Delphi-Source.de Tutorial

TList-Tutorial von Johannes Tränkle

Einleitung

Herzlich willkommen bei meinem kleinen Listentutorial. Das Tutorial soll euch TList ein klein wenig näher bringen und euch zeigen, wie ihr TList sinnvoll einsetzen könnt.

Zu diesem Zweck schreiben wir uns ein kleinen "Vektorzeichenprogramm". Ihr werdet damit zwar kein Marktführer, was wir erreichen wollen ist aber folgendes:

- Rechtecken, Ellipsen und Bildern sollen angezeigt werden.
- Diese sollen mit der Maus und der Tastatur verschoben werden können
- Die verschiedenen Elemente sollen nach vorne bzw. nach hinten geschoben werden können.

Ihr seht also, "Vektorzeichenprogramm" ist ein klein wenig hoch gegriffen, aber man soll sich seine Ziele ja hoch stecken.

TList allgemein

Zuerst will ich kurz die grundlegenen Funktionen von TList beschreiben. Danach werden wir unser Programm schreiben.

TList speichert wie uns die Hilfe verrät ein Array von Zeigern. Während diese Funktion früher noch von Hand programmiert werden mußte, nimmt uns TList jetzt die Arbeit ab.

Ein TList-Objekt erstellt Ihr wie folgt:

```
var UnsereListe:TList;
...
UnsereListe:=TList.Create;
```

Wenn wir fertig sind, entfernen wir das ganze wieder aus dem Speicher:

```
UnsereListe.Free;
```

Um einen Pointer an die Liste anzufügen oder zu löschen, geben wir folgenden Code ein:

```
UnsereListe.Add(Zeiger1); // Hinzufügen
UnsereListe.Delete(0); // Löscht den Zeiger Nummer 0, der erste Index ist 0

{ bevor der Zeiger aus der List gelöscht wird, muß der von
ihm belegte Speicher wieder freigegeben werden, sonst haben wir es mit
einer
Speicherleiche zu tun}
```

Die Zeiger können auch innerhalb der Liste bewegt werden:

```
UnsereListe.Move(0,3); // verschiebt Element 0 an Position 3
```

Um noch mal alles auf einmal vor uns zu haben, folgendes kleines Programm:

```
var UnsereListe:TList;
    Zahl:^Integer; // Ein Pointer auf eine Integer-Variable
    loop:Integer; // einfach nur ne Variable

begin

 UnsereListe:=TList.Create;
 New(Zahl); // Speicher für Zahl reservieren
 Zahl^:=5; // irgendeinen Wert zuweisen
 UnsereListe.Add(Zahl); // Element 0

 New(Zahl); // Speicher für Zahl nochmals reservieren
 Zahl^:=10; // irgendeinen Wert zuweisen
 UnsereListe.Add(Zahl); // Element 1

 { die Liste beinhaltet jetzt also zwei Zeiger auf Integervariablen
 mit folgendem Inhalt 5, 10 }

 UnsereListe.Move(0,1); // verschiebt Element 0 an Position 1

 {die Liste sieht jetzt also so aus: 10, 5 }

 for loop:=0 to UnsereListe.Count-1 do
  begin
   Zahl:=UnsereListe.Items[loop];
   Dispose(Zahl);
  end;
 { wir haben jetzt also den Speicher für jede Integer-Variable wieder
freigegeben }

 while UnsereListe.Count>0 do
  UnsereListe.Delete(0);
 { alle Elemente werden wieder aus der Liste gelöscht }

 UnsereListe.Free; // raus aus dem Speicher

end;
```

Jetzt wo wir wissen, wie TList prinzipiell funktioniert, kann es losgehen.

Schritt 1

Jetzt gehts los:
Wir erstellen ein Formular (Name: HauptForm). Am oberen Rand platzieren wir ein Panel, dessen Align-Eigenschaft wir auf alTop setzen. Auf dieses Panel ziehen wir drei Buttons, die wir mit "Rechteck", "Kreis" und "Bild" beschriften.

In die Mitte des Formulars setzen wir jetzt noch eine TImage-Komponente (Name: Zeichenflaeche) und einen OpenPicture-Dialog (in Delphi 5, ansonsten einfach einen OpenDialog) (Name:PictureDialog).

