# Concevoir ses propres composants

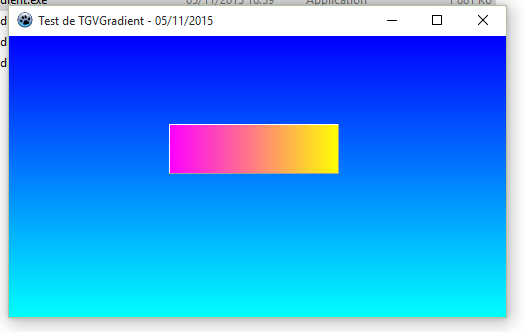
**Objectifs** : dans ce chapitre, vous allez apprendre à concevoir des composants capables d’enrichir la palette de ceux fournis pas **Lazarus**.

**Sommaire :** Un composant visuel de A à Z : ***TGVGradient*** – Un composant non visuel : ***TGVSizeMover***

**Ressources** : les programmes de test sont présents dans le sous-répertoire *composants* du répertoire e*xemples*.

## Un composant visuel de A à Z : TGVGradient

Le premier composant à concevoir sera simple, mais bâti à partir du minimum exigible pour un composant graphique : il descendra de ***TGraphicControl***. Son objectif est de proposer un fond en dégradé pour un contrôle fenêtré. L’utilisateur devra pouvoir contrôler les couleurs de début et de fin, ainsi que l’orientation horizontale ou verticale du dégradé.



La principale difficulté pourrait être de dessiner le dégradé lui-même, mais ce serait sans compter sur les outils déjà fournis par **Lazarus**. En l’occurence, la propriété *canvas* des contrôles graphiques propose d’une méthode *GradientFill* tout à fait adaptée à notre propos puisqu’elle remplit un rectangle avec des couleurs et une direction à fournir !

Le développement de composants est grandement facilité par une connaissance étendue des unités et paquets déjà fournis par **Free Pascal** et **Lazarus**. À condition que les outils proposés soient efficaces et fiables, il ne s’agit pas de perdre de l’énergie à reproduire ce qui existe déjà, surtout s’il s’agit du cœur du travail.

**[Exemple Comp\_04]**

La syntaxe de cette procédure publique attachée à la classe ***TCanvas*** est :

TCanvas.GradientFill(ARect : TRect ; AStart : TColor; AStop: TColor; ADIrection: TGradientDirection);

Dans l’unité *Graphics*, le type ***TGradientDirection*** est défini ainsi :

**type**

TGradientDirection = (gdVertical, gdHorizontal) ;

Les paramètres sont d’une compréhension immédiate et correspondent exactement au cahier des charges du composant à créer. Les deux seules difficultés tiennent à l’affichage.

Tout d’abord, le composant aura besoin de connaître son propriétaire et son parent pour pouvoir s’afficher et devra s’adapter à la surface de ce dernier. Ce travail sera effectué lors de la création composant :

**constructor** TGVGradient.Create(AOwner: TComponent);

// \*\*\* construction du composant \*\*\*

**begin**

**inherited** Create(AOwner); // création à partir du propriétaire

Parent := (AOwner **as** TWinControl); // transtypage vers le parent

Align := alClient;

// [Initialisations ici]

**end**;

Le propriétaire est le composant sur lequel est déposé celui créé. Le parent qui permet l’affichage est le même.

Le transtypage de *AOwner* de ***TComponent*** en ***TWinControl*** est nécessaire puisque la propriété *Parent* exige un contrôle fenêtré comme donnée.

Second problème : il faut retrouver la surface à remplir afin de la passer en paramètre à la méthode *GradientFill*. Heureusement, les contrôles fenêtrés disposent d’une propriété qui renvoie le rectangle qu’ils occupent : *ClientRect* qui est du type ***TRect***, celui qui convient exactement à la méthode *GradientFill* !

C’est la méthode *Paint* du composant qui sera chargée de dessiner le contrôle conformément à ce qui est voulu :

**procedure** TGVGradient.Paint;

// \*\*\* dessin du fond \*\*\*

**begin**

**inherited** Paint;

**if** Enabled **then** // actif ?

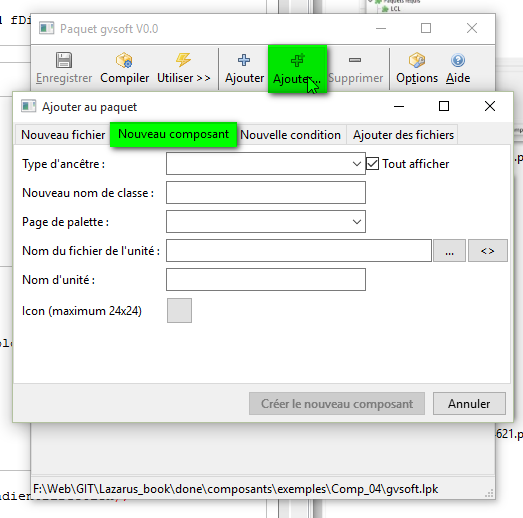
Canvas.GradientFill(ClientRect, BeginColor, EndColor, Direction);

end;

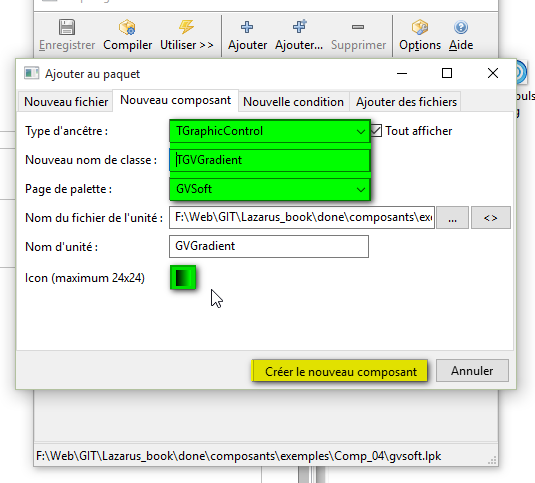
C’est en effet cette méthode qui peint le contrôle quand nécessaire et constitue un emplacement privilégié pour personnaliser son affichage.

Le reste du travail ressemble beaucoup à celui qui a été fait pour le composant ***TGVUrlLabel***. Il s’agit de définir le nouveau composant, en particulier les propriétés à publier afin qu’elles soient disponibles dans l’EDI, et de fournir une procédure d’enregistrement.

