FMX, utilisation du composant TPath

Exploiter des SVG

Par Serge Girard

Date de publication : 19 juin 2019

Hors de mes sentiers habituels, je veux vous faire part de comment j'ai découvert le composant **TPath**, des « triturations » que j'ai pu faire sur ce composant et quelles destinations j'envisage pour celui-ci.

Tout est parti d'un constat lorsque j'ai voulu ajouter un simple triangle monochrome à un composant et de la possibilité d'en changer couleur et orientation. Ma première solution fut d'ajouter un **TImageList** mais une image pour chaque couleur, chaque orientation cela devenait « lourd » et vite ingérable sans parler du fait qu'il fallait, en plus, penser multi-résolution ! J'avais déjà utilisé **TPath** au cours de mon **tutoriel sur les grilles FMX** mais, noyé dans la masse, il était certainement passé inaperçu.

Ce que je vous propose ici est une sorte de cahier de développement autour de ce composant. Mes tests ont été écrits avec la version Delphi 10.3.

I - Introduction

J'avoue, ce n'est pas tout à fait la première fois que j'utilise des composants **TPath** mais, jusque-là, je ne m'étais contenté, la plupart du temps, que de récupérer des fichiers tout prêts, en particulier sur le site **Material Design Icons** . Le plus souvent pour mes images de boutons je me contentais même de charger directement des icônes gratuites sur **le site icones8.fr**.

*Quel rapport entre **TPath** et des icônes ? Patience vous le découvrirez bientôt.*

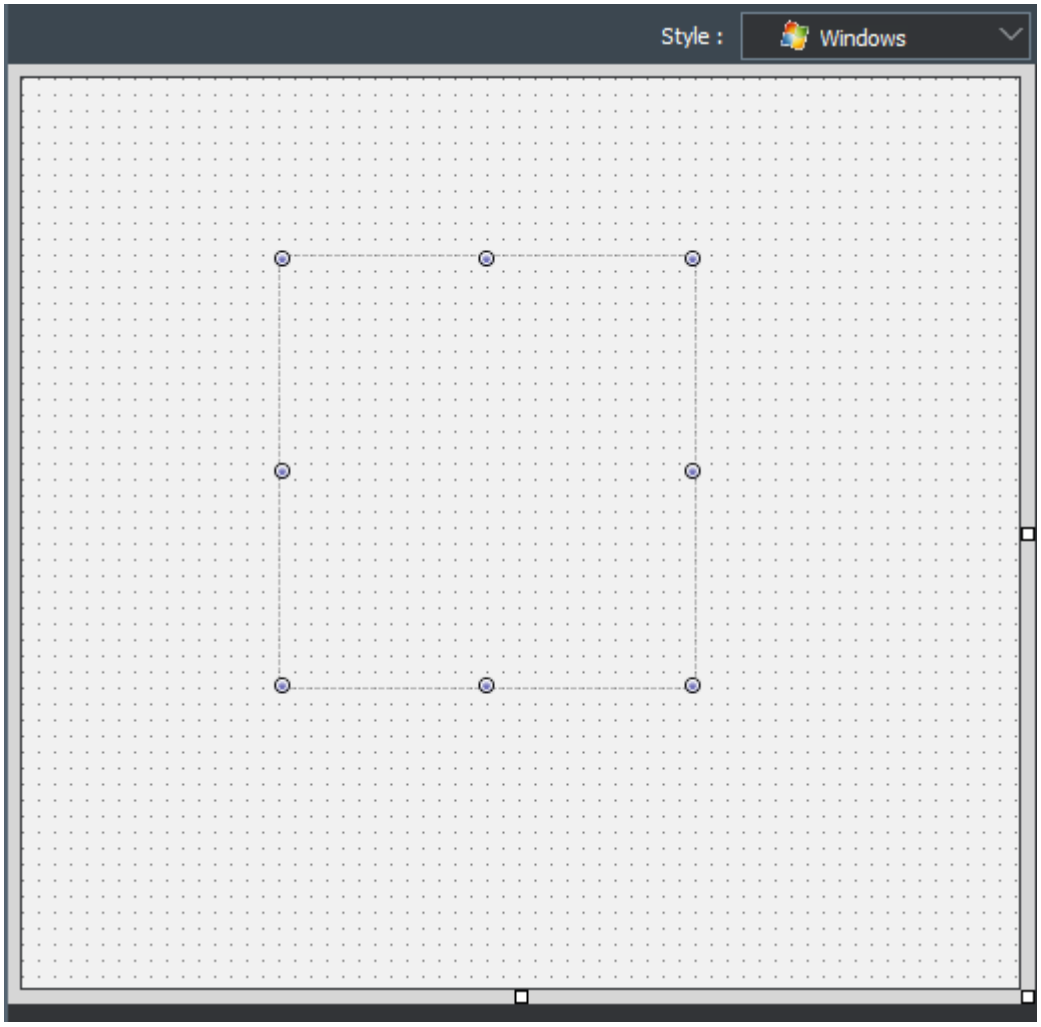
I-A - Qu'est-ce ?

La définition, « **TPath** définit des formes de type chemin 2D représentant des groupes de courbes et de lignes connectées » est peu explicite au profane. Je décrirai plutôt cela comme un dessin 2D composé de lignes et de courbes dont les spécifications sont inscrites dans la propriété **Data** de type **TPathData.** Ces spécifications suivent une norme précise, celle du SVG (Scalable Vector Graphis) 1.0 et encore, malheureusement, s'agit-il uniquement de l'attribut **d** de l'élément **path**, soit une liste d'instructions *moveto (M)*, *line (L, l)*, *curve (C, c)*, *arc* (A, a) et *closepath (Z).*

Décrit ainsi c'est pas très parlant ! Le mieux est encore de faire une petite démonstration rapide.

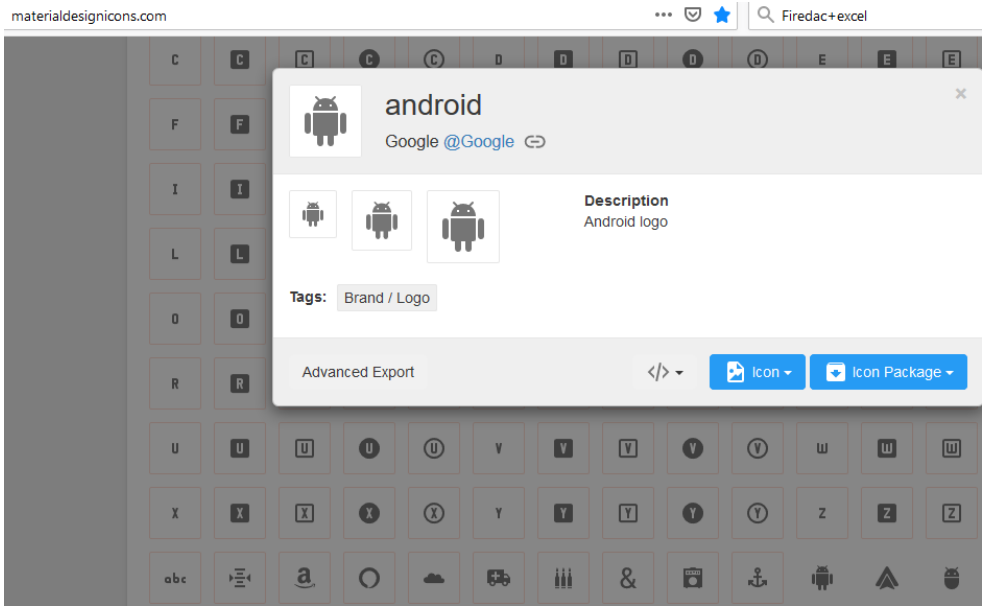
Étapes :

1. Sur une forme FMX vierge poser un **TPath**.



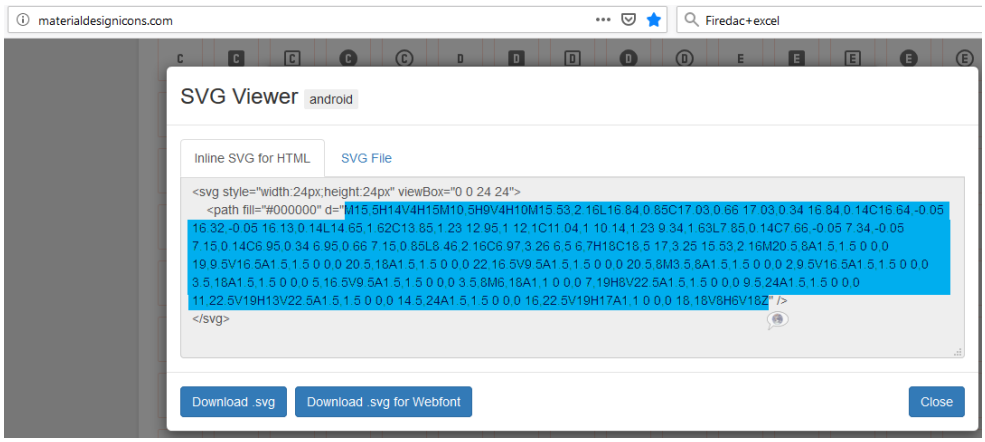
2. Aller sur le site **Material Design Icons**.

3. Sélectionner le dessin de son choix.

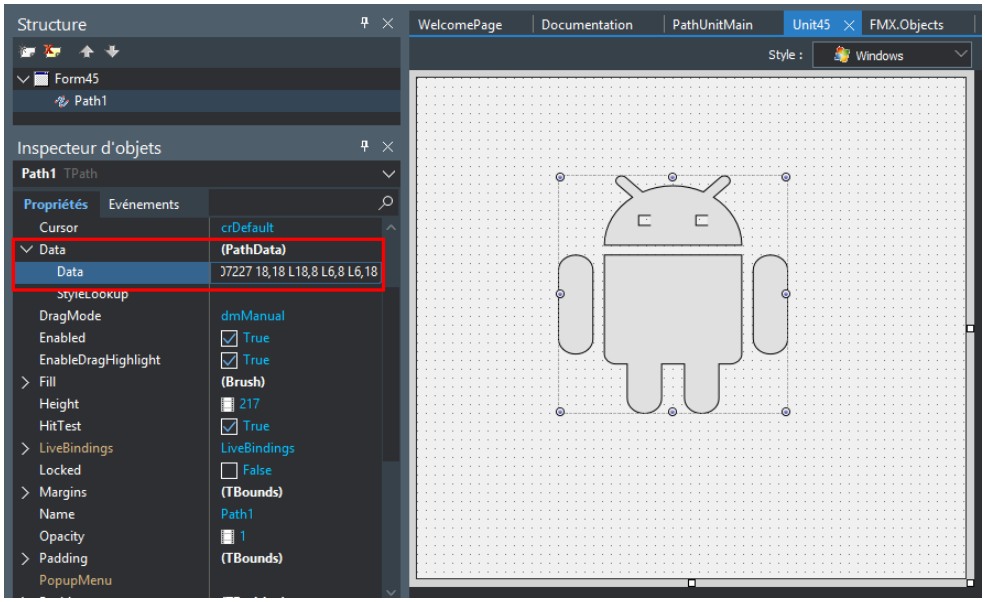


4. Utiliser le bouton [</>] permettant de voir le texte du fichier SVG (Scalable Vector Graphics).

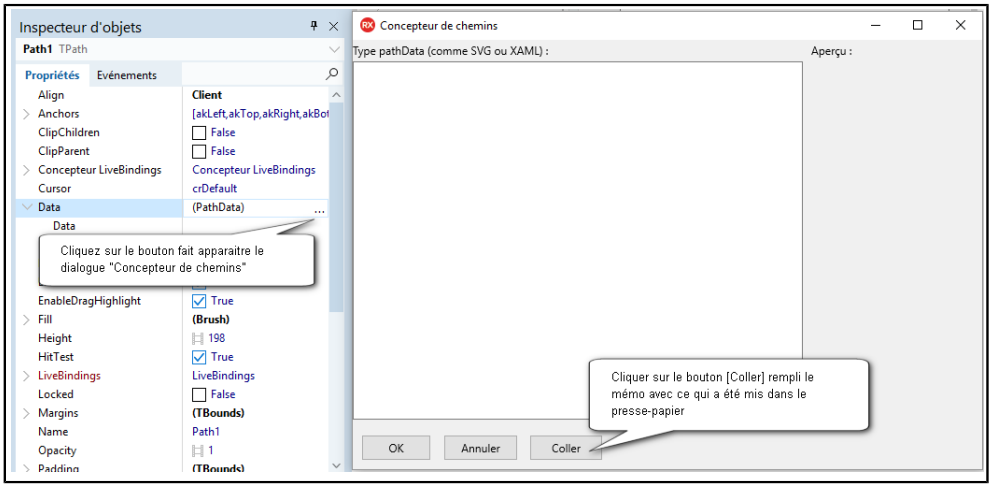
5. Sélectionner la partie données.



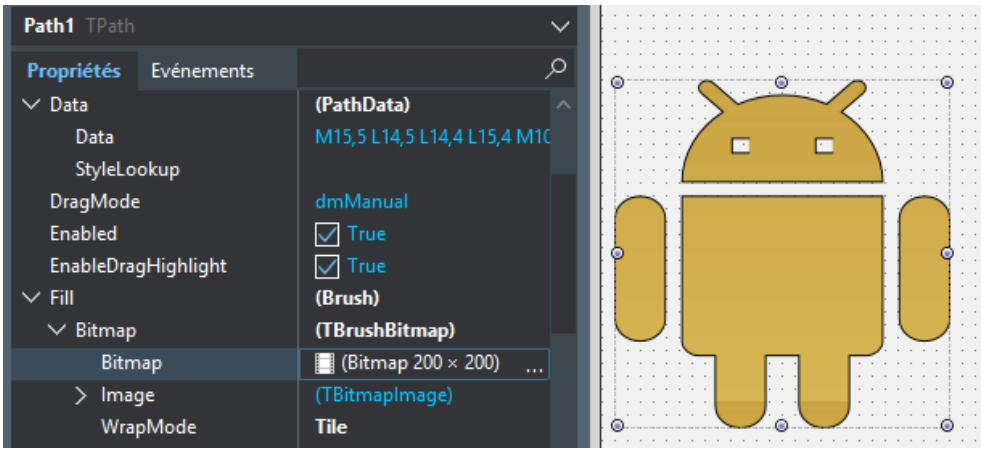
6. Coller cette sélection dans la propriété **Data.Data** et valider (touche <Entrée>).



6'. En alternative il est possible d'obtenir un dialogue « Concepteur de chemins » en cliquant sur le bouton […] de la propriété Data et de saisir les diverses instructions.



7. Enfin, colorier à son goût grâce aux propriétés **Fill** et l'on obtient le trophée « Droïd d'or », en quelques clics de souris et sans une seule ligne de code.



I-B - SVG, ai-je mentionné SVG ?

Cela fait déjà plusieurs fois que j'utilise cet acronyme, il serait peut-être temps d'en savoir un peu plus. Raccourci de ***Scalable Vector Graphics*** que je traduirai par « graphique vectoriel évolutif », c'est un format de données ASCII conçu pour décrire des ensembles de graphiques vectoriels 2D, basé sur XML et répondant à une **norme** précise.

Considérez cela comme une sorte de langage graphique basé sur le texte qui décrit les images avec des formes vectorielles, du texte et des graphiques matriciels intégrés.

Les fichiers d'extension .SVG fournissent donc des graphiques indépendants de la résolution d'un écran et dans un format compact. La possibilité de styler SVG avec CSS et de prendre en charge les scripts et l'animation fait de SVG une partie intégrante de la plate-forme Web.

Loin de moi l'idée d'expliquer ces normes je préfère vous renvoyer vers les articles **wikipédia** en mettant en exergue **l'image explicative** de l'article et les différents sites de l'**organisation W3C**.

*Je rappelle, de ce format, qui commence à être bien utilisé des designers web, **TPath** ne prend en charge que la partie données (**d=**) de l'élément **<path>**.*

Cette mention est fort dommage car le **SVG** contient bien d'autres informations comme les couleurs, les styles, etc.

*Je vous recommande la lecture de **ce tutoriel** sur les bases de la syntaxe de **Path**, ou, plus complet, **celui-ci***

I-C - Que va-t-on aborder alors ?

Dans ce tutoriel je vais vous présenter deux petites applications qui tenteront de démontrer les avantages de ce composant par rapport à l'utilisation d'images classiques.

La première application a surtout été écrite dans le but d'étudier les propriétés **Fill** et **Stroke** du composant, en bref la colorisation du dessin, en bonus : un chargement possible à partir d'un fichier **SVG** (dans la mesure où il s'agira d'un fichier simple) mais aussi l'utilisation des composants de gestion des couleurs et dégradés.

Le second programme est plus orienté utilisation pratique et aperçu des avantages. Par effet de bande ce programme fournira une approche de la raison des images multiples (**multiresbitmap**).

