas fixas, que indicam a região não rolante da grade (conforme você pode ver na Figura 14). As grades estão entre os componentes mais complexos disponíveis no Delphi, conforme indicado pelo alto número de propriedades e métodos que elas contêm. Existem muitas opções e propriedades excelentes para as grades, controlando sua aparência e seu comportamento.

FIGURA 14

Quando você coloca um novo componente grade em um formulário, ele contém uma linha fixa e uma coluna fixa como padrão.



Em sua aparência, a grade pode ter linhas de diferentes tamanhos ou pode não ter linhas. Você pode configurar o tamanho de cada coluna ou linha independentemente das outras, pois as propriedades `RowSize`, `ColWidth` e `RowHeight` são arrays. Para o comportamento da grade, você pode deixar que o usuário redimensione as colunas e as linhas (`goColSizing` e `goRowSizing`), arraste colunas e linhas inteiras para uma nova posição (`goRowMoving` e `goColumnMoving`), selecione a edição automática e permita seleções de intervalos. Como várias opções permitem que os usuários executem diversas operações nas grades, também existem vários elementos relacionados às grades, como `OnColumnMove`, `OnDrawCell` ou `OnSetEditText`.

O evento mais importante provavelmente é `OnDrawCell`. Em resposta a esse evento, um programa deve pintar determinada célula da grade. Apenas as grades de strings podem apresentar automaticamente seu conteúdo. Na verdade, o componente DrawGrid não tem suporte ao armazenamento de dados. Ele é simplesmente uma ferramenta para organizar uma parte da tela, para apresentar informações em um formato regular. Trata-se de uma ferramenta simples, mas também poderosa. Métodos como `CellRect`, que retorna o retângulo correspondente à área de uma célula, ou `MouseToCell`, que retorna a célula em uma posição específica, são uma satisfação de usar. Manipulando linhas e colunas

redimensionáveis e grades rolantes, eles simplificam tarefas complexas e liberam o programador de cálculos maçantes.

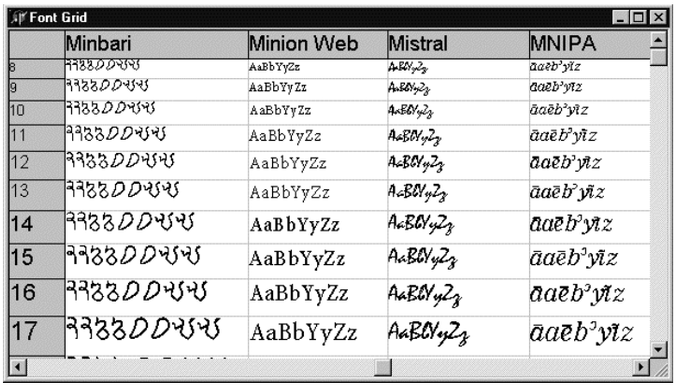
Para que você pode usar uma grade? Construir uma planilha eletrônica provavelmente é a primeira idéia que vem à mente, mas isso talvez seja um pouco complexo demais para um exemplo. Decidimos usar o controle StringGrid em um programa que mostra as fontes instaladas no sistema, e o controle DrawGrid em um programa que simula o jogo MineSweeper (Campo minado).

Uma Grade de Fontes

Se você colocar um componente StringGrid em um formulário e configurar suas opções corretamente, terá um editor funcional completo de strings organizadas em uma grade, sem realizar qualquer programação. Para tornar o exemplo mais interessante, decidimos desenhar cada célula da grade com uma fonte diferente, variando seu tamanho e seu tipo. Você pode ver o resultado do programa FontGrid na Figura 15.

FIGURA 15

Um exemplo da saída do aplicativo FontGrid. A largura de cada coluna pode ser alterada em tempo de execução.



O formulário desse programa é muito simples. Você precisa apenas colocar um componente grade em um formulário, alinhá-lo com a área cliente, configurar algumas propriedades e opções, e deixar o programa fazer o resto. Na verdade, o número de colunas e linhas e seu tamanho são calculados em tempo de execução. As propriedades importantes que você precisa configurar são `DefaultDrawing`, que deve ser `False` para nos permitir pintar a grade como quisermos, e `Options`:

```
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
```

Como normalmente acontece no Delphi, quanto mais simples for o formulário, mais complexo será o código. Este exemplo segue essa regra, embora ele tenha apenas dois métodos, um para inicia-

lizar a grade na inicialização e outro para desenhar os itens. A edição, na verdade, não foi personalizada e ocorre usando a fonte de sistema. O primeiro dos dois métodos é `FormCreate`. No início, esse método usa o objeto global `Screen` para acessar as fontes instaladas no sistema.

A grade tem uma coluna para cada fonte, assim como uma coluna fixa com números representando tamanhos de fonte. O nome de cada coluna é copiado do objeto `Screen` para a primeira célula de cada coluna (que possui um índice zero):

```
procedure TForm1.FormCreate(Sender: TObject);
var
   I, J: Integer;
begin
   {o número de colunas é igual ao número de fontes mais
   1 para a primeira coluna fixa, que tem um tamanho igual a 20}
   StringGrid1.ColCount := Screen.Fonts.Count + 1;
   StringGrid1.ColWidths [0] := 50;

   for I := 1  to Screen.Fonts.Count do
   begin
   // escreve o nome da fonte na primeira linha
   StringGrid1.Cells [I, 0] :=
     Screen.Fonts.Strings [I-1];

   {calcula o tamanho máximo exigido para a coluna, obtendo a largura do texto com o maior tamanho
   da fonte nessa coluna}
   StringGrid1.Canvas.Font.Name :=
   StringGrid1.Cells [I, 0];
StringGrid1.Canvas.Font.Size := 32;
StringGrid1.ColWidths [I] :=
   StringGrid1.Canvas.TextWidth ('AaBbYyZz');
end;
...
```

Na última parte do código anterior, o programa calcula a largura de cada coluna. Isso é feito através da avaliação do espaço ocupado pela string de texto personalizada *AaBbYyZz*, usando-se a fonte da coluna (escrita na primeira célula, `Cells [I, 0]`) e o maior tamanho de fonte usado pelo programa (32). Para calcular o espaço exigido pelo texto, você pode aplicar os métodos `TextWidth` e `TextHeight` em um canvas, com a fonte correta selecionada.

As linhas, em vez disso, têm sempre uma largura 26 e uma altura que aumenta, calculada com a fórmula aproximada: `15 + I x 2`. Na verdade, calcular o texto mais alto significa verificar a altura do texto em cada coluna, certamente uma operação complexa demais para este exemplo. A fórmula aproximada funciona suficientemente bem, conforme você pode ver executando o programa. Na primeira célula de cada linha, o programa escreve o tamanho da fonte, que corresponde ao número da linha mais sete.