Pour ajouter le nouveau composant au paquet gvsoft, il faut en premier lieu ouvrir ce dernier et choisir de créer un nouveau composant :



Comme lors de la création du premier composant, il faut renseigner les trois premières lignes et, optionnellement, choisir une icône :



Cliquez sur « *Créer le nouveau composant* » après avoir vérifié que vous avez bien choisi ***TGraphicControl*** comme type d’ancêtre.

Voici le listing complet de l’unité de ***TGVGradient*** tel qu’elle doit être tapée :

**unit** gvgradient;

{$mode objfpc}{$H+}

**interface**

**uses**

Classes, SysUtils, LResources, Forms, Controls, Graphics, Dialogs;

**type**

{ TGVGradient }

TGVGradient = **class**(TGraphicControl)

**strict private**

fDirection: TGradientDirection;

fEnabled: Boolean;

fBeginColor, fEndColor: TColor;

**procedure** SetBeginColor(AValue: TColor);

**procedure** SetDirection(AValue: TGradientDirection);

**procedure** SetEndColor(AValue: TColor);

**protected**

**procedure** SetEnabled(AValue: Boolean); **override**;

**procedure** Paint; **override**;

**public**

**constructor** Create(AOwner: TComponent); **override**;

**published**

// actif/inactif

**property** Enabled: Boolean **read** fEnabled **write** SetEnabled **default** True;

// couleur de départ

**property** BeginColor: TColor **read** fBeginColor **write** SetBeginColor **default** clBlue;

// couleur de fin

**property** EndColor: TColor **read** fEndColor **write** SetEndColor **default** clAqua;

// direction du dégradé

**property** Direction: TGradientDirection **read** fDirection **write** SetDirection **default** gdVertical;

**end**;

**procedure** Register;

**implementation**

**procedure** Register;

**begin**

{$I gvgradient\_icon.lrs}

RegisterComponents('GVSoft',[TGVGradient]);

**end**;

{ TGVGradient }

**procedure** TGVGradient.SetBeginColor(AValue: TColor);

// \*\*\* couleur de début \*\*\*

**begin**

**if** fBeginColor = AValue **then** // inchangée ?

Exit; // on sort

fBeginColor := AValue; // nouvelle valeur

Repaint;

**end**;

**procedure** TGVGradient.SetDirection(AValue: TGradientDirection);

// \*\*\* direction du dégradé \*\*\*

**begin**

**if** fDirection = AValue **then** // même valeur ?

Exit; // on sort

fDirection := AValue; // nouvelle valeur

Repaint;

**end**;

**procedure** TGVGradient.SetEndColor(AValue: TColor);

// \*\*\* couleur de fin \*\*\*

**begin**

**if** fEndColor = AValue **then** // inchangée ?

Exit; // on sort

fEndColor := AValue; // nouvelle couleur

Repaint;

**end**;

**procedure** TGVGradient.SetEnabled(AValue: Boolean);

// \*\*\* activité du composant \*\*\*

**begin**

**if** fEnabled = AValue **then** // valeur inchangée ?

Exit; // on sort

**inherited** SetEnabled(AValue);

fEnabled := AValue; // nouvelle valeur

Repaint;

**end**;

**procedure** TGVGradient.Paint;

// \*\*\* dessin du fond \*\*\*

**begin**

**inherited** Paint;

**if** Enabled **then** // actif ?

Canvas.GradientFill(ClientRect, BeginColor, EndColor, Direction);

**end**;

**constructor** TGVGradient.Create(AOwner: TComponent);

// \*\*\* construction du composant \*\*\*

**begin**

**inherited** Create(AOwner); // création à partir du propriétaire

Parent := (AOwner **as** TWinControl); // transtypage vers le parent

Align := alClient;

BeginColor := clBlue;

EndColor := clAqua;

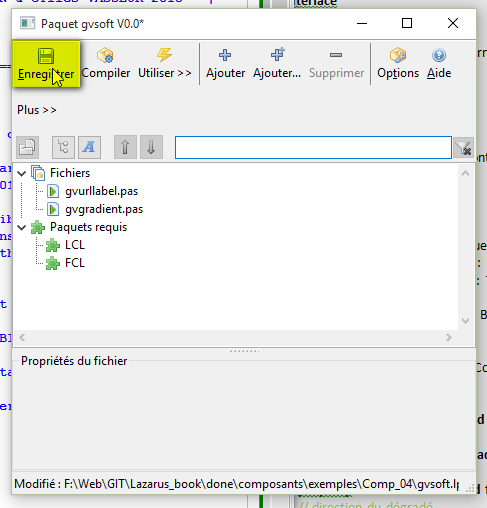
Enabled := True;

Direction := gdVertical;

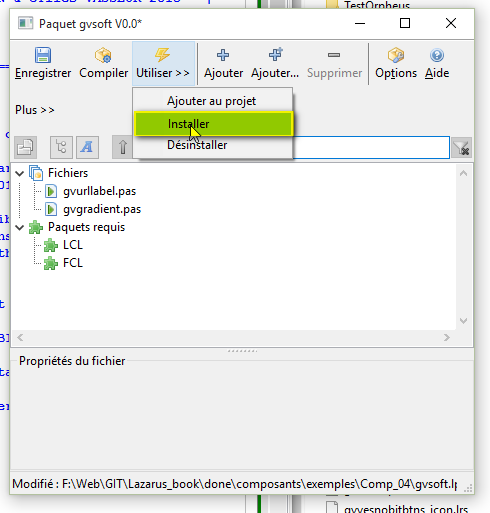
**end**;

**end**.

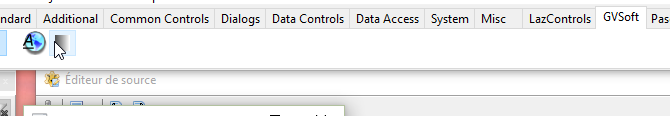
Pensez à enregistrer votre travail :



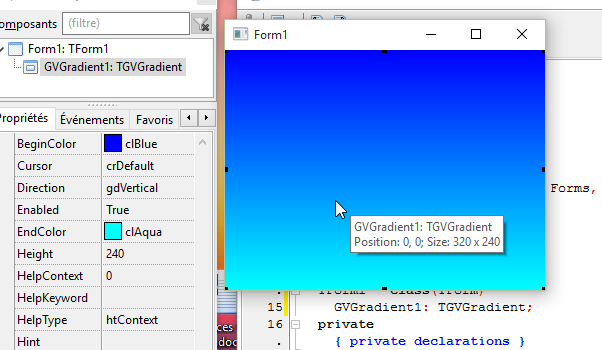
Vous pouvez alors installer votre paquet modifié selon la procédure déjà utilisée :