II - Application 1 : Obtenir des images sympas

Charger une image, et même la coloriser, je l'ai montré lors de mon introduction. Inconvénient tout se faisait pendant le design il me fallait donc trouver un moyen de faire la même chose au cours de l'exécution, ce que j'ai réalisé en plusieurs étapes :

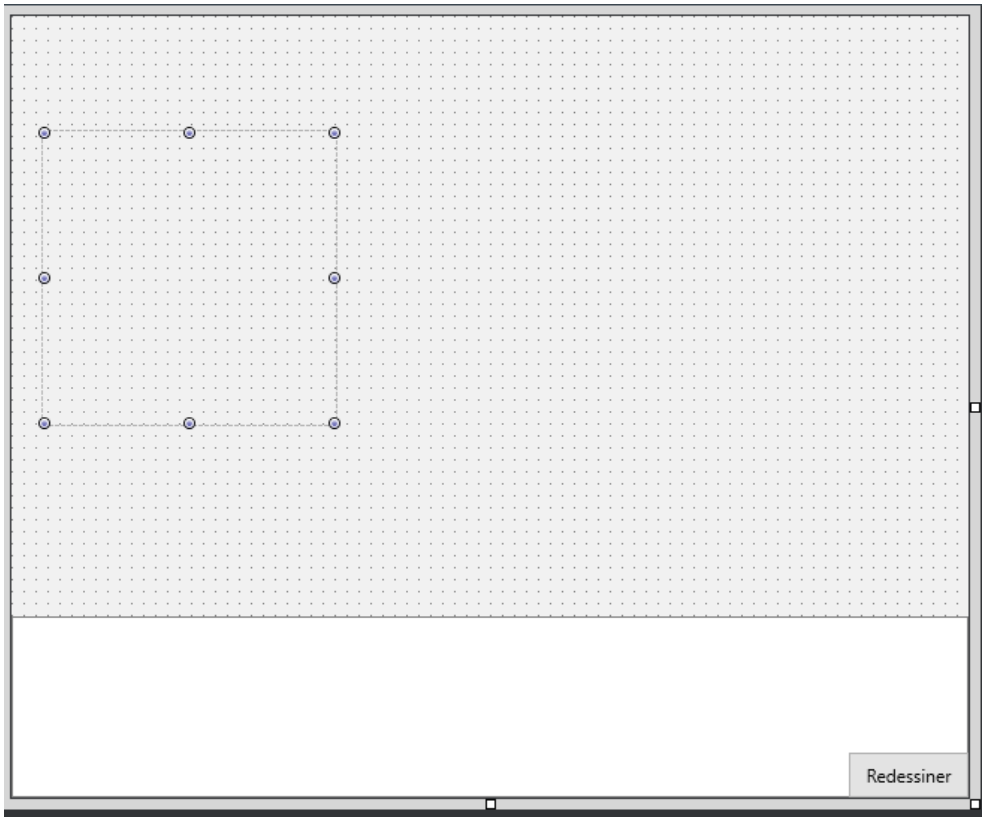
1. Utiliser un **TMemo** pour saisir (ou copier) la partie données (**d**) de l'élément **<path..>** d'un fichier **SVG** ouvert dans un éditeur de texte ou visible directement via l'explorateur internet.
2. Faire une colorisation simple sur la propriété **Fill.Color**. Le but était d'avoir le même comportement que sur **le site icones8.fr**. Cette étape, je l'abandonnerai plus tard au profit d'un dialogue un peu plus poussé permettant d'utiliser les dégradés.
3. Sauvegarder le résultat dans une image m'a ensuite semblé un bon exercice, en obtenir plusieurs tailles fut un gadget supplémentaire.
4. Ayant ensuite marre, au cours de mes essais, de faire du copier-coller ou même des essais de dessins en saisissant directement des instructions, je me suis ensuite attaqué à un chargement direct à partir d'un fichier **SVG**.

   Attention : uniquement des fichiers simples, ne contenant la partie data comme cela peut nous être fourni par le site.

5. Enfin obtenir des images encore plus sympathiques en utilisant la possibilité des dégradés m'a tenté.

II-A - Chargement simple

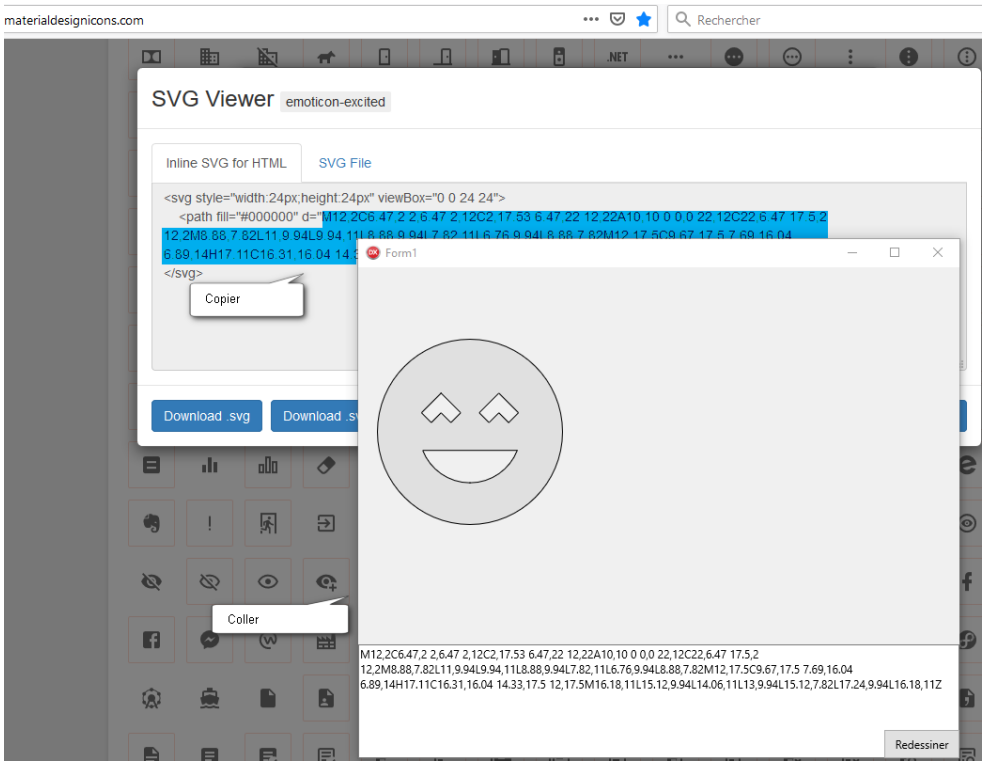
À ma forme déjà présentée chapitre **I.A**, je rajoute un **TMemo** et un **TButton**, le tout légèrement agrémenté.



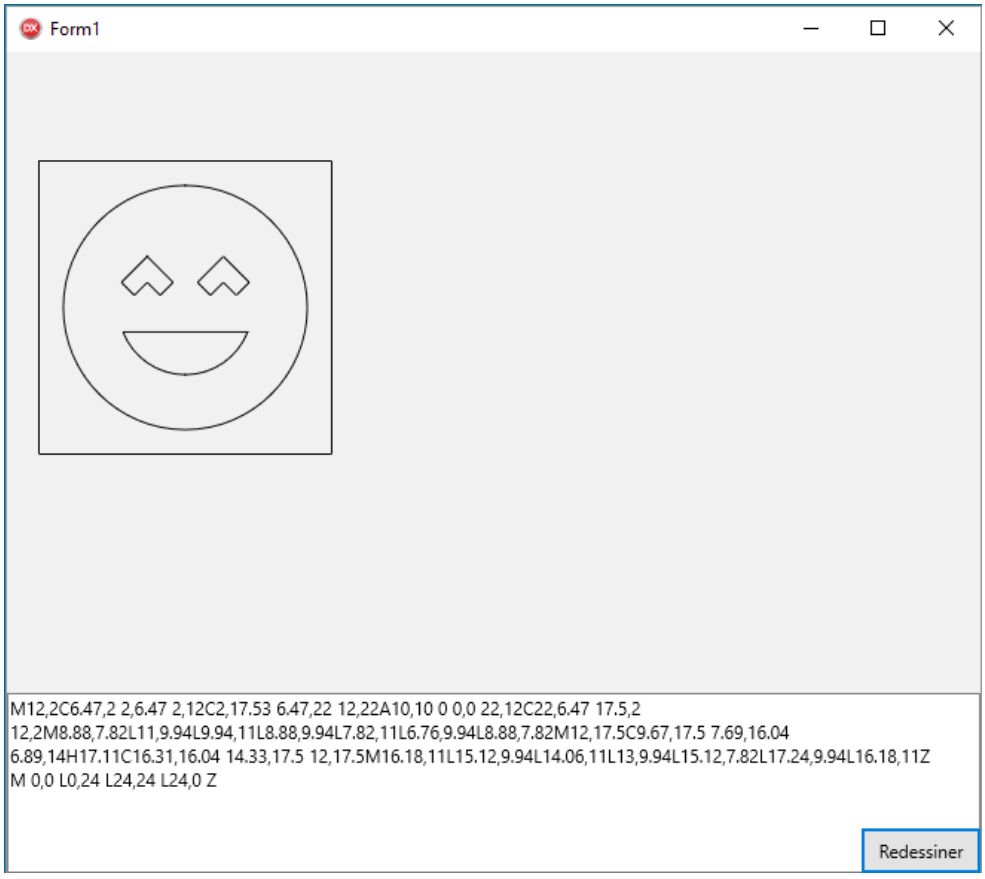
Un peu de code pour le bouton.

```
 procedure TMainForm.btnRepaintClick(Sender: TObject);
// Charge le nouveau dessin contenu dans le mémo
begin
  Path1.BeginUpdate;
  Path1.Data.Data := Memo1.Text;
  Path1.EndUpdate;
end;
```

Et je peux réitérer l'expérience de l'introduction, sélectionner la partie données d'un fichier **SVG**, coller celle-ci dans le mémo, et tester le résultat en cliquant sur le bouton.



Je peux même saisir quelques données supplémentaires afin d'inscrire le dessin dans un rectangle.



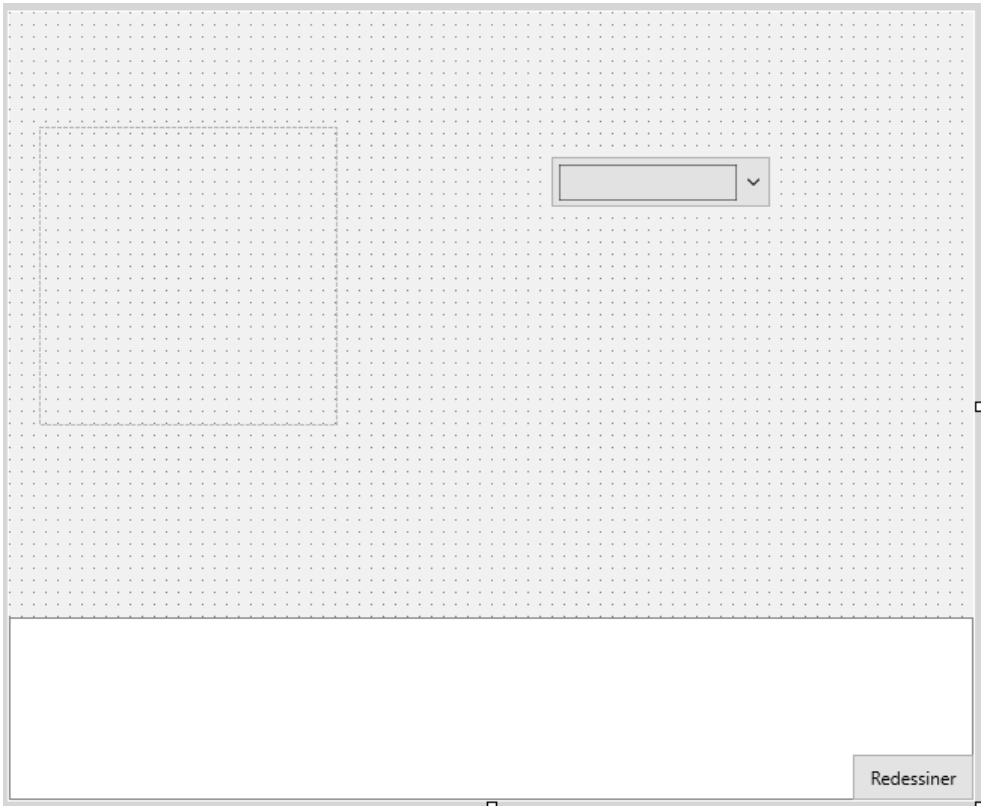
Bien sûr, on peut réussir à le faire par tâtonnements successifs mais, en regardant de plus près les données du fichier SVG affichées,

la première ligne nous fourni un indice viewBox= "0 0 24 24"

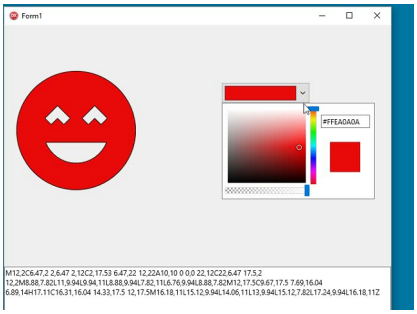
indiquant que le dessin est inscrit dans un carré de 24x24, les commandes suivantes M 0,0 L0,24 L24,24 L24,0 Zdessineront le carré

II-B - Colorisation simple

Premier pas supplémentaire, coloriser le **TPath** autrement qu'au moment du design. Pour cela je rajoute un **TColorComboxBox** à ma forme.



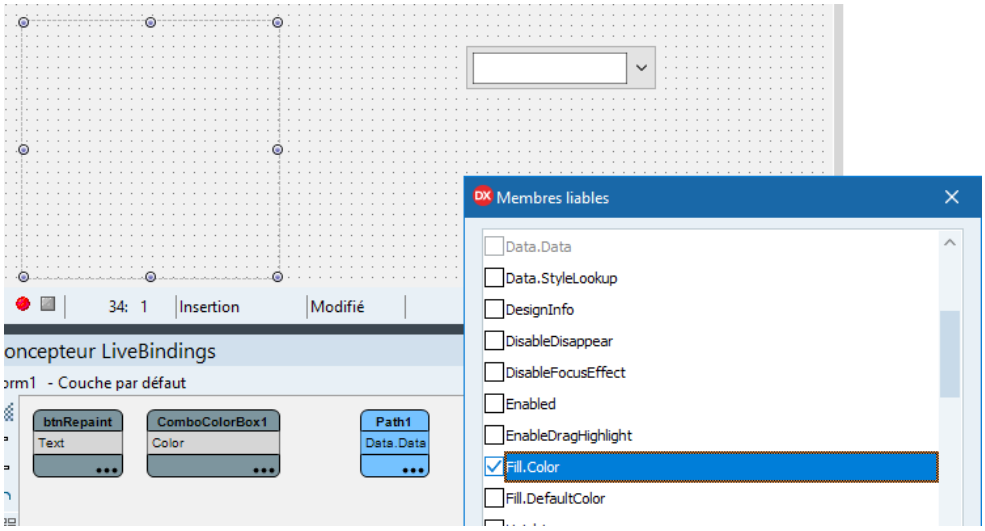
Comment ensuite appliquer la couleur choisie à la propriété **Fill.Color** de mon **TPath** ? Il y a bien sûr la méthode classique utilisant l'évènement **OnChange** du composant **ColorComboBox**.



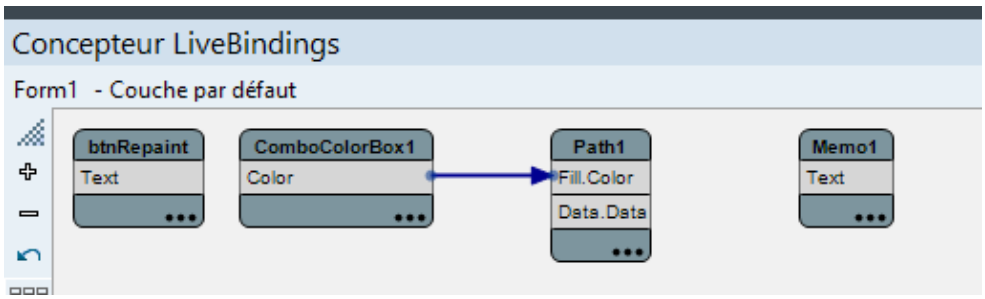

```
procedure TForm1.ComboColorBox1Change(Sender: TObject);
begin
Path1.Fill.Color:=ComboColorBox1.Color;
end;
```

Chantre de l'utilisation des **LiveBindings** je préfère montrer ici comment, en quelques étapes, passer par ceux-ci. À peine plus de clics de souris mais pas de code.

1	Appelez le concepteur de liaison, soit *via* le menu contextuel du composant TPath (clic droit, puis sélection de l'option « Lier visuellement … », soit *via* le menu principal de l'IDE « Voir »« Fenêtres d'outils »« Concepteur LiveBindings » ou encore via le menu qui se trouve en bas de l'inspecteur d'objet, l'option « Lier visuellement …»
2	Ajoutez la propriété **Fill.Color** aux membres observables de **Path1**



3	Établissez le lien entre la propriété **Color** de **ColorComboBox1** et la propriété **Fill.Color** de **Path1**.



*Au passage, notez que **Path1** propose la propriété **Data.Data** qu'il serait possible de lier à la propriété **Text** de **Memo1**. Je ne le ferais pas afin de permettre la saisie de données « à la main » donc forcément incomplète tant qu'elles ne sont pas validées en utilisant le bouton.*

*Si vous avez eu l'occasion d'aller sur **le site icones8.fr**, vous aurez pu voir qu'il est possible de changer la couleur (et même plus).*

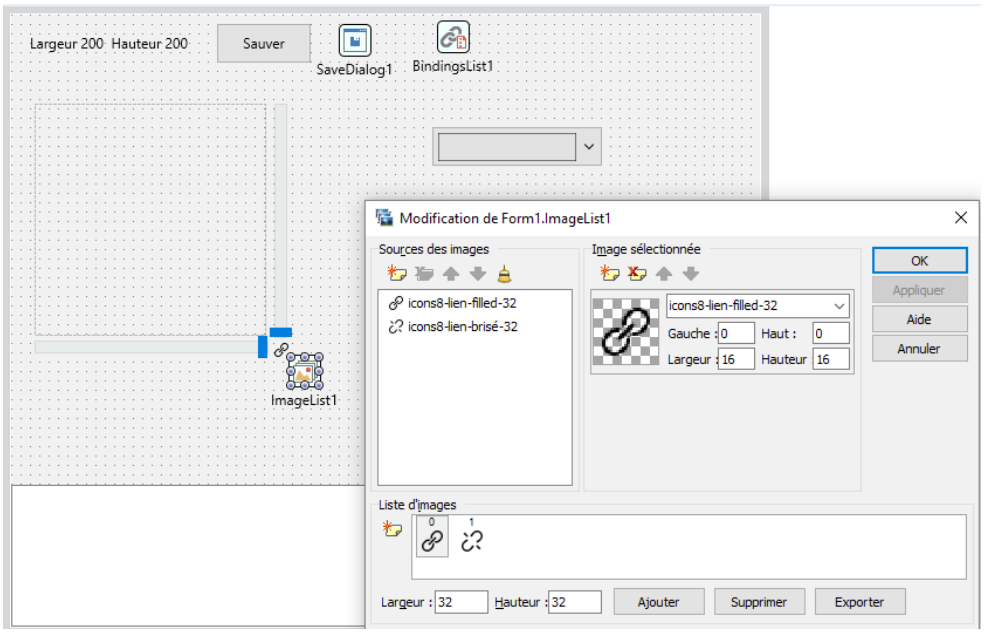
*Astuce bonus, cliquez sur le bouton **</HTML>** et vous retrouverez les données du **path**.*

II-C - Sauvegarder le résultat

Cette étape me paraissait évidente, maintenant que je pouvais obtenir des images coloriées à volonté, les sauvegarder pour une utilisation dans de futurs programmes devenait nécessaire.

