LA BIBLIOTHÈQUE BGRABITMAP

Objectifs : dans ce chapitre, vous apprendrez à dessiner avec la bibliothèque BGRABitmap.

Sommaire : Introduction – Création et sauvegarde d'une image – Méthodes de dessin des objets Canvas et CanvasBGRA – Méthodes de dessin propres à la classe TBGRABitmap – Méthodes pour afficher du texte – Opérations sur une image préexistante – Animation utilisant un écran virtuel

Ressources : les programmes de test sont présents dans le sous-répertoire *BGRABitmap* du répertoire *exemples*.

INTRODUCTION

La bibliothèque BGRABitmap est un ensemble d'outils pour le dessin, l'écriture, le traitement des images, etc. Les possibilités offertes par cette bibliothèque sont infinies. Pour s'en rendre compte, il suffit de faire un essai du logiciel libre LazPaint, qui est basé sur cette bibliothèque et développé par son auteur.

Les principaux avantages de la bibliothèque BGRABitmap, comparée à la bibliothèque standard de Lazarus, sont la gestion de la transparence et le dessin avec anti-crénelage (en anglais *anti-aliasing*).

INSTALLATION

La bibliothèque BGRABitmap n'est pas livrée avec Lazarus. Avant de pouvoir l'utiliser, il faut la télécharger et l'installer. Pour télécharger les sources de la bibliothèque, rendez-vous sur le site du logiciel LazPaint :

http://sourceforge.net/projects/lazpaint/files/src/

Une fois la bibliothèque téléchargée, il faut décompresser l'archive ZIP et placer les sources de la bibliothèque à un endroit soigneusement choisi.

Le chemin de ce dossier devra être indiqué au compilateur, par exemple en ajoutant ce chemin dans les options de votre projet Lazarus (« Projet », « Options du projet », « Chemins », « Autres fichiers unités »).

L'autre possibilité est d'ouvrir l'un des paquets inclus dans les sources de la bibliothèque. Dans un premier temps, vous n'aurez besoin que du paquet bgrabitmappack.lpk. Une fois ce paquet ouvert, ajoutez-le au projet (« Utiliser », « Ajouter au projet »).

Une fois que vous avez indiqué au compilateur, d'une façon ou d'une autre, le chemin des sources de la bibliothèque, ajoutez l'unité *BGRABitmap* dans la liste des unités utilisées par le programme, puis faites un premier essai de compilation.

CRÉATION ET SAUVEGARDE D'UNE IMAGE

Maintenant que vous avez installé la bibliothèque BGRABitmap et que vous avez fait un premier essai de compilation couronné de succès, vous pouvez commencer à explorer les possiblités offertes par cette bibliothèque.

Vous allez donc faire connaissance avec la classe TBGRABitmap. La classe TBGRABitmap permet de créer des images, de dessiner. Vous allez commencer par utiliser la classe TBGRABitmap dans de simples programmes qui créeront des images en mémoire et les enregistreront dans des fichiers.

Le plus simple programme utilisant la classe TBGRABitmap est un programme qui crée une image, avec ses dimensions et sa couleur par défaut, la sauvegarde dans un fichier, et libère la mémoire allouée à l'image.

En plus de l'unité *BGRABitmap,* le programme utilise l'unité *BGRABitmapTypes,* qui contient des définitions nécessaires la plupart du temps. Dans l'exemple ci-dessous, l'unité *BGRABitmapTypes* fournit la définition de la constante BGRAWhite.

```
program bgrabitmap_001;


uses
  SysUtils, BGRABitmap, BGRABitmapTypes;


const
  LARGEUR = 100;
  HAUTEUR = 100;


var
  image: TBGRABitmap;


begin
  { Création d'un carré blanc de cent fois cent pixels. }
  image := TBGRABitmap.Create(LARGEUR, HAUTEUR, BGRABitmapTypes.BGRAWhite);


  { Sauvegarde de l'image dans un fichier. }
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));
```

```
  { Libération de la mémoire allouée à l'image. }
  image.Free;
end.
```