Um die von uns später eingefügten Zeichenelemente zu speichern, tragen wir in der public-Sektion unseres Formulars ein TList-Objekt namens Objekte und eine Variable "Angewaehlt" ein:

```
public
 { Public-Deklarationen }
 Objekte:TList;
 AnGewaehlt:Integer; // welches Objekt ist gerade angewählt
```

Wir müssen uns jetzt überlegen wie wir unsere Objekte, also die Ellipsen, Rechtecke und Bilder, die wir zeichnen wollen, im Speicher ablegen.

Denkbar wäre z.B. einen Typ wie folgt zu definieren:

```
type
  ObjektTyp=Record
   Typ:Integer; // 1 =Bild 2 = Recheck 3 = Ellipse
   X,Y,Height,Width:Integer; // Position und Ausmaße
   Image:TBitmap; // für den Fall, daß es ein Bild ist
   LinienFarbe,FuellFarbe:TColor; // für den Fall, daß es eine Form ist
  end;
```

Auch wenn es bei unserem Projekt nicht allzu tragisch wäre, fällt doch auf, daß wir einige Dinge speichern, die wir eigentlich nicht bräuchten. Handelt es sich um ein Bild, dann brauchen wir die Daten über die Linien- und Füllfarbe nicht, bei einer Form brauchen wir Image nicht.

Es bietet sich also an, für Bilder und Formen zwei separate Typen zu definieren (natürlich könnte man auch drei Typen - Bild, Rechteck, Ellipse - definieren, das wollen wir hier aber nicht tun):

```
type
  BildTyp=Record
   Typ:Integer;
   X,Y,Height,Width:Integer;
   Image:TBitmap;
  end;
  FormTyp=Record
   Typ:Integer;
   X,Y,Height,Width:Integer;
   FormTyp:Integer; // 1 = RechtEck 2 = Ellipse
   LinienFarbe,FuellFarbe:TColor;
  end;
```

Außerdem definieren wir noch einen Basistyp, der z.T. genau die gleiche Datenstruktur aufweißt wie unsere anderen Typen:

```
  BasisTyp=Record
   Typ:Integer; // 1=Bild 2= Form
   X,Y,Height,Width:Integer;
  end;
```

Wofür wir den brauchen, werden wir bald merken.

Schritt 2

Jetzt fügen wir folgenden Code in unser OnCreate-Ereignis ein:

```
 Randomize;
 Objekte:=tList.Create;
 ZeichenFlaeche.Align:=alClient;
```

Im OnClose-Ereignis fügen wir folgendes ein:

```
  while Objekte.Count>0 do
  begin
    LoescheObjekt(Objekte[0]);
    Objekte.Delete(0);
  end;
  Objekte.Free;
```

Die LoescheObjekt-Prozedur werden wir später definieren.

Zuerst werden wir aber die verschiedenen Objekte erstellen. Jedesmal, wenn der Benutzer auf einen der Buttons klickt soll ein Rechteck etc. erstellt werden.

Hierzu definieren wir zwei Prozeduren, eine für Bilder, eine für Formen:

```
 public
    { Public-Deklarationen }
    Objekte:TList;
    AnGewaehlt:Integer; // welches Objekt ist gerade angewählt
    procedure NeuesBild(FileName:String);
    procedure NeueForm(WelcheForm:Integer);
  end;

procedure THauptform.NeuesBild(FileName:String);
var Bild:^BildTyp;
begin
 New(Bild);
 with Bild^ do
  begin
   Typ:=1;
   X:=0;
```

```
   y:=0;
   Image:=TBitmap.Create; // Speicher zuweisen
   Image.LoadFromFile(FileName); // Laden
   Width:=Image.Width;
   Height:=Image.Height;
  end;
 Objekte.Add(Bild); // zu unseren Objekten hinzufügen
 AnGewaehlt:=Objekte.Count-1; // das neueste wird angewählt
end;

procedure THauptform.NeueForm(WelcheForm:Integer);
var Form:^FormTyp;
begin
 New(Form);
 with Form^ do
  begin
   Typ:=2;
   X:=Random(ZeichenFlaeche.ClientWidth div 2); { wir
   positionieren es einfach zufällig}
   y:=Random(ZeichenFlaeche.ClientHeight div 2);
   Width:=Random(ZeichenFlaeche.ClientWidth div 4)+10;
   Height:=Random(ZeichenFlaeche.ClientHeight div 4)+10;
   FormTyp:=WelcheForm;
   LinienFarbe:=clBlack;
   FuellFarbe:=clBlue;
  end;
 Objekte.Add(Form);
 AnGewaehlt:=Objekte.Count-1; // das neueste wird angewählt
end;
```