A última operação é armazenar a string "*AaBbYyZz*" em cada célula não fixa da grade. Para fazer isso, o programa usa um laço `for` aninhado. Espere usar laços for aninhados freqüentemente quando trabalhar com grades. Aqui está a segunda parte do método `FormCreate`:

```
// define o número de colunas
StringGrid1.RowCount := 26;
for I := 1  to 25 do
begin
   //  escreve o número na primeira coluna
   StringGrid1.Cells [0, I] := IntToStr (I+7);
   // configura uma altura que aumenta para as linhas
```

```
   StringGrid1.RowHeights [I] := 15 + I*2;
   // insere texto padrão em cada coluna
   for J := 1  to StringGrid1.ColCount do
      StringGrid1.Cells [J, I] := 'AaBbYyZz'
   end;
   StringGrid1.RowHeights [0] := 25;
end;
```

Agora, podemos estudar o segundo método, `StringGrid1DrawCell`, que corresponde ao evento `OnDrawCell` da grade. Esse método tem vários parâmetros:

- `Col` e `Row` se referem à célula que estamos pintando atualmente.
- `Rect` é a área da célula que vamos pintar.
- `State` é o estado da célula, um conjunto de três flags que podem estar ativos ao mesmo tempo: `gdSelected` (a célula está selecionada), `gdFocused` (a célula tem o foco de entrada) e `gdFixed` (a célula está na área fixa, que normalmente tem uma cor de fundo diferente). É importante conhecer o estado da célula, pois isso normalmente afeta sua saída.

O método `DrawCell` pinta o texto do elemento da grade correspondente, com a fonte usada pela coluna e com o tamanho usado para a linha. Aqui está a listagem desse método:

```
procedure TForm1.StringGrid1DrawCell (Sender: TObject;
   Col, Row: Integer; Rect: TRect; State: TGridDrawState);
begin
   // seleciona uma fonte, dependendo da coluna
   if (Col = 0) or (Row = 0)  then
      StringGrid1.Canvas.Font.Name := I
   else
      StringGrid1.Canvas.Font.Name :=
         StringGrid1.Cells [Col, 0];

   // seleciona o tamanho da fonte, dependendo da linha
   if Row = 0  then
      StringGrid1.Canvas.Font.Size := 14
   else
      StringGrid1.Canvas.Font.Size := Row + 7;

   // seleciona a cor de fundo
   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;

   // exibe o texto
   StringGrid1.Canvas.TextRect (
      Rect, Rect.Left, Rect.Top,
      StringGrid1.Cells [Col, Row]);

   // desenha o foco
   if gdFocused in State  then
      StringGrid1.Canvas.DrawFocusRect (Rect);
end;
```

O nome da fonte é recuperado pela linha 0 da mesma coluna. O tamanho da fonte é calculado somando-se 7 ao número da linha. As colunas fixas usam alguns valores padrão. Tendo configurado a fonte e seu tamanho, o programa seleciona uma cor para o fundo da célula, dependendo de seus estados possíveis: selecionada, fixa ou normal (isto é, nenhum estilo especial). O valor do flag `gdFocused` do estilo é usado algumas linhas depois para desenhar o retângulo de foco típico. Quando tudo estiver configurado, o programa poderá produzir alguma saída em si, desenhando o texto e, se necessário, o retângulo de foco, com as duas últimas instruções do método `StringGrid1DrawCell` anterior.

DICA Para desenhar o texto na célula da grade, usamos o método `TextRect` do canvas, em vez do método `TextOut`, mais comum. O motivo é que `TextRect` recorta a saída no retângulo dado, evitando o desenho fora dessa área. Isso é particularmente importante no caso de grades, pois a saída de uma célula não deve ultrapassar suas bordas. Como estamos pintando no canvas da grade inteira, quando estivermos desenhando uma célula, podemos acabar danificando também o conteúdo de células vizinhas.

Como uma observação final, lembre-se de que, quando você decidir desenhar o conteúdo da célula de uma grade, não deve desenhar apenas a imagem padrão, mas também fornecer uma saída diferente para o item selecionado, desenhar o foco corretamente etc.

Minas em uma Grade

O componente StringGrid usa o array `Cells` para armazenar os valores dos elementos e também possui uma propriedade `Objects` para armazenar dados personalizados de cada célula. O componente DrawGrid, por sua vez não tem um armazenamento predefinido. Por esse motivo, o exemplo a seguir define um array bidimensional para armazenar o valor das células da grade — isto é, do campo de reprodução.

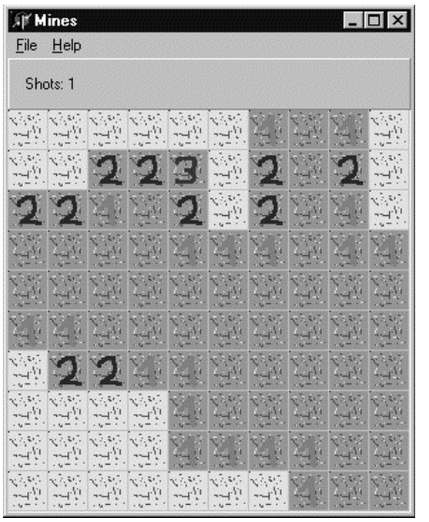
O exemplo Mines é um clone do jogo Campo minado incluído com o Windows. Se você nunca jogou esse jogo, sugerimos que tente fazer isso e leia suas regras no arquivo de ajuda, pois forneceremos apenas uma descrição básica. Quando o programa começa, ele apresenta um campo vazio (uma grade) em que existem algumas minas escondidas. Dando um clique com o botão esquerdo do mouse em uma célula, você testa se há ou não uma mina nessa posição. Se você encontrar uma mina, ela explodirá, e o jogo termina. Você perde.

Se não houver mina na célula, o programa indicará o número de minas nas oito células vizinhas a ela. Conhecendo o número de minas próximas à célula, você tem um bom palpite para a próxima jogada. Para ajudá-lo ainda mais, quando uma célula não possui minas na área vizinha, o número de minas dessas células é apresentado automaticamente e, se uma delas não tiver minas nas vizinhanças, o processo será repetido. Assim, se você tiver sorte, com um único clique de mouse, poderá descobrir um bom número de células vazias (veja a Figura 16).

Quando você achar que encontrou uma mina, basta dar um clique com o botão direito do mouse na célula; isso colocará uma bandeira lá. O programa não diz se sua inferência está correta; a bandeira é apenas um indício para suas tentativas futuras. Se posteriormente você mudar de idéia, pode dar novamente um clique com o botão direito do mouse na célula, para remover a bandeira. Quando tiver encontrado todas as minas, você ganhará, e o jogo termina.

Essas são as regras do jogo. Agora, precisamos implementá-las usando um componente DrawGrid como ponto de partida. Nesse exemplo, a grade é fixa e não pode ser redimensionada ou modificada de qualquer maneira, durante a execução. Na verdade, ela possui células quadradas de 30x30 pixels, que serão usadas para apresentar bitmaps do mesmo tamanho.

FIGURA 16
O programa Mines após um único clique de sorte. Um grupo de células sem minas é apresentado simultaneamente.