Si vous ne spécifiez pas de couleur lors de l'appel du constructeur, la couleur par défaut de l'image sera la couleur transparente (si l'on peut dire).

```
  { Création d'un carré transparent de cent fois cent pixels. }
  image := TBGRABitmap.Create(LARGEUR, HAUTEUR);
```

Vous pouvez choisir explicitement une image transparente en passant la valeur BGRAPixelTransparent au constructeur.

```
  { Création d'un carré transparent de cent fois cent pixels. }
  image := TBGRABitmap.Create(LARGEUR, HAUTEUR, BGRAPixelTransparent);
```

Quant au fichier dans lequel l'image est sauvegardée, ce peut être un fichier de type BMP, JPG ou PNG. Cela dépend tout simplement du nom de fichier que vous passez à la méthode SaveToFile().

```
  { Sauvegarde de l'image dans un fichier. }
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.bmp'));
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.jpg'));
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));
```

Cependant le type PNG est généralement préféré, parce qu'il permet d'exploiter pleinement l'un des atouts de la bibliothèque BGRABitmap, à savoir la transparence. En effet BGRA est l'acronyme de « blue, green, red, alpha », où alpha est un octet représentant le degré de transparence du pixel.

Vous avez donc appris à créer une image, à la sauvegarder dans un fichier et à libérer la mémoire allouée à l'image. Vous pouvez maintenant commencer à dessiner. Cependant la bibliothèque BGRABitmap met à votre disposition une telle variété d'outils que vous ne savez pas trop lequel choisir pour commencer.

Vous allez commencer par le plus facile et même, comme vous allez le voir, par le plus familier. Vous allez commencer par utiliser les méthodes de l'objet TCanvas.

MÉTHODES DE DESSIN DES OBJETS CANVAS ET CANVASBGRA

Car l'auteur de la bibliothèque BGRABitmap a eu la lumineuse idée d'équiper la classe TBGRABitmap d'un objet Canvas identique extérieurement à l'objet homonyme de la classe TForm de la bibliothèque standard de Lazarus.

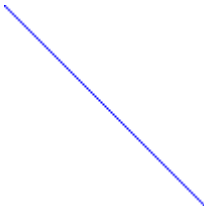
Par conséquent, pour peu que vous soyez familier de l'objet Canvas de la LCL (la bibliothèque de composants visuels de Lazarus, qui est une réécriture de la VCL de Delphi), vous savez d'ores et déjà dessiner avec la classe TBGRABitmap !

Voici par exemple un programme qui trace une ligne à travers le carré blanc que vous avez fabriqué out à l'heure :

```
program bgrabitmap_005;


uses
  SysUtils, BGRABitmap, BGRABitmapTypes, BGRAGraphics;


procedure Operations(image: TBGRABitmap);
begin
  image.Canvas.Pen.Color := BGRAGraphics.clBlue;
  image.Canvas.MoveTo(0, 0);
  image.Canvas.LineTo(image.Width, image.Height);
end;


const
  LARGEUR = 100;
  HAUTEUR = 100;


var
```

```
  image: TBGRABitmap;


begin
  image := TBGRABitmap.Create(LARGEUR, HAUTEUR, BGRAWhite);
  Operations(image);
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));
  image.Free;
end.
```

Et voici le résultat :



L'objet Pen, les méthode MoveTo() et LineTo() s'utilisent exactement comme leurs homonymes de la LCL. Quant à l'image obtenue, elle est exactement semblable à celle qu'on aurait obtenu en dessinant dans l'objet Canvas d'une fenêtre standard.

C'est à la fois le point fort et le point faible de l'objet Canvas : d'être parfaitement compatible avec l'objet Canvas standard, et donc d'offrir des procédures de dessin basiques, c'est-à-dire notamment sans transparence ni anti-crénelage.