Après confirmation de la reconstruction de Lazarus, le nouveau composant ***TGVGradient*** est disponible dans la palette aux côtés de ***TGVUrlLabel*** :



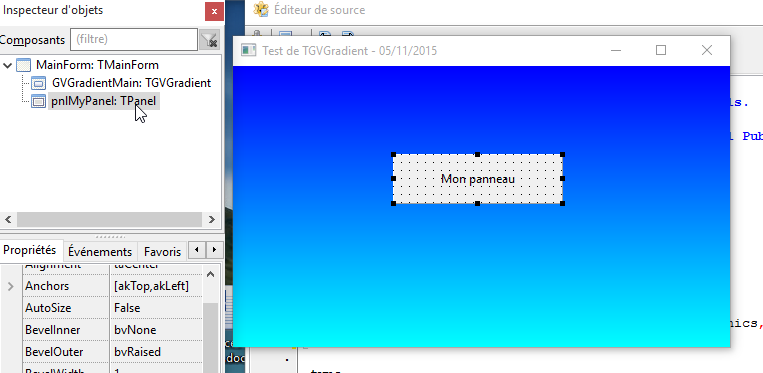
La simple action de déposer un composant ***TGVGradient*** sur la fiche provoquera un changement notable d’affichage :



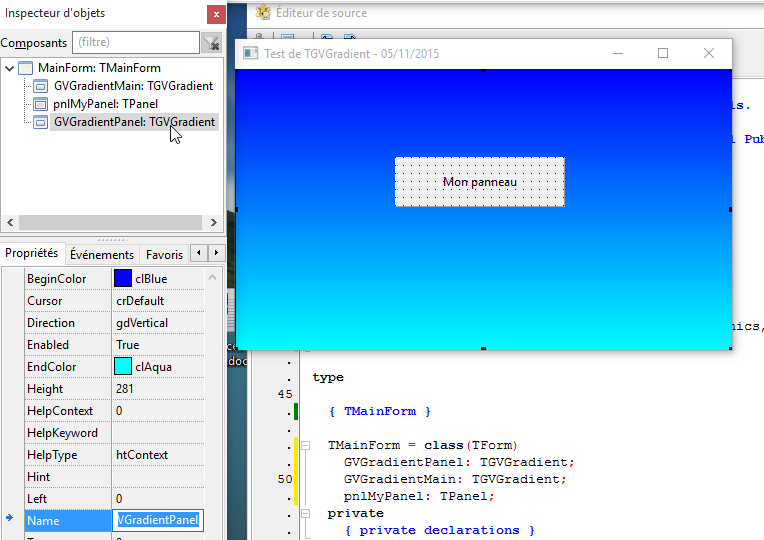
Il vous reste à modifier les principales propriétés : *BeginColor*, *EndColor*, *Direction* et *Enabled* pour modifier l’aspect de votre travail. Par ailleurs, déplacez et redimensionnez la fenêtre du programme d’exemple afin de vérifier que le composant de dégradé s’adapte bien aux changements de son parent.

Si vous changez le parent du composant en choisissant un contrôle graphique comme un ***TPanel***, vous pouvez cumuler les dégradés.

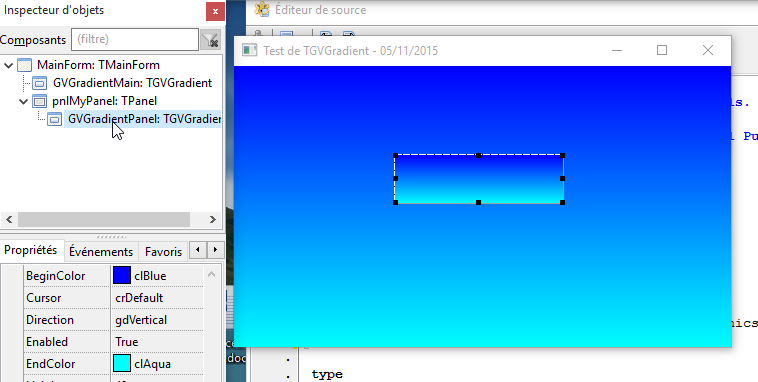
Commencez par déposer un ***TPanel*** sur la fiche :



Poursuivez en y déposant un nouveau TGVGradient :

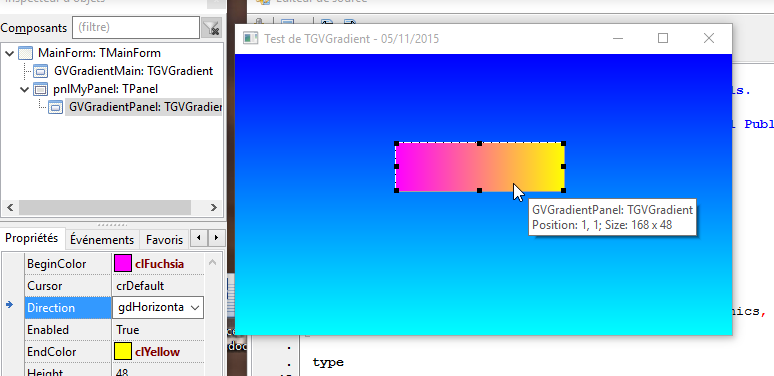


Pour le moment, rien ne semble se passer. C’est normal puisque le nouveau composant de type ***TGVGradient*** a par défaut la fiche principale pour parent. Afin de changer cela, faites glisser le nom de ce nouveau composant dans l’inspecteur d’objets jusqu’au nom du panneau récemment ajouté et lâche-le sur lui. Vous devriez obtenir ceci :



Le nom du second composant de dégradé apparaît comme décalé et dépendant du panneau : c’est ce qui était désiré. À l’affichage, le nouveau dégradé est à présent apparent : son parent est le panneau.

Afin de montrer que les deux dégradés sont indépendants, il est souhaitable de modifier leurs valeurs :



## Un composant non visuel : TGVSizeMover

L’objectif est à présent de réaliser un composant autorisant les contrôles qu’il aura enregistrés à être déplacés et redimensionnés. Il s’agira d’un descendant de ***TComponent*** puisqu’il n’est pas visuel, c’est-à-dire visible à l’exécution.