O código desse programa é complexo e não é fácil encontrar um ponto de partida para descrevê-lo. Por isso, incluímos mais comentários que o normal no código-fonte que acompanha o capítulo, para que você possa percorrê-lo a fim de entender o que ele faz. Contudo, descreveremos seus elementos mais importantes. Primeiro, os dados do programa são armazenados em dois arrays (declarados como campos `private` do formulário):

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

O primeiro é um array de valores boleanos que indicam se um item deve ser apresentado ou permanecer oculto. Note que o número de linhas e colunas desse array é `NItems`. Você pode mudar livremente essa constante, mas deve redimensionar a grade de acordo. O segundo array, `Map`, contém as posições das minas e bandeiras, e os números das minas vizinhas. Ele usa códigos de caractere, em vez de um tipo de dados de enumeração correto, para utilizar os dígitos de 0 a 8 para indicar o número de minas em torno da célula. Aqui está uma lista dos códigos:

- *M: Mina* oculta indica a posição de uma mina que o usuário ainda não encontrou.
- *K: Mina conhecida* indica a posição de uma mina já encontrada pelo usuário e que tem uma bandeira.
- *W: Mina incorreta* indica uma posição onde o usuário colocou uma bandeira, mas onde não existe mina.
- *0 a 8: Número de minas* indica o número de minas nas células vizinhas.

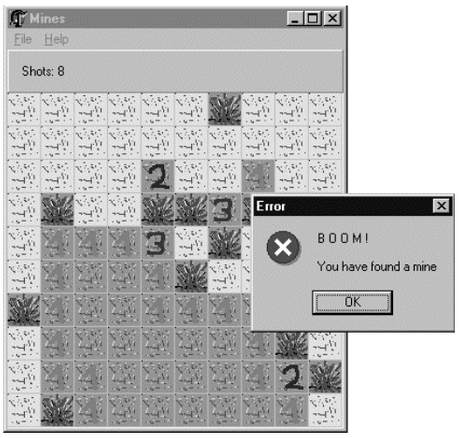
O primeiro método a explorar é `FormCreate`, executado na inicialização. Esse método inicializa vários campos da classe de formulário, preenche os dois arrays com valores padrão (usando dois laços `for` aninhados) e depois estabelece as minas na grade. Para o número de vezes definido em uma constante (isto é, o número de minas), o programa insere uma mina em uma posição aleatória. Entretanto, se já havia uma mina, o laço deve ser executado mais uma vez, pois o número final de minas no array `Map` deve ser igual ao número solicitado. Caso contrário, o programa nunca terminará, pois ele testa quando o número de minas encontradas é igual ao número de minas incluídas na grade. Aqui está o código do laço; ele pode ser executado mais do que `NMines` vezes, graças ao uso da variável inteira `MinesToPlace`, que é aumentada quando tentamos colocar uma mina sobre uma já existente:

```
Randomize;
// coloca 'NMines' minas que não se sobrepõem
MinesToPlace := NMines;
while MinesToPlace > 0 do
begin
   X := Random (NItems);
   Y := Random (NItems);
   // se não houver uma mina
   if Map [X, Y] <> 'M' then
   begin
      // insere uma mina
      Map [X, Y] := 'M';
      Dec (MinesToPlace)
   end;
end;
```

O último trecho do código de inicialização calcula o número de minas vizinhas de cada célula que não tem mina. Isso é feito chamando-se o procedimento `ComputeMines` para cada célula. O código dessa função é bastante complexo, pois ele precisa considerar os casos especiais das minas junto a uma ou duas bordas da grade. O efeito dessa chamada é armazenar no array `Map` o caractere que representa o número de minas vizinhas a cada célula.

O próximo procedimento lógico é `DrawGrid1MouseDown`. Esse método primeiramente calcula a célula em que foi dado o clique de mouse, com uma chamada ao método `MouseToCell` da grade. Em seguida, existem três trechos alternativos de código: um pequeno para quando o jogo tiver terminado e os outros dois para os botões do mouse. Quando o botão esquerdo do mouse é pressionado, o programa verifica se existe uma mina (oculta ou não) e, se houver, ele apresenta uma mensagem e termina com uma explosão (veja a Figura 17).

FIGURA 17
Ai! Você pisou em uma mina!



Se não houver mina, o programa configura o valor `Display` da célula como `True` e, se houver um `0`, ele iniciará o procedimento `FloodZeros`. Esse método apresenta os oito itens próximos a uma célula visível que tenha o valor 0, repetindo a operação seguidamente, caso uma das células vizinhas também tenha o valor 0. Essa chamada recursiva é complexa, pois você tem de fornecer um modo de terminá-la. Se houver duas células próximas, ambas tendo o valor 0, cada uma estará na área vizinha à outra; portanto, elas poderiam continuar para sempre a pedir para que a outra célula exiba a si própria e às suas células vizinhas. Novamente, o código é complexo, e a melhor maneira de estudá-lo pode ser percorrê-lo no depurador.

Quando o usuário pressiona o botão direito do mouse, o programa muda o status da célula. A ação do botão direito do mouse é alternar a bandeira na tela, de modo que o usuário sempre possa remover uma bandeira existente, caso ele ache que a decisão anterior esteja errada. Por isso, o status de uma célula que contém uma mina pode mudar de *M* (mina oculta) para *K* (mina conhecida) e vice-versa. Quando todas as minas tiverem sido encontradas, o programa terminará com uma mensagem de felicitações.

Um trecho muito importante do código está no final do método de resposta ao evento `OnMouseDown`. Sempre que o usuário dá um clique em uma célula e seu conteúdo muda, essa célula deve ser repintada. Se você repintar a grade inteira, o programa ficará mais lento. Por isso, usamos a função de API do Windows `InvalidateRect`:

```
MyRect := DrawGrid1.CellRect (Col, Row);
InvalidateRect (DrawGrid1.Handle, @MyRect, False);
```

O último método importante é `DrawGrid1DrawCell`. Já usamos esse procedimento de pintura no último exemplo, de modo que você deve se lembrar de que ele é chamado para cada célula que precisa de repintura. Fundamentalmente, esse método extrai o código correspondente à célula, o que mostra um bitmap correspondente, carregado a partir de um arquivo. Mais uma vez, preparamos um bitmap para cada uma das imagens em um novo arquivo de recursos, que é incluído no projeto graças ao Project Manager melhorado do Delphi 5.