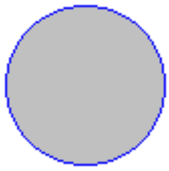
Heureusement l'auteur de la bibliothèque BGRABitmap a eu une autre lumineuse idée : celle d'équiper la classe TBGRABitmap d'un objet Canvas, de type TCanvasBGRA, qui lui aussi reprend la syntaxe de l'objet Canvas standard, mais fournit un résultat plus sophistiqué, en faisant appel de façon cachée aux fonctions avancées de la bibliothèque BGRABitmap.

Voici un second exemple de dessin dans l'objet Canvas de type TCanvas :

```
procedure Operations(image: TBGRABitmap);

begin

  with image.Canvas do
```

```
  begin
    Pen.Color := BGRAGraphics.clBlue;
    Brush.Color := clSilver;
    Ellipse(10, 10, 90, 90);
  end;
end;
```

Le résultat est une ellipse basique (crénelée) :



Remplacez dans le code « Canvas » par « CanvasBGRA ».

```
procedure Operations(image: TBGRABitmap);
begin
  with image.CanvasBGRA do
```

Le reste du code ne change pas. En revanche, le résultat obtenu n'est plus le même :

Un lissage (ou *anti-aliasing*) a été appliqué à l'ellipse.

MÉTHODES DE DESSIN PROPRES À LA CLASSE TBGRABITMAP

Vous avez appris à utiliser la bibliothèque BGRABitmap avec la syntaxe héritée de l'objet Canvas standard.

Vous allez maintenant faire connaissance avec le style de programmation propre à la classe TBGRABitmap. Les fonctions que vous allez découvrir ont une syntaxe un peu plus complexe que les précédentes, parce qu'elles offrent des possibilités supérieures.

Voici, pour commencer, comment tracer une ligne :

```
procedure Operations(image: TBGRABitmap);

const

  EPAISSEUR = 10;

var

  couleur: TBGRAPixel;

begin

  couleur := BGRA(0, 0, 255, 255);

  image.DrawLineAntialias(10, 10, 90, 90, couleur, EPAISSEUR);

end;


const

  LARGEUR = 100;

  HAUTEUR = 100;


var

  image: TBGRABitmap;


begin

  image := TBGRABitmap.Create(LARGEUR, HAUTEUR, BGRAWhite);

  Operations(image);

  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));

  image.Free;

end.
```

Et voici le résultat :

Comme vous pouvez le constater, la méthode DrawLineAntiAlias permet de choisir l'épaisseur de la ligne et aussi son degré d'opacité. En outre, comme le nom de la méthode l'indique, et comme on le voit dans l'image ci-dessus, la ligne est lisse et non pas crénelée comme les lignes de l'objet Canvas standard.

Après la ligne, le rectangle :

```
procedure Operations(image: TBGRABitmap);

const

  EPAISSEUR = 10;

var

  couleur: TBGRAPixel;

begin

  couleur := BGRA(0, 0, 255); { Par défaut, le quatrième paramètre vaut 255. }

  image.RectangleAntialias(10, 10, 90, 90, couleur, EPAISSEUR);

end;
```

Pour un rectangle plein, vous utiliserez la méthode FillRectAntiAlias :

```
procedure Operations(image: TBGRABitmap);
var
  couleur: TBGRAPixel;
begin
  couleur := BGRA(0, 0, 255);
  image.FillRectAntialias(10, 10, 90, 90, couleur);
end;
```

Pour un rectangle arrondi, vous utiliserez la méthode RoundRectAntiAlias :

```
procedure Operations(image: TBGRABitmap);
const
  EPAISSEUR = 10;
begin
  image.RoundRectAntialias(
    10, { Abscisse du sommet supérieur gauche. }
    10, { Ordonnée du sommet supérieur gauche. }
    90, { Abscisse du sommet inférieur droit.  }
    90, { Ordonnée du sommet inférieur droit.  }
    20, { Rayon horizontal de l'arrondi.       }
    20, { Rayon vertical de l'arrondi.         }
    BGRA(0, 0, 255),
    EPAISSEUR,
```