De plus il faut tenir compte des diverses résolutions (dpi (dots per inch ou points par pouces)) possibles pour mes différentes cibles, il me fallait donc aussi prévoir la possibilité de changer les tailles (hauteur et largeur) de mon **TPath**.

Je vais ajouter à ma forme deux **TTrackBar**s, un **TLabel** qui contiendra les indications de taille, un **TButton** qui déclenchera l'opération de sauvegarde via un **TSaveDialog** et enfin un ultime petit **TButton**, associé à un **TImageList**, afin de dissocier ou non les hauteurs et largeurs de mon **TPath**.



II-C-1 - La taille de la future image

Je commence par renommer les deux **TTrackBar**s en **Largeur** et **Hauteur**, je mets également leurs maximums à ***200***, leurs minimums à ***20***. Le **TLabel** est renommé en **LabelTaille** et le petit bouton en **btnLien**.

Je code ensuite les évènements nécessaires au redimensionnement du **TPath**.

```
procedure TForm1.btnLienClick(Sender: TObject);
// Associer/dissocier les réglages de hauteur et largeur de l'image
begin
  if btnLien.Imageindex = 0 then
```

```
    btnLien.Imageindex := 1
  else
    btnLien.Imageindex := 0;
end;

procedure TForm1.HauteurChange(Sender: TObject);
begin
// l'index de l'image m'indique s'il faut associer ou non les valeurs
  if btnLien.Imageindex = 0 then   // associées
  begin
    Largeur.Value := Hauteur.Value;
    Path1.Width := Largeur.Value;
  end;
  Path1.Height := Hauteur.Value;
  LabelTaille.Text := Format('Largeur %0.0f  Hauteur %0.0f',
    [Largeur.Value, Hauteur.Value]);
end;

procedure TForm1.LargeurChange(Sender: TObject);
begin
// l'index de l'image m'indique s'il faut associer ou non les valeurs
  if btnLien.Imageindex = 0 then // associées
  begin
    Hauteur.Value := Largeur.Value;
    Path1.Height := Hauteur.Value;
  end;
  Path1.Width := Largeur.Value;
  LabelTaille.Text := Format('Largeur %0.0f  Hauteur %0.0f',
    [Largeur.Value, Hauteur.Value]);
end;
```

*Oui, je n'ai pas utilisé les **LiveBindings**, dans cette partie ce serait peu pratique du fait que hauteur et largeur peuvent être liées ou non.*

II-C-2 - Sauver l'image obtenue

Du bouton pour la sauvegarde, renommé **btnSave** j'utiliserai son évènement OnClick.

```
procedure TForm1.btnSaveClick(Sender: TObject);
// Sauvegarde du dessin dans un fichier
var
  ABitmap: TBitmap; // un bitmap
  ACodec: TBitmapCodecSaveParams; // Qualité
  ACodecManager: TBitmapCodecManager; // filtres possibles

begin
  ACodecManager := TBitmapCodecManager.Create;
  try
    // Fourni les filtres possibles
    SaveDialog1.Filter := ACodecManager.GetFilterString;
    if SaveDialog1.Execute then
    begin
      ACodec.Quality := 100;
      ABitmap := Path1.MakeScreenshot; // Récupère le dessin
      try
        ABitmap.SaveToFile(SaveDialog1.FileName, @ACodec); // sauvegarde
      finally
        FreeAndNil(ABitmap);
      end;
    end;
  finally
    FreeAndNil(ACodecManager);
  end;
end;
```

La seule partie un peu plus technique est dans l'utilisation du **TBitmapCodecManager** pour récupérer les filtres (type de fichier) possibles selon la plateforme cible et dans le paramétrage de la qualité de sauvegarde *via* un **TBitmapCodecSaveParams**.

L'important est surtout dans la méthode utilisée pour obtenir l'image : **MakeScreenShot**.

Il est maintenant temps de faire quelques tests de sauvegarde sous différents formats.

Si la sauvegarde se passe correctement, par contre selon le type de fichier retenu nous n'obtenons pas le même résultat visuel. La transparence semble gardée pour les formats .GIF, .ICO, .PNG et .TIF (confirmé en utilisant un programme comme **Gimp**) cette même transparence est remplacée par la couleur noire pour les formats .JPG, .JPEG et .BMP. Quelques manipulations sur l'image peuvent améliorer le rendu de ces derniers formats. La procédure suivante, alliée à une demande de la couleur de remplacement fait l'affaire.

```
procedure ChangeCouleur(aBmp: TBitmap;
  FromColor: TAlphaColor = TAlphaColors.Null;
  ToColor: TAlphaColor = TAlphaColors.Null);
// Changer une couleur particulière
var
  Data: TBitmapData;
  X, Y: Integer;
  AColor: TAlphaColor;
begin
  aBmp.Map(TMapAccess.ReadWrite, Data); // récupère les pixels
  if FromColor = TAlphaColors.Null then
    FromColor := Data.GetPixel(0, 0); // premier point = transparence
  for X := 0 to aBmp.Height do
    for Y := 0 to aBmp.Width do
    begin
      AColor := Data.GetPixel(Y, X);
      if AColor = FromColor then
        Data.SetPixel(Y, X, ToColor); // modification
    end;
  aBmp.UnMap(Data); // Applique les modifications
end;
```

Il vous faudra ajouter une nouvelle forme modale pour le dialogue et faire un test sur l'extension souhaitée pour le déclencher. Par exemple :

```
        if Sametext(ExtractFileExt(SaveDialog1.FileName), '.bmp') then
         begin
           // si bmp demande quelle sera la couleur de fond
           if FormSelectColor.ShowModal = mrOK then
             ChangeCouleur(ABitmap, TAlphaColors.Null,
               FormSelectColor.ColorListBox1.Color);
         end;
```

Vous retrouverez cette partie dans le source complet.

Reste un petit bémol à cette procédure, l'utilisation de l'instruction FromColor := Data.GetPixel(0, 0);,

permettant d'obtenir comme couleur de référence le premier pixel de l'image peut être, de fait, un élément du dessin et non un pixel de fond d'image.

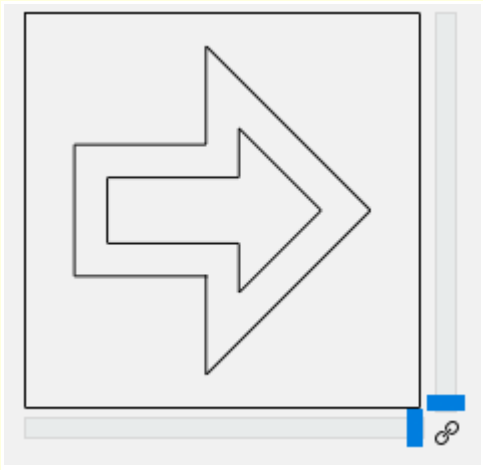
Dessinez un rectangle encadrant un dessin.

Par exemple en saisissant les données suivantes :

M11,16 L3,16 L3,8 L11,8 L11,2 L21,12 L11,22 L11,16 M13,7 L13,10 L5,10 L5,14 L13,14 L13,17 L18,12 L13,7 Z

M0,0 L0,24 L24,24 L24,0 Z

Vous obtiendrez



*Sauvegardez au format **bmp**, vous obtiendrez une image monochrome.*

Pour m'assurer que le premier pixel soit toujours un pixel transparent mon astuce consiste à mettre le composant **TPath** à l'intérieur (***Align = Client)*** d'un **TLayout** (que je nomme ***PathContainer***). Je change ensuite les propriétés **Padding** de ce **TLayout** de façon à obtenir une bordure de un pixel. Il faut alors modifier le code gérant la taille qui doit maintenant concerner le **TLayout**, et celui de la sauvegarde `ABitmap := PathContainer.MakeScreenshot;`

L'image sauvegardée est maintenant celle du **TLayout**.

*Cette idée de récupérer la couleur du premier pixel est bien sûr la solution la plus simple. Une autre solution consisterait à récupérer la première couleur d'un pixel qui ne fasse pas partie des contours (différente de la couleur de la propriété **stroke)** et non contenu dans la zone définie avec la fonction **PointInObject**.*

II-D - Charger un fichier SVG

*Mon intention n'est pas de faire un chargeur de fichier complet, juste de pouvoir charger les fichiers SVG « simples » obtenus via **Material Design Icons**.*

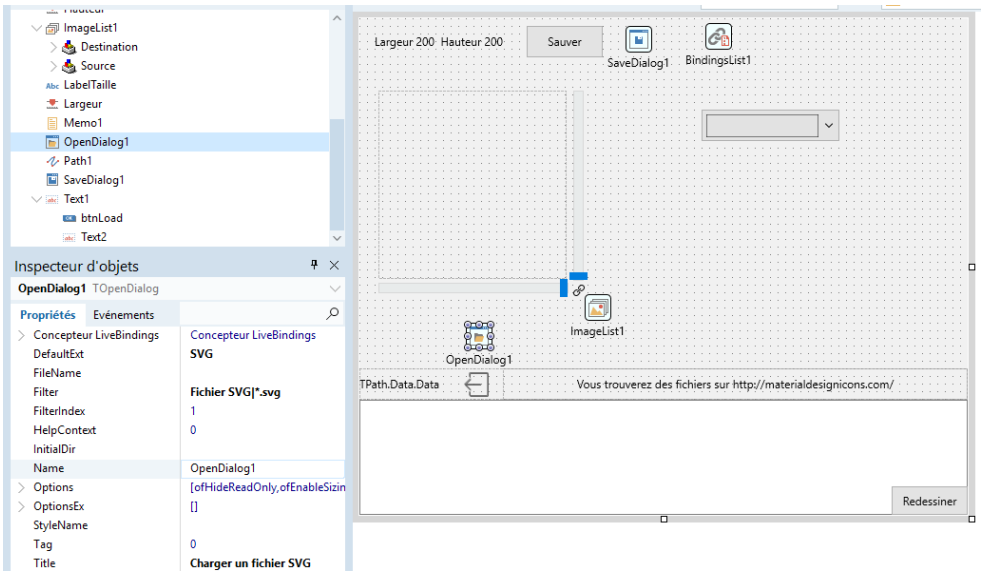
Que contient un fichier SVG ?

Android.svg
```
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//
EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"><svg xmlns="http://www.w3.org/2000/
svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1" width="24" height="24" viewBox="0
 0 24 24"><path d="M15,5H14V4H15M10,5H9V4H10M15.53,2.16L16.84,0.85C17.03,0.66 17.03,0.34
 16.84,0.14C16.64,-0.05 16.32,-0.05 16.13,0.14L14.65,1.62C13.85,1.23 12.95,1 12,1C11.04,1
 10.14,1.23 9.34,1.63L7.85,0.14C7.66,-0.05 7.34,-0.05 7.15,0.14C6.95,0.34 6.95,0.66
 7.15,0.85L8.46,2.16C6.97,3.26 6,5 6,7H18C18,5 17,3.25 15.53,2.16M20.5,8A1.5,1.5 0 0,0
 19,9.5V16.5A1.5,1.5 0 0,0 20.5,18A1.5,1.5 0 0,0 22,16.5V9.5A1.5,1.5 0 0,0 20.5,8M3.5,8A1.5,1.5 0
 0,0 2,9.5V16.5A1.5,1.5 0 0,0 3.5,18A1.5,1.5 0 0,0 5,16.5V9.5A1.5,1.5 0 0,0 3.5,8M6,18A1,1 0 0,0
 7,19H8V22.5A1.5,1.5 0 0,0 9.5,24A1.5,1.5 0 0,0 11,22.5V19H13V22.5A1.5,1.5 0 0,0 14.5,24A1.5,1.5
 0 0,0 16,22.5V19H17A1,1 0 0,0 18,18V8H6V18Z" /></svg>
```

Un fichier **xml** donc, qu'il serait possible de traiter *via* un **XMLDocument**. Seulement, cette étude ne se base que sur des fichiers simples et j'ai voulu tenter le diable en ouvrant d'autres fichiers SVG de structure plus complexe et la plupart des chemins n'étaient pas détectés.

L'objectif final étant d'ouvrir le fichier et d'en récupérer la substantifique moelle c'est-à-dire ce qui se trouve entre les guillemets, derrière le **d="**, et pour faire simple, j'ai préféré utiliser les expressions régulières (d="(.*)") pour accéder à ces données.

D'une manière basique je rajoute un **TButton (*btnLoad*)** pour le chargement et un **TOpenDialog** pour rechercher les fichiers. Pour améliorer mon design je positionnerai le bouton dans un **TText**



Reste à codifier l'événement **OnClick** de ***btnLoad.***

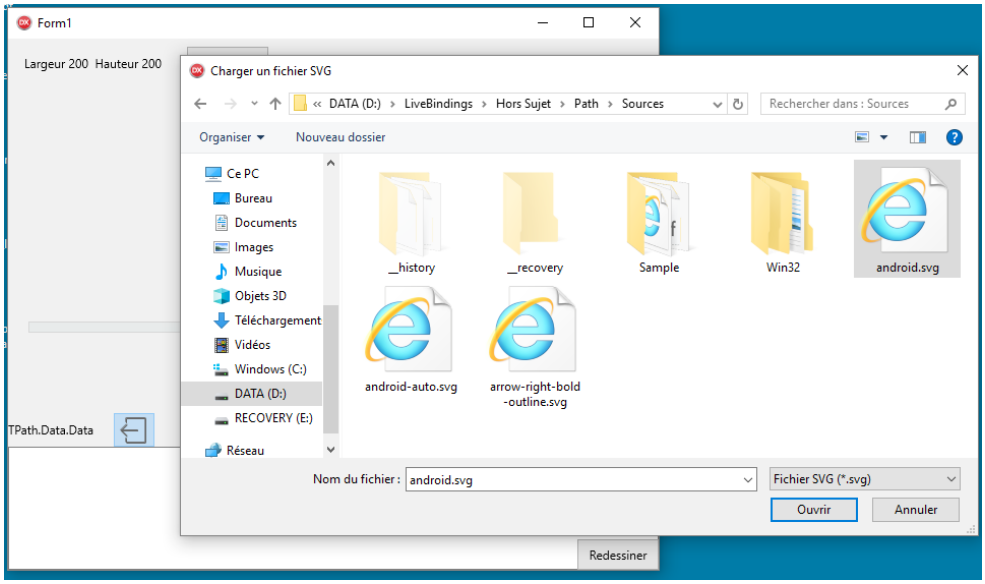
```
procedure TMainForm.btnLoadClick(Sender: TObject);
// Charger un fichier SVG
// Attention tous les SVG ne sont pas forcément possible à traiter
// chargement très basique à améliorer
var
  SL: TStringList;
  Match: TMatch;
begin
  if OpenDialog1.Execute then
  begin
    // Prépare le composant TPath
    Path1.BeginUpdate;
    Path1.Data.Clear;   // efface
    // remet les valeur par défaut
    Path1.Fill.Kind := TBrushKind.Solid;
    Path1.Fill.Color := Path1.Fill.DefaultColor;
    Path1.Stroke.Kind := TBrushKind.Solid;
```

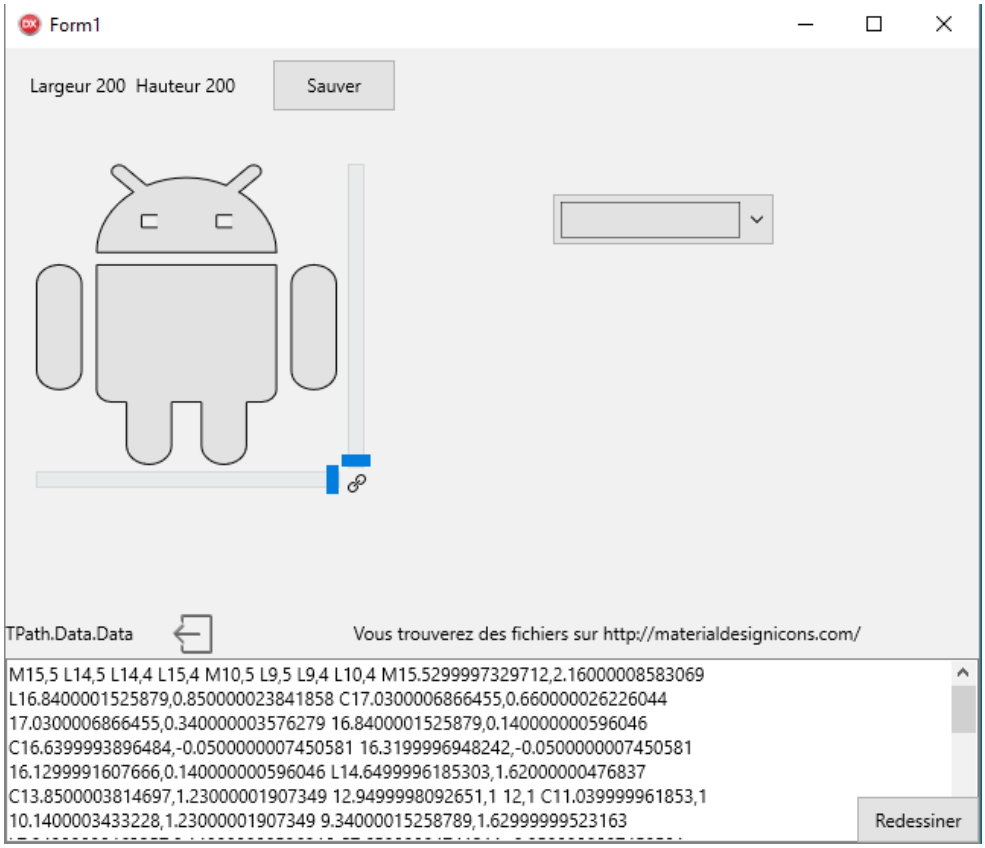
```
    Path1.Stroke.Color := Path1.Stroke.DefaultColor;

    Memo1.Lines.Clear; // efface le chemin affiché

    // Charge le fichier dans une TstringList préalablement crée
    SL := TStringList.Create;
    try
      SL.LoadFromFile(OpenDialog1.FileName);
      // utilisation de l'expression régulière
      Match := TRegEx.Match(SL.Text, 'd="(.*)"');
      while Match.Success do
      begin
        Path1.Data.Data := Path1.Data.Data + Match.Groups.Item[1].Value;
        Match := Match.NextMatch;
      end;
    finally
      FreeAndNil(SL);
    end;
    // affichage des données dans le mémo
    Memo1.Lines.Add(Path1.Data.Data);
    Path1.EndUpdate; // dessin
  end;
end;
```

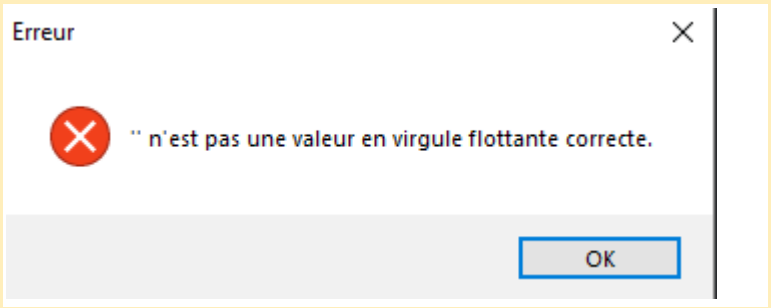
*Qui dit expressions régulières dit utilisation de l'unité **System.RegularExpressions** ce qu'il ne faudra pas oublier d'indiquer*

```
implementation

{$R *.fmx}

uses System.RegularExpressions;
```

Et à vérifier cela à l'exécution.