Im nächsten Schritt werden wir die Prozedur schreiben, die unsere Objekte anzeigt (PaintIt), vorerst weisen wir den Buttons jeweils folgenden OnClick-Code zu:

```
Ein Rechteck:
 NeueForm(1);
 PaintIt;
Ein Kreis:
 NeueForm(2);
 PaintIt;
Ein Bild:
 if PictureDialog.Execute then NeuesBild(PictureDialog.FileName);
 PaintIt;
```

Schritt 3

Wir wollen jetzt die Prozedur schreiben, die das ganze zeichnet.

Nun werden wir auch sehen, wieso wir einen Basistypen definiert haben.
Da der Benutzer ja in beliebiger Reihenfolge Objekte einfügen kann, wissen wir nicht, welche Objekte sich an welcher Stelle der Liste befinden.

Um herauszufinden, um was für ein Objekt es sich handelt, weisen wir einer Variable des Types Basistyp einfach den Zeiger auf das zu zeichnende Objekt zu. Da es sich bei BasisTyp um die gleiche Datenstruktur wie bei BildTyp und FormTyp handelt, können wir ohne Probleme auf die Daten zugreifen:

```

var Basis:^BasisTyp;
```

```
begin
  Basis:=Objekte[0]; // das erste Objekt ist ein??
  case Basis^.Typ of
  ...
```

Das Ganze sieht dann so aus:

```
public

    ...
    procedure PaintIt;
    ...
  end;

procedure THauptForm.PaintIt;
var loop:Integer;
    Basis:^BasisTyp;
    bild:^BildTyp;
    Form:^FormTyp;

begin
 with ZeichenFlaeche.Canvas do
  begin
   // Löschen
   Brush.Color:=clWhite;
   Pen.Color:=clWhite;
   Rectangle(0,0,ZeichenFlaeche.Width,zeichenFlaeche.Height);

   for loop:=0 to Objekte.Count-1 do // das letzte zuerst zeichnen
    begin
     Basis:=Objekte[loop];
     case Basis^.Typ of
      1: begin // Bild
          Bild:=Objekte[loop];

StretchDraw(Rect(Bild^.X,Bild^.Y,Bild^.X+Bild^.Width,Bild^.Y+Bild^.Height),
         Bild^.Image);
          //Draw(Bild^.X,Bild^.Y,Bild^.Image);
         end; // Bild;
      2: begin // Form
          Form:=Objekte[loop];
          Brush.Color:=Form^.FuellFarbe;
          Pen.Color:=Form^.LinienFarbe;
           case Form^.FormTyp of
            1:
Rectangle(Form^.X,Form^.Y,Form^.X+Form^.Width,Form^.Y+Form^.Height);
            2:
Ellipse(Form^.X,Form^.Y,Form^.X+Form^.Width,Form^.Y+Form^.Height);
           end; // Rechteck oder Kreis
         end; // Form
     end; // case

     if loop=AnGewaehlt then
      begin // um das gerade gewählte Objekt wird ein roter Rahmen
gezeichnet
          Brush.Style:=bsClear;
          Pen.Color:=clRed;

Rectangle(Basis^.X,Basis^.Y,Basis^.X+Basis^.Width,Basis^.Y+Basis^.Height);
          Brush.Style:=bsSolid;
```

```
      end; // angewaehlt
    end; // jedes Objekt
  end;
end;
```

Durch die gleiche Datenstruktur war es uns also möglich zu erfahren, um was für einen Typen es sich überhaupt handelte. Im BasisTyp sind ja noch weitere, bei jedem Objekt gleiche, Daten wie die Position und die Größe gespeichert, dies brauchen wir im nächsten Schritt.

Schritt 4

Um das angewählte Objekt zu wechseln fügen wir im OnMouseDown-Ereignis der Zeichenfläche folgenden Code ein:

```
var loop:Integer;
    Basis:^BasisTyp;
begin

   for loop:= Objekte.Count-1 downto 0 do // das vorderste zuerst prüfen
    begin
     Basis:=Objekte[loop];
     if X>Basis^.X then if X<Basis^.X+Basis^.Width then
      if Y>Basis^.Y then if Y<Basis^.Y+Basis^.Height then
       begin
        AnGewaehlt:=loop;
        PaintIt;
        exit; // raus; nicht schön, aber es funktioniert ;-)
       end; // gefunden
    end;
end;
```

Aufgrund des allgemein gehaltenen Basistypen brauchen wir hier also gar nicht zu wissen, um was für ein Objekt es sich handelt. Das ist uns völlig egal!