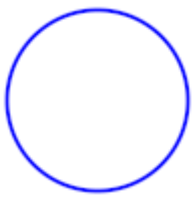
```
    []
  );
end;
```

Par défaut, les quatre coins du rectangle sont arrondis. Le dernier paramètre de la méthode permet de modifier ce comportement. Voici un exemple d'un rectangle plein, dont seuls les coins supérieur gauche et inférieur droit sont arrondis :

```
procedure Operations(image: TBGRABitmap);
var
  couleur: TBGRAPixel;
begin
  couleur := BGRA(0, 0, 255);

  image.FillRoundRectAntialias(
    10,
    10,
    90,
    90,
    20,
    20,
    couleur,
    [rrTopRightSquare, rrBottomLeftSquare]
  );
end;
```

Pour tracer une ellipse (et donc aussi un cercle), vous utiliserez la méthode EllipseAntiAlias :

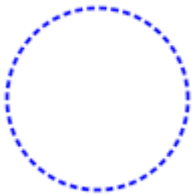
```
procedure Operations(image: TBGRABitmap);

const

  EPAISSEUR = 2;

begin

  image.Fill(BGRAWhite);


  image.EllipseAntialias(

    image.Width / 2,       { Abscisse du centre. }

    image.Height / 2,      { Ordonnée du centre. }

    image.Width / 2 - 5,   { Rayon horizontal.   }

    image.Height / 2 - 5,  { Rayon vertical.     }

    CSSBlue,               { Couleur.            }

    EPAISSEUR

  );

end;
```



Remarquez qu'au lieu de choisir la couleur de fond de l'image lors de l'appel du constructeur Create (comme le faisaient les exemples précédents), le programme ci-dessus utilise la méthode Fill.

La propriété CustomPenStyle permet de modifier le style du trait, par exemple pour dessiner en pontillés :

```
  { Style de la ligne. }
  image.CustomPenStyle := BGRAPenStyle(2, 1);
```



Pour dessiner une ellipse pleine (et donc aussi un disque), vous utiliserez la méthode FillEllipseAntiAlias :

```
procedure Operations(image: TBGRABitmap);
var
  couleur: TBGRAPixel;
begin
  couleur := BGRA(0, 0, 255);

  image.FillEllipseAntialias(
    49.5,
    49.5,
    49,
    49,
    couleur
  );
end;
```

Enfin, pour une ellipse pleine et coloriée de façon dégradée, vous utiliserez la méthode FillEllipseLinearColorAntialias :

```
procedure Operations(image: TBGRABitmap);

begin

  image.FillEllipseLinearColorAntialias(

    image.Width / 2,

    image.Height / 2,

    image.Width / 2 - 5,

    image.Height / 2 - 5,

    BGRA(0, 0, 255),     { Couleur de la périphérie. }

    BGRAPixelTransparent { Couleur du centre.        }

  );

end;
```

Vous voici bien loin de l'objet Canvas standard, n'est-ce pas ? Et encore ces quelques exemples ne donnent-ils qu'un premier aperçu des possibilités offertes par la bibliothèque BGRABitmap.

MÉTHODES POUR AFFICHER DU TEXTE

Vous allez à présent vous familiariser avec les fonctions d'affichage de texte.