La loi de Murphy : « tout ce qui peut foirer va foirer » s'applique à ce code.

Une chaîne de caractères trop longue, une coupure (saut de ligne) au sein de la séquence à récupérer, un format différent ou encore une norme différente et c'est l'erreur.

II-D-1 - Addenda

Après publication du programme, j'ai voulu résoudre le téléchargement en utilisant les possibilités de lecture de XML offertes et donc augmenter le nombre de fichiers SVG que je pourrais charger.

Un puriste préférera certainement ce code à la méthode précédente.

Ce n'est en aucun cas une mise en œuvre complète du standard SVG, mais cela s'en rapproche ! En tout cas assez pour quelques images SVG ne contenant que des données de chemin.

```
// nécessite les unités  Xml.XMLIntf, Xml.adomxmldom, Xml.XMLDoc

procedure SVGLoadFromFile(const FileName: string;
                          const DestPath : TPath);
var
  Doc: IXMLDocument;
  Data,Node: IXMLNode;
  I : Integer;
  APathData : TPathData;

  procedure g(ANode : IXMLNode);
  var j : integer;
      CNode : IXMLNode;
      ASubPathData : TPathData;
  begin
   ASubPathData := TPathData.Create;
   for j:=0 to ANode.ChildNodes.Count-1 do
     begin
      CNode:=ANode.ChildNodes[j];
      if CNode.HasAttribute('d')
      then begin
          ASubPathData.Data:=CNode.Attributes['d'];
          DestPath.Data.AddPath(ASubPathData);
      end
      else g(CNode);
     end;
  end;

begin
  DestPath.Data.Clear;
  APathData:=TPathData.Create;

  DefaultDOMVendor:='ADOM XML v4';
  Doc := LoadXMLDocument(FileName);
  Data:=Doc.DocumentElement;

  for I := 0 to Data.ChildNodes.Count-1 do
  begin
    Node := Data.ChildNodes[I];
    if Node.HasAttribute('d')
      then begin
          APathData.Data:=Node.Attributes['d'];
          DestPath.Data.AddPath(APathData);
      end
      else g(Node);
  end;
end;
```

Même si plus de fichiers sont correctement lus, le rendu ne sera pas toujours correct du fait de certaines transformations (attribut **transform**) peuvent être indiquées.

*Plusieurs transformations sont possibles, j'en ai trouvé au moins deux types : **translate** et **matrix.** La première me semble facile à appliquer grâce à la fonction **TPath.Data.Translate**. Par contre c'est sans succès que j'ai tenté **TPath.Data.Matrix** même si le nom de la fonction était prometteur.*

*Plus d'informations sur les transformations**ici**.*

II-E - Les dégradés et autres remplissages

Le monochrome c'est déjà pas mal, mais il est temps de s'interroger sur les autres propriétés possibles tant sur le remplissage (**Fill**) que sur les contours (**Stroke**).

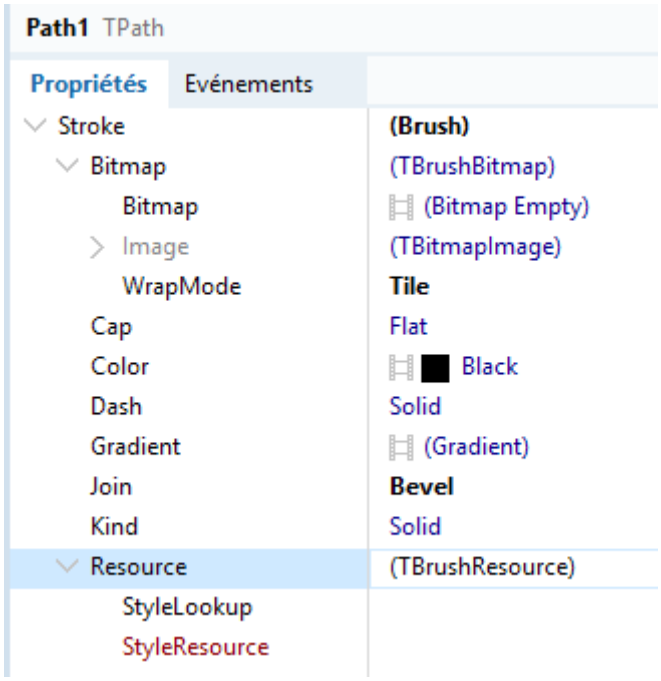
Il est temps pour moi de supprimer le **TColorCombobox** du chapitre **II.B** et de le remplacer par un dialogue qui va couvrir plus de possibilités.

II-E-1 - Dialogue de colorisation

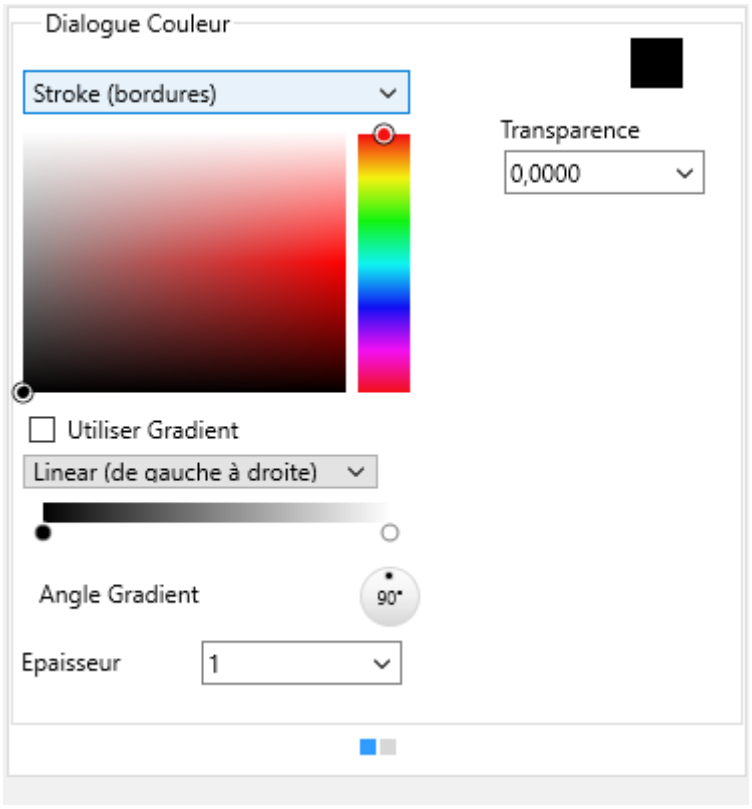
Ce dialogue, une ensemble de deux panneaux à pour but de couvrir un maximum de possibilité de changements pour les propriétés ***Fill*** et ***Stroke*** à l'exécution du programme. Les propriétés ***Fill.Resource*** et ***Stroke.Resource*** ne seront pas abordées dans cette première partie. J'ai également jugé qu'il était inutile, vu la taille maximum de mon **TPath**, de jouer sur ***Bitmap.WrapMode***.

En image quelques possibilités couvertes par ce dialogue :

Pour le contour (propriété ***Stroke***)

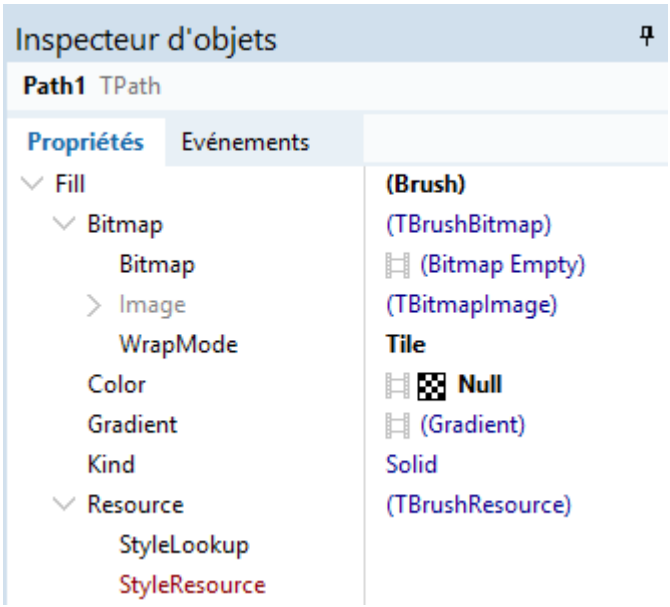


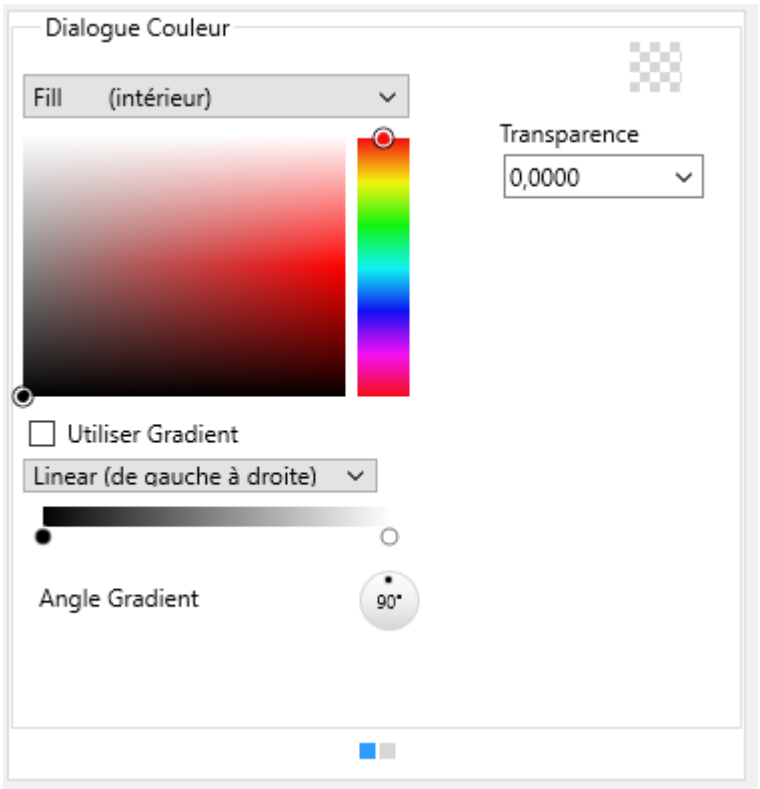
Propriété Stroke de Tpath

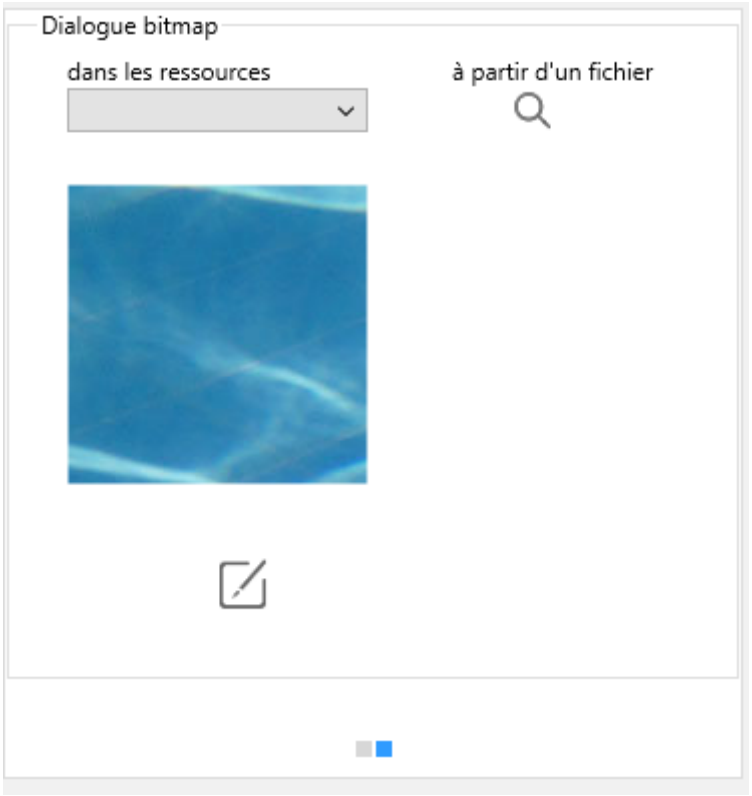


Dialogue couleur pour Stroke

Et pour le remplissage (propriété ***Fill***)







Fill.Bitmap

II-E-1-a - La sélection de la couleur

Partie délicate, refaire le dialogue de la **TColorComboBox** n'est pas si facile que cela en a l'air. Il faut comprendre les enchaînements entre les différents composants **TColorQuad**, **TColorPicker** et **TColorBox**.

La seule piste fournie est l'indication de la propriété manquante quand l'on pose le composant.

- Un **TColorPicker** sera lié à un **TColorQuad** ;
- Le **TColorQuad** nécessitera un **TColorBox** ;
- Et c'est la propriété **Color** du **TColorBox** que j'utiliserai.

À cela s'ajoute une valeur de transparence (propriété **Alpha** du **TColorQuad**) qui sera, dans mon cas, géré par un **TComboTrackBar**.

*Il est facile de s'y perdre ! Heureusement presque tout est réactif et seul le changement de valeur du **TComboTrackBar** est à gérer.*

```
procedure TMainForm.ColorAlphaChange(Sender: TObject);
// Changement de la transparence
begin
  ColorQuad1.Alpha := ColorAlpha.Value;
end;
```

Pas mal de changements pouvant se produire j'introduis un record qui me servira à mémoriser les attributs.

```
type
 TPathGradient = record
    Fill: TBrush;
    FillAngle: Single;
```

```
    Stroke: TStrokeBrush;
    StrokeAngle: Single;
  end;

 TMainForm = class(TForm)
    Path1: TPath;

  private
    { Déclarations privées }
    memPG: TPathGradient;

   public
    { Déclarations publiques }
  end;
```

II-E-1-b - Les dégradés

Là encore il a fallu faire face à un joli challenge pour comprendre comment fonctionnait un **TEditGradient** et mais aussi quels calculs allaient entrer en jeu dès qu'il s'est agit de modifier l'angle de rotation.

II-E-1-b-i - TEditGradient

La première chose qu'il m'a fallu découvrir fut de comprendre comment initialiser un **TGradientEdit**. Il faut savoir que ce composant a une propriété **Gradient** qui fait référence à un ensemble de **TGradientPoints**, ses propriétés : **Gradient.Color** et **Gradient.Color1** représentant respectivement les points de début et fin du tableau.

J'ai codé cette initialisation ainsi :

```
    GradientEdit1.BeginUpdate;
    // suppression des points intermédiaires
    for i := 1 to GradientEdit1.Gradient.Points.Count - 2 do
      GradientEdit1.Gradient.Points.Delete(i);
    // initialisation des couleurs de début
    GradientEdit1.Gradient.Color := ColorBox1.Color;
    // et de fin
    GradientEdit1.Gradient.Color1 := TAlphaColors.White;
    GradientEdit1.EndUpdate;
```

II-E-1-b-ii - Dégradé Linéaire et rotations

Lorsque l'on regarde le dialogue de création des dégradés proposé par l'EDI, une molette permet de faire tourner celui-ci. Je m'attendais donc à retrouver une propriété pour cela mais il y en a aucune. La question se posait donc de savoir comment faire. Le « secret », je l'ai trouvé en retrouvant les sources du dialogue Embarcadero\Studio\20.0\source\Property Editors\fmx.design.brush.pas

Avec cette recherche, j'ai presque eu des regrets d'avoir créé mon propre dialogue ! Réutiliser l'existant aurait certainement été plus facile. D'un autre côté je serais certainement passé à côté de plusieurs trucs.