Genauso werden wird die Objekte auch bewegen. Hierzu fügen wir im OnKeyDown-Ereignis des Formulars (vorher Keypreview=true im Objektinspektor einstellen) ein:

```
var Basis:^BasisTyp;
begin
 if AnGewaehlt=-1 then Exit; // noch kein Objekt erstellt, also raus

 Basis:=Objekte[AnGewaehlt];
 if not (ssShift in Shift) then // die Cursor-Tasten bewegen das gerade
angewählte
                               // Objekt
  begin
   if key=vk_Left then Dec(Basis^.X,5);
   if key=vk_Right then Inc(Basis^.X,5);
   if key=vk_Up then Dec(Basis^.Y,5);
   if key=vk_Down then Inc(Basis^.Y,5);
  end else begin // Shift und Cursor verändert die Größe des aktuellen
Objektes
   if key=vk_Left then Dec(Basis^.Width,5);
   if key=vk_Right then Inc(Basis^.Width,5);
   if key=vk_Up then Dec(Basis^.Height,5);
```

```
   if key=vk_Down then Inc(Basis^.Height,5);
  end;

 PaintIt;
end;
```

Um außerdem noch Objekte nach vorne und hinten stellen zu können, geben wir noch folgendes ein:

```
 ...
 if key=vk_F1 then // nach hinten
  begin
   Objekte.Move(AnGewaehlt,0);
   AnGewaehlt:=0;
  end;
  if key=vk_F2 then // nach vorne
   begin
    Objekte.Move(AnGewaehlt,Objekte.Count-1);
    Angewaehlt:=Objekte.Count-1;
   end;
 PaintIt;
end;
```

Schritt 5

Damit wir unsere Objekte auch noch mit der Maus bewegen können, gehen wir wie folgt vor.

In der Public-Sektion fügen wir noch folgende Variablen hinzu:

```
 ...
 if key=vk_F1 then // nach hinten
  begin
   Objekte.Move(AnGewaehlt,0);
   AnGewaehlt:=0;
  end;
  if key=vk_F2 then // nach vorne
   begin
    Objekte.Move(AnGewaehlt,Objekte.Count-1);
    Angewaehlt:=Objekte.Count-1;
   end;
 PaintIt;
end;
```

Wenn nun ein OnMausDown-Ereignis auf unserer Zeichenfläche eintritt, stellen wir zuerst fest, daß die Maus gedrückt wurde, an welcher Position und ob vielleicht ein neues Objekt anvisiert wurde:

```
procedure THauptform.ZeichenflaecheMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var loop:Integer;
    Basis:^BasisTyp;
begin
  MausDown:=true;
  MausX:=X;
```

```
  MausY:=Y;
   for loop:= Objekte.Count-1 downto 0 do // das vorderste zuerst prüfen
                                          // es überlagert u.U. ja weiter
hinten
                                          // liegende Objekte
    begin
     Basis:=Objekte[loop];
     if X>Basis^.X then if X<Basis^.X+Basis^.Width then
      if Y>Basis^.Y then if Y<Basis^.Y+Basis^.Height then
       begin
        AnGewaehlt:=loop;
        PaintIt;
        exit; // raus
       end; // gefunden
    end;
end;
```

Dabei ergibt sich folgendes kleines Problem: Wenn wir mit den Cursortasten die Größe eines Objektes verändern, können wir es auch "Spiegeln", d.h. ihm eine Größe <0 zuweisen. Dann funktioniert der obige Code zum Erkennen, ob ein Objekt anvisiert wurde, nicht. Wenn ihr das ausbessern wollt, müßt ihr entweder verhindern, daß ein Objekt eine Größe kleiner als 0 zugewiesen bekommt oder ihr müßt obige Routine ein klein wenig umschreiben, damit sie auch in solchen Fällen funktioniert. Das würde hier aber zu weit führen.