Le cahier des charges est proche de celui de l’EDI **Lazarus** lui-même : un contrôle visuel pourra être redimensionné grâce à des poignées et déplacé en maintenant le bouton de la souris enfoncé. De plus, l’affichage, les dimensions et la couleur des poignées seront configurables.

Une première difficulté est vite identifiable : comment utiliser des gestionnaires d’événements d’un contrôle pour le déplacer et/ou le redimensionner alors qu’ils peuvent déjà être utilisés par le programme pour d’autres fins ? En effet, on peut imaginer que cliquer sur le contrôle affiche en temps ordinaire une boîte de dialogue : cliquer sur le même contrôle pour le déplacer ne doit évidemment pas afficher cette boîte… Sachant que le déplacement et le redimensionnement sont des états transitoires, il suffit en fait de créer un tableau de méthodes géré par le composant ***TGVSizerMover*** dans lequel seront stockées les méthodes modifiées afin d’être restituées après le déplacement/redimensionnement.

**[Exemple Comp\_05]**

Le tableau des méthodes sera défini ainsi :

// type tableau de méthodes pour la sauvegarde

TMethods = **array** **of** TMethod;

Le type *TMethod* défini dans l’unité *System* définit un enregistrement de deux pointeurs, le premier pointant vers le code de la méthode et le second vers l’instance à laquelle elle appartient.

La seconde difficulté est plus difficile à cerner et à résoudre : afin de gérer les événements liés au contrôle à déplacer/redimensionner, il est nécessaire de capturer les messages de la souris. Manque de chance, la propriété adaptée *MouseCapture*, ainsi que les gestionnaires d’événements concernant la souris sont protégés, donc seulement accessibles par un descendant de ***TControl***. La solution réside en une astuce : on va définir un descendant vide de ***TControl***, uniquement pour récupérer les propriétés et méthodes nécessaires :

**type**

// classe pour récupérer les méthodes protégées

TMoveControl = **class**(TControl)

**end**;

La classe ***TGVSizerMover*** est alors définie ainsi :

{ TGVSizerMover }

TGVSizerMover = **class**(TComponent)

**strict private**

fEnabled: Boolean; // drapeau actif/inactif

fHandlesColor: TColor;

fHandlesOnMove: Boolean; // visibilité des poignées en mouvement

fHandlesSize: Integer; // taille des poignées

fOnChange: TNotifyEvent; // changement notifié

fMoving: Boolean; // drapeau de déplacement

fControls: TComponentList; // contrôles déplaçables

fHandles: TObjectList; // liste des zones de saisie

fGettingHandle: Boolean; // poignées en cours ?

fOldPos: TPoint; // ancienne position d'un contrôle

fCurrentControl: TMoveControl; // contrôle en cours

// stockage des méthodes

fOnClickMethods : TMethods;

fOnChangeMethods : TMethods;

fMouseDownMethods : TMethods;

fMouseMoveMethods : TMethods;

fMouseUpMethods : TMethods;

// actions de la souris sur les poignées

**procedure** HandleMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

**procedure** HandleMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);

**procedure** HandleMouseUp(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

**procedure** SetHandles(AroundControl: TMoveControl); // poignées autour

**procedure** SetEnabled(AValue: Boolean); // activation/désactivation

**procedure** SetHandlesColor(AValue: TColor);

**procedure** SetHandlesOnMove(AValue: Boolean); // visibles/invisibles en déplacement ?

**procedure** SetHandlesSize(AValue: Integer); // taille des poignées

**procedure** SetHandlesVisible(AValue: Boolean); // visibilité des poignées

**protected**

**procedure** Change; // changement

**public**

**constructor** Create(AOwner: TComponent); **override**; // constructeur

**destructor** Destroy; **override**; // destructeur

**procedure** Add(AControl: TControl); // ajout d'un contrôle

**property** Moving: Boolean **read** fMoving; // en mouvement ?

**property** Enabled: Boolean **read** fEnabled **write** SetEnabled; // actif ?

**published**

**property** OnChange: TNotifyEvent **read** fOnChange **write** fOnChange;

**property** HandlesOnMove: Boolean **read** fHandlesOnMove **write** SetHandlesOnMove;

**property** HandlesSize: Integer **read** fHandlesSize **write** SetHandlesSize **default** CHSize;

**property** HandlesColor: TColor **read** fHandlesColor **write** SetHandlesColor **default** CHColor;

**procedure** ControlMouseDown(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

**procedure** ControlMouseMove(Sender: TObject; Shift: TShiftState; X, Y: Integer);

**procedure** ControlMouseUp(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

**end**;

La partie privée de la classe contient essentiellement les champs nécessaires à la gestion des contrôles, ainsi que des drapeaux permettant de savoir si le composant est actif et si un mouvement est en cours. C’est à ce niveau que sont définies les méthodes concernant les événements liés à la souris et celles liées aux poignées.

La partie protégée de la classe ne comprend que la méthode *Change* en charge de la notification d’un changement.

La partie publique, outre les habituels constructeur et destructeur, abrite la méthode *Add* en charge de l’ajout d’un contrôle à la liste de ceux qui sont gérés par le composant et deux propriétés : *Moving* en lecture seule permet de savoir si un contrôle est en mouvement et Enabled permet d’activer/désactiver le composant.

Enfin, la partie des publications propose les propriétés permettant d’accéder aux événements liés à la souris, ainsi que celles définissant les poignées (*HandlesOnMove*, *HandlesSize* et *HandlesColor*).

### Création et destruction du composant

Le constructeur est principalement chargé d’initialiser la liste des contrôles à gérer dans le champ privé *fControls* et de construire une liste de poignées (en fait, de ***TPanel***) stockées dans le champ privé *fHandles* qui est de type ***TObjectList***. Le curseur associé à chaque poignée est adapté à la position prévue de la souris sur le contrôle. Les gestionnaires de la souris (*OnMouseDown*, *OnMouseMove* et *OnMouseUp*) sont alors renseignés :

**constructor** TGVSizerMover.Create(AOwner: TComponent);

// \*\*\* création du composant \*\*\*

**var**

Li: Integer;

LPanel: TPanel;

**begin**

**inherited** Create(AOwner);

fHandles := TObjectList.Create(False);

fControls := TComponentList.Create(False);

fHandlesSize := CHSize; // taille par défaut des poignées

fHandlesColor := CHColor; // couleur par défaut des poignées

**for** Li := 0 **to** 7 **do** // on balaie les poignées

**begin**

LPanel := TPanel.Create(Self); // on crée le panneau

**with** LPanel **do** // traitement

**begin**

Name := 'Handle' + IntToStr(Li); // nom unique

BevelOuter := bvNone; // plat

Caption := EmptyStr;