Pour cette série d'exemples, vous allez créer, non plus une simple application console, mais une application graphique qui permettra de voir les images créées dans une fenêtre. Voici le modèle sur lesquels les exemples suivants seront bâtis :

```
unit bgrabitmap_018_u;


{$mode objfpc}{$H+}


interface


uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs,
  BGRABitmap, BGRABitmapTypes;


type


  { TForm1 }


  TForm1 = class(TForm)
    procedure FormPaint(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;


var
  Form1: TForm1;


implementation


{$R *.lfm}
```

```
{ TForm1 }


procedure TForm1.FormPaint(Sender: TObject);
var
  image: TBGRABitmap;
begin
  image := TBGRABitmap.Create(100, 100, BGRAWhite);


  { Dessin de l'image dans la fenêtre }
  image.Draw(
    Canvas, { TForm1.Canvas                                    }
    0,      { abscisse du sommet supérieur gauche de l'image }
    0,      { ordonnée du sommet supérieur gauche            }
    TRUE    { dessin opaque                                  }
  );


  { Sauvegarde de l'image dans un fichier }
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));


  image.Free;
end;
```

Pour la commodité du lecteur, on a rassemblé dans la méthode FormPaint tout le code relatif à l'image (création, affichage, libération), sauf les opérations de dessin que vous mettrez à part comme précédemment.

Voici un premier exemple d'affichage de texte :

```
procedure Operations(image: TBGRABitmap);
begin
  with image do
  begin
    FontHeight := 24;
    FontAntialias := TRUE;
    FontStyle := [fsBold]; { fsItalic, fsStrikeOut, fsUnderline }
```

```
    TextOut(5, 5, 'Hello', CSSBlue);
  end;
end;
```

Et le résultat obtenu :

La méthode TextOut admet un cinquième paramètre facultatif permettant de choisir l'alignement du texte, par exemple l'alignement à gauche :

```
    TextOut(5, 5, 'Hello', CSSBlue, taLeftJustify);
```

Ou au centre :

```
    TextOut(Width / 2, 5, 'Hello', CSSBlue, taCenter);
```

Pour centrer l'image verticalement, il y a plusieurs façons de faire, par exemple celle-ci, qui exploite la propriété FontFullHeight :

```
procedure Operations(image: TBGRABitmap);
begin
  with image do
  begin
    FontHeight := 24;
    FontAntialias := TRUE;
    FontStyle := [fsBold];
    TextOut(Width / 2, (Height - FontFullHeight) / 2, 'Hello', CSSBlue, taCenter);
```

```
  end;
end;
```

Si vous désirez en savoir plus dès à présent sur les différentes méthodes permettant de centrer horizontalement et verticalement un texte, consultez le code source des exemples 23 à 26.

Pour choisir une autre police que la police par défaut, vous utiliserez la propriété FontName :

```
procedure Operations(image: TBGRABitmap);
begin
  with image do
  begin
    FontHeight := 24;
    FontName := 'Courier New';
```

Pour donner une inclinaison au texte, vous utiliserez la méthode TextOutAngle :

```
procedure Operations(image: TBGRABitmap);
begin
  with image do
  begin
    FontHeight := 24;
    FontAntialias := TRUE;
    FontStyle := [fsBold];

    TextOutAngle(
      25,
      5,
      -450, { angle en dixièmes de degré }
      'Hello',
      CSSBlue,
      taLeftJustify
    );
  end;
end;
```

Ou bien vous changerez la valeur de la propriété FontOrientation :

```
procedure Operations(image: TBGRABitmap);
begin
  with image do
  begin
    FontHeight := 24;
    FontAntialias := TRUE;
    FontStyle := [fsBold];

    FontOrientation := -450;

    TextOut(25, 5, 'Hello', CSSBlue, taLeftJustify);
  end;
end;
```

La méthode TextRect permet d'inscrire le texte dans un rectangle :

```
    image.TextRect(
      Rect(5, 5, 95, 95),
      'Texte avec retour automatique à la ligne',
      taCenter, { taLeftJustify, taRightJustify }
      tlCenter, { tlTop, tlBottom               }
      CSSBlue
    );
```