Lembre-se de que, ao usar recursos, o código tende a ser mais rápido do que o uso de arquivos separados e, novamente, acabamos com um único arquivo executável para instalar. Os bitmaps têm nomes correspondentes ao código da grade, com um caractere (‘*M*’) na frente, pois o nome ‘*0*’ seria inválido. Os bitmaps podem ser carregados e desenhados na célula com o seguinte código:

```
Bmp.LoadFromResourceName (HInstance, 'M' + Code);
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

É claro que isso ocorrerá apenas se a célula estiver visível — isto é, se `Display` for `True`. Caso contrário, um bitmap indefinido padrão será apresentado. (O nome do bitmap é ‘*UNDEF*’.) Carregar os bitmaps a partir dos recursos toda vez parece lento, de modo que o programa poderia ter armazenado todos os bitmaps em uma lista na memória, como fez o exemplo World2, já visto neste capítulo. Entretanto, desta vez, decidimos usar uma estratégia diferente, embora ligeiramente menos eficiente: uma cache. Isso faz sentido, pois já usamos recursos em lugar de arquivos para acelerar as coisas.

A cache de bitmap do programa Mines é pequena, pois ela tem apenas um elemento, mas sua presença acelera o programa consideravelmente. O programa armazena o último bitmap que usou e seu código; então, sempre que ele precisar desenhar um novo item, se o código for o mesmo, usará o bitmap da cache. Aqui está a nova versão do código anterior:

```
if not (Code = LastBmp) then
begin
   Bmp.LoadFromResourceName (HInstance, 'M' + Code);
   LastBmp := Code;
end;
DrawGrid1.Canvas.Draw (Rect.Left, Rect.Top, Bmp);
```

Aumentar o tamanho dessa cache certamente melhorará a velocidade do programa. Você pode considerar uma lista de bitmaps como uma cache grande, mas isso provavelmente é inútil, pois alguns bitmaps (aqueles com números altos) são raramente usados. Conforme você pode ver, algumas melhorias podem ser feitas para acelerar o programa, e muito também pode ser feito para melhorar sua interface com o usuário. Se você tiver entendido esta versão do programa, achamos que poderá aprimorá-lo consideravelmente.

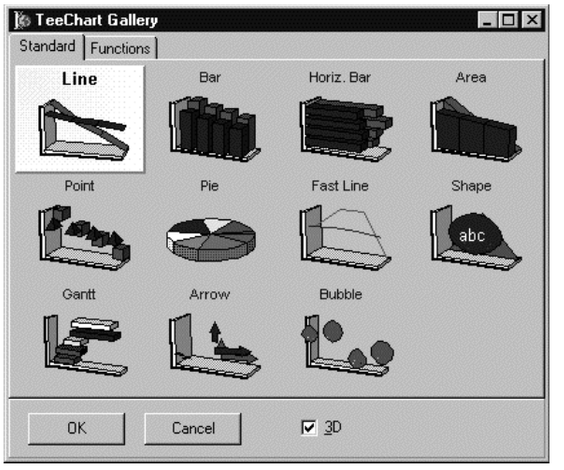
Usando TeeChart

O TeeChart é um componente baseado na VCL, criado por David Berneda e licenciado para a Borland, para a inclusão nas versões Developer e Enterprise do Delphi. O componente TeeChart é muito complexo: o Delphi inclui um arquivo de ajuda e outros materiais de referência para esse componente; portanto, não perderemos tempo listando todos os seus recursos. Construiremos apenas dois exemplos. O TeeChart aparece em três versões: o componente independente (na página Additional da paleta de componentes), a versão consciente de dados (na página Data Controls) e a versão relatório (na página QReport). O Delphi Enterprise também inclui um controle DecisionChart na página Decision Cube da paleta. A versão consciente de dados do TeeChart foi apresentada no Capítulo 13 deste livro, e a usaremos novamente em um exemplo orientado à Web.

NOTA É claro que seria mais simples construir um exemplo usando o TeeChart Wizard, mas ver todos os passos dará a você um melhor entendimento da estrutura desse componente.

O componente TeeChart fornece a estrutura básica para a produção de gráficos, através de uma base complexa de gráficos e uma série de classes e do contêiner visual para gráficos (o controle em si). Os gráficos são objetos da classe `TChartSeries` ou de classes derivadas. Uma vez que você tenha colocado o componente TeeChart em um formulário, deve criar uma ou mais séries. Para fazer isso, você pode abrir o Chart Component Editor: selecione o componente, dê um clique com o botão direito do mouse para apresentar o menu local do projetista de formulário e escolha o comando Edit Chart. Agora, pressione o botão Add (na página Séries da guia Chat) e escolha o gráfico (ou série) que você deseja incluir dentre os muitos disponíveis (conforme você pode ver na Figura 18).

FIGURA 18
A TeeChart Gallery permite que você escolha o tipo de gráfico ou uma série.



Assim que você cria uma nova série, um novo objeto de uma subclasse `TChartSeries` é incluído em seu formulário. Esse é o mesmo comportamento do componente MainMenu, que inclui objetos da classe `TMenuItem` no formulário. Você pode então editar as propriedades do objeto `TSeries` no Chart Component Editor, ou pode selecionar o objeto `TChartSeries` no Object Inspector (com a caixa de combinação Object Selector) e editar suas muitas propriedades.

As diferentes subclasses `TChartSeries` — isto é, os diferentes tipos de gráfico — têm diferentes propriedades e métodos (embora alguns deles sejam comuns a mais de uma subclasse). Lembre-se de que um gráfico pode ter várias séries: se elas forem todas do mesmo tipo, provavelmente vão se in-

tegrar melhor, como no caso de várias barras. De qualquer forma, você também pode ter um layout complexo com gráficos de diferentes tipos visíveis simultaneamente. Às vezes, essa é uma opção extremamente poderosa.

Construindo um Primeiro Exemplo

Para construir este exemplo, colocamos um componente TeeChart em um formulário e depois simplesmente incluímos quatro séries 3D Bar — isto é, quatro objetos da classe `TBarSeries`. Em seguida, configuramos algumas propriedades, como o título do gráfico etc. Aqui está um resumo dessas informações, extraídas da descrição textual do formulário:

```
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
```

Em seguida, incluímos no formulário uma grade de string e um botão de seleção chamado *Update*. Esse botão é usado para copiar os valores numéricos da grade de string no gráfico. A grade é baseada em uma matriz de 5x4, assim como uma linha e uma coluna para os títulos. Aqui está sua descrição textual:

```
object StringGrid1: TStringGrid
   ColCount = 6
   DefaultColWidth = 50
   Options = [goFixedVertLine, goFixedHorzLine,
     goVertLine, goHorzLine, goEditing]
   ScrollBars = ssNone
   OnGetEditMask = StringGrid1GetEditMask
end
```

O valor 5 para a propriedade `RowCount` é o padrão, e ele não aparece na descrição textual. (O mesmo vale para o valor 1 para as propriedades `FixedCols` e `FixedRows`.) Um elemento importante dessa grade de strings é a máscara de edição usada por todas as suas células. Isso é ajustado usando-se o evento `OnGetEditMask`:

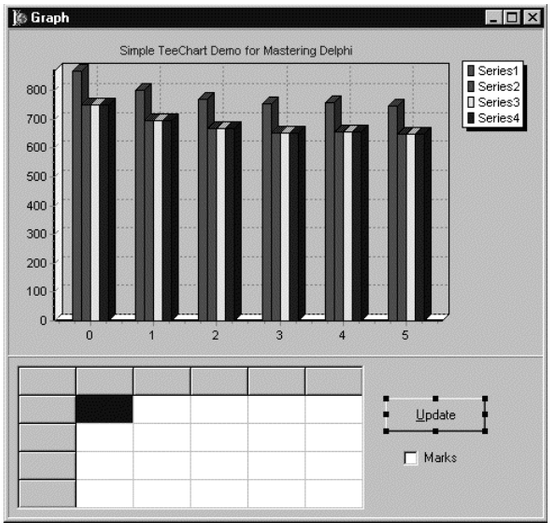
```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject;
  ACol, ARow: Longint; var Value: string);
```