Color := fHandlesColor; // couleur par défaut

Width := HandlesSize; // taille

Height := HandlesSize;

Visible := False; // invisible par défaut

**case** Li **of** // curseur adapté

0,4: Cursor := crSizeNWSE;

1,5: Cursor := crSizeNS;

2,6: Cursor := crSizeNESW;

3,7: Cursor := crSizeWE;

**end**;

// gestionnaires suivant l'état de la souris

OnMouseDown := @HandleMouseDown;

OnMouseMove := @HandleMouseMove;

OnMouseUp := @HandleMouseUp;

**end**;

fHandles.Add(LPanel); // on l'ajoute à la liste des poignées

**end;**

**end;**

Le destructeur ne pose pas de problèmes particuliers : il se contente de détruire les objets construits par *Create* :

**destructor** TGVSizerMover.Destroy;

// \*\*\* destruction \*\*\*

**begin**

fControls.Free;

fHandles.Free;

**inherited** Destroy;

**end**;

### Le dessin des poignées

La méthode chargée de calculer les coordonnées des 8 poignées d’un contrôle est *SetHandles* définie comme suit :

**procedure** TGVSizerMover.SetHandles(AroundControl: TMoveControl);

// \*\*\* poignées autour du contrôle \*\*\*

**var**

Li,LTop,LLeft: Integer;

LTopLeft: TPoint;

**begin**

fCurrentControl := nil; // pas de composant en cours

**for** Li := 0 **to** 7 **do** // on balaie les poignées à dessiner

**begin**

**with** AroundControl **do** // contrôle à entourer

**begin**

// calcul des emplacements de base

**case** Li **of**

0: **begin**

LTop := Top - (HandlesSize - 1);

LLeft := Left - (HandlesSize - 1);

**end**;

1: **begin**

LTop := Top - (HandlesSize - 1);

LLeft := (Width div 2) + Left - ((HandlesSize div 2) - 1);

**end**;

2: **begin**

LTop := Top - (HandlesSize - 1);

LLeft := Left + Width - 1;

**end**;

3: **begin**

LTop := (Height div 2) + Top - ((HandlesSize div 2) - 1);

LLeft := Left + Width - 1;

**end**;

4: **begin**

LTop := Top + Height - 1;

LLeft := Left + Width - 1;

**end**;

5: **begin**

LTop := Top + Height - 1;

LLeft := (Width div 2) + Left - ((HandlesSize div 2) - 1);

**end**;

6: **begin**

LTop := Top + Height - 1;

LLeft := Left - (HandlesSize - 1);

end;

7: **begin**

LTop := (Height div 2) + Top - ((HandlesSize div 2) - 1);

LLeft := Left - (HandlesSize - 1);

**end**;

**end**;

// point supérieur gauche

LTopLeft := Parent.ClientToScreen(Point(LLeft,LTop));

**end**;

**with** TPanel(fHandles[Li]) **do** // panneau placé

**begin**

Parent := AroundControl.Parent; // pour l'affichage

LTopLeft := Parent.ScreenToClient(LTopLeft); // coordonnées locales

Top := LTopLeft.Y; // emplacement réel

Left := LTopLeft.X;

**end**;

**end**;

fCurrentControl := AroundControl; // composant en cours

SetHandlesVisible(True); // les poignées sont visibles

**end**;

Une nouvelle fois, l’affichage d’un contrôle exige de définir son parent. C’est pourquoi le parent des ***TPanel*** qui constituent les poignées est défini au parent du contrôle entouré.

Il faut par ailleurs contrôler le dessin de ces poignées suivant l’état de la souris. On aura à gérer la souris sur le contrôle, lorsqu’elle le quitte ou qu’elle se déplace.

*OnMouseDown* sera contrôlé grâce à *HandleMouseDown* :

**procedure** TGVSizerMover.HandleMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

// \*\*\* souris cliquée sur poignée \*\*\*

**begin**

**if** Enabled **and** (Sender **is** TControl) **then** // actif et un contrôle ?

**begin**

fGettingHandle := True; // poignées actives

// capture des messages de la souris

TMoveControl(Sender).MouseCapture := True;

// enregistrement de la position du curseur

GetCursorPos(fOldPos);

**end**;

**end**;

La capture des messages de la souris s’effectue par le contrôle à entourer.

*OnMouseUp* sera géré grâce à *HandleMouseUp* dont la tâche essentielle est de remettre le contrôle dans l’état où était avant le clic :

**procedure** TGVSizerMover.HandleMouseUp(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

// \*\*\* relâchement du bouton de la souris \*\*\*

**begin**

**if** fGettingHandle **then** // poignées actives ?

**begin**

Screen.Cursor := crDefault; // curseur par défaut

TMoveControl(Sender).MouseCapture := False; // capture libérée

TMoveControl(Sender).Repaint; // contrôle redessiné

fGettingHandle := False; // fin de l'activité des poignées

**end**;

**end**;

*OnMouseMove* sera traité grâce à *HandleMouseMove* qui est un peu plus compliquée que ses homologues, car elle doit aussi s’occuper du redimensionnement du contrôle actif :

**procedure** TGVSizerMover.HandleMouseMove(Sender: TObject; Shift: TShiftState; X,

Y: Integer);

// \*\*\* redimensionnement \*\*\*

**var**

LNewPos, LPoint: TPoint;

LOldRect: TRect;

**begin**

**if** fGettingHandle **then** // poignées actives ?

**begin**

**with** TMoveControl(Sender) **do** // contrôle en cours

**begin**

GetCursorPos(LNewPos); // nouvelle position du curseur

**with** fCurrentControl **do** // contrôle enregistré

**begin** //redimensionnement

// point en cours

LPoint := Parent.ScreenToClient(Mouse.CursorPos);

// rectangle en cours

LOldRect := BoundsRect;

**case** fHandles.IndexOf(TMoveControl(Sender)) **of** // quelle poignée ?

0: **begin**

LOldRect.Left := LPoint.X;

LOldRect.Top := LPoint.Y;

**end**;

1: LOldRect.Top := LPoint.Y;

2: **begin**

LOldRect.Right := LPoint.X;

LOldRect.Top := LPoint.Y;

**end**;

3: LOldRect.Right := LPoint.X;

