Curso de criação de componentes em Delphi

Unidade 5. O objeto Canvas e o método Paint: TGradiente

Voltar ao índice

Por Luis Roche

Nesta unidade nós criaremos um componente gráfico do tipo que é usado como fundo na instalação de aplicações: TGradient. Mas nosso componente irá mais adiante ele terá possibilidades novas para o gráfico standard. Nós aprenderemos a utilizar o canvas que prepara todos os componentes gráficos. Nós estudaremos o método paint, como usar isto em nossos componentes.

Objetivo do componente

Nosso propósito é criar um componente no qual nós puxaremos um gradient semelhante de cores para qual é mostrado na maioria dos programas de instalação de aplicações (e em outro tipo de programas). Entre as características adicionais que nosso componente deve ter é necessário destaquar que o gradiente de cores não concluirá na cor preta, mas poderá começar e terminar em duas cores quaisquer. Também, o gradiente poderá estar horizontal ou vertical. Ultimamente, o tamanho de nosso desenho (quer dizer, do canvas do componente) será variável, deste modo nós podemos combinar dois ou mais componente para criar efeitos de cores espetaculares.

Nosso componente derivará então de TGraphicControl do que nós precisamos que tem Canvas, mas não é necessário que tenha manipulador (o handler). Esta escolha faz isto baseado no que foi visto na unidade 2.

A base da programação gráfica em Delphi: o Canvas do objeto.

Tradicionalmente, a programação gráfica em janelas foi levada por meio do uso da interface gráfica de dispositivo (GDI). Esta interface é uma ferramenta poderosa que permite o desenho de gráficos por meio do uso de pincéis, escovas, retângulos, elipses, etc.

Mas o GDI tem uma inconveniência: a programação dele é muito laboriosa. Quando eles fazem uso da função GDI, é necessária uma handle, como também criar e destruir as ferramentas de desenho diversos (recursos) isso é usado. Por último , quando concluindo, o contexto de dispositivo deve ser restabelecido a seu estado original antes de destruir-lo.

Delphi encapsula o GDI e faz que nós não tenhamos que nos preocupar sobre contextos de dispositivo nenhum e se nós liberamos ou não os recursos usados. Deste modo nós podemos nos concentrar em nosso objetivo principal: puxar os gráficos. De todos os modos, se nós queremos isto, nós podemos continuar usando as funções GDI se nos interessa. Nós temos um grande leque de possibilidades, Delphi ou GDI, podemos usá-los do modo que nos convenha.

Como foi mencionado no parágrafo anterior, o Delphi nos provê uma interface gráfica completo. O objeto principal desta interface é o objeto **Canvas** . **O objeto Canvas se encarrega de ter um contexto de dispositivo válido, como também de liberar -se quando nós não usarmos mais. Também, o objeto Canvas (ou simplesmente Canvas) tem propriedades diversas que representam o lápis atual (pen), escova (brush) e fonte (font).**

O canvas administra todos estes recursos para nós, razão por que nós temos bastante o que informar à aquela classe de caneta e ele se encarrega do resto. Também, deixando o Delphi fazer o serviço pesado de criar e liberar os recursos gráficos, em muitos casos um aumento de velocidade acontecerá se nós mesmos administrarmos estes recursos.

A tela de objeto encapsula a programação gráfica a três níveis de profundidade que é o seguinte:

Nível	**Operação**	**Ferramentas**
Alto	Desenho de linhas e formas Visualização e modificação de texto Recursos de áreas	Métodos MoveTo, LineTo, Retângulo, Elipse, Métodos TextOut, TextHeight, TextWidth, TextRect, Métodos FillRect, FloodFill,
Médio	Personalizar texto e gráficos Pixels de manipulação Copia e união de imagens	Propriedades Pen, Brush e Font Propriedade Pixels Draw, StretchDraw, BrushCopy, CopyRect, CopyMode,
Baixo	Chamadas para funções GDI	Handle de propriedade

Agora nós não vamos explicar tudo e cada ferramenta disponível para a criação de gráficos, mas nós faremos isto como nas proximas unidades. De todos os modos, uma descrição detalhada de cada um deles está na ajuda on-line do delphi

Bem, nós já sabemos que a tela de objeto nos provê uma interface gráfica muito poderosa de criação gráfica. Mas agora as perguntas seguintes podem ser esboçadas: Onde o objeto Canvas reside?, todos os componentes têm isto?, e se não é deste modo, como os componentes incorporam isto?