Voilà donc la procédure telle que je l'ai recopiée :

```
procedure CalcPosition(Angle: Single; Gradient: TGradient);
// calcul des positions de début et de fin en fonction d'un angle
var
  X, Y, Koef: Single;
  Radian: Single;
  CosRadian: Single;
  SinRadian: Single;
begin
```

```
  Radian := DegToRad(Angle);
  CosRadian := Cos(Radian);
  SinRadian := Sin(Radian);
  if (CosRadian <> 0) and (Abs(1 / CosRadian) >= 1) and
    (Abs(1 / CosRadian) <= 1.42) then
    X := Abs(1 / CosRadian)
  else
    X := 1;

  if (SinRadian <> 0) and (Abs(1 / SinRadian) >= 1) and
    (Abs(1 / SinRadian) <= 1.42) then
    Y := Abs(1 / SinRadian)
  else
    Y := 1;

  Koef := Max(X, Y);
  Koef := Koef * 0.5;
  Gradient.StartPosition.Point := PointF(0.5 - (CosRadian * Koef),
    0.5 + (SinRadian * Koef));
  Gradient.StopPosition.Point := PointF(0.5 + (CosRadian * Koef),
    0.5 - (SinRadian * Koef));
end;
```

Et son utilisation au sein de l'évènement **OnChange** de mon bouton variateur.

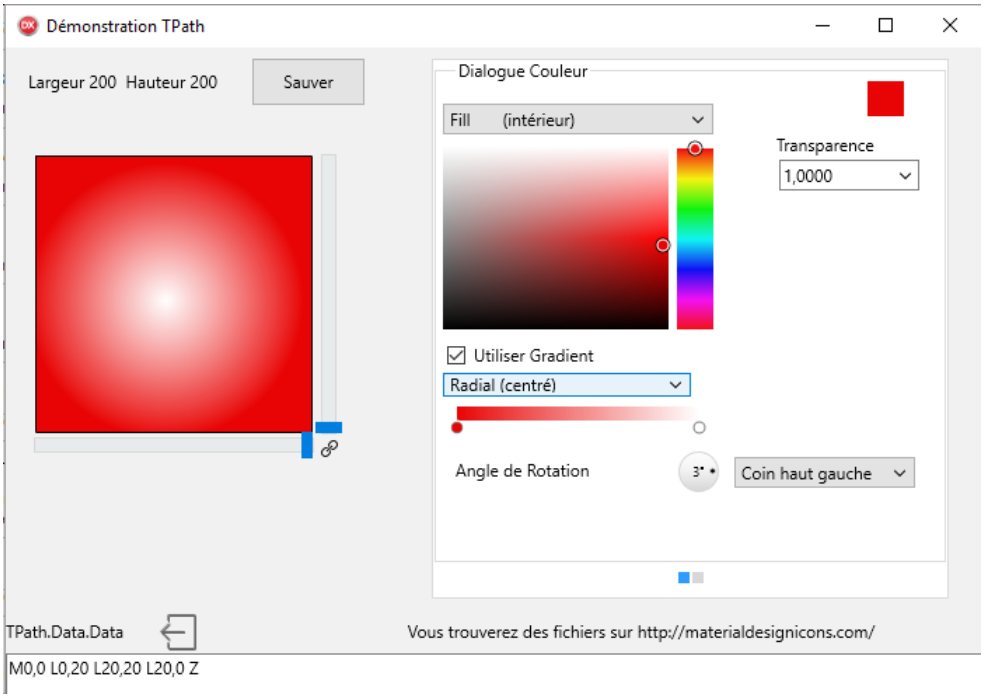
```
procedure TMainForm.AngleGradientChange(Sender: TObject);
// Changement de l'angle du gradient linéaire
begin
  Path1.BeginUpdate;
  if Objet.ItemIndex = 0 then // Remplissage (Fill)
  begin
    CalcPosition(AngleGradient.Value, memPG.Fill.Gradient);
    memPG.FillAngle := AngleGradient.Value;
    Path1.Fill.Gradient := memPG.Fill.Gradient ;
  end
  else
  begin // Contour (Stroke)
    CalcPosition(AngleGradient.Value, memPG.Stroke.Gradient);
    memPG.StrokeAngle := AngleGradient.Value;
    Path1.Stroke.Gradient:= memPG.Stroke.Gradient ;
  end;
  // Redessiner
  Path1.EndUpDate;
end;
```

II-E-1-b-iii - Dégradé Radial et rotations

Contrairement au dégradé linéaire, le dégradé de type radial a bien un angle de rotation accessible (obtenu *via* la propriété **RadialTransform** du gradient).

Si vous avez eu la curiosité de regarder le source indiqué chapitre précédent ou, plus simplement, en lançant le dialogue de création d'un dégradé, vous remarquerez qu'à ce sujet Embarcadero a botté en touche et cache le bouton.

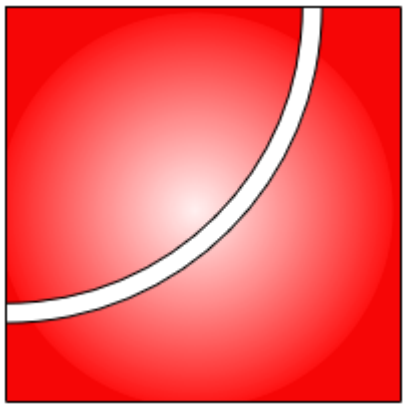
J'ai, malgré tout, voulu tenter le coup.



L'évènement OnChange codé ainsi :

```
procedure TMainForm.RotationDialChange(Sender: Tobject);
// on joue sur Gradient.RadialTransform.RotationAngle
begin
Path1.BeginUpdate ;
if Objet.ItemIndex = 0 then
   begin
      memPG.Fill.Gradient.RadialTransform.RotationAngle := RotationDial.Value
      Path1.Fill.Gradient.RadialTransform.RotationAngle:= RotationDial.Value ;
   end
   else begin
      memPG.Stroke.Gradient.RadialTransform.RotationAngle := RotationDial.Value;
      Path1.Stroke.Gradient.RadialTransform.RotationAngle:= RotationDial.Value ;
   end ;
Path1.EndUpdate ;
```

Cela fonctionne, le point central se déplace selon un cercle dont le centre passe par le point en haut et à gauche du composant **TPath**.



(En blanc le chemin suivi)

La question fut alors : « Est-il possible de déplacer ce centre de rotation ? »

A priori, à la vue des propriétés de **RadialTransform**, il y a bien un **RotationCenter** associé au **RotationAngle**. Là je dois déchanter car mes tests de changement du point n'ont strictement rien changé.

*À la lecture de **cet article**il semblerait que ce bogue soit apparu à partir de X8 et resterait non résolu à ce jour **[RSP-10426]**. Soit c'est le cas, soit j'ai mal fait mes tests, une chose est sûre, douché par l'article je n'ai pas continué dans cette voie.*

II-F - Bilan

Ce programme, pratique pour obtenir des images personnalisées, a été réalisé en première approche du composant **TPath**. Le chapitre suivant sera consacré à des idées d'utilisation et divers éléments non abordés. Les seules réserves que j'émettrai concernent le fait qu'il m'ait fallu écrire une procédure de chargement d'un fichier et que le composant n'accepte qu'un seul remplissage. Je ne peux pas écrire que le composant est monochrome avec les possibilités de dégradés ou de remplissage avec un bitmap ce serait faux, mono-texture serait peut-être un terme plus approprié.

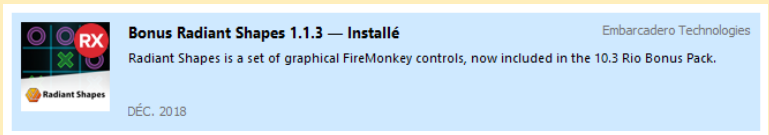
Quand je vois les possibilités offertes par le format SVG sur le web, je ne peux que déplorer que les indications de remplissage de couleur, sorte de CSS, ne puissent être prises en compte.

*Vous retrouverez les sources de ce programme dans l'archive PathProject.zip téléchargeable ici ou encore dans mon **dépôt GitHub** où l'exécutable est aussi dans une archive (ExePathProject.zip).*

III - Utilisation pratique

Après avoir joué avec le composant en apprenant à le colorier, il est temps de passer au côté pratique du composant **TPath** et de découvrir quels peuvent être ses avantages. Le programme que je vous présente maintenant est un condensé de mes diverses expériences dans ce domaine. Encore une fois il s'agit d'un programme qui s'est ébauché de fil en aiguille, une idée en entraînant une autre, chaque idée étant développée dans un onglet.

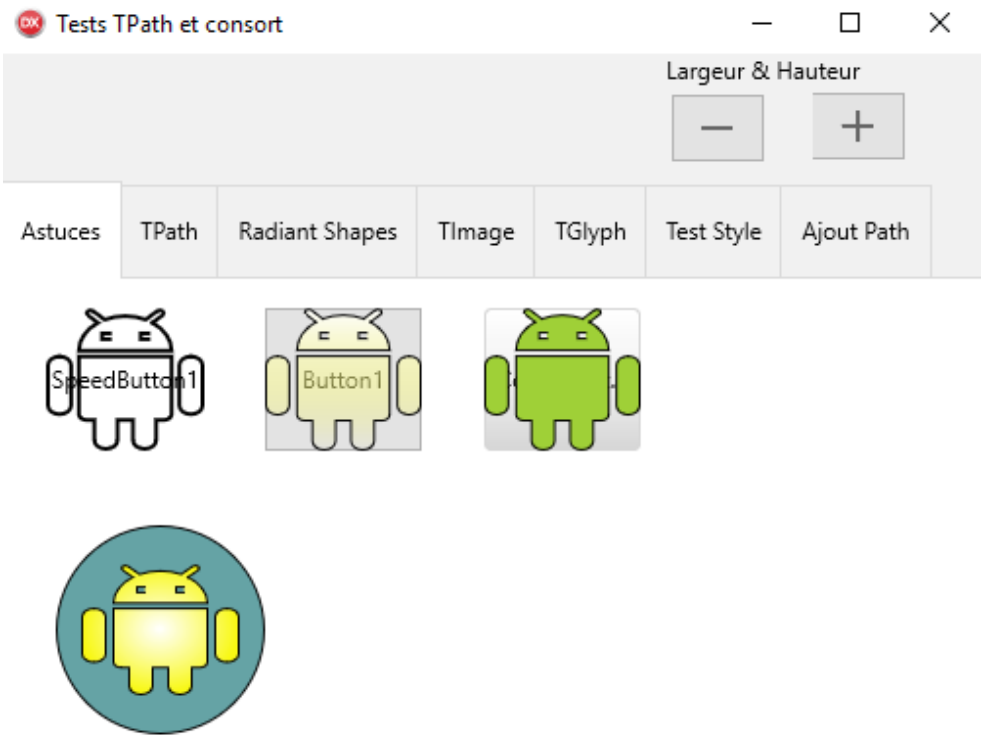
*Dans ce programme, j'ai aussi voulu tester l'utilisation des composants **RadiantShapes** obtenu par **Getit**.*



Si vous ne voulez pas les installer, vous devrez ôter toutes références à ceux-ci dans les sources du programme avant de compiler.

III-A - Astuce simple

Plus qu'une utilisation « remarquable », j'ai d'abord voulu tester l'utilisation de **TPath** dans des boutons ou autres composants. Sur cette première page j'ai donc positionné tout d'abord plusieurs types de boutons (**TSpeedButton**, **TButton**, **TCornerButton**) et adjoint un **TPath** à chacun d'eux. Le dernier « bouton » est en fait un **TCircle** dans lequel aussi insérer un **TPath** pour obtenir un design à la mode « réseaux sociaux ».



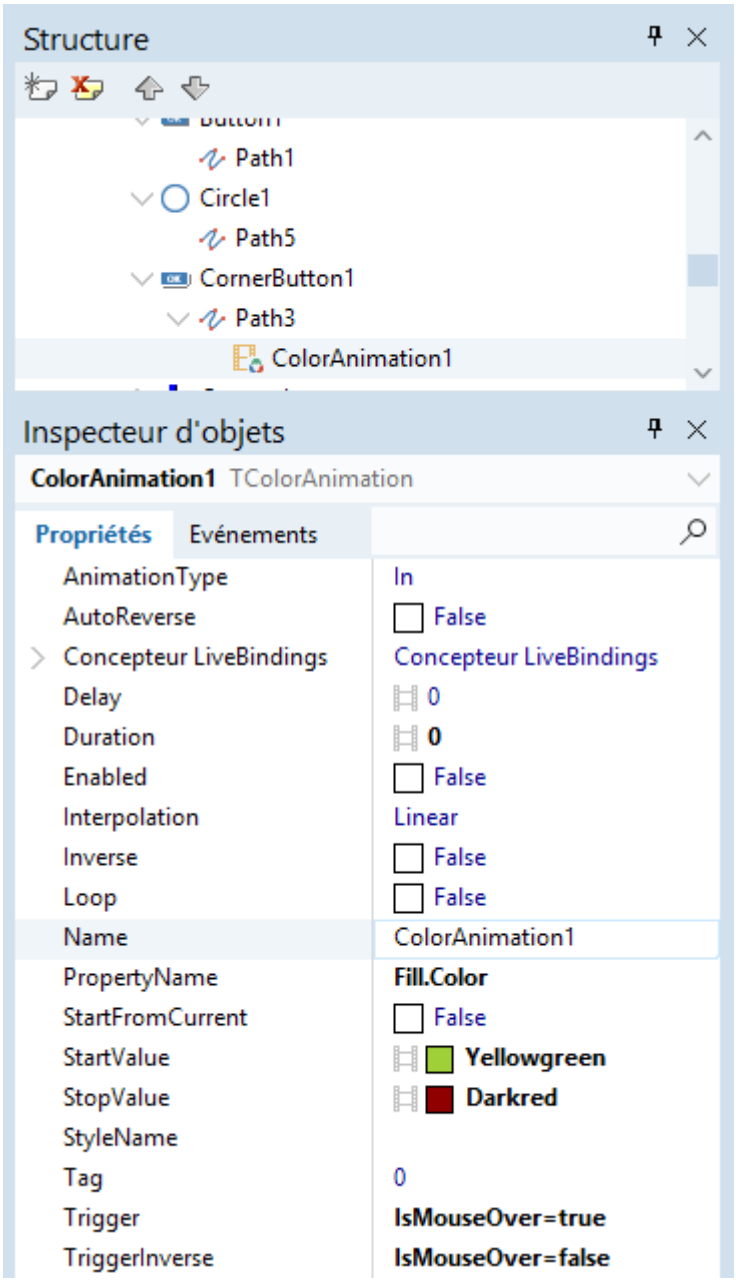
En mode design utilisez les propriétés Locked pour pouvoir sélectionner le conteneur plutôt que le contenu
N'oubliez pas la propriété HitTest :
- positionnez vous sur le robot vert, il change de couleur, cliquez,
- positionnez vous en dehors des contours il redevient vert, cliquez.
Agrandissement , *sur le seul cercle
La propiété Align du robot va jouer
Client : le robot va finir par déborder du cercle
Center : la taille du robot ne change pas
...

En soit, rien que l'on n'aurait pu faire avec des **TGlyph** ou **TImage** et en y associant un **TImageList**.

Deux petits tests vont changer cette impression mais, tout d'abord, une petite astuce qui s'applique à tout composant enfant d'un autre : la propriété **Locked**. Découverte, ou plutôt redécouverte, quand j'ai voulu programmer des évènements d'un bouton. En mode design, chaque fois que je sélectionnais le bouton, c'était le **TPath** enfant qui était en fait sélectionné, il fallait utiliser la touche Escape pour obtenir les propriétés du bouton. Mettre la propriété du composant enfant ***Locked*** à ***true*** règle cet inconvénient.

III-A-1 - La zone de TPath

Premier test, « jouer » avec les couleurs du petit Androïd vert. Je me suis posé la question : « Est-il possible d'ajouter des effets à un **TPath** ? » Pour ce faire j'ai ajouté au **TPath** un **TColorAnimation**.



Le résultat confirme mes souhaits, dès que la souris entre dans le **TPath** le petit robot change de couleur et ce sans avoir écrit une seule ligne de code, dois-je le préciser ?

*Dans le **TPath** ? En fait pas exactement il serait plus juste d'écrire dans la surface délimitée par les données de celui-ci, soit la partie colorée. Dès que la souris se retrouve dans un espace elle n'est plus considérée comme dans la zone.*

Cette notification se confirme en codant deux évènements **OnClick**, le premier sur le bouton le second sur l'image.

```
procedure TMainform.CornerButton1Click(Sender: TObject);
/// test souris et click
begin
Showmessage('click '+TComponent(sender).name);
end;

procedure TMainform.Path3Click(Sender: Tobject);
```

```
/// Path3 est enfant de CornerButton1
/// test souris et click
/// si la souris se trouve à l'intérieur de la zone
/// c'est cet évènement qui est levé
begin
ShowMessage('Android touché');
end;
```

Tout de suite cette particularité m'a fait penser à des jeux d'arcades comme « Space Invaders » ou autres jeux d'arcades avec des sprites ! Et vous ?

III-A-2 - Dessin dans un cercle

Deuxième expérience que je voulais réaliser, un bouton rond, mode qui fleurit sur les réseaux sociaux. Rien de plus facile que de poser un **TCircle** et d'y ajouter un composant **TPath**. Plus délicat est de choisir comment aligner ce composant à l'intérieur du cercle.

Tant que l'on n'a pas à modifier la taille du cercle jouer sur les propriétés **Padding** du cercle ou les propriétés **Margins** du **TPath** cela fonctionne. Cependant j'ai aussi voulu tester des tailles variées, que ce soit au design ou par code (boutons + et – de l'interface). Force est de constater qu'il faut réajuster ces marges à chaque fois.

J'ai donc coder l'évènement **OnResize** du composant **TCircle** afin de toujours obtenir mon image inscrite dans le cercle.