```
begin
  // máscara de edição para as células da grade
  Value := '09;0';
end;
```

Na verdade, existe mais um componente importante, uma caixa de verificação usada para alternar a visibilidade dos marcadores da série. (Os *marcadores* são pequenos rótulos amarelos que descrevem cada valor; você precisará executar o programa para vê-los.) Você pode ver o formulário em tempo de projeto na Figura 19. Neste caso, as séries são populadas por valores aleatórios; esse é um excelente recurso do componente, pois ele permite que você veja uma prévia da saída sem introduzir dados em si.

FIGURA 19

O exemplo Graph1, baseado no componente TeeChart, em tempo de projeto.



Incluindo Dados no Gráfico

Agora, basta inicializarmos os dados da grade de string e copiá-los nas séries do gráfico. Isso ocorre no manipulador do evento `OnCreate` do formulário. Esse método preenche os itens fixos da grade e os nomes das séries, em seguida, preenche a parte relativa aos dados da grade de strings e, por fim chama o manipulador do evento `OnClick` do botão *Update*, para atualizar o gráfico:

```
procedure TForm1.FormCreate(Sender: TObject);
var
   I, J: Integer;
begin
   with StringGrid1 do
   begin
     {preenche a coluna e a fileira fixas,
     e os nomes da séries do gráfico}
     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;
```

```
      // preenche a grade com valores aleatórios
      Randomize;
      for I := 1  to 5 do
         for J := 1  to 4 do
            Cells [I, J] := IntToStr (Random (100));
   end; //  com

   // atualiza o gráfico
   UpdateButtonClick (Self);
end;
```

Podemos acessar as séries usando o nome do componente (como `Series1`) ou usando a propriedade de array `Series` do gráfico, como em `Chart1.Series[J-1]`. Nessa expressão, note que os dados em si da grade de strings começam na linha e na coluna 1 — a primeira linha e coluna, indicadas pelo índice zero, são usadas para os elementos fixos — e o array de gráficos `Series` é baseado em zero.

Outro exemplo de atualização de cada série está presente no manipulador do evento `OnClick` da caixa de seleção; esse método alterna a visibilidade dos marcadores:

```
procedure TForm1.ChBoxMarksClick(Sender: TObject);
var
   I: Integer;
begin
   for I := 1  to 4 do
     Chart1.Series [I-1].Marks.Visible :=
        ChBoxMarks.Checked;
end;
```

Mas o código realmente interessante está no método `UpdateButtonClick`, que atualiza o gráfico. Para fazer isso, o programa primeiro remove os dados existentes de cada gráfico e depois inclui novos dados (ou *pontos de dados*, para usar o jargão correto):

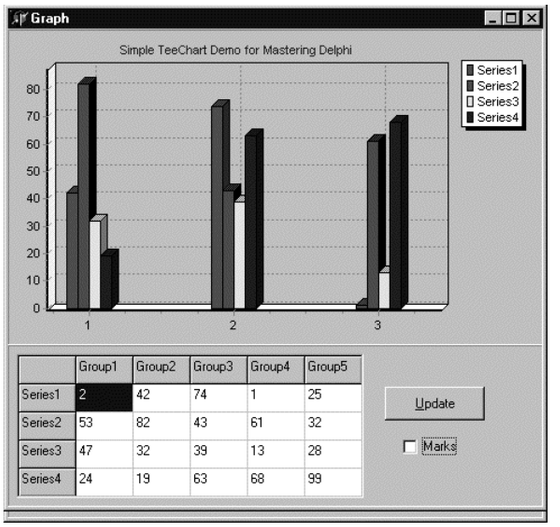
```
procedure TForm1.UpdateButtonClick(Sender: TObject);
var
   I, J: Integer;
begin
   for I := 1  to 4 do
   begin
      Chart1.Series [I-1].Clear;
      for J := 1  to 5 do
         Chart1.Series [I-1].Add (
            StrToInt (StringGrid1.Cells [J, I]),
            '', Chart1.Series [I-1].SeriesColor);
   end;
end;
```

Os parâmetros do método `Add` (usados quando você não quer especificar um valor `X`, mas apenas um valor Y) são o valor em si, o rótulo e a cor. Neste exemplo, o rótulo não é usado; portanto, simplesmente o omitimos. Poderíamos ter usado o valor padrão `clTeeColor` para obter a cor correta das séries. Você poderia usar cores específicas para indicar diferentes intervalos de dados.

Uma vez que você tenha construído o gráfico, o TeeChart permite muitas opções de visualização. Você pode facilmente ampliar a visualização (basta indicar a área com o botão esquerdo do mouse), reduzir (usando o mouse do modo oposto, arrastando em direção ao canto superior esquerdo) e usar o botão direito do mouse para mover a visualização. Você pode ver um exemplo de zoom na Figura 20.

FIGURA 20

O formulário do exemplo Graph1 em tempo de execução. Note que demos um zoom no gráfico.



Criando Séries Dinamicamente

O exemplo Graph1 mostra alguns dos recursos do componente TeeChart, mas ele tem por base um único tipo de gráfico fixo. Poderíamos tê-lo aprimorado, permitindo alguma personalização da forma das barras verticais; em vez disso, escolhemos uma estratégia mais geral, permitindo ao usuário escolher diferentes tipos de séries (gráficos).

Inicialmente, o componente TeeChart tem os mesmos atributos que no exemplo anterior. Mas agora o formulário tem quatro caixas de combinação, uma para cada linha da grade de strings. Cada caixa de combinação tem quatro valores (Line, Bar, Area e Point), correspondentes aos quatro tipos de séries que queremos manipular. Para manipular essas caixas de combinação de uma maneira mais flexível no código, incluímos um array desses controles nos campos privados do formulário:

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

Esse array é preenchido com o componente em si no método `FormCreate`, que também seleciona o item inicial de cada um deles. Aqui está o novo código de `FormCreate`:

```
// preenche o array Combos
Combos [0] := ComboBox1;
Combos [1] := ComboBox2;
Combos [2] := ComboBox3;
Combos [3] := ComboBox4;
// mostra o tipo de gráfico inicial
for I := 0  to 3 do
   Combos [I].ItemIndex := 1;
```

DICA Esse exemplo demonstra um modo comum de criar um array de controles no Delphi, algo que os programadores Visual Basic desejam freqüentemente. Na verdade, o Delphi é tão flexível que os arrays de controles não são incorporados; você pode criá-los como quiser. Essa estratégia conta com o fato de que geralmente é possível associar o mesmo manipulador de evento a diferentes eventos, algo que o VB não permite fazer.

Todas essas caixas de combinação compartilham o mesmo manipulador de eventos `OnClick`, que destrói cada uma das séries atuais do gráfico, cria as novas quando solicitadas e depois atualiza suas propriedades e dados:

```
procedure TForm1.ComboChange(Sender: TObject);
var
   I: Integer;
   SeriesClass: TChartSeriesClass;
   NewSeries: TChartSeries;
begin
   // destrói as séries existentes (em ordem inversa)
   for I := 3 downto 0 do
      Chart1.Series [I].Free;
   // cria as novas séries
   for I := 0  to 3 do
   begin
      case Combos [I].ItemIndex of
         0: SeriesClass := TLineSeries;
         1: SeriesClass := TBarSeries;
         2: SeriesClass := TAreaSeries;
      else // 3: e padrão
         SeriesClass := TPointSeries;
      end;
      NewSeries := SeriesClass.Create (self);
      NewSeries.ParentChart := Chart1;
      NewSeries.Title :=
         Format ('Series %d', [I + 1]);
   end;
   // atualiza os marcadores e os dados
   ChBoxMarksClick (self);
   UpdateButtonClick (self);
   Modified := True;
end;
```

A parte central desse código é a instrução `case`, que armazena uma nova classe na variável de referência de classe `SeriesClass`, usada para criar os novos objetos de série e configurar as propriedades `ParentChart` e `Title` de cada um. Também poderíamos ter usado uma chamada ao método `AddSeries` do gráfico em cada desvio da instrução `case` e depois configurado a propriedade `Title` com outro laço `for`. Na verdade, uma chamada como

```
Chart1.AddSeries (TBarSeries.Create (self));
```

cria os objetos da série e ao mesmo tempo configura seu gráfico progenitor.

Note que essa nova versão do programa permite que você mude o tipo de gráfico para cada série independentemente. Você pode ver um exemplo do efeito resultante na Figura 21.

Por fim o exemplo Graph2 permite salvar os dados atuais que está apresentando em um arquivo e carregar os dados de arquivos existentes. O programa tem uma variável booleana `Modified`, usada para saber se o usuário mudou qualquer um dos dados. O suporte a arquivos é baseado em streams e não é particularmente complexo, pois o número de elementos a salvar é fixo (todos os arquivos têm o mesmo tamanho). Aqui estão os dois métodos conectados aos itens de menu Open e Save:

```
procedure TForm1.Open1Click(Sender: TObject);
var
   LoadStream: TFileStream;
   I, J, Value: Integer;
```

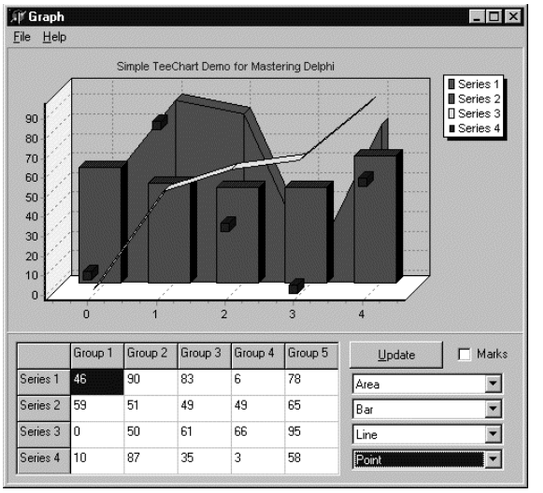
```
begin
   if OpenDialog1.Execute  then
   begin
      CurrentFile := OpenDialog1.Filename;
      Caption := 'Graph [' + CurrentFile + ']';
      //  carrega do arquivo atual
      LoadStream := TFileStream.Create (
        CurrentFile, fmOpenRead);
 try
    // lê o valor de cada elemento de grade
    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;
 //  carrega o status da caixa de seleção e das caixas de combinação
 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;
      // dispara atualizar eventos
      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 // chama "Salvar como"
         SaveAs1Click (Self)
      else
      begin
         // salva o arquivo atual
         SaveStream := TFileStream.Create (
            CurrentFile, fmOpenWrite or fmCreate);
         try
            //  escreve o valor de cada elemento da grade
            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;
// salva a caixa de seleção e as caixas de combinação
Value := Integer (C BoxMarks.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;
```

FIGURA 21

Vários tipos de gráficos ou séries de gráfico apresentados pelo exemplo Graph2.



	Group 1	Group 2	Group 3	Group 4	Group 5
Series 1	46	90	83	6	78
Series 2	59	51	49	49	65
Series 3	0	50	61	66	95
Series 4	10	87	35	3	58

Um Gráfico de Banco de Dados na Web

No Capítulo 22 deste livro, vimos como criar uma figura simples e retorná-la a partir de um aplicativo CGI. Podemos aplicar a mesma estratégia para retornar um gráfico complexo e dinâmico, construído com o componente `TDBChart`. Usar esse componente na memória é um pouco mais complexo do que configurar todas as suas propriedades em tempo de projeto, pois você terá que configurar as propriedades no código Pascal. (Você não pode usar um componente visual, como um DBChart, em um módulo Web ou em qualquer outro módulo de dados.)

No aplicativo ISAPI WebChart, usamos a tabela Country.DB para produzir um gráfico de pizza com a área e a população dos países da América, como no exemplo ChartDb do Capítulo 13 deste livro. Os dois gráficos são gerados por duas ações diferentes, indicadas pelos caminhos `/population` e `/area`. Como a maior parte do código é usada mais de uma vez, o reunimos nos eventos `OnCreate` e `OnAfterDispatch` do módulo da Web.

ALERTA Da maneira como está escrito, este programa não oferece suporte a usuários concorrentes. Você precisará incluir algum código de linha de execução (threading) e sincronismo nessa DLL ISAPI, para fazê-lo funcionar com vários usuários simultaneamente. Uma alternativa é colocar todo o código nos manipuladores de eventos Action, para que nenhum objeto global seja compartilhado entre vários pedidos. Ou, então, você pode transformá-lo em um aplicativo CGI.

O módulo de dados tem um objeto tabela, que é corretamente inicializado em tempo de projeto, e três campos privados:

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

Os objetos correspondentes a esses campos são criados junto ao módulo da Web (e usados por chamadas subseqüentes):

```
procedure TWebModule1.WebModule1Create(Sender: TObject);
begin
   // abre a tabela de banco de dados
   Table1.Open;
   // cria o gráfico
   Chart := TDBChart.Create (nil);
   Chart.Width := 600;
   Chart.Height := 400;
   Chart.AxisVisible := False;
   Chart.Legend.Visible := False;
   Chart.BottomAxis.Title.Caption := 'Name';
   // cria a série de pizza
   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);
   // cria o bitmap de memória
   Image := TImage.Create (nil);
   Image.Width := Chart.Width;
   Image.Height := Chart.Height;
end;
```

O próximo passo é executar o manipulador da ação específica, que configura a série do gráfico de pizza para o campo de dados específico e atualiza alguns títulos:

```
procedure TWebModule1.WebModule1ActionPopulationAction(
   Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
   // configura os valores específicos
   Chart.Title.Text.Clear;
   Chart.Title.Text.Add ('Population of Countries');
   Chart.LeftAxis.Title.Caption := 'Population';
   Series.Title := 'Population';
   Series.PieValues.ValueSource := 'Population';
end;
```

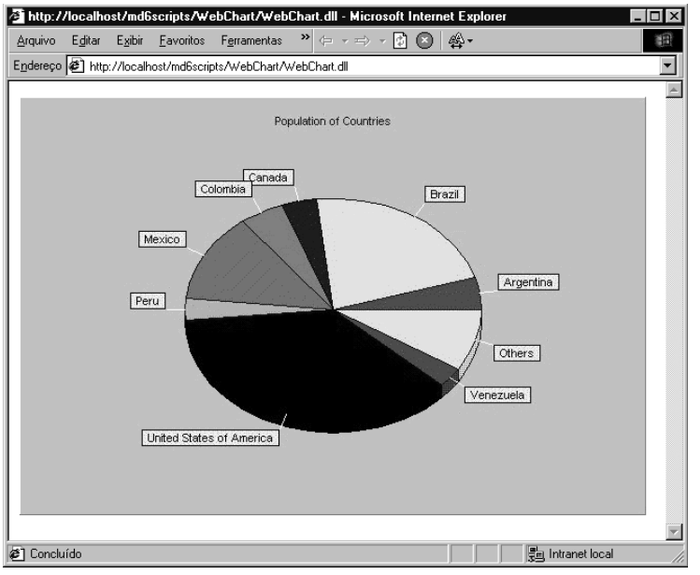
Isso cria o componente DBChart correto na memória. O último passo, novamente comum para as duas ações, é salvar o gráfico em uma imagem de bitmap e depois formatá-lo como um JPEG em um stream, para ser posteriormente retornado do aplicativo no lado do servidor. O código é parecido com aquele do exemplo anterior:

```
procedure TWebModule1.WebModule1AfterDispatch(
   Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
var
   Jpeg: TJpegImage;
   MemStr: TMemoryStream;
begin
   // pinta o gráfico no bitmap de memória
   Chart.Draw (Image.Canvas, Image.BoundsRect);
   // cria o jpeg e copia a imagem nele
   Jpeg := TJpegImage.Create;
   try
      Jpeg.Assign (Image.Picture.Bitmap);
      MemStr := TMemoryStream.Create;
      // salva em um stream e o retorna
      Jpeg.SaveToStream (MemStr);
      MemStr.Position := 0;
      Response.ContentType := 'image/jpeg';
      Response.ContentStream := MemStr;
      Response.SendResponse;
   finally
      Jpeg.Free;
   end;
end;
```

O resultado, visível na Figura 22, é certamente interessante. Como opção, você pode estender esse aplicativo vinculando-o a uma tabela HTML que mostra os dados do banco de dados. Basta escrever um programa com uma ação principal que retorne a tabela HTML e uma referência à figura incorporada, que será retornada por uma segunda ativação da DLL ISAPI com um caminho diferente.

FIGURA 22

O JPEG com o gráfico de população gerado pelo aplicativo WebChart.



Usando Meta-Arquivos

Os formatos de bitmap abordados anteriormente neste capítulo armazenavam o status de cada pixel de um bitmap, embora eles normalmente compactassem a informação. Um tipo totalmente diferente de formato gráfico é representado pelos formatos orientados a vetores. Nesse caso, o arquivo armazena a informação exigida para recriar a figura, como os pontos inicial e final de cada linha ou o cálculo matemático que define uma curva. Existem muitos formatos de arquivo orientados a vetores diferentes, mas o único suportado pelo sistema operacional Windows é o WMF (Windows Metafile Format) — formato de meta-arquivos do Windows. Esse formato foi estendido no Win32 para o EMF (Extended Metafile Format) — formato de meta-arquivo estendido, que armazena informações extras relacionadas aos modos de mapeamento e ao sistema de coordenadas.

Um meta-arquivo do Windows é basicamente uma série de chamadas às funções primitivas da GDI. Depois de ter armazenado a seqüência de chamadas, você pode *executá-las*, reproduzindo as figuras. O Delphi suporta meta-arquivos do Windows através das classes `TMetafile` e `TMetaFileCanvas`. Construir um exemplo é muito simples.

A classe `TMetafile` é usada para manipular o arquivo em si, com métodos para carregar e salvar os arquivos, e propriedades que determinam os principais recursos do arquivo. Uma delas é a propriedade `Enhanced`, que determina o tipo de formato de meta-arquivo. Note que, quando o Windows está lendo um arquivo, a propriedade `Enhanced` é configurada dependendo da extensão de arquivo — WMF para meta-arquivos do Windows 3.1 e EMF para os meta-arquivos estendidos do Win32.

Para gerar um meta-arquivo, você pode usar um objeto da classe `TMetafileCanvas`, conectado ao arquivo através de seus construtores, como se vê no seguinte trecho de código:

```
Wmf := TMetafile.Create;
WmfCanvas := TMetafileCanvas.CreateWithComment(
   Wmf, 0, 'Marco', 'Demo metafile');
```

Uma vez que você tenha criado os dois objetos, pode pintar sobre o objeto canvas com chamadas regulares e, no final, salvar o meta-arquivo conectado a um arquivo físico.

Uma vez que você tenha o meta-arquivo (um novo que acabou de criar ou um construído com outro programa), pode apresentá-lo em um componente Image ou pode simplesmente chamar os métodos `Draw` ou `StretchDraw` de qualquer canvas.

No exemplo WmfDemo, escrevemos algum código simples, apenas para mostrar a você os fundamentos dessa estratégia. O manipulador do evento `OnCreate` do formulário cria o meta-arquivo estendido, um objeto simples que é usado para operações de leitura e escrita:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   Wmf := TMetafile.Create;
   Wmf.Enhanced := True;
   Randomize;
end;
```

O formulário do programa tem dois botões e os componentes PaintBox, mais uma caixa de seleção. O primeiro botão cria um meta-arquivo, gerando uma série de linhas parcialmente aleatórias. O resultado aparece no primeiro componente PaintBox e também é salvo em um arquivo fixo:

```
procedure TForm1.BtnCreateClick(Sender: TObject);
var
   WmfCanvas: TMetafileCanvas;
   X, Y: Integer;
begin
```

```
   // cria o canvas virtual
   WmfCanvas := TMetafileCanvas.CreateWith Comment(
      Wmf, 0, 'Marco', 'Demo metafile');

   try
      // limpa o fundo
      WmfCanvas.Brush.Color := clWhite;
      WmfCanvas.FillRect (WmfCanvas.ClipRect);

      // desenha 400 linhas
      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
      // finaliza a operação de desenho
      WmfCanvas.Free;
   end;

   // apresenta o desenho atual e o salva
   PaintBox1.Canvas.Draw (0, 0, Wmf);
   Wmf.SaveToFile (ExtractFilePath (
      Application.ExeName) + 'test.emf');
end;
```

ALERTA Se você desenhar ou salvar o meta-arquivo antes que o canvas do meta-arquivo conectado seja fechado ou destruído, essas operações não produzirão nenhum efeito! Esse é o motivo pelo qual chamamos o método `Free` antes de chamar `Draw` e `SaveToFile`.