A resposta para estas perguntas é simples: Todos os objetos derivados de TGraphicComponent possuem Canvas. Em relação a outros componentes, depende. No caso de dúvida, a coisa mais simples é consultar o Object Browser e verificar se o componente em questão possui canvas ou não (se é declarado em um modo protegido ou público). Deste modo por exemplo, o componente Tlabel, Timage e TPanel possuem isto, enquanto tais componentes como TButton ou TRadioButton não. Este aspecto determinará o ascendente a ser escolhido quando nós queremos desenvolver um componente que devera ter Canvas.

A forma de consentir o canvas de um objeto é muito simples. Vamos supor que nós temos um componente (TGradient) derivou da classe TGraphicControl. E vamos supor que nós queremos

tirar no canvas uma linha do ponto (0,0) até o ponto (20,20). Para esta tarefa nós deveríamos escrever a seguinte linha:

```
TGradient.Canvas.MoveTo(0,0);
TGradient.Canvas.LineTo(20,20);
```

Vamos ver agora as propriedades do Canvasa que nós usaremos no desenvolvimento de nosso componente. Basicamente TGradient deveria puxar uma série de retângulos coloridos com uma certa cor. Para isso nós usaremos o método FillRect. O método FillRect recebe como parâmetro um objeto do tipo TRect com as coordenadas superiores esquerdas e direito inferior para pintar. Deste modo, o código que nós usaremos em nosso componente será semelhante ao seguinteo (nós ainda não nos definimos a cor de preenchimento)

```
Canvas.FillRect(TRect);
```

Para escolher a cor de preenchimento nós usaremos a propriedade brush do canvas. A propriedade brush determina a cor padrão que o canvas usa para preencher formas gráficas e fundos. Em nosso caso, nós usaremos a propriedade color do brush para escolher a cor que preencherá o canvas.

Nós também precisaremos usar a propriedade pen do canvas. O objeto pen determina a classe de lápis que nós usaremos no canvas para puxar linhas e pontos. Em nosso caso, nós usaremos a propriedade estilo do pen que determina o tipo de linha para puxar (em nosso componente psSolid) e o modo de propriedade que determina o modo com que o lápis utilizará o canvas (em nosso componente pmCopy)

Criando o gradiente de cores.

No Delphi, uma cor é representada através de 4 bytes hexadecimais. O byte mais alto é usado para determinar o ajuste que o Windows faz da paleta e não veremos isto com mais detalhe. Os outros três bytes (os três bytes mais baixos) eles representam a quantidade de vermelho, verde e azul que forma a cor. Nosso componente devera calcular um gradiente de cores de um inicial (FColorDesde no código fonte) até outro fim (FColorHasta). O melhor modo para calcular este gradiente é decompondo as cores RGB (Vermelho, Verde e Azul) em seu componente. Deste modo, nós saberemos a quantidade de cada uma destas três cores básicas que irão formar uma certa cor. Por exemplo a cor vermelha pura é representada por 255,0,0 (255 de vermelho, 0 de verde e azul), e um tom cinza tem os três valores de vermelho, verde e mesmo azul (p.e. 150,150,150).

O decomposição de uma cor em seus três componentes básicos depende de três funções: *GetRValue, GetGValue e GetBValue que* pertencem ao API de Windows. Estas funções levam como parâmetro a cor da qual nós queremos obter o decomposição e eles devolvem a " quantidade " respectivamente de vermelho, verde e azul que compõe a cor.

No código fonte, uma vez decompostas as cores inicial e final nas variáveis RGBDesde[0 ..2] e RGBHasta[0 ..2] (0 para cor vermelha, 1 para verde e 2 para azul), o processo de calculo do gradiente é o seguinte:

1. **Calcular a diferença em valor absoluto entre RGBDesde e RGBHasta para cada uma das cores básicas e manter isto na variável RGBDif[0 ..2]. Adicionalmente, no fator variável nós manteremos +1 se RGBHasta for maior que RGBDesde (gradiente superior) e -1 caso contrário (gradiente descendente). Este processo é feito para cada uma das três cores**

básicas.