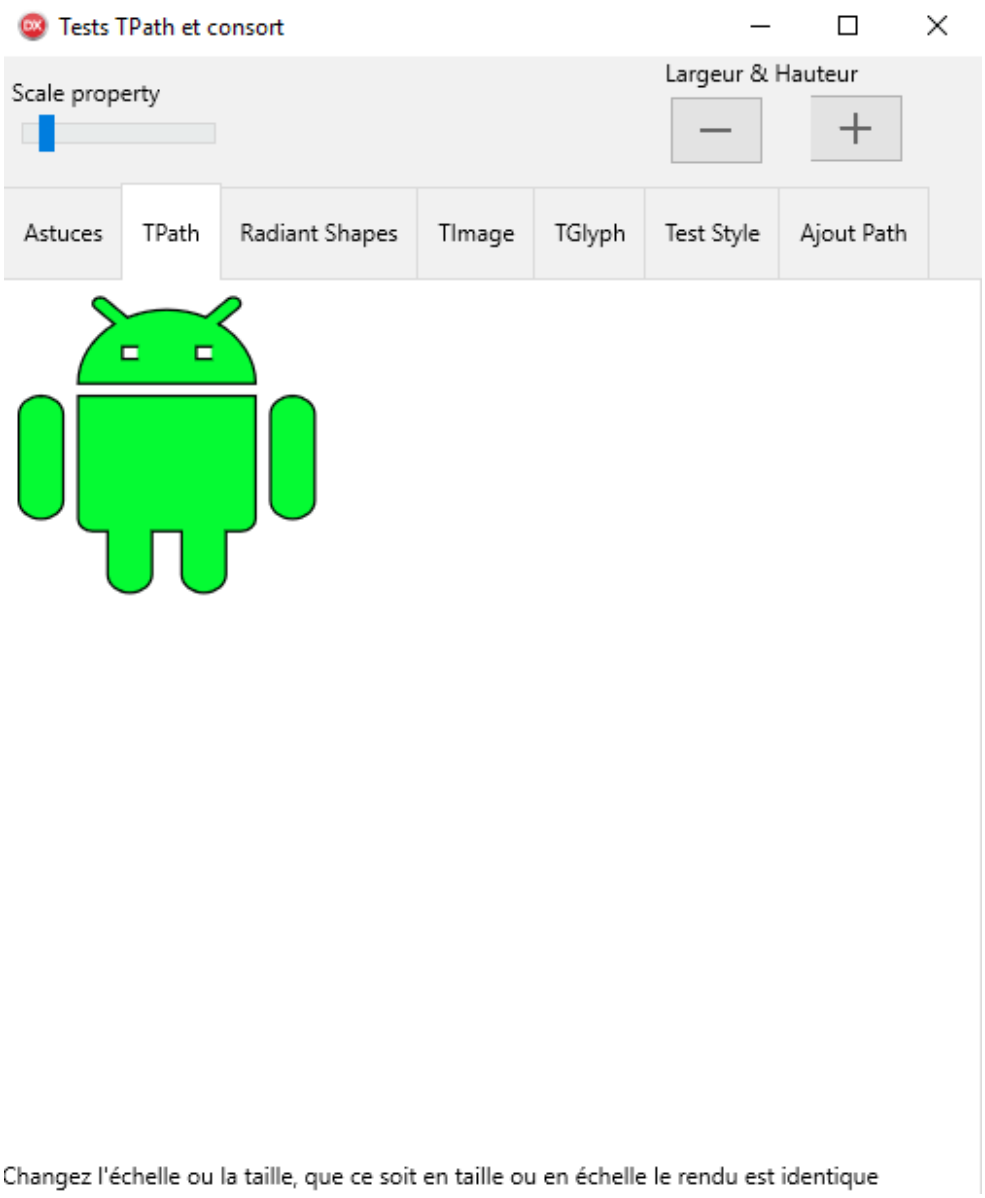
```
procedure TMainform.Circle1Resize(Sender: TObject);
/// Changer les marges pour inscrire le robot dans le cercle
///  si l'on considère que le robot (TPath) est un carré
///  le côté d'un carré a est égal à d/√2
var a,p : single;
begin
// Exit ; // désactive l'évènement
a:=Circle1.Width/sqrt(2);
p:=(Circle1.width-a)/2;   // calcul de la marge (padding)
Circle1.Padding.Bottom:=p;
Circle1.Padding.Left:=p;
Circle1.Padding.Right:=p;
Circle1.Padding.Top:=p;
end;
```

III-B - Les avantages de TPath

Il est temps de tester en quoi **TPath** peut être intéressant. J'ai fait plusieurs tests différents, d'où les nombreux onglets et même une unité séparée.

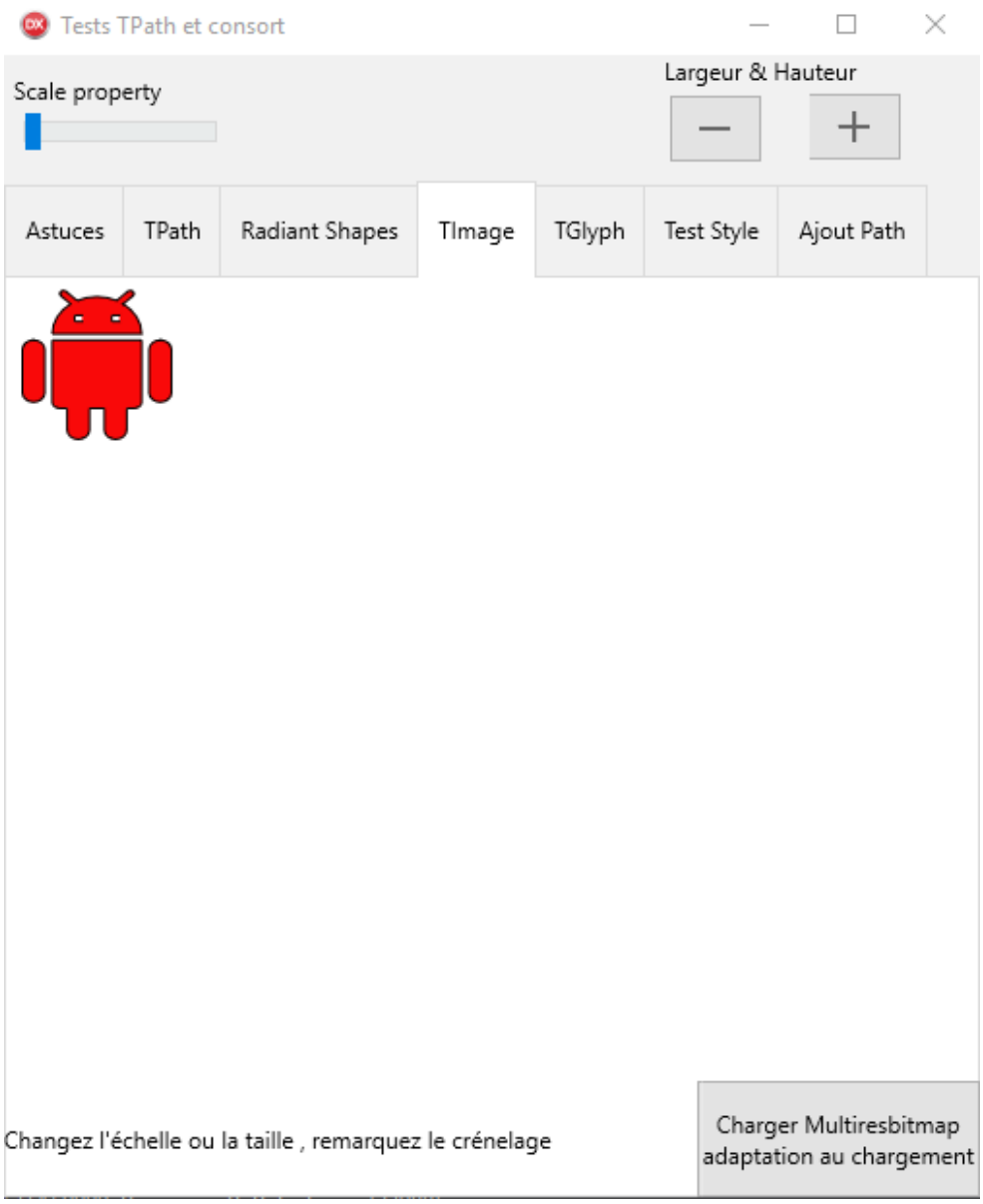
III-B-1 - Le crénelage

Trois onglets (**TPath**, **TImage**, **TGlyph**) pour comprendre ce que changer la taille du composant, l'échelle ou la résolution d'un écran peuvent faire. Pour les deux premiers j'ai fait jouer à la fois la taille du composant (boutons + et -) mais aussi son échelle (curseur).

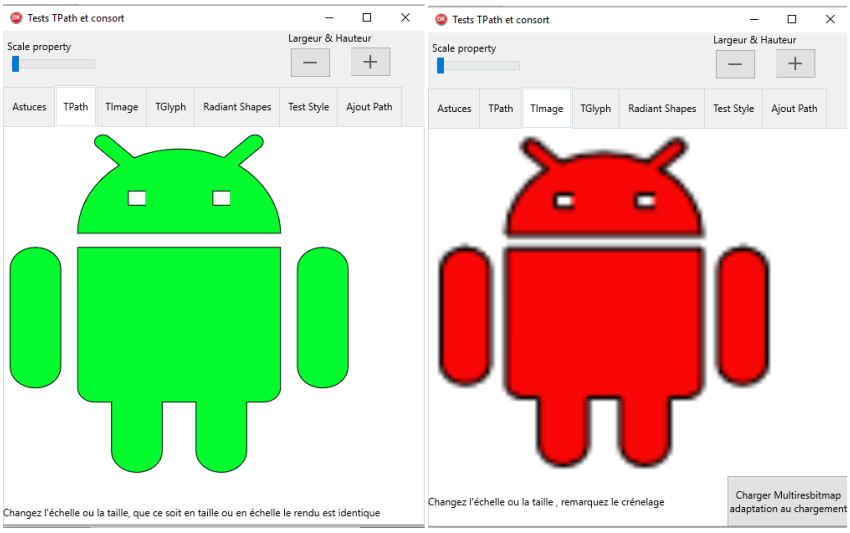


Quelque soit les valeurs utilisées l'image restera toujours impeccable.

Par contre si l'on utilise un **TImage**, dès que les valeurs changent un effet de crénelage disgracieux apparaît.



Comparatif, tailles identiques

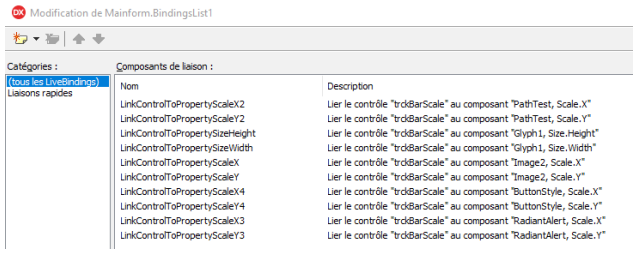


Effet de crénelage sur TImage

III-B-2 - Échelle contre Taille

Sur l'onglet **TPath** je peux également voir la différence entre changer la taille (hauteur et largeur) du composant et l'échelle.

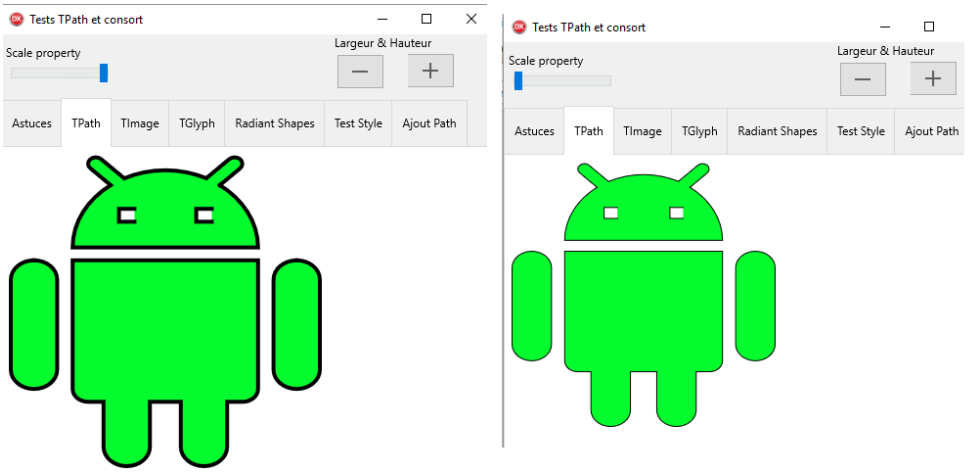
Pour le changement d'échelle, j'ai préféré passer par les LiveBindings, réduisant ainsi le code à une portion congrue se contentant de redessiner l'écran.



Liaisons avec le TTrackBar

```
procedure TMainform.trckBarScaleChange(Sender: TObject);
begin
// rafraichir l'écran
BeginUpdate;
EndUpdate ;
end ;
```

La différence entre les deux processus est minime en augmentant l'échelle le contour s'épaissit contrairement à ce qui se passe en changeant de taille.



(à gauche un changement d'échelle, à droite un agrandissement)

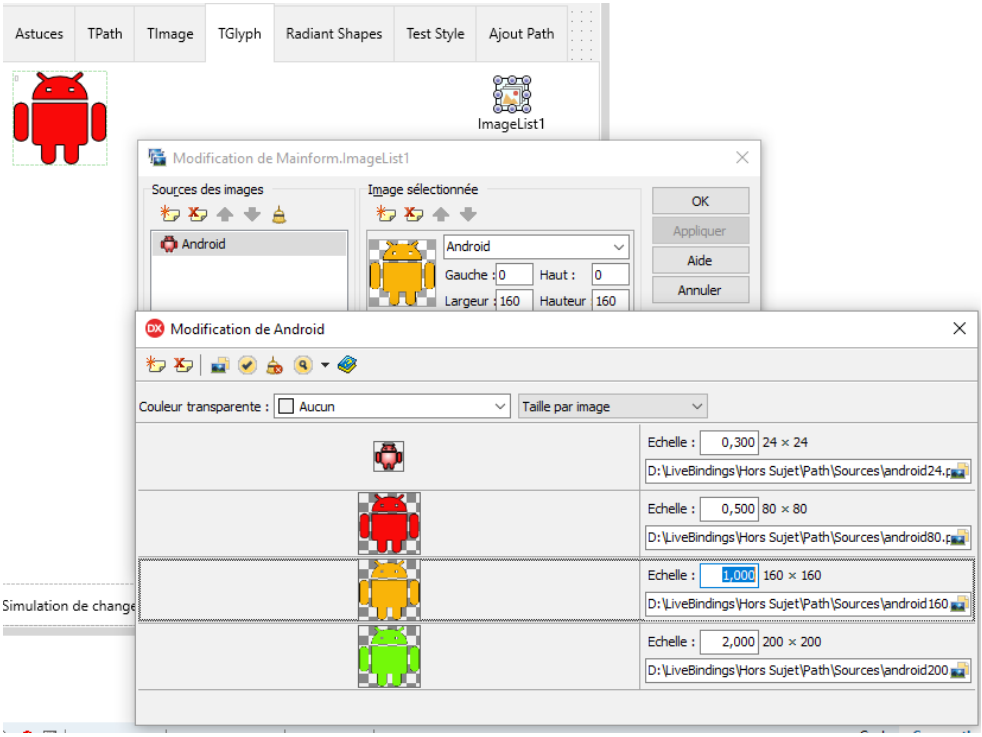
III-B-3 - MultiresBitmap pour les nuls

Constatations faites des problèmes de crénelage, j'ai voulu en apprendre un peu plus sur ce que le **MultiresBitmap** apportait. J'ai d'abord commencé ce travail sur le **TGlyph**, la raison en étant qu'avant la version Tokyo seul le **TGlyph** pouvais se lier à une liste d'images.

Complètement néophyte en ce qui concerne cette fonctionnalité, je croyais que seul un changement de résolution d'écran pouvait démontrer l'utilité de **MultiresBitmap** jusqu'à ce que je me souvienne d'une portion de vidéo d'Andrea

Magni qui en faisait une démonstration lors du « Delphi CE BootCamp 2018 (1) : GUI Fundamentals (FMX and VCL) part 3 Common Controls » (à environ 40 minutes de celle-ci).

Remplir le **TImagelist** m'a donc grandement été facilité par mon premier programme qui m'a permis de sauvegarder les images, en différentes tailles et couleurs pour les distinguer.



Diminuer ou augmenter la taille de l'image forcera le programme à changer d'image pour celle de résolution la plus proche.

Quelques remarques :

- À l'usage il est à remarquer que le passage d'un taille à l'autre peut produire un effet saccadé ;
- **TGlyph** ne propose pas d'accès direct à la propriété **Scale** pourtant il m'a été possible de l'atteindre via Livebindings ;
- À l'exécution sur mon mobile Androïd le robot est jaune et non rouge, preuve que l'écran de ce dernier à un dpi plus important que mon écran Windows.