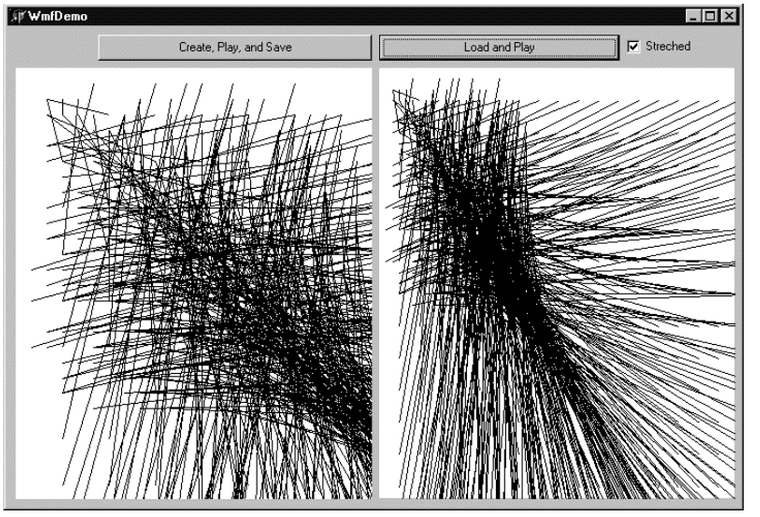
Recarregar e repintar o meta-arquivo é ainda mais simples:

```
procedure TForm1.BtnLoadClick(Sender: TObject);
begin
   //  carrega o meta-arquivo
   Wmf.LoadFromFile (ExtractFilePath (
      Application.ExeName) + 'test.emf');

   // desenha-o ou estica-o
   if cbStretched.Checked  then
      PaintBox2.Canvas.StretchDraw PaintBox2.Canvas.ClipRect, Wmf)
   else
      PaintBox2.Canvas.Draw (0, 0, Wmf);
end;
```

Note que você pode reproduzir exatamente o mesmo desenho, mas também modificá-lo com a chamada a `StretchDraw`. (O resultado dessa operação aparece na Figura 23.) Essa operação é diferente do esticamento de um bitmap, que normalmente degrada ou modifica a imagem, pois aqui estamos mudando a escala através da alteração do mapeamento de coordenadas. Isso significa que, ao imprimir um meta-arquivo, você pode esticá-lo para preencher a página inteira com um efeito muito bom, algo muito difícil de fazer com um bitmap.

FIGURA 23
A saída do exemplo WmfDemo com um meta-arquivo alongado.



Girando Texto

Neste capítulo, abordamos muitos exemplos diferentes do uso de bitmaps e criamos figuras de muitos tipos. Entretanto, o tipo mais importante de figura com o qual normalmente tratamos nos aplicativos Delphi é texto. Na verdade, mesmo quando apresenta um rótulo ou o texto de uma caixa de edição, o Windows ainda o pinta da mesma maneira como qualquer outro elemento gráfico. Já apresentamos um exemplo de pintura de fonte anteriormente neste capítulo, no exemplo FontGrid. Agora, vamos voltar a esse assunto com uma estratégia ligeiramente menos comum.

Quando você pinta texto no Windows, não há meios de indicar a direção da fonte: o Windows parece desenhar o texto apenas horizontalmente. Entretanto, para sermos precisos, o Windows desenha o texto na direção aceita por sua fonte, que é horizontal como padrão. Por exemplo, podemos mudar o texto apresentado pelos componentes de um formulário, modificando a fonte do próprio formulário, como fizemos no exemplo SideText. Na verdade, você não pode modificar uma fonte, mas pode criar uma nova fonte, semelhante a uma já existente:

```
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; // falso
   ALogFont.lfUnderline := 0; // falso
   ALogFont.lfStrikeOut := 0; // falso
   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));
   Font := CreateFontIndirect (ALogFont);
   Font.Handle :=  Font;
end;
```

Esse código produziu o efeito desejado no rótulo do formulário do exemplo, mas, se você incluir outros componentes nele, o texto geralmente será impresso fora da parte visível do componente. Em outras palavras, você precisará fornecer esse tipo de suporte dentro de componentes se quiser que tudo apareça corretamente. Para os rótulos, entretanto, você pode evitar criar um novo componente; em vez disso, basta mudar a fonte associada ao `Canvas` do formulário (e não o formulário inteiro) e usar os métodos de desenho de texto padrão. O exemplo SideText muda a fonte do `Canvas` no método `OnPaint`, que é semelhante a `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;
```

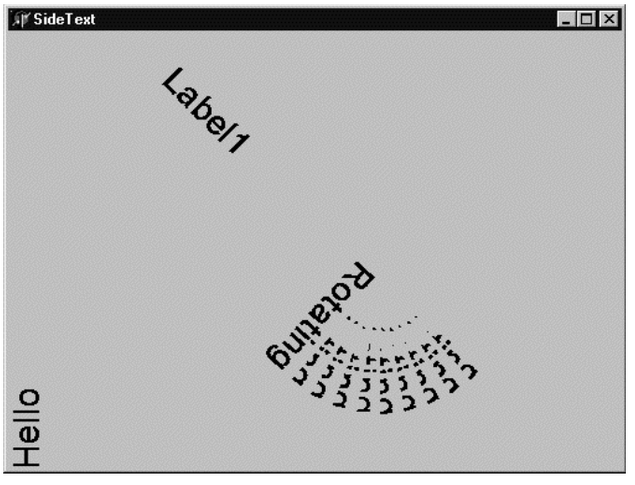
A orientação da fonte também é modificada por um terceiro manipulador de eventos, associado a um temporizador. Seu efeito é girar o formulário com o passar do tempo, e seu código é muito parecido com o procedimento anterior, com exceção do código para determinar o *escape* de fonte (o ângulo da rotação da fonte):

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

Com essas três técnicas diferentes de rotação de fonte (título de rótulo, texto pintado e texto girando com o passar do tempo), o formulário do programa SideText em tempo de execução é semelhante à Figura 24.

FIGURA 24

Os efeitos do exemplo SideText, com texto realmente girando.



O Que Vem a Seguir?

Neste capítulo, exploramos várias técnicas diferentes que você pode usar no Delphi para produzir saída gráfica. Usamos o `Canvas` do formulário, bitmaps, meta-arquivos, componentes gráficos, grades e outras técnicas. Certamente existem muito mais técnicas relacionadas à programação gráfica no Delphi e no Windows em geral, incluindo a grande área da programação de jogos de alta velocidade.

Permitindo que você se vincule diretamente à API do Windows, o suporte do Delphi a figuras é certamente amplo. Entretanto, a maioria dos programadores Delphi nunca faz chamadas diretas ao sistema GDI, mas, em vez disso, conta com o suporte oferecido pelos componentes existentes no Delphi. Esse assunto já foi apresentado no Capítulo 12 do livro.

Se você já leu o livro, esperamos que também tenha aproveitado este capítulo extra. Caso tenha começado por este capítulo, o restante do livro tem muito a oferecer, mesmo no contexto dos recursos gráficos, mas certamente não apenas limitado a isso. Consulte os endereços www.pearsonedbrasil.com/delphi6_cantu para obter mais informações sobre o livro e `www.marcocantu.com` (em inglês) para pegar o código-fonte gratuito deste capítulo e do livro inteiro (que também está no CD que acompanha o livro).