4: **begin**

LOldRect.Right := LPoint.X;

LOldRect.Bottom := LPoint.Y;

**end**;

5: LOldRect.Bottom := LPoint.Y;

6: **begin**

LOldRect.Left := LPoint.X;

LOldRect.Bottom := LPoint.Y;

**end**;

7: LOldRect.Left := LPoint.X;

**end**;

SetBounds(LOldRect.Left, LOldRect.Top,LOldRect.Right -

LOldRect.Left, LOldRect.Bottom - LOldRect.Top);

**end**;

// nouvelle position

Left := Left - fOldPos.X + LNewPos.X;

Top := Top - fOldPos.Y + LNewPos.Y;

fOldPos := LNewPos;

**end**;

// affichage des poignées

SetHandles(fCurrentControl);

**end**;

**end**;

Les autres méthodes concernant les poignées contrôlent leur aspect. Ainsi, l’utilisateur a la maîtrise de l’affichage des poignées, de leur couleur et de leur taille :

**procedure** TGVSizerMover.SetHandlesColor(AValue: TColor);

// \*\*\* couleur des poignées \*\*\*

**var**

Li: Integer;

**begin**

**if** fHandlesColor = AValue **then** // pas de changement ?

Exit; // on sort

fHandlesColor := AValue; // nouvelle valeur

**for** Li := 0 **to** 7 **do** // on balaie les poignées

TMoveControl(fHandles[Li]).Color := AValue;

**end**;

**procedure** TGVSizerMover.SetHandlesOnMove(AValue: Boolean);

// \*\*\* poignées visibles lors d'un déplacement ? \*\*\*

**begin**

**if** fHandlesOnMove= AValue **then** // valeur inchangée ?

Exit; // on sort

fHandlesOnMove := AValue; // nouvelle valeur

**end**;

**procedure** TGVSizerMover.SetHandlesSize(AValue: Integer);

// \*\*\* taille des poignées \*\*\*

**var**

Li: Integer;

**begin**

**if** fHandlesSize = AValue **then** // même valeur ?

Exit; // on sort

fHandlesSize := AValue; // nouvelle valeur

**if** Enabled **then**

SetHandlesVisible(False);

**for** Li := 0 **to** 7 **do** // on balaie les poignées

**begin**

TMoveControl(fHandles[Li]).Height := AValue; // hauteur

TMoveControl(fHandles[Li]).Width := AValue; // largeur

**end**;

**end**;

**procedure** TGVSizerMover.SetHandlesVisible(AValue: Boolean);

// \*\*\* visibilité des poignées \*\*\*

**var**

Li: Integer;

**begin**

**for** Li := 0 **to** 7 **do** // on balaie les poignées

TMoveControl(fHandles[Li]).Visible := AValue; // visibilité fixée

**end**;

### La gestion des contrôles

Il reste bien évidemment à traiter les contrôles que le composant doit gérer. En premier lieu, il faut les intégrer à une liste :

**procedure** TGVSizerMover.Add(AControl: TControl);

// \*\*\* ajout d'un contrôle \*\*\*

**begin**

fControls.Add(AControl);

**end**;

Il faut, comme pour les poignées, s’occuper de la souris dès qu’elle affecte un contrôle de la liste, aussi bien dans le cas d’un clic que lors d’un déplacement :

**procedure** TGVSizerMover.ControlMouseDown(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

// \*\*\* souris cliquée sur contrôle \*\*\*

**begin**

**if** Enabled **and** (Sender is TControl) **then** // actif et un contrôle ?

**begin**

fMoving := True; // déplacement

// capture des messages de la souris

TMoveControl(Sender).MouseCapture := True;

// contrôle devant les autres contrôles

TMoveControl(Sender).BringToFront;

// enregistrement de la position du curseur

GetCursorPos(fOldPos);

// poignées dessinées

SetHandles(TMoveControl(Sender));

**end**;

**end**;

**procedure** TGVSizerMover.ControlMouseMove(Sender: TObject; Shift: TShiftState;

X, Y: Integer);

// \*\*\* déplacement du contrôle \*\*\*

**var**

LNewPos, LFrmPoint: TPoint;

**begin**

**if** Moving **then**

**begin**

**with** TControl(Sender) **do**

**begin**

GetCursorPos(LNewPos);

**if** ssShift **in** Shift **then** // taille si touche majuscules

**begin**

Screen.Cursor := crSizeNWSE;

LFrmPoint := ScreenToClient(Mouse.CursorPos);

Width := LFrmPoint.X;

Height := LFrmPoint.Y;

**end**

**else** // déplacement

**begin**

Screen.Cursor := crSize;

Left := Left - fOldPos.X + LNewPos.X;

Top := Top - fOldPos.Y + LNewPos.Y;

fOldPos := LNewPos;

**end**;

**end**;

Change; // changement notifié

**if** HandlesOnMove **then**

SetHandles(TMoveControl(Sender)) // poignées visibles

**else**

SetHandlesVisible(False); // poignées invisibles

**end**;

**end**;

**procedure** TGVSizerMover.ControlMouseUp(Sender: TObject; Button: TMouseButton;

Shift: TShiftState; X, Y: Integer);

// \*\*\* fin de clic sur contrôle \*\*\*

**begin**

**if** Moving **then**

**begin**

Screen.Cursor := crDefault; // curseur normal

ReleaseCapture; // souris libérée

fMoving := False; // fin du mouvement

**if** **not** HandlesOnMove **then** // poignées à dessiner ?

SetHandles(TMoveControl(Sender)); // on les dessine

**end**;

**end**;

Enfin, lors de l’activation du composant, il est nécessaire de mémoriser les gestionnaires de la souris des contrôles gérés, de même qu’il faut les restituer lors de la désactivation. La méthode *SetEnabled* qui effectue ce travail est copieuse, mais le schéma de son fonctionnement est simple et s’appuie sur le fait que les méthodes sont stockées sous la forme d’un enregistrement ***TMethod*** qui comprend un pointeur vers leur code et un autre vers leurs données :

**procedure** TGVSizerMover.SetEnabled(AValue: Boolean);

// \*\*\* composant actif ? \*\*\*

**var**

Li: Integer;

LOldM, LNewM, LNilM: TMethod;

**begin**

**if** fEnabled = AValue **then** // même valeur ?

Exit; // on sort

fEnabled := AValue; // nouvelle valeur

**if** Enabled **then**

**begin**

fOnClickMethods := nil; // nettoyage des tableaux de méthodes

fOnChangeMethods := nil;

fMouseDownMethods := nil;

fMouseMoveMethods := nil;

fMouseUpMethods := nil;