III-B-4 - Chargement d'un MultiResBitmap à partir d'une liste

Une fois que la liste d'image faite il m'a semblé intéressant de l'utiliser pour le **TImage** déposé dans l'onglet précédent. À cet effet j'ai rajouté un bouton qui me permet de le faire avec le code suivant :

```
procedure TMainform.btnloadClick(Sender: TObject);
/// Charge les différentes images déjà contenue dans le composant ImageList
begin
Image2.MultiResBitmap:=TFixedMultiResBitmap(ImageList1.Source[0].MultiResBitmap);
end;
```

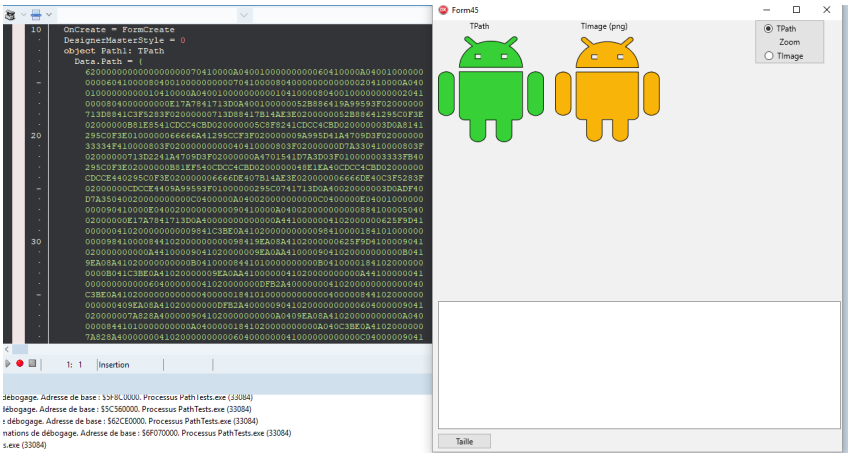
En l'utilisant si l'image change bien de couleur (du rouge au jaune) l'agrandir ensuite n'a pas le même effet que pour **TGlyph** , aucun changement de couleur ne se fait, c'est la seconde différence trouvée en changement d'échelle et agrandissement, en largeur et hauteur, de l'image.

III-B-5 - Économie d'octets

Je sais que l'économie sur les octets n'est plus tellement à la mode, toutefois je voulais quand même savoir ce qu'il en était, bien que cela paraisse évident dès que l'on utilise plusieurs images pour régler les problèmes de dpi, qu'en était-il entre une image et un **TPath** de même taille à l'écran ?

Pour ce faire, j'ai préféré créer une unité différente afin de retrouver facilement dans le texte du fichier **fmx** les données des deux composants.

Pour tester cette partie, intervertissez l'ordre de créations des fiches du projet.



Utilisation en mode debug

Exécuter le programme ainsi créé en mode debug permet de récupérer le texte des données (**Data.Path** pour **TPath**, **PNG** pour **TImage**). Il me suffit ensuite de coller ces données dans le **TMemo** puis de cliquer sur le bouton pour obtenir la taille.

	TPath.Data.Path	**TImage.MultiresBitmap PNG**
Nombre d'octets	2648	12567

Résultat flagrant, avantage à **TPath** presque cinq fois moindre, sans compter le meilleur rendu en fonction des tailles ou dpi et la facilité de changement de couleurs !

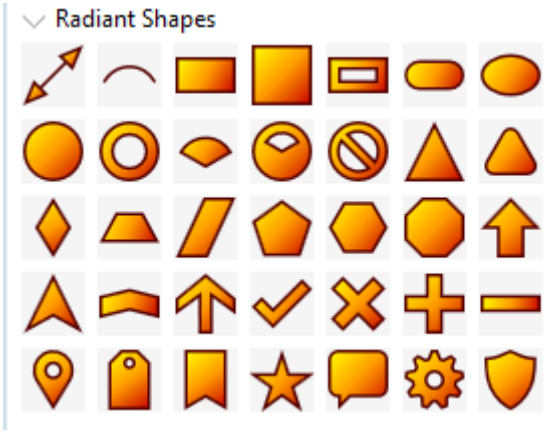
Où se trouve le loup ? À mon avis il est certainement caché dans l'utilisation du CPU puisqu'il y a calculs.

III-C - Utilisation des Radiant Shapes

*Vous pouvez très bien sautez ce chapitre. Pour les tests suivants le paquet de composants **Raize Radiant Shapes** doit être installé.*

Pourquoi ajouter cette étude ? En fait je tentais de créer plusieurs formes simples (triangles, rectangles, cercles, rectangles aux coins arrondis) en saisissant des commandes SVG. Si les premières ne m'ont pas posées trop de soucis, le rectangle aux coins arrondis par contre était loin d'être une réussite et débordait très nettement de mes

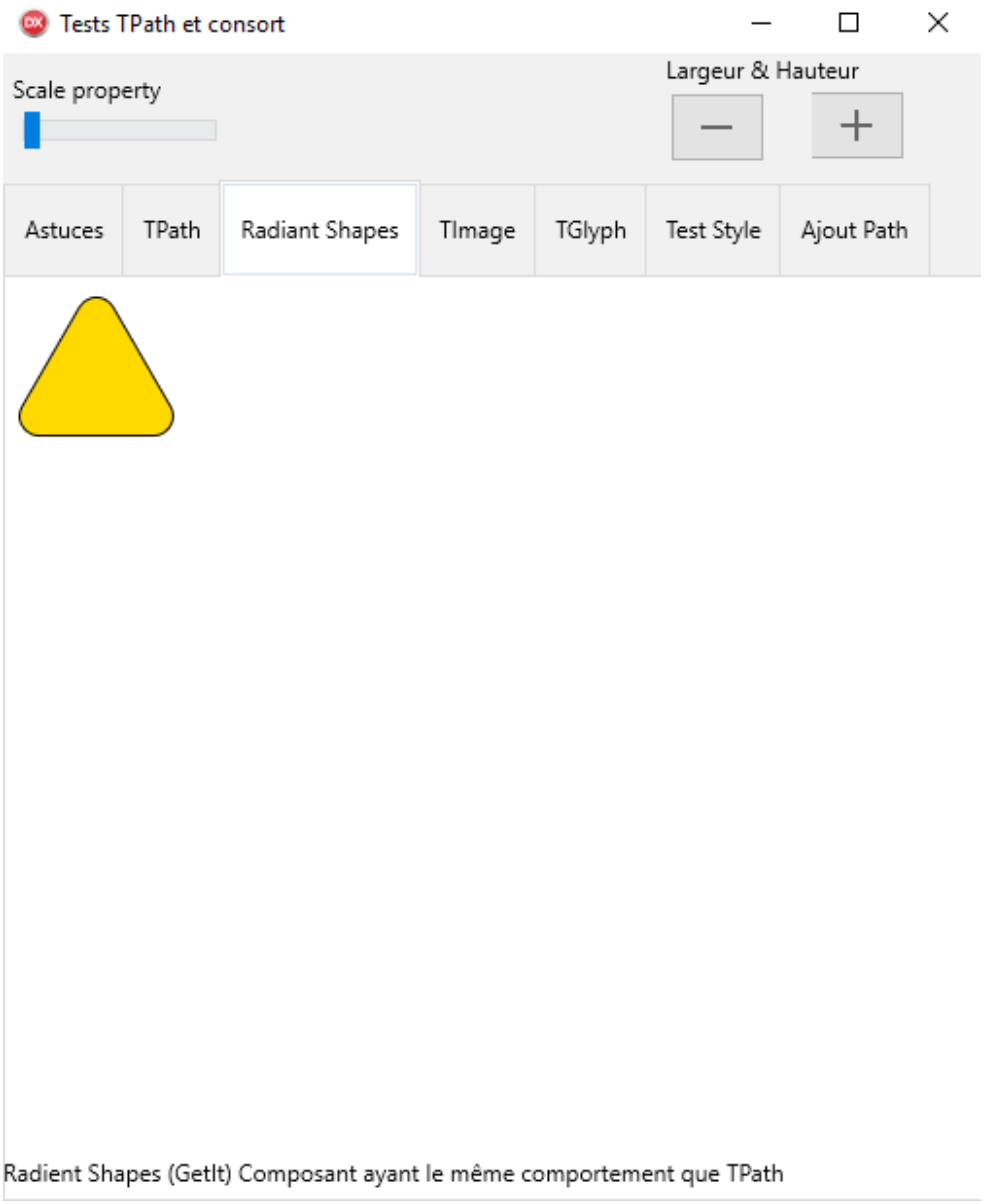
connaissances. C'est alors que je me suis souvenu de ces composants et me suis posé la question : « Utilisent-ils des **TPath**s ? » J'ai donc procédé à l'installation de ce paquet de composants, *via* **Getit** c'est extrêmement facile.



Palette des composants Radiant Shapes

III-C-1 - Essais d'utilisation

Un onglet de plus au niveau du design, la pose d'un composant et coder les mêmes fonctionnalités que pour le **TPath** ou le **TImage** n'a pris que peu de temps.



Le résultat est remarquable, comparable au **TPath**. Fouiller dans les sources, puisque fournies, m'a permis de confirmer que Ray Konopka utilise bien les primitives d'un **TPath** mais pas d'une utilisation de chemins de données comme je le faisais jusqu'à présent. Dommage, ce n'est pas là que je trouverais les commandes **SVG** de mon rectangle aux coins arrondis !

*Si vous voulez plus d'informations sur comment l'auteur de ces composants procède (et si vous êtes anglophones) je vous invite à regarder **les deux vidéos de ce lien**.*

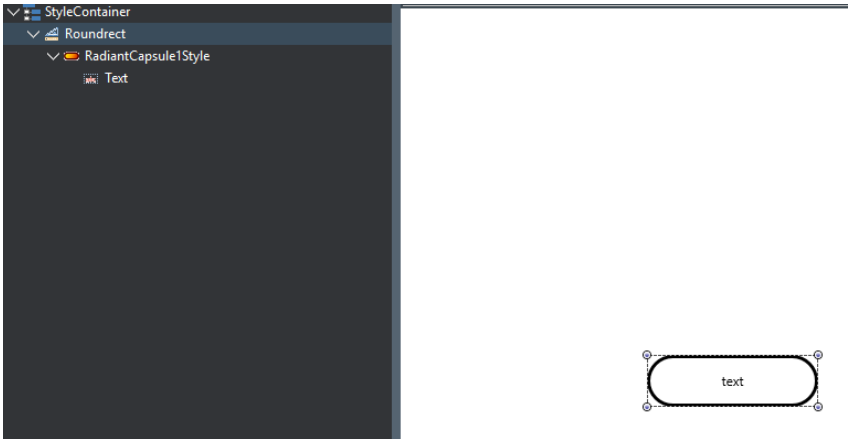
III-C-2 - Objectif de cette recherche

Pourquoi avoir voulu créer un rectangle aux coins arrondis ? En fait cette envie découle d'une idée : il est possible d'utiliser un **TPath** dans un élément de style (genèse de tout cet article). Un membre du forum ayant constaté que pour tout ce qui était style un rectangle « classique » avait des problèmes de crénelage une fois la solution déployée sous Androïd je voulais voir ce que cela pourrait donner.

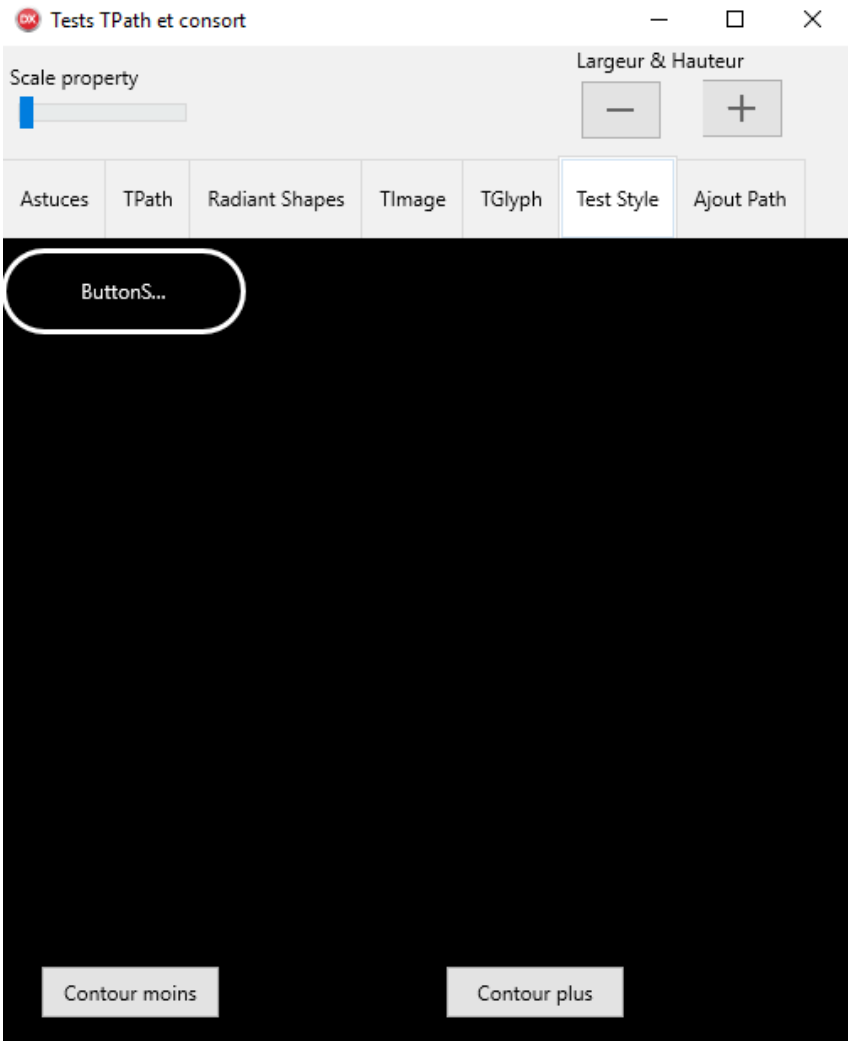
*Les échanges se trouvent dans cette **discussion**.*

Pour résumer le problème une petite image, empruntée, résume la situation.

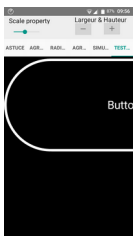
Si, malheureusement je n'ai pas réussi à créer un **TPath** acceptable, l'utilisation d'un **TRadiantCapsule** dans un style donne un rendu impeccable.



Création du style



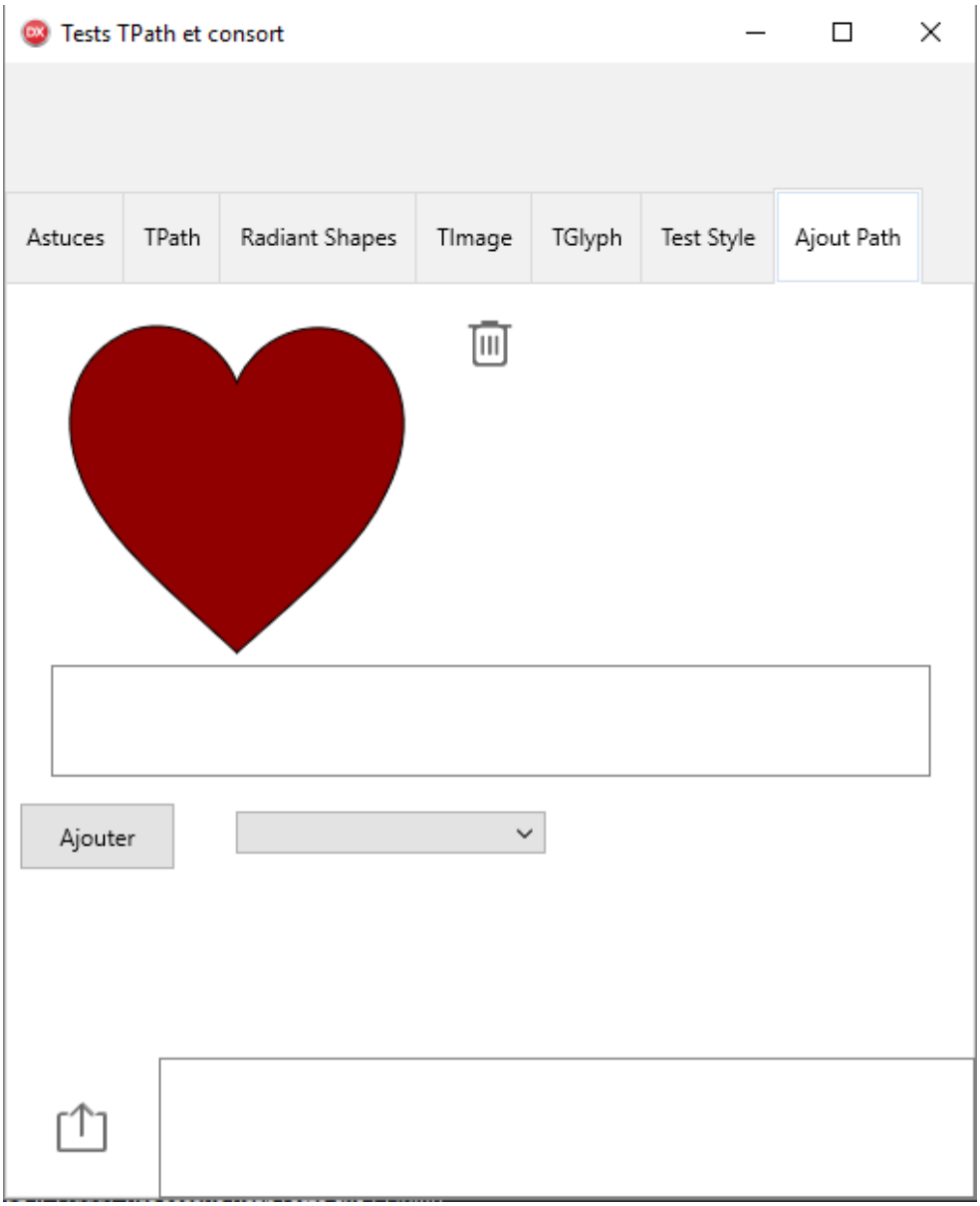
Utilisation au design



Test sous Androïd

III-D - Diverses manipulations sur le PathData

Dans le programme du chapitre **II**, je donnais déjà la possibilité de dessiner en utilisant le mémo pour saisir des commandes SVG. J'ai simplement voulu aller un peu plus loin, était-il possible de fusionner deux dessins chargés ? De même, il y avait encore une propriété de ***Data*** non étudiée : ***StyleLookup***



III-D-1 - Fusion de dessins

L'idée était de tester une hypothèse, était-il possible de fusionner deux dessins à l'exécution et ce sans passer par la case saisie ? Pour réaliser cela j'ai créé une page contenant deux **TPath**s, le premier contiendra l'image résultat, le second l'image à ajouter.

Bien sûr, il eut été facile de faire une simple concaténation de chaînes mais j'ai voulu explorer un peu la classe **TPathData**.