La méthode TextShadow permet d'obtenir un effet d'ombre :

```
procedure Operations(image: TBGRABitmap);

var

  texte: TBGRABitmap;

begin

  texte := TextShadow(

    100,       { Largeur de l'image.  }

    100,       { Hauteur de l'image.  }

    'Hello',   { Texte.               }

    24,        { Taille de la police. }

    CSSBlue,   { Couleur du texte.    }

    BGRABlack, { Couleur de l'ombre.  }

    5,         { Décalage horizontal. }

    5,         { Décalage vertical.   }

    5          { Rayon de l'ombre.    }

  );


  image.PutImage(0, 0, texte, dmDrawWithTransparency);

  texte.Free;

end;
```

Voilà un premier aperçu des méthodes d'affichage de texte. Si vous souhaitez donner de l'épaisseur à vos lettres et obtenir un effet d'éclairage, avec ombre et reflets, consultez l'exemple 32.

OPÉRATIONS SUR UNE IMAGE PRÉEXISTANTE

Jusqu'ici vous avez dessiné et écrit dans un carré blanc ou transparent. Vous allez maintenant effectuer diverses opérations sur une image préexistante, à savoir la photographie d'un tableau. Il s'agit d'un tableau de J.B. Camille Corot intitulé « L' Étang de Ville-d'Avray ». Ce tableau est conservé au musée des Beaux-Arts de Strasbourg :

http://www.musees.strasbourg.eu/index.php?page=XIXe-siecle-francais

Pour créer une image de type TBGRABitmap à partir du fichier corot.jpg, le plus simple est de passer le nom du fichier, comme unique paramètre, au constructeur de la classe TBGRABitmap :

```
program bgrabitmap_033;


uses
  SysUtils, BGRABitmap, BGRABitmapTypes;


const
  FICHIER = 'images\corot.jpg';


var
  image: TBGRABitmap;


begin
  { Création d'une image à partir d'un fichier. }
  image := TBGRABitmap.Create(FICHIER);


  { Sauvegarde de l'image dans un fichier. }
  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));


  { Libération de la mémoire allouée à l'image. }
  image.Free;
end.
```

Le fichier de type PNG produit par le programme est la réplique exacte de l'imae originale.

La bibliothèque BGRABitmap s'est occupée pour vous des dimensions de l'image, et la conversion du type JPG vers le type PNG s'est faite automatiquement à partir du nom de fichier passé à la méthode SaveToFile.

Vous auriez pu procéder autrement, à savoir créer une image vierge à partir des dimensions de l'image JPG, puis appeler la méthode LoadFromFile :

```
procedure Operations(image: TBGRABitmap);

const

  FICHIER = 'images\corot.jpg';

begin

  image.LoadFromFile(FICHIER);

end;


var

  image: TBGRABitmap;


begin
```

```
  image := TBGRABitmap.Create(800, 571);
```

Maintenant que, d'une façon ou d'une autre, vous savez charger une image depuis un fichier, vous allez effectuer différentes manipulations sur cette image, et d'abord en découper un morceau. Pour ce faire vous utiliserez la fonction GetPart(), qui reçoit un paramètre de type TRect :

```
procedure Operations(image: TBGRABitmap);

var

  partie: TBGRABitmap;

begin

  partie := image.GetPart(Rect(300, 400, 400, 500)) as TBGRABitmap;

  partie.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));

  partie.Free;

end;
```

Ce qui donne le résultat suivant :

Soit dit en passant, le même résultat peut être obtenu avec un code plus court, au moyen de la très commode fonction BGRAReplace() :

```
const

  FICHIER = 'images\corot.jpg';


var

  image: TBGRABitmap;


begin

  image := TBGRABitmap.Create(FICHIER);


  BGRAReplace(

    image,
```

```
    image.GetPart(Rect(300, 400, 400, 500))
  );


  image.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));
  image.Free;
end.
```

À présent vous allez faire une image en noir et blanc, au moyen de la méthode FilterGrayScale() :

```
procedure Operations(image: TBGRABitmap);
var
  noir_et_blanc: TBGRABitmap;
begin
  noir_et_blanc := image.FilterGrayScale as TBGRABitmap;
  noir_et_blanc.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));
  noir_et_blanc.Free;
end;


const
  FICHIER = 'images\corot.jpg';


var
  image: TBGRABitmap;


begin
  image := TBGRABitmap.Create(FICHIER);
  Operations(image);
  image.Free;
end.
```

À vrai dire, « noir et blanc » n'est pas très une formule très exacte, car il s'agit plutôt de différents tons de gris, comme l'indique le nom de la méthode.