2. Entre 0 a 255 (nosso gradiente consistirá em 256 cores), nós calculamos a cor que corresponde a cada requadro a puxar (com o método FillRect) por meio da expressão:

```
Vermelho:=RGBDesde[0]+factor[0]*MulDiv(contador,RGBDif[0],255);
```

(de um modo semelhante para o verde e o azul)
Nota: MulDiv é uma função do Api de Windows que multiplica os primeiros dois valores passados como parâmetros e o resultado é dividido pelo terceiro parâmetro que devolve o valor desta divisão em forma de 16 bits arredondados.

3. Nós colocamos a cor calculada na propriedade color do objeto brush:

```
Canvas.Brush.Color:=RGB(Vermelho,Verde,Azul);
```

Como é fácil supor, a função RGB leva três valores de vermelho, verde e azul, e forma a cor que corresponde a estas cores básicas.

4. O requadro é tirado:

```
Canvas.FillRect(Banda);
```

Banda é um tipo de variável TRect que mantém as coordenadas do requadro para puxar.

Ultimo detalhe: como foi mencionado, nosso gradiente consistirá em 256 cores. Porém, dependendo do modo gráfico que nós configurarmos o Windows e da existência de outros objetos com as próprias cores , o Windows ajustará as cores disponíveis de forma que o resultado final sera a mais proxima possível.

Quando desenhar no Canvas. A mensagem WM_PAINT e o método Paint.

Quando nós queremos utilizar o canvas, nós devemos colocar o código correspondente dentro do método paint que pertence ao objeto TGraphicControl (o objeto TCustomControl também tem este método). Isto é deste modo porque quando nosso componente deveria ser puxado (p.e. o gradiente de cores), bem porque é a primeira vez em que ou em resposta para uma aplicação de Windows, o Windows enviará uma mensagem do tipo WM_PAINT para nosso componente. O componente, de um modo automático (herança), faz uma ligação ao método paint para desenhar o componente (em nosso caso, o gradiente).

Em geral nós não teremos a necessidade de investigar mais a mensagem WM_PAINT. Como nós há pouco explicamos, a única coisa que nos interessa dele (pelo menos para nosso componente), é que quando esta mensagem é recebida, o Delphi executará o método paint associado ao componente. E é dentro deste método que nós devemos adicionar o código necessário para puxar o gradiente.

Então, os passos que nós vimos anteriormente que são necessários puxar o gradiente, eles serão localizados dentro do método TGradiente.Paint que será declarado como nula.O método paint declarará isto no tipo protegido, desde que não se possa usá-lo fora de nosso componente.

Outros detalhes no desenvolvimento do componente. Publicando propriedades herdadas.

- **Um dos detalhes que nós veremos agora é o alinhamento do componente. O objeto TGraphicControl possui a propriedade Align (então nosso componente herdará isto), mas será declarado como público. Para nós é interessante declarar isto como published de forma que ela apareca no Object Inspector em design-time. Para isto nós devemos redeclarar a propriedade align na seção published:**

```
published
  property align
```

É importante fazer uma advertência em dois aspectos: o primeiro é que uma redeclaração pode serr só menos restritivo o acesso para uma propriedade (p.e protected a public), mas não mais restritivo (public a protected).
O segundo aspecto é ao redeclarar, não é necessário especificar o tipo da propriedade, apenas indicar seu nome. O Que nós podemos fazer no momento do redeclaração é definir um valor novo por padrão para esta propriedade.
- **O resto dos detalhes do codigo fonte não precisa de mais detalhes. São declarados as propriedades Direção, ColorDesde e ColorHasta e são escritos os valores correspondentes...**