Le principe : le mémo en bas de page sert à saisir un chemin, le bouton situé à gauche de celui-ci permettant d'afficher dans le **TPath** source le résultat de la saisie. Le bouton **[Ajouter]** enclenche l'action que je cherche à étudier.

```
procedure TMainform.btnAddPathClick(Sender: TObject);
/// Ajoute ce qui est dans la miniature (PathtoAdd)
/// au composant PathToDraw
begin
PathToDraw.Data.AddPath(PathToAdd.Data);
MemoResult.Text:=PathtoDraw.data.Data;
end;
```

*Hyper simple à réaliser, mais ayez la curiosité d'aller voir **toutes les méthodes** proposées. Cela ne vous rappelle pas un peu, et en mieux, l'utilisation de **Canvas** pour dessiner ?*

Piste non explorée mais prometteuse :

*Quand je pense que je saisissais le chemin d'un rectangle alors qu'une méthode **AddRectangle** existe ! Voilà une piste que je n'ai pas explorée et qui aurait peut-être répondu à mes tentatives de dessin avant d'utiliser les Radiant Shapes du chapitre **III.C.2**.*

L'ajout de chemin étant aisée, je me suis ensuite proposé d'avoir au sein de mon programme des chemins tout prêts, que je pouvais rajouter directement. C'est le but de la boite de choix ajoutée. Cette boîte, à ce stade, ne contenait que deux options : un cœur et un rectangle.

J'avoue m'être fourvoyé, mon objectif était de tester la propriété **resource** que j'avais entre-aperçue d'où mon utilisation de chaînes de ressources.

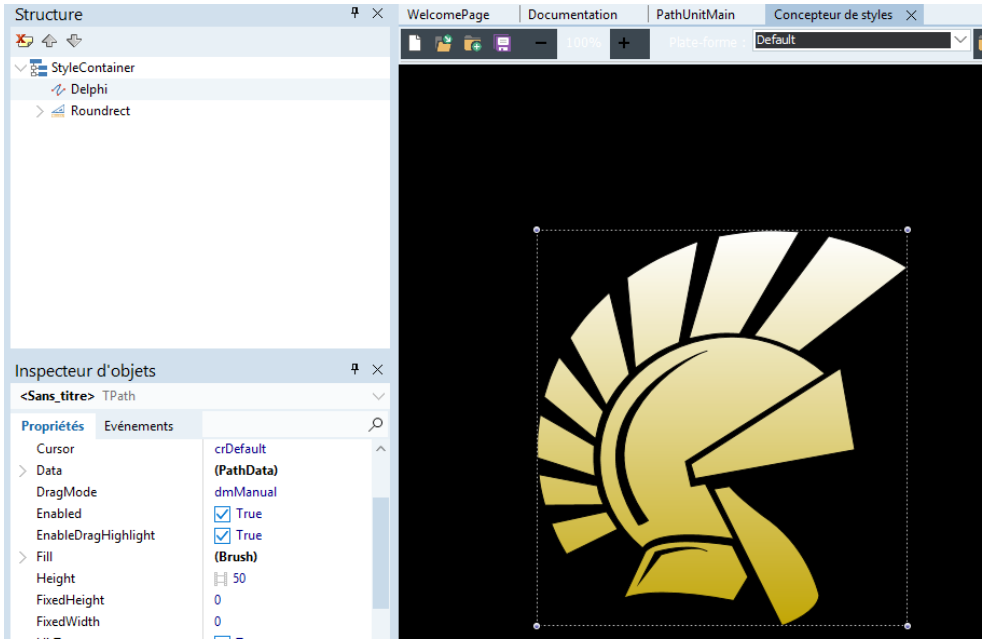
```
resourcestring
  StrCoeur = 'M 213.1,6.7 c -32.4-14.4-73.7,0-88.1,30.6 C 110.6,4.9,67.5-9.5' +
             ',36.9,6.7 C 2.8,22.9-13.4,62.4,13.5,110.9 C 33.3,145.1,67.5,170.3,125,217 '+
             'c 59.3-46.7,93.5-71.9,111.5-106.1 C 263.4,64.2,247.2,22.9,213.1,6.7 z';
  StrRectangle = 'm0 1 H10 v10 h-10 v-10 Z';
```

Sauf que, cette propriété est en fait pour le remplissage (**fill**) ou le contour (**stroke**) et qu'il s'agit d'un **TBrushResource**, grosse confusion et cela n'a donc rien apporté de nouveau. Ce **TBrushResource** restera un élément à explorer.

III-D-2 - Utilisation de StyleLookup

L'utilisation de cette propriété m'intéressait fortement dans le cadre d'autres projets, pour ne citer que celui déclencheur de cette recherche l'indication du tri au sein d'un entête de grille.

J'ai donc extrait le casque d'hoplite (2) stylisé **fourni par Jim McKeith** pour l'insérer dans un élément de style.



Ce casque, je l'ai même mis en chaîne de ressource mais vous pourrez constater que la chaîne de données est extrêmement longue. À tel point que, débordant des tailles autorisées, j'ai été obligé de la scinder en deux.

J'ai ensuite ajouté une option (index 3) dans ma boite de choix afin de pouvoir affecter la style au **TPath**.

```
procedure TMainform.ComboBox1ClosePopup(Sender: TObject);
/// Utilisation de TPathData ("TPath invisible")
/// pour charger des données de dessin
var APathData : TPathData;
begin
APathData:=TPathData.Create;
Case ComboBox1.ItemIndex of
  0 : APathData.Data:=StrCoeur;        // ressource
  1 : APathData.Data:=strRectangle;    // ressource
  2 : APathdata.data:=strCasquea+strCasqueb; // ressource
  3 : APathData.StyleLookup:='Delphi'; // style
End;
if ComboBox1.ItemIndex>-1 then
 begin
  PathToDraw.Data.AddPath(APathData);    // ajout du TPathData
  MemoResult.Text:=PathtoDraw.data.Data;
 end;
end;
```

Et cela fonctionne, mais pas tout à fait comme espéré :

1 Le casque ne s'inscrit pas dans le cœur mais écrase celui-ci#;
2 La couleur que j'avais mise dans le style (d'un joli or) n'est pas utilisée ;
3 Impossible d'effacer le style ensuite ;

Force est donc de constater que les propriétés **Data.Data** et **Data.StyleLookup** seraient exclusives et qu'une fois utilisé **stylelookup** impossible de s'en débarrasser (3) .

III-E - Bilan

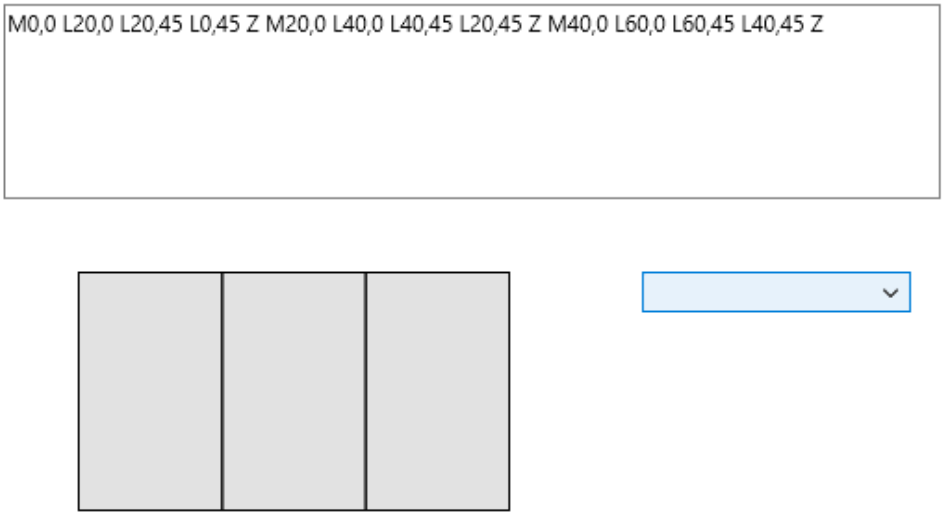
De ces essais, il faut retenir que **TPath** est une alternative efficace aux problèmes que pourrait apporter des changements de résolution surtout que le gain en octets est loin d'être négligeable.

Vous retrouverez les sources de ce chapitre dans l'archive PathTests.zip

Attention, le contenu est sujet à différences entre le code et les extraits dans ce document.

IV - Pour aller plus loin

Lors du bilan du chapitre **II.F** j'indiquais que **TPath** était « mono-texture » plutôt que mono-couleur c'est vrai, il n'empêche qu'il est possible de jouer sur les dégradés. Ainsi il est possible d'obtenir le drapeau français de cette manière :

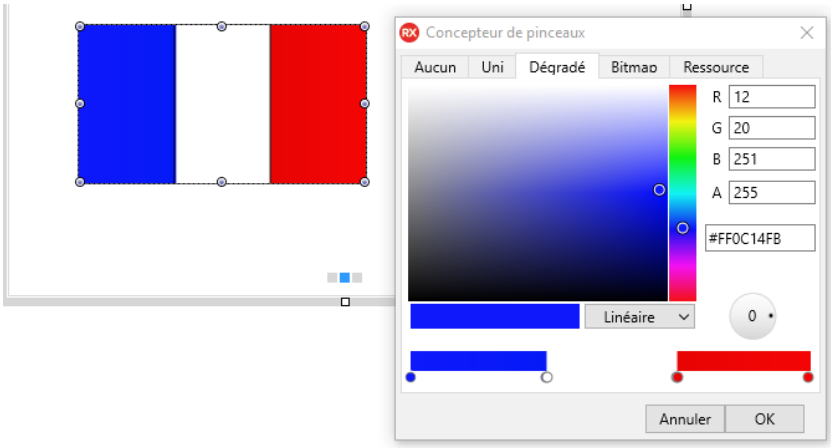


J'utilise ici un seul dessin et ce pour voir les contours des divers rectangles.

J'aurais tout aussi bien pu ne faire qu'un seul rectangle, les barres de délimitations étant inutiles.

Notez aussi que le nombre de points pour les dégradés est limité à 8 pour une application Androïd.

Et en manipulant le dégradé de remplissage (avec un peu de patience).



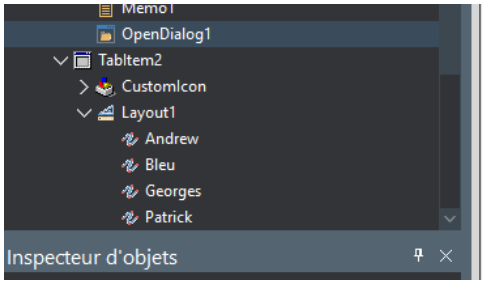
Dégradé manuel

Tous les drapeaux à bandes verticales égales sont donc possibles à obtenir.

```
// Technique du gradient
var AGradient : TGradient;
    Gpoint : TCollectionItem;
begin
AGradient:=TGradient.Create;
try
// Angle 0°
AGradient.StartPosition.X:=0;
AGradient.StartPosition.Y:=0.5;
AGradient.StopPosition.X:=1;
AGradient.StopPosition.Y:=0.5;

case Pays.ItemIndex of
  1 : begin  // France
        AGradient.Color:=TAlphaColors.Blue;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=1/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Blue;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(1/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.White;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=2/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.White;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(2/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Red;
        AGradient.Color1:=TAlphaColors.Red;
      end;
  2 : begin  // Belgique
        AGradient.Color:=TAlphaColors.Black;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=1/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Black;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(1/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Yellow;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=2/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Yellow;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(2/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Red;
        AGradient.Color1:=TAlphaColors.Red;
      end;
  3 : begin // Italie
        AGradient.Color:=TAlphaColors.Green;
        GPoint:=AGradient.Points.Add;
```

```
        TGradientPoint(GPoint).Offset:=1/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Green;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(1/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.White;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=2/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.White;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(2/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Red;
        AGradient.Color1:=TAlphaColors.Red;
      end;
  4 : begin  // Roumanie Bleu plus clair
        AGradient.Color:=TAlphaColor($FF0C90FB);
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=1/3;
        TGradientPoint(GPoint).Color:=TAlphaColor($FF0C90FB);
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(1/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Yellow;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=2/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Yellow;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(2/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Red;
        AGradient.Color1:=TAlphaColors.Red;
      end;
  5 : begin    // Tchad
        AGradient.Color:=TAlphaColors.blue;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=1/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Blue;
        GPoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(1/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Yellow;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=2/3;
        TGradientPoint(GPoint).Color:=TAlphaColors.Yellow;
        Gpoint:=AGradient.Points.Add;
        TGradientPoint(GPoint).Offset:=(2/3)+0.001;
        TGradientPoint(GPoint).Color:=TAlphaColors.Red;
        AGradient.Color1:=TAlphaColors.Red;
      end;

  else begin
         AGradient:=nil;
       end;
end;
if AGradient=nil
   then Path2.Fill.Kind:=TBrushKind.Solid
   else begin
     Path2.Fill.Kind:=TBrushKind.Gradient;
     Path2.Fill.Gradient:=AGradient;
   end;
finally
  AGradient.Free;
end;
```

Une autre solution serait de mettre, dans un même conteneur, plusieurs **TPath**s en fonction de la couleur de remplissage (attribut **fill** ou **style**). Pour l'Union Jack c'est cette technique que j'ai testée.



Des améliorations dans la routine de lecture des fichiers SVG pourraient certainement apporté cela, les indications de couleurs étant notifiées par un attribut.

En parlant d'attributs, un certain nombre de fichiers SVG contiennent d'autres directives telle que **transform** dont j'ai déjà parlé, mais aussi d'autres types de nœuds comme **rect** découvert lors de mes recherches pour le drapeau français.

france
```
   <rect
       style="fill:#000080;fill-opacity:1;stroke:#000000;stroke-width:0.26499999;stroke-
miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
       id="rect4504"
       width="20"
       height="45"
       x="0"
       y="0" />
    <rect
       style="fill:#f2f2f2;fill-opacity:1;stroke:#000000;stroke-width:0.26499999;stroke-
miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
       id="rect4506"
       width="20"
       height="45"
       x="20"
       y="0" />
    <rect
       style="fill:#ff0000;fill-opacity:1;stroke:#000000;stroke-width:0.26499999;stroke-
miterlimit:4;stroke-dasharray:none;stroke-opacity:1"
       id="rect4508"
       width="20"
       height="45"
       x="40"
       y="0" />
```

Pas de **path** dans ce SVG pour obtenir les chemins je suis passé par un code

```
procedure TFormProspective.Drapeau3Click(Sender: TObject);
var ARect : TRectF;
begin
Path1.Data.Clear;
ARect:=TRectF.Create(TPointF.Create(0,0));
ARect.Width:=20;
ARect.Height:=45;
Path1.Data.AddRectangle(ARect,0,0,[],TCornerType.Round);
ARect.TopLeft:=TPointF.Create(20,0);
ARect.Width:=20;
ARect.Height:=45;
Path1.Data.AddRectangle(ARect,0,0,[],TCornerType.Round);
ARect.TopLeft:=TPointF.Create(40,0);
ARect.Width:=20;
ARect.Height:=45;
Path1.Data.AddRectangle(ARect,0,0,[],TCornerType.Round);
MemoData.Lines.Text:=Path1.Data.Data;
end;
```

Il serait certainement possible de gérer ce type de structure dans la routine de chargement d'un fichier SVG.

Ces essais se retrouvent dans cette archive (prospectives.zip)

V - Conclusion et remerciements

Si vous avez lu **le tutoriel à l'intention des développeurs Mozilla** vous aurez vite compris que **TPath** était loin d'offrir quelque chose de complet par rapport à ce que peut offrir le format SVG. Néanmoins, est-il permis de rêver et de voir un jour les **TPath**s comme une alternative aux images que nous utilisons pour nos interfaces ? Je l'espère. Dans le même ordre d'idée je verrais bien les images ressources des styles remplacées elles-aussi.

Pour ceux qui veulent rester VCL je ne peux qu'espérer que ce composant fera son apparition dans les prochaines versions, néanmoins sachez qu'il existe des projets open-source sur **GitHub**.

J'ai également lu, au cours de mes recherches, que **TPath** pouvait être une alternative intéressante pour dresser des camemberts et autres figures proposées par **TChart** et il y a certainement encore beaucoup d'utilisations auxquelles je n'ai pas pensé.

J'espère en tout cas vous avoir mis l'eau à la bouche et que le partage de mes explorations vous sera profitable.

Une petite mise en garde amicale s'impose. Triturer ce composant comme j'ai pu le faire est aussi addictif que peuvent l'être les composants 3D.

Je remercie l'équipe de **www.developpez.net** pour l'hébergement accordé à mes articles et l'implication des correcteurs techniques xxxxx et grammaticaux xxxx

VI - Notes de mise à jour

Depuis mes tests (juin 2019) de l'eau a coulé sous les ponts.

La version 10.4.1 est devenue la version en cours et l'offre de composants tiers concernant les SVG s'est étoffée, composants qui, il faut le noter, sont aussi bien pour VCL que FMX.

Je n'en retiendrai que celui proposé via Getit : **SVGIconImageList VCL & FMX**disponible sur GitHub dans ce **dépôt**, compatible depuis XE4 (apparition de **TPath**). Mon banc d'essais à base de drapeaux (4) , dont certains très complexes, me fait écrire que ce paquet est un des plus aboutis, je le recommande. Pour en savoir un peu plus lisez ce **billet**.

De mon côté, malgré divers bogues constatés (et signalés) j'ai plus ou moins réussi en n'utilisant que **TPath** à faire un rendu de fichiers au format SVG sous FMX qui pourrait faire l'objet d'un nouveau tutoriel sur le sujet, bien que l'apparition de composants « tout fait » rende la chose obsolète, démonter les mécanismes reste quand même formateur.

1 : Delphi CE Bootcamp 2018 est accessible gratuitement, après inscription, via **Embarcadero Academy**
2 : Hoplite, un soldat corinthien m'évitant ainsi de choisir entre athéniens et spartiates.
3 : Du moins est-ce le cas avec la version 10.3.2
4 : SVG téléchargeables depuis Wikipedia