À présent vous allez redimensionner l'image originale, au moyen de la méthode Resample() :

```
procedure Operations(image: TBGRABitmap);

var

  reduction: TBGRABitmap;

begin

  reduction := image.Resample(

    350,

    250,

    rmFineResample { rmSimpleStretch }

  ) as TBGRABitmap;


  reduction.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));

  reduction.Free;

end;
```

L'option rmFineResample donne un résultat de meilleure qualité ; l'option rmSimpleStretch donne un résultat plus rapide.

Si vous souhaitez apprendre comment accéder directement aux pixels de l'image, reportez-vous directement au code source de l'exemple 40.

Si vous souhaiter apprendre comment utiliser un masque afin d'obtenir une image inscrite dans une forme autre qu'un rectangle, par exemple un ovale, reportez-vous à l'exemple 41.

ANIMATION UTILISANT UN ÉCRAN VIRTUEL

Pour terminer, vous allez apprendre à réaliser une animation basée sur les fonctions de la bibliothèque BGRABitmap. On entend par animation une image qui change à intervalles de temps réguliers et qui produit l'illusion d'un mouvement.

Vous allez donc réaliser une application graphique. Cette application permettra à l'utilisateur de déplacer une dame sur une ligne d'échiquier, au moyen des flèches gauche et droite du clavier.

L'image de la dame, en se déplaçant, ne devra pas effacer les rayures des cases noires, sauf bien entendu pour la partie que le corps de la dame couvrira. Les parties couvertes devront être restaurées après le passage de la dame. Cela sera fait de la façon la plus économique, c'est-à-dire en restaurant seulement la partie de l'image concernée, et non pas tout l'échiquier.

Toutes ces opérations se feront dans un écran virtuel, une image de type TBGRABitmap. Une fois ces opérations terminées, le contenu de l'écran virtuel sera copié dans l'objet Canvas de la fenêtre.

Voici le code complet de l'unité :

```
unit bgrabitmap_042_u;
```

```
{$MODE objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, ExtCtrls,
LMessages, LCLType,
  BGRABitmap, BGRABitmapTypes;

type
  { TForm1 }
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
    procedure FormPaint(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure WMEraseBkgnd(var message: TLMEraseBkgnd); message LM_ERASEBKGND;
  private
    { private declarations }
    ecran_virtuel,           { Écran virtuel.                                  }
    dame,                    { Image de la dame.                               }
    sauvegarde: TBGRABitmap; { Image pour la restauration de l'arrière-plan. }
    fX, fY: integer;         { Position de la dame.                            }
  public
    { public declarations }
    procedure DessinDame(const aX, aY: integer);
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.lfm}

{ TForm1 }

const
  STEP = 5;

procedure TForm1.FormCreate(Sender: TObject);
var
  fond: TBGRABitmap;
begin
  { Création de l'écran virtuel. }
  ecran_virtuel := TBGRABitmap.Create(320, 40);

  { Création de l'image de la dame à partir du fichier. }
  dame := TBGRABitmap.Create('images\dame.bmp');

  { Remplacement du bleu par de la "couleur transparente". }
  dame.ReplaceColor(CSSMidnightBlue, BGRAPixelTransparent);

  { Création de l'image de fond à partir du fichier. }
  fond := TBGRABitmap.Create('images\ligne.bmp');

  { Dessin du fond sur l'écran virtuel. }
  ecran_virtuel.PutImage(0, 0, fond, dmSet);

  { Libération de l'image de fond. }
  fond.Free;

  { Dessin de la dame à sa position initiale. }
  DessinDame(0, 0);
end;
```