Bewegt sich jetzt die Maus (OnMouseMove) führt das Programm folgenden Code aus:

```
var Basis:^BasisTyp;
begin
  if MausDown then
  begin
    if AnGewaehlt=-1 then Exit; // noch kein Objekt
    Basis:=Objekte[AnGewaehlt];
    Basis^.X:=Basis^.X+(X-MausX);
    Basis^.Y:=Basis^.Y+(Y-MausY);
    MausX:=X;
    MausY:=Y;
    PaintIt;
  end;
end;
```

Ich wiederhole mich zwar, aber auch hier interessiert es uns wieder nicht, um was für eine Art Objekt es sich den handelt. Die Maus-Koordinaten speichern wir übrigens, um festzustellen, um wieviel Pixel die Maus bewegt wurde (X-MausX).

Wird die Maus wieder losgelassen, stellen wir das einfach fest:

```
var Basis:^BasisTyp;
begin
  if MausDown then
  begin
    if AnGewaehlt=-1 then Exit; // noch kein Objekt
    Basis:=Objekte[AnGewaehlt];
    Basis^.X:=Basis^.X+(X-MausX);
    Basis^.Y:=Basis^.Y+(Y-MausY);
    MausX:=X;
```

```
    MausY:=Y;
    PaintIt;
  end;
end;
```

Schritt 6

Zum Schluß fügen wir jetz wie versprochen noch die LoescheObjekt-Prozedur ein, damit wir keine Speicherleichen hinterlassen:

```
var Basis:^BasisTyp;
begin
  if MausDown then
  begin
    if AnGewaehlt=-1 then Exit; // noch kein Objekt
    Basis:=Objekte[AnGewaehlt];
    Basis^.X:=Basis^.X+(X-MausX);
    Basis^.Y:=Basis^.Y+(Y-MausY);
    MausX:=X;
    MausY:=Y;
    PaintIt;
  end;
end;

procedure THauptform.LoescheObjekt(Objekt:Pointer);
var Basis:^BasisTyp;
    bild:^BildTyp;
    Form:^FormTyp;
begin
 Basis:=Objekt;
 case Basis^.Typ of
  1: begin
      Bild:=Objekt;
      Bild^.Image.Free;
      Dispose(bild);
     end; // bild
  2: begin
      Form:=Objekt;
      Dispose(Form);
     end; // form
 end; // case
end;
```

Ausblick

Wir sind nun also am Schluß unseres kleinen Tutorials angekommen. Ich hoffe, ich konnte euch näher bringen, wieso ihr TListen ganz geschickt einsetzen könnt und wozu ein BasisTyp gut ist.

Folgendes ließe sich (u.a.) noch am Programm verbessern:

- Das schon angesprochene Problem mit der Erkennung negativer Höhen oder Breiten
- Freie Wahl der Rahmen- und Füllfarbe wenn eine Fom erstellt wird. Seien wir ehrlich: schwarzer Rand, blauer Hintergrund ist nicht immer prickelnd.

- Eine Größenveränderung mit der Maus.
- Mehrere Objekte anwählen.
  Hierzu würde ich dem Basistyp eine Variable namens Angewaehlt beifügen (**nicht vergessen**: auch die Bild- und Formtypen brauchen dann diese Variable an genau derselben Stelle wie der Basistyp, sonst bekommt ihr entweder abstruse Ergebnisse oder böse Lesefehler).
- Keine zufällige Position oder Größe der Objekte sondern ein "Aufziehen" mit der Maus.
- "PaintIt" erst auf einem unsichtbaren Image oder Bitmap zeichnen und dann als Ganzes auf die Zeichenfläche kopieren, so kann ein Flimmern verhindert werden.
- Ich bin mir nicht ganz sicher, aber ich glaube, eine Paintbox wäre von der Zeichengeschwindigkeit schneller als ein TImage. Es müßte aber bei jedem OnPaint des Formulars neu gezeichnet werden.

Wozu ihr (wieder mal u.a.) das hier Gezeigte noch gebrauchen könnt:

- Um Spiele zu schreiben, bei denen sich mehrere Objekte (Gegenstände, Truppen etc.) auf einem Feld befinden.
- Ein Autorensystem (vielleicht schreib ich ja noch mal ein kleines Tutorial "Wir schreiben uns DelphiLight").

Zum Abschluß gibt's hier jetzt auch endlich den ganzen Code als Zip-File (19 KB).

Das Ganze wurde mit Delphi 5 geschrieben, kann also mit älteren Versionen u.U. nicht gelesen werden. Ihr könnt aber einfach den Code aus "main.pas" in euer Projekt rüberziehen.