LNilM.Data := nil; // méthode à nil

LNilM.Code := nil;

// longueur des tableaux dynamiques

SetLength(fOnClickMethods, fControls.Count);

SetLength(fOnChangeMethods, fControls.Count);

SetLength(fMouseDownMethods, fControls.Count);

SetLength(fMouseMoveMethods, fControls.Count);

SetLength(fMouseUpMethods, fControls.Count);

// inversion des méthodes concernant la souris

**for** Li := 0 **to** (fControls.Count - 1) **do**

**begin**

// OnClick

LOldM := GetMethodProp(TControl(fControls[Li]), 'OnClick');

fOnClickMethods[Li].Code := LOldM.Code;

fOnClickMethods[Li].Data := LOldM.Data;

SetMethodProp(TControl(fControls[Li]), 'OnClick', LNilM);

// OnChange

**if** IsPublishedProp(TControl(fControls[Li]), 'OnChange') **then**

**begin**

LOldM := GetMethodProp(TControl(fControls[Li]), 'OnChange');

fOnChangeMethods[Li].Code := LOldM.Code;

fOnChangeMethods[Li].Data := LOldM.Data;

SetMethodProp(TControl(fControls[Li]), 'OnChange', LNilM);

**end**;

// OnMouseDown

LOldM := GetMethodProp(TControl(fControls[Li]), 'OnMouseDown');

fMouseDownMethods[Li].Code := LOldM.Code;

fMouseDownMethods[Li].Data := LOldM.Data;

LNewM.Code := Self.MethodAddress('ControlMouseDown');

LNewM.Data := Pointer(Self);

SetMethodProp(TControl(fControls[Li]), 'OnMouseDown', LNewM);

// OnMouseMove

LOldM := GetMethodProp(TControl(fControls[Li]), 'OnMouseMove');

fMouseMoveMethods[Li].Code := LOldM.Code;

fMouseMoveMethods[Li].Data := LOldM.Data;

LNewM.Code := Self.MethodAddress('ControlMouseMove');

LNewM.Data := Pointer(Self);

SetMethodProp(TControl(fControls[Li]), 'OnMouseMove', LNewM);

// OnMouseUp

LOldM := GetMethodProp(TControl(fControls[Li]), 'OnMouseUp');

fMouseUpMethods[Li].Code := LOldM.Code;

fMouseUpMethods[Li].Data := LOldM.Data;

LNewM.Code := Self.MethodAddress('ControlMouseUp');

LNewM.Data := Pointer(Self);

SetMethodProp(TControl(fControls[Li]), 'OnMouseUp', LNewM);

**end**;

**end**

**else**

**begin**

// récupération des anciennes valeurs

**for** Li := 0 **to** (fControls.Count - 1) **do**

**begin**

// OnClick

LOldM.Code := fOnClickMethods[Li].Code;

LOldM.Data := fOnClickMethods[Li].Data;

SetMethodProp(TControl(fControls[Li]), 'OnClick', LOldM);

// OnChange

**if** IsPublishedProp(TControl(fControls[Li]), 'OnChange') **then**

**begin**

LOldM.Code := fOnChangeMethods[Li].Code;

LOldM.Data := fOnChangeMethods[Li].Data;

SetMethodProp(TControl(fControls[Li]), 'OnChange', LOldM);

**end**;

// OnMouseDown

LOldM.Code := fMouseDownMethods[Li].Code;

LOldM.Data := fMouseDownMethods[Li].Data;

SetMethodProp(TControl(fControls[Li]), 'OnMouseDown', LOldM);

// OnMouseMove

LOldM.Code := fMouseMoveMethods[Li].Code;

LOldM.Data := fMouseMoveMethods[Li].Data;

SetMethodProp(TControl(fControls[Li]), 'OnMouseMove', LOldM);

// OnMouseUp

LOldM.Code := fMouseUpMethods[Li].Code;

LOldM.Data := fMouseUpMethods[Li].Data;

SetMethodProp(TControl(fControls[Li]), 'OnMouseUp', LOldM);

**end**;

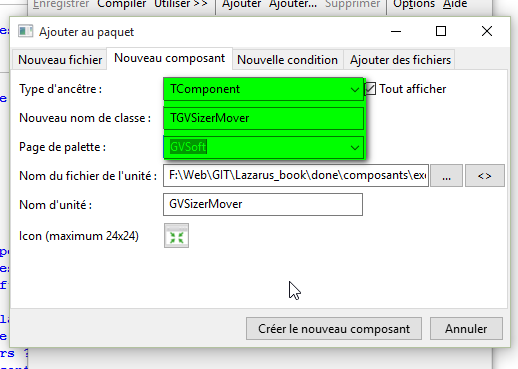
SetHandlesVisible(False); // poignées invisibles

**end**;

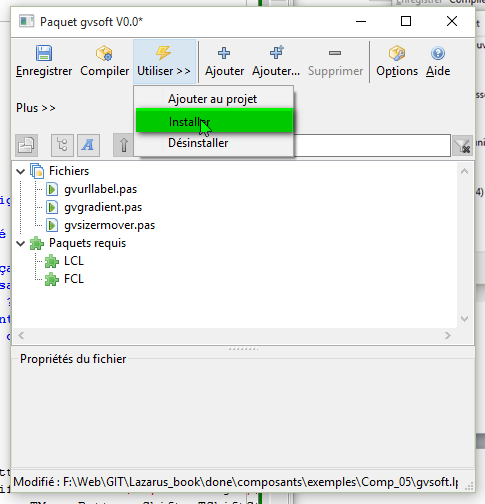
**end**;

### Intallation du composant TGVSizerMover

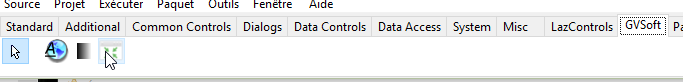
Les procédures de création et d’installation sont celles déjà étudiées précédemment. Chargez le paquet *gvsoft.lpk*, cliquez sur « *Ajouter…* », puis sur « *Nouveau composant* » que vous définirez ainsi :



Cliquez alors sur « *Nouveau composant* » afin d’obtenir le squelette du composant à compléter. Après la frappe de l’unité, il ne reste qu’à cliquer sur « *Utiliser* » et « *Installer* » le paquet, avant de confirmer la reconstruction de **Lazarus** :