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
  ecran_virtuel.SaveToFile(ChangeFileExt(ParamStr(0), '.png'));
  ecran_virtuel.Free;
  dame.Free;
  sauvegarde.Free;
end;

procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  case Key of
    VK_LEFT : if fX >= 000 + STEP then DessinDame(fX - STEP, fY);
    VK_RIGHT: if fX <= 280 - STEP then DessinDame(fX + STEP, fY);
  end;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin
  { Dessin de l'écran virtuel dans la fenêtre }
  ecran_virtuel.Draw(Canvas, 0, 0);
end;

procedure TForm1.FormShow(Sender: TObject);
begin
  BorderStyle := bsSingle;
  ClientWidth := ecran_virtuel.Width;
  ClientHeight := ecran_virtuel.Height;
end;

procedure TForm1.WMEraseBkgnd(var message: TLMEraseBkgnd);
begin
  { Procédure créée et laissée vide à dessein, pour que le fond de la fenêtre ne
soit pas effacé }
```

```
end;


procedure TForm1.DessinDame(const aX, aY: integer);
begin
  { Restauration du fond sauf au premier appel de la procédure }
  if Assigned(sauvegarde) then
    ecran_virtuel.PutImage(fX, fY, sauvegarde, dmSet);
  { Mise à jour de l'image de sauvegarde en vue de la prochaine restauration }
  BGRAReplace(sauvegarde, ecran_virtuel.GetPart(Rect(aX, aY, aX + 40, aY + 40)));
  { Dessin de la dame à sa nouvelle position }
  ecran_virtuel.PutImage(aX, aY, dame, dmDrawWithTransparency);
  { Enregistrement de la position }
  fX := aX;
  fY := aY;
  { Rafraîchissement de la fenêtre }
  Invalidate;
end;


end.
```

Arrêtons-nous sur quelques endroits intéressants de ce code. L'image de la dame provient du fichier dame.bmp, dont la couleur est un certain bleu arbitrairement choisi dont on passe la valeur à la méthode ReplaceColor() :

```
  { Création de l'image de la dame à partir du fichier. }
  dame := TBGRABitmap.Create('images\dame.bmp');


  { Remplacement du bleu par de la "couleur transparente". }
  dame.ReplaceColor(CSSMidnightBlue, BGRAPixelTransparent);
```

Les opérations sur l'écran virtuel se font dans la procédure DessinDame() :

```
procedure TForm1.DessinDame(const aX, aY: integer);
```

```
begin
  { Restauration du fond sauf au premier appel de la procédure }
  if Assigned(sauvegarde) then
    ecran_virtuel.PutImage(fX, fY, sauvegarde, dmSet);
  { Mise à jour de l'image de sauvegarde en vue de la prochaine restauration }
  BGRAReplace(sauvegarde, ecran_virtuel.GetPart(Rect(aX, aY, aX + 40, aY + 40)));
  { Dessin de la dame à sa nouvelle position }
  ecran_virtuel.PutImage(aX, aY, dame, dmDrawWithTransparency);
  { Enregistrement de la position }
  fX := aX;
  fY := aY;
  { Rafraîchissement de la fenêtre }
  Invalidate;
end;
```

Le transfert de l'écran virtuel vers la fenêtre se fait dans la méthode FormPaint() :

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  { Dessin de l'écran virtuel dans la fenêtre }
  ecran_virtuel.Draw(Canvas, 0, 0);
end;
```

La dame glisse vers la gauche ou vers la droite, en fonction des touches sur lesquelles l'utilisateur appuie.

Les rayures à l'arrière-plan sont préservées.

Voilà ! Vous avez fait connaissance avec la bibliothèque BGRABitmap.