O Código Fonte do Componente

```
unit Gradient;

interface

uses
  Classes, Controls, Graphics, WinTypes, WinProcs;

type
  TDireccion = (dHorizontal, dVertical);  {Tipo de dirección del gradiente}
  TGradiente = class(TGraphicControl)
  private
    FDireccion : TDireccion;   {Dirección del gradiente}
    FColorDesde, FColorHasta : TColor;     {Color del gradiente}
    procedure SetDireccion(valor : TDireccion);
    procedure SetColorDesde(valor : TColor);
    procedure SetColorHasta(valor : TColor);
  protected
    procedure Paint; override;
  public
    constructor Create(AOwner : TComponent); override;
  published
    property Direccion : TDireccion read FDireccion write SetDireccion default dHori
    property ColorDesde : TColor read FColorDesde write SetColorDesde default clBlue
    property ColorHasta : TColor read FColorHasta write SetColorHasta default clBlac
    property Align;     {Redeclaração da propriedade como publicada}
  end;

procedure Register;

implementation

constructor TGradiente.Create(AOwner : TComponent);
begin
  inherited Create(AOwner);        {Sempre a primeira coisa a fazer}
  FDireccion:=dHorizontal;         {Valores padrões}
```

```
  FColorDesde:=clBlue;
  FColorHasta:=clBlack;
  Width:=100;
  Height:=100;
end;

procedure TGradiente.SetDireccion(Valor : TDireccion);
begin
  if FDireccion <> valor then
  begin
    FDireccion := Valor;
    Repaint;              {Força o desenho do gradiente}
  end;
end;

procedure TGradiente.SetColorDesde(Valor : TColor);
begin
  if FColorDesde <> Valor then
  begin
    FColorDesde := Valor;
    Repaint;
  end;
end;

procedure TGradiente.SetColorHasta(Valor : TColor);
begin
  if FColorHasta <> Valor then
  begin
    FColorHasta := Valor;
    Repaint;
  end;
end;

procedure TGradiente.Paint;
var
  RGBDesde, RGBHasta, RGBDif : array[0..2] of byte;  {Cores inicial e final - difere
  contador, Vermelho, Verde, Azul : integer;
  Banda : TRect;                                     {Coordenadas do requadro a pint
  Factor : array[0..2] of shortint;                  {+1 se gradiente é crescente e
begin
  RGBDesde[0]:=GetRValue(ColorToRGB(FColorDesde));
  RGBDesde[1]:=GetGValue(ColorToRGB(FColorDesde));
  RGBDesde[2]:=GetBValue(ColorToRGB(FColorDesde));

  RGBHasta[0]:=GetRValue(ColorToRGB(FColorHasta));
  RGBHasta[1]:=GetGValue(ColorToRGB(FColorHasta));
  RGBHasta[2]:=GetBValue(ColorToRGB(FColorHasta));
  for contador:=0 to 2 do          {Calculo de RGBDif[] e factor[]}
  begin
    RGBDif[contador]:=Abs(RGBHasta[contador]-RGBDesde[contador]);
    If RGBHasta[contador]>RGBDesde[contador] then factor[contador]:=1 else factor[co
  end;
  Canvas.Pen.Style:=psSolid;
  Canvas.Pen.Mode:=pmCopy;
  if FDireccion = dHorizontal then
  begin
    Banda.Left:=0;
    Banda.Right:=Width;
    for contador:=0 to 255 do
    begin
```

```
      Banda.Top:=MulDiv(contador,height,256);
      Banda.Bottom:=MulDIv(contador+1,height,256);
      Vermelho:=RGBDesde[0]+factor[0]*MulDiv(contador,RGBDif[0],255);
      Verde:=RGBDesde[1]+factor[1]*MulDiv(contador,RGBDif[1],255);
      Azul:=RGBDesde[2]+factor[2]*MulDiv(contador,RGBDif[2],255);
      Canvas.Brush.Color:=RGB(Vermelho,Verde,Azul);
      Canvas.FillRect(Banda);
    end;
  end;
  if FDireccion = dVertical then
  begin
    Banda.Top:=0;
    Banda.Bottom:=Height;
    for contador:=0 to 255 do
    begin
      Banda.Left:=MulDiv(contador,width,256);
      Banda.Right:=MulDIv(contador+1,width,256);
      Vermelho:=RGBDesde[0]+factor[0]*MulDiv(contador,RGBDif[0],255);
      Verde:=RGBDesde[1]+factor[1]*MulDiv(contador,RGBDif[1],255);
      Azul:=RGBDesde[2]+factor[2]*MulDiv(contador,RGBDif[2],255);
      Canvas.Brush.Color:=RGB(Vermelho,Verde,Azul);
      Canvas.FillRect(Banda);
    end;
  end;
end;

procedure Register;
begin
  RegisterComponents('Curso',[TGradiente]);
end;

end.
```

Luis Roche revueltaroche@redestb.es

Ultima modificación 20.12.1996