Le nouveau composant ***TGVSizerMover*** doit alors être disponible dans la palette de composants sous l’onglet « *gvoft* » :



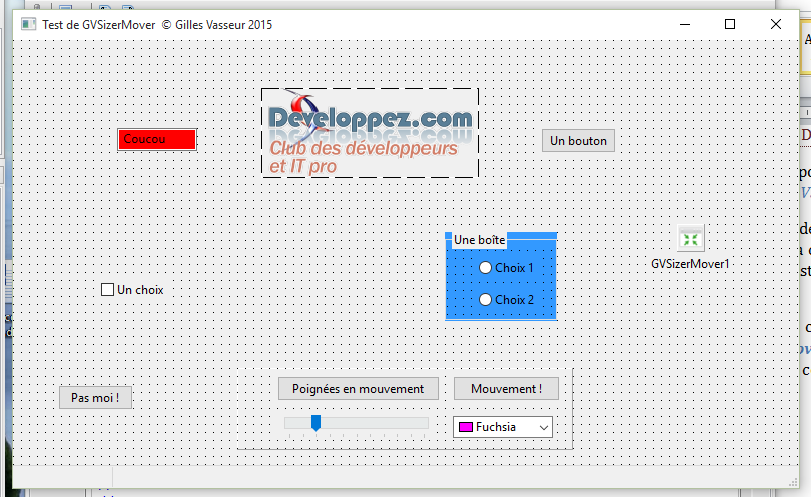
### Test du composant TGVSizerMover

Afin de tester le comportement de ce nouveau composant, vous allez créer un nouveau projet nommé *TestGVSizerMover*.

Tout d’abord, vous déposerez toute une série de contrôles sur la fiche principale : ce sont eux qui seront sous la coupe du composant ***TGVSizerMover***. Un seul contrôle ne sera pas enregistré dans la liste des contrôles à manipuler : un bouton dont la légende sera « *pas moi !* ».

Un panneau (***TPanel***) contiendra des contrôles qui détermineront les propriétés du composant ***TGVSizerMover*** : taille, couleur et visibilité des poignées, ainsi qu’activation ou non du composant. Ce panneau sera lui aussi déplaçable et redimensionnable.

L’interface du programme de test donnera ceci :



Le programme lui-même est relativement simple : il enregistre les contrôles dans la méthode de création de la fiche *FormCreate* et gère les modifications relatives au composant ***TGVSizerMover*** *via* les contrôles qu’accueille le ***TPanel***. Une barre des statuts ***TStatusBar*** affichera l’état du composant testé, ainsi que les coordonnées du contrôle actif.

**unit** main;

{$mode objfpc}{$H+}

**interface**

**uses**

Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,

ComCtrls, ExtCtrls, ColorBox, gvsizermover;

**type**

{ TfrmMain }

TfrmMain = **class**(TForm)

btnMove: TButton;

Button1: TButton;

btnHandles: TButton;

btnNo: TButton;

CheckBox1: TCheckBox;

ColorBox1: TColorBox;

Edit1: TEdit;

GroupBox1: TGroupBox;

GVSizerMover1: TGVSizerMover;

Image1: TImage;

Panel1: TPanel;

RadioButton1: TRadioButton;

RadioButton2: TRadioButton;

StatusBar: TStatusBar;

tbSize: TTrackBar;

procedure btnMoveClick(Sender: TObject);

procedure btnHandlesClick(Sender: TObject);

procedure ColorBox1Change(Sender: TObject);

procedure Edit1Click(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure GVSizerMover1Change(Sender: TObject);

procedure tbSizeChange(Sender: TObject);

**private**

{ private declarations }

**public**

{ public declarations }

**end**;

**var**

frmMain: TfrmMain;

**implementation**

{$R \*.lfm}

{ TfrmMain }

**procedure** TfrmMain.btnMoveClick(Sender: TObject);

// \*\*\* activation/désactivation des mouvements \*\*\*

**begin**

GVSizerMover1.Enabled := **not** GVSizerMover1.Enabled;

btnHandles.Enabled := GVSizerMover1.Enabled;

**if** **not** GVSizerMover1.Enabled **then**

Statusbar.Panels[0].Text := 'Attente...'

**else**

Statusbar.Panels[0].Text := 'Mouvement...'

**end**;

**procedure** TfrmMain.btnHandlesClick(Sender: TObject);

// \*\*\* poignées visibles/invisibles lors des déplacements

**begin**

GVSizerMover1.HandlesOnMove := **not** GVSizerMover1.HandlesOnMove;

**end**;

**procedure** TfrmMain.ColorBox1Change(Sender: TObject);

// \*\*\* changement de couleur des poignées \*\*\*

**begin**

GVSIzerMover1.HandlesColor := ColorBox1.Selected;

**end**;

**procedure** TfrmMain.Edit1Click(Sender: TObject);

// \*\*\* clic sur contrôle \*\*\*

**begin**

Statusbar.Panels[1].Text := 'Clic sur : ' + Sender.ClassName;

**end**;

**procedure** TfrmMain.FormCreate(Sender: TObject);

// \*\*\* création de la fiche \*\*\*

**begin**

GVSizerMover1.Add(Button1);

GVSizerMover1.Add(CheckBox1);

GVSizerMover1.Add(Edit1);

GVSizerMover1.Add(GroupBox1);

GVSizerMover1.Add(Image1);

GVSizerMover1.Add(Panel1);

**end**;

**procedure** TfrmMain.GVSizerMover1Change(Sender: TObject);

// \*\*\* changement \*\*\*

**begin**

**with** Statusbar.Panels[1] **do**

Text := format('X : %d Y : %d', [(Sender **as** TControl).Left, (Sender **as** TControl).Top]);

**end**;

**procedure** TfrmMain.tbSizeChange(Sender: TObject);

// \*\*\* taille des poignées \*\*\*

**begin**

GVSizerMover1.HandlesSize := tbSize.Position + 2;

**end**;

**end**.

L’exécution du programme devient intéressante en cliquant sur le bouton « *Mouvement* » qui active le composant ***TGVSizerMover***:



Vous pouvez alors changer de place ou redimensionner la plupart des éléments de la fiche dans la mesure où ils auront été recensés par la méthode *Add* dans *FormCreate*. Vous pouvez aussi adapter les poignées à vos besoins en utilisant les contrôles du ***TPanel***.

Voici le genre d’écran que vous obtiendrez en vous amusant un